Asymmetric Multiprocessing Real Time Operating System on Multicore Platforms

by

Girish Rao Bulusu

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved September 2014 by the
Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Georgios Fainekos
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

The need for multi-core architectural trends was realized in the desktop computing domain fairly long back. This trend is also beginning to be seen in the deeply embedded systems such as automotive and avionics industry owing to ever increasing demands in terms of sheer computational bandwidth, responsiveness, reliability and power consumption constraints. The adoption of such multi-core architectures in safety critical systems is often met with resistance owing to the overhead in migration of the existing stable code base to the new system setup, typically requiring extensive re-design. This also brings about the need for exhaustive testing and validation that goes hand in hand with such a migration, especially in safety critical real-time systems.

This project highlights the steps to develop an asymmetric multiprocessing variant of Micrium μC/OS-II real-time operating system suited for a multi-core system. This RTOS variant also supports multi-core synchronization, shared memory management and multi-core messaging queues.

Since such specialized embedded systems are usually developed by system designers focused more so on the functionality than on the coding standards, the adoption of automatic production code generation tools, such as SIMULINK's Embedded Coder, is increasingly becoming the industry norm. Such tools are capable of producing robust, industry compliant code with very little roll out time. This project documents the process of extending SIMULINK's automatic code generation tool for the AMP variant of μC/OS-II on Freescale's MPC5675K, dual-core Microcontroller Unit. This includes code generation from task based models and multi-rate models. Apart from this, it also describes the development of additional software tools to allow semantically consistent communication between task on the same kernel and those across the kernels.

To all who supported me

ACKNOWLEDGEMENTS

I am extremely grateful to my supervisors, Prof. Yann-Hang Lee, Prof. Fainekos and Prof. Wu for their continuing support and invaluable advice throughout. Also to my teammate Dhiraj Shetty.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 INTRODUCTION 1

1.1 Motivation 1

1.1.1 Need for Multicore in Embedded Systems 1

1.1.2 Problems With Multi-Core Adoption in Embedded Systems ... 2

1.1.3 AMP-RTOS and SMP-RTOS on Multi-Core 4

1.1.4 Need for Automatic Code Generation 8

1.2 Contribution and Method 9

1.3 Document Outline 10

2 BACKGROUND 11

2.1 Multicore Microcontroller Units 11

2.1.1 True Parallelism with Multicore 12

2.2 PowerPC and E200Z7 Processor 13

2.2.1 Atomic Instructions 15

2.3 Freescale MPC5675K 18

2.3.1 Lock Step Mode and Decoupled Parallel Mode 18

2.3.2 Hardware Semaphore Peripheral 19

2.4 Real-Time Operating Systems and μC/OS-II 20

2.4.1 Real-Time Kernels and RTOS 20

2.4.2 Scheduler and Rate Monotonic Scheduling 22

2.4.3 Lifetime of a μC/OS-II Task 23

2.4.4 Priorities and Ready List 24

CHAPTER Page

2.4.5 Time Management 25
2.4.6 Mutual Exclusion and Synchronization 26
2.4.7 Message Queues 28
2.4.8 Memory Management 31
2.5 Automatic Code Generation and μC/OS-II 33
2.5.1 Model Based Design and Rapid Prototyping 33
2.5.2 Code Generation Architecture and TLC Process 34
2.5.3 Program Execution of Generated Code 39
3 LITERATURE REVIEW AND RELATED WORKS 45
3.1 FreeRTOS and Multicore 45
3.2 Multiprocessors Synchronization protocol 47
3.3 Semantics Preserving Multi-Task Implementation 49
3.4 Three-Slot Asynchronous Reader-Writer Mechanism for Multicore 50
4 PROBLEM ANALYSIS AND REQUIREMENTS 52
4.1 Overview of Objectives 52
4.2 Functional Requirements 53
4.2.1 μC/OS-II System Level Requirements 53
4.2.2 μC/OS-II Additional Services Requirements 53
4.2.3 Requirements of μC/OS-II Support on SIMULINK 53
5 DESIGN 55
5.1 μC/OS-II for MPC5675k in AMP mode 56
5.1.1 Memory Layout 56
5.1.2 Initialization and Bootstrapping 59
5.2 Inter-Core Communication 62

CHAPTER Page

5.3 Mutual Exclusion - Spinlocks 63
5.3.1 Mutual Exclusion using Semaphore Peripheral 63
5.3.2 Mutual Exclusion using Atomic CAS 64
5.4 Inter-Core Barriers 65
5.4.1 Barriers using Semaphore Peripheral 65
5.5 μC/OS-II Global Counting Semaphores 66
5.5.1 Global Semaphore Data Structure Setup 67
5.5.2 Global Semaphore Algorithm 69
5.5.3 Global Semaphore with PCP Algorithm 76
5.6 μC/OS-II Shared Memory Management 78
5.6.1 Shared Memory Management Data Structure and Initial Setup . 79
5.6.2 Shared Memory Management Algorithm 80
5.7 μC/OS-II Shared Memory Queues 82
5.7.1 Shared Memory Queues Data Structure Setup 84
5.7.2 Shared Memory Queues Algorithm 87
5.8 Task Model for μC/OS-II on Simulink embedded coder 96
5.9 Multirate Model for μC/OS-II on Simulink Embedded Coder 98
5.9.1 Initial Setup for Multi-rate Code Generation 99
5.9.2 Multi-rate Process Execution Algorithm 100
5.10 Dynamic Buffering Protocol-Single Core RT SIMULINK Block 102
5.10.1 Data Structures Required 104
5.10.2 Intra-core Communication Protocol Algorithm 106
5.11 Multicore Rate Transition SIMULINK Block 111
5.11.1 Initial Setup and Data Structures Required 113

CHAPTER Page

5.11.2 Inter-Core Communication Protocol Algorithm 115
6 IMPLEMENTATION 119
6.1 Porting μC/OS-II for MPC5675K in AMP Mode 119
6.1.1 Project Directory Structure 119
6.1.2 Porting steps in μC/OS-II/BSP 126
6.1.3 Porting Steps for Hardware Specific and Startup Code 130
6.2 Codewarrior Setup and Debugging Tools 132
6.2.1 Codewarrior Setup for AMP Variant Of μC/OS-II 132
6.2.2 Importing and Compiling the Code 133
6.2.3 Setting Core B Entry Point 134
6.2.4 Debugging the Code 135
7 EVALUATION 136
7.1 High Level Testing Strategy 136
7.2 Testing the μC/OS-II porting for Core_A and Core_B 137
7.2.1 Verify μC/OS-II Porting: OSTaskStkInit() and OSStartHighRdy() 137
7.2.2 Verify μC/OS-II Porting: OSCtxSw() 140
7.2.3 Verify μC/OS-II Porting: OSIntCtxSw() and OSTickISR() 141
7.2.4 Performance Improvement with Respect To Single Core 143
7.3 Testing Mutual Exclusion Primitives 146
7.3.1 Testing Mutual Exclusion Functionality Based on Atomic CAS 147
7.3.2 Testing Mutual Exclusion Functionality - Hardware Semaphore 149
7.3.3 Comparison of Spin Locks-Atomic CAS and HWSEMA4 150
8 CONCLUSION and FUTURE WORK 152

CHAPTER Page

REFERENCES 154

LIST OF TABLES

Table Page

6.1 Folder Structure:Application Specific Code 120

6.2 Folder Structure:μC/OS-II and μC/CPU Configuration 121

6.3 Folder Structure:μC/OS-II Source Code 122

6.4 Folder Structure:μC/LIB Libraries 123

6.5 Folder Structure:Processor Specific Code 124

6.6 Folder Structure:Board Support Package 124

6.7 Folder Structure:Hardware Specific and Startup Code 125

6.8 Folder Structure:Global Functions 125

7.1 Performance Measurement of Test Case(a) 145

7.2 Performance Measurement of Test Case(b) and Test Case(d) 146

7.3 Performance Measurement of Test Case(d) 146

7.4 Performance Comparison of Spin-locks 150

LIST OF FIGURES

Figure		Page
2.1	Atomic Compare and Swap Function	17
2.2	Hardware Semaphore Gate Finite State Machine	20
2.3	Code Generation Architecture and TLC Process	35
2.4	Single Rate Program Execution	40
2.5	Multi-Rate Program Execution	42
2.6	Tornado- rtwdemo_mrmtos_main Task	43
2.7	Tornado tBaseRate- Base Rate Function	44
5.1	Memory Layout	57
5.2	Boot Sequence	60
5.3	Spinlocks using Hardware Semaphore Peripherals	64
5.4	Global Semaphore Data Structures Setup	67
5.5	Global Semaphore Timing Diagram: No Resource Contention	71
5.6	Global Semaphore Timing Diagram: With Resource Contention	73
5.7	Global Semaphore with PCP Timing Diagram	77
5.8	Global Queues Data Structures Setup	84
5.9	Global Queues Case 1: Producer Enqueues Before Consumer Dequeues	89
5.10	Global Queues Case 2: Consumer Tasks Pend Before Producer Enqueues	91
5.11	μC/OS-II Task - SIMULINK Block	98
5.12	μC/OS-II Multi-Rate Model Initialization	102
5.13	DBP based SIMULINK RT Block- Data Structures Required	104
5.14	DBP Based Rate Transition Block: Model Initilzation	108
5.15	DBP Based Rate Transition Block: Timer Callback Function	109
5.16	DBP Based Rate Transition_Block: Reader Writer Functions	110
5.17	3-ACM Global Rate Transition_Block: Model Setup and Data Structures	114

Figure Page

5.18 3-ACM Global Rate Transition_Block: Writer side algorithm 117

5.19 3-SAM Global Rate Transition_Block: Reader side algorithm 118

7.1 Initialization of Task Stack: CPU Registers 139

7.2 AMP vs Single Core Performance: Test Setup (a) and (b) 144

7.3 AMP vs Single Core Performance: Test Setup (c) and (d)..... 145

7.4 Test Setup: Spin-lock using Atomic CAS..... 147

Chapter 1

INTRODUCTION

The central idea behind this project revolves around the use of μC/OS-II, a real-time operating systems (RTOS), in an Asymmetric Multiprocessing (AMP) mode of operation on a multi-core system. The hardware used in this project is the Freescale Qorivva MPC5675K, Power Architecture 32-bit Micro-controller Units (MCU). The choice of hardware and the operating system is representative of a typical industry setup seen especially in the automotive domain. However, the concepts explored within the scope of this project can be easily extended to any other configuration. This project serves to document the process of porting an existing uni-processor real-time operating system to an AMP mode of operation along with support for multi-core synchronization and message passing capabilities. The project also describes the process of extending SIMULINK's Embedded Coder to support this above mentioned configuration. Such an automatic code generation framework would serve as the starting point for a more customized code generation frameworks based on the applications intended to be designed and developed.

1.1 Motivation

1.1.1 Need for Multicore in Embedded Systems

Traditionally, the prevalent trend within the semiconductor industry in the design of processors revolved around the shrinking of the die size and increasing the operating frequency. However in the earlier years of computing, particularly in the 1970s, the physical limitation that uni-processor systems inherently possessed was realized. This notion became more concrete in the 1980s because of which a number of parallel computing machines were built for commercial purposes. Many prominent researcher such as Stone

and Cocke proposed that 250 MHz would be the highest operating frequency attainable on a uni-processor system [44] and any further increase would be constrained by physical factors. However, this upper-limit on the CPU clock rate was soon disproved and the market focus again shifted back to the ramp up of the uni-processor system's clock rates. As the demand for high performance computing began to emerge in the embedded systems domain, such as sophisticated signal processing, the need for higher MIPS performance was seen. This increase in complexity of systems drove the operating frequency of micro-processors till it began to peak at several billion operations per second, well within the giga-hertz operating range.

The sheer increase in operating frequency led to increased computational bandwidth in terms of instruction executed per second, however the negative effects of pushing the limits could no longer be ignored. The most critical problem being that of heat dissipation. Also, there was an inherent upper limit to this approach as most chip designers had hit an architectural wall in terms of designing sophisticated pipelining techniques. These techniques had reached a point where any further optimization in terms of pipelining for the best case scenario would only result in much greater degradation in performance during worst case execution [23]. Keeping in mind the above mentioned thermal, architectural, physical and systemic software issues, the need to transition to multi-processor systems seemed as the next logical step.

1.1.2 Problems With Multi-Core Adoption in Embedded Systems

On narrowing the discussion to deeply embedded systems such as Automotive Embedded Systems, there are many more factors which influence the move from single core to multi-core systems. According to a study performed by VDC [19] the adoption of multi-processor in the communications processors market was almost 3 times more likely than in the automotive domain. The study highlighted that the main cause for this slow adop-

tion was the software rework required to migrate the legacy code from the uni-processor to multi-processor systems. This migration, especially in safety critical systems, is met with great resistance owing to the need for extensive and thorough re-testing and validation. Another cause to this slow adoption was the nature of the applications and the inherent sequential nature of C/C++ programs, which is the most prevalent language for such domains. Parallel programming considerations proved to be major paradigm shift for most experienced developers who were accustomed to sequential programming.

Despite the issues mentioned above, the 2 main reasons tipping the scales in the favor of adoption of mulit-processor systems in the deeply embedded domain are the economic and technical reasons [47]. Taking the example of the automotive industry, on an average, today's vehicles incorporate up to 70 ECUs with a sum total of about 20 million lines of code running across these ECUs. This steady increase in the number of ECUs and the effort spent in the software development for these ECUs in a car is estimated to account for almost 40% of the automobile's production cost [18]. Such an approach is at a watershed owing to limited composability, increased communication delays and decreasing probability of fault isolation and error containment [24].

The move to multi-core SoCs that communicate via some form message passing or shared memory schemes eliminates the need for multiple ECUs. Such a consolidation helps in minimizing communication delays between the ECUs, driving down production costs and facilitates precise fault and error detection. Also, such a multi-core setup can be used to provide more robust fault tolerance mechanisms by utilizing multi-core SoCs for error detection and to maintain high availability in the event of a failure of one of the processors. However, the issue of increasing complexity of software design and development in such a multi-processor setup still needs to be addressed. The study of some of these software issues was an area of focus during the course of this project.

1.1.3 AMP-RTOS and SMP-RTOS on Multi-Core

A real-time system, in its most general form, can be defined as a computer systems that is required to adhere to some bounded response time constraints in a deterministic manner. On failing to do so, it can run the risk of causing some catastrophic system failures [27]. This definition of real-time systems brings into focus three key aspects, namely, temporal correctness, deterministic behavior and the severity of system failure associated if these criteria are not met. The ability of a system to cope with certain temporal and behavioral failures enables one to categorize these systems as "hard", "firm" and "soft" real time systems. These terms are defined in detail in the following sections.

The common underlying thread to all these categories of real-time systems is the adherence to some form of temporal behavioral in a deterministic manner. Having said that, an over-simplified argument would be that greater the computational capacity of the system, less likely would be the occurrence of missed deadlines and an increase in responsiveness.

However, with multi-core real time system, designers would have a major challenge in terms of determining the worst case execution times in such an environment. The system could essentially lose its determinism and reliability. Hence the most important thing that can be made available to such systems designers is an easy to use application development environment. A Real Time Operating System (RTOS) can provide some degree of abstraction to the designers hence reducing the development effort of a real-time system.

A Real Time Operating System (RTOS) acts like the management system between the application and the hardware resources. Most of the hardware related design time considerations can be abstracted from the application designers in this manner. The RTOS would also provide the capability to schedule the real-time tasks hence allowing the resource requests by tasks to be serviced in a timely manner. The APIs exposed by the

RTOS would allow application designers to focus on the functionality and their use in designing more complex system, rather than being concerned about the underlying implementation details of these services.

The two broad classification amongst RTOS are Uni-Processor RTOS (UP-RTOS), and Multi-Processor RTOS (MP-RTOS). MP-RTOS can further be classified as Symmetric Multi-Processor RTOS (SMP-RTOS) and Asymmetric Multi-Processor RTOS (AMP-RTOS).

SMP RTOS and Global Scheduling

In a SMP-RTOS setup all the underlying processors share a common memory and one instance of the thread safe operating system code. An SMP-RTOS typically has a global queue containing all tasks and at any one point of time 'm' highest priority tasks would be dynamically scheduled on the 'm' available processors. Following is a listing of some of the advantages and disadvantages to global scheduling schemes typically seen in SMP-RTOS. Based on the inherent disadvantages of an SMP-RTOS setup with global scheduling, the decision to adopt an AMP-RTOS setup with partitioned scheduling was seemed obvious.

Advantages:

1. SMP-RTOS with global scheduling are best suited in a memory constrained systems since only one instance of the RTOS is needed in memory [8].

2. Programming software for an SMP-RTOS with synchronization routines, such as spin-locks, mutexes and semaphores, becomes easier since only one set of these synchronization routines would be used to manage the tasks on the multicore system [8].

3. Extensive study done in the area of queuing theory [22] has shown that tasks scheduled using a single global FIFO queue of tasks on a multi-processor setup exhibit better average response times. Since there exists a single ready queue maintained by one kernel, all ready to run tasks would be scheduled to execute on the available processors and if needed tasks can be migrated from one core to another. This allows for better CPU utilization and load balancing between the processors [7].

4. In a multi-core setup with m processors, if more than m tasks have individual task utilization greater than 0.5, it is empirically seen that global scheduling policies offer better overall utilization as compared to portioned scheduling schemes [2].

5. Other scheduling schemes such as the PFAIR scheduling proposed by Baruah, Cohen, Plaxton and Varvel [3] have been proven to be optimal for scheduling periodic tasks on a multiprocessor setup. These scheduling protocols have has a linear-time complexity to perform the necessary and sufficient schedulability test. However since the PFAIR scheduling algorithm relies upon the breaking up of tasks into subtasks of smaller time slices it consequently leads to high implementation overhead [2].

Disadvantages

1. Since there is a single FIFO queue to schedule the tasks, synchronization overheads are considerably higher when scheduling tasks concurrently for multiple processors. Repeated kernel level locks to make the shared operating system code reentrant can hit the execution time bounds of the tasks adversely, leading to high worst case latencies [2].

2. Since task migration is essential in a dynamic scheduling scheme, inter-processor interrupts and cache reloading that is required for such a task migration can prove to be expensive when resuming a task on different processor [2].

3. There are no known necessary and sufficient schedulability tests available for global fixed priority scheduling which can be performed in less than exponential time [2]. Some test sets, which even though have only close to 1 total utilization demand, cannot be scheduled using the global EDF or global RM policies on an multi-core system (here utilization demand is lower than the number of processors). Such tasks sets typically contain a mix of both high and low utilization tasks present together. This phenomena is called the Dhall's effect [? , p. 281sha2006embedded]

AMP RTOS and Partitioned Scheduling

In way of comparison, each processor executes a distinct instance of the RTOS on an AMP-RTOS setup. In an AMP setup typically a partitioned scheduling scheme is followed where each task may only execute on a fixed processor and separate dispatch queues for each processor.

Advantages:

1. Most techniques for single-processor scheduling are also applicable here especially if the task sets are completely independent [2].

2. There is wide body of work which proves that partitioning-based scheduling algorithms on multi-core systems offers better real-time performance, especially with respect to hard real-time systems, as compared to global scheduling algorithms [48].

3. Facilitates integration of legacy code written for 2 or more independent UP-RTOSs into an AMP-RTOS without considerable re-design.

Disadvantages:

1. Such a duplication of the RTOS code leads to a greater memory footprint [7].

2. Synchronization routines becomes difficult to implement due independent RTOS code and partitioned queues

3. Such an analysis to split up tasks for each of the dispatch queues, translates to a greater design time over-head [45]. This analysis required to find an optimal distribution of tasks across the cores falls in the class of bin-packaging problems is known to be a NP-hard problem and cannot be performed in polynomial time [38]. However some heuristics based algorithms such as the Rate Monotonic First Fit (RMFF) proposed by Dhall and Liu offer close to optimal solutions [21].

Despite the above mentioned disadvantages, the inherent advantages to AMP-RTOS and the disadvantages of SMP-RTOS have made AMP-RTOS the choice of setup for this project.

1.1.4 Need for Automatic Code Generation

Taking the case of embedded systems design and development, where often cross-functional designers from the mechanical, electrical and software domains work in collaboration, the ability to understand the details of each of these domains can prove to be challenging.

With this in mind it has become the need of the day to increase the abstraction level of the projects and provide faster code production techniques which is also compliant with industry standards. With the help of tools allowing Model Based Design and Automatic Production Code Generation, consistent co-design of components is possible starting all the way from the design phase to the code production phase [35]. Also, model-based

designs can facilitate code-reuse and allow greater testing coverage. These factors are vital in driving down the cost of such complex embedded systems.

With the help of tools like SIMULINK's Embedded Coder the process of system design and production code generation can become a seamless process allowing easy integration of components at any phase - from requirements gathering to the final product integration phase. This thesis project includes the setup, and some of the additional functionalities, added to the Embedded Coder framework to allow easy development of applications on the Freescale MPC5675K running μC/OS-II in AMP mode.

1.2 Contribution and Method

This thesis document presents the study done in taking some of the design decisions over the course of the project. Some of these decisions were driven owing to the advantages and optimality they offered and some chosen owing to their ease of implementation which might not be the most optimal solutions. However with this framework in place, it would not require considerable change to implement other more optimal solutions, as suited by the need of the application.

Unlike the commonly followed scheme of requirement gathering, wherein within the first few iterations of the development cycle all of the requirements are identified, in this project the requirements were captured in an evolutionary and adaptive fashion. Starting with the core requirements, which was the need to support a real-time operating system on the provided multi-core hardware and to support code generation for μC/OS-II, other requirements were added to the pool over every development iteration. Further requirements were mostly identified keeping in mind the typical issues a system designer or application developer, using such a framework, would face over the course of a similar project. The finalized set of requirements presented within this body of work should be

the base functional requirements upon which more specific functional and non-functional requirements can be added.

1.3 Document Outline

The rest of the document is organized as follows. Chapter 2 provides Background information about μC/OS-II, its architecture.It also explains the code generation process for single rate and multi-rate system using SIMULINK.

Chapter 3 mentions the related work done in the area of AMP RTOS setups, dynamic buffering protocol and the three-slot asynchronous data sharing semantics.

Chapter 4 provides a list of the formalized requirements of this thesis project.

Chapter 5 talks about the design of the additional μC/OS-II related capabilities that have been added in this project. It also provides details of the multi-rate code generation support for μC/OS-II from SIMULINK models and the semantic preserving data sharing protocols implemented as blocks within SIMULINK.

Chapter 6 describes the implementation details of the porting process of μC/OS-II on the MPC5675K MCU.

Chapter 7 gives the summary of the test cases that have been employed in the testing of various aspects of the project.

Chapter 8 gives as a concluding note, some of the advantages of this setup and also future work that would be done in continuation to this project.

Chapter 2

BACKGROUND

This section of the document provides some definitions and brief description about some of the concepts upon which further work has been done over the course of the project. For a reader familiar with concepts based on μC/OS-II, the configuration of the MPC5675K automotive dual-core MCU, atomic instructions provided by PowerPC architecture, SIMULINK automatic code generation for multi-rate models and its existing buffering techniques, this section can be skipped.

2.1 Multicore Microcontroller Units

A processor core is the computing unit within the integrated circuit, also known as the central processing unit, responsible for fetching and executing instructions to perform a specific function. These instructions are a part of the instruction set architecture of the processor which allows the programmer or compiler designers to interact with the processor to implement a functionality to be performed by the processor.

Before any further discussion is undertaken it is important to make a critical distinction between microprocessors and microcontrollers. A microcontroller or a Microcontroller Unit (MCU) is a complete computing system which consists of a processor, memory and other peripherals. Microcontrollers are used for embedded systems applications. Since embedded system typically have a defined fixed relationship between the outputs to given inputs to the system and they do not serve a general purpose computing role the need for extensive resources is eliminated. Owing to these factors the micro-controllers often operate in a much lower frequency domain, of the order of a couple of hundred megahertz (like 0-180 MHZ for the e200z7d processors on the MPC5675K). Micropro-

cessors on the other hand consists of only the CPU or the core, and the other components such as RAM, ROM have to be incorporated by the designers externally as a part of SoC to make it a functional unit. Microprocessors usually are used in general purpose computing domain which demands higher RAM, ROM and I/O resources. The clock speeds seen on the microprocessors often run in the gigahertz range. The choice of using microcontrollers in embedded systems is driven primarily by cost considerations and low-power constraints. However these days with the increase in complexity of embedded systems, microprocessors SoCs are often being used in embedded systems. This is particularly seen in the mobile computing domain.

The next distinction to be considered is that of a multi-processors and multi-core processors. Though often used interchangeably, a multi-core processors is a setup where two or more cores are integrated onto a single integrated circuit die instead of having two or more independent processors on the motherboard interfaced using multiple sockets. There are several advantages of a multi-core setup, which has made this architecture design the de-facto standard within the industry. The proximity of the cores greatly decreases the cache coherency latency allowing them to run at a much higher rate. Also the die area required on the printed circuit board is also greatly reduced as opposed to that taken up by a multi-chip design. The proximity of the cores also ensures that the power required to drive the off-chip signals are reduced, leading to greater to power saving [5].

2.1.1 True Parallelism with Multicore

In a dual-core processor or MCU, like the one present within the MPC5675K board used in the project, there are two independent cores present. With this true parallelism can be achieved rather than just an illusion of parallelism which is provided by multi-threaded programs on a single core system. To explain this further consider single core setup running 2 tasks of dual priority. In a single core system the execution of these tasks

would be interleaved, wherein a context switch between the tasks could be triggered every fixed quanta of time. Thus it is easy to conclude that each task gets at the most 50 per-cent of the processor time. Here we can for the time being ignore the context switch overhead which would also eat up on the CPU utilization. Now on scaling up the number of task to say 100, each task gets only about 1per-cent of the CPU time. On a fast enough processor it would still seem as though tasks are executing parallel however there would be clear reduction in the responsiveness of the system. On the other hand going back to the 2 task setup but this time on a dual core system. It is possible to schedule each of the tasks on each of the cores wherein almost 100 % of each of the cores utilization is towards the execution of each of the tasks. Here parallelism in its true sense is seen. With the help of a scheduling policy, such as EDF scheduling, it is possible to run 100 tasks on the 2 cores, where a simplistic division of tasks would be to allocate 50 tasks to each of the cores. In such a scenario, each task gets 2 % of the CPU time as compared to 1 %, thereby increasing the responsiveness of the system by almost a factor of 2.

However, multi-core systems bring about the need of synchronization and other such design considerations owing to true simultaneity. This can make the design of application more complicated and also such synchronization requirements to protect shared resources often prove to be computationally expensive.

2.2 PowerPC and E200Z7 Processor

A collaboration between Apple, IBM and Motorola on a common RISC architecture to create a new architecture aimed at the high performance, low cost computers led to the creation of the PowerPC architecture [11]. It was based on IBM's existing Power architecture, which underwent a number of significant changes to improve performance such as increase in clock rates, a simplified instruction set, and a higher degree of superscalar exe-

cution, 64-bit support and multiprocessor support. This improved form of the POWER architecture was branded as PowerPC (POWER Performance Computing) [11].

There are a number of distinguishing factors between PowerPC and other popular architectures because of which it has become one of the most pervasive architecture in the embedded systems domain such as avionics, automobile, telecommunication etc. Before we discuss the key features of PowerPC and in specific the e200 series of micro-processors it is important to loosely define the terms architecture and micro-architecture. An architecture or Instruction Set Architecture (ISA) is a specification of the functionality that a microprocessor has to provide. From a purely architecture point of view the implementation of these functionalities in hardware hold very little importance to the programmer or the compiler designer. However with varying demands the hardware implementation of the micro-architectures would also vary. These hardware variations such as the depth of the pipeline, superscalar design, branch prediction are a part of the micro-architecture.

Some of the key distinguishing features that makes PowerPC still a favorite within the industry is the broad range of microprocessor families it has to offer and the use of a common instruction set architecture (Power ISA) which covers this gamut of microprocessor families [4].

PowerPC architecture also allows great flexibility in terms of the number of I/O peripherals that can be interfaced, this makes these implementations highly scalable. PowerPC also does not rigidly define the implementation of the standard peripherals, such as the debug controller or the interrupt controller units, which provides the vendor a great deal of flexibility in their implementation. Compared to this the ARM architecture imposes a strong integration in the design of such units. This would no doubt improve performance owing to the strongly coupled design but at the expense of versatility and diversity in hardware solutions.

Finally and probably one of key advantages are that PowerPC based microprocessors and MCUs owing to their simplicity and compact design make it more reliable. This allows them to operate under a greater range of temperatures. For example the operating temperatures for the e200 series microprocessor within the MPC5675K board ranges from -40 to 120 degrees Celsius making it ideal for automotive and high temperature industrial applications [16].

The e200 family of processor and more specifically the e200z7 variant of this processor used within the MPC5675K was designed to be simpler compared to the other higher performance PowerPC processors and suited for a real-time environments [12]. An example of this simple design is the in-order execution and retirement of instructions. The core proves to be highly cost effective as it is highly customizable.

Some of the features worth mentioning as they are relevant in this project. Each core is a dual issue 32-bit processor with an independent instruction and data buses. The 64-bit path to cache supports fetching of two 32-bit instruction per clock. It also has dedicated program counter incrementers supporting pre-fetching of instructions. The load and store units support both unit and multiple word with a latency of 3 cycles for a load latency. Each core individually supports interrupts and exception handling with extensive vectored interrupt and nested interrupt capability [17, p. 136].

2.2.1 Atomic Instructions

From this projects point of view the need for atomic instruction is vital for the synchronization of the two cores when running in AMP mode. Based on these basic instructions a shared memory access (both read and write) can be made atomic. With this it is possible to implement busy-waiting mutual exclusion routines. With these mutual exclusion routines across cores, more sophisticated synchronization solutions between the two

kernels such as the blocking counting semaphore and shared memory blocking queues, can be implemented.

Spin-lock implementations of locks across cores described in the upcoming section relies on shared memory. A pre-determined location in shared memory is used to hold a flag value indicating that the critical section has been entered or is free for access to the requesting process.

lwarx (Load Word and Reserved Instruction) and stwcx (Store Word Conditional Indexed) instructions work in conjunction. lwarx instruction loads the word held at the effective address (shared memory) location into the required register and places a reservation on this memory location [14, p. 80]. During the execution of stwcx instruction, it is checked if the reservation held by lwarx on the memory location has not been corrupted by another access [14, p. 137]. If it has not been corrupted, then the stwcx instruction stores the value held in a register into the effective address pointing to the shared memory, sets bit 0-2 of the cr0 (conditional register 0) to 0b001 and clears the reservation set by the lwarx instruction. However, if the reservation was corrupted by another access then the store to the effective address does not occur and the sets bit 0-2 of the cr0 to 0b000. Based on the value set into this conditional register it can be checked if the flag value set in the shared memory location was atomically accessed or not.

The isync instruction is also essential along with the use of the above mentioned instructions. This instruction prevents the execution of the instructions following the isync instruction till all the instruction issued prior to the isync instruction have completed execution and have taken effect. Following the isync instruction the instruction queue is flushed and the instructions are fetched again [13].

Atomic Compare and Swap:

```
1  int Atomic_CAS(volatile int * p, int oldval, int newval)
2  {
3   __asm__ __volatile__ (
4           "0: lwarx %0, 0, %1   \n\t"
5           "   xor. %0, %3, %0   \n\t"
6           "   bne 1f            \n\t"
7           "   stwcx. %2, 0, %1  \n\t"
8           "   bne- 0b           \n\t"
9           "   isync             \n\t"
10          "1:                       "
11          : "=&r"(fail)
12            : "r"(p), "r"(newval), "r"(oldval)
13              : "cr0");
14   return fail;
15 }
```

Figure 2.1: Atomic Compare and Swap Function

Atomic Compare and Swap (CAS) function with the help of atomic instructions have been used widely over the years and for the purpose of this project the following code snippet has been taken from [46]. Based on this function, a spin-lock implementation has been incorporated into the project.

Here in this code-snippet first at (Fig 2.1, 4), the contents of the memory address calculated by adding address offset 0 to %1 (parameter 1 i.e. *p) into %0 (here 0% represents "fail"). Apart from doing this, a reservation is also done for the memory location now currently held by 0% (0 + %1).

Next, the contents of the 0% are compared to "ldval" (Fig 2.1, 5). If the contents are equal, then the code continues to run without any branch. But if the values are not equal then it branches forward to the label "1:" at line (Fig 2.1, 10) and continues to return back from the function with the fail value set as 1.

At (Fig 2.1, 7), it then checks that the reservation made for the memory location still holds or not. If it is successful then no other access to the memory have been done and continues to write the newval back to the memory address (0 + %1) and releases the

reservation and finally returns back from the function with "fail" set as `0`. However if the reservation fails then it branches back to the label "`0:`" and retries the entire operation.

As explained above the isync instruction at (Fig 2.1, 9) ensure that all instruction such as the memory reservation, the checks performed and the release of the reservation is completed before the function return back.

2.3 Freescale MPC5675K

The MPC5675K microcontroller is ideal for application that adhere to the Safety Integrity Level 3 with the help of providing a great level of on-chip redundancy [17, p. 33]. Some of these critical systems include redundant CPU cores, Interrupt Controllers, DMA controller, crossbar bus systems, memory protection unit, peripheral bus bridge, system timers and the watchdog timers. Apart from this peripherals such as FLEXCAN and LIN-CAN are also redundant. This redundancy can greatly reduce the software complexity, especially in an AMP setup, by statically allocating peripheral usage to each kernel.

2.3.1 Lock Step Mode and Decoupled Parallel Mode

The MPC5675K MCU can be booted up by in two modes of operations, namely Lock Step Mode (LSM) and Decoupled Parallel Mode (DPM). This selection needs to be done statically at system startup.

In the LSM mode of operation is well suited for applications requiring high levels of fault tolerance through redundancy. When the MCU runs in LSM, both cores run synchronously also known as in lock-step and execute the same commands. Here each pair of the memory mapped peripherals share the same address. As mentioned before this MCU offers a great deal of redundancy, and in the LSM mode each redundant sub-system executes the same operations as the other. Following the execution of the same

commands, a set of dedicated Redundancy Control Checker Units (RCCU) detect failures by making sure the outputs to the SRAM, flash or external buses are consistent.

Decoupled Parallel Mode (DPM) of operation is meant for additional performance. In this mode, each CPU core and each connected channel run independently [40]. While in this mode, the RCCU is disabled and all of the peripherals are mapped to different address allowing individual control. Also, SRAM is relocated and split into half, and the single 512KB SRAM array is also split into two 256KB arrays. Along the same lines, the redundant Interrupt Controllers are mapped to different memory location, where each is dedicated to one of the cores. Another peripheral of significant use within this project, namely the Hardware Semaphore Module (SEMA4), also becomes available in DPM.

2.3.2 Hardware Semaphore Peripheral

As discussed earlier, the need for atomic instructions is crucial for providing any kind of mutual exclusion and synchronization between the cores and the use of the peripherals. Since the MPC5675K MCU has a dual core setup and also has the DPM more of operation, for the sake of making application development easier and providing atomicity a dedicated SEMA4 peripheral has been added to provide such atomicity. With such an implementation, the need for an architecture specific read-modify-write instructions is eliminated and the solution becomes architecture-neutral. [17, p. 1601] Each of the 16 gates provided within the SEMA4 unit are capable of holding 3 states representative of unlocked (0), locked by Core_A (0b01) and locked by Core_B (0b10). To illustrate the states of the gates, say Core_A (2.2) sets the state of a Gate_n as locked, then only when Core_A writes zero or when either cores initiate and complete the entire reset sequence can this gate be unlocked [17, p. 1602].

When the gate is locked by a core, and then attempts to lock or unlock the gate by the other care take no effect as the buses are monitored to track source of access.

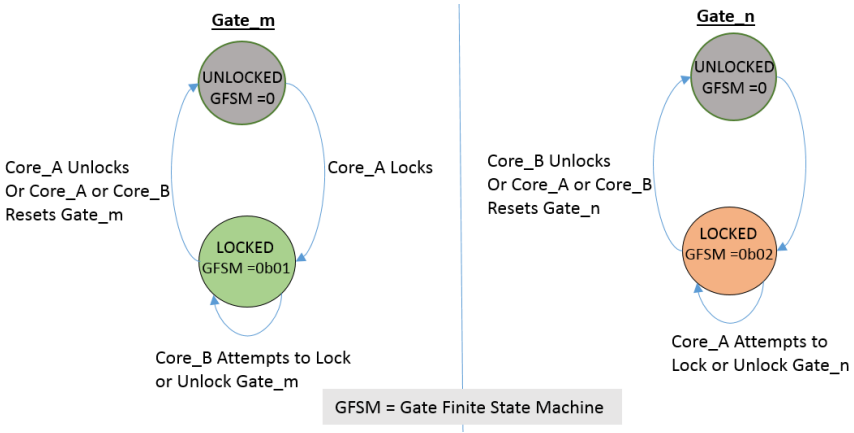


Figure 2.2: Hardware Semaphore Gate Finite State Machine

2.4 Real-Time Operating Systems and μC/OS-II

This section talks about some general real-time concepts and provides various features available on a typical RTOS and specific details of some of these concepts with respect to μC/OS-II.

2.4.1 Real-Time Kernels and RTOS

A real time kernel is software that manages the time and resources of a microprocessor, micro-controller or Digital Signal Processor [26, p. 17]. The management of resources can also be defined in terms of the management of the application tasks, and by adhering to some degree of the temporal reliability. The way to achieve this kind of temporal reliability is to ensure deterministic execution of the RTOS code and by minimizing the jitters or fluctuations in the execution of these tasks. Here a task is an application typically implemented as an infinite loop, which assumes that the CPU is completely available for its execution and is designed to achieve a per-determined functionality. Here such tasks and task level applications comprise of the background part of the real-time systems. On the other hand Interrupt Service Routines make up the interrupt level part of the real-time system also known as the foreground system.

Another important characteristic of real-time systems is the deterministic functional behavior. This essentially means, the system should ensure that given the same inputs the response of the system should always be the same unless the system is designed to behave differently. It is important to note that different behavior owing to system design at different instances of time is still deterministic behavior. To make this clearer let us take an example of the Airbags deployment system in an automobile. In the detection of a crash by the sensors present on the chassis, these sensor values are provided to the system eventually leading to the deployment of the airbags. However at another instances of time such identical sensor inputs are again given to the system but the system does not deploy the airbags. This would make the system unpredictable and certainly make this system unsuited for hard real time systems.

Based on the degrees to which an RTOS can provide guarantees in their real-time characteristics categorizes them into the 3 categories. These categories are Hard, Soft and Firm Real-time systems.

A "hard" real time system shows the least degree of tolerance to failures and any failure to conform to the requirements leads to catastrophic system failure [27, p. 6]. "Soft" real time systems are ones where on missing some temporal constraints the system does not fail completely, but continues to run in a compromised or degraded state with respect to the ideal case where no such deadline misses were encountered [27, p. 6].

The middle grounds between the hard and soft real time system are classified as "Firm" real time systems. This is where the distinctions between the three tends to become blurry. The definition of a firm real time system would be a system where few missed deadlines can be tolerated by the system and would not lead to complete catastrophic failure. However missing more than a few deadlines can in-fact lead to a catastrophic failure. Here the problem arises in defining the value of few deadlines. It can be assumed to be a parameter defined by the nature of the application under consideration.

Real-time kernels is the software responsible for managing the tasks within a real-time system. The presence of more than one task, which typically is the case, makes this a multi-tasking environment. With the help of multi-tasking parallelism can be achieved making the system more responsive and maximizes CPU utilization. It also makes the system design modular instead of being monolithic making it easier for application programming.

Apart from managing the tasks and their scheduling it also responsible for managing inter-task communication and management of system resources. The services provided by the kernel come at the cost of overhead in execution time but are an indispensable part of any RTOS. This kernel, such as the one in Micrium's μC/OS-II, uses between 2% to 4% of the CPU time [26, p. 18].

2.4.2 Scheduler and Rate Monotonic Scheduling

The scheduler or also known as the Dispatcher is the component of the RTOS responsible for deciding which task, of all the available tasks, will run next. Like most real-time kernels even the μC/OS-II kernel is preemptive and priority based. Here each task is assigned a priority based solely on the criticality of the task or could be decided by other schemes such as Rate Monotonic Scheduling.

Rate Monotonic Scheduling (RMS) has been adopted within the scope of this project, more specifically in the work involving code generation for μC/OS-II using SIMULINK Automatic Code Generation. There are certain assumptions that need to be made with RMS which are:

- All tasks within the system have to be periodic.
- Tasks should synchronize with one another and data transfers should only occur with lock-free, semantics preserving techniques.

- The kernel should be preemptive and allocate the CPU for execution to the highest priority task available within the ready list.

With RMS scheduling as long as the CPU utilization is bounded by 0.693, for a task set containing numerous tasks, schedulablity can be achieved. Other non-critical tasks, presumably with lower priority, can also be executed to make the CPU utilization closer to 100%.

2.4.3 Lifetime of a μC/OS-II Task

Tasks as define earlier serve the purpose of performing some sort of functionality such as monitoring inputs, updating outputs, reading sensor data, react to external events, react in a time-define manner, interface with peripherals and communicate with other systems

For a Task to be set up in in μC/OS-II it requires a Task Control Block (TCB), a stack and a priority should be assigned [25]. Apart from this there other parameter that are required to be defined when calling the OSTaskCreate() function [25]. Once the task is created successfully it is placed onto the ready-list, which would be described in the upcoming sections.

Task stacks in μC/OS-II are defined in a static manner and it is preferred not to allocate stack memory to the task using C compilers malloc. The main reason for this being the problem of fragmentation of heap memory that arises with the use of malloc() [25].

Tasks in μC/OS-II can be in any one of the 5 following states: Dormant, Ready, Running, Waiting for Event, Interrupted [25].

Dormant state corresponds to the state when the task has been created and resides in memory but has not been made available to the kernel yet. Dormant state also applies to task which have been deleted. The task still remains in memory but it is just that the scheduler is unaware of its existence any more.

Ready state corresponds to the state when it is ready to execute however owing to its priority being less it cannot preempt the low priority task.

Running State is held by only one task within a kernel and this task has control over the CPU. At this moment no other higher priority task would not be in the ready state giving the task the opportunity to run.

Waiting state or also called the blocked state is when the task is waiting for an Event to occur after which its state would be changed to ready. Events could be based on inter-task synchronization and communication services provided by the kernel, timer expiry etc. More about events would be discussed in the upcoming sections.

Finally the Interrupted state corresponds to when a task is interrupted and the CPU is currently processing the interrupt.

2.4.4 Priorities and Ready List

Each task within the μC/OS-II kernel is allocated a unique priority [25]. Ranging from 0 to OS_LOWEST_PRIO value set within the OS_CFG.H. However the upper limit to priorities is 64. In this setup priority 0 is equivalent to the highest priority and 64 represents the lowest possible priority that can be set. An important point to note here is that, since only task can exist at each priority level, the priority values in themselves can be used as unique identifiers to the task.

In an effort to reduce the amount of RAM used to maintain the Ready-list a table like data structures is maintained to preserve the state of the tasks. Also to increase the speed of access of these data structure every time a scheduling point occurs an additional data structure is maintained called the OSRdyGrp.

Each row within the ready table represents groups of tasks. To place a task within the ready list first the group of the task is is set. This is done using the following:

$OSRdyGrp \mid= OSMapTbl[prio >> 3];$

This indicated that a task within the corresponding group is in the ready state. Next it is required that the specific bit within the OSRdyTbl is set, thus specifying the exact task that is ready. This is done using the following:

OSRdyTbl[prio $>>$ 3] $|=$ OSMapTbl[prio & 0x07];

2.4.5 Time Management

Like for any Operating system, even in the case of a RTOS, the need for periodic timer interrupts is essential to provide services such as time delays and timer timeouts [25]. Especially in this implementation where we follow a RMS scheme in the SIMULINK related aspects of the project. The low level setup of the decrementer timer provided by the e200z7 processor would be discussed in the porting section, however once this setup is complete periodic timer interrupt is triggered by the hardware. But through the operating system we have greater control over the resolution of this timer. The resolution can be set to 10 to 1000 timer per second as typically seen in most RTOS. This value can be set by assigning the appropriate value to the OS_CFG_TICK_RATE within os_cfg_app.h. It is not always advisable to keep the timer resolution to 1000 timer per second as it can lead to considerable overheads in the handling of these recurring interrupts.

Apart from the usual services such as delay OSTimeDly we need the periodic timer service for the implementation of a RMS scheme . Once these timers are enabled (by setting OS_CFG_TMR_EN to 1) and the timers are created (using OSTmrCreate()), by providing the required periodicity and the call back function and initial offset the timers can be started (using OSTmrStart()). Once such a timer is created the operating parameters cannot be changed on the fly and requires that the timer be deleted and a new timer be created [26, p. 201].

The timers are maintained with the help of an internal timer task- OS_TmrTask() which is spawned only when the timer are enabled [25]. The source for this timer task is

also the same interrupts used for he OS timer ticks. As shown in the figure below, every time a timer interrupt is fired, the Tick ISR is called which in turn signals the release of the Timer Task with the help of a dedicated semaphore. This timer task is responsible for updating the timer values, which were created, and to invoke the required call back functions of the timers have expired.

Here the timer call back functions are run within the context of the timer task, and if there are multiple timers then it is essential that sufficient stack space is allocated.

2.4.6 Mutual Exclusion and Synchronization

There are a number a number of way through which mutual exlcusion (critical section access) in μC/OS-II can be achieved between the tasks [25, p. 134].

- **Disabling Interrupts:** To access critical sections shared between tasks interrupts can be disabled (using OS_ENTER_CRITICAL) before entering the critical section and then enabled again (using OS_EXIT_CRITICAL) after exiting the critical section. By disabling interrupts since the scheduler can no longer be invoked unless done so explicitly. This ensure that the critical section is protected from other tasks and interrupt service routines. However if the length of the critical sections is large this can severely downgrade the responsiveness of the system as it would not be able to service any interrupts during this period. Such an approach is suited for very small critical sections of code.

- **Scheduler locking:** A more moderate form of protecting critical sections is with the use of Scheduler locking (OSSchedLock()) before entering and Scheduler unlock after exiting (OSSchedUnLock()). By doing so task rescheduling is no longer possible while in the critical section giving complete CPU control to the currently

running task. However interrupts are still enabled and would be serviced if triggered while in the critical section. This helps improve the responsiveness of the system.

- **Semaphores, message queues and mailboxes:** By protecting critical sections with semaphores it is possible that tasks can wait in the blocked or waiting state till the task which is currently holding the lock decides to leave it. Here it is possible for even ISR to be able to release locks. With the use of semaphores not only critical sections be protected but also complex synchronization between the tasks can be achieved. More details regarding Semaphores and Message Queues would be given in the coming sections as their understanding is important to better understand the work done in the project.

The concept of semaphores in the context of computing was invented by Dijkstra in 1965. Semaphores can be thought of conceptually as keys with which a resource access can be restricted. A semaphore could be configured to work as a binary or counting semaphores. The counter within the binary semaphores can either hold `0` or `1`. In the counting semaphore with 32-bit counters, as is implemented in the μC/OS-II setup of the project [25], the counter can hold values up to 65535. Every time a task intends to acquire the semaphore it decrements the counter and continues on with its normal execution. However if the counter reaches a value `0`, the task can no longer continue and goes into waiting state.

When the task releases a semaphore it increments the counter. If there are tasks currently waiting on the semaphore then the highest priority task of all the waiting tasks are put into the ready-list with the help of Event Control Block (ECB). This being a preemptive kernel, if this recently released task is of a higher priority than the task currently running an immediate context switch occurs.

The signals that allow tasks and ISRs to synchronize with each other are called events within the context of μC/OS-II. Task can wait for other tasks or ISRs to send an event signal with the help of kernel data structures called as Event Control Blocks (ECB) [25, p. 134]. It is important to note that only tasks can be in the waiting state and ISRs are not allowed to wait on the ECB.

μC/OS-II maintains the task dependent on an event and other crucial data structures such as semaphore counters, pointer arrays for message queues etc with the help of an ECB. Each semaphore, mailbox and queue is assigned an ECB with which inter-task synchronization can be made possible.

Using the OSEventTbl[] and OSEventGrp data structures, which are very similar to the OSRdyGrp and OSRdyTbl of the ready list, it maintains the list of waiting tasks. The same operations as mentioned earlier to place or remove task from the ready list can be used to place or remove tasks from the event wait list.

OSEventTaskWait() function is called when it is needed to remove a task from the ready list and instead be placed on the wait list while it waits for an event to occur. When the event, such as the release of a semaphore, occurs task can be removed from the wait list and put on the ready list by calling OSEventTaskWait().

Semaphores in μC/OS-II can be access with the help of 5 services namely OSSemCreate(), OSSemPend(), OSSemPost(), OSSemAccept and OSSemQuery.

2.4.7 Message Queues

To be able to send one or more messages between tasks μC/OS-II provides the messaging queue service [25]. The use of global variables to exchange data between tasks and ISRs has certain restrictions. It is possible for tasks to share data between each other by protecting global variables with semaphores. However for a task and an ISR to exchange

data, semaphores cannot be used to protect the global variables since ISR cannot block on a semaphore. The way to overcome this problem is with the help of message queues.

A message in the context of μC/OS-II consists of a pointer to data and as is the case with semaphores there is a wait list associated with each message queue. The message queue is essentially an array of void type capable of storing address of the memory blocks which have the required data within them. This avoids the overhead of copying the data and allows pointers to be passed around making the implementation faster.

A task that expects to read a message from the queue but if the message is not available the task will be suspended and is placed on the wait list. This wait period can also be specified in the form of a timeout period. If the message is not received within this timeout period the waiting task is put into the ready-list and returns back with an error code indicating no message. Here again ECB would be required for the wait list.

When the message is delivered into the queue by the producer task then two things can occur. If there are no tasks waiting for the message then this message gets buffered within the queue and corresponding data structures to maintain the circular queue are updated. However, if there is a task on the wait list, then this task is put on the ready-list and the message is directly given to the task by passing the pointer to the tasks TCB.

Apart from the ECB we also need another kernel data structure called the Queue Control Block (QCB) [25]. This holds the required data to manage the queue.

.OSQStart — Holds the pointer to the start of the message queue.

.OSQEnd — Holds the pointer to one position beyond the end of the message queue, allowing circular buffer semantics to be followed.

.OSQIn — Holds the pointer to the location within the queue where the next message will be inserted. Since this is a circular queue once OSQIn value equals the OSQEnd positions is it reset to OSQStart value.

.OSQOut — Holds the pointer to the location within the queue where the next message can be retrieved from. Since this is a circular queue once OSQOut value equals the OSQEnd positions is it reset to OSQStart value.

.OSQSize — This value is set at the time of creation of the message queues and indicated the maximum number of message entries possible within the queue. μC/OS-II allows up to 65535 entries within each queue.

.OSQEntries — Holds the number of message currently in the message queue.

The functions that would be required to be understood for the scope of this project are the following:

OSQCreate() — This function creates the queue. In the process of creation of a queue an ECB and a Q_OS is acquired from the kernel. It then links the ECB and the OS_Q data structures by passing the pointer of the QCB into ECBs OSEventPtr field. Then it initializes the QCB data structures such as the size of the queue to OSQSize, start and end positions and sets the in and out values to 0 indicating that the queue is empty.

OSQPost() — This function is used to deposit a messgae to a waiting task or the queue. When a message is generated by the producer this function first checks the ECB to see if there are any tasks currently waiting on the message. If there are then it puts the message onto the task's TCB and removes the task from the wait list and puts it onto the ready list.

If there are no task waiting in the wait list. It then places the data ontot the circular buffer, makes changes to the OSQIn and OSQEntries values.

OSQPend() — This function is used to wait for a message. When the message becomes available and no task is on the wait list it decrements the value of OSQEntries and moves the OSQOut value. If the message is not available then the task puts itself on the wait list and calls the OSSched() function to schedule the next highest priority task to run. After the timeout has occurred the task wakes up to see if there is a message on its TCB.

If there is no message it returns with an error message to be appropriately handled by the application.

OSQAccept() — This function is similar to OSQPend() except that if there is no message available it does not put the task to sleep and return back with a null pointer indicating that no message was available. This makes it ideal to be used from within an ISR since blocking is not permitted.

2.4.8 Memory Management

Dynamic memory allocation is generally done with the use of ANSI C compilers malloc() and free() functions. However these schemes have a problem of fragmentation. In a case where a large contiguous memory is demanded by the task and because of fragmentation no such contiguous memory is not available then the application execution can break down. Another reason for having a special dynamic memory allocation scheme apart from the use of malloc() and free() is the non-determinism in the execution time of these functions [26, p. 299]. The memory management scheme followed in μC/OS-II for allocation and de-allocation is done in constant time and is deterministic.

μC/OS-II manages memory in the form of partitions. Here partition is the contiguous block of memory that is global to the tasks. This block of memory is known as a partition in the context of μC/OS-II. This partition is initialized and managed with the help of the various memory management services about which further discussion would be taken up. However like in the case of semaphores or message queues, there is kernel data structure which is needed to manage these partitions. This data structure is called the Memory Control Block (MCB). Now, based on the size of each of the memory block required by the application, this partition is further divided into memory blocks. Say, if we allocate a 1024Bytes of memory to a partition and each block is of size 32bits (4Bytes) then there would be 256 (1024/4) such memory blocks available to the application tasks. To better

understand the memory management techniques employed within μC/OS-II the fields of the MCB need to be analyzed.

OSMemAddr — Holds the beginning address of the memory partition. During the OSMemCreate() function call this entire partition is initialized by stetting the value to 0.

OSMemFreeList — Holds the pointer of the next available free memory block from the linked list of free memory blocks.

OSMemBlkSize — Is set only at time when OSMemeCreate() is called and holds the size of each memory block.

OSMemNBlks — This value is also set during the call to OSMemCreate() and holds the number of memory blocks within the partition.

OSMemNFree — Holds the number of free memory blocks available in the partition.

Now that the function of each of the data members has been discussed, a complete picture of how memory management is done in μC/OS-II can be gathered with a discussion on the memory management services provided.

OSMemeCreate() — It is assumed that the memory partitions are created before this function is called. As a part of the initialization done during this function call, the memory blocks are cleared and set with zero. More importantly a linked list of the memory blocks is created. This linked list represents the free blocks available within the partition.

As a part of the function call the starting address of the partition is passed within the function call along with the number of memory blocks required and size of each block. On the successful initialization, of this partition and the creation of the memory blocks each of these values are assigned to OSMemAddr, OSMemNBlks and OSMemBlkSize member of the MCB respectively.

OSMemGet() — This function call is equivalent of the malloc() system call. Based on the partition MCB passed into the function call as a parameter, a memory block from the partition is returned. If a memory block is available within the partition then the

OSMemNFree value is decremented and a pointer to the memory block is returned. If no memory block is available then a well-known error value is returned indicating that there are no longer any free memory block available.

OSMemPut() — This function call is equivalent of the free() system call. As a part of the parameters the pointer to the partition's MCB as well as the memory blocks address is passed. As a part of this call it also checked if the memory partition is already full. If it is not full then OSMemNFree value is incremented and success value is returned back. However, if the partition is already full then this indicates an error condition and return the error OS_MEM_FULL. An important point to note here is that it is possible that, owing to an error in programming, a block belonging to another memory partition can be returned through OSMemPut to another partition. Such a situation should be avoided.

2.5 Automatic Code Generation and μC/OS-II

2.5.1 Model Based Design and Rapid Prototyping

Model based design (MBD) is increasingly being adopted as a methodology for designing complex embedded systems since it allows advanced verification and validation [1]. MBD can be defined in simple terms as the use of diagrams, mathematical or other visual methods to represent the components of a complex control system [41]..

Another key approach that is used in the development and validation, especially for embedded control systems, is the use of rapid prototyping. In an multi-domain environment where all the of required sub-systems and components might not be available during the early stages of the project, the use of prototyping allows the design teams to work with customers and continuously refine the designs and validate them as well.

In a typical industry setup, say for the development of a new controller module, during the early stages of the development process an initial prototype is often required. De-

signer usually incorporate the functional algorithms and early iterations of the software on an FPGA or a generic microprocessor with similar specifications to demonstrate the feasibility and performance of the controller [20]. This process repeats over many iteration till the software and hardware is finally ready for deployment in the production environment.

Within each iteration of this rapid prototyping approach an idealized form of the model is created, containing the most refined versions of the algorithm and the functional logic. This model can then be tested and validated in either a simulation environment also called as model-in-loop testing or with the help of automatic code generation tools the code can be deployed on the controller hardware for validation. This hardware could be the production hardware, if available, or an FPGA. With this approach, during every iteration the model parameters can be fine-tuned, optimized and tested for functionality, compliance to standards and robustness.

SIMULINK can be used for the purpose of automatic code generation if the requirements include the fixed point and timing behaviors. The following sections would discuss more in details regarding this code generation process from models.

2.5.2 Code Generation Architecture and TLC Process

The Embedded Coder is a product offered by Mathworks for the purpose of automatic code generation targeting the embedded domain specifically. Another product that offers almost the same capabilities as the embedded coder is the SIMULINK coder. SIMLUNK coder generates C and C++ code from the designs created in the form of models, Stateflow charts and MATLAB functions. This generated code is more general purpose in nature and can be used in real-time and non-realtime systems. Embedded Coder provides additional configuration and optimization options over SIMULINK coder providing a

greater degree of control over the generated code. Embedded Coder also offers built in support to some industry software standards, such as AUTOSAR and ASAP2 [33].

The code generation process in SIMULINK is controlled with the help of its Target Language Compiler (TLC) feature [30]. This compiler provides a set of TLC files for generating the ANSI C Code. The code generation and the TLC process can be represented with the help of the following Figure 2.3. This process presented within this figure is an adaptation of the process presented within the Target Language Compiler documentation provided by Mathworks [30].

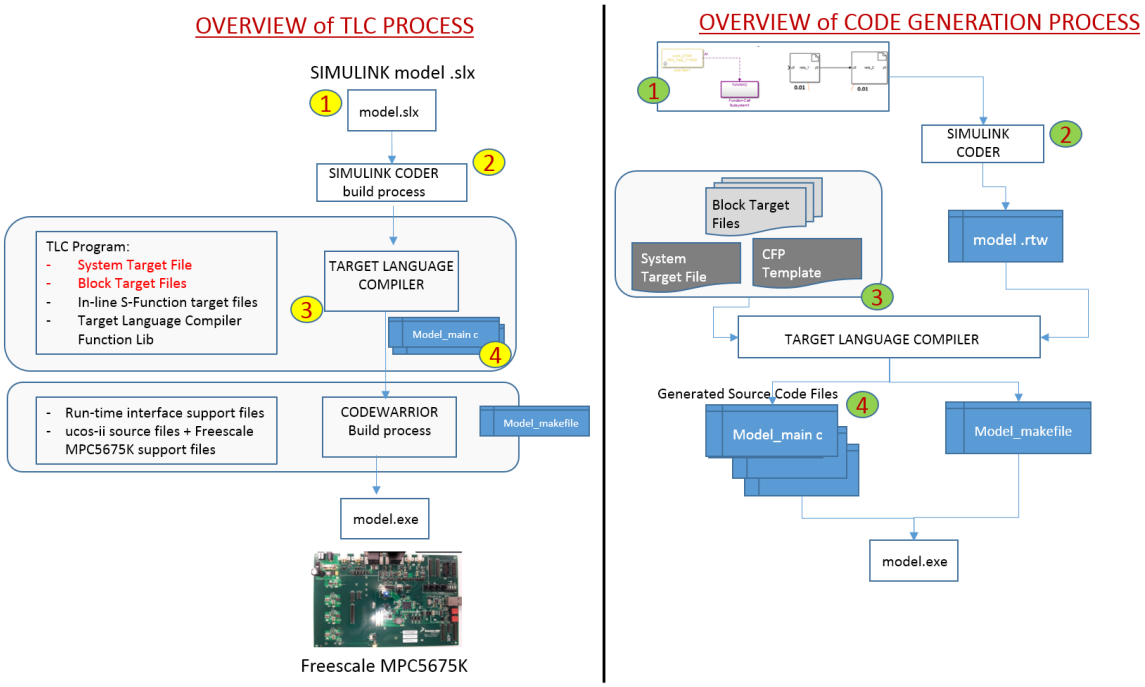


Figure 2.3: Code Generation Architecture and TLC Process

1. The model, which essentially is a block diagram of the system, is first created and verified by the designers. This model contains the semantics of execution of the various components of the system and the communication between these sub-systems.

2. Next in the process, the Target Language Compiler with the help of the TLC files, which constitute the target files and the model-wide files, generates the code. These TLC files can be used to specify the nature of the real-time system, which includes the operating system and the processor, on which it is intended to be executed. During this step of the code generation process, the target language Compiler uses the block target files, specific to each block present within the model, to transform the corresponding entry within the model.rtw into its ANSI C code. Model-wide target files are not specific to any one block but are used for the global customization of the entire code generated. The system target file is the entry point for the code generation process. Through these files the generated code can be fine-tuned to make it best suited for the underlying system on which it is to be executed. Apart from the above mentioned TLC files, there also exists a template makefile which can be modified with required flags to conform to the targets specific requirements.

3. Once the code generation process is complete the source files are obtained along with a custom make file the source can be compiled and deployed onto the target. However, for the sake of this project the makefile was not needed and only the generated code was needed to be integrated into the Codewarrior project.

One of the requirements of the project included the customization of the files generated through automatic code generation to be compatible with the underlying μC/OS-II operating system. Apart from this, additional blocks were also created to aid in the design and code generation process for this project. To achieve this customization of the generated code two kinds of TLC files need to be modified, namely the File Customization Template and the individual block level TLC files. As mentioned above the entry point for the code generation process is called the system target file. The following is brief description these three categories of TLC files.

1. **System Target File:** The system target file provides the entry point to the code generation process [30]. This is done with the invocation of the codegenentry.tlc file from within the system target file. Following this a number of other TLC files are invoked. By modifying these files the overall code generation process can be customized. In the course of this project the overall structure of code generation is set to its default configuration.

   The code generation process can also be configured by assigning appropriate values to the Compiler Configuration Variables. With the help of these configuration variables the code format, target type and language of code generated can be set [30].

   The system target file selected for this project is the embedded real-time target (ert.tlc). The embedded real-time target file allows for the generation of production code, with minimal usage of RAM/ROM. However it does not support continuous time semantics. As opposed to this the generic real-time target is often used for the purpose of rapid-prototyping and also supports continuous time semantics. However the code is not ready for production and does not minimize the RAM and ROM usage [29].

2. **Custom File Processing Template (CFP):** After the selection of the system target file, the next step is to customize the overall structure of the generated code. This is possible with the help of the ERT Custom File Templates also called the CFP template. These files are used to primarily arrange the code that needs to be generated, into buffers. Once these buffers are populated they are called at various desired locations and the code is generated in those sections. These locations can include header sections, global data sections, and specific sections of generated code. The CFP template, which by default is set as the example_file_process.tlc, is set to only generate the main function.

Depending on whether the model is a single rate or a multi-rate model, bareboard_srmain.tlc and bareboard_mrmain.tlc is called respectively. These template files can also be modified to produce custom main function.

Apart from just defining the code generated for the main function additional code customization is possible by enabling the ERTCustomFileTest flag as TRUE within this file. One such customization that is possible is the addition of source files with custom functions within them.

3. **Block Target Files**: SIMULINK provides S-Function (system function) blocks with which custom blocks can be created to extend the existing SIMULINK capability. This Level 2 S-functions is typically a C, C++ or MATLAB code which are then compiled into MEX files. These MEX files can then work in tandem with the SIMULINK engine to provide the desired functionality within the model. For instance a very common example the SIMULINK provided is the timestwo block [31]. This block is responsible for multiplying the data coming in through the input-port with a factor of two and provides these values on the output port. This block, with this custom functionality can then be added within the model to perform this functionality. These s-functions can also take parameters as inputs as well allowing greater control on its behavior. However, the custom blocks that were created over the course of this project had no simulation time functionality but primarily were needed for the sake of code generation.

   To enable these blocks for customized code generation through Embedded Coder, block target files need to be written. Similar to the Custom File Processing templates, these block TLC files also work in the form of buffers, where in the code is populated, and then emitted at the desired location. In such blocks, where the primary purpose is for code generation, the C files are typically used for evaluating

the input parameters to the blocks, determining the sample times of the ports of the block, finally pass the block parameters to the SIMULINK coder during the translation of the model into the model.rtw file. There are a number of other functionality supported by the s-function and further information about this can be obtained within the SIMULINK documentation [31] and [29] . A point to note here is that, it is not only the custom blocks that have the block TLC files, but all blocks present within the SIMULINK's Embedded Coder library also contain the TLC files to support the automatic code generation of the block logic.

2.5.3 Program Execution of Generated Code

The code generated by the Embedded Coder can be run on a bare-board environment, without an underlying RTOS, as a standalone piece of code. The code generation architecture can generate code for single rate and multi-rate models. However, to be able to support a specific RTOS, such as the μC/OS-II, the code generation process has to be customized as needed. SIMULINK Embedded Coder does provide support for VxWorks operating system under the Tornado Environment for Industrial Automation support. This support of VxWorks does provide a good starting point to build the custom code generation framework to support μC/OS-II.

Like in the case for any typical C program, the starting point of the generated code is also the main loop. This main loop is periodically interrupted by the timer ISR, named as rt_onestep. The frequency at which rt_onestep is interrupted defines the base rate of the generated code. Within the rt_onestep ISR the model_step function is called. This model_step function contains all the computational logic associated with one step of the model from which the code was generated.

Depending on whether the model is a single or multi-rate, the structure of the rt_onestep function would vary. In the following figures the pseudo-code of the each of the possible

variations of the rt_onestep functions is provided with line numbers for the convenience of discussion.

Single-rate Operation

First upon entering the model_step function the timer interrupts are disabled till the overrun condition has been checked for. An overrun occurs in a scenario where the rt_onestep function is called, once the timer interrupt is received, before the previous iteration of the model_step function has completed. In such a case the system fails to adhere to the timing constraints of the model. On detecting the overrun or any other error an appropriate error flag is set and return immediately from the function (Fig 2.4, 1).

If an overrun or any other error has not occurred then the timer interrupts are enabled again (Fig 2.4, 2). After doing so, the model_setp function is called which contains the in-lined code of the blocks present within the model (Fig 2.4, 3).

```
rt_OneStep()
{
  Check for interrupt overflow or other error              (1)
  Enable "rt_OneStep" (timer) interrupt                    (2)
  ModelStep-- Time step combines output,logging,update.    (3)
}
```

Figure 2.4: Single Rate Program Execution

Multi-rate Operation

In a multi-rate system, when the rt_onestep ISR is called again the interrupts are disabled till the overrun condition is checked for only base rate and not for any other sub-rates (Fig 2.5, 1). If there are no overruns or error detected the interrupts are then re-enabled again (Fig 2.5, 2).

An important point to note in the implementation of the multi-rate operation is that a pre-emptive multi-tasking scheme is used where the tasks corresponding to each rate are prioritized based on the rate of the task. Faster the rate higher is the priority (Rate Monotonic Scheme). Each of these tasks running at different rates are assigned a task identifier (tid). The base rate, which is the fastest rate and consequently the highest priority task, is given tid = 0. The next fastest sampling task is given tid = 1 and so on. The total number of such tasks are maintained with the NUMST variable.

Now, after the interrupts have been enabled the task corresponding to the base rate, with tid = 0, is run (Fig 2.5, 3). Since this is the highest priority task every time the rt_onestep ISR is called it would be run. Since each of the sub-rates are defined as sub-multiples of the base rate during every call of the rt_onestep function tasks with tid = 0 and one or more of the other tasks are run. This is seen by iterating over all of the sub-rates of the tasks present in the model, except for base rate (Fig 2.5, 4).

For each of the sub-rate tasks event and counter flags are maintained. The counter flags essentially work as dividers to the base rate, and during every iteration of the rt_onestep count up to the required sub-rate. Once the required sub-rate value is reached on the counter, the event flag for the task is set. This event flag indicates the next sub-rate task that needs to execute. If an overrun condition is detected for the sub-rate task then the overrun flag is set. This detection and handling of the overrun is similar to the handling of overrun for the base rate task (Fig 2.5, 5). If no overrun is detected then the model_step corresponding to the tid is called (Fig 2.5, 4 to 5).

VxWorks-Tornado Environment Multi-Rate Operation

The above 2 methods of program execution described are those generated by using the ert.tlc system target file in SIMULINK automatic code generation process. However, SIMULINK also provides support for VxWorks operating system. This requires that

```
rt_OneStep()
{
  Check for interrupt overflow or other error                (1)
  Enable "rt_OneStep" (timer) interrupt                      (2)
  ModelStep-- Time step combines output,logging,update.      (3)
}


rt_OneStep()
{
  Check for base-rate interrupt overflow                     (1)
  Enable "rt_OneStep" interrupt                              (2)

  ModelStep(tid=0)     --base-rate time step.                (3)

  For i=1:NumTasks    -- iterate over sub-rate tasks         (4)
    Check for sub-rate interrupt overflow                    (5)
    If (sub-rate task i is scheduled)                        (6)
      ModelStep(tid=i)    --Sub-rate time step.              (7)
    EndIf
  EndFor
}
```

Figure 2.5: Multi-Rate Program Execution

instead of bare-board target, VxWorks target is selected as the operating system. On doing so the code generated does provide multi-rate multi-tasking support which is very similar to the kinds required within this project. However the monolithic characteristic, where the base rate task controlling the execution of the other sub-rate tasks is maintained here to some degree as well. For the purpose of extending multi-rate support for μC/OS-II the code generation templates as a part of the Tornado environment were drawn upon as the starting point. The program execution order for VxWorks is described below. It is required as a part of the Tornado Automation Environment that a separate task, namely rtwdemo_mrmtos_main, is spawned. After doing so the rest of the initialization is done by this task.

```
rtwdemo_mrmtos_main()
{

  Turn off auxiliary clock interrupts                                          (1)
  Set and check the auxiliary clock rate                                       (2)

  for each sub-rate task (and also the base rate task)                         (3)
        Initialize the task release Semaphore
        spawn a task for each with priority corresponding to Rate Monotonic Scheme
  end for

  Connect the ISR to the auxiliary clock interrupt                             (4)
  Turn on auxiliary clock interrupts                                           (5)

  While(not Error) and (time < final time)                                     (6)
    Background task.
  EndWhile

  Turn off auxiliary clock interrupts                                          (7)
  Delete Tasks
  Delete Task release semaphores
  Disable interrupts (Disable rt_OneStep from executing.)

  Shutdown
}
```

Figure 2.6: Tornado- rtwdemo_mrmtos_main Task

The annotated pseudo-code within Figure 2.6 mentions the initialization process undertaken by the rtwdemo_mrmtos_main task. First, to allow the proper setup of the tasks and the timer ISR the auxiliary clock interrupts are disabled (Fig 2.6, 1). Based on the base sampling rate specified within the model the same sampling rate is also assigned to the auxiliary clock timer. After setting the sampling rate, it is rechecked again to see if the setup was done correctly (Fig 2.6, 2). Then depending on the number of sub-rates required, the VxWorks tasks are spawned. Here the tasks associated with the base rate, i.e. tBaseRate, runs at the highest priority and the other sub-rates tasks are assigned priorities based on the frequencies of sampling rate. Also, semaphores to control the execution of these tasks are created and initialized (Fig 2.6, 3). Next the ISR, associated with the auxiliary clock interrupt is installed. This ISR is only responsible for releasing a semaphore allowing the tBaseRate to run (Fig 2.6, 4). This concludes the basic setup of the task framework and on successful completion the auxiliary timer interrupts can be enabled again (Fig 2.6, 5). Now, this task runs in background mode, only waiting for the occurrence of an error or

program exit (Fig 2.6, 6). Once these conditions are encountered the house-keeping is performed by deleting tasks, deleting semaphores and the disabling of the

```
tBaseRate()
{
  Check for errors                                          (1)

 Wait till semaphore released for tBaseRate                 (2)

  For i=1:NumTasks    -- iterate over sub-rate tasks
    Check for sub-rate interrupt overflow                   (3)
    If (sub-rate task i is scheduled)
      Give semaphore to run Sub-rate task(tid=i)            (4)
    EndIf
  EndFor

  step the model for base rate (tid=0)                      (5)
}
```

Figure 2.7: Tornado tBaseRate- Base Rate Function

In the beginning of every iteration of this task, first error conditions are checked for. If any errors are found the error status is returned (Fig 2.7, 1). Next overrun conditions are checked for the base rate task. If no such condition is encountered then the task blocks itself till the semaphore to run tBaseRate task is not released by the auxiliary clock timer ISR (Fig 2.7, 2). Next, overrun condition for the sub-rate tasks is checked for (Fig 2.7, 3). If no overruns are found then similar to the rt_Onestep function in the multi-rate program execution process, with the help of counters it is determined which of the sub-rate tasks need to run. The sub-rate tasks are then allowed to run by releasing the semaphores that they are blocked on (Fig 2.7, 4). After doing so the function call to run the model step function for the base rate is called (Fig 2.7, 5).

Chapter 3

LITERATURE REVIEW AND RELATED WORKS

This chapter investigates some of the work that is relevant to the body of work undertaken within the scope of this thesis. Certain design patterns mentioned within these works have been adopted as well and this section would highlight some of the reasons for doing so.

3.1 FreeRTOS and Multicore

There is a lot of work that has been done validating the advantages of multicore setups as compared to uni-processor ones. Some of the core ideas behind focusing on multi-core real time operating systems have been discussed in the previous sections. The work presented by Mistry [34] in this report primarily involves the implementation of an open source real-time operating system, namely FreeRTOS, adapted for multi-core support. The design choice in this work was also Asymmetric Multiprocessing Real Time Operating systems. However the main aspect that distinguishes his work from this thesis project is that both instances of the RTOS on a dual core setup shared the same ready-list. This implementation of sharing a ready list makes it an attractive proposal keeping the task execution semantics almost similar to that of uni-processor systems.

However, a key disadvantage observed with shared ready list for multi-core processors is that the access to the ready list needs to be protected across the cores since it is a shared resource. This mutual exclusion is again a busy waiting solution which can eat into the execution time of the task which is about to be context switched in into the processor. With this approach as the number of cores keep increasing in the hardware, more would

be the contention, eventually making the implementation unsuited for further scale up owing to the bottle neck in the shared list access.

However, having a shared ready list makes synchronization of tasks easier to implement. For instance, say that a task which is busy waiting on the vTaskAcquireNamedMutex, as implemented in the work, and unless a timer tick is triggered forcing a context switch it holds the first processor. Then a concurrently running task on the other core which had held the same lock now releases it. Here the wake up of the first task is a fairly straightforward process as the busy waiting loop just exits. However this being a busy waiting solution, it eats of valuable CPU time, at least till timer tick enforcing a context switch occurs. A lower priority task can get starved for CPU time in such a situation.

Here Mistry does suggest that an option of a voluntary yield by the processor could be explored, but this would just complicate the use of such synchronization APIs.

Another interesting feature implemented by Mistry[34] is the option to assign core affinities to certain tasks. This essentially makes the classification of tasks into two categories. Tasks bound to a processor based on the affinity and the other free to run on any instance core or in other words free to run within any instance of the RTOSs. Here this approach allows a great degree of load balancing to be accomplished. But with such an implementation the task with no affinity either should make the data structures purely contained within the stack of the task or have these data structures be placed in shared memory. Now in an event that more than one such tasks with no core affinity require access to these data structures, the need for a lock requiring mutual exclusion across the cores would be required. These across the core mutex locks as described within the work are more expensive than the locks required when mutex protection is required only within the kernel. This flexibility of task execution on either cores also require duplication of functions, that the dependent on, across the kernels.

3.2 Multiprocessors Synchronization protocol

The synchronization protocol, namely Multiprocessor Priority Ceiling Protocol, proposed by Rajkumar [39] is an extension of the well-studied Priority Ceiling Protocol for a multiprocessor system which relies on shared memory between multiple processors. This has been the standard synchronizing protocol for a system consisting of global view of priorities an in a multi-processor system, where the tasks are scheduled using the Partitioned Scheduling Scheme [28, p. 256]. The Multiprocessors Synchronization protocol for real-time Open Systems (MSOS) proposed by Nemati et al [37] is an improved version of the MPCP with improvements in pre-emption overhead times and most importantly assumes that each system is independent of each other. This setup is similar to the AMP-RTOS setup that we have presented within this project.

The MPCP protocols deals with a system wherein the priority scheme followed by the individual systems are consistent with each other. And by incorporating the Priority Ceiling Protocol (PCP) within the synchronization protocol, the blocking time for high priority tasks is reduced.However, since the basic assumption made in this thesis was that the priority schemes on the two independent kernels were not related, a task of priority 1 on Core_A might not be of a higher priority in the global view with respect to a task of priority 2 on Core_B. Owing to this reason the Priority Ceiling aspect of the protocol has been omitted from the work. If required the PCP protocol can also be incorporated within this body of work if the priority schemes can be formalized on the 2 kernels.

The work done by Nemati et al [37] in the MSOS provides an overview of a framework in terms of the required data structures and the basic rules to be followed in the implementation. However, the MSOS protocol focuses on bounding the blocking time of tasks waiting on the global resource. This is achieved with the help of the Priority Ceiling Protocol, that essentially relies on boosting the priority of the tasks higher its

normal priority. This upgrade of priority does ensure that the task holding the lock is not pre-empted by a task not holding requiring lock, however there is an fundamental problem with this scheme. The temporal characteristics of a task not holding the lock, which includes tasks that are not dependent on the lock at all, depends on the length of the critical section of tasks holding the lock currently [6]. This could impact the tasks which are latency sensitive.

This problem can be explained with the help of the following example. Say in a system with 4 tasks of priorities ranging from 1 to 4 (Where priority 1 is higher than 2 and so on). Task with priority 1 and 4 share a lock. Say at an instant of time Task 4 has acquired the lock and has entered the critical section and then when Task 1 is currently blocked on the resource queue. With the PCP protocol the priority of task 4 would be boosted up to 1 ensuring that Task 4, now with priority 1, is not pre-empted by Task 2 and Task 3. This does ensure that priority inversion does not occur but also delays the execution of these tasks, which are not dependent on the lock, to be delayed till the length of the critical section of Task 4. If Task 3 and Task 4 were time sensitive this phenomenon would hurt the temporal characteristics of these tasks.

Keeping in mind the above mentioned points about the implementation of any PCP related protocol, during the course of this project two versions of the global sempahore API have been created. One without the PCP aspect and the other with the PCP feature incorporated within them.

Also a practical issue was addressed with respect to the communication scheme between processors. Since the works provided in the domain of multi-processor synchronization provides only the high level guidelines to implement the shared resource management, certain extensions were required to be incorporated to make the global semaphore setup scalable to beyond one global semaphore.

To explain this scenario, let us assume that on Core_A all incoming external interrupts have been disabled, typically in the case when a task enters a critical section. During this time more than one global locks could be released by the tasks running on Core_A. The solution to this problem was the release of all locks at the same time when core_A does enable interrupts. This was done with the help of additional data structures, details about which would be given in the upcoming sections. This issue could have also been addressed by assigning individual software interrupt lines to each global semaphores, but since the number of such software settable interrupts is limited on the MPC5675K and on most other hardware platforms, it would not be scalable solution.

3.3 Semantics Preserving Multi-Task Implementation

Most high level models assume that the ideal synchronous semantics are preserved[9]. In such ideal semantics each execution of a system component to produce an output from a given input is considered to be instantaneous, i.e. the release and execution is carried out within one unit instant of time. It is expected that even when such multi-rate blocks communicate with each other these semantics are persevered. However such synchronous behaviors do not apply when the real implementation is generated from the model since actual execution times can interfere with such ideal semantics.

SIMULINK and other synchronous languages provide monolithic implementations of multi-rate models as described in earlier sections (Section 2.5.3). Another approach adopted is in the use of independent mult-tasking implementations where each component of the system maintains its dedicated time driven task. Communication between tasks in such a system relies upon simple buffering techniques [36, p. 32] with execution time semantics. This issue with the buffering framework provided by such Reactive Synchronous based tools was the motivation behind the work by Paul et al in [9]. The

issues with data buffering in multitasking frameworks is well explored and documented by SIMULINK as well at [32].

These issues have been addressed by SIMULINK in a very restrictive manner with the help of rate transition blocks. Sofronis et al [43] discuss the ways SIMULINK has come around this problem and provided solutions to allow deterministic behavior of the multi-rate model with the help of Rate Transition blocks. They mention the need for a unit-delay between slower blocks (lower priority blocks in Rate Monotonic priority scheduling) and faster blocks. Another issue that they highlight is the need for a zero-order-hold block be inserted between fast and slower blocks. However with the use of such simple buffering techniques ideal semantics are not always preserved as has been highlighted in the paper by Paul et al [9]. They also go ahead to provide a range of communication protocols between various configurations of reader and writer blocks such as low to high priority, high to low priority etc., which not only perform the buffer maintenance actions during the execution times but also during release times.

The semantics preserving buffering schemes proposed by Paul et al has been one of the focus areas of this project and the buffering schemes mentioned have been incorporated within the multi-tasking framework, based on Rate Monotonic scheduling scheme. The Dynamic Buffering Protocol described in these works rely on the fact that the underlying task framework generated by code generation supports multiple independent time triggered tasks. The setup of such a framework for μC/OS-II within this project makes the DBP an ideal buffering protocol.

3.4 Three-Slot Asynchronous Reader-Writer Mechanism for Multicore

The lock-free Dynamic Buffering Protocol is suited for communication with tasks on a single processor. However in the setup like the one presented in this work, wherein two separate kernels handle the scheduling of tasks independently, DBP would not work

because it is applicable to a single processor setup. DBP is based on the assumption that at any time only one task is executing, i.e., either the reader or the writer. The scheme is strongly dependent on the pattern of release and execution times of the tasks. In the current setup since there is no determinism in the scheduling of the tasks on different cores, as they are being managed by different schedulers on different cores, another scheme more general in nature with respect to release and execution times, needs to be considered. It is important to note that the underlying idea behind this mencanism is the recent value

The lock-free three-slot mechanism provided within the works by Chen et al [10] assumes single read and writer pairs and the presence of an atomic compare and swap operation for the successful implementation of the 3 slot buffer asynchronous communication mechanism. The environment provided within this project provides this atomic compare and swap mechanism for the access of the buffer slots. Also since the PowerPC architecture ensures single atomic read and writes of boundary aligned bytes, word and half-words the sequential consistency model of the memory operations by tasks across the cores.

Since SIMULINK does not particularly have the notion of a true multi-tasking setup let alone multi-core and multi-tasking setup this buffering scheme was also provided as a custom block in this project.

The assumptions of this asynchronous scheme of communication between two concurrently running tasks is discussed in further detail in the design section.

Chapter 4

PROBLEM ANALYSIS AND REQUIREMENTS

4.1 Overview of Objectives

The principal objective of this implementation is to have an AMP ready setup of μC/OS-II on the dual-core Freescale MPC567K MPU. These μC/OS-II kernels would behave in a standalone manner, with an independent code and data section dedicated for each core. Shared resources, such as peripherals, or task synchronization between tasks across the kernels would be handled with the help of inter-core semaphores, referred to as Global Semaphores within this project. Any inter-task data sharing across the kernels can be achieve with the global queues relying upon shared memory management.

The additional services, mentioned above, rely on shared memory accesses. To prevent concurrent accesses this shared memory needs to be protected by mutual exclusion primitives. Also, in this AMP setup some aspects of the initializations are partially carried out by each core. To ensure that the μC/OS-II kernels and the services are initialized correctly -barrier primitives are also required.

This project also aims to support μC/OS-II automatic production code generation with the help of SIMUINK's Embedded Coder. This code generation process would support both a task view and multi-rate view in terms of the models from which code would be generated. To support semantics preserving data passing between the μC/OS-II tasks within a kernel- Dynamic Buffering Protocol has been incorporated as a replacement for the existing generic Rate Transition Blocks provided by SIMULINK. Also, for communication across tasks residing on different kernels, a multi-core variant of the rate transition block has also been included in the code generation framework.

The above mentioned broad requirements have been listed in the following section. The convention followed for identifying each of the requirements is of the form "REQ-x-y": where x represents the parent level numeric unique identifier and y represents the numeric unique identifier for each of the child level requirements.

4.2 Functional Requirements

4.2.1 μC/OS-II System Level Requirements

- REQ-1-1 Setup two independent μC/OS-II kernels running concurrently on each core of the dual-core MPC5675K MPU.
- REQ-1-2 Provide mutual exclusion primitives to make operating system components accessing shared memory safe.
- REQ-1-3 Provide inter-core barrier primitives to allow system setup and OS initialization to be completed by the kernel responsible.
- REQ-1-4 Provide inter-core communication using software settable interrupts.

4.2.2 μC/OS-II Additional Services Requirements

- REQ-2-1 Provide API for blocking counting semaphores between the two kernels
- REQ-2-2Provide API to manage shared memory between kernels.
- REQ-2-3 Provide API for shared memory message queues between tasks and ISRs on the 2 kernels.

4.2.3 Requirements of μC/OS-II Support on SIMULINK

- REQ-3-1 Provide μC/OS-II task block within SIMULINK for code-generation through task-view modeling.

- REQ-3-2 Provide code generation support for μC/OS-II from multi-rate SIMULINK models.
- REQ-3-3 Provide semantic preserving data transferring mechanism between μC/OS-II tasks within a single kernel.
- REQ-3-4 Provide semantics preserving rate transition blocks between μC/OS-II tasks across kernels.

Chapter 5

DESIGN

In this project, it is known that the multiprocessor platform is composed of equal capacity processors and that there exists a section of shared memory between these processors. Each core here executes an independent piece μC/OS-II RTOS code. Within each instance of μC/OS-II there exists a distinct set of tasks.

First a high level view of the memory model is provided, allowing easy understanding of the shared memory architecture and the code placement approach followed in this project. Then, the initialization done during the boot up process is provided here to highlight some of the aspects of porting and the areas where changes have been made to enable the AMP setup of μC/OS-II.

The last segment of this section provides the design of the μC/OS-II Task block for SIMULINK allowing task modeling of a system. Next, it provides design details of the process of code generation from mixed-rate SIMULINK models for μC/OS-II. Following which, details of the custom rate transition blocks for intra-core and inter-core data sharing have been provided.

For the purpose of explaining the μC/OS-II related services that have been added during the course of this project, some cases would be employed to represent the use of the API created and also for the explanation of the underlying implementation.

The representation of some of the common elements, such as the processor, task, priorities, etc. is provided here. A task Ti allocated to a processor (core) P_k, with priority 'ρ' and having periodicity of 't' is denoted with the notation Ti (P_k, ρ, t). Since, within this project we have dual core setup, named as Core_A and Core_B, let us assume the values taken by Pk = P_A, P_B. Also, since each core has a distinct μC/OS-II kernel, where each

kernel can also support 0-64 task priorities, there can be more than one task with the same priority within the entire task set. However these 2 tasks have to be present on different kernels/cores. In other words, no two tasks within one instance of μC/OS-II can share the same priority level.

5.1 μC/OS-II for MPC5675k in AMP mode

5.1.1 Memory Layout

When the MPC5675K MCU is running in the Decoupled Parallel Mode, the static RAM of 512 KB is split into two 256 KB memory sections. These 2 arrays are completely decoupled and mapped to different SRAM controllers, namely SRAM_0 and SRAM_1. The SRAM_0 array starts at 0x40000000 and SRAM_1 array starts at 0x50000000.

Since we are aiming to setup an AMP mode of operation with 2 independent RTOS running on each core, which do not share any of the code base, 2 independent linker files have been created. Two linker files are used for the creation of two separate executable program files, one for each core. The linker file, which is called the Linker Command File (LCF), along with the help of the Codewarrior Qorivva compiler directives, places the pieces of code into SRAM.

A brief introduction about the two important segments within the LCF, namely the memory and sections segment, allow us to understand the linker file setup described below. The memory segment is used to divide the memory of the MCU into various memory areas. Next, the Sections segment is used to define the contents of the target-memory area. By placing pragma directives within the source code, specific code can be placed in the desired memory sections. These sections can be named as required; however, care needs to be taken that the pragma directives are properly mapped to these section names.

In this project each linker file, named MPC5675K_RAM.lcf for both cores, creates a memory section for each 256 KB partition of the SRAM. The memory areas, also called memory blocks, specified within the memory segments on each LCF file are mirrored with slightly different labels and the addressing is only offset based on the start address of the SRAM_0 and SRAM_1 arrays. The starting addresses of these memory blocks have been highlighted within Figure 5.1

The memory blocks contain one or many grouped sections within them. The high level purpose of each of these memory sections is given below.

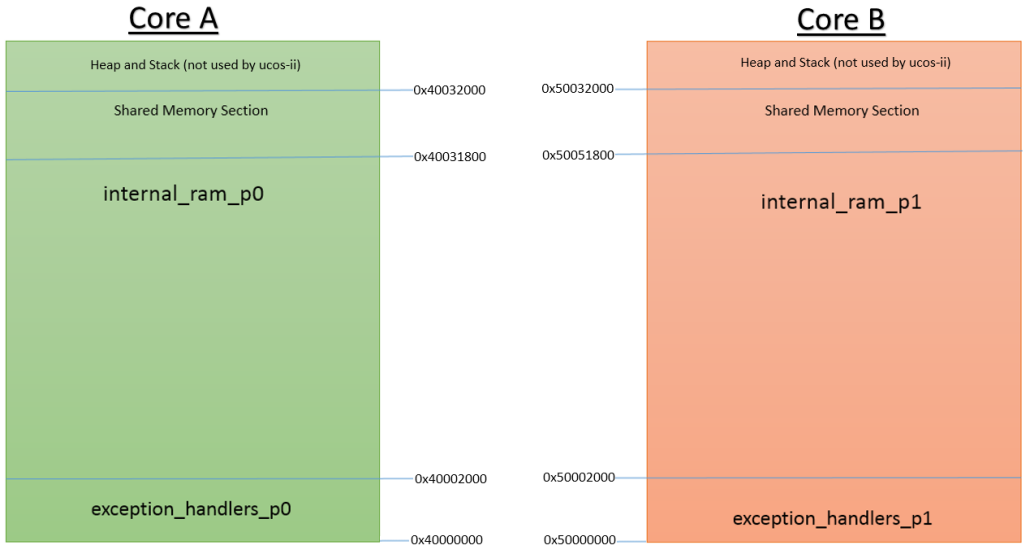


Figure 5.1: Memory Layout

1. **Exception Code:** The code responsible for handling exceptions are placed within the exception_handlers_p0 and exception_handlers_p1 sections of the corresponding LCF files. This section contains the code for the interrupt vector branch table, the ISR handlers and the supporting functions for the exception handling mechanism such as the function for registering the ISR to an interrupt line.

2. **Initialization Routines, Code and Data Placement in RAM:** This section holds the hardware related initialization routines, the application code including the operating system services and the constants used in the application and RTOS code. This section is named as internal_ram_p0 and internal_ram_p1 for Core_A and Core_B respectively. The hardware initialization done by Core_A, such as DPM mode startup code, initialization of RAM and most importantly the ppc_eabi_init object code, is present within the init_vle section. Details of each of these code sections are provided in the porting section 6.1.3.

   The text_vle_p1 on Core B holds the start_p1 object code which is responsible for Core_B specific hardware initialization such as the initialization of the MMU and the setup of the non-maskable interrupts.

   The application specific code and the RTOS code is placed within the text_vle memory sections on the respective SRAM memory regions. An important point to note here is that the entry point for the respective cores are also placed within this section. These entry points are named as __startup and __start_p1 respectively for Core_A and Core_B. internal_ram_p0 and internal_ram_p1 also holds the initialized and un-initialized data that is to be placed in SRAM.

3. **Shared Memory:** The MEMORY command, described above, provides the location and the size of the blocks of memory on the target memory. With it the Linker can be instructed where to place the sections of the code and which areas should be avoided. Taking advantage of this, a portion of memory has been omitted from the linker file for both cores. As shown in the Figure 5.1, 2 KB of memory from both SRAM_0 and SRAM_1 arrays have been excluded from the linker file, hence these memory regions do not contain any code or data within it. These sections of memory are dedicated as shared memory and using raw memory access data objects for

implementation of global semaphores, mutual exclusion (using atomic CAS) etc. can be placed here.

4. **Heap and Stack:** The standard C library implementation of dynamic memory allocation using malloc() and free() is not used and instead the μC/OS-II functions for dynamic memory allocation are used. Also, the stack of the tasks is placed within the code area of the memory using static stack allocation. This avoids problems of fragmentation that dynamic stack allocation imposes. More regarding the reasons for avoiding the use of standard C dynamic memory allocation functions can be found within the subsection 2.4.8 describing the memory management approach in μC/OS-II.

   Owing to these reasons the allocated heap memory in the linker can be reduced to a minimal amount and if required, with a few simple modification to the LCF, can be incorporated as a part of the shared memory internal_ram_p0 or internal_ram_p1 to place the code and global data.

   The mainline stack is used, however only for very limited purposes. When both cores enter through their respective application entry points, the stack used is the mainline stack. Any calls such as the BSP initialization and the μC/OS-II specific initialization is performed on this stack. Once the first task is loaded, on each of the cores, the mainline stack is not modified again and the tasks execute from within their static stacks.

5.1.2 Initialization and Bootstrapping

To run an AMP setup of μC/OS-II on the MPC5675K board the DPM mode needs to be enabled and the corresponding entry points needs to be set for the program counters on each of the cores. After the necessary initializations are done each core enters through the

respective entry points. The steps shown within Figure 5.2 represent a high level view of the entire booting process, details of which have been provided within this section. With the help of this booting process the key areas where porting and modifications to the existing startup code have been made can be understood clearly.

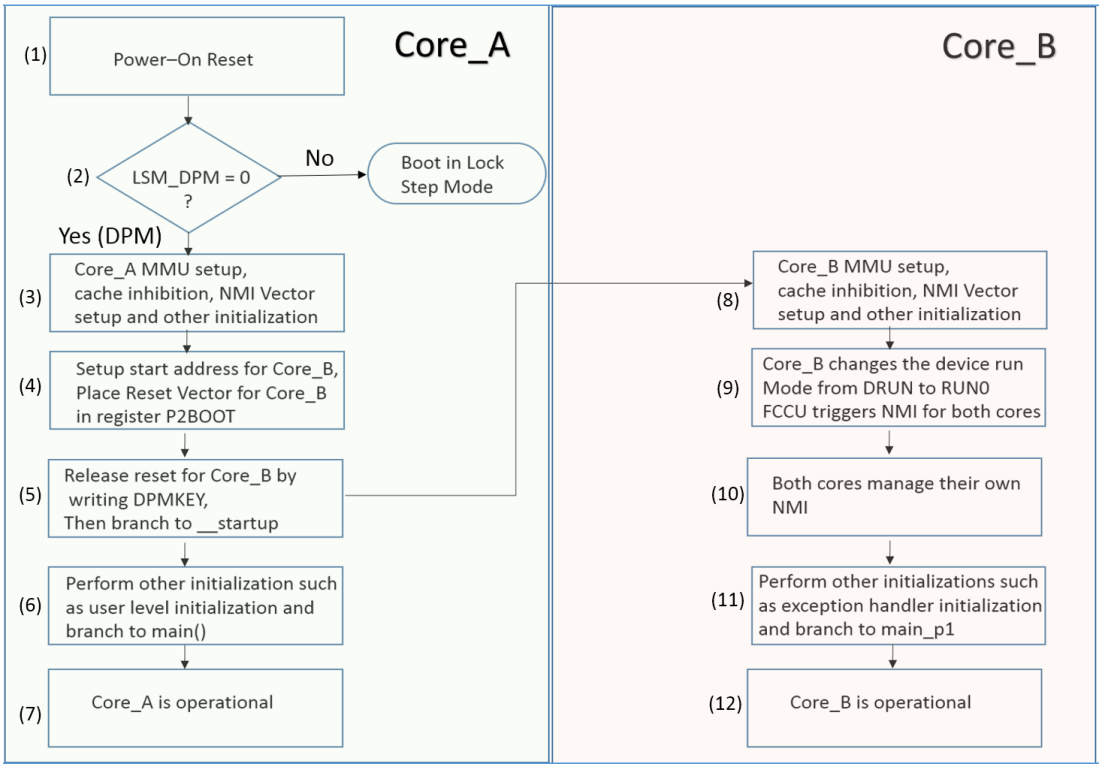


Figure 5.2: Boot Sequence

At power-on reset Core_A beings to run while Core_B continues to run in the held state (Fig 5.1, 1).

At this stage if the LSM_DPM user option bit within the shadow flash region is checked (Fig 5.1, 2). Based on the value of this bit the MCU can go into DPM or LSM mode. In this setup, the LSM_DPM bit has been set in the shadow flash region with the help of the debugger unit (which also contains the flash programmer), such that DPM mode is enabled. Following this, the required initialization specific to Core_A such as the MMU,

cache and interrupt vector initialization is done. Also, the SRAM initialization for both SRAM_0 and SRAM_1 is done at this stage (Fig 5.1, 3).

Once the initialization for the primary or master core (Core_A) is complete, to make the second core operational the DPMBOOT and DPMKEY registers must be configured. The reset vector for Core_B, which is start_p1, needs to be set into the DPMBOOT[P2BOOT] register (Fig 5.1, 4). After doing so, to be able to release the reset state of Core_B a unique sequence is written into the DPMKEY[KEY] register after which Core_1 jumps to the required reset vector. At this stage the MCU is running in DPM mode (Fig 5.1, 5).

Now, on Core_A other initializations such as the exception handler initialization, user specified initialization (within ppc_eabi_init) etc. is done. Following this it branches to main() and continues to run the application code (Fig 5.1, 6). At this stage Core_A can be considered to be completely operational.

One Core_B simultaneously its own set of initializations such as the MMU, cache, and non-maskable interrupts setup continues to run (Fig 5.1, 8). At this point it is important to note that the system is still running in the DRUN mode of operation, where there is full accessibility to the system, allowing full configuration. Once Core_B also finishes its initializations it changes the mode of operation to RUN0 (Fig 5.1, 9). Since a software change request of the chip mode is triggered, a non-maskable interrupt (NMI) is signaled to both cores by the FCCU. At this point both cores are capable of handling this NMI (Fig 5.1, 10). Further initialization specific to Core_B can be carried out after this(Fig 5.1, 11). Now Core_B can also be considered to be fully functional.

Further configurations, such as the setup involved in the Board Support Package, on both cores can be done following this as well if required.

This booting process primarily highlights some of the high level aspects of the MCU initialization and configuration. More details regarding this would be provided in sections 6.1.2 which describes the porting process detail.

5.2 Inter-Core Communication

One of the crucial components for implementing any kind of synchronization between the cores in an AMP mode is the need for the 2 cores to be able to communicate. An example of such a need would be, say a task on Core_A has currently acquired mutually exclusive access to a shared resource while a task on Core_B has been denied access. A simple approach would be that Core_B continues to spin on a shared memory variable, which could be setup as a flag, to see when the shared resource is free. However, such busy waiting approaches are wasteful of CPU utilization. The more desired approach would be that the task on Core_B blocks. However, the need for inter-processor communication arises when the task on Core_A does not require the shared resource any longer and has released the lock. In such an event, through inter-processor interrupts Core_B can be notified and can possibly acquire the shared resource.

There could be other use cases for the software settable interrupts, for instance to be able to able to invoke a lower priority software interrupt to handle the bottom half of a computationally intensive interrupt service routine. Thus allowing any subsequent higher priority interrupts to be handled in a timely manner.

In MPC5675K, software interrupts are treated as external interrupts and for each interrupt controller there are 8 such software settable interrupts. With the help of the INTC_InstallINTCInterruptHandler and INTC_InstallINTCInterruptHandler_p1 function on Core_A and Core_B respectively, the vector address of the ISR and priorities of these software interrupts can be set.

5.3 Mutual Exclusion - Spinlocks

The below mentioned algorithms ensure that the cores remain mutually exclusive of each other in the critical sections. However it is important to note that to achieve full mutual exclusion between tasks not only across cores but also within cores, and for the success of the below mentioned algorithms, interrupts also have to be disabled before calling these mutual exclusion primitives to disallow any context switches in the event of calling these functions.

5.3.1 Mutual Exclusion using Semaphore Peripheral

To implement spinlocks using the hardware semaphore peripheral, a simple algorithm based on the characteristics of the SEM4 peripheral suggested by Freescale has been used [42].

There is one pre-defined semaphore gate which has been used to implement spinlock for the entire application code on both Core_A and Core_B. In the call to SEMA4_SpinLock() first the gate value is checked to see if it is unlocked (Fig 5.3, 1-2). If the gate value is seen to be non-zero indicating that the gate has been locked the process continues to check in tight while loop till the value becomes zero indicating that the lock is free.

Next, once the lock is free the process tries to set the gate value with the core identifier (Fig 5.3, 3) (1 for Core_A and 2 for Core_B). After setting the gate value, it again checks to ensure that the gate lock has been acquired (Fig 5.3, 4). If the check shows that the setting of the gate has failed, i.e. the gate shows a value other than the core id that was set in the previous step, it restarts all over again. However if the check at (Fig 5.3, 4) shows that the gate value is the same as set in the previous step it exits and returns from the SEMA4_SpinLock() function and enters the global critical section.

```
void SEMA4_SpinLock()
{
    while(1)
    {
        do
        {
            current_valueA = SEMA4_0.GATE[1].B.GTFSM;     (1)
        }while(current_valueA != UNLOCKA);                (2)


        SEMA4_0.GATE[1].B.GTFSM = CPU0_LOCKA;             (3)

        current_valueA = SEMA4_0.GATE[1].B.GTFSM;
        if(current_valueA == CPU0_LOCKA)                  (4)
        {
            break;
        }
    }
}

void SEMA4_SpinUnlock()
{

        do
        {
            SEMA4_0.GATE[1].B.GTFSM; = UNLOCKA;           (5)
        }while(current_valueA != LOCKA);                  (6)
}
```

Figure 5.3: Spinlocks using Hardware Semaphore Peripherals

When the process intends to exit the global critical section, it calls the SEMA4_SpinUnlock() function. Within this function the process sets the gate value as unlocked and checks again to see if the gate is no longer locked by the core (Fig 5.3, 5-6).

5.3.2 Mutual Exclusion using Atomic CAS

The mutual exclusion primitives created for this project are based completely on the conventional spinlocks based on Compare and Swap algorithms. To implement spinlocks a pre-determined shared memory location needs to be known and be initialized with a value `0`. This initialization would be done within a section of code which is executed

before the spinlocks are called. This, within the scope of this project, is performed within ppc_eabi_init.c.

Within the AtomicCAS_SpinLock() function, this shared memory location is passed as a parameter to the Atomic_CAS() function defined in Section2.2.1 To this shared memory location the process attempts to write 1 if the value currently at the same location is 0. If the value at the shared memory location is not zero, indicating that the locks has already been acquired, then the function returns with a value 0 indicating failure to acquire the lock. To implement spinlocks it is required that the function be called repeatedly till it is able to successfully acquire the lock.

The AtomicCAS_SpinUnLock() function here does a simple write to the shared memory location with the value 0, indicating that the lock is free. However it is advised that an isync instruction be included before the value 1 is loaded to the shared memory location to allow the completion of the instructions in the pipeline before the process exits the critical section.

5.4 Inter-Core Barriers

5.4.1 Barriers using Semaphore Peripheral

The implementation of barriers is again based on the characteristics of the hardware semaphore peripheral as described in Section 2.3.2. The design of the barrier primitives used in this project is based on the application note provided by Freescale on the use of the hardware semaphore peripheral [42]. Barriers are critical within this project and hence a brief description of their design has been presented within this section.

To implement barriers two hardware semaphore gates have been used (other than the one already used in the implementation of the mutual exclusion primitive in the previous section) where each gate is managed by one of the cores. To implement the barri-

ers, both cores first set the status of their respective gates as locked, i.e. Core_A sets the SEMA4_0.GATE[2] value as locked by setting its core id and Core_B sets SEMA4_0.GATE[3] as locked. Then both cores check the status of each other's gates and if the value indicates unlocked it implies that the other core has not yet reached the barrier. The process on the core continues to spin on a tight while loop checking for the status of the gate held by the other core. When Core_B observes GATE[2] status as locked by Core_A and similarly when Core_A observes GATE[3] status as locked by Core_B it can be assumed that the barrier condition has been met. To reuse these barriers, each of the cores should set the state of their respective gates as unlocked after the barrier has been crossed.

5.5 μC/OS-II Global Counting Semaphores

The global semaphores described within this section are an extension of the existing semaphore implementation of μC/OS-II. With the help of the global semaphores, tasks across cores can be synchronized and global critical sections can be protected from concurrent access by multiple tasks.

Following are the pre-requisites for the implementation of global semaphores:

1. Mutual exclusion primitives should be available for the access of shared memory.

2. Inter-processor communication using interrupts should be established.

There are two version of the global semaphores provided within this implementation, one without the Priority Ceiling Protocol (PCP) semantics and the other without. The data structure setup for both the global semaphore (no PCP) and global semaphore with PCP are identical. Also, the in terms of the algorithm, except for the priority boosting aspect of the Global Semaphore with PCP algorithm the underlying working is identical for both implementations.

5.5.1 *Global Semaphore Data Structure Setup*

The basic setup required to have the global semaphore operational are the data structures mentioned within this section. The need for the each of these would be explained in further detail in the discussion of the Global Semaphore Algorithm. Figure 5.4 contains these data structures along with their appropriate location in memory. Details of these data structures is also provided below.

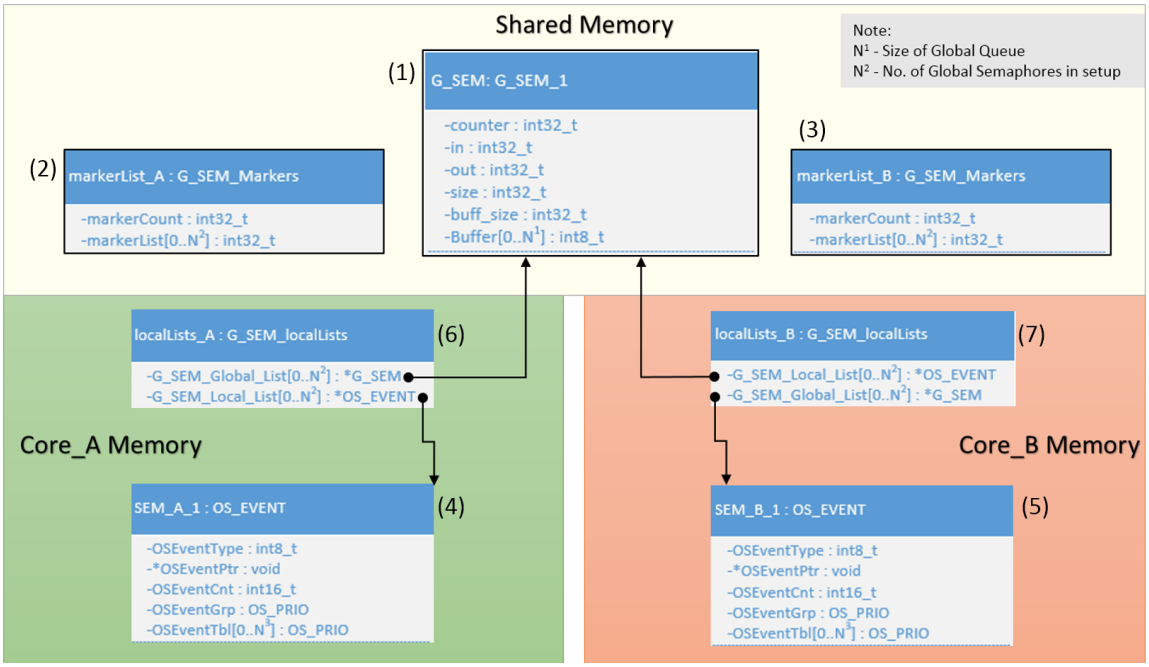


Figure 5.4: Global Semaphore Data Structures Setup

1. Each global semaphore consists of a global queue (G_SEM_1.Buffer) in shared memory capable of holding a unique integer identifier (Fig 5.4,1). This unique identifier would represent the core IDs for each of the cores. Since the MPC5675K has only two cores, Core_A is represented as 1 and Core_B as 2 within this list.

2. Each global semaphore also requires a global counter (G_SEM_1.counter) present in shared memory to hold the semaphore counter value (Fig 5.4,1). The global

queue and the global counter have been encapsulated into a single data structure called G_SEM. In this example a single instance of this structure is created called G_SEM_1 representative of one global semaphore.The global queue would also have the accompanying data structures such as in, out, size etc. to support the FIFO functionality of the semaphore queue.

3. Two list, one for each core, called the markerList[index] is created in shared memory. Each entry within this list is used for holding the release counter for each of the global semaphores. The indexed entries within this list are mapped to the global semaphores and the local semaphores lists present on each core (described below).The markerList is encapsulated within the G_SEM_Markers structure along with the another variable to hold the number of non-zero entries within each of the lists. Irrespective of the number of the Global Semaphores present in the system only two data structures of type G_SEM_Markers are required in the shared memory, one for each core, named as markerList_A and markerList_B (Fig 5.4, 2 & 3). These lists allow the scale up of the implementation to accommodate more than one global semaphores.

4. A local Event Control Block (referred to as the OS_EVENT structure within μC/OS-II) is created and initialized to the type Semaphore on each core (Fig 5.4, 4 & 5). With these local ECBs (semaphores), the state of the local tasks on each core can be controlled. The tasks can either be placed into the waiting state or into the ready list based on the Global Semaphore algorithm decried below.

   For the sake of explanation we would refer to these ECB entities as SEM_A_1 and SEM_B_1. Based on the number of global semaphores required, that many local SEM_A_n and SEM_B_n would need to be created on each core.

5. Each core also contains two lists to maintain all the global and local semaphores created. These lists are indexed from 0 to N, where N (referred to as N^2 within Fig 5.4) is the number of global semaphores present within the system. Entry "i" within the list holds the pointer to the respective local and global semaphores. Let us name these lists as G_SEM_Local_List[index] and G_SEM_Global_List[index]. These lists are encapsulated into a structure called G_SEM_localLists (Fig 5.4, 6 & 7).

5.5.2 Global Semaphore Algorithm

With the description of the required data structures given in the previous section, this section would provide the details of the algorithm in terms of cases. For the sake of explanation let us assume that the global semaphores are only working in the capacity of binary semaphores.

However, this implementation also supports counting semaphores as well. The only change that would need to be incorporated to support the counting semaphores is the initialization of global semaphore counter value within ppc_eabi_init. Here, instead of the value being 1, it can be initialized to any other value. Similarly, the local semaphores on each of the cores associated with this global semaphore would also have to be created and initialized with the same value as that of the global semaphore counter.

To explain the various aspects of the algorithm let us assume there are two tasks, one on each core contending to acquire the same shared resource to enter into a global critical section protected by global semaphores. Let these task be represented as T_1 (P_A, 1) and T_2 (P_B, 3). We assume that they are scheduled according to the fixed priority scheduling policy within each of the μC/OS-II kernels. Both these task are sharing a global resource G_SEM_1.

Initial Setup:

Each of the global semaphore counter values are set to the required value within ppc_eabi_init section during the initialization process. This is required to be done before any of the involved tasks are created to ensure that the global semaphore setup is ready when these tasks start to run. Since it is a binary semaphore that we are considering, G_SEM_1.counter value is set 1. Apart from this all of the queue management data structures such as G_SEM_1.in, G_SEM.out etc. have also been initialized appropriately (according to the semantics of the FIFO lists).

Before each of these tasks are spawned the local semaphores SEM_A_1 and SEM_B_1 are created and assigned to the local lists G_SEM_Local_List[index] on each of the respective cores with the help of the G_OSSemCreate() API. Also, using G_OSSemCreate() API, the pointer to G_SEM_1 is added to the G_SEM_Global_List[index]. Since there is only one global semaphore, the index value is set to 0. In the case where there are more than one global semaphores, corresponding entries within these lists would be populated.

Case 1: Acquiring the Lock When the Resource Is Free:

In the case that none of the contending tasks have acquired the global semaphore and now task T_1 (P_A, 1) has started its execution (Fig 5.5, 1) and calls the G_OSSemPend() API (Fig 5.5, 2) to acquire the global semaphore. In this process first the global counter value is decremented to indicate a request. After decrementing it checks to see if the G_SEM_1.counter value is less than 0. This decrement and check is done with the help of global mutual exclusion primitive described earlier to ensure consistent values. In this case the value would indicate 0 (Fig 5.5, 3), hence allowing the task to acquire the semaphore. Once the task returns (Fig 5.5, 4) from the G_OSSemPend() function call it begins to run within the global critical section (Fig 5.5, 5).

After the task has acquired the global semaphore it can however be pre-empted by other higher priority tasks within the same kernel.

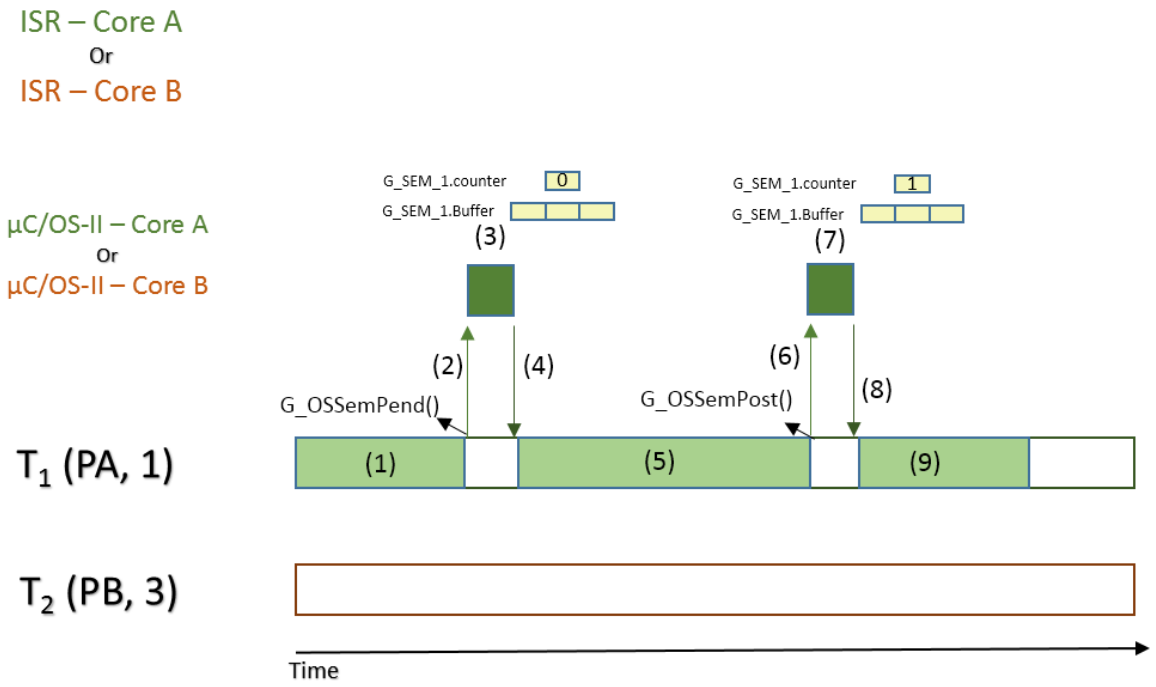


Figure 5.5: Global Semaphore Timing Diagram: No Resource Contention

Case 2: Releasing the lock when no other task is waiting:

This case represents the situation where only task T_1 (P_A, 1) has acquired the semaphore and no other tasks, either on the same core or the other core, are waiting on the semaphore. Now, when task T_1 (P_A, 1) exits the global critical section it calls the G_OSSemPost() API (Fig 5.5, 6). Within this function call it first checks the global counter value (Fig 5.5, 7). This check is done by first locally copying the G_SEM_1.counter value and incrementing it. If the value of the locally held global counter value still is less than or equal to zero, it indicates that a task is waiting on the resource. This case does not apply in this scenario as we have assumed no other task has requested for the global semaphore yet. If the value of the locally held global counter value is greater than zero it indicates that no

task is waiting on the global semaphore. This applies in the currently considered scenario and the counter would hold a value of 1. It then commits this local global counter value to the G_SEM_1.counter value in shared memory and returns back (Fig 5.5, 8) from the function call. After doing so, task T_1 (P_A, 1) now runs outside the scope of the global critical section (Fig 5.5, 9).

Case 3: Blocking when resource is not free:

Let us assume that task T_1 (P_A, 1) has already entered into the global critical section (Fig 5.6, 1) and now task T_2 (P_B, 3) begins to run on Core_B. Then it attempts to acquire the global binary semaphore by calling G_OSSemPend() (Fig 5.6, 3). Within this function it first decrements the global counter value first and then checks the G_SEM.counter with the help of the global mutual exclusion primitive to ensure consistent values. Here, since it sees the counter value as -1 which indicates that the resource is no longer available. In such a case the task T_2 (P_B, 3) needs to block. Before blocking it first adds the Core_B ID '2' onto the global queue at G_SEM_1.in position and increments the 'in' value as well. It also then increments the G_SEM_1.size value and finally sets the status of the task to Wait state by calling the native μC/OS-II API OS_EventTaskWait() (Fig 5.6, 4).

After the current task's state has been changed to waiting the scheduler is called again to run the next available task from the local ready list of Core_B.

Case 4: Releasing the lock when other task is waiting:

In the situation where task T_1 (P_A, 1) has acquired the global semaphore and when it is in the global critical section, task T_2 (P_B, 3) tries to acquire the semaphore. In this case it gets blocked and the SEM_B_1 local ECB maintains the state of this task as waiting. Also, in the global semaphore queue an entry exists with core B's ID (2). When task T_1 (P_A, 1) intends to exit the global critical section it first calls the G_OSSemPost() function (Fig 5.6, 5).

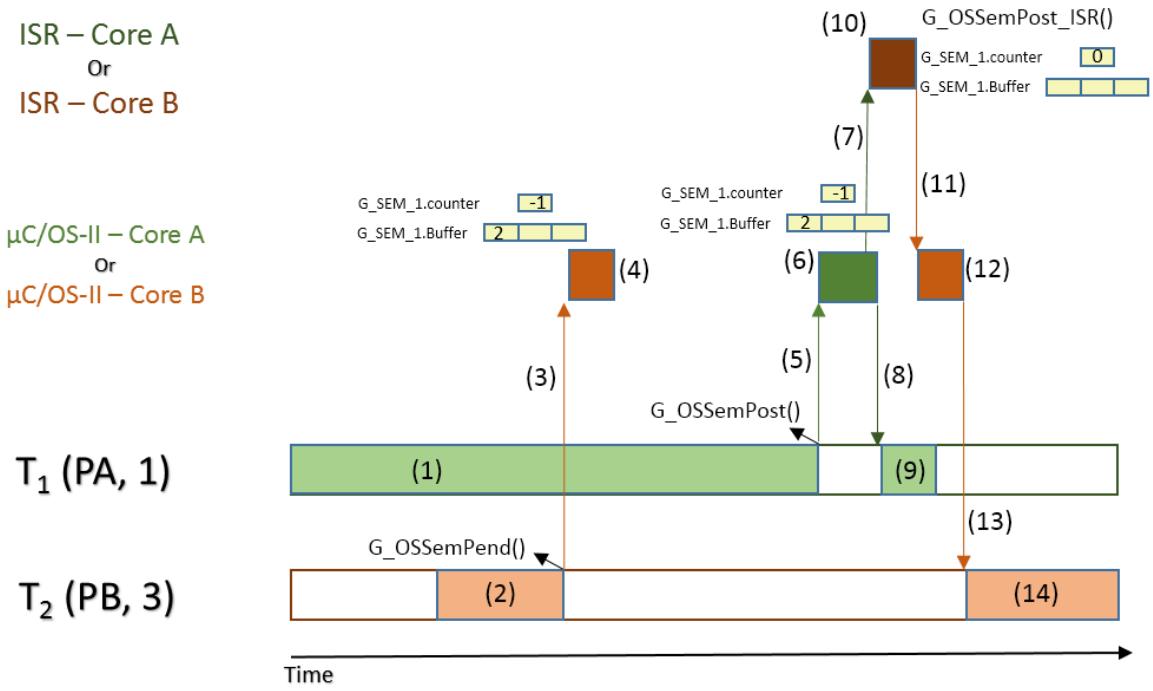


Figure 5.6: Global Semaphore Timing Diagram: With Resource Contention

Again, similar to the steps mentioned in Case 2, within this function call it first it checks the global counter value (Fig 5.6, 6). This check is done by first locally copying the G_SEM_1.counter value and then incrementing it. This time the value of the locally held global counter value is equal to -1 which indicates that a task is waiting on the resource. It then proceeds to check the head of the global semaphore queue and retrieve the core ID. In this case it sees that the core ID is 2, indicating that a task from Core_B needs to be woken up. This function then increments the release counter value within the markerList[index] list at the corresponding index for the global semaphore. In this scenario the index would be zero as no other global semaphores have been considered. After doing so, an inter-processor interrupt is sent from Core_A to Core_B (Fig 5.6, 7).

It is important to note that the global counter value in the shared memory has not been incremented yet.

After T_1 (P_A, 1) raises the IPI, it returns back and has exited the global critical section (Fig 5.6, 8). It then continues to run outside the global critical section protected by G_SEM_1 (Fig 5.6, 9).

On Core_B, as a part of the ISR to handle the inter-processor interrupt sent by Core_A (Fig 5.6, 10), all the indices within the markerList[index] list are checked for non-zero values. Here at index zero a value 1 would be observed indicating that G_SEM_1 has been released once. Following this, data structures corresponding to the global semaphore, i.e. G_SEM_1, is retrieved from G_SEM_Global_List[index] at index 0. Also with this index the local ECB, i.e. SEM_B_1, is retrieved from G_SEM_Local_List[index].

Then the markerList[index] value is set to zero to indicate that the IPI has been handled. With the retrieved index value (index = 0) from the markerList[index] as a parameter, G_OSSemPost_ISR() is called from within the ISR. This G_OSSemPost_ISR(), essentially behaves like a proxy function for the G_OSSemPost() done by task T_1 (P_A, 1).

Within the G_OSSemPost_ISR() function (Fig 5.6, 10) again the local copy of the global counter value is incremented and checked. This would hold a value of 0 indicating a waiting task. However, since the head of the global queue contains Core_B's ID, it indicates that a task locally present on Core_B needs to be granted the semaphore. It then goes ahead and commits the local value of the global counter to the G_SEM_1.counter in shared memory.

Following this the G_SEM_1.out value is updated and G_SEM_1.size value is decremented to show dequeue of the head entry in the global queue. After doing so, the waiting task T_2 (P_B, 3) is put onto the ready queue by calling the native μC/OS-II function OS_EventTaskRdy() with SEM_B_1 event control block as the parameter. Finally, the scheduler is called again and if the recently woken up task is the highest priority task in the ready list it would get scheduled (Fig 5.6, 14) or the current task which was interrupted with the IPI continues to run.

Case 5: Releasing the lock with interrupts disabled on other cores:

To understand this scenario let us assume that there are two task on Core_A, say T_1 (P_A, 1) and T_2 (P_A, 2). On Core_B there exists two tasks, say T_3 (P_B, 3) and T_4 (P_B, 4). There also exists another task on Core_B, say T_5 (P_B, 5), which does not share any critical sections with any of the above mentioned 4 tasks. However, this task contains a critical section of its own requiring that the interrupts be disabled on core_B.

Now, let us assume that T_1 (P_A, 1) and T_3 (P_B, 3) share a global critical section protected by G_SEM_1. Similarly, tasks T_2 (P_A, 2) and T_4 (P_B, 4) share a global critical section protected by global semaphore G_SEM_2.

Next, let us assume that T_1 (P_A, 1) and T_2 (P_A, 2) have acquired exclusive access to their respective global critical sections. Following which, when tasks T_3 (P_B, 3) and T_4 (P_B, 4) try to acquire the locks, they get blocked on their local ECBs. This would allow task T_5 (P_B, 5) to run.

In the course of execution of task T_5 (P_B, 5), disables interrupts on Core_B as mentioned before. This would mean that all IPI inbound to Core_B would be disabled. During this period, it is possible that both task T_1 (P_A, 1) and T_2 (P_A, 2) release their respective global semaphores. The interrupt request would be latched but would indicate only one occurrence of the IPI. We have to ensure that both releases are handled and correspondingly both tasks on Core_B are placed on the ready-list.

To avoid the above mentioned issue, when tasks on Core_A release the global semaphores, as a part of the G_OSSemPost() calls, they would not only raise the IPI but also increment the appropriate release counter value within the markerList[index] list for the global semaphore that they are releasing. In this case both entry 0 and entry 1 within this list would be incremented to 1.

When the interrupts do get enabled on Core_B, as a part of the ISR to handle this IPI, it would go through the entire list and check for all non-zero entries and call the

G_OSSemPost_ISR() for those local ECBs. This would put both task, which were waiting state, into the ready queue. If this approach would not have been followed, some of the semaphore releases would have been lost in the event of interrupts being disabled.

5.5.3 Global Semaphore with PCP Algorithm

Before the discussion of the algorithm it is important to realize the importance of the priority ceiling protocol. The MSOS algorithm [37] proposed by Nemati et al and other multi-processor synchronization protocols [39] mention the need for bounded blocking and that the maximum blocking time should be a function of only the global critical sections. This enables a certain degree of guarantees in the schedulability of the system.

To explain the various aspects of the algorithm let us assume there are 4 tasks, two on each core. Let these tasks be represented as $T_1(P_A, 17)$, $T_2(P_A, 16)$, $T_3(P_B, 15)$, $T_4(P_B, 14)$. Of these tasks $T_1(P_A, 17)$ and $T_3(P_B, 15)$ share a global critical resource, say G_SEM_1. The priority boosting principle proposed by the MSOS [37] algorithm is as follows:

- Priority of Task T_i is increased immediately to $p_i + max_i(p_i \mid T_i$ on Processor $P_n)$

However, the priority ceiling protocol followed in this approach is slightly different, and assumes that if there are N tasks in the system then each of the tasks would be assigned a priority equal to N, N+1, N+2 etc. Now, when a task Ti with its original priority as N+ i requires a global resource, its priority gets boosted to i. As an example, the priority of $T_1(P_A, 17)$ on attempting to acquire the global resource immediately becomes 7.

For the purpose of implementing the PCP feature within the Global Semaphore implementation a custom function called G_OSTaskChangePrio() has been created. This is a custom function and is similar to the native μC/OS-II OSTaskChangePrio() API used to change the priority of a task. With the help of G_OSTaskChangePrio function it is

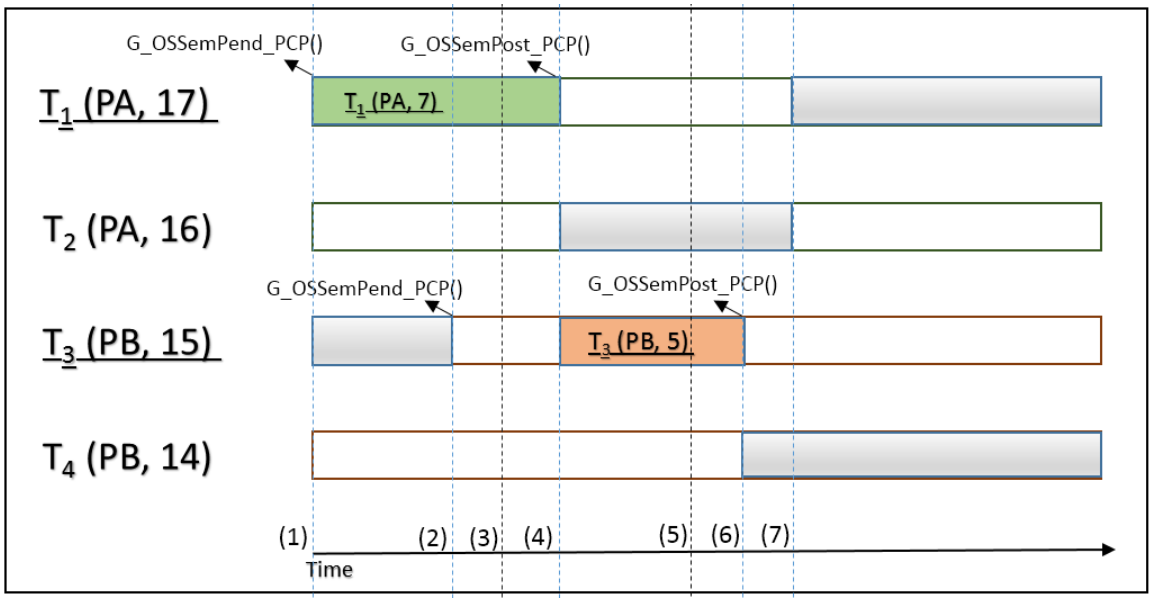


Figure 5.7: Global Semaphore with PCP Timing Diagram

possible to increase or decrease the priority of the calling task by the desired amount, as long as the new priority value is not already taken by another task.

In this example, first when $T_1(P_A, 17)$ tries to acquire the global resource (Fig 5.7, 1) by calling the G_ OSSemPend_PCP(), its priority gets updated to 7. This is done by calling the G_OSTaskChangePrio function and passing 10 as a parameter. This increase the priority of the task by a value of 10. Now, in this case since there are no other tasks which have acquired the global semaphore, all of the data structure updates which occurred during the call to the G_OSSemPend() are carried out. After the task returns from the G_OSSemPend_PCP() function cal it is running within the global critical section with a priority of 7.

Now, at (Fig 5.7, 2) say task $T_3(P_B, 15)$ requests for access to the global critical section by calling the G_ OSSemPend_PCP(), within which again its priority is upgraded to 5 as a part of this function call. Next it checks the global semaphore counter value and realizes

that it cannot be granted access. With this task T_3 blocks, in the same manner as described in the global semaphore implementation, till the global semaphore becomes available.

Say, at (Fig 5.7, 3) another task $T_2(P_A, 16)$ gets released on Core_A, however owing to the priority ceiling protocol it is not allowed to run since task T_1 is already running with an upgraded priority of 7.

Now, when T_1 intends to exit the global critical section by calling the G_OSSemPost_PCP() (Fig 5.7, 4), first its priority is downgraded to its original priority of 17 and then it releases the semaphore with the help of an IPI as described in the previous global semaphore algorithm. Within this call after the priorities have been restored and the IPI has been sent, OS_Sched() is called which schedules the highest priority task to begin executing immediately. This highest priority task would be $T_2(P_A, 16)$.

Simultaneously, on Core_B (Fig 5.7, 4) when the IPI is serviced it grants access to task T_3 (which is blocked with a priority of 5).

Now, let us assume that a task $T_4(P_B, 14)$ gets released at (Fig 5.7, 5), however since T_3 is running with priority of 5 T_4 is not allowed to run. Only after T_3 releases the global semaphore and reduces its priority to 15 again is $T_4(P_B, 14)$ allowed to run (Fig 5.7, 6).

With this example provided above, it is seen how the Priority Ceiling Protocol Semantics are preserved and the worst case blocking time of tasks dependent on the global resource is limited only at most to the total execution time of all the global critical sections, without the interference of tasks not involved in the global critical section.

5.6 μC/OS-II Shared Memory Management

To explain the various aspects of the shared memory management algorithm let us us assume that there are multiple tasks on Core_A, say T_1 (P_A, 1) and T_2 (P_A, 2), which are responsible for gathering sensor data over the CAN channel from external sensor units. In this setup, tasks on Core_B are responsible for only error logging and handling in the

case of catastrophic errors occurring on Core_A. Let the task responsible for error logging on Core_B be T_3 (P_B, 1). Each task on Core_A checks for the incoming sensor values and if the values exceed a certain threshold it wakes up the task on the other core using global semaphores described in the previous section. After the error logging task wakes up, it reads the values of the erroneous sensor data from a shared memory queue. This data is present in the form of a message on the message queue and the tasks then logs the time stamp and possibly wakes up other error handling tasks.

The workings of the shared memory queue would be described in the upcoming sections, but for now we would not go into the specifics and assume that the queue works like any other producer-consumer queue that is shared between two or more tasks either on the same core or on different cores. Here the message is the pointer to the memory block to which the tasks on Core_A have recorded the erroneous sensor data.

5.6.1 Shared Memory Management Data Structure and Initial Setup

Following are the the steps to initialize the data structures:

- Here the initialization is taken care of within Core_A. As a part of the initialization once the starting address of the memory partition has been statically assigned in the shared memory, it is required that this memory partition be broken down into smaller memory blocks of the required block size. Say in this example we require 10 blocks of size 32 bytes each. Hence it should be taken care that memory partition is at least 32 * 10 bytes in size. To setup this memory partition we call the G_OSMemCreate() function. This function is the same as the native function except that it does not allocate the MCB from the OSMemFreeList, but instead the statically created MCB is passed as a parameter to the function. This function is typically called before the tasks requiring these memory blocks are spawned.

- Also within G_OSMemCreate() function a linked list of the memory blocks is created. Any access of shared memory should be mutually exclusive not only between the tasks on Core_A but also from Core_B hence it is required this creation of the linked list is protected within the inter-core mutual exclusion primitive.

- The start address of the memory partition, which is also the start address of the first memory block of size 32 bytes, is saved within the PartitionMgr.OSMemAddr. Other data structures required for the housekeeping of the shared memory such as the number of blocks created, number of blocks free and the size of each block size is also saved within the PartitionMgr data structure.

- Here the inter-core barrier is placed after the call to G_OSMemCreate() and the creation of any tasks on Core_A and before the creation of any tasks on Core_B. This ensures that the tasks on the other core do not start to execute and work with the shared memory before the setup is complete. This barrier is only a one-time requirement for the initialization.

- Along with the initialization of the shared memory partition, the shared memory queue would also need to be initialized. This can be done with the call to G_OSQCreate(). Details regarding the initialization of the shared memory queues would be given in the following sections.

5.6.2 Shared Memory Management Algorithm

Following is the scenario based description of the Shared Memory Algorithm.

Acquiring a free shared memory block:

When a task on core PA detects that the sensor data has crossed the threshold value, it first acquires a free memory block from the partition present in shared memory. This

is done with the help of the G_OSMemGet(). This call requires that the shared MCB, namely PartitionMgr, is passed as an argument to this function. Within this function call, after the preliminary check have been made to see if the MCB passed is valid, first it is checked that there exists a free memory block within the linked list of free memory block. This pointer to this free list is present within the MCB in the field called G_PartitionPtr->OSMemFreeList. After a memory block is available it adjust this free list pointer to point to the updated free list. After this the number of free memory blocks contained within the G_PartitionPtr->OSMemNFree field is decremented. Finally the pointer to the retrieved free memory block is returned back to the tasks requesting the memory block. All of the above changes were made within the critical section guarded by disabling interrupts and the global mutual exclusion primitive.

Following this the task records the data onto the shared memory block and posts the pointer to this block into the shared memory queue by calling the G_OSQPost() function. The tasks wake up on core PB would be handled by the event of posting a message onto the message queue.

No free memory blocks available:

To check if there are any free memory blocks available within the G_PartitionPtr->OSMemFreeList linked list the G_PartitionPtr->OSMemNFree value is checked. In the event that this value is not greater than zero the function returns immediately after the enabling interrupts and exiting the inter-processor mutual exclusion primitive with an error code OS_ERR_MEM_NO_FREE_BLKS. With the help of this error message the task can be made either made to skip the posting of the erroneous data onto the queue for another time or it could continually keep checking for the availability of the free block. Such use of error messages would be an application design consideration.

Releasing a memory block:

When the error logging task, which was waiting for the message to become available on the message queue, receives the message by calling G_OSQPend() it performs the required error logging activity.

Other tasks could also be woken as a consequence to perform the error handling activity as well. Once the message has been processed by the error logging task on Core_B it is responsible for returning back this shared memory block. This is done with the help of calling the G_OSMemPut() function. As a part of this function call, the shared memory MCB is passed as a parameter along with the pointer to the memory block which needs to be freed up. Within this function again preliminary check are performed to see if the MCB is valid or not. Next G_PartitionPtr->OSMemNFree value is compared to the G_PartitionPtr->OSMemNBlks to check if all the blocks are already present within the free memory block free linked list (G_PartitionPtr->OSMemFreeList). If this value is greater than or equal then it implies that the returned memory block does not belong to shared memory partition under consideration. Following this it return immediately with an error code - OS_ERR_MEM_FULL.

However, if the G_PartitionPtr->OSMemNFree value is not greater or equal to G_PartitionPtr->OSMemNBlks then this memory block is added to the G_PartitionPtr->OSMemFreeList and the G_PartitionPtr->OSMemNFree value is incremented to show the addition of this recently freed block to the pool of available memory blocks.

5.7 μC/OS-II Shared Memory Queues

The implementation of the global shared memory queues can have numerous advantages in a setup such as the one presented within this project. Since separate task sets exists on each of the cores, to co-ordinate any kind of functionality involving the use of data manipulation involving more than one task, a protocol for inter-task communication across

kernels needs to be established. The messaging queue service provided by μC/OS-II can be extended to support this multi-kernel setup. With this message queue implementation tasks as well as ISRs can produce or consume data. The implementation offered here supports multiple readers and writer, either on the same core or distributed across cores, to the same queue.

Following are the pre-requisites and assumptions that apply for the implementation of global shared memory queues:

1. Mutual exclusion primitives should be available for the access of shared memory.

2. Inter-processor communication using interrupts should be established.

3. The tasks across the cores for the purpose of shared memory queues are assumed to follow the same priority patterns. More regarding this would be discussed within the section explaining he algorithm of shared memory queues.

4. Shared Memory Management across the kernels should be available. A point to note here is that it is not necessary to use the shared memory blocks as the data objects to be passed. Instead pointers to other data structures present in shared memory can also be passed within this queue. However care has to be taken that these shared memory data structures remain static till the reader has completed consuming the data.

Since we have already discussed the shared memory partition management in the previous section, we would be utilizing the dynamically allocated shared memory within this discussion of shared memory queues.

Since both tasks as well as ISR are capable of queueing and de-queuing data packets from the global shared memory queues, they would be referred to as producer entities and consumer entities when referred to in general.

5.7.1 Shared Memory Queues Data Structure Setup

The basic setup required to have the global shared memory queues is provided within this section. The following Figure 5.8 also contains these data structures and contains numbered markers to correspond to each of the data structures described below.

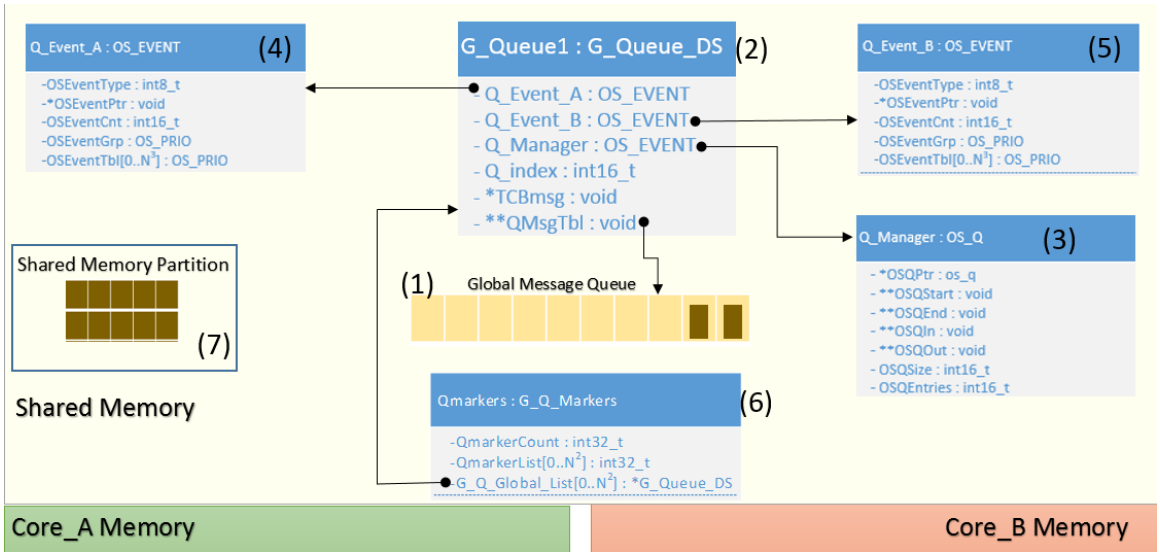


Figure 5.8: Global Queues Data Structures Setup

1. The data structure of type G_Queue_DS is created for each intended global shared memory queue. This data structure consists encapsulates all of the required data structures such as the global shared memory queue, ECBs, QCB and other supporting data structures which allow inter-processor communication. Since we are considering only one global shared memory queue, only one instance of this queue has been created within this example, namely G_Queue1 (Fig 5.8, 2). However, with the help of G_Q_Global_List[index] and G_Q_Global_List [index], the system supports multiple such shared memory queues as well. The following points regarding the data structures covers all of the constituent data structures of the G_Queue_DS, needed to implement the global shared memory queue mechanism.

2. Each global shared memory queue requires an array of pointers to hold the data packets deposited by the producer entities and those that would be de-queued and read by the consumer entities. This queue structure within this implementation is called as the .QMsgTbl (Fig 5.8, 1). This array is created statically as a member of the G_Queue1 data structure of type G_Queue_DS (Fig 5.8, 2).

3. A queue control block (QCB) is also required to manage the queue. The QCB control block is named as .Q_Manager (Fig 5.8, 3). The functionality of each of the constituent members of this OS_Q type data structure has been provided in section 2.4.7. Instead of allocating a free QCB from the .OSQFreeList that the μC/OS-II kernel maintains, this is also statically created in shared memory as a member of the G_Queue1 data structure of type G_Queue_DS (Fig 5.8, 2).

4. An Event Control Block (ECB) is required within a typical implementation of the μC/OS-II message queues to maintain a wait-list of the tasks which are currently awaiting the producer entity to deposit a data packet. In this implementation since there are two kernels with two distinct set of tasks, two separate event control blocks have been statically created in memory. The OSEventTbl[] table, present within Q_Event_A (Fig 5.8, 4) and Q_Event_B (Fig 5.8, 5), maintains the list of tasks waiting on a message on Core_A and Core_B respectively. Details about how these wait-list tables are maintained can be found at [25, p. 295]. These Q_Event_A and Q_Event_B data structures of type OS_EVENT are also created as a member of G_Queue1 of type G_Queue_D (Fig 5.8, 2).

5. All of the above mentioned data structures are members of the G_Queue_DS. If N (represented as N^2 within (Fig 5.8, 3) such queues are required, then N instances of G_Queue_DS need to be statically created in shared memory.

6. Apart from the above mentioned data structures, another data structures of type G_Q_Markers, namely Qmarkers (Fig 5.8, 6) is needed to be placed in shared memory, to support the entire global shared memory queue mechanism. This data structures is created only once for the entire setup and with this it is possible to scale up the global queues mechanism to more than one shared memory queues. At the time of initialization queues are created using the G_OSQCreate function by either Core_A or Core_B. Within this function call pointer to each of the global queue data structures (G_Queue1, G_Queue2 etc.) is passed along with the index value. Based on the index value provided as a parameter, the pointers to these global queue data structures are stored into the Qmarkers.G_Q_Global_List[index] array at the index provided.

7. The G_Q_Markers structure also contains another data structure called the QmarkersList[index] (Fig 5.8, 6). Each entry within this array at any point of time represents the status of the global shared memory queue corresponding to the index value. Each entry within this array would either hold the value 0, 1 or 2. Value of "0" represents that no message is pending to be delivered to a task on either core. Value "1" represents that a message is pending to be delivered to a task on Core_A and a value of "2" represents that a message is pending for a task on Core_B.

8. Since in the upcoming cases, which have been presented to discuss the mechanism of shared memory queues, the data packets that would be en-queued into the global queue would be memory blocks, this diagram also represents the memory partition. To manage this memory partition, the shared memory management concepts discussed in section 5.6 can be referred to.

5.7.2 Shared Memory Queues Algorithm

This section would provide the details of the shared memory message queue capability that has been added to μC/OS-II with the help of certain cases. The intricacies of the implementation as well as the use of the API have been discussed within these cases.

In many ways the implementation for the shared memory queues, in its approach, is similar to the global semaphores discussed in the previous section. However, the key distinctions worth mentioning here are that, unlike the global semaphores where the task wake up pattern was based on FIFO, in this setup tasks are allowed to read the message based on their priority. Say for instance there are 2 tasks waiting on a message irrespective of the fact that they belong to the same core or to separate cores, on the production of the message the highest priority task gets access to this message. It is important to note here that the tasks across the cores now follow the same priority convention, where a task T_1 (P_A, 1) and T_2 (P_B, 1) have the same priority. On the same lines, T_2 (P_B, 3) has a lower priority than T_1 (P_A, 2).

In the first case where both tasks have the same priority across the cores, let us assume that both these tasks are consumers waiting for a message. If the producer entity resides on Core_A then it prefers to deliver the message to T_1 (P_A, 1) and if the producer entity resides on Core_B then it would prefer to deliver the message to T_2 (P_B, 1). This design decision was taken to minimize inter-processor interrupts being raised. However this can logic can be tweaked with minor modifications to the code, to have a more fair decision making process. In the situation where tasks of equal priority are contending to acquire the next message a simple toggle bit can ensure that half the times the message gets delivered to tasks on the same core and the other half to the tasks on the other core.

Since, in all the cases mentioned below only global queue has been created and used for the message passing between tasks and ISRs, the initial setup given below is only provided

once and applies to all cases. Also, it is assumed that the data being passed on the global queue, in all of the cases below, are dynamic shared memory blocks which have been acquired by the producer entities using the G_OSMemGet() API. When the consumer entity, on receiving and processing this data, no longer needs this shared memory block it releases it by calling the G_OSMemPut() API. Details of these API calls be found in the previous section of shared memory management.

Also all access to the QCB, ECBs and the global queues are performed within the protection of the global mutual exclusion primitive to ensure integrity of these shared memory data structures.

Initial Setup:

1. The data global queue data structure G_Queue1 of type G_Queue_DS needs to be initialized before the tasks or the ISR begin to utilize the global message queue service. Since typically these queues would be used for communication across cores, the initialization of these data structures by calling G_OSQCreate() needs to be done by either of the cores. It is preferred however that Core_A does this initialization within the ppc_eabi_init.c along with the help of inter-core barriers to prevent the other core from starting up before the initialization is complete.

2. As a part of the call to G_OSQCreate() the global queue data structure is passed along with the index. Based on this index value the pointer to the global queue data structure gets added into the Qmarkers.G_Q_Global_List[index] (Fig 5.8, 6) and the Q_index value within the global queue data structure is also set to the value "index" (Fig 5.8, 2). Apart from this function also initializes the QCB and the two ECBs.

Case 1: Producer Enqueues Before Consumer Dequeues:

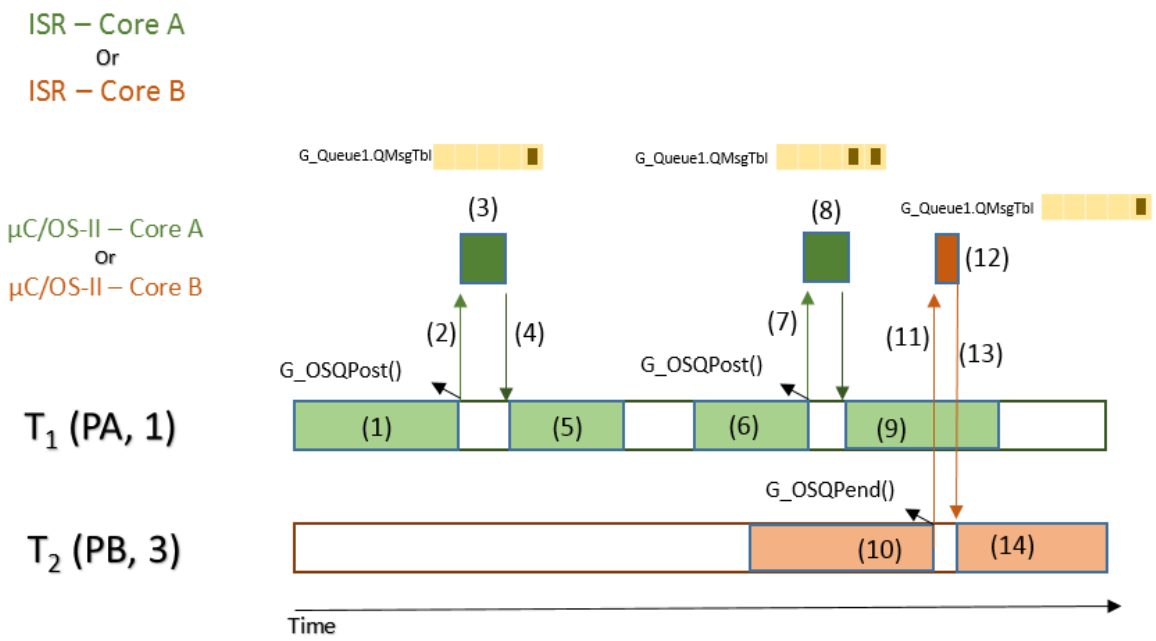


Figure 5.9: Global Queues Case 1: Producer Enqueues Before Consumer Dequeues

This case intends to show the working of the queue mechanism when the producer entity, say T_1 (P_A, 1), creates the message and en-queues the data packet onto the global queue. Now, when the consumer entity, say T_2 (P_B, 3), attempts to acquire a message it does so by de-queueing from the global queue.

As seen in (Fig 5.9, 1), T_1 (P_A, 1) begins to run on Core_A and first acquires a shared memory block by calling the G_OSMemGet() and populates this memory block with the required data. Then this tasks calls G_OSQPost() along with the pointer to this memory block and the pointer to the global queue data structure (Fig 5.9, 2).

Within this call first it is checked if any task either on Core_A or Core_B is waiting on this message. This is done by checking the Q_Event_A.OSEventGrp and Q_Event_B.OSEventGrp values (Fig 5.9, 3). If these value are zero, which applies in this case, it indicates that no task on either core is waiting for the message.

It then checks if the queue is full by comparing the Q_Manager.OSQEntries with the Q_Manager.OSQSize (Fig 5.9, 3). On comparing if it is seen that the queue is full it return from the function call with an error message indicating that the queue is full (OS_ERR_Q_FULL). However, if the queue is not full, it then adds the pointer to the memory block at the OSQin index within the global queue (QMsgTbl[ ]) and then updates the other queue management data structures indicating an en-queue. After this the task return back from the G_OSQost function call and continues to run (Fig 5.9, 5).

As shown in the figure, let us assume T_1 (P_A, 1) again goes through this process of en-queuing of a message pointer onto the global queue without any tasks waiting on the message (Fig 5.9, 6 to 9).

Now, after two messages have been en-queued, task T_2 (P_B, 3) begins to run and attempts to acquire a message from the global queue (Fig 5.9, 10). For this it calls the G_OSQPend() with the pointer to G_Queue1 as the parameter indicating the queue from which it intends to retrieve the message and the timeout period (Fig 5.9, 11). Within this call it first checks if the call has been made from within an ISR and if so it returns back with an error message OS_ERR_PEND_ISR. This is important since blocking for a message within an ISR is not permitted. To acquire a message from within an ISR the G_OSQAccept() non-blocking API needs to be called. More about this would be discussed in the upcoming cases.

After performing the necessary checks, it then check to see if there are any message entries within the global queue. This is done by checking the Q_Manager.OSQEntries (Fig 5.9, 12). Here, if the value indicates zero then no message is present in the queue and the task needs to be put into the waiting state. However, since in this case 2 message entries are present within the global queue the Q_Manager.OSQEntries would indicate a value of 2. It then increments Q_Manager.OSQOut and decrements the Q_Manager.OSQEntries

indicating a dequeue (Fig 5.9, 12). After doing so it returns back to the T_2 (P_B, 3) along with the pointer to the memory block which was at the head of the queue (Fig 5.9, 13).

After task T_2 (P_B, 3) has completed its work with the acquired memory block it must ensure that the memory block is returned to the global shared memory pool by calling OS_MemPut()(Fig 5.9, 14).

Case 2: Consumer Tasks on Both Cores Pend Before Producer Enqueues:

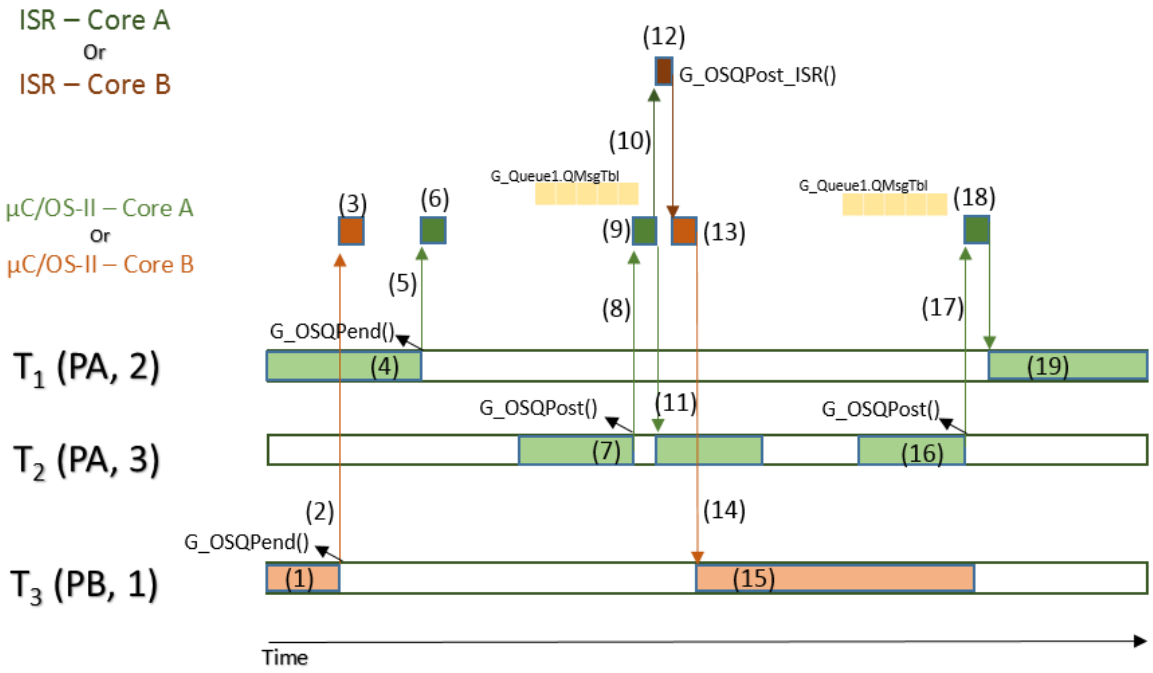


Figure 5.10: Global Queues Case 2: Consumer Tasks Pend Before Producer Enqueues

Using this scenario, the task release patterns of the tasks which are already blocked waiting for the message is described in detail. In this case more than one consumer tasks, say T_1 (P_A, 2) and T_3 (P_B, 1), try to acquire a message but get blocked. Following this when the producer task, namely T_2 (P_A, 3), begins to generate the required message the blocked tasks are made ready to run based on their priority rather than their time of arrival. Here it is important to note that since tasks are already waiting on the availability of a message,

the μC/OS-II implementation does not en-queue the packet, and instead directly posts the message onto the tasks TCB (at *OSTCBMsg within the tasks TCB) and then puts the task onto the ready queue. The same approach has been adopted in the global queue implementation.

As seen in (Fig 5.10, 1), first T_3 (P_B, 1) begins to run and tries to acquire a message from the global queue. For this it calls G_OSQPend() along with the global queue data structure and the timeout period as parameters (Fig 5.10, 2). Within this call, it first check to see if the function has been called from within an ISR. Next, it validates the ECB data structures and then continues on to check if there are any messages available on the global queue by checking the Q_Manager.OSQEntries value (Fig 5.10, 3). Since in this case no message has been produced previously, it would indicate a value of zero. At this point it is required that T_3 (P_B, 1) be blocked, for the timeout time specified as the parameter, waiting for a message to be deposited to its task control block (Fig 5.10, 3).

To put T_3 (P_B, 1) in the waiting state (i.e. being taken off the ready list) it first sets the OSTCBStat value as 0x04 indicating that the task is pending on a message from a queue. It then sets the OSTCBStatPend value within the TCB of T_3 (P_B, 1)'s as zero indicating that the task needs to be put into the pending state. Finally the OSTCBDly value within the task's TCB is set the timeout value provided as one of the parameters.

After setting the required task TCB values the task status is changed to waiting by calling the native OS_EventTaskWait() function. Also now the ECB for Core_B, i.e. Q_Event_B is updated to reflect that T_3 (P_B, 1) is in the waiting state (Fig 5.10, 3).

Now, task T_1 (P_A, 2) also requests for the message from the shared memory queue by calling the G_OSQPend()(Fig 5.10, 5). Here, similar to the steps done as a part of the G_OSQPend() for T_3 (P_B, 1) the Q_Manager.OSQEntries are checked to see the availability of a message (Fig 5.10, 6). Here again the value would indicate zero, since no message has yet been en-queued. Then the TCB of T_1 (P_A, 2) is also updated by setting OSTCB-

Stat value as 0x04, OSTCBStatPend as 0 and finally setting the OSTCBDly value as the timeout value passed as a parameter within the G_OSQPend() call. After doing so the state of this task is also changed to waiting and the same is also reflected within the ECB for Core_A, i.e. Q_Event_A (Fig 5.10, 4).

After T_1 (P_A, 2) and T_3 (P_B, 1) have attempted to acquire a message and have consequently gotten blocked, task T_2 (P_A, 3) begins to run on Core_A. It first acquires a shared memory block by calling the G_OSMemGet() and populates this memory block with the required data. Then this tasks calls G_OSQPost() along with the pointer to this memory block and the pointer to the global queue data structure (Fig 5.10, 7 to 8).

Within this call first it is checked if any task either on Core_A or Core_B is waiting on this message (Fig 5.10, 9). This is done by checking the Q_Event_A.OSEventGrp and Q_Event_B.OSEventGrp values. Here both values would hold 0b00000001 indicating that tasks ranging from priority 0 to 7 are in the waiting state for the queue related event (which is the post of a message by the producer entity) on Core_A and Core_B.

Now, with the help of the OS_NxtTask utility function created as a part of this global queue implementation the priority value of the highest priority task waiting on this queue related event is retrieved from both Q_Event_A and Q_Event_B ECBs. After retrieving these priorities it is then seen which of the tasks on either core are of a higher priority (Fig 5.10, 9). In this case it would be observed that the task T_3 (P_B, 1) on Core_B is of a higher priority than task T_1 (P_A, 2) on Core_A. Based on the priority based implementation of the global queue mechanism task T_3 (P_B, 1) should get the first message produced.

Since it is not possible to directly post onto the TCB of T_3 (P_B, 1) from Core_A, with the help of inter-processor interrupt the other core is notified of this message. However, before raising an IPI the pointer to the message is saved within the G_Queue_1.TCBmsg variable (Fig 5.10, 9).

Then Qmarkers.QmarkerList[index] value is changed from 0 to 2 indicating that a message is pending for a task on Core_B. Here, the index value is retrieved by referring to the G_Queue1.Q_index value which was set in the G_OSQCreate() call in the initial setup. Along with this, the G_Queue_1.QmarkerCount value is also incremented to indicate that one of the global message queues (i.e. G_Queue1) has a pending message (Fig 5.10, 9). After the above mentioned data structures have been updated an IPI is raised from Core_A to Core_B (Fig 5.10, 10). After the IPI has been raised task T_2 (P_A, 3) returns back and continues to run (Fig 5.10, 11).

On Core_B, as a part of the ISR to handle the inter-processor interrupt (Fig 5.10, 12) sent by Core_A all the indices within the Qmarkers.QmarkerList[] are checked to see if any of them hold the value 2. Since at index 0 the value is found to be 2, it indicates that a message on a global queue is meant for a task on Core_B. With the retrieved index value the it is now possible to get global queue data structure from the G_Q_Global_List[index] (Fig 5.10, 12).

It is important to note here that if an IPI would have been raised from Core_B to Core_A, then within the ISR running on Core_A all the indices within the QmarkerList[] would be checked to see if any of them hold the value 1.

Now, from within the ISR the G_OSQPost_ISR() is called along with the global queue data structure as the parameter (Fig 5.10, 12). This call to G_OSQPost_ISR(), essentially behaves like a proxy function for the G_OSQPost() done by task T_2 (P_A, 3).

Within the G_OSQPost_ISR() function (Fig 5.10, 12) again with the help of the OS_NxtTask both the OS_Event_A and OS_Event_B are checked to see which of the cores has a task of higher priority waiting on the message. Here, it would again see that a task on Core_B is of higher priority, i.e. task (T_3 (P_B, 1). Since this task is local to Core_B it then sets the value at index 0 within the Qmarkers.QmarkerList[] as 0, indicating that the message has been received on Core_B and no pending message remains to be handled for this

global queue. Following this it calls the native μC/OS-II OS_EventTaskRdy() function to change the state of (T_3 (P_B, 1) from waiting to ready. As a part of this function call the message is also passed, which was retrieved from G_Queue1.TCBmsg within the global queue data structure. This message is then placed on (T_3 (P_B, 1) task's OSTCBMsg which is a data structure dedicated to hold messages sent to tasks directly in the implementation of μC/OS-II message queues.

When the ISR completes all of the above mentioned steps it exits and calls μC/OS-II again i.e. the OSIntExit function where checks to see if a higher priority task is now ready to run or should the context of the task previously running be restored (Fig 5.10, 13). In this example it is assumed that T_3 (P_B, 1) is of higher priority and begins to run (Fig 5.10, 15).

Now, task T_2 (P_A, 3) creates another message to be en-queued on the global queue by calling G_OSQPost() (Fig 5.10, 16).

Here, within the G_OSQPost call again it checks to see if any task either on Core_A or Core_B is waiting on this message (Fig 5.10, 18). At this instant only Q_Event_A.OSEventGrp would hold 0b00000001 indicating that one or more tasks on Core_A ranging from priority 0 to 7 are in the waiting state for the queue related event (which is the post of a message by the producer entity). At this instant only task T_1 (P_A, 2) is waiting on the message to be deposited. Since this task is local to Core_A it directly places the pointer to the message onto the TCB (.OSTCBMsg) of T_1 (P_A, 2) and changes the state of the task to ready by calling the OS_EventTaskRdy() native μC/OS-II function. After doing so T_1 (P_A, 2) begins to run since it is of a higher priority than T_2 (P_A, 3) (Fig 5.10, 19).

Case 3: ISR to ISR communication:

In this case let us assume the producer entity is an ISR, namely ISR_A on Core_A and the consumer entity is another ISR, namely ISR_B on Core_B. Also, let us assume that no other actors are present in this setup so as to make this discussion simpler.

When ISR_A gets triggered owing to an interrupt it calls the G_OSQPost() API to en-queue the message onto the global queue. Since no other tasks are pending for a message on either core, this message gets en-queued as shown in Case1.

Now, when ISR_B begins to run owing to an interrupt on Core_B it tries to acquire the message suing the G_OSQAccept() API. As was mentioned in Case1, if the G_OSQPend() call is placed within an ISR it returns back with an error message OS_ERR_PEND_ISR. The reason for placing such a restriction is because it is not possible to block within the context of an ISR.

For this reason μC/OS-II provides the OSQAccept API with which an ISR can attempt to get a message if available. This same functionality has been extended for shared memory messaging queues as well.

In this call, first the Q_Manager.OSQEntries is checked. If this value is greater than zero, it indicates that a message is available within the global queue. Then the message at the head of the queue is de-queued as in the case of G_OSQPend() and the OSQOut variable is updated.

However, if there is no message available the function call return immediately with a NULL pointer as the message indicating that no message was available on the global queue.

5.8 Task Model for μC/OS-II on Simulink embedded coder

SIMULINK provides task blocks for VxWorks, Windows, Linux and other operating system, however for μC/OS-II no such task exists. Having a task block to model the system in the form of independent tasks can be a very useful feature for designers who prefer

a task view of the system. The intended purpose of the task block can be broken into the following and a brief description of how some of this functionality can be achieved would be given here. In the upcoming sections of implementation a pseudo-code to highlight the process execution of these tasks would be given.

As a part of this implementation there are 2 capabilities offered for the generation of such a task. A simple task which runs on the fixed priority scheduling policy and the other allows the tasks to be created following the rate monotonic scheme of scheduling with an associated timer driven wake up. The details of the two task creation capabilities using this μC/OS-II task block is given below:

1. The custom made μC/OS-II task block is connected to a function call subsystem, for which when the code is generated an independent μC/OS-II task is created using OSTaskCreateExt. All of the blocks present within the down-stream function-call subsystem are in-lined within the task body. The code added to this sub-system is wrapped into an infinite while loop, like any typical real-time task. As parameters the user needs to provide only the task name, priority and stack size. Depending on the complexity of the task, the stack size can be changed to accommodate more data structures.

2. If the user wishes to create a timer driven task then by selecting the appropriate option in the block parameter window an option to enter the time-out period of the timer along with any initial offset that needs to be set. The timer functionality is implemented using the OSTmrCreate API. For timers to be enabled a timer call back function also is created and a dedicated semaphore is released with the help of OSSemPost. This allows the corresponding task, which is blocked on the OSSem-Pend function call, to be woken up on the triggering of the timer. This then allows the task to run one iteration till it again waits for the timer.

An important point to note here is that the task and timer creation API calls are done within the model_initialize function called from main. These tasks created using this block relies on the initialization of the μC/OS-II services beforehand. Following is the explanation of the parameters passed to the task blocks. These blocks are also shown within Fig 5.11.

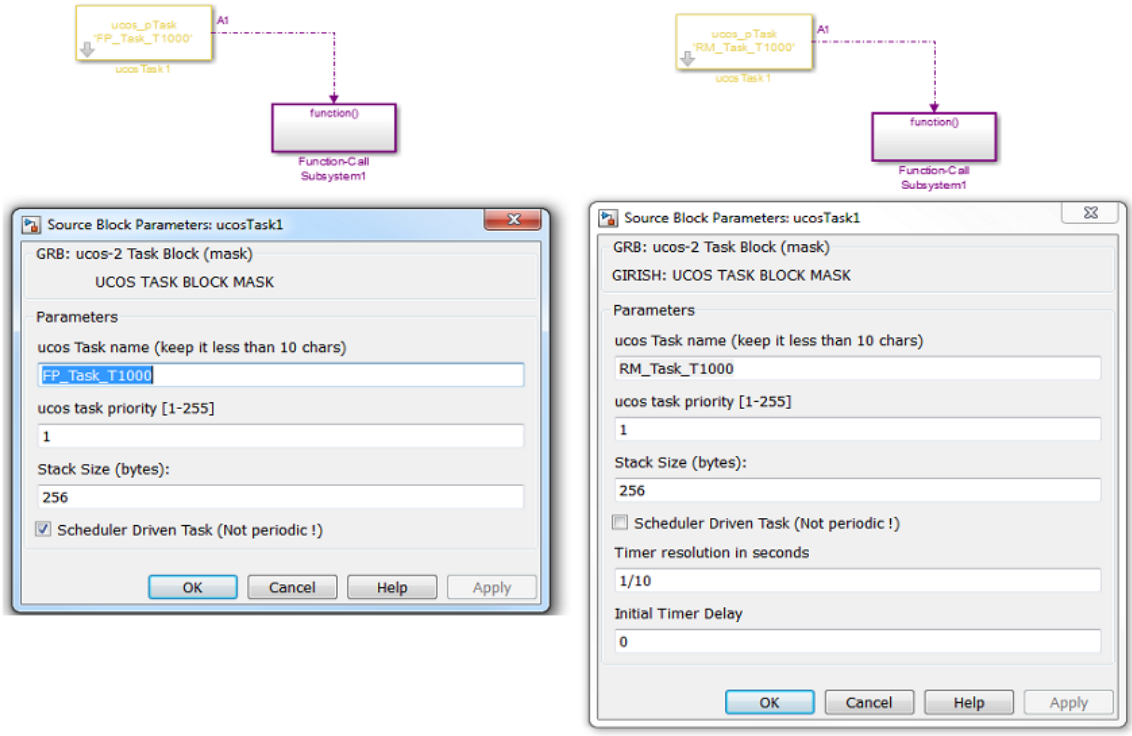


Figure 5.11: μC/OS-II Task - SIMULINK Block

5.9 Multirate Model for μC/OS-II on Simulink Embedded Coder

SIMULINK provides code generation support for models with multiple rates. In its simplified form the pseudo-multitasking approach is followed wherein one task is responsible for calling in the various model_step functions corresponding to the sample rates as and when required. The Real-time Workshop provides support for code generation

running on a RTOS, such as VxWorks. The code generation scheme followed here can be classified as truly multi-tasking since each of the model_step functions, corresponding to the sub-rates, are called within separate tasks. However, since the execution of the sub-rate tasks again relies on the task running at the base rate any overruns or errors in this task can cause the entire system to break-down. To avoid this, the multi-rate framework developed for μC/OS-II is, in its true sense, completely multi-tasked. Here each task, apart from the need for inter-task communication, are independent of each other. With the help of the following pseudo-code the execution process for the custom μC/OS-II framework can be described. The necessary TLC script changes and the precise nature of the code generated would be discussed within the implementation section.

5.9.1 Initial Setup for Multi-rate Code Generation

To support this multi-rate framework timers need to be enabled as mentioned within the Time Management section of this document. However there are a few key points that are worth mentioning regarding the setup of timers again. The timer service provided by μC/OS-II is managed with the help of a timer task OS_TmrTask. This is an internal task created when timers are enabled. To improve the responsiveness of the tasks the timer task has been assigned a priority higher than all of the tasks created as part of the code generation process. This is done by setting the OS_TASK_TMR_PRIO macro within the app_cfg.h file. This ensures that as soon as a timer interrupt is raised- the timer task is allowed to run.

The data structures required for the creation of the tasks, timers and semaphores are also declared within the global section of the code. These data structures would be used in the code present within multirate_ucos_init function described below.

5.9.2 Multi-rate Process Execution Algorithm

Since this code generation framework relies on the initial setup of the μC/OS-II kernel, it does not contain a main function. Instead the task, timer and semaphore setups are done within an initialization function, namely multirate_ucos_init. This function is called from within an already running task in the underlying AMP μC/OS-II setup, named as AppTaskCreate within this project. The priority of this task is the second highest in the system only next to the priority of the timer task.

As a part of this initialization function first the timer values, or the sample times for each of the tasks need to be specified. Since a monolithic structure is not followed it is not necessary that the sampling rates need to be integer multiples of the base rate task. These timer values need to be assigned within an array, named as timer_rates[]. Each entry within this array corresponds to each of the task within the system. The first entry belongs to the task running the model_step function for the fastest sampling rate and so on (Fig 5.12, Line 1).

Next, depending based on the number of rates present within the mixed-rate model, the process of creating the timers, tasks and the initialization of task release semaphores is done (Fig 5.12, Line 2). During this the error conditions are also checked to ensure proper creation and initialization of the tasks, timers and semaphores.

Now that the required tasks as well as the timers have been created, to ensure that all tasks start with zero initial offset first the interrupts are disabled (Fig 5.12, Line 3) and then each of the created timers are started with the OSTmrStart function call. Since during this period the OS timer ticks are not incremented there is no skew in terms of timer start times. After all of the timers have been started the interrupts are enabled again.

After doing so the system is ready and the tasks are present within the read-list. Since the program is still running in the context of the AppTaskCreate task the other tasks are

not yet able to run. Next, the AppTaskCreate blocks on the error detection semaphore (Fig 5.12, Line 4). Every time an error is detected, such as an overrun, the task responsible for the sets an error flag and error releases a semaphore on which the AppTaskCreate task is blocking. Then the AppTaskCreate function deletes the timer for the task, deletes the task and finally also deletes the task release semaphore within the timer callback function (Fig 5.12, Line 5 to 7). This behavior is optional and specific to this implementation and can be modified as required by the application. Care needs to be taken before the deletion of a task, all resources held by the task needs to be released. The AppTaskCreate function then again waits on the semaphore for error detection by other tasks. It continues to do so till there are no other task is available within the system and then finally returns itself.

The task body for each of the task, like any typical real-time task, contains an infinite loop. At the beginning of each of these loops the task blocks for the task release semaphore. After, the task gets the semaphore to start execution, it sets the overrun flag as TRUE. Following this the model_step function corresponding to the task is called. After the model_step function returns as a last step of the task body the overrun flag is set to FALSE.

As a part of timer call back function associated with each task, first an overrun condition is checked. This is determined by the status of the overrun flag. If it is TRUE, when checked from within the timer call back function, an overrun is indicated and the required error semaphore is released waking up the AppTaskCreateTask task. This check for an overrun is performed by disabling interrupts and then on completing the check, interrupts are enabled again. If no overruns are detected then the task release semaphore is released allowing the task to run one iteration of the task body till it blocks again for the next release of the task release semaphore.

```
multirate_ucos_init()
{

  Assign the real-time sampling rates to each rate                              (1)

  for each sub-rate task (and also the base rate task)                          (2)
        Initialize the task release Semaphore
        create a timer with the required sampling rate and connect call-backs
        Spawn a task with priority corresponding to Rate Monotonic Scheme
        check for errors in creation of semaphore, timer and task
  end for


  Disable all interrupts                                                        (3)
  for each sub-rate task (and also the base rate task)
        Start the timer associated with the task release                        (4)
  end for
  Enable all interrupts

  While(model tasks exists in system)                                           (5)
    Block on semaphore for errors                                               (6)
    If (sub-rate task i has error)                                              (7)
      Delete timer for task
      Delete task
      Delete Task release semaphores
    EndIf
  EndWhile

  Shutdown
}
```

Figure 5.12: μC/OS-II Multi-Rate Model Initialization

5.10 Dynamic Buffering Protocol-Single Core RT SIMULINK Block

With the multi-task implementation described in the previous section, where instead of a single tasks there exist multiple tasks which are triggered by the timer events, the semantics of task execution are preserved as long as there are no inter-task communications. However, like any typical real-time system, communication between tasks of different periods becomes imperative in complex designs. In order to preserve synchronous semantics SIMULINK provides rate transition blocks which provide close to ideal solutions. However in certain situations do not offer deterministic behavior as described in the earlier sections[9]. The work proposed within [9], namely the Dynamic Buffering Protocol, of-

fers lock-free inter-task communication protocol which also preserves ideal synchronous semantics.

Unlike the approach adopted for SIMULINK's rate transition blocks, where the pointer are manipulated only at execution time of either the producer or consumer task, in DBP pointers are manipulated at the release of task. This release event in the multi-rate implementation described within this document is the expiry of a timer, which in-turn executes the timer-call back function. Within this timer call back function the task release semaphore is also released allowing the task to run. This implementation based on DBP works with the Earliest Deadline First and Static Priority Scheduling as well [9].

Even though the DBP [9] works with any arrival pattern and the general case is applicable for the one writer and multi-reader scenario as well. Following are the assumptions that are made owing to the implementation of the DBP protocol itself and some due to the scope of the implementation within the project.

DBP Assumptions:

1. This protocol works on the basic assumption that no task overruns occur.

2. When both reader and writer tasks are released at the same time, then the writer actions at release time are performed first and then the reader actions are performed.

Project Level Assumptions:

1. The case of one-writer and multiple readers can be treated as a case of multiple writer-reader pairs. This assumption, though semantically correct, might not be optimal in terms of memory usage.

2. The creation of tasks and timers for the multi-rate system assumes that there is no initial offset in the start of the timers.

Based on the above assumptions the 3 cases considered in the DBP protocol have also been implemented within this project. These cases are as follows:

Case 1: Low priority producer and high priority consumer.

Case 2: High priority producer and low priority consumer.

Case 3: High priority producer and low priority consumer with unit delay.

5.10.1 Data Structures Required

Inter-task communication through DBP relies primarily on shared buffers between tasks and a pointer manipulation logic to control this communication protocol. Based on the case, the size of the buffer required would vary.

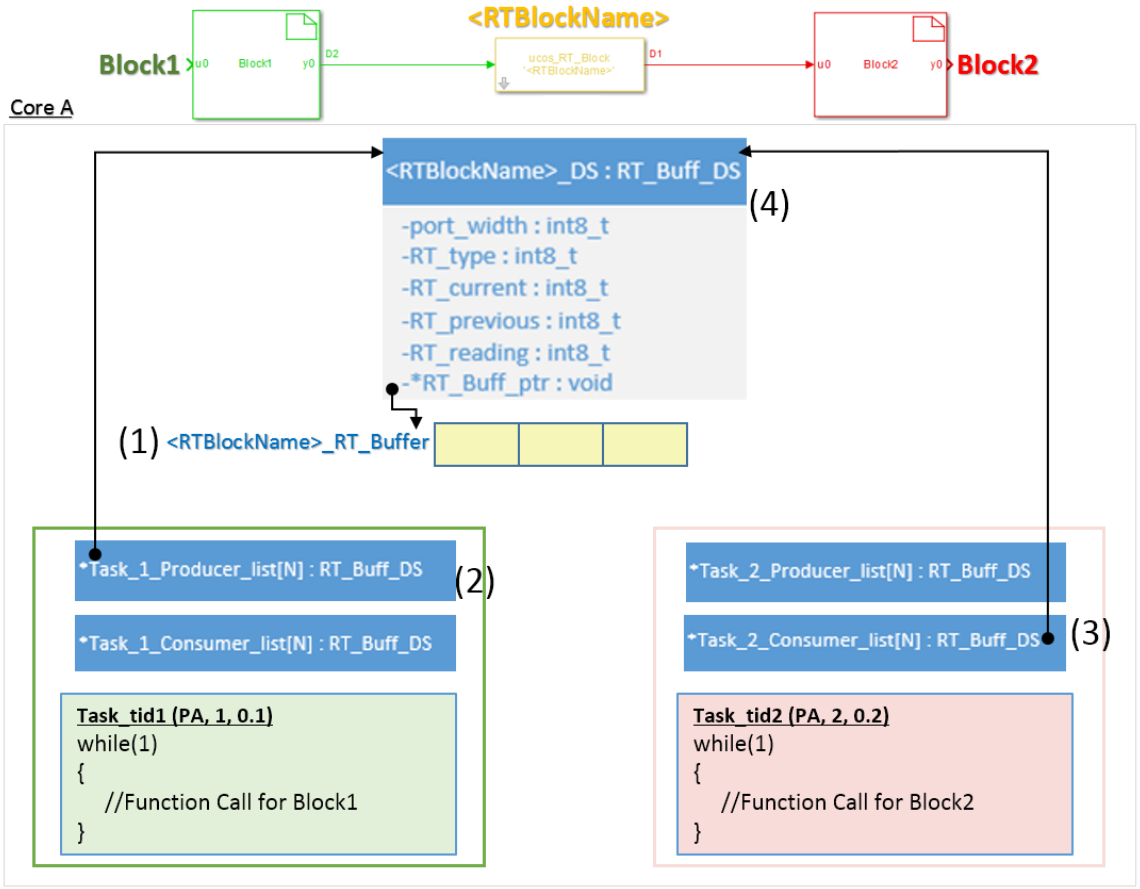


Figure 5.13: DBP based SIMULINK RT Block- Data Structures Required

In the case of communication between tasks with low priority writers to high priority reader i.e. case 1, and in the case of high priority writer to low priority reader without unit delay, i.e. case 2, a double buffer is required. In case 3, with high priority writer and low priority reader with a unit delay between tasks, triple buffers are used. Each entry within these buffers have to be wide enough to accommodate the data on the output port of the producer blocks. Say for instance if the producer outputs 4 data values of size double each time it runs then, each entry within the buffer should also be able to hold 4 doubles. Let the buffer be named as <RTBlockName>_RT_Buffer[][], where <RTBlockName> is the name of the DBP based Rate transition block in the SIMULINK model (Fig 5.13, 1).

To maintain these buffers, 3 additional data structures have been created, namely <RT-BlockName>_RT_Buff_DS, Task_<tid>_Producer_list and Task_<tid>_Consumer_list. The last 2 list data structure are arrays of the type RT_Buff_DS. Here <tid> represents the task id of the task to which the block functionality would be assigned to as explained in the multi-rate code generation process in the earlier sections.

Since a task can play the role of the writer in one task pair and a reader in another task pair, 2 data structures are maintained per task. Let us consider, as an example, the task created for blocks running at one sampling rate higher than the base rate, then tid = 1. If this task is a writer in a reader-writer pair, then the data structures responsible for the management of the <RTBlockName>_RT_Buffer[][] is present within the Task_1_Producer_list[] associated with this task (Fig 5.13, 2).

Now, if the reader, in the reader-writer pair, has a sampling rate two sampling rates higher than the base sampling rate, then tid = 2. The data structures responsible for the management of the <RTBlockName>_RT_Buffer[][] is present within the Task_2_Producer_list[] (Fig 5.13, 3).

A point to note here is that for each reader-writer pair, an entry for the buffer management data structure is made within two lists, one for the reader on its consumer list and one for the writer task on its producer list.

<RTBlockName>_RT_Buff_DS of type RT_Buff_DS is a data structure which holds the address of the buffer associated between the producer and consumer tasks and other data structures required for the implementation of the DBP protocol (Fig 5.13, 4). The constituents of this data structure along with their function is given below.

- **RT_type:** Based on the type of transition, i.e. high to low or low to high, a unique identifier (1, 2 or 3) is assigned to this variable.
- port_width: This structure also holds the port width to determine the number of elements present as the output from the writer task.
- **RT_Buffer_ptr:** This data structure holds the pointer to the <RTBlockName>_RT_Buffer[][].
- **RT_current:** In Case 1 and Case 3, this variable holds the buffer index to which the writer task is currently writing to or last wrote to.
- **RT_previous:** In Case 1 and Case 3, this variable holds the buffer index to which the previous occurrence of the writer deposited the data into.
- **RT_reading:** In Case 3, this variable holds the buffer index from which the reader task is currently reading data from, or would read data from in the immediate future.

5.10.2 Intra-core Communication Protocol Algorithm

The discussion of the algorithm can be done with the help of taking a typical example of each of the cases. To explain the various aspects of the algorithm let us us assume that there are 2 tasks on core PA, say T_W (P_A, 2) and T_R (P_A, 1) for case 1. For case 2 and 3 let the tasks be T_W (P_A, 1) and T_R (P_A, 2). For the sake of discussion, let us

call the <RTBlockName>_RT_Buffer[] as just RT_Buffer and the data structures within <RTBlockName>_RT_Buff_DS directly by their variable names.

Low priority producer and high priority consumer (RT_type = 1):

Here it is assumed that a unit-delay exits between the low priority writer T_W (P_A, 2) and the high priority reader T_R (P_A, 1). In this case the writer maintains the RT_Buffer and the RT_current value. On the other hand the reader maintains the RT_previous value and updates it as and when required.

Within the model_initialize function the data structures are first initialized. This is done for all the DBP transition blocks present within the system. Since the transition in this example is that of Low to High, the initialization involves setting the RT_current = 0. Also, the entire buffer is initialized with the default value 0. (Fig 5.14, Line 1 to Line 3).

As mentioned in the data structure setup above, the data structures responsible for managing the RT_Buffer would have to be associated to both the reader and writer tasks. This is done by assigning the pointer of the <RTBlockName>_RT_Buff_DS to the reader T_R (P_A, 1) task's Task_<tid>_Consumer_list[]. Similarly, the pointer to the <RTBlock-Name>_RT_Buff_DS is assigned the writer T_W (P_A, 2) task's Task_<tid>_Producer_list[] list (Fig 5.14, Line 4 to Line 5).

The key distinguishing factor of this algorithm from SIMULINKs rate transition blocks is that pointers are manipulated during task release times. However, since this algorithm assumes that there are no overrun conditions in the task execution, this needs to be verified before any other modifications can be done to the pointers (Fig 5.15, Line 6).

As mentioned before as a part of the assumptions of the DBP protocol, if both the reader and writer task get released at the same time, then the writer side release time activities need to be carried out first. Also, the current multi-task code generation frame-

```
model_init()
{
  for each RT_DBP block in model                (1)
    Initialize the elements of RT_Buff_DS       (2)

    Set the RT_Buffer[][] with initial value    (3)
    Initialize the Task_<tid>_Producer_list[]   (4)
    Initialize the Task_<tid>_Consumer_list[]   (5)
 end for

}
```

Figure 5.14: DBP Based Rate Transition Block: Model Initilzation

work for Multi-rate models presented within this project starts the timers in the order of the priority of the tasks. Hence, in the event that a low priority and a high priority task get released at the same time, then the high priority task's timer call back function is called first. Now keeping in mind both these design aspects, in the case of Low to High transitions, where the reader task gets released first, the reader task in its timer call back function does the writer side activities corresponding to RT_Type = 1 (as specified by the DBP protocol). Correspondingly when the writer gets released, presumably after the release of the reader, it performs the reader side release time activities within its timer call back function. This reversal in roles during the task release function, only for RT_Type 1, ensures that the writer side activities are always performed before the reader side activities.

Within the case being considered here, when the reader task T_R (P_A, 1) gets released the Task_<tid>_Consumer_list[] would contain the pointers of all the DBP Rate Transition Blocks where it behaves like a reader. Since it reads the value of RT_Type as case 1 it toggles the RT_current value (Fig 5.15, Line 11 to Line 12). Note, as mentioned before, this is the writer side logic performed by the reader.

Similarly, when the writer task is released, all of the DBP Rate Transition associated to it need to be updated. The Task_<tid>_Producer_list[] contains pointers for all the DBP Rate Transition blocks where the task acts like a writer. Based on the RT_Type

```
Timer_CallBack_Func()
{
    check for task overrun condition                          (6)

    for each entry within Task_<tid>_Producer_list[]          (7)
        If RT_Type == 1                                       (8)
            RT_previous = not RT_current

        ElseIf RT_Type == 2                                   (9)
            if  (RT_current == RT_next)
                RT_next = not RT_next

        ElseIf RT_Type == 3                                   (10)
            RT_previous = RT_current
            RT_current = x  [0,1,2] such that
                    (x != RT_previous and x!= RT_reading)
        EndIf
    end for

    for each entry within Task_<tid>_Consumer_list[]          (11)
        If RT_Type == 1                                       (12)
            RT_current = not RT_current

        ElseIf RT_Type == 2                                   (13)
            RT_current = RT_next
        EndIf

        ElseIf RT_Type == 3                                   (14)
            RT_reading = RT_previous
        EndIf
    end for
}
```

Figure 5.15: DBP Based Rate Transition Block: Timer Callback Function

value from within the entries of this list, which here is Case1, the appropriate changes are made to the pointers. Since here in this case since RT_Type is equal to 1, within the timer call back of the task T_W (P_A, 2) RT_previous is set as the toggled value of RT_current (Fig 5.15, Line 7 to Line 8). (i.e. if RT_ current = 0 then RT_previous is set as 1 and vice versa). These are the reader side activities, as mentioned within the DBP protocol.

Now, when the tasks begin to run within the context of their task body the actual data points are transferred between tasks. Once the writer block, present as a function within the T_W (P_A, 2), has produced the required data, it then copies it to the buffer at the index value held by RT_current (Fig 5.16, Line 15).

Similarly, the reader block, present as a function call within T_R (P_A, 1) task body, copies the data present within RT_Buffer at the RT_previous index (Fig 5.16, Line 17).

An important aspect to note here is that, since the reader task is of a higher priority, it would execute first owing to rate monotonic scheduling semantics. In the first iteration of this logic when the writer has still not produced the data points the reader reads the default values set in the buffer at the time of initialization.

```
model_step_Writer()
{
    ...
    If RT_Type == 1 or RT_Type == 3
    RT_Buffer[RT_current][port_width] =                    (15)
                producer_outport[port_width]
    ElseIf RT_Type == 2
    RT_Buffer[RT_next][port_width] =                       (16)
                producer_outport[port_width]
    ...
}

model_step_Reader()
{
    ...
    If RT_Type == 1                                        (17)
        consumer_inport[port_width] =
                RT_Buffer[RT_previous][port_width]
    ElseIf RT_Type == 2                                    (18)
        consumer_inport[port_width] =
                RT_Buffer[RT_current][port_width]
    ElseIf RT_Type == 3                                    (19)
        consumer_inport[port_width] =
                RT_Buffer[RT_reading][port_width]
    EndIf
    ...
}
```

Figure 5.16: DBP Based Rate Transition_Block: Reader Writer Functions

High Priority Producer and Low Priority Consumer (RT_type = 2):

The data structures RT_current and RT_next are initialized to zero (Fig 5.14, Line 2).

At the release time of the writer, which in this case is T_W (P_A, 1), the RT_next value is toggled only if the RT_current is equal to RT_next (Fig 5.15, Line 9). At the release of the reader, i.e. TR (PA, 2), the RT_current value is set equal to the value held by RT_next (Fig 5.15, Line 13).

During execution time, while in the task body T_W (P_A, 1) deposits the data into RT_Buffer at the index held by RT_current (Fig 5.16, Line 16). However within the task T_R (P_A, 2), the values are read from the RT_Buffer at the index held by RT_reading (Fig 5.16, Line 18).

High Priority Producer and Low Priority Consumer with Unit Delay

(RT_type = 3):

The data structures RT_current, RT_reading and RT_previous are all initialized to zero (Fig 5.14, Line 2).

At the release time of the writer, which in this case is T_W (P_A, 1), the RT_previous value is set to the value held by RT_current. After doing so, RT_current value is set to a value other than that held by RT_reading and RT_previous (Fig 5.15, Line 10). On the release of reader task, i.e. TR (PA, 2), the RT_reading value is set equal to the value held by RT_previous (Fig 5.15, Line 14).

During execution time, while in the task body T_W (P_A, 1) deposits the data into RT_Buffer at the index held by RT_current (Fig 5.16, Line 15). However within the task T_R (P_A, 2), the values are read from the RT_Buffer at the index held by RT_reading (Fig 5.16, Line 19).

5.11 Multicore Rate Transition SIMULINK Block

Data sharing between tasks managed by one kernel was discussed in the previous section. However, in an AMP setup such as the one in this project the synchronous semantics

cannot be guaranteed for tasks residing across the cores. The shared memory queues, also discussed earlier, were lock based implementations. Such an implementation is accepted as a standard to ensure that all of the data points produced by the producer are captured by the consumer. However the latest value transfer semantics only require that the reader receives the most recent non-corrupted value. The lock based approaches impose a certain degree of inter-dependence in the execution of tasks, reducing the parallelism and enforcing a certain degree of serialization of the task execution patterns [10].

The Three-slot Asynchronous Reader-writer Communication mechanism (3-ACM) designed by Chen et al [10] stores the shared data in a memory location which is accessible by the reader and writer tasks in a multi-core setup. This facility is supported in the AMP implementation of μC/OS-II, owing to the presence of shared memory accessible by both cores.

The 3-ACM Rate Transition Block, called as the Global Rate Transition Block within this project addresses the following basic requirement of any data sharing algorithm. It ensures that the data values transferred between the writer and the reader should be uncorrupted and also display correct temporal ordering. Irrespective of whether or not the reader and writer are present on the same or on different cores, the data transfer should be consistent. Again, since it is an AMP setup, there are two separate schedulers handling the tasks on each of the cores. The worst case in such a setup would be the concurrent execution of both the reader and writer tasks on each of the cores. Also, in a setup such as the one followed in this project, the behavior of the tasks on each of the kernels, such as the relative speed among the tasks and the duration of read and writes cannot be assumed. Keeping in mind the above mentioned execution semantics the 3-ACM approach would be a good solution.

Assumptions of the 3-ACM Mechanism:

1. Since communication has to be established between tasks running on two different cores, shared memory is required. Such a setup is available within this project.

2. All shared memory access, should be mutually exclusive. With the availability of the mutual exclusion primitives in this project, as discussed before, shared memory access can be secured.

3. The hardware should support an atomic compare and swap operation. This is supported by the hardware and has been extensively used in other areas of this project. This function has been called as the Atomic_CAS(mem, v1, v2). This function takes 3 parameters and ensures that the condition check as well as the data swap happens atomically. With this function, only if the current value held at the location pointed by mem and v1 are the same, the value of v2 is written into memory location mem. If it is not equal it returns back without modifying the memory location mem.

5.11.1 Initial Setup and Data Structures Required

The code generation setup for AMP presented within this thesis uses two separate models, one model to generate code for Core_A and the other model for Core_B. As show in Figure 5.17, two separate models have been created with one block in each of them. Here the block Block_PA in Core_A's model (Fig 5.17, 1) is the writer and the block Block_PB in Core_B's model (Fig 5.17, 2) is the reader. Now to be able to use this communication scheme between blocks across two models, the global rate transition block needs to be added to each of the models separately. Also, since these buffers exist in shared memory, the data buffers along with the data structures required to implement the 3-ACM algorithm need to be created beforehand in shared memory. The buffers and the

control variables are encapsulated into a single structure called RT_G_Buffer_0 of type RT_GBuff_DS (Fig 5.17, 4).

In this setup four global rate transition blocks have been statically created in memory and are presented to the user , as seen in this Figure 5.17 . Care has to be taken that on both sides, the same global rate transition buffers be selected to establish communication between the tasks.

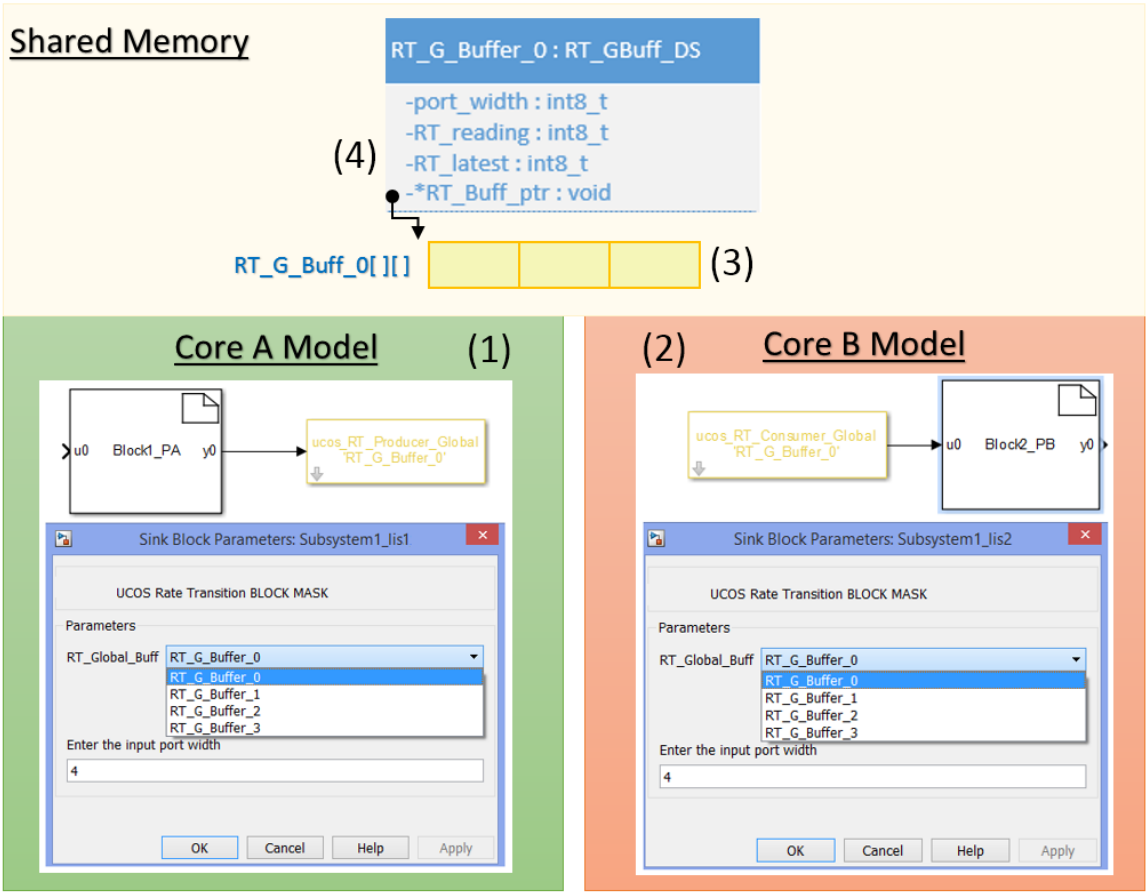


Figure 5.17: 3-ACM Global Rate Transition_Block: Model Setup and Data Structures

The data structures required to implement this 3-ACM based inter-task communication across cores are as follows.

RT_G_Buff: This is the three slot global data buffer, where the writer deposits the data and reader reads data from (Fig 5.17, 3).

RT_latest: This variable holds the index at which the latest and consistent value of the data deposited by the writer exists (Fig 5.17, 4). This can assume the value 0, 1 and 2 depending representative of the indices of the buffer. This value is initialized to a value of 0 within ppc_eabi_init.c on Core_A.

RT_reading: This variable shows the index within the buffer from which the reader is currently reading data from or the slot from which it had read the data most recently (Fig 5.17, 4). This variable can assume the values 0, 1, 2 and 3. Here RT_reading holds the value 3 when the reader is in the process of deciding which slot of the buffer it should read the data from. This value is initialized to a value of 3 within ppc_eabi_init.c on Core_A.

5.11.2 Inter-Core Communication Protocol Algorithm

For the explanation of the working of the Global Rate Transition block, some of the features of the 3-ACM algorithm need to be discussed. Firstly, the reader task can be in one of the three possible states with respect to the global rate transition logic. These three states are, updating the RT_reading value before it begins the actual read, or it is reading the buffer slot containing the data deposited by the writer or could be in the inactive state, where there is no rate transition logic being executed.

The writer task needs to determine which of the 3 slots can be written into safely. The slot that the writer finally deposits the data into should be mutually exclusive of the RT_reading and the last slot that the writer deposited the value into (held by RT_latest). To obtain this mutually exclusive value a utility function has been incorporated called NextFunction (Fig 5.18, 2). This aspect of the algorithm ensures that the reader never reads from a slot the writer is currently in the process of updating. Finally, the RT_reading can be updated by the reader as well as the writer, this is useful in the event that reader gets context switched out before it gets an opportunity to update RT_reading. By doing so

the algorithm ensures that the RT_reading value holds the buffer slot index from which the reader is going to read from.

Since the working of this algorithm does not rely on priorities or release times of tasks, owing to the lack of precedence relationship between tasks, the discussion of the algorithm would be undertaken in terms of the reader and writer side algorithms. These algorithms would work on any arrival patterns of the read and writer tasks.

Let us assume the same setup as described in Figure 5.17, where the writer block Block1_PA resides in a task T_W (P_A, 1), which is created as a part of the multi-rate code generation framework. The reader block, Block2_PB is present within a task T_R (P_B, 1), which is also created as a prt of the multi-rate code generation framework. The working of the algorithm is described in the next section.

Writer Side Algorithm:

For the internal workings of the writer side algorithms first a local variable is created to hold the temporary value. Let this variable be called next (Fig 5.18, 1). When the writer task T_W (P_A, 1) begins to execute the lines of code responsible for the inter-task communication it first determines determines a safe slot to write into based on the values held by RT_reading and RT_latest. This is done so by calling the NextFunction with RT_reading and RT_latest as parameters (Fig 5.18, 3). Within this function a safe index is retrieved based on a simple table lookup (Fig 5.18, 2). Next, the writer deposits the data into RT_G_Buff global buffer, at the index value held by variable “next” (Fig 5.18, 4). This data is the output from the block logic Block1_PA.

After the writing is complete, the RT_latest value is updated with the next value (Fig 5.18, 5). The need for the local variable becomes clear here, as the need to hold an inter-core lock for extended periods of time can be avoided. Next the Atomic_CAS function is called to update the RT_reading value atomically (Fig 5.18, 6). This update only

```
int next;  //local variable                                    (1)

int NextFunction(int x, int y)
{

    const int8_T next[4][3] = { { 1, 2, 0 },                   (2)
                                { 1, 2, 1 },
                                { 2, 2, 1 },
                                { 1, 0, 0 } };
    return next[x][y];
}

Writer_CoreA()
{
    ...
    next = NextFunction(RT_reading, RT_latest);                (3)
    RT_Buff[temp_write][port_width] =                          (4)
                        producer_outport[port_width];
    RT_latest = next;                                          (5)
    Atomic_CAS(RT_reading, 0, next);                           (6)
    ...
}
```

Figure 5.18: 3-ACM Global Rate Transition_Block: Writer side algorithm

succeeds if the RT_reading value at this point of time is 3, indicative of the fact that the reader task is currently in the process of deciding which slot to read from. This function allows the job of updating the RT_reading value to be distributed across the two tasks.

Reader Side Algorithm:

Similar to the implementation of the writer task, even on the reader side task T_R (P_B, 1) a local variable called current is created (Fig 5.19, 1). Now, since the reader block logic, i.e. Block2_PB, depends on the value from the writer block as its input, this value needs to be retrieved first from the global buffer according to the 3-ACM reader side algorithm. For this first, the RT_reading value is set to 0, indicating that the reader has started the

process of deciding which slot of the buffer to read from (Fig 5.19, 2). Next, the "current" variable is updated with the RT_latest and then it attempts to set the RT_reading value with the value held in current. This is done by again calling the Atomic_CAS function. Here, if the writer has updated the RT_reading value with the RT_latest value as shown in Figure (Fig 5.18, 6), then reader is not required to set the RT_latest and returns back from the Atomic_CAS function. In this scenario it can be assumed that the RT_reading value is as recent as the value it got at (Fig 5.19, 3). However, if the RT_reading value is zero, then the value held in current is copied into RT_reading atomically.

```
int current;                                        (1)

Reader_CoreB()
{
    ...
    RT_reading = 0;                                 (2)
    current = RT_latest;                            (3)
    Atomic_CAS(RT_reading, 0, current);             (4)
    current = RT_reading;                           (5)
    consumer_inport[port_width] =                   (6)
                              RT_Buff[current]

    ...
}
```

Figure 5.19: 3-SAM Global Rate Transition_Block: Reader side algorithm

Now, that the reader task has retrieved the index from which it needs to read from within the RT_G_Buff global buffer (Fig 5.19, 4), it goes ahead and copies the data locally and passes it as input to the Block2_PB logic (Fig 5.19, 6).

Chapter 6

IMPLEMENTATION

6.1 Porting μC/OS-II for MPC5675K in AMP Mode

Following section describes the steps involved to adapt the μC/OS-II RTOS for the MPC5675K in the AMP mode. Since the porting steps are widely documented within the μC/OS-II user guide [25, p. 313] this section only highlights some of the design changes from the available μC/OS-II ports. For execution and debugging of this port Codewarrior Development Studio for MCU version 10.5 has been used. As suggested by Micrium, the best way to obtain a port for a new MPU is by modifying an existing port from a similar processor. For this reason the μC/OS-II port for the MPC5643L MCU containing e200z4 series CPU running in LSM mode was chosen as the base port to be modified. To perform this porting, guidelines provided by Micrium have been followed to ensure that the available port can be easily adapted to the MPC5675K MCU.

The advantage of using the PowerPC architecture becomes evident with the fact that the CPU specific porting has been made considerably easy owing to the architectural similarities between the e200z4 CPU present on the MPC5643L MPU and the e200z7 CPU present on the MPC5675K MPU.

6.1.1 Project Directory Structure

The project which was developed within Codewarrior has been broken into a number of directories and sub-directories to modularize the source code into various sections. The purpose of each section and the relevant directories within them are shown in the fol-

Core_A	Core_B
app0.c	app1.c
includes.h	includes.h

Table 6.1: Folder Structure:Application Specific Code

lowing section. Each section is accompanied by a table indicating the files present within them.

An important aspect to the directory structure is that most of the software components are replicated for both cores, except for a few sections which are only present on core PA. These sections which are exclusive only to core PA would also be present within the software of core PB however the functions within them would not be utilized.

The broad classification of the software component is as follows:

1. Application Specific Code:

This is the application specific code which utilizes the μC/OS-II services exposed in the form of APIs. Also, if there is a need to perform additional peripheral initializations apart from the default initializations, such as for UART or CAN channels, they can be done here or within the startup section of the software component. The application code consists of the program entry points for both cores- main() and main_p1() for core PA and Core PB respectively.

2. μC/OS-II and μC/CPU Configuration:

The os_cfg.h file contains macros which are used to enable or disable various components of μC/OS-II such as whether or not to enable message queues, semaphores, memory management and timer management. It is also used to define certain variables and data structures which are required to be defined for the operation of the μC/OS-II code.

Core_A	Core_B
os_cfg.c	os_cfg.c
cpu_cfg.h	cpu_cfg.h

Table 6.2: Folder Structure:μC/OS-II and μC/CPU Configuration

cpu_cfg.h file contains macros to define the CPU characteristics such as the whether the CPU is running in little endian or big endian mode. It also allows us to configure the CPU timer time-stamp word size.

3. μC/OS-II Source Code:

This is the processor independent μC/OS-II source code written in ANSI C. This is where the native μC/OS-II services such as memory management, timers, queues, semaphores etc. are defined. It also contains error code macros and other data structure values (within ucos-ii.h) which would be required for the operation of the μC/OS-II services and application development.

4. μC/LIB Libraries:

These library files are the generic compiler functions such as ASCII, memory copy, string related functions. These functions are occasionally used as replacements to the standard library function that the complier provides. These functions can be fine-tuned and used as replacement for the standard compiler functions.

5. μC/OS-II and μC/CPU Processor Specific Code:

The files contained within this section are the μC/OS-II specific files which also has certain elements specific to the architecture. There are some CPU specific functionality as well which has been encapsulated in the form of μC/OS-II functions. An example of

Core_A	Core_B
os_core.c	os_core.c
os_flag.c	os_flag.c
os_mbox.c	os_mbox.c
os_mem.c	os_mem.c
os_mutex.c	os_mutex.c
os_q.c	os_q.c
os_sem.c	os_sem.c
os_task.c	os_task.c
os_time.c	os_time.c
os_tmr.c	os_tmr.c
ucos_ii.h	ucos_ii.h

Table 6.3: Folder Structure:μC/OS-II Source Code

such functions, and possibly the most important of such functions, are the enabling and disabling of the CPU interrupts. There are CPU initializations also contained within these files which are required to be done by both cores individually.

6. Board Support Package:

The files contained within BSP section are the support code responsible for the integration between the board peripherals and the operating system. It also contains certain key initializations such as the setup of the external and the auxiliary clock configurations along with the PLL setups. Also the decrementer specific to each CPU, responsible for providing the tick to the RTOS, is also setup within the initialization functions within

Core_A	Core_B
lib_ascii.c	lib_ascii.c
lib_ascii.h	lib_ascii.h
lib_def.h	lib_def.h
lib_math.c	lib_math.c
lib_math.h	lib_math.h
lib_mem.c	lib_mem.c
lib_mem.h	lib_mem.h
lib_str.c	lib_str.c
lib_str.h	lib_str.h

Table 6.4: Folder Structure:μC/LIB Libraries

the BSP section. More in detail regarding the changes would be taken up in the following section.

7. Hardware Specific and Startup Code:

Freescale has provided the library functions to access the peripherals on the MCU. Apart from that the key aspects handled within the code are the initializations during the boot-up process, the setup of entry points to the application code, setup of the interrupt vector table. Finally it also contains the linker files which define the linker file which is used during the compilation of the source code into the executable binary. The source provided by Freescale is available as a single software package but for the sake of this project the source corresponding to each core has been segregated into the appropriate projects for each core. These specific files have been provided with a suffix _p0 or _p1 indicative of the cores they belong to.

Core_A	Core_B
os_cpu_a.s	os_cpu_a.s
os_cpu_c.c	os_cpu_c.c
os_cpu.h	os_cpu.h
os_dbg.c	os_dbg.c
cpu_core.c	cpu_core.c
cpu_core.h	cpu_core.h
cpu_def.h	cpu_def.h
cpu_a.s	cpu_a.s
cpu.h	cpu.h

Table 6.5: Folder Structure:Processor Specific Code

Core_A	Core_B
bsp.c	bsp.c
bsp.h	bsp.h

Table 6.6: Folder Structure:Board Support Package

Also certain one-time initialization, such as the one contained within __ppc_eabi_init.c, are undertaken by core PA alone. Due to this reason some of the source files are not contained within the Core PB source.

Here since some of the initializations are specific to each CPU the code provided by Freeescale has been split into two projects.

8. μC/OS-II Global Functions- Additional Functionality Added:

All of the additional functionality added to the native μC/OS-II services as a part of the project have been added within these files. Since these files are not completely agnostic of the underlying memory addressing they cannot be classified into the hardware

Core_A	Core_B
Exceptions.c	Exceptions_p1.c
Exceptions.h	Exceptions_p1.h
IntcInterrupts_p0.c	IntcInterrupts_p1.c
IntcInterrupts.h	IntcInterrupts_p1.h
ivor_branch_table_p0.c	ivor_branch_table_p1.c
MPC5675K_HWInit.h	MPC5675K_HWInit.h
MPC5675K.h	MPC5675K.h
typedefs.h	typedefs.h
MPC5675K_P0_DPM_RAM_VLE.tcl	__start_p1.c
MPC5675K_RAM.lcf	MPC5675K_RAM.lcf
__ppc_eabi_init.c	
MPC5675K_DPM_Startup.c	
MPC5675K_HWInit.c	
MPC5675K_init_ram.c	

Table 6.7: Folder Structure:Hardware Specific and Startup Code

Core_A	Core_B
Global_functions_p0.c	Global_functions_p1.c
Global_functions_p0.h	Global_functions_p1.h

Table 6.8: Folder Structure:Global Functions

independent μC/OS-II code. Due to this they are considered as a separate software section.

6.1.2 Porting steps in μC/OS-II/BSP

The board support package is the code which is associated to the target board. Within this section of the software the configuration of the CPU and peripheral clock frequencies is done. Apart from this any other peripherals which might be useful such as the initialization of the watch dog timer, CAN or UART channel can also be done. The three functions within BSP where changes were made are discussed in detail below.

BSP, BSP_LowLevelInit(void)

The primary purpose for this function is to disable the software watchdog timer (SWT) module on the MPC5675K board. Since the project is still not ready for production the watchdog timer has been disabled. However with a few initializations it is possible to turn on the watchdog timer as well from within this function. The primary use of the SWT is to ensure that no system lockups happen and this is ensured by periodic interrupts sent by the SWT peripheral. If 2 consecutive timeout interrupts are not handled it leads to the reset of the system.

This function is called from within each core where each of these functions disables one of the two software watchdog timers (SW_0, SW_1).

BSP, BSP_Init(void)

This function is called only by core PA since the initializations and configurations done are not specific to each core and apply to the device as a whole. Care should be taken that only after these BSP related configurations are done should core PB be allowed to proceed further. This can be achieved with the help of inter-core barriers.

The primary functionality achieved within this function is the configuration of the clocks and enabling user run modes with these clock configurations. At power on reset of the MCU the clock source for system clock is by default set to the internal 16MHz

RC oscillator (IRCOSC). However with this BSP initialization function the system clock would be configured to run at the higher frequency of the PLLs. This function also sets up the clock tree to distribute and divide the clock sources, such as the external output clock and Auxiliary clocks, for the peripherals and buses on the MCU.

The details of the clocking setup is as follows.

- Enable user mode RUN1 and RUN0 for the device with the required clock configurations and ensure that the configurations take effect in the required run modes.
- Change the run mode to RUN1, where for the purpose of configuring the clocks PLL0 and PLL1 clocks are disabled.
- Assign the external oscillator (XOSC) as the reference clock source for the two Frequency Modulated Phase Locked Loop (FMPLL) modules on the target. Here with the help of the clock generation module the output source is selected as the 40 MHz crystal oscillator [17, p. 379]
- Obtain the outputs from the two PLLs by providing the appropriate values to the Loop Division Factor (LDF), Input Division Factor (IDF) and the Output Division Factor (ODF). Here the output clock from the PLL0, also called the system PLL, module is fed as the source for the system clock at 120 MHz. However this value can be changed by manipulating the factors.

  The formula followed [17, p. 1129] to obtain the required frequency of 120 MHz and 80 MHz for PLL0 and PLL1 respectively is:

$$\frac{\text{xosc . ldf}}{\text{idf . odf}} = phi$$

  Here to XOSC is the frequency of the external clock (selected to be 40 MHz) which is the reference clock. The IDF value is set to 6 and 8 respectively for PLL0 and

PLL1. For both PLL0 and PLL1 ODF value is set to 4. The LDF values with which the 40 MHz external oscillator signal is multiplied with is set to 72 and 64 respectively.

- Once the PLL has been setup, it is not advisable to change the clocks instantaneously to the new operating frequencies as it causing potentially un-regulated voltage spikes in device. Hence with the help of Progressive Clock Switching (PCS) the system clock is switched to the required operating frequencies gradually by stepping through the various division factors[17, p. 1130].

- After the PLLs have been configured it is required that the two run modes, namely RUN0 and RUN1, are also run with the system PLL and PLL1 enabled for them. Here an important thing to note is that the external oscillator should also be enabled when the system PLL is enabled since it is the clock source for the PLLs.

- In the process of changing the run modes care should be taken that the system be allowed to transition completely into the required mode. A timeout, in the form of a countdown decrementer, can be provided to ensure that the transition completes successfully.

- The clock generation module also generates an output clock signal as well for off-chip use. To enable this external output clock pad muxing of one of the pins has to be performed to operate in the external output clock mode. After performing the required pin muxing the clock source for this external clock is also set as the system PLL. Also this output clock has been configured with a divider of a factor of 8.

- Along with the setup of the system clock and external clocks auxiliary clocks also need to be setup. Three clocks, namely Auxiliary Clock 0, Auxiliary Clock 1, and Auxiliary Clock 2 are configured in an identical manner. These clocks would be

used by the peripherals. For instance the FlexCAN module would be using Aux Clock 2. Out of the 4 clock sources that these Auxiliary Clocks can be fed with, PLL0 is taken as the source for all three Aux Clocks. The dividers for each of the auxiliary clocks have been enabled and have been set with a dividing factor of 1.

BSP, BSP_Tick_Init(void)

This function is called from both kernels independently as they setup up the core specific decrementer and timer control registers.

Each of the e220z7 cores on the dual core MPU is equipped with a time base/decrementer counter. This decrementer is also coupled with core specific interrupt capability allowing this setup to be utilized as the clock tick used by the μC/OS-II kernel. However this decrementer needs to be initialized by providing the number of cycles to the auto-reload register. In this setup the base clock frequency has been set to the CPU execution speed, i.e. 180 MHz

Based on the number of ticks required per second set by the macro OS_TICKS_PER_SEC the number of cycle that elapse between each tick can be calculated. The decrementer auto-reload register is then initialized with this value. Next the 2 time base register, one to hold the higher and the other to hold the lower 32 bits, for each core is reset in 2 steps. The 64-bit structure provides a way to maintain the time of day and the values held by the interval timers [15, p. 65]

Next the CPU is configured, by setting the Hardware Implementation Dependent Register 0 (HID0), such that the periodic time base increments are enabled. Next within the timer control register the interrupts are enabled for the decrementer and the auto-reload functionality is also enabled.

With this setup, every time the decrementer value expires an interrupt is raised at interrupt vector number 10. As a part of the interrupt handler the tick count is incremented.

6.1.3 Porting Steps for Hardware Specific and Startup Code

As mentioned earlier, the hardware initialization package provided by Codewarrior has been split and incorporated into two separate projects, one for each core. Some of the basic initializations are still carried out by Core_A, however the core specific initializations are carried by the cores themselves through the segregated initialization code. The details of the linker files have been provided with the Memory Model section of the document. This section would highlight other important aspects of the hardware initialization required for the setup of the AMP mode of operation.

ivor_branch_table_p0.c and ivor_branch_table_p1.c

Within these files the interrupt handler or ISR need to be installed to the corresponding interrupt vector within the branch table. Both cores need to install their versions of the interrupts handlers. The handlers that are required are the OSExtIntISR, OSCtxSw and OSTickISR at interrupt vectors 4, 8 and 10 respectively. The OSExtIntISR is needed to handle all interrupts external to the CPU. These interrupts could include the software settable interrupts, CAN and UART interrupts amongst other sources.

OSCtxSw is used to handle the exceptions raised by system calls. These are used in μC/OS-II for context switch when called from within the OS_Sched() function. In the event that a higher priority tasks is available on the ready-list then an immediate context switch is required. In such an event a se_sc instruction is executed indicating a system call and as a part of the handler the context switch is done.

The OSTickISR is used when the decrementer causes an exception when the specified counter value expires and is reset again. This is essential in setting the OS ticks for μC/OS-II.

__ppc_eabi_init.c (only Core_A)

All of the pre-main initializations required such as the setup of the global semaphores etc. and other shared memory initialization can be performed here. Since it is not advisable to modify the __start.c file, which is a part of the Codewarrior standard code containing the __start function, this acts like a hook from within the __start.c allowing any pre-main initializations.

However, since the initialization here are called prior to the call to main function for Core_A only, though highly unlikely, it does not ensure that the initialization would be complete before Core_B's tasks start. Hence a barrier on both core_A and core_B becomes imperative to ensure that tasks get created and begin to execute on both cores only after all the user level initializations are complete.

MPC5675K_P0_DPM_RAM_VLE.tcl (only Core_A)

For RAM-based projects all the initializations relevant to the SRAM for both Core_ A and Core_B is done within the mpc567xK_init procedure of the debugger scripts. Through the debugger scripts the entry points for the application code is also provided for both Core_A (__startup) and Core_B (__start_p1) by setting the program counter to the appropriate addresses.

Within this script the caches are inhibited and the MMU setup is also done for Core_A by setting up a single 4GB TLB page. Also the interrupts are initialized to enable the capture of any exception during the setup of the MCU.

__start_p1.c (only Core_B)

This file contains the entry point for Core_B (__start_p1). As a part of this function which is run only on Core_B again the MMU initialization is done in a similar way as done for Core_A by setting the special purpose registers to control the MMU. Also, the interrupts are also initialized to capture any exceptions during the setup of Core_B. After the initialization is done for Core_B it branches to main_p1, which is the entry point to the application code.

6.2 Codewarrior Setup and Debugging Tools

6.2.1 Codewarrior Setup for AMP Variant Of μC/OS-II

This section of the document outlines the IDE used, which was the CodeWarrior for MCU Version 10.5. It is an integrated environment that allows us to write, compile and debug the code for our AMP implementation on the target Freescale board. It is based on the Eclipse IDE and is available both as a licensed version and a 60-day evaluation version.

Starting development on a new board with Codewarrior is not too difficult a task, since it comes with startup/initialization code for a number of supported cores/boards. In addition to this, most of the peripheral registers have been described in header files as part of the board specific package. This makes accessing peripheral registers easier when compared to directly dereferencing register addresses.

Another invaluable tool was the PEMicro USB Qorivva Multilink Interface for MPC55xx/56xx Devices, which connects the evaluation board to the host machine via USB. It connects to the board's JTAG header. This debugger allows us to step through code, pause cores individually and provides information about the state of all CPU and peripheral registers.

In the following sections, we take a look at how to import, compile and debug the code on both of the MPC5675K's dual cores.

To download the CodeWarrior IDE, please visit the link mentioned below.

http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME

6.2.2 Importing and Compiling the Code

This section assumes that the reader has a copy of the code for the μC/OS-II AMP implementation.

Following are the steps to be followed to import and compile the code.

1. The first step is to import the code into the CodeWarrior IDE. To do this, first open CodeWarrior IDE, right-click on the white space under Project Explorer and click Import.

2. Next, choose Existing Projects Into Workspace and hit Next.

3. Now, click on Browse and navigate to the directory that contains the code. Select the directory that contains the code and press OK. In the example below, the directory is CW_Documentation. Once the code has been selected, click Select All and then Finish.

4. Next, we need to ensure that the code is set to run from RAM and not Flash. To do this, select both the projects that you just imported, right-click and then click on Build Configurations > Set Active > RAM.

5. As before, select both projects, right-click and then click on Build.

6. After the build, make sure there are no errors. This can be verified by looking at the Problems tab at the bottom of the IDE.

6.2.3 Setting Core B Entry Point

1. Next, we need to find the entry point of the μC/OS-II kernel that would be running on Core B. You can find this information in the ucos-dpm-again.MAP file for the kernel.

   This file can be found in ucos-dpm-ram-b > RAM > ucos-dpm-again.MAP

2. In the ucos-dpm-again.MAP file, get the address of __start_p1.To this address add 0x2.

3. Next, we need to modify the startup script to instruct Core B to start executing code at the entry point we identified above. The startup script can be found at ucos-dpm-ram-a > Project_Settings > Debugger > MPC5675K_P0_DPM_RAM_VLE.tcl. In this file, modify the entry for the value of reg $GPR_GROUPPC and set it equal to the address of start_p1 + 0x2, which was identified above.

4. Next both the project need to be linked so that both can be downloaded onto the target. To do this, select ucos-dpm-ram-a in the Project Explorer, then click on Run > Debug Configurations

5. In the window that opens up, click on ucos-dpm-again_RAM_PnE USB-ML-PPCNEXUS > Debugger > Other Executables > Add.

6. Click on the Workspace button, navigate to ucos-dpm-ram-b > RAM, choose ucos-dpm-again.elf and click OK.

7. check the Load Symbols and Download to Device boxes and then click OK

6.2.4 Debugging the Code

To begin the debugging of the above code on the target, please follow the instructions given below.

1. Ensure that the debugger is connected to the host machine and the The target board is powered on.

2. Select the project for Core_A and start and click debug.

3. At this point the the Debug window should appear with Core_A paused at main() function and Core_B paused at start_p1().

4. To begin multi-core execution click the Multicore Resume button within the debugger window.

5. To read CPU/peripheral register values, click on Multicore Pause and then click on the Registers tab

Chapter 7

EVALUATION

7.1 High Level Testing Strategy

As described in section 1.2 the requirements were gathered over multiple iterations of the development process. Since the implementation within each iteration often relied on the work previously done, the testing strategy followed for this project also involved testing multiple inter-dependent features together. The tests presented within this section are some of the key test cases, which were used to check the basic functionality of each of the components developed over the course of the project. Apart from these high level functionality tests, many minor and unit level test cases were created in every incremental build. However documenting these tests is beyond the scope of this report.

Each of the tests described below also mention the requirements covered within the tests. Wherever possible, metrics comparing multiple approaches of implementation have been provided along with relevant test results to show proper functioning have also been provided.

The board is also equipped with 4 on-board LEDs. These have been extensively used to represent the error scenarios in many of the test cases described below.

To measure the performance of some of the features developed the system timer module has been used. In DPM mode two independent timers are made available, namely STM_0 and STM_1. This are essentially up-counters and the clock driving these counters is the system clock. The system clock has been pre-scaled by a factor of 256 to ensure that the counters do not over-flow during long tests.

Also, the PPC_NEXUS source level debugger is also useful for reading the register values, SRAM memory locations and most importantly setting break-points and single stepping through the code on each of the cores, providing execution control of each core independently.

7.2 Testing the μC/OS-II porting for Core_A and Core_B

Micrium provides a certain set of tests in-order to validate that the porting process was successful. One of the first tests performed over the course of the project was the testing of the kernels (Core_A and Core_B kernels) itself.

The approach to perform these tests was to not include any application level code so as to enable easy error isolation. With the presence of application code, the confidence in the testing of the kernel can be reduced as the application code itself might be the cause of the error.

Requirement Coverage:

Setup two independent μC/OS-II kernels running on each core concurrently on the dual-core MPC5675K MCU

7.2.1 Verify μC/OS-II Porting: OSTaskStkInit() and OSStartHighRdy()

This is to verify that the task stack initialization and the correct restoration of the CPU registers from the stack is done when a task begins to run. This test is based on the Micrium port testing guidelines and is performed for both Core_A and Core_B.

This test is done on each core at a time, while the other core continues to execute an infinite for(;;) loop.

Test Setup:

a. Setup the Includes.h and OS_CFG.h files as per the Micrium μC/OS-II documentation [25, p. 338]. Also disable the OS_TASK_EN macro within the OC_CFG.h since there are no applications created within this test.

b. Remove all of the application level code and create one TEST.C file containing the main function. Within this main function only OSInit(); and OSStart() function is called.

c. Compile and Run the code.

d. With the help of the debugger, first step over the OSInit() function.

e. Step into the OSStart() function and continue to single step till OSStratHighRdy() is reached.

f. Continue to step into OSStartHighRdy() function till the epilogue function is called.

g. Continue to step through the OSStratHighRdy() function till the last instruction, which is the return from interrupt.

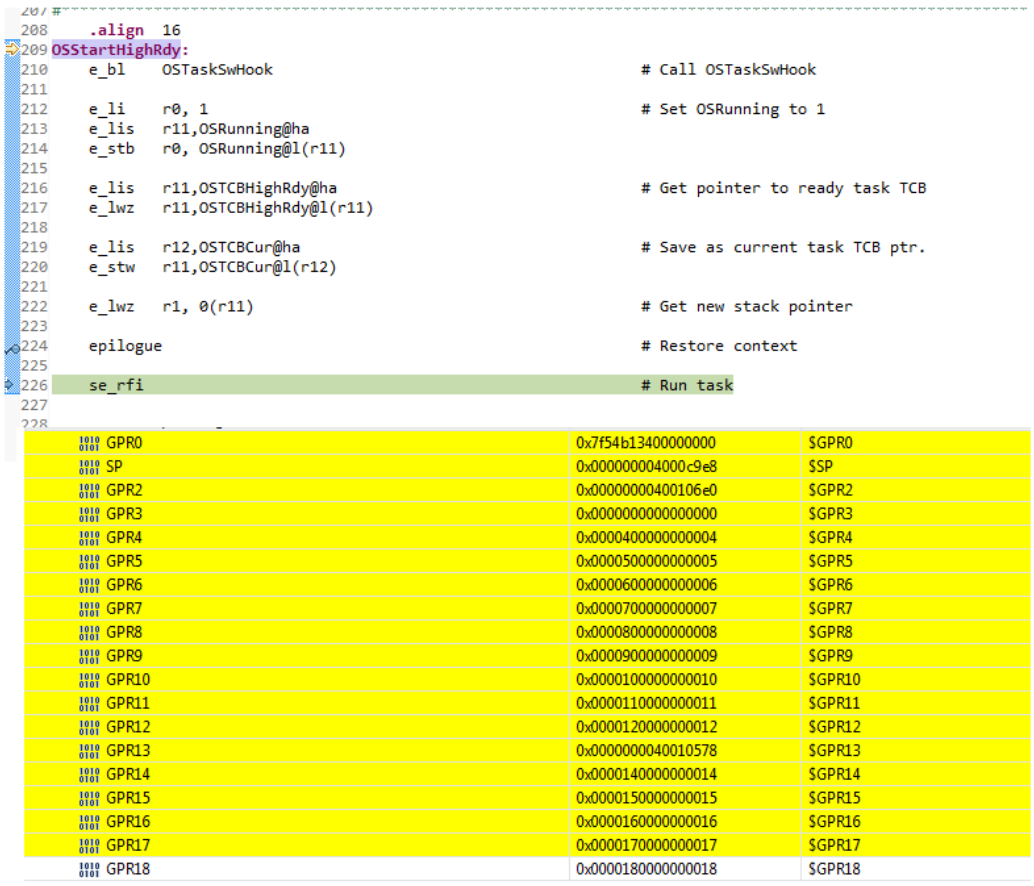
Expected Results:

a. On stepping through the epilogue assembly function (Test Setup, f) the CPU registers should begin to get populated. This is in accordance to how the task stack was initialized within the OSTaskStkInit() function.

b. After continuing to step through the OSStartHighRdy() function and finally on executing the return from interrupt instruction (Test Setup, g), the first instruction within the idle task should start to execute. Here since the idle task is created by default, it is expected that it begins to run as it is the only task present within the system.

Results:

a. The CPU register are populated as expected with the initialized values when the register values on the stack are popped out as shown in the figure.



```
208     .align  16
209 OSStartHighRdy:
210     e_bl    OSTaskSwHook                            # Call OSTaskSwHook
211
212     e_li    r0, 1                                   # Set OSRunning to 1
213     e_lis   r11,OSRunning@ha
214     e_stb   r0, OSRunning@l(r11)
215
216     e_lis   r11,OSTCBHighRdy@ha                     # Get pointer to ready task TCB
217     e_lwz   r11,OSTCBHighRdy@l(r11)
218
219     e_lis   r12,OSTCBCur@ha                         # Save as current task TCB ptr.
220     e_stw   r11,OSTCBCur@l(r12)
221
222     e_lwz   r1, 0(r11)                              # Get new stack pointer
223
224     epilogue                                        # Restore context
225
226     se_rfi                                          # Run task
227
```

GPR0	0x7f54b13400000000	$GPR0
SP	0x000000004000c9e8	$SP
GPR2	0x00000000400106e0	$GPR2
GPR3	0x0000000000000000	$GPR3
GPR4	0x0000400000000004	$GPR4
GPR5	0x0000500000000005	$GPR5
GPR6	0x0000600000000006	$GPR6
GPR7	0x0000700000000007	$GPR7
GPR8	0x0000800000000008	$GPR8
GPR9	0x0000900000000009	$GPR9
GPR10	0x0000100000000010	$GPR10
GPR11	0x0000110000000011	$GPR11
GPR12	0x0000120000000012	$GPR12
GPR13	0x0000000040010578	$GPR13
GPR14	0x0000140000000014	$GPR14
GPR15	0x0000150000000015	$GPR15
GPR16	0x0000160000000016	$GPR16
GPR17	0x0000170000000017	$GPR17
GPR18	0x0000180000000018	$GPR18

Figure 7.1: Initialization of Task Stack: CPU Registers

b. The idle task, which is the only tasks and hence also the highest priority task, begins to run on returning back from the interrupt called at the last instruction of the OSStratHighRdy() function. Hence the OSTaskStkInit() and OSStartHighRdy() are verified.

7.2.2 Verify μC/OS-II Porting: OSCtxSw()

This test is to verify that the context switch functionality of the μC/OS-II kernel works as expected. This test is based on the Micrium port testing guidelines and is performed for both Core_A and Core_B.

Test Setup:

a. Ensure that the OSCtxSw() interrupt handler is installed correctly to handle system calls for a context switch.

b. Create another task, say TestTask1, apart from the idle task (which is created by default and is the lowest priority task). This task contains an infinite while loop which repeatedly calls the OSTimeDly(1) function.

c. Start the test program. Here the highest priority task, TestTask1 should begin to run.

d. Step through the OSTimeDly() function, till the OS_Sched() function call is made. This function schedules and readies the next highest priority task to run.

e. Step through the OS_Sched() function till the OS_TASK_SW() function is called. This function contains the system call to invoke a context switch.

f. Since the OSCtxSw() interrupt handler has been installed, when the system call is raised, this interrupt is handled and after stepping through the entire ISR finally the Idle tasks begins to run.

Expected Results:

a. When the system starts to run, the first task to begin running should be the TestTask1() since it is the highest priority task.

b. When the system call (software interrupt) is raised, it should be handled properly by the invocation of the OSCtxSw() interrupt handler.

c. On steeping through the OSCtxSw() interrupt handler, the registers for TestTask1 should be saved onto the stack in preparation for a context switch.

d. After stepping through the entire OSCtxSw() ISR, idle task should begin to execute.

Results:

a. TestTask1 is the first task to start running.

b. The software interrupt mechanism to initiate a context switch works as expected.

c. The CPU register within the context of the TestTask1 are saved onto the stack as expected.

d. After stepping through the OSCtxSw() function Idle task beings to execute.

7.2.3 Verify μC/OS-II Porting: OSIntCtxSw() and OSTickISR()

OSIntCtxSw() is called from within the external interrupt handler. This performs a context switch to a task, which has been made ready to run, from within the interrupt service routine. The OSTickISR is the interrupt handler for the interrupt raised by the decrementer on each of the CPUs. Each call of the OSTickISR represents an OS Tick.

This test, tests the above two functions and is based on the Micrium port testing guidelines. It is performed for both Core_A and Core_B.

Here instead of the debugger we would be using the LED to validate the correct behavior.

Test Setup:

a. Setup the interrupt vector for clock tick ISR.

b. Set up the OS_TICKS_PER_SEC macro within OS_CFG.h to the required rate. For the sake of this test is was recommended by the Micrium testing procedure to set it to 10 Hz.

c. Create a task named TestTask2 (as in the previous test). Within this task initialize the OS clock tick and enable periodic interrupts from the decrementer (done within BSP section).

d. Within the TestTask2 body (infinite while loop section) add a OSTimeDly(1), following which toggle the LED.

Expected Results:

a. As seen in the previous tests, since TestTask2 is the higest priority task in the system it would begin to run.

b. After the OSTimeDly() function is called from TestTask2 the idle task should begin to run. This can be seen with the help of the debugger if needed.

c. After the delay time expires the TestTask2 function should begin to run and execute the LED toggle function and the LED should begin to blink.

Results:

a. TestTask2 did begin to run as expected.

b. The led started to blink and continued to blink (toggled between every call) till the program execution was halted or stopped.

c. It was seen that the tick interrupt invoked that OSTickISR() interrupt handler.

d. Within this ISR the OSTimeTick function is called which decrements the .OSTCBDly count for TestTask2 till it reaches 0. When it reaches 0, it signifies that the task needs to be woken up and is ready to run. During the last OSTickISR call, when the OSTCBDly reached 0, within the OSIntExit function it is seen that instead of returning to the idle task it needs to return to TestTask2. On realizing so it calls the OSIntCtxSw() to force a context switch to TestTask2. Since the LED begins to blink as expected it can be assumed that the OSTickISR as well as the OSIntCtxSw functionality, as explained above, works as expected.

7.2.4 Performance Improvement with Respect To Single Core

This test is to showcase that a high parallel tasks sets where there is no inter-dependence of tasks an multi-core AMP setup such as the one provided within the project provides 1.6 times, and often more, the performance improvement over a single core setup.

Three version of the tests were performed, namely, short, long and very long versions. Depending on the version the memory and mathematical operation were performed 1 million, 10 million or 100 million times.

The measurements of execution times was taken with the help of the OS tick values on each core and also using the STM_0 and STM_1 counter values on each core respectively.

Test Setup:

a. To emulate a single core setup we pause the normal execution of one of the cores with the help of an infinite for loop or with the help of the debugger. Then on the core allowed to run we create 2 tasks each of which perform relatively long memory accesses in order to achieve relatively long execution times. This setup of tasks is shown in Fig 7.2, Test Setup (a).

b. In the fully functioning AMP setup both cores are functioning and each of the core now contains one of these tasks. Since these tasks completely parallelizable owing to no inter-dependency between the tasks this split is possible. Following diagram shows the setup. This setup of tasks is shown in Fig 7.2, Test Setup (b).

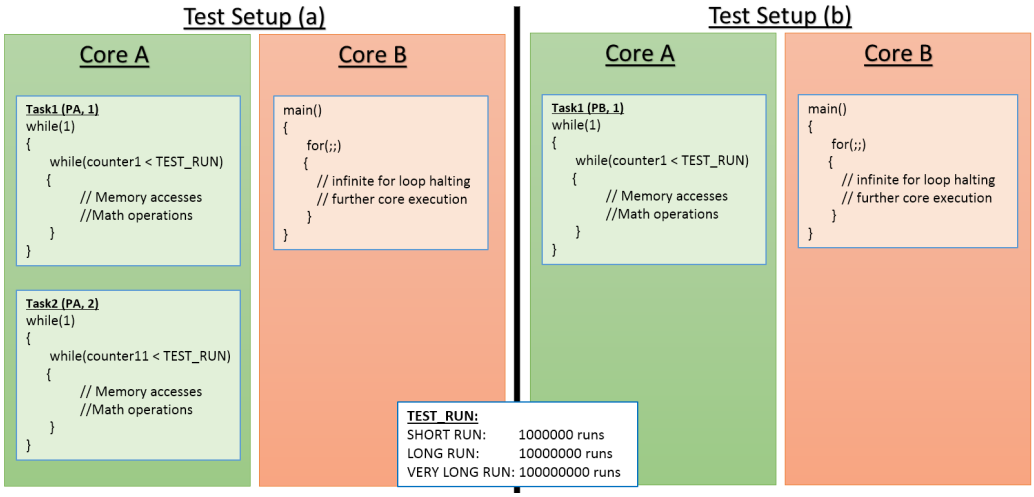


Figure 7.2: AMP vs Single Core Performance: Test Setup (a) and (b)

c. This test is to show that the execution load performed by one task alone on one core if split up amongst 2 tasks, where one task runs on each core, load balancing and faster overall execution times can be achieved to perform that same job. For this setup first to emulate a single core environment we pause the normal execution of one of the cores with the help of an infinite for loop or with the help of the debugger. Then on the core allowed to run we create a tasks which now runs the memory and math operations 2 times as many as the value held by TEST_RUN macro. This setup of tasks is shown in Fig 7.3, Test Setup (c).

d. This setup is the same as setup done in setup scenario (b). Each tasks on each core performs the memory and math operations only TEST_RUN number of times. This value is half the number of iterations performed by the single task in setup scenario (c). This setup of tasks is shown in Fig 7.3, Test Setup (d).

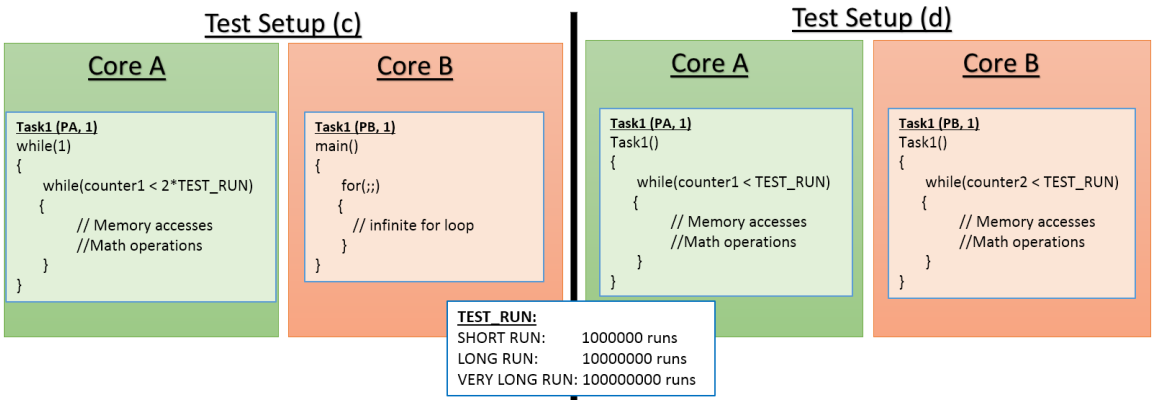


Figure 7.3: AMP vs Single Core Performance: Test Setup (c) and (d)

Test RUNS	Core_A	Core_B
1000000	1351ms	1293ms
10000000	13553ms	12936ms
100000000	130.68s	129.98s

Table 7.1: Performance Measurement of Test Case(a)

Expected Results:

a. On comparing the execution times from the Test Setup-a and Test Setup-b a speed up of at least 1.6 should be seen.

b. On comparing the execution times from the Test Setup-c and Test Setup-d a speed up of at least 1.6 but the speed up factor should be slightly lower than in the previous test case, owing setup (c).

Results:

a. On comparing the execution times from the Test Setup-a and Test Setup-b a speed up of 1.88 was seen.

b. On comparing the execution times from the Test Setup-c and Test Setup-d a speed up of 1.71 was seen.

Test RUNS	Core_A	Core_B
1000000	688ms	686ms
10000000	6882ms	68756ms
100000000	68.82s	68.77s

Table 7.2: Performance Measurement of Test Case(b) and Test Case(d)

Test RUNS	Core_A	Core_B
1000000	1176ms	1174ms
10000000	11765ms	117526ms
100000000	117.65s	117.53s

Table 7.3: Performance Measurement of Test Case(d)

7.3 Testing Mutual Exclusion Primitives

The following test case is used to validate the functionality of mutual exclusion through spinlock. These spinlocks are implemented using Atomic CAS and the Hardware semaphore peripheral.

Three version of the tests were performed, namely, short, long and very long versions. Depending on the version the below mentioned checks and incrementing of the shared_var variable was performed 1 million, 10 million or 100 million times.

By incorporating a check to see if the shared_var value is 1000 before every iteration of setting it back to zero and incrementing it back to 1000, it is possible to check if the global critical section has been breached by more than one core at a time.

Requirement Coverage:

Provide mutual exclusion primitives to make operating system components accessing shared memory safe

7.3.1 Testing Mutual Exclusion Functionality Based on Atomic CAS

The mutual exclusion APIs developed in this project called AtomicCAS_Spinlock() and AtomicCAS_SpinUnlock() have been tested in this test case.

The task setup and pseudo code for each task is provided within Fig 7.4.

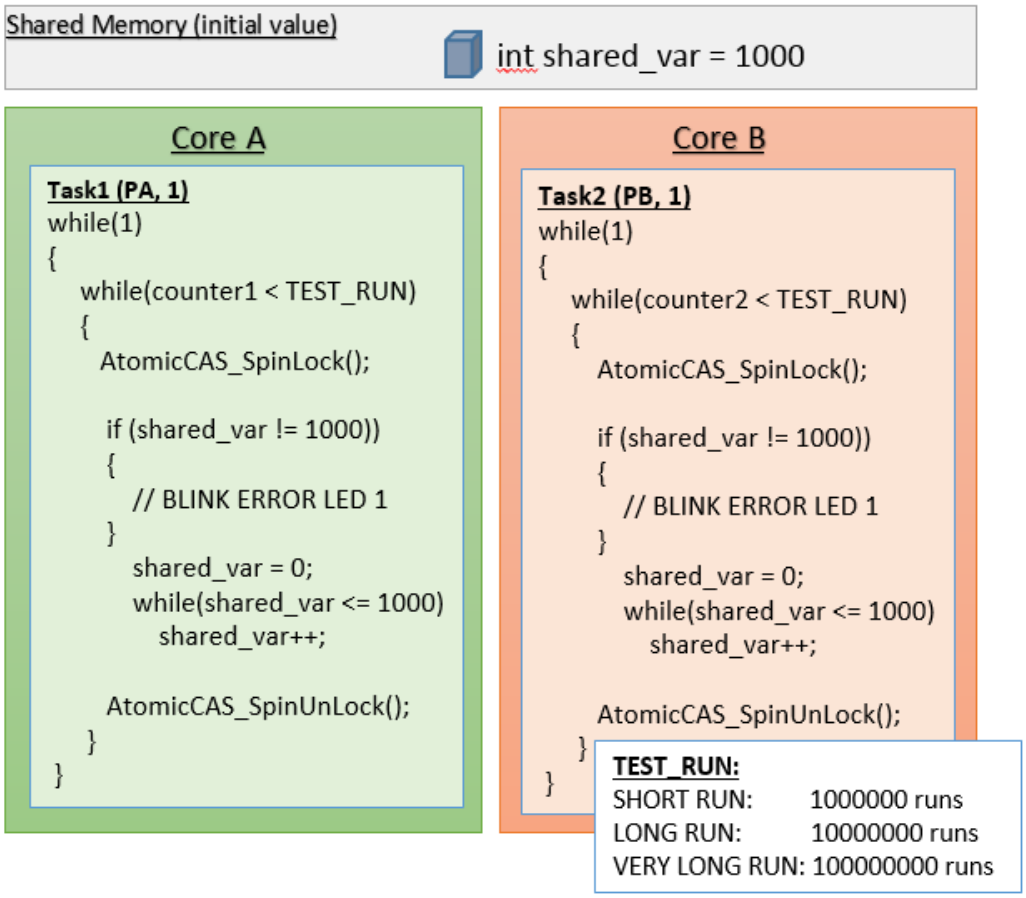


Figure 7.4: Test Setup: Spin-lock using Atomic CAS

Test Setup:

a. A shared variable of integer type, named shared_var, is created at a pre-determined shared memory address. This shared variable is initialized to a value of 1000.

b. One tasks is created on each core, namely T1 (PA, 1) and T2 (PB, 1).

c. Each of these tasks are responsible for incrementing the shared_var value from 0 to 1000. Before this incrementing is done, first the value held inshared_var is checked to see if it is equal to 1000. Only if it is equal the incrementing process within the task begins to execute. However, if the check shows that the value of shared_var is not 1000, then an LED is switched on each core indicating an error has occurred. Each core has a distinct LED assigned to it, say LED1 and LED2 for Core_A and Core_B respectively.

d. All of the above mentioned cheeks and incrementing operation on each core is protected with the help of AtomicCAS_Spinlock() and AtomicCAS_SpinUnlock() API to ensure the access to shared_var is not corrupted by the other core.

e. Steps (c) and (d) are repeated TEST_RUN number of times.

Expected Results:

a. Here the process of acquiring the global critical section depends on the processor which reaches the call for acquiring the lock first, i.e. the AtomicCAS_Spinlock() function call. Once a task enters the global critical section, the other task on the other core cannot enter this global critical section till the first task releases the spinlock using the AtomicCAS_SpinUnlock() function.

b. Say task T1 (PA, 1) has entered the global critical section and T2 (PB, 1) continues to spin within AtomicCAS_Spinlock() function, preventing it from entering the global critical section. Here, T1 (PA, 1) continues to run till it runs the test TEST_RUN number of times and only after doing so would it release the lock. In the event of the failure of the mutual exclusion primitive, T2 (PB, 1) would also enter the global critical section where it would first check the shared_var value. In

this case T2 (PB, 1) would see that the value is not equal to 1000 and would turn LED1 on, indicating an error.

Results:

a. In any version of the test performed the error LED was not turned on. This indicates that at any point of time the global critical section was not breached by more than one core at a time.

b. Since this mutual exclusion primitive is fair to both cores the likelihood of Core_A entering the global critical section is equal to that of the Core_B. Hence in a situation of a race to acquire the lock, Core_A and Core_B both have approximately 50 per-cent chance of getting ownership of the lock.

7.3.2 Testing Mutual Exclusion Functionality - Hardware Semaphore

The task setup and pseudo code for each tasks relevant to this test is provided within Fig 7.4. Only the API calls have been modified wherein the spinlock implementation using the hardware semaphore peripherals have been used.

Test Setup:

a. The test setup for this test case is identical to the one provided within the Test Case 7.3.1. The only difference in this case is the use of the SEMA4_Spinlock() and SEMA4_SpinUnlock() function instead of the AtomicCAS_SpinLock() and AtomicCAS_SpinUnLock() functions.

Expected Results:

a. For all the TEST_RUN versions of the test, none of the error LEDs should blink, indicating that when a core has entered a global critical section the other core has also not entered it. These expected results are the same as with Test Case 7.3.1.

Test Feature	Core_A	Core_B
Average time to acquire lock- ATOMIC CAS	1.62μs	1.58μs
Average time to acquire lock- SEMA4	4.29μs	4.23μs

Table 7.4: Performance Comparison of Spin-locks

Results:

a. In any version of the test performed the error LED was not turned on. This indicates that at any point of time the global critical section was not breached by more than one core at a time.

7.3.3 Comparison of Spin Locks-Atomic CAS and HWSEMA4

Test Setup:

In this test two identical tasks were created on each core were created, namely T_1 (P_A, 1) and T_2 (P_B, 1). Each of these tasks continually attempt to acquire access to a global critical section protected with the help of the mutual exclusion primitives based on the Atomic CAS and SEMA4 peripherals. The amount of time required, to acquire the lock in this case can be viewed as a pessimistic value as both tasks, owing to their identical setup and their concurrent execution, would simultaneously try to acquire the lock. This test was performed 10 times and the average time to acquire the lock by either core was taken.

Expected Results:

It is expected that the Atomic CAS implementation of the mutual exclusion primitives should generally perform better with respect to the SEMA4 based implementation.

Results:

As is seen in the Table 7.4 the performance of the Atomic CAS based mutual exclusion locks generally performed better. This behavior can be explained because of the latency in accessing the SEMA4 peripheral.

Chapter 8

CONCLUSION AND FUTURE WORK

The transition into multicore architectures on embedded systems is imperative as has been established by a large body of work. However, the software aspect of such a transition still seems to be lagging behind. The steps taken to setup an AMP variant μC/OS-II can be applied to any other underlying hardware.

A point worth mentioning here is that though the complexity of porting μC/OS-II to a multi-core setup was greatly reduced owing to the advantages that AMP provides, the synchronization and message passing functionality between 2 distinct tasks sets running on different kernels became considerably more complex as compared to an SMP setup. This is so because in an SMP setup, a single kernel manages all task synchronization and message passing capabilities. An AMP setup also brings about the need for greater design time consideration in task allocation on the cores to ensure optimal usage of the individual cores.

Future work in this project would involve creating a task profiling tool with which task segregation amongst the cores can be made automated, more robust and easy to define. This setup would prove useful in performing detailed worst-case execution times analysis in support of this task partitioning scheme.

Also, since the AMP setup is considerably more expensive in terms of memory usage, a major focus of the work would also involve making the newly added capabilities as optimal as possible in an attempt to reduce the memory footprint. It would also be interesting to see how this implementation performs in a further scale up to more than 2 cores and possibly on a heterogeneous multi-processor hardware.

Another future area that can be explored is to allow the AMP setup to run in a high availability mode or fail-over mode. This means that if one of the kernels fails, the other fully-functional kernel should be able to take over. This can be done in a number of possible ways, with one solution being akin to the concept of a watchdog timer. One task on kernel A can keep signaling kernel B at regular intervals. If the signal is not received by kernel B, it can be assumed that kernel A has failed and hence kernel B should take over the processing functionality. Until this time, kernel B (Core_B) could be in a barely running state, to conserve power.

In keeping with the distributed nature of contemporary computing, one topic that could be explored is the concept of distributed locks over CANBus. This setup could consist of one of the cores/boards/kernels acting as a lock server, and the other actors being the clients who wish to acquire a distributed lock. The design would be based on the client-server model, with both the sends and receives being of the blocking type. The lock server would maintain information of the current lock holder and also a queue of all the waiting clients, so it knows who is to get the lock once it is released.

From a Simulink code-generation point of view, the next evolutionary step to the multi-task implementation is to incorporate multi-task and multi-core support within one model. This would allow designers to design a complete system, unaware of the underlying hardware configuration. The code generation process along with the task profiling tool would automatically allocate the block functionality to the appropriate cores to ensure optimal CPU usage.

REFERENCES

[1] Z. Al-bayati, Haibo Zeng, M. Di Natale, and Zonghua Gu. Multitask implementation of synchronous reactive models with earliest deadline first scheduling. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 168–177, June 2013. doi: 10.1109/SIES.2013.6601489.

[2] Theodore P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. Technical report, 2005.

[3] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6): 600–625, 1996.

[4] Fawzi Behmann. Power architecture combines rich features for embedded. http://www.rtcmagazine.com/articles/view/102566, September 2012. Online; accessed 20-July-2014.

[5] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.934110.

[6] B.B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 292–302, July 2013. doi: 10.1109/ECRTS.2013.38.

[7] John Carbone. A smp rtos for the arm mpcore multiprocessor. Technical report, Express Logic, 2005.

[8] Brian Carlson and Steve Jahnke. Leveraging the benefits of symmetric multiprocessing (smp) in mobile devices. *Texas Instruments white paper*, 2009.

[9] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Trans. Embed. Comput. Syst.*, 7(2):15:1–15:40, January 2008. ISSN 1539-9087. doi: 10.1145/1331331.1331339. URL `http://doi.acm.org/10.1145/1331331.1331339`.

[10] Jing Chen and Alan Burns. A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical report, 1997.

[11] Keith Diefendorff, Rich Oehler, and Ron Hochsprung. Evolution of the powerpc architecture. *IEEE Micro*, 14(2):34–49, 1994. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/40.272836.

[12] Freescale. *Freescale PowerPC Architecture Primer*. Freescale Semiconductor, Inc., June 2005. URL `http://www.freescale.com/files/32bit/doc/white_paper/POWRPCARCPRMRM.pdf`.

[13] Freescale. *Synchronizing Instructions for PowerPCâĎć Instruction Set Architecture*. Freescale Semiconductor, Inc., November 2011. URL http://cache.freescale.com/files/32bit/doc/app_note/AN2540.pdf.

[14] Freescale. *MPCxxx Instruction Set*. Freescale Semiconductor, Inc., November 2011. URL http://cache.freescale.com/files/product/doc/MPC82XINSET.pdf.

[15] Freescale. *e200z760n3 Power ArchitectureÂő Core Reference Manual*. Freescale Semiconductor, Inc., June 2012. URL http://cache.freescale.com/files/32bit/doc/ref_manual/e200z760RM.pdf.

[16] Freescale. *Qorivva MPC5675K Microcontroller Data Sheet*. Freescale Semiconductor, Inc., October 2013. URL http://cache.freescale.com/files/32bit/doc/data_sheet/MPC5675K.pdf.

[17] Freescale. *Qorivva MPC5675K Microcontroller Reference Manual*. Freescale Semiconductor, Inc., November 2013. URL http://cache.freescale.com/files/32bit/doc/ref_manual/MPC5675KRM.pdf.

[18] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 203–210, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. doi: 10.1145/1017753.1017787. URL http://doi.acm.org/10.1145/1017753.1017787.

[19] Eric Heikkila and J. Eric Gulliksen. Multi-core computing in embedded applications: Global market opportunity and requirements analysis. Technical report, VDC Research Group (VDC) Embedded Hardware and Systems Group, September 2007.

[20] Martin Hein. Accelerating sensor development with rapid prototyping and model-based design. http://www.mathworks.com/company/newsletters/articles/accelerating-sensor-development-with-rapid-prototyping-and-model-based-design.html, 2013. Online; accessed 19-July-2014.

[21] Andreas Karrenbauer and Thomas Rothvoß. *An average-case analysis for rate-monotonic multiprocessor real-time scheduling*. Springer, 2009.

[22] Leonard Kleinrock. Queueing systems, volume ii: computer applications. 1976.

[23] W. Knight. Two heads are better than one [dual-core processors]. *IEE Review*, 51:32–35(3), September 2005. ISSN 0953-5683. URL http://digital-library.theiet.org/content/journals/10.1049/ir_20050903.

[24] Hermann Kopetz, R. Obermaisser, C. El Salloum, and B. Huber. Automotive software development for a multi-core system-on-a-chip. In *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS '07. Fourth International Workshop on*, pages 2–2, May 2007. doi: 10.1109/SEAS.2007.2.

[25] Jean J. Labrosse. *MicroC OS II: The Real Time Kernel (With CD-ROM)*. CRC Press, 2 edition, 6 2002. ISBN 9781578201037. URL http://amazon.com/o/ASIN/1578201039/.

[26] Jean J Labrosse. *uC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs (Board NOT Included)*. Micrium Press, 9 2009. ISBN 9780982337530. URL http://amazon.com/o/ASIN/0982337531/.

[27] Phillip A. Laplante and Seppo J. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley-IEEE Press, 4 edition, 11 2011. ISBN 9780470768648. URL http://amazon.com/o/ASIN/0470768649/.

[28] Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors. *Principles of Distributed Systems: 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings (Lecture Notes in ... Computer Science and General Issues)*. Springer, 2010 edition, 12 2010. ISBN 9783642176524. URL http://amazon.com/o/ASIN/3642176526/.

[29] MathWorks. *Real-Time Workshop UserâĂŹs Guide*. The MathWorks, Inc., January 1999. URL http://www.clemson.edu/ces/crb/ece496/fall2000/rtw_ug.pdf.

[30] MathWorks. *Target Language Compiler For Use with Real-Time WorkshopÂő*. The MathWorks, Inc., 2000. URL http://radio.feld.cvut.cz/matlab/pdf_doc/rtw/targetlanguagecompiler.pdf.

[31] MathWorks. *Basic C MEX S-Function R2014a Documentation*. The MathWorks, Inc., March 2014. URL http://www.mathworks.com/help/simulink/sfg/example-of-a-basic-c-mex-s-function.html.

[32] MathWorks. *Rate Transition R2014a Documentation*. The MathWorks, Inc., March 2014. URL http://www.mathworks.com/help/simulink/slref/ratetransition.html.

[33] MathWorks. *Simulink Coder User's Guide*. The MathWorks, Inc., 2014. URL http://www.mathworks.com/products/simulink-coder/index.html.

[34] James Mistry, Matthew Naylor, and Jim Woodcock. Adapting freertos for multicores: an experience report. *Software: Practice and Experience*, 2013.

[35] T.G. Moreira, M.A Wehrmeister, C.E. Pereira, J. Petin, and E. Levrat. Automatic code generation for embedded systems: From uml specifications to vhdl code. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 1085–1090, July 2010. doi: 10.1109/INDIN.2010.5549590.

[36] Marco Di Natale. Simulink, simulation, code generation and tasks. http://retis.sssup.it/ marco/files/lesson23_Simulink.pdf, 2013. Online; accessed 19-July-2014.

[37] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 251–261, July 2011. doi: 10.1109/ECRTS.2011.31.

[38] Farhang Nemati. *Partitioned Scheduling of Real-Time Tasks on Multi-core Platforms*. School of Innovation, Design and Engineering, Mälardalen University, 2010.

[39] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991. ISBN 0792392116.

[40] Mark Ruthenbeck. *Qorivva MPC5643L Dual Processor Mode*. Freescale Semiconductor, Inc., March 2011. URL http://cache.freescale.com/files/32bit/doc/app_note/AN4034.pdf.

[41] B. Schatz, A. Pretschner, F. Huber, and J. Philipps. Model-based development of embedded systems. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 298–311. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-44088-8. doi: 10.1007/3-540-46105-1_34. URL http://dx.doi.org/10.1007/3-540-46105-1_34.

[42] Mong Sim. *A Practical Approach to Hardware Semaphores*. Freescale Semiconductor, Inc., January 2014. URL http://cache.freescale.com/files/32bit/doc/app_note/AN4805.pdf.

[43] Christos Sofronis. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *In 6th ACM International Conference on Embedded Software (EMSOFTâĂŹ06*, 2006.

[44] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *Computer*, 24(9):30–38, September 1991. ISSN 0018-9162. doi: 10.1109/2.84897. URL http://dx.doi.org/10.1109/2.84897.

[45] Hiroyuki Tomiyama. Real-time operating systems for mpsocs. http://www.mpsoc-forum.org/previous/2009/slides/Tomiyama.pdf, 2009. Online; accessed 19-July-2014.

[46] Chaojie Xiao Xiaofeng Guan. Inline assembly - start from scratch. https://www.ibm.com/developerworks/aix/library/au-inline_assembly/, November 2012. Online; accessed 19-July-2014.

[47] Jiao Yu and B.M. Wilamowski. Recent advances in in-vehicle embedded systems. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 4623–4625, Nov 2011. doi: 10.1109/IECON.2011.6120072.

[48] Yi Zhang, Nan Guan, Yanbin Xiao, and Wang Yi. Implementation and empirical comparison of partitioning-based multi-core scheduling. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, pages 248–255. IEEE, 2011.