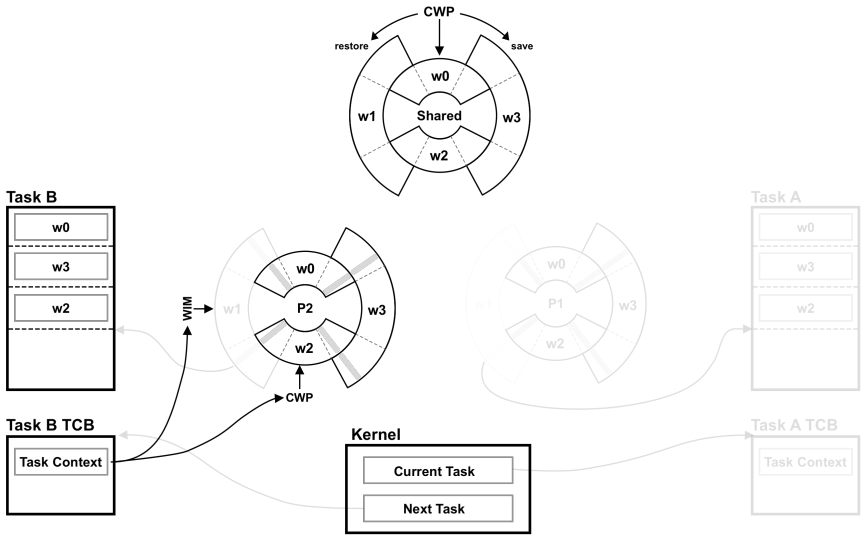
DEGREE PROJECT

Porting Zephyr RTOS to the LEON/GRLIB SoC SPARC v8 architecture



Nikolaus Huber

Space Engineering, master's level (120 credits)
2019

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Abstract

The aim of this thesis is to create a port of the Zephyr realtime operating system for the LEON processor platform. The LEON is a frequently used computing core for spaceflight applications, with ample flight heritage. It is based upon the well established SPARC v8 instruction set, and offers many extensions to ease software development and increase overall processor performance. An overview of the necessary steps towards a functional architecture port is given in this report. Special emphasis is put upon the interrupt handling and context switching. One LEON specific feature introduced with the GR716 LEON3-FT microcontroller, register window partitioning, is used to increase the performance of the context switching mechanism in the operating system. By using this feature, context switching time has shown to decrease significantly, while easing verification of the overall software system by providing dedicated partitions for tasks with hard realtime requirements.

Keywords: RTOS, SPARC, Zephyr, LEON, Register Window Partitioning

Sammanfattning

Det övergripande målet med examensarbetet är att porta Zephyr realtidsoperativsystem (OS) till LEON processorplattformen. LEON processorn är ursprungligen designad för och förekommer ofta i datorsystem inom rymd p.g.a. sina feltoleranta egenskaper. LEON är kompatibel med den öppna SPARC v8 instruktionsuppsättningen vilken också tillåter utökning och anpassningar. Rapporten ger läsaren en överblick av vilka steg som är nödvändiga för att skapa en fungerande arkitekturport av ett OS. Vidare beskriver rapporten mer i detalj designen kring trådväxling och avbrottshantering, samt hur dessa anpassas för att utnyttja LEON specifika utökningar av SPARC till att nå högre prestanda. GR716 LEON3-FT introducerar partitionering av SPARC registerfönster för att kunna minska tiden det tar operativsystemet att växla trådar. Denna funktion har inte använts tidigare i något OS, och är därför av särskilt intresse att studera och karakterisera. Resultaten visar att trådväxlingstiden minskat signifikant, samtidigt som determinismen blivit bättre och därigenom är det nu enklare att designa system med hårda realtidskrav.

Acknowledgements

I would like to express my genuine gratitude to my academic supervisor, Anita Enmark, at Luleå University of Technology, for not only agreeing to examine my master thesis project, but for inspiring me to dive into the fascinating world of spacecraft onboard data handling in the first place. Thank you for the invaluable help during the last two years, with all the projects I was involved in.

Furthermore, I would like to thank my supervisor, Daniel Hellström, from Cobham Gaisler, for suggesting this thesis topic, as well as the amazing support on the way. Also, my sincere thanks to my second supervisor, Martin Åberg, for your input and your help, and the many interesting discussions we had about the SPARC platform during my time at Gaisler.

There are too many people, who have made my years of studying as enjoyable as they were, therefore, I just want to name a few. To Cassidy Thompson, a big thank you for your friendship, and your help whenever I struggle with the English language. To Florine Enengl, for all the fun we had in Vienna, and continue to have during our abroad adventures. To Kiki, Fabian, Siiri, Hampus, Georges, Natalie, August, Ludvig, John, Terese, Joar, Daniel, Niels, Elin, and Anna, for making my time in Kiruna unforgettable.

Finally, I must express my very profound gratitude to my parents, Anita and Wolfgang Huber, and to my brother, Laurenz, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of moving to Sweden. This accomplishment would not have been possible without them. Thank you.

Contents

Acronyms

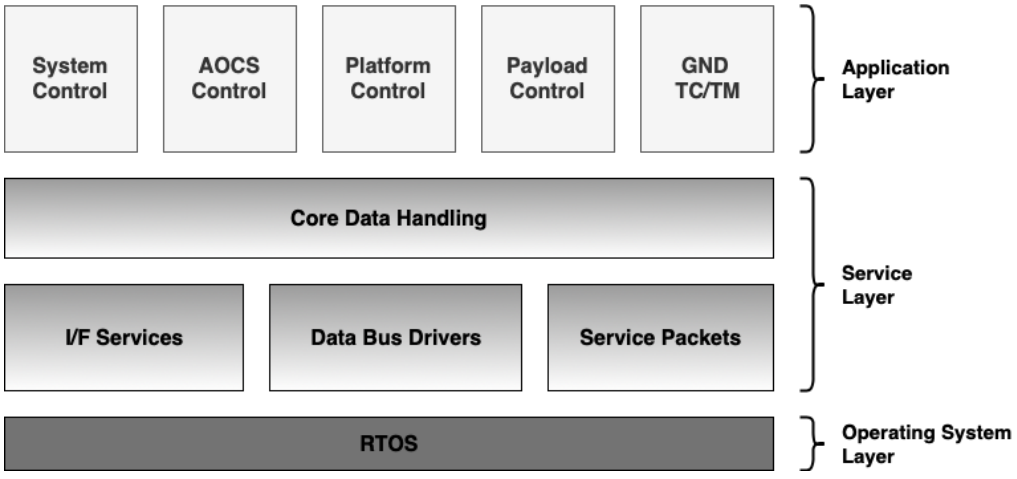
ADC	Analog-Digital Converter
ABI	Application Binary Interface
AOCS	Attitude and Orbit Control System
ASIC	Application Specific Integrated Ciruit
BIOS	Basic Input Output System
CPU	Central Processing Unit
CWP	Current Window Pointer
FPGA	Field Programmable Gate Array
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LSB	Least Significant Bit
nPC	Next Program Counter
OS	Operating System
PC	Program Counter
PIL	Processor Interrupt Level
PROM	Programmable Read Only Memory
PSR	Processor Status Register
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operating System
SoC	System on Chip
TBR	Trap Base Register
TCB	Task Control Block
TM/TC	Telemetry/Telecommand
WCET	Worst Case Execution Time
WIM	Window Invalid Mask

List of Figures

CHAPTER 1

Introduction

Similar to modern software running on ground equipment, spacecraft onboard software today is also often divided into different layers. An example for such an architecture is illustrated in Figure 1.1.



Source: [4, page 90]

Figure 1.1: Example of a layered software architecture for onboard data handling systems.

The *application layer* is usually the highest level of onboard software. Different applications are responsible for handling different tasks commonly found on modern spacecrafts, such as keeping the spacecraft oriented (Attitude and Orbit

Control System (AOCS)), communicating with other spacecrafts or ground stations (Telemetry/Telecommand (TM/TC)), supervising the status and health of the spacecraft (Platform Control), commanding and receiving data from scientific instruments (Payload Control), to name a few.

In order to ease the development of these applications, the onboard computing platform usually offers interfaces for accessing the spacecraft's hardware (e.g. reaction wheel control, thermistor readings, etc.) and the upper layers of common communication protocol stacks (e.g. CAN, MIL-STD-1553, etc.). These routines and drivers are gathered in the *service layer*.

The level closest to hardware in a typical onboard software architecture is the *operating system layer*. As software running on a spacecraft is subject to stringent timing requirements, a special class of operating systems is used, which is usually referred to as Real Time Operating System (RTOS). An RTOS is a piece of software, that can schedule multiple applications (usually then called tasks or threads) on a limited number of processors. In contrast to a general purpose Operating System (OS), an RTOS can guarantee deterministic timing behaviour for its different operations (e.g. switching from one execution thread to another takes a fixed number of clock cycles). This allows for the software system to be analysed during design and implementation, so that it can be verified, that every task always finishes within its predefined deadline. This thesis deals with a specific RTOS, called Zephyr. The main goal of this thesis was to port this RTOS to a previously unsupported architecture, the LEON platform.

1.1 Thesis outline

This thesis report is divided into the following parts:

Chapter 2 gives an introduction to realtime operating systems in general, a short overview of the Zephyr RTOS, and a description of the underlying hardware platform.

Chapter 3 illustrates the overall porting process of the OS, gives an overview of the important steps towards a functioning system, and highlights some of the design choices made along the way.

Since every hardware architecture has its specific design traits, chapter 4 introduces the concept of *window partitioning*, which is available on certain LEON processors. The impact of using this feature on the performance of the operating system is analysed in chapters 5 and 6.

Finally, a conclusion of this work is given in chapter 7 together with suggestions for possible future improvements in chapter 8.

1.2 Copyright notice for code samples

All software development for this work has been done during a thesis internship at Cobham Gaisler AB in Gothenburg, Sweden. As such, all the code samples found in this report are subject to the following copyright notice:

CHAPTER 2

Background

2.1 Realtime operating systems

"An embedded system is a combination of computer hardware and software [...] designed to perform a dedicated function." [2]

Embedded systems are everywhere. They range from the smallest sensor configurations up to control systems for cars, aeroplanes, and satellites. As mentioned in the above quote, embedded systems are designed to perform dedicated functions, which separates them from general computing systems, such as *servers* or *personal computers*. General computing systems are versatile, the same PC can be used for different tasks, such as gaming, office work, or scientific computations. Embedded systems are dedicated to the environment they are embedded in.

Embedded systems often have stricter requirements than general purpose systems regarding performance, reliability, and predictability. For example, the software used for controlling the Anti-lock Breaking System (ABS) in modern cars must be computationally efficient (performance), is not allowed to ever reach a dead-lock state (reliability), and must be able to work with a deterministic timing behaviour to guarantee a controlled, periodic break release pattern (predictability).

Early embedded systems often featured dedicated processors for each important function. This means, that the ABS for example, would have its own computing chip, whose sole duty is to control the before mentioned break release patterns. Fortunately, the majority of time while driving is not spent on emergency breaking. This implies, that the processor used for this emergency feature is idle most of the

times. Dedicating a processor to each feature a car has results in an overall increase in costs, weight, and power consumption. On the other hand, predictability is easy to verify, since the overall response time of the system is never longer than the Worst Case Execution Time (WCET) of the one software routine executing on it. Especially with space applications in mind, the increased weight and power consumption may impose problems on the overall system design, so that a different approach might be advantageous.

In order to minimise the amount of processors present in a system design, multiple software routines (usually termed processes, threads, or tasks) share the available computational resources. This means, that there must be a special piece of software, that distributes these resources among different processing jobs, usually called a *scheduler*:

"The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput and processor efficiency" [18]

The scheduler is part of an *operating system*, which usually includes other software as well (e.g. libraries for accessing the hardware, routines for memory management, etc.). Many (especially safety-critical) embedded systems have realtime requirements. Realtime computing systems can be characterised in the following way:

"The correctness of a computation depends not only on the logical correctness but also on the time at which the results are produced." [16]

It is common to distinguish hard realtime and soft realtime systems. In hard realtime systems, each computational task has a deadline, which must be kept under all circumstances. Missing a deadline might result in economical disaster or loss of human lives [16]. Soft realtime systems have less strict timing requirements in the sense, that missing a deadline from time to time is tolerable.

The stringent timing requirements of realtime systems necessitate the usage of a particular kind of OS, commonly termed Real Time Operating System. Over the years, a multitude of different RTOS have been developed, ranging from small microkernels, to realtime extensions of desktop grade operating systems (e.g. Realtime Linux).

This thesis work was dedicated to one specific RTOS, Zephyr, which will be presented shortly in the following section.

2.1.1 An RTOS for the age of IoT - Zephyr

"The Zephyr Project strives to be the best-of-breed, open source RTOS for connected, resource-constrained devices, and built with security and safety in mind." [22]

Zephyr is an RTOS, that is built and maintained by different companies working in the Internet of Things (IoT) field. The core of the OS is taken from Wind River's operating system kernel *Rocket* [23].

The kernel itself can be divided into a very small nano-kernel, and a slightly bigger micro-kernel, which incorporates additional features to ease software development for Zepyhr.

Zephyr has been developed with the following goals in mind [14]:

- CPU architecture independence
- Small footprint (can run from 10 kB of memory)
- Security
- Connectivity
- Integration of powerful development tools
- Open source code

Zephyr offers a wide range of different features [11]:

- Multithreading (including support for POSIX pthreads API)
- Interrupt services (both registering interrupt service routines at compile time and runtime)
- Memory allocation services
- Inter-Thread synchronisation services (e.g. semaphores, mutexes)
- Inter-Thread data passing services (e.g. message queues, byte streams)
- Power management services
- Support for multiple scheduling policies (e.g. rate monotonic, earliest deadline first, cooperative, preemptive, etc.)

Developing applications for Zephyr is done using the cmake[1] tool. Cmake itself creates build systems in different formats, with Zephyr supporting the well established Unix makefiles, or the more modern Ninja format. For an example of how to build a working application for Zephyr, refer to [10] or Appendix B.

2.2 Hardware architecture

2.2.1 SPARC v8

The Scalable Processor ARChitecture (SPARC) does not refer to a single chip, or even family of chips. It is, what is usually called an Instruction Set Architecture (ISA):

"An abstract interface between the hardware and the lowest level software of a

[1] https://cmake.org

machine that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on." [15]

An ISA can be imagined as a contract between the manufacturer of a processor and the software developer. It guarantees, that an application written according to the ISA will run on any chip that is compliant with the ISA. As such, it can be seen as the programmer's view of the underlying processor platform.

Over the years, many different implementations of the SPARC architecture have been developed, ranging from small microcontrollers to processors meant for high performance computing. There are multiple versions of the SPARC ISA, since it has gone through various design iterations over the years. This thesis focuses on version 8 of the SPARC instruction set, as it forms the base for the LEON3 processor core.

The instruction set guarantees, that a machine language program will always yield the same result, independent of the actual hardware used. However, in modern software development, applications are rarely written in machine language anymore. Instead, higher level languages are used to ease the development process.

These higher level languages are then translated by a specific piece of software, called the *compiler*. The compiler takes an application written in a higher level language and translates it to the desired machine language. In case that the desired machine language is not the same as the machine language understood by the host platform (i.e. the computer which is running the compiler), this process is often also referred to as cross-compilation (which is usually the case in software development for embedded systems).

To make it possible for compiled programs to be linked together with other machine level code, and to allow for standardised ways of calling operating system functions, a second level of agreement has to be established, referred to as the Application Binary Interface (ABI):

"The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers." [15]

The SPARC ABI [19] includes important details, such as function calling conventions (i.e. how arguments are transported to and from function calls), the usage of the system's stack memory (e.g. stack frame sizes and relative addresses of specific values within each stack frame), and operating system calls.

Within the scope of this thesis it is neither practical, nor necessary to include a full description of the SPARC v8 design. Only the parts, which are inevitably important for the understanding of the described algorithms will be presented in the following sections. For a full overview of the underlying architecture, please refer to the SPARC v8 manual [21] and the SPARC ABI [19].

2.2.1.1 Registers

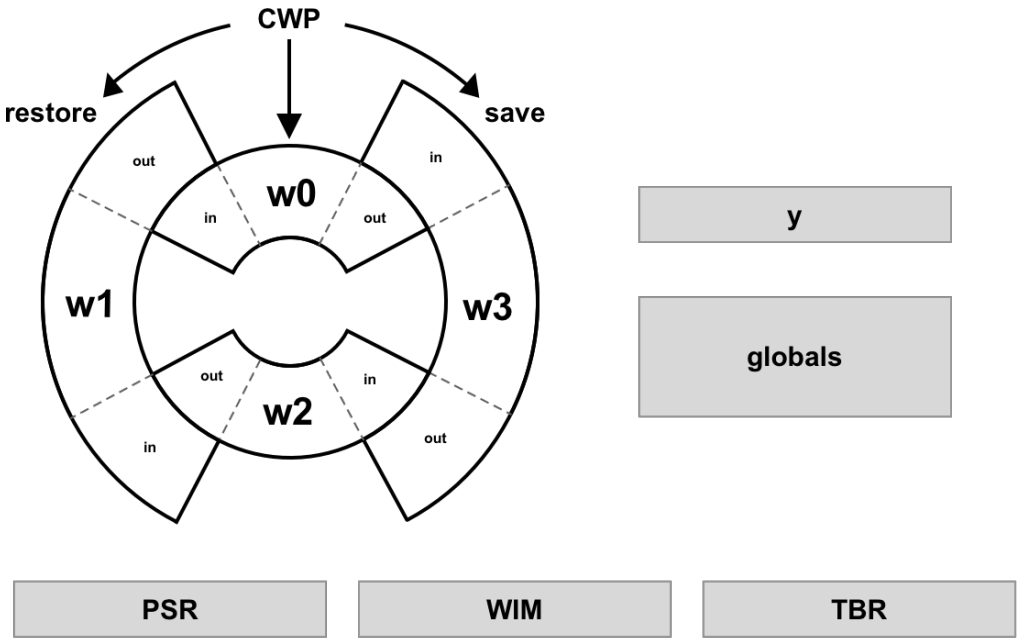


Figure 2.1: Typical SPARC register file.

SPARC v8 defines a 32 bit (big endian) Harvard architecture. One design characteristic of the SPARC platform is the use of a windowed register file. In many other Reduced Instruction Set Computer (RISC) architectures, a bank of general purpose working registers is provided as operands for instructions. Whenever a new function is called, at least part of these registers must be saved out to the main memory, to provide a blank working environment for the new function. The particular registers needed to be saved are defined in a platform's ABI.

In the SPARC architecture, another approach is used, which is demonstrated in Figure 2.1. At any point, a function has access to 8 global registers, and 24 work registers, which form a window. Each register window is divided into three groups: 8 input, 8 local, and 8 output registers. An important design trait, which allows for a certain convenience when doing function calls, is the overlapping of windows. The 8 output registers of the current window (**w0** in Figure 2.1, which is pointed to by the Current Window Pointer (CWP)) are the same as the 8 input registers of the adjacent window (**w3** in the example). When executing a function call with parameters, these parameters can be put into the output registers of the current window, and upon execution of a **save** instruction, these parameters are available in the input registers of the new window. This mechanism makes it possible to minimise the amount of memory loads and stores that need to be performed (which on most systems are slower than accessing working registers). If functions do not need more registers, than provided in their window (together with the global registers), not a

single access to the main memory has to be done. As such, the register windows can be seen as a cache for the system's stack memory. The SPARC ISA specifies, that the number of available windows on a compliant system must be between 2 and 32. Algorithms are therefore developed in a way, that they do not depend on the actual number of windows present on a specific chip. For demonstration purposes, all algorithms in this thesis report will be shown on the 4 window register file illustrated in Figure 2.1. This example figure furthermore indicates the global registers, the **y** register (which is used during certain arithmetic operations), the Processor Status Register, the Window Invalid Mask, and the Trap Base Register.

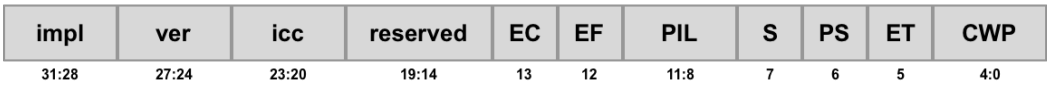


Figure 2.2: Processor Status Register fields.

The structure of the Processor Status Register (PSR) is shown in Figure 2.2. It consists of the following fields:

impl These bits identify the specific implementation or class of implementations of the architecture
ver Identifies a distinct version of the chip within an implementation class
icc Holds the condition codes for the *integer unit*. Those bits are used for conditional branching
res. Reserved for future iterations of the SPARC ISA
EC If set, the (optional) coprocessor is enabled
EF If set, the (optional) floating point unit is enabled
PIL Processor interrupt level. Sets the interrupt level above which the processor will transfer control to a trap handler (see section 3.3)
S If set, the processor is in supervisor mode
PS Contains the value of the S bit during the most recent trap
ET If set, traps are enabled (see section 3.3)
CWP Number of the current window in use by the system

Some SPARC instructions are only allowed to be executed if the processor is in a supervisor mode. The **S** and **PS** bits can be used to implement automatic switching between the user and the operating system code. In the current implementation, all code is executed in supervisor mode, therefore, these two bits are just set to 1 during boot up (see section 3.2). The **PIL** and **ET** are used for controlling the way the processor reacts to interrupts. Those two fields are further explained in section 2.2.1.2. Since neither a coprocessor, nor the floating point unit have been used during this thesis work, the respective enable bits **EC** and **EF** are not further discussed here. The CWP encodes the currently used window. It is a 5-bit number, which explains the maximum of 32 register windows in a SPARC v8 system.

One problem with the windowed register file arises, once a **save** instruction causes a wrap around on the register ring. Therefore, the last window, which has empty

local and input registers (OBSERVE: the output registers already hold valid data, since the coincide with the input registers of the first used window) must be marked in a way, that the operating system can take care of the imminent overflow. This marking is done in the WIM register:

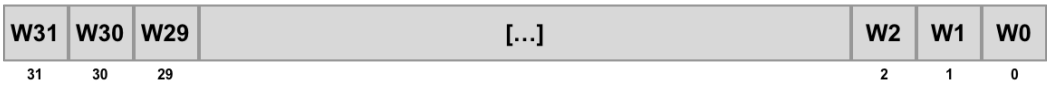


Figure 2.3: Window Invalid Mask fields.

The WIM is used to mark windows as invalid, thereby guarding the system against overflows. When a **save** instruction would cause entering a window, that already holds valid data, a trap is caused by the architecture, and execution is given to a specific function of the operating system to resolve this issue. This mechanism is further described in section 3.3.2. In normal operation, only one bit is ever set in the WIM register, illustrated in Figure 2.3. If a specific system does not have the full 32 windows available, the upper bits of WIM always read as zero, and any writes to them are neglected.

The Trap Base Register (TBR) will be explained in the following section. Although not currently implemented, it shall be mentioned, that for changing the values of PSR, WIM, and TBR, the processor must be in supervisor mode.

2.2.1.2 Traps

A trap is a transfer of control to a special part of the operating system. It is used to handle unexpected events, such as errors and interrupts. The SPARC v8 defines three different categories of traps:

Precise traps are caused by a particular instruction. The trap code is executed before any software visible state changes due to the trapping instruction. An example for a precise trap is the *window overflow trap*, further explained in section 3.3.2. Whenever the CWP is decremented (e.g. due to a function call), and it thereby enters an invalid window, the processor transfers control to the *window overflow trap handler* to change the state of the system in a way, so that execution can be resumed.

Deferred trap are similar to precise traps, with the exception, that by the time the processor transfers control to the trap handler, the program visible state of the system might have changed. Examples for deferred traps are error traps caused by the floating point unit.

Interrupting traps can be caused by an external interrupt request, implementation specific states (e.g. breakpoint mechanisms), or an exception caused by an earlier instruction (e.g. an ECC data error after a load instruction).

The SPARC v8 offers 16 different external interrupt request lines. In order to cause an external interrupt, two conditions must be met:

- Traps must be globally enabled by setting the **ET** bit in the PSR
- The asserted request line number must be above the current **PIL** (encoded as a 4 bit field in the PSR)

The SPARC v8 ISA defines 256 different trap types. All trap types are listed in a *trap table*, that holds the first four instructions of each trap handler routine. Usually, these four instructions are used to jump to the actual trap handler code that is somewhere else in the instruction memory. The trap table itself can also be put at different memory locations. When a trap happens, the processor will look up the location of the trap table through the TBR. This register must be set up to point to the beginning of the trap table during the early boot up sequence (see section 3.2).

Figure 2.4: Trap Base Register fields.

The TBR register consists of 3 different fields, bits 32 to 12 are the Trap Base Address (TBA), containing the most significant 20 bits of the trap table address. Bits 11 to 4 encode the trap type (thereby offering 256 different types). Bits 3 to 0 are always zero (therefore each trap type entry has 4 instructions available in the trap table).

Since the upper 20 bits of the TBR are set up during boot up, the processor only has to adjust the trap type field whenever a trap happens, and the TBR will automatically point towards the address of the respective trap handler within the table.

Given the number of trap types and the number of instructions per trap type in the table, the trap table always has a footprint of 4 kB. Especially for memory restricted systems, this might cause memory space issues. Therefore, the SPARC v8 Embedded Extension [17] suggests a different trap handling strategy, *single vector trapping*. When using single vector trapping, control is always transferred to the address specified by the upper 20 bits in the TBR, independent of the trap type. A software routine located at this address must then take care of looking up the correct trap handler routine. Since usually not all 256 trap types have a dedicated trap handler assigned to them, the majority of them can point to the same default handler routine, thereby greatly reducing the memory footprint of the trap table.

2.2.1.3 Assembly language

In this following section, only a very short introduction into SPARC assembly will be given to ease the understanding of the supplied code snippets throughout this report. For a full list of available instructions on SPARC architectures refer to [21].

SPARC assembly language instructions are interpreted left to right. A typical addition for example can be done with:

```
add    %l1, %l2, %l3
```

The **add** instruction causes the values currently present in local registers **%l1** and **%l2** to be added together, and the result will be put into local register **%l3**. In the example above, all three operands are given as registers. The second operand could be given as an immediate value as well (e.g. to accommodate different memory addressing modes):

```
add    %l1, 0x10, %l3
```

Here, the value **0x10** will be added to the value in **%l1** and the result saved in **%l3**. It is important to notice, that since each instruction is 32 bits wide, there is a limit on the value that can be immediately supplied as an operand. Given the design of the **add** instruction in the SPARC ISA, immediate arguments are restricted to 13 bit numbers [21].

If a higher number shall be used as an operand, it first has to be placed into a register. For a 32 bit number, this is done in two steps:

```
sethi   %hi(0x7FFFFFFF), %l1
or      %l1, %lo(0x7FFFFFFF), %l1
```

The **sethi** instruction sets the highest 22 bits of the supplied destination register **%l1**. The consecutive **or** is used to set the lower 10 bits. The second line of this example also demonstrates, that the very same register can both be used as a source and as a destination. The macros **%hi** and **%lo** are provided by the preprocessor of the assembler for convenience to mask out the upper 22 or the lower 10 bits of a 32 bit number. Since the operation of setting a 32 bit number in a register is very common (e.g. for setting up memory addresses), there is a synthetic instruction, that can be used when writing assembly code. The following code will automatically be translated into the same machine code as the statement given above:

```
set     0x7FFFFFFF, %l1
```

Some instructions, such as the store (**st**) and load (**ld**) instruction, are also available in a double word version (**std**, **ldd**). In order to use the double word instructions, the supplied memory address must be double word aligned.

SPARC features a delayed branching strategy. This means, that the instruction

following immediately after a branch instruction will always still be executed (there is a way of annulling the delay slot when doing conditional branching [21]). The delay slot can, for example, be used to supply an argument to a called subroutine by writing to the output register:

```
call    function1
 wr     0x10, %o0
```

In the above example, **function1** is called and there is a write instruction in the delay slot, which will put the value **0x10** into output register **%o0**. This write instruction will be executed before the first instruction of **function1**. To mark delay slots, it is common to indent the instruction.

2.2.2 LEON3

The LEON3 is a fully SPARC v8 compliant 32-bit processor core developed by Cobham Gaisler AB. It is designed for embedded applications, offering multiple ISA extensions suggested in the SPARC Embedded Extension [17]. Especially with space applications in mind, the processor soft core can be configured to enable or disable fault-tolerance against Single Event Upset (SEU) errors at design time.

The LEON3 is part of a bigger ecosystem, called GRLIB [9]. GRLIB is a collection of VHDL libraries of different Intellectual Property (IP) cores. GRLIB based designs are built around the AMBA-2.0 AHB/APB bus architecture. The idea is, that embedded system designers can select all the IPs/pheripherals they need for their specific application, and connect those to a central bus. The finished designs can be synthesised in FPGAs or ASICs.

Cobham Gaisler also already offers different System on Chip (SoC) designs, which are radiation tested and qualified for space flight applications. One of the newest additions to this line of SoCs is the GR716 microcontroller.

2.2.3 GR716

The GR716 is a new microcontroller designed by Cobham Gaisler for the space and aeronautics market. It is based on the LEON3-FT processor design, and offers many standard interfaces used in space applications:

- MIL-STD-1553
- CAN 2.0
- UARTs, SPI, I2C
- SpaceWire

Proposed usage areas include propulsion system control, sensor bus control, power control, thermal control, AOCS, and instrumentation control, among others.

The high number of register windows (31 windows) motivates the usage of a special LEON3 feature, *register window partitioning*, which will be used for enhancing context switching speed (see section 4.2).

For a full description of the GR716, refer to its datasheet [6].

2.2.4 The porting process

Zephyr is developed with portability in mind. The majority of the operating system code is written in high level C, therefore does not need to be adapted when running on a different platform. Nevertheless, specific parts of the operating system are depending on the underlying hardware. It shall be mentioned here, that the current implementation of the Zephyr RTOS is not developed to be compliant with any coding standards for safety-critical systems (e.g. MISRA-C [2]).

The Zephyr documentation provides a porting guide [1], and the overview of the porting process illustrated in chapter 3 will mostly follow the steps highlighted there. Especially the early boot process (section 3.2) was inspired by the C runtime environment included in Cobham Gaisler's GCC compiler extension for the LEON platform, called BCC [5]. As indicated before, one of the most important jobs of an operating system is switching from one execution thread to another. This ability ultimately allows for multiple (independently developed) software function routines to share one processing core.

A lot of research effort has gone into developing efficient algorithms for the SPARC platform to achieve this context switching behaviour. A good overview of what is understood as a task's context can be found in [12]. It also gives an introduction into how light-weight threads are implemented in the desktop operating system SunOS. [13] presents multiple different context switching algorithms for SPARC like processors, together with an estimate of their complexity and runtime behaviour. The lazy context switching mechanism, that was developed as part of this thesis (see section 4.2), is based on an algorithm specified in the mentioned report. A good introduction to the SPARC platform itself is given in [3]. It describes different hardware implementations of the SPARC ISA, and presents important, hardware-vendor independent software routines, such as the window overflow and underflow trap handlers (see section 3.3).

[2] `https://www.misra.org.uk`

CHAPTER 3

Porting Zephyr

3.1 Overview

The word porting usually refers to the action of adapting parts of an application, in order to make it executable on a different platform. Depending on how different said platform is from the original one, this process might involve a different workload. Generally, when talking about porting for embedded systems, three different things might be meant:

- **Application porting**

Whenever an application, that was written for a specific OS shall be run on a different OS, the process to achieve this is called application porting. If the underlying OS conforms to a well established standard (e.g. POSIX), applications might not need to go through any change in order to run on a different system as well. This kind of porting will not be further discussed in the scope of this work.

- **Board porting**

If there already exists a port of the operating system for the underlying hardware, only parts of the code need to be adapted to the specific board that is used. This usually includes certain device drivers and protocol stacks.

- **Architecture Port**

In case that the operating system itself has not been ported to a specific architecture

before, this usually requires porting of all parts of the OS that require close interaction with the underlying hardware. Which parts of an OS need adaption in order to run on a different hardware depends on the pecularities of the used system. Certain parts of the OS almost always need altering, like the boot up sequence, error state handling, interrupt handling, and context switching.

In order to facilitate code reusability, the majority of modern operating systems are usually implemented in a higher level language, such as C or C++. A part of the code, however, needs to be adapted for each underlying platform the OS shall be used for (and is frequently written in the respective architecture's assembly language).

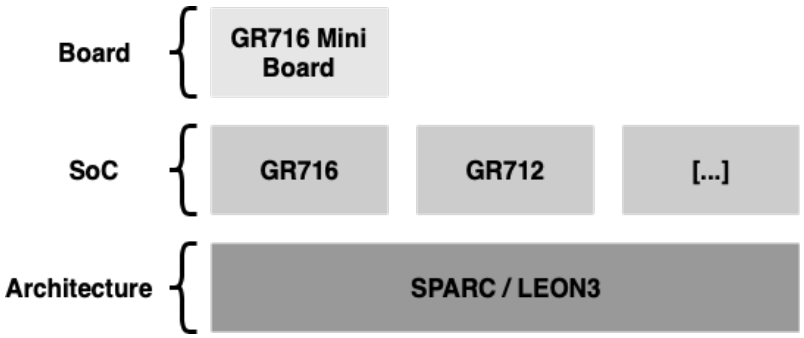


Figure 3.1: Zephyr RTOS hardware abstraction hierarchy for the LEON3 platform.

Zephyr features a layered design, where each level offers a higher abstraction of the underlying one. A visualisation of this hierarchy for the LEON3 architecture is illustrated in Figure 3.1. The SPARC v8 ISA and ABI build the basis of the LEON3 processor design. There are different implementations of the LEON3 processor, such as the GR716 or the GR712, and since the processor core is available as generic VHDL code, it can be implemented in other designs as well. The GR716 Mini Board is a physical evaluation platform developed by Cobham Gaisler to ease the development of applications based on this particular processor. It is used during hardware evaluation (see chapter 5).

An architecture port is needed to enable Zephyr to run on an ISA or an ABI that is not currently supported [1]. In this section, the most important steps towards a functional port of Zephyr on the LEON3 processor architecture will be described. The mentioned steps are mostly focusing on the architecture port, certain parts from the specific GR716 SoC port are included as well. The code of some of the most important algorithms developed during this thesis work is shown in Appendix A.

3.2 Early boot up sequence

After a system reset (e.g. initial boot up, reboot after error or power outage) the *early boot up sequence* is responsible for putting the system into a state, in which C code can be executed. In this section, a short overview of the necessary steps on the LEON3 platform is presented.

After power is available and stabilised in the system, the processor will start to execute instructions from the instruction memory. The first instruction to be executed is called the *entry point*. There are multiple ways of setting the memory address of this first instruction, most commonly, the **ENTRY** symbol is defined in the linker script (see section 3.8):

```
ENTRY(__leon_entry_point}
```

By defining the symbol **___leon__entry__point** as the entry point in the linker script, the linker takes care of installing the address of the first instruction correctly. After reset, the system will therefore resume execution at the following function:

```
FUNC_BEGIN __leon_entry_point
    mov     %g0, %g4
    sethi   %hi(__leon_trap_reset), %g4
    jmp     %g4 + %lo(__leon_trap_reset)
     nop
FUNC_END __leon_entry_point
```

As can be seen in the code snipped above, the entry point routine is used to call the *reset trap handler* (OBSERVE: at this stage, the trap table has not been installed yet, therefore, causing a reset trap by software would not work, instead, the address of the handler is used to jump into the routine).

In the reset trap handler, the system is first configured to use single vector trapping by setting the SV bit (capitalised expressions such as **ASR17__SV** in the example are used throughout this report for symbolic bitvectors) in the Ancillary State Register **%asr17** (as described in [17]):

```
rd    %asr17, %g1
set   ASR17_SV, %g2
or    %g1, %g2, %g1
wr    %g1, %asr17
```

Now the trap table can be installed by setting the TBR:

```
set   __leon_trap_table, %g1
wr    %g1, %tbr
```

Next, the CWP is set to point to **w0**, therefore, the invalid window is **w2** (OBSERVE: if **wr** is supplied with two source operands, those two are combined through

logic XOR before writing the result to the destination register). So first the WIM is set (by toggling the second bit from 0 to 1, **%g0** is hardwired to all zeros)

```
wr    %g0, 2, %wim
```

and then the PSR register is altered:

```
rd    %psr, %g1
set   (PSR_PIL | PSR_ET | PSR_CWP), %g3
andn  %g1, %g3, %g2
wr    %g2, PSR_ET, %psr
```

PSR_PIL, **PSR_ET**, and **PSR_CWP** are bit-masks for their respective fields within the PSR. By using them as the second argument to the **andn** instruction, those bits are effectively masked out (i.e. all set to zero). Therefore, when writing back to the PSR, this will set the CWP to **w0**, Processor Interrupt Level (PIL) to zero, and enable traps (since **PSR_ET** is used as second argument to **wr**).

From now on, the C runtime system can be initialised. Assuming, that the initial stackpointer was set up by the bootloader, the first stack frame can be allocated:

```
and    %sp, 0xfffffff0, %l0
sub    %l0, 96, %sp
clr    %fp
```

The stackpointer of the first frame is set to 96 bytes below the initial stackpointer value, and the frame pointer is cleared (According to [19] only the deepest stack frame of the system has a frame pointer to zero). After setting up the stack frame, some system specific information can be read from the configuration registers and the number of windows (and the number of windows minus one) can be set at the addresses labelled as **___leon_nwindows** and **___leon_nwindows_min1** respectively. Those are used by other routines, such as the *window overflow* and *window underflow* trap handlers, or the *context switching* mechanism.

At this point, the C runtime environment is successfully set up, and the first C routine can be called:

```
call    _PrepC
 nop
```

The **_PrepC** function can be used if there are further initialisation steps to be taken, which can be done in higher level C code. For now, it is enough to call the early kernel initialisation function **_Cstart** provided by Zephyr [1].

```
void _PrepC(void)
{
        _Cstart();
}
```

3.3 Traps

3.3.1 Trap table

As described in section 2.2.1.2, for memory efficiency reasons, it was chosen to use single vector trapping for this Zephyr port. Since only a few of the available 256 traps (trap type 0x00 to 0xFF) are going to be implemented, the trap table is divided into subtables. Each subtable is characterised by the first hexadecimal digit of the respective trap type. This means, there are 16 different subtables, and each subtable holds the addresses of 16 different trap handlers. For a basic implementation, only a few subtables are of interest, since the first one holds the addresses of important OS routines (e.g. window overflow, window underflow, etc.) and the second one holds the addresses of the 16 different interrupt service level routines. One more table is used for holding the addresses of often used routines (e.g. functions to alter the PIL).

The remaining 13 subtables can all point to one and the same table, which only holds addresses to the *unimplemented default trap*, which for now will just put the system into an error mode for debugging purposes.

This implies, that if a trap happens, execution will resume at the address specified by the upper 20 bits of the TBR, which was set up during early boot up (see section 3.2). From there, a software routine has to look up the memory location of the required trap handler routine. This is done in multiple steps:

```
rd     %tbr, %l6
and    %l6, TBR_TT_MASK, %l6
srl    %l6, TBR_TT_SHIFT, %l6
```

First, the current value of the TBR is read, then a mask is used to get the trap type from it. As the trap type is located at bits 4 to 11, it is also shifted to the right (**srl**).

```
sethi  %hi(__leon_trap_table_svt_level0), %l3
or     %l3, %lo(__leon_trap_table_svt_level0), %l3
```

Next, the address of the subtable is calculated, and the respective subtable is fetched:

```
and    %l6, 0xF0, %l5
srl    %l5, 2, %l5          ! Table offset
ld     [%l3+%l5], %l3       ! Fetch subtable
```

Now the actual trap handler address can be fetched from the subtable:

```
and    %l6, 0x0F, %l5
sll    %l5, 2, %l5          ! Table offset
ld     [%l3+%l5], %l3       ! Fetch trap handler
```

Finally, a jump instruction to the trap handler routine can be issued. In the delay slot the PSR is read into local register **%l0** as is required by the SPARC ABI [19]:

```
jmp    %l3
 rd    %psr, %l0
```

Given their importance, the remaining part of this section will highlight two different trap handler routines, the *window overflow* and *window underflow* trap handlers.

3.3.2 Window overflow trap

When a **save** instruction executes, the current value of the CWP is compared against the WIM. If the **save** instruction would cause the CWP to point to an invalid register window, that is, one whose corresponding WIM bit equals 1 (WIM[CWP] = 1), a window overflow trap is caused [21].

The overflow handler code is shown in appendix A.1. The working of this fairly short, but important, function will be illustrated in this section.

Both after the initial setup of the system through the Boot-PROM, and after a new thread is placed in the register file by the context switching mechanism (see section 3.4), only a single window is filled with valid data, and the window right behind it (CWP+1) is set in the WIM as invalid. The frame pointer (**%i7**) will point to the base address of a full stack frame in the task's stack (or the initial stack upon boot up), the stack pointer (**%o7**) will point to an address at least 96 bytes below the frame pointer (which is the minimum stack space per window as described in [19]). This situation is demonstrated in Figure 3.2. In the visualisations throughout this report, an empty rectangle symbolises an empty stack frame, if the frame is coloured, it holds valid data.

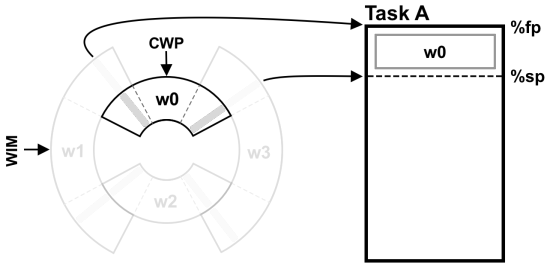


Figure 3.2: Initial setup for execution.

When a **save** instruction is executed, the CWP decreases by one, and in most cases, the **save** is accompanied by a subtraction instruction, which will set the stack pointer of the new window to at least 96 bytes below its frame pointer. For

convenience, these two operations can be combined, as the **save** instruction also acts as an addition if it is defined with operands (i.e. **save %sp, -96, %sp**). It is worth noticing, that both the source and destination register are defined as **%sp**, since in the case of **save**, the source is taken from the old, and the destination from the new register window. When no operands are defined, it will only decrease the CWP, but will not manipulate any registers in the windows. This is useful for stepping through the windows without changing their data, a feature, which will be used in the window overflow and underflow trap handlers. The state of the register file after one **save %sp, -96, %sp** instruction can be seen in Figure 3.3.

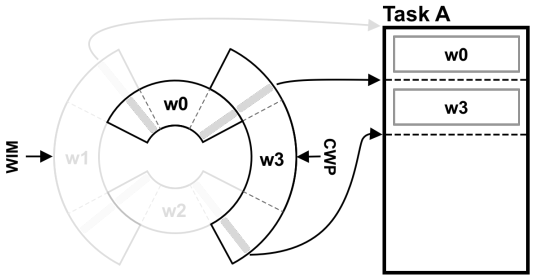


Figure 3.3: Decreasing the current window pointer.

The same procedure can be done one more time, which leads to a state shown in Figure 3.4.

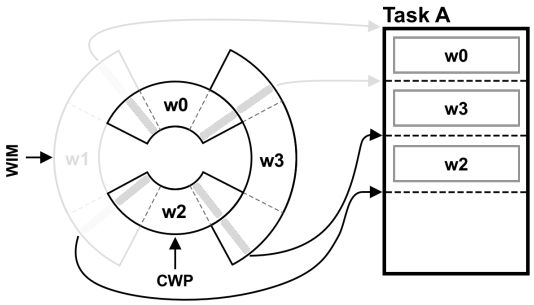


Figure 3.4: State of register file before window overflow.

If another **save** instruction is to be executed, this would align CWP and WIM, so instead a trap is caused. According to the SPARC ISA [21], the trap decrements the CWP (so the invalid window is entered), and the Program Counter (PC) and Next Program Counter (nPC) are written in local registers **%l1** and **%l2** respectively (illustrated in red in Figure 3.5).

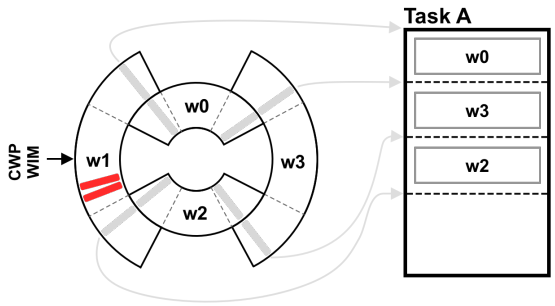
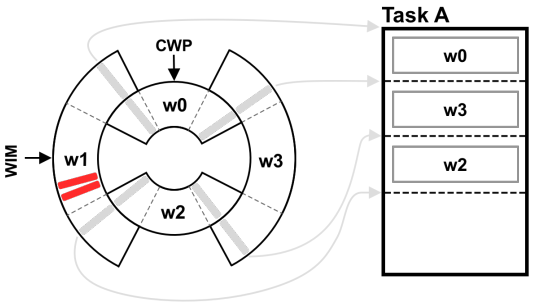


Figure 3.5: Window Overflow Trap.

This is the state, that is prepared by hardware, before execution is resumed at the trap table base address, which will look up the corresponding handler address in the trap table, and copy the PSR into local register **%l0** (see section 3.3). It will then jump to the window overflow trap handler code, whose address is installed in the trap table.

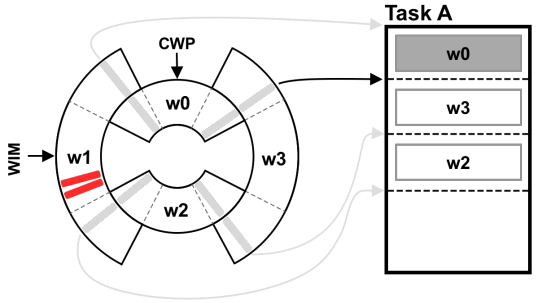
From here on, the function shown in Appendix A.1 will be executed. The code will be divided and shown alongside the visualisation of the algorithm, to ease understanding (the line numbers correspond to their original in the full listing).




```
2  save
```

Figure 3.6: Entering already used window.

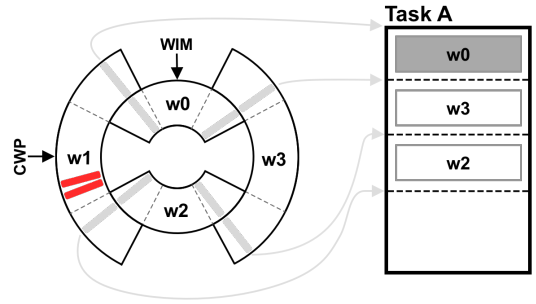
Through a **save** instruction, **w0** is entered (which already holds valid data). The stack pointer of this window still points to the end of the frame, which was reserved for **w0** when it was entered.




```
3   std         %l0, [%sp + 0x00]
4   std         %l2, [%sp + 0x08]
5   std         %l4, [%sp + 0x10]
6   std         %l6, [%sp + 0x18]
7   std         %i0, [%sp + 0x20]
8   std         %i2, [%sp + 0x28]
9   std         %i4, [%sp + 0x30]
10  std         %i6, [%sp + 0x38]
```

Figure 3.7: Saving input and local registers to memory.

Through the use of mentioned stack pointer, the input and local registers of window **w0** are saved to their respective positions within the stack frame reserved for this window (the position for each register relative to the stack pointer is defined in the SPARC ABI [19]).




```
13  set __leon_nwindows_min1, %l0
14  ld      [%l0], %l1
15  rd      %wim, %l3
16  sll     %l3, %l1, %l2
17  srl     %l3, 1, %l3
18  wr      %l3, %l2, %wim
19  nop
20  nop
21  nop
22
23  restore
```

Figure 3.8: Rotating WIM and incrementing CWP.

In the next part of the trap handler, the WIM is readjusted to point to the next window. This is done in multiple steps:

Line 13 Set the address of a specific memory location, where the number of windows in the system minus one is stored, into register **%l0**

Line 14 Load the number of windows minus one from address **%l0** into **%l1**

Line 15 Read current value of WIM into **%l3**

Line 16 Shift current value of WIM to the left by the number of windows minus one and save into **%l2**

Line 17 Shift current value of WIM to the right by one and save into **%l3**

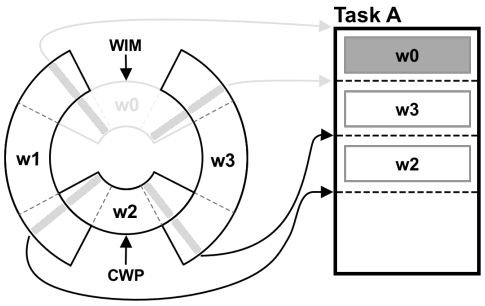
Line 18 Take the value in **%l3**, execute a logical OR with **%l2**, and store the result in WIM

Line 19-21 Wait for three clock cycles. This is the maximum time it can take for the new value of WIM to be systemwide ready [21]

Line 23 Through a restore operation, set back CWP to the window that was entered by the trap

There are multiple things worth mentioning in this part of the algorithm. The

seemingly complicated readjustment of WIM is caused by the fact, that a potential underflow of the invalid bit must be taken care of. In case that WIM is set to window **w0** (bit 0 set in WIM), the next invalid window would be **w3** (bit 4 set in WIM). Therefore, the value of the current invalid window is rotated once to the right, and number of windows minus one times to the left, and those two values are combined through a logical OR. In case of WIM $\neq$ **w0**, this will result in two bits set after this OR operation, but one of them (the one with higher significance) will be outside of the range of valid windows, and therefore a write to that bit in WIM will be ignored.




```
25  jmp     %l1
26   rett   %l2
```

Figure 3.9: Trap epilogue.

The code shown in Figure 3.9 is the standard trap epilogue as defined in [21]. The jump (**jmp**) instruction will cause execution to continue at the address given in the supplied register operand (in this case **%l1** which holds the PC from before the trap). However, before resuming at this address, the instruction in the delay slot is executed. The return-from-trap (**rett**) instruction increments the CWP, so that it points to the window used before the trap happened (refer to Figure 3.4), and enables traps again.

The PC points to the instruction that caused the trap (in this case a **save** instruction), which will be reexecuted after the trap. Since the WIM was adjusted to point to the next window, this time, the **save** instruction can be done without causing a trap, which leads to the state shown in 3.10.

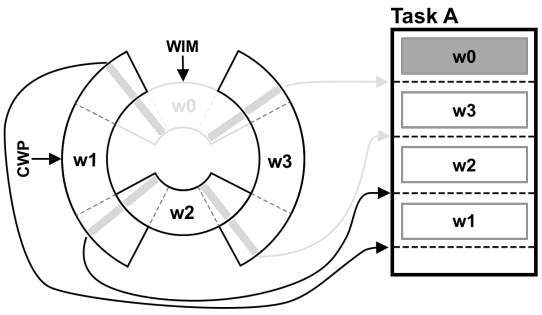


Figure 3.10: Re-executing SAVE instruction.

3.3.3 Window underflow trap

Similarly to the procedure mentioned in the section before, when a **restore** instruction executes, the current value of the CWP is compared against the WIM. If the **restore** instruction would cause the CWP to point to an invalid register set, that is, one whose corresponding WIM bit equals 1 (WIM[CWP] = 1), a window underflow trap is asserted. [21]

From a logical point of view, the window underflow trap must restore the input and local registers of a window, which where previously saved on the stack by either a window overflow trap, or the context switching mechanism. The full algorithm can be seen in appendix A.2.

The algorithm is broken down into different steps again to ease understanding.

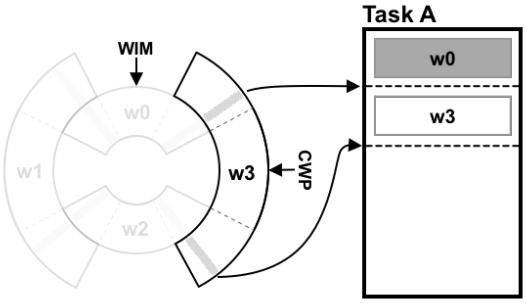


Figure 3.11: Register file setup before window underflow.

The status of the register file before a window underflow is illustrated in Figure 3.11. The CWP is pointing to **w3**, and a stack frame is reserved for it on the task's stack. It can be seen, that there is another frame already on the stack (**w0**), which is currently not in a register window. This situation occurs, for example, if previously the task used up all the available register windows, so that a *window overflow trap* was caused, which put the registers of **w0** onto the task's stack.

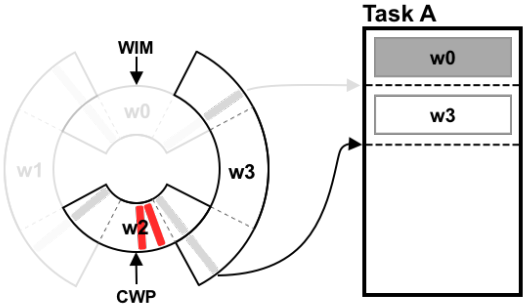
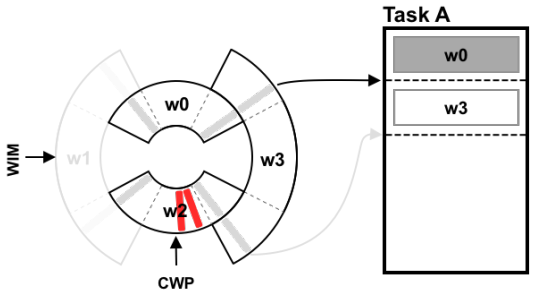


Figure 3.12: Register file setup before entering window underflow trap handler.

If another **restore** instruction is executed, the CWP aligns with the WIM, which

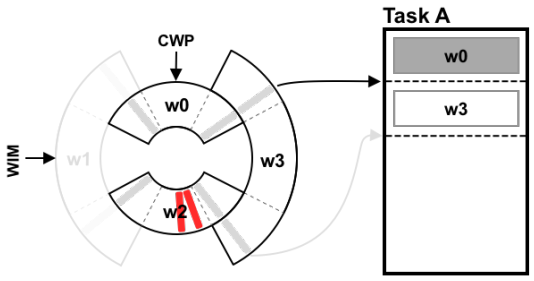
causes a *window underflow trap*. Similar to the *window overflow trap*, the hardware responds by decreasing the CWP by one (now pointing to **w2**), putting the PC and nPC into registers **%l1** and **%l2** respectively, before calling the *window underflow trap handler* routine installed in the trap table. This setup is shown in Figure 3.12.




```
2  set __leon_nwindows_min1, %l6
3  ld  [%l6], %l7
4  rd  %wim, %l3
5  sll %l3, 1, %l4
6  srl %l3, %l7, %l5
7  wr  %l4, %l5, %wim
8  nop
9  nop
10 nop
```

Figure 3.13: Adjusting WIM.

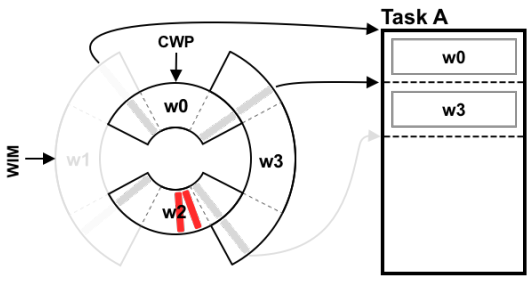
As illustrated in Figure 3.13, as a first step, the WIM is rotated once to the left. The mathematical background is the same as for the right rotation mentioned before in the *window overflow handler*. The current set bit in WIM is shifted once to the left, and number of windows minus one to the right. Those two numbers are combined through logical OR and written back into WIM. Three **nop** instructions follow the write to WIM, since it may take up to three clock cycles for the updated WIM to propagate through the system.




```
12 restore
13 restore
```

Figure 3.14: Entering underflowed window.

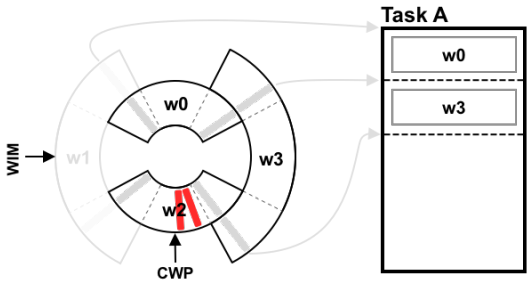
Next, the window which needs to be filled with values from the stack memory must be entered. Therefore, two consecutive **restore** instructions are issued as depicted in Figure 3.14.




```
15  ldd         [%sp + 0x00], %l0
16  ldd         [%sp + 0x08], %l2
17  ldd         [%sp + 0x10], %l4
18  ldd         [%sp + 0x18], %l6
19  ldd         [%sp + 0x20], %i0
20  ldd         [%sp + 0x28], %i2
21  ldd         [%sp + 0x30], %i4
22  ldd         [%sp + 0x38], %i6
```

Figure 3.15: Filling window with values from stack.

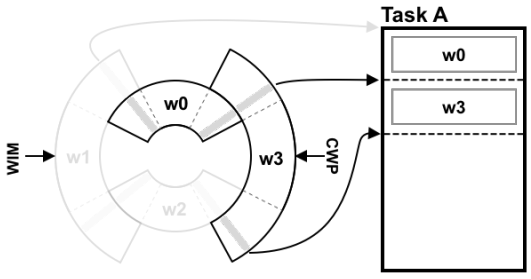
Since the frame pointer of the adjacent window is also the stack pointer of the current window, **%sp** can be used to read the before saved register values from the stack memory and load them into their respective registers. This step can easily be understood by comparing Figure 3.15 with 3.7 from the *window overflow handler* routine, since it is literally just the reversion of that step.




```
24  save
25  save
```

Figure 3.16: Going back to original trap window.

In order to continue the execution of the trapped task, the CWP must be adjusted to point to the window that was entered upon the start of the *window underflow handler*. Therefore, two consecutive **save** instructions are issued as illustrated in Figure 3.16.




```
27  jmp         %l1
28   rett         %l2
```

Figure 3.17: Standard trap epilogue.

The trap epilogue (Figure 3.17) is the same as for the *window overflow handler*.

3.4 Context switching

One of the most important abilities of an OS is to switch from one executing thread to another. This mechanism allows for multiple different processing jobs to run on a single execution platform. At this stage in the porting process, a cooperative context switching mechanism is added to the architecture port. At a later stage, this algorithm is extended to allow for preemptive context switches as well.

Whenever the currently running thread decides to give up executing (e.g. by issuing a call to **k__yield()**, or trying to get access to a shared resource (semaphore, mutex, etc.) that is already in use), the operating system routine

```
unsigned int __swap(int key)
```

is called internally. The argument *key* is the current interrupt level, which will be restored once the current thread starts executing again in the future.

In order to illustrate the algorithm, it is assumed, that the current running thread (Task A) is yielding, and another thread (Task B) is the next, ready to run thread, waiting to start executing. The algorithm will only be presented visually here, the respective code can be found in Appendix A.3.

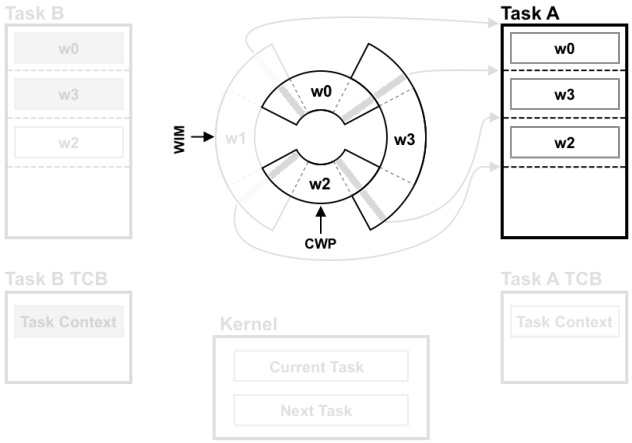


Figure 3.18: Task A is yielding to Task B.

Upon entering the **___swap** routine, Task A has been running for a while, and is currently using the first three register windows. As can be seen in Figure 3.18, another **save** instruction would cause a *window overflow trap*. Each task has a certain memory area reserved as its stack, and a Task Control Block (TCB), where important task parameters (name, priority, etc.) are saved. Tasks can be uniquely distinguished by their TCBs, therefore, the kernel holds a reference to all tasks in the system in a queue-like structure.

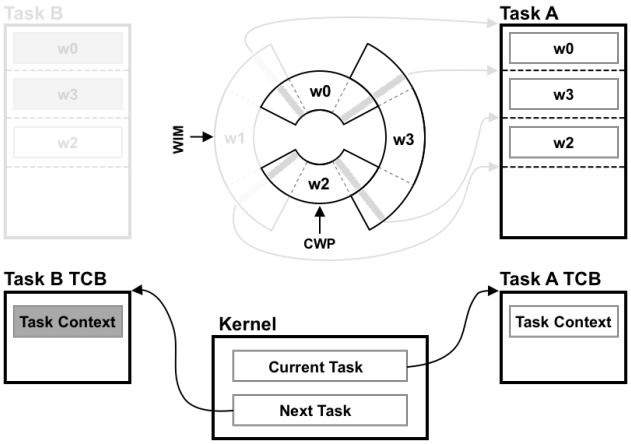


Figure 3.19: The Kernel structure holds pointers to the TCBs of Task A and Task B.

The kernel structure offers two entries, which are important for the context switching algorithm. There is a pointer to the TCB of the current running task, and a pointer to a queue with all tasks, that are currently ready to run. The tasks in this queue are ordered according to the used scheduling policy (e.g. rate monotonic). The first element of this queue therefore points to the TCB of the task, that shall take over the Central Processing Unit (CPU) next. In Figure 3.19, this first element of the queue is referred to as *Next Task*.

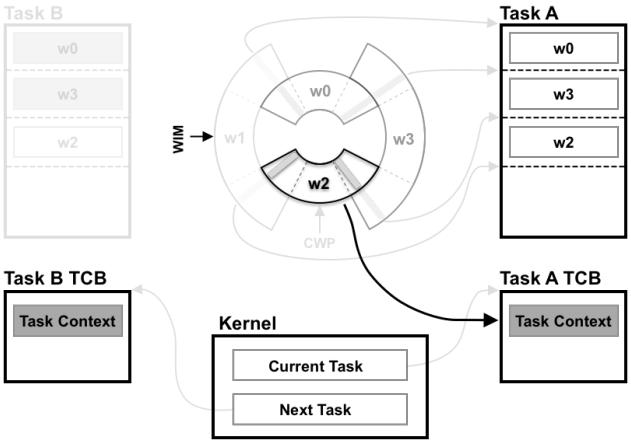


Figure 3.20: Task A's context is saved in its TCB.

As a first step, the context of the currently running task is saved in its TCB. The context of a task includes:

- The input registers **%i0 - %i7** of the current window
- The local registers **%l0 - %l7** of the current window

- The output registers **%o6** and **%o7** of the current window
- The PSR
- The *key* argument provided to the **___swap** routine
- The standard return value for the swap method

The standard return value is used as a way to indicate to a task, that certain states might have changed while the task was out of execution. It is used internally by the OS. The reason for only saving the output registers **%o6** (the stackpointer) and **%o7** (the return address for the most recent **call** instruction) is, that all other output registers are volatile across function calls. Since the context switch is cooperative, these registers, together with the **%y** and the global registers, do not have to be considered. In the given example, illustrated in Figure 3.20, the respective contents of **w2** are saved in Task A's TCB, together with the PSR, key, and standard return value.

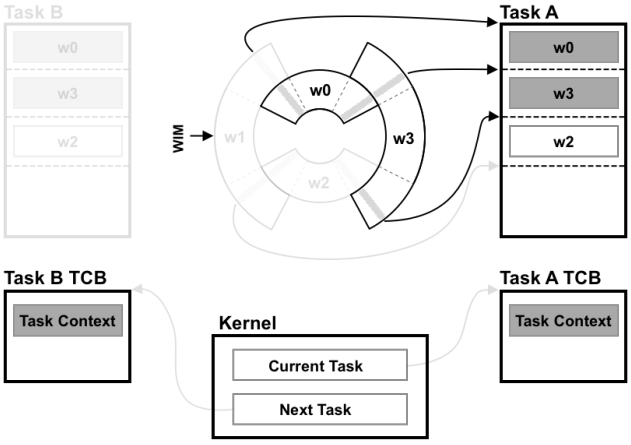


Figure 3.21: Window saving loop.

After storing the context, there might be multiple windows left, which were used by Task A. In a loop, these register windows are stored to the task's stack memory, similar to a repeated *window overflow*. Before going into the loop, traps must be disabled (since the CWP has to be manipulated to access the different windows).

The algorithm checks before each **restore** instruction, if switching to the preceding window would cause a *window underflow trap*. In that case, all used register windows have been stored to the stack memory, and Task A is therefore fully removed from the execution environment.

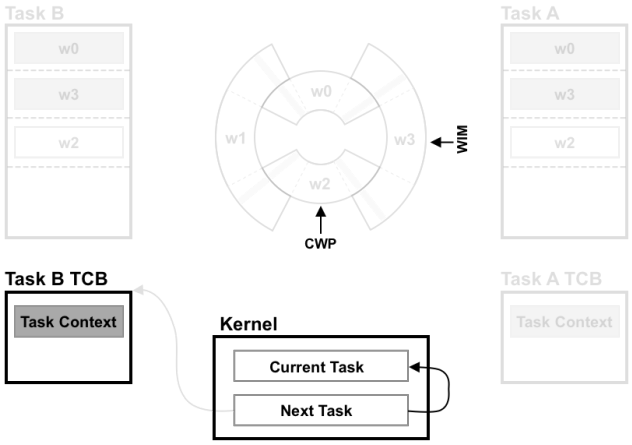


Figure 3.22: Setting new Current Task pointer.

After leaving the register window saving loop, the CWP is restored and the WIM is set to the preceding window. This is a stylistic choice, one could also just leave the CWP pointing to the latest used window during the saving loop.

At this stage, Task A is fully removed from the execution environment, and therefore no reference to its TCB is needed anymore. Thus, the *Current Task* field in the kernel structure can be updated to point to the TCB of the next to run task (see Figure 3.22).

Figure 3.22 shows the final configuration of the register file after saving all register windows, and resetting CWP and WIM.

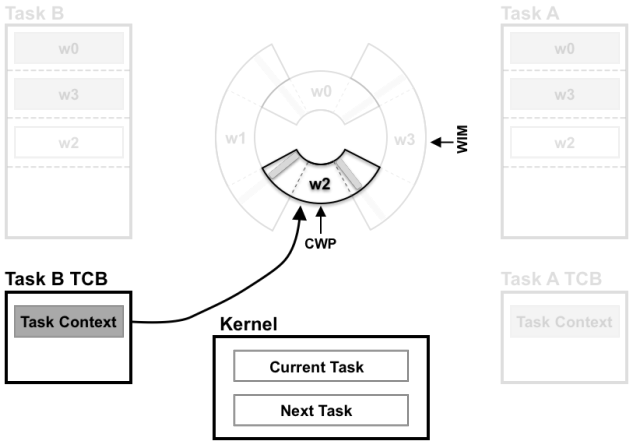


Figure 3.23: Filling in Task B's context into the register window.

From here on, the new task (Task B) needs to be put into an executable state. Therefore, the *Task Context* is loaded from the TCB of Task B. The respective contents of the saved register window (**%i0 - %i7**, **%l0 - %l7**, **%o6**, **%o7**) are

loaded into the current window, the PSR is restored (with adjusted CWP, this also enables traps again), the key is used to set the correct interrupt level, and the return value for the **___swap** function is set according to the value saved in the TCB.

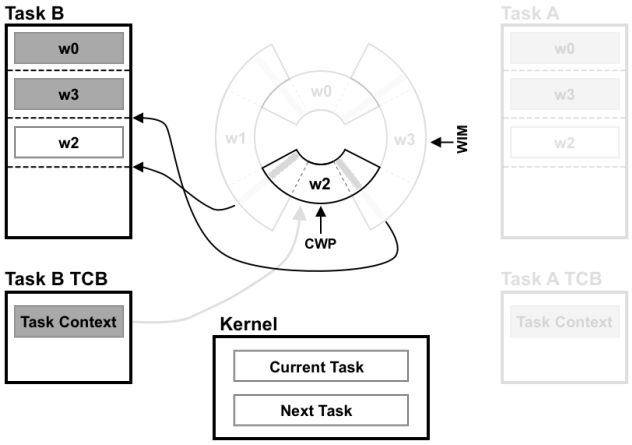


Figure 3.24: Task B is switched in and ready to run.

As can be seen in Figure 3.24, filling the current window with the register values from the TCB automatically also sets the correct pointers to the task's stack memory. At this stage, the **___swap** routine can issue a **jmp** instruction to continue executing Task B.

It is worth mentioning, that only one (i.e. the most recent) window is restored for switching in a new task. That makes it clear, why the WIM has been reset before. If the newly switched in task issues a **restore** operation, a *window underflow trap* will restore the preceding window. This has multiple advantages, such as a deterministic timing behaviour for switching in tasks (OBSERVE: the switching out of a task is not deterministic, since the number of windows to store in the saving loop cannot be known prior to runtime).

3.5 Thread creation

After introducing the context switching algorithm in the previous section, it is time to add the ability to create new threads. This is part of the architecture port, since the context of a task is architecture dependent. Internally, a new thread is created by the operating system by calling the following function:

```
void _new_thread(struct k_thread *thread, k_thread_stack_t *stack,
                 size_t stack_size, k_thread_entry_t thread_func,
                 void *parameter1, void *parameter2, void *parameter3,
                 int priority, unsigned int options)
```

This function must be implemented by the architecture port. Only a high level description of the different steps is given here:

1. The task's stack memory must be allocated according to **stack_size**
2. The stack memory pointer **stack** must be pointed to the task's stack
3. The internal OS function **__new__thread__init** must be called to register the task's TCB within the OS
4. The context (as described in 3.4) in the task's TCB must be set up as if the task was switched out by the **___swap** routine

By setting up the context field in the task's TCB as if the task was switched out by the context switching mechanism, it can be switched in by the very same when the task is ready to run.

3.6 Device drivers

The amount of device drivers included in a final application depends on the requirements of said application. Zephyr supports a wide range of high level drivers (e.g. communication stacks like Bluetooth, CAN, I2C, etc.), which can be added before compilation through the *kconfig*[1] system.

It is debatable, if these device drivers are actually part of the operating system, or if they should be seen as add-on libraries.

For a functional port of the RTOS, only the drivers for two distinct peripherals are needed. These are a driver for the interrupt controller of the processor, and the timer, which will be presented in the following sections.

3.6.1 Interrupt controller

As mentioned in section 2.2.1.2, whenever an interrupting trap happens, the trap table dispatch mechanism will load the *interrupt trap handler* routine according to the interrupt line that was triggered. It is a possibility, to put different trap handlers for different interrupt lines into the mentioned subtable directly. However, this would mean, that the respective interrupt service routine is tied to that specific interrupt line.

The IRQMP [9] interrupt controller comes with the ability, to dynamically map a wider range of interrupt request lines into the 16 interrupt levels available at the processor side.

[1] Kconfig is a configuration system, which was originally developed for the Linux kernel. It lets the user choose different configuration options, either through a supplied *.config* file, or through a graphical user interface (*menuconfig*)

In order to facilitate the dynamic mapping feature, and to make it possible to add interrupt service routines at runtime, all 16 entries of the subtable in the trap table will point to the same interrupt dispatching routine, which will configure the system for interrupt execution, and then load the respective interrupt handler. The full code of the interrupt dispatcher is shown in Appendix A.4. Here, only a high level description of the algorithm is given:

1. As with any other trap, the interrupt will cause the CWP to be decremented by one. If this means, that CWP now points to an invalid window, do a manual window overflow
2. Create a special *interrupt stack frame* on the interrupted task's stack, where all registers are saved, that are volatile across function calls (global registers **%g1** - **%g5**, input registers **%i0** - **%i7**, **%y**)
3. Switch to the system's dedicated interrupt stack
4. Raise PIL so that no other interrupt can be issued while the current one is executing
5. Enable traps
6. Use the trap type in **%l3** to find the correct interrupt service routine and execute it
7. If preemption is enabled, check if a context switch must be issued, and if so, call **___swap**
8. Recreate the previous state by setting the volatile registers from the values in the *interrupt stack frame*
9. If returning from trap would cause CWP to point to an invalid window, do a manual window underflow
10. Return from trap

Step 6 can be done in a high level C function, by using the trap type to find the respective interrupt service routine in a linked list. This makes it easy, to include new service routines during runtime.

3.6.2 Timer

The timer device driver is used to generate the RTOS tick interrupt, which can be seen as the heart beat of the operating system.

There are three important routines, that must be implemented by the architecture port.

```
static void _timer_isr(void* arg)
```

This method is the actual interrupt service routine, which will be called every time the counter value for the used timer underflows. All this function does is clearing the respective interrupt pending bit and calling **z__clock__announce()** to inform the kernel about the time that has progressed.

```
uint32_t z_clock_elapsed(void)
```

A call to **z__clock__elapsed** returns the number of ticks, that occurred since the last time the current tick value was announced by the timer driver.

```
uint32_t z_clock_driver_init(struct device *device)
```

This function is used to initialise the timer driver. The **device** structure holds pointers to the configuration registers of the timer peripheral. Within this function, the pointer to **__timer__isr** is added to the linked list of interrupt service routines mentioned in section 3.6.1.

3.7 CPU idling

Whenever there is no thread, that is ready to take over the CPU, the context switching mechanism switches in an *idle task*, which is automatically created by the RTOS during system boot up. It is possible to define a job, that shall be done whenever the system is idle, otherwise, the system can either be kept in a spinning state (i.e. an infinite loop), or be put into a power saving mode.

The SPARC ISA does not define a dedicated power saving mode, however, the LEON can be configured to include such a feature. If available on the respective hardware implementation, these features can be added in the SoC port. For a generic SPARC implementation, the following function is included in the architecture port:

```
void __weak k_cpu_idle(void)
{
        /* Do nothing but unconditionally unlock interrupts and return to the
         * caller.
         */
        leon_set_pil_inline(0);
}
```

The function is defined with the **___weak** attribute, so that it can be overwritten by an implementation in the SoC port. As specified in [6], the GR716 offers a low power mode by writing a zero value into **%asr19**:

```
static ALWAYS_INLINE void leon_idle(unsigned int key)
{
        irq_unlock(key);
        __asm__ volatile("wr  %g0, %asr19");
}

void k_cpu_idle(void)
{
        leon_idle(0);
}
```

Calling **leon_idle** with argument 0 causes all interrupt lines to be unlocked, so that any interrupt (especially the timer tick interrupt) can wake up the processor again.

3.8 Linker scripts and toolchain

Since Cobham Gaisler offers a GCC based development environment ([5]) for the LEON3 processor core, the cross-compile mechanism offered by the Zephyr build system can be used. This is done by setting two different environment variables, before issuing any commands of the build system:

```
export ZEPHYR_TOOLCHAIN_VARIANT=cross-compile
export CROSS_COMPILE=/opt/bcc-2.0.5-gcc/bin/sparc-gaisler-elf-
```

The second line of the above bash code snippet must be adjusted to point to the bin folder of the installed BCC compiler [5].

The linker script is used to "glue together" the final application. It is possible to divide the linker script into different parts, one for the architecture, one for the specific SoC, and one for the board in use. The Zephyr build system will look for a linker script in the board folder first, if none is found there, it will continue looking in the SoC folder, and the architecture folder, consecutively.

The linker script is also responsible for setting up the application in a way, so that it can be loaded into the microcontroller later on (e.g. by setting the entry point as described in section 3.2).

CHAPTER 4

Window Partitioning

The SPARC ISA only specifies, that the number of available register windows must be between 2 and 32. Most commonly, processor designs based on the SPARC specification feature 8 windows, which has empirically proven to be a good compromise between processor design complexity and typical function call depths.

Especially with the GR716 in mind (which features 31 register windows), a different approach to context switching might be advisable, since in the worst case, there could be 30 windows (one is always marked as invalid), that need to be saved to a task's stack when the task is switched out by the scheduler.

Thus, the LEON3 can be configured to include a *window partitioning* mechanism. As described in [9], the ancillary state register **%asr20** can be used to separate the register window into multiple, independent parts. Register **%asr20** features a 5 bit field, which encodes the start window of the partition, and a 5 bit field for the maximum number of CWP within this partition (i.e. size of the partition minus one).

By changing these fields in **%asr20**, the output registers of the last register window of the partition are mapped to the input registers of the first window, effectively creating a (smaller) register window circle. Through this feature, it is possible to divide the overall windowed register file into multiple partitions.

In the following two sections, the interrupt trap handler, and the context switching algorithm are adapted to make use of the window partitioning mechanism.

4.1 Interrupt handling

The algorithm described in section A.4 is changed in the following way:

1. Check if we are in the invalid window, if so, do a manual window overflow
2. Save the global registers and **%y** in an interrupt stack frame on the interrupted task's stack (the input registers do not have to be saved, since the rest of the ISR will run in a dedicated partition)
3. Store the current CWP and WIM in the TCB of the interrupted task
4. Switch to dedicated interrupt partition
5. Setup stackpointer to point to system's interrupt stack
6. Look up ISR for the given trap type and execute it
7. Switch back to task partition (and set up CWP and WIM again)
8. If preemption is enabled, check if a context switch is needed, and if so, call **___swap**
9. Reinstall values from interrupt stack frame
10. If returning from trap would cause the CWP to align with the invalid window, do a manual window underflow
11. Return from trap

The advantage of running the ISR in a dedicated partition is the deterministic way of execution. Before, the performance of an ISR was dependent on how many unused windows there were still left. In the worst case (i.e. when the interrupt trap already enters the invalid window) each subroutine call in the ISR will cause a window overflow trap, which decreases the performance of the interrupt routine significantly. By choosing the size of the interrupt partition according to the ISR with the deepest call depth, it can be guaranteed, that all ISRs can run without a single window overflow. The code for the interrupt trap with window partitioning is shown in Appendix A.6.

4.2 Context switching

Similar to dedicating a partition to interrupt handling, which was presented in the previous section, the remaining part of the register window might be further divided. By creating dedicated partitions for different tasks, the context switching algorithm can be simplified for the case, that the scheduler shall transfer execution to a task, that has a partition assigned to it.

To understand, how this partitioning simplifies context switching, the principle idea is presented in the following section. The code sample for this algorithm is given in Appendix A.5.

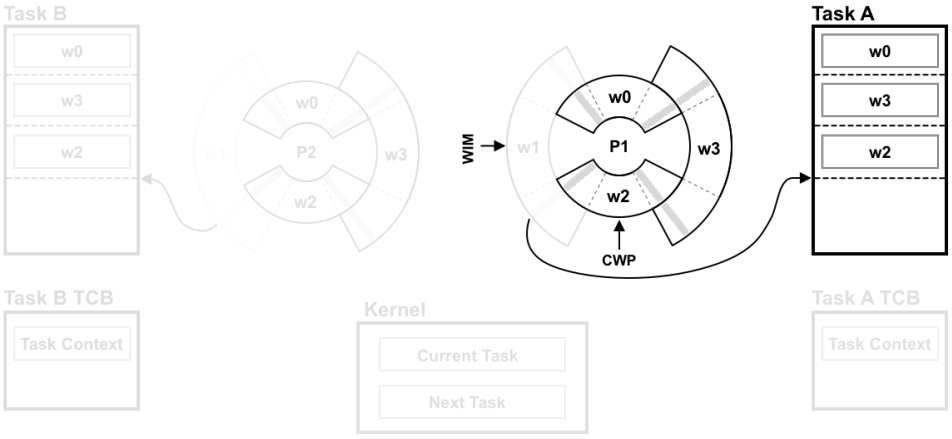


Figure 4.1: Initial state before context switching. Partition P1 is dedicated to Task A, partition P2 is dedicated to Task B.

Figure 4.1 demonstrates the state of the system at the beginning of **___swap**. Task A has been running in its dedicated partition **P1**, and is yielding. The scheduler shall now transfer execution to Task B, which has been running previously in its own dedicated partition **P2**.

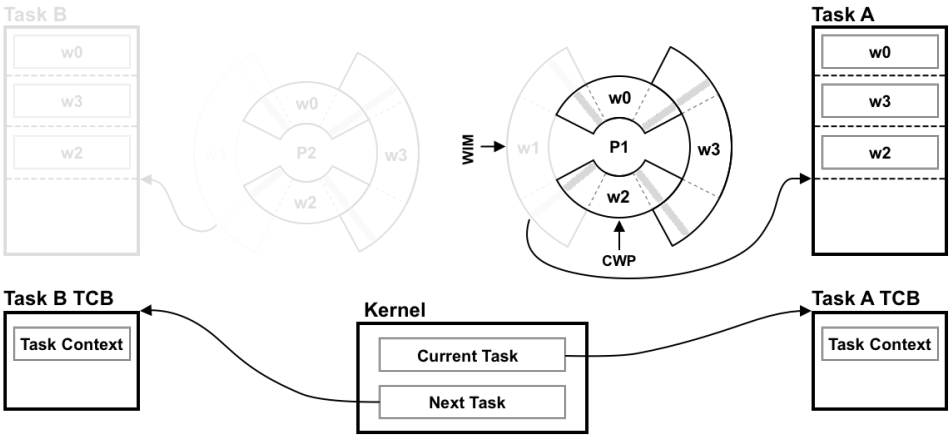


Figure 4.2: Getting a reference to currently running and next to run task.

As a first step, a reference to the TCB of the currently running task's and the next to run task's TCB is taken from the kernel structure as shown in Figure 4.2.

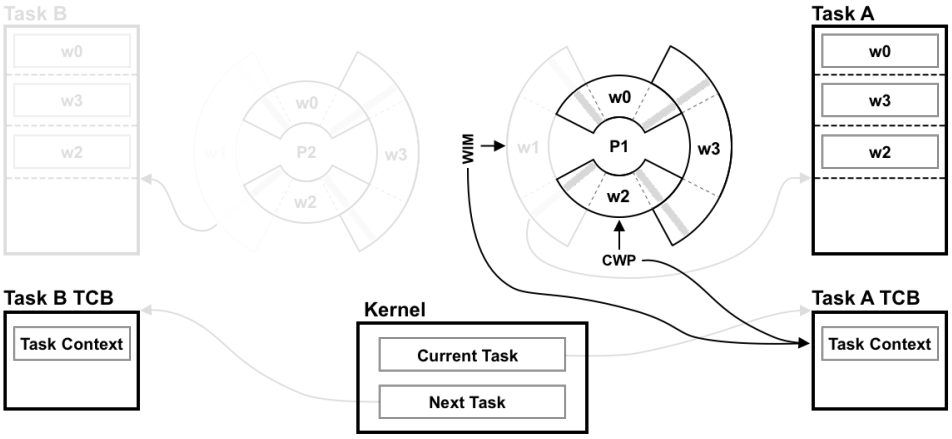


Figure 4.3: Saving the task context.

The CWP, the WIM, the standard return value, and the PSR are saved as Task A's context. No other values have to be saved, since Task A has its own partition dedicated to it, so that the local and input registers can just remain in their respective register window. This step is shown in Figure 4.3.

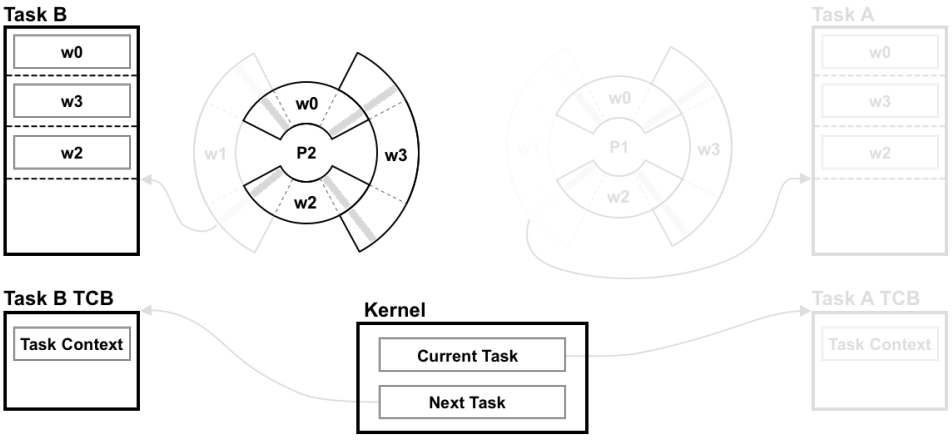


Figure 4.4: Switching partitions.

Now the scheduler can switch to Task B's dedicated partition (Figure 4.4).

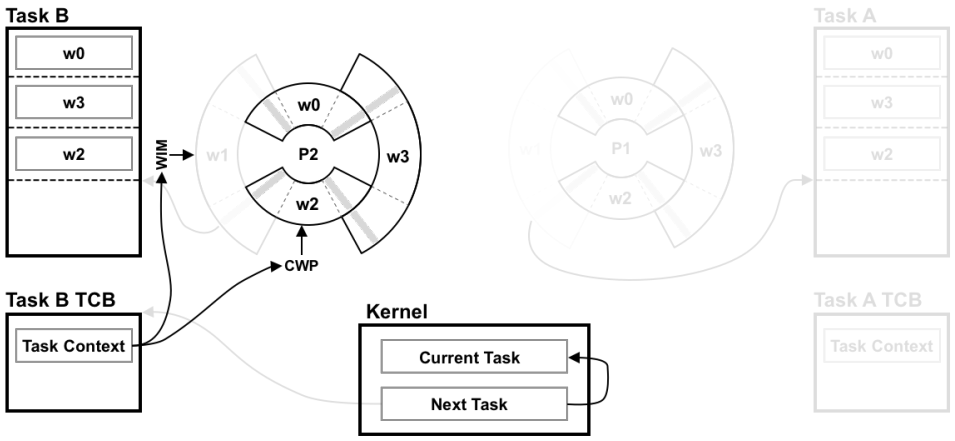


Figure 4.5: Setting up Task B for execution.

Finally, the global CWP and WIM can be set according to the values saved in Task B's TCB. At this stage, Task B is ready to run.

It is clear, that the overall complexity of the context switching mechanism was notably reduced by introducing the partitioning feature. Not a single window had to be written out to or loaded from a task's stack memory, which (as shown in section 6) will greatly reduce the time it takes to switch from one execution thread to another.

The above case only covers the context switch between tasks, which are both running in dedicated partitions. If the overall system only has a very limited set of tasks, it might be feasible to create partitions for each of them. The current implementation offers up to two dedicated partitions, and one obligatory shared partition.

All tasks, which do not have an assigned partition, will run in this shared partition. The implementation of the context switching algorithm was done in a way to achieve lazy context switching behaviour. This means, that upon switching from the shared partition to a dedicated one, the task running in the shared partition will remain there. If at a later stage, execution is transferred back to the task currently in the shared partition, this context switch has the same timing behaviour as if this task would have its own partition (since it is already present in the shared one). Only if another task (i.e. one that is neither in a dedicated nor currently in the shared partition) shall be switched in, the task present in the shared partition will be flushed out, and the new task will be set in. This case is exactly the same as the normal context switch described in section 3.4.

It should be mentioned, that the thread creation mechanism (see section 3.5) must also be altered to facilitate the window partitioning feature. Therefore, the *options* field in the thread creation function is used to define two new options: **K_PART_1** and **K_PART_2**. If a task is created with either option (and the window parti-

tioning is enabled through *kconfig*), it will be put directly into its partition by the thread creation mechanism.

To see an example of how to set up an application with window partitioning, refer to Appendix B.

CHAPTER 5

Performance Evaluation

There are two different aspects for evaluating Zephyr on the LEON platform. First, it must be verified, that the port itself is running correctly on the new platform. Consecutively, it can be evaluated *how well* the port is performing.

The functional verification of the port is done by using Zephyr's testing framework Ztest [20]. Depending on the feature list, that is selected by the architecture's configuration file, Ztest selects different test scenarios out of a pool of test cases, builds the respective code projects, and loads and evaluates the output through the use of a simulator.

For the given platform, Cobham Gaisler's cycle-accurate instruction simulator TSIM [8] has been used extensively during the development process, and finally for the evaluation of the correct functioning of the operating system.

In order to evaluate, how well the system is performing, a measurable quantity has to be defined. For the given architecture port (especially keeping the window partitioning feature in mind), a benchmarking for the context switch timing and the interrupt service timing has been performed. These two benchmark tests were done on the GR716 Mini Board [7].

5.1 LEON benchmarking

In order to evaluate the performance of Zephyr on the GR716, a benchmarking test was performed. This benchmarking test consisted of the sampling of context switch-

ing times, and interrupt latency measurements. The GR716 has tightly-coupled, constant access-time, SRAM for instructions and data [6]. This was used during all performance evaluations. It means, that there are no temporal or spatial variations in execution times caused by cache misses.

For measuring the context switching time, two instances of the same task function were created. Within the task function, an alternating amount of **save** instructions were issued (to simulate different function call depths), before calling **k_yield** to transfer control to the other task. To allow a comparison between different execution scenarios, three different test cases were evaluated. First, the test was done without partitioning enabled. Then, one task was put into a dedicated partition (8 windows), the other was running in the shared partition (also 8 windows). And finally, both tasks got a dedicated partition (with 8 windows each) assigned.

For measuring the interrupt latency, the timestamping feature offered by the interrupt controller [6] is used. When activated, the timestamp automatically records the assertion and acknowledge time of a predefined interrupt. In the interrupt trap handler, those numbers are read out together with the current cycle count just before the respective ISR is entered.

For the purpose of this evaluation, the timestamp has been configured to trigger whenever the timer tick interrupt line is asserted.

CHAPTER 6

Benchmarking Results

Figure 6.1 illustrates the measured context switching times for the three different test cases mentioned in section 5:

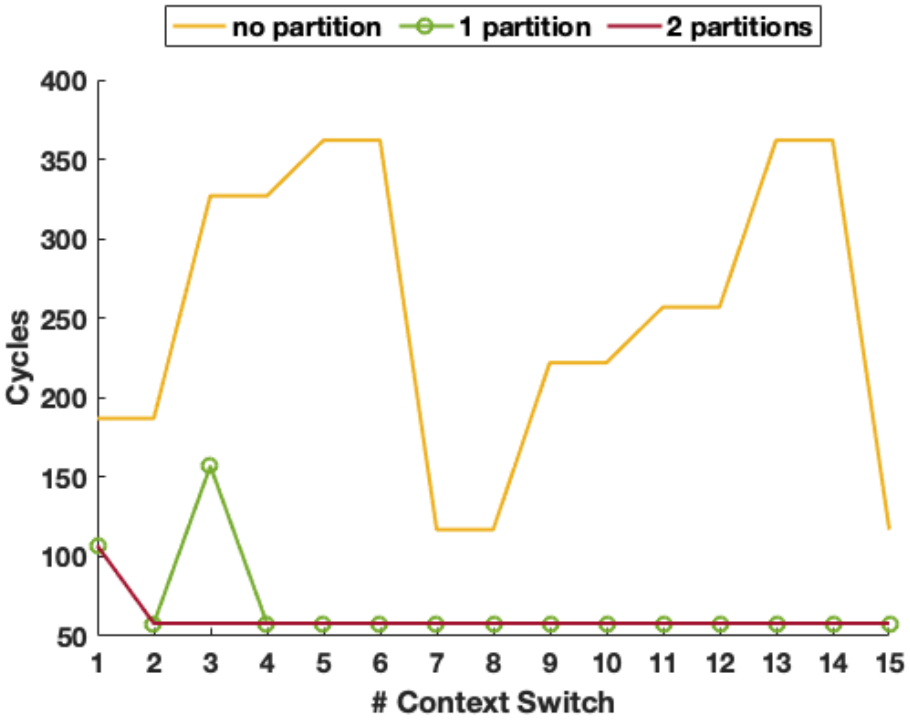


Figure 6.1: Context switch timing benchmark results.

- both tasks running in the same, unpartitioned register file
- one task assigned to a dedicated partition and the other task running in the shared partition
- both tasks assigned to dedicated partitions

As the amount of windows in use changes each time before a task yields, the context switch timing fluctuates when not using the partitioning feature (yellow curve). This is due to the window saving loop having to save a different amount of windows every time.

When assigning both tasks to a dedicated partition (red curve), the context switching time remains constant after the first switch to a dedicated partition (first context switch is done from the main context (in the shared partition) to the first task running in a dedicated partition). Since there are only context switches between the two dedicated partitions afterwards, the context switching time is constant from the second instant on.

Of special interest is the case of one task running in a dedicated partition, and one task running in the shared one (green curve). As can be seen in Figure 6.1, it shows almost the same performance as the aforementioned case of having two dedicated partitions. This behaviour is achieved through the lazy context switching strategy mentioned in section 4.2. The one context switch, that is different (switch number 3 in Figure 6.1), is caused by the switch from the main context to the task context. From here on, it has the same performance as if it was running in a dedicated partition, since it never has to be removed from the shared partition again.

Overall, it is clearly visible, that the window partitioning feature greatly increases the performance of the context switching mechanism. It shall be noted here, that the maximum amount of windows, which had to be saved in these test cases, was 7 windows. In the case of an unpartitioned register file, and if a task would have a very high call depth, up to 29 windows might have to be saved during the window saving loop, which would yield an even worse performance when not using partitioning.

For the evaluation of interrupt latency, two different times are of interest. First, the time between the assertion of the interrupt, and the acknowledgement by the processor is measured through the timestamp feature mentioned in chapter 5. The recorded results are illustrated in Figure 6.2. It can be seen, that there is no difference between the recorded times with regards to partitioning, which is expected, since the time between interrupt assertion and acknowledgement is purely depending on the hardware.

The more interesting timing is the one between the acknowledgement by the processor and the start of the ISR. The recorded times are shown in Figure 6.3. Here, when using partitioning, the minimum and average times slightly increase compared to the case without partitioning. This increase can be expected, since when using partitioning, the added complexity of switching partitions must be accounted for. The benefit of using partitioning, however, is clearly visible through the recorded

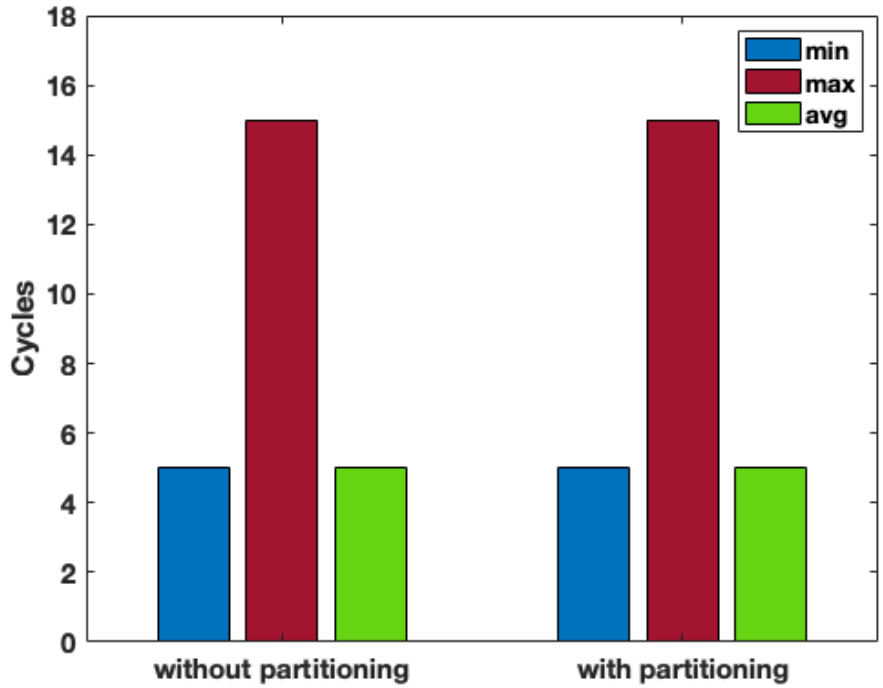


Figure 6.2: Measured timing between timer interrupt assertion and acknowledge for the cases without and with partitioning.

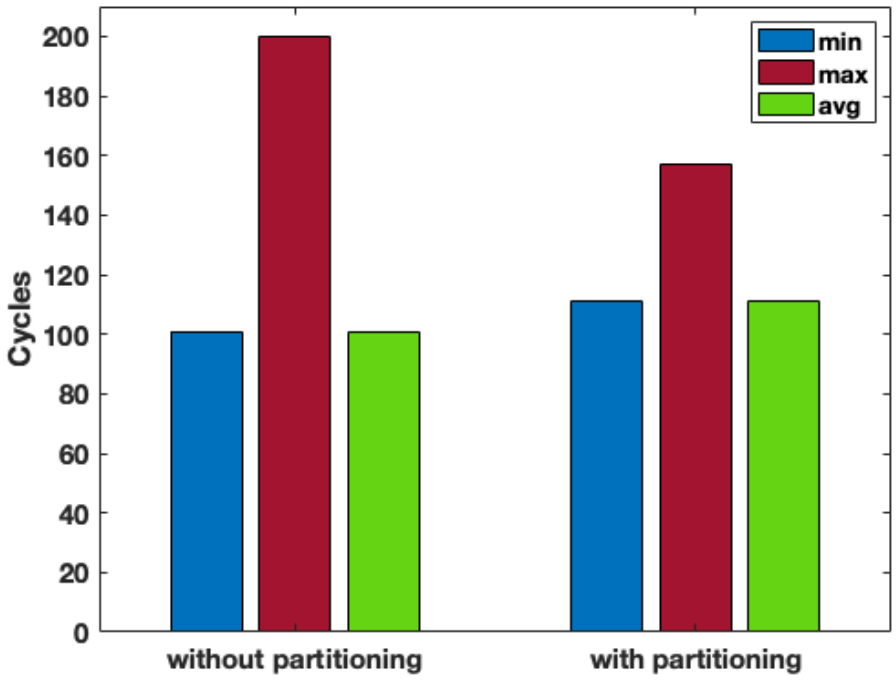


Figure 6.3: Measured timing between timer interrupt acknowledge and start of ISR for the cases without and with partitioning.

maximum times. This timing instant is caused, when the trap enters an invalid window, which means, it has to perform a manual window overflow. This step is the same for both the case with and without partitioning. As described in section 3.6.1, the ISR dispatching is done in a high level C function, therefore, an additional window is used by the interrupt trap handler. When not using partitioning, calling this high level C function will cause a window overflow trap, whose additional time increases the maximum value shown in Figure 6.3. However, when using a sufficiently sized interrupt partition (in this benchmarking 7 windows), the additional call to a C function will work without a window overflow.

It is worth mentioning, that the above illustrated benchmarking only covers the interrupt trap handler, not the ISR performance itself. Since, in the worst case, every additional function call in the service routine would cause a window overflow, the performance of the ISR will significantly decrease when not using partitioning. The added penalty of window overflows can be circumvented by sizing the interrupt partition appropriately for the respective application.

CHAPTER 7

Conclusion

The aim of this thesis project was to create a port of the Zephyr RTOS for the LEON platform, which was previously not supported by this operating system. Through the porting procedure detailed in section 3, a functional state of the operating system was achieved.

Zephyr comes with an extensive testing framework, Ztest, mentioned in chapter 5. Depending on the architecture settings (e.g. the included peripherals), Ztest automatically selects a number of tests to run whenever the tool is invoked.

The current implementation passes 84 out of the 91 selected tests. All test cases that are failing, are due to unimplemented features, such as stack sentinel checking, XIP, or POSIX compliance. The port has proven to be stable during all performed tests and sample applications.

As shown in chapter 6, through the introduction of window partitioning, the performance of the context switching mechanism could be increased significantly. It is worth mentioning, that the current implementation only offers cooperative context switching in a deterministic way (see chapter 8).

Overall, Zephyr is a good fit for the GR716, considering footprint and processing capability.

CHAPTER 8

Future work

This section shall give a short overview of the next steps, that could be taken to enhance the current version of the Zephyr port for the LEON platform.

- **Device drivers**

During this thesis project, only a minimal amount of device drivers was developed. Currently, only the IRQMP interrupt controller, the timer, and a basic UART driver are included. Especially drivers for common communication stacks (e.g. CAN), and the memory management unit could be interesting to include.

- **Improve interrupt handling**

The current implementation of the interrupt handler trap with partitioning still needs to check, if the trap causes the CWP to point to an invalid window, and if so, has to issue a manual window overflow before executing the actual interrupt handling. This has multiple disadvantages, such as a non-deterministic timing behaviour of the trap handler itself, and further also a change in timing for the interrupted task (since it will have to do a window underflow at a certain point to gain back the data).

In theory, there is no problem using the invalid window of a task partition to switch to the interrupt partition, issues occur when a call to **___swap** is done at the end of the interrupt. This function has an input parameter, which will overwrite data in the adjacent window of the task partition. It might be possible, to rewrite the interrupt trap handler and the context switching routine to only use global registers, and calling **___swap** from within the interrupt partition.

Support for interrupt nesting could also be implemented.

- **SoC port for GR712**

Currently, the only specific LEON implementation supported is the GR716. Other processors, such as the GR712, would need their own SoC port. This would be especially interesting, since the GR716 is a single-core, and the GR712 a dual-core processor. Thus, parts of the architecture port will most likely need adjustment to allow for Symmetric-Multi-Processing (SMP) as well.

Bibliography

[1] *Architecture Porting Guide*. 2019. URL: https://docs.zephyrproject.org/latest/guides/porting/arch.html (visited on 06/03/2019).

[2] M. Barr and A. Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, 2006.

[3] B.J. Catanzaro. *The SPARC Technical Papers*. Sun Technical Reference Library. Springer New York, 2012. URL: https://books.google.se/books?id=ESLoBwAAQBAJ.

[4] Jens Eickhoff. *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*. Springer Publishing Company, Incorporated, 2011.

[5] Cobham Gaisler. *BCC User's Manual*. 2019. URL: https://www.gaisler.com/doc/bcc2.pdf (visited on 04/18/2019).

[6] Cobham Gaisler. *GR716 - 2019 Advanced Data Sheet and User Manual*. 2019. URL: https://www.gaisler.com/doc/gr716/gr716-ds-um.pdf (visited on 05/29/2019).

[7] Cobham Gaisler. *GR716-MINI Development Board User's Manual*. 2019. URL: https://www.gaisler.com/doc/gr716/GR716-MINI_user_manual.pdf (visited on 08/20/2019).

[8] Cobham Gaisler. *TSIM2 Simulator User's Manual*. 2019. URL: https://www.gaisler.com/doc/tsim-2.0.64.pdf (visited on 05/29/2019).

[9] Gobham Gaisler. *GRLIB IP Library User's Manual*. 2018. URL: https://www.gaisler.com/products/grlib/grlib.pdf (visited on 05/10/2019).

[10] *Getting Started Guide*. 2019. URL: https://docs.zephyrproject.org/latest/getting_started/index.html (visited on 08/16/2019).

[11] *Introduction*. 2019. URL: https://docs.zephyrproject.org/latest/introduction/index.html (visited on 07/15/2019).

[12] David Keppel. *Register Windows and User-Space Threads on the SPARC*. Tech. rep. Department of Computer Science and Engineering University of Washington, 1991.

[13] Jochen Liedtke. *Lazy Context Switching Algorithms for Sparc-like Processors*. Tech. rep. German National Research Center for Computer Science, 1993.

[14] *Meet Linux's little brother: Zephyr, a tiny open-source IoT RTOS*. 2016. URL: http://linuxgizmos.com/zephyr-a-tiny-open-source-iot-rtos/ (visited on 07/02/2019).

[15] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[16] K.G. Shin and P. Ramanathan. "Real-time computing: a new discipline of computer science and engineering". In: *Proceedings of the IEEE 82,* 1 (1994), pp. 6–24. URL: http://ieeexplore.ieee.org/iel1/5/6554/00259423.pdf.

[17] *SPARC-V8 Suppliment. SPARC-V8 Embedded (V8E) Architecture Specification*. 1st. 1996.

[18] W. Stallings. *Operating Systems: Internals and Design Principles*. 4th. Prentice Hall, 2001.

[19] *SYSTEM V APPLICATION BINARY INTERFACE. SPARC Processor Supplement*. 3rd. 1996. URL: https://www.gaisler.com/doc/sparc-abi.pdf (visited on 05/29/2019).

[20] *Test Framework*. 2019. URL: https://docs.zephyrproject.org/1.12.0/subsystems/test/ztest.html (visited on 08/01/2019).

[21] *The SPARC Architecture Manual. Version 8*. SPARC International Inc. 535 Middlefield Road, Suite 210 Menlo Park, CA, 1992. URL: https://www.gaisler.com/doc/sparcv8.pdf (visited on 05/25/2019).

[22] *WHAT IS THE ZEPHYR PROJECT?* 2019. URL: https://www.zephyrproject.org/what-is-zephyr/ (visited on 05/15/2019).

[23] *Wind River Welcomes Linux Foundation's Zephyr Project*. 2016. URL: https://blogs.windriver.com/wind_river_blog/2016/02/wind-river-welcomes-linux-foundations-zephyr-project.html (visited on 06/12/2019).

APPENDIX A

Code Samples

A.1 Window Overflow

```
1  FUNC_BEGIN __leon_trap_window_overflow
2          save
3          std    %l0, [%sp + 0x00]
4          std    %l2, [%sp + 0x08]
5          std    %l4, [%sp + 0x10]
6          std    %l6, [%sp + 0x18]
7          std    %i0, [%sp + 0x20]
8          std    %i2, [%sp + 0x28]
9          std    %i4, [%sp + 0x30]
10         std    %i6, [%sp + 0x38]
11
12
13         set    __leon_nwindows_min1, %l0
14         ld     [%l0], %l1
15         rd     %wim, %l3
16         sll    %l3, %l1, %l2
17         srl    %l3, 1, %l3
18         wr     %l3, %l2, %wim
19         nop
20         nop
21         nop
22
23         restore
24
25         jmp    %l1
26          rett  %l2
27 FUNC_END __leon_trap_window_overflow
```

A.2 Window Underflow

```
1  FUNC_BEGIN __leon_trap_window_underflow
2          set    __leon_nwindows_min1, %l6
3          ld     [%l6], %l7
4          rd     %wim, %l3
5          sll    %l3, 1, %l4
6          srl    %l3, %l7, %l5
7          wr     %l4, %l5, %wim
8          nop
9          nop
10         nop
11
12         restore
13         restore
14
15         ldd    [%sp + 0x00], %l0
16         ldd    [%sp + 0x08], %l2
17         ldd    [%sp + 0x10], %l4
18         ldd    [%sp + 0x18], %l6
19         ldd    [%sp + 0x20], %i0
20         ldd    [%sp + 0x28], %i2
21         ldd    [%sp + 0x30], %i4
22         ldd    [%sp + 0x38], %i6
23
24         save
25         save
26
27         jmp    %l1
28          rett  %l2
29 FUNC_END __leon_trap_window_underflow
```

A.3 Context Switching

```
    .section    ".text"
    .global     __swap

/**
 * Interrupts are already locked with key as argument to __swap.
 * This is a leaf procedure, so only out registers
 * can be used without saving their context first.
 */

 /* unsigned int __swap(unsigned int key) */
FUNC_BEGIN __swap

    /**
     * Get a reference to currently running and next to run
     * thread context into %o1 and %o2.
     */
    sethi   %hi(_kernel), %o3
    or      %o3, %lo(_kernel), %o3

    ld     [%o3 + _kernel_offset_to_current], %o1
    ld     [%o3 + _kernel_offset_to_ready_q_cache], %o2

    /* put standard return value into arch structure */
    sethi  %hi(_k_neg_eagain), %o4
    or     %o4, %lo(_k_neg_eagain), %o4
    ld     [%o4], %o4
    st     %o4, [%o1 + _thread_offset_to_retval]

    /* save all local registers */
    std    %l0, [%o1 + _thread_offset_to_l0_and_l1]
    std    %l2, [%o1 + _thread_offset_to_l2]
    std    %l4, [%o1 + _thread_offset_to_l4]
    std    %l6, [%o1 + _thread_offset_to_l6]

    /* save all input registers */
    std    %i0, [%o1 + _thread_offset_to_i0]
    std    %i2, [%o1 + _thread_offset_to_i2]
    std    %i4, [%o1 + _thread_offset_to_i4]
    std    %i6, [%o1 + _thread_offset_to_i6]

    /* save output registers */
    std    %o6, [%o1 + _thread_offset_to_o6]

    rd     %psr, %o4
    st     %o4, [%o1 + _thread_offset_to_psr]

    st     %o0, [%o1 + _thread_offset_to_key]

    and     %o4, PSR_CWP, %g3       /* %g3 = CWP */
    andn    %o4, PSR_ET, %g1        /* %g1 = psr with traps disabled */
    wr      %g1, %psr               /* disable traps */
    rd      %wim, %g2               /* %g2 = wim */
```

```
53      mov     1, %g4
54      sll     %g4, %g3, %g4           /* %g4 = wim mask for CW invalid */
55
56
57      /* load number of windows -1 into g7 */
58      set     __leon_nwindows_min1, %g7
59      ld      [%g7], %g7
60
61  save_frame_loop:
62      sll     %g4, 1, %g5             /* rotate wim left by 1 */
63      srl     %g4, %g7, %g4           /* %g7 = NWINDOWS-1 */
64      or      %g4, %g5, %g4           /* %g4 = wim if we do one restore */
65
66      /* if restore would not underflow, continue */
67      andcc   %g4, %g2, %g0           /* window to flush? */
68      bnz     done_flushing           /* continue */
69       nop
70
71      restore                         /* go one window back */
72
73      /* essentially the same as window overflow */
74      std     %l0, [%sp + CPU_STACK_FRAME_L0_OFFSET]
75      std     %l2, [%sp + CPU_STACK_FRAME_L2_OFFSET]
76      std     %l4, [%sp + CPU_STACK_FRAME_L4_OFFSET]
77      std     %l6, [%sp + CPU_STACK_FRAME_L6_OFFSET]
78
79      std     %i0, [%sp + CPU_STACK_FRAME_I0_OFFSET]
80      std     %i2, [%sp + CPU_STACK_FRAME_I2_OFFSET]
81      std     %i4, [%sp + CPU_STACK_FRAME_I4_OFFSET]
82      std     %i6, [%sp + CPU_STACK_FRAME_I6_OFFSET]
83
84      ba      save_frame_loop
85       nop
86
87  done_flushing:
88      /**
89       * wait three instructions after the write to PSR before
90       * using non-global registers or instructions affecting the CWP
91       */
92      wr      %g1, %psr               /* restore cwp */
93      add     %g3, 1, %g2             /* calculate desired wim */
94      cmp     %g2, %g7                /* check if wim is in range */
95      bg,a    wim_overflow
96       mov    0, %g2
97
98  wim_overflow:
99
100     mov     1, %g4
101     sll     %g4, %g2, %g4           /* %g4 = new wim */
102     wr      %g4, %wim
103
104     /* put new thread context into current */
105     st      %o2, [%o3 + _kernel_offset_to_current]
106
107     /* restore local registers */
```

```
108     ldd     [%o2 + _thread_offset_to_l0_and_l1], %l0
109     ldd     [%o2 + _thread_offset_to_l2], %l2
110     ldd     [%o2 + _thread_offset_to_l4], %l4
111     ldd     [%o2 + _thread_offset_to_l6], %l6
112
113     /* restore input registers */
114     ldd     [%o2 + _thread_offset_to_i0], %i0
115     ldd     [%o2 + _thread_offset_to_i2], %i2
116     ldd     [%o2 + _thread_offset_to_i4], %i4
117     ldd     [%o2 + _thread_offset_to_i6], %i6
118
119     /* restore output registers */
120     ldd     [%o2 + _thread_offset_to_o6], %o6
121
122     /* get function return value */
123     ld      [%o2 + _thread_offset_to_retval], %o0
124
125     /* %g1 = new thread psr */
126     ld      [%o2 + _thread_offset_to_psr], %g1
127
128     /* reset PIL to key */
129     ld      [%o2 + _thread_offset_to_key], %o4
130
131     sll     %o4, PSR_PIL_BIT, %o4
132     andn    %g1, PSR_PIL, %o3
133     or      %o3, %o4, %g1
134
135     andn    %g1, PSR_CWP, %g1       /* psr without cwp */
136     or      %g1, %g3, %g1           /* psr with new cwp */
137     wr      %g1, %psr               /* restore status register and ET */
138     nop
139     nop
140     nop
141
142     /* jump into thread */
143     jmp     %o7 + 8
144      nop
145
146 FUNC_END __swap
```

A.4 Interrupt Trap

```
/*
 * Interrupt trap handler
 *
 * - IU state is saved and restored
 * - FPU state is not saved or touched
 * - Interrupt nesting is not supported. (Future Work)
 *
 * On entry:
 * %l0: psr
 * %l1: pc
 * %l2: npc
 * %l3: SPARC interrupt request level (bp_IRL)
 */
FUNC_BEGIN __leon_trap_interrupt_svt
        /* We came from an SVT trap dispatcher with trap type in %l6 */
        sub     %l6, 0x10, %l3

FUNC_BEGIN __leon_trap_interrupt
        /* %g2, %g3 used during manual window overflow */
        mov     %g2, %l4
        mov     %g3, %l5

        /* We are in our own register window, which could be the invalid one.
         * If so, save the next.
         */
        rd      %wim, %g2
        srl     %g2, %l0, %g3
        cmp     %g3, 1
        bne     .Lwodone
         nop

        /* Manual window overflow */
        sethi   %hi(__leon_nwindows_min1), %g3
        ld      [%g3 + %lo(__leon_nwindows_min1)], %g3
        sll     %g2, %g3, %g3
        srl     %g2, 1, %g2
        or      %g2, %g3, %g2

        /* Enter window to save */
        save

        /* set new wim */
        mov     %g2, %wim
        nop
        nop
        nop

        /* Put registers on stack */
        std     %l0, [%sp + CPU_STACK_FRAME_L0_OFFSET]
        std     %l2, [%sp + CPU_STACK_FRAME_L2_OFFSET]
        std     %l4, [%sp + CPU_STACK_FRAME_L4_OFFSET]
        std     %l6, [%sp + CPU_STACK_FRAME_L6_OFFSET]
```

```
53
54          std     %i0, [%sp + CPU_STACK_FRAME_I0_OFFSET]
55          std     %i2, [%sp + CPU_STACK_FRAME_I2_OFFSET]
56          std     %i4, [%sp + CPU_STACK_FRAME_I4_OFFSET]
57          std     %i6, [%sp + CPU_STACK_FRAME_I6_OFFSET]
58
59          /* exit window */
60          restore
61          nop
62
63  .Lwodone:
64          /* ISR context save */
65
66          /*
67           *  At this point:
68           *          %l4 = old %g2
69           *          %l5 = old %g3
70           */
71
72          /*
73           * Save the state of the interrupted task
74           * including global registers on the task stack
75           *
76           * Note: this could also be done in the callee_saved structure
77           */
78
79          /* get stack to safe isr context
80           * includes normal frame for window traps
81           */
82          sub     %fp, CPU_INTERRUPT_FRAME_SIZE, %sp
83
84          std     %l0, [%sp + ISF_PSR_OFFSET]     ! psr, PC
85          st      %l2, [%sp + ISF_NPC_OFFSET]     ! nPC
86          st      %g1, [%sp + ISF_G1_OFFSET]      ! g1
87          std     %l4, [%sp + ISF_G2_OFFSET]      ! g2, g3
88          std     %g4, [%sp + ISF_G4_OFFSET]      ! g4, g5
89
90          std     %i0, [%sp + ISF_I0_OFFSET]      ! i0, i1
91          std     %i2, [%sp + ISF_I2_OFFSET]      ! i2, i3
92          std     %i4, [%sp + ISF_I4_OFFSET]      ! i4, i5
93          std     %i6, [%sp + ISF_I6_OFFSET]      ! fp, i7
94
95          rd      %y, %g1
96          st      %g1, [%sp + ISF_Y_OFFSET]       ! y
97
98          /* Get a reference to _kernel */
99          sethi   %hi(_kernel), %g2
100         or      %g2, %lo(_kernel), %g2
101
102         /* switch to interrupt stack */
103         mov     %sp, %fp
104         ld      [%g2 + _kernel_offset_to_irq_stack], %sp
105
106         /* Allocate full C stack frame */
107         sub     %sp, SPARC_MINIMUM_STACK_FRAME_SIZE, %sp
```

```
108
109         /* TODO: Interrupt nesting, for now disable interrupts */
110         mov     %l5, %o2
111         sll     %o2, PSR_PIL_BIT, %o2
112         andn    %l0, PSR_PIL, %o3
113         or      %o3, %o2, %o0
114
115         /* Enable traps */
116         wr      %o0, PSR_ET, %psr
117         nop
118         nop
119         nop
120
121 /** ISR DISPACH BEGIN **/
122         /* %l3 holds interrupt request line */
123         mov     %l3, %o0
124
125         call    _enter_irq
126          nop
127 /** ISR DISPATCH END **/
128
129         /* Interrupts are still disabled */
130
131         /* TODO: Nesting */
132
133 /* switch back to interrupted task stack */
134         mov     %fp, %sp
135         add     %fp, CPU_INTERRUPT_FRAME_SIZE, %fp
136
137 #ifdef CONFIG_PREEMPT_ENABLED
138 /* Determine if context switch in nescessary */
139         sethi   %hi(_kernel), %l5
140         or      %l5, %lo(_kernel), %l5
141
142         ld      [%l5 + _kernel_offset_to_current], %l6
143         ld      [%l5 + _kernel_offset_to_ready_q_cache], %l7
144
145         /* See if current and ready context are the same */
146         cmp     %l6, %l7
147         beq     no_reschedule
148          nop
149
150         /* A context reschedule is required */
151         call    __swap
152          mov    0xf, %o0               /* PIL = 15 */
153
154 no_reschedule:
155 #endif /* CONFIG_PREEMPT_ENABLED */
156
157         /* Interrupts are disabled still for this thread */
158
159         /* Reverse context save */
160         ld      [%sp + ISF_Y_OFFSET], %g1
161         wr      %g1, 0, %y
162
```

```
163         ldd     [%sp + ISF_PSR_OFFSET], %l0     ! psr, PC
164         ld      [%sp + ISF_NPC_OFFSET], %l2     ! nPC
165         rd      %psr, %l3
166         and     %l3, PSR_CWP, %l3       ! current CWP
167         andn    %l0, PSR_CWP, %l0       ! rest of psr from task
168         or      %l3, %l0, %l0
169         andn    %l0, PSR_ET, %l0
170
171         mov     %sp, %g1
172
173         ldd     [%sp + ISF_G2_OFFSET], %g2
174         ldd     [%sp + ISF_G4_OFFSET], %g4
175
176         ldd     [%sp + ISF_I0_OFFSET], %i0
177         ldd     [%sp + ISF_I2_OFFSET], %i2
178         ldd     [%sp + ISF_I4_OFFSET], %i4
179         ldd     [%sp + ISF_I6_OFFSET], %i6
180
181
182         /*
183          * Register usage
184          *
185          *         Restored:
186          *          All global registers except g1
187          *                 All input registers
188          *
189          *         l0 = original psr
190          *  l1 = resturn address (i.e. PC)
191          *  l2 = nPC
192          *  l3 = CWP
193          */
194
195         /* Disable traps */
196         mov     %l0, %psr
197         nop
198         nop
199         nop
200
201         /* Determine if we must prepare the return window */
202         rd      %wim, %l4
203         /* l6 := cwp + 1 */
204         add     %l0, 1, %l6
205         and     %l6, PSR_CWP, %l6
206         /* Handle wrap-around */
207         sethi   %hi(__leon_nwindows), %l5
208         ld      [%l5 + %lo(__leon_nwindows)], %l7
209         cmp     %l6, %l7
210         bge,a   .Lwrapok
211          mov    0, %l6
212
213
214 .Lwrapok:
215         /* %l5 := %wim >> (cpw +1 ) */
216         srl     %l4, %l6, %l5
217         /* %l5 is 1 if (cwp+1) is an invalid window */
```

```
218         cmp     %l5, 1
219         bne     .Lwudone
220          nop
221
222         /* Manual window underflow */
223         /* %wim = rol(%wim) */
224         /* %l7 := __leon_nwindows - 1 */
225         sub     %l7, 1, %l7
226         srl     %l4, %l7, %l5
227         sll     %l4, 1, %l4
228         wr      %l4, %l5, %wim
229         nop
230         nop
231         nop
232
233         restore
234         ldd     [%sp + CPU_STACK_FRAME_L0_OFFSET], %l0
235         ldd     [%sp + CPU_STACK_FRAME_L2_OFFSET], %l2
236         ldd     [%sp + CPU_STACK_FRAME_L4_OFFSET], %l4
237         ldd     [%sp + CPU_STACK_FRAME_L6_OFFSET], %l6
238
239         ldd     [%sp + CPU_STACK_FRAME_I0_OFFSET], %i0
240         ldd     [%sp + CPU_STACK_FRAME_I2_OFFSET], %i2
241         ldd     [%sp + CPU_STACK_FRAME_I4_OFFSET], %i4
242         ldd     [%sp + CPU_STACK_FRAME_I6_OFFSET], %i6
243         save
244
245         /* Manual window underflow completed */
246
247 .Lwudone:
248         /* Restore %psr since we may have trashed condition codes */
249         /* also disables traps */
250         wr      %l0, %psr
251         nop
252         nop
253         nop
254
255         /* restore g1 */
256         ld      [%g1 + ISF_G1_OFFSET], %g1
257
258         jmp     %l1
259         rett    %l2
260
261 FUNC_END __leon_trap_interrupt
262 FUNC_END __leon_trap_interrupt_svt
```

A.5 Context Switching with Partitioning

```
        .section        ".text"
        .global         __swap
        .global         __setup_partition

/* unsigned int __swap(unsigned int key) */
FUNC_BEGIN __swap
        set     _kernel, %o3

        /* Get a reference to current and next to run thread */
        ld      [%o3 + _kernel_offset_to_current], %o1
        ld      [%o3 + _kernel_offset_to_ready_q_cache], %o2

        set     _k_neg_eagain, %o4
        ld      [%o4], %o4
        st      %o4, [%o1 + _thread_offset_to_retval]

        rd      %psr, %o4
        st      %o4, [%o1 + _thread_offset_to_psr]

        st      %o0, [%o1 + _thread_offset_to_key]

        and     %o4, PSR_CWP, %g3       /* CWP */
        andn    %o4, PSR_ET, %g1        /* psr with traps disabled */
        wr      %g1, %psr               /* DISABLE TRAPS */

        rd      %wim, %g2
        st      %g2, [%o1 + _thread_offset_to_wim]

        ld      [%o2 + _thread_offset_to_is_in_partition], %g1

        cmp     %g0, %g1
        bne     switch_partitions
         nop

        /* next to run thread is not in a partition right now */
        /* switch to shared partition */
        mov     %o3, %g3

        mov     CONFIG_LEON_SHARED_PART_STWIN, %g5
        sll     %g5, ASR20_STWIN_BIT, %g5

        mov     CONFIG_LEON_SHARED_PART_CWPMAX, %g4
        sll     %g4, ASR20_CWPMAX_BIT, %g4

        or      %g5, %g4, %g5
        mov     %g5, %asr20
        nop
        nop
        nop

/** Now in shared partition **/
        /* get reference to thread that is currently in shared partition */
```

```
        ld      [%g3 + _kernel_offset_to_current_in_shared], %g4

        cmp     %g4, %g0
        be      done_flushing
         nop

        ld       [%g4 + _thread_offset_to_psr], %g1
        ld       [%g4 + _thread_offset_to_wim], %g5

        andn     %g1, PSR_ET, %g1               /* psr with traps disabled */
        wr       %g1, %psr
        wr       %g5, %wim
        nop
        nop
        nop

        /* save all local registers */
        std      %l0, [%g4 + _thread_offset_to_l0_and_l1]
        std      %l2, [%g4 + _thread_offset_to_l2]
        std      %l4, [%g4 + _thread_offset_to_l4]
        std      %l6, [%g4 + _thread_offset_to_l6]

        /* save all input registers */
        std      %i0, [%g4 + _thread_offset_to_i0]
        std      %i2, [%g4 + _thread_offset_to_i2]
        std      %i4, [%g4 + _thread_offset_to_i4]
        std      %i6, [%g4 + _thread_offset_to_i6]

        /* save output registers */
        std      %o6, [%g4 + _thread_offset_to_o6]

        mov      %y, %o4
        st       %o4, [%g4 + _thread_offset_to_y]

        and      %g1, PSR_CWP, %g1
        mov      1, %g7
        sll      %g7, %g1, %g7             /* wim mask for CW invalid */

        set      CONFIG_LEON_SHARED_PART_CWPMAX, %g2

save_frame_loop:
        sll      %g7, 1, %g1
        srl      %g7, %g2, %g7
        or       %g7, %g1, %g7

        /* if restore would underflow, stop */
        andcc    %g7, %g5, %g0
        bnz      done_flushing
            nop

           restore

        /* essentially the same as window overflow */
        std      %l0, [%sp + CPU_STACK_FRAME_L0_OFFSET]
        std      %l2, [%sp + CPU_STACK_FRAME_L2_OFFSET]
```

```
        std     %l4, [%sp + CPU_STACK_FRAME_L4_OFFSET]
        std     %l6, [%sp + CPU_STACK_FRAME_L6_OFFSET]

        std     %i0, [%sp + CPU_STACK_FRAME_I0_OFFSET]
        std     %i2, [%sp + CPU_STACK_FRAME_I2_OFFSET]
        std     %i4, [%sp + CPU_STACK_FRAME_I4_OFFSET]
        std     %i6, [%sp + CPU_STACK_FRAME_I6_OFFSET]

        ba      save_frame_loop
         nop

done_flushing:

        /* unset is_in_partition for flushed out thread */
        st      %g0, [%g4 + _thread_offset_to_is_in_partition]

        /* set CWP to 0 and WIM to w1 */
        rd      %psr, %g1
        and     %g1, 0xfffffff0, %g1
        wr      %g1, %psr
        mov     0x2, %wim
        nop
        nop
        nop

        /* put new thread context into current */
        ld      [%g3 + _kernel_offset_to_current], %o1
        ld      [%g3 + _kernel_offset_to_ready_q_cache], %o2

        st      %o2, [%g3 + _kernel_offset_to_current]
        mov     1, %o1
        st      %o1, [%o2 + _thread_offset_to_is_in_partition]
        st      %o2, [%g3 + _kernel_offset_to_current_in_shared]

        ldd     [%o2 + _thread_offset_to_y], %o4
        mov     %o4, %y

        /* restore local registers */
        ldd     [%o2 + _thread_offset_to_l0_and_l1], %l0
        ldd     [%o2 + _thread_offset_to_l2], %l2
        ldd     [%o2 + _thread_offset_to_l4], %l4
        ldd     [%o2 + _thread_offset_to_l6], %l6

        /* restore input registers */
        ldd     [%o2 + _thread_offset_to_i0], %i0
        ldd     [%o2 + _thread_offset_to_i2], %i2
        ldd     [%o2 + _thread_offset_to_i4], %i4
        ldd     [%o2 + _thread_offset_to_i6], %i6

        /* restore output registers */
        ldd     [%o2 + _thread_offset_to_o6], %o6

        /* get function return value */
        ld      [%o2 + _thread_offset_to_retval], %o0
```

```
        ld      [%o2 + _thread_offset_to_psr], %g1

        /* reset PIL to key */
        ld      [%o2 + _thread_offset_to_key], %o4

        sll     %o4, PSR_PIL_BIT, %o4
        andn    %g1, PSR_PIL, %o3
        or      %o3, %o4, %g1

        andn    %g1, PSR_CWP, %g1       /* psr without cwp */
        wr      %g1, %psr               /* restore status register and ET */
        nop
        nop
        nop

        ba      swap_return
         nop

switch_partitions:
        /* next to run thread is in own partition, just switch to that one */
        ld      [%o2 + _thread_offset_to_stwin], %g1
        ld      [%o2 + _thread_offset_to_cwpmax], %g2
        ld      [%o2 + _thread_offset_to_psr], %g3
        ld      [%o2 + _thread_offset_to_wim], %g4
        ld      [%o2 + _thread_offset_to_key], %g5
        ld      [%o2 + _thread_offset_to_retval], %g7

        /* reset PIL to key */
        sll     %g5, PSR_PIL_BIT, %g5
        andn    %g3, PSR_PIL, %g3
        or      %g3, %g5, %g3

        /* put next thread context into current */
        st      %o2, [%o3 + _kernel_offset_to_current]

        /* switch partitions */
        sll     %g1, ASR20_STWIN_BIT, %g1
        sll     %g2, ASR20_CWPMAX_BIT, %g2
        or      %g1, %g2, %g1
        wr      %g1, %asr20

        /** Now in next thread partition **/
        wr      %g3, %psr           /* this also sets CWP */
        wr      %g4, %wim
        nop
        nop
        nop

        mov     %g7, %o0

swap_return:
        jmp     %o7 + 8
         nop

FUNC_END __swap
```

A.6 Interrupt Trap with Partitioning

```
FUNC_BEGIN __leon_trap_interrupt_svt
        sub     %l6, 0x10, %l3

FUNC_BEGIN __leon_trap_interrupt

        mov     %g2, %l4
        mov     %g3, %l5

        /* check if we are in invalid window */
        rd      %wim, %g2
        srl     %g2, %l0, %g3
        cmp     %g3, 1
        bne     .manual_win_ov_done
         nop

        /* Manual window overflow */

        rd      %asr20, %g7
        set     ASR20_CWPMAX, %g6
        and     %g7, %g6, %g7
        srl     %g7, ASR20_CWPMAX_BIT, %g3
        sll     %g2, %g3, %g3
        srl     %g2, 1, %g2
        or      %g2, %g3, %g2

        /* Enter window to save */
        save

        /* set new wim */
        mov     %g2, %wim
        nop
        nop
        nop

        /* Put registers on stack */
        std     %l0, [%sp + CPU_STACK_FRAME_L0_OFFSET]
        std     %l2, [%sp + CPU_STACK_FRAME_L2_OFFSET]
        std     %l4, [%sp + CPU_STACK_FRAME_L4_OFFSET]
        std     %l6, [%sp + CPU_STACK_FRAME_L6_OFFSET]

        std     %i0, [%sp + CPU_STACK_FRAME_I0_OFFSET]
        std     %i2, [%sp + CPU_STACK_FRAME_I2_OFFSET]
        std     %i4, [%sp + CPU_STACK_FRAME_I4_OFFSET]
        std     %i6, [%sp + CPU_STACK_FRAME_I6_OFFSET]

        /* exit window */
        restore
        nop

.manual_win_ov_done:
         /* We need to save a minimum context on the task stack */
```

```
 /* get stack to save isr context */
sub     %fp, CPU_INTERRUPT_FRAME_SIZE, %sp

st      %g1, [%sp + ISF_G1_OFFSET]      ! g1
std     %l4, [%sp + ISF_G2_OFFSET]      ! g2, g3
std     %g4, [%sp + ISF_G4_OFFSET]      ! g4, g5

rd      %y, %g1
st      %g1, [%sp + ISF_Y_OFFSET]       ! y

/* get a reference to kernel */
sethi   %hi(_kernel), %g2
or      %g2, %lo(_kernel), %g2

/* get a reference to current running thread */
ld      [%g2 + _kernel_offset_to_current], %g5

/* store CWP in TCB */
rd      %psr, %g1
and     %g1, PSR_CWP, %g3

st      %g3, [%g5 + _thread_offset_to_cwp]

/* store WIM in TCB */
rd      %wim, %g3
st      %g3, [%g5 + _thread_offset_to_wim]

/* move interrupt request line into global reg */
mov     %l3, %g7


/* switch to dedicated ISR partition */
mov     CONFIG_LEON_INT_PART_STWIN, %g3
sll     %g3, ASR20_STWIN_BIT, %g3

mov     CONFIG_LEON_INT_PART_CWPMAX, %g4
sll     %g4, ASR20_CWPMAX_BIT, %l4

or      %g3, %l4, %g3
mov     %g3, %asr20
nop
nop
nop

/** now in ISR partition **/

/* change CWP to 0 and disable interrupts */
and     %g1, 0xfffffe0, %g1
mov     15, %l5
sll     %l5, PSR_PIL_BIT, %l5
andn    %g1, PSR_PIL, %l6
or      %l6, %l5, %l6
mov     %l6, %psr

mov     1, %g3
```

```
        sll     %g3, 1, %g3
         mov     %g3, %wim            ! wim = w1
         nop
         nop
         nop

         /* setup fp and sp */
         mov     %g0, %fp
         ld      [%g2 + _kernel_offset_to_irq_stack], %sp
         sub     %sp, SPARC_MINIMUM_STACK_FRAME_SIZE, %sp

         /* enable traps */
         mov     %psr, %l4
         wr      %l4, PSR_ET, %psr
         nop
         nop
         nop

/** ISR DISPACH BEGIN **/
        /* %l3 holds interrupt request line */
        mov     %g7, %o0

        call    _enter_irq
         nop
/** ISR DISPATCH END **/
        /* switch back to task partition */

        /* get a reference to kernel */
        sethi   %hi(_kernel), %g2
        or      %g2, %lo(_kernel), %g2

        /* get a reference to current running thread */
        ld      [%g2 + _kernel_offset_to_current], %g4

        ld      [%g4 + _thread_offset_to_stwin], %g3
        sll     %g3, ASR20_STWIN_BIT, %g3

        ld      [%g4 + _thread_offset_to_cwpmax], %g1
        sll     %g1, ASR20_CWPMAX_BIT, %g1

        or      %g3, %g1, %g3
        mov     %g3, %asr20
        nop
        nop
        nop

        /** now in Task partition **/

        /* restore CWP from TCB */
        rd      %psr, %g1
        ld      [%g4 + _thread_offset_to_cwp], %g3
        andn    %g1, PSR_CWP, %g1
        or      %g1, %g3, %g3
        mov     %g3, %psr
```

```
        /* restore WIM from TCB */
        ld      [%g4 + _thread_offset_to_wim], %g3
        mov     %g3, %wim
        nop
        nop
        nop

#ifdef CONFIG_PREEMPT_ENABLED
/* Determine if context switch in necessary */
        ld      [%g2 + _kernel_offset_to_current], %g3
        ld      [%g2 + _kernel_offset_to_ready_q_cache], %g4

        /* See if current and ready context are the same */
        cmp     %g3, %g4
        beq     no_reschedule
         nop

        /* A context reschedule is required */

        call    __swap
         mov    0xf, %o0             /* PIL = 15 */

no_reschedule:
#endif /* CONFIG_PREEMPT_ENABLED */

        rd      %psr, %l3
        and     %l3, PSR_CWP, %l3       ! current CWP
        andn    %l0, PSR_CWP, %l0       ! rest of psr from task
        or      %l3, %l0, %l0
        andn    %l0, PSR_ET, %l0

        /* Disable traps */
        mov     %l0, %psr
        nop
        nop
        nop

        /* Reverse context save */
        ld      [%sp + ISF_Y_OFFSET], %g1
        wr      %g1, 0, %y

        ld      [%sp + ISF_G1_OFFSET], %g1
        ldd     [%sp + ISF_G2_OFFSET], %g2
        ldd     [%sp + ISF_G4_OFFSET], %g4

        /* Determine if we must prepare the return window */
        rd      %wim, %l4
        add     %l0, 1, %l6
        and     %l6, PSR_CWP, %l6
        rd      %asr20, %g7
        set     ASR20_CWPMAX, %g6
        and     %g7, %g6, %g7
        srl     %g7, ASR20_CWPMAX_BIT, %g7
        add     %g7, 1, %g7
        cmp     %l6, %g7
```

```
        bge,a   .wim_wrap_around
         mov    0, %l6

.wim_wrap_around:
        srl     %l4, %l6, %l5
        cmp     %l5, 1
        bne     .trap_wu_done
         nop

        /* manual window underflow */
        sub     %g7, 1, %g7
        srl             %l4, %g7, %l5
        sll     %l4, 1, %l4
        wr      %l4, %l5, %wim
        nop
        nop
        nop

        restore
        ldd     [%sp + CPU_STACK_FRAME_L0_OFFSET], %l0
        ldd     [%sp + CPU_STACK_FRAME_L2_OFFSET], %l2
        ldd     [%sp + CPU_STACK_FRAME_L4_OFFSET], %l4
        ldd     [%sp + CPU_STACK_FRAME_L6_OFFSET], %l6

        ldd     [%sp + CPU_STACK_FRAME_I0_OFFSET], %i0
        ldd     [%sp + CPU_STACK_FRAME_I2_OFFSET], %i2
        ldd     [%sp + CPU_STACK_FRAME_I4_OFFSET], %i4
        ldd     [%sp + CPU_STACK_FRAME_I6_OFFSET], %i6
        save

.trap_wu_done:

        /* may have trashed cc */
        wr      %l0, %psr
        nop
        nop
        nop

        jmp     %l1
         rett    %l2

FUNC_END __leon_trap_interrupt
FUNC_END __leon_trap_interrupt_svt
```

APPENDIX B

How to create an application with window partitioning

In this section, a short overview of the necessary steps for creating a simple Zephyr application with window partitioning for the GR716 is presented.

First, the folder structure is created for the application. In the Zephyr repository, there is a folder called *samples*, where the subdirectory for this example will go:

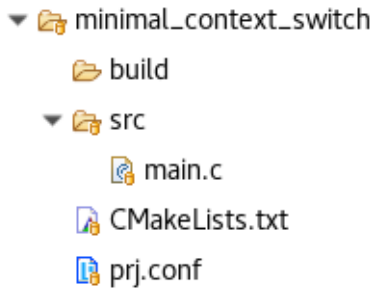


Figure B.1: Folder structure for example application.

Figure B.1 shows the minimal structure for a working Zephyr application. The different files are explained below.

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.13.1)
2
3 include($ENV{ZEPHYR_BASE}/cmake/app/boilerplate.cmake NO_POLICY_SCOPE)
4 project(minimal_context_switch)
5
```

```
6 target_sources(app PRIVATE src/main.c)
```

The *CMakeLists.txt* file configures the cmake tool to produce the necessary files for the build system. Through the **include** function the Zephyr standard application cmake template is loaded. The **project** directive sets the name of the project, and **target_sources** includes the source code of the application.

prj.conf

```
1  CONFIG_STDOUT_CONSOLE=y
2  CONFIG_PRINTK=y
3  CONFIG_MAIN_STACK_SIZE=4096
4  CONFIG_CONSOLE=y
5  CONFIG_CONSOLE_HAS_DRIVER=y
6  CONFIG_UART_CONSOLE_ON_DEV_NAME="APBUART0"
7  CONFIG_SERIAL=y
8  CONFIG_UART_APBUART=y
9  CONFIG_APBUART0=y
10 CONFIG_LEON_TIMER=y
```

The *prj.conf* file sets the standard configuration values used by kconfig. Here, basically only the UART driver and the LEON timer driver are selected. The stack memory size for the main task is also configured.

The application code itself goes into the *main.c* file:

main.c

```
1  #include <zephyr.h>
2  #include <string.h>
3  #include <stdio.h>
4
5
6  #define STACKSIZE 4096
7  #define PRIORITY 2                        /* coop threads */
8
9  void out_task(const char* output, void* param2, void* param3)
10 {
11         for(int i = 0; i < 3; i++)
12         {
13                 printf(output);
14                 k_yield();
15         }
16 }
17
18 K_THREAD_DEFINE(
19         task1_id,
20         STACKSIZE,
21         out_task,
22         "Hello from Task 1!\n",
23         NULL,
24         NULL,
25         PRIORITY,
26         K_PART_1,
```

```
27          K_NO_WAIT
28  );
29
30  K_THREAD_DEFINE(
31          task2_id,
32          STACKSIZE,
33          out_task,
34          "Hello from Task 2!\n",
35          NULL,
36          NULL,
37          PRIORITY,
38          K_PART_2,
39          K_NO_WAIT
40  );
```

Some comments to the code above:

Line 7 Using a positive integer, the threads are cooperative (preemptive threads have negative priority in Zephyr)

Line 9 Both tasks are using the **out_task** function as their workload job. In Zephyr, a task function must except up to 3 arguments. The first argument here is used to point to the output string

Line 14 A call to **k_yield** will cause the scheduler to remove the current thread and put in the next to run thread instead

Line 18 **K_THREAD_DEFINE** is used to define threads at compile time. It is also possible to create threads dynamically at runtime in the main function

Line 19 The task ID. This is used by the compiler and linker

Line 20 The stack size required by the task

Line 21 The task function

Line 22-24 The three optional task parameters. The first one is used to define the output string for the different tasks

Line 25 The priority of the task. Higher (absolute) numbers correspond to lower priorities

Line 26 The task options. Here, **K_PART_1** is used to dedicate the first partition to this task

Line 27 An optional task delay before the first time it is scheduled. Here, no delay is configured

The second thread is set up exactly the same way as the first one, but will be put in the second dedicated partition instead.

Within the Zephyr repository base directory issue the following bash commands to set up the build system:

```
source zephyr-env.sh
export ZEPHYR_TOOLCHAIN_VARIANT=cross-compile
export CROSS_COMPILE=/opt/bcc-2.0.5-gcc/bin/sparc-gaisler-elf-
```

The last line of the above code snippet must be adapted, if a different version of BCC is used, or if it is in a different directory. Now, within the *build* directory of

the *minimal_context_switch* example folder, issue the following commands:

```
cmake -DBOARD=tsim_leon3 ..
```

This will invoke cmake to create the build system. Now the graphical kconfig tool can be used:

```
make menuconfig
```

This will load the the interface shown in B.2. From here, every configuration value that can be changed for the current architecture setting can be adapted. In order to use the window partitioning feature, the values under *LEON Partitioning Options* must reflect the state shown in Figure B.3.

As can be seen in Figure B.3, partitioning is enabled, and in total four partitions are created: one *interrupt partition* with 7 windows (**w0 - w6**), one *shared partition* with 8 windows, (**w7 - w14**), *dedicated partition 1* with 8 windows (**w15 - w22**), and *dedicated partition 2* with 8 windows (**w23 - w30**).

Different values for the different partitions are also possible, as long as there is no overlap. Furthermore, no partition can overflow the register window circle, i.e. the first window (**w0**) must be the start, and the last window (**w30**) must be the tail of a partition.

Once the configuration is saved, the application ca be compiled and linked:

```
make
```

If everything was set up correctly, the make tool will compile and link the application. Towards the end of the output, an overview of the static memory footprint is given (see Figure B.4).

If there is a version of the LEON simulator TSIM installed, the following command can be used to run the application:

```
make run
```

This will lead to the output shown in Figure B.5.

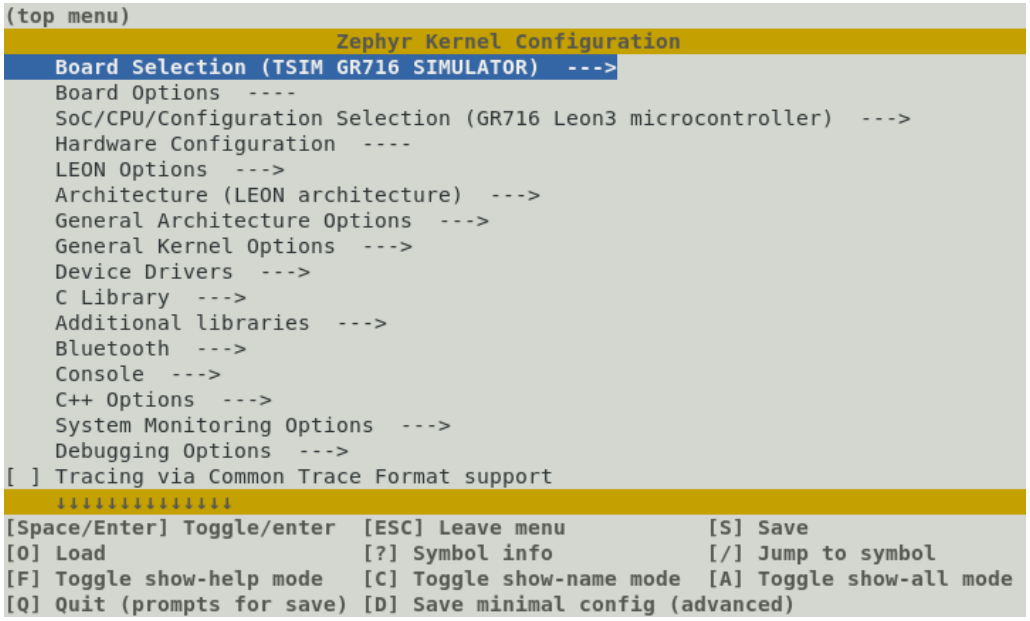


Figure B.2: Kconfig top menu window.

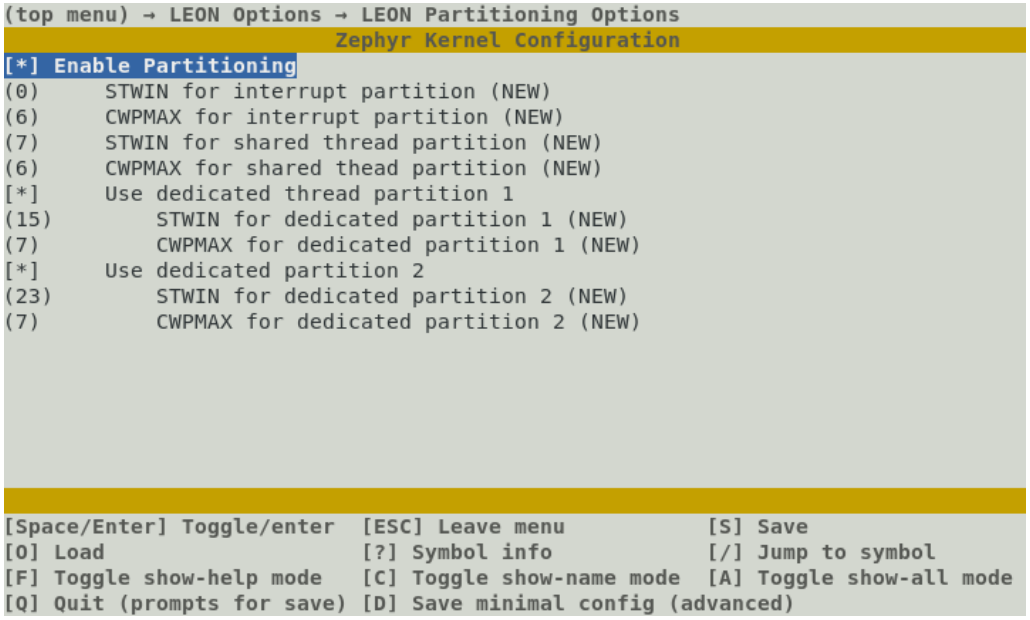


Figure B.3: Kconfig LEON partitioning options.

```
[ 94%] Linking C executable zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
        bootprom:          0 GB         4 KB      0.00%
         extprom:          0 GB        16 MB      0.00%
            spi0:          0 GB        32 MB      0.00%
            spi1:          0 GB        32 MB      0.00%
            dram:       17992 B        64 KB     27.45%
            iram:       18096 B       128 KB     13.81%
          extram:          0 GB       256 MB      0.00%
        IDT_LIST:           9 B         2 KB      0.44%
[ 94%] Built target zephyr_prebuilt
Scanning dependencies of target linker_pass_final_script_target
[ 95%] Generating linker_pass_final.cmd
[ 95%] Built target linker_pass_final_script_target
[ 96%] Generating isr_tables.c
Scanning dependencies of target kernel_elf
[ 97%] Building C object zephyr/CMakeFiles/kernel_elf.dir/misc/empty_file.c.obj
[ 98%] Building C object zephyr/CMakeFiles/kernel_elf.dir/isr_tables.c.obj
[100%] Linking C executable zephyr.elf
Generating files from zephyr.elf for board: tsim_leon3
[100%] Built target kernel_elf
```

Figure B.4: Output of make.

```
***** Booting Zephyr OS zephyr-v1.13.0-4181-g79dec29 *****
Hello from Task 1!
Hello from Task 2!
Hello from Task 1!
Hello from Task 2!
Hello from Task 1!
Hello from Task 2!
```

Figure B.5: TSIM output.