SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Adapting FreeRTOS for Multicore: an Experience Report

James Mistry[1]*, Matthew Naylor[2], and Jim Woodcock[2]

[1]*British Telecom, Adastral Park, Ipswich, IP5 3RE*
[2]*Department of Computer Science, University of York, YO10 5GH.*

SUMMARY

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 paper documents our experience of adapting FreeRTOS, a popular embedded real–time operating system, to support multiple 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. Mutual exclusion is achieved in an almost completely platform–agnostic manner, preserving one of FreeRTOS's most attractive features: *portability*.

Received ...

KEY WORDS: embedded systems; operating systems; multicore computing; task scheduling; software-based mutual exclusion; field-programmable gate-arrays.

1. INTRODUCTION

FreeRTOS is a popular open–source embedded real–time operating system that has been ported to over 30 different hardware platforms and receives over 75,000 downloads per year. The aim of this work is to produce a version of FreeRTOS that supports multicore hardware, an increasingly important requirement given that such hardware is now appearing widely even in embedded devices. The software developed during the course of the work 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. This paper, based on the first author's MSc thesis [1], documents our experience of the whole software development process, including requirements, design choices, implementation details, critical evaluation, and the lessons learned.

*Correspondence to: james.mistry@bt.com

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 [2, p4]. Examples of such systems include those responsible for controlling airbags in cars, control surfaces in aircraft and missile early warning alerts.

In order to prove that real–time systems meet their deadlines it is necessary to identify their worst–case performance scenarios. If systems become too complex, the cost of performing such analysis can become 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. As with any Real–Time Operating System (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.

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 extending FreeRTOS to support multicore hardware 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.

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 [5]. 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" 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 [6]. More fundamentally, a lack of action has the potential to cause the usefulness of embedded real–time systems to hit a wall. What will the system designers do when future iterations of the application must perform more tasks, process larger data and provide faster response times without having access to matching improvements in single–core hardware? Migration to multicore is inevitable.

1.2. Method

This paper 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.

Although MicroBlaze has been chosen as the test–bed, it is important to point out that the majority of the multicore modifications made to FreeRTOS are fully general, not specific to any particular

DOI: 10.1002/spe

platform. In fact, making the modifications easily portable to a range of different multicore architectures is one of the primary design goals in this work. Portability is, after all, one of the main features of the FreeRTOS design, and this work aims to preserve it.

Having said that, the MicroBlaze platform does have attractions. MicroBlaze FPGA designs are readily available and can be implemented with little effort. And by being able to strip down the hardware configuration (for example by disabling 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. Another attraction is that the number of processor cores need not, in principle, be limited to just two: modern FPGAs are capable of hosting tens of MicroBlaze cores.

1.3. Outline

Section 2 discusses the background issues, technologies and research surrounding this topic. Section 3 details and analyses the requirements. Section 4 explains the system design, detailing significant algorithms and other key design decisions. Section 5 discusses the components of the system, how they work and the justifications for the exact method of their implementation. Section 6 evaluates the adequacy of the implementation with regard to the defined requirements. Section 7 summarises the work, recapping the main lessons learned, and suggests avenues for future work.

2. BACKGROUND

2.1. Multicore processors

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"). 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 that share the processor, the fewer of each task's instructions the processor can execute in a given amount of time.

In contrast, a multicore processor has the ability to execute tasks with *true* simultaneity. The same number of tasks as cores can be executed literally at the same time. 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.

To illustrate, consider a system with many running tasks. Suppose that one task needs to execute a critical section of code involving access to shared variables in an atomic manner, i.e. without interference from any other task. In a single–core system, the programmer can simply write:

```
disableInterupts()
// Critical section of code
enableInterrupts()
```


DOI: 10.1002/spe

Disabling interrupts is all that is necessary to ensure exclusive access because the only way another task may run and therefore gain access to the shared variables on a single–core processor is if the scheduler interrupts the current task and resumes another.

However, on a multicore processor this is not the case. Because all cores execute instructions at the same time, the potential for the simultaneous use of shared resources is not connected to the swapping of tasks. In order to protect against this, an alternative mechanism for creating a critical section that works across all cores in the system is required.

2.2. Mutual Exclusion

The term "mutual exclusion" refers to the desirable situation in which multiple processes are denied simultaneous access to a single shared resource. Usually, it is achieved in an architecture–specific manner. 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.

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 [7]. Because this exchange is guaranteed to be atomic across all cores, it can be used to implement mutual exclusion 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 mutual exclusion.

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 [8]. 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 make maintaining the system harder.

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 [9, p115] for synchronisation between two processes, and is illustrated in Figure 1.

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

DOI: 10.1002/spe

```
// 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 for P0

 // 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 for P1

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

}
```

Figure 1. Peterson's 2-process Mutual Exclusion Algorithm

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.

Generalisations of Peterson's 2-process Algorithm Peterson's 2-process algorithm can be generalised to support *n* competing processes. Perhaps the easiest way to appreciate this is to arrange $n-1$ instances of the 2-process algorithm into a tournament of $log_2(n)$ rounds, as illustrated in Figure 2. In each round, each process competes with one other process for a 2-process lock, and half the processes proceed to the next round. After $log_2(n)$ rounds, only one process remains and it may enter its critical section.

A more direct generalisation of the 2-process algorithm is provided by Peterson although he acknowledges that it requires more memory than other solutions and, most importantly, satisfies fewer constraints [9, p116]. In particular, his generalised algorithm does not satisfy the constraint of being free from starvation, [10, p38] despite some suggestions to the contrary [11, p20]. 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, as coded in Figure 3.

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.

DOI: 10.1002/spe

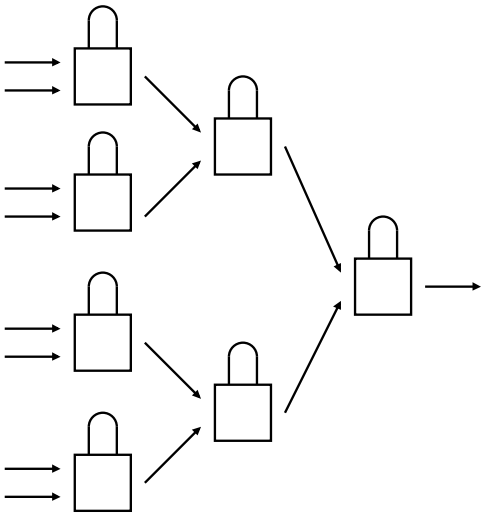

Figure 2. Implementing an n-process lock using $n-1$ instances of 2-process lock. Arrows represent processes. The winning process must acquire (or pass through) $log_2(n)$ locks. In this example, $n = 8$.

```
// Let i be the current process ID.
// Let n be the number of processes.

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

for (int j=1; j<n; j++) {
  Q[i] = j;
  turn[j] = i;

  do {
    stageClear = 1;

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

      if (Q[k] >= j) {
        stageClear = 0;
        break;
      }
    }
  } while ((stageClear == 0) && (turn[j] == i));
}

// Critical section for process i

Q[i] = 0; // Exit protocol
```

Figure 3. Peterson's n–Process Mutual Exclusion Algorithm

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

DOI: 10.1002/spe

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 [10, p36]. 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 [10, p38]. 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 [10, p38]. Alagarsamy introduces his own algorithm that promises "optimal bypasses" and solves this inefficiency by incorporating the modifications made by Block and Woo [10, p38]. Alagarsamy's solution to the unbounded bypass problem works by having processes "promote" [10, p38] 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.

Memory Barriers A challenge that exists with software-based mutual exclusion 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.

2.3. FreeRTOS

FreeRTOS is an RTOS written in C. As of August 2011, it has been ported to twenty–seven different architectures [3]. 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.

Overview The division between the hardware independent and portable layers is expressed in terms of the source files in Figure 4.[†] 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:[‡]

[†]Header files are not included in this illustration.
[‡]Components marked † are excluded from the scope of this work.

Prepared using speauth.cls
DOI: 10.1002/spe

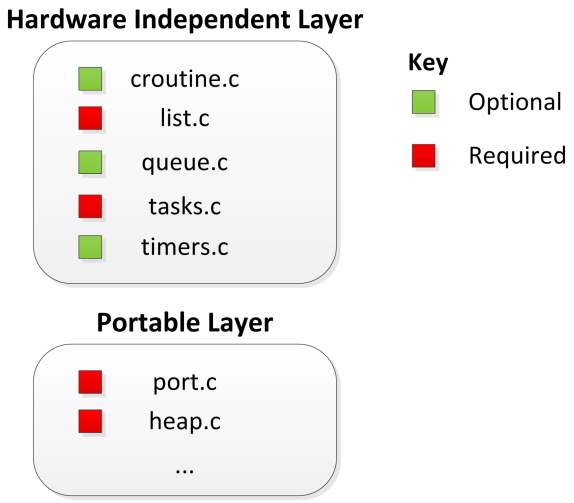

Figure 4. FreeRTOS source structure

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

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 4. This includes a global entry point to the system, taking the form of a `main` function, and 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.

Scheduler FreeRTOS employs a fixed priority pre–emptive scheduler [12]. 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 remains the same.

DOI: 10.1002/spe

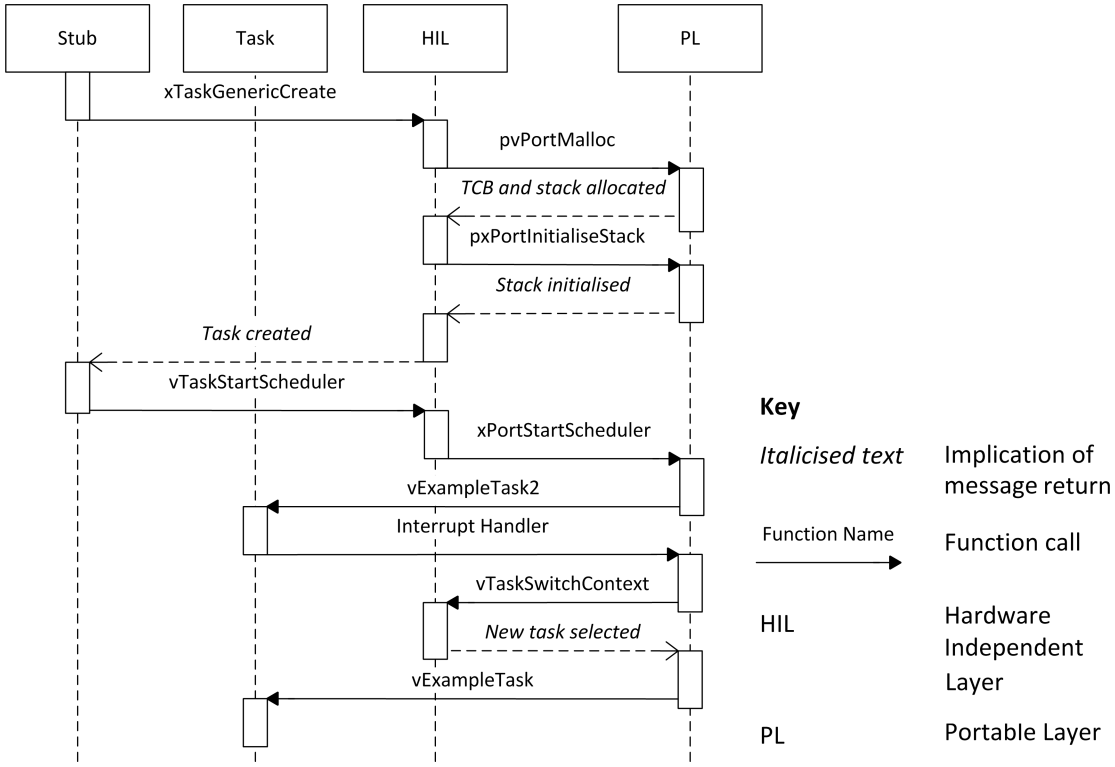

Figure 5. Pseudo–UML sequence diagram showing a task's creation and execution

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 being the highest priority task ready for scheduling, this item 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, the "current item" pointer is advanced to point to the next item, meaning that the scheduling of a task automatically moves another task behind it forward in the list.

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

A task can be defined using the `portTASK_FUNCTION` macro.

```
static portTASK_FUNCTION( vExampleTask, pvParameters ) {
  // Perform task processing here
}
```

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:

DOI: 10.1002/spe

```
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.

The 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 `main` function (referred to as 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.

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.

On MicroBlaze, `_interrupt_handler` is the function to which the processor jumps when an interrupt occurs. This calls `vPortSaveContext` which transfers the contents of the processor's registers, including the address at which to continue execution, into memory. 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 the flag indicates a yield should occur, then `vTaskSwitchContext` is called 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`.

Interrupt Controller The MicroBlaze port requires an interrupt controller to be included in the hardware design. An interrupt controller accepts multiple interrupt sources as input, forwarding

DOI: 10.1002/spe

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 [13] — 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`.

3. OBJECTIVES

The objectives of this work are formulated by the following requirements.

3.1. Functional Requirements

REQ-01 *FreeRTOS must allow the concurrent scheduling of tasks on a dual–core processor.* Ignoring optional FreeRTOS 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.

REQ-02 *Operating system components must be implicitly thread safe.* Operating system components will utilise mutual exclusion features automatically when required to ensure thread-safety, and will not expose such concerns to application tasks.

REQ-03 *Application tasks must have access to an API providing mutual exclusion.* 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. Using no more than 2 API calls, application tasks will be able to enter and exit a critical region for protected access to a named resource.

REQ-04 *The MicroBlaze port of FreeRTOS must be modified to be compatible with the multicore-enabled hardware independent layer.* 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.

REQ-05 *Example applications must be created to demonstrate concurrency and synchronisation.* To demonstrate the use of concurrency and synchronisation implemented by the FreeRTOS modifications, appropriate applications will be created to exemplify the features added.

REQ-06 *Modifications made to FreeRTOS must scale for use with* n *cores.* It would be unnecessarily limiting to restrict multicore support in FreeRTOS to merely support for dual-core processors. Modifications should scale transparently for use on processors with more than 2 cores.

3.2. Non-Functional Requirements

REQ-07 *Source code must be made publicly accessible*. In order to provide a record of the work'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 work will be made publicly accessible via an Internet-based source code repository.

DOI: 10.1002/spe

REQ-08 *Application development must not be made more difficult*. One of the virtues of FreeRTOS 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 work. For example, 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 work must take a developer familiar with both versions no longer than 5 minutes.

REQ-09 *The results of this work should be reproducible.* Complex hardware requirements have the potential to make any results 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. The time required to reproduce the modified FreeRTOS system developed in this work will not exceed an hour for a competent software engineer with detailed instructions, the required apparatus and no experience of hardware design.

4. DESIGN

4.1. Hardware

A fundamental design decision that spans the hardware and software aspects of this work is that of using an SMP (Symmetric Multiprocessing) architecture in which a single instance of the operating system is stored in shared memory and accessed by all cores. The alternative option (AMP, Asymmetric Multiprocessing) is for each core to have its own instance of the operating system running in its own private memory area.

One attraction of AMP is a reduction in the need for synchronisation since less memory needs to be shared between cores. Another is increased scope for parallelism since cores can in principle fetch instructions simultaneously from parallel memory units. However, any cross-task communication still requires full concurrent synchronisation. And in order to truly benefit from multiple cores, an AMP 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. 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. The queue of waiting tasks can be accessed by all cores so each core can execute any task regardless of whether or not it 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. Furthermore, since the SMP architecture requires less memory, it is arguably more appropriate for the real-time domain where memory constraints are often tight.

The only exception to the use of shared memory is in the initial startup stage of the system, 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 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

DOI: 10.1002/spe

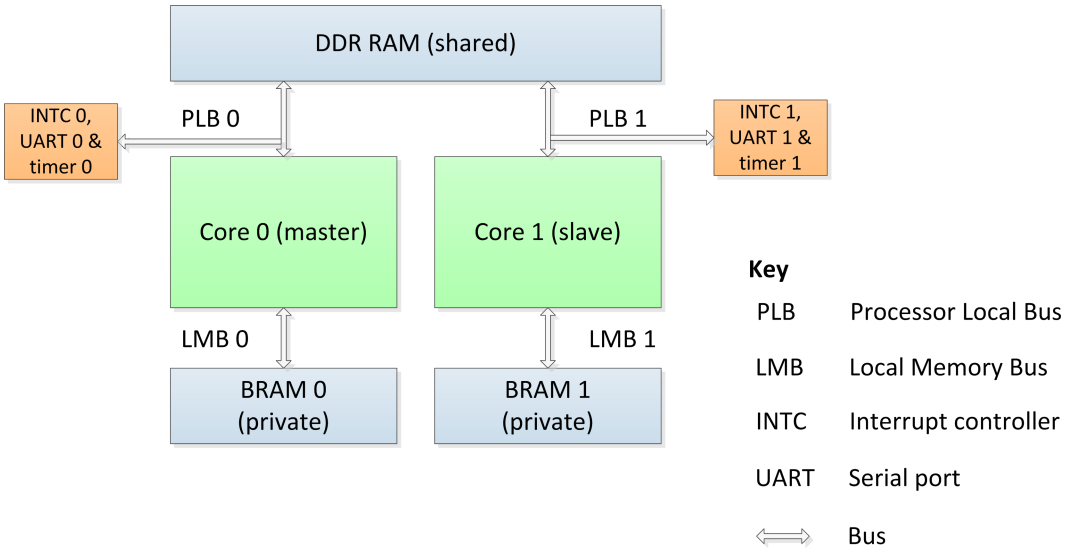

Figure 6. Hardware design

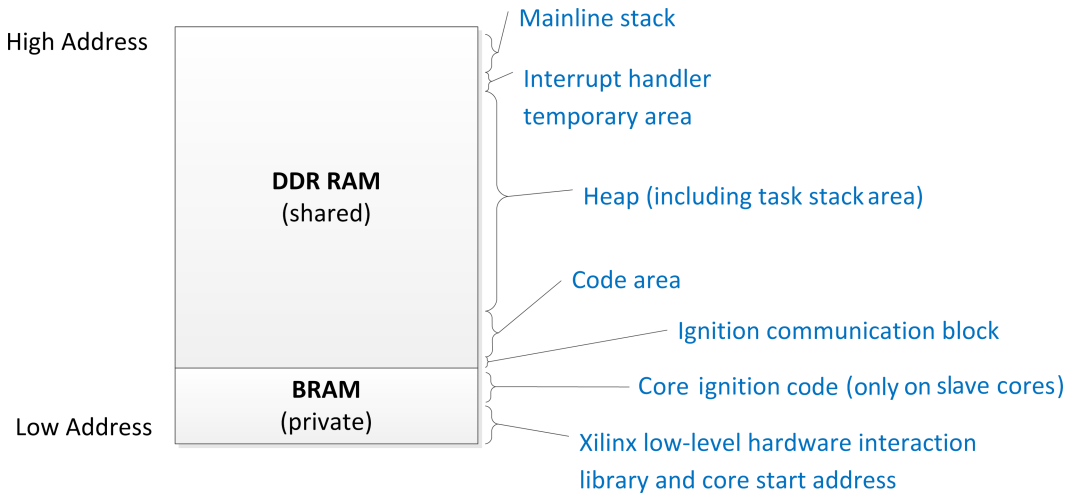

Figure 7. Memory model (not to scale)

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-09, 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) rather than a shared bus for access to memory.

4.2. Core Ignition

Each core begins by executing code located in its own small private block RAM (BRAM), shown at the bottom of Figure 7. The code for the master core immediately jumps to the application entry point (the `main` function), located alongside FreeRTOS and application tasks in a large shared RAM, and performs application–specific initialisation such as the creation of tasks. It also performs some hardware configuration, such as the set up of the master core’s interrupt controller and hardware timer, the drivers for which are provided by Xilinx and reside in the core’s private BRAM. It then starts the FreeRTOS scheduler and begins executing the first task.

DOI: 10.1002/spe

The code for the slave core performs the hardware configuration of the slave core's own peripherals, as well as using a small area of shared RAM (the ignition communication block) 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 in the slave core's BRAM runs and immediately passes execution to the master interrupt handler in FreeRTOS.

4.3. Memory Layout

The code of each task is located in the code area at the bottom of shared memory just after the ignition communication block. The heap is the area used for memory dynamically allocated by tasks at runtime. Perhaps a bit confusingly, it is also 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.

FreeRTOS allocates stack space for each task because, 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 mainline stack would thus grow and contract depending on the nesting of function calls. While generally a transparent and fundamentally important mechanism of the C 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); a single stack would quickly become corrupted if used by multiple tasks in separate threads of execution. 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.[§]

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.

4.4. Task Control Block

The cornerstone of the multicore FreeRTOS modifications is a simple 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

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

DOI: 10.1002/spe

```
// 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];
```

Figure 8. Obtaining the current task's TCB in Multicore FreeRTOS

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.

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. 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. 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.

4.5. 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 [14].

The same is true of multicore FreeRTOS. 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 our 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.

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 4.6), the former two issues are closely related with the indexing of processor cores in the `pxCurrentTCBs` array. The affinity of a task can simply be the integer identifier of the core to which it is bound. A lack of affinity can simply be expressed by a constant greater than the highest TCB array index. Because the TCB array is 0–based, 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 the "no affinity" constant). 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.

4.6. 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. The affinity–aware scheduler behaves as follows:

DOI: 10.1002/spe

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).

4.7. Synchronisation

As discussed in Section 2.2, software–based synchronisation solutions provide a viable alternative to relying on hardware components for the implementation of mutual exclusion, easing portability. Although this option comes with an inevitable overhead, it does not preclude the addition of alternative platform–specific synchronisation in the portable layer if desired.

The synchronisation features are implemented in software using Alagarsamy's improved version of Peterson's Algorithm, discussed in Section 2.2. There is still, however, a practical consideration that must be addressed: 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.

One simple option is to use an integer as a target: on entry to a critical section a task provides 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.

```
// 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.

DOI: 10.1002/spe

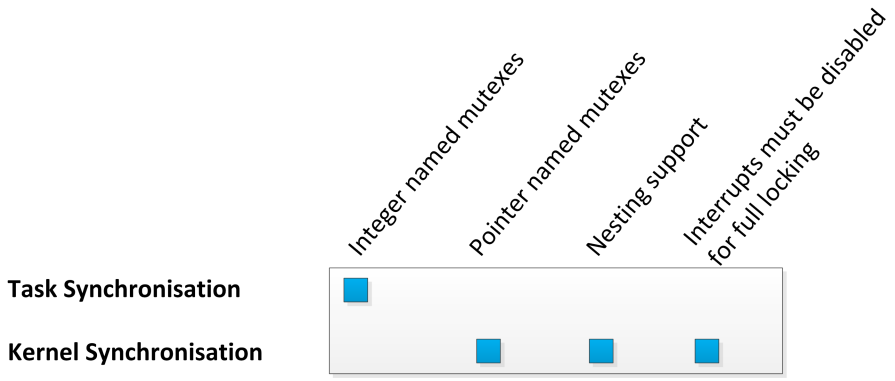

Figure 9. Synchronisation features

The mutex API is divided in two: the first part is for synchronisation by tasks using the functions `vTaskAcquireNamedMutex` and `vTaskReleaseNamedMutex`, while the second part is for synchronisation 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 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:

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.

DOI: 10.1002/spe

```
typedef struct {
  volatile long interruptBridgeSwap[portNUM_PROCESSORS];
  volatile void * deviceHandlers[portNUM_PROCESSORS];
  volatile long initMagic;
  volatile void * interruptHandlerPtr;
  volatile void * startFirstTaskPtr;
} cpuComBlock;
```

Figure 10. Definition of the Ignition Control Block

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 for simplicity. Additionally, several peripherals are included: (1) one 16 KB private BRAM peripheral for each core, used to store core–specific code, such as the ignition communication block; (2) one serial port I/O peripheral for each core; and (3) one shared 256 MB DDR RAM peripheral where all operating system and application code, as well as working memory, is stored. The DDR RAM 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, to determine the ID of the processor which executed it.

5.2. Core Ignition

When a slave core starts it needs 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 FreeRTOS code that starts a core's first task. Because the Xilinx tools require that separate software projects are loaded onto each core, and the main FreeRTOS project is the one loaded onto the master core, this information is not available to the slave core at design–time.

The slave core 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 the `struct` in Figure 10.

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 initial synchronisation of the master and slave cores. When a slave core starts (we assume it is started manually 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.

The interrupt hander of the slave core is a "stub" handler which jumps straight to the address of the FreeRTOS master interrupt handler as specified in the `interruptHandlerPtr` member of the ignition communication block. Unfortunately, the process of loading an address from memory and branching to it cannot be done without using a general–purpose CPU register. Because a slave core's

DOI: 10.1002/spe

```
_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
```

Figure 11. Slave core interrupt handler

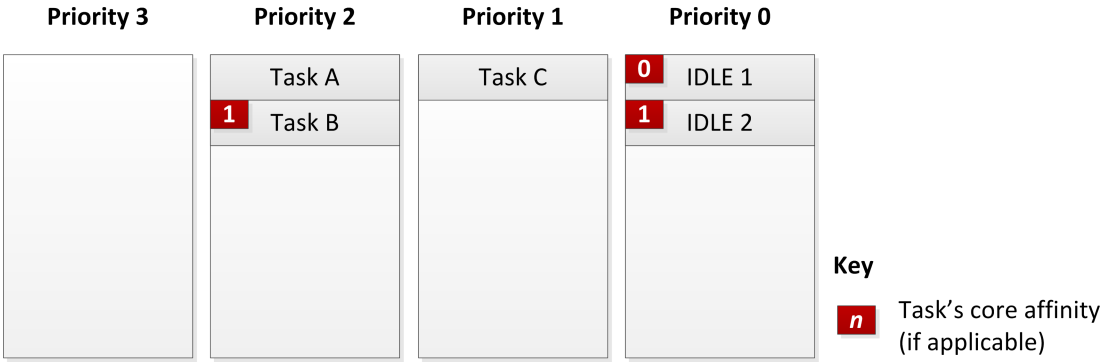

Figure 12. Multicore scheduler ready lists example

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 slave core's handler (`r18`) is first saved to shared memory in the `interruptBridgeSwap` member of the ignition communication block before jumping to the master handler. See Figure 11. The master handler then includes code to check whether it is running on the slave core and if so restores the register (`r18`) with this saved value before continuing to save the task context.

5.3. Scheduler

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.

The way this is implemented in the multicore scheduler is best demonstrated using an example. Figure 12 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

DOI: 10.1002/spe

```
vPortRestoreContext:

mfs     r18,    rpvr0
andi    r18,    r18,    0xFF
muli    r18,    r18,    4

lwi     r3,     r18,    pxCurrentTCBs
lwi     r1,     r3,     0

/* Restore the task's context */
```

Figure 13. Getting the TCB of the task to swap in

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, as coded in Figure 13. 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.

5.4. Synchronisation

Memory access re–ordering presents a particularly significant problem for software mutual exclusion algorithms, as they rely on memory reads and writes occurring in the exact order specified (as explained in Section 2.2). 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 application (explained in Section 6.1). 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 mutual exclusion 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.

Making the API easy to use is also very important. As such, mutex acquisition and release is performed with just two simple function calls, illustrated in Figure 14. The call to the function `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:

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


DOI: 10.1002/spe

```
// An example mutex identifier declaration
static const unsigned portBASE_TYPE EX_MUTEX = 1;

static portTASK_FUNCTION( vExampleTask, pvParameters ) {
  for (;;) {
    vTaskAcquireNamedMutex(
      ((systemTaskParameters *)(pvParameters))->taskHandle, EX_MUTEX);

    // Critical section code goes here

    vTaskReleaseNamedMutex(
      ((systemTaskParameters *)(pvParameters))->taskHandle, EX_MUTEX);
  }
}
```

Figure 14. Using the multicore synchronisation API

```
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());
```

Figure 15. Kernel–level synchronisation in the memory manager.

As explained in Section 4.7, the kernel–level synchronisation acquires mutexes on behalf of processor cores (as opposed to tasks) and interrupts have to be actively disabled to obtain full mutual exclusion. The code in Figure 15 taken from the modified memory manager, demonstrates this. Notice 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 the function `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.

5.5. Reproducing the Results

A comprehensive 50-step guide to building and running our multicore-enabled FreeRTOS in a dual-core MicroBlaze setup is available in the associated MSc thesis [1]. The source code for the software we have developed is available online at `http://sourceforge.net/projects/freertosxcore`.

DOI: 10.1002/spe

Table I. Concurrency test results when measured using an external timer

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

6. EVALUATION

6.1. Demo Applications

Two simple demo applications have been created to exemplify the effectiveness of the FreeRTOS modifications. 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* consists of three tasks (two worker tasks and a *check* task):

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 simpler. It consists of three tasks as well, with two 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 and seconds.Three versions of the tests were used: SR or short run (1 million runs),¶ LR or long run (10 million runs) and VLR or very long run (100 million runs). An equivalent version of the conc test application has been created for the single–core version of FreeRTOS. Table I compares the execution time of the two versions. 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 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.

Applications have also been created to test the scheduler, as well as various API calls. These tests are explained in the associated MSc thesis [1]. Some optional elements of the FreeRTOS API have

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

Prepared using speauth.cls
DOI: 10.1002/spe

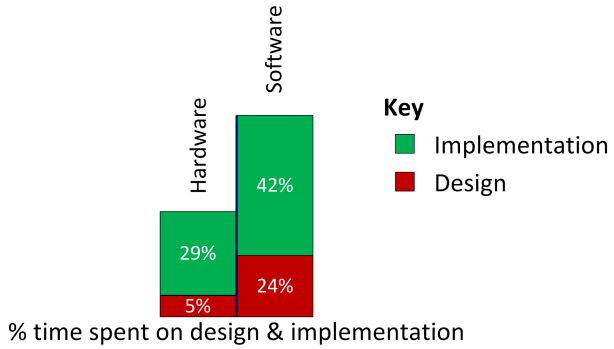

Figure 16. Comparison of time spent on hardware and software design/implementation

not been tested. The relevant code is located in `tasks.c` and is marked with comments reading "NOT FULLY MULTICORE TESTED".

6.2. *Development Time*

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.

About 64% of software development time was spent on the implementation, compared to about 85% of hardware development time. As displayed in Figure 16, 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, including a fully operational dual–core processor system with the necessary peripherals to implement and test multicore FreeRTOS. However, a bottleneck occurred in the hardware implementation due to a lack of familiarity with the tools and the available FPGA platforms.

6.3. *SMP Architecture*

Although the predicted benefits of using an SMP architecture were ultimately sound, some of these were not 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. This is due to the Xilinx tools requiring that all the MicroBlaze cores runs entirely separate software projects. 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.

6.4. *Concurrency*

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 also been applied 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.

DOI: 10.1002/spe

Considering performance optimisation has not been included within the scope of this paper, the execution time results are good. 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.

6.5. Mutual Exclusion

Alagarsamy's algorithm was successfully implemented to provide software synchronisation, and tests tentatively suggest that the implementation is thread–safe. 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.

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 $n - 1$ [10, p36]. 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 disassembled 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.

DOI: 10.1002/spe

6.6. Multicore Application Development

One of our aims has been to ensure that multicore application development is no more difficult than single-core development. For the application developer, there are only four small differences to be aware of: (a) an additional "CPU affinity" parameter when calling `xTaskCreate`; (b) the use of the `systemTaskParameters` struct to access application parameters passed in to a task (see Section 5.4); (c) the inclusion of a task handle parameter to some API functions, accessible via a pointer to a `systemTaskParameters` struct available to every task; and (d) the use of two functions for entering and exiting critical sections for use in ensuring thread safety, as required.

7. CONCLUSION AND FUTURE WORK

7.1. Conclusion

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 (SMP) 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 very attractive, 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 [15, p227]. 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

DOI: 10.1002/spe

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 application tasks and OS components alike.

7.2. Future Work

Proving the Software Synchronisation With access to more complete information about MicroBlaze out–of–order execution semantics and its `mbar` instruction for introducing memory barriers, 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.

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 overridden in the portable layer.

In order to determine the performance gap between software–based and hardware–level synchronisation mechanisms it would be very worthwhile to conduct an empirical comparison between the two, for example by comparing lock-acquisition delays in Alagarsamy's algorithm against those of the MicroBlaze mutex peripheral over a range of lock-intensive benchmarks. It would also be worthwhile to add other software–based synchronisation algorithms to the empirical comparison, for example the tournament algorithm mentioned in Section 2.2.

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.

Scheduler Optimisation As discussed in Section 6.4, it may be possible to improve the scheduler's run-time performance. 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.

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.

REFERENCES

DOI: 10.1002/spe

1. Mistry J. 2011. FreeRTOS and Multicore. MSc Thesis, Department of Computer Science, University of York.
2. Laplante PA. 2004, Real-time systems design and analysis. Wiley-IEEE.
3. Real Time Engineers Limited. August 2011. Introduction to FreeRTOS. http://www.freertos.org/.
4. ChibiOS/RT. August 2011. RTOS Concepts. http://www.chibios.org/.
5. Lomas N. August 2011. Dual-core smartphones: The next mobile arms race. http://www.silicon.com/technology/mobile/2011/01/12/dual-core-smartphones-the-next-mobile-arms-race-39746799/.
6. Andrews D, Bate I, Nolte T, Otero-Perez C, and Petters, SM. Impact of embedded systems evolution on RTOS use and design. In *Proceedings of the 1st Workshop on Operating System Platforms for Embedded Real-Time Applications*.
7. Chynoweth M and Lee MR. Intel Corporation. August 2011. Implementing Scalable Atomic Locks for Multi-Core Intel EM64T and IA32 Architectures. http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures/.
8. Xilinx Inc. 2011. XPS Mutex Documentation. http://www.xilinx.com/support/documentation/ip_documentation/xps_mutex.pdf.
9. Peterson GL. 1981. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, vol. 12, no. 3, pp. 115-116.
10. Alagarsamy K. 2005. A mutual exclusion algorithm with optimally bounded bypasses. *Information Processing Letters*, vol. 96, no. 1, pp. 36–40.
11. Hofri M. 1989. Proof of a mutual exclusion algorithm – a classic example. *ACM SIGOPS Operating Systems Review*, vol. 24, no. 1, pp. 18–22.
12. Real Time Engineers Limited. August 2011. Task Priorities. http://www.freertos.org/a00015.html#TaskPrior.
13. Agron J. August 2011. How to Create and Program Interrupt-Based Systems. https://wiki.ittc.ku.edu/ittc/images/d/df/Edk_interrupts.pdf.
14. Microsoft. August 2011. SetThreadAffinityMask Function. http://msdn.microsoft.com/en-us/library/ms686247%28v=vs.85%29.aspx
15. Xilinx Inc. 2011. MicroBlaze Processor Reference Guide.

DOI: 10.1002/spe