Design and Implementation of Multi-core Support for an Embedded Real-time Operating System for Space Applications

Master of Science Thesis

KTH Information and Communication Technology

KTH ROYAL INSTITUTE OF TECHNOLOGY

Author:	Wei Zhang, KTH, Sweden
Supervisor:	Ting Peng, DLR, Germany
Examiner:	Assoc. Prof. Ingo Sander, KTH, Sweden

Abstract

Nowadays, multi-core processors are widely used in embedded applications due to the advantages of higher performance and lower power consumption. However, the complexity of multi-core architectures makes it a considerably challenging task to extend a single-core version of a real-time operating system to support multi-core platform.

This thesis documents the process of design and implementation of a multi-core version of RODOS - an embedded real-time operating system developed by German Aerospace Center and the University of Würzburg - on a dual-core platform. Two possible models are proposed: Symmetric Multiprocessing and Asymmetric Multiprocessing. In order to prevent the collision of the global components initialization, a new multi-core boot loader is created to allow that each core boots up in a proper manner. A working version of multi-core RODOS is implemented that has an ability to run tasks on a multi-core platform. Several test cases are applied and verified that the performance on the multi-core version of RODOS achieves around 180% improved than the same tasks running on the original RODOS. Deadlock free communication and synchronization APIs are provided to let parallel applications share data and messages in a safe manner.

Key words: **real-time** operating system, multi-core architecture, embedded system

Acknowledgment

This thesis is dedicated to my parents whose support and help over the years let me study abroad and pursue my dream.

I am sincerely thankful to my supervisor Ting Peng, who guided me to understand how the RODOS real-time operating system works. She also gave me some invaluable ideas on the implementation of multi-core version of RODOS.

I wish to thank Daniel Lüdtke, for his encouragement and kept the progress of the project, as well as provided some feedback regarding the synchronization and communication mechanisms.

I am supremely grateful to my examiner Ingo Sander, for his continuous feedback in writing the thesis, without his detailed comments, this thesis was unable to be finished in time.

Last but not least, I wish to thank Dr.Andreas Gerndt and all scientists and researchers in the Simulation and Software Technology Department at German Aerospace Center in Braunschweig, Germany, for allowing me to conduct my master thesis under such excellent environment.

Braunschweig, May 27, 2015

Contents

List of Figures

Listings

List of Abbreviations

BSP	Board Support Package
CPU	Central Processing Unit
DLR	Deutsches Zentrum fÃijr Luft und Raumfahrt
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
FSBL	First Stage Boot Loader
ICD	Interrupt Control Distributor
IDE	Integrated Development Environment
IRQ	Interrupt Request
JTAG	Joint Test Action Group
MMU	Memory Management Unit
OBC-NG	On Board Computer - Next Generation
OS	Operating System
PL	Programmable Logic
PPI	Private Peripheral Interrupt
PS	Processing System
RAM	Random Access Memory
RODOS	Real-time Onboard Dependable Operating System
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SCU	Snoop Control Unit
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on Chip
SRAM	Static Random Access Memory
SSBL	Second Stage Boot Loader
TCB	Task Control Block
UART	Universal Asynchronous Receiver/Transmitter
WCET	Worst-Case Execution Time

1

Chapter 1

Introduction

1.1 Motivation

An embedded system is a computer system that is specially designed for particular tasks, as the name indicated, it is embedded as a part of a complete device. Nowadays, embedded system devices are widely used in commercial electronics, communication devices, industrial machines, automobiles, medical equipment, avionics, etc. A typical embedded system consists of four parts (illustrated in Figure 1.1): embedded processing unit, hardware peripherals, embedded operating system and application software.

Some unique properties offered by embedded systems when compared with general purpose computers are:

- Limited resources: Normally, the hardware running an embedded application is very limited in computing resources such as RAM, ROM, and Flash memory. The design of embedded hardware tends to be very specific, which means that these systems are designed for the specific tasks and applications in order to get advantages of the limited resources.
- Real-time constraints: An embedded system always behaves in a deterministic manner [1, p. 2]. Such examples include airbag control systems and networked multimedia systems. Compared to the so-called hard real-time system, like airbag system, whose tasks are guaranteed to meet their deadlines, the soft real-time systems are more common in our daily life, the time limits are much weaker, and the consequence of deadline missing is not so severe as the hard real-time system's [1, p. 6-7].

As an essential element of embedded systems, an embedded operating system is designed to be compact, efficient at resource usage and reliable [2]. Many functions that existed in the general-purpose operating system are not necessary in the embedded operating system since embedded operating system only need to take care of some specialized applications. In order to fully use limited resources and maximize

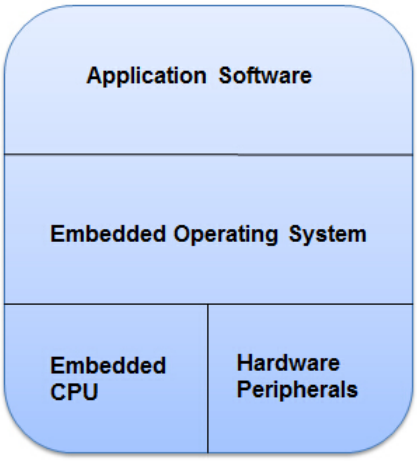


Figure 1.1: Embedded system structure

the responsiveness of the whole system, an embedded operating system is commonly implemented in assembly language or C language [2].

An embedded operating system is usually referred as a real-time operating system, because it not only has to perform a critical operation for a limited period of time, for instance, interrupt handling, but also needs to prioritize tasks to let them meet their deadlines.

This project idea was first proposed by Simulation and Software Technology Department of German Aerospace Center (DLR) in Braunschweig, Germany. This project work is a part of project OBC-NG (On-Board-Computer-Next Generation) [3], which is currently being conducted at DLR. The motivation to extend RODOS real-time operating system to multi-core support is not to change its deterministic property, but to meet the trend of current embedded system development - being able to run tasks concurrently [4, p, 3]. Multi-core platform provides an ideal environment for parallel execution. However, in reality, the improvement in performance gained by switching to multi-core platform mainly depends on the software construction and implementation, only specially designed software can take the most advantages of multi-core architecture.

1.2 Outline

Chapter 2 presents the necessary backgrounds and literature reviews to realize the underlying problems of multi-core architectures and multi-core versions of real-time operating system. Chapter 3 discusses the project requirements. Chapter 4 describes the system design and suggests two available solutions: symmetric multiprocessing model and asymmetric multiprocessing model. Chapter 5 discusses the implementation of these two solutions. Chapter 6 tests and evaluates the outcome of modified RODOS, verifies the requirements proposed in chapter 3. Chapter 8 summarizes the whole thesis, as well as suggests several points of future improvement.

Chapter 2

2 Fundamentals and Related Work

2.1 The Impact of Multi-core

A processor core is a unit inside the CPU that executes instructions. Tasks, consisting of multiple consecutive instructions, are performed one after the other in a single-core environment. An operating system, which has the ability to switch between tasks, make it possible for a single-core processor to execute multiple tasks. Because of the frequency at which tasks are switched, the operating system gives an impression that a single-core processor runs multiple tasks at the same time.

However, the situation will become worse if more tasks need to share the processor. For instance, if we have two tasks, the processor only needs to divide its time by two. Thus, each one gets 50% of the total execution time. However, if ten tasks to be run on a single-core processor, each task shares 10% of the execution time, and apparently, takes a long time to finish. So, in order to complete all tasks as soon as possible, the CPU's performance needs to be improved.

$$CPU\_Performance = Clock\_Frequency * IPC$$

The equation indicates that the processor's performance depends on two factors: clock frequency and IPC (Instruction Per Clock) [5, p, 42]. Before 2000, almost all the CPU manufacturers attempted to increase the performance by raising the CPU clock frequency. From the first electromechanical computer Z3[1] to the latest Intel core i7 processor[2], clock frequency rose significantly due to the development of semiconductor technology.

However, in the early 2000 [6], semiconductor scientists and researchers found that they could no longer achieve a faster single-core processors by the way they did in the past. One of the obstacles is the power wall [7]:

$$CPU\_Power = C * f * V^2$$

---

[1] The clock frequency of the Z3 is 5.3Hz

[2] The clock frequency of the i7 processor is up to 3.3GHz

Where C represents the value of capacitance, f represents the value of system clock frequency, and V represents the value of system voltage. Based on the current semiconductor technology, a higher frequency normally relies on the higher voltage. The CPU power consumption will be dramatically increased by a factor 8 when doubling the voltage, which is unacceptable due to the fact that the chip will become too hot. The power wall, as well as the memory wall[3] has forced the chip vendors to shift their attentions from frequency increment to IPC increase.

During the last two decades, several new technologies have been invented to achieve a higher IPC, such as pipelining, Instruction-level parallelism (ILP) [5, p, 66], Thread-level parallelism (TLP) [5, p, 172], etc.

Another much better solution is to use a multi-core processor. Multi-core processors have an ability to execute more than one instruction at one time. Consider we have two independent tasks which each of them needs an equal execution time, the dual-core processor can save up to 50% of the execution time than that by the single-core processor, since the dual-core processor naturally has an ability to run two tasks in parallel. Another significant advantage is power saving [8, p, 11-13]. According to the power theory above, the quantity of power consumption is closely related to the number of the clock frequency. It is indeed not a good idea to increase the system clock frequency to achieve a better performance, because power consumption is a very sensitive factor for the modern embedded system development. Thus, using the dual-core system with the original clock frequency, the overall performance doubled by handling more work in parallel, and the power consumption only goes up by a factor of 2, one-quarter of single-core's solution. Thus, in terms of power consumption, a multi-core solution is more power efficient.

2.1.1 Multicore Processor Architecture

There exist two types of multi-core architectures: symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP). As their names indicate, the differences are quite straightforward.

In SMP systems, each core has the same hardware architecture. They share the main memory space, have full access to all I/O devices. Each core, although, has its private resources, such as L1-cache memory, private timers and memory management unit. The whole system is controlled by a single operating system instance which treats all cores equally. Due to the shared memory architecture, the operating system has to provide some common interfaces for all cores to access the main memory, as well as some communication mechanisms for task synchronization [10].

Typically, SMP solutions are employed when an embedded application simply needs more CPU power to manage its workload, in much the way that multi-core proces-

---

[3] The increasing gap between processor speed and memory speeds

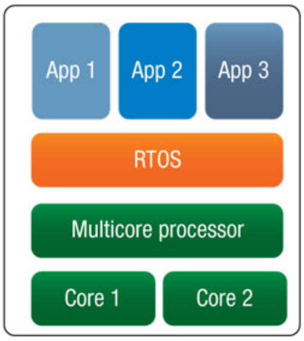


Figure 2.1: SMP Architecture
[9]

sors are used in desktop computers, as well as modern mobile phones and tablets [11].

In AMP systems, each core may have different hardware architectures, and each core may deploy different operating system. Each of them has memory space. They are to a large extent independent of each other, although they are physically connected. In AMP systems, not all cores are treated equally, for example, a system might only allow (either on the hardware or operating system level) one core to execute the operating system code or might only allow one core to perform I/O operations [10].

AMP models are used when different hardware architectures or multiple operating systems are needed for some specific tasks, like DSP and MCU integrated on one chip.

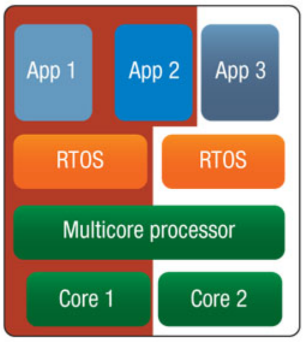


Figure 2.2: AMP Architecture
[9]

2.2 ARM Cortex-A9 Processor

The ARM Cortex-A9 processor is a power-efficient and high-performance processor that is widely used in low power and thermally constrained cost-sensitive embedded devices [12].

The Cortex-A9 processor provides an increment in performance of greater than 50% compared to the old Cortex-A8 processor [12]. The Cortex-A9 processor can be configured with up to four cores. Configurability and flexibility make the Cortex-A9 processor suitable for a broad range of applications and markets.

The Cortex-A9 processor has the following features [12]:

- Out-of-order speculating pipeline.
- 16, 32 or 64KB four-way set associative L1-caches.
- Real-time priority controlled preemptive multithreading.
- Floating-point unit.
- NEON technology for multi-media and SIMD processing.
- Available as a speed or power optimized hard macro implementation.

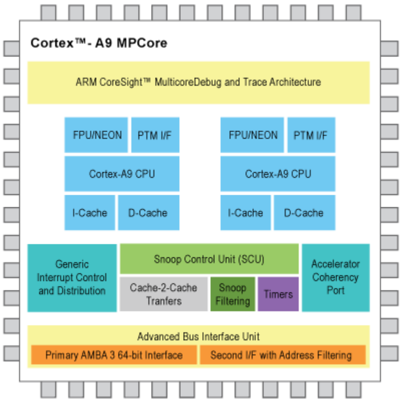

Cortex™- A9 MPCore
ARM CoreSight™ MulticoreDebug and Trace Architecture
FPU/NEON
PTM I/F
FPU/NEON
PTM I/F
Cortex-A9 CPU
Cortex-A9 CPU
I-Cache
D-Cache
I-Cache
D-Cache
Generic Interrupt Control and Distribution
Snoop Control Unit (SCU)
Cache-2-Cache Tranfers
Snoop Filtering
Timers
Accelerator Coherency Port
Advanced Bus Interface Unit
Primary AMBA 3 64-bit Interface
Second I/F with Address Filtering

Figure 2.3: ARM Cortex-A9 Processor
[13]

2.2.1 Xilinx MicroZed Evaluation Kit

The Xilinx MicroZed evaluation kit is used as a target hardware platform for this thesis project. MicroZed is a low-cost development board based on the Xilinx Zynq-7000 All Programmable SoC [14] which integrated an ARM cortex-A9 dual-core processor. Each processor implements two separate 32 KB L1-caches for instruction and data. Additionally, there is a shared unified 512 KB level-two (L2) cache for instruction and data. In parallel to the L2-cache, there is a 256 KB on-chip memory (OCM) module that provides a low-latency memory [14].

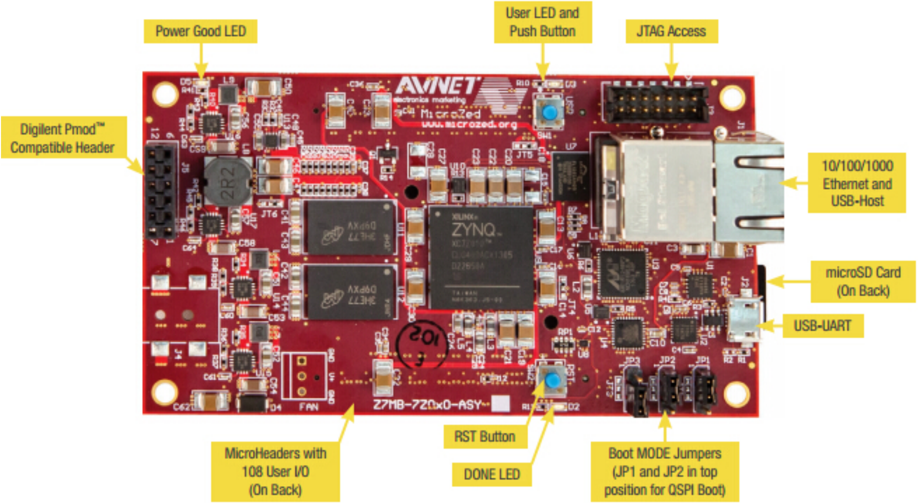

Power Good LED
User LED and Push Button
JTAG Access
Digilent Pmod™ Compatible Header
10/100/1000 Ethernet and USB-Host
microSD Card (On Back)
USB-UART
MicroHeaders with 108 User I/O (On Back)
RST Button
DONE LED
Boot MODE Jumpers (JP1 and JP2 in top position for QSPI Boot)

Figure 2.4: MicroZed main board

With a traditional processor, the hardware platform is pre-defined. The manufacturer selected the processor parameters and built-in peripherals when the chip was designed. To make use of this pre-defined processor, users need only target that particular hardware platform in the software development tools. The Zynq-7000 All Programmable SoC is different. These chips provide multiple building blocks and leave the definition to the design engineer [14]. This adds flexibility, but it also means that a little bit of work needs to be done before any software development can take place. Xilinx provides the Vivado Design Suite[4] which allows the engineers start with MicroZed in a stand-alone mode as a learning platform and then quickly expand their building blocks, customize own IP cores and set up all peripherals [14].

The Xilinx SDK[5] is provided with the tools to support the software development for MicroZed. It consists of a modified Eclipse distribution, including a plugin for C and C++ support, as well as MicroZed versions of the GCC toolchain. SDK also includes a debugger that provides full support for multi-core debugging over a connection to the MicroZed.

---

[4] Version 13.2 was used in this project

[5] Version 14.6 was used in this project

2.3 Real-Time Operating System

In contrast to non real-time operating systems such as Windows or Linux, a real-time operating system (RTOS) is designed to serve real-time application requests. The key role of an RTOS is to execute application tasks in predictable quantities of time [1, p, 118-119] in order that all tasks can meet their deadlines.

Of course, all types of RTOS contain some functions to provide an interface to switch tasks in order to coordinate virtual concurrency in a single-core processor environment, or even true concurrency in multi-core processors. Such an RTOS has the following goals [1, p, 79]:

- To offer a reliable, predictable platform for multitasking and resources sharing.
- To make the application software design easier with less hardware restriction.
- To make it possible for designers to concentrate on their core product and leave the hardware-specific problems to board vendors.

Generally, a real-time operating systems provides three crucial functions [1, p, 79]: scheduling, interrupt handling, inter-task communication and synchronization. The RTOS kernel is the key part that provides these three functions. The scheduler determines which task should be assigned a high priority and executed first, which low-priority task might be preempted by a high-priority task. The scheduler should schedule all the tasks to let them meet their time restrictions. Interrupt handling is an ability to handle the unexpected internal and external interrupts in time. Inter-task communication and synchronization guarantees that parallel execution tasks should share information in a safe manner.

2.3.1 Preemptive Priority Scheduling

A running lower-priority task can be preempted by a higher-priority task. Scheduler deployed preemption schemes instead of round-robin [15] or first-come-first-serve schemes are called preemptive priority schedulers. The priorities are assigned based on the urgency of the tasks [1, p, 90]. For instance, in a car, the engine monitoring task is more important than the air-condition control task. Therefore, the engine monitoring task will be assigned as a high priority than the air-condition control task.

Prioritized scheduling can be either a fixed priority scheduling or a dynamic priority scheduling. In the fixed priority systems, the priorities of tasks are determined during the system initialization phase, they cannot be changed afterward [1, p, 90]. Dynamic priority system, by contrast, allows the priority of tasks to be adjusted during the run-time environment in order to meet the dynamic time requirements [1, p, 90]. So, usually, a dynamic priority system is more flexible than a fixed priority system. However, dynamic priority schedulers are much more complicated, have higher requirements for the hardware platform. Hence, most embedded systems are equipped with a fixed priority scheduler. Another reason is that there are limited situations in which systems need the dynamic priority scheduler [1, p, 90].

Two types of scheduling policies exist [1, p, 90]: pre-runtime scheduling and run-time scheduling. In pre-runtime scheduling, top designers will generate a feasible schedule offline, which keeps the order of all tasks and prevents conflicting access to the shared resources [16, p, 51-52]. One of the advantages of this type is low context switch overhead. However, in the run-time scheduling, fixed or dynamic priorities are assigned during the run-time. This scheduling algorithm mainly relies on a relatively complex run-time environments for inter-task communication and synchronization.

Preemptive priority schedule approaches are widely researched in academia, some favorite examples such as Rate-Monotonic Scheduling Approach and Earliest Deadline First Approach [17, p, 46-61]. The detailed description of these algorithms will not be covered here since these topics already beyond the scope of the thesis.

2.3.2 Interrupt Handling

A real-time operating system normally includes some pieces of code for interrupt handler. The interrupt handler prioritizes the interrupts and stores them in a waiting queue if more than one interrupt is required to be handled. There are two kinds of interrupts: hardware interrupts and software interrupts [1, p, 87]. The major difference between them is the trigger mechanism. The trigger of a hardware interrupt is an electronic signal from an external peripheral. While the source of a software interrupt is the execution of particular instructions, typically a piece of machine-level instructions [1, p, 87]. Another special type of internal interrupt is triggered by the program's attempt to perform an illegal or unexpected operation, which is called *exceptions* [6] [1, p, 87]. When an interrupt occurs, the processor will suspend current execution code, jump to a pre-defined location [7] and execute the code associated with each interrupt source.

Hardware interrupts are asynchronous, which means, they can happen at any time. The developers need to write interrupt handlers for each hardware interrupt, so that the processor could know which interrupt handler should be invoked when an explicit hardware interrupt occurred [1, p, 88].

Access to shared resources in the interrupt handlers is unsafe under the multitasking environment since these shared resources might be accessed by other tasks' interrupt handlers at the same time. A code that is performing access to the shared resources

---

[6] ARM cortex-A9 processors have five exception modes: fast interrupt, interrupt, SWI and reset, Prefetch Abort and Data Abort, and Undefined Instruction.

[7] Normally defined in the vector table

is called the critical section [18]. Engineers need to disable interrupt before entering the critical section, and enable it again after exiting from the critical section. The critical section code should be optimized as short as possible since the system might miss some important interrupt requests during the time interrupt disabled [1, p, 88].

The current system status, also called *context*, must be reserved in processor's registers when switching tasks, so that they can be restored after resuming the interrupted task. Context switching is thus process storing and restoring the status of tasks, and by this way, the execution could be resumed from the same point at a later time. In RTOS, the context switching is usually implemented by assembly code since it has to directly access the processor's local registers. Another reason is that context switching time is a major contributor to the system's response time, and thus should be optimized as fast as possible [1, p, 88].

2.3.3 Synchronization

It is assumed that the ideal task model should be like that all the tasks are independent execution. However, this assumption is unrealistic from a practical perspective. Task interaction is quite common in most modern real-time applications [1, p, 106]. So, some communication and synchronization mechanisms should be applied to guarantee that the tasks interaction are safe. A variety of approaches can be used for transferring data between different tasks, ranging from simple one[8] to more sophisticated ideas[9].

In most situations, shared resources can only be occupied by one task at one time and cannot be interrupted by other tasks. If two tasks access the same shared resource simultaneously, some unexpected behavior might occur. The details are discussed in Section 2.4.

Mutual exclusion (normally abbreviated to **Mutex**) is a method that widely used to avoid the simultaneous use of the shard resources. Several software based mutual exclusion algorithms have been designed, which all rely on the same basic idea: the algorithm allows threads to determine whether another competing thread has already occupied the target shared resource or attempting to hold this resource. One of the simple solutions designed by Gary L. Peterson for two threads is presented below [19, p, 115-116].

```
//declare shared variables
volatile bool turn = 0;
volatile bool Q[2] = {0, 0};

// for thread 0
void accessSharedResourceThread0()
```

---

[8] like global variable

[9] like Ring Buffers, Mailboxes, Semaphores

```
{
  Q[0] = 1;
  turn = 1;
  //waiting in a loop if resource is occupied by another thread
  while(turn && Q[1]);

  //the critical section code here
  //release the signal when exit
  Q[0] = 0;
}

// for thread 1
void accessSharedResourceThread1()
{
  Q[1] = 1;
  turn = 0;
  //waiting in a loop if resource is occupied by another thread
  while(!turn && Q[0]);

  //the critical section code here
  //release the signal when exit
  Q[1] = 0;
}
```

Listing 2.1: Gary L. Peterson's two threads mutual exclusion algorithm

The **Q** array represents the intention of a thread to enter the critical section. The **turn** variable indicates whether another thread has already entered the critical section or about to enter it [19, p, 115-116]. When a thread attempts to access the critical section, the element of **Q** array indexed by the thread Id will be set to true, and the value of the **turn** variable will equal to the index of the competing thread.

It is also possible that both threads may attempt to go into the critical section at the same time. In this case, one thread will modify the **turn** variable first than the other thread, and will exit the while loop first because the competing thread will toggle the **turn** variable, so the while expression for first thread will evaluate to False. As a consequence, the first thread will jump out of the loop and go into the critical section. At the same time, the competing thread has to wait inside the while loop before the first thread exits the critical section.

2.4 Challenges for multi-core and multiprocessor programming

In the past, software engineers could just wait for transistors to be squeezed smaller and faster, allowing processors to become more powerful. Therefore, code could run faster without taking any new effort. However, this old era is now officially over.

Software engineers who care about performance must learn the new parallel concept and attempt to parallelize their programs as much as possible.

However, a program that runs on the five-core multiprocessor more likely reaches far less than a five-fold speedup performance than the single-core processors. Also, complications may arise in terms of additional communication and synchronization overhead.

2.4.1 The realities of parallelization

In an ideal world, upgrading from a single-core processor to an n-core processor supposed to provide an n-fold increment in computational power. In fact, unfortunately, this never happens. One reason for this is that most real-world computational problems cannot be effectively parallelized.

Let us take a real-world example, considering four cleaners who decide to clean four cars. If all the cars have the same size, it makes sense to assign each cleaner to clean a car. An assumption is made that each person has the same clean rate. We would expect a four-fold speedup over the single cleaner's case. However, the situation becomes a little bit complicated if one car is much bigger than the others, then, the overall completion time is decided by the larger car which definitely takes longer time to complete.

The formula we need to analyze the parallel computation is called **Amdahl's low** [20, p, 483-485]:

$$s = \frac{1}{(1-p) + \frac{p}{n}}$$

where p represents the ratio of the parallel portion in one program, n represents the number of processors, s represents the maximum speedup that can be reached.

Considering the car-cleaning example, assume that each small car is one unit, and the large car is two units. Assigning one cleaner per car means four out of five units are executed in parallel. So, Amdahl's law states the speedup:

$$s = \frac{1}{(1-p) + \frac{p}{n}} = \frac{1}{\frac{1}{5} + \frac{\frac{4}{5}}{4}} = 2.5$$

Unexpectedly, only a 2.5-time speedup reached by four cleaners working parallel. However, things can get worse. Assuming ten cleaners need to clean ten cars, but one car is twice the size of others, the speedup is:

$$s = \frac{1}{(1-p) + \frac{p}{n}} = \frac{1}{\frac{1}{11} + \frac{\frac{10}{11}}{10}} = 5.5$$

applying ten cleaners for a job only yields a five-fold speedup, roughly half of the value we expect.

A better solution is that cleaners might help others to clean the remaining cars as soon as their assigned work is completed. However, in this case, the additional time overhead should be counted when considering the coordination among all cleaners.

Here is what Amdahl's law teaches us about the utilization of multi-core system. In general, for a given task, Amdahl's law indicates that even if you parallelize 90% parts of one job, the remaining 10% will only yields a five-fold speedup, not as you expected, ten-fold speedup. In other words, the remaining part will dominate the overall time consumption. Therefore, you can only achieve a full ten-fold speedup by dividing a task into ten equally sized pieces. So, it seems worthwhile to explore an effort to derive as much parallelism from the remaining part as possible, although it is difficult.

2.4.2 Atomic Operations

An operation performing in a shared memory space is atomic if it completes in a single step [21]. For example, when an atomic load is carried out on a shared memory space, it reads the entire value as it appeared at a single moment in time [21]. Non-atomic load does not make that guarantee, which means a non-atomic load operation needs several steps to perform. Additionally, atomic operations normally have a succeed-or-fail definition when they either successfully change the state of the system or just return if have no apparent effect [21].

However, in the multiprocessing environment, which instructions are atomic, what kind of atomic mechanisms provided by processors? For example, Intel x86 processors provide three mechanisms to guarantee atomic operations [22]:

- Some guaranteed atomic instructions, such as reading or writing a byte instructions.
- Bus locking. Using the LOCK instruction and LOCK signal prefix.
- Cache coherency protocols ensure that atomic operations can be carried out on the cached data structures (cache lock).

The non-atomic operations might cause unexpected behavior under multiprocessing environment. Considering a non-atomic operation below:

```
// demo for non-atomic operation
//c code:
  x = x + 2;
//assembly code in X86 platform
  mov eax, dnord ptr[x]
  add eax, 2
  mov dnord ptr[x], eax
```

Listing 2.2: Demo code for non-atomic operation

An adding operation in C code will be represented three instructions in assembly code, which indicates the *addition* operation in C is a non-atomic operation. Consider two threads running the same code with one global element x, the unexpected result will appear.

```
// demo for two threads running concurrently
//            x = 1
// thread 1: x = x + 2
// thread 2: x = x + 2
//time      thread 1                    thread 2
   1        mov eax, dnord ptr[x]       ---------
   2        -------------               mov eax, dnord ptr[x]
   3        add eax,  2                 add eax, 2
   4        mov dnord ptr[x], eax       ----------
   5        -------------               mov dnord ptr[x], eax
 //
```

Listing 2.3: Demo code for two threads running under non-atomic operation concurrently

The return result of thread one will equal to 5 instead of 3, because of the overwritten of value x by thread two. One simple solution is bus locking, using the *LOCK* signal to lock the whole bus to guarantee that only one operation is performed inside this bus at a time. [23]. However, one of the disadvantages is that this method mainly relies on the underlying hardware implementation, in another word, is hardware-specific.

ARM Cortex A9 processor provides a number of atomic instructions as well [12], but, a core running these instructions does not coordinate with other cores. Thus, it does not provide atomicity across multiple cores. By this reason, those atomic instructions are inappropriate for the multi-core synchronization.

2.4.3 Cache Coherence

When using multiple cores with separate L1-caches, it is significance to keep the caches in a state of coherence by guaranteeing that any shared elements that are modified in one cache need to be updated throughout the whole cache system. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the entire system [24]. Normally, this is implemented in two ways: through a directory-based system or a snooping system. In a directory-based system, the data being shared is stored in a common directory which maintains the coherence between caches. This directory acts as a filter through which the processor needs to require permission to load an entry from the primary memory to its cache [24]. When the entry is changed, the directory either updates the entry or invalidates the other caches with that entry. In a snooping system, all caches monitor (also called snoop) the bus to check whether they have a copy of the block of data which is required on the bus [24].

The Snoop Control Unit inside the ARM Cortex A9 processor series is responsible for cache coherence and system memory transfers [25, p, 26]. It provides multiple APIs that let developers configure the caches and memory systems based on their requirements without the knowledge of underlying implementation.

2.4.4 Sequential Consistency

Between the time when typing in some C/C++ code and the time it executes on a processor, the code sequence might be reordered without modifying the intended behavior. Changes are made either by the compiler at compile time or by the processor at run time, both by the reason of allowing your code run faster.

The primary job of a compiler is to convert human-recognized code to machine-recognized code. During the conversion, the compiler is free to take any liberties, one of them is memory reordering, which typically happened when the compiler optimization flags are enabled. Consider the following code:

```
// c code:
 int x, y

 voud fun(){
   x = y + 1;
   y = 0;
 }

//assemby code:
//by gcc 4.7.1 without enabling the optimization flags
fun:
   ....
   movl   y(%rip), %eax
   addl   $1, %eax
   movl   %eax, x(%rip)
   movl   $0, y(%rip)
   ....
//
//by gcc 4.7.2 with enabling the optimization flags by -O2
fun:
   ....
   movl   y(%rip), %eax
   movl   $0, y(%rip)
   addl   $1, %eax
   movl   %eax, x(%rip)
   ....
```

Listing 2.4: Demo code for memory reordering when enabling the compiler optimization

Under the compiler optimization flags disabling condition, you can find that (in line 16) the memory store to global variable y occurs right after the memory store to x, which keeps the sequential consistency with the original c code.

When enabling the compiler optimization flag (-O2), the underlying machine code is reordered, store to y will perform before storing to x. The reason compiler reorders the instructions in this case is writing back to memory will take several clock cycles, so it will reduce some latency if executing this instruction in the beginning [5].

However, such reordering can cause problems in the multiprocessing environment. Below is a commonly-cited example, where a shared flag is used to indicate whether some other variables are available or not.

```

int sharedValue;
bool isPublished = 0;

void sendValueToAnotherThread(int x)
{
    sharedValue = x;
    IsPublished = 1;
}
```

Listing 2.5: Example of synchronization with shared flag

Normally, the flag value *isPublished* will toggle to true after *sharedValue* assigned a new value. Imagine what will happen if the compiler reorders the store to *IsPublished* before the store to *sharedValue*. A thread could very well be preempted by operating system between these two instructions, leaving other threads believe *sharedValue* has been already available which, in fact, is still not updated[26].

Apart from happened at compile time, memory reordering occurred at run time as well. Unlike compiler reordering, the effects of processor reordering are only visible in multi-core and multiprocessor environment [27].

A common way to enforce correct memory ordering on the processor is using some special instructions which serve as memory barriers. More sophisticated types of memory barrier, like fence instructions [28], will not be presented here.

2.4.5 Multi-core Scheduling

Tasks activation depends on the scheduling algorithm. Unlike single-core scheduling algorithms, multi-core ones not only have to decide in which order the tasks will be executed, but also need to decide on which core a task will be performed. This comes up the definition of two categories: *global* and *partitioned*, allowing or not allowing migrations of tasks over the cores.

To be acceptable for an embedded aircraft system, a scheduling schemes should meet the following requirements [29, p, 23]

- **Feasibility**: All tasks should be scheduled to meet their deadlines.
- **Predictability**: The response time of the set of tasks does not increase if the execution time of one task decreases.

As we discussed in Section 2.3, priority-based preemptive scheduling algorithms are fit for single-core processors. They are relatively easy to implement, and their worst-case execution time can easily be calculated. Although some multi-core scheduling algorithms still verify those two requirements, for example, *Global Rate Monotonic* or *Global Deadline Monotonic*. However, they are both the cases for fixed priority algorithm instead of dynamic priority algorithms.

When considering the partitioned algorithms, the problem to some extent remains equivalent to single-core versions of the algorithm. Therefore, they are predictable. Global scheduling algorithms, although, have some advantages over partitioned algorithms, the additional overhead will be involved in terms of tasks migration, which may lead to unpredictable behavior.

2.4.6 Architectural Considerations

As discussed before, in the symmetric architecture, a single operating system instance is deployed among all cores. The SMP privileged operating system executed under a non-disjoint execution environment on each core. Two cores, for example, the services on core0 are isolated from the duplicated services on core1. However, core1 shares the memory space with core0. Which means that all core0's did may have some influences on core1. The address that accessed by core0 is accessible by core1 as well. So, particular attention has to be taken to prevent unnecessary memory access. Moreover, many additional issues need to worry about despite the attractive: the added complexity of real-time determinism, the reuse of existing software and the complexity of multitasking communication and synchronization [10].

Asymmetric architectures are used when several independent operating system instance are deployed on different cores. Each operating system instance has its private context which means on each core, the memory space is not visible from the other cores. This feature enables the reuse of single-core operating system with minimal modification.

2.5 RODOS

RODOS (Real-time Onboard Dependable Operating System) [30] is a real-time operating system for embedded systems and was designed for application domains demanding high dependability. It supports all the fundamental features that one can expect from a modern real-time operating system. Which includes features such as resource management, communication and interrupt handling. Besides, a fully preemptive scheduler is implemented, which supports both fixed priority-based scheduling and round-robin scheme for threads within the same priority level.

2.5.1 Introduction

RODOS was developed at the German Aerospace Center(DLR) [30] and has its roots in the operating system BOSS. Now, RODOS is enhanced and extended at the German Aerospace Center as well as the department of aerospace information technology at the University of Würzburg [31].

The features RODOS offers [31]:

- Object-oriented C++ interfaces
- Ultra fast booting and execution
- Real-time priority controlled preemptive multithreading
- Time events
- Thread safe communication and synchronization

2.5.2 Directory Structure

The directory structure of the current RODOS distribution is divided into four directories:

- **rodos-core-master**: includes the source code of RODOS kernel for many different hardware platforms.
- **rodos-doc-master**: includes several related documents.
- **rodos-support-master**: includes some programs and libraries used in space application, such as GUI, Matlab support libraries, filesystem, etc.

- **rodos-tutorials-master**: includes several necessary tutorials and examples for beginners.

Inside the directory *rodos-core-master*, The source structure of RODOS kernel is consisted of three sub-directories: *bare-metal*, *bare-metal-generic* and *independent*. Files inside the directory *independent* are all hardware-independent files. These files provided the high-level abstraction needed for implementation based on the specific hardware platform. Files in the *bare-metal* and *bare-metal-generic* are supported for RODOS running on bare-metal targets, instead on the host operation system, like Linux and Windows. The difference between these two directories is that files inside the *bare-metal* are completely hardware related, one file may have different versions for different hardware platforms. However, the *bare-metal-generic* directory only contains some general definition files which compatible with all hardware platforms. The detailed discussion regarding each file's functionality will be presented in Section 2.6.

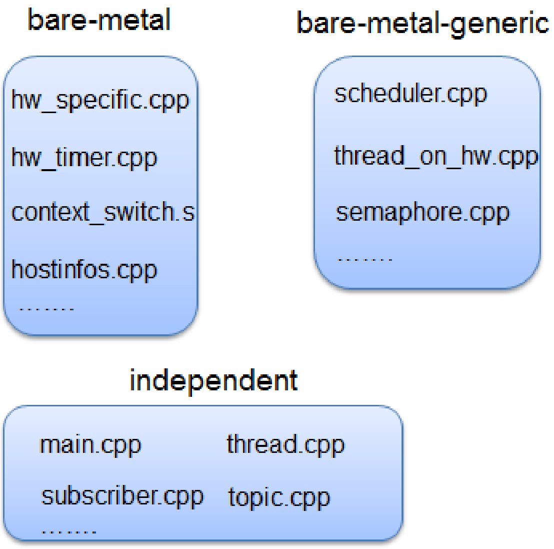


Figure 2.5: RODOS source code structure

2.5.3 Threads

Threads in RODOS are user defined parts of the software that contain logic to fulfill a specific purpose.

RODOS uses fair priority controlled preemptive scheduling [31].The running threads with a lower priority are preempted by the ready thread with the higher priority. If there is more than one thread with the same priority, each of them gets a fixed share of computing time, and they are executed one by one [31].

Another possible interruption of a thread is periodical execution. If one thread has finished its tasks, it can suspend for a defined amount of time, after that time, the scheduler resume the suspended thread which is very useful for actions that have to be executed periodically. While one thread is suspended, another ready thread can be executed and, therefore, no CPU time is wasted in waiting.

Considering the following code, which creates two threads with different priorities. The higher priority thread preempts the lower thread periodically.

```
class HighPriorityThread: public Thread{
public:
  HighPriorityThread() : Thread("HiPriority", 25) { }
  void init()
  {
    PRINTF(" highpriority = '*'");
  }
  void run()
  {
    while(1)
   {
    int64_t now = NOW();
    PRINTF("*");
    AT(now + 1*SECONDS);
    }
  }
};

class LowPriorityThread: public Thread {
public:
  LowPriorityThread() : Thread("LowPriority", 10) { }
  void init()
  {
    PRINTF(" lowpriority = '.'");
  }
  void run()
  {
    long long cnt = 0;
    while(1)
    {
       cnt++;
       if (cnt % 10000000 == 0) {
       PRINTF(".");
      }
    }
  }
};

HighPriorityThread highPriorityThread;
LowPriorityThread  lowPriorityThread;
```

Listing 2.6: Demo code of RODOS priority threads

The *LowPriorityThread* continuously outputs the character "." and is interrupted every second by the *HighPriorityThread*, which writes the character "*".

New thread is created and initialized by the *thread* constructor. RODOS allocates stack for the new task, sets up the context with the pre-defined parameters. The scheduler starts after thread constructor's work completes, picks up the highest priority thread among all the ready-list threads, and runs. Below is the UML sequence diagram for thread's creation and execution:

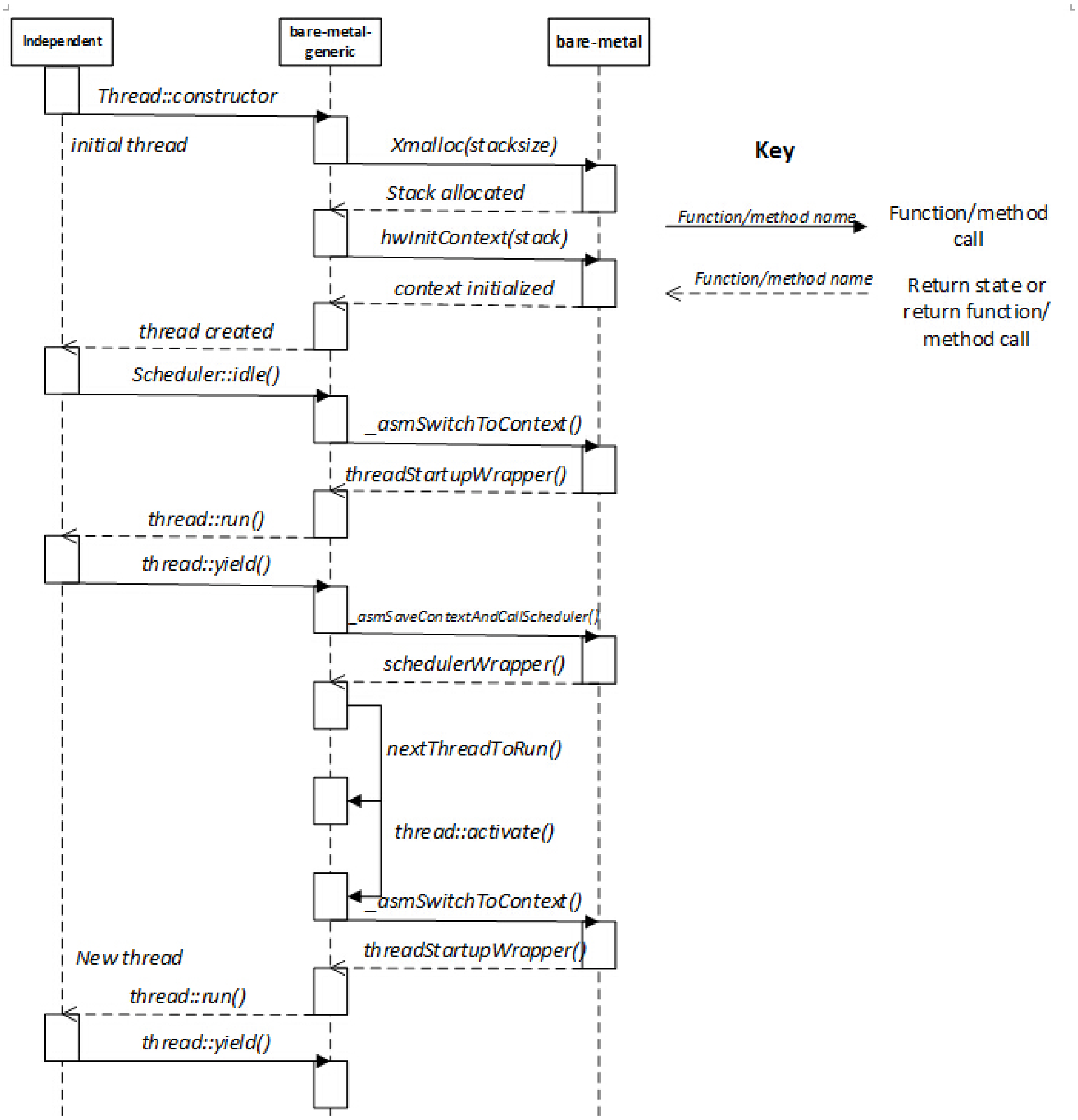


Figure 2.6: UML sequence diagram for thread's creation and execution

2.5.4 Middleware

Modern real-time operating systems often include not only a core kernel, but also a middleware, a set of software frameworks that provides additional services for application developers [32, p, 7]. RODOS uses a middleware for communication between local threads and threads on distributed RODOS systems. The middleware follows the publisher/subscriber mechanism [32][33]. The interface between the publisher and the subscriber is called *Topic*. A thread publishes information using a topic, for example, the current temperature value. Every time the publisher provides new temperature data, the thread that subscribes to this topic will receive the data. However, there can also be multiple publishers and subscribers. The advantage of this middleware is that all publishers and subscribers can work independently without any knowledge about others. Consequently, publishers can easily be replaced in case of a malfunction. The subscribers do not notice this replacement.

Chapter 3

3 Problem Analysis and Requirements

3.1 Overview of requirements

The principal requirement of this thesis is to implement a multi-core version of RODOS operating system that schedules threads for concurrent execution on dual-core MicroZed platform.

Two categories of requirements are proposed: functional requirements and non-functional requirements. The description of each requirement is showed below. Each requirement is signed either primary or secondary based on the relative significance. All the primary requirements should be considered first, the secondary requirements might be met depending on the time constraints.

3.2 Functional Requirements

3.2.1 Porting RODOS to on one core as a start point of the multi-core version implementation

Type: Primary
Description: Modifying some hardware-specific files to allow original RODOS to run on one core of MicroZed board successfully.

3.2.2 RODOS should be booted on a multi-core platform

Type: Primary
Description: Modifying the hardware-dependent boot loader and setting the system boot sequence to let the operating system be booted on a dual-core MicroZed board without any collisions of shared components.

3.2.3 Modified RODOS should have an ability of concurrent execution of threads in multi-core platform

Type: Primary
Description: RODOS source code will be extended (either for SMP or AMP) to enable threads running in multiple processors concurrently. The initial idea is adapting RODOS on SMP, solution for AMP sets as an alternative.

3.2.4 New communication and synchronization APIs should be provided

Type: Primary
Description: New APIs should be provided for parallel applications, and these APIs guarantee that threads communication and synchronization is determinate and deadlock free.

3.2.5 Test sets should be generated to demonstrate the new features of multi-core version of RODOS

Type: Primary
Description: Several test sets should be created to determine the performance improved over the original RODOS and to demonstrate the concurrency and synchronization features.

3.2.6 Modifications made to RODOS should be applied for n-cores platform with minimum modification

Type: Secondary
Description: It would be possible to extend the current solution to a platform with more than two cores without too much modification.

3.3 Non-functional Requirements

3.3.1 The testing result should be reproducible

Type: Primary
Description: The source code must be modified under the open source license and managed by the version control software (Git or Subversion). A detailed tutorial should be provided for application engineers to duplicate the results.

3.3.2 The whole project should be completed in five month

Type: Secondary
Description: In order to leave enough time for the final report writing, the whole project will take no longer five month. So, in order to efficient use of limited time, the primary requirements will be considered as a higher priority requirements which should be met first.

Chapter 4

4 Design

4.1 Principle

After considering the hardware and software aspects, the initial design decision is applying the SMP architecture for this project. Some benefits offered when both cores are running under the same operating system instance. Tasks are free to be executed in either core. Memory space could be shared directly without additional memory segmentation. However, this freedom may lead to some special attention on synchronization issues since variables defined in core0 are visible from core1. So, in order to avoid both cores to access the same region of memory, a synchronization mechanism needs to be applied.

Another advantage offered by SMP is high execution efficiency. SMP's global scheduling algorithm allows the ready tasks can be executed by arbitrary core, even tasks can be moved from one busy core to another idle core. Therefore, the situation that one core has a long queue of ready tasks while another core is waiting in the idle stage will never occur.

Alternatively, AMP will be a backup solution for this thesis. Since each core has its operating system instance, which allows the reuse of single-core version of RODOS with minimal modification. Besides, AMP solution is deterministic because the scheduling algorithm will keep the consistency of the original version of RODOS. However, for the sake of reaching task load balancing, the top-level application engineers should have some knowledge about each task's execution time, and then allocate tasks to explicit core manually. Moreover, special care must be taken to prevent both cores from the collision for the shared resources.

4.2 Dual-core Boot Sequence

In the multi-core system, the boot sequence should be redesigned since shared components only needed to be initialized once. One core will be started as a primary core.

The primary core is in charge of performing hardware early initialization. It executes any initialization code which only need to be performed once, to initialize all global resources, such as global timer, L2-cache, snoop control unit (SCU), interrupt control distributor (ICD), etc. Another core serves as a secondary core which only needs to be set up its own private components, such as L1-cache, private timers, private peripheral interrupts (PPI) and memory management unit (MMU), etc. After all the necessary initialization is done, a signal will be sent back to the primary core to notify it that the secondary core wakes up successfully. Afterward, the operating system will be activated. in more detail:

Primary core will:

- Start execution at the address in exception vector table.
- Install its private MMU translation table entries and initialize the MMUs.
- Initialize the private L1-caches.
- Initialize the private peripheral interrupts (PPI)
- Initialize the snoop control unit (SCU).
- Initialize the interrupt control distributor (ICD).
- Initialize the L2-cache, RAM and Flash.
- Initialize the global timer.
- Send a wake-up signal to secondary core.
- Waiting for a response signal from secondary core.
- Initialize operating system.
- Activate scheduler.

Secondary core will:

- Waiting for the wake-up signal from primary core.
- Start execution at the address predefined by exception vector table.
- Initialize the MMU.
- Initialize the L1-cache.
- Send back a notified signal to primary core.
- Waiting for the scheduler to be activated.

Once released, the secondary cores will enter *idle_ thread* where they are ready to run tasks. *idle_ thread* act as a "dummy thread". This is the lowest priority thread that just serves as a placeholder when there is no thread to run on that CPU. So when no real threads are scheduled to execute on that core, the core will go into *idle|_ thread.* When a real thread is ready to run, the dummy thread will be switched out by the scheduler.[34, p, 13-14].

The primary core goes on to execute *main* function, where operating system will be activated. The global scheduler is responsible for taking care of scheduling the various threads across the available cores, Figure 4.1 illustrates the process of dual-core boot sequence.

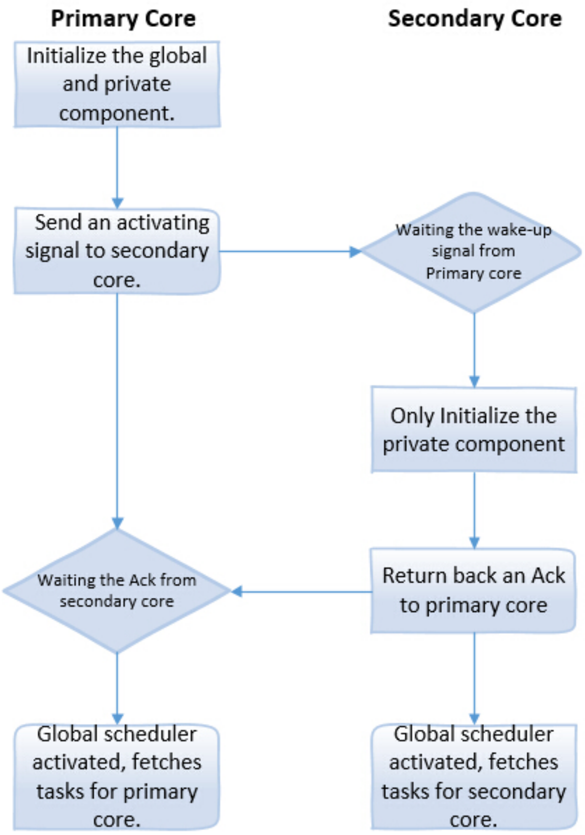

Primary Core
Secondary Core
Initialize the global and private component.
Send an activating signal to secondary core.
Waiting the wake-up signal from Primary core
Only Initialize the private component
Waiting the Ack from secondary core
Return back an Ack to primary core
Global scheduler activated, fetches tasks for primary core.
Global scheduler activated, fetches tasks for secondary core.

Figure 4.1: Dual-core boot sequence

4.3 Memory Model of SMP

The memory model of SMP illustrated by Figure 4.2 is quite intuitive. The factory-programmed bootROM will be executed first. The bootROM determines the boot

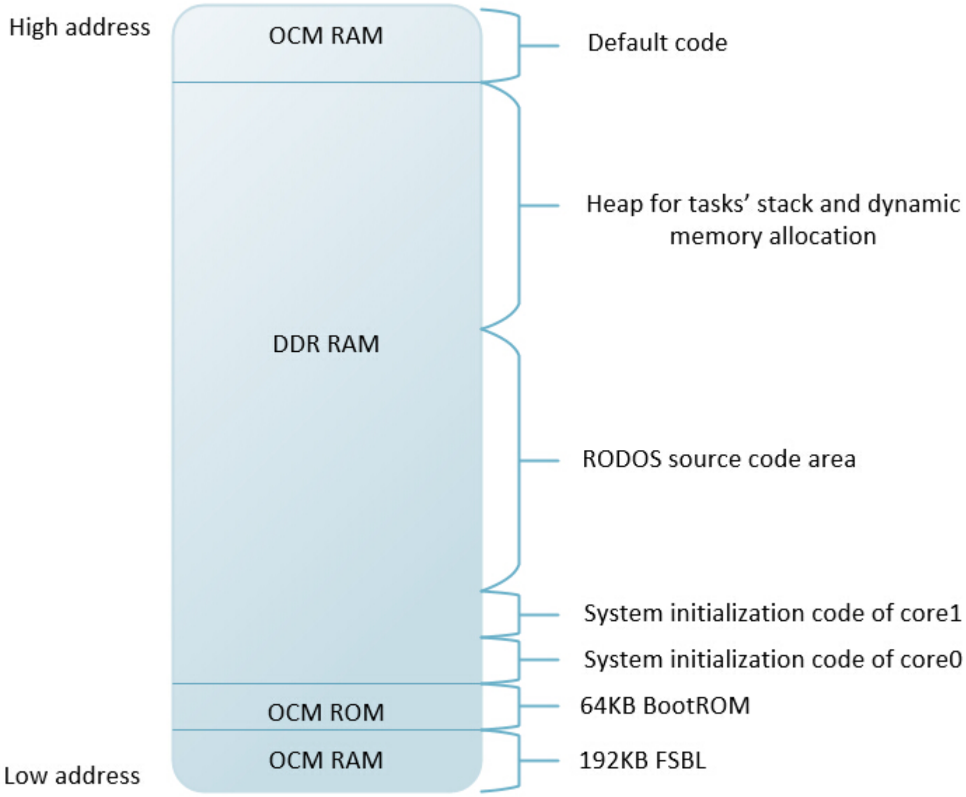

High address
Low address
OCM RAM
DDR RAM
OCM ROM
OCM RAM
Default code
Heap for tasks' stack and dynamic memory allocation
RODOS source code area
System initialization code of core1
System initialization code of core0
64KB BootROM
192KB FSBL

Figure 4.2: Memory map for SMP

device, performs some necessary system initialization. Followed is the First-Stage Boot Loader (FSBL), which located at the bottom of shared memory space. The process of FSBL enables engineers to configure the entire SoC platform (contains PS and PL) [35]. Additionally, the JTAG interface is enabled to allow engineers to access the platform's internal components for the test and debug purpose [35].

The code located just after the FSBL is called system initialization code of core0, this piece of code is a low-level code provided by Xilinx standalone board support package (BSP). The code starts execution at the vector table that defines a branch of instructions for different type of exceptions, performs global and private component initialization, and, if everything is correct, will eventually jump to the *main* function, which is the entry point of RODOS source code.

Followed area is the system initialization code of core1. It first waits for a wake-up signal from core0, after received this signal, the dedicated initialization code for core1's private resource will be executed.

The source code of RODOS is seated in the middle area of memory space just after the system initialization code. The heap region is used for stack allocation at the run-time when a new thread is created by thread's constructor. A pre-defined quantity of memory on the heap area will be used for the thread's stack. The reason each thread needs to have its stack is quite simple. In a multitasking environment, threads are frequently swapped in or out by the operating system. Thread-specific stacks need to be used to reserve their parameters of current states. If only one stack is used among all the threads, the parameters of a thread being swapped in might be corrupted by the thread being swapped out. Therefore, allocating a private stack for each thread can avoid this issue. Moreover, heap is also an area used for memory allocation dynamically at run-time.

4.4 RODOS Modification for SMP

This section deals with the actual modifications of the source code of RODOS to support the SMP model.

4.4.1 Thread Control Block

The Thread Control Block (TCB), the same as task control block in other operating systems, is a critical data structure containing several parameters of the thread[1], which used by the scheduler to identify threads. In single-core version of RODOS, a pointer called *nextThreadToRun* exists to determine that the thread next will be executed by the processor, in another word, *nextThreadToRun* serves as a representation of what the processor will execute. By referring to this pointer, the scheduler can

---

[1] Such as name, priority, stack size, address of context, etc.

know which threads should be switched in next. The content of this pointer always up-to-date by the scheduler to make sure the thread that will be switched in next is indeed the highest priority thread among all the waiting threads. It is thread safe since additional synchronization mechanism is not required. The thread pointed by *nextThreadToRun* is always switched in by the context switching code before executed by the processor.

Since multiple tasks will run simultaneously on multi-core platform, the idea of expanding *nextThreadToRun* is quite straightforward, By extending *nextThreadToRun* from a reference pointer to a thread control block into an array of pointers to thread control blocks [4, p, 48-49]. The size of the array is equal to the number of cores, so that each core's ready list can be pointed by an element of the array. For instance, *nextThreadToRun[1]* only connect to the threads that will be executed by core1. By this design, all the features will be inherited naturally from the original version of RODOS. However, one problem that arises is how to let the cores themselves distinguish by themselves. There must be some methods to identify themselves although all cores run the same piece of code. One option is writing the core ID to a special-purpose register in each core during the hardware implementation and configuration since registers are private resources for each core. Therefore, core's ID can be identified directly by accessing each core's special-purpose register.

```
//get coreID
unsigned char coreID = getCurrentCUPID();
//get the next to run thread for each core.
thread* currentThreadToRun = nextThreadToRun[coreID]
//do something each below
............
```

Listing 4.1: Modified *nextThreadToRun* pointer

4.4.2 Core Affinity

In Linux, processor affinity enables the binding of a thread to a particular processor so that this thread only scheduled and executed on that processor [36]. Here, a similar idea is applied for RODOS modification. In many situations, it seems to be a bad idea since the binding thread with one core might make the scheduler inflexible, and an overview reduce the efficiency of the whole system. However, in multi-core environment, certain threads may benefit from having this ability. For example, there is only one serial port in the MicroZed board, and this serial port is only connected to the first core. So, with this, it makes no sense to schedule a thread that is trying to communicate with host computer via serial port to the second core[2]. Moreover, threads can be carefully assigned core affinity to optimize the balance of each core.

---

[2] Normally happened in FPGA based hardware platform

Although, it is worth to reminder that the majority of threads can be executed by arbitrary core. Thus, core affinity is unnecessary for them.

Threads must have awareness of their affinities during the entire life. So, there should be a way to express their affinities, as well as a lack of affinity. Fortunately, as mentioned in Section 4.4.1, each core will assign an integer on its special-purpose register to identify itself, this number can also be used to represent the core affinity as well, just simply by adding one additional element into the thread control block data structure. An integer value larger than the number of cores can be adopted to represent the lack of affinity, because no core associated with this affinity value, which is quite intuitive to identify.

By adding the element *CoreAffinity* to the thread control block structure, each thread can be assigned with an integer value for scheduling.

4.4.3 Scheduling

The new version of scheduling seems similar with the original version. Indeed, the affinity-based scheduling works as follow loops:

- Get the current core ID.
- Disable interrupt, enter the critical section.
- Fetch the next thread with the highest priority in the thread ready-list. Check thread's affinity, if matched with core ID, put this thread into *nextThreadToRun[coreID].*
- If not matched, fetch the next highest priority thread and check thread's affinity again. If it is lack-affinity task, put this into *nextThreadToRun[coreID]*, since this task can run in arbitrary core.
- If there is no thread in the ready-list, select the previously executing thread which might be preempted by a high-priority thread.
- Exit the critical section. Now, the selected thread will be executed.
- After the current thread complete execution, return to the first stage and continues again.

Let's take an example to illustrate how the new scheduler works.

As in the original design of the RODOS scheduler, the idle thread will be executed first before activating the scheduler. A local counter *idleCnt++* inside the *IdleThread::run()* is used to record how many times the *IdleThread* is invoked by the scheduler, This value can be used as a reference to determine the efficiency of the scheduler. The scheduler is activated immediately after return from the *idle thread.* Based upon this example, we assume all threads need the same execution

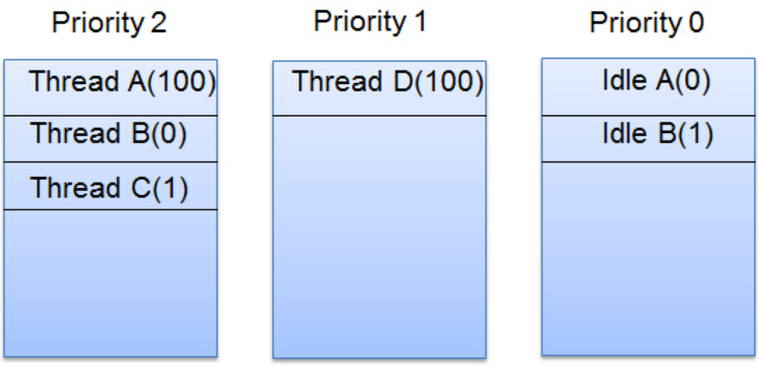


Figure 4.3: Scheduler example

time and core0 runs first in a dual-core platform. The scheduler first gets the first element of highest priority ready queue, that is Thread A, checks the Thread A's affinity, as it is a lack-affinity thread, selects thread A for scheduling on the current core. So, now *NextThreadToRun[0]* = Thread A. After that, the core1 works, the scheduler picks up the first element of highest priority ready queue, now it is Thread B. However, the affinity doesn't match with core ID, thus, gives up Thread B and fetches Thread C. Thread C which fits the affinity with core1, this will be pointed by the *NextThreadToRun[1]*. The scheduler will repeat these steps until all tasks are scheduled. For this example, after scheduling, the result should be as follows:

Core0: NextThreadToRun[0] = Thread A -> Thread B -> Idle A;

Core1: NextThreadToRun[1] = Thread C -> Thread D -> Idle B;

4.4.4 Synchronization

As discussed in the section 2.3.3, Gary L. Peterson's approach works quite well on the two threads environment. However, in terms of multi-core environment, more than two threads will be executed at the same time. Thus, the original version of Gary L. Peterson's mutual exclusion algorithm is no longer valid. Fortunately, in paper [19], an extended n-threads version of Gary L. Peterson's algorithm was proposed.

```
//
 int i = getCurrentThreadId();
 int n = getThreadNum();

//global declaration
 int Q[n];
 int turn[n-1];

//for thread Ti
 for(int j=1; j<n; j++)
 {
   Q[i] = j;
   turn[j] = i;
   bool checkFlag;

   do{
     checkFlag = 1;

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

       if(k == i) continue;

       if(q[k] >= j){
         checkFlag = 0;
         break;
       }
     }
   }while(!checkFlag && (turn[j] == i));
 }

//code for critical section below
//exit critail section
 Q[i] = 0;
```

Listing 4.2: Gary L. Peterson's n-threads mutual exclusion algorithm

The *Q* array still represents the statement of the associated threads. The *turn* variable extended to be an array of size *n-1*, which *n* represents the amount of overall threads. The underlying principle remains the same: In order to let a thread enter the critical section, two conditions must be satisfied [19]:

- No other threads are inside the critical section. The target shared resources are free to access.
- All the other threads are "behind" in the process of entering.

However, it is necessary to notice that this algorithm is very inefficient since only one critical section used to protect all shared resources. As in the multi-core and multiprocessor system, multiple shared resources widely existed. For example, two threads attempt to access two separate shared memory locations at the same time, one thread successfully entering its memory would be forced to prevent another thread accessing its entirely different memory space, because only one general critical section is defined (one global variable used) in this algorithm, rather than several different critical sections identified.

One of the solutions is quite straightforward, letting each shared resource have its mutex, the thread who wants to access an explicit shared resource needs to obtain the associated mutex first. So, by different mutexes, accessing a global resource will have no influence on accessing another different global resource.

An alternative solution is using binary semaphore. In RODOS, a class *semaphore* is implemented as a binary semaphore with two operations included:

- Enter(): Taking a binary semaphore before accessing a shared resource, the caller thread will be blocked if semaphore is occupied by another thread.
- Leave(): Releasing a binary semaphore after exiting from the critical section, let the semaphore available for other threads.

The next problem raised with binary semaphore is called *unbounded priority inversion* [1, p, 118]. Priority inversion typically occurs when a low-priority thread occupies a semaphore, a high-priority thread which attempts to access the same shared memory space is forced to wait on the semaphore until the low-priority thread releases it. However, during the time of waiting, if the low-priority thread is preempted by another mid-priority thread. Then, unbounded priority inversion occurred, because the blocking delay of the high-priority thread is no longer predictable.

One useful approach, the *priority ceiling protocol* [1, p, 118], offers a simple solution to the problem of unbounded priority inversion. Each shared resource is assigned a priority [3] that equals to the priority of the highest priority thread that can occupy it. When a lower-priority thread attempts to access a shared resource, its priority updates to the ceiling's priority, no other threads can preempt this lower-priority inside the critical section. So, in this case, the execution time of lower-priority thread is deterministic. Below is the code for semaphore implementation in RODOS, it might notice the priority ceiling algorithm is applied.

```
void Semaphore::enter() {
  Thread* caller = Thread::getCurrentThread();
  long callerPriority = caller->getPriority();
  PRIORITY_CEILING {
    // Check if semaphore is occupied by another thread
    if ((owner != 0) && (owner != caller) ) {

      // Avoid priority inversion
      if (callerPriority > owner->getPriority()) {
        owner->setPriority(callerPriority);
      }
      // Sleep until wake up by leave
      while(owner != 0 && owner != caller)
   Thread::suspendCallerUntil(END_OF_TIME, this);
      ownerEnterCnt = 0;
    }
    owner = caller;
    ownerPriority = callerPriority;
    ownerEnterCnt++;
```

---

[3] the priority ceiling

```
  } // end of prio_ceiling
  caller->yield(); // wating with prio_ceiling, maybe some one more important wants to work?
}
/**
 *  caller does not block. resumes one waiting thread (enter)
 */
void Semaphore::leave() {
  Thread* caller = Thread::getCurrentThread();
  Thread* waiter = 0;

  if (owner != caller) { // User Programm error: What to do? Nothing!
    return;
  }

  ownerEnterCnt--;
  if (ownerEnterCnt > 0) { // same thread made multiple enter()
    return;
  }

  PRIORITY_CEILING {
    _previusPriority = ownerPriority + 1; // _previusPriority is defined & used in PRIORITY_CEILING, it substracts 1
    ownerPriority = 0;
    owner = 0;
    waiter = Thread::findNextWaitingFor(this);

    if (waiter != 0) {
      owner = waiter; // set new owner, so that no other thread can grep the semaphore before thread switch
      waiter->resume();
    }
  } // end of PRIORITY_CEILER, Restores prio set in _previusPriority
  if ( (waiter != 0) && (ownerPriority == 0) ) { caller->yield(); }
}
```

Listing 4.3: Class semaphore implementation in RODOS

4.5 Memory Model of AMP

One of the biggest differences between the SMP model and the AMP model is the number of operating system instance. Only one operating system instance is executed among all the cores in SMP model, while, at least two operating system instances (either the same or different) running in AMP model. The memory model exposes this difference. As Figure 4.3 indicates, the whole DDR RAM of MicroZed is split into two parts, each part consists of three subparts: an RODOS source code, a low-level system initialization code and a private heap area. The 64kb BootROM and 192kb FSBL keep the same as SMP's counterpart, and both of them located at the bottom of memory space.

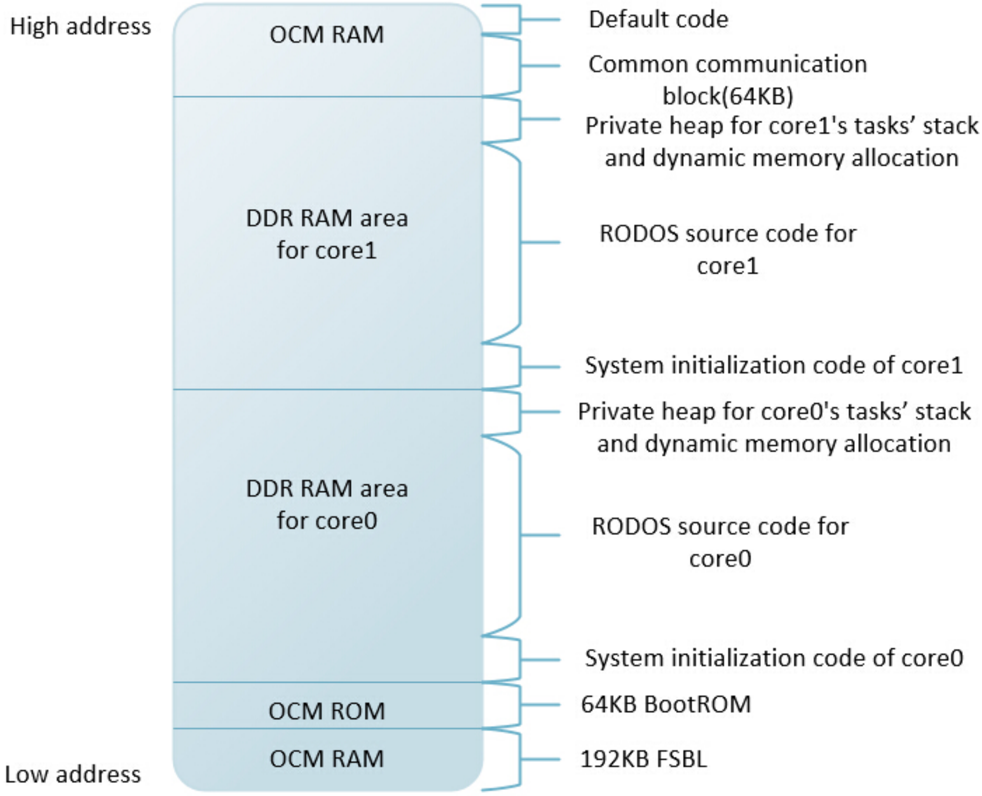

High address
Low address
OCM RAM
DDR RAM area for core1
DDR RAM area for core0
OCM ROM
OCM RAM
Default code
Common communication block(64KB)
Private heap for core1's tasks' stack and dynamic memory allocation
RODOS source code for core1
System initialization code of core1
Private heap for core0's tasks' stack and dynamic memory allocation
RODOS source code for core0
System initialization code of core0
64KB BootROM
192KB FSBL

Figure 4.4: Memory map of AMP

A common communication block located at the top of memory space is designed for communication and synchronization purpose since no shared variables existed among all the cores. The detailed description is presented in Section 5.3.3.

4.6 Multicore Interrupt Handling

As discussed in Section 2.3, an interrupt-driven approach is widely used in many embedded systems that have a number of push buttons, sensors, etc. Inputs from these devices are virtually asynchronous to process currently executing, and a polled-I/O approach is inefficient for systems with many input devices, depending upon the amount of processing required. It is impossible to predict when the event will occur, so the system needs to check I/O status frequently by a polled-I/O approach. Using interrupts enables the processor to continue with other tasks until an event occurs, thus dramatically improving execution efficiency [14].

As processors have become more advanced, the number of interrupt sources has exploded. Consequently, as showed in Figure 4.5, the dual-core Xilinx Zynq SoC uses an ARM Generic Interrupt Controller (GIC) to process interrupts, which can come from [14]:

- Software-Generated Interrupts, 16 for each core. Software-generated interrupts can interrupt themselves and either or both cores.
- Shared Peripheral Interrupts, 60 in total. These interrupts can come from the I/O Peripherals in the Zynq SoC's Processor System (PS) or from the Programmable Logic (PL) side of the device. The two ARM Cortex-A9 MPCore processor's share these interrupts.
- Private Peripheral Interrupts, 5 interrupts, which are private to each core (e.g. private timer, and watchdog timer).

Each core contains a set of private peripheral interrupts (PPIs) with private access by banked registers. The PPIs include the global timer, private timer, watchdog timer, and FIQ/IRQ from the PL. Software generated interrupts (SGIs) are routed to one or both cores. The SGIs are generated by writing to the registers in the generic interrupt controller (GIC) [37].

The generic interrupt controller (GIC) is a centralized resource unit for managing interrupts sent to the processors from the PS and PL. The controller enables, disables, prioritizes and masks the interrupt sources and distributes them to the selected core in a programmed manner as the core interface accepts the next interrupt [14].

The GIC ensures that an interrupt targeted to two cores can only be sent to one core at a time. All the interrupt sources are identified by a unique interrupt ID numbers [14].

The routine of interrupt handling by ARM Cortex-A9 MPCore processor is following:

1. Set up the related register to enable the interrupt.
2. When interrupt occurs, the processor stops executing the current thread.
3. The processor stores the state of the current thread on the stack to allow the suspended thread to continue after the interrupt has been handled.
4. The processor jumps to the interrupt service routine (ISR), which determines how the interrupt is to be handled.
5. The processor resumes operation of the interrupted thread after restoring it from the stack.

Chapter 5

5 Implementation

5.1 Porting RODOS to Single-Core

One of the primary stages of this project is porting original version of RODOS operating system to one core of MicroZed board that sets the basis for the later multi-core implementation. In this section, interfaces that must be provided for a port of the RODOS to the Microzed board are documented.

5.1.1 Development Environment

A GNU cross-compiler is required for a new platform since development and boot image creation is not done from the same architecture we are targeting. So for the MicroZed, this is done by building them with the *arm-xilinx-eabi-g++* target triplet [38, p, 1]. As for the linker, instead of specifying a long list of parameters we create a script that defines the relevant settings. A linker script is parsed every time we link application software against the RODOS libraries. Fortunately, Xilinx SDK automatically generates a linker script based on our system configuration.

5.1.2 System Configuration

There are a number of RODOS-related configuration parameters that define hardware restrictions and the behavior of the operating system in the header file *params.h* which contains a list of pre-processor directives. An exemplary list of some relevant parameters and their values is provided below.

```
//how much memory is going to be available all together
#define XMALLOC_SIZE (1024000)
//the stack size for every thread
#define DEFAULT_STACKSIZE (10240)
//time interval between timer interrupts in ms
#define PARAM_TIMER_INTERVAL (10000)
```

```
//time slice to switch between the same priority threads
#define TIME_SLICE_FOR_SAME_PRIORITY 100*MILLISECONDS
//threads priority
#define DEFAULT_THREAD_PRIORITY (100)
//the highest priority allowed for threads
#define MAX_THREAD_PRIORITY (1000)
```

Listing 5.1: Examples of RODOS-related configuration parameters

In the most recent RODOS versions, there have been some additional interfaces added that acquire information about the host's environment at run-time. These interfaces are very self-explanatory in general. In our case *getHostBasisOS()* returns *"baremetal"* and *getHostCpuArch()* will be *"ARM Cortex-A9"*, and *getIsHostBigEndian()* will return true. Function *getSpeedKiloLoopsPerSecond()* should be set to a fixed value according to benchmark results. To make it clear that this value is currently useless as we return -1 since a negative value represents an impossible benchmark result[38, p, 2].

5.1.3 Startup and Control Routines

Static locals and global objects have to be initialized before we can jump to the kernel's main function. For every new thread, *hwInitContext(long* stack, void* object)* inside the thread constructor is called to set up the thread's context with certain parameters. The first argument of this function is a pointer to the start address of the thread's stack and provides the basis to compute an appropriate stack pointer [38, p, 3-4]. The second argument is the thread object pointer (*this) which is used to be passed to a callback function. It is stored in the register that makes up the first argument of a function call. The initial entry point is the callback function *threadStartupWrapper()*, its address has to be stored within the context structure so it will be loaded to the program counter when context switching occurred [38, p, 4]. *threadStartupWrapper()* begins the thread execution by jumping to its *run()* function which is a virtual function defined by the users. *hwInitContext()* returns a pointer to the newly created context which is the address of struct containing several registers that are allocated inside the stack [38, p, 4].

5.1.4 Timing Interface

Inside the RODOS, a timer is implemented using an interrupt handler that accumulates the past nanoseconds. The initialization of time and registration of a respective interrupt handler is done by *Timer::init()*. The *timer* class provides *start()* and *stop()* interface as well. The timer interval, used for the time between two consecutive timer interrupts, should be set with the method *setInterval()* [38, p, 5].

```
long long unsigned int hwGetNanoseconds();
long long unsigned int hwGetAbsoluteNanoseconds();
void Timer::init();
void Timer::start(); /* may be a stub */
void Timer::stop(); /* may be a stub */
void Timer::setInterval(const long long interval);
```

Listing 5.2: RODOS timer interfaces

Function *hwGetNanoseconds()* returns the value of passed nanoseconds since the system was booted up, the resolution is selected to nanoseconds, since this is the time unit that RODOS relies on. *hwGetAbsoluteNanoseconds()* returns the value of an internal real-time clock if present. Otherwise, it just returns the same value as *hwGetNanoseconds()* [38, p, 5].

5.1.5 Context Switching

In a multitasking environment, threads are frequently switched by the operating system. Responsible routines are mostly implemented in assembly code since we need to directly access and modify the register contents. But before switch, the control should be passed to a thread. This is done by the scheduler with a call to the *___asmSwitchToContext()* routine [38, p, 6]. Here, saved registers are restored and finally a jump to the old program counter is made. As mentioned earlier, when the thread is entered in the first time, it initially jumps to the *threadStartupWrapper()*, which calls the implemented virtual *Thread::run()*.

Context switches can be voluntary, initiated by a call to *Thread::yield()* from currently executed thread. This method invokes a direct call to a function usually implemented in assembly code called *___asmSaveContextAndCallScheduler()*. In case of a voluntary context switch we arrive in the appropriate routines similarly to a normal function call. Thus, only those registers need to be saved which need preserving across function calls [38, p, 6]. After everything is stored in the respective context structure, the context pointer is jumped from the thread which are switching from to the *schedulerWrapper()* [38, p, 6].

These two routines are showed below in detail, are absolutely essential and have to be implemented for every new hardware platform. The routines might be invoked very frequently, which makes it a good idea to optimize them for a minimal number of instructions.

```
void ___asmSwitchToContext(long* context);
void ___asmSaveContextAndCallScheduler(long* context);
```

Listing 5.3: RODOS context switching interfaces

5.2 The Reasons of Switching to AMP

Several unexpected bottlenecks occurred during the attempts of SMP implementation. The degree of difficulty is underestimated.

Unlike the most types of RTOS, RODOS was implemented by C++. The custom threads are initialized by a particular function called thread constructor. The compiler automatically calls that thread's constructor at the point a new thread is created. The thread initialization process contains several memory and registers specific operations, such as context creation and stack allocation. For a thread, *hwInitContext()* is called to set up the context with certain parameters. In SMP model, all threads need to be declared in one processor first, and then, to be dispatched to their target processor based on the value of *CPUAffinity*. However, the problem is that, the context of threads will not move from one core to another automatically. A new low-level function should be designed to help move the context from one core to another. However, unfortunately, this function still hasn't implemented till the end of this project.

```
class samplethread : public Thread {
public:

  samplethread() : Thread("samplethread", 20) { }
  void init() {
    xprintf("a sample thread skeleton");
    }
  void run(){
    ............
    }
  }
} samplethread;
```

Listing 5.4: Thread skeleton frame in RODOS

If we examine the skeleton of a thread, you will observe that the code inside the virtual method *run* determines what actions the thread will perform. The start address of *run* method can be passed to another core when the thread is required to be executed in another core. So, by this way, the thread initialization and execution can be separately performed on different cores. In fact, a small prototype was implemented based on this idea. Indeed, it works in terms of concurrent execution. However, some faults should not be ignored:

- Pre-emption execution is not fully supported. The context switching code cannot be performed smoothly in secondary core because all threads in secondary core are missing the necessary initialization processes.
- Low execution efficiency. All scheduling work will be done by primary core, the only job of secondary core is executing code of the *run* methods that

supposed to be executed on secondary core. This job distribution might result in an unbalancing workload: primary core may running a heavy thread while secondary core just waiting a new thread from primary core's scheduler.

So, due to the difficulties of the SMP implementation, the design decision was forced to change from the SMP model to the AMP model. All the rest parts of this section are only for the AMP implementation.

5.3 Multicore Awareness

5.3.1 BootRom and First-Stage Boot Loader

The BootROM is the first piece of code to run on a processor immediately after the system resets [35]. The BootROM only executes on core0, copy the boot image First-Stage Boot Loader (FSBL) from the boot device to the OCM, and jump the code execution to the OCM.

The FSBL executes after the BootROM is done. The FSBL operations [35]:

- Initialize the PS using the PS7 Init data which is generated by Xilinx IDE.
- Program the SoC‘s PL if the appropriate bit file is provided.
- Load either a Second-Stage Boot Loader (SSBL) if a Linux operating system is used or loads an application into DDR SDRAM.
- Start the execution of the SSBL, or the application loaded.

In order to keep the complexity of the system implementation to a minimum, the RODOS operation system is treated as an entire application to get rid of the complicated SSBL.

5.3.2 Operation Systems

As AMP required, two RODOS operation system instances are executed on core0 and core1 respectively. Core0 is selected as a primary core for initializing all shared resources and starting up the core1. A modified Board Support Package (BSP), named *standalone_ v3_ 10_ a* is used for booting the second core in a correct order. A pre-processor defined constant *USE_ AMP* is imported to support this option. This constant prevents core1 over-initializing the shared resources.

Another benefit of using this constant is that the Memory Management Unit (MMU) mapping is adjusted to create an alias of memory where the physical memory address 0x2000000 is mapped to the virtual address 0x00000000, which is done by the file *boot.s* [39].

RODOS in core0 does in the following order:

1. Disabling cache for high 64KB OCM in the address range of 0xFFFF0000 to 0xFFFFFFFF. The purpose of this will be detailed discussed in the section 5.3.3.
2. Initializing the Interrupt Control Distributor (ICD)
3. Writing the value of core1's start address to the address 0xFFFFFFF0.
4. Sending a Set Event (SEV) instruction to core1.
5. Waiting the response from core1.
6. Starting own RODOS operating system instance.

After the PS powers up, core1 will be jumped to 0xFFFFFE00, where is a predefined piece of code in OCM. This code merely is an infinite loop that executes Wait-For-Event (WFE) instructions [40]. An event will be notified by changing the value of the address 0xFFFFFFF0. If this address contains a positive value, core1 will leave this continuous loop and jump to the start address of its application which written by core0 in advance [40].

Before core1 starts running, core0 should copy the start address of core1 (0x02000000) to the address 0xFFFFFFF0. Then, core0 executes a Send-Event (SEV) instruction to wake core1 up. core1 will get the value 0x02000000 from the address 0xFFFFFFF0 and jump to this address to execute [39].

After the wake-up signal from core0, core1 will start fetching the instructions from the address 0x02000000. Operating system in core0 does in the following order:

1. Disabling cache for high 64KB OCM in the address range of 0xFFFF0000 to 0xFFFFFFFF.
2. Initializing the Interrupt Control Distributor and interrupt subsystem
3. Notifying to core0 that core1 has already started.
4. Starting own RODOS operating system instance.

5.3.3 Inter-Core Communication and Synchronization

In AMP architecture, both cores have their private memory spaces. Therefore, communication APIs should be provided for data transmission over cores. A communication interface was designed dedicated for this purpose. Both cores coordinated access to a reserved area called *common communication block* when performing sending and receiving operation. The common communication block was defined as a struct:

```c
typedef struct {
  volatile bool isDataFromCore0Ready;
  volatile bool isDataFromCore1Ready;
  volatile bool isSecondCoreStart;
  volatile void *dataFromCore0;
  volatile void *dataFromCore1;
} cpuComBlock;
```

Listing 5.5: Common communication block struct definition

The last two pointers **dataFromCore0* and **dataFromCore1* are used as two channels for data transmission from core0 and core1 respectively. The reason for defining these pointers as *void* type is allowing arbitrary types of data can be transmitted. However, instead of using a shared pointer for holding the data from two directions, two pointers that each of them serves as a uni-direction channel dedicated for one core to avoid the contention when both cores attempt to send data to the opposite cores. The first two members *isDataFromCore0Ready* and *isDataFromCore1Ready* act as the flag signals to indicate whether the expected data are available or not. One of them is supposed to be set to true immediately after the associated sending operation performed. The third member is not designed for communication purpose, it is used for synchronization between two cores. As discussed in section 5.3.2, the primary core will wait until the secondary core changes the value of *isSecondCoreStart*, which implies the secondary core has already wakened up.

```c
  //declare a shared common block printer for multicore communication
  cpuComBlock * cpuComBlockPtr = (cpuComBlock *)(0xFFFF0000);
```

Listing 5.6: Common communication block declaration

A shared common communication block pointer that declared explicitly at the address 0xFFFF0000 is defined to reach the consistency among both cores. The value 0xFFFF0000 is the start address of OCM. One of the reasons we are not using DDR memory is because DDR memory has a higher latency for accesses and also is less deterministic due to the additional refresh cycles in the background.

Several communication APIs are provided for application engineers to transmit data over cores. In core0, for instance, *waitAndSendDataToCore1()* provides one option for sending data from core0 to core1. By this function, core0 is forced to wait if the previous data still exists in the common communication block. The sending operation is only performed when the channel is free. For contrast, the sending action in *dontWaitAndSendDataToCore1()* will be performed as soon as this function is called. The previous data in the channel will be overwritten by the new data in this case. In the receiving function *revDataFromCore1()*, an argument *waiting_time* that represents how much time should wait for is required to passed in. The function will return *NULL* if no data available during this explicit period. A challenge that discussed in Section 2.4.4 arises from the fact that ARM processors normally will re-order the instructions internally either at compile-time or run-time to reach higher performance. A "memory barrier" micro applied here to force the processors to execute instructions in the order that the code present. Moreover, hardware interrupts are disabled before accessing the shared memory space.

```
void waitAndSendDataToCore1(void * data);
void dontWaitAndSendDataToCore1(void * data);
volatile void* revDataFromCore1(int64_t waiting_time);
```

Listing 5.7: Multi-core communication APIs

5.3.4 Interrupt Handling

As a dual-core processor, both cores should have the ability to handle the interrupt. However, only primary core is selected to receive an interrupt from PL as for the demo purpose here.

In order to implement interrupt handling correctly, two functions are needed:

- Interrupt setting up function. This function configures and enables a PL interrupt.
- Interrupt service routine function (ISR). This function defines the actions when the interrupt occurs.

```
// Initialize driver instance for PL IRQ
    PlIrqInstancePtr.Config.DeviceId = PL_IRQ_ID;
    PlIrqInstancePtr.Config.BaseAddress = IRQ_PCORE_GEN_BASE;
    PlIrqInstancePtr.IsReady = XIL_COMPONENT_IS_READY;
    PlIrqInstancePtr.IsStarted = 0;
  /*
   * Connect the PL IRQ to the interrupt subsystem so that interrupts
   * can occur
   */
  Status = SetupIntrSystem(&IntcInstancePtr,
          &PlIrqInstancePtr,
          PL_IRQ_ID);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }
```

Listing 5.8: Interrupt handling interface

The code above treats as an interrupt setup routine to initialize PL, connect the PL to the interrupt subsystem and enable the interrupt.

Function *SetupIntrSystem()* setups the interrupt system such that PL interrupt can occur for the peripheral. This feature is application specific since the actual system may or may not have an interrupt controller [14].

```
/*
 * Connect the interrupt handler that will be called when an
 * interrupt occurs for the device.
 */
Status = XScuGic_Connect(IntcInstancePtr, IntrId,
       (Xil_ExceptionHandler)PlIntrHandler,
       PeriphInstancePtr);
if (Status != XST_SUCCESS) {
  return Status;
}
/*
 * function for interrupt handler
 *
 */
static void PlIntrHandler(void *CallBackRef)
{
  XPlIrq *InstancePtr = (XPlIrq *)CallBackRef;
  /*
  * Clear the interrupt source
  */
  Xil_Out32(InstancePtr->Config.BaseAddress, 0);

  irq_count = 1;
}
```

Listing 5.9: Interrupt handler routine

Function *P1INtrHandlier()* is an interrupt handler routine which will be called when an interrupt occurs. As a demo here, the purpose of this handler function is merely turn off the interrupt source and set a global variable equals to 1. This function was registered by the function *SetupIntrSystem()*. The interrupt handler was triggered by PL, a more detail about testing and evaluation will be covered in Chapter 6.

5.4 Results Reproducing

This section provides a step-by-step tutorial for readers to reproduce the results. Although the MicroZed board is used as a target platform, this tutorial is also compatible with the other boards based on the Xilinx zynq-700 SoC.

5.4.1 Tools Setup

The software used for this project are:

- Windows-7 64-bit (also works on 32-bit) with at least 8GB free memory space.
- Xilinx Vivado development kit, version 2013.2. The newly updated version also works fine with minor modification.
- MicroZed Board Awareness archive for Vivado 2013.2 which can be download from *www.microzed.org.*
- MicroZed Zynq PS Preset for Vivado 2013.2 (TCL) which can be download from *www.microzed.org* as well.
- Terminal emulator **Tera Term**, This is for monitoring serial output from the MicroZed board.

The hardware used for this project are:

- Avnet MicroZed board with power supply.
- USB cable (Type A to Micro-USB Type B) for connection between board and host computer.
- JTAG Programming Cable (Platform Cable, Digilent HS1 or HS2 cable) for downloading and debugging purpose.

5.4.2 Producing the hardware

The first step in completing a MicroZed design is to define and build the hardware platform.

Vivado development kit has awareness of several Xilinx development boards built-in, as well as the MicroZed board. Although, version 2013.2 does not have full support for 3rd-party boards built-in, it does have partial support. An XML file must be copied to the proper location for Vivado to find it.

1. Download the board awareness files and unzip on your computer.
2. Launch Vivado by selecting **Start** -> **All Programs** -> **Xilinx Design Tools** -> **Vivado 2013.2** or double chick the icon of Vivado.
3. Click on **Create New Project**.
4. Click the browse icon and navigate to the Project location to your desired project location and click **Select**.
5. Set the Project name to your desired name.
6. The project will be RTL based, so leave the radio button for RTL Project selected. Click **Next**.
7. Next, the Default Part is selected. MicroZed will be an option shows up in the list if the board awareness files installed successfully.

8. Set the Board Vendor to **em.avnet.com** in the next page and single-click the **MicroZed 7010 Board**.
9. Next, a project summary is displayed. Click **Finish**. Now a blank MicroZed project is created, to access the ARM processing system, some necessary IP sources should be added by IP integrator.
10. Select **Create Block Design**, give the block design a name and click **OK**.
11. Click the **Add IP** text in the diagram window. The **Add Sources** window opens. Scroll to the bottom of the window. Find the **ZYNQ7 Processing System** IP, select by double-click. Now the Microzed processing system will appear in the Diagram window.

    As Vivado 2013.2 does not automatically imports the PS configuration. For engineers creating their custom board, they need to configure the entire PS manually. Here, since we used is MicroZed board, a standard TCL file helps us to import the MicorZed PS parameters in a fast way.
12. Select the **Tcl Console** tab at the bottom of Vivado cockpit. Type the directory where you copy the TCL file and type "*source MicroZed_ PS_ properties_ v01.tcl*", click **Enter**.
13. Now, you can see the Zynq PS configuration that was created by the TCL script.
14. Back in the Vivado Block Design, click **Run Block Automation** link at the top of the window and select **processing_ system7_ 1** to connect all block I/O.
15. Now, it is time to validate our design. Click the **Validate Design icon**. If everything is correct, a successful validation window will appear. Click **OK**.
16. The next three steps are not intuitive but must be done. We must generate output products of our MicroZed block design. To do this, we need to switch back to the **Sources** tab, save the project first.
17. In the source window, right-click on **System.bd** and select **Generate Output Products**, click **OK**.
18. Right-click on **System.bd** again and select **Create HDL wrapper**, click **OK**.
19. Once the top-level wrapper is created, select it in the **Sources window** then click **Generate Bitstream** in the Flow Navigator window. The Bitstream generation may take several minutes to complete. When finished, the hardware that needed for further development is done. For allowing this platform available to the Software Development Kit (SDK), this is done by exporting the hardware platform.

20. Select **File** -> **Export** -> **Export Hardware** for SDK.

5.4.3 Building the Source Code

Once a MicroZed Hardware Platform is created and exported from Vivado, next is to build the RODOS source code at the platform and see it operates on hardware.

1. Download the RODOS source code package from the URL: https://github.com/zwdtc/RODOS_ source_ code/releases[1], unzip it to a local directory.
2. Launch SDK by selecting **Start** -> **All Programs** -> **Xilinx Design Tools** -> **Vivado 2013.2** -> **SDK** -> **Xilinx SDK 2013.2**.
3. Select a workspace, click **OK**.
4. Select **File** -> **New** -> **Other** -> **Hardware Platform Specification**, click **Next**.
5. Input a name for the project name.
6. Click browse and select the **System.xml** file generated during the Export process from Vivado. This will be included in the archive provided by the **producing the hardware** section.
7. Since this was a PS-only system, there will not be a Bitstream or BMM. Click **Finish**, notice the PS7 MicroZed hardware platform is now available in the Project Explorer.
8. Now, Select **Xilinx_ tools** -> **repositories** -> **New**, browse to and select the directory of RODOS source code/RODOS_ source_ code-master/Rodos_ zynq/repo. The custom *mid* and *tcl* files in the repository, which contains directives for customizing software libraries and generating Board Support Packages (BSP) for RODOS operating system. Click **Apply** -> **OK**.
9. Repeat the step 8, add the directory of RODOS source code/RODOS_ source_ code-master/Rodos_ zynq_ amp/repo.
10. Now, In SDK select **File** -> **New** -> **Board Support Package**. select hw_ platform_ 0 and pick the ps7_ cortexa9_ 0, which indicates the core0 of processor. Two additional BSPs are available now, select rodos_ zynq for core0 as a primary core. Click **finish**. A RODOS BSP for core0 is created.
11. Repeat the step 10, generate an RODOS BSP for core1 by selecting the rodos_ zynq_ amp BSP package.

---

[1] The source code is temporarily unavailable due to the lack of permission, It will be available in the near future.

12. Now, we need to create two applications based on these two BSPs. In SDK, select **File** -> **New** -> **Application Project**. Select processing core and existing RODOS BSP respectively. click **Finish** at last. Now, source code can be built by SDK.

5.4.4 Running multi-core version of RODOS

We take the testing tasks as top-level applications to demonstrate how to run applications on the multi-core version of RODOS.

1. Copy the file *sync_ Test_ core0.cc* and *sync_ Test_ core1.cc* from the ../RODOS_ source_ code-master/testing_ code, overwrite the template code in application files *core0.cc* and *core0.cc* respectively.
2. Build these two project, if everything is in the correct way, two *elf* images are generated respectively.
3. Create a FSBL project by click **File** -> **New** -> **New Application Project**, name the file and click **Next**, select **Zynq FSBL** and click **Finish**.
4. Build newly created FSBL project.
5. Select **Xilinx Tools** -> **Create Zynq Boot Image**. This will preload the FSBL and applications elf images. The order of the files is important. The FSBL should be loaded before the application.
6. Click **Create Image**.
7. Navigate to the application directory, then go into the newly created **bootimage** directory. Notice that three files have been generated: .bif, .bin, and .mcs.
8. Next, the generated boot file need to be download to the flash memory. Select **Xilinx Tools** -> **Program Flash**. It takes ten seconds to complete.
9. Now, power on the MicorZed board, connect to your host computer by USB-UART cable, open the terminal emulator **Tera Term**, configure with the 115200-8-n-1-n settings.
10. Push the Reset button. Output results should be showed in the terminal.

Chapter 6

6 Testing and Evaluation

6.1 Testing Strategy

As an essential element of this thesis project, software testing provides objective, independent information about the quality of the modified software [41]. Due to the relative complexity of this project implemented, the primary testing strategy is revealing the bugs and errors as early as possible after one development phase was done.

Several test sets were created to test the quality of modified system components, as well as some newly created features. These customized tasks were designed to provide a fast and concrete feedback to ensure the success of implementation made at the end of each development phase. Specifically, four categories of test tasks were designed for this project: interrupt handling test tasks, synchronization test tasks, concurrent execution test tasks and communication test tasks.

6.2 Interrupt Handling Testing

For testing purpose only, a Chipscope Virtual I/O core [42] is included in this test set, in order to provide a simple interrupt source. Engineers could generate interrupt sources towards the PS and measure the latency of interrupt response in an intuitive way by the Chipscope Virtual I/O core. However, engineers should always keep in mind that this virtual core would not exist in a real project, instead, the interrupt might be sourced from the variety of peripheral devices.

The Chipscope Virtual I/O core is configured to connected with the MicroZed's PL. The VIO core provides a mechanism for the user to interact with hardware for the Chipscope Analyzer application. In this design, when the VIO generates an interrupt signal, it forwards to the PS core1_nIRO pin. The ore1_nIRO pin connects directly to core1's Private Peripheral interrupt (PPI), so there is no need the set the shared Interrupt Control Distributor (ICD) for interrupt distribution. Core1 access the

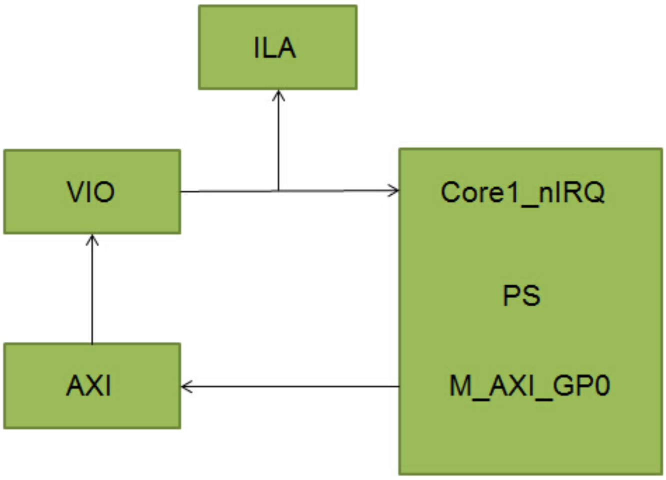


Figure 6.1: PL block diagram

control register to remove the IRQ flag during the Interrupt Service Routine (ISR). A Chipscope AXI core is connected as well for the latency measurement purpose. A Chipscope ILA core is also included in this design to monitor the IRQ signal. After several steps configuration (which will not present here), push the virtual button called *SyncOut[0]* and core1 will service the interrupt. It is notable that every time when virtual button is pressed, an interrupt signal is created, and core1 will perform the instructions defined in the ISR, prior to that, the IRQ signal should be cleaned. However, some interesting noticed that the interrupt response time for the first IRQ and the subsequent IRQ are different. In Figure 6.2, there is a delay of over 65 clocks between the interrupt being asserted and the ISR removing the signal

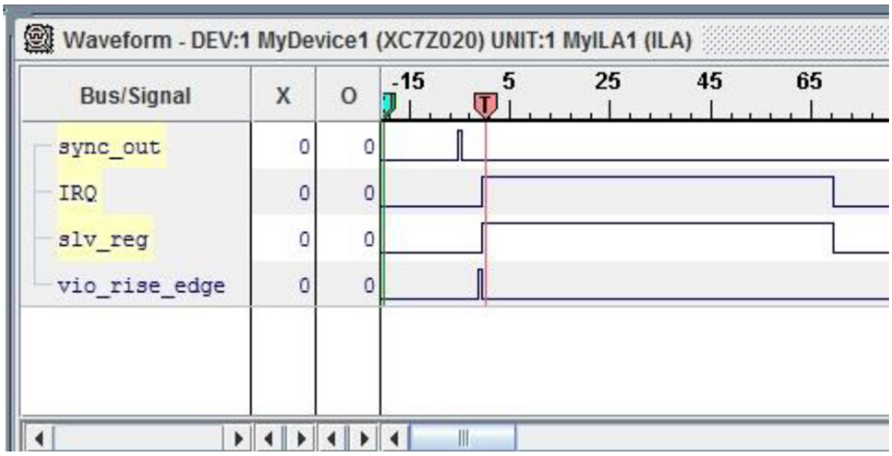


Figure 6.2: Chipscope output capture for first IRQ

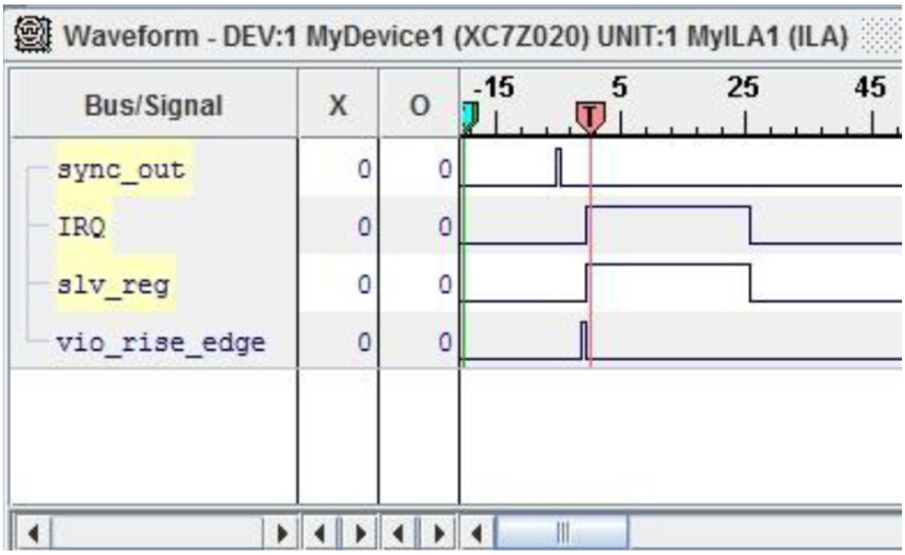


Figure 6.3: Chipscope output capture for sebsequent IRQ

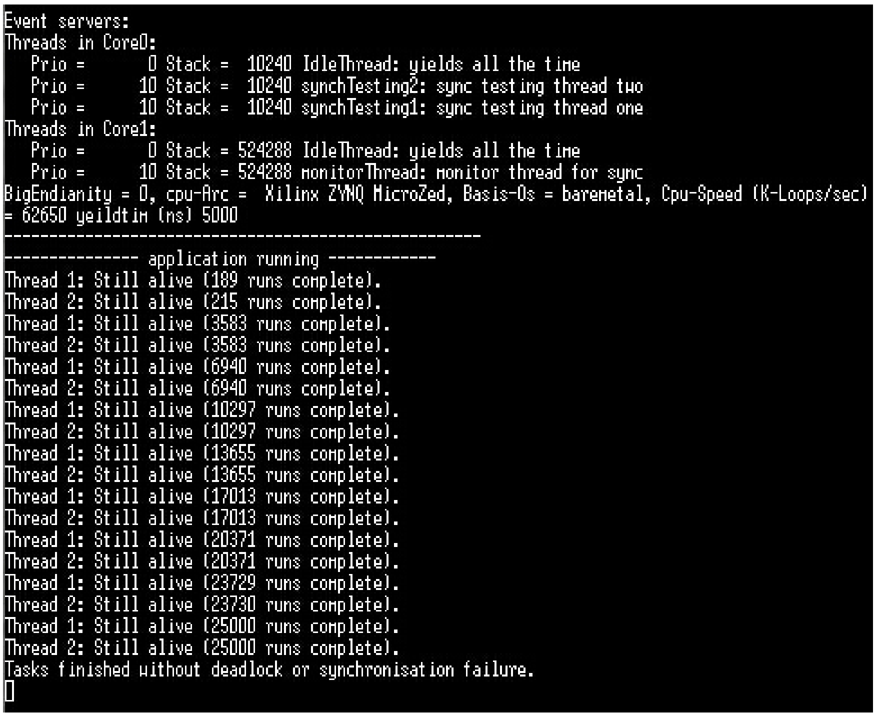
bit. The delay time of subsequent IRQ is around 25 clocks, almost one-third of the first time.

The reason is IRQ service routine being cached. The application which services interrupts is located in DDR memory, when the first interrupt occurs, the instructions for ISR will be read into the cache and executed. After the first IRQ occurs, the service routine will be located in the cache, so fetches of instructions for the routine will be sourced by cache rather than DDR memory, which is slower and less deterministic.

The time difference between the first and later ISR could be reduced by moving the ISR code into non-cached on-chip memory.

6.3 Synchronization Testing

Three test tasks are used for synchronization testing. Two of them serve as slave tasks, both of them did the same thing: obtaining a shared semaphore before entering the critical section, increasing the value of a shared value *shareVar* inside the critical section, and releasing the semaphore after exiting the critical section. Each time when entering the critical section, the *shareVar* is checked whether the value is multiple of 1000. If the answer is no, another variable *failedSync* is set to True to indicate something wrong with the synchronization. By this way, the master task can check the quality of the synchronization mechanism.



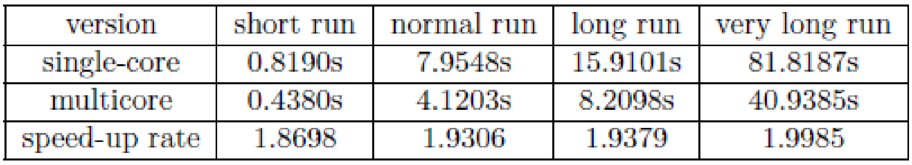
```
Event servers:
Threads in Core0:
   Prio =       0 Stack =  10240 IdleThread: yields all the time
   Prio =      10 Stack =  10240 synchTesting2: sync testing thread two
   Prio =      10 Stack =  10240 synchTesting1: sync testing thread one
Threads in Core1:
   Prio =       0 Stack = 524288 IdleThread: yields all the time
   Prio =      10 Stack = 524288 monitorThread: monitor thread for sync
BigEndianity = 0, cpu-Arc =  Xilinx ZYNQ MicroZed, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec)
= 62650 yeildtim (ns) 5000
-----------------------------------------------------
--------------- application running ------------
Thread 1: Still alive (189 runs complete).
Thread 2: Still alive (215 runs complete).
Thread 1: Still alive (3583 runs complete).
Thread 2: Still alive (3583 runs complete).
Thread 1: Still alive (6940 runs complete).
Thread 2: Still alive (6940 runs complete).
Thread 1: Still alive (10297 runs complete).
Thread 2: Still alive (10297 runs complete).
Thread 1: Still alive (13655 runs complete).
Thread 2: Still alive (13655 runs complete).
Thread 1: Still alive (17013 runs complete).
Thread 2: Still alive (17013 runs complete).
Thread 1: Still alive (20371 runs complete).
Thread 2: Still alive (20371 runs complete).
Thread 1: Still alive (23729 runs complete).
Thread 2: Still alive (23730 runs complete).
Thread 1: Still alive (25000 runs complete).
Thread 2: Still alive (25000 runs complete).
Tasks finished without deadlock or synchronisation failure.
▯
```

Figure 6.4: Synchronization test result

The master task is merely for checking the status of two slave tasks and showing some necessary information over the serial port. The master task also determines the iteration round of the critical section code in two slave tasks. As performing an exhausting test, the iteration round is set to 25000. If the master task notified from the slave tasks that an error of synchronization failure occurred, a warning message would be passed to users. Otherwise, a summary of testing results will be showed after both worker tasks are completed.

6.4 Concurrent Execution Testing

Concurrent execution is one of the most important features of multi-core system, so, particular attention is paid for the concurrency testing. Test cases consisted of two tasks that performed the same function, but one for the test with the original single-core version of RODOS, another for test with modified multi-core version of RODOS. The content of the test is quite simple, only performing repeated memory access (repeated times determined by variable t as an example here, in the real code, by variable *calc1Var* and *calc2Var* respectively) which intent to create a relatively long execution time. In order to explore the relationship between task's execution time and system's performance, each task contains four versions: short run (t = 5 million), normal run (t = 50 million), long run (t = 100 million) and very long run (t = 500 million). The execution time of each version is listed below:



version	short run	normal run	long run	very long run
single-core	0.8190s	7.9548s	15.9101s	81.8187s
multicore	0.4380s	4.1203s	8.2098s	40.9385s
speed-up rate	1.8698	1.9306	1.9379	1.9985

Figure 6.5: Concurrent execution test result

As we observed from the table, the performance on multi-core version improved 87% in short run test. This improvement increased in the normal run test and long run test, and reached a peak by a factor 1.9985 in a very long run test. So, the efficiency provided by multi-core version of RODOS is closely related to the task's execution time, and the more execution time needed, the more efficiency can be obtained.

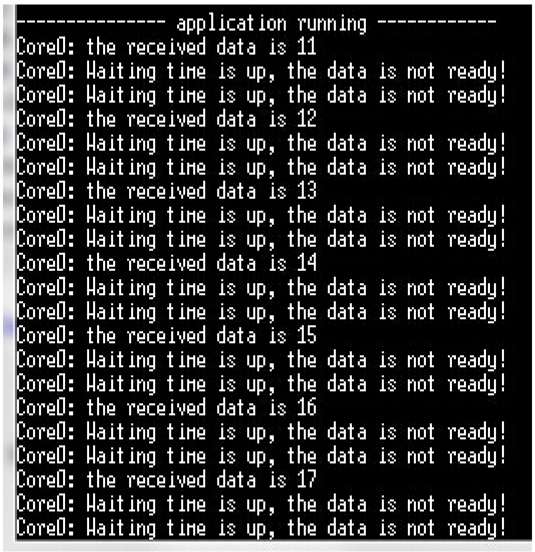
6.5 Communication Testing

A sending task and a receiving task are designed for the communication test. A local integer variable T which defined by the sending task in core1 will be transfer to core0 in every three seconds. To identify the received data is indeed moved from core0, the value of T intentionally increased by one after one sending operation is performed. In core1, the function *revDataFromCore1()* is called every one second, and the receiving function only forced to wait up to 10000 nanoseconds if the data is not available. The expecting result should be like that: core0 attempts to collect new data by every second, however, only one out of three attempts will succeed. The result of communication testing is showed in Figure 6.6.

The result verified our expectation. core0 will perform two failure attempts before one successful try in every three seconds. A message will be prompt out to indicate the failure of these attempts.

6.6 Evaluation

The time distribution of this project is illustrated in Figure 6.7. A considerable proportion of time was spent on the software development. One reason is that too many efforts made on SMP implementation, which produces no result in the end. Another reason might be that the majority part of hardware design was



```
-------------- application running ------------
Core0: the received data is 11
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
Core0: the received data is 12
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
Core0: the received data is 13
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
Core0: the received data is 14
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
Core0: the received data is 15
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
Core0: the received data is 16
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
Core0: the received data is 17
Core0: Waiting time is up, the data is not ready!
Core0: Waiting time is up, the data is not ready!
```

Figure 6.6: Communication test result

automatically generated by Xilinx tool sets. All the processes were standardized that allows users to construct desired hardware in a very quick way. However, several factors extend the period of software development: unfamiliar with the process when porting original RODOS to one core, unfamiliar with the *tcl* file and *mld* file when implementing a customized BSP by Xilinx SDK, some ad-hoc events happened[1]. The initial outcome was supposed to be an SMP solution. However, due to some reasons discussed in section 5.2, a new version of RODOS was implemented in the AMP model. Based on the testing results, AMP solution offers an improved performance by around 90%, which very close to the theoretical limitation. Another interesting

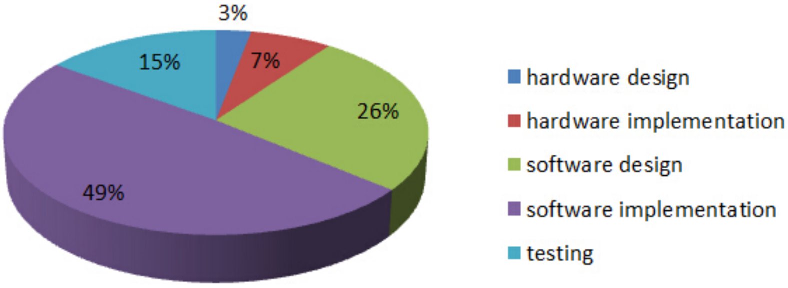


Figure 6.7: Time distribution diagram

---

[1] Such as preparing the intermediate presentation, retaking an exam

pattern observed from the test results is that the more execution time required for tasks, the better performance could gain, although the reason for this relationship is still not fully understand and need to be proved by the formal verification in the future.

6.6.1 Requirements Evaluation

All primary requirements are met! Two secondary requirements are not met due to the time restriction.

Requirement **3.2.1** was met. The single-core applications were demonstrated in the intermediate presentation, although not illustrated in this report.

Requirement **3.2.2** was met. RODOS was extended to AMP architecture successfully. The modified version offers an improvement performance around 190% over the original version.

Requirement **3.2.3** was met. A new lock-free communication and synchronization approach was implemented.

Requirement **3.2.4** was met. The *testing-code* package were provided.

Requirement **3.2.5** was not met. Theoretically, the AMP version of RODOS is easy to scale to n-cores since the key elements of operating system will keep the same of the original version. However, this point was not fully verified by the experiment due to the limited time left after fulfilling the primary requirement.

Requirement **3.3.1** was met. A tutorial is demonstrated in Section 5.4. The source code of modified RODOS is available via git at https://github.com/zwdtc/RODOS_ source_ code[2].

Requirement **3.3.2** was not meet, the total amount of time used for this project is six months, additional one month was dedicated for writing final report.

---

[2] Temporarily unavailable due to the lack of permission, will available in the near future

7

Chapter 7

Conclusions and Future Improvement

7.1 Summary

The current work revealed the possibility to extend RODOS to a particular dual-core platform. A comprehensive literature review and background study was performed before the real job starts. A principle design decision was made that system should be configured as symmetric multiprocessing model. The asymmetric multiprocessing model was set as a backup solution. The differences between these two models were detailed discussed in Chapter 4. However, several bottlenecks occurred during the SMP implementation, the unexpected difficulty forced the design decision switched from SMP model to AMP model, in which each core runs its operating system instance. This decision to a large extent guaranteed the whole project could go forward to catch up the original schedule.

A major disadvantage of AMP model is that more memory space required and inflexible of task scheduling than in the equivalent SMP model. The boot loader is modified to support multi-core boot sequence. Cores are classified as the primary core and secondary core. The primary core is on duty for global resources initialization and starting secondary core. The secondary code only needs to take care of its private resources. Moreover, since variables defined in one core are invisible from another core, a common communication block is designed to accommodate multi-core communication and synchronization issues. Several test sets are created to evaluate the performance of modified version of RODOS. In result, an around 90% improvement performance is gained.

7.2 Future Work

7.2.1 SMP Model Implementation

The implementation of the project is limited to the AMP model. However, considering the advantages offered by SMP model (less memory consumption, more intelligent scheduler), the SMP implementation is definitely worth to try in the future.

7.2.2 Formal Analysis of Real-Time Properties

A formal method should be applied to investigate the real-time feature of modified multi-core version of RODOS. The worst-case execution time (WCET) should be determined by an upper bound of system's execution time. Simple-core architectures allow WCET determination by using static analysis techniques[29, p, 26]. However, the lack of knowledge on the MicroZed processor (such as how the out-of-order instructions performed) may have some influence on the WCET estimation. In the multi-core architecture, performing this analysis even became harder since execution time of one application on one core might rely on the application executed on another core.

7.2.3 Comprehensive Testing

A comprehensive testing strategy can be applied to provide a higher confidence on software synchronization before it used in the real space application.

7.2.4 Implementation Scales to N-Core

Because of the restrictions of hardware, only dual-core processor is used for this project. In terms of features offered by AMP model, the operating system can be scaled to n-core processor without too much modification. However, communication and synchronization APIs should be re-designed to adapt to n-core architecture.

Bibliography

[1] S. J. O. Phillip A. Laplante, *Real-Time Systems Design and Analysis.* Wiley, 2012.

[2] Wikipedia, “Embedded operating system.” *Available at http://en.wikipedia.org/wiki/Embedded_operating_system*, 2014. Last visited on 17, Nov 2014.

[3] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, T. Peng, B. Weps, G. Fey, and A. Gerndt, “OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft,” *IEEE Aerospace Conference*, 2014.

[4] J. Mistry, “FreeRTOS and multicore,” Master’s thesis, Department of Computer Science University of York, 2011.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach, Fourth Edition.* Elsevier, 2007.

[6] Wikipedia, “Multi-core processor.” *Available at http://en.wikipedia.org/wiki/Multi-coreprocessor*, 2014. Last visited on 10, Oct 2014.

[7] Intel Corporation, “Enhanced Intel Speedstep Technology for the Intel Pentium M Processor,” 2004.

[8] D. Geer, “Chip makers turn to multicore processors,” *IEEE Computer Society*, vol. 38, pp. 11–13, 2005.

[9] LynuxWorks, “Hypervisors ease the world of multicore processors.” *Available at http://http://rtcmagazine.com/articles/view/101663*, 2013. Last visited on 27, may 2015.

[10] M. Gondo, “Blending asymmetric and symmetric multiprocessing with a single os on arm11 mpcore,” *Information Quarterly*, vol. 5, 2007.

[11] “Understanding multicore basics: AMP and SMP.” *Available at http://www.eetasia.com/ART_8800699683_499495_TA_d8cee7a6.HTM*. Last visited on 2, Sep 2014.

[12] ARM, *ARM Cortex-A Series, Programmer’s Guide*, 2013.

[13] “Dual-core arm cortex-a9 mpcore processor,” 2014. Last visited on 10, Oct 2014.

[14] *Zynq-7000 All Programmable SoC,Reference Manual.*

[15] Wikipedia, "Round-robin scheduling." *Available at* `http://en.wikipedia.org/wiki/Round-robin_scheduling`, 2014. Last visited on 12, Oct 2014.

[16] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multicore processors," *CommunicAtions of the ACM*, vol. 53, no. 1, pp. 51–52, 2010.

[17] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, pp. 46–61, 1973.

[18] wikipedia, "Round-robin scheduling." *Available at* `http://en.wikipedia.org/wiki/Round-robin_scheduling`, 2014. Last visited on 4, Dec 2014.

[19] G. Peterson, "Myth about the mutual exclusion problem," in *Information Processing Letters*, vol. 12, (Drpartment of Computer Science, University of Rochester, Rochester, NY 14627, USA.), 1981.

[20] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, pp. 483–485, 1967.

[21] Wikipedia, "Linearizability." *Available at* `http://en.wikipedia.org/wiki/Linearizability`, 2014. Last visited on 27, Nov 2014.

[22] *Intel 64 and IA-32 Architectures Software DeveloperâĂŹs Manual*, 2011.

[23] "Bus lock." *Available at* `https://software.intel.com/sites/products/documentation/doclib/iss/2013/amplifier/lin/ug_docs/GUID-004AFC99-EBBA-43D5-98A9-43A97997F9B5.htm`. Last visited on 15, Nov 2014.

[24] Wikipedia, "Wikipedia cache coherence." *Available at* `http://en.wikipedia.org/wiki/Cache_coherence`, 2014. Last visited on 25, Nov 2014.

[25] ARM, *Cortex-A9 MPCore Technical Reference Manual*, 2011.

[26] J. Preshing, "Memory ordering at compile time." *Available at* `http://preshing.com/20120625/memory-ordering-at-compile-time/`, 2012. Last visited on 11, Nov 2014.

[27] "Memory barriers." *Available at* `http://preshing.com/20120710/memory-barriers-are-like-source-control-operations`. Last visited on 22, Sep 2014.

[28] D. Lea, "The jsr-133 cookbook for compiler writers." *Available at* `http://g.oswego.edu/dl/jmm/cookbook.html`, 2011. Last visited on 29, Nov 2014.

[29] T. AVIONICS, "The use of multicore processors in airborne systems," tech. rep., 2011.

[30] "Rodos real time kernel design for dependability." *Available at* `http://www.dlr.de/irs/en/desktopdefault.aspx/tabid-5976/9736_read-19576/`, 2013. Last visited on 27, Nov 2014.

[31] "Rodos (operating system) online source." *Available at* `http://en.wikipedia.org/wiki/Rodos_(operating_system)`. Last visited on 12, Oct 2014.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System concepts.* Wiley, 2005.

[33] D. C. Schmidt and C. O.Ryan, "Patterns and performance of distributed real-time and embedded publisher/subscriber architectures," *Journal of Systems and Software*, vol. 66, pp. 213–223, 2002.

[34] ARM, "Arm mpcore boot and synchronization code," tech. rep., 2009.

[35] A. Taylor, "Figuring out the microzed boot loader âĂŞ adam taylor microzed chronicles, part 5," 2014. Last visited on 10, Oct 2014.

[36] Wikipedia, "Processor affinity." *Available at* `http://en.wikipedia.org/wiki/Processor_affinity`, 2014. Last visited on 7, Oct 2014.

[37] D. Dhanalaxmi and V. R. Reddy, "Implementation of secured data transmission system on customized zynq soc," *International Journal of Science and Research (IJSR)*, 2012.

[38] B. Andre, "Porting the real-time operating system rodos to sparc v9," 2012.

[39] J. McDougall, "Simple amp: Bare-metal running on both zynq cpus," tech. rep., 2012.

[40] *Zynq-7000 All Programmable SoC Technical Reference Manual*, 2014.

[41] C. Kaner, "Quality assurance institute worldwide annual software testing conference," 2006.

[42] *ChipScope Pro Software and Cores User Guide*, 2012.