BENCHMARKING OF XILKERNEL, FREERTOS AND μC/OS-II ON THE SOFT PROCESSOR PLATFORM MICROBLAZE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÖKHAN UĞUREL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JUNE 2012

Approval of the thesis:

BENCHMARKING OF XILKERNEL, FREERTOS AND μC/OS-II ON THE SOFT PROCESSOR PLATFORM MICROBLAZE

submitted by **GÖKHAN UĞUREL** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. İsmet Erkmen
Head of Department, **Electrical and Electronics Engineering** ________________

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Supervisor, **Electrical and Electronics Engineering Department** ________________

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr. Ece Schmidt
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering Dept., METU ________________

Dr. Atilla Özgit
Computer Engineering Dept., METU ________________

Date: ________________

Name, Last Name:	GÖKHAN UĞUREL
Signature :	

ABSTRACT

BENCHMARKING OF XILKERNEL, FREERTOS AND μC/OS-II ON THE SOFT PROCESSOR PLATFORM MICROBLAZE

Uğurel, Gökhan
M.Sc., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Cüneyt Bazlamaçcı

June 2012, 78 pages

In real time embedded systems, more and more developers are choosing the soft processor option to save money, power and area on their boards. Reconfigurability concept of the soft processor gives more options to the designer, also solving the problem of processor obsolescence. Another increasing trend is using real time operating systems (RTOSs) for microprocessors or microcontrollers. RTOSs help software developers to meet the critical deadlines of the real time environment with their deterministic and predictable behaviour. Providing service APIs and fast response times for task management, memory and interrupts; RTOSs decrease the development time of on going, and also future, projects of software developers. Comparing RTOSs on RTOS-specific benchmark criteria, called RTOS benchmarking in the literature, helps software developers to choose the appropriate RTOS for their requirements and provokes RTOS companies to strengthen their products on areas where they are weak. This study will compare three popular RTOSs on Xilinx's soft processor platform MicroBlaze. Xilkernel, μC/OS-II and FreeRTOS are selected among nine available RTOSs for MicroBlaze and are compared against critical RTOS benchmarking criteria, which are task preemption time, task preemption time under load, get/release semaphore time, pass/receive message time, get/release fixed sized dynamic memory time, UART RS-422 message interrupt serving time, RTOS initialization time and memory footprint data. Results are interpreted using architectural concepts of the RTOSs considered.

Keywords: Soft Processors, Real Time Systems, RTOS Benchmarking, MicroBlaze

ÖZ

SANAL İŞLEMCİ MICROBLAZE ÜZERİNDE XILKERNEL, FREERTOS VE μC/OS-II KARŞILAŞTIRMASI

Uğurel, Gökhan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Cüneyt Bazlamaççı

Haziran 2012, 78 sayfa

Gerçek zamanlı gömülü sistemlerde, elektronik kartlar üzerinde maliyet, güç ve yer bakımından tasarruf etmek isteyen tasarımcılar sanal işlemcileri daha sık kullanmaya başladılar. Sanal işlemcilerin tekrar tekrar biçimlendirilebilmesi ve yapılandırılabilmesi tasarımcılara daha fazla seçenek sunmakta ve tasarımcıların işlemcilerin artık kullanılmaz olması gibi sorunlarla karşılaşmalarını önlemektedir. Bir diğer yükselen eğilim de, işlemciler üzerinde gerçek zamanlı işletim sistemi (GZİS) koşturmaktır. Bu tip işletim sistemleri, gerçek zamanlı bir ortamın getirdiği kritik zamanlama gereksinimlerini karşılamada kararlı ve tahmin edilebilir bir davranış sergiler. Görev, bellek ve kesme yönetimi için sağladıkları komut kümeleri ve hızlı tepki süreleri ile GZİS'ler projelerde yazılım geliştirme sürelerini önemli ölçüde düşürmektedir. GZİS karşılaştırma çalışmaları, tasarımcıya gereksinimlerine uygun işletim sistemini tercih etmesinde zaman kazandıracak, ayrıca GZİS firmalarını eksik olan alanlarında uyararak tasarımcılar için daha yüksek performanslı işletim sistemlerinin geliştirilmesinde rol oynayacaktır. Bu çalışma, Xilinx firmasının sanal işlemcisi MicroBlaze üzerinde üç popüler GZİS'yi karşılaştırmaktadır. MicroBlaze desteği veren dokuz GZİS arasından seçilen Xilkernel, μC/OS-II ve FreeRTOS, görev değişimi zamanı, yüklü çalışmada görev değişimi zamanı, semafor alma/bırakma zamanı, mesaj yollama/alma zamanı, sabit boyutlu dinamik bellek alma/bırakma zamanı, RS-422 seri mesaj kesme işleme zamanı, GZİS ilklendirme zamanı ve bellek ayakizi verilerine göre karşılaştırılmış, sonuçlar işletim sistemlerinin mimari yapılarına ve özelliklerine göre yorumlanmıştır.

Anahtar Kelimeler: Sanal İşlemciler, Gerçek Zamanlı Sistemler, Gerçek Zamanlı İşletim Sistemleri Karşılaştırmaları, MicroBlaze

To My Family...

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor Assoc. Prof. Dr. Cüneyt Bazlamaçcı for his guidance and support in my study. His inspiring suggestions and meticulous feedback in every step of this thesis enabled me to write it and made it an invaluable experience for me. It has been a pleasure to write this thesis under his guidance.

I would also wish to express my sincere gratitude to my thesis committee members Prof. Dr. Semih Bilgen, Assoc.Prof.Dr. Ece Schmidt, Assoc.Prof.Dr. İlkay Ulusoy and Dr. Atilla Özgit as they kindly accepted to share their invaluable comments and helpful suggestions with me.

I also thank to Emre Turgay and Murat Kalkan in ASELSAN who have contributed to improve myself in embedded software and given me valuable advice in every step of my thesis work. Also, my dear friend İlker Erçin, you were there with me during the hard and stressful days.

I deeply express my gratitude to Fatih İzciler who is the designer of the target platform that is used on this study and I am also grateful to all my other colleagues for their encouragement and support.

I express my dearest thanks to Burcu Uzun who did not leave me alone getting through the hardest times. Her presence and her belief in me have been reassuring throughout this study. I am deeply indebted to her for her understanding, patience, respect in what I am doing and encouragement.

Last but not least, most special thanks and love go to my family, my mother Ayşe and my sister Nilüfer, who have supported me in everything I have done in my life. It could have been impossible to write this thesis without their love and support. They are the true possessors of my success.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xiv

LIST OF FIGURES xv

CHAPTERS

1 INTRODUCTION 1

1.1 BACKGROUND 1

1.2 APPROACH 3

1.3 RELATED WORK 4

1.4 OUTLINE 5

2 SOFT PROCESSORS 6

2.1 ADVANTAGES 7

2.1.1 CONFIGURABILITY 7

2.1.2 LOW-COST 7

2.1.3 HARDWARE SOLUTIONS TO SOFTWARE PROBLEMS 8

2.2 DISADVANTAGES 8

2.2.1 DESIGN TOOLS 8

2.2.2 JTAG CHAIN PROBLEMS 9

2.3 PROPRIETARY AND OPEN-SOURCE SOFT PROCESSORS . . . 9

2.3.1 PROPRIETARY SOFT PROCESSORS 9

2.3.2 OPEN-SOURCE SOFT PROCESSORS 10

2.4 SAMPLE APPLICATIONS USING SOFT PROCESSORS 10

3 REAL TIME OPERATING SYSTEMS 11

3.1 ADVANTAGES 13

3.1.1 RESOURCE MANAGEMENT 13

3.1.2 HARDWARE SUPPORT 14

3.1.3 TIME TO MARKET 14

3.2 DISADVANTAGES 14

3.2.1 PERFORMANCE OVERHEAD 14

3.3 EXAMPLE RTOSs AND SAMPLE APPLICATIONS 15

3.4 ARCHITECTURE OF AN RTOS 15

4 XILKERNEL 17

4.1 FEATURES 17

4.1.1 SCHEDULING 17

4.1.2 SYSTEM APIs 18

4.1.3 INTERRUPT HANDLING 19

4.2 LAUNCHING XILKERNEL 20

5 μC/OS-II 22

5.1 FEATURES 22

5.1.1 SCHEDULING 22

5.1.2 SYSTEM APIs 23

5.1.3 INTERRUPT HANDLING 25

5.2 LAUNCHING μC/OS-II 25

6 FREERTOS 27

6.1 FEATURES 27

6.1.1 SCHEDULING 27

6.1.2 SYSTEM APIs 28

6.1.3 INTERRUPT HANDLING 28

6.2 LAUNCHING FREERTOS 30

7 TARGET PLATFORM AND CONFIGURATION OF SELECTED RTOSs . 32

7.1 REASONS FOR SELECTION OF XILKERNEL, μC/OS-II AND FREERTOS 32

7.2 A BRIEF COMPARISON OF THREE RTOSs 33

7.3 TARGET PLATFORM 35
7.4 CONFIGURATION OF SELECTED RTOSs 37
8 BENCHMARKING CRITERIA 39
8.1 TASK PREEMPTION TIME 40
8.2 TASK PREEMPTION TIME UNDER LOAD 42
8.3 GET/RELEASE SEMAPHORE TIME 43
8.4 PASS/RECEIVE MESSAGE TIME 44
8.5 GET/RELEASE FIXED SIZED DYNAMIC MEMORY TIME . . . 45
8.6 UART RS-422 MESSAGE INTERRUPT SERVING TIME 47
8.7 RTOS INITIALIZATION TIME 48
8.8 MEMORY FOOTPRINT 49
9 RESULTS 51
9.1 TASK PREEMPTION TIME 52
9.2 TASK PREEMPTION TIME UNDER LOAD 52
9.3 GET/RELEASE SEMAPHORE TIME 54
9.4 PASS/RECEIVE MESSAGE TIME 55
9.5 GET/RELEASE FIXED SIZED DYNAMIC MEMORY TIME . . . 55
9.6 UART RS-422 MESSAGE INTERRUPT SERVING COUNT 56
9.7 RTOS INITIALIZATION TIME 57
9.8 MEMORY FOOTPRINT 58
9.9 OVERALL 58
10 CONCLUSION AND FUTURE WORK 60

REFERENCES 62

APPENDICES

A APPENDIX A 65
A.1 XILKERNEL CODE SNIPPETS 65
A.1.1 GET/RELEASE SEMAPHORE TIME 65
A.1.2 PASS/RECEIVE MESSAGE TIME 66
A.1.3 GET/RELEASE FIXED SIZED MEMORY TIME 67
A.1.4 UART RS-422 MESSAGE INTERRUPT SERVING TIME 68

A.1.5 RTOS INITIALIZATION TIME 68

B APPENDIX B 69

B.1 μC/OS-II CODE SNIPPETS 69

B.1.1 TASK PREEMPTION TIME 69

B.1.2 GET/RELEASE SEMAPHORE TIME 70

B.1.3 PASS/RECEIVE MESSAGE TIME 71

B.1.4 GET/RELEASE FIXED SIZED MEMORY TIME 72

B.1.5 UART RS-422 MESSAGE INTERRUPT SERVING TIME 73

B.1.6 RTOS INITIALIZATION TIME 73

C APPENDIX C 74

C.1 FREERTOS CODE SNIPPETS 74

C.1.1 TASK PREEMPTION TIME 74

C.1.2 GET/RELEASE SEMAPHORE TIME 75

C.1.3 PASS/RECEIVE MESSAGE TIME 76

C.1.4 GET/RELEASE FIXED SIZED MEMORY TIME 77

C.1.5 UART RS-422 MESSAGE INTERRUPT SERVING TIME 78

C.1.6 RTOS INITIALIZATION TIME 78

LIST OF TABLES

TABLES

Table 5.1 System Services of μC/OS-II 24

Table 7.1 Available RTOSs on MicroBlaze 32

Table 7.2 First Look on the Three RTOSs 34

Table 8.1 Task Preemption Time Measurement Method 40

Table 8.2 API Calls for Task Preemption Time Measurement 41

Table 8.3 API Calls for Get/Release Semaphore Time Measurement 44

Table 8.4 API Calls for Pass/Receive Message Time Measurement 45

Table 8.5 API Calls for Get/Release Fixed Size Dynamic Memory Time Measurement 47

Table 8.6 UART RS-422 Message Interrupt Serving Time Method 48

Table 9.1 Benchmarking Results 51

Table 9.2 RTOS Initialization Functions 58

LIST OF FIGURES

FIGURES

Figure 1.1 FPGA Designs With/Without Embedded Processors 2

Figure 3.1 Architecture of an Application Using an RTOS 12

Figure 3.2 RTOS Usage Statistics 13

Figure 4.1 Xilkernel Architecture 18

Figure 4.2 Scheduling Architecture of Xilkernel 19

Figure 4.3 Interrupt Handling in Xilkernel 20

Figure 4.4 Building Applications in Xilkernel with One Separate File Method 21

Figure 5.1 Architecture of μC/OS-II 23

Figure 5.2 Interrupt Handling in μC/OS-II 25

Figure 6.1 Scheduling Algorithm of FreeRTOS 29

Figure 6.2 Interrupt Handling Algorithm in FreeRTOS 30

Figure 7.1 Target Board 35

Figure 7.2 IP Cores Inside FPGA 36

Figure 7.3 FPGA and External Chip Connections 37

Figure 8.1 Measurement Method and the Drawback 42

Figure 8.2 Measurement Methods for Get/Release Semaphore Time 43

Figure 8.3 Measurement Methods for Pass/Receive Message Time 45

Figure 8.4 Measurement Methods for Get/Release Fixed Sized Dynamic Memory Time 46

Figure 8.5 Interrupt Serving Flow 48

Figure 8.6 RTOS Initialization Time Measurement 49

Figure 9.1 μC/OS-II's OS Tick Code 53

Figure 9.2 Task Preemption Count in 10 ms while Increasing Total Task Number . . . 54

Figure 9.3 Critical Region Concept 57

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Nowadays, more and more embedded systems are using FPGAs (Field Programmable Gate Arrays) to control and process data by making use of the parallelism and flexibility concepts inherent in FPGAs. Designers using FPGAs can choose exact peripherals needed for the requirements of the application, having also the freedom of changing them during the design process. Assume that a system designer prepares the requirements of an incoming project before the start of the design phase of the product as usual. The possibility of the change of these requirements is very high after the design phase has started since the customers can frequently change their minds. If system designers have planned using microprocessors at the beginning of the project, software engineers may experience difficulties to fulfill the incoming requirements because of the inflexible hardware architecture of the microprocessor. If FPGAs have been used on the other hand, software of the product will be protected and will not be processor dependant, and the designers will not suffer from processor obsolescence. Also, if performance is an issue, microprocessors run at clock rates roughly 4 or 5 times faster than FPGAs. But, FPGAs can perform tens of thousands of operations at each clock cycle because of their parallelism while microprocessors can perform 4 or 8 operations in one clock cycle. Microprocessors are limited in the number of operations they can perform in one cycle. If power is in question, FPGAs can perform 50 to 100 times the performance per watt of power consumed than a microprocessor. The disadvantage of FPGAs is the limited processing abilities needed for complicated algorithmic processes, which can alternatively be satisfied with the use of a soft processor [1].

If the decision is to use FPGAs, designers can have more advantages by using embedded processors in FPGAs. Using embedded processors in designs is an increasing trend among developers (Figure 1.1). These embedded processors can be hard or soft. A soft processor is an IP core, generated from the logic cells of FPGA however hard processors are already fabricated inside FPGA, a dedicated part of the chip. Today's embedded systems must be power-efficient, sufficiently small and above all, cheap, to be commercially successful. If a design uses an embedded processor, board area is saved by removing the microprocessor from the board and the power consumption is also decreased. Using an embedded processor

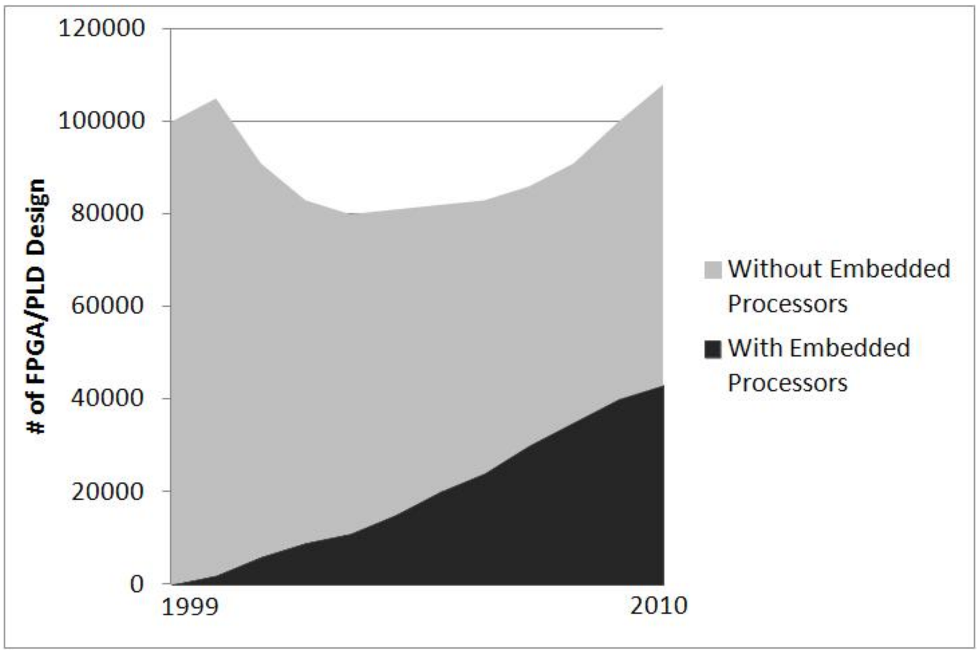


Figure 1.1: FPGA Designs With/Without Embedded Processors [2]

on the other hand has some disadvantages. Because of the integration of the hardware and software platform design, the design tools are more complex and relatively immature compared to standard processor design tools. Also downloading and debugging the software on the embedded processor can sometimes be tedious.

Running an RTOS on processors is a trend that designers increasingly follow due to RTOSs' deterministic behaviour and efficient resource management characteristics. Creation of tasks to handle and distribute huge sized codes, scheduling algorithms to manage the tasks, efficient interrupt handling and faster memory allocation are some of the many advantages of using an RTOS. API calls written specifically for the needs of real time systems simplify and accelerate the design process of software engineers. Time to market value significantly decreases if RTOSs are used. The software may be platform independent since RTOSs support many processors. If the decision is to use an RTOS, it is highly probable that the software code will not change at all, or slight changes will be needed, in case of a processor change.

RTOS comparison according to real time related benchmark criteria is useful for those who want to use RTOSs on their systems but are not sure which one to use. Designers choose an appropriate RTOS for their design by considering their requirements. For example, if memory is critical for a designer, it is wise to choose an RTOS with the lowest memory footprint specification. If interrupt handling times are critical, then the RTOS with the lowest interrupt handling time should be a rational choice. RTOS comparison studies will also provoke RTOS companies to improve their drawbacks and introduce better products. In literature, there are various benchmarking criteria for RTOSs defined for various purposes but we believe that there is not enough research done on porting these criteria to a specific processor and evaluate the performance of RTOSs according to these criteria. If you are a soft processor user and decide to use an RTOS on your processor, your chance of finding a survey including a detailed comparison of RTOS products on soft processors is quite low.

In this study, three RTOS candidates (Xilkernel, μC/OS-II and FreeRTOS) which can be used with MicroBlaze (soft processor of Xilinx) are compared according to their task preemption time, task preemption time under load, get/release semaphore time, pass/receive message time, get/release fixed sized dynamic memory time, UART RS-422 message interrupt serving time, RTOS initialization time and memory footprint data and then the results are interpreted. First, the characteristics of Xilkernel, μC/OS-II and FreeRTOS are given, namely their architectures, their scheduling infrastructure, system APIs and interrupt handling concepts are investigated. In light of this information the best RTOS on each benchmarking criterion is identified and the results are interpreted.

1.2 APPROACH

Xilkernel, μC/OS-II and FreeRTOS are selected from nine available RTOSs which are among the most popular RTOSs ported to run on MicroBlaze. The reasons of this selection are given in Section 7.1 in detailed. Three RTOSs will be configured to run on the target platform that is described in Section 7.2. The benchmarking test procedures, which are designed according to real time system developers' needs, will be performed on these selected RTOSs and the results will be interpreted using the detailed characteristics of RTOSs presented in Chapters 4, 5 and 6.

1.3 RELATED WORK

In [3], Xilkernel and FreeRTOS are used for developing a dynamic scheduling method for Kahn Process Networks. Two MicroBlaze processors are used on a many-to-one mapping approach, where the primary aim is to increase the efficiency of Embedded System-Level Platform Synthesis and Application Mapping (ESPAM) tool. The secondary and minor aim is to compare Xilkernel and FreeRTOS in terms of general concepts like performance, complexity, kernel customization and memory footprint.

Ronnholm [4], compared Xilkernel, uClinux and Asterix on MicroBlaze platform. Although the benchmark criteria are comprehensive and detailed according to [3], this study can be regarded as 'out-of-date' in view of the RTOSs compared. uClinux is not rated as a 'real time operating system' by the users of MicroBlaze because of its low timing performance and high memory footprint need [5]. Also Asterix is used mostly for educational purposes and is not popular among RTOS users.

In [6], RTOSs on small microcontrollers (microcontrollers with a maximum ROM of 128Kbytes and maximum RAM of 4Kbytes) are benchmarked in terms of their features and detailed performance criteria. Sixteen RTOSs are compared according to language support, tool compatibility, system service APIs, memory footprint (ROM and RAM usage), performance, device drivers, OS-aware debugging tools, technical support, source/object code distribution, licensing scheme and company reputation and finally four of them are selected for future performance benchmarking such as context switching, memory management calls and so on.

So, we believe that a renewed study on MicroBlaze is needed which should research the available RTOSs supporting MicroBlaze, compare them and eliminate some of them according to numerous criteria and perform an in-depth comparison on the rest using the critical criteria that real time system usage imposes. Preliminary results of the present work appeared in [7].

1.4 OUTLINE

The outline of this thesis is as follows. In Chapter 2, soft processor concept will be introduced. Real time systems and characteristics of real time operating systems will be summarized in Chapter 3. Chapters 4, 5 and 6 present brief overviews and key features of the RTOSs under consideration within this study, namely Xilkernel, μC/OS-II and FreeRTOS. The reasons for selecting Xilkernel, μC/OS-II and FreeRTOS, the target platform upon which the benchmarking is performed and the configuration options for the selected RTOSs are given in Chapter 7. In Chapter 8, the benchmarking criteria and the methods used in the comparison tests are detailed. The comparison results of our benchmarking tests are presented and interpreted in Chapter 9. Finally, Chapter 10 concludes the study. In Appendix part, the code snippets that are used in benchmarking tests are given for the three RTOSs.

CHAPTER 2

SOFT PROCESSORS

The demand for smaller devices in technology appears to be getting no slower. Mobile phone, computer and other electronic device companies aim to present more compact solutions in their areas. This argument is also valid for military applications. Light-weight, small, hand-held devices are preferred by military officials. These compact devices should also be power-efficient. For example, the features of today's mobile phones are rapidly increasing. Internet connection, GPS and other applications need more processor performance and hence more battery usage. This situation is also valid for notebooks, tablet computers and so on. So, power consumption in these devices should be minimal for a better design. Most importantly, these devices should be cheap to be a desirable option in the market. If a company is capable of providing the same service to its customers with a lower price, then surely their product is designed successfully.

Developers working on these departments mentioned above can save area, power, money and more by using an embedded processor inside FPGAs. Embedded processors can be hard or soft according to their architectures. Hard processors are embedded in hardware as dedicated silicon in either FPGAs or ASICs (Application Specific Integrated Circuit). However, soft processors are designed in a hardware description language such as VHDL or Verilog and then synthesized similar to all other FPGA blocks. They are simply VHDL codes. The features of the processor are up to the developer, that is to say, soft processors are technology independent. Reconfigurability is an important advantage, the designers can change the hardware architecture of their soft processors only by changing and compiling the FPGA code. Soft processors are used in areas like communication, advertising, defence industry, security-authentication and so on [8]. Some application examples using soft processors will be given in Section 2.4.

2.1 ADVANTAGES

2.1.1 CONFIGURABILITY

Soft processors are flexible. Based on the application, for example, one can change the number of UART controllers on the processor, add a CAN bus controller or an Ethernet infrastructure, select which memory controller to use and so on. The decision of which features the processor will have, depends on the developer. Surely, the more features added to the soft processor, the less performance you get from it. But looking from the perspective of configurability, being able to design the processor any time you want brings many advantages. First of all, it's a well-known phenomenon that the requirements of a project won't stay the same in the design phase. The customer will want an extra feature, system designers add this feature to the requirements and hardware or software engineers must be able to add this feature to the product even though their design phase have already started or maybe even finished. Let's assume that, we work at a company which is producing an air conditioner for an automotive company and the automotive company uses CAN bus infrastructure in the design of their cars. Requirements of the air conditioner are prepared by the system engineers and the decision is to use a hard microprocessor that supports CAN bus infrastructure. Everything goes well and the project is completed successfully. After a year, the automotive company changes its mind and decides to use an Ethernet platform in their cars instead of the CAN bus platform. The large amount of microprocessors stocked in the company are going to trash because microprocessors at hand may have no support for Ethernet protocol. New microprocessors should be bought, new electronic boards should be designed and the project cost may increase significantly. If soft processors were selected in the first place, all one has to do is simply disable the CAN bus support, enable the Ethernet support and compile the FPGA code again.

2.1.2 LOW-COST

If an FPGA is platform is already installed on the development board, then there is no extra fee to use a soft processor embedded in FPGA. If one has a licence from an FPGA company such as Xilinx or Altera for FPGA code development purposes, then the soft processor feature comes as a part of this licence. Furthermore, one can decrease the cost more by reducing the number of components on the board and minimizing the development board's size with the help of a soft processor. FPGA block RAMs and embedded Flash memory inside FPGAs can be used for development instead of using an extra RAM and FLASH memory on the board. But both type of memories inside an FPGA are limited in size (depending on the target FPGA).

Processor obsolescence is not an issue on soft processors since the VHDL code of the new released soft processors can always be purchased from the FPGA companies. For example, a soft processor which is released one week ago by an FPGA company, can be purchased and run on the FPGAs, which has been bought five years ago. Hence, development companies will not pray that their good old processor will not be obsolete in the near future, when soft processors are used in their designs.

2.1.3 HARDWARE SOLUTIONS TO SOFTWARE PROBLEMS

Possibly the most exciting reason to choose a soft processor is the freedom of developers in solving their problems in many different ways. For example, some problematic software parts of the project can be transferred and solved on the hardware part [9]. A custom VHDL core that uses the parallelism concept of FPGAs can be designed for solving a software problem which possibly can not easily be solved on the software part of the project. Also these FPGA cores can communicate with soft processors through shared memories or GPIOs. In summary, developers have more solution options to their problems if soft processors are used.

2.2 DISADVANTAGES

2.2.1 DESIGN TOOLS

Designing a soft processor based process includes enabling all the necessary features of the processor, assigning their parameters carefully and adding and adjusting the peripherals needed for the processor. So, at first, such processes can be hard for a software developer who is not competent in hardware design also. The design tools for soft processor based development processes should be easily understandable, helpful, and rich in documentation for the software developers. Since soft processor concept is a relatively new topic compared to designing and working with a standard processor, FPGA companies continue to improve the quality of their software design tools, which are used for generating a soft processor based project [9].

2.2.2 JTAG CHAIN PROBLEMS

Software code that runs on soft processors is downloaded to the target board using JTAG interface. In a complex system, when the components on the JTAG chain increases, debugging the software code on the soft processor gets even more difficult. Debugging speed decreases, making it hard to work efficiently. Also, if one of the components on the JTAG chain fails, then it is impossible to renew and download the software code. In other words, some other component in another board may affect code renewing process of a soft processor.

2.3 PROPRIETARY AND OPEN-SOURCE SOFT PROCESSORS

2.3.1 PROPRIETARY SOFT PROCESSORS

The most commonly used and well-known commercial soft cores are Xilinx's MicroBlaze and PicoBlaze, Altera's NIOS 2 and Tensilica's Xtensa. PicoBlaze is an embedded 8-bit soft processor, which is specialized for simple applications. MicroBlaze is an embedded 32-bit soft processor, which is optimized for embedded applications and can be configured according to requirements, namely, a fast processor or a size optimized processor can easily be generated. Error correction codes, floating point unit, memory management unit, instruction and data caches, branch optimization, hardware acceleration, hardware exception support, hardware debugging, TCP/IP, UDP and CAN bus infrastructure are some of the features that MicroBlaze supports [10]. NIOS 2 is a configurable embedded 32-bit soft processor for Altera FPGAs. Economy, standard and fast core options are available for selection according to the requirements at hand. Like MicroBlaze, NIOS 2 also has a rich instruction set optimized for embedded applications. Tensilica's Xtensa is not as popular as the soft processors mentioned above but offers a very useful feature to developers, a user-customizable instruction set. The four soft core processors mentioned above have similar design flows, so if a developer is familiar with using one of them, then it is easy to learn and use another one.

2.3.2 OPEN-SOURCE SOFT PROCESSORS

Either by using the architecture of commercial soft processors or by creating one from scratch, open source soft processors are also designed to fulfill the specialized needs of developers. SecretBlaze [11] is one such example, designed for secure applications, namely for investigating hardware and software solutions that are robust against cryptanalysis techniques. The other most popular open source soft processors are OpenRISC 1200, LEON3, aeMB, OpenFire and LatticeMico [8].

2.4 SAMPLE APPLICATIONS USING SOFT PROCESSORS

Soft processors can be used instead of hard processors in nearly all applications areas. For example, MicroBlaze running on a Virtex-6 FPGA can achieve a running frequency of 307 MHz in its performance-optimized configuration [10]. If an FPGA platform is already used on the board, using a soft processor saves area on the board and decreases power consumption. At ASELSAN Inc., MicroBlaze is used for example in hand-held thermal cameras where battery life is important and portability is an issue. Advanced Electronic Design produced a LED sign for JPMorgan which has nearly two million pixels, making it the highest resolution signage application in the world in 2006 [8]. In this application, a thousand PicoBlaze processors are communicating using Ethernet, sending and receiving data to different parts of the sign. Another application example is CISCO Systems' CISCO Carrier Routing System, in which 192 Xtensa cores are used in CISCO Silicon Packet Processor, which helps the scaling up the routing system to 92 Terabits per second. Also Chelsio Communication is using Xtensa processors for its next-generation 10 Gb Ethernet Terminator ASIC [12]. Altera's Nios II embedded soft processor is used in THALES' safety critical avionics system. Altera's HardCopy ASIC which includes Nios II, is used instead of a custom ASIC solution to decrease development time and manage system obsolescence [13].

CHAPTER 3

REAL TIME OPERATING SYSTEMS

A real time system is defined as an application which responds in a (timely) predictable way to all individual unpredictable external stimuli arrivals [14]. Processing information and giving a response to outside world must be within a critical specified interval. The duration of such intervals and the requirements of the application determine the type of the real time system. A hard real time system's unpredictable behaviour may cost lives or large amount of money, namely, missing a deadline is unacceptable. For example, a brake-by-wire system in an automobile is a real time system with hard constraints. Missing one critical deadline in the electronic infrastructure may yield a late brake and afterwards an accident. In soft real time systems, failing to meet the response time decreases the performance, but in general doesn't affect the way the system works. For example, GPS on a mobile phone can lose the connection for one or two seconds, eventually leading the user to miss new location information for a while. When the connection recovers from the error, the user may not even be aware that the connection has been lost, hence GPS on a mobile phone can be thought as a soft real time system.

To meet the critical deadlines that the real time systems impose, real time operating systems (RTOSs) are used. An RTOS is a piece of software with a set of APIs for users to develop applications [6]. The most important function of an RTOS is to manage the system resources such as memory and CPU efficiently. Multitasking, memory protection, various scheduling techniques, hardware libraries such as TCP/IP, CAN or USB are some of the features of RTOSs to help real time systems become more deterministic and predictable. In Figure 3.1, RTOS functions within an embedded system architecture is visualized. Some people may think that, using an RTOS is a waste of time and money, and necessary libraries or pieces of codes can be written by the user instead of using an RTOS. Imagine a USB flash drive company who wants to develop a USB interface protocol for a specific microcontroller used on their boards. Starting from scratch and designing the protocol for this microcontroller obviously takes time and requires a significant amount of person-hour. Assume that, the microcontroller that is used becomes obsolete some time later and the company chooses another microcontroller company, which necessitates a new USB stack code development. Then the amount of work that is done for the obsolete microcontroller is wasted. The codes of RTOS companies are shipped and used by lots of developers, so bugs are continuously fixed and workarounds are continuously found [15]. A USB protocol on a well-known RTOS option, which is developed and used for many years, might possibly be the most efficient and predictable option for implementing a USB platform. According to VDC Research Group [17],

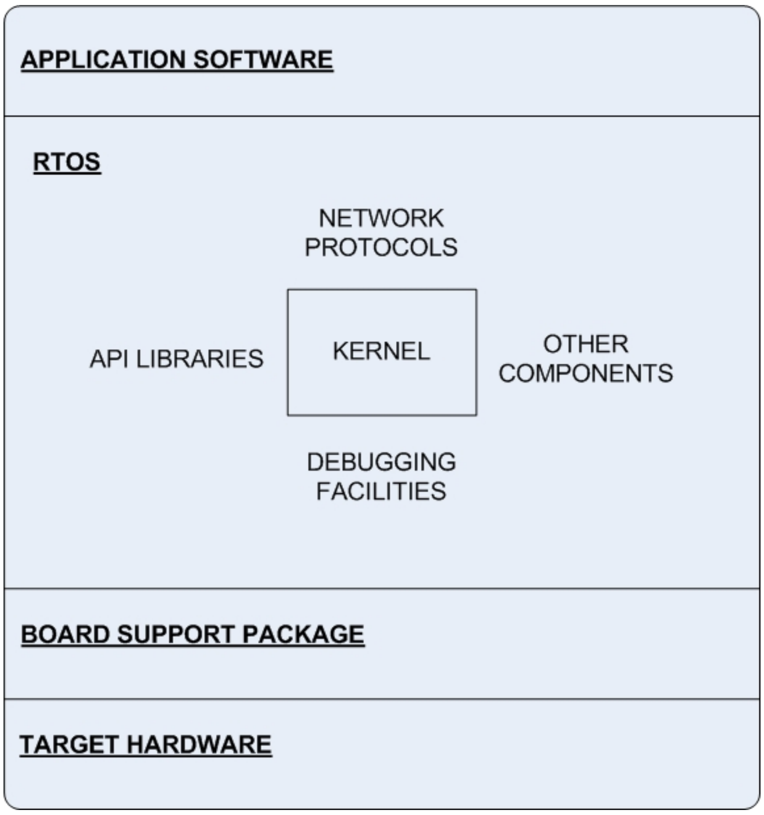


Figure 3.1: Architecture of an Application Using an RTOS [16]

using an RTOS is an increasing trend among system developers. The market for real time operating systems is increasing at an annual rate of %7. Nearly %50 of current no-RTOS users are thinking of using an RTOS for their future projects. In Figure 3.2, the results of a RTOS usage survey performed by VDC Research Group is given. Commercially licensed RTOS users are happy with their choices, so they will continue using commercially licensed

RTOSs. Open source RTOS usage is an increasing trend, since the percentage of using an open source RTOS in future is bigger than the current open source RTOS usage. The interest in using chip/vendor based RTOSs and in house developed RTOSs are decreasing. Also, in [6] it is stated that RTOS usage in 16-bit and 8-bit processors are also increasing, although there is a common belief that RTOSs are for powerful 32-bit processors.

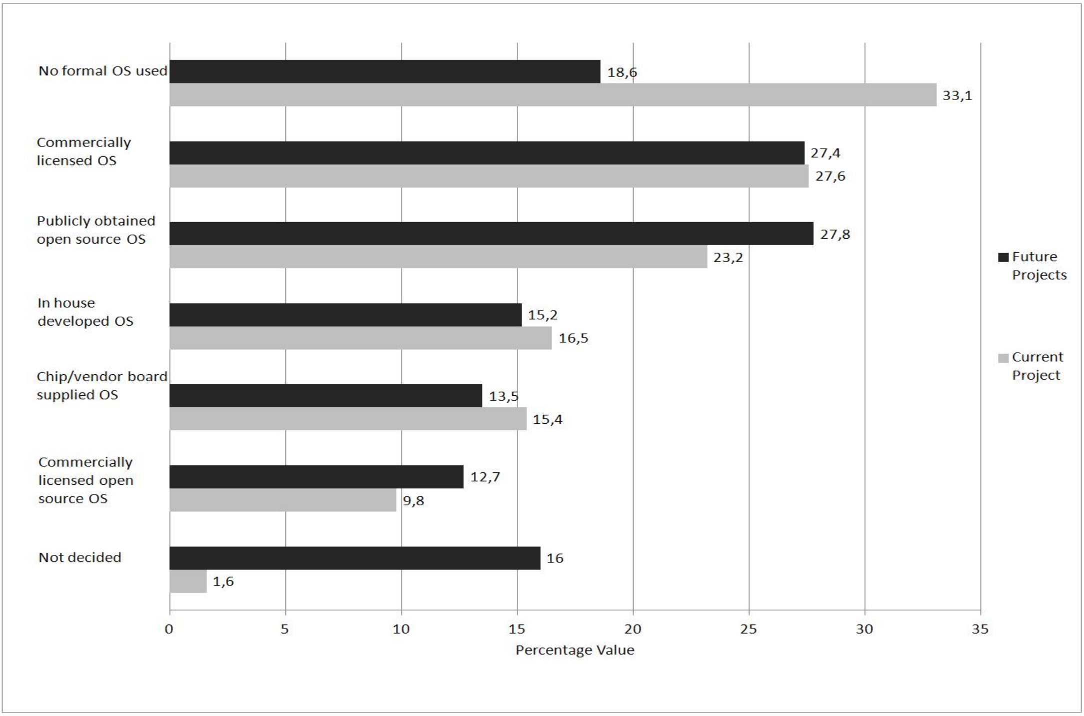


Figure 3.2: RTOS Usage Statistics [17]

3.1 ADVANTAGES

3.1.1 RESOURCE MANAGEMENT

Real time embedded systems generally have more than one event to respond. In order to handle these events in a time-predictable and deterministic fashion, all kinds of resources on the system should be managed efficiently. Multitasking, task synchronization tools like semaphores and mailboxes, memory protection, various scheduling techniques, faster interrupt handling mechanisms are some of the features that RTOSs provide for resource management purposes. By using the rich set of APIs that RTOSs provide, deterministic latency times are achieved, leading to more reliable systems. API functions of RTOSs can differ but in fact they perform similar operations for the same mechanisms, like semaphores or task scheduling.

3.1.2 HARDWARE SUPPORT

RTOSs support different kinds of processors, so any software that is developed for one type of processor is portable to another [14]. This increases reusability and decreases the time that is needed to start a new project.

3.1.3 TIME TO MARKET

Time to market significantly effects the overall cost of a project. EMF survey [18] states that, the average number of software developers for a project is 25.7. If the average cost for one engineer in USA is approximately \$150,000 per year, then average cost of delay per month is \$321,250. The average estimation of project delay is 4.1 months, so the total loss caused by this delay is calculated to be \$1,317,125. These figures do not take into account the possible loss caused by the business contracts because of the delay. Using RTOSs, products can be designed faster, less lines of code can be written with the help of easily understandable API calls and maintenance time of the project can be decreased. Bugs will be fewer since the system calls of RTOSs have worked and been tested over years. In summary, RTOSs decrease time to market and help to shorten the production time.

3.2 DISADVANTAGES

3.2.1 PERFORMANCE OVERHEAD

Using an RTOS more qualified than needed increases processor utilization so the system's power consumption increases. RTOS features running on background like scheduling can decrease the processor utilization of the user code. Enabling more features on an RTOS can increase the initialization time and memory footprint of an application. Also, RTOSs with more features need more time to get familiar with. So, a developer should analyse the requirements of project under consideration carefully and select the optimum RTOS for his/her project.

3.3 EXAMPLE RTOSs AND SAMPLE APPLICATIONS

RTOSs support various types of processors. For example, μC/OS-II supports up to 60 processors, hard or soft, 8-bit to 32-bit, etc. For a large scale like this, it is easy to guess that RTOSs are used in various application areas. In ASELSAN Inc., PowerPC processors are used with VxWorks and Integrity RTOSs. VxWorks company claims that VxWorks RTOSs power more than 1 billion embedded devices on earth [19]. Green Hills company states that Integrity RTOSs are used in areas like automotive, avionics, industrial safety, medical devices, secure mobile devices, secure networking, software defined radio and wireless devices [20]. In summary, in many different applications, one can easily find an embedded device running an RTOS in embedded systems area.

3.4 ARCHITECTURE OF AN RTOS

There are four application levels on a project that uses an RTOS. These are the hardware platform, board support package, RTOS and the application code. RTOSs lie between the BSP (Board Support Package) and the application code level which are all visualized in Figure 3.1. On the lowest level, hardware platforms reside, which can be communicated with assembly language. On top of that BSP level is available, which helps to communicate with the hardware platform. BSP codes are processor specific, which means, if the processor changes on the hardware platform level, then the BSP code should be updated. On top of the BSP level, RTOS is running which has no idea about the hardware platform. If the processor changes, there is no need to change the RTOS code, the API calls of RTOSs are hardware independent. On top of the RTOS level lies the application code. This is the code that handles the requirements of the project using RTOS API calls. Since RTOSs provide the user a portable platform, in case of a processor change, there is no need to change the application code which increases reusability.

If RTOS level is further investigated, we have to introduce the operating system tick concept (OS tick). OS tick is the software interrupt that every RTOS needs for scheduling and kernel specific operations. It can be thought as the heartbeat of an RTOS. On every OS tick, tasks are checked whether a context switching needs to be performed and ready/blocked task lists are updated. In summary user's application code is interrupted by the OS tick at a frequency defined on the configuration files of RTOSs. Multitasking in RTOSs is done with the help of this architecture, but with the drawback of the increased processor usage.

OS tick period is typically 1 or 10 milliseconds. Increasing the frequency of the OS tick decreases the performance of the kernel, the application code could spend all of it's time switching in and out of the kernel and the system response could be decreased [21].

CHAPTER 4

XILKERNEL

Xilkernel is Xilinx's small and modular RTOS for Xilinx embedded processors MicroBlaze and PowerPC. The most important difference of Xilkernel from other RTOSs ported to MicroBlaze is that it is integrated with Xilinx design tools and no extra licence is needed if you are already a MicroBlaze designer. POSIX API is used for real time support, which is easy to use or get familiar with. It's an advantage to use a worldwide standard interface since porting the software code to another RTOS will be easy. Scalability, which is a critical issue for embedded designers, is also covered in Xilkernel. Extra features can be added to the RTOS or some features can be excluded from compiling, which effects the memory footprint value of Xilkernel. In Figure 4.1, Xilkernel architecture is visualized. Modules seen on the figure can be customized, i.e. removed or added again.

4.1 FEATURES

4.1.1 SCHEDULING

Xilkernel has two scheduling options, one of which should be selected before kernel starts to run [22]. First one is preemptive scheduling (SCHED_PRIO). The thread, which has higher priority among other threads runs until the thread itself decides to leave the processor to other threads or the defined time slice is over. If a lower priority thread is running and a higher priority thread becomes ready to run, then lower priority thread is preempted and higher priority thread starts to run. Second one is round-robin scheduling (SCHED_RR). In this technique, processes have no priority, instead processor control is being passed over between processes in a cyclic manner. If the selection is SCHED_PRIO, then a priority queue is used

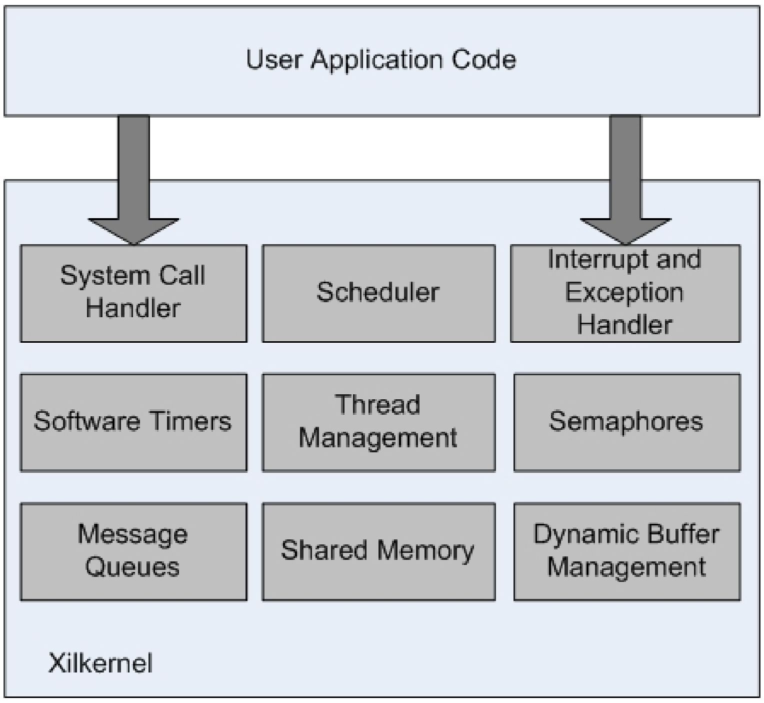


Figure 4.1: Xilkernel Architecture [22]

for deciding which process should run next. This priority queue is formed from ready queues which are as many as priority levels. The process which is at the top of the highest priority's ready queue is scheduled to run. If there are more than one process on the same priority ready queue, then the scheduling is round-robin and time-sliced between that processes. If a priority ready queue is empty, then the next priority queue is checked. Blocked processes are deleted from the ready queues and wait on the wait queues. When blocked processes are ready again, they enter the ready queues according to their priority. This type of architecture does not bring any delay theoretically when running a system under load (with many tasks). The scheduling architecture of Xilkernel can be seen on Figure 4.2.

4.1.2 SYSTEM APIs

Xilkernel supports a portion of the POSIX API standard, since the aim of this RTOS is to be small and suitable for simple embedded applications. For thread management, Xilkernel provides creating, exiting and joining options, also scheduling properties and variable stack sizes of threads. Xilkernel semaphore interface supports classical semaphore functions like posting and pending semaphores, and furthermore semaphores can be destroyed and their

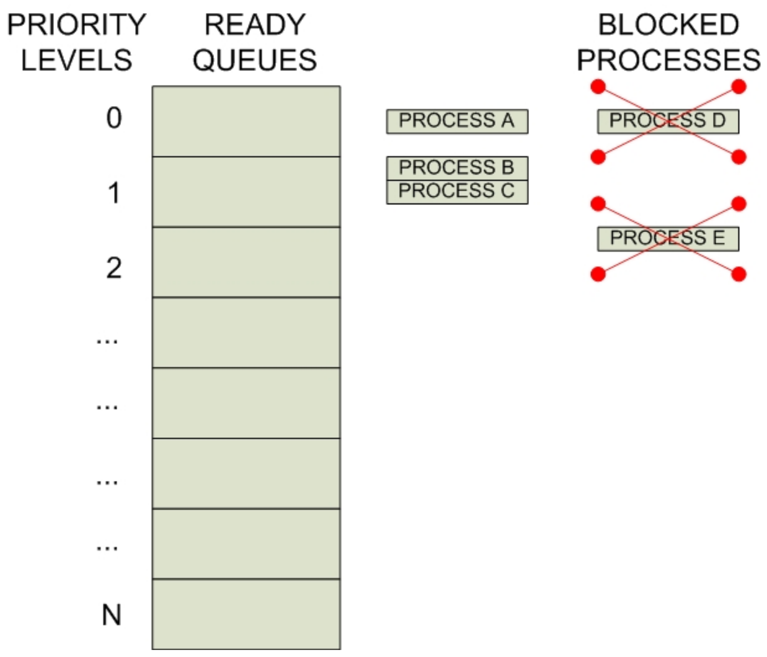


Figure 4.2: Scheduling Architecture of Xilkernel [22]

values can be read. Message queue API functions can be used if semaphore and dynamic memory modules are also enabled in the kernel. Sending or receiving messages and reading or configuring the queue are the available API functions while using message queues. Dynamic memory allocation is another module that Xilkernel supports. ‘Malloc’ and ‘Free’ functions are provided but, a user should handle them with care for the sake of thread-safety issue. Interrupts and context switching are advised to be disabled while handling dynamic memory issues. Shared memory, mutexes and software timers are other modules that are supported by Xilkernel. Locking, unlocking and configuring mutexes, returning the kernel clock ticks elapsed and sleeping processes are some of the API functions these modules provide.

4.1.3 INTERRUPT HANDLING

Xilkernel needs a software timer to perform scheduling, such that, a timer interrupt is generated from the software timer at a predefined frequency and connected to the kernel before run time (OS tick). Multiple external interrupts can be handled with the kernel code and the interrupt controller core together. When an external interrupt is active, the execution stack is switched to the separate kernel stack [22]. The advantage of this procedure is that, the interrupt execution does not use the user application stack. Meanwhile, the drawback of the method is that there will be a time overhead while serving the interrupts. After stack switch, the kernel stack gives the control to the interrupt controller and hence interrupt handler of that interrupt is called. Interrupt handler function should include minimum size of code as possible and also no blocking system calls should be made from the handler which results in an exception otherwise [22]. After the handler function is served, the scheduler of the kernel is called if there is a need for rescheduling the threads. The whole process is summarized in Figure 4.3.

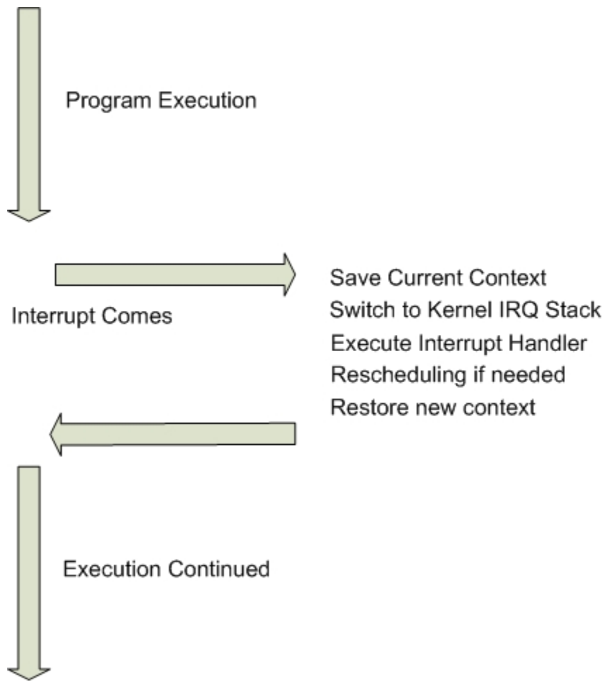
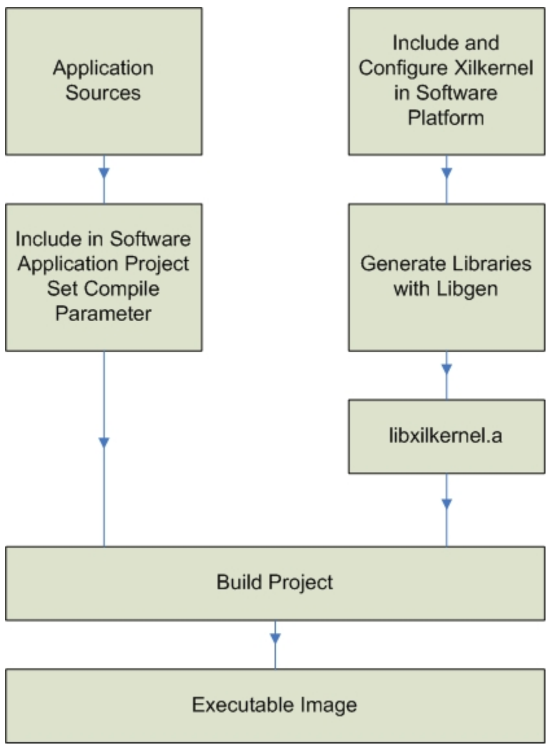


Figure 4.3: Interrupt Handling in Xilkernel [22]

4.2 LAUNCHING XILKERNEL

Applications using Xilkernel can be formed in two ways. First way is linking Xilkernel library file ‘libxilkernel.a’ with the software application file and generating one ‘.elf’ file. This end file is the only file which is downloaded to the target platform. Having one ‘.elf’ file helps the developer in debugging, downloading and bootloading [22]. The flow of this method is depicted in Figure 4.4. The second way, which is a bit harder than the first one, is downloading separate ‘.elf’ files to the target. All application files and the kernel image have separate ‘.elf’ files which are all downloaded to the target platform. Once the method of using Xilkernel is chosen, changes should be done on the application to work with Xilkernel accordingly. Customization of the kernel should be made, namely, by adding and removing the desired features of the kernel. A timer interrupt should be connected to the kernel for scheduling purposes. Static threads should be declared if desired and thread stack sizes should be defined. After designing the kernel, the developer should include the header 'xmk.h' to all application files. Finally, in main(), the developer should call xilkernel_main() for starting off the kernel.



Application Sources
Include and Configure Xilkernel in Software Platform
Include in Software Application Project Set Compile Parameter
Generate Libraries with Libgen
libxilkernel.a
Build Project
Executable Image

Figure 4.4: Building Applications in Xilkernel with One Separate File Method [22]

CHAPTER 5

μC/OS-II

μC/OS-II is Micrium's portable, scalable, preemptive, real time deterministic kernel for microprocessors, microcontrollers and DSPs [23]. Supporting up to 60 different processors, μC/OS-II is widely used in industry such as avionics systems that obey safety standard DO-178B, medical equipments, automotive systems, etc. It is also popular among academic researches, since source code can be used without license as long as it is not used for commercial applications.

μC/OS-II can be scaled to only contain the features you need for your application and thus provide a small footprint. Also, the execution times for most of the services provided by μC/OS-II is claimed to be both constant and deterministic. Architecture of μC/OS-II is depicted in Figure 5.1. μC/OS-II port block is a processor specific code, which changes according to different types of processors.

5.1 FEATURES

5.1.1 SCHEDULING

In μC/OS-II, each task has a unique priority level (meaning that a task cannot have a priority value equal to another one) so round-robin scheduling is not supported in the kernel. Alternatively, μC/OS-II supports preemptive scheduling, always running the highest priority task. Scheduling of tasks are handled by the kernel function OSSched(). Since the code in OSSched() is considered to be a critical section, interrupts are disabled before task scheduling process. Also, μC/OS-II claims that increasing task number does not decrease the task

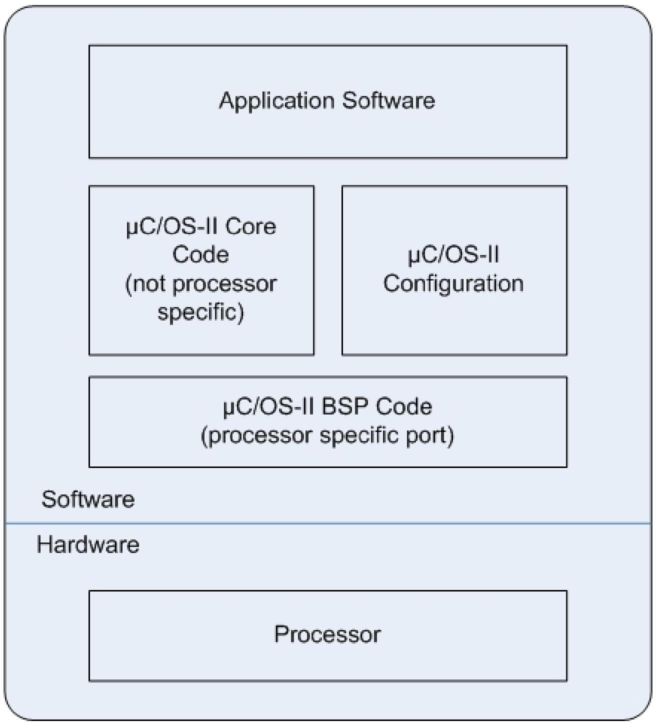

Application Software
µC/OS-II Core Code (not processor specific)
µC/OS-II Configuration
µC/OS-II BSP Code (processor specific port)
Software
Hardware
Processor

Figure 5.1: Architecture of μC/OS-II [24]

scheduling performance. Tasks can have 64 priority levels and 56 different tasks can be created in an application by the user [25]. Scheduling code of μC/OS-II is deterministic, that means the number of tasks created in the application does not affect the context switching time. More information about the scheduler and the prove of the above statement can be found in [26]. However, the execution time of the OS tick function is directly proportional to the number of tasks created in the application. So, one should be careful about the timing requirements when working with many tasks.

5.1.2 SYSTEM APIs

μC/OS-II supports more than 65 system functions related to semaphores, mutexes, event flags, mailboxes, message queues, memory management, task management, time management, timer management and scheduling. In Table 5.1, the most commonly used system services (some of which are also used in our benchmarking codes with Xilkernel and FreeRTOS) are summarized.

Table 5.1: System Services of μC/OS-II

System APIs	Description
Semaphores	
OSSemCreate	Creates a semaphore
OSSemDel	Deletes a semaphore
OSSemPend	Waits for a semaphore, if semaphore is not available
OSSemPost	Increases semaphore value, lets semaphore waiting tasks to run
Mutual Exclusion Semaphores	
OSMutexCreate	Creates a mutex
OSMutexDel	Deletes a mutex
OSMutexPend	Waits for a mutex, if mutex is not available
OSMutexPost	Posts mutex, lets mutex waiting tasks to run
Event Flags	
OSFlagCreate	Creates an event flag group
OSFlagDel	Deletes an event flag group
OSFlagPend	Waits for a combination of bits to be set in an event flag group
OSFlagPost	Sets or clears bits in an event flag group
Message Queues	
OSQCreate	Creates a message queue
OSQDel	Deletes a message queue
OSQFlush	Cleans the contents of a message queue
OSQPend	Pends for a message
OSQPost	Posts a message to the queue
Memory Management	
OSMemCreate	Creates a memory partition
OSMemGet	Obtains a memory block from the partition
OSMemPut	Returns a memory block to a partition
Task Management	
OSTaskChangePrio	Changes the priority of a task
OSTaskCreate	Creates a task
OSTaskDel	Deletes a task
OSTaskResume	Resumes a task
OSTaskSuspend	Suspends a task
Time Management	
OSTimeDly	Delays a task for a number of clock ticks
OSTimeGet	Gets the system clock in terms of clock ticks
OSTimeSet	Sets the system clock in terms of clock ticks
OSTimeTick	Processes the clock tick
Timer Management	
OSTmrCreate	Creates a timer
OSTmrDel	Deletes a timer
OSTmrStart	Starts timer
OSTmrStop	Stops timer
Miscellaneous	
OSInit	Initializes μC/OS-II variables and data structures
OSStart	Starts multitasking
OSVersion	Returns the version of μC/OS-II

5.1.3 INTERRUPT HANDLING

Interrupt handling in μC/OS-II is simply as follows. When a task is interrupted, execution of the task is suspended and ISR takes control of the CPU. The first duty of ISR is to save the registers of the task to the task's stack area. This saving takes approximately 300 nanoseconds on a MicroBlaze processor running at 150MHz [27]. Since μC/OS-II allows the usage of nested interrupts, second duty of ISR code is to control if this interrupt is a nested one or not. After this, user code of the corresponding interrupt's handler is executed. When the user code finishes its execution, the next running task is decided in the service call OSIntExit() function and context switching of tasks is performed if necessary. Finally, the registers of the completed task are restored and handling of the interrupt is finished. The interrupt handling process is summarized in Figure 5.2.

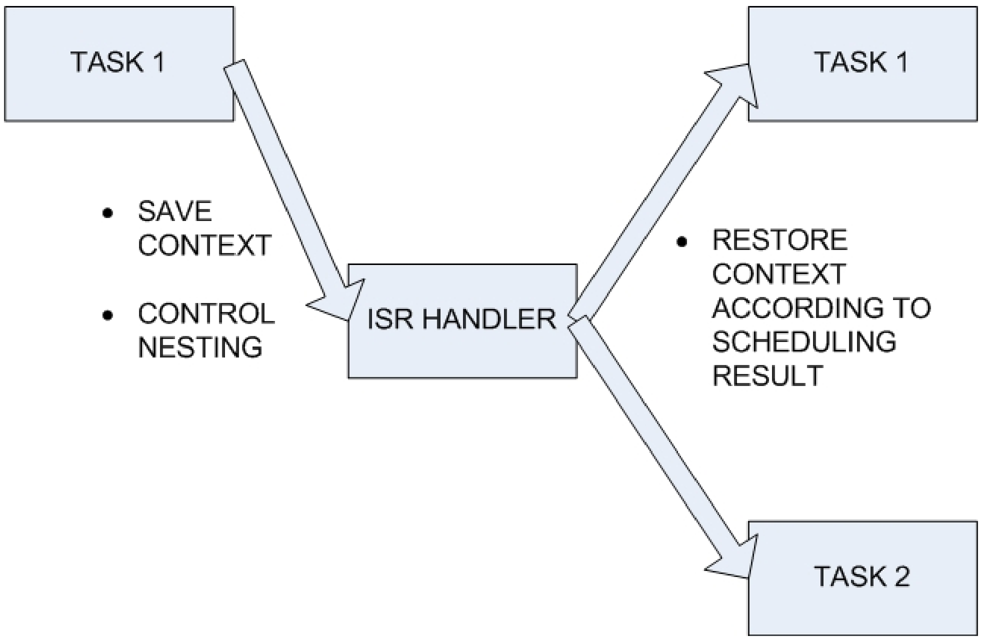


Figure 5.2: Interrupt Handling in μC/OS-II [24]

5.2 LAUNCHING μC/OS-II

For running μC/OS-II on an arbitrary processor smoothly, all the architecture blocks (application software, processor independent code, configuration, processor specific code, CPU, timer) which are defined on Figure 5.1 should be accurate. Since CPU and timer is assumed to be available on the hardware and processor independent code is provided by μC/OS-II website, a designer has to configure the rest of the blocks, i.e. processor specific code, configuration and application software. Board support packages, which will ease the process of writing processor specific code, are available for up to 60 processors on μC/OS-II website. For example, MicroBlaze BSP that is used in our hardware platform is downloaded from μC/OS-II website. If a processor is not supported on the website, a designer should change and modify the BSP code of a randomly selected processor. According to Jean J. Labrosse, this process consists of writing or changing 50 to 300 lines of code [25]. The rest of the blocks, application software and configuration, are provided by the designer according to the design requirements.

CHAPTER 6

FREERTOS

FreeRTOS is an open source RTOS while Xilkernel and μC/OS-II are propriety ones. This real time mini kernel receives 77500 downloads per year and is one of the most popular kernels among embedded system designers. According to 2011 EETimes Embedded Market Survey, FreeRTOS is the most frequently used kernel and also became the leading kernel, which is being considered currently as the next RTOS candidate to be used among embedded system users [28]. Increasing demand of RTOSs for small microcontrollers and the idea of having an RTOS with no license fee at all helped FreeRTOS to be popular among other RTOSs.

Using FreeRTOS instead of another commercial RTOS has two major advantages. First of all, it is a free and open source RTOS that means a designer can download the code and start developing with absolutely no cost. The other advantage is that a designer can create a working prototype of a product quite fast. Supported nearly for 60 different processors and 17 toolchains and having only three .c files for the core of the kernel are among the main reasons for this 'reduced time to market' value.

6.1 FEATURES

6.1.1 SCHEDULING

FreeRTOS has two modes of scheduling algorithms, cooperative and preemptive. In both modes, any number of tasks can have the same priority value, including the idle task. Unlike μC/OS-II, the user can create any number of tasks desired, the only restriction being the memory. In cooperative scheduling, a task runs until it voluntarily gives control to another task. It's more difficult to implement cooperative scheduling in a design, but when succeeded, a more reliable and deterministic system may be achieved. The other mode is preemptive scheduling, which was explained in earlier chapters. The scheduling code of FreeRTOS runs at every OS tick of the system. The algorithm controls the configuration attribute 'configUSE_PREEMPTION', and then decides which scheduling algorithm to use. According to this selection, necessary context switching operations are performed or not, and the timer tick is increased. The whole scheduling process is summarized in Figure 6.1. If the selection is to use preemption then the context switching times are critical for the user. The scheduler architecture is similar to the architecture that is used in Xilkernel. Ready queues, which are as many as priority levels on the system, contain the ready tasks. The scheduler runs the highest priority task that is available on the application by examining the ready queues. Blocked processes wait on the wait queues for their initiative interrupt. This architecture does not bring any extra delays when the system is under load.

6.1.2 SYSTEM APIs

Like Xilkernel and μC/OS-II, FreeRTOS has various system APIs for use. For task and scheduling purposes, FreeRTOS provides functions such as creating, delaying or yielding a task, entering and exiting critical regions inside a task and disabling or enabling interrupts. Also, scheduler statistics can be traced with system APIs 'vTaskStartTrace' and 'vTaskEndTrace'. Like many other RTOSs, queue and semaphore architectures are supported with simple system APIs. Software timer, which is a necessary RTOS property is also supported with 12 system APIs.

6.1.3 INTERRUPT HANDLING

FreeRTOS handles interrupts with the help of two RTOS configuration parameters; 'configKERNEL_INTERRUPT_PRIORITY' and 'configMAX_SYSCALL_INTERRUPT_PRIORITY'. First one, 'configKERNEL_INTERRUPT_PRIORITY' is the OS tick's interrupt priority. This should be set to the lowest interrupt priority level that the hardware platform supports. The other configuration parameter, 'configMAX_SYSCALL_INTERRUPT_PRIORITY', shows the highest interrupt priority level that interrupt safe system APIs can be called inside the interrupt handler. If an interrupt has an interrupt priority level higher than this constant, then

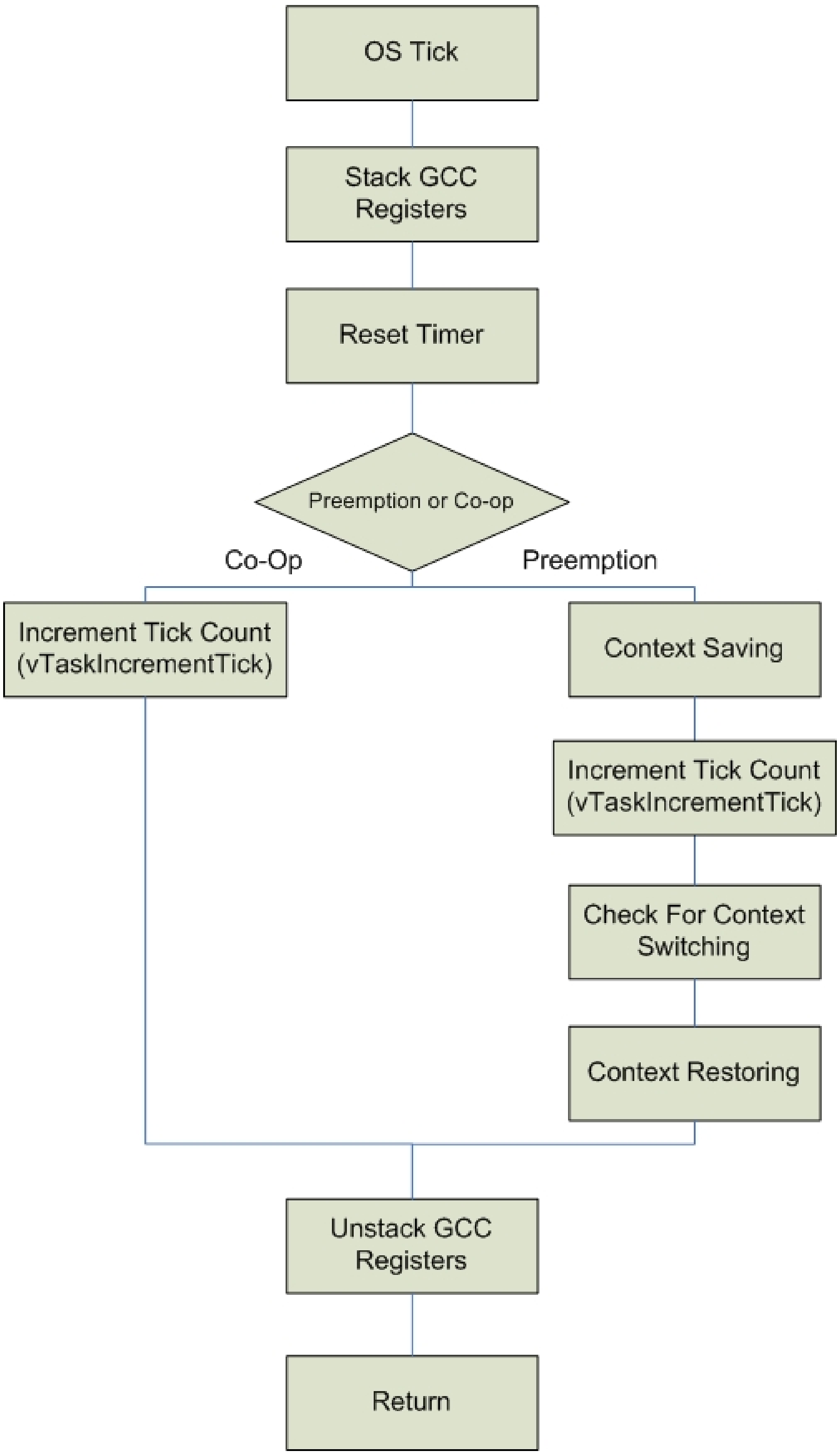


Figure 6.1: Scheduling Algorithm of FreeRTOS

no system APIs can be called from that interrupt's ISR handler, namely this interrupt will never be delayed because of the FreeRTOS kernel activity [28]. If an interrupt has no system calls inside the ISR handler, then that interrupt can have all the interrupt priority levels, namely, no restrictions come from the two configuration constants. Assuming an example microcontroller with 8 interrupt priority levels (0 being the lowest, 7 being the highest), the interrupt handling mechanism of this case can be visualized as in Figure 6.2. In summary, FreeRTOS serves interrupts with minimum timing overhead if configuration constants are arranged properly. Even the critical sections of the kernel can be interrupted, if the interrupt priority level is above 'configMAX_SYSCALL_INTERRUPT_PRIORITY'.

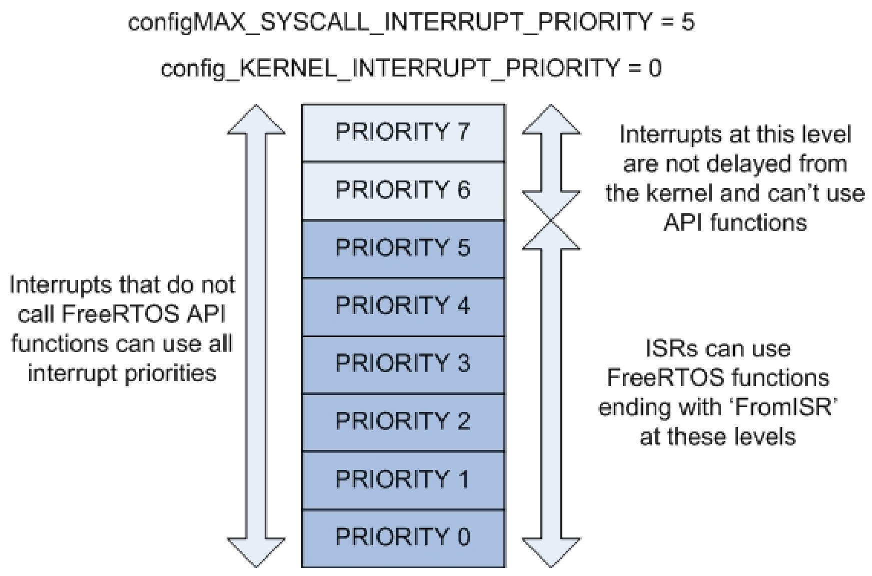

configMAX_SYSCALL_INTERRUPT_PRIORITY = 5
config_KERNEL_INTERRUPT_PRIORITY = 0
PRIORITY 7
PRIORITY 6
PRIORITY 5
PRIORITY 4
PRIORITY 3
PRIORITY 2
PRIORITY 1
PRIORITY 0
Interrupts at this level are not delayed from the kernel and can't use API functions
Interrupts that do not call FreeRTOS API functions can use all interrupt priorities
ISRs can use FreeRTOS functions ending with 'FromISR' at these levels

Figure 6.2: Interrupt Handling Algorithm in FreeRTOS

6.2 LAUNCHING FREERTOS

Launching FreeRTOS is a similar process to launching μC/OS-II. Application software, processor independent code, configuration, processor specific code, CPU and a timer is needed for running FreeRTOS smoothly on a hardware platform. CPU and the timer interrupt are assumed to be available on the hardware platform. Like all other RTOSs, FreeRTOS needs a timer interrupt for the kernel scheduling code. Processor independent code is available for download on FreeRTOS website which contains only three .c files (four if co-routines are used). Processor specific code may be the one with the heaviest workload above if porting to a specific processor is not supported on the FreeRTOS website. If a designer uses a processor that is already supported, all that's need to be done is to download the board support package files from FreeRTOS website and change some platform specific constants. Else, the designer should follow the instructions on FreeRTOS website and create the porting codes of the unsupported processor. The rest of the items, application software and configuration, are provided by the designer according to the design requirements at hand.

CHAPTER 7

TARGET PLATFORM AND CONFIGURATION OF SELECTED RTOSs

7.1 REASONS FOR SELECTION OF XILKERNEL, μC/OS-II AND FREERTOS

There are many RTOSs supporting MicroBlaze on the market. The leading ones and the companies distributing these RTOSs are listed in Table 7.1.

Table 7.1: Available RTOSs on MicroBlaze

Company	RTOS
FreeRTOS Team	FreeRTOS
Micrium	μC/OS-II
Mentor Graphics ESD	Nucleus Plus
Petalogix	uClinux
eSol Co. Ltd	PrKernel
Express Logic	ThreadX
MiSPO	NORTi
Xilinx	Xilkernel
eCosCentric	eCos

FreeRTOS is open source and very popular among above mentioned RTOSs according to 2011 EETimes Embedded Market Survey [28]. The survey also states that embedded system users selected FreeRTOS as their first choice for their future projects. So, FreeRTOS should definitely be on the comparison list. PrKernel and NORTi use μITRON architecture for standardization of API calls and software portability. Both RTOSs are poor in terms of documentation and support, and are not popular among MicroBlaze users. Even that, NORTi has a website that does not have language support for English. So PrKernel and NORTi are eliminated. uClinux stands for MicroController Linux, and is used by developers who are familiar with the Linux environment. However, μClinux is criticized as being 'not a real time OS' by embedded system users [5]. In [4], uClinux has 10 times worse interrupt latency times than Xilkernel and memory footprint values of uClinux are very high compared to Xilkernel. Also, a designer needs a Linux host to download and build the kernel, so μClinux is also eliminated. eCos needs a bootloader called Redboot to load program from the ROM to the RAM [6]. This brings up extra RAM and ROM space necessity, which is not suitable for a real time MicroBlaze system so eCos will not be on our comparison list. Xilkernel is highly integrated with the design tools of Xilinx and is using POSIX thread architecture. Every designer who uses MicroBlaze can use Xilkernel with no extra fee and can easily get familiar with that RTOS so comparison results of Xilkernel concern majority of the MicroBlaze community. We believe that an RTOS using the global POSIX API architecture should definitely be on the comparison list hence Xilkernel is in. Nucleus Plus supports the design tool of MicroBlaze (EDK) up to version 9.1. EDK version 12.4 will be used when comparing RTOSs on MicroBlaze so Nucleus Plus is eliminated. ThreadX also has the same problem. An example design of MicroBlaze is given on the website supporting EDK 11.4. Also, the design is given as a library, so the comparison will not be reasonable since the RTOSs to be compared should be compiled with the same compiler. The website does not give information about the MicroBlaze port for EDK 12.1 so ThreadX is eliminated. μC/OS-II is popular among MicroBlaze users and MicroBlaze port for EDK 12.4 is available on the website. Another advantage is that, μC/OS-II is free for educational purposes so μC/OS-II is on our comparison list.

7.2 A BRIEF COMPARISON OF THREE RTOSs

On Chapter 4, 5 and 6, we have given the features of Xilkernel, μC/OS-II and FreeRTOS but before we move further and compare them according to RTOS concepts we believe it is important to compare them according to general concepts like scheduling, licence type and documentation. This comparison will give a basic idea of that RTOS and will help us to understand the further detailed comparisons' results. For example, an open source RTOS may be thought of having a minimal implementation at first or an RTOS with a licence fee may be expected to have a better performance, but of course these are all reasonable predictions. Our general comparison criteria are given below.

- **Processor Support:** RTOSs support different processor architectures. Supporting more processors means that RTOS is more generic and better-designed, also applicable to real time systems with a wider range of complexity.
- **Licence Type:** RTOSs are provided according to different licence standards; free for all purposes, free for educational purposes only or fee-based for all purposes. This criterion may give an idea about the development status of an RTOS, whether that RTOS is a robust and mature RTOS or it is still progressing.
- **Documentation:** Being well-documented increases the understandability of the RTOS APIs which leads to more reliable systems and it decreases the learning period of an RTOS. This criterion will detail the documentation resources of RTOSs.
- **Language Support:** This criterion will give the programming languages supported by that RTOS. Supporting more languages means that RTOS is more advanced and applicable to more developers.
- **Scheduling Support:** Scheduling algorithms of the RTOSs will be given, whether preemptive, cooperative, round-robin and other scheduling techniques are supported.

The result of the comparison according to the criteria listed above is given in Table 7.2.

Table 7.2: First Look on the Three RTOSs

	Xilkernel	**μC/OS-II**	**FreeRTOS**
Processor Support	Only MicroBlaze and PowerPC	Up to 60	Up to 60
Licence Type	Fee-based	Free For Educational Purposes Only	Free
Documentation	Online	Book and Online	Online
Language Support	C/C++	C/C++	C/C++
Scheduling Support	Preemptive and Round-Robin	Preemptive	Preemptive and Cooperative

7.3 TARGET PLATFORM

The target platform that the benchmarking tests will be performed is an electronic board designed for controlling the Tank Driver Surveillance System in ASELSAN Inc., and this board consists of an FPGA, an external SRAM, Parallel Flash, serial communication chips, and a JTAG connection.

The requirements of the project impose the use of a MicroBlaze as the microprocessor and

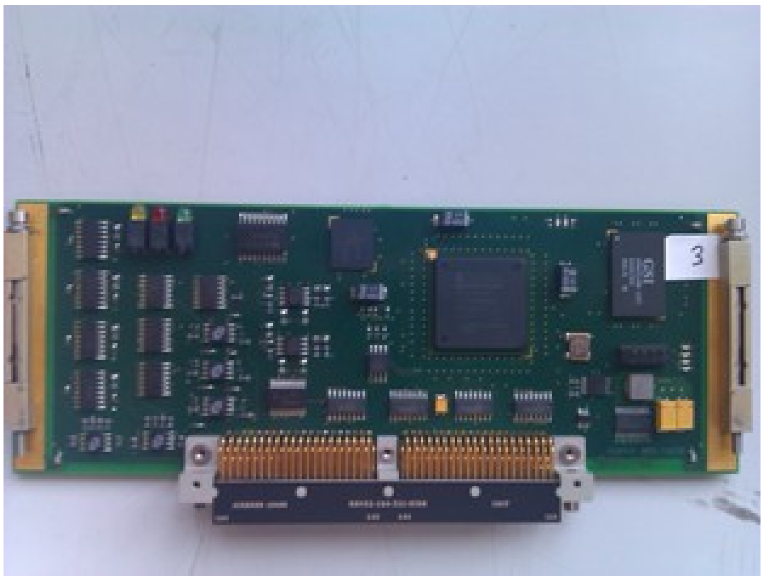


Figure 7.1: Target Board

no hard microprocessor component is placed on the control board. The board is built around Xilinx Spartan XC3S1400AN FPGA. Inside this FPGA, MicroBlaze is configured to run at a frequency of 80MHz. External Parallel Flash on the board is not used on the benchmarking tests since benchmarking codes are downloaded to the external SRAM on the board, in other words no ROM memory is used. Namely, benchmarking test codes and RTOS codes are located on the external SRAM. This external SRAM is 1Mbit, a size that is more than enough for both test and RTOS codes. So, no memory problems should be encountered on our benchmarking process. The FPGA code is compiled with the necessary configuration options so on the benchmarking process; there is no need to download the FPGA code again and again. Since FPGA code is located on internal FPGA Flash memory of the FPGA, only the output file of the compiled benchmarking codes should be downloaded to SRAM when the system is powered on. The target board can be seen in Figure 7.1. In order to explain the target plat-

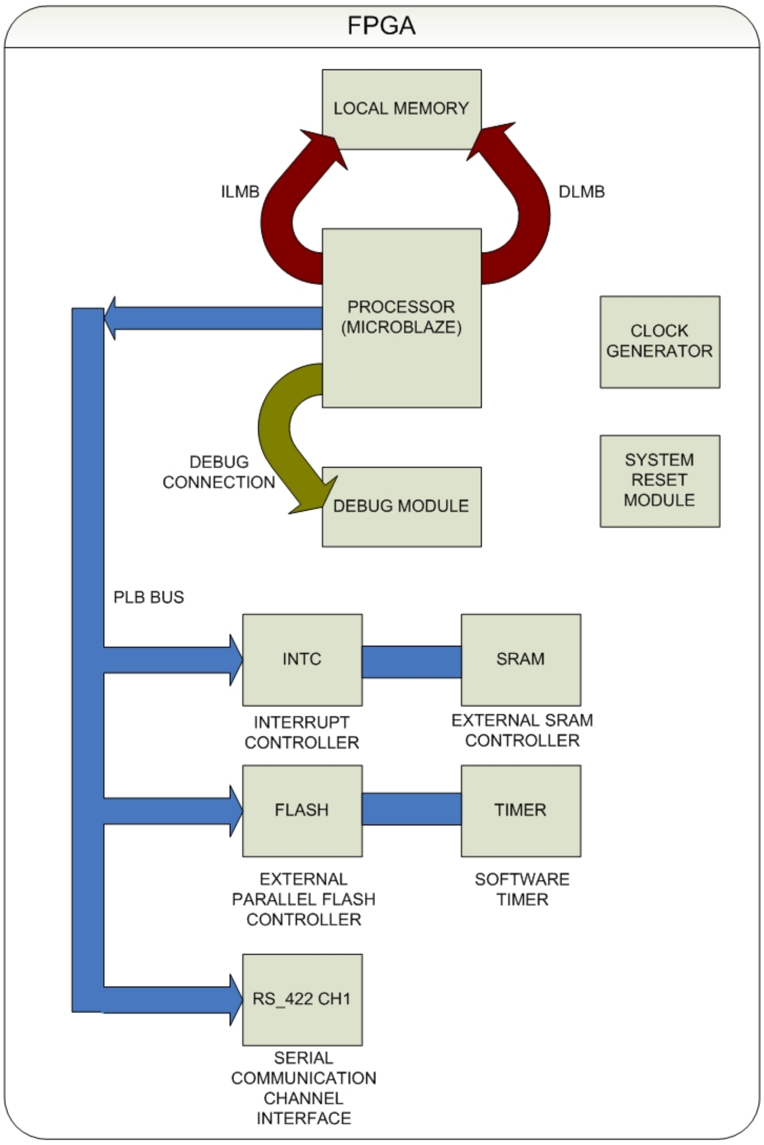


Figure 7.2: IP Cores Inside FPGA

form more detailed, the block diagram showing the IP cores (Intellectual Property) that are present on the FPGA system design are given in Figure 7.2. Our soft processor MicroBlaze is connected to the local memory through instruction and data local memory buses. Using the debug module, MicroBlaze can be debugged via JTAG connection outside from FPGA. The PLB (Processor Local Bus), provides separate 32-bit address and 64-bit data buses for the instruction and data parts. This architecture supports read and write data transfers between master and slave devices which have a PLB bus interface and connected through PLB signals. Supporting multiple master and slave devices, user can connect various IP cores to MicroBlaze using the PLB. As shown on the Figure 7.2, external SRAM and flash memory controllers, timer controller, interrupt controller and UART interface are connected to MicroBlaze with the PLB. Using the SRAM and flash memory controller interfaces, we can communicate with the external SRAM and parallel flash on our target board. Finally, clock generator generates the necessary clock signals that are needed for the IP cores and processor reset module generates the reset signal of the system.

To visualize the connections of FPGA and the external chips on the board, another block diagram is given on Figure 7.3. In Figure 7.2 the inner IP cores of FPGA are given, but in Figure 7.3, the bigger picture is given, which shows the chips reside on the electronic board.

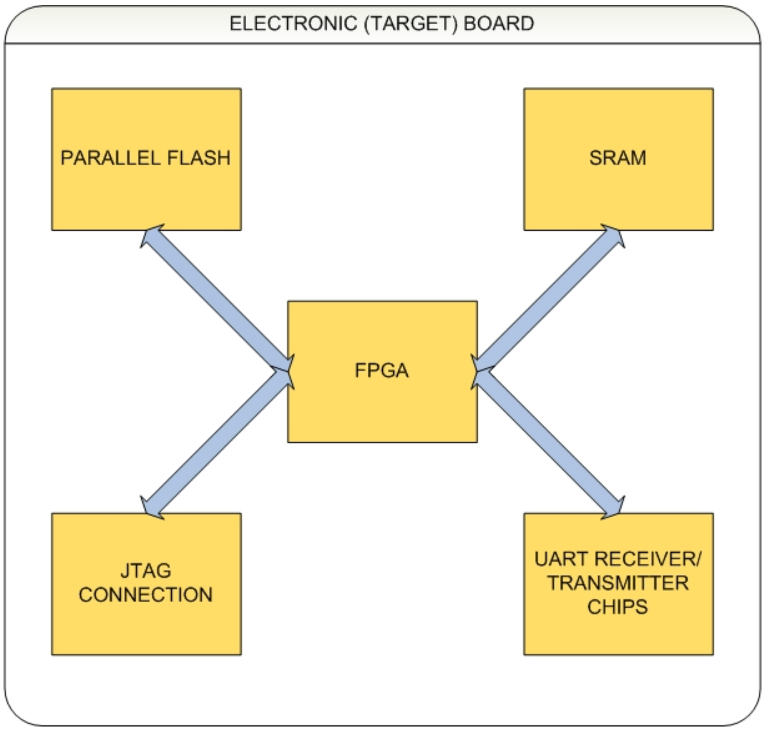


Figure 7.3: FPGA and External Chip Connections

7.4 CONFIGURATION OF SELECTED RTOSs

RTOS configuration options related with the selected benchmark criterion should be same on all RTOSs for reliable benchmarking results. GNU-based compiler tool-chain and GNU debugger (GDB) are used on the EDK environment (MicroBlaze code development environment) and the compiler version is GCC-4.1.2. The configuration features that are handled on RTOSs are listed below.

- The OS tick, which is the software interrupt given to RTOSs for kernel specific operations (such as scheduling) is generated every 1 ms.
- The compiler settings are default.
- Task stack size is configured as 2 KB.
- Stack and heap sizes of the program are both selected as 64 KB.
- Xilkernel Options: Xilkernel version 5.0.a is used for our benchmarking tests. The code of the Xilkernel comes with the licence of the code development environment.
- μC/OS-II Options: μC/OS-II version 2.90.a with a MicroBlaze port is used for our benchmarking tests. Full source code is downloaded from the website of Micrium and compiled with GCC-4.1.2.
- FreeRTOS Options: FreeRTOS version 7.0 with a MicroBlaze port is used for our benchmarking tests. Full source code is downloaded from the website of FreeRTOS and compiled with GCC-4.1.2.
- On each benchmarking test, unused RTOS features are disabled before compiling the RTOS.

CHAPTER 8

BENCHMARKING CRITERIA

RTOS benchmarking studies are useful for embedded designers, since selection of an RTOS at the beginning of the project can significantly effect the time to market duration of a product. Of course, the designers desire to select the most appropriate RTOS according to their requirements, so they research the available RTOSs on the market at first. They examine the web-sites, read the data sheets and they think they have all the technical information needed to decide which RTOS to use. However, the performance data of RTOSs on data sheets and web-sites are generally optimistic, namely, that performance will not be verified by the designers on their applications. If the selection is wrong, the designers most probably will not understand that they had the wrong RTOS choice, since all of the RTOSs have a learning period before using them efficiently.

Studies which compare the available RTOSs on a specific processor can help the designers who plan to run an RTOS on that processor for their projects. These studies should include all of the real time specific criteria for the sake of completeness. Not only designers make use of these studies, the RTOS companies will also have the chance to see the underperforming parts of their RTOSs. Valuable data from the application field can provoke the RTOS companies to improve the codes of their RTOSs and provide better RTOS products to designers.

Now, the benchmarking criteria will be detailed. Appendix A,B and C provide the code snippets that further detail the benchmarking criteria given below. Benchmarking codes are enriched with comments to increase understandability. To set an example, RTOS API calls are clearly stated with comments. Also, critical variables and constants are defined and explained.

8.1 TASK PREEMPTION TIME

Task preemption time is defined here as the time spent by the RTOS to save the context of a currently running low-priority task and switch the execution to a higher-priority task. Task switching time is defined as the time taken by the RTOS to save the context of a task and switch to another task which have equal priorities. Task switching time can not be taken as a benchmark criteria in this study, since tasks can not have equal priorities in μC/OS-II. Task preemption time is crucial on multitasking embedded systems which have strict timing requirements, so an RTOS should be switching tasks rapidly to give more execution time to user code.

On MicroBlaze platform, task preemption time performance of Xilkernel, μC/OS-II and FreeRTOS will be compared using the measurement method in Table 8.1. For 10 mil-

Table 8.1: Task Preemption Time Measurement Method

Task1 (Higher Priority)	**Task2 (Lower Priority)**
Running	Sleeping
Switch control to Task2	Sleeping
Sleeping	Running
Sleeping	When Task2 starts to run, switch control back to Task1
Running	Sleeping

liseconds, this measurement algorithm will run for each of the RTOSs on MicroBlaze and the number of task preemption operations will be counted separately. On μC/OS-II, 'OSTaskSuspend' and 'OSTaskResume' system calls are used for the purpose of switching tasks. On Xilkernel, which uses POSIX thread architecture, there is no system call to suspend and resume a task. The available commands for task management are 'yield','pthread_join' and 'pthread_exit'. The function 'yield' suspends the running task, changing the state of a task from 'running' to 'ready'. If this function is called from a higher priority task, the scheduler again gives the execution to the same task since that task is ready and waiting for execution [3]. In summary, 'yield' function is designed for round-robin scheduling, it is not suitable for our task preemption benchmark criterion. The function 'pthread_exit' terminates a task, and that task is deleted from the scheduling queues forever until that task is created again. Also, 'pthread_join' function keeps a task in a suspended state until another desired task is terminated. Since no tasks will be terminated in our benchmark test, this function is also not suitable for use. Semaphores, mutexes and conditional variables can be used for scheduling tasks, but using these features will cause comparing different types of architectures for task preemption criterion so Xilkernel will not be benchmarked on this criterion. For designers, who want to use Xilkernel to perform task preemptions, we performed a task preemption scenario with semaphores just to give an idea. The results of this test will be given on the results section. On FreeRTOS, similar to μC/OS-II, system calls of suspending and resuming tasks are used. These are ‘vTaskSuspend’ and ‘vTaskResume’. The API calls that are used by RTOSs are summarized in Table 8.2. To criticise the measurement algorithm, one

Table 8.2: API Calls for Task Preemption Time Measurement

	Xilkernel	**μC/OS-II**	**FreeRTOS**
Sleep Task	-	OSTaskSuspend	vTaskSuspend
Resume Task	-	OSTaskResume	vTaskResume

can argue that measuring the exact preemption time would give more reliable benchmarking results. First of all, software timers of RTOSs that will measure the exact preemption time have low time resolution. Time is measured in terms of ‘OS ticks’ so the time resolution is in the order of milliseconds because of the restrictions on the OS tick frequencies of RTOSs [21]. Task preemption time for RTOSs is on the order of microseconds so software timers are not suitable for measurement purposes. One can ask why the standalone software timer of MicroBlaze is not used for measurement purposes because this timer’s resolution is on the order of nanoseconds (depending on the running frequency of MicroBlaze). But this method is also not suitable since RTOS companies strongly advice that standalone service calls of MicroBlaze should not be used when an RTOS is running on MicroBlaze because of reliability issues. So, our measurement method does not aim to give the exact task preemption times, but our aim is to give the count of task preemption times performed in 10 milliseconds and find the best RTOS with respect to this criterion instead.

When comparing task preemption times, our measurement is affected from the variable context switching times that is caused by switching from the higher priority task to lower priority task. This can be seen as the drawback of our method which is visualized in Figure 8.1. However, context switching time is mostly affected from the number of registers that should be saved while switching tasks [25]. Our tasks which are used on the preemption test are simple tasks with no functions and furthermore their stack sizes are equal. So, we can safely assume that the time passed between switching from a high priority task to a low priority task and vice versa are equal. To conclude, we believe this drawback will not effect the results of our task preemption time comparison.

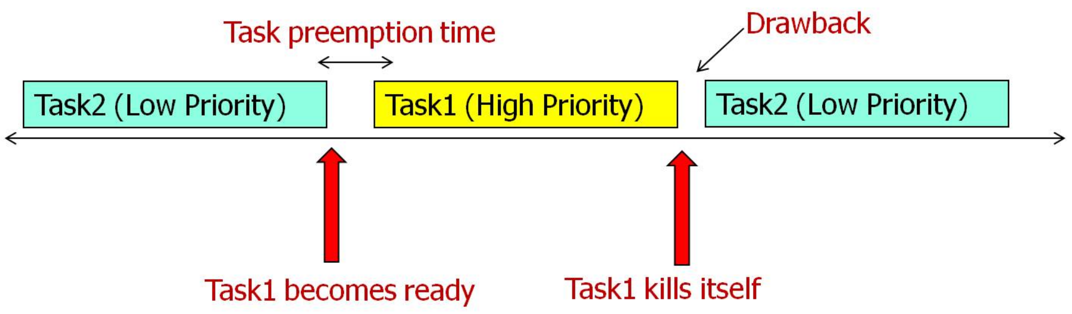


Figure 8.1: Measurement Method and the Drawback

8.2 TASK PREEMPTION TIME UNDER LOAD

The term 'real time' does not mean 'as fast as possible' but rather 'real time' demands consistent, repeatable, known timing performance [29]. Real time operating systems should be deterministic, that means they should offer load-independent performance. Timings should not change when the number of tasks, semaphores and other operating system related architectures increase on the application. Especially the scheduling performance of RTOSs should not be affected when the load increases. As we have seen on Chapter 4, 5 and 6; benchmarked RTOSs claim deterministic task preemption times. On this benchmarking criteria, we will repeat the previous task preemption time test, but with more tasks available on RTOSs. Xilkernel will not be on the comparison list, due to the reasoning on Section 8.1. RTOSs will have five, ten and fifteen tasks statically created at the start of the application, and we will examine the effects of these tasks by comparing the results of this benchmarking test with the previous two task case. The extra tasks will have lower priorities than the two tasks which perform context switching, so they won't take control of the scheduler. They will conserve their 'ready' states during the benchmark test.

For 10 milliseconds the same algorithm on the 'Task Preemption Time' test will run for each of the RTOSs on MicroBlaze and the number of task preemption operations will be counted separately. This benchmark criterion will be performed for five, ten and fifteen total number of tasks. Finding out which RTOS is faster on multitasking and understanding the effect of increasing total task number are aimed in this experiment. The service calls that will be used to resume and suspend the tasks on RTOSs will be same as given in Table 8.2.

8.3 GET/RELEASE SEMAPHORE TIME

Semaphores are widely used in today's embedded systems for the purpose of synchronization of resources and signalling events. Getting semaphore time is defined as the time required for an RTOS to take an available semaphore with the service call defined for that purpose. Releasing semaphore time is the time required for an RTOS to release the semaphore with the service call defined for that purpose.

On MicroBlaze platform, get/release semaphore time performance of Xilkernel, μC/OS-II and FreeRTOS will be compared using the measurement methods in Figure 8.2. For 10

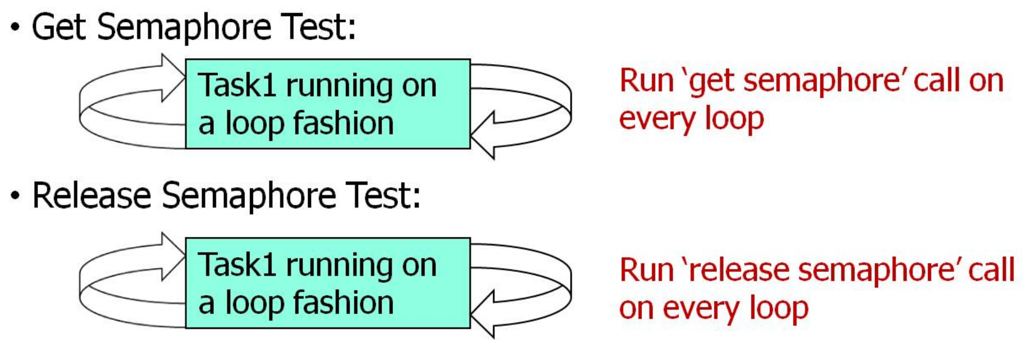

• Get Semaphore Test:
Task1 running on a loop fashion
Run 'get semaphore' call on every loop
• Release Semaphore Test:
Task1 running on a loop fashion
Run 'release semaphore' call on every loop

Figure 8.2: Measurement Methods for Get/Release Semaphore Time

milliseconds, 'get semaphore' method will run for each of the RTOSs on MicroBlaze and the number of 'get semaphore' operations will be counted separately. Another 10 milliseconds, 'release semaphore' algorithm will run for each of the RTOSs on MicroBlaze and the number of 'release semaphore' operations will be counted. For both tests, only one task is enough, which runs in an endless loop, only handling the functions of getting or releasing semaphores. For getting semaphore test, a counting semaphore will be created before Task1 starts to run and after 10 milliseconds of executing 'get semaphore' service call, the value of the counting semaphore will be investigated. Then, the number of executed 'get semaphore' calls will be found. Similar approach will be followed for releasing semaphore test. The API calls that are used by RTOSs are given in Table 8.3.

Table 8.3: API Calls for Get/Release Semaphore Time Measurement

	Xilkernel	**μC/OS-II**	**FreeRTOS**
Get Semaphore	sem_wait	OSSemPend	xSemaphoreTake
Release Semaphore	sem_post	OSSemPost	xSemaphoreGive

8.4 PASS/RECEIVE MESSAGE TIME

Message passing is a popular synchronization and communication infrastructure between tasks. Tasks can send and receive bytes, data structures, pointers and code segments to each other via message passing API calls that RTOSs support. For example, 10 bytes of critical information may be received from outside world via an external interrupt inside TaskA, which should be processed by TaskB immediately after reception. This 10 byte information can be passed to the internal RTOS queues from TaskA to TaskB, which is waiting for the message to run. Message passing is a similar synchronization tool to semaphore passing, but including information exchange between users.

RTOSs provide message passing API functions for handling message passing operation. These functions, like most functions of RTOSs, should be deterministic in time. Any unexpected delay can cause failures on the system whose extent depends on the requirements of the system. On MicroBlaze platform, pass/receive message time performance of Xilkernel, μC/OS-II and FreeRTOS will be compared using the measurement methods in Figure 8.3. For 10 milliseconds 'pass message' method will run on each of the RTOSs on MicroBlaze and the number of 'pass message' operations will be counted separately for each RTOS using a global variable. Same method will be repeated for releasing message test. For both tests, only one task is enough, which will run in an endless loop, only handling the functions of getting or releasing fixed sized messages. Since μC/OS-II and Xilkernel allow only message pointer passing mechanism, which means passing 32-bit address value between tasks, our algorithm should pass 32-bit values in all of the three RTOSs. In Xilkernel and μC/OS-II, a pointer which points to a 32-bit integer will be sent and in FreeRTOS 32-bit integer will be directly sent, through the queue generated inside the kernel. So, basicly all three RTOSs will send and receive the

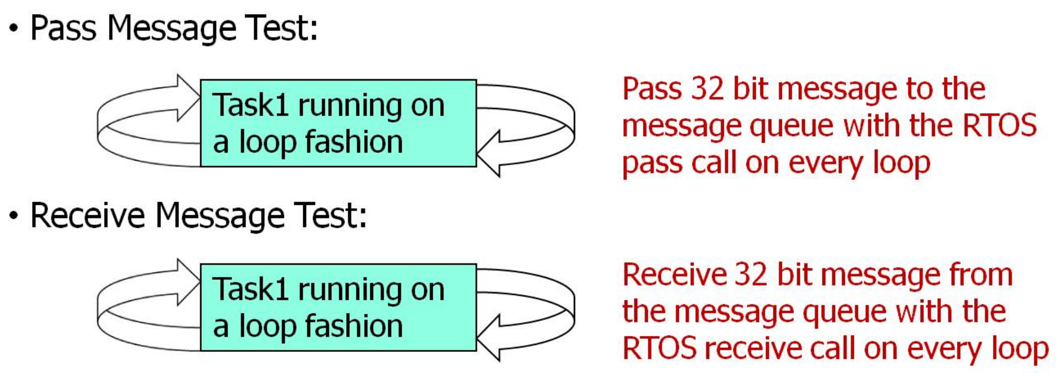

• Pass Message Test:
Task1 running on a loop fashion
Pass 32 bit message to the message queue with the RTOS pass call on every loop
• Receive Message Test:
Task1 running on a loop fashion
Receive 32 bit message from the message queue with the RTOS receive call on every loop

Figure 8.3: Measurement Methods for Pass/Receive Message Time

same amount of data. Since all of the RTOSs have a heap size of 64kB, 'pass message' test should not last until the heap memory is full. The precautions for the danger of filling the heap memory with messages should be taken and handled in our algorithm. So, a period of 10 milliseconds is verified to be enough for a safe heap memory usage. For 10 milliseconds, at maximum, only one fifth of the heap memory is filled by examining the number of 'pass message' operations. The API calls that are used by RTOSs are given in Table 8.4.

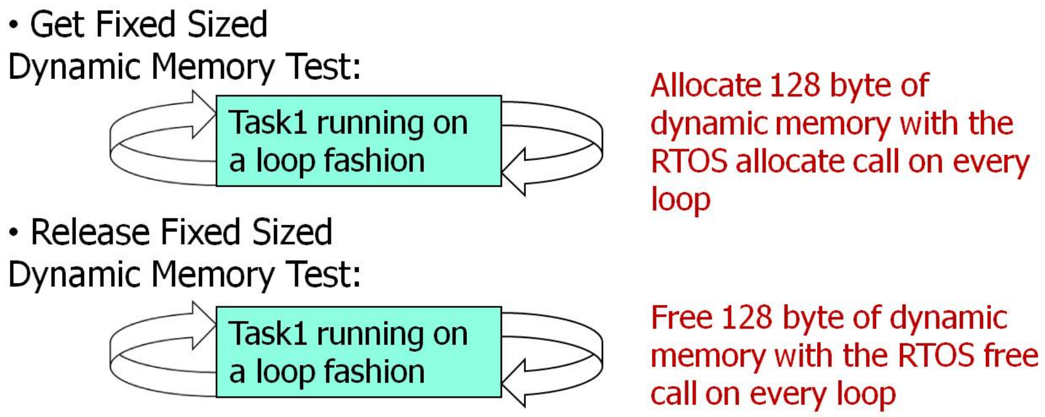
Table 8.4: API Calls for Pass/Receive Message Time Measurement

	Xilkernel	**μC/OS-II**	**FreeRTOS**
Get Dynamic Memory	bufmalloc	OSMemGet	pvPortMalloc
Release Dynamic Memory	buffree	OSMemPut	vPortFree

8.5 GET/RELEASE FIXED SIZED DYNAMIC MEMORY TIME

When writing a software application and memory is needed during run-time, dynamic memory allocation libraries are used for allocating and deallocating memory. If variables on the software code are created statically, then these variables are created on stack memory when the code starts to run and are released when the code exits. But if dynamic memory usage is needed, memory is created on heap dynamically using the allocation functions and freed under the control of the user, again using the library functions. So, one should be careful while dealing with dynamic memory, since not freed memories can lead to memory exhaustion problems in long term. Usage of dynamic memory is advantageous since pieces of memory can be reused by the application, but it should be handled with care.

RTOSs provide dynamic memory allocation functions to simplify dynamic memory management. These functions must be deterministic in time like most of the functions of RTOSs to fulfill the critical timing requirements of real time systems. Selected RTOSs will be compared according to 'get/release fixed sized dynamic memory' functions with the methods given in Figure 8.4. For 10 milliseconds 'get fixed size memory' method will run for each of the



• Get Fixed Sized Dynamic Memory Test:
Task1 running on a loop fashion
Allocate 128 byte of dynamic memory with the RTOS allocate call on every loop
• Release Fixed Sized Dynamic Memory Test:
Task1 running on a loop fashion
Free 128 byte of dynamic memory with the RTOS free call on every loop

Figure 8.4: Measurement Methods for Get/Release Fixed Sized Dynamic Memory Time

RTOSs on MicroBlaze and the number of 'get fixed size memory' operations will be counted separately for each RTOS. Same method will be repeated for 'release fixed size memory' operations. For both tests, only one task is enough, which will run in an endless loop, only handling the functions of getting or releasing fixed sized memories. The fixed size memory mentioned is decided to be 128 bytes. For 'get fixed sized memory' test, the application will allocate 128 bytes of dynamic memory in every loop, and using a global variable, the number of 'get fixed sized memory' calls will be counted. For 'release fixed sized memory' test, the application will free 128 bytes of dynamic memory in every loop, and using a global variable, the number of 'release fixed sized memory' calls will be counted. Since all of the RTOSs have a heap size of 64kB, 'get fixed sized memory' test should not last until the heap memory is full. So, a period of 10 milliseconds is tested to be enough for a safe heap memory usage. In μC/OS-II and Xilkernel, dynamic memory partitions should be created before the kernel starts to run, but in FreeRTOS, using the memory allocation functions, dynamic memory is allocated directly from the heap (heap_3.c is used as memory management source for FreeRTOS). The API calls that are used by RTOSs are given in Table 8.5.

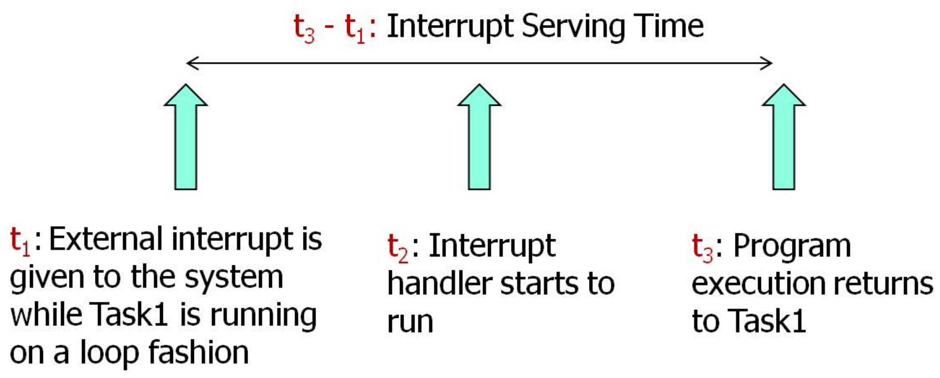
Table 8.5: API Calls for Get/Release Fixed Size Dynamic Memory Time Measurement

	Xilkernel	**μC/OS-II**	**FreeRTOS**
Receive Message	msgrcv	OSQPend	xQueueReceive
Post Message	msgsnd	OSQPost	xQueueSend

8.6 UART RS-422 MESSAGE INTERRUPT SERVING TIME

When working with real time embedded systems, serving external interrupts in a fast and deterministic fashion is very crucial. According to the requirements of the application, real time systems have to respond to outside events within a critical specified interval. Serving interrupts faster is not enough for a good RTOS, also the serving times should be predictable. In this benchmark criterion, all of the RTOSs will be compared against their UART RS-422 message interrupt serving time. UARTs are pieces of integrated circuits that are used for serial communication realizing the standards RS-232, RS-422, RS-485 and so on. A UART RS-422 message interrupt will be generated in this benchmark test. The time needed to serve that interrupt and return the execution of the program to where it is interrupted will be measured (Figure 8.5). Note that, this criterion is highly dependant on the hardware architecture of the processor, but RTOSs can effect the serving time by introducing extra delays because of their usage of the processor. So, instead of measuring the exact time of serving an interrupt, we will run the processor for 10 milliseconds and count how many interrupt servings are done on each RTOS. By this way, we will take into account the inner architecture of the RTOS, which is the timer tick code that runs for scheduling purposes. The algorithm of the UART RS-422 message interrupt serving time criterion will be as follows (Table 8.6).

- There will be one task on all of the RTOSs.
- This task will run on an endless loop fashion.
- The duty of the task is to send an RS-422 message (1 byte long) from a UART channel on the hardware platform.
- When a message is sent from UART RS-422 channel, an interrupt is generated inside MicroBlaze which leads the flow of the program to the handler function of this interrupt.



t_3 - t_1: Interrupt Serving Time
t_1: External interrupt is given to the system while Task1 is running on a loop fashion
t_2: Interrupt handler starts to run
t_3: Program execution returns to Task1

Figure 8.5: Interrupt Serving Flow

- Inside the handler function, a global boolean attribute will be set to 'true' which helps us to understand that the interrupt is served.
- Then, the flow of the program will be returned to the task again and the same RS-422 serial message will be sent via UART RS-422 channel if the global boolean attribute is set on the handler.
- This loop will continue for 10 milliseconds, and how many interrupts are served on all of the RTOSs will be compared.

Table 8.6: UART RS-422 Message Interrupt Serving Time Method

	Task1	**Interrupt Handler**
1.	Running	Waiting for the interrupt
2.	UART RS-422 message sent	Waiting for the interrupt
3.	Interrupt generated when message sent	Waiting for the interrupt
4.	Not Running	Program comes to this handler
5.	Not Running	Bool attribute set
6.	Running	Waiting for the interrupt
7.	If Bool attribute set, return to step 2	Waiting for the interrupt

8.7 RTOS INITIALIZATION TIME

After the power-up of the system, RTOSs have to initialize their architecture and the hardware platform before making any RTOS API calls. Initialization of the global variables, creation of the idle thread, user static threads and any other system threads are some of the duties RTOSs have to perform before scheduling starts. Also, initialization of the hardware units on the board has to be performed, namely memory controllers and processor specific initializations should be handled by the RTOS. The initialization process finishes when a jump is made to the scheduler to start running the highest priority thread [30]. This RTOS initialization time is surely important for developers who need a fast start-up on their systems. Our benchmarking test will measure the time between the power-up of the system and the first instruction of the first task and find the fastest RTOS on this criterion. The measurement method is visualized in Figure 8.6.

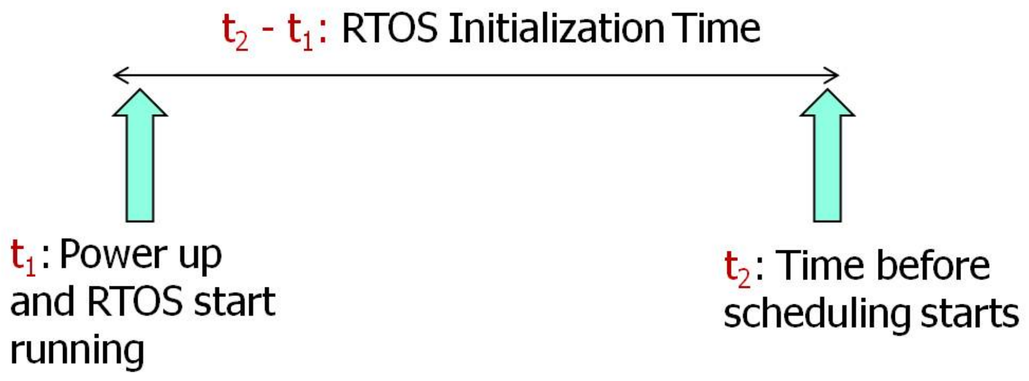

t_2 - t_1: RTOS Initialization Time
t_1: Power up and RTOS start running
t_2: Time before scheduling starts

Figure 8.6: RTOS Initialization Time Measurement

8.8 MEMORY FOOTPRINT

Memory footprint is the RAM usage of the RTOS codes. Memory restrictions are critical for real time systems, so if an RTOS is decided to be used on a real time system, a developer expects a low memory footprint from that RTOS. The memory usage of an RTOS depends on the applications it supports so it is expected that an RTOS with more features enabled, needs more memory. We will compare the memory footprint needs of all RTOSs by comparing the text section memory of the application under the load of one task running on a forever loop. Text section of the application contains the executable instructions generated from the source codes of RTOSs whose size gives the best idea about the memory footprint of an RTOS. All the unused features of all RTOSs should be disabled for a meaningful comparison. As FreeRTOS is not reconfigurable, it needs all the source files to be compiled for any application, so FreeRTOS will not be on the memory footprint comparison. μC/OS-II and Xilkernel will be compared, and having the lowest memory footprint RTOS will be found.

CHAPTER 9

RESULTS

In this chapter, the results of benchmarking tests will be given. The RTOSs will be compared against the criteria defined in Chapter 8, the winner of each criterion will be identified, and the results will be interpreted using the information of RTOS features given in Chapter 4, 5 and 6. Total results are summarized in Table 9.1. All the tests except memory footprint comparison are iterated for a number of times to ensure a confidence level. The number of iterations have been selected to achieve a confidence level of %99 with a gap of %1 ($\Delta < 0.01$) [31].

Table 9.1: Benchmarking Results

	Xilkernel	**μC/OS-II**	**FreeRTOS**
Task Preemption Count in 10 ms	-	**59**	30
Task Preemption Count Under Load in 10 ms (5/10/15 tasks)	-	**57/55/50**	30/30/30
Get Semaphore Count in 10 ms	131	**302**	156
Release Semaphore Count in 10 ms	132	**280**	180
Pass Message Count in 10 ms	63	**308**	125
Receive Message Count in 10 ms	47	**312**	112
Get Fixed Sized Dynamic Memory Count in 10 ms	243	**438**	86
Release Fixed Sized Dynamic Memory Count in 10 ms	89	**407**	104
UART RS-422 Message Interrupt Serving Count in 10 ms	**75**	54	53
Initialization Time in ms	**1,64**	7,81	5,96
Memory Footprint in bytes	**16490**	17086	-

9.1 TASK PREEMPTION TIME

'Task preemption time' comparison gives the number of task preemptions performed by each RTOS in 10 milliseconds. μC/OS-II performs more task preemptions than FreeRTOS if results are interpreted. The result is expected if the section includes the brief comparison of RTOSs is examined carefully (Section 7.2) since μC/OS-II supports only priority scheduling, the context switching code of μC/OS-II does not need to control other scheduling techniques when switching tasks. However, FreeRTOS support priority and round-robin scheduling, so when switching tasks, extra conditional statements and code is written for controlling both scheduling techniques. Both RTOSs enter critical region (disabling interrupts on MicroBlaze) when performing context switching operation to prevent from setting the ready bit of one or more tasks accidently. Also, both RTOSs use assembly language functions to save and restore task's contexts, which are the state registers of MicroBlaze. Since enabling/disabling interrupts and reading/writing state registers of MicroBlaze are port specific functions, a major timing difference is not expected arising from those code snippets.

As stated on Section 8.1, we performed task preemption time benchmark test on Xilkernel using semaphore architecture just to give an idea although it should not be compared with other RTOSs directly. The number of task preemptions performed by Xilkernel using semaphore architecture is 26.

9.2 TASK PREEMPTION TIME UNDER LOAD

Results show that, FreeRTOS is not affected by the increase of the load that is caused by incremented number of tasks however μC/OS-II's performance drops by % 3.51 when total number of tasks are five, % 7.02 when total number of tasks are ten, and % 12.28 when total number of tasks are fifteen if we compare the results with the previous benchmarking criterion (Figure 9.2). Namely, increasing the total number of tasks on the system decreases the task preemption performance of μC/OS-II, which decreases the determinism of μC/OS-II. In spite of its high performance, μC/OS-II is not deterministic on this criterion.

We know from Chapter 4, 5 and 6 that theoretically the scheduler of all the RTOSs are deterministic, so there must not be any performance degradation if the number of tasks increase.

However, our benchmarking test does not measure the exact task preemption time, our aim is to find out which RTOS is better on run-time on a specific timing period. Apart from the scheduling codes, the OS tick function also affects the performance on this criterion. As we have explained on Chapter 5, the execution time of μC/OS-II's timer tick function is directly proportional to the number of tasks on the application. Delayed task control mechanism on the scheduler of μC/OS-II brings extra delay while examining all the tasks available on the application (Figure 9.1). When the execution time of the timer tick function increases, the count of the total task preemptions decrease as expected. However FreeRTOS does not

```
void OSTimeTick(void)
{
        OS_TCB *ptcb;
        OSTimeTickHook();
        ptcb = OSTCBList;
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO)
        {
                OS_ENTER_CRITICAL();
                if (ptcb->OSTCBDly != 0)
                {
                        if (--ptcb->OSTCBDly == 0)
                        {
                                if (!(ptcb->OSTCBStat &
                                OS_STAT_SUSPEND))
                                {
                                        OSRdyGrp |= ptcb->OSTCBBitY;
                                        OSRdyTbl[ptcb->OSTCBY]
                                        |= ptcb->OSTCBBitX;
                                }
                                else
                                {
                                        ptcb->OSTCBDly = 1;
                                }
                        }
                }
                ptcb = ptcb->OSTCBNext;
                OS_EXIT_CRITICAL();
        }
        OS_ENTER_CRITICAL();
        OSTime++;
        OS_EXIT_CRITICAL();
}
```

Figure 9.1: μC/OS-II's OS Tick Code [25]

spend time on controlling other tasks (which are defined statically on the system) on timer tick function, that makes it deterministic on this criterion. There is no time loss on entering critical region for all static tasks available on the system. Besides being deterministic, FreeRTOS still slower than μC/OS-II on this criterion. In conclusion, μC/OS-II is faster than FreeRTOS on the task preemption time under load benchmarking criterion however when the number of tasks defined on the system increases, the task preemption performance of μC/OS-II decreases.

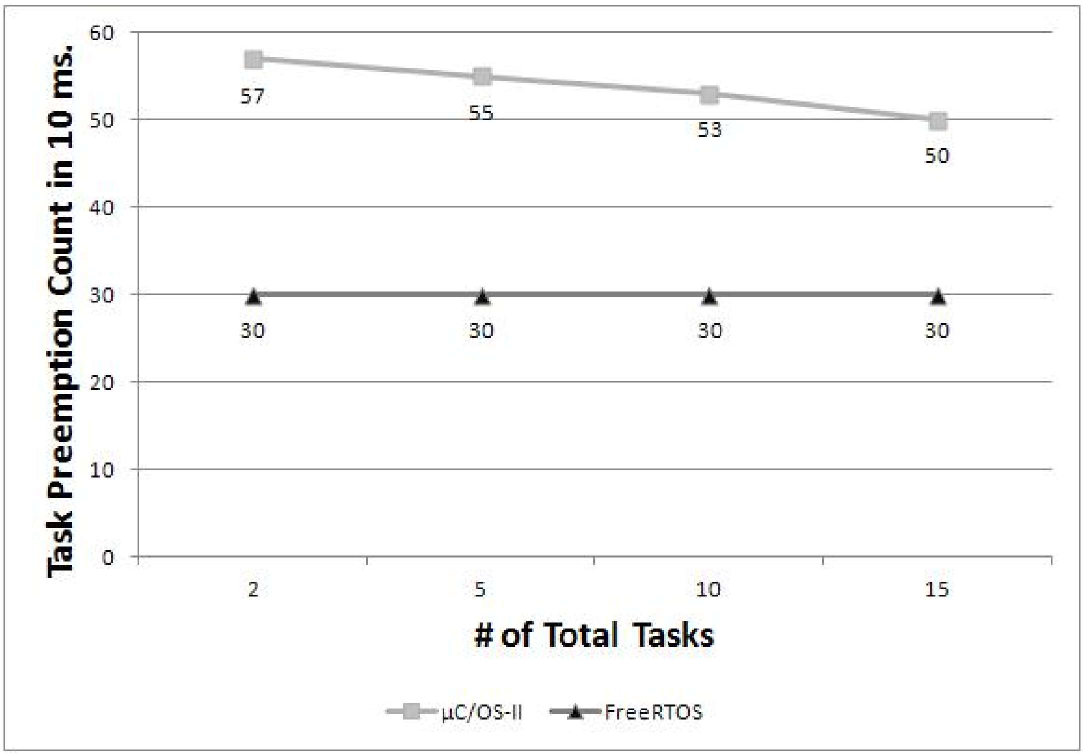


Figure 9.2: Task Preemption Count in 10 ms while Increasing Total Task Number

9.3 GET/RELEASE SEMAPHORE TIME

‘Get/release semaphore time’ comparison gives the number of ‘get semaphore’ and ‘release semaphore’ operations performed on each RTOS in 10 milliseconds. The reasons of low performance of Xilkernel can be the extra validity control of the semaphore when starting semaphore operations. FreeRTOS and μC/OS-II assume that the semaphore is initialized correctly. FreeRTOS semaphores use queues as their underlying mechanism. The queue architecture brings extra delays when performing semaphore operations.

9.4 PASS/RECEIVE MESSAGE TIME

‘Pass/receive message time’ results will compare the number of ‘pass message’ and ‘receive message’ operations performed on each RTOS in 10 milliseconds. Receiving a message from a queue should be faster than passing a message to a queue in μC/OS-II according to the execution time table in [25]. So, our results for ‘pass/receive message time’ test in μC/OS-II are reasonable. Xilkernel makes extra control operations before starting to pass/receive message if the related code segment of Xilkernel is examined. Also, even if no tasks are waiting for a message, the tasks are signalled with the information of the new message. This drops the performance of Xilkernel. FreeRTOS uses the same queue infrastructure as for semaphores and also suspends all tasks when before performing queue operations, then resumes them after operations have been completed. These are the reasons of the slow performance of FreeRTOS on this criterion.

9.5 GET/RELEASE FIXED SIZED DYNAMIC MEMORY TIME

‘Get/release fixed sized dynamic memory time’ results will compare the number of ‘get dynamic memory’ and ‘release dynamic memory’ operations performed on each RTOS in 10 milliseconds. FreeRTOS uses wrapper functions for the ‘malloc’ and ‘free’ implementations of the target compiler, while Xilkernel and μC/OS-II have specialized memory API functions designed for better performance and safety. So, a low performance on FreeRTOS is expected on this criterion from the start. Also, Xilkernel and μC/OS-II can get and release fixed sized dynamic memory partitions that are defined initially while creating the dynamic memory partition, however FreeRTOS can get and release variable sized dynamic memory partitions. For example in Xilkernel and μC/OS-II, if a dynamic memory partition is created having the number of blocks option is 1000 and the size of each block is 32 bytes, the user can only get/release 32 bytes of memory partitions. However, FreeRTOS uses the whole heap memory; if 20 bytes of memory partition is taken from the memory, the user can release 10 bytes of that memory partition. This algorithm of FreeRTOS can cause memory fragmentation problems and can decrease the performance.

Another reason of the low performance of FreeRTOS on this benchmark criterion can be the suspending/resuming operation of all tasks when get/release memory operation command is given. μC/OS-II enters critical region (disabling all the interrupts that may harm the memory operations) and Xilkernel has no protection while performing get/release functions. If the performance data of μC/OS-II is further investigated, we see that getting memory partition operation is faster than releasing it. This claim can be supported with the given execution times of API functions in [25]. To explain the performance difference of Xilkernel and μC/OS-II, we can argue that the function codes of both RTOSs are very similar however Xilkernel checks the memory partition's validity every time the get/release command is issued. μC/OS-II assumes that the dynamic memory created initially is not null and valid.

9.6 UART RS-422 MESSAGE INTERRUPT SERVING COUNT

Interrupt serving results will compare the number of interrupt servings performed on each RTOS in 10 milliseconds. Xilkernel is the fastest RTOS on UART RS-422 message interrupt serving time criterion, while μC/OS-II and FreeRTOS are comparable. On every OS tick, all of the RTOSs control the interrupt controller core for a pending interrupt. When the interrupt is available, the context of the active task is saved and the handler of the interrupt is activated. After the user interrupt code is handled, rescheduling is done and the context of the currently selected task is restored. Since the interrupt handling codes of all RTOSs are practically similar and they are mostly dependant on the hardware architecture of MicroBlaze, the results are actually close. However there is a difference in stack handling concept; FreeRTOS uses a separate interrupt handler stack however the other RTOSs use the kernel stack for interrupt handling process. But since all the RTOSs do not use the running task's stack, this architecture diversity will not bring any performance difference. Interrupt handler code of FreeRTOS uses more hardware specific interrupt controller functions of MicroBlaze than Xilkernel, so this brings extra delay. Note that, if the external UART RS-422 message interrupt occurs when an RTOS is in critical section, the serving time of that interrupt can increase. Critical regions are the code parts, that must not be interrupted for safety of the kernel. So, before entering a critical region, RTOSs disable interrupts and after handling the critical code, the interrupts are enabled again. There is a trade-off between system responsiveness and reliability in critical region handling. So, a kernel with more critical regions inside, decreases the responsiveness of the application. μC/OS-II enters critical region on every OS tick, so if the external interrupt comes during this region, serving time of that interrupt will be delayed. The possibility of an extra delay caused by the critical region concept in μC/OS-II is visualized in Figure 9.3.

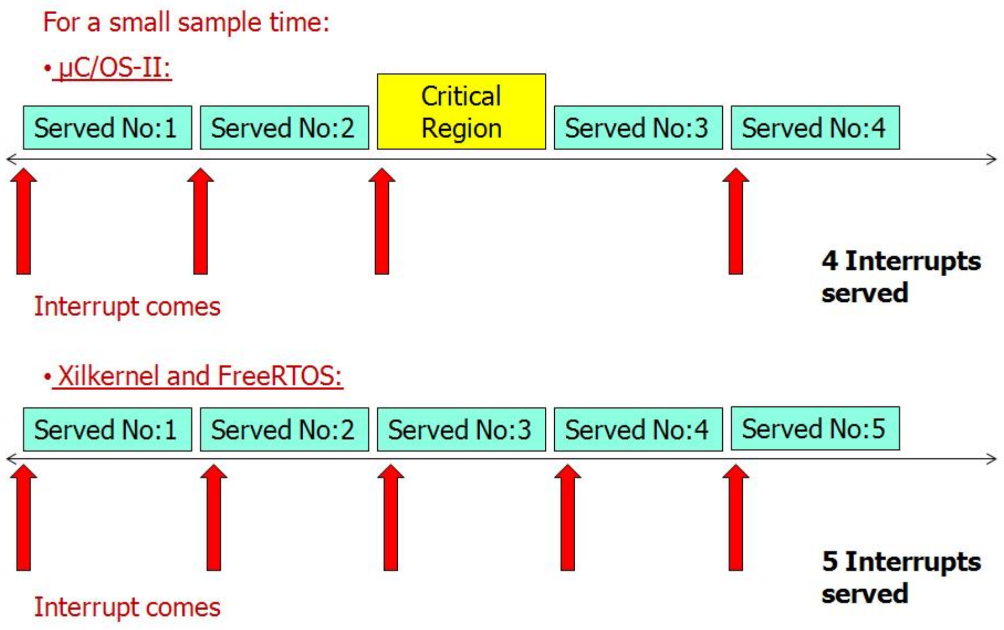


Figure 9.3: Critical Region Concept

9.7 RTOS INITIALIZATION TIME

Xilkernel initialized faster than the other RTOSs on RTOS initialization criterion, while FreeRTOS is the second and μC/OS-II is the slowest one. As we have explained, RTOS initialization time is directly proportional to the complexity of the architecture of an RTOS. A longer initialization time is expected on μC/OS-II since μC/OS-II is rather complicated and more suitable for large-scale projects. To set an example, μC/OS-I has DO-178B certificate which is an avionics certificate related to the safety and quality of the software. This means having more complicated architectures and initializing more functions and global variables. If we examine the codes of μC/OS-II deeper, it is seen that event control blocks (ECB) are used for semaphore, queue and mailbox operations. This ECB architecture is typical to μC/OS-II only, and provides safer background to the user. In summary, having more reliable and complex architectures on the RTOS background increases the initialization time as expected.

Xilkernel is the simplest kernel, which is suitable for small-scale applications, so the initialization time of Xilkernel is faster than the other RTOSs as expected. If we examine the codes of FreeRTOS, we see that a separate interrupt stack is used for interrupt handling operations, so this stack should be initialized and allocated at start-up. This increases the initialization time of FreeRTOS. The functions that are called by RTOSs during initialization are shown on Table 9.2.

Table 9.2: RTOS Initialization Functions

	FreeRTOS	**μC/OS-II**	**Xilkernel**
1.	Idle Task Create	Disable Interrupts	Setup Scheduler
2.	Disable Interrupts	Global Variables Initialization	Idle Task Create
3.	Setup Scheduler	Setup Scheduler	Start First Task
4.	Setup Interrupt Handler Stack	ECB Initialization	
5.	Start First Task	Idle Task Create	
6.		Start First Task	

9.8 MEMORY FOOTPRINT

Xilkernel and μC/OS-II are comparable on the memory footprint criterion. The results have been obtained under same configuration options with no optimization flag on compiler. Both RTOSs have task management and scheduling codes available, which are required for having one task running on an endless loop. Unused configuration options are disabled, which decreases the source code size of RTOSs. These are semaphore, queue, memory and timer management functions and other unused features of RTOSs. If enabled features increase, the code size of RTOSs increase to an extent which is still suitable for small-scale applications so we can conclude that both RTOSs are not limited on memory footprint criterion.

9.9 OVERALL

μC/OS-II has performed better than Xilkernel and FreeRTOS on nearly all of the benchmarking criteria. However Xilkernel is the best on interrupt serving, initialization time and memory footprint comparisons.

If the requirements of a project impose faster response on any of these benchmarking criteria except UART RS-422 message interrupt serving and RTOS initialization time then the selection of RTOS should be μC/OS-II. As far as documentation is concerned, μC/OS-II offers more comprehensive and apparent documentation than the other two RTOSs. Service calls are easily understandable, which simplify designers' work and save their time.

The only disadvantage is that, μC/OS-II has a licence fee. If users want to have that performance on their systems, they have to pay the price.

FreeRTOS came second in overall, only performed worst on acquiring fixed-sized dynamic memory test and UART RS-422 message interrupt serving test. Since it is open-source and free, if money is a critical requirement but also you want to have the advantages of having an RTOS on your system, you can choose FreeRTOS.

Finally, Xilkernel came last in overall. But Xilkernel came top on UART RS-422 message interrupt serving time, RTOS initialization time and memory footprint benchmark tests. This result proves that Xilkernel is a small-scale RTOS and it is highly integrated with the MicroBlaze hardware. If the timing requirements of your project is not strict and you do not want to pay a licence fee, you can use Xilkernel on your system.

CHAPTER 10

CONCLUSION AND FUTURE WORK

Nowadays, designers working on embedded systems are considering using soft processor option more frequently since based on the application, designers feel free to add or remove peripherals according to the requirements of their projects. Also processor performance can be adjusted according to the needs, components on the board can be reduced and processor obsolescence problem can be eliminated. If the requirements force the designer to manage hard timing constraints on their applications and to have precise latencies in terms of context switching, memory handling and interrupts, designers are heading to the RTOS solution increasingly. Selecting an RTOS that most suits the requirements of the project should not be difficult for the designer, which means, no time should be wasted by the designer for researching all of the available RTOSs. A comparison of available RTOSs using the benchmark criteria that fulfill designers' need will accelerate designers' work and provoke RTOS companies to strengthen their products on areas where they are weak.

In this study, three important RTOS products on MicroBlaze are compared according to critical benchmark criteria, i.e. task preemption time, task preemption time under load, get/release semaphore time, pass/receive message time, get/release fixed sized dynamic memory time, UART RS-422 message interrupt serving time, RTOS initialization time and memory footprint data. μC/OS-II is the clear winner of the comparison, since in eight of the eleven benchmarking criteria, μC/OS-II outperformed the other RTOSs. Also, μC/OS-II has the most detailed and easily understandable documentation.

Only drawbacks of μC/OS-II are the low UART RS-422 message interrupt serving and RTOS initialization time performance and licence fee.

FreeRTOS came second overall. With satisfactory real-time performance and being open-source and free, FreeRTOS can be the selection of embedded designers if the advantages of RTOSs are needed on a project.

Xilkernel came last if performance is in question, although it is proved that it is suitable for small-scale applications and it is highly integrated with the hardware architecture and the design tools of MicroBlaze.

For future work, we plan to increase the number of benchmark criteria. Although, all of the crucial benchmarking criteria for real time operating systems are covered in this study, there can be additions to the benchmarking list. Also, these benchmarking criteria can be evaluated on another soft processor platform and the results can be compared with the results obtained in this study.

REFERENCES

[1] Edwards, S., “Microprocessors or FPGAs?: Making the Right Choice,” *RTC Magazine*, pp. 22–25, 2011.

[2] Lattice, “Open and Easy Microprocessor Designs Using the LatticeMico32,” Online: http://www.latticesemi.com/corporate/newscenter/newsletters/newsjanuary2007/mico-32trendschallenges.cfm, last visited on 09.05.2012.

[3] Cannella, E., “Performance Evaluation of Multi-threading Operating Systems in MP-SoCs Generated by ESPAM,” Master’s thesis, University of Udine, Italy, 2008.

[4] Rönnholm, A., “Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU,” Master’s thesis, Malardalens University, Sweden, 2006.

[5] R. Klenke, “Experiences Using the Xilinx Microblaze Softcore Processor and Uclinux in Computer Engineering Capstone Senior Design Projects,” in *Microelectronic Systems Education (MSE’07), IEEE International Conference on*, pp. 123–124, IEEE, 2007.

[6] Anh, T.N.B. and Tan, S.L., “Real-Time Operating Systems for Small Microcontrollers,” *Micro, IEEE*, vol. 29, no. 5, pp. 30–45, 2009.

[7] Ugurel, G. and Bazlamacci, CF, “Context Switching Time and Memory Footprint Comparison of Xilkernel and μC/OS-II on MicroBlaze,” in *Electrical and Electronics Engineering (ELECO), 7th International Conference on*, pp. II–62, IEEE, 2011.

[8] Tong, J.G. and Anderson, I.D.L. and Khalid, M.A.S., “Soft-Core Processors for Embedded Systems,” in *Microelectronics, 2006. ICM’06. International Conference on*, pp. 170–173, IEEE, 2006.

[9] Fletcher, B.H., “FPGA Embedded Processors-Revealing True System Performance,” in *Embedded Systems Conference*, pp. 1–18, 2005.

[10] Xilinx, “MicroBlaze Soft Processor Core,” Online: http://www.xilinx.com/tools/microblaze.htm, last visited on 10.12.2011.

[11] L. Barthe, L. Cargnini, P. Benoit, and L. Torres, “The Secretblaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor,” in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 310–313, IEEE, 2011.

[12] Tensilica, “Tensilica’s HiFi Audio DSP First IP Core to Support SRS’ Advanced StudioSound HD Audio Suite for HDTVs,” Online: http://www.tensilica.com/news/340/330/Tensilica-s-HiFi-Audio-DSP-First-IP-Core-to-Support-SRS-Advanced-StudioSound-HD-Audio-Suite-for-HDTVs.htm, last visited on 10.12.2011.

[13] Altera, “Altera’s DO-254/ED-80 Certifiable Nios II Processor Leveraged in Thales Safety-Critical Avionics System Certified by EASA,” Online: http://www.altera.com/corporate/news_room/releases/2010/products/nr-do-254.html, last visited on 10.12.2011.

[14] D. Timmerman and L. Perneel, “RTOS State of the Art,” *Dedicated Systems*, vol. 9, pp. 1–24, 2009.

[15] R. Moore, “Selecting an Embedded RTOS,” Online: http://www.eg3.com, last visited on 08.07.2012.

[16] Your Electronics Open Source, “RTOS for Embedded Systems,” Online: http://dev.emcelettronica.com/rtos-embedded-systems, last visited on 08.07.2012.

[17] S. Balacco, “VDC Research Webcasts, Embedded/Real-time Operating Systems,” Online: http://www.vdcresearch.com/market_research/embedded_software/freeresearch.aspx, last visited on 10.12.2011.

[18] J. Krasner, “RTOS Selection and Its Impact on Enhancing Time-To-Market and On-Time Design Outcomes.” Online: http://www.embeddedforecast.com, last visited on 08.07.2012.

[19] VxWorks, “The RTOS That Powers More Than 1 Billion Embedded Systems Around the Globe.” Online: http://www.windriver.com/products/vxworks, last visited on July 2012.

[20] Green Hills Software, “Real-Time Operating System.” Online: http://www.ghs.com/products/rtos/integrity.html, last visited on 08.07.2012.

[21] Microchip, “RTOS Selection Guide,” Online: http://www.microchip.com, last visited on 22.04.2012.

[22] Xilinx, “Xilkernel Documentation,” Online: http://forums.xilinx.com/xlnx/attachments /xlnx/EMBEDDED/4789/1/xilkernel_v5_00_a.pdf, last visited on 10.12.2011.

[23] Micrium, “μC/OS-II brochure,” Online: http://micrium.com/newmicrium/uploads/file/datasheets/ucosii_datasheet.pdf, last visited on 10.12.2011.

[24] Holenderski, M., “μC/OS-II,” Online: www.win.tue.nl/~mholende/ooti/ooti_rt_course_ucos.pdf, last visited on 10.12.2011.

[25] Labrosse, J.J., *MicroC/OS-II: The Real-Time Kernel*. Newnes, 2002.

[26] M.-J. Jung, M.-H. Cho, Y.-H. Kim, and C.-H. Lee, “Generalized Deterministic Task Scheduling Algorithm for Embedded Real-Time Operating Systems,” in *ESA Conference*, pp. 79–82, 2006.

[27] Labrosse, J.J., “Use an RTOS on Your Next MicroBlaze-Based Product,” *Xcell Journal*, no. 48, pp. 35–37, 2004.

[28] FreeRTOS, “The FreeRTOS Project,” Online: www.freertos.org, last visited on 08.07.2012.

[29] Kalinsky, D., “Basic Concepts of Real-Time Operating Systems,” Online: http://www.kalinskyassociates.com/Wpaper1.html, last visited on 08.07.2012.

[30] Penumuchu, C.V., *Simple RTOS: A Kernel Inside View for a Beginner.* Trafford, 2007.

[31] Bilgen, S., “Confidence of Simulation Results.” Unpublished Lecture Note, 1986.

APPENDIX A

APPENDIX A

A.1 XILKERNEL CODE SNIPPETS

A.1.1 GET/RELEASE SEMAPHORE TIME

```
void* first_thread(void *arg)
{
        while(1)
        {
                //wait code
                time1 = xget_clock_ticks();
                if(time1 > TIME_A)
                //(TIME_B - TIME_A) is defined as 10ms
                {
                        sem_wait(&sem1);
                        //Xilkernel API call
                }
                if(time1 > TIME_B)
                {
                        sem_getvalue(&sem1,&SemValue);
                        //Xilkernel API call
                }
                //post code
                time1 = xget_clock_ticks();
                if(time1 > TIME_A)
                {
                        sem_post(&sem1);
                        //Xilkernel API call
                }
                if(time1 > TIME_B)
                {
                        sem_getvalue(&sem1,&SemValue);
                }
        }
}
```

A.1.2 PASS/RECEIVE MESSAGE TIME

```
void* first_thread(void *arg)
{
        while(1)
        {
                if(!StartRelease)
                {
                        time1 = xget_clock_ticks();
                        if(time1 > TIME_A)
                        //test starts after a predefined TIME_A
                        {
                                var = msgsnd(q1, &QueueVariable,
                                sizeof(QueueVariable), 0);
                                //Xilkernel API call
                                if (var == 0)
                                {
                                        SendCount++;
                                        if(time2 > TIME_B)
                                                //(TIME_B - TIME_A)
                                                //is defined as 10ms
                                                StartRelease = 1;
                                                //Start releasing
                                                //messages

                                }
                        }
                }
                else
                {
                        time1 = xget_clock_ticks();
                        if(time1 > TIME_C)
                        {
                                var = msgrcv(q1, &RecvVariable,
                                sizeof(RecvVariable), 0, 1);
                                //Xilkernel API call
                                if (var == 0)
                                {
                                        ReceiveCount++;
                                        if(time2 > TIME_D)
                                        //(TIME_D - TIME_C) is
                                        //defined as 10ms
                                                TestEnd = 1;
                                }
                        }
                }
        }
}
```

A.1.3 GET/RELEASE FIXED SIZED MEMORY TIME

```
void* first_thread(void *arg)
{
        while(1)
        {
                if(!StartRelease)
                {
                        time1 = xget_clock_ticks();
                        if(time1 > TIME_A)
                        {
                                msg = bufmalloc(mbuft,128);
                                //Xilkernel API call
                                if (msg != (void *)0)
                                {
                                        GetCount++;
                                        if(time1 > TIME_B)
                                                //(TIME_B - TIME_A)
                                                //is defined as 10ms
                                                StartRelease = 1;
                                                //Start Releasing
                                                //Memory
                                }

                        }
                }
                else
                {
                        time1 = xget_clock_ticks();
                        if(time1 > TIME_C)
                        {
                                buffree(mbuft,msg);
                                //Xilkernel API call
                                ReleaseCount++;
                                if(time1 > TIME_D)
                                //(TIME_D - TIME_C)
                                //is defined as 10ms
                                        EndTest = 1;
                        }
                }
        }
}
```

A.1.4 UART RS-422 MESSAGE INTERRUPT SERVING TIME

```
void* first_thread(void *arg)
{
        while(1)
        {
                time1 = xget_clock_ticks();
                if(time1 > TIME_A)
                {
                        if(SendMessage)
                        {
                                XUartLite_Send(&RS422TestChannel,
                                TestArray, SendCount);
                                SendMessage = false;
                        }
                        if(time1 > TIME_B)
                        //(TIME_B - TIME_A) is defined as 10ms
                                TestEnd = 1;
                }
        }
}

void TestHandler()
{
        InterruptCounter++;
        SendMessage = true;
}
```

A.1.5 RTOS INITIALIZATION TIME

```
void* first_thread(void *arg)
{
        while(1)
        {
                if(Initialized)
                {
                        XUartLite_Send(&RS422TestChannel,
                        TestArray, SendCount);
                        Initialized = 0;
                }
        }
}
```

APPENDIX B

APPENDIX B

B.1 μC/OS-II CODE SNIPPETS

B.1.1 TASK PREEMPTION TIME

```
static void FirstTask(void *p_arg)
{
        while(1)
        {
                time1 = OSTimeGet();
                if(time1 > TIME_A)
                {
                        TaskPreemptionCount++;
                        OSTaskSuspend(TASK1_PRIO);
                        //RTOS API call
                }
                if(time1 > TIME_B)
                //(TIME_B - TIME_A) is defined as 10ms
                {
                        TestEnd = 1;
                }
        }
}

static void SecondTask(void *p_arg)
{
        while(1)
        {
                OSTaskResume(TASK1_PRIO);
                //RTOS API call
        }
}
```

B.1.2 GET/RELEASE SEMAPHORE TIME

```
static void FirstTask(void *p_arg)
{
        while(1)
        {
                //wait code
                time1 = OSTimeGet();
                if(time1 > TIME_A)
                {
                        OSSemPend(SharedDataSem, 0, &err2);
                        //RTOS API call
                }
                if(time1 > TIME_B)
                //(TIME_B - TIME_A) is defined as 10ms
                {
                        OSSemQuery(SharedDataSem, &sem_data);
                        //RTOS API call
                        SemCount = sem_data.OSCnt;
                }

                //post code
                time1 = OSTimeGet();
                if(time1 > TIME_A)
                {
                        OSSemPost(SharedDataSem);
                        //RTOS API call
                }
                if(time1 > TIME_B)
                //(TIME_B - TIME_A) is defined as 10ms
                {
                        OSSemQuery(SharedDataSem, &sem_data);
                        //RTOS API call
                        SemCount = sem_data.OSCnt;
                }
        }
}
```

B.1.3 PASS/RECEIVE MESSAGE TIME

```
static void FirstTask(void *p_arg)
{
        while(1)
        {
                if(!StartRelease)
                {
                        time1 = OSTimeGet();
                        if(time1 > TIME_A)
                        {
                                err = OSQPost(CommQ,
                                (void *)&QueueValue);
                                //RTOS API call
                                if (err == 0)
                                {
                                        SendCount++;
                                        if(time1 > TIME_B)
                                        //(TIME_B - TIME_A)
                                        //is defined as 10ms
                                                StartRelease = 1;
                                                //Start releasing
                                                //messages
                                }
                        }
                }
                else
                {
                        time1 = OSTimeGet();
                        if(time1 > TIME_C)
                        {
                                msg = OSQPend(CommQ, 0, &err);
                                //RTOS API call
                                if(err == 0)
                                {
                                        ReceiveCount++;
                                        if(time1 > TIME_D)
                                        //(TIME_D - TIME_C)
                                        //is defined as 10ms
                                                TestEnd = 1;
                                }

                        }
                }
        }
}
```

B.1.4 GET/RELEASE FIXED SIZED MEMORY TIME

```
static void FirstTask(void *p_arg)
{
        while(1)
        {
                if(!StartRelease)
                {
                        time1 = OSTimeGet();
                        if(time1 > TIME_A)
                        {
                                msg = OSMemGet(CommMem, &err);
                                //RTOS API call
                                if (msg != (INT8U *)0)
                                {
                                        GetCount++;
                                        if(time1 > TIME_B)
                                        //(TIME_B - TIME_A) is
                                        //defined as 10ms
                                                StartRelease = 1;
                                                //Start releasing
                                                //memory
                                }
                        }
                }
                else
                {
                        time1 = OSTimeGet();
                        if(time1 > TIME_C)
                        {
                                OSMemPut(CommMem, (void *)msg);
                                //RTOS API call
                                ReleaseCount++;
                                if(time1 > TIME_D)
                                //(TIME_D - TIME_C) is
                                //defined as 10ms
                                        EndTest = 1;
                        }
                }
        }
}
```

B.1.5 UART RS-422 MESSAGE INTERRUPT SERVING TIME

```
static void FirstTask(void *p_arg)
{
        while(1)
        {
                time1 = OSTimeGet();
                if(time1 > TIME_A)
                {
                        if(SendMessage == 1)
                        {
                                XUartLite_Send(&RS422TestChannel,
                                TestArray, SendCount);
                                SendMessage = 0;
                        }
                        if(time1 > TIME_B)
                        //(TIME_B - TIME_A) is defined as 10ms
                                EndTest = 1;
                }
        }
}

void TestHandler()
{
        InterruptCounter++;
        SendMessage = true;
}
```

B.1.6 RTOS INITIALIZATION TIME

```
static void FirstTask(void *p_arg)
{
        while (1)
        {
                if(Initialized)
                {
                        XUartLite_Send(&RS422TestChannel,
                        TestArray, SendCount);
                        Initialized = 0;
                }
        }
}
```

}

APPENDIX C

APPENDIX C

C.1 FREERTOS CODE SNIPPETS

C.1.1 TASK PREEMPTION TIME

```
static void prvTaskA(void* pvParameters)
{
        while(1)
        {
                vTaskResume(TaskB);
                //FreeRTOS API call
        }
}

static void prvTaskB(void* pvParameters)
{
        while(1)
        {
                time1 = (unsigned int)xTaskGetTickCount();
                if(time1 > TIME_A)
                {
                        TaskPreemptionCount++;
                        vTaskSuspend(TaskB);
                        //FreeRTOS API call
                }
                if(time1 > TIME_B)
                //(TIME_B - TIME_A) is defined as 10ms
                {
                        TestEnd = 1;
                }
        }
}
```

C.1.2 GET/RELEASE SEMAPHORE TIME

```
static void prvTaskA(void* pvParameters)
{
        while(1)
        {
                //post code
                time1 = (unsigned int)xTaskGetTickCount();
                if(time1 > TIME_A)
                {
                        if(xSemaphoreGive(SemaphoreA) == pdFALSE)
                        //FreeRTOS API call
                        {
                                time2 = (unsigned int)
                                xTaskGetTickCount();
                        }
                }

                //wait code
                time1 = (unsigned int)xTaskGetTickCount();
                if(time1 > TIME_B)
                //(TIME_B - TIME_A) is defined as 10ms
                {
                        if(xSemaphoreTake(SemaphoreA,
                        (portTickType)0) == pdFALSE)
                        //FreeRTOS API call
                        {
                                time2 = (unsigned int)
                                xTaskGetTickCount();
                        }
                }
        }
}
```

C.1.3 PASS/RECEIVE MESSAGE TIME

```
static void prvTaskA(void* pvParameters)
{
        while(1)
        {
                if(!StartRelease == 1)
                {
                        time1 = (unsigned int)xTaskGetTickCount();
                        if(time1 > TIME_A)
                        {
                                noerr = xQueueSend(xQueue1,
                                (void *) &QueueSendData,0);
                                //FreeRTOS API call
                                if (noerr == 1)
                                {
                                        SendCount++;
                                        if(time1 > TIME_B)
                                        //(TIME_B - TIME_A) is
                                        //defined as 10ms
                                                StartRelease = 1;
                                }

                        }
                }
                else
                {
                        time1 = (unsigned int)xTaskGetTickCount();
                        if(time1 > TIME_C)
                        {
                                noerr = xQueueReceive(xQueue1,
                                &QueueReceiveData,0);
                                //FreeRTOS API call
                                if (noerr == 1)
                                {
                                        ReceiveCount++;
                                        if(time1 > TIME_D)
                                        //(TIME_D - TIME_C) is
                                        //defined as 10ms
                                                TestEnd = 1;
                                }
                        }
                }
        }
}
```

C.1.4 GET/RELEASE FIXED SIZED MEMORY TIME

```
static void prvTaskA(void* pvParameters)
{
        while(1)
        {
                if(!StartRelease == 1)
                {
                        time1 = (unsigned int)xTaskGetTickCount();
                        if(time1 > TIME_A)
                        {
                                msg = pvPortMalloc(128);
                                //FreeRTOS API call
                                if (msg != (void *)0)
                                {
                                        GetCount++;
                                        if(time1 > TIME_B)
                                        //(TIME_B - TIME_A) is
                                        //defined as 10ms
                                                StartRelease = 1;
                                }
                        }
                }
                else
                {
                        time1 = (unsigned int)xTaskGetTickCount();
                        if(time1 > TIME_C)
                        {
                                vPortFree(msg);
                                //FreeRTOS API call
                                ReleaseCount++;
                                if(time1 > TIME_D)
                                //(TIME_D - TIME_C)
                                //is defined as 10ms
                                        EndTest = 1;
                        }
                }
}
```

C.1.5 UART RS-422 MESSAGE INTERRUPT SERVING TIME

```
static void prvTaskA(void* pvParameters)
{
        while(1)
        {
                time1 = (unsigned int)xTaskGetTickCount();
                if(time1 > TIME_A)
                {
                        if(SendMessage == 1)
                        {
                                XUartLite_Send(&RS422TestChannel,
                                TestArray, SendCount);
                                SendMessage = 0;
                        }
                        if(time1 > TIME_B)
                        //(TIME_B - TIME_A) is
                        //defined as 10ms
                                EndTest = 1;
                }
        }
}
void TestHandler()
{
        InterruptCounter++;
        SendMessage = true;
}
```

C.1.6 RTOS INITIALIZATION TIME

```
static void prvTaskA(void* pvParameters)
{
        while(1)
        {
                if(Initialized)
                {
                        XUartLite_Send(&RS422TestChannel,
                        TestArray, SendCount);
                        Initialized = 0;
                }
        }
}
```