# FreeRTOS and Multicore

James Mistry

Department of Computer Science

University of York

A thesis submitted for the degree of

*Master of Science*

2nd September 2011

Supervised by Prof. Jim Woodcock and Dr. Matthew Naylor

*The body of this thesis is 28,961 words in length, as counted by wc -w.*

This thesis is dedicated to my parents, whose help and support over the years has been invaluable in allowing me to pursue my passion, and to Robert and Fran, whose belief in me has been an unquantifiable gift.

# Acknowledgements

I am supremely grateful to my supervisors, Matt and Jim, for their enthusiastic support and invaluable advice throughout; I am also indebted to the researchers in the Real–Time Systems Group at York who responded with such patience to my questions and provided domain expertise that saved me days of frustration.

# Abstract

Multicore processors are ubiquitous. Their use in embedded systems is growing rapidly and, given the constraints on uniprocessor clock speeds, their importance in meeting the demands of increasingly processor–intensive embedded applications cannot be understated. In order to harness this potential, system designers need to have available to them embedded operating systems with built–in multicore support for widely available embedded hardware.

This report documents the process of developing a multicore version of FreeRTOS, a popular embedded real–time operating system, for use with programmable circuits configured with multicore processors. A working multicore version of FreeRTOS is presented that is able to schedule tasks on multiple processors as well as provide full mutual exclusion support for use in concurrent applications.

# Statement of Ethics

No human or animal subjects were involved in experiments, no property was harmed or put at risk and no personal data was collected as part of this project.

This project has involved the modification of a real-time operating system. The modifications have not been subjected to independent review and testing to validate the accuracy of the results presented. As such, the software presented must not, without further rigorous assessment as to its safety and suitability for use in a critical context, be used in situations in which life or property may depend on its correct operation.

The software produced as part of this project is based on open–source software released under the GNU General Public License by multiple authors, including but not limited to Real Time Engineers Ltd and Tyrel Newton. In accordance with the terms of this licence, the modifications in source code form are made freely available at the following address: `https://sourceforge.net/projects/freertosxcore/`

As a scientific venture, great attention has been given to the importance of allowing others to reproduce the work presented. In addition to making the source code of the software modifications publicly available, section 5.3 on page 66 contains detailed instructions explaining how to reproduce the hardware, configure the necessary tools and run the software.

“On two occasions I have been asked, ‘Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?’ I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.”

– *Charles Babbage*

# Contents

# List of Figures

# Nomenclature

| | |
|---|---|
| *AMP* | Asymmetric Multiprocessing |
| *BRAM* | Block Random Access Memory |
| *FPGA* | Field-Programmable Gate Array |
| *HDL* | Hardware Description Language |
| *PVR* | Processor Version Register |
| *ROM* | Read-Only Memory |
| *RTOS* | Real-Time Operating System |
| *SMP* | Symmetric Multiprocessing |
| *TCB* | Task Control Block |
| *XPS* | Xilinx Platform Studio |

# Chapter 1

# Introduction

From an idea first proposed by Sir Tony Hoare, this project aims to produce a version of FreeRTOS, an open–source real–time operating system, that supports multicore processors. Using a relatively basic multicore hardware configuration customised using a programmable circuit, the software developed during the course of the project has been created to serve as a starting point from which a full–featured version of FreeRTOS can be developed to provide a comprehensive operating system solution for embedded devices with multiple processors.

## 1.1 Motivation

Real–time software is fundamental to the operation of systems in which there exist requirements to impose temporal deadlines and behave in a deterministic manner.[1, p. 4] Examples of such systems include those responsible for controlling airbags in cars, control surfaces in aircraft and missile early warning alerts. In addition to such "hard" real–time systems, for which there must be an absolute guarantee that deadlines are met,[1, p. 5] there exist "firm" and "soft" real–time applications. Software in these categories provide weaker guarantees about their meeting of deadlines,[1, p. 6] usually because the effect of a deadline miss is deemed to be less harmful: unusable results (in the case of the former) and degraded performance (in the case of the latter).[1, p. 6]

The application of such definitions, while well–established, can nevertheless appear rather odd when contextualised. Hard real–time software is often framed as the tool to which engineers resort when human life is the cost of failure. By contrast, soft real–time systems become the reserve of mundane applications for which deadlines only serve as a yard stick for performance. Laplante, for example, suggests that cash machines fall into this category. He explains:

> Missing even many deadlines will not lead to catastrophic failure, only degraded performance.[1, p. 6]

While he is admittedly providing a simplified example, surely such a classification borders on the meaningless. What operations do the deadlines he mentions govern? How regularly might they be missed? By how much might they be missed (and therefore by how much will performance degrade)? Similarly, the notion of "catastrophic" failure is very much open to interpretation. Recall that failure in this sense refers not to the degree to which the system itself fails internally, but rather to the effects of missed deadlines on the external world. This raises important questions. What is the purpose of a deadline if it may be regularly missed by an infinite quantity of time? And if there is an upper–bound to the quantity of time for which a deadline may be missed, is this not the *actual* (hard) deadline?

The apparent vagueness of these classifications illustrates an important point: every piece of software, whether explicitly or not, is supposed to adhere to deadlines of some kind. The property of reliably meeting these deadlines follows from the determinism of the system, and this is of far greater significance than the existence of the deadlines themselves. While a real–time operating system may schedule tasks in such a way that they have an optimal chance of meeting their deadlines, if the underlying hardware behaves in a non–deterministic way then there is nothing the software can do to guarantee that deadlines will be met.

A good example of this is presented by Zlokolica *et al.* in their evaluation of real–time applications sharing an embedded environment with other (non–real–time) CPU–intensive tasks.[2] The software they evaluate is responsible for processing high definition video. Deadline misses can result in the processing of individual frames being skipped and screen artefacts[1] developing as a result:[2]

> Besides handling interrupts for scheduling tasks for the RTAs[2], [the kernel] is also responsible for handling a great number of other tasks related to video applications [...] which can cause a processing delay of the RTA module execution[2]

[1]Garbled components of an on–screen image
[2]Real–time algorithms

After analysing the source and occurrence of these delays and essentially building a model of the system's deterministic properties, they ultimately concluded that the incidence of missed deadlines was acceptable within the parameters of the application.[2] However, had they not been able to build such a model, they could not have identified the worst–case execution scenario and would have been unable to draw conclusions about the reliability of the system's performance. Similarly, had the system been more complex, the costs of performing such an analysis may have been prohibitively high.

Thus, simplicity is a core principle of the open–source embedded real–time operating system, FreeRTOS.[3] Developing applications based on FreeRTOS involves leveraging an easy–to–use API and simple, low–footprint real–time kernel. FreeRTOS allows the development of hard or soft real–time software, the exact classification depending on the application–specific implementation details, including the other components of the system (such as the hardware). As with any RTOS, it must be understood that it serves as a *tool* to be used correctly or incorrectly by system developers. An RTOS is not a magic elixir from which normal software automatically derives real–time properties. While the FreeRTOS task scheduler uses task priorities to run important tasks with greater preference to unimportant ones, it cannot ensure that their deadlines are met if doing so is infeasible given inherent restrictions on the system (such as the speed of the processor).

Indeed, it is important to make the point that processing throughput (the amount of processing a system can do per unit of time) does not define a real–time system.[4] A hard real–time system that can guarantee its deadlines will be met becomes no "more" real–time with additional processing throughput, although it may well become faster. The motivation for implementing FreeRTOS on a multicore architecture is thus not to change the way in which it handles task deadlines or exhibits predictable properties, but rather to meet the realities of the ever–increasing demands being made of embedded systems. For real–time and non–real–time applications alike, being able to run tasks concurrently promises performance advantages not possible on single–core architectures due to the limit processor manufacturers have encountered in maintaining increases in clock speeds on individual chips.[5] Moore's Law does continue, but in a "fundamentally different" way.[5][6] Multicore chips have allowed transistor numbers to continue to increase exponentially,[5] but not in a way that offers transparent performance gains for software. Indeed, the performance improvements promised by multicore architectures can only be realised by software specifically designed for execution in a concurrent environment.[1]

[1] A software application designed for execution in a single task cannot be automatically split into multiple

Perhaps the most striking examples of this in embedded systems can be found in the incredible explosion of processing power, and specifically the recent mainstream adoption of multicore architectures, on smartphones.[7] It is clear that all embedded real–time systems must adapt in the same way as those in the consumer market. Andrews *et al.* contend that RTOS designers have been "fighting"[8] Moore's Law and must now instead look to make use of processor advancements to help avoid resorting to ever more complex software solutions designed to squeeze the required real–time performance from slower hardware, as well as the associated maintenance problems that arise from this.[8] More fundamentally, a lack of action has the potential to cause the usefulness of embedded real–time systems to hit a wall. Consider the high–definition video processing applications mentioned above: the sheer size of the system's workload had brought it close to failure.[2] What will the system designers do when future iterations of the application must perform more tasks, process higher–resolution video and provide faster response times without having access to matching improvements in single–core hardware? Migration to multicore is inevitable.

## 1.2 Method

This thesis documents the process of modifying the MicroBlaze FreeRTOS port to run tasks concurrently on multiple processors. MicroBlaze is a soft processor architecture implemented through the use of Field Programmable Gate Arrays (FPGAs), special integrated circuits whose configuration is changeable "in the field". This technology allows for the rapid design and implementation of customised hardware environments, allowing complete control over the processing logic as well as system peripherals such as I/O devices and memory. Because of how configurable and relatively inexpensive they are, FPGAs provide an attractive way to develop highly parallel hardware solutions, with which a multicore embedded RTOS would be incredibly useful in providing a platform to abstract both the real–time scheduling behaviour as well as the underlying hardware away from application code.

As well as enabling discussions with researchers at York who are very familiar with the MicroBlaze platform, this processor architecture provides another important benefit: simplicity. MicroBlaze FPGA designs are readily available and can be implemented with relatively little effort, particularly important given the author's lack of experience with this technology. Also, by being able to strip down the hardware configuration (for example by disabling processor

tasks for simultaneous execution on a multicore processor.

caching and limiting the number of hardware peripherals), it is possible to greatly reduce the complexity of the software components that interact with the hardware at a low level, thus reducing implementation time and limiting the risks of non–completion in the relatively little time available.

Inevitably, the identification of design and implementation issues of one component during the design or implementation of another would sometimes need to retroactively be used to inform previous design, requirements or implementation details. It was thus deemed important that allowances be made in the project for experimental cycles of design, development and requirements refinement. The overarching methodology and how each iteration step fits into it is explained below.

**Planning** An exploration of the technical feasibility of the project, as well as an assessment of the technology required for the following stages.

**Requirements** An analysis of the current iteration's requirements as a decomposition of the broader project aims, combined with experience from previous iterations. Before the next stage, this is incorporated as a tentative agreement of what should be accomplished within the current iteration.

**Design** Applies the agreed requirements for the current iteration as a technical design appropriate to the subject matter. This could be in the form of code sketches, an initial hardware design or actions planned to resolve tool configuration issues on the basis of consultation with a domain expert.

**Implementation** The production of a new version of a prototype deliverable, either in the form of a hardware design or code. In addition, the incorporation of this deliverable with the other project deliverables to create a complete test environment.

**Testing and Analysis** The running of purpose–built test tasks to test the modifications made in this iteration. This includes standard test tasks which test the core context switching features and task management API in both pre–emptive and co–operative scheduling modes. Note that due to time limitations, FreeRTOS co–routines and the queue API are *not* included within the scope of this project. Also included is an analysis of the testing results which informs an assessment of the work required to be carried forward to the next iteration.

## 1.3 Outline

Chapter 2 discusses the background issues, technologies and research surrounding this topic. Chapter 3 details and analyses the project requirements. Chapter 4 explains the system design, detailing significant algorithms and other key design decisions. Chapter 5 discusses the components of the system, how they work and the justifications for the exact method of their implementation. Chapter 6 evaluates the adequacy of the implementation with regard to the defined requirements. Chapter 7 evaluates the implementation in a wider context, suggesting future improvements and how the work done might be continued. Chapter 8 summarises the project, taking a holistic view of the work done.

# Chapter 2

# Literature Review: Real–Time and the Multicore Environment

## 2.1 Multicore Processors

A processor "core" is the unit within a CPU that executes instructions, collections of which constitute software. By reading specially encoded values from memory that map to instructions in the processor's instruction set[1], the processor performs primitive actions that can be combined to create more advanced behaviour. To put the speed at which these primitive actions are executed into perspective, the MicroBlaze processor used in this project can execute up to 125,000,000 instructions per second (not taking into account time delays as a result of the processor waiting on another component, such as memory).

Traditionally, most processors have contained only one core. With some exceptions beyond the scope of this discussion, single–core processors can only execute instructions *synchronously*, one after the other. In order for a single–core processor to run multiple tasks, it has to rely on software, usually an operating system, to switch between tasks either at a regular interval (known as "pre–emptive scheduling") or when a currently executing task is ready to release control of the processor (known as "co–operative scheduling"). This discussion will focus on pre–emptive scheduling.

Because of the speed at which processors execute instructions, pre–emptive scheduling gives the impression that multiple tasks are executing at the same time. However, the more tasks

[1]An instruction set is the set of available instructions for a specific type of processor.

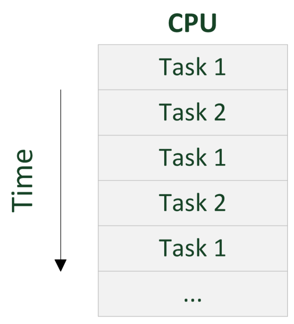

Figure 2.1: Executing two tasks on a single–core processor

that share the processor, the fewer of each task's instructions the processor can execute in a given amount of time. In Figure 2.1, for example, the processor only needs to divide its time in two because there are only two tasks that need to be executed. Thus, each one gets a 50% share of the processor's time. If three tasks were sharing the processor, this would be reduced to a third and thus it would take more time for each of the tasks to complete. Now consider a single–core processor with 10,000 long–running tasks to complete. In this situation it could well be that their sheer quantity would destroy the illusion of simultaneous execution, because each task would only benefit from one–ten thousandth of the processor's time, and take very long to finish.

Until the early 2000s,[5] this problem was often lessened in each new generation of hardware by reducing the time it would take for processors to execute their instructions. However, due to various manufacturing factors beyond the scope of this thesis, there came a point when an upper limit on the speed at which instructions could be executed was reached. It was then, for example, that mainstream processor manufacturers such as AMD and Intel began developing multicore processors to continue to offer improved performance and faster execution time.

Processors with multiple cores can execute instructions *asynchronously*, more than one at a time. Because each core executes instructions independently, the same number of tasks as cores can be executed simultaneously in a multicore system. Note that, unlike a single–core system, this is *true* simultaneity: multiple instructions are actually executed at the same time.

In Figure 2.2, both tasks have a 100% share of their respective core's time. Assuming both tasks require an equal quantity of time to complete, this means that the processor is able to execute them both in the same time as it would take to execute just one of them, and in half the time it would take a single–core processor of the same speed to complete them both. Note

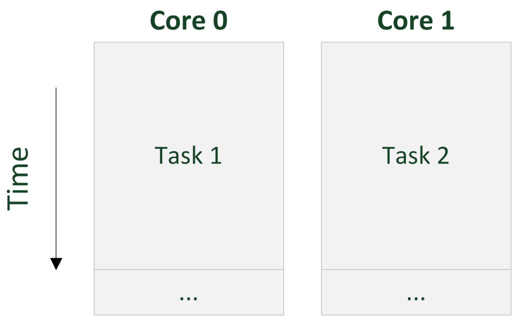

Figure 2.2: Executing two tasks on a dual–core processor

that in practice, multicore systems also make use of task swapping (switching between tasks) in order to be able to run many more tasks than there are available cores.

Unfortunately, the advantages of multicore processors cannot be used transparently by software not specifically designed to take advantage of simultaneous execution. Unlike with single–core processors, in which a new processor that executes instructions more quickly would automatically make compatible software run faster, there are several issues that make using multicore more complicated. Apart from having to carefully divide (or sub–divide) tasks in an appropriate way, the execution of multiple instructions at once presents challenges when accessing shared resources.

Consider the following code which consists of two tasks, both of which were initially intended for execution on a single–core processor.

Listing 2.1: Two tasks intended for use with one core

```

int sharedResults = 0; // 0 indicates no calculation result

/* First Task (for execution on Core 0) */
int main() {

  // Used to store the results of the calculation
  int results = 0;

  for (;;) {

    // Perform a calculation. Returns > 0 if successful,
    // < 0 if an error occurred
    results = performCalculation();

    disableInterrupts();
    // Communicate the results to the second task
    sharedResults = results;
    enableInterrupts();

  }

}

/* Second Task (for execution on Core 1) */
int main() {

  for (;;) {

    while(sharedResults != 0) { // Busy wait }

    disableInterrupts();
    if (sharedResults < 0) {

      writeToLog("Calculation error.");

    } else {

      writeToLog(sprintf("Calculation result: %i", sharedResults));

    }

    sharedResults = 0; // Reset the shared variable
    enableInterrupts();

  }
```

```
47
48 }
```

The job of the first task is to perform a calculation, the details of which are not important, and communicate the results to the second task using shared memory (the `sharedResults` variable). The second task polls this shared variable for a non–zero value, after which it determines whether or not the calculation was successful and writes this information to a log. It then resets the shared variable and waits for the next results.

Note that on lines 15 and 31 the `disableInterrupts` function is called. This effectively creates a "critical section", a region of code in which a task has exclusive access to some shared resource (in this case, the processor). This prevents the processor from interrupting the task and switching to another one until a corresponding call to `enableInterrupts` is made (lines 18 and 43). This is necessary to ensure that while the `sharedResults` variable is in use by one task, it is not changed by another. Because the only way another task may run and therefore gain access to a shared resource on a single–core processor is if the current task is interrupted and another resumed, disabling interrupts (provided at intervals by the hardware) is all that is necessary to ensure shared resources are used safely.

However, on a multicore processor this is not the case. Because all cores execute instructions at the same time as each other, the potential for the simultaneous use of shared resources is not connected to the swapping of tasks — a shared variable in a multicore system could, in fact, be modified by multiple tasks at any time. In order to protect against this, an alternative mechanism for creating a critical section that works across all cores in the system is required. This is discussed further in section 2.6 on page 19.

## 2.2 SMP and AMP Architectures

There are two principal types of multicore architecture: SMP (Symmetric Multiprocessing) and AMP (Asymmetric Multiprocessing). The difference is significant, but relatively straightforward: in SMP systems, all processors are connected to the same shared memory and execute the same instance of the operating system;[1] in AMP systems, each processor executes its own instance of the operating system using its own dedicated memory area, with special allowances made for communication between tasks running on separate processors.

[1]The same OS code in memory is executed by all processors.

Although AMP systems do not require the same quantity of synchronisation by virtue of the fact that memory and code is inherently private to each processor, the process of implementing separate OS instances and segregating memory adds significant complexity to system design, implementation and maintenance. For example, the same mechanisms that prevent memory from being accessed by multiple processors must also be circumvented or mitigated in order to allow cross–processor communication. Also, changes to the operating system code must be implemented for each processor, taking into account any differences in the separate code bases.

## 2.3 FPGAs

A Field–Programmable Gate Array (FPGA) is an integrated circuit whose behaviour can be reprogrammed *ad infinitum* after manufacture. Internally, an FPGA consists of large quantities of logic blocks that can be configured *en masse* to work together and behave in a useful way.[10] An FPGA itself is usually mounted on a board, upon which other hardware components such as memory and I/O ports are also located for use by the chip. Two types of FPGA boards have been used in this project: the Xilinx Spartan–3e Starter Board and the Xilinx Virtex–5 XUPV5–LX110T Development Board. The differences between them and the reasons for their use are discussed in section 4.1.1 on page 41. For now, it is sufficient to regard them as equivalent for the purposes of this discussion.

At this point it is helpful to consider more carefully the difference between hardware and software. When software is compiled it is converted to "machine" code, a series of bits that represent instructions and operands that a processor can interpret and execute. At the most fundamental level, software uses arithmetic (such as adding and subtracting), logic (such as or–ing and xor–ing) and memory manipulation (such as loading data from memory) operations provided by the processor to manipulate numbers. By combining together many of these operations, more high–level behaviour (such as drawing on a screen) can be produced.

The behaviour of hardware can be defined in much the same way. Consider the following piece of code:[11]

Listing 2.2: Describing hardware using Handel–C

```
void main(void) {
  uart_t uart;
  unsigned 4 i;
  unsigned 8 x;
  unsigned 8 xs[16];

  par {

    uart_driver(uart);

    seq {

      for (i = 0; i < 8; i++) {
        uart.read ? x;
        xs[i] = x;
      }

      par (k = 0; k < 8; k++) {
        xs[k] = xs[k] + 1;
      }

      for (i = 0; i < 8; i++) {
        uart.write ! xs[i];
      }

    }

  }

}
```

While it may seem at first that this code defines the operation of a piece of software, it is actually written in Handel–C, a hardware description language (HDL) that specifies the behaviour of an integrated circuit. The first loop reads 8 bytes from the UART (serial port). The second loop (using the `par` statement) adds one to each received byte. Note that this is a compile–time instruction, or macro, that indicates that all iterations of the loop should be expanded before compilation and should execute in parallel. The third loop writes the modified bytes out to the UART.

Once compiled to a binary format, this design is programmed into an FPGA, causing it to modify its logic blocks and routing structure in such a way that it produces the behaviour specified. The creation of a hardware design that allows the FPGA to behave as a processor follows the same principles, although the design itself is much more complex. Instead of performing some dedicated task like that illustrated in the example above, the behaviour detailed in a processor design concerns reading instructions from an address in memory, interpreting them and then acting on them (the fetch–decode–execute cycle). Because both hardware and software are based on the same fundamental set of logic operations, designs typically used to specify hardware behaviour can be replicated in software (for example, emulators written in software that execute instructions intended for a different hardware architecture) and *vice–versa* (for example, hardware implementations of the Java Virtual Machine).

This important principle is what makes the flexibility of FPGAs so useful. In addition to programming an FPGA with a processor design to allow it to run software, custom additional "peripherals" (connected but independent hardware designs) can be used to provide specialised functionality implemented using the bare hardware. Needless to say, without significant tool support the process of creating and implementing these designs can be extremely difficult. As such, FPGA manufacturers provide extensive toolsets consisting of design software that allows designs to be created and programmed directly onto an FPGA using a PC or other general–purpose computer.

One such manufacturer is Xilinx, whose FPGA boards have been used in this project. Using their design software (Xilinx Platform Studio or XPS), it is possible to create complex hardware designs quickly by selecting the desired components using a wizard–like interface. Once a basic design project has been created by doing this, it is then possible to modify specific features of the included peripherals (such as the quantity of memory available) and customise the design for a specific application. Once programmed onto an FPGA, it is then possible to communicate with the hardware using an I/O interface on the board (such as its serial port) from software running on a PC.

At York, a custom laboratory consisting of an array of FPGAs (called the VLAB) has been created to allow hardware designs to be implemented and communicated with remotely. While used to some extent in this project, the VLAB's lack of support for the Xilinx debugging tools (discussed in section 2.4) as well as the delay in downloading hardware designs to boards (due to latency imposed by the network) posed significant problems when dealing with complex hardware and software implementations, ultimately resulting in the latter part of the work being done using a direct board connection.

## 2.4 MicroBlaze

MicroBlaze is a processor implemented using a hardware design that can be programmed onto Xilinx FPGAs. It consists of a relatively small instruction set (compared to mainstream architectures such as x86) and support for its designs is included out–of–the–box by the Xilinx tools, allowing them to be quickly built and programmed onto FPGAs, including in a dual–core configuration.

Dual–core MicroBlaze configurations do, however, present a specific challenge inherent to the architecture. There are no built–in synchronisation features that allow mutual exclusion to be implemented across multiple processors.[9, p. 227] The closest things available are the `lwx` and `swx` instructions which are intended to allow software to synchronise on a memory location when running on a single–core system. Unfortunately, due to the fact that the synchronisation data is stored internally in the processor, it is not possible to extend these features for use with multiple cores. Although mutual exclusion can be implemented through the use of a separate peripheral, this is entirely specific to the MicroBlaze platform.

In addition, the built–in hardware configurations available for the board used in this project do not allow the configuration of certain shared peripherals, such as UARTs and interrupt controllers. This means, for example, that a multicore system must be able to perform some sort of synchronisation between cores at the bootstrapping stage in order to ensure that each core's interrupts are installed in the correct order. This is discussed in more detail in section 5.2.1 on page 57.

The Xilinx SDK (version 13.1 was used in this project) is provided with the tools to support the development of software for MicroBlaze and the initialisation of FPGAs with compiled executables. It consists of a customised Eclipse distribution, including the CDT plugin for C and C++ support, as well MicroBlaze versions of the GCC toolchain. This includes a debugger which provides full support for debugging software over a connection to the FPGA. Unfortunately, there is a presumption in the implementation of the debugging tools that multiple cores will not ever execute the same code. This means that source code from a project targeted at one core cannot be used to step through execution of another core. The only alternative is to use the live disassembler, which supports the debugging features by displaying the assembler code being executed as it is in memory at the time, as opposed to the original C source. Although sometimes useful, it suffers from instability and often leads to very time–consuming debugging sessions, especially when inspecting complex code.

## 2.5 Real–Time Operating Systems

Fundamentally, the role of an RTOS is to execute application tasks in predictable quantities of time[1, p. 4] in order that they be able to reliably meet deadlines. The key to achieving this is in the deterministic execution of the RTOS code, ensuring as little fluctuation in execution time as possible.[12] Most importantly, it must be possible to have a *guarantee* about a real–time system's worst–case execution time. In terms of an RTOS, this means ensuring, for example, that a guarantee of the maximum number of processor cycles used to execute any given system code can be determined. In addition, RTOSes are also responsible for co–ordinating the scheduling of tasks in keeping with their assigned priorities.

The extent to which an RTOS can guarantee that task deadlines are met, a factor which also depends on the system's hardware and application software, determines its "hardness".[12] Hard real–time systems are those for which missed deadlines may result in catastrophic consequences, and are therefore not permitted; firm real–time systems are those for which deadline misses are permissable, but for which results gained after a missed deadline are unusable; soft real–time systems are those for which deadline misses are permissable but the quality of results gained after a deadline miss degrades.

The RTOS itself is not wholly, or even largely, responsible for making a system "real–time". The onus is on system developers to use the RTOS's features, such as priority–based scheduling, in order to build a real–time system using appropriate hardware and application software. However, any modifications made to an RTOS must take into account the need for predictability and determinism. They must also ensure the time spent performing OS–level tasks, particularly frequent ones such as interrupt handling, is as low as possible in order to minimise the effect on the ability for tasks to meet their deadlines.

### 2.5.1 Kernel

The kernel is the core component of an operating system. It provides a layer of abstraction between the application software and the hardware. Features such as memory allocation, interrupt handling, context–switching and task management are provided for use by application software through dedicated kernel APIs.

### 2.5.2 Scheduler

The scheduler is the operating system component responsible for deciding which tasks are allocated to the system's processors for execution. The way in which a scheduler decides *how* to perform this allocation is the key factor in determining its effect on elements of the system's performance, such as the equality with which tasks are given processor time, the task throughput, the time it takes for tasks to run to completion and the delay between requests and responses (also known as thread flyback time).[12]

In an RTOS, tasks are assigned numeric values that correspond to priorities. Tasks with higher priorities are supposed to execute in preference to those with lower ones, the priorities being assigned by system developers in a way intended to be optimal for the application being built. For instance, consider an application consisting of three tasks: the first waits for a network command indicating that log information should be sent to a waiting technician, the second monitors the status of the other system tasks, while the third performs important safety–critical processing upon the receipt of a message from an external source. The task responsible for the safety–critical processing would be assigned the highest priority as its responsibilities are such that it should always run in preference to the other tasks.

FreeRTOS employs a fixed priority pre–emptive scheduler.[13] A "fixed priority" scheduler is one that, when making a decision on which task to schedule on a processor, always chooses the one with the highest priority ready for execution. A scheduler that is "pre–emptive" instigates this decision at regular intervals, notifications of which are delivered to the operating system by means of interrupts generated by dedicated clock hardware. The FreeRTOS scheduler can also be made to run in co–operative mode, in which the process of suspending one task and choosing another to schedule is instigated only by tasks themselves. The tasks do this by explicitly "yielding" control of their processor core using the `taskYIELD` macro. Note that tasks running in pre–emptive mode are also able to do this. The scheduling behaviour of FreeRTOS after an interrupt–triggered yield is identical to that exhibited on an explicit yield. Modifications to FreeRTOS would need to be careful to preserve these semantics across all processors, ensuring that the scheduler's use of priority information remained the same.

At the heart of the FreeRTOS scheduler are "ready task (linked) lists". For each priority, there is one of these lists. The lists themselves consist of a data structure with pointers to the "current" item (task) in the list. Each item is a structure pointing to the next and previous list items, as well as consisting of a member for use in sorting the list. The scheduler determines the next task to schedule by retrieving the item at the head of the highest priority non–empty list. As well as ensuring that this item represents the highest priority task ready for scheduling, it is also guaranteed to be the one with the closest deadline, as each list is sorted according to its items' deadlines. Before an item is retrieved from the list, its "current item" pointer is incremented to point to the next item, meaning that the scheduling of a task automatically moves another task behind it forward in the list.

### 2.5.3 Context switching

"Context switching" is the process of changing the task being executed on a processor core. It consists of five steps:

1. **Initiate the context switch:** The context switch is initiated by calling the context switching code. This is normally done either by an interrupt handler (for example in response to a tick by the clock hardware), or indirectly by a running task that decides to yield control of the processor.

2. **Save the context:** Save the contents of the processor's registers to an area of memory reserved for use by the task being swapped out, such as its stack.

3. **Select a new task to schedule:** Delegate to the scheduler in order to determine the task to be swapped in.

4. **Restore the context:** Load the contents of the processor registers from the memory area associated with the task to be swapped in. This memory area contains the contents of the registers as they were saved when the task was last swapped out.

5. **Continue task execution:** This step is performed automatically as part of the previous one, but it is worth highlighting. The PC (program counter) register within the processor indicates the memory address containing the next instruction that the processor should execute. This register is included in those that are saved and loaded. This means that when the context is restored, the processor resumes execution at the address loaded into the PC register from when the task's context was last saved.

The performance of context–switching code is of special significance to all operating systems[14][15] due both to the frequency with which it is executed and to the large number of potentially time–consuming memory operations required. Although throughput is not the principle concern of RTOSes, ensuring that context–switching code is both predictable and fast is important in keeping task execution efficient.

## 2.6 Synchronisation

On multitasking systems, "thread safety" is a term used to describe the property of software that allows it to execute in multiple threads with freedom from race conditions arising from the interference of one thread with another. Such interference comes about when multiple threads access a shared resource in a sequence that causes them to behave in an undesired way.

Synchronisation (locking a resource on behalf of a thread of execution) is essential for making code thread safe by allowing access to shared resources to be carefully controlled. Synchronisation always occurs "on" a target. This target represents the shared resource to be locked. On single–core systems, the target on which tasks synchronise can simply be the processor core — as long as the processor is locked by an executing task on a system with only one processor, no other task will be allowed to execute and thus will have no opportunity to access the shared resource. FreeRTOS accomplishes this by disabling processor interrupts, thus preventing another task from being pre–emptively scheduled, on entry to a critical section.

Using the processor core as the synchronisation target on multicore systems is not feasible. Synchronising on the current processor core in a multicore system would serve only to restrict other tasks running on the same core from accessing the shared resource. It would provide no protection against access by tasks running on other processors. Synchronising on *all* processor cores in the system, while protecting the shared resource by preventing any other task from running, would be very inefficient. Even tasks not requiring any access to the shared resource would be unable to execute.

Instead, the synchronisation target needs to be refined. By uniquely identifying the shared resource, synchronisation can be implemented in such a way that a task in a critical section will only prevent other tasks from running if they require access to the same shared resource, regardless of which processor core they are running on.

Usually, synchronisation is architecture–specific. MicroBlaze, in particular, requires a special configuration for mutual exclusion to be supported at the hardware level. In order to understand the issue fully, it is important to consider the various options available in different system configurations.

### 2.6.1 Built–in Atomicity

Some processor architectures provide built–in operations that are atomic (indivisible) and that work across all cores. For example, on x86 the `xchg` instruction can be used to to exchange a register value with one at a specified memory location.[16] Because this exchange is guaranteed to be atomic across all cores, it can be used to implement synchronisation primitives such as semaphores.

MicroBlaze also provides instructions that allow atomic operations on memory (namely `lwx` and `swx`), but a core executing these instructions does not co–ordinate its actions with other cores in the processor. It thus does not provide atomicity across multiple cores, making these instructions inappropriate for implementing multicore synchronisation.

### 2.6.2 Mutex Peripheral

To enable mutual exclusion on multicore MicroBlaze processors at the hardware level, Xilinx provide a "mutex peripheral" that can be programmed into their FPGAs.[17] This peripheral is configured at design–time with various parameters, including the quantity of desired mutex objects. Software running on the processor cores can then interface with the mutex peripheral in order to request that a mutex be locked or released.

Rather than simply being hardware–specific (after all, the x86 atomic instructions are also hardware–specific), this method of implementing mutual exclusion is *configuration*–specific. The mutex peripheral is a prerequisite to supporting mutual exclusion using this method, and requires additional FPGA space as well as software specific to the peripheral in order to operate. In addition, because certain options (such as the number of supported mutexes) must be configured at the hardware level, related changes post–implementation may have a negative impact on the ease of maintenance of a system using this peripheral.

### 2.6.3 Software–based Mutual Exclusion

It is also possible to implement mutual exclusion in software alone, without the use of hardware–level atomic instructions. Several algorithms exist which allow this, and they all operate in the same basic way: using carefully–positioned memory reads and writes, the algorithms allow separate threads to determine whether or not the program counter of a thread competing for a shared resource has entered, or is about to enter, a critical section for the resource in question. The most simple of these solutions was created by Gary L. Peterson for synchronisation between two processes, and is best illustrated by way of an example:[18, p. 115]

Listing 2.3: Peterson's Algorithm

```

// Global, shared variables
volatile int Q[2] = {0, 0};
volatile int turn = 0;

// Called by process 0
void accessSharedResourceP0() {

  // Trying protocol for P0
  Q[0] = 1;
  turn = 1;
  while ((Q[1] == 1) && (turn == 1)) {  }

  // Critical section
  // Perform processing based on the shared variable

  // Exit protocol for P0
  Q[0] = 0;

}

// Called by process 1
void accessSharedResourceP1() {

  // Trying protocol for P1
  Q[1] = 1;
  turn = 0;
  while ((Q[0] == 1) && (turn == 0)) {  }
```

```

    // Critical section
    // Perform processing based on the shared variable

    // Exit protocol for P1
    Q[1] = 0; // Exit critical section

}
```

The algorithm is based on two variables: the `Q` array, for representing the intention of a process to enter the critical section, and the `turn` variable, for identifying whether a competing process is inside a critical section or about to enter it. Note that these variables are declared with the `volatile` keyword that prevents the compiler from optimising code in a way that might change the order in which these variables are accessed and thus break the algorithm. When a process wants to enter the critical section, it sets the element of `Q` indexed by its processor ID (in this case `0` or `1`) to `1` (denoting `true`). It then assigns the `turn` variable the index of the competing process.

At this point, both processes may be attempting to modify the `turn` variable at the same time. One will succeed before the other, and this process will exit the `while` loop first because as the competing process's write to the `turn` variable takes effect, the loop expression will no longer evaluate to true for the first process. The first process will then exit the loop and be within the critical section. Once finished, the process will set its "intention" in the `Q` array to 0 (denoting `false`), and a competing process busy waiting in the `while` loop will detect this and enter the critical section itself. Note that all elements in the `Q` array are initialised to `0` to indicate that in the initial state of the system, no process intends to enter the critical section.

While incredibly useful for explaining the principle of implementing mutual exclusion in software, this algorithm only supports two concurrent processes. Peterson includes a generalised version of the algorithm to support $n$ processes,[18, p. 116] although he acknowledges that it requires more memory than other solutions and, most importantly, satisfies fewer constraints.[18, p. 116]

In particular, his generalised algorithm does not satisfy the constraint of being free from starvation,[19, p. 38] despite some suggestions to the contrary.[20, p. 20] That is, the algorithm cannot guarantee that a process using it will not be perpetually denied resources. To understand why this is the case, it is necessary to understand the way in which Peterson generalised his two–process solution.[1][18, p. 116]

Listing 2.4: $n$–Process Peterson's Algorithm

```

int i = getCurrentProcessId();
int n = getNumProcesses();

int Q[n]; // Initialised to 0
int turn[n - 1]; // Initialised to 1

/* Protocols for P1 */
for (int j=1; j<n; j++) {

  Q[i] = j;
  turn[j] = i;

  int stageClear;

  do {

    stageClear = 1;

    for (int k=0; k<n; k++) {

      if (k == i) continue;

      if (Q[k] >= j) {

        stageClear = 0;
        break;

      }

```

[1]The syntax of the algorithms presented in Peterson's paper have both been modified to represent a C implementation.

```
31      }
32
33    } while ((stageClear == 0) && (turn[j] == i));
34
35  }
36
37  // Critical section
38
39  Q[i] = 0;
```

The `Q` variable is still an array of size $n$, where $n$ is the number of processes. The `turn` variable becomes an array of size $n$ - 1. The principle remains the same: in order to enter the critical section, a process must satisfy itself that all the other processes are "behind" it in the process of entering. The value assigned to a process's entry in `Q` denotes its progress in checking where the other processes are relative to the critical section. It is helpful to think of the values in `Q` as denoting the "stage" at which each process is currently working to determine its eligibility to continue. A process busy waits at each stage until it is able to progress to the next.

If a process finds another that provisionally appears to be ahead of it (line 24), the `stageClear` variable will be set to `0` (denoting false). The `do...while` loop expression will then test both this variable and the `turn` array, which will indicate if the other process detected to be ahead has fallen behind since the assignment to `stageClear` on line 26. If the other process is still ahead, the current one must continue to wait by repeating the `do...while` loop. However, if the other process has fallen behind or if there are no other processes ahead of it, the current process can continue to the next stage. If the current stage (denoted by the `j` variable) is the last one, the process enters the critical section.

As Alagarsamy explains, Peterson's generalised algorithm does not enforce any upper–bound on the number of "bypasses" that can occur.[19, p. 36] A bypass is a situation in which a process that is made to busy wait at the first stage is "overtaken" by other processes before it can progress to a higher stage.[19, p. 38] This can occur because a process blocked at the first stage, regardless of the duration for which it has been blocked, is at the same stage as a process showing interest in the critical section for the first time, and can thus be "beaten" to the second stage. A process that is continually overtaken in this manner can thus find itself perpetually denied access to the critical section.

Alagarsamy also demonstrates an inefficiency first highlighted by Block and Woo caused by the fact that each process must cross $n$ - 1 stages, meaning that even processes that may have no interest in the critical section must have their elements in `Q` checked by a process wishing to enter.[19, p. 38] Alagarsamy introduces his own algorithm that both promises "optimal bypasses" and solves this inefficiency by incorporating the modifications made by Block and Woo.[19, p. 38] Alagarsamy's solution to the unbounded bypass problem works by having processes "promote"[19, p. 38] others at lower stages when they enter the critical section. This ensures that even a process blocked at the first stage will be guaranteed to move to the next once a process enters the critical section.

A challenge that exists with all of the algorithms discussed above arises from the fact that processors will often re–order memory accesses internally, for example to improve performance. This behaviour cannot be controlled at design–time and must be mitigated using a "memory barrier". A memory barrier is a point in a program, represented by the issuing of a special–purpose instruction, after which the processor must guarantee that all outstanding memory operations are complete. On x86, the `mfence` instruction performs this function; on MicroBlaze the equivalent is the `mbar` instruction. By using these instructions strategically, it should be possible to avoid the effects of re–ordering on the algorithms.

While software–based mutual exclusion arguably introduces certain inefficiencies into the implementation of synchronisation features, the benefits of providing largely hardware–agnostic mutual exclusion cannot be ignored, particularly in a system ported to as many platforms as FreeRTOS.

## 2.7 FreeRTOS

FreeRTOS is an RTOS written in C. As of August 2011, it has been ported to twenty–seven different architectures.[3] As well as supporting fixed–priority pre–emptive and co–operative scheduling, the FreeRTOS kernel API also exposes useful functionality to application software, such as queue manipulation and task management. However, perhaps the most impressive aspect of FreeRTOS is that which has enabled its use on so many different platforms: its division into two architectural layers, the "hardware independent" layer, responsible for performing the majority of operating system functions, and the "portable" layer, responsible for performing hardware–specific processing (such as context switching). The hardware independent layer is the same across all ports — it expects a standard set of functions to be exposed by the portable layer so that it can delegate platform–specific processing to it, regardless of which hardware the port is intended for.

As well as being useful in practice (through allowing code re–use and platform–independent application code), this aspect of FreeRTOS has allowed some of the modifications made in this project to be initially implemented and tested on a port that uses Windows to simulate access to the hardware. As well as making debugging easier, this "simulation" port provides a very good opportunity to learn about how FreeRTOS works, the use of the Windows API providing a relatively easy, high–level perspective of the portable layer's responsibilities.

## 2.7.1 Overview

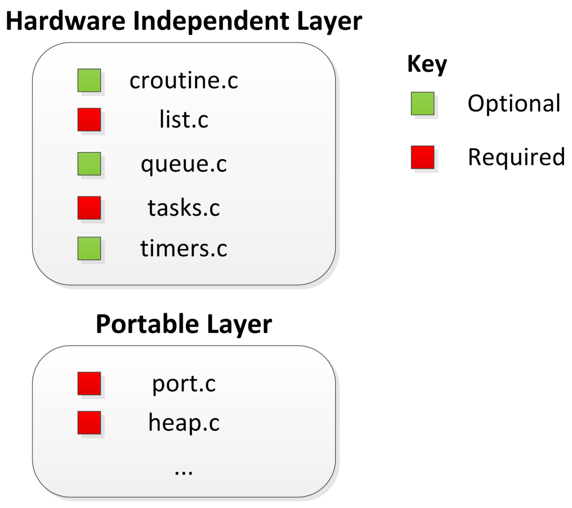

Figure 2.3: FreeRTOS source structure

The division between the hardware independent and portable layers is expressed in terms of the source files in Figure 2.3.[1] The two required files, "list.c" and "tasks.c", provide the minimum high–level functionality required for managing and scheduling tasks. The function of each source file is explained below:

**croutine.c** Provides support for "co–routines". These are very limited types of tasks that are more memory efficient.[21]†

[1]Header files are not included in this illustration.

**list.c** Implements a list data structure for use by the scheduler in maintaining task queues.

**queue.c** Implements a priority queue data structure accessible to application code for use in message passing.†

**tasks.c** Provides task management functionality to both application tasks and to the portable layer.

**timers.c** Provides software–based timers for use by application tasks.†

**port.c** Exposes the standard portable API required by the hardware independent layer.

**heap.c** Provides the port–specific memory allocation and deallocation functions. This is explained below in more detail.

Note that the application code is not illustrated in Figure 2.3. This consists of a global entry point to the system, functions representing the system's various tasks and the inclusion of FreeRTOS header files for access to system APIs.

The hardware–independent layer is started by this global entry point located in the application code. It takes the form of a `main` function and, as the entry point to the entire system, is responsible for creating all the initial application tasks and starting the FreeRTOS scheduler. Note that a FreeRTOS task is most analogous to a thread in a conventional operating system: it is the smallest unit of processing within FreeRTOS. Indeed, due to the fact that FreeRTOS distributions are typically highly specialised, no distinction is made between tasks and processes.

## 2.7.2 The Life of a Task

When a task is created using the `xTaskGenericCreate` function, FreeRTOS first allocates memory for the task. This involves both allocating memory for the task's stack (the size of which can be specified in the function call) as well as for the task control block (or TCB) data structure which, among other things, stores a pointer to the task's stack, its priority level and also a pointer to its code. The only hard, design–time restriction on this code is that its variables must fit within the stack size specified when the task is created. A task should also not end

† These components are excluded from the scope of this project.

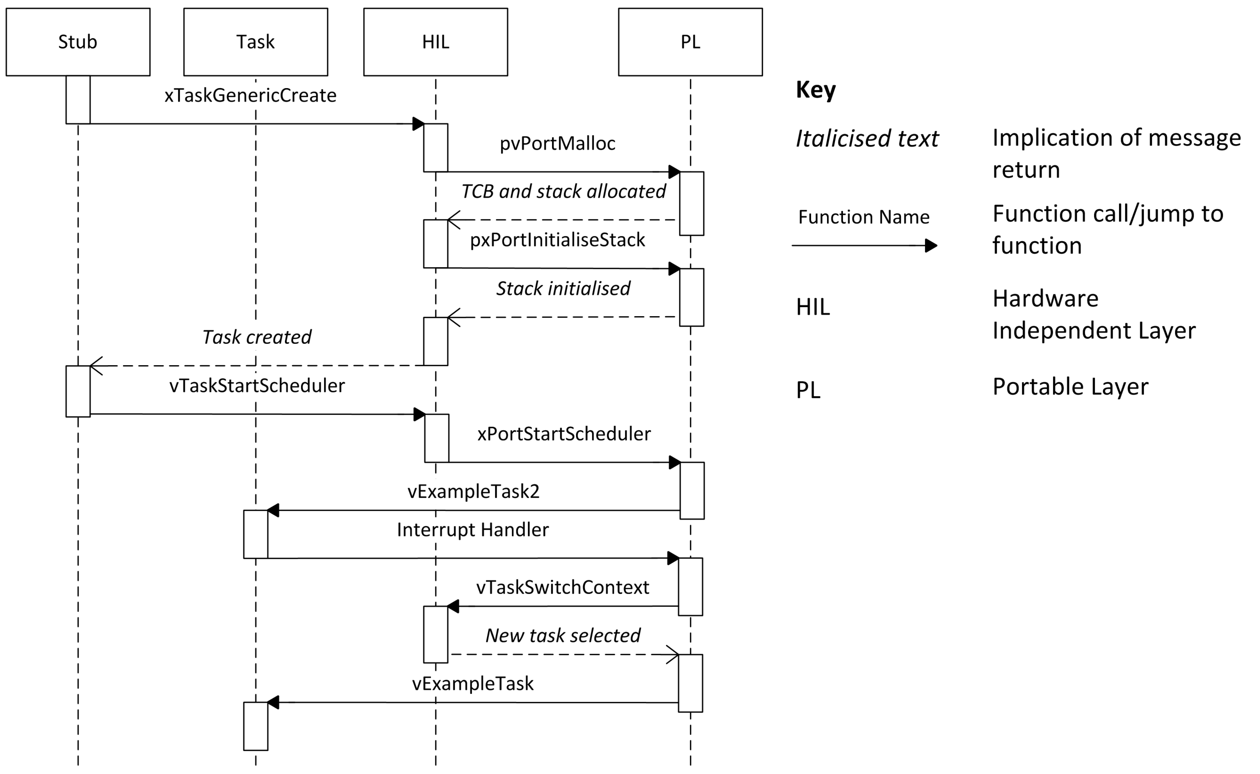

Figure 2.4: Pseudo–UML sequence diagram showing a task's creation and execution

without first deleting itself using the vTaskDelete API function in order to free up memory it uses.

Listing 2.5: Skeleton FreeRTOS task

```

static portTASK_FUNCTION( vExampleTask, pvParameters ) {

  for (;;) {

    // Perform task processing here

  }

}
```

The skeleton task shown above consists of a private (static) function declaration created by the use of the portTASK_FUNCTION macro. This macro accepts the name of the task function as the first parameter and the name of the task “parameter” argument as the second (used for passing information to the task when it first starts). Expanded when using the MicroBlaze port, the above macro looks as follows:

Listing 2.6: Task signature after macro expansion

```
void vExampleTask( void *pvParameters )
```

This macro is defined in the portable layer and its use improves the portability of code by ensuring that task function signatures are always correct for the platform on which the code is compiled.

Once created, the actual memory allocation for a task is delegated to the bundled memory manager, which is a component of the portable layer. The memory manager is expected to expose two functions: pvPortMalloc (for memory allocation) and pvPortFree (for memory deallocation). How these functions manage the memory internally is up to the specific implementation: included with FreeRTOS are three different memory managers, each with different behaviour. The first and most simple allows memory allocation but not deallocation; the second permits releasing without combining small adjacent blocks into larger contiguous ones; and the third relies on the compiler’s own malloc and free implementations but does so in a thread–safe way.

Once allocation is done, `xTaskGenericCreate` then delegates the initialisation of the stack area to the portable layer using `pxPortInitialiseStack`. The purpose of this function is to prepare the stack area of the newly created task in such a way that it can be started by the context switching code as if it was a suspended task being resumed. Once created, the task is then added to the ready queue in preparation for it to be scheduled at a later time. When no tasks run, a system–created idle task takes control of the processor until it is swapped out. This can occur either as the result of a context switch, or because the idle task is configured to yield automatically by setting the `configIDLE_SHOULD_YIELD` macro to `1`.

Eventually, the stub will start the FreeRTOS scheduler using `vTaskStartScheduler` in the hardware independent layer, although the notion of a scheduler entirely provided by FreeRTOS is not accurate. Scheduling, by its very nature, straddles the line between the hardware independent and portable layers. On the one hand, it is heavily dependent on the handling of interrupts (when in pre–emptive mode) and the performance of context switching, things firmly in the realm of the portable layer. On the other, it involves making high level decisions about which tasks should be executed and when. This is certainly something that would not be worth re–implementing in each portable layer.

As such, both the hardware independent and portable layers are partially responsible for correctly scheduling tasks. After performing port–specific initialisation, the portable layer's `xPortStartScheduler` function installs timer interrupts (if in pre–emptive mode) and then starts the first task. If not in pre–emptive mode, context switches will occur when tasks yield by using the `taskYIELD` macro. When a yield occurs (whether as a result of an interrupt or an explicit yield), the portable layer is invoked. It saves the current context, calls the hardware independent layer to determine the task to swap in, and then restores this task's context in order to resume it. The hardware independent layer stores a pointer to the TCB of the currently executing task in the `pxCurrentTCB` variable.

### 2.7.3 MicroBlaze Port

The MicroBlaze port requires an interrupt controller to be included in the hardware design. An interrupt controller accepts multiple interrupt sources as input, forwarding received interrupts to a processor in a priority order. The main reason this is necessary on MicroBlaze is that the architecture only has one interrupt line[22] — the interrupt controller is relied upon to receive and queue interrupts before the processor is notified. As a result, the portable layer must configure and enable the processor's interrupt controller before any tasks can be started. This is done by applications in their `main` function by calling `xPortSetupHardware`. This function is also responsible for initialising and enabling the processor's data and instruction caches, if used. However, this is beyond the scope of this project.

Once the FreeRTOS scheduler has been started by application code using the hardware independent layer, the `xPortStartScheduler` function in the portable layer is called. This then configures the hardware clock responsible for generating tick interrupts. This allows FreeRTOS to keep track of time, and when in pre–emptive mode, to trigger task re–scheduling. The first task is then started by a call to `vPortStartFirstTask`.

`vPortStartFirstTask` is the first of several functions written in assembler to perform the low–level swapping in and out of tasks and interrupt handling. Recall that after the allocation of each task's stack, the memory is initialised to appear as though the task had been previously swapped out. `vPortStartFirstTask` serves two important purposes:

1. It assigns a pointer to point to the "global stack frame". This is the memory address at which the stack of the calling function (`vPortStartFirstTask`) ends. A pointer to this location is needed in order to provide somewhere for the interrupt handler to create a temporary stack frame for use in storing parameters passed to a function called to handle low–level aspects of the interrupt handling (`XIntc_DeviceInterruptHandler`).[23] This is necessary because in the MicroBlaze architecture, a function (the caller) calling another function (the callee) stores parameters to be passed to the callee on its own (the *caller's*) stack.[9, p. 129][23] Note that this does not cause conflict with the stacks of tasks because their stacks are located on the heap and dynamically allocated by the hardware independent layer.

2. It calls `vPortRestoreContext` in order to load the context of the first task to schedule and jump to its entry point.

The other of these low–level assembler functions perform context switching as described in section 2.5.3. `vPortSaveContext` transfers the contents of the processor's registers, including the address at which to continue execution, into memory; `vPortRestoreContext` loads a saved context from memory into the processor's registers, before either jumping to the address saved or jumping to the address saved *and* enabling interrupts (depending on whether interrupts were enabled when the context was last saved).

_interrupt_handler is the function to which the processor jumps when an interrupt occurs. This saves "volatile" registers (those which may change before the dedicated context–saving code is executed) to memory, after which it calls vPortSaveContext. Once this returns, it clears the yield flag (a flag denoting whether or not a task should yield control of the processor) and then calls the low–level handler, XIntc_DeviceInterruptHandler. This is provided by Xilinx in a small support library. Its purpose is to perform the necessary housekeeping in order to ensure that interrupts are correctly acknowledged. It also calls an optionally defined "callback" function associated with each interrupt. For clock interrupts, this is configured by the portable layer to be the vTickISR function which simply increments the system tick counter and, if in pre–emptive mode, sets the yield flag. Control then returns to _interrupt_handler which tests the yield flag. If it indicates a yield should occur (which it always does in pre–emptive mode), it calls vTaskSwitchContext in the hardware independent layer which selects a task to "swap in" by pointing to it using the pxCurrentTCB pointer. vPortRestoreContext is then called, which restores the context of the task denoted by pxCurrentTCB. Note that if the yield flag indicates that a yield should *not* occur, vTaskSwitchContext is not called and the task that was interrupted is resumed by vPortRestoreContext.

vPortYieldASM performs the same function as _interrupt_handler, with two important exceptions: it does so in response to an explicit call by application code or the hardware independent layer, as opposed to an interrupt; and by its very nature, it assumes that a yield is desired.

### 2.7.4 Windows Simulator

The Windows simulator port consists of a FreeRTOS portable layer that uses the Windows API to perform low–level operations, allowing it to be run "on top" of Windows as a console application. This port has been used to both learn about the operation of FreeRTOS and investigate the changes required to allow it to execute in a multicore environment. Apart from being less complex and more high level, the port also lends itself well to experimentation by virtue of the fact that it can be debugged more easily using the Windows port of the GDB debugger.

The port uses Windows threads to simulate the behaviour of bare hardware. A pool of threads is created on startup, each of which represents an application task. Tasks not selected for execution by the FreeRTOS scheduler (in the hardware independent layer) are kept in the blocked state, while the executing task is unblocked. Meanwhile, another Windows thread runs to simulate the hardware clock, using `sleep` system calls to block for a specified quantity of time before notifying yet another Windows thread (of a higher priority) responsible for interrupt handling. Once notified, this thread executes the registered interrupt callbacks, delegates to the scheduler to pick a new task to resume and then simulates this behaviour by suspending and resuming the appropriate threads representing tasks.

Mutual exclusion is achieved by synchronising on Windows mutex objects created by calling the `CreateMutex` API function. The portable layer uses the Windows API to block a thread representing a task wanting to enter a critical section until the appropriate mutex object is available (using the `WaitForSingleObject` or `WaitForMultipleObjects` functions).

Note that while this port may be incredibly useful in investigating high–level modifications to FreeRTOS, there are many things specific not just to a particular architecture, but to bare hardware in general, that it cannot simulate. For example, the details of how context–switching occurs, the configuration of hardware components (such as interrupt controllers) and the unrestricted access to system memory are all aspects of a real operating system that cannot be represented in this kind of simulation.

# Chapter 3

# Problem Analysis

## 3.1 Overview of Objectives

The principal objective of the implementation is to create a version of FreeRTOS that schedules tasks for concurrent execution on a MicroBlaze processor with two cores, and provides a method of safe communication between tasks regardless of the core on which they are executing. By combining the kernel modifications necessary to achieve this with a modified version of Tyrel Newton's MicroBlaze portable layer, the aim is to demonstrate the efficacy of the modifications and provide a platform upon which FreeRTOS can be further developed for use in multicore environments.

The project's functional and non-functional requirements are detailed below. All requirements are assigned, among other things, an identifier and a type. The identifier is of the form "REQ-$xy$-$n$": $x$ is either the character "P" (denoting a primary requirement) or "S" (denoting a secondary requirement); $y$ is either the character "F" (denoting a functional requirement) or "N" (denoting a non-functional requirement); and $n$ is an integer uniquely identifying the requirement. The requirement type denotes its relative importance: primary requirements must be met; secondary requirements may be met, depending on the remaining implementation time after all primary requirements have been completed. Note that all requirements are numbered sequentially, regardless of their type or their "functional" status.

## 3.2 Functional Requirements

### 3.2.1 FreeRTOS must allow the concurrent scheduling of tasks on a dual–core processor

**Identifier** REQ-PF-01

**Type** Primary

**Description** Excluding the optional hardware independent layer components (such as the queue API), modifications will be made to the FreeRTOS hardware independent layer so that it is able to schedule tasks on multiple processors concurrently.

**Source** Project Description

**Fit Criterion** Multiple tasks, for which parallel execution would provide observable execution time advantages, execute at least 1.5 times faster using the modified version of FreeRTOS on a dual-core processor than on the unmodified version of FreeRTOS on a single-core processor (with all other hardware configuration being equal). See section 6.1 on page 74 for details of the test strategy.

### 3.2.2 Operating system components must be thread safe

**Identifier** REQ-PF-02

**Type** Primary

**Description** Relevant operating system components will have access to synchronisation functionality in order to provide thread-safety within the FreeRTOS API.

**Source** Requirements Analysis

**Fit Criterion** Within 25,000 synchronisations, the kernel synchronisation features do not cause deadlock or allow concurrent access to the protected resource. See section 6.1 on page 74 for details of the test strategy.

### 3.2.3 OS-level thread-safety must be implicit

**Identifier** REQ-PF-03

**Type** Primary

**Description** Operating system components will perform synchronisation automatically when required, and will not rely on application tasks to ensure their thread safety.

**Source** Requirements Analysis

**Fit Criterion** With the exception of API changes unrelated to synchronisation, API calls in the unmodified version of FreeRTOS will remain the same and will use the synchronisation features specified in REQ-PF-02.

### 3.2.4 Application tasks must have access to an API providing mutual exclusion

**Identifier** REQ-PF-04

**Type** Primary

**Description** Tasks must be able to access shared resources in a thread-safe manner without implementing mutual exclusion solutions themselves, and thus must be able to synchronise using an API exposed by FreeRTOS.

**Source** Requirements Analysis

**Fit Criterion** Using no more than 2 API calls, application tasks will be able to invoke a mutual exclusion API for the protection of a named resource.

### 3.2.5 The task mutual exclusion API must be thread safe

**Identifier** REQ-PF-05

**Type** Primary

**Description** Tasks will have access to synchronisation functionality in order to allow the implementation of thread-safety within applications.

**Source** Project Description

**Fit Criterion** Within 25,000 synchronisations, the task–level synchronisation features do not cause deadlock or allow concurrent access to the protected resource. See section 6.1 on page 74 for details of the test strategy.

### 3.2.6 The MicroBlaze port of FreeRTOS must be modified to be compatible with the multicore-enabled hardware independent layer

**Identifier** REQ-PF-06

**Type** Primary

**Description** In order to run the modified version of the hardware independent layer on bare hardware (i.e. without using a simulated environment), the MicroBlaze portable layer developed by Tyrel Newton will be modified to support a multicore architecture and made compatible with the changes in the hardware independent layer.

**Source** Requirements Analysis

**Fit Criterion** As for REQ-PF-01 on page 35, with the constraint that the software is running directly on MicroBlaze processors programmed into an FPGA.

### 3.2.7 Tasks must be created to demonstrate concurrency and synchronisation

**Identifier** REQ-PF-07

**Type** Primary

**Description** To demonstrate the use of concurrency and synchronisation implemented by the FreeRTOS modifications, appropriate tasks will be created to exemplify the features added.

**Source** Project Description

**Fit Criterion** Demonstration tasks will illustrate the use and effectiveness of both the concurrent scheduling and the shared resource synchronisation features in the modified version of FreeRTOS. See section 6.1.4 on page 78 for details of the demo tasks.

### 3.2.8 The implementation must be completed within 45 days

**Identifier** REQ-PF-08

**Type** Primary

**Description** In order that enough time is allowed for the thesis write-up, the implementation will take no longer than 45 days. By assigning priorities to the project objectives, it will be possible to ensure that a minimum set of requirements are met within this timeframe. Additional time will be used to meet secondary requirements.

**Source** Initial Schedule

**Fit Criterion** Primary requirements take no longer than 45 days to complete.

### 3.2.9 Modifications made to FreeRTOS must scale for use with $n$ cores

**Identifier** REQ-SF-09

**Type** Secondary

**Description** It would be unwise to restrict multicore support in FreeRTOS to merely support for dual-core processors. The reason that this has been done so far is to limit the potential project scope given the highly restricted time available in which to do the work. In actual fact, modifications should scale transparently for use on processors with more than 2 cores.

**Source** Requirements Analysis

**Fit Criterion** Fit criteria of all functional requirements hold when the modifications are executed on a processor with 3 cores, 4 cores, 5 cores, 6 cores, 7 cores and 8 cores (the maximum supported by the Xilinx software development tools).

## 3.3 Non-Functional Requirements

### 3.3.1 Source code must be made publicly accessible

**Identifier** REQ-PN-10

**Type** Primary

**Description** In order to provide a record of the project's achievements, as well as to ensure compliance with the open-source licence under which FreeRTOS is distributed, the modified version of FreeRTOS created as part of this project will be made publicly accessible via an Internet-based source code repository.

**Source** Requirements Analysis/Ethical Considerations

**Fit Criterion** All deliverable source code is available publicly at the following URL: `http://sourceforge.net/projects/freertosxcore/`

### 3.3.2 Task development must not be made more difficult

**Identifier** REQ-PN-11

**Type** Primary

**Description** One of the virtues of FreeRTOS extolled in section 1.1 on page 1 is its simplicity. In order to minimise the effect on application development time, it is important that this aspect of the operating system is preserved through the modifications made in this project.

**Source** Requirements Analysis

**Fit Criterion** The process of converting a task of between 250 and 350 lines in length designed for use with the unmodified version of FreeRTOS to make it compatible for use with the modifications made in this project must take a developer familiar with both versions no longer than 5 minutes.

### 3.3.3 The hardware design used to run the modifications must be simple

**Identifier** REQ-SN-12

**Type** Secondary

**Description** Complex hardware requirements have the potential to make any modifications difficult to reproduce. Thus, the hardware design used must be reproduceable using the XPS base system builder (an automated design helper) with as few manual changes required as possible.

**Source** Requirements Analysis

**Fit Criterion** Excluding the compilation of the hardware design to a bitstream, the time required to reproduce the hardware design used in this project will not exceed 15 minutes for a competent software engineer with detailed instructions, the required apparatus and no experience of hardware design.

# Chapter 4

# Design

## 4.1 Multicore Hardware

### 4.1.1 FPGA Board

FPGA boards, the circuit boards connecting together FPGA chips with other peripherals such as off-chip memory and I/O components, are available in various configurations and specifications. While peripheral support is one aspect of the differences between boards, this project's use of peripherals is very limited — only memory (BRAM and DDR RAM) and serial ports are required by the software produced. More significant, then, are the specifications of the FPGA chip bundled with the circuit boards.

The FPGA chips available for use in this project were limited to two Xilinx models: the Spartan-3e and the Virtex-5. These chips both occupy opposite ends of the spectrum in terms of performance and capacity, with the Spartan-3e designed to be affordable for high-volume applications and the Virtex-5 for applications requiring high performance.

At first, a board using the Spartan-3e was used for initial work and experimentation. This chip is widely used within the department at York, and the VLAB contains a relatively large array of them. This means that as well as being abundant, there are many researchers at the department experienced with the use of these chips and able to provide expertise in solving related problems. This proved invaluable at the earlier stages of the project, but during the course of investigating the type of memory to use for the modified FreeRTOS implementation the Spartan-3e was found to have an insufficient quantity of on-chip memory (or "BRAM"), particularly when additional libraries useful for debugging were compiled. Also, the available resources on a Spartan-3e chip are relatively sparse, to the extent that dual-core hardware designs leave little room for additional on-chip peripherals beyond the processor itself.

The Virtex-5 solves both of these problems, providing twice the quantity of BRAM and a far greater number of resources available to hardware designs. Although the BRAM issue ultimately became irrelevant (due to the fact that DDR memory was chosen for FreeRTOS instead), it was nevertheless decided to use the Virtex-5 chip in order to provide an implementation that could be easily extended in the future to support additional peripherals such as networking hardware. This decision was supported by the fact that hardware designs had been successfully implemented using the Virtex-5 while it was still thought that the additional BRAM was required, allowing experience with the chip to be accrued. Another advantage to this chip presented itself when it was decided to stop using the VLAB in preference to a direct FPGA board connection: the Virtex-5 board is owned privately by one of the supervisors of this project and is not used by anyone else in the department.

Because the department only has one Virtex-5 board, this decision might be seen as something of a double–edged sword. Using the hardware design produced in this project may require the purchase of additional boards, although this is less of an inconvenience than it might at first seem. For a start, the hardware design *will* fit onto a Spartan-3e chip, and can be easily replicated for the chip using XPS without significantly deviating from the Virtex-5 implementation steps detailed in section 5.3 on page 66. In addition, the extension of the work presented will likely involve the use of more peripherals and therefore more complex hardware designs that may not fit onto the Spartan-3e, in which case a similar board with more resources would be required anyway.

### 4.1.2 Hardware Design

A fundamental design decision that spans the hardware and software aspects of this project is that of using an SMP architecture. There are several advantages to having both cores in the system execute the same code: as well as simplifying the software components by eliminating the need for separate core–specific instances of FreeRTOS, it allows memory to be directly shared by both cores without having to implement complex memory segregation. For example, using the alternative asymmetric architecture memory would have to either be segregated at the hardware level (using separate memory peripherals for each core) or at the software level (by restricting memory allocation to memory areas reserved for the core doing the allocation).

This complexity may to some extent be justified by a reduction in the need for synchronisation, as most of the memory access occurs on memory that is private to the executing core. However, any cross-task communication would still require full concurrent synchronisation. Also, in order to truly benefit from multiple cores, an asymmetric system would need a way to transfer tasks (and their associated memory) from one core to another. This is because it must be possible for waiting tasks to be executed by any core, regardless of which core's memory area the task first allocated its memory on. Consider the following: a waiting task was previously scheduled on core 0 and therefore allocated its memory in that core's private memory area. However, after being swapped out by this core and left waiting to be executed, core 1 becomes available and ready to start executing the task. At this point, core 1 must gain access to the task's allocated memory in order to start executing it, but the memory is located in an area private to core 0.

If core 1 were to simply ignore the task and start executing another that (more conveniently) was using its own memory area, the efficiency of the system would depend on an adequate balance of tasks assigned to each core. If one core finished executing all tasks using its memory area, that core would then become idle and be unable to share the processing load of tasks on other cores. Instead, it would make much more sense to transfer the task's memory to a location accessible from the core ready to start executing it. Apart from implying a degree of complexity in the context-switching code, there are performance issues with this as well. In pre-emptive mode, context switches happen very frequently (many times per second) and the implications on speed of moving even moderate quantities of memory this often are significant.

An SMP architecture avoids these issues. A single operating system instance is located in shared RAM accessible to all cores, with each task's memory also allocated there. This allows the queue of waiting tasks, for example, to be accessed by all cores and therefore allows each one to execute any task, regardless of whether or not they initially allocated the task's memory. In order to prevent one core from simultaneously allocating the same memory as another, synchronisation is used to protect the memory allocation code with a critical section.

The only exception to the use of shared memory is in the initial startup stage of the system, in this project termed the "core ignition". This is a necessary bootstrapping process that puts the cores into a state in which they are ready to begin executing the operating system and tasks symmetrically.

In order to provide support for pre-emptive scheduling, timer and interrupt controller peripherals are provided for each core. The timer sends interrupt signals at regular intervals

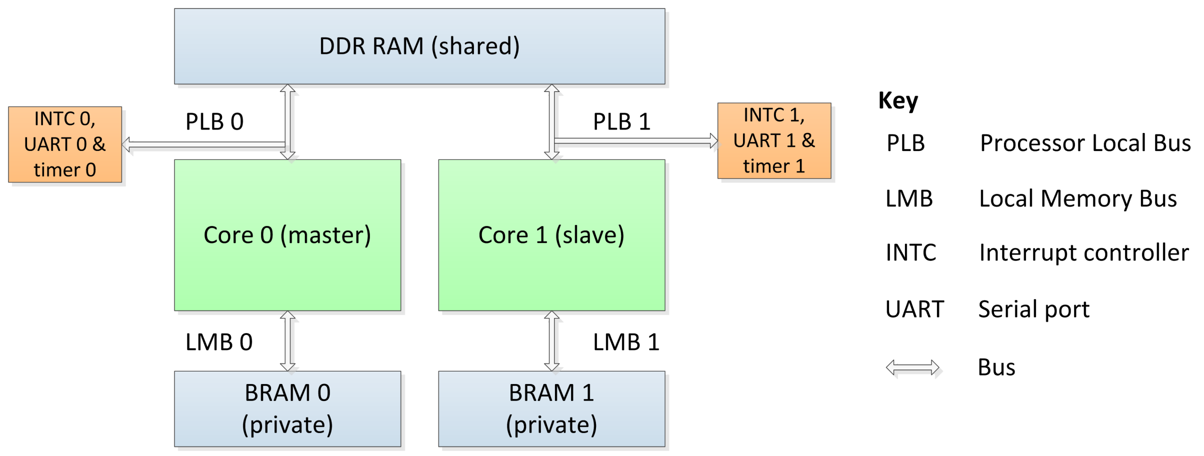

Figure 4.1: Hardware design

which are received by the interrupt controller. This then queues them until the core is ready to respond, at which point the core begins executing its dedicated interrupt handler as registered during the hardware configuration on startup. Unfortunately, the XPS tools do not allow interrupt controllers to be shared, meaning that separate timers and interrupt controllers for each processing core are included in the hardware design. Whether or not this restriction is artificially imposed by the automated XPS components or inherent to MicroBlaze has not been explored: as outlined by requirement REQ-PN-12 (section 3.3.3 on page 40), the priority was to keep the design simple, and in particular ensure that it varied as little as possible from the stock dual-core design provided by Xilinx.

This also explains the use of separate "buses" (lines of communication between cores and peripherals). Usually, SMP systems make use of shared buses for access to memory. However, with core–specific peripherals such as interrupt controllers inherently requiring their own core buses, these are also used for access to shared memory (PLB 0 and PLB 1 in Figure 4.1). Again, in the interests of creating a hardware platform resembling the Xilinx stock design as closely as possible, the feasibility of implementing a shared bus has not been explored.

## 4.2 FreeRTOS Modifications

### 4.2.1 Core Ignition

When the master core first starts, it enters the application entry point (the `main` function) and performs application–specific initialisation (such as the creation of tasks), as well as some hardware configuration, such as the set up of the master core's interrupt controller and hardware timer, the code for which is located in the core's private BRAM. It then starts the FreeRTOS scheduler and begins executing the first task.

Simultaneously, the slave core starts. When it does, it begins executing the ignition code located in its small, private BRAM. This ignition is responsible for performing the hardware configuration of the slave core's own peripherals (this can only be done by itself), as well as using a small area of shared RAM to communicate with the master core in order to determine when the FreeRTOS scheduler has been started, as it is only after this happens that the slave core can begin executing the FreeRTOS code in shared memory and scheduling tasks itself. At this point, the cores are executing symmetrically and the code in BRAM is never executed again with one exception: when an interrupt is fired, a small interrupt handler "stub" in the slave core's BRAM runs and immediately passes execution to the master interrupt handler in FreeRTOS.

### 4.2.2 Memory Model

The memory model used is quite simple. Core-specific code used to interact with the hardware is located in BRAM. This code is called for use in configuring peripherals, such as starting the hardware timer, assigning an interrupt handler to the interrupt controller and acknowledging hardware interrupts. The code is provided by Xilinx as a small library of C code, and is used both during the core ignition phase and by the interrupt handler.

When the core first starts, it jumps to the start address. This address contains a branch instruction causing the core to jump to a software entry point: in the case of the master core, this is the `main` function that performs hardware configuration and then ultimately starts the FreeRTOS scheduler; in the case of a slave core, this is the "core ignition" `main` function (located in BRAM) that also performs hardware configuration, but then waits until it is notified by the master core that the scheduler has been started.

Once the slave core has detected the start of the FreeRTOS scheduler (the cores communicate

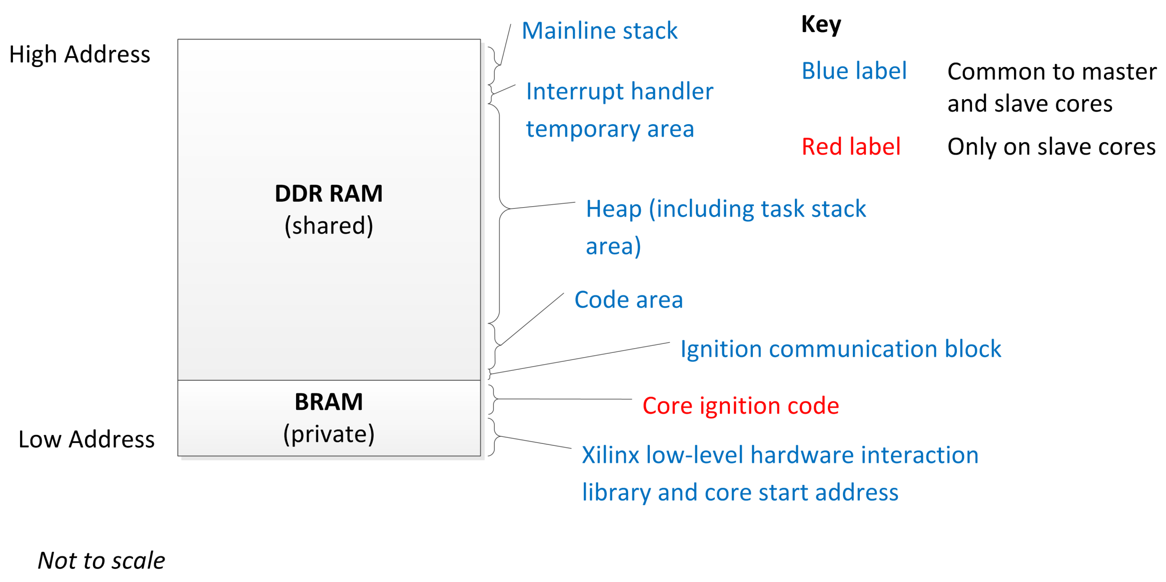
High Address
Low Address
DDR RAM
(shared)
BRAM
(private)
Mainline stack
Interrupt handler temporary area
Heap (including task stack area)
Code area
Ignition communication block
Core ignition code
Xilinx low-level hardware interaction library and core start address
Key
Blue label
Common to master and slave cores
Red label
Only on slave cores
Not to scale

Figure 4.2: Memory model

using an area of shared memory called the ignition communication block), it begins executing FreeRTOS. The slave core jumps to the scheduler, selecting a task for execution on the slave core, loading its context and then enabling interrupts for the core. When an interrupt is fired, the core jumps to the interrupt handler code which, in the case of the timer interrupt, increments the system tick count. In pre-emptive mode, these timer interrupts will ensure that the core automatically divides its time between waiting tasks.

The code of a task is located in the code area at the bottom of shared memory just after the ignition communication block. This consists of the actual instructions that make up FreeRTOS and the application tasks. The heap, perhaps a bit confusingly, is used for the storage of statically–allocated task variables defined at design–time. This is because, on creation of a task, FreeRTOS allocates a quantity of memory on the heap to be used as the task's stack. However, this is also the area used for memory dynamically allocated by tasks at runtime.

This is because, typically, if tasks were executed as simple functions outside of the context of FreeRTOS, their stacks would be located in the "mainline stack" area at the top of shared memory. The stack pointer would automatically be maintained by code generated by the compiler on entry to and return from functions, which would of course execute asynchronously. The mainline stack would thus grow and contract depending on the call stack. While generally a transparent and fundamentally important mechanism of a compiler, this is of little use to FreeRTOS when executing tasks. As tasks are swapped in and out, task–specific stacks need to be used to maintain their local data (such as variables). This is because a single stack would quickly become corrupted if used by multiple tasks in separate threads of execution — the information specific to a task being swapped in would overwrite the information of the task being swapped out. Note that the exact method of allocation is left to the compiler's implementations of `malloc` and `free`, with critical sections used to make the process thread–safe.

The mainline stack *is* used, albeit not very much. When the master core enters the application entry point, its stack is the mainline stack. Any calls from this function (for example, to create tasks and start the scheduler) use the mainline stack area. Once the first task is loaded, however, the mainline stack is never modified again and future code is executed within the stack of the currently executing task.[1] The area between the mainline stack and the heap called the "interrupt handler temporary area" is a product of a design decision from the original MicroBlaze FreeRTOS port. By reserving some memory here for use by the interrupt handler rather than relying on tasks' stacks, it was intended that tasks could be made to use less memory. For example, rather than allocating an extra 250 bytes to each task's stack for use by the interrupt handler, reserving one 250 byte area of memory would be sufficient for all tasks, as the interrupt handler would only ever execute in the context of one task at a time. This has, in principal, been preserved in the multicore modifications, although to ensure thread safety this temporary area is now divided into as many sub-areas as there are processor cores. The interrupt handler then only uses the temporary area corresponding to the core currently executing it.

### 4.2.3 Multicore Awareness

Section 4.2.1 on page 45 explains how multiple cores progress from initial system startup to concurrent execution of FreeRTOS. This section deals with the actual modifications to FreeRTOS itself and how the software has been designed to support multiple cores. Note that within the greater goal of multicore support, a guiding principle has been to preserve the way in which single-core FreeRTOS does things. Put another way, great effort has been made to ensure that multicore FreeRTOS behaves in a manner coherent with that of the original, and as such that the multicore semantics should follow logically from the single-core ones. Simplicity has been a key aspect of achieving this.

[1]Note that for slave cores, the mainline stack is located in BRAM and is not illustrated.

#### 4.2.3.1 Task Control Block

It is from this simplicity that is derived the cornerstone of the modifications: a change in the "current TCB" multiplicity. A TCB, or task control block, is a data structure containing information about a task used by the scheduler to identify it in queues (when waiting to be scheduled, for example) and store and retrieve pertinent details about it, such as its priority. In single-core FreeRTOS, a pointer called `pxCurrentTCB` exists to identify the location of the TCB for the task currently being executed by the processor. This is extremely useful: when the scheduler comes to switch tasks, it can simply refer to this pointer to know which task to swap out, and then once a new task is swapped in, merely assign the address of the new task's TCB to the pointer before restoring its context. Whenever something about the currently executing task needs to be queried or updated, FreeRTOS merely needs to refer to the TCB via this pointer, and can be sure that it will always point to the correct task (if it points to a different task, then it means that a different task is running).

The reason that this works so elegantly in single-core FreeRTOS is that `pxCurrentTCB` serves as a representation of what the processor is doing that is always guaranteed to be accurate when accessed by the kernel. It is also inherently thread safe and access does not require any synchronisation: as it will always be changed by the context switching code before a new task is scheduled, a thread of execution can never accidentally refer to a TCB that is not owned by the currently executing task.

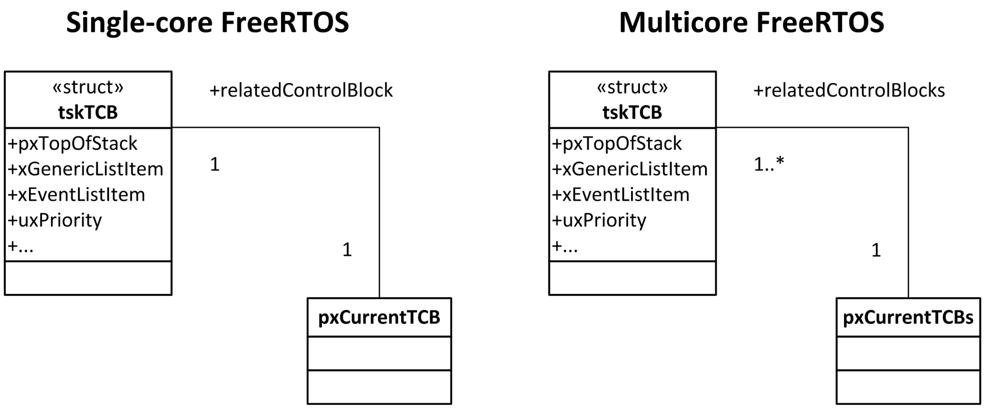

Figure 4.3: TCB pointer multiplicities

Extending this for use with multiple cores is straightforward. By turning `pxCurrentTCB` from a pointer to a TCB into an array of pointers to TCBs, all the benefits of the single–core

TCB pointer can be preserved. The size of the array is equal to the number of processor cores, the 0–based index of each core serving as a key to the TCB of the task it is executing. It remains thread safe because each core only ever accesses its own TCB pointer, safe–guarding concurrent access issues across cores while retaining the original semantics ensuring thread safety across tasks on the same core.

This design does make an assumption about the hardware. In order for the relevant code to access the element of the TCB array corresponding to the current core, there must be some way of uniquely identifying the core. Because all cores execute the same code, such identification must be made available from a source which is addressable in the same way across all cores, but which yields different data based on the core on which execution occurs. There are two potential options: firstly, using the private BRAM which resides in the same address space for all cores but obviously refers to different memory depending on the core from which the access is performed; or secondly, using special–purpose processor registers in each core containing values specified in the hardware design that are readable at runtime.

Hung *et al.* opt for an approach similar to the former, citing non-invasiveness and simplicity.[25, p. 242] They implement a small block of ROM (read-only memory) for each core, pre-populating them with processor core IDs.[25, p. 242] There are two problems with following this method. Firstly, initialising a part of the BRAM with a core identifier requires either a modification of the compiled binaries before they are loaded onto the FPGA, or the execution of a hard–coded BRAM memory assignment before FreeRTOS is executed. Both of these things are rather onerous, placing the burden of processor identification on arbitrary software behaviour. Secondly, the numerous points at which FreeRTOS needs to determine the current core's ID (many of which occur in the interrupt handler) could mean that adding the overhead of a memory access each time would have an undesirable effect on performance.

By contrast, using a special–purpose register avoids the possibility of memory latency and places the responsibility for processor identification, in many ways more instinctively, on the hardware. This also allows more flexibility in extending the use of the modifications to systems with many cores, of which only a subset might need to be used with FreeRTOS. For instance, an 8–core processor might be tasked with running FreeRTOS only on cores 6 and 7. The actual configuration could be made completely transparent to the software by simply assigning the appropriate values to the special–purpose registers on the two cores on which execution is to occur. These values can then be used directly as indices to elements in the `pxCurrentTCBs` array.

Listing 4.1: Using processor core IDs

```c
// Get the ID of the current core
unsigned char coreId = portGetCurrentCPU();

// Get a pointer to the TCB of the current task on this core
tskTCB * pxCurrentTask = pxCurrentTCBs[coreId];

// Do something with the TCB...
pxCurrentTask->uxPriority = pxCurrentTask->uxPriority + 1;
```

#### 4.2.3.2 Core Affinity

Processor or core affinity is the ability to bind a task to a specific core so that it is only ever scheduled on that core. In many situations this is a decidedly bad idea, as the Windows documentation points out:

> Setting an affinity [...] for a thread can result in threads receiving less processor time, as the system is restricted from running the threads on certain processors. In most cases, it is better to let the system select an available processor.[26]

The same is true of multicore FreeRTOS — a task given a core affinity will certainly not get as much processor time as physically possible in most circumstances. However, due to the fact that in a multicore environment idle tasks are needed for each core, there has to be some mechanism to ensure that they only get scheduled on the correct ones. Certain applications may also benefit from being able to tie a task to a core. For example, in the hardware design only the first core is connected to the primary serial port. Thus, to send data through this port, a task must be running on the first core. It may also be, for example, that the performance profile of an application is such that tasks can be carefully given core affinity to optimise performance.

This feature raises important design implications: tasks must have an awareness of their affinity for the duration of their life; as well as expressing affinity, there must be a way to express a *lack* of affinity; and the scheduler must reflect a task's affinity in an appropriate way. While the latter is discussed in detail in the following section (section 4.2.3.3), the former two issues are closely related with the indexing of processor cores in the `pxCurrentTCBs` array. With each core identified by an integer, this can be used to express task affinity as well. A lack of affinity can simply be expressed by a constant greater than the highest index. Because the TCB array is 0–based (the first element is indexed by 0, the second by 1 and so on), an obvious choice for this constant is the number of cores in the system.

By amending the TCB data structure to include a member called `uxCPUAffinity`, each task can be associated with a core ID (or with the "no affinity" constant) for use in scheduling. The process of using FreeRTOS from an application's perspective does not need to change much at all: when creating a task, the application must either specify a core index to assign the created task an affinity, or specify the `portNO_SPECIFIC_PROCESSOR` constant defined by the portable layer.

#### 4.2.3.3 Scheduling

The scheduler is the FreeRTOS component that must actually act upon the affinity values by selecting the appropriate task for execution on a particular processor core. This can be viewed as a constraint on the existing (single–core) scheduler's operation. The affinity–aware scheduler behaves as follows:

1. Get the current core ID.
2. Enter a critical section (because global task lists will be manipulated).
3. Retrieve the next task in the highest priority non–empty ready list which is not executing on any core. If there is no task matching these criteria in the highest priority list, repeat for the list with the next lowest priority.
4. **If no affinity is set for the retrieved task in the ready list:** select the task for scheduling on the current core.
5. **Otherwise, and if the affinity of the retrieved task is set to the current core ID:** select the task for scheduling on the current core.
6. If no new task was selected for scheduling, select the previously executing task.
7. Exit the critical section.

Thus, a task is only scheduled on a core when it is (a) not executing on any other processor; and (b) either has no core affinity or has a core affinity indicating that it should execute on the current core; and (c) is the highest priority task satisfying conditions (a) and (b). If no task can be found, then the task previously scheduled on the core is resumed until the next yield. This ensures that the highest priority task ready for scheduling is always the one selected for execution.

### 4.2.4 Synchronisation

As discussed in section 2.6 on page 19, software–based synchronisation solutions provide a viable alternative to relying on hardware components for the implementation of mutual exclusion. Apart from allowing the hardware design to remain as simple as possible, implementing shared resource protection in software allows for the possibility of providing thread–safety in an almost completely platform–agnostic manner. Although this comes with an inevitable overhead, it must be remembered that the goal of these modifications is not absolute performance, but rather predictable behaviour. In any case, this option would not preclude the addition of alternative platform–specific synchronisation in the portable layer if desired.

As such, the synchronisation features are implemented in software using Alagarsamy's improved version of Peterson's Algorithm. This has been discussed in detail in section 2.6.3 on page 20, and so the specifics of the algorithm theory and its improvements on Peterson's original will not be repeated here. There are still, however, practical considerations that must be addressed.

As one would expect, the algorithm presented by Alagarsamy focuses on the semantics of the synchronisation and does not address the practical aspect of using a single instance of the algorithm for synchronisation on a range of named objects. Recall that synchronisation always has a "target", exclusive access to which is acquired when its critical section is entered. In single–core FreeRTOS, this target is simply the processor. In a direct implementation of Alagarsamy's algorithm, the target is implicitly represented by the shared global variables the critical section entry and exit code use. These global variables are fixed.

However, it is important to allow applications to easily specify a custom synchronisation target, as it would be very inefficient for one system–wide critical section to be used to protect all resources. For instance, consider two tasks that operate on different locations in shared memory. One task wanting to gain exclusive access to its memory would be forced to prevent another task from locking its completely separate part of memory, all because the synchronisation target would be implied to be the algorithm's global variables rather than something more specific provided by the tasks. It must be possible for multiple critical sections to be active simultaneously as long as they refer to different targets.

Two methods of providing this functionality have been considered. The most simple but least flexible for application developers is to simply use an integer as a target, while the more complex and more flexible option involves allocating memory dynamically and using the address of the memory as the target. In the first solution, on entry to a critical section a task must provide an integer parameter in a predefined range to uniquely identify a synchronisation target. Other tasks will only be prevented from entering critical sections if they attempt to use the same integer parameter in their entry call. It is possible to modify Alagarsamy's algorithm to do this quite easily: by placing an upper–bound restriction on the number of named mutexes used by the system (defined in the portable layer), the `Q` and `turn` arrays can be given an extra dimension indexed by the name of the mutex. Note that whether the actual target is shared memory, an I/O peripheral or something else is application–specific and not relevant to FreeRTOS: the tasks themselves are responsible for enforcing the association between the integer and the resource it represents.

Listing 4.2: Alagarsamy's turn array with named mutex support

```
// Original algorithm turn array declaration (task IDs start at 1)
portBASE_TYPE mutexTurns[portMAX_TASKS + 1];

// A named mutex version of the array declaration
portBASE_TYPE mutexTurns[portMAX_TASKS + 1][portMAX_MUTEXES];
```

A benefit of this approach is that it avoids the need for dynamic memory allocation, which can be slow and indeterministic. It also allows the memory usage of the synchronisation code to be clearly limited to a predefined size at design–time by the modification of the `portMAX_TASKS` and `portMAX_MUTEXES` portable layer macros. At the same time, this pre–allocation means that memory usage may be higher than if the space for the synchronisation variables was dynamically allocated. For example, using a long integer as the `portBASE_TYPE`, the turn array alone would require 440 bytes of memory for the whole life of the system in a configuration supporting up to 10 tasks and 10 mutexes, regardless of how many tasks would actually use the named mutexes and how regularly they would do so. Nevertheless, in the interests of preserving the software's real-time properties and keeping the modifications as simple as possible, the first method is used to support named mutexes.

The mutex API is divided in two: the first part is for synchronisation by tasks using the `vTaskAcquireNamedMutex` and `vTaskReleaseNamedMutex` functions, while the second part is for synchronisation internally by the kernel using the `vCPUAcquireMutex` and `vCPUReleaseMutex` functions. There are several differences, the most important of which is the component on whose behalf the mutex is acquired. When tasks acquire mutexes, they do so on their own behalf — the mutex will only be released when the same task releases it (tasks are identified by a unique ID assigned on their creation and stored in their TCB, a handle to which is passed to the mutex acquisition and release functions).

However, when the kernel acquires a mutex, it does so on the behalf of the current processor core. This is to allow mutexes to be acquired outside of the context of a task, for example when the initialisation code calls the FreeRTOS API before the scheduler has been started. Unfortunately this means that while threads of execution running on other cores will be unable to acquire the mutex, other threads that become scheduled on the same core (as a result of a context switch, for example) *will* have access to the critical section. Much of the kernel code that makes use of this executes while interrupts are disabled, meaning that there is no possibility of such a context switch occurring. However for some code, such as the memory allocation and deallocation functions, interrupts must be explicitly disabled (if enabled to begin with) after entering the critical section in order to create a "full" lock and guard against other threads running on the same core.

There are also other design differences between the two synchronisation implementations:

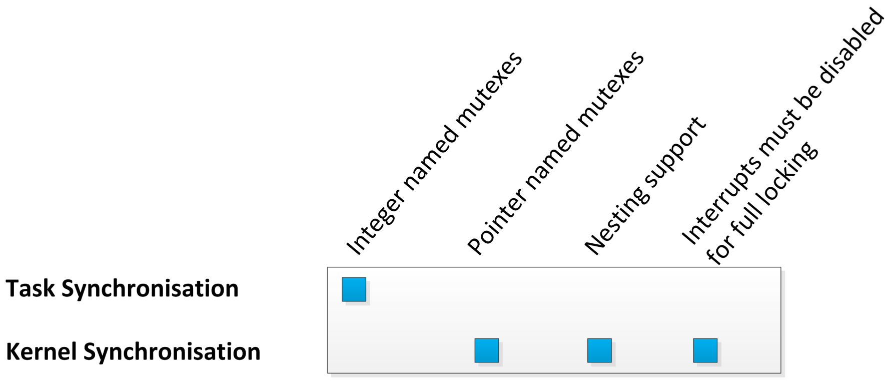

Figure 4.4: Synchronisation features

1. While the task synchronisation relies on an integer to identify mutexes to simplify work

for application developers, the kernel synchronisation uses a pointer to global variables for use by the synchronisation algorithm. This is because the kernel synchronisation does not need to be as flexible, with all mutexes predefined at design–time. Identifying these using pointers simplifies the synchronisation code and improves its clarity.

2. To preserve the semantics of single–core FreeRTOS, the kernel synchronisation explicitly supports critical section nesting. This allows the same critical section to be entered multiple times by the same thread of execution without first being exited as long as, eventually, each entry call is matched by an exit call. For the sake of simplicity, nesting is not supported in task synchronisation.

# Chapter 5

# Implementation

## 5.1 Hardware

The Xilinx stock dual–core MicroBlaze configuration serves as the basis of the implemented hardware design. This includes two MicroBlaze cores running at a clock speed of 125 MHz with instruction and data caching disabled. Disabling the caching features of the processor simplifies the implementation by eliminating the need for cache management in the software modifications. Additionally, several peripherals are included:

- One 16 KB private BRAM peripheral for each core. This is used to store core–specific code, such as the slave core ignition.
- One serial port I/O peripheral for each core. This potentially allows simultaneous serial communication by software running on separate cores. Due to the lack of support in XPS for a shared serial peripheral, if communication over only one port is required then this must be delegated to software running on the core connected to the desired port.
- One shared 256 MB DDR RAM peripheral. This is where all operating system and application code, as well as working memory, is stored. As it is shared by both cores, it also serves as the medium for cross–task communication.

In addition, the special–purpose read–only basic PVR (Processor Version Register) is enabled for both cores. This is a register that can be assigned a constant value at design–time, and interrogated by software at runtime using the dedicated `mfs` MicroBlaze instruction. For the first core, the data in the USER1 section of the PVR register is set to `0x00`; for the second core, it is set to `0x01`. These values serve as the IDs for use by FreeRTOS in identifying the core on which it is executing. The exact way in which these values are used by the software is explained in section 5.2.2 on page 60 below.

## 5.2 Software

### 5.2.1 Core Ignition

The purpose of the core ignition used by the slave core is to both configure the peripherals specific to the core, such as the interrupt controller and timer, and to wait until the master core has started the FreeRTOS scheduler before starting to execute tasks. In order to configure itself for interrupts, a slave core must be able to determine the location in shared memory of the "master" interrupt handler, the component of FreeRTOS that handles interrupts for all cores, as well as the location of the code in FreeRTOS that starts a core's first task. Because the Xilinx tools require that separate software projects are loaded onto the FPGA when each separate core is started and the main FreeRTOS project is the one loaded when the master core starts, this information is not available to the core ignition at design–time.

Instead, the core ignition must rely on the master core to communicate these addresses at runtime. It does this by using co-ordinated access to a reserved area of shared memory called the "ignition communication block". This reserved area is located in the low 2 KB of shared memory. The data structure used for communication is defined by a struct in the code for both cores:

Listing 5.1: Ignition communication block struct

```
typedef struct {

  volatile long interruptBridgeSwap[portNUM_PROCESSORS];
  volatile void * deviceHandlers[portNUM_PROCESSORS];
  volatile long initMagic;
  volatile void * interruptHandlerPtr;
  volatile void * startFirstTaskPtr;

} cpuComBlock;
```

The fourth and fifth members define the location in memory of the master interrupt handler and task starting code, respectively. The second member allows slave cores to publish the memory address of their specific low–level device interrupt handlers for execution by the scheduler. The third member is used for communication between the master and slave core. When the slave core starts (and it must start before the master core), it assigns the “slave magic code” to the `initMagic` member. It then waits until the master core changes the `initMagic` member to the value of the “master magic code”. Once it does this it means that the master core has finished performing its hardware initialisation, has started the FreeRTOS scheduler and has written the relevant code addresses to the other members in the communication block. At this point, the slave core will begin executing the code at the location specified by the `startFirstTaskPtr` pointer and will start running a task.

When the master core is notified of an interrupt, it jumps immediately to the FreeRTOS master interrupt handler located in shared memory. This behaviour is enforced by the compiler automatically, as the FreeRTOS interrupt handler is part of the software project loaded onto the FPGA when the master core starts. It is therefore regarded by the compiler as overriding the default interrupt handler. However, because the compiler cannot “see” the FreeRTOS code when compiling the software project for the slave core, the master interrupt handler cannot automatically override the slave’s default handler. Instead, a separate interrupt handler “stub” located in BRAM is used to initially handle slave interrupts. Its only purpose is to direct the processor to the address of the FreeRTOS master interrupt handler as specified by the master core in the `interruptHandlerPtr` member of the ignition communication block.

Unfortunately, as well as solving a problem this also creates one. The process of loading the address of the master interrupt handler from memory and branching to it cannot be done without using a general–purpose CPU register. Because this stub handler is executed before a task’s context is saved, allowing this to happen would cause corruption of the task context. To solve this problem, the contents of the processor register used in the stub handler (`r18`) is first saved to shared memory in the `interruptBridgeSwap` member of the ignition communication block before jumping to the master handler.

Listing 5.2: Slave core interrupt handler stub

```
_interrupt_handler:

  // First save r18 (the register to use for temporary data)
  swi   r18,    r0,   BRIDGE_SWAP_ADDR

  // Load the address of the master interrupt handler
  lwi   r18,    r0,   masterInterruptHandler

  // Branch to the master interrupt handler
  bra   r18

.end _interrupt_handler
```

The master handler is then responsible for restoring the register with this saved value before continuing to save the task context. Note that the master handler will only do this for the slave core — because the master core jumps directly to the master interrupt handler, it does not need to use a stub handler and therefore does not use `interruptBridgeSwap`.

Listing 5.3: Master interrupt handler bridge swap use

```
 1    swi    r3,    r1,   portTASK_CONTEXT_OFFSET(3)
 2
 3    mfs    r3,    rpvr0
 4    andi   r3,    r3,   0xFF
 5
 6    beqi   r3,    _no_bridge_swap
 7    muli   r3,    r3,   4
 8    lwi    r18,   r3,   BRIDGE_SWAP_ADDR
 9
10  _no_bridge_swap:
11
12  /* Context saving code goes here */
```

In order to make a register available for use in swapping data, the master interrupt handler first saves the `r3` register to memory (line 1). It then loads the core ID into `r3` (lines 3 and 4). If the ID is equal to 0 (i.e. if the current core is the master core), it jumps to the `_no_bridge_swap` label (lines 6 and 10). Otherwise, if the current core is the slave core, `r3` is assigned the value of the core ID multiplied by 4 — the size in bytes of a pointer (line 7). At this point, `r3` denotes the offset from `interruptBridgeSwap[0]` constituting the location in memory at which `r18` was temporarily stored by the slave stub handler. This value is then restored by loading the value at the memory address denoted by the BRIDGE_SWAP_ADDR constant (equivalent to the address of interruptBridgeSwap[0]) plus the offset into r18 (line 8).

### 5.2.2 Scheduler

Understanding the single–core scheduler in detail is very useful in appreciating how it has been adapted for multicore. Indeed, the scheduling algorithm in single–core FreeRTOS is beautifully simple. For each possible task priority there is a list containing pointers to all TCBs of tasks that are both assigned that priority and schedulable (including, if applicable, the currently executing task). If a list is empty, it means that there are no schedulable tasks assigned a priorty represented by that list. As tasks are created and assigned priorities, FreeRTOS keeps track of the highest priority list that can contain tasks ready for execution. This is referred to as the "top ready priority" list.

When a yield occurs and the scheduler is called to determine which task to swap in, it simply cycles through the ready lists starting at the top ready priority list and working down towards the idle priority list until it encounters a list that is non–empty. When it does so, it selects the task at the head of the list for scheduling. In this way, higher priority tasks are always executed over lower priority ones: only if a list is empty will the scheduler consider executing tasks in a lower priority list. By always ensuring that idle tasks run at the lowest priority (known as the "idle priority" and represented by the tskIDLE_PRIORITY macro), the scheduler automatically selects the idle task for execution if it is unable to find application tasks.

Listing 5.4: The heart of the single–core scheduler

```
while(listLIST_IS_EMPTY(&( pxReadyTasksLists[uxTopReadyPriority])))
{
  configASSERT( uxTopReadyPriority );
  --uxTopReadyPriority;
}


/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so
the tasks of the same priority get an equal share of the
processor time. */
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,
    &( pxReadyTasksLists[ uxTopReadyPriority ] ) );
```

The while loop in the listing above iterates through the ready lists, decrementing the global `uxTopReadyPriority` variable until it finds a list that is non–empty. The `listGET_OWNER_OF_NEXT_ENTRY` macro then fetches a pointer to the TCB of the task at the head of the non–empty list found and assigns it to `pxCurrentTCB`. Execution will then eventually return to the low–level context–switching code which will swap in the task whose TCB is pointed to by this variable.

The multicore environment imposes two additional constraints on the scheduler. For a task to be scheduled on a core:

1. The task must not be executing on any other core; and

2. The task must either have no core affinity or have an affinity for the core in question.

While significant modifications have been required to accommodate these constraints, the solution has been implemented with the intention of producing semantics which follow logically from those found in single–core FreeRTOS. Simply put: the scheduler will always select the available task with the highest priority for scheduling. Using the task ready lists to do this in a multicore environment is not as simple as in a single–core environment.

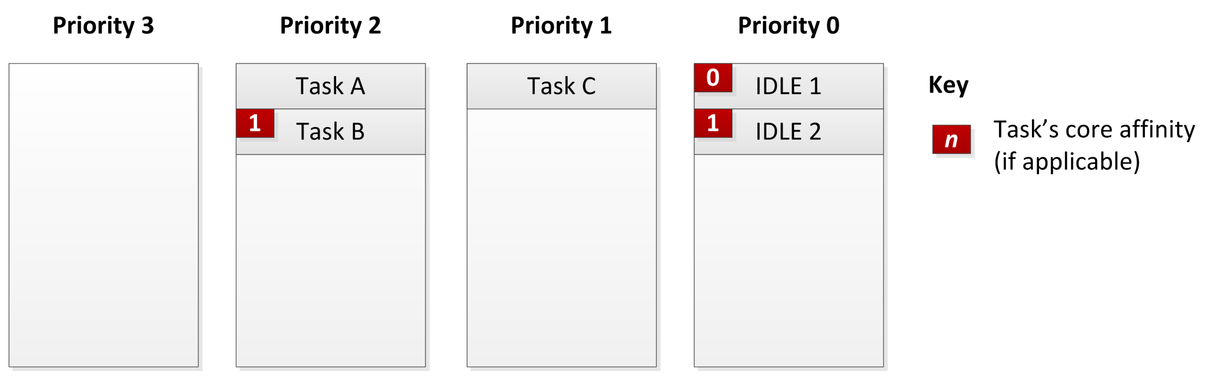

Figure 5.1: Multicore scheduler ready lists example

The way this is implemented in the multicore scheduler is best demonstrated using an example. Figure 5.1 illustrates a hypothetical set of ready lists in a system configured to use four different priorities, with priority 0 assigned as the idle priority. The highest priority non–empty list displayed is of priority 2. Assume that a yield occurs on core 0 and the scheduler is invoked to determine which task to next schedule on the core. The scheduler first considers the Priority

2 list, but finds that Task A is already executing on core 1. It then considers Task B, but cannot select it for scheduling due to the fact that it has an affinity for core 1. At this point the scheduler must consider tasks of lower priority, otherwise it will be unable to give core 0 any more tasks to execute. As such, it moves down one priority level to the Priority 1 list — it retrieves Task C from this list. This task has no core affinity and is not being executed by any other core, allowing the scheduler to select it for execution. If Task C for some reason could not be selected for execution, the scheduler would move down to the Priority 0 list at which it would be guaranteed to select the idle task with affinity for core 0.

Once a task is selected for execution, the scheduler passes control to the portable layer. The element for the current core in the `pxCurrentTCBs` array is then used to swap the selected task in.

Listing 5.5: Getting the TCB of the task to swap in

```
 1  vPortRestoreContext:
 2
 3    mfs    r18,   rpvr0
 4    andi   r18,   r18,   0xFF
 5    muli   r18,   r18,   4
 6
 7    lwi    r3,    r18,   pxCurrentTCBs
 8    lwi    r1,    r3,    0
 9
10  /* Restore the task's context */
```

The core ID is retrieved from the PVR (lines 3 and 4) and is then multiplied by the size of a pointer (line 5). This is used to retrieve the element of `pxCurrentTCBs` corresponding to the current core (line 7), after which the first word in the TCB (the stack pointer) is read into `r1`. The part of the stack containing the contents of the processor core registers are then loaded into each register and the processor core is made to jump to the location in memory as specified by the program counter saved when the task was swapped out.

### 5.2.3 Synchronisation

There were two key practical aspects to implementing Alagarsamy's mutual exclusion algorithm:

1. Ensuring that the effects of out–of–order memory accesses were neutralised correctly.

2. Making the implementation easy to use by application developers.

Memory access re–ordering presents a particularly significant problem for mutual exclusion algorithms, as they rely on memory reads and writes occurring in the exact order specified. As explained in section 2.6.3 on page 20, this is because memory accesses are used to communicate the position of a core's program counter to tasks on different cores, allowing the mutual exclusion algorithm to determine which task to allow into a critical section. Memory barriers are used to force the processor to execute the memory accesses in the order defined in the code. Whenever one is required, the `vPortMemoryBarrier` function defined in the portable layer can be called to create one. It is possible to demonstrate the importance of the memory barriers by commenting out the code within this function in the multicore portable layer and running the sync test demo task (explained in section 6.1.4 on page 78). The result is a failure of the synchronisation code, causing deadlock.

Because the details of how MicroBlaze re–orders instructions are not well documented, the strategy for using memory barriers in the algorithm has been to have them to protect all shared memory accesses. This consists of creating a memory barrier after each memory write and before each memory read of a global variable.

Note that memory operations can also be re–ordered by the compiler at compile–time. This is prevented by ensuring that all global variables are marked `volatile`, indicating to the compiler that they may change at any time (for example by a memory store operation in a separate thread of execution). This prevents the compiler from trying to optimise the code by making assumptions about the values of the variables at particular points in the program.

Making the API easy to use is also very important. As well as generally limiting the effort required to use the synchronisation features and thus making development more efficient, it helps to encourage the use of mutual exclusion in the first place. An onerous or difficult–to–use API implementation could make developers more likely to overlook synchronisation issues and avoid using critical sections (whether consciously or not). As such, mutex acquisition and release is performed with just two simple function calls:

Listing 5.6: Using the multicore synchronisation API

```

// Mutex identifier declaration
static const unsigned portBASE_TYPE EXAMPLE_MUTEX = 1;

```

```
// The task
static portTASK_FUNCTION( vExampleTask, pvParameters ) {

  for (;;) {

    vTaskAcquireNamedMutex(
      ((systemTaskParameters *)(pvParameters))->taskHandle,
      EXAMPLE_MUTEX);

    // Critical section code goes here

    vTaskReleaseNamedMutex(
      ((systemTaskParameters *)(pvParameters))->taskHandle,
      EXAMPLE_MUTEX);

  }

}
```

The call to `vTaskAcquireNamedMutex` passes in the task's handle (a pointer to its TCB) and the "value" of the mutex (the integer identifying the resource being locked). Once control returns from this function, the mutex has been acquired and all subsequent code is executed within the named mutex's critical section until `vTaskReleaseNamedMutex` is called and returns. Notice the use of the `pvParameters` argument to allow the task to refer to its handle: unlike in the original version of FreeRTOS in which task parameters are simply user–defined pointers, in multicore FreeRTOS they are always supplied by the system as a pointer to a `systemTaskParameters` struct that provides access to the task's handle. User–defined parameters are passed as void pointers in a member of this struct and can be accessed as follows:

Listing 5.7: Accessing user–defined parameters in multicore FreeRTOS

```
int * pIntParameter = (int *)(((systemTaskParameters *)(pvParameters))
  ->applicationParameters);
```

As explained in section 4.2.4 on page 54, the kernel–level synchronisation acquires mutexes on behalf of processor cores (as opposed to tasks) so that it can be used when no tasks are running. If used when interrupts are enabled, however, this can result in other threads of execution running on the same core as the owner of the mutex entering the critical section at the same time. This is because the kernel–level synchronisation algorithm cannot differentiate between them — it identifies a mutex holder by the core on which it is running. As a result, interrupts have to be actively disabled when kernel code that may execute when they are enabled creates a critical section. The following code, taken from the modified memory manager, demonstrates this.

Listing 5.8: Kernel–level synchronisation when interrupts might be enabled

```
portBASE_TYPE intsEnabled = xPortAreInterruptsEnabled();
portBASE_TYPE currentCPU = portGetCurrentCPU();

taskENTER_CRITICAL(currentCPU, pvGetMemoryMutexObject());
{
  if (intsEnabled == pdTRUE) {
    portDISABLE_INTERRUPTS();
  }
  pvReturn = malloc( xWantedSize );
  if (intsEnabled == pdTRUE) {
    portENABLE_INTERRUPTS();
  }
}
taskEXIT_CRITICAL(currentCPU, pvGetMemoryMutexObject());
```

A new portable layer function, `xPortAreInterruptsEnabled`, is used to determine whether or not interrupts are currently enabled. This is important, because if they are not then the code must avoid enabling them out of turn. The `taskENTER_CRITICAL` macro acquires a kernel–level mutex for the given memory mutex object (returned by `pvGetMemoryMutexObject`) and then, if interrupts are enabled, disables them. It then performs the memory allocation before enabling interrupts if they were enabled to begin with, after which it exits the critical section.

### 5.2.4 Looking Ahead to *n* Cores

Although the requirements (in particular REQ-PF-01 on page 35) limit the scope of the project to modifying FreeRTOS for use with a dual–core processor, this limitation was something of a precautionary measure imposed by severe time restrictions. It was deemed of most importance to have a working multicore version of FreeRTOS by the end of the project, even if it was restricted to processors with only two cores. There was a concern that the process of building and maintaining multiple hardware configurations with different numbers of cores, and then testing each one with the FreeRTOS modifications, would simply not be feasible within the time available.

Because of this, and as discussed in more detail in Chapter 6, the secondary requirement (REQ-SF-09 on page 38) of ensuring the modifications scale to $n$ cores has not strictly been met. However, every design and implementation decision has been made with a view to ensuring that no further changes are required to use multicore FreeRTOS on processors with more than two cores. This point cannot be emphasised enough. From the core identification to the multicore scheduler and modified portable layer, every single modification has been pervaded by the principal of ensuring compatibility with $n$ cores. However, due to a lack of testing on hardware with more than two cores, this compatibility is nonetheless theoretical, although it is worth noting that the author's confidence in this theory is very high.

## 5.3 Reproducing the Results

This section provides step–by–step instructions to enable the implementation to be reproduced by the reader. Although this project uses a Virtex-5 FPGA board, alternative instructions are provided where applicable to allow the hardware design to be created for use with a Spartan-3e board.

### 5.3.1 Required Apparatus

- Administrative access to a PC running Windows Vista or above with at least 2 spare USB ports, or 1 spare USB port and 1 spare RS–232 serial port. Note that these instructions have only been tested on Windows, although with minimal modification they should work with the Linux port of the Xilinx software tools.
- A Xilinx Virtex-5 or Spartan-3e FPGA board with a power supply, JTAG USB cable and serial cable (with USB converter as required).
- A USB serial converter software driver (if using a converter). This should be supplied by the manufacturer of the converter cable and is required to allow the system to transparently

read data from a USB connection as if it was sent over a serial port.

- Access to the Xilinx ISE Design Suite 13.1 software tools for Windows. These can either be already installed or available in an installation package or disc. Note that these instructions have only been tested with version 13.1 of the tools — although it should be possible to use other versions, deviation from the instructions will probably be required.
- The most recent version of PuTTY, available at http://www.chiark.greenend.org.uk/~sgtatham/putty/. This is for monitoring serial output from the FPGA; equivalent software capable of this (such as HyperTerminal) would also be adequate.
- An SVN client for downloading the multicore FreeRTOS source code.

### 5.3.2 Tool Configuration

1. If the Xilinx ISE Design Suite is not yet installed on your development system, install it now. Although you may be prompted to connect the FPGA board during installation, this is optional and can be done afterwards.
2. Once installation is complete, the tools need to be configured to run with administrative privileges. This is because when the installer adds the necessary tool paths to the "PATH" environment variable, it does so as the administrative user, meaning that these paths are only visible from this user account. To perform this configuration, find the shortcuts you will use to open the Xilinx Platform Studio and Xilinx Software Development Kit (in the Start menu, these are placed in the "Xilinx ISE Design Suite 13.1 EDK" folder by default). For each shortcut:
   (a) Right–click on the shortcut and select "Properties".
   (b) On the "Shortcut" tab, click "Advanced".
   (c) In the dialog window that opens, tick the "Run as administrator" box, then click "OK".
   (d) Click "OK" again to dismiss the "Properties" dialog.
3. If you are using a serial–to–USB converter and you have not yet installed the converter driver, do so now.

4. If you have not yet installed PuTTY, do so now.
5. Once installed, open PuTTY and select the "Serial" option in the "Session" section.
6. Refer to the serial–to–USB driver documentation to determine which COM port is used to send and receive data to the USB device, then type the name of this port into the "Serial line" box. Ensure the "Speed" box is set to 9600.
7. In the list on the left, select the "Terminal" section. Tick the "Implicit CR in every LF" box — this causes single linefeeds sent over the serial port to be interpretted by PuTTY as Windows line breaks (linefeed + carriage return) and display them as such. Return to the "Session" section by selecting it in the list on the left.
8. In the "Saved Sessions" box, enter a name for this connection (e.g. "FPGA serial connection"). Click the "Save" button to save this connection profile, and then exit PuTTY by clicking "Cancel".

### 5.3.3 Producing a Hardware Design

9. Open the Xilinx Platform Studio using a shortcut set to run the program as an administrator above.
10. In the dialog displayed on startup, select the "Base System Builder wizard" option and click "OK".
11. To specify a location at which to save the project, click "Browse" and choose an empty folder on a local disk. Click "Save" and then "OK".
12. Select the "PLB system" option and click "OK".
13. Select the "I would like to create a new design" option and click "Next".
14. Select the "I would like to create a system for the following development board" option.
15. **If using a Virtex–5 board:**
    (a) Under board vendor, select "Xilinx".
    (b) Under board name, select "Virtex 5 ML505 Evaluation Platform".

(c) Under board revision, select "1" and click "Next".

(d) Select the "Dual–Processor System" option and click "Next".

(e) Leave the default options, ensuring that the "Enable Floating Point Unit" options are unticked for both processors, then click "Next".

(f) The next pane displayed allows the hardware peripherals to be customised. In each processor peripheral list, remove all peripherals (by selecting them and clicking the "Remove" button to the left of the list), except: "RS232_Uart_1", "RS232_Uart_2", "dlmb_cntlr" and "dlmb_cntlr_1".

(g) In the list of available peripherals on the far left, select the "xps_timer" peripheral and add it to Processor 1's list by clicking the respective "Add" button. Once added, select the "xps_timer_0" item in the list and tick the "Use interrupt" box. This will ensure that an interrupt controller is included in the hardware design and configured for use with the timer. Repeat this step for processor 2.

(h) In the "Shared Peripherals" list, remove all listed peripherals.

(i) In the list of available peripherals on the far left, select the "DDR2_SDRAM" peripheral and add it to the "Shared Peripherals" list by clicking the respective "Add" button. Then click "Next".

(j) Ensure that all caching options are unticked and click "Next", then click "Finish". A blueprint for the hardware design will be created and loaded into the XPS editor.

(k) On the far left, select the "Project" tab (next to the "IP Catalog" tab) and right–click on the "Device: xc5vlx50tff1136-1" item.

(l) Select the "Project Options" item.

(m) In the dialog that opens, set the "Architecture" to "virtex5", the "Device Size" to "xc5vlx110t", the "Package" to "ff1136" and the "Speed Grade" to "-1". Then click "OK".

(n) Open the "system.ucf" file by double–clicking the "UCF File: data/system.ucf" item in the "Project Files" list under the "Project" tab on the far left.

(o) Find the the lines displayed in Appendix A on page 93 in the opened file, and replace them with the lines displayed in Appendix B on page 96.[27]

16. **If using a Spartan–3e board:**

(a) Under board vendor, select “Xilinx”.

(b) Under board name, select “Spartan-3E Starter Board”.

(c) Under board revision, select “D” and click “Next”.

(d) Select the “Dual–Processor System” option and click “Next”. After doing this, note the warning displayed at the bottom of the Base System Builder window about the selected board having limited resources.

(e) Leave the default options, ensuring that the “Enable Floating Point Unit” options are unticked for both processors, then click “Next”.

(f) The next pane displayed allows the hardware peripherals to be customised. In each processor peripheral list, remove all peripherals (by selecting them and clicking the “Remove” button to the left of the list), except: “RS232_DTE”, “RS232_DCE”, “dlmb_cntlr” and “dlmb_cntlr_1”.

(g) In the list of available peripherals on the far left, select the “xps_timer” peripheral and add it to Processor 1’s list by clicking the respective “Add” button. Once added, select the “xps_timer_0” item in the list and tick the “Use interrupt” box. This will ensure that an interrupt controller is included in the hardware design and configured for use with the timer. Repeat this step for processor 2.

(h) In the “Shared Peripherals” list, remove all listed peripherals except the “DDR_SDRAM” peripheral. Then click “Next”.

(i) Ensure that all caching options are unticked and click “Next”, then click “Finish”. A blueprint for the hardware design will be created and loaded into the XPS editor.

17. On the right–hand pane in XPS, select the “System Assembly View” tab, then select the “Bus Interfaces” tab.

18. Right–click the “microblaze_0” list item and select “Configure IP”.

19. In the dialog that appears, click the “Advanced” button and select the “PVR” tab.

20. Set the “Specifies Processor Version Register” option to “Basic”, then set the “Specify USER1 Bits in Processor Version Register” to `0x00`. Click “OK”.

21. Repeat steps 17 – 19 for the “microblaze_1” item in the “System Assembly View”, instead setting the “Specify USER1 Bits in the Processor Version Register” option to `0x01`.

22. The hardware blueprint is now fully customised and can be compiled to a binary representation ready for programming onto an FPGA (called a "bitstream"). To do this, select the "Project" menu then select "Export Hardware Design to SDK". Click "Export Only". This process may take several minutes to complete.

### 5.3.4 Building the Source Code

23. Using an SVN client, check out the following URL to an empty local directory: `https://freertosxcore.svn.sourceforge.net/svnroot/freertosxcore`. The SVN client will download the source code of the modified version of FreeRTOS to your system.

24. Open the Xilinx Software Development Kit using a shortcut set to run the program as an administrator.

25. In the "File" menu, select "New" then "Xilinx C Project".

26. When prompted to specify a hardware platform specification, click "Specify". Type a name to identify the hardware platform created, then click browse to select the XML file specifying the hardware configuration created earlier. This should reside in the following path: "*hardware path*\ SDK\ SDK_Export\ hw\ system.xml", where "*hardware path*" is the path to the directory in which the hardware design was saved by XPS. Then click "Finish".

27. In the "Project Name" field enter "freertos_xcore".

28. In the "Processor" field, select "microblaze_0".

29. In the list under "Select Project Template", select the "Empty Application" item and click "Next". Then click "Finish".

30. Repeat steps 25 – 29, specifying "core_ignition" for the "Project Name" field and selecting "microblaze_1" for the "Processor" field.

31. In the Xilinx SDK, right–click the "core_ignition" folder in the "Project Explorer" pane and select "Import".

32. Under the "General" folder in the dialog that appears, double–click "File System".

33. Click the “Browse” button next to the “From directory” field. Navigate to the directory to which you checked the SVN repository out earlier.

34. Select the “core_ignition” sub–directory and click “OK”, then click “Finish”.

35. Repeat steps 31 – 34 for the “freertos_xcore” folder in the “Project Explorer”, importing the “freertos_xcore” SVN sub–directory instead of “core_ignition”.

36. The source code is now ready to be built. Right–click the “core_ignition” project item in the “Project Explorer” and select “Build Project”. Once the build is complete, do the same for the “freertos_xcore” project item.

37. To avoid confusion it may be best to disable automatic building, an option that causes projects to be built whenever contained source files are saved. To do this, select each software project (“core_ignition” and “freertos_xcore”) and untick the “Build Automatically” item in the “Project” menu.

### 5.3.5 Running FreeRTOS Multicore

38. Ensure the FPGA board is connected to its power supply, then plug the power supply into a wall socket. Plug the serial cable into the FPGA’s serial port and, if using a USB converter, attach it to the opposite end of the cable, then plug the cable into the PC. Finally, connect the JTAG cable to the FPGA and plug it into a USB port on the PC.

39. Return to the Xilinx Platform Studio, ensuring that the hardware design created earlier is open. In the “Design Configuration” menu, select the “Download Bitstream” option. This will cause XPS to program the bitstream into the FPGA.

40. Open PuTTY and load the serial connection profile saved earlier. The PuTTY session window that opens will display the standard output of the software projects executed on the hardware.

41. Return to the Xilinx SDK. Open the “main.c” file in the “freertos_xcore” project (located in “DemoTasks\ MicroBlaze\ main.c”). Locate the line containing a call to `vStartSyncTestTasks`. Leaving this line uncommented runs the demo tasks showing the synchronisation features at work; to run the tasks that demonstrate concurrency, comment out this line and then uncomment the one below it (a call to `vStartConcTestTasks`).

42. The concurrency demonstration requires the execution time of the single–core and multicore versions to be compared. This requires the creation of a single–core hardware project to execute the single–core tasks included in the “single_core_conc_tasks” SVN sub–directory. The steps to do this are not included here.

43. Right–click on the “core_ignition” project in the “Project Explorer” and select “Run As” then “Run Configurations”.

44. Under the “Device Initialization” tab, select “Reset Entire System” for the “Reset Type” option. This will cause the FPGA to stop all processors when the second core is started and the “core_ignition” software is loaded into its BRAM. Click “Apply”.

45. In the list on the left, select “freertos_xcore.elf”.

46. Under the “Device Initialization” tab, select “Reset Processor Only” for the “Reset Type” option. Click “Apply”, then click “Close”.

47. Repeat steps 40 – 43 for the debug configurations by using the “Debug Configurations” item in the “Debug As” menu.

48. Right–click the “core_ignition” project item and select “Run As”, then select “Launch on Hardware”. Always ensure that the “core_ignition” project is executed first, as otherwise the cross–core initialisation will not complete correctly.

49. The SDK will display a warning that the bitstream was not downloaded to the FPGA from the SDK. Tick the box indicating that the bitstream was downloaded from another program and dismiss it.

50. Repeat step 48 for the “freertos_xcore” project. Switch to the PuTTY terminal to see the output from the FreeRTOS tasks.

# Chapter 6

# Evaluation

## 6.1 Testing

### 6.1.1 Strategy

As described in section 1.2 on page 5, formalised testing in the form of purpose–built test tasks in each development iteration has been a core element of the development model used in this project. Due to the relative complexity of the software being modified, a testing strategy that would reveal problems caused by modifications as soon as possible after implementation was absolutely essential.

As such, a suite of FreeRTOS test tasks was used to test the effects of modifications at the end of each iteration in the development cycle. Before this stage in each cycle, custom test tasks were developed to provide specific (and faster) feedback on the success of modifications made in the implementation stage. In practice, components of the end–of–cycle tests were often also run during the implementation stage to provide greater confidence in certain modifications. As well as generally increasing the effectiveness of the testing strategy, this helped contain problems in the cycle from which they were introduced.

The test tasks were designed to test the functionality of the modified system components (the scheduling and context–switching code) and the newly introduced features (software–based synchronisation and new elements of the task API). Synchronisation tests were created in “long run” and “short run” versions. Short run tests ran quickly (within minutes) but were not thorough; long run tests took a long time to complete (up to an hour for a single test) but provided results with a higher degree of confidence than short run tests. Long run tests were always used at the end of a development cycle, but apart from this, test use was not formally prescribed. The tests consisted of:

1. **2–task Synchronisation:** Test using two worker tasks assigned separate core affinities. The critical section created by each task contained code that repeatedly incremented a shared variable in what was a highly divisible operation. A third "check" task monitored the worker tasks for deadlock and evidence that the shared variable increment operations had not been performed atomically. Upon timer interrupt, the scheduler would always resume the interrupted task. The short run version consisted of 10,000 runs; the long run version consisted of 25,000 runs (i.e., this many acquisitions and releases of the shared mutex object per task).

2. **4–task Synchronisation:** Test using four worker tasks assigned separate core affinities, two on each core. Semantics are as above. Upon timer interrupt, the scheduler would switch between the worker tasks assigned on the each core. The short run version consisted of 10,000 runs; the long run version consisted of 25,000 runs.

3. **API:** Test used to verify the accuracy of specific API calls. API calls tested were `vTaskDelay`, `xTaskGetTickCount`, `xTaskCreate`, `pvPortMalloc` and `vPortFree`.

4. **Scheduling:** Test used to verify the performance of the scheduler. Four tests were used:

   (a) **Affinity Sharing:** Tested the scheduling behaviour when 2 tasks with core affinity were sharing a processor with 2 tasks without core affinity;

   (b) **Free For All:** Tested the scheduling behaviour when 4 tasks without core affinity were sharing a processor;

   (c) **Normal Prioritisation:** Tested the scheduling behaviour when 3 tasks shared the processor, one of which had a lower priority;

   (d) **Multicore Prioritisation:** Tested the scheduling behaviour when 3 tasks shared the processor, one of which had a core affinity and another of which had a lower (but non–idle) priority. On multicore FreeRTOS, the lower priority task in this situation was expected to get *some* processor time when one of the cores for which no task had any affinity found that the higher priority tasks were either executing on the other core or had an incompatible affinity.

None of the scheduling tests were automated and thus required manual observation for the results to be determined.

5. **Concurrency:** Test consisted of two tasks that performed the same function, one for use with the original FreeRTOS and the other for use with multicore FreeRTOS. They differed only in the ways necessary to make them compatible with the separate operating systems. They performed repeated memory accesses with the intention of creating a relatively long and therefore measurable execution time. The execution times of the two versions were then compared to observe the performance difference between FreeRTOS on single–core and FreeRTOS on dual–core. Interrupt frequency was the same on both systems. Three versions of the tests were used: SR or short run (1 million runs),[1] LR or long run (10 million runs) and VLR or very long run (100 million runs).

Table 6.1: Concurrency test results when measured using internal clock ticks

| Version | SR Time (ticks) | LR Time (ticks) | VLR Time (ticks) |
|---|---|---|---|
| Single–core | 817 | 7,797 | 79,084 |
| Multicore | 506 | 4,142 | 41,419 |

Execution time improved on multicore FreeRTOS by a factor of approximately 1.6 in the short run test. This was the lowest improvement observed. In the long run test, execution time on the multicore version improved by a factor of 1.88 over the single–core version. In the very long run test, execution time improved by a factor of just over 1.9. This suggests that the efficiencies provided by multicore FreeRTOS are related in some way to the elapsed execution time.

To control for possibly skewed tick durations, these tests were repeated but measured in seconds using an external timer (results displayed below).

These results validate those from the initial tests in which the timing, based on system ticks, was used to automatically generate results. The execution time improvements are indeed related to the elapsed execution time. The short run test shows an improvement factor of 1.6; the long run one of 1.85; and the very long run one of nearly 1.87. The exact relationship between the efficiencies provided by multicore FreeRTOS and the elapsed

[1] "Runs" in this context refers to the number of memory operations performed by the test.

Table 6.2: Concurrency test results when measured using an external timer

| Version | SR Time (sec) | LR Time (sec) | VLR Time (sec) |
|---|---|---|---|
| Single–core | 8 | 87 | 816 |
| Multicore | 5 | 48 | 437 |

execution time have not been investigated further, although these results can be seen to show that for highly parallel tasks, very significant execution time improvements are possible with multicore FreeRTOS.

### 6.1.2 Testing the Tests

In order to test the synchronisation tests, two counter–examples were created to demonstrate their suitability in detecting problems they were designed for. The first simply consisted of the same tasks without the calls to the synchronisation API, while the second consisted of the same tasks with calls to “dummy” synchronisation functions which simulated deadlock by causing the tasks to enter an infinite loop.

The API tests were relatively simple. The code tested was actually covered by other tests, but including dedicated tests made problems with these components obvious sooner than if they were left to manifest themselves in other tests. The production of these tests relied on the use of assertions to test pre–conditions and post–conditions to ensure the API acted as expected.

The scheduling tests relied on tasks with different priorities and core affinities “checking in” to notify the tester of their execution. Testing this mechanism involved first running all tasks in each test with affinities and priorities allowing them to execute with equal slices of processor time. Once satisfied that they “checked in” correctly, they were modified to test the specific scenarious detailed above.

### 6.1.3 Limitations

Unfortunately, there are inherent limitations, some of which were purposely imposed, on the tests used. The limited available time and high frequency of test runs required to ensure the modifications were correctly regression tested restricted how thorough each test could be.

In particular, synchronisation tests with much higher run counts are necessary to provide the requisite degree of confidence in the implementation of the mutual exclusion algorithm. Due to the lack of specific information about how MicroBlaze implements out–of–order execution, the use of memory barriers to mitigate memory access problems has probably been over–used, with liberal estimates as to the required positioning of memory barriers guiding the implementation. Trial and error was considered as a way to determine the MicroBlaze's behaviour, but due to the very long duration of high–confidence synchronisation tests (consisting of millions of runs) on the development hardware, this simply was not feasible. Extensive work still needs to be done to assess the properties of this part of the implementation.

Some optional elements of the FreeRTOS API have not been tested. It is important that before these features are used, they are correctly modified and tested for compatibility with the rest of the system. The relevant code is located in `tasks.c` and is marked with comments reading "NOT FULLY MULTICORE TESTED".

### 6.1.4 Demo Tasks

As per requirement REQ-PF-07, demo tasks have been created to exemplify the effectiveness of the FreeRTOS modifications. These are simplified versions of some of the test tasks described above. The demo tasks are split into two "applications": the first demonstrates and tests the synchronisation features (termed the "sync test"), while the second illustrates the existence of concurrency through execution time comparison (termed the "conc test").

The sync test application consists of three tasks:

1. **Core 0 Worker:** Created with an affinity for core 0, this task acquires a mutex, increments the value of a shared variable 1,000 times in what is a trivial but relatively time–consuming operation and then releases the mutex before repeating the process. Because the shared variable incrementation occurs within a critical section, it should be indivisible. By checking whether or not the shared variable is a multiple of 1,000 on entry to the critical section, the task can detect if synchronisation has failed and both worker tasks are executing within the critical section simultaneously. If this happens, it notifies the check task. The task only ever finishes once the mutex has been acquired and released 25,000 times. Every time the critical section is entered, the check task is notified that it is still alive. When the critical section is exited, the check task is notified of the number of runs it has executed so far; when finished, it notifies the check task of this as well.

2. **Core 1 Worker:** Performs the same function as the Core 0 Worker, except has an affinity for core 1. This task will run concurrently with the Core 0 Worker.

3. **Check Task:** Monitors the progress of the two worker tasks and is responsible for displaying program output over the serial port, and is thus given an affinity for core 0. Approximately every hundred runs it updates the user as to the status of the worker tasks. If either of the worker tasks fail to check in within 5,000 ticks of the previous check in, the task warns that deadlock may have occurred. If the task receives a notification from one of the worker tasks that there has been a synchronisation failure, a warning to this effect is passed on to the user. When both worker tasks are finished, this task outputs a summary of the results.

The conc test is more simple. It consists of three tasks as well, with both worker tasks assigned an affinity for different cores. They perform memory writes a pre–defined number of times and each notify the check task when they have finished. The check task then displays the execution time in ticks. An equivalent version of the conc test application has been created for the single–core version of FreeRTOS. By comparing the execution time on both versions, it is possible to observe the same tasks executing more quickly when being scheduled on two cores. Note that because these tasks rely on pre–emptive scheduling, the interrupt frequency (specified in the `configTICK_RATE_HZ` macro) must be the same in both the single–core and dual–core applications for the execution times to be comparable.

In fact, it is worth mentioning the general significance of the interrupt frequency to application performance. The interrupt frequency defines how often a tick interrupt will occur, and therefore how often a processor core pre–emptively switches tasks (in pre–emptive mode). There is an unavoidable overhead in this: the context–switching code must execute every time a task is switched, and so if the tick frequency is too high the processor cores will spend too great a proportion of their time performing context–switches as opposed to executing tasks.

## 6.2 Hardware

### 6.2.1 Expectations

The complexity of hardware design presented a potentially serious problem with producing an appropriate platform upon which to implement multicore FreeRTOS. The author's lack of experience with FPGAs and hardware design in general, combined with the very limited time available, meant that a ground–up implementation of the hardware was simply not feasible. This was anticipated very early on in the requirements analysis, and it was decided that to mitigate the problem the Xilinx toolset and stock hardware designs would be relied on as much as possible. It was expected that by using a design requiring minimal manual configuration, the Xilinx tools (which are extremely powerful) would make the process of rapidly creating a suitable hardware configuration possible.

### 6.2.2 Results

The use of the Xilinx tools and stock designs almost entirely removed the necessity for manual configuration of the hardware design. All hardware blueprints were generated automatically for the specific FPGA board being used, meaning that they compiled to bitstreams without any compatibility issues requiring manual intervention. The implemented hardware provided a fully operational dual–core processor system with the necessary peripherals to implement and test multicore FreeRTOS.

### 6.2.3 Evaluation

While the time spent on hardware design and implementation was significantly lower than that spent for the software, a significantly larger proportion of hardware development time was spent on implementation than design when compared to software development.

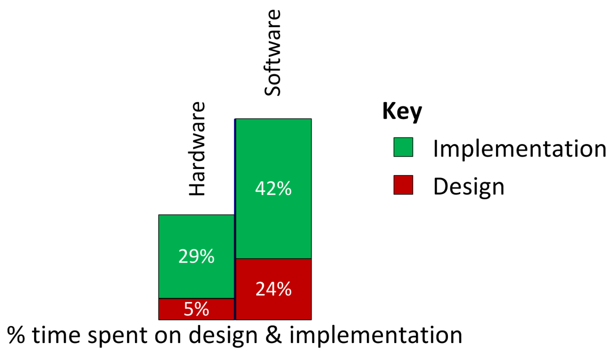

Figure 6.1: Comparison of time spent on hardware and software design/implementation

About 64% of software development time was spent on the implementation, compared to about 85% of hardware development time. As displayed in Figure 6.1, this accounts for 29% of the total design and implementation time. There are several factors causing this disparity. For a start, the hardware design was largely automated and hugely simplified thanks to the provision of stock designs by Xilinx, resulting in an extremely quick design phase. However, a bottleneck occurred in the hardware implementation due to a lack of familiarity with the design tools, configuration problems in the laboratory installations of the Xilinx tools and various issues using the VLAB. While the fundamental approach of leveraging known working hardware designs certainly allowed the hardware development to be completed much faster, the avoidable inefficiencies in the process were not adequately foreseen. These manifested themselves as significant opportunity cost, evident in the lack of attention given to secondary requirements.

## 6.3 Architecture

### 6.3.1 Expectations

The implementation of an SMP architecture was intended to produce a system with desirable traits on many fronts. By having all processor cores executing the same code, it was anticipated that the system would offer better maintainability than if it used an AMP architecture. For example, kernel modifications would be centralised and by default (i.e., unless protected by a critical section) all memory would be accessible to all tasks. As well as making it easy to implement inter–task communication using shared memory, it was also expected that this would greatly simplify the problem of changing the core on which a task is running (in an AMP system with private working memory for each core, doing this would require the task's memory to be copied to the private memory of the new core). Using an SMP architecture it was also expected that memory usage would be lower, an important consideration when dealing with embedded systems. Simply, with only one instance of FreeRTOS being executed, only one instance of its code is required in memory.

### 6.3.2 Results

The implemented shared memory model implicit in the SMP architecture did provide the benefits anticipated. Operating system and task code is shared by all processor cores, meaning that they only need to be maintained within a single code base and only one instance needs to be resident in memory at once. This also means that the scheduler is able to schedule tasks on different cores without performing expensive copy operations to move their stacks to different areas of memory. In addition, however, memory areas private to each processor core are required for initialisation code and low–level hardware interaction libraries. Using the implemented synchronisation API, shared memory can also be used as a medium for tasks to communicate with one another without additional hardware components.

### 6.3.3 Evaluation

Although the justifications were ultimately sound, some of the design expectations did not offer benefits that were quite as definitive in practice as they might have at first seemed. Although a single FreeRTOS and task code base is shared by all cores, the ignition code required to initialise all slave cores has to be maintained separately from the operating system and application tasks. While far from burdensome, this does complicate the creation of multicore applications a bit, although this has been mitigated to some extent by ensuring that the core ignition code is application–agnostic and performs only the minimum of necessarily core–specific configuration.

Maintainability was one of the key components of the decision to opt for an SMP architecture. By allowing all cores to execute the same OS and application code, it was expected that modifying and testing it in a multicore environment would be easier by virtue of the fact that memory was an intrinsically shared resource that was readily accessible from within normal application code without having to address a special shared component separately. However, due to the way in which the Xilinx tools operate, debugging a system in an SMP configuration is significantly more difficult than one in an AMP configuration. This is due to the fact that while the debugger can operate on multiple processor cores at the same time, only the code sent to the FPGA when a given core is started is visible to the debugger when running on that core.

Because the OS and application code is always sent to the FPGA when the first core is started, debugging the execution of this code on slave cores, as far as the author is aware, is not possible via the Xilinx development environment unless the live disassembler is used. The live disassembler disassembles the machine code in memory being executed and displays the corresponding assembler mnemonics for use in manipulating breakpoints and stepping through instructions. As well as being more tedious than stepping through the original C code, a bug in the tool (observed on the development system used in this project, at least) causes it to become unstable after the first few step commands, making it essentially unusable for most debugging tasks involving live execution.

It is certainly conceivable that this might be a serious hindrance to application developers: convenient and easy–to–use debugging tools make application development quicker and bug fixing more simple. It must be acknowledged that in an AMP architecture, operating system code would be visible to the debugger regardless of the core on which it is executing. It is likely, however, that task code would still only be visible to one core, unless each core was given its own version of every task in the system. Even so, this would imply quite significant memory requirements, all in the interests of providing a workaround for a problem specific to a certain platform's tools. Perhaps from a practical perspective, a development team might deem it adequate justification. However, the aim of this project has been to create a starting point from which many FreeRTOS portable layers may be given multicore support. Quite simply, to sacrifice what is otherwise a sound design decision to enhance the use of a single vendor's tools is simply not compatible with this aim.

## 6.4 Software

### 6.4.1 Expectations

It was expected that by modifying the multiplicity of the data structure responsible for tracking the TCB of the currently executing task to reflect all currently executing tasks across all cores, the remaining modifications required (with the exception of those relating to mutual exclusion) would follow directly from this. Thread safety, it was anticipated, could then be applied to the modifications as required.

It was also expected that by using mutual exclusion, the scheduler could be modified to correctly schedule tasks on any core using shared priority lists. The semantics of the single–core FreeRTOS scheduler could then be preserved, ensuring that tasks fixed at higher priorities would always run in preference to those fixed at lower ones.

The intention behind the implementation of the software–based synchronisation API was to provide a largely platform–independent way of safely accessing shared resources in a predictable manner. This could then be used to make the FreeRTOS kernel thread–safe and expose the necessary functions to task code for the same purpose.

### 6.4.2 Results

At the most basic level, the data structure multiplicity was key in providing the kernel code with "awareness" of multiple cores. By indexing the TCBs relating to executing tasks by the core ID and restricting the modification of this structure to elements relating to the core on which the code performing the modification is running, the kernel is able to manipulate the TCBs of executing cores without the need for mutual exclusion. Indeed, this principle has been replicated in the portable layer: the "yield flag" indicating the desire of a processor core to perform a context–switch has been modified in the same way.

The scheduler's modifications, while effective in providing multicore priority–based scheduling, have been revealed to suffer from some performance issues. Fully parallelisable tasks running on multicore FreeRTOS with two available processing cores achieve a 160% execution time improvement over the same tasks executing on the originial, single–core FreeRTOS. This is significantly below the theoretical 200% improvement and is indicative of the existence of bottlenecks requiring optimisation throughout the context switching and scheduling code.

Alagarsamy's algorithm was successfully implemented to provide software synchronisation, tests tentatively suggesting that the implementation is thread–safe. See section 6.1.1 above for more details.

Platform independence, however, was not total: deferral to the portable layer is required in order create memory barriers, although this is the only hardware–dependent element of the implementation.

### 6.4.3 Evaluation

Considering performance optimisation has not been included within the scope of this project, the execution time results are good. Indeed, the lowest observed execution time performance exceeds the criteria defined by REQ-PF-01 on page 35, while the highest execution time improvements (achievable after the first seven minutes of execution) approach the theoretical maximum. While the cause of the relationship between the performance improvements and the elapsed execution time is not fully understood, it seems that the code likely to most improve performance resides within the scheduler, in which a high proportion of time can be spent waiting for access to critical sections. This could be mitigated to some extent by a flat reduction in code covered by critical sections and by increasing the specificity of the synchronisation targets. For example, the current implementation uses a single mutex object to denote the "scheduler's lock", but this could be expanded into two objects responsible for locking access to priority lists and the currently executing tasks array separately. Although arguably a symptom of the decision to share the priority lists amongst all cores, it was implemented in this way in order to make changing the core on which a task is executed easier. Without this it is likely that the synchronisation problem would rear its head elsewhere, as at some point a critical section would be required to negotiate the selection of a task from another core's list for execution.

Additionally, when checking if a task is being executed by another core there is no distinction made by the multicore scheduler between a task executing and a task waiting to enter the scheduler's critical section. It could be that a lower priority task, having yielded and entered the scheduler's critical section, might prevent a higher priority task from being scheduled because it perceives it to be executing on another core. Although this perception is technically correct, it arguably gives rise to an inefficiency: the task is so close to non–execution (having already been swapped out and simply be waiting for the scheduler to select another task to swap in) that treating it differently from a task that is not executing at all is somewhat arbitrary, yet prevents it from being scheduled. This could potentially be addressed by notifying the scheduler when a task starts waiting to enter its critical section, allowing it to reschedule a higher–priority task waiting on the critical section on another core rather than ignore it. Care would have to be taken to ensure that if another core did "take" a task in this way, the core originally executing the task did not attempt to resume it.

Through static code analysis, it is possible to determine the synchronisation algorithm's worst–case execution time. This is because Alagarsamy's mutual exclusion algorithm promises bounded bypasses, limiting their number to $n$ - 1.[19, p. 36] This translates directly into the maximum number of times a compiled instruction will be executed. The compiled code can then be disassembled and the processor documentation for the target platform used to identify the number of clock cycles required by each instruction. A profile can then be built of the maximum number of clock cycles required. By using the processor's clock speed and hardware clock frequency, this information can be expressed in terms of ticks and therefore used in the evaluation of the code's potential effect on task deadlines.

For example, consider a processor core with a clock speed of 100 MHz. It is capable of performing 100 million cycles per second. For the sake of argument, consider that the code being analysed consists of various instructions that require a maximum of 1.5 million cycles to execute (determined from analysis of the disassembed code). The objective is to determine what effect this might have on a task's relative deadline, measured in ticks, on execution of the code. If the system clock is configured to tick at a rate of 1,000 Hz, a tick occurs one thousand times per second, meaning that during execution of the code being analysed, up to 15 ticks may occur. This information can then be incorporated into the application design to ensure that code is not executed that may put a deadline at risk.

## 6.5 Requirements

### 6.5.1 Functional Requirements

**REQ-PF-01 (page 35)** This requirement was met. Highly concurrent applications benefit from an execution time improvement of 160%.

**REQ-PF-02 (page 35)** This requirement was met. Refer to the `vCPUAcquireMutex` and `vCPUReleaseMutex` functions in `tasks.c`.

**REQ-PF-03 (page 36)** This requirement was met. The kernel features modified by this project use mutual exclusion internally as required and do not impose any burden on application code to ensure the thread–safety of API calls.

**REQ-PF-04 (page 36)** This requirement was met. Refer to the `vTaskAcquireNamedMutex` and `vTaskReleaseNamedMutex` functions in `tasks.c`.

**REQ-PF-05 (page 36)** This requirement was met. See `synctest.c` for a demonstration.

**REQ-PF-06 (page 37)** This requirement was met.

**REQ-PF-07 (page 37)** This requirement was met. See `synctest.c` and `conctest.c`.

**REQ-PF-08 (page 38)** This requirement was not met due to the fact that requirement REQ-SF-09, part of the implementation, was not met.

**REQ-SF-09 (page 38)** This requirement was not met. A lack of remaining time after the implementation of the primary requirements meant that this secondary requirement could not be tested. Note, however, that it has been implemented, as explained in section 5.2.4 on page 65.

### 6.5.2 Non–functional Requirements

**REQ-PN-10 (page 39)** This requirement was met. The source code is available via SVN at https://sourceforge.net/projects/freertosxcore/.

**REQ-PN-11 (page 39)** This requirement was met. Required code modifications consist of: (a) an additional "CPU affinity" parameter for `xTaskCreate`; (b) the use of the `systemTaskParameters` struct to access application parameters passed in to a task; (c) the inclusion of a task handle parameter to some API functions, accessible via a pointer to a `systemTaskParameters` struct; and (d) the use of two functions for entering and exiting critical sections for use in ensuring thread safety, as required.

**REQ-PN-12 (page 40)** This requirement was partially met. With all tool configuration complete, it is possible to reproduce the hardware design within 15 minutes. However, without correctly installed or configured tools, it takes approximately 25 minutes. Refer to section 5.3 on page 66 for detailed instructions.

# Chapter 7

# Future Work

## 7.1 Proving the Software Synchronisation

With access to more complete information about MicroBlaze out–of–order execution semantics, the implementation of Alagarsamy's algorithm could well be proven. In addition, very high confidence tests consisting of millions of runs and hundreds of tasks are necessary to provide the confidence required for use of this code in a production environment.

## 7.2 Scheduler Optimisation

As discussed in section 6.4.3 on page 84, the scheduler's performance leaves room for improvement. Analysing in detail how the scheduling code can be improved to reduce the quantity of time that processor cores find themselves waiting for access to a critical section, while maintaining thread safety, would be essential to improving the performance of the system.

## 7.3 Formal Analysis of Real–Time Properties

While guided by the importance of creating deterministic code that follows real–time principals, the implementation's adherence to these principals has by no means been proven. Formally analysing the efficacy of the implementation in this regard would be essential to preparing it for use in a real application. The extension of the Z model of single–core FreeRTOS built by Shu Cheng, a researcher under Professor Jim Woodcock's supervision, to incorporate the multicore amendments might provide an interesting way not only to investigate their adequacy in a real–time context, but also to formally define how they cause the modified version of FreeRTOS to differ in operation from the original.

It would also be particularly interesting to perform a detailed investigation, using static analysis, of the worst–case execution time of the various FreeRTOS components. The MicroBlaze port would lend itself very well to this, given its relative simplicity. Perhaps an automated tool could be developed to parse the disassembly and calculate (with a degree of naïvety, of course) the worst–case execution time in cycles. Output from such a tool could then be used to perform a more detailed analysis of the code from which a profile of each FreeRTOS component could be built to inform application development and future kernel improvements.

## 7.4 Providing Synchronisation Alternatives

While the use of software–based synchronisation certainly has its benefits, particularly on MicroBlaze, many architectures provide extensive hardware–level synchronisation that is more efficient. Extending the multicore modifications to allow synchronisation functions to be defined in the portable layer in addition to those built in to the kernel would allow ports targeted at this kind of hardware to make full use of it. This could be done by having task code use macros to access the synchronisation features which could then be overriden in the portable layer.

The synchronisation algorithm implemented forces a processor core waiting to enter a critical section to busy wait. Although it may well be swapped out if a tick occurs during this time, it would be worth investigating the implications of explicitly yielding at this point and to what degree this might improve the system's performance.

## 7.5 MicroBlaze Cacheing Support

The original MicroBlaze port of FreeRTOS supports instruction and data cacheing. In the interests of time, this support was not preserved in the multicore modifications. Assessing the implications of introducing it and how to effectively manage the caches in a concurrent environment would be an extremely worthwhile exercise, particularly as caching can provide significant performance advantages.

## 7.6 Extending Multicore Support

The modifications in this project have been limited to the minimum set of FreeRTOS components, and have only been implemented for the configuration specified in the source code (code excluded by configuration macros has not been modified). Extending the multicore support to include not only all of FreeRTOS, but also additional peripherals (such as networking components) and third–party software libraries (such as those providing video encoding), would make an excellent candidate for future work. Similarly, the creation of multicore versions of the other available FreeRTOS portable layers would also have the potential to be extremely useful to the FreeRTOS community. Fully testing the modifications on systems with $n$ cores would also be highly desirable, particularly as this would allow for the possibility of implementing massively parallel applications, particularly on FPGA platforms.

# Chapter 8

# Conclusions

With no multicore version of FreeRTOS currently available, the modifications required for the operating system to schedule tasks on multiple processors were extensive, and consideration had to be given at each level of design to the interplay between the system's hardware and software requirements. For example, a principal high–level design decision was that the system would use a symmetric multiprocessing architecture, in which all processors execute the same "instance" of the operating system. A major advantage of this is that only one instance of the operating system is required in memory, making the overall memory footprint much lower than in an equivalent asymmetric model (in which each processor executes its own instance of the OS). It also helped keep the scheduler modifications simple, avoiding the onerous task of having to copy task stacks between private memory areas when moving a task from one processor core to another.

However, this had implications on the hardware design: in order for the operating system to be aware of the processor on which it is currently executing, special–purpose read–only registers had to be configured to allow the processors to identify themselves to the software. Indeed, the entire notion of recognising "the current processor" had to be built into FreeRTOS. As well as being necessary for obvious things like ensuring the scheduler selects a task for execution on the correct core, the processor ID is used as an index to core–specific data that allows much of the kernel to avoid the complications (and performance penalties) of synchronisation. It also allowed the implementation of core affinity, the ability to bind a task to a core for the duration of its life. Borne from a need to ensure that each idle task only ever executes on one processor core, core affinity also allows applications to be tuned for optimal performance or to accommodate core–specific hardware. For example, a quirk of the hardware design implemented as part of this project was that only the first core had access to the serial port, meaning that tasks needing to communicate with the development PC had to be given affinity for it.

This affinity, as with all the properties of multicore, had to be reflected in the scheduler. While the simplicity of the original FreeRTOS scheduler is truly magnificent, the realities of multicore meant confronting inevitable complications. With the constraint of core affinities and the possibility that tasks in the shared ready lists might already be executing on other cores when retrieved by the scheduler, careful consideration had to be given as to how the scheduler should be modified to accommodate these things while remaining a spiritual successor, semantically, to the original code. By remaining a fixed priority scheduler but with a remit to look beyond tasks with the "top ready" priority when unable to select anything for execution, the multicore scheduling algorithm exhibits behaviour that is a natural extension of the original.

Implementing mutual exclusion correctly was essential, but MicroBlaze does not currently support any built–in synchronisation features that work across multiple processors.[9, p. 227] Although an additional mutex peripheral could have been included in the hardware design, it was decided that due to FreeRTOS's popularity on so many different platforms the kernel itself should provide a platform–agnostic implementation. Thanks to the collective works of Peterson, Block, Woo and Alagarsamy, an optimal and truly elegant algorithm has been developed to do just this. Using only shared memory and memory barriers, a tentative implementation has been produced that is integrated into the FreeRTOS kernel and available to tasks and OS components alike.

It has been suggested that something be written about engineering lessons learnt during the course of this project. In what is perhaps an indictment of the author, the lessons imparted by this work have been old rather than new: that time is not just the enemy of software projects, but of everything — every hour should be used with care; that careful thought and design is the most valuable way to start the development of software; that no matter how certainly it seems that a computer is exhibiting erratic, inexplicable behaviour, the likelihood of this actually being the case (and not being quickly apparent) is infinitesimally low; and, perhaps most importantly, that these lessons have proven to leave impressions of a fleeting nature, and must therefore be given all the more attention.

# Appendix A

Listing 1: Lines to remove from the system.ucf file

```
#######################################
# DQS Read Postamble Glitch Squelch circuit related constraints
#######################################

#######################################
# LOC placement of DQS-squelch related IDDR and IDELAY elements
# Each circuit can be located at any of the following locations:
#  1. Ununsed "N"-side of DQS diff pair I/O
#  2. DM data mask (output only, input side is free for use)
#  3. Any output-only site
#######################################

INST "*/gen_dqs[0].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y56";
INST "*/gen_dqs[0].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y56";
INST "*/gen_dqs[1].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y18";
INST "*/gen_dqs[1].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y18";
INST "*/gen_dqs[2].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y22";
INST "*/gen_dqs[2].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y22";
INST "*/gen_dqs[3].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y60";
INST "*/gen_dqs[3].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y60";
INST "*/gen_dqs[4].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y62";
INST "*/gen_dqs[4].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y62";
INST "*/gen_dqs[5].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y216";
INST "*/gen_dqs[5].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y216";
INST "*/gen_dqs[6].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y220";
INST "*/gen_dqs[6].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y220";
INST "*/gen_dqs[7].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y222";
INST "*/gen_dqs[7].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y222";

##########################################
# LOC and timing constraints for flop driving DQS CE enable signal
# from fabric logic. Even though the absolute delay on this path is
# calibrated out (when synchronizing this output to DQS), the delay
# should still be kept as low as possible to reduce post-calibration
# voltage/temp variations - these are roughly proportional to the
# absolute delay of the path
##########################################

INST "*/u_phy_calib_0/gen_gate[0].u_en_dqs_ff"  LOC = SLICE_X0Y28;
INST "*/u_phy_calib_0/gen_gate[1].u_en_dqs_ff"  LOC = SLICE_X0Y9;
INST "*/u_phy_calib_0/gen_gate[2].u_en_dqs_ff"  LOC = SLICE_X0Y11;
INST "*/u_phy_calib_0/gen_gate[3].u_en_dqs_ff"  LOC = SLICE_X0Y30;
INST "*/u_phy_calib_0/gen_gate[4].u_en_dqs_ff"  LOC = SLICE_X0Y31;
INST "*/u_phy_calib_0/gen_gate[5].u_en_dqs_ff"  LOC = SLICE_X0Y108;
INST "*/u_phy_calib_0/gen_gate[6].u_en_dqs_ff"  LOC = SLICE_X0Y110;
INST "*/u_phy_calib_0/gen_gate[7].u_en_dqs_ff"  LOC = SLICE_X0Y111;
```

# Appendix B

Listing 2: Lines to add to the system.ucf file

```

###########################################
# LOC placement of DQS-squelch related IDDR and IDELAY elements
# Each circuit can be located at any of the following locations:
#  1. Ununsed "N"-side of DQS diff pair I/O
#  2. DM data mask (output only, input side is free for use)
#  3. Any output-only site
###########################################

INST "*/gen_dqs[0].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y96";
INST "*/gen_dqs[0].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y96";
INST "*/gen_dqs[1].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y58";
INST "*/gen_dqs[1].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y58";
INST "*/gen_dqs[2].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y62";
INST "*/gen_dqs[2].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y62";
INST "*/gen_dqs[3].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y100";
INST "*/gen_dqs[3].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y100";
INST "*/gen_dqs[4].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y102";
INST "*/gen_dqs[4].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y102";
INST "*/gen_dqs[5].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y256";
INST "*/gen_dqs[5].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y256";
INST "*/gen_dqs[6].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y260";
INST "*/gen_dqs[6].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y260";
INST "*/gen_dqs[7].u_iob_dqs/u_iddr_dq_ce"  LOC = "ILOGIC_X0Y262";
INST "*/gen_dqs[7].u_iob_dqs/u_iodelay_dq_ce"  LOC = "IODELAY_X0Y262";

INST "DDR2_SDRAM/DDR2_SDRAM/gen_no_iodelay_grp.
gen_instantiate_idelayctrls[1].idelayctrl0" LOC = IDELAYCTRL_X0Y2;
INST "DDR2_SDRAM/DDR2_SDRAM/gen_no_iodelay_grp.
gen_instantiate_idelayctrls[0].idelayctrl0" LOC = IDELAYCTRL_X0Y6;
INST "DDR2_SDRAM/DDR2_SDRAM/gen_no_iodelay_grp.
gen_instantiate_idelayctrls[2].idelayctrl0" LOC = IDELAYCTRL_X0Y1;

###########################################
# LOC and timing constraints for flop driving DQS CE enable signal
# from fabric logic. Even though the absolute delay on this path is
# calibrated out (when synchronizing this output to DQS), the delay
# should still be kept as low as possible to reduce post-calibration
# voltage/temp variations - these are roughly proportional to the
# absolute delay of the path
###########################################

INST "*/u_phy_calib_0/gen_gate[0].u_en_dqs_ff"  LOC = SLICE_X0Y48;
INST "*/u_phy_calib_0/gen_gate[1].u_en_dqs_ff"  LOC = SLICE_X0Y29;
INST "*/u_phy_calib_0/gen_gate[2].u_en_dqs_ff"  LOC = SLICE_X0Y31;
INST "*/u_phy_calib_0/gen_gate[3].u_en_dqs_ff"  LOC = SLICE_X0Y50;
INST "*/u_phy_calib_0/gen_gate[4].u_en_dqs_ff"  LOC = SLICE_X0Y51;
INST "*/u_phy_calib_0/gen_gate[5].u_en_dqs_ff"  LOC = SLICE_X0Y128;
INST "*/u_phy_calib_0/gen_gate[6].u_en_dqs_ff"  LOC = SLICE_X0Y130;
INST "*/u_phy_calib_0/gen_gate[7].u_en_dqs_ff"  LOC = SLICE_X0Y131;
```

# References

[1] P. A. Laplante, *Real-time systems design and analysis*. New York: Wiley-IEEE, 2004.

[2] V. Zlokolica, R. Uzelac, G. Miljkovic, T. Maruna, M. Temerinac, and N. Teslic, “Video processing real-time algorithm design verification on embedded system for HDTV”, *Engineering of Computer Based Systems, 2009. ECBS-EERC ’09.*, pp. 150 –151, Sep. 2009.

[3] (Aug. 2011). Introduction to FreeRTOS, [Online]. Available: `http://www.freertos.org/`.

[4] (Aug. 2011). RTOS concepts, [Online]. Available: `http://www.chibios.org/`.

[5] H. Sutter. (Aug. 2011). The free lunch is over: a fundamental turn toward concurrency in software, [Online]. Available: `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[6] D. Geer, “Chip makers turn to multicore processors”, *Computer*, vol. 38, no. 5, pp. 11 –13, Sep. 2005.

[7] N. Lomas. (Aug. 2011). Dual-core smartphones: the next mobile arms race, [Online]. Available: `http://www.silicon.com/technology/mobile/2011/01/12/dual-core-smartphones-the-next-mobile-arms-race-39746799/`.

[8] D. Andrews, I. Bate, T. Nolte, C. Otero-Perez, and S. Petters, “Impact of embedded systems evolution on RTOS use and design”, presented at the Proceedings of the 1st Workshop on Operating System Platforms for Embedded Real-Time Applications.

[9] Xilinx, *MicroBlaze Processor Reference Guide*. San Jose, CA: Xilinx, 2011.

[10] (Aug. 2011). FPGAs - under the hood, [Online]. Available: http://zone.ni.com/devzone/cda/tut/p/id/6983.

[11] (Aug. 2011). VirtualLab introduction, [Online]. Available: http://www.jwhitham.org.uk/c/vlab/.

[12] (Aug. 2011). Response time and jitter, [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:jitter.

[13] (Aug. 2011). Task priorities, [Online]. Available: http://www.freertos.org/a00015.html#TaskPrior.

[14] R. Mall, *Real-Time Systems: Theory and Practice*. Delhi: Pearson Education India, 2009, p. 68.

[15] H. Huang, *Exploiting Unused Storage Resources to Enhance Systems' Energy Efficiency, Performance, and Fault-Tolerance*. Delhi: ProQuest, 2006, p. 23.

[16] (Aug. 2011). Implementing scalable atomic locks for multi-core Intel EM64T and IA32 architectures, [Online]. Available: http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures/.

[17] Xilinx. (2011). XPS mutex documentation, [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/xps_mutex.pdf.

[18] G. L. Peterson, "Myths about the mutual exclusion problem", *Information Processing Letters*, vol. 12, no. 3, pp. 115 –116, 1981.

[19] K. Alagarsamy, "A mutual exclusion algorithm with optimally bounded bypasses", *Information Processing Letters*, vol. 96, no. 1, pp. 36 –40, 2005.

[20] M. Hofri, “Proof of a mutual exclusion algorithm – a classic example”, *ACM SIGOPS Operating Systems Review*, vol. 24, no. 1, pp. 18 –22, 1989.

[21] (Aug. 2011). Tasks and co-routines, [Online]. Available: http://www.freertos.org/index.html?http://www.freertos.org/taskandcr.html.

[22] (Aug. 2011). How to create and program interrupt-based systems, [Online]. Available: https://wiki.ittc.ku.edu/ittc/images/d/df/Edk_interrupts.pdf.

[23] (Aug. 2011). On the use of the mainline stack by the interrupt handler in the MicroBlaze port of FreeRTOS, [Online]. Available: http://www.freertos.org/index.html?http://interactive.freertos.org/entries/200807.

[24] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, “A design kit for a fully working shared memory multiprocessor on FPGA”, presented at the Proceedings of the 17th ACM Great Lakes symposium on VLSI, ACM, 2007, pp. 219 –222.

[25] A. Hung, W. Bishop, and A. Kennings, “Symmetric multiprocessing on programmable chips made easy”, in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, IEEE Computer Society, 2005, pp. 240–245.

[26] (Aug. 2011). Setthreadaffinitymask function, [Online]. Available: http://msdn.microsoft.com/en-us/library/ms686247%28v=vs.85%29.aspx.

[27] fpgadeveloper.com. (2011). Convert an ml505 edk project for the xupv5, [Online]. Available: http://www.fpgadeveloper.com/tag/tutorial.