Implementation of Asymmetric Multiprocessing Support in a Real-Time Operating System

By

Islam Abdalla Elamin Elsheikh
17880

Final Report

Bachelor of Engineering (Hons)
(Electrical and Electronics)
September 2015

Supervisor: Lo Hai Hiung

CERTIFICATION OF APPROVAL

Implementation of Asymmetric Multiprocessing Support in a Real-Time Operating System

by

Islam Abdalla Elamin Elsheikh
17880

A project dissertation submitted to the
Electrical and Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(Electrical and Electronics)

Approved by,

____
Lo Hai Hiung

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

April 2016

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

ISLAM ABDALLA ELAMIN ELSHEIKH

ABSTRACT

The semiconductor industry can no longer afford to rely on decreasing the size of the die, and increasing the frequency of operation to achieve higher performance. An alternative that has been proven to increase performance is multiprocessing. Multiprocessing refers to the concept of running more than one application or task on more than one central processor. Multi-core processors are the main engine of multiprocessing. In asymmetric multiprocessing, each core in a multi-core systems is independent and has its own code that determines its execution. These cores must be able to communicate and synchronize access to resources.

In real-time applications, where the output of the system should be deterministic, and with the increasingly power-intensive embedded systems applications, asymmetric multiprocessing can provide a balance between increasing the system performance and maintaining its determinability.

This project documents the process and provides the results of implementing asymmetric multiprocessing support in FreeRTOS, a popular real-time operating system. The platform of choice is Altera Nios II soft-core processor running on a Field Programmable Gate Array.

ACKNOWLEDGEMENTS

I would like to address my utmost gratitude to my lecturer and supervisor, Lo Hai Hiung, without whom this project would not have seen the light of the day. Since the project was a mere idea, his support and guidance have always shed the light on the path to be taken, and helped me to overcome the difficulties and obstacles I came to face.

I am also forever thankful to my parents for their love and support, and my family and friends for always being by my side. Finally, I would also like to extend my warmest regards to Universiti Teknologi PETRONAS for providing me with all the tools and help I needed during this project.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS AND NOMENCLATURES

AMP	Asymmetric Multiprocessing
SMP	Symmetric Multiprocessing
OS	Operating System
RTOS	Real-Time Operating System
IP	Intellectual Property
FPGA	Field Programmable Gate Array
RAM	Random Access Memory
BRAM	Block Random Access Memory
SDRAM	Synchronous dynamic random-access memory
JTAG	Joint Test Action Group
UART	Universal Asynchronous Receiver/Transmitter
IDE	Integrated Development Environment

CHAPTER 1

INTRODUCTION

The aim of this project is to extend the open-source FreeRTOS kernel capabilities to support Asymmetric Multiprocessing (AMP). The hardware platform selected to run the kernel is Altera Nios II soft processor.

1.1 Background Of Study

A real time system is a system whose output is deterministic. That is, the system must respond within a specified time, known as the deadline. Normally, embedded systems have real-time requirements. The “hardness” of a real-time system depends on the severity of the consequences of missing a deadline. Hence, a hard real-time system must absolutely meet its deadline always, whereas a soft real-time system should generally meet the deadline, however, a deadline miss is not as severe as it is in hard real-time systems [1]. Due to their deterministic respond time, hard real-time systems are used for critical applications such as aerospace and missile control applications.

A real-time operating system, or RTOS is an operating system with real-time capabilities. The RTOS acts as a bridge between the hardware and the user applications, or threads of execution. RTOSes in general require multitasking, the ability to assign priority to threads, and having an enough number of interrupt levels [2].

FreeRTOS is one of the most widely used RTOSes [3] and currently it has more than 35 microcontroller ports. FreeRTOS was selected to be used in this study. In addition to being free and open source, FreeRTOS has a very small footprint and it is written almost entirely in C, except for few assembly lines where needed.

Despite its wide use in the industry, FreeRTOS is built from the ground up to run on a single core. A microprocessor contains a unit, called a “core” that can execute instructions sequentially. Traditionally, most embedded systems used single-core processors [4]. A multi-core system can execute code instructions in parallel. For an example, a dual-core processor can run a piece of code in half the time that a single-core requires to execute the same code. With the increase in complexity and amount of processing power required for embedded systems, it is not uncommon to find homogeneous and heterogeneous multicore embedded systems, and as such, RTOSes need to be able to take advantage of these cores.

The hardware platform selected for this project is Altera Nios II processor. Nios II is a 32-bit RISC architecture soft core that can be implemented on Altera FPGAs (Field Programmable Gate Array).

1.2 Problem Statement

To increase the system performance, the semiconductor industry has traditionally relied on making the size of the die smaller and increasing its operating frequency [1]. However, the limitation of using a single processor was soon realized, with the greatest limitation being heat dissipation when decreasing the die size and increasing the frequency. Due to heat dissipation problem and the increasing complexity of chip architecture, multi-core systems were inevitable.

While symmetric multiprocessing can improve the performance of the system, it can only be implemented on homogeneous cores. Asymmetric multiprocessing on the other hand can be implemented on heterogeneous and homogeneous cores alike. Another critical problem of SMP is that it is not deterministic as system functions and threads are highly dependent on load distribution [5]. This will limit the applications of SMP in embedded systems.

Therefore, the need for an AMP implementation on RTOS arises to overcome the problems and limitations of a single core, and the drawbacks of SMP in embedded systems. This paper aims to develop an AMP setup of FreeRTOS and investigate its benefits in embedded systems.

1.3 Objectives And Scope Of Study

The principal objective of this paper is to develop an AMP implementation of the FreeRTOS kernel on a dual-core Nios II processor. The detailed objectives of this study are as follows.

- Develop and implement a dual-core Nios II processor on a simulation platform.
- To set up an instance of the FreeRTOS kernel on one core of the Nios II dual-core processor.
- To provide an inter-core communication method for managing synchronization routines between the two cores and allowing the master core to control the slave core execution.

Chapter 2 in this project is the literature review where previous work has been discussed and analyzed. After that is the methodology in chapter 3. This chapter focuses on the implementation of the project and presents the detailed steps undertaken to complete the project. The results of the project are presented and discussed in chapter 4. Finally, the lasts chapter is the conclusion, where a summary of the project and recommendations for future work are given.

CHAPTER 2

LITERATURE REVIEW

This chapter analyzes the literature for this project. It has been divided into sections where each section discusses the part of the literature related to its topic. The chapter starts with a discussion about multicore processors and then multiprocessing in RTOS in details. Synchronization plays a major role in this project, and as such, synchronization mechanisms are analyzed in the following section. The basics of FreeRTOS is then discussed in the third chapter. Afterwards, the SMP FreeRTOS booting process in multi-core system is discussed.

2.1 Multicore Processors

In a single core processor, only one task can run at any given time, however, processors achieve multitasking by relying on the speed of execution being very fast. The “illusion” of multitasking can be achieved in single-core processors by switching between tasks rapidly. Multicore systems can, however, achieve real multitasking by having more than one core executing code at the same time. This can increase the performance of the system and reduce the time required to execute tasks.

2.1.1 Multiprocessing in RTOS

Multiprocessing on multicore systems can be either Symmetric Multiprocessing (SMP) or Asymmetric Multiprocessing (AMP). In SMP configuration, all processors will be connected to a shared memory where all cores execute the same code. One OS controls all the cores, schedules tasks and synchronizes resources access. In AMP, on the other hand, each core will have its own OS, memory space and it will execute a different code. Each OS is responsible for scheduling tasks for its respective core [1].

The most noticeable difference between SMP and AMP is memory footprint. Since

SMP requires only one instance of the OS, it takes less memory space than AMP where each core requires its own OS. Managing synchronization routines such as mutex, and spin-locks is also easier in SMP since one kernel controls the system resources. However, due to the shared queue that is used for scheduling tasks, scheduling overhead for accessing the OS code is much greater in SMP [6].

Other than requiring more memory space, managing synchronization routines is also more difficult in AMP as each core has its own code. On the other hand, AMP in real-time applications is more deterministic as studies have shown that algorithms for single-core scheduling (in contrast to global scheduling in SMP) can offer a better real-time performance [7]. Also, AMP allows for applying single-core scheduling techniques since each core schedules its own tasks.

Mistry in his thesis [8] develops a modified version of FreeRTOS to support SMP on dual processors. He used the MicroBlaze port to base his project upon, as the platform for implementation he is using is the MicroBlaze processor running on Xilinx FPGAs. His implementation allows the FreeRTOS scheduler to schedule tasks on the two core simultaneously. The main primary difference of this project, is that in Mistry's project, the kernels shares the same ready list of tasks. The fact that the protection of this list is implemented with mutual exclusion, which is a blocking synchronization, reduced the efficiency of the code due to the busy waiting of the processor [6], [9]. Thus, by increasing the number of cores, the time spent in the busy waiting state will increase as well, as each core will be waiting for the list to be released. This solution, on the other hand, makes synchronization easier. Suppose that there is a task that is busy waiting for a resource held by another task on the other core. Having the task in this state, makes synchronization easier as the task will be awaken by simply exiting the waiting loop. An additional feature that Mistry [8] implemented is core affinity. Core or processor affinity is the ability to bind tasks to a certain core to run on [10].

Another example of implementing multiprocessing on an RTOS is Bulusu's thesis [6]. The RTOS of choice he used is uC-OS-II and he used a dual-core Freescale MPC567K MPU as the platform. The major difference of Bulus's and Mistry's works is that the former is using an asymmetric multiprocessing setup, where two kernels are running; one on each core of the SoC. Every kernel is independent with its own code and data sections. Bulusu's implementation shares many similarities with this project, with the main difference being the RTOS of choice. Consequently, a different hardware platform is used in this paper.

2.2 Synchronization

In computer science, thread or process synchronization refers to the concept of allowing processes to communicate or shake hands in order to agree on a sequence of action [11]. In the case of sharing resources for example, synchronization insures that only one task can access this resource at a time. Techniques for synchronization among processes can be software-based or hardware-based [12]. Software-based techniques are implemented completely in code, and hence, it is hardware independent and can be used on any processor. Such techniques include software locks, ticket locks, and MCS (Mellor-Crummey and Scott) locks. Hardware locks on the other hand require the processor to have atomic operations such as test-and-set and compare-and-swap. A comparison of software and hardware synchronization mechanism [12] shows that hardware synchronization mechanisms are faster than software mechanisms where they require around one fourth (25%) only of the time required for software mechanisms.

For shared resources such as peripherals, or for synchronization of the tasks between the kernels, Bulusu [6] uses inter-core semaphores, which he calls Global Semaphores. He uses shared memory for inter-tasks communication. Basically, he implemented global queues that are protected by the global semaphores. In addition to semaphores, he also uses barrier primitives for synchronization. For an example to ensure that the kernels and the services are initialized correctly, a barrier primitive will ensure that all kernels reach a certain point before proceeding.

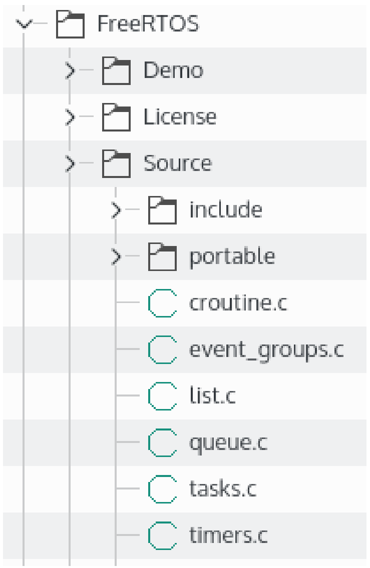
Priority ceiling protocol is a synchronization protocol that is used to prevent deadlock resulting from priority inversion [13]. Rajkumar proposes an extension for this protocol, namely Multiprocessor Priority Ceiling Protocol [14]. Bulusu [6] uses MPCP for his implementation. However, due to the fact that his project is AMP, and thus, kernels are not sharing tasks ready-list, he neglected the Priority Ceiling concept of the protocol. The Priority Ceiling is based on the concept that a task with the highest priority on one core, might not have the highest priority of the global view [6].

For synchronization, Mistry [8] opted for use of software based techniques. As he discussed, mutex peripherals are hardware specific, and in the case of his paper (and this project as well) it is configuration-specific, since it requires a hardware intellectual property (IP) to be configured into the FPGA. This IP peripheral will take an additional FPGA space.

2.3 FreeRTOS

2.3.1 Source Directory Structure

FreeRTOS is a widely used RTOS and has many applications [3]. To run an application demo, many files might be required, but the whole FreeRTOS kernel is contained within 3 files only. These files are list.c, tasks.c, and port.c [1], [15].



```
The demo application files

The core kernel files
The core kernel header files
Processor specific code (contains port.c)
```

Figure 1: FreeRTOS directory structure

One of FreeRTOS strong features is its separation between the hardware independent layer, and the portable layer [1]. The hardware-independent layer is common across all ports and it performs most of the system functions. The portable layer performs the hardware specific functions and exposes a set of functions that the hardware independent layer can interface with.

2.3.2 Tasks

Embedded applications can be structured as a set of “tasks” [16]. Each task can run without any dependency on another task to perform a specific function. On a single core, only one tasks can be running at any given moment. Other than being in the “running” state, tasks can also be ready, blocked, or suspended. A task in the running state is the task that is actually being executed [17].

A ready task is a task that is able to run, but it is not running because another task with an equal or higher priority is being executed. A blocked task is a task that cannot run because it is waiting for a temporal or external event. Finally, suspended tasks are similar to blocked tasks in that they cannot be scheduled, however, a task needs to be explicitly set in and out of the suspended state through vTaskSuspend() and xTaskResume() respectively. The figure to the right shows the valid state transitions.

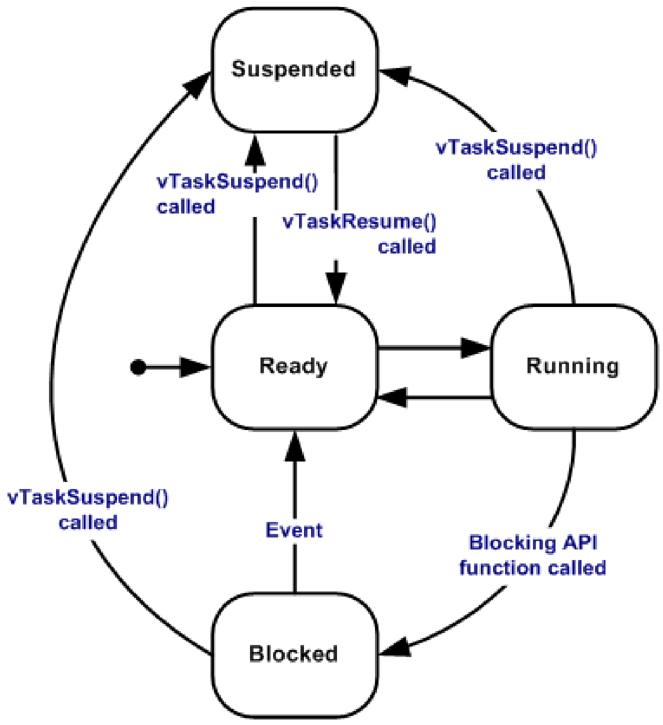
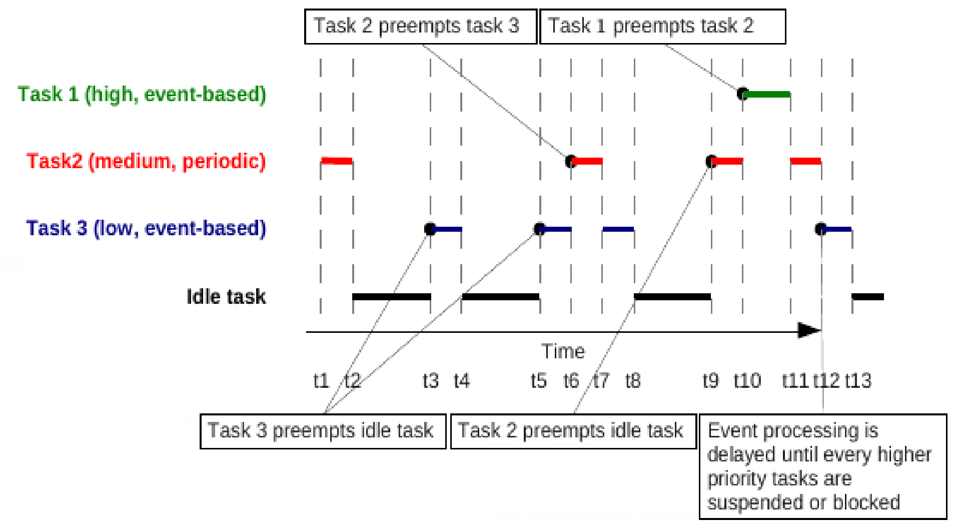


Figure 2: State transitions [17]

FreeRTOS uses preemtive multitasking where every task is given a processor time to run or suspended involuntary by the scheduler. Every task is given a priority when it is created [18]. At any given moment, the task with the highest priority (that is not blocked or suspended) will be running. The following figure explains how tasks are preemted by showing three tasks running.



Task 2 preempts task 3
Task 1 preempts task 2
Task 1 (high, event-based)
Task2 (medium, periodic)
Task 3 (low, event-based)
Idle task
Time
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13
Task 3 preempts idle task
Task 2 preempts idle task
Event processing is delayed until every higher priority tasks are suspended or blocked

Figure 3: FreeRTOS scheduling [6]

2.4 Multiprocessors Booting

Mistry's [1] setup consists of 2 processors connected to a shared RAM. Each processor however has its own small private block RAM (BRAM) memory. Upon booting up, the master core starts executing the code in its private BRAM. After initializing the necessary hardware, such as the interrupt controller and hardware timer, the master core starts executing the FreeRTOS scheduler code. At the same time, the slave core also starts executing the code in its private BRAM to initialize its hardware. The two cores use the shared RAM for communication. When the slave core finishes initializing the hardware, it waits for the master core to start the scheduler, as it is only then the slave can start executing the code of FreeRTOS in the shared RAM and can schedule tasks itself.

In his setup, interrupts are handled by the master core. When an interrupt occurs on the slave core, a small interrupt handler runs that simply passes the interrupt to the master core, which will be handled by the FreeRTOS interrupt handler. The slave then returns to its normal execution.

2.5 Altera's Nios II

Nios II is a soft core that can be programmed entirely on an FPGA. FPGAs are integrated circuits that contains memory and logic blocks. These logic blocks can be configured as digital systems [19]. Using Altera's tools and software, Nios II can be implemented in Altera's FPGAs.

Out of the box, FreeRTOS provides an official port and demo applications for Nios II implemented on a Altera Cyclone III FPGA with as specific design [20]. However, in this paper, FreeRTOS is run on DE2 board, which has Cyclone II on board, and the hardware design will be developed to suit the project needs. As such, some changes are required.

2.6 Summary Of The Literature Review

In this chapter, previous work that is related to this project has been reviewed. The chapter started by discussing multicore processors and multiprocessing in RTOSes, and discussed some implementations of SMP and AMP versions of popular RTOSes. Then synchronization techniques are compared, and it was shown that hardware-based techniques are much faster than hardware techniques. FreeRTOS basics was discussed and then the booting process of SMP RTOS was explored. Finally, important point about Altera Nios II processor were highlighted.

CHAPTER 3

METHODOLOGY

This chapter focuses the implementation methodology of the project. The first section discusses Altera's IP cores that are used as the platform for this project. Next section discusses the implementation and flow chart of the project, whereas the third chapter shows the detailed steps of implementation.

3.1 Altera's IP Cores

The project uses Alter's FPGA as the platform for implementing the system. Many soft cores are used to construct the system. The processor core used is Altera's Nios II/e processor. The processor is used without cache memory for simplicity. Two types of memories are used. The first one is the on chip memory, which is the memory inside the FPGA. The size of this memory is limited hence, the second memory is used which the SDRAM on the DE2 board. To interface the processor with this memory, an SDRAM Controller IP core is used.

For communication with the host computer, JTAG UART IP cores are used. These cores allow the processors to send and receive data serially to and from the computer. Interval Timers cores are connected to the processors to provide a constant “tick” that is required by the FreeRTOS. Finally, for synchronization, Altera Avalon Mutex core is used.

3.2 Project Implementation And Flow Chart

In this project, two processors are implemented on an FPGA. One processor is a master, and the other is a slave. The flow chart below shows the behavior of the two cores. Each processor is independent and runs its own code, however the slave will process requests coming from the master. As shown below, the slave is waiting in an infinite loop checking a request queue. When there is a request, the slave will process what is required, and then send the results to the master.

The master processor runs tasks that might needs the slave to do extra processing. If a task needs the slave, it will send a request, and waits for the results. If the results are not available, the tasks should yield the processor and go into the blocked state, so it will not consume time busy waiting. Whenever the tasks is run again, it will check for the results from the slave. If results are available, the task will run, otherwise, it continues to be blocked and other task will take turns.

If a task does not require the slave for processing, it will run normally on the master core.

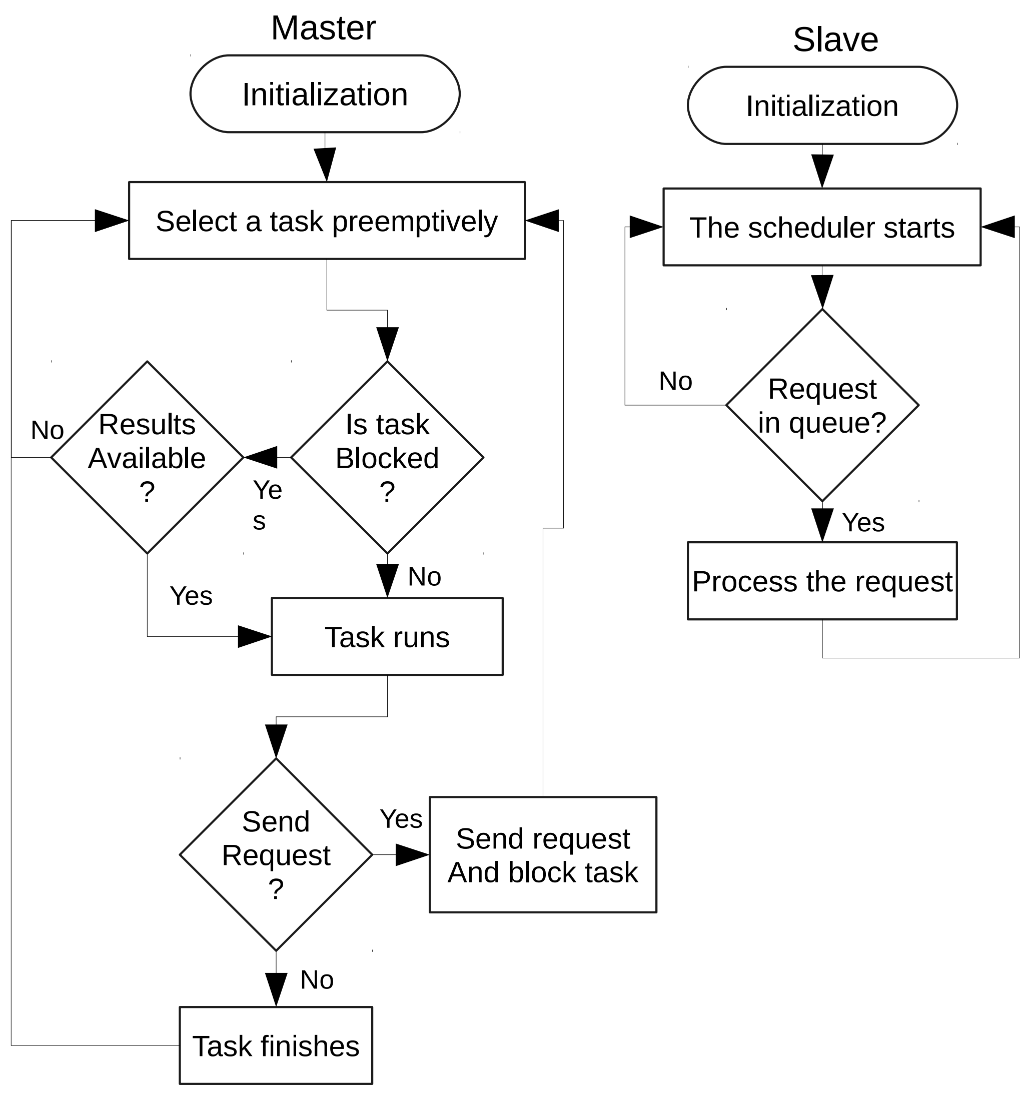


Figure 4: Flow chart of the system operation

Sending the request from the tasks on the master to the slave can be achieved via message passing. Sending the results from the slave to the master can be achieved via either message passing or shared memory. In this project, message passing is implemented with global queues.

The advantage of this configuration is that the master core in this case is running an RTOS, however, the slave processor can be running an RTOS or just a bare-metal application. Therefore, the slave core can be a more powerful core that is intended for heavy-processing applications.

Another configuration has been developed for the project where the slave core runs FreeRTOS as well. This method provides many benefits as discussed in the next chapter.

3.3 Research Methodology

This section details the methodology for implementing this project. The general framework for implementation includes first understanding the project background. Then, the tools used for the project including the FreeRTOS, and the Nios II processor are studied in details. Afterwards, different implementations for AMP has been explored to determine the most suitable approach for implementation. Before actual modifications were made to the FreeRTOS kernel, a detailed design for the project should be developed. Then the FreeRTOS kernel was modified and tested. At last, final results was then recorded and analyzed.

3.3.1 Preliminary Research Work

Before any progress was made, many aspects of the project background has been understood. This background includes the concept of real-time systems, SMP and AMP, heterogeneous and homogeneous cores, FPGAs and soft cores.

3.3.2 Learning the required tools

Many tools are required to accomplish this project. The first of which is the FreeRTOS. In addition to studying its documentation, a useful method for understanding the FreeRTOS operation was by using its POSIX/Linux simulator. This simulator allows the FreeRTOS to be run in a POSIX environment. This is useful as it allows for advanced debugging using GDB.

Other tools include the Nios II core, Altera Quartus II, Nios II Integrated Development Environment, and Altera Embedded Design Suite (EDS).

3.3.3 Exploring possible AMP implementations

Different options and algorithms have been developed for AMP. At this stage, a suitable implementation has been determined, such as which inter-core communication method is to be used.

3.3.4 Project design

In this stage, the project was designed in details and verified before development started. Making modifications at this step is easier than later. Once the design was approved, the project implementation started.

3.3.5 Implementation

At this stage FreeRTOS kernel was extended to implement AMP. The implementation step includes two parts. First, the hardware design has been developed. It determines how the two processors are connected, and what other IP cores are required for the design. The FPGA then was configured with the design. The second step was to modify and extend the kernel code of FreeRTOS. Based on the design developed in the previous chapter, the code has beem modified to achieve the project objectives.

3.3.6 Results analysis

The results of the project have been recorded and analyzed and different simple application has been developed to demonstrate the AMP implementation.

3.4 Key Milestones

The following steps have been identified as key milestones for the project.

For FYP1, the key milestones are as follows.

- KM1 - Implementing a single Nios II core on an FPGA (W5).
- KM2 - Submission of extended proposal (W6)
- KM3 - Running FreeRTOS on a single Nios II core (W9)
- KM4 - Implementing a dual-core Nios II processor on an FPGA (W12)

- KM5 - Submission of Interim Draft Report (W13)
- KM6 - Submission of Interim Report (W14)

For FYP2, the key milestones are as follows.

- KM1 - Running FreeRTOS on the dual-core Nios II processor (W1)
- KM2 - Finishing the AMP system design (W4)
- KM3 - Submission of Progress Report (W7)
- KM4 - Modifying the FreeRTOS code to support AMP (W9)
- KM5 - Pre-SEDEX (W10)
- KM6 - Submission of Draft Final Report (W11)
- KM7 - Recording the system results (W11)
- KM8 - Submission of Dissertation (soft bound) (W12)
- KM9 - Submission of Technical Paper (W12)
- KM10 - Viva (W13)
- KM11 - Submission of Project Dissertation (Hard Bound) (W15)

3.5 Gantt Chart

The Gantt chart for FYP1 and FYP2 are available in appendix 1.

CHAPTER 4

RESULTS AND DISCUSSION

This chapter details the results of the project until the time of writing. The project progress is on time with the Gantt chart developed in the methodology section. The key milestones for FYP1 have all been achieved, in addition to all the milestones of FYP2 until week 8. This chapter first discusses running FreeRTOS on a single Nios II processor. Next, it shows the dual processor setup running asymmetrically using the developed framework.

4.1 Running FreeRTOS On A Single Nios II Core

This section discusses the implementation of a Nios II core that is able to run an RTOS. Then it focuses on the modifications made to the FreeRTOS to be able to run on the Nios II processor.

4.1.1 Hardware implementation

When creating a Nios II processor, it needs to be connected to a clock source and to a memory to run the code stored in it. These are the very basic requirements for the system. A JTAG-UART module is usually added to send and receive data from the PC to the board. For the FreeRTOS to run, it requires a timer that is able to interrupt the processor at specified intervals of time. To implement this, Interval Timer module is also created and connected to the processor as an interrupt.

The system used to run the FreeRTOS consist of the Nios II processor, connected to the on-chip memory, with the address 0x00000000 for the reset vector, and 0x00000020 for the exception vector. A JTAG UART, timer are also added and connected appropriately to the processor.

The figure below shows the block diagram of the system and the memory map.

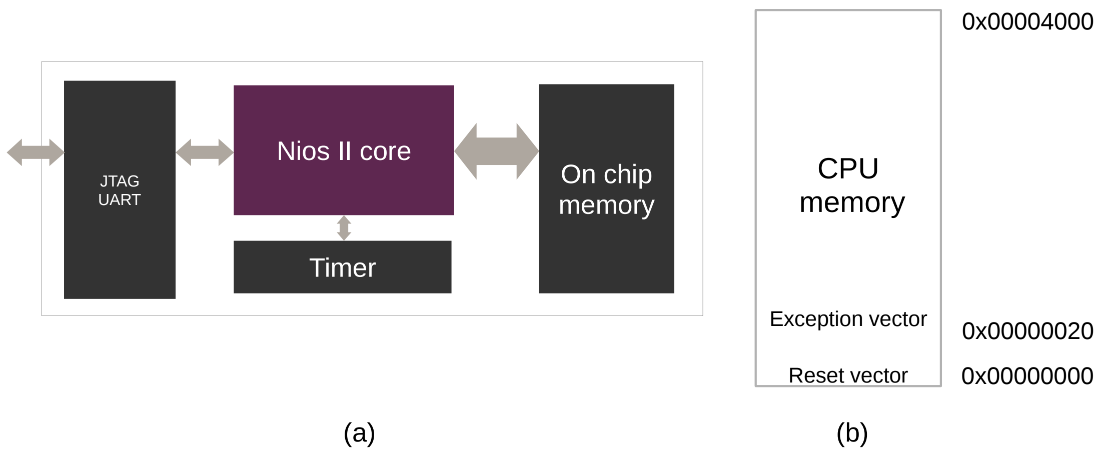


Figure 5: Single core Nios II processor: (a) Block diagram (b) Memory map

4.1.2 Software implementation

The official FreeRTOS Nios II port is used in the paper as the base for all the modifications. However, the hardware-specific portion of the port depends on the hardware system developed to run the FreeRTOS. As such, some modification were required to make it operate on the system described above.

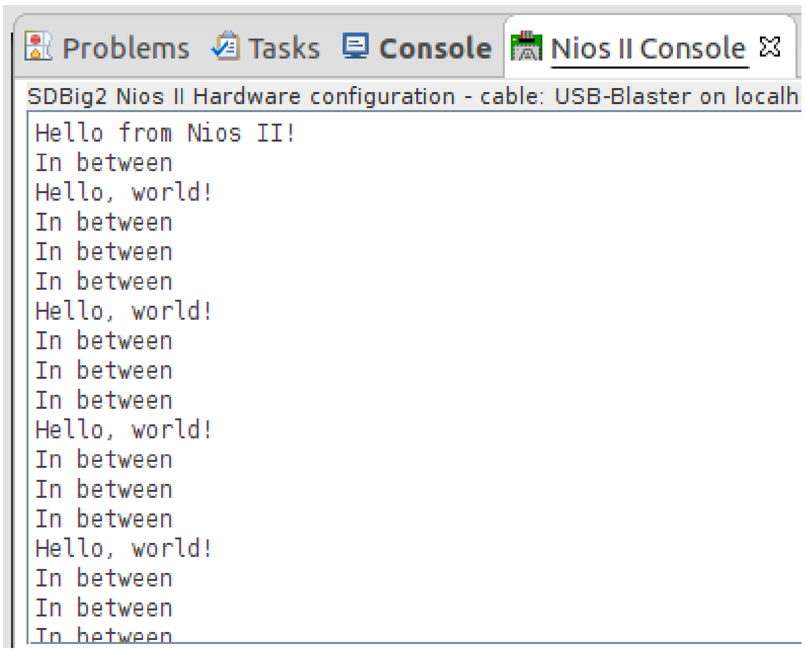
After importing all the necessary files, and modifying the hardware-specific portions, FreeRTOS was then run on the Nios II processor. To demonstrate the task switching of FreeRTOS, two tasks were created that use the JTAG module to print messages on the PC. Each task has a different delay time.

The first task is called “sayHello”, which prints “Hello, world!” and has a delay of 1000ms. Its code is as follows.

```
void sayHello( void *p){
while(1){
 alt_putstr("Hello, world!\n");
 vTaskDelay(1000);
 }
}
```

The second task is called “sayInBetween”, which prints “In between” and has a delay of 300 ms. Its code is as follows.

```
void sayInBetween (void *p){
while(1){
 alt_putstr("In between\n");
 vTaskDelay(300);
 }
}
```
Not surprisingly, the output shows that the second task sayInBetween prints its message 3 times every time the first task prints its message. The output on the PC when running the tasks is shown below.



SDBig2 Nios II Hardware configuration - cable: USB-Blaster on localh

```
Hello from Nios II!
In between
Hello, world!
In between
In between
In between
Hello, world!
In between
In between
In between
Hello, world!
In between
In between
In between
Hello, world!
In between
In between
In between
```

Figure 6: FreeRTOS tasks output

4.2 Running FreeRTOS On The Dual-core Nios II Processor

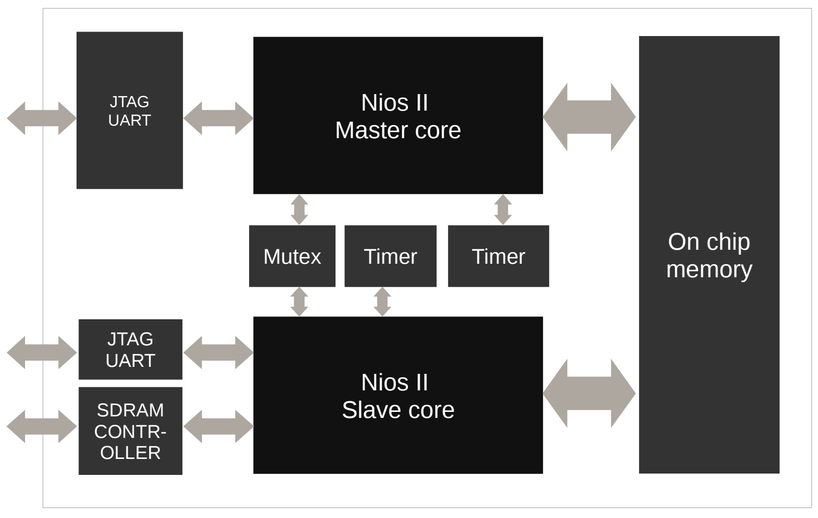
This section focuses on the connections of two Nios II cores. The software part will also be discussed briefly.

4.2.1 Hardware implementation

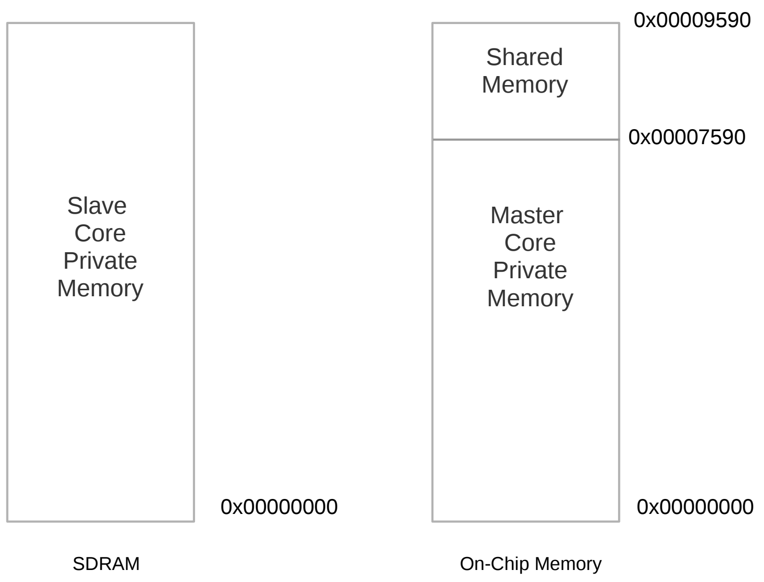
Each core of the Nios II processor requires a timer and a memory. However, they can be connected to the same physical memory, but each has its own section. However, due to the limited size of the on chip memory, the slave processor uses the SDRAM as its private memory. The master processor is connected to the on chip memory and has the addresses 0x00000000 and 0x00000020 for the reset and exception vectors respectively.

A part of the on chip memory is shared by the two processors. Both processors have their data lines connected to this memory so they can use it for communication. To synchronize the access to this shared memory, the two processors uses an Altera Avalon Mutex. Finally, each processor is connected to a JTAG UART IP core to allow it to communicate with the PC.

The figure below shows the block diagram of the dual core setup as well as their memory map.



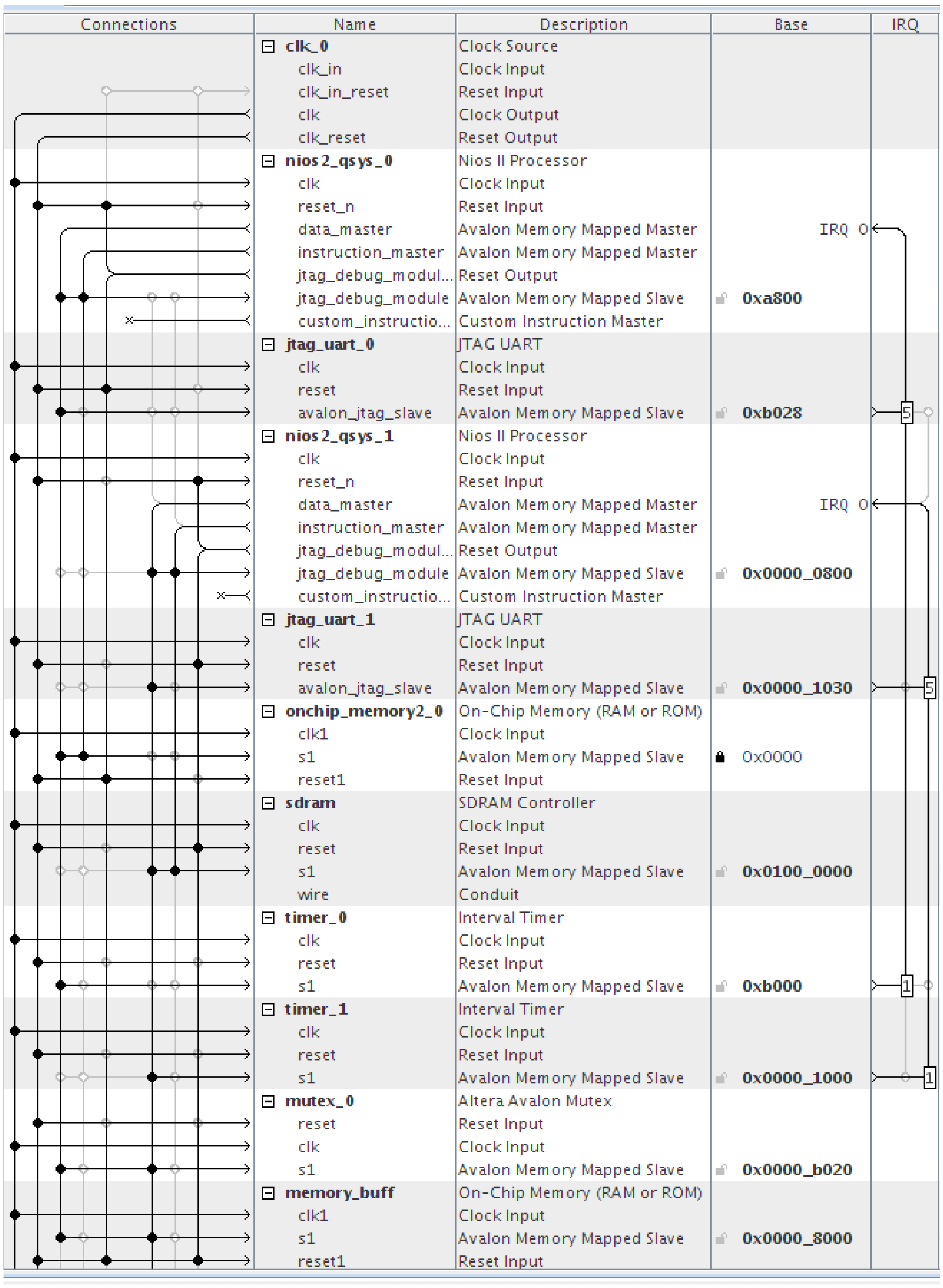
(a)



(b)

Figure 7: Dual core Nios II processor: (a) Block diagram (b) Memory map

The following figure shows the actual connections of the processors taken from Qsys.



Connections	Name	Description	Base	IRQ
	clk_0	Clock Source		
	clk_in	Clock Input		
	clk_in_reset	Reset Input		
	clk	Clock Output		
	clk_reset	Reset Output		
	nios2_qsys_0	Nios II Processor		
	clk	Clock Input		
	reset_n	Reset Input		
	data_master	Avalon Memory Mapped Master	IRQ 0	
	instruction_master	Avalon Memory Mapped Master		
	jtag_debug_modul...	Reset Output		
	jtag_debug_module	Avalon Memory Mapped Slave	**0xa800**	
	custom_instructio...	Custom Instruction Master		
	jtag_uart_0	JTAG UART		
	clk	Clock Input		
	reset	Reset Input		
	avalon_jtag_slave	Avalon Memory Mapped Slave	**0xb028**	5
	nios2_qsys_1	Nios II Processor		
	clk	Clock Input		
	reset_n	Reset Input		
	data_master	Avalon Memory Mapped Master	IRQ 0	
	instruction_master	Avalon Memory Mapped Master		
	jtag_debug_modul...	Reset Output		
	jtag_debug_module	Avalon Memory Mapped Slave	**0x0000_0800**	
	custom_instructio...	Custom Instruction Master		
	jtag_uart_1	JTAG UART		
	clk	Clock Input		
	reset	Reset Input		
	avalon_jtag_slave	Avalon Memory Mapped Slave	**0x0000_1030**	5
	onchip_memory2_0	On-Chip Memory (RAM or ROM)		
	clk1	Clock Input		
	s1	Avalon Memory Mapped Slave	0x0000	
	reset1	Reset Input		
	sdram	SDRAM Controller		
	clk	Clock Input		
	reset	Reset Input		
	s1	Avalon Memory Mapped Slave	**0x0100_0000**	
	wire	Conduit		
	timer_0	Interval Timer		
	clk	Clock Input		
	reset	Reset Input		
	s1	Avalon Memory Mapped Slave	**0xb000**	1
	timer_1	Interval Timer		
	clk	Clock Input		
	reset	Reset Input		
	s1	Avalon Memory Mapped Slave	**0x0000_1000**	1
	mutex_0	Altera Avalon Mutex		
	reset	Reset Input		
	clk	Clock Input		
	s1	Avalon Memory Mapped Slave	**0x0000_b020**	
	memory_buff	On-Chip Memory (RAM or ROM)		
	clk1	Clock Input		
	s1	Avalon Memory Mapped Slave	**0x0000_8000**	
	reset1	Reset Input		

Figure 8: Dual core system connections

4.2.2 Software implementation

The asymmetric multiprocessing framework supports two modes of operation. The first one consists of the master core running FreeRTOS whereas the slave core is running a custom kernel. The second mode consist of the two cores running FreeRTOS. These two modes are discussed below.

4.2.3 Running FreeRTOS on the master core

The asymmetric multiprocessing framework consists of two files, asym.c and asym.h. They define functions that the master core and the slave core can use to communicate and synchronize actions. Some functions should be called from the master processor only, whereas other should only be called from the slave. Instead of having two sets of files, one for each core, the same file can be compiled for the master and slave core as well. The only change that needs to be done is defining IS_MASTER or IS_SLAVE macros at the top of the file when compiling it for the master and for the slave respectively.

The framework defines a global queue structure for communications. A global queue (request queue) is created for holding the tasks that the master sends to the slave. The user is free to change the size of the queue to match its application as the only limitation is the memory size. The global queue resides in the shared memory and accessible by both the master and slave core.

Sending requests

Upon boot up, the master core calls two functions to instantiate the mutex and the global queue using the functions

```
xAsymMutexInit();

xAsymReqQueuInit();
```

Then the master core should continue to create the FreeRTOS tasks and finally starts the FreeRTOS scheduler.

Tasks running on the master core can send requests to the slave core using the function xAsymSendReq() which has the prototype

```
bool_t xAsymSendReq( int8_t xReqValue , bool_t xReturn );
```

The first parameter of the function xReqValue is the request index that the master requires the slave to perform. The other value xReturn can be either zero or one. This value specifies if the task should return immediately or should wait until the slave finishes servicing the request. This is important as the tasks might only need to send a request to the slave without the need for it to wait. An example might be if a task requires the slave to show some output such as displaying a message or playing a sound. The master task does not need to wait to the end of the execution of the slave task. For other application, and especially if the master task requires some output from the slave, the task might need to wait until the slave finishes servicing the request.

When a task waits for its request to be serviced by the slave, it goes to a blocked state until the slave finishes servicing the request. This is achieved by utilizing FreeRTOS binary semaphores. When the task sends the request, a binary semaphore is created, and the task tries to take it using FreeRTOS function xSemaphoreTake(). While in this state, FreeRTOS does not allocate any CPU time to the task and other tasks can run instead. Once the task request is serviced, the binary semaphore associated with this request is given using xSemaphoreGive(), and the task will be put in the ready-list to run.

If the request queue is full, the function does not return but wait until there is a location for the new request. The task returns True if the request was sent successfully, and false otherwise.

Example code

An example of a simple master code is as follows.

```c
#include "sys/alt_stdio.h"
#include "altera_avalon_mutex.h"
#include "system.h"
#include "stdlib.h"

// FreeRTOS
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

// asym includes
```

```c
#include "asym.h"

void masterTask( void *p){
	int i = 0;
	int task;
	while(i < 18){
		task = rand() % 3;
		taskENTER_CRITICAL();
		alt_printf("Sending Task %x\n", task );
		taskEXIT_CRITICAL();
		xAsymSendReq( task, 1  );
		vTaskDelay(20);
		i++;
	}
	alt_printf("Finished sending tasks\n");
	while(1);
}

void readyTask( void *p){
	while(1){
		taskENTER_CRITICAL();
		alt_printf("Ready is now running\n" );
		taskEXIT_CRITICAL();
		vTaskDelay(1000);
	}
}

int main()
{
	xAsymMutexInit();
	xAsymReqQueuInit();
	xTaskCreate(masterTask, "masterTask", 100, NULL, 2, NULL);
	xTaskCreate(readyTask, "readyTask", 100, NULL, 1, NULL);
	alt_putstr("Starting scheduler\n");
	vTaskStartScheduler();
	return 0;
}
```

4.2.4 Running a custom kernel on the slave core

As mentioned previously, the slave processor can run either a custom kernel or FreeRTOS. When running the custom kernel, the slave will be busy waiting for the master request.

Adding tasks to the scheduler

Upon boot up, the slave core instantiate the mutex and request queue using the functions

```c
	xAsymMutexInit();
	xAsymReqQueuInit();
```

Then the tasks that the slave should run are added using the function xAsymTaskCreate which has the following prototype

```c
bool_t xAsymTaskCreate( void (* pxTask )( void *p ) , xTaskIndex_t
xTaskIndex);
```

xAsymTaskCreate takes two parameters. The first parameter pxTask is a pointer to the function of the type

```
void taskName( void *p);
```

This pointer should points to the task that the slave execute. The second parameter is the task index. The master tasks send requests using this task index. The tasks that are defined in the slave are similar to the FreeRTOS tasks. They are normal C functions with the above prototype, and then added for scheduling using xAsymTaskCreate( ) instead of FreeRTOS xTaskCreate( ).

In this custom kernel, the slave create an array of function pointers, psTasks

```
PRIVILEGED_DATA static void (* pxTasks[ NUMBER_OF_TASKS ] )( void *p
);
```

This array holds the pointers to the functions that the slave can execute. The function xAsymCreateTask( ) adds the function pointer to this array.

Running the scheduler

After adding the tasks, the schedule is run by calling the function vAsymStartScheduler( ). This function never returns, and waits for the master to sends a request. When a request is sent, the function calls vAsymServeReq(xToServe), where xToServe is the index of the request that is to be serviced.

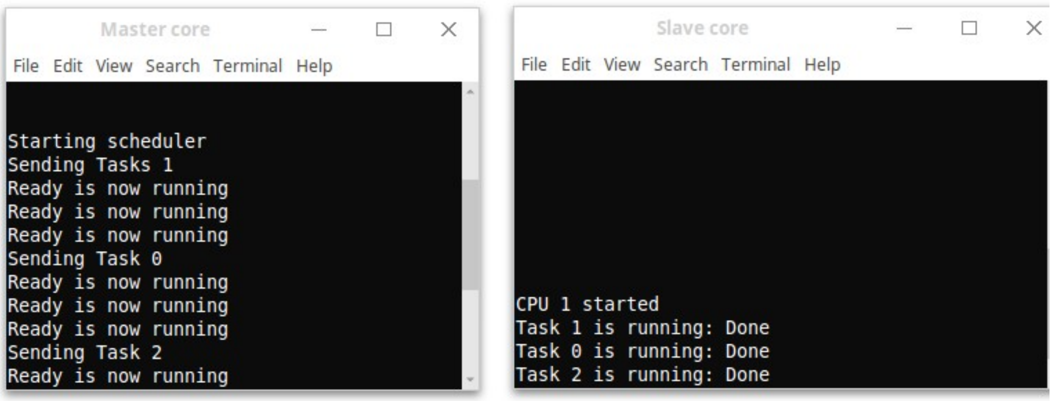
Example code

The following example shows a simple code running on the slave core.

```
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_mutex.h"
#include "system.h"
#include "unistd.h"

// Asym
#include "asym.h"

void xZerothTask( void * data){
        alt_printf("Task 0 is running: ");
        usleep(1500000);
        alt_printf("Done\n");
}

void xFirstTask( void * data){
        alt_printf("Task 1 is running: ");
        usleep(1500000);
        alt_printf("Done\n");
}
void xSecondTask( void * data){
        alt_printf("Task 2 is running: ");
        usleep(2500000);
        alt_printf("Done\n");
}

int main()
{
        xAsymMutexInit();
        xAsymReqQueuInit();
        alt_putstr("CPU 1 started\n");
        /* Adding tasks */
        xAsymTaskCreate(xZerothTask , Task0 );
        xAsymTaskCreate(xFirstTask , Task1 );
        xAsymTaskCreate(xSecondTask , Task2 );

        /* Starting the scheduler */
        vAsymStartScheduler();
        return 0;
}
```

Output of the code

After running the two examples of code on the master and the slave cores, here is the output on the PC terminal. The master core sends tasks to the slave core. When a task is sent, the “Ready” task runs indicating that the “Master” now is in a blocked state. The slave waits for requests from the master, and then runs the corresponding tasks.


Master core
File Edit View Search Terminal Help
Slave core
File Edit View Search Terminal Help

```
Starting scheduler
Sending Tasks 1
Ready is now running
Ready is now running
Ready is now running
Sending Task 0
Ready is now running
Ready is now running
Ready is now running
Sending Task 2
Ready is now running
```

```
CPU 1 started
Task 1 is running: Done
Task 0 is running: Done
Task 2 is running: Done
```

Figure 9: The slave processor running the custom kernel

4.2.5 Running FreeRTOS on the slave core

Instead of running a custom kernel, the slave can run FreeRTOS and service the master requests. This configuration allows the slave to be running its own tasks instead of busy waiting for the master requests. Another important advantage of this setup is that the requests are prioritized. That is, if the slave is servicing a low priority request, and a request with higher priority occurs, the slave will drop the low priority request and service the one with the highest priority. To achieve this, the framework utilizes FreeRTOS tasks priorities.

Handling requests

Now that FreeRTOS is running on the slave processor, some tasks might be required to run regardless of the presence of the master requests, while others should run only when requested. Tasks that service the master request call the function vAsymServeRequest( ). This function takes one parameter, which is the index of the requests that is to be serviced. When a task calls vAsymServeRequest function, it goes to the blocked state (similar to what happens on the master core) waiting for a request from the master. This is achieved by creating a binary semaphore which the task will take it. This semaphore is only given when the master sends a request with an index matching the index that the task sent when calling the vAsymServeRequest function.

The framework defines a queue xToServeQueue that contains the slave tasks that are waiting for requests from the master. When a task calls vAsymServeRequest(ucRequestedTask), it will be added to this queue with the requested task index ucRequestedTask, and a pointer to the semaphore that the task is trying to take. When a request occurs, it will be compared with the queue of tasks on the slave core that are awaiting. If task has a requested task index ucRequestedTask that matches the request index from the master, the task will be removed from xToServeQueue, and its semaphore will be given so the task can run.

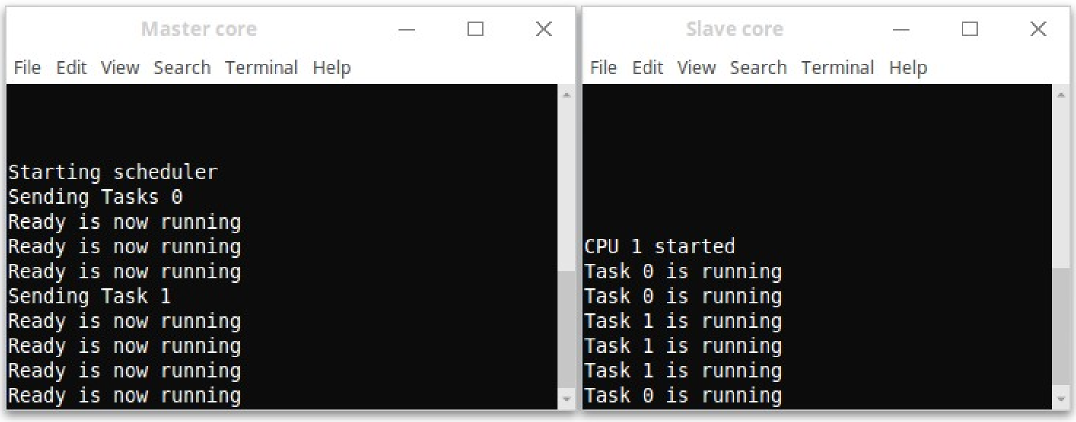
Example code

Below is an example where the slave is running FreeRTOS.

```
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_mutex.h"
#include "system.h"
#include "unistd.h"
// FreeRTOS
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
// Asym
#define IS_SLAVE
#include "asym.h"

void xZerothTask( void * data){
        int8_t ucIndex;
        while(1){
                vAsymServeRequest(Task0);
                for (ucIndex = 0; ucIndex < 3 ; ucIndex++ ){
                        taskENTER_CRITICAL();
                        alt_printf("Task 0 is running\n");
                        taskEXIT_CRITICAL();
                        vTaskDelay(2000);
                }
        }
}

void xFirstTask( void * data){
        int8_t ucIndex;
        while(1){
                vAsymServeRequest(Task1);
                for (ucIndex = 0; ucIndex < 3 ; ucIndex++ ){
                        taskENTER_CRITICAL();
                        alt_printf("Task 1 is running\n");
                        taskEXIT_CRITICAL();
                        vTaskDelay(2000);
                }
        }
}

void xSecondTask( void * data){
        int8_t ucIndex;
        while(1){
                vAsymServeRequest(Task2);
                for (ucIndex = 0; ucIndex < 3 ; ucIndex++ ){
                        taskENTER_CRITICAL();
                        alt_printf("Task 2 is running\n");
                        taskEXIT_CRITICAL();
                        vTaskDelay(2000);
                }
        }
}

int main()
{
        xAsymMutexInit();
        xAsymReqQueuInit();
        alt_putstr("CPU 1 started\n");
        xTaskCreate(xZerothTask , "Task0" , 100, NULL, 1, NULL);
        xTaskCreate(xFirstTask , "Task1" , 100, NULL, 2, NULL);
        xTaskCreate(xSecondTask , "Task2" , 100, NULL, 3, NULL);
        vTaskStartScheduler();
        return 0;
}
```

Output of the code

After running the two examples of code on the master and the slave cores, here is the output on the PC terminal. A slight modification was made to the master code so its tasks will not be blocked when sending requests.


Master core
File Edit View Search Terminal Help
Slave core
File Edit View Search Terminal Help

```
Starting scheduler
Sending Tasks 0
Ready is now running
Ready is now running
Ready is now running
Sending Task 1
Ready is now running
Ready is now running
Ready is now running
Ready is now running
```

```
CPU 1 started
Task 0 is running
Task 0 is running
Task 1 is running
Task 1 is running
Task 1 is running
Task 0 is running
```

Figure 10: FreeRTOS running on the slave core

When task 0 is sent, the slave start executing it. However, when task 1 is sent, the slave stops executing task 0 and start task 1 since task 1 has a higher priority. When task 1 finished, task 0 resumes execution.

4.3 Qualitative Analysis

The proposed AMP framework allows the master core to be running FreeRTOS in a multi-core system without compromising its determinability by having ready lists of tasks that are private for each core. All processing-intensive tasks can be offloaded to the slave core. Moreover, the AMP framework has a very small memory footprint of only 808 bytes after compilation. The global queue sizes can be defined by the user.

Despite these advantages, in this proposed design when the slave processor runs a custom kernel, the processor is idle when the queue is empty, hence, wasting a valuable CPU time. Also, the requests do not have priorities, hence, they are serviced on First Come First Served basis. The second mode addresses these problems by having FreeRTOS running on the slave core. The priorities of the requests are handled by FreeRTOS tasks priorities, and when the requests queue is empty the slave can perform lower priority tasks.

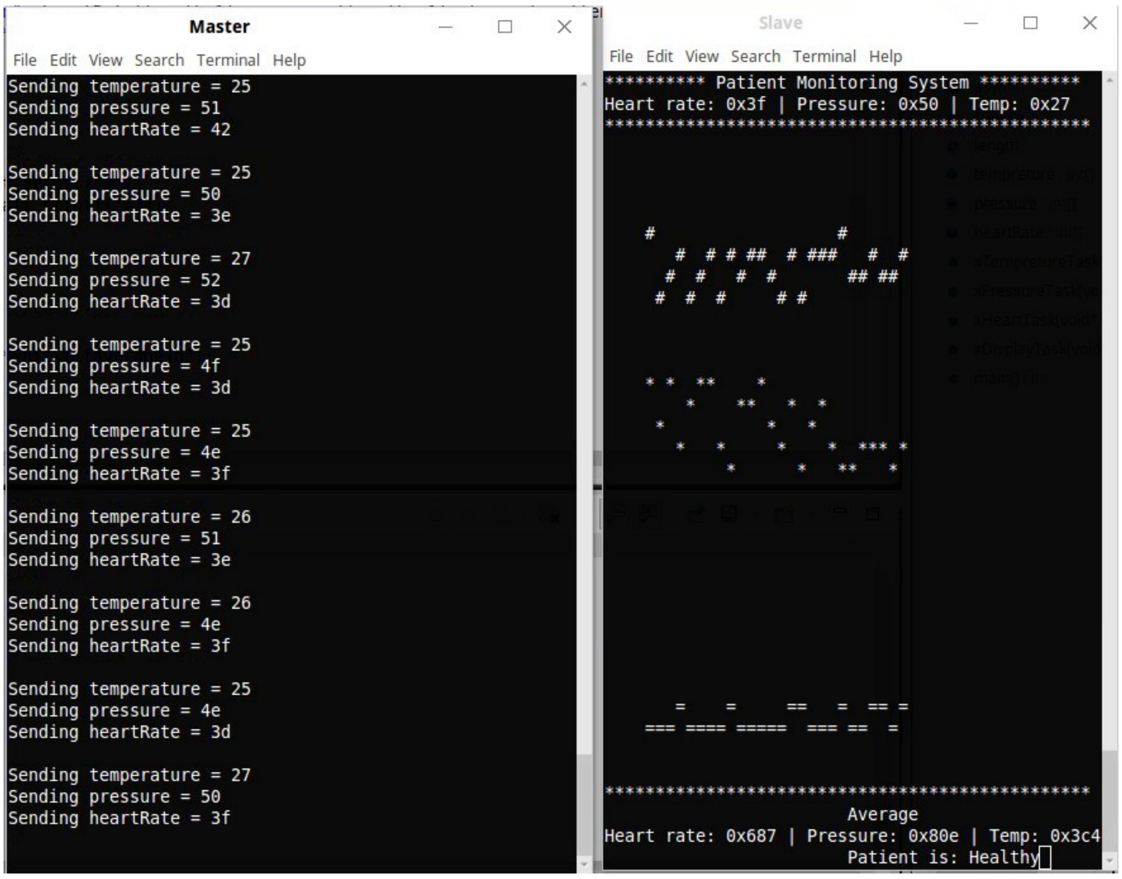
This proposed design can have applications in robotics or in systems where there is intensive processing while real-time performance is required. The slave core in this case can even be a Digital Signal Processor that processes data (e.g. image from a camera), and the master core can handle input and output devices.

4.4 Sample Application

To demonstrate the usage of the developed framework, a simple patient monitoring application has been developed. The application uses the functions and the features of the framework, and showcases how the project might be utilized for a real application.

RTOSes are commonly used in medical applications, such as in insulin pumps, cardio-vascular and hypertension monitors, and self-monitoring blood glucose devices [21]. With the increasing amount of data that needs processing, these applications require high processing power. To maintain the RTOS determinability, an AMP configuration can be used where one core runs the RTOS and the other cores can run custom kernels, different RTOSes, or a high level operating system such as Linux. In such case, the other core can even be a digital signal processor to provide much higher processing power for data analysis.

The sample application showcases such an example, whereas the master core can be connected to many sensors (heart rate, pressure, and temperature sensors) to monitor a patient health. The core can also be controlling other devices that are critical to the patient health. The master then sends all the data to the salve core where it can perform different functions on the data to analyze it and then conclude a certain course of action to be taken. In this application however, the slave only plots the data coming from the master core and calculate the average for each sensor's readings.


Master
File Edit View Search Terminal Help
Slave
File Edit View Search Terminal Help

```
Sending temperature = 25
Sending pressure = 51
Sending heartRate = 42

Sending temperature = 25
Sending pressure = 50
Sending heartRate = 3e

Sending temperature = 27
Sending pressure = 52
Sending heartRate = 3d

Sending temperature = 25
Sending pressure = 4f
Sending heartRate = 3d

Sending temperature = 25
Sending pressure = 4e
Sending heartRate = 3f

Sending temperature = 26
Sending pressure = 51
Sending heartRate = 3e

Sending temperature = 26
Sending pressure = 4e
Sending heartRate = 3f

Sending temperature = 25
Sending pressure = 4e
Sending heartRate = 3d

Sending temperature = 27
Sending pressure = 50
Sending heartRate = 3f
```

```
********** Patient Monitoring System **********
Heart rate: 0x3f | Pressure: 0x50 | Temp: 0x27
************************************************



    #                  #
       #  # # ##  # ###   #  #
      #  #   #  #       ## ##
     #  #  #     # #



    * *  **    *
        *    **   *  *
     *          *   *
       *   *     *    *  *** *
            *      *   **   *










       =    =     ==   =  == =
    === ==== =====  === ==  =



************************************************
                        Average
Heart rate: 0x687 | Pressure: 0x80e | Temp: 0x3c4
                        Patient is: Healthy
```

Figure 11: Output of the sample application

The figure above shows the output of the master and the slave core while executing the application.

In conclusion, this simple demonstration shows the ability of the framework to be applied in real life applications where high processing power is needed, while the system has hard real-time requirements.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

In addition to symmetric multiprocessing, asymmetric multiprocessing is an alternative approach for achieving multiprocessing. Admittedly, neither approach is without its drawbacks, however depending on the application, one might be the better option. In real-time applications, asymmetric multiprocessing seems promising as it can guarantee the determinability of the output.

This report documents the implementation of asymmetric multiprocessing on Nios II soft core. The project has been divided into subtasks that are achievable within the time frame. The project consists of 3 objectives. The first two objectives are related, whereas the first objective is to develop a dual-core Nios II processor, and the second objective is to run FreeRTOS on this setup. The third objective of the project is to develop an inter-core communication framework for managing synchronization between the two cores. All objectives have been successfully achieved. A system including dual-core Nios II processor, memory and peripherals was developed and tested. Some modification were made to FreeRTOS Nios II port to adapt it to run on the developed system. Then an asymmetric multiprocessing framework was developed that allows the two cores to communicate. A simple demonstration application has been developed as well to show how the projects can be implemented in real-life applications.

For future work, this project can be improved upon by building more complex applications that can demonstrate the use of this framework. Also, these applications, can be designed to run on two heterogeneous cores. Finally, for awaking the tasks that are blocked, the current implementation of the project uses software polling to detect when a request is received or serviced. An improvement to this project is to use interrupts instead of polling which will reduce the time that the processor wastes when polling.

REFERENCES

[1] James Mistry, Matthew Naylor, and Jim Woodcock, “Adapting freertos for multicores: an experience report,” 2013.

[2] “What is real-time operating system (RTOS)? - Definition from WhatIs.com,” *SearchDataCenter*. [Online]. Available: http://searchdatacenter.techtarget.com/definition/real-time-operating-system. [Accessed: 30-Oct-2015].

[3] UBM Tech, “2014 Embedded Market Study. Then, Now: What’s Next?,” 2014.

[4] M. Khalgui, *Embedded Computing Systems: Applications, Optimization, and Advanced Design*. IGI Global, 2013.

[5] Michael Christofferson, “Using an asymmetric multiprocessor model to build hybrid multicore designs,” *Embedded*. [Online]. Available: http://www.embedded.com/design/mcus-processors-and-socs/4006482/Using-an-asymmetric-multiprocessor-model-to-build-hybrid-multicore-designs. [Accessed: 30-Oct-2015].

[6] Girish Rao Bulusu, “Asymmetric Multiprocessing Real Time Operating System on Multicore Platforms,” Arizona State University, 2014.

[7] Yi Zhang, Nan Guan, Yanbin Xiao, and Wang Yi, “Implementation and empirical comparison of partitioning-based multi-core scheduling,” in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, 2011.

[8] J. Mistry, “FreeRTOS and Multicore,” University of York, 2011.

[9] A. Strake, “Locking in OS Kernels for SMP Systems,” 2006.

[10] TMurgent Technologies, “Processor Affinity.” 03-Nov-2003.

[11] “Synchronization (computer science) | Project Gutenberg Self-Publishing - eBooks | Read eBooks online.” [Online]. Available: http://self.gutenberg.org/article/whebn0004726017/synchronization%20(computer%20science). [Accessed: 16-Dec-2015].

[12] C. John, C.-C. Kuo, and R. Kuramkote, “A comparison of software and hardware synchronization mechanisms for distributed shared semory multiprocessors.” 24-Sep-1996.

[13] K. Renwick and B. Renwick, “How to use priority inheritance | Embedded.”

[Online]. Available: http://www.embedded.com/design/configurable-systems/4024970/How-to-use-priority-inheritance. [Accessed: 16-Dec-2015].

[14] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheri- tance Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1991.

[15] “FreeRTOS - Free RTOS Source Code Directory Structure.” [Online]. Available: http://www.freertos.org/a00017.html. [Accessed: 24-Dec-2015].

[16] “Tasks and Co-routines.” [Online]. Available: http://www.freertos.org/taskandcr.html. [Accessed: 24-Dec-2015].

[17] “FreeRTOS task states and state transitions described.” [Online]. Available: http://www.freertos.org/RTOS-task-states.html. [Accessed: 30-Oct-2015].

[18] “RTOS task priorities in FreeRTOS for pre-emptive and co-operative real time operation.” [Online]. Available: http://www.freertos.org/RTOS-task-priority.html. [Accessed: 24-Dec-2015].

[19] National Instruments, “FPGA Fundamentals - National Instruments.” [Online]. Available: http://www.ni.com/white-paper/6983/en/. [Accessed: 30-Oct-2015].

[20] “FreeRTOS Nios II Port and Demo.” [Online]. Available: http://www.freertos.org/FreeRTOS-Nios2.html. [Accessed: 17-Dec-2015].

[21] “Medical Market,” *High Integrity Systems*, 30-Oct-2015. [Online]. Available: https://www.highintegritysystems.com/embedded-rtos-for-medical-devices/. [Accessed: 10-Apr-2016].

APPENDIX A

Gantt Chart for FYP1.

	Activities/ Milestone	Week No.													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	Selection of Project Topic	▒	▒												
2	Preliminary Research Work		▒	▒	▒	▒									
3	Implementing a single Nios II core on an FPGA					■									
4	Submission of extended proposal						■								
5	Understanding FreeRTOS kernel						▒	▒	▒	▒					
6	Running FreeRTOS on a single Nios II core									■					
7	Understanding the Nios II ports and operation										▒	▒			
8	Designing a dual-core Nios II processor											▒	▒		
9	Implementing a dual-core Nios II processor on an FPGA.												■		
10	Submission of Interim Draft Report													■	
11	Submission of Interim Report														■

■ Milestone

▨ Process

Gantt Chart for FYP2.

	Activities/ Milestone	Week No.														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Running FreeRTOS on the dual-core Nios II processor	■														
2	Designing the AMP system		▨	▨	▨											
3	Finishing the AMP system design				■											
4	Submission of Progress Report							■								
5	Modifying the FreeRTOS code to support AMP					▨	▨	▨	▨	▨						
6	FreeRTOS supports AMP									■						
7	Testing the system									▨	▨	▨				
8	Pre-SEDEX										■					
9	Submission of Draft Final Report											■				
10	Recording the system results											■				
11	Submission of Dissertation (soft bound)												■			

12	Submission of Technical Paper												■			
13	Viva													■		
14	Submission of Project Dissertation (Hard Bound)															■

■	Milestone
▨	Process