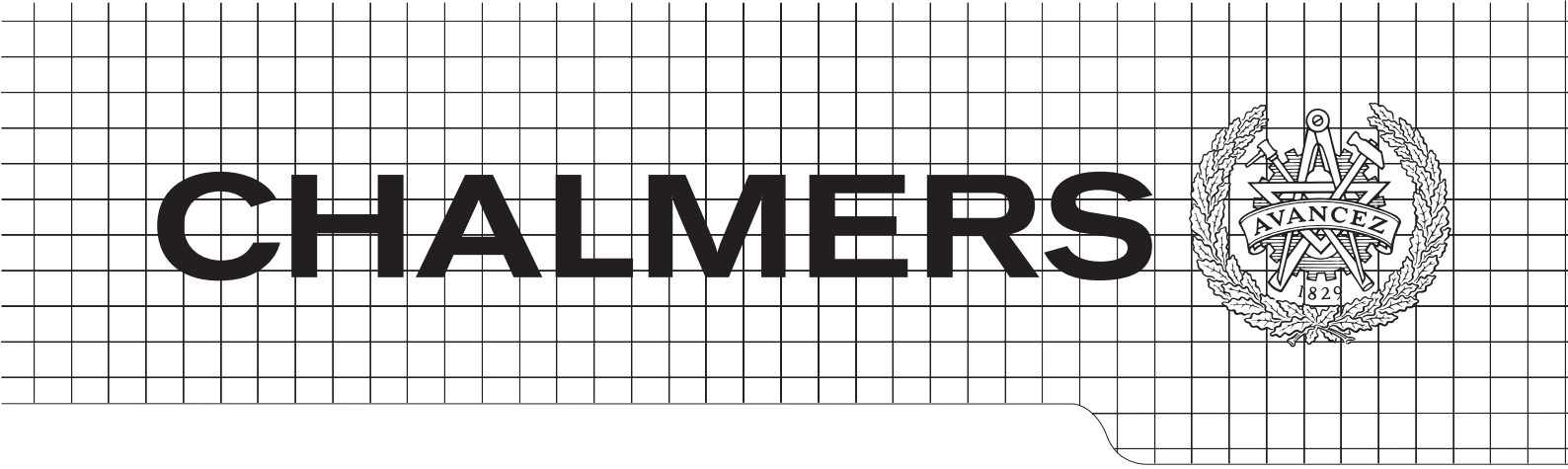

Performance Evaluation and Modeling of a Multicore AUTOSAR System

On theoretical modelling of speedup gain in heterogeneous multicore systems

Abdollah Safaei Moghaddam

Master of Science Thesis in Networks and Distributed Systems

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2013

Performance Evaluation and Modeling of a Multicore AUTOSAR System

Abdollah Safaei Moghaddam

Examiner: Jan Jonsson

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2013

Abstract

Multicore processors have become common in personal computers and workstations for the past few years, and they are making their way to embedded devices. Meanwhile, Electronic Controller Unit (ECU) suppliers have also introduce multicore solutions in the automotive Electrics and Electronics(E/E) domain. The automotive E/E architectures are expected to adapt themselves to this change. This leads the AUTomotive Open System ARchitecture (AUTOSAR) standard to introduced multicore support in release 4.0. Because of the close ties and dependencies between the software and hardware, this adaptation is a complex task. The dependencies between hardware and software need to be handled carefully for any well performing multicore software.

Based on the AUTOSAR solution, we believe that the cross-core communication could be a potential bottleneck and hence, this study measures SoftWare Components(SWC) communication time in inter-core and intra-core. In order to achieve this, a mocking of an AUTOSAR software was designed, implemented and tested on a dualcore MPC551x processor. Furthermore, a theoretical model for speedup gain prediction on heterogeneous dualcore systems is proposed. The model considers a scenario in which a task is fragmented into so-called slave tasks among cores in order to achieve speedup. By using this model, once can predict the possible speedup gain when migrating a software from a single-core to a multicore platform. The model is driven by extending Amdahl's law and addressing the cross-core communication overhead in AUTOSAR and the heterogeneous nature of the MPC551x processor. The results show that cross-core communication has an overhead of 54%. The speedup curve shows that in tasks with large execution times, the speedup is 1.74 and that speedup is unity for tasks with an execution time about $28\mu s$. The proposed model is evaluated by carrying out several test scenarios and comparing the results with the model which shows the model is more than 90% accurate.

Keywords: AUTOSAR, automotive, multicore ECU, dualcore, speedup, heterogeneous, theoretical modeling, Amdahl, software migration

Acknowledgments

I would like to thank my supervisor, Anders Svensson, for his endless support, kindness and enthusiasm. Needless to say, without his close cooperation and keen ideas that shed light on dark corners of this research, this would not have been possible. Furthermore, I would like to thank Mafijul Islam and Tommy Andersson for their in depth support and their time throughout this work. Also, I like to thank Johan Haraldsson, Sigurjón Þorvaldsson, Peyman Barazandeh and all my colleagues for the great times that we had together and the interesting discussions that broadened my understanding of the automotive industry. And last but not least, I am thankful for my family and friends for their support and understanding.

Contents

1 Introduction **1**
- 1.1 Thesis delimitations and problem definition . . . 2
- 1.2 Thesis structure . . . 2

2 Background **4**
- 2.1 Introduction to AUTOSAR . . . 4
  - 2.1.1 Application layer . . . 5
  - 2.1.2 Communication between software layers . . . 5
  - 2.1.3 VFB layer . . . 6
  - 2.1.4 RTE layer . . . 7
  - 2.1.5 BSW layer . . . 8
- 2.2 Multi-core support in AUTOSAR . . . 8
  - 2.2.1 Communication in multi-core AUTOSAR . . . 9
  - 2.2.2 The Operating System . . . 9
  - 2.2.3 Inter OS-Application communication . . . 11

3 Test and Development Environment **18**
- 3.1 Software environment . . . 18
  - 3.1.1 ArcticStudio and ArcCore . . . 18
  - 3.1.2 CodeWarrior . . . 19
  - 3.1.3 FreeMASTER . . . 19
- 3.2 Hardware environment . . . 22

4 Inter-core versus intra-core communication **23**
- 4.1 Test description . . . 23
- 4.2 Intra-core communication . . . 24
- 4.3 Inter-core communication . . . 26
- 4.4 Communication time measurement results . . . 30

5 Speedup gain on an AUTOSAR 4.0 implementation with dualcore support **32**
5.1 Introduction to speedup and Amdahl's law 32
5.2 Measuring speedup gain 33
5.2.1 Single-core scenario 34
5.2.2 Dualcore scenario 37
5.2.3 Results 37
5.3 Extending Amdahl's law to a heterogeneous multicore AUTOSAR system 41
5.3.1 Earlier studies 41
5.3.2 Model overview and Assumptions 41
5.3.3 Using and evaluating the extended speedup model 45

6 Conclusions **49**

Bibliography **53**

Appendix A How to use FreeMASTER **55**
A.1 Introduction to FreeMASTER 55
A.1.1 Memory read and write 56
A.1.2 Oscilloscope 56
A.1.3 Recorder 56
A.2 FreeMASTER on target driver 56
A.2.1 Establishing a serial communication 56
A.2.2 Using FreeMASTER on target driver 61
A.3 Introduction to PC side software 63
A.3.1 Communication setup in PC application 63
A.3.2 Reading and Modifying variables 63
A.3.3 Recorder 63

Appendix B Verification experiments on the Extended speedup model **65**

1

Introduction

Nowadays, electronics play a key role in vehicles and we are driving computers, rather than cars. The electronic equipment that were normally used in luxury cars before, are used in conventional vehicles, today. As a result, the functions that were special to luxury cars, are expected to be seen, as standard functions, in middle-class cars over time. Growth of functionalities in cars results in an increase of electronics complexity. As complexity increases in electrical and electronics (E/E) architecture of a vehicle, it has become more and more necessary to standardize automotive E/E architectures to manage this complexity. Therefore, the "AUTomotive Open System ARchitecture", AUTOSAR, was founded to fulfill this requirement. The growth in software size not only impacts software, but more powerful hardware is also needed.

As software required faster hardware, processor manufactures provided faster processors, mainly by increasing the clock frequency. However, higher clock frequency led to a greater power consumption. In the early 2000s, the clock frequency trend started flatting out by hitting the power consumption ceiling according to Krste Asanovic *et al* [1]. Thereby, increasing clock frequency was no longer possible. Hence, new techniques had to be introduced for increasing computational power, but, with the same power consumption. One solution to this problem is multi-core processors as Krste Asanovic *et al* describe in [1]. The automotive industry is no exception in this trend and hence, multi-core ECUs were introduced. As ECU software developers will need to adopt to this change, the AUTOSAR standard needs to support a multi-core architecture for ECU suppliers and vendors, also. Therefore, multi-core support was introduced in AUTOSAR 4.0.

1.1 Thesis delimitations and problem definition

The study was accomplished at Advanced Technology and Research at Volvo Group Trucks Technology and the thesis delimitations were defined in alignment with the Volvo's research projects.
The AUTOSAR design was based on single-core micro-controllers up to version 3.1. As the need of multi-core processors became trivial, the AUTOSAR introduced the first version of multi-core AUTOSAR in version 4.0. This project investigates and evaluates the performance of the current AUTOSAR multi-core architecture. The project is carried on an MPC5517E micro-controller, addressing the followings:

- *What are the possible bottlenecks in communication among cores and what is the communication overhead in a cross-core communication compared to an intra-core communication?* An architectural study is carried out and based on the prospective bottlenecks found, an experimental mocking model of an AUTOSAR implementation is done by having only the necessary modules for an AUTOSAR software, to support the findings.

- *What is the possible speedup gain achieved in the current implementation of AUTOSAR 4.0?* When migrating from a single-core to a multi-core software, it is important to know what advantage is gained in such a migration. A test scenario is designed and implemented in order to define the upper bound of this advantage gain.

- *Is it possible to predict the speedup gain of an AUTOSAR application, when migrating the software from a single-core to a multicore architecture?* Preliminary steps are taken towards a theoretical modeling of the speedup gain achieved by migrating from a single-core to a dualcore implementation. The model is based on hardware specific factors of the MPC551x processor and the cross-core communication overhead. The theoretical model will extend Amdahl's law such that heterogeneous systems and the AUTOSAR multi-core architecture are addressed. Given an already implemented set of software components, the model will predict what speedup can be achieved, if the implementation is migrated from a single-core to a dual-core system.

1.2 Thesis structure

The remainder of this thesis is structured as the following: Chapter 2 describes the AUTOSAR layers and studies the key modules introduced in AUTOSAR 4.0 for supporting multi-core software. Chapter 3 explains the "Test and development environment". In this chapter, various tools that are used throughout the project are described and the software development life cycle is explained. Chapter 3 also describes an overview of the MPC551x processor hardware architecture. In Chapter 4, the cross-core communication overhead is measured by doing an experiment. Chapter 5 describes the speedup gain in a multi-core implementation of an

AUTOSAR software and explains the theoretical modeling developed for predicting speedup gain. Chapter 6 gives a conclusion and summarizes the findings and results.

2

Background

The "AUTomotive Open System ARchitecture", AUTOSAR, was founded during 2002-2004 as an agreement between core partners, BMW, Bosch, Continental, Daimler Chrysler, Ford, Toyota and Volkswagen[2]. Today, 9 Core partners, 50 premium members, 78 associate members and 21 development members support AUTOSAR[3]. This chapter briefly describes the AUTOSAR standard. Section 2.1 gives an introduction to the AUTOSAR layers and interfaces. Section 2.2 explains how multi-core support is introduced in AUTOSAR 4.0 and briefly describes the newly introduced modules that are necessary to support multi-core.

2.1 Introduction to AUTOSAR

The main goal of AUTOSAR, as stated in the official website is:

> *"The primary goal of the AUTOSAR development cooperation is the standardization of basic system functions and functional interfaces, the ability to integrate, exchange and transfer functions within a car network and to substantially improve software updates and upgrades over the vehicle lifetime. Having this goal in mind, AUTOSAR pushes the paradigm shift from an ECU based to a function based system design attempt in automotive software development and enables the management of the growing E/E complexity with respect to technology and economics."* - AUTOSAR.org, [4]

AUTOSAR is designed based on a layered, modular, stack structure such that each layer is independent from the other layers in order to satisfy modularity among functions, ease of software mobility and migration, hardware independence, and etc. Figure 2.1 illustrates the AUTOSAR stack architecture. A brief explanation is given on different layers of the AUTOSAR architecture in the following sections.

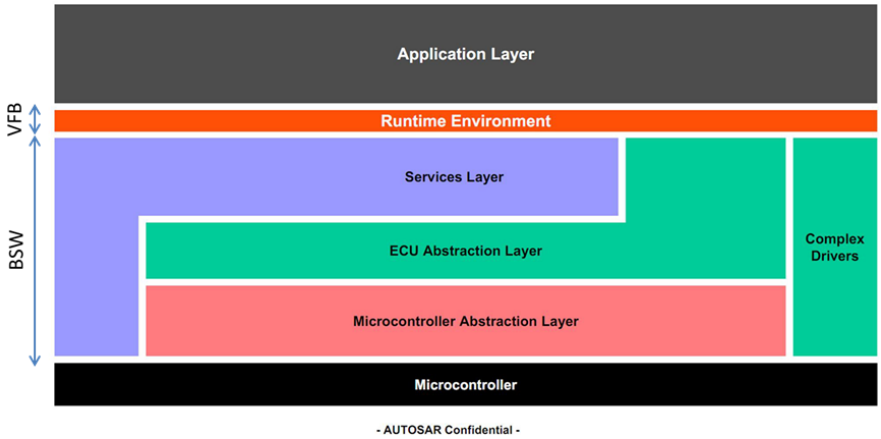

Figure 2.1: *Overview of software layers in AUTOSAR. The BSW is shown in sub-layers depicting a high level structure of the BSW itself ([2]).*

2.1.1 Application layer

Conceptually, the application layer consists of the so called SoftWare Components (SWC) that each of which contain one or more *runnable*. Each SWC is an application software that provides a small functionality of the vehicle. SWCs are location independent and ECU independent such that a SWC could be moved to another ECU without the need of modification. Communication between SWCs is supported through standardized communication interfaces.

2.1.2 Communication between software layers

AUTOSAR states that for software re-usability and component mobility, AUTOSAR modules shall communicate with each other through predefined structured *interfaces*. Two different types of interfaces are introduced in AUTOSAR, *"Sender/Receiver"* interface (message passing), and *"Client/Server"* interface (function call). The sender/receiver interface makes an asynchronous data exchange whereas the client/server interface makes a synchronous call. As for now, we are concerned about the sender/receiver communication interface and within this project a communication refers to a sender/receiver communication unless specified otherwise. Also, communication between SWCs and modules within the underlying layers is done through the standard interfaces introduced. Figure 2.2 illustrates standard schematics for each type of interface.

Each SWC consists of a number of runnables, and each runnable may have access to a number of *ports*. The sender/receiver interfaces may have a 1:1 relation (e.g. one sender writing to 1 receiver), 1:N relation (e.g. on sender writing to multiple receivers) or an N:1 relation, (e.g. multiple senders writing to one receiver)[1].Figures 2.3 and 2.4 illustrate the schematic for each case.

[1]See [5] section 4.3.1.4 for more details.

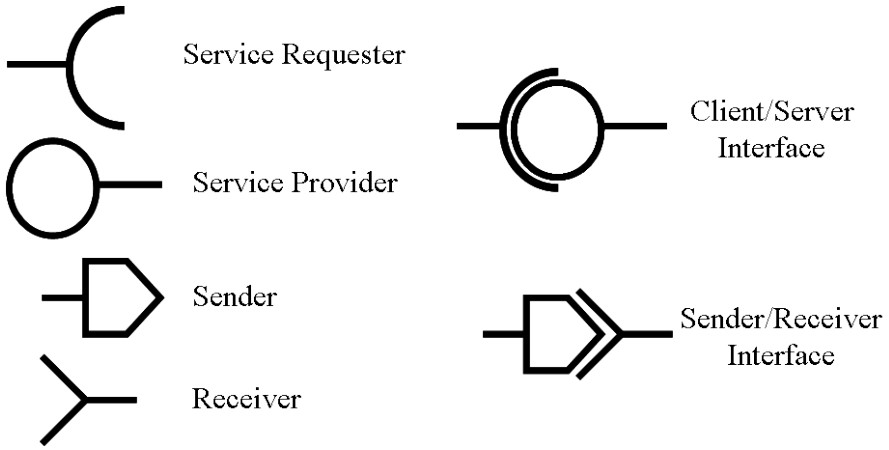

Figure 2.2: *Standard illustration of Sender/Receiver and Client/Server interfaces in AUTOSAR.*

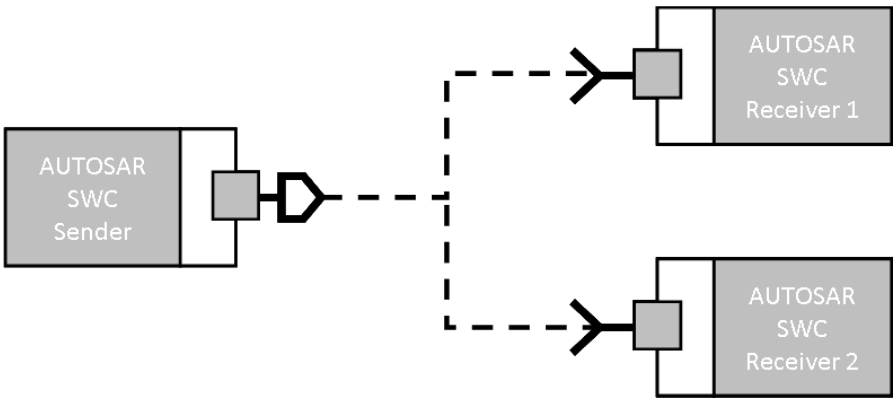

Figure 2.3: *High level illustration of data propagation among SWCs demonstrating a 1 to 2 (1:2) relation. The data provided by SWC-Sender is accessible by two SWCs, SWC-Receiver1 and SWC-Receiver2.*

2.1.3 VFB layer

Virtual Function Bus (VFB), is the communication concept that separates the communication mechanism between SWCs and the infrastructure beneath the application layer and amongst SWCs themselves. Communication amongst SWCs is done through the same mechanisms as between SWCs and the lower layers in the AUTOSAR stack[6]. Figure 2.5 demonstrates an arbitrary number of SWCs and the communication interfaces between them. From the SWC point of view, a communication is done via the VFB, thus, a SWC has no awareness of the communication end-point location, (i.e the receiving port) providing mobility among SWCs. In Figure 2.5 port F writes to two ports, D and B, establishing a 1:2 relation.

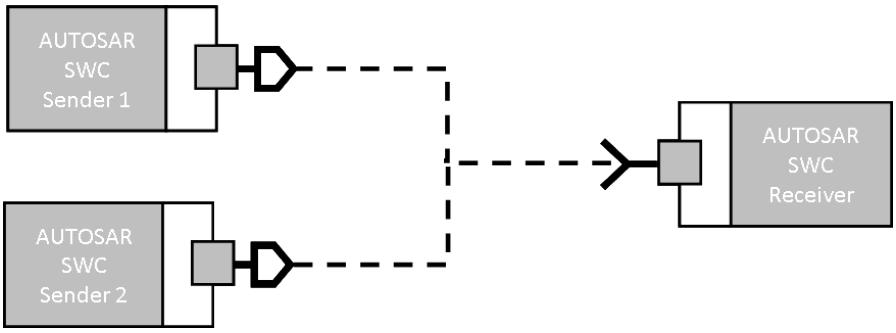

Figure 2.4: *High level illustration of multiple SWCs writing to 1 Receiver demonstrating a 2 to 1 (2:1) relation. Two SWCs, SWC-Sender1 and SWC-Sender2 both can provide data for SWC-Receiver. SWC-Sender1 and SWC-Sender2 might overwrite each others data.*

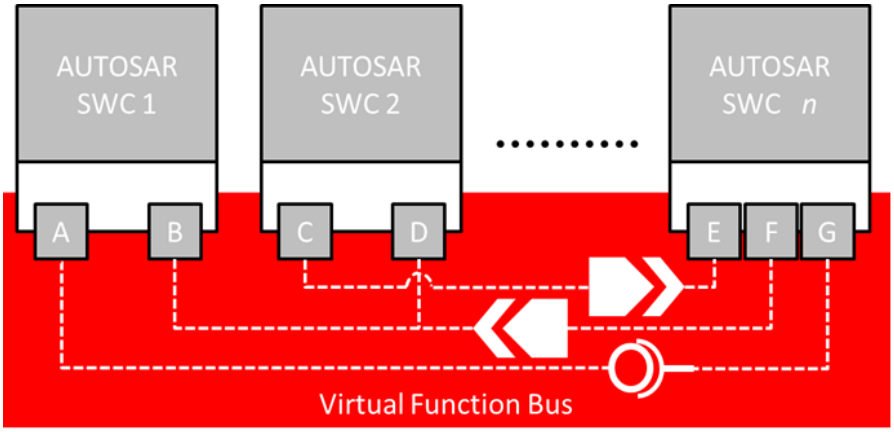

Figure 2.5: *High level illustration of Virtual Function Bus, (VFB), with a number of SWCs, interfaces and ports.*

2.1.4 RTE layer

The Run-Time Environment (RTE) is seen as the heart and the veins of the system which makes communication possible amongst SWCs. Also, the communication between SWCs and the underlying layers (Basic Software components and services) is done through the RTE layer. The RTE is generated for each ECU based on the ECU configuration settings, SWCs that are mapped to the ECU, and the underlying layers' settings and configuration. The RTE is responsible for running runnables, task execution and etc[5]. Figure 2.6 depicts the RTE within the VFB context. The communication between port B and C, is done through the RTE layer and *may or may not* require additional communication through the underlying layers. While the communication between two ECUs (i.e port D and E) conceptually goes through the VFB and *does* require communication through underlying layers.

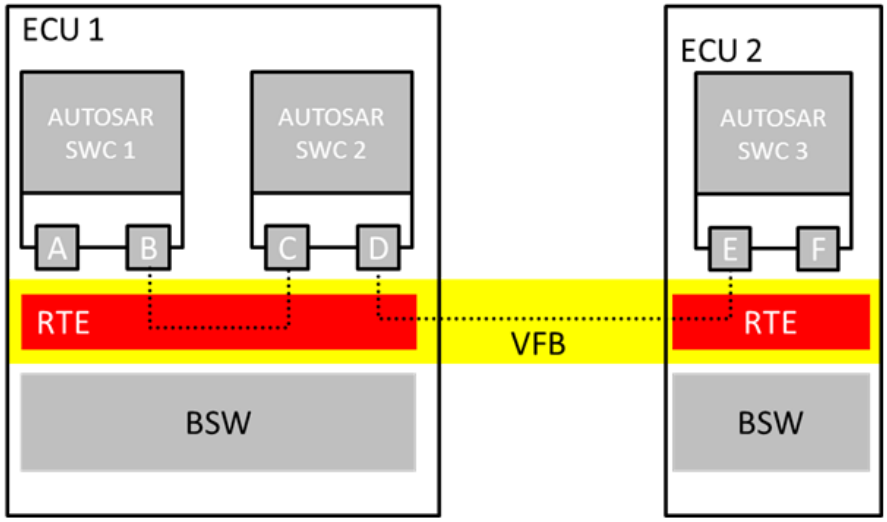

Figure 2.6: *Demonstration of inter-ECU and intra-ECU communication through RTE and VFB layers. Inter-ECU communication is handled conceptually through the VFB layer, while intra-ECU communication is handled only within the RTE of that ECU.*

2.1.5 BSW layer

The Basic SoftWare(BSW), as shown in Figure 2.1, consists of several component modules and services. It provides functionalities to applications through various standard/hardware-specific components and services. In terms of vehicle functionality, the BSW does not provide any specific job itself, rather, it is the engine of the AUTOSAR stack. Within the BSW layer, the "Services layer" consists of services such as communication protocols and memory management. The "ECU abstraction layer" makes ECU specific configurations transparent to higher layers. The "Microcontroller abstraction layer" contains standard communication interfaces between upper layers and the microcontroller registers and configuration. The "Complex Device Drivers", (CDD), brings in functionalities for direct hardware access to upper layers for specific purposes.

2.2 Multi-core support in AUTOSAR

As the computation demand is going higher and increasing in the automotive industry on one hand, and increasing clock frequency is not possible due to the need of low power consumption in ECUs on the other hand, the need of multi-core ECUs becomes more and more clear. Multi-core ECUs may be used for more processing power and redundancy in safety critical applications (as recommended in ISO26262). Multi-core support was first introduced in AUTOSAR release 4.0 in 2009. In this section, a brief introduction is given to aspects of AUTOSAR, such as modules and aspects, with respect to multi-core support.

2.2.1 Communication in multi-core AUTOSAR

Communication between SWCs is divided as the following:

$$\begin{cases} Inter-ECU \quad communication \\ Intra-ECU \quad communication \begin{cases} Inter-core \quad communication \\ Intra-core \quad communication \end{cases} \end{cases}$$

As the scope of this project is limited to ECU level architecture, inter-ECU communication is not discussed. Inter-core communication is extensively studied as it plays a critical role in future multi-core systems. Figure 2.7 illustrates a high level of all four cases of communication.

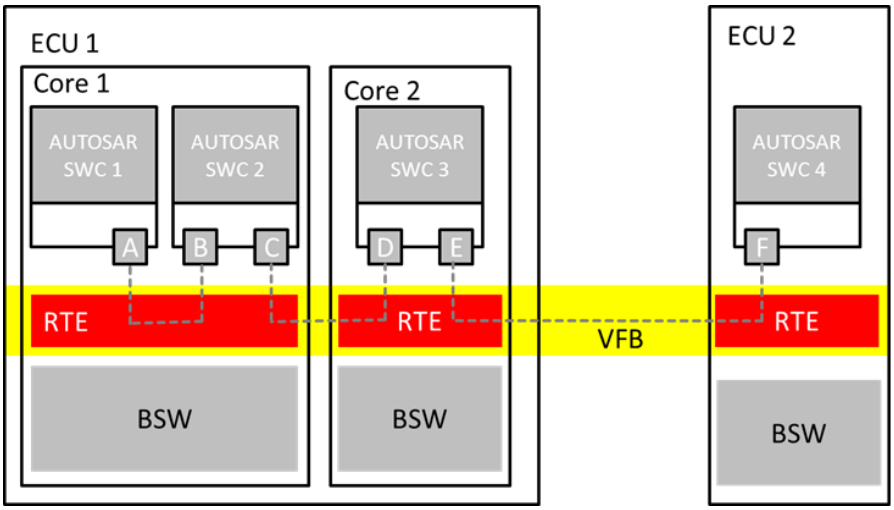

Figure 2.7: *High level demonstration of inter/intra core communication and inter-ECU communication. Ports A and B illustrate an intra-core communication, ports C and D illustrate a cross-core communication, and ports E and F illustrate an inter-ECU communication.*

2.2.2 The Operating System

The AUTOSAR Operating System, (OS) is a manipulation of the OSEK[7] operating system. Most of the general properties and attributes of OSEK apply to AUTOSAR-OS also. The rest of this section explains briefly some aspects of the AUTOSAR-OS.
As discussed in [8] Section 3.37, from an operating system point of view, there are three different main approaches to increase processing power within AUTOSAR. That is:

- One Operating system per core(e.g. A case in which multiple ECUs are merged into one ECU, each running on one core, individually. The drawbacks of this approach may appear in shortage in I/O and memory, complexity in core synchronization, computation overhead for shared resource handling and etc.)
- One operating system controlling more than one core(e.g. Running SWCs with parallel distributed algorithms).
- Having cores with no operating system(e.g. A core may be used to do simple tasks such as I/O without the need of an operating system).

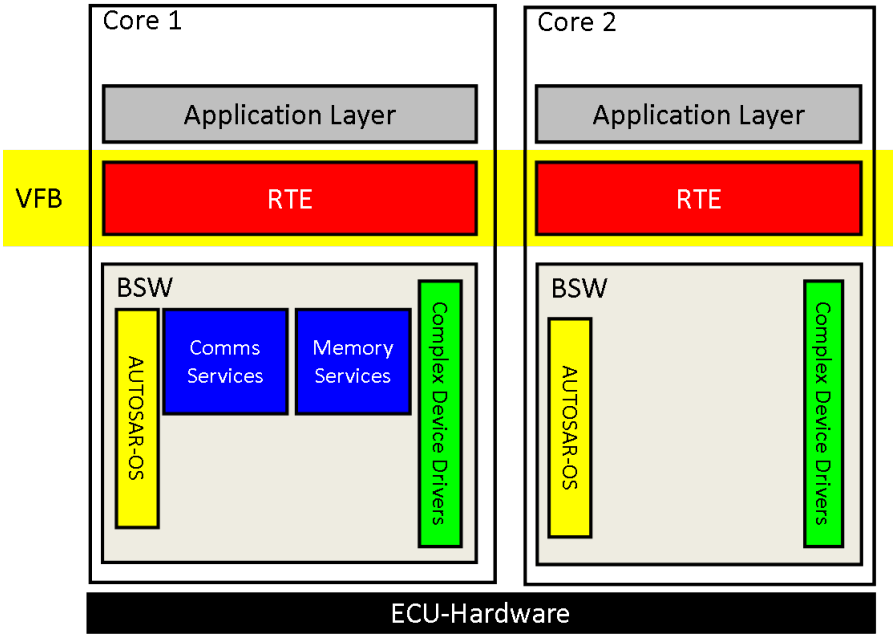

Figure 2.8: *High level demonstration of AUTOSAR layers on a dual-core ECU with one OS per core. Core1 is the master core and thus, Core2 has much less modules and services than Core1.*

The scope of this research is only within the first approach, i.e. one OS per core. Figure 2.8 illustrates the BSW on a dual-core ECU. The following gives a terminology on some aspects of the OS, that are used throughout this study:

- Tasks: Each OS has at least one task. Tasks may be "basic" or "extended". A basic task is a task which is terminated after completion. An extended task is a task that may be activated again after completion. Generally, the OS task is a basic task and all other periodic tasks are extended tasks[7]. Tasks may be activated periodically by the scheduler or by an event.
- Scheduler: Each OS has a scheduler which implements the scheduling policy among tasks. In AUTOSAR, static priority scheduling policy is used.

- Event: An event may be sent to the scheduler to activate a specific task. The scheduler will then activate the naming task if there is no other task with a higher priority to be executed. The scheduler itself has a higher priority than any other task.

2.2.3 Inter OS-Application communication

"OS Application" refers to a set of tasks, counters, Interrupt Service Routines(ISRs), and other OS modules that form an object together and hold an adherence together. Each OS has at least one OS-Application. Hence, in multi-core systems with one OS per core, each core has at least one OS-application. Within the context of this writing, OS-applications are used *only* to assign objects to a specific core. Thus, each core has one OS-application and every task, runnable, counter and etc shall belong to one, and only one OS-Application, which determines the core in which the object executes on. The "Inter OS Communicator"(IOC), is necessary and was introduced in [9] Section 7.10 for multi-core support. Since there is only one OS per core, inter-os communication is seen as inter-core communication[2]. We focus on two major purposes and functionalities of the IOC:

- The IOC is responsible for guaranteeing data consistency between two cores in inter-core communication.
- The IOC provides a mechanism for event triggering between the two cores while keeping the two OSs synchronized.

2.2.3.1 Data consistency in IOC

Figure 2.9 depicts a configuration of SWCs emphasizing a 2:1 relation between SWC-1, SWC-2 and SWC-3, in which SWC-1 and SWC-2 try to send a data to the same port on SWC-3. This means a memory location is shared among the three SWCs. In such a configuration, if SWC-1 and SWC-2 attempt to send a data to SWC-3 simultaneously, SWC-1 might overwrite the data of SWC-2, and vice-versa. Hence, a mechanism is needed to support mutual exclusion between the two SWCs(i.e SWC-1 and SWC-2). The IOC may be implemented in various ways to support mutual exclusion, however, AUTOSAR standard suggests using "spinlocks". Within this project, all mutual exclusions are handled by using spinlocks. Lakshmanan *et al* in [10] have discussed various approaches for supporting data consistency(and mutual exclusion in general) within the AUTOSAR context. While Figure 2.9 illustrates a high level structure of how the IOC handles data consistency between two cores, Figure 2.10 depicts the sequence diagram of the same configuration[3]. Figure 2.9 shows that SWC-1 and SWC-2 write to their corre-

[2]Although the standard does allow multiple OS-Applications on one core, we use OS-application as an identification for objects to determine which core they belong to and thus, having such an approach, requires not to have multiple OS-Applications per core.

[3]Labels used in Figures 2.9 and 2.10 are *not* AUTOSAR standard. Shortened names have been used for ease of readability.

sponding RTE layer. The RTE layer knows that destination port is shared among other SWCs and hence, it will request a write permission to its corresponding OS and the OS passes the request to the IOC module. So far, all events execute in parallel on both cores and a race condition occurs between the two SWCs trying to access the IOC. The IOC grants write permission to the RTE by an acknowledgment. The IOC grants write permission to each RTE layer such that it guarantees data consistency among the two SWCs. Each RTE layer writes its data to the destination port after it has received IOCs acknowledgment. Figure 2.10 shows the sequence diagram of the same configuration as in Figure 2.9. The following describes the chain of events in Figures 2.9 and 2.10:

1. **RTE_Write_<Runnable>_<Port>():** A runnable in a SWC writes sends a data to the RTE layer with the destination port.

2. **OS_Request_IOC_Write_<Runnable>_<Port>():** The RTE requests write permission on the destination port from the IOC module. Note that a race condition may occur if multiple calls are made to the IOC module.

3. **Ack_IOC_Write_<Runnable>_<Port>():** IOC grants write permission to the RTE layer by acknowledging its request.

4. **RTE_Write_Buffer_<Runnable>_<Port>():** The RTE writes the data to the memory buffer location, which will be read by the receiver.

5. **Ack_Completion:** The RTE acknowledges the IOC that it has finished its job and now releases the IOC module.

6. **RTE_OK:** The RTE acknowledges the SWC that the write operation has completed successfully and the data is available to the reader.

Note that reading the data by SWC-3 has no impact on the communication mechanism whatsoever. How SWC-3 reads the data and how frequently it reads the data does not affect the concern of data consistency and hence, we do not address the complete chain of send and receive. Figure 2.10 illustrates the sequence diagram of a cross-core communication with guaranteed data consistency(See [9], Section 7.10.3.1)[4]

[4]As it is stated in the AUTOSAR OS Specification [9] page 108, on 1:1 relations, data consistency is not necessary. However, the current implementation done by ArcCore[11] does not follow this approach and the RTE generated still acquires the lock and so assures data consistency.

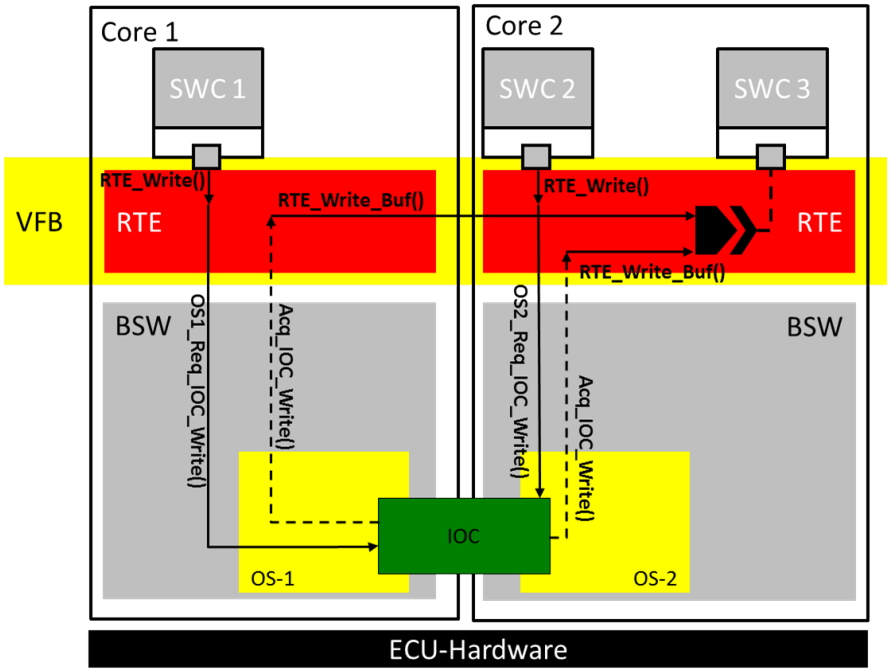

Figure 2.9: *High level demonstration of data consistency handling between two cores through the IOC service. The IOC has to grant write permit to SWC-1 and SWC-2 since they try to write to a shared memory location, before any data is actually written to SWC-3.*

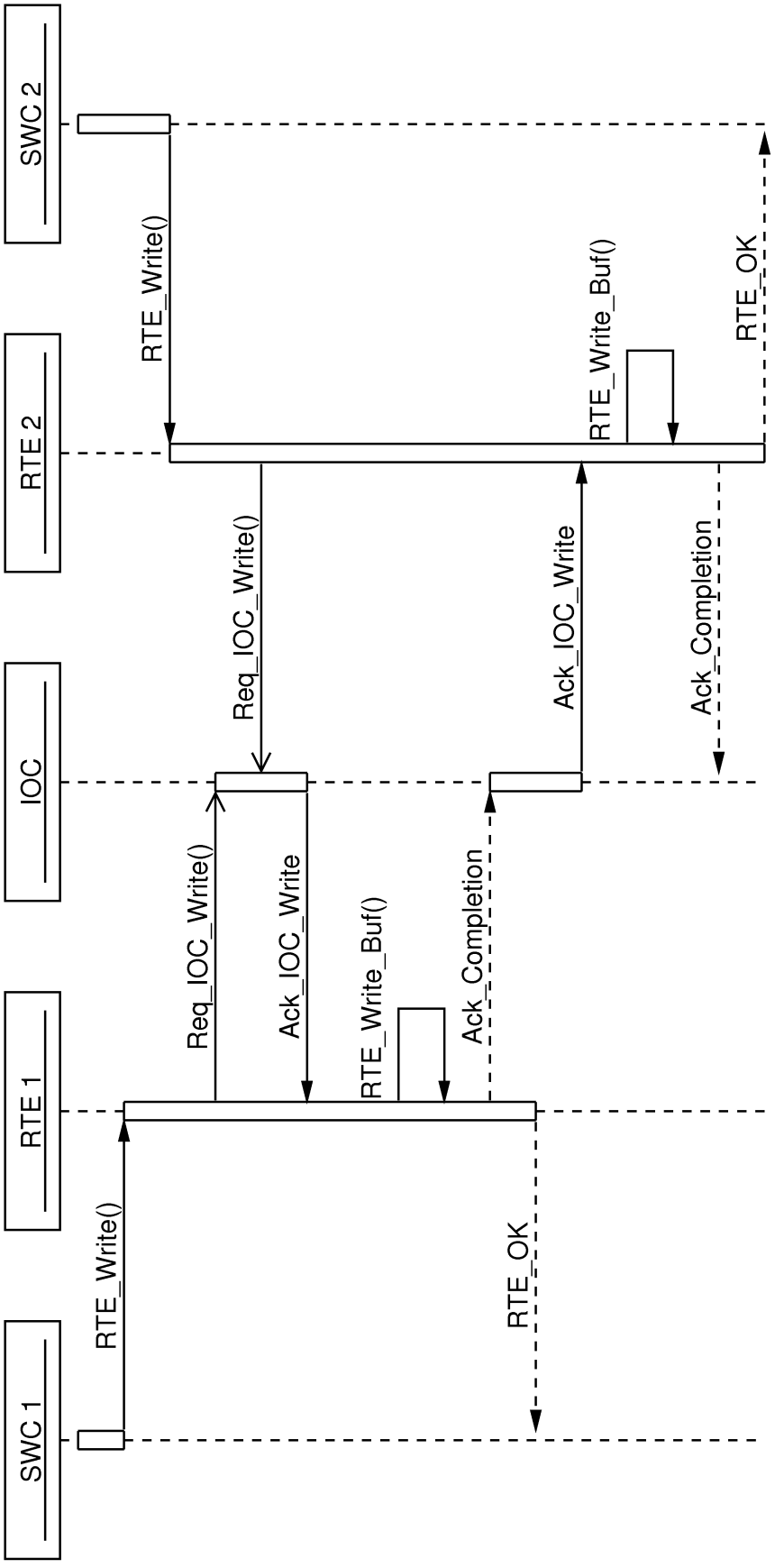

Figure 2.10: Sequence diagram demonstrating data consistency handling between two cores through the IOC service. The IOC has to grant write permit to SWC-1 and SWC-2 since they try to write to a shared memory location, before any data is actually written to SWC-3.

2.2.3.2 Inter-core notification in IOC

In a sender/receiver communication, a sender may optionally notify the availability of new data to the receiver. There are various ways for implementing such a mechanism depending on the hardware architecture. However, the standard suggests using interrupts. Upon completion of data write, the sender shall trigger an "event" to the task in which the corresponding receiver port is running on. The event is then passed to the "scheduler", which will eventually activate the receiving task. As mentioned in Section 2.2.2 each OS has its own scheduler, and thus, accessing the scheduler of another OS shall be handled such that it will not cause any affect on the other scheduler. Figure 2.11 illustrates a configuration of inter-core sender/receiver communication with notification support. As in Figure 2.11, if SWC-1 and SWC-2 intend to send a data to SWC-3 with notification simultaneously, a race condition may occur in accessing the scheduler. Therefore, the notification shall be handled through the IOC. Here, accessing the scheduler is accessing a shared resource. Moreover, another rationale supporting the need of the IOC, is that even if the notification is handled independent of the scheduler, since each core has an OS of its own and the OS has its own Interrupt Vector Offset Register (IVOR), again a conflict may occur if the notification is not handled via the IOC. Having said that, in essence, an inter-core notification problem is a shared resource handling problem, and it is handled in the same manner of guaranteeing data consistency (i.e using spinlocks when acquiring the shared resource).

Figure 2.11 shows the high level configuration that SWC-1 and SWC-2 write a data to SWC-3. Both SWCs use the inter-core notification mechanism. Both SWCs, notify the destination port once data write has completed and is available to the receiver. The data write mechanism(i.e. *RTE_Write_Buf()*) follows the same mechanism as explained earlier in Section 2.9, but it is not shown in Figures 2.11 and 2.12 for simplicity. Each SWC sets an *event* to the OS on its core, and hence communicates to its scheduler. However, the scheduler module on Core1 realizes that destination port resides on another OS module, and hence, requires access to the scheduler of the destination port(i.e. Scheduler2). Once Scheduler1 has acquired access to Scheduler2, it sets an event on it by an IVOR call(i.e. *NotifyCore()*). Note that similar to the IOC method for assuring data consistency, the IOC uses spinlocks to handle the race condition between SWC1 and SWC2. Figure 2.12 shows the sequence diagram of the same configuration in Figure 2.11. The following describes the chain of events in Figures 2.11 and 2.12:

1. **RTE_Write_<Runnable>_<Port>():** A runnable in a SWC writes sends a data to the RTE layer with the destination port.

2. **RTE_Write_Buffer_<Runnable>_<Port>():** The RTE writes the data to the memory buffer location, which will be read by the receiver.

3. **Set_Event():** The RTE has to notify the availability of new data to the receiver. This is done by triggering an event to the relevant OS.

4. **IOC_Request_Scheduler():** The OS requests access to the scheduler. This results in a race condition if two cores try to access the scheduler simultaneously. Since the scheduler is a shared resource, it needs to be handled via the IOC.

5. **IOC_Acquire_Scheduler:** The IOC grants scheduler access to the OS. The OS can then make changes to the scheduler, or access another scheduler if needed. In Figure 2.12 OS1 needs to access another scheduler while OS2 does not need to do so.

6. **NotifyCore():** The NotifyCore() function is implemented as an IVOR. It is used to interrupt and modify the scheduler on another core. For example, Scheduler1 has to notify a modification to Scheduler2, since the receiving port(i.e. SWC-3) is running on Core2. This illustrates that both schedulers are seen as *shared resource*, despite the necessity of a cross-core notification. The reasoning from the scheduler point of view, is that the scheduler is not aware that a receiving event will require a cross-core notification or not.

7. **Ack_Completion:** The OS acknowledges the IOC about completion of scheduler modification. The scheduler is now free.

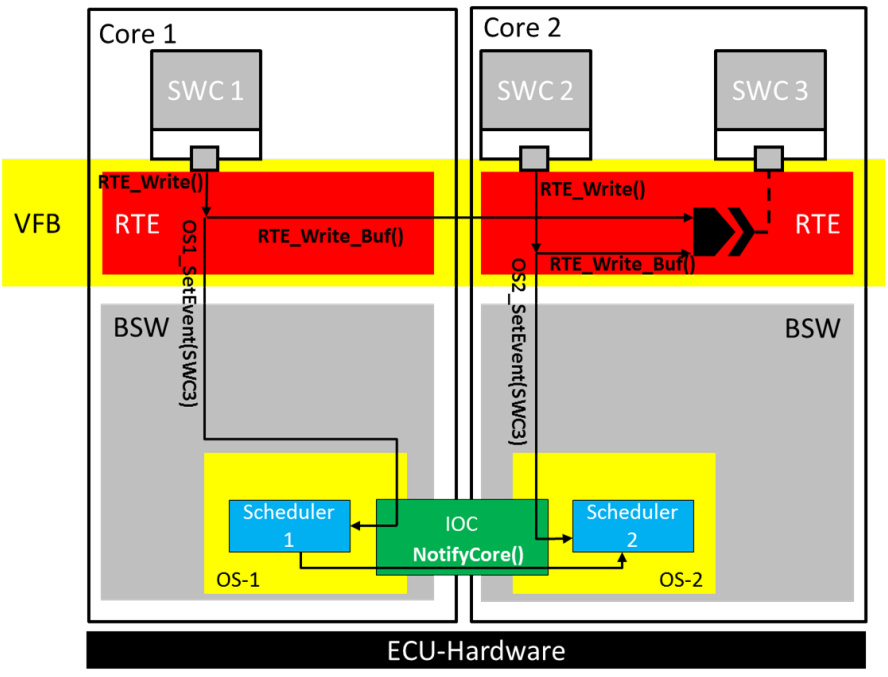

Figure 2.11: *High level demonstration of inter-core, data-write notification through the IOC service. In such a case, OS-1 needs to access the scheduler on another OS, and this results in multiple processes(SWC1 and SWC2) accessing a shared resource(Scheduler 2).*

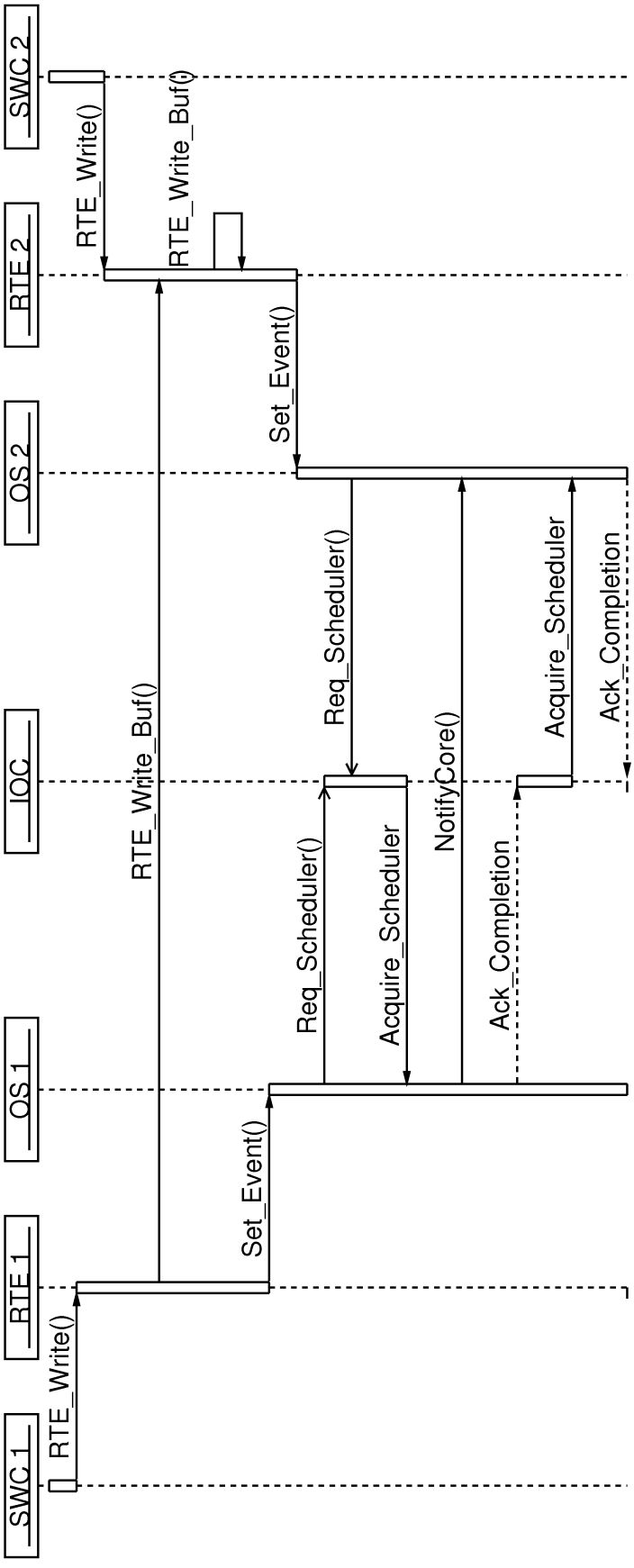

Figure 2.12: *Sequence diagram demonstrating inter-core, data-write notification through the IOC service. In such a case, OS-1 needs to access the scheduler on another OS, and this results in multiple processes(SWC1 and SWC2) handling a shared resource(Scheduler 2).*

3

Test and Development Environment

This chapter describes the development environment, both software and hardware. In order to simulate an AUTOSAR stack, we first need an AUTOSAR software and a hardware to execute the software on. Among the AUTOSAR component builders such as Vector, Davinci and ArcticStudio[11], we chose ArcCore because we already had access to an implementation with multicore support based on AUTOSAR 4.0[1] developed in ArcticStudio and provided by Ecore. The hardware target is an MPC5517E micro-controller on the MPC5510EVB evaluation board. The following sections explain the development environment. Section 3.1 describes the software tools and environment and Section 3.2 briefly describes the target hardware.

3.1 Software environment

This section describes the software tools and environment that are used through the project. That is, the development environment, the debugging and analysis tools and environment, and hardware communication software and tools.

3.1.1 ArcticStudio and ArcCore

ArcticStudio is an eclipse based, free and open-source[2] Integrated Development Environment (IDE) providing SWC, RTE and BSW module code generators. ArcCore, which is the heart of the BSW modules, drivers and interfaces supports a number of target hardware and evaluation boards which is freely available. Through this project, different configurations and implementations of AUTOSAR

[1] Current version is preliminary evolved version of 3.1 with some functionalities of version 4.0

[2] Some features of the IDE, basically, code generators are not freely available as the time of this writing.

are generated through ArcticStudio and evaluated. The ArcticStudio uses "gcc" compiler and is capable of directly generating a binary file for the target hardware by using the target hardware cross-compiler directly within the IDE. ArcticStudio is *not* a debugging tool(it cannot connect to the hardware directly). Hence, the generated binary file (*.elf) has to be uploaded on the hardware using other tools. Also, other tool and software are needed for debugging purposes.

3.1.2 CodeWarrior

CodeWarrior 5.9[12] is a fairly simple IDE by Freescale[13] for development on Freescale micro-controllers. Since we are using a Freescale product, we use CodeWarrior cross-compiler. The procedure is such that ArcticStudio uses gcc compiler, then it is linked to CodeWarrior for generating the elf file. Hence, the user does not need to do anything directly within the CodeWarrior IDE. The generated binary file(*.elf) is uploaded to the target hardware using "NEXUS DEBUGGER for CodeWarrior". NEXUS DEBUGGER's communication with hardware is done through the "NEXUS/USB Multilink JTAG Debugger".

3.1.3 FreeMASTER

One of the difficulties in parallel real-time software development, is detecting run-time bugs that may or may not appear in runtime due to the realtime and parallel nature of software and cause an abnormal behavior in software. Such an abnormality is extremely difficult to detect using static debuggers[3] simply because by *stopping* and *starting* the software, the software behavior changes and the bug will not appear. Moreover, in real-time applications, sometimes it is necessary to see the system behavior in runtime, which again is not possible with static debuggers. The problem is solved by logging(which in essence modifies the software) or by using realtime debuggers. FreeMASTER[15] is an open-source run-time debugging tool, also provided by Freescale, which makes it possible to fetch and modify variables in realtime by communicating through RS232 channel or the CAN bus with the target hardware using an on target driver. Through the communication with target, the desired variables are sampled, read and modified in a user-defined rate and downloaded to the PC. FreeMASTER also provides a graphical user interface for data visualization. Through this project, FreeMASTER is used both for debugging and monitoring purposes to analyze our test cases. In easy words, FreeMASTER is seen as a digital oscilloscope which can probe, modify, and plot desired variables on PC. In this section a brief explanation is given on how FreeMASTER modularities are embedded within the AUTOSAR stack. Generally, FreeMASTER has two parts:

[3]Most common debuggers are static debuggers which execute the program step by step or by evaluating a condition such as breakpoints. Dynamic debuggers instead do not need to stop the execution flow, and it is possible to monitor some states of registers, stack pointer and etc *while* the program flow is executing[14].

- The PC based FreeMASTER client running on PC.
- The on target driver and communication software.

Once the on target FreeMASTER software is embedded within the AUTOSAR stack and is running on the target, the on target software could be controlled and monitored from the PC. However, FreeMASTER has its own limitations and does not come without a cost. Basically, embedding FreeMASTER within the original software is *altering* the original AUTOSAR software that was intended to be monitored. Hence, it is the designer's responsibility to design and implement the monitoring on target software with least overhead cost, and to estimate the cost caused by software manipulation. Figure 3.1 depicts a schematic of the development environment. As it seen in Figure 3.1, "FreeMASTER Driver and Application" are implemented on the RTE and Application layers. This depends on which layers are intent to be monitored by FreeMASTER. A detailed explanation is given on how to use FreeMASTER in Appendix B.

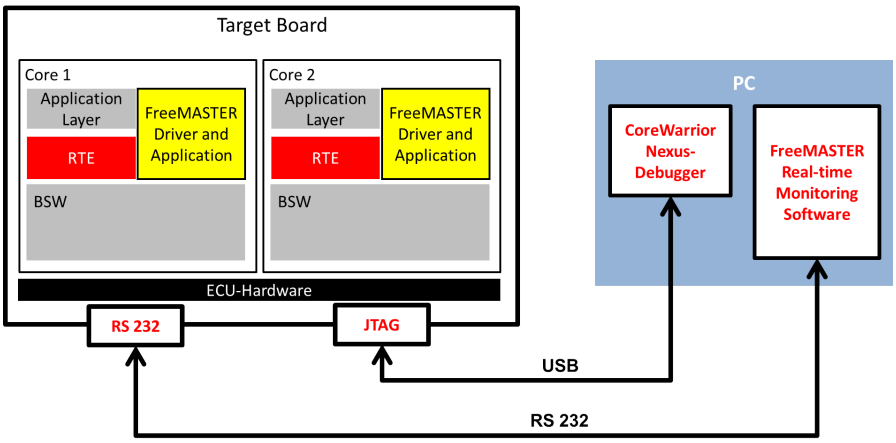

Figure 3.1: *Schematic view of development environment. The software is uploaded and debugged using the JTAG connector through USB, while the on target software is monitored in realtime by FreeMASTER PC client via RS232 channel.*

Figure 3.2 shows the complete tool chain from code generation through running an AUTOSAR software on the hardware. The following describes development steps as shown in Figure 3.2:

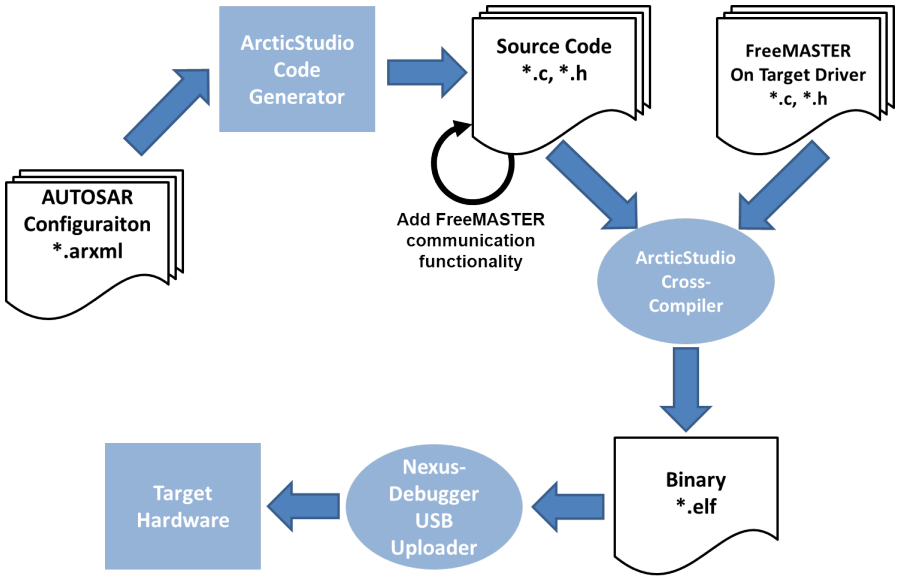

Figure 3.2: *Complete software development tool chain.*

1. An AUTOSAR software design is made.
2. Using ArcticStudio code generators, the source code is generated from AUTOSAR configuration description files, “ECU Extract Builder”, “SWC Builder” and “ECU Configuration”.
3. FreeMASTER driver and protocol files are added to the source code, according to the FreeMASTER implementation design.
4. FreeMASTER monitoring application is added to AUTOSAR source code and necessary changes are made.
5. The binary file is generated within ArcticStudio using the CodeWarrior cross-compiler.
6. The binary file (*.elf) is uploaded to the target hardware using Nexus Debugger USB uploader.

3.2 Hardware environment

The hardware used for evaluation in this project is an MPC5510EVB[16] evaluation board equipped with an MPC5517E[17] micro-controller from Freescale. As explained in [17][4] and [18], the MPC551x micro-controllers family are heterogeneous dualcore processors. Software analysis is not independent from hardware and thus, it is necessary to understand the hardware architecture which the software is executing on. The following briefly describes some aspects of the naming micro-controller. The MPC551x micro-controller consists of two non-identical cores, Core1 and Core2. Core2 has less processing power and features than Core1 and thus, it is known as an auxiliary node, or I/O node. Figure 3.3 shows some of the key differences between Core1 and Core2:

- Core1 has a bigger instruction buffer than Core2.
- Core1 has an independent data bus access to flash memory as shown in Figure 3.3, whilst data communication in Core2, is handled via shared "instruction and data" bus through crossbar.
- Core1 has more branching capabilities than Core2, see [17] for more details.

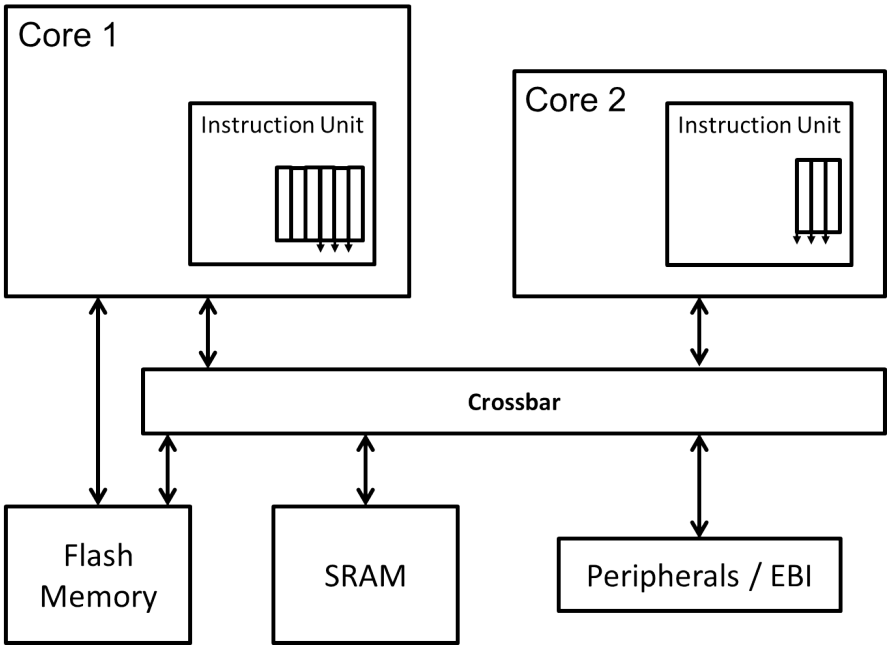

Figure 3.3: *The dualcore MPC551x micro-controller block diagram. Direct access to flash memory and a bigger instruction buffer brings more processing power to Core1 compared to Core2.*

[4] See Sections 10 and 11.

4

Inter-core versus intra-core communication

This chapter studies the communication time in an inter-core communication and an intra-core communication. The additional overhead introduced in cross-core communication is an interesting factor from a multi-core perspective and therefore it is investigated. Hence, two simple configurations of SWCs on the AUTOSAR stack were designed and implemented. One configuration with an *intra-core* communication and another, with an *inter-core* communication is developed. In the following of this chapter, in Section 4.1 the scenarios and test configurations are explained, Section 4.2 describes the intra-core communication test and Section 4.3 describes the inter-core communication test. The results and discussions are presented in Section 4.4.

4.1 Test description

The communication mechanism used in the two tests is *intra/inter core communication with notification* as described earlier in Section 2.2.3.2. The communication time is measured by using FreeMASTER and reading the "Timebase" register on the MPC5510x micro-controller. The timebase register provides system clock resolution. An example for using the timebase register is available in [19]. The overhead of actual *reading* the timebase register, is neglected.

Figures 4.1 and 4.3 illustrate a high level structure of intra-core and inter-core communication as implemented in the software. The configuration consists of two software components, SWC1 and SWC2. As the dotted arrows show, each SWC, consists of one runnable, Runnable1 and Runnable2. Runnable1 is mapped to TaskA and Runnable2 is mapped to TaskB. Both TaskA and TaskB have the same priority. The "communication time" refers to a time that a data is written to the output port (i.e. *Writing point* in Figure 4.2) until the data is read by the receiving port (i.e. *Reading point* in Figure 4.2). Listing 4.1 shows the source code

for Runnable1 and Runnable2. In both test scenarios, the source code for both runnables is the same and the difference is in the RTE layer. Runnable1 makes an implicit "RTE_IWrite" call which will write the data on the outgoing port. Runnable2 makes an implicit "RTE_IRead" call which reads the written data. As mentioned earlier, the "communication time" refers to the time difference between the "RTE_IWrite" call and the "RTE_IRead" call.

4.2 Intra-core communication

In this section, the intra-core communication test is described in details. In Figure 4.1, both TaskA and TaskB are mapped to "Scheduler1" and run on Core1, and so Core2 is idle. Figure 4.2 illustrates the sequence diagram of the same configuration as shown in Figure 4.1. The following describes the sequence of events based on Figures 4.1 and 4.2.

1. **Activate_Runnable1:** OS1 activates Runnable1 and thus, TaskA is activated and Runnable1 starts its execution.

2. **RTE_IWrite_Runnable2:** Runnable1 writes an 8 bit data on its port[1].

3. **RTE_IWrite_Buffer():** After "data_write" completion the RTE activates Runnable2 by running "SetEvent(Runnable1)".

4. **GetSpinlock(OS):** Accessing the scheduler is handled through the IOC. "GetSpinlock()" is interpreted as *request IOC service*, and "GetSpinlock(OS)" is interpreted as *Request Scheduler*. The function continues examining the availability of OS lock in a busy loop until the lock is available.

5. **Irq_Save:** OS1 has locked the OS lock and hence it is granted access to Scheduler1, and Scheduler1 is activated. The *scheduler* task itself, is an IVOR. The intention is that the scheduler shall have highest priority and hence, it is implemented as an interrupt routine. Irq_save() disables all interrupts while the *critical* section is executing(i.e. Scheduler1). This is needed in order to avoid deadlocks and the reason lies behind the nature of spinlocks.

6. **Scheduler:** The scheduler starts its execution which will examine which task shall start next.

7. **Activate_Runnable2:** The scheduler activates the next task that should start its execution and so it activates Runnable2.

8. **ReleaseSpinlock():** The execution of the critical section is finished and the lock is released.

[1]The "RTE_Write" handling through IOC as described in Section 2.2.3.2 is not shown for simplicity.

9. **Irq_Restore:** The interrupts are enabled since the critical has ended.
10. **RTE_IRead_Runnable1:** Runnable2 is actually activated and reads the written data to its port by Runnable1.

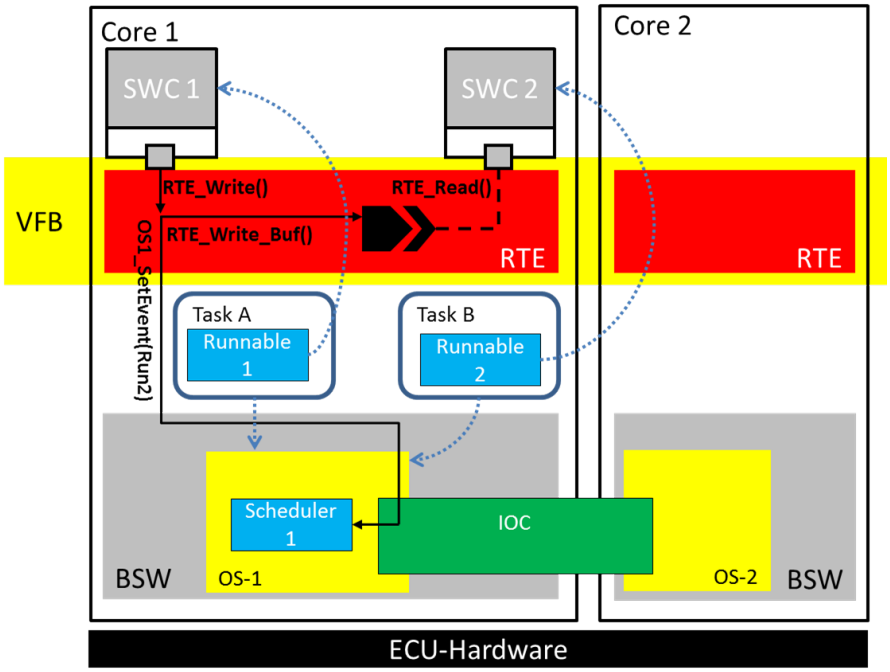

Figure 4.1: *High level illustration of implemented software for "intra-core communication with notification" measurement. The communication time is measured from the time SWC1 writes a data, and the data is read by SWC2.*

```
1 void Runnable1(){
    static UInt8 val = 11;
3   Rte_IWrite_Runnable1_senderPort_dataElem_1(val);
  }
5
  void Runnable2() {
7   static UInt8 val = 33;
    UInt8 val = Rte_IRead_Runnable2_senderPort_dataElem_1();
9 }
```

Listing 4.1: *Source code for Runnable1 and Runnable2 as is in the implementation on application layer. Runnable1 writes "val" to "senderPort" via an "RTE_IWrite". The RTE_IWrite, will send an event to TaskB which will eventually activate Runnable2. Runnable2 reads "val" on "senderPort" by making an RTE_IRead call.*

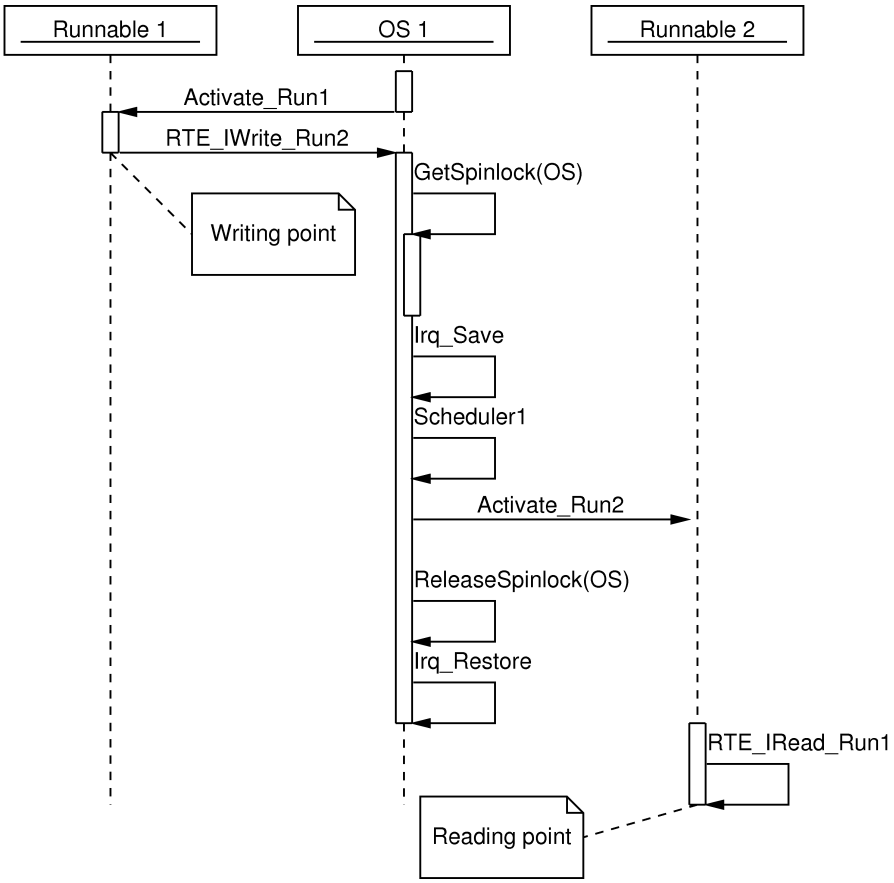

Figure 4.2: *Sequence diagram illustration of implemented software for "intra-core communication with notification" measurement. Execution time is measured between the "Writing point" and the "Reading point".*

4.3 Inter-core communication

The intra-core communication configuration is altered such that SWC1 and SWC2 will establish an inter-core communication and the communication time is measured. Figure 4.3 depicts the high level structure of the implemented software and Figure 4.4 shows the sequence diagram of the same configuration. The following describes the sequence of events for in an inter-core communication based on Figures 4.3 and 4.4.

1. **Activate_Runnable1:** OS1 activates Runnable1 and thus, TaskA is activated and Runnable1 starts its execution.
2. **RTE_IWrite_Runnable2:** Runnable1 writes an 8 bit data on its port[2].
3. **RTE_IWrite_Buffer():** After "data_write" completion the RTE activates Runnable2 by running "SetEvent(Runnable1)".

[2]The "RTE_Write" handling through IOC as described in Section 2.2.3.2 is not shown for simplicity.

4. **GetSpinlock(OS):** Accessing the scheduler is handled through the IOC. "GetSpinlock()" is interpreted as *request IOC service*, and "GetSpinlock(OS)" is interpreted as *Request Scheduler*. The function continues examining the availability of OS lock in a busy loop until the lock is available.

5. **Irq_Save:** OS1 has locked the OS lock and hence it is granted access to Scheduler1, and Scheduler1 is activated. The *scheduler* task itself, is an IVOR. The intention is that the scheduler shall have highest priority and hence, it is implemented as an interrupt routine. Irq_save() disables all interrupts on the core it is called while the *critical* section is executing(i.e. Sceduler1). This is needed in order to avoid deadlocks and the reason lies behind the nature of spinlocks.

6. **Scheduler1:** The scheduler starts its execution which will examine which task shall start next.

7. **NotifyCore()/IVOR:** Scheduler1 tries to activate "TaskB", but, it does not have access to TaskB since it belongs to Core2. Thus, Scheduler1 shall notify the *owner* of TaskB about the receiving event. Hence, an interrupt is triggered, and any running task on Core2 is preempted. The interrupt activates "Scheduler2".

8. **Scheduler2:** Scheduler2 is activated and examines the event sent by Scheduler1.

9. **Activate_Runnable2:** Scheduler2 activates the next task that should start its execution and so it activates Runnable2.

10. **Ack:** The Notification message is acknowledged. According to [9], scheduler execution is a synchronization point. Hence, an acknowledgment message is sent back to the calling core.

11. **ReleaseSpinlock():** The execution of the critical section is finished and the lock is released.

12. **Irq_Restore:** The interrupts are enabled since the critical has ended.

13. **RTE_IRead_Runnable1:** Runnable2 is actually activated and reads the written data to its port by Runnable1.

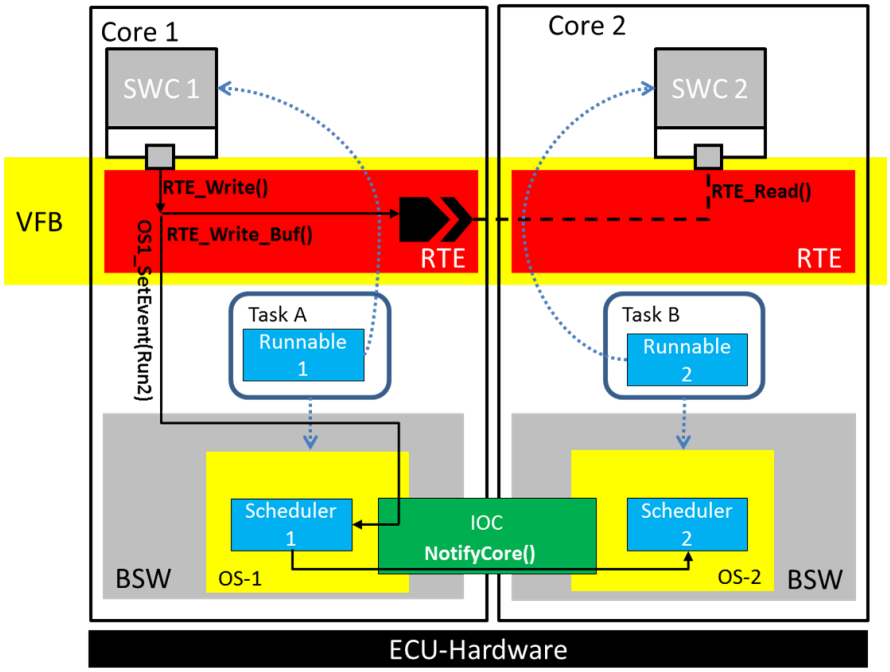

Figure 4.3: *High level illustration of implemented software for "inter-core communication with notification" measurement. The communication time is measured from the time SWC1 writes a data, and the data is read by SWC2.*

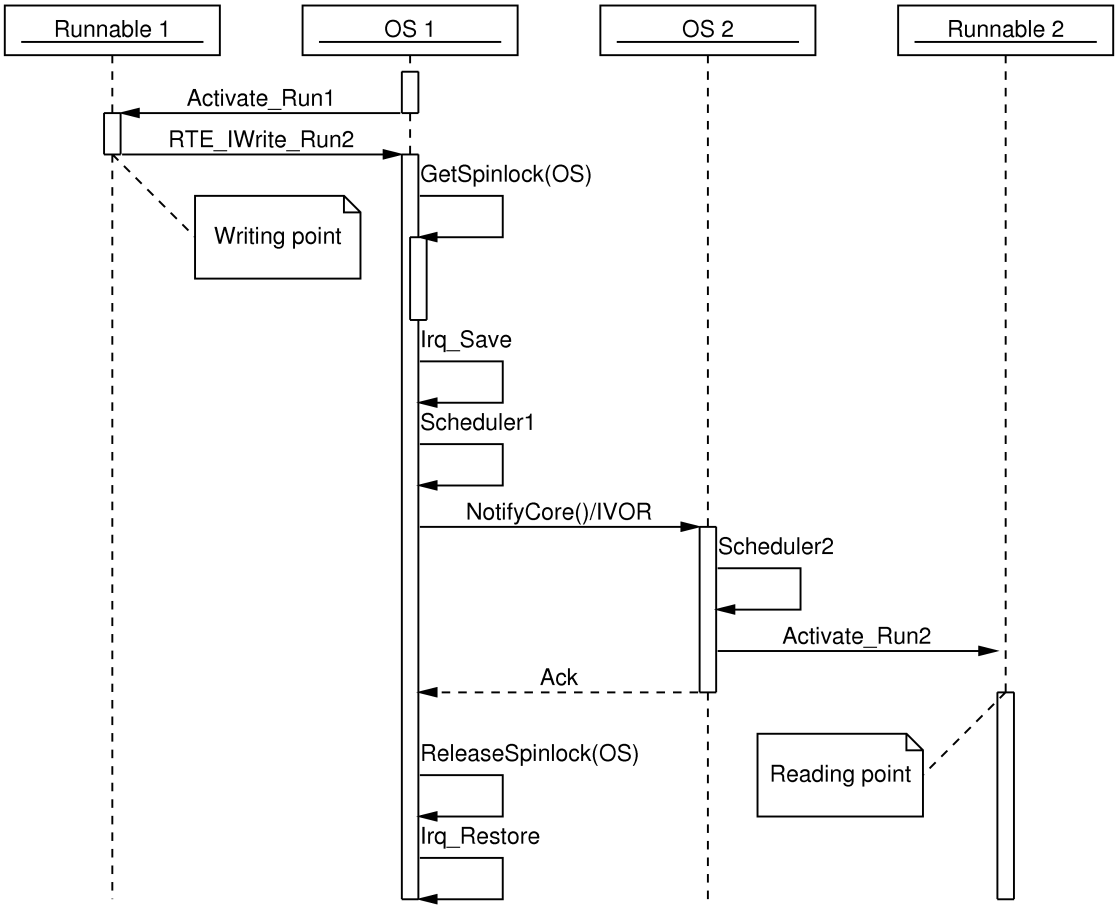

Figure 4.4: *Sequence diagram illustration of implemented software for "inter-core communication with notification" measurement. Execution time is measured between the "Writing point" and the "Reading point".*

4.4 Communication time measurement results

Measuring the time difference on inter-core communication between *writing point* and *reading point* according to Figures 4.2 and 4.4 is not possible using the Timebase register since it is not accessible on Core2(See [17] Chapters 10 and 11). In order to overcome this problem in a simple manner, the two designs were modified such that SWC2 will write back a dummy data on SWC1 through another interface and SWC1 will read back the data. Figure 4.5 illustrates a high level sequence diagram of the modified design in intra-core communication. The inter-core communication design is modified in a similar manner. The "Round Trip Time" (RTT) is introduced as the time between the "Writing point" and the "Reading point" according to Figure 4.5. It is assumed that the time spent in Runnable2 between "RTE_IRead_Run1" and "RTE_IWrite_Run1" in Figure 4.5 is negligible. Thus, intra-core communication time and inter-core communication time is the RTT divided by two. Table 4.1 shows the measured RTTs in intra-core communication and inter-core communication time. The last column in Table 4.1 shows the intra-core and inter-core communication time normal form over intra-core communication time. As it is seen in the last column, a cross-core communication is 54% more expensive than an intra-core communication.

Communication Architecture	**Communication Time** (μs)	**Normalized**
Intra-core	$\frac{110}{2} = 55$	1
Inter-core	$\frac{170}{2} = 85$	$\frac{85}{55} \approx 1.54$

Table 4.1: Intra-core versus Inter-core communication time. The results show that the inter-core communication time is 54% higher than the intra-core communication time.

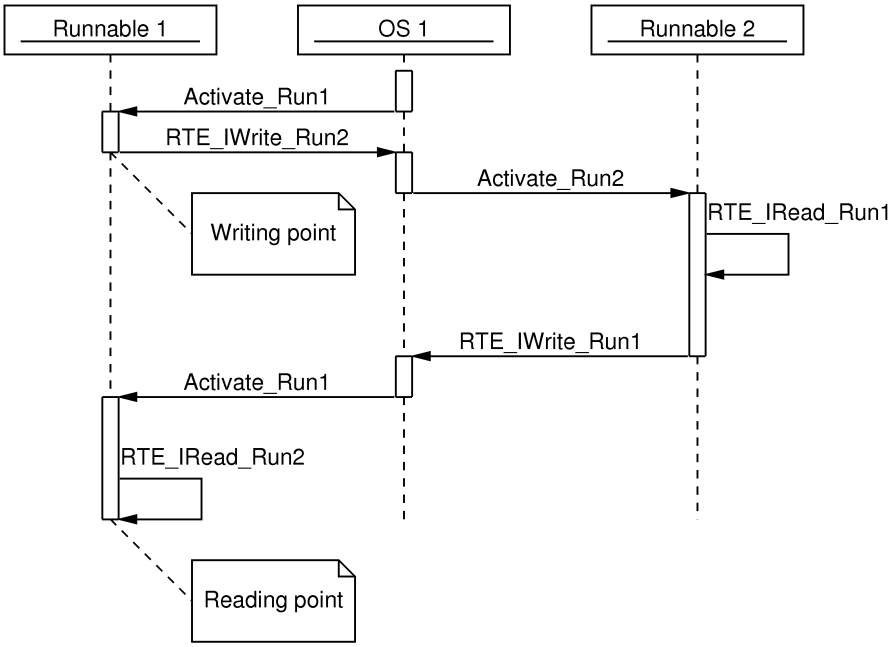

Figure 4.5: *High level sequence diagram of modified SWC1 and SWC2 communication mechanism. In this modified scenario, SWC2 writes back a dummy data to SWC1 so that time measurement is done always on Core1.*

5

Speedup gain on an AUTOSAR 4.0 implementation with dualcore support

A fundamental question that shall be answered in addressing multi-core systems is the processing power gained by a multi-core system compared to a single-core system. This chapter, studies the processing power gained by introducing multicore support in AUTOSAR 4.0. Later, preliminary studies are carried out towards a development of a theoretical model of speedup gain, based on the multi-core AUTOSAR design requirements and implementation factors. The theoretical model takes into account hardware specific factors specifically considering the heterogeneous design of the MPC551x microcontroller family. This chapter is structured as the following: Section 5.1 gives an introduction to speedup and describes the terminology. Section 5.2 describes the test scenarios for measuring speedup gain and explains the findings and results. Section 5.3 illustrates how the theoretical model for speedup gain is developed based on Amdahl's law.

5.1 Introduction to speedup and Amdahl's law

In parallel programming, the computational capacity gained by having multiple cores is known as "speedup". Speedup, is introduced according to Equation 5.1.

$$S_N = \frac{T_1}{T_N} \tag{5.1}$$

Where:

- N is the number of cores.
- T_1 is the execution time of the program having only one core.

- T_N is the execution time of the program having multiple cores.
- S is speedup.

Later, Amdahl's argument in [20] led to extending the speedup gain equation by differentiating between the parallel portion of the program and the serial portion of the program. Amdahl's law enhanced the speedup such that:

$$S(N,P) = \frac{1}{(1-P) + \frac{P}{N}} \tag{5.2}$$

Where:

- N is the number of cores.
- P is the parallel portion of the program such that: $0 \leqslant P \leqslant 1$
- S is the Speedup.

Amdahl's law considers only symmetric systems with identical cores with the same characteristics. From Equation 5.2 it is observed that by having a smaller serial portion in the program, a higher speedup will be achieved. Thus, when seeking software speedup by using extra cores, it is critical to know what portion of the software executes in a sequential manner, and what portion is executable in parallel.

In the following of this chapter, two test scenarios were designed and implemented, mocking workload division and parallelism in order to measure the possible speedup gain. In Section 5.3 a theoretical model is driven based on the heterogeneous hardware architecture of the MPC551x, such that it could be applied on already implemented single-core based applications. The model presents the speedup boost on migration of those applications to a dualcore system.

5.2 Measuring speedup gain

In multi-core systems, task fragmentation is usually done by a master task that fragments a workload of a task on a number of slave tasks, which each run on a different core, usually, known as slave cores. Each slave task carries out a part of the total work and reports back the result to the master task. Afterwards, the master task collects the results and puts the results together. In first sight, it seems that the more slave tasks are used, the parallel portion of the software, P in Equation 5.2 is increased and hence, a higher speedup is gained. However, the task fragmentation itself, introduces an extra overhead to the serial portion of the software, that is S in Equation 5.2 is increased and hence, the speedup is reduced. In the following we propose and implement a test scenario, in which, we measure the actual speedup gained in a similar manner as explained. We introduce SWC-Master, SWC-Slave1 and SWC-Slave2. SWC-Master consists of two runnables,

"DivideRunnable" that would do the workload fragmentation on other cores, and *"CollectRunnable"* that would put the results together. SWC-Slave1 and SWC-Slave2 will each do a part of the total fragmented workload.

Two scenarios are carried out, firstly, on a single-core system and secondly on a dualcore system with the same workload as in the single-core scenario but with the workload being fragmented on two cores. Figures 5.1 and 5.3 illustrate the SWCs configuration, port mapping and runnable to task mapping on the two scenarios. The following sections describe each scenario in details.

5.2.1 Single-core scenario

The software architecture is shown in Figure 5.1, since there is no active communication between Core1 and Core2, and there is no active task on Core2, Core2 is idle. Figure 5.2 shows the sequence diagram in which *"DivideRunnable"* puts a workload on *"Runnable1"*. The following explains the single-core scenario based on Figure 5.1 and Figure 5.2.

Step 1: *"DivideRunnable"* which is mapped to SWC-Master, writes a message by an "RTE_IWrite" to port "S2". Once the message becomes available on the receiving port, Task Slave1 is activated. The message is read on the receiving port by *"Runnable1"* and *"Runnable1"* starts its execution.

Step 2: *"Runnable1"* completes the execution of the simulated workload and reports back upon completion by an "RTE_IWrite" to *"CollectRunnable"*, which is mapped to SWC-Master and port R2.

Step 3: *"CollectRunnable"* measures the total execution time(i.e. the time difference between writing to *"Runnable1"* and reading the data written back to *"CollectRunnable"* by *"Runnable1"*). As it is seen in Listing 5.1, the "CollectRunnable " does the measurement (line No. 20) once TaskSlave1(i.e. task1 in source code) has completed its execution.

The scenario is repeated for different workloads. Hence, the total workload (i.e E) is increased, and the scenario starts again from "Step 1". Time measurement starts before "DivideRunnable" fragments the workload and stops after "CollectRunnable" has read the data on its ports(i.e. R2 in Figure 5.1). Hence, the communication time is included in the workload execution time itself. Listing 5.1 shows source code of runnables. The source code is simplified in Listing 5.1 and Listing 5.2.

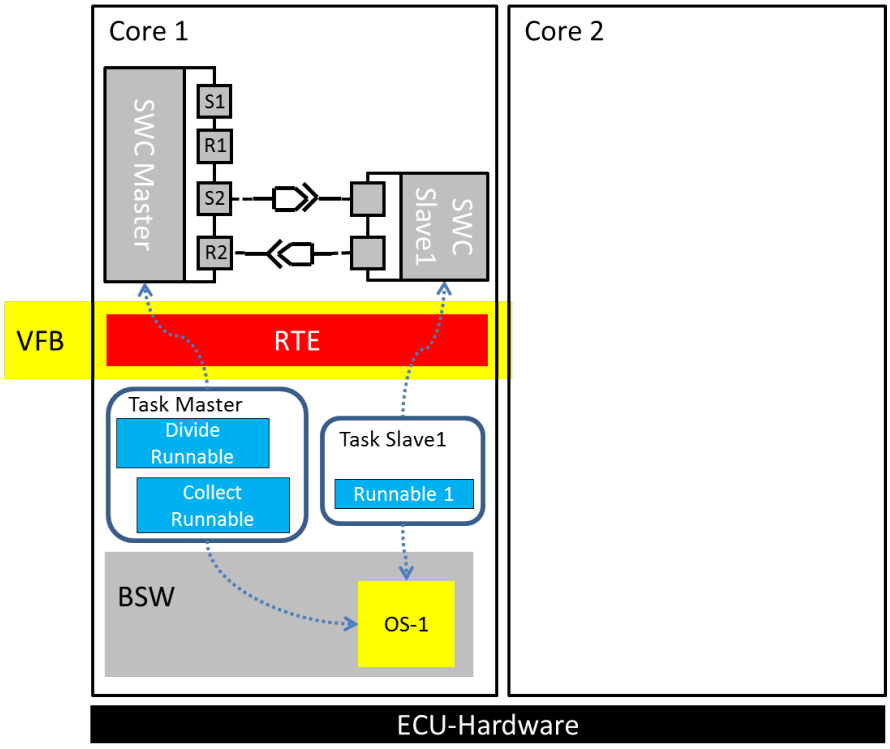

Figure 5.1: *High level illustration of implemented software for single-core scenario configuration on the AUTOSAR stack. In this scenario, only Core1 is used and no workload fragmentation is done and the total workload is carried out by Task Slave1. SWC-Master writes a data to SWC-Slave1, and SWC-Slave1 reports back to SWC.Master, once it has finished its execution.*

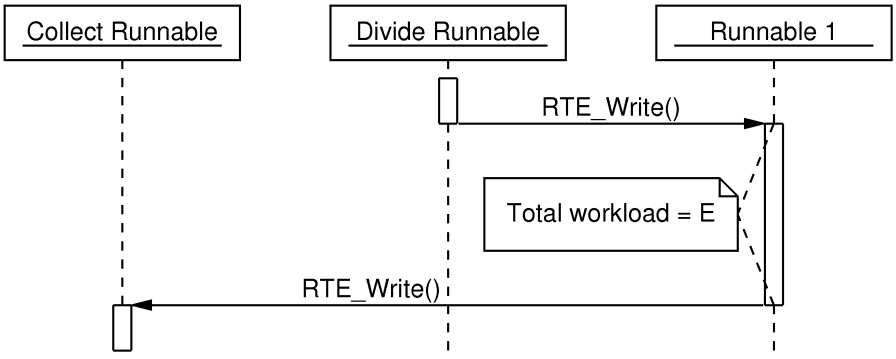

Figure 5.2: *Sequence diagram of single core scenario illustrating workload propagation between the Master task and one Slave task. No workload division occurs in this scenario.*

```
1  void DivideRunnable(){
2
3
4    Rte_IWrite_Sender1 Runnable_senderPort_dataElem_1(task1Counter);
5
6  }
7
8  void CollectRunnable(){
9
10   UInt8 val1 = 99;
11   UInt8 val2 = 99;
12   val1 = Rte_IRead_Sender2Runnable_R1senderPort_dataElem_1();
13
14   if(val1 == task1Counter)
15   {
16     task1Counter = 1;
17     task1_done = 1;
18   }
19
20   if((task1_done==1))
21   {
22     /* Measure time, Increase workload, repeat the scenario*/
23   }
24 }
25 void Runnable1(){
26   static UInt8 val = 11;
27   taskCounter = Rte_IRead_DivideRunnable_senderPort1_dataElem_1();
28   int dummy = 0;
29   for(int i = 0; i < taskCounter; i++){
30     dummy++;  /*workload simulation*/
31   }
32   Rte_IWrite_CollectRunnable_senderPort_dataElem_1(taskCounter);
33 }
```

Listing 5.1: *Listing of runnables source code. CollectRunnable is called when a data is written on its port. After checking that the data written is correct, it will set the relative task to done and so the experiment continues by increasing the workload.*

5.2.2 Dualcore scenario

Figure 5.3 illustrates the software configuration in the dual core scenario. As mentioned earlier, in this scenario, the workload is fragmented among the two cores and is completed by "Slave Task1" and "Slave Task2". The following describes the dual-core scenario based on Figure 5.3 and Figure 5.4.

Step 1: *"DivideRunnable"* which is mapped to SWC-Master, writes a message to port "S2" and a message to port "S1" by making an RTE_IWrite on each port. Once the message becomes available on the receiving ports, Task Slave1 and Task Slave2 are activated. The message is read on the receiving port by *"Runnable1"* and *"Runnable2"*, and the two runnables start their execution.

Step 2: *"Runnable1"* and *"Runnable2"* complete their execution of the simulated workload and each reports back the completion to *"CollectRunnable"*.

Step 3: *"CollectRunnable"* starts its execution when a data is written on its port *and* Core1 is idle. For example, if *"Runnable1"* completes its execution it will trigger the *"CollectRunnable"*, but the *"CollectRunnable"* will realize that *"Runnable2"* has not reported the result yet and hence, Core1 will go back to idle. Eventually, *"Runnable2"* will also complete its execution and trigger *"CollectRunnable"*, but this time it will realize that both runnables have reported back, and hence it will end the scenario. By contrast, in Figure 5.4, *"Runnable2"* completes its execution before *"Runnable1"* but it can not activate *"CollectRunnable"*. This is because Core1 is executing *"Runnable1"* and *"Task Master"* and *"Task Slave1"* have the same priority(See Figure 5.3). Hence, *"CollectRunnable"* is executed when actually both slave tasks have completed their execution, but as explained in the first example, this might not be the case. Therefore, as seen in Listing 5.2(line No. 27) this is examined and *"CollectRunnable"* measures the total execution time when both *"Runnable1"* and *"Runnable2"* have completed their workload(i.e. time difference Writing to *"Runnable1"* and *"Runnable2"*, and reading the data written back to *"CollectRunnable"* by *"Runnable1"* and *"Runnable2"*). As it is seen in Listing 5.2(line No. 27), the "CollectRunnable " does the measurement when both task have completed their execution.

The scenario is repeated for different workloads. Hence, the total workload (i.e , E) is increased, and the scenario starts again from "Step 1".

5.2.3 Results

The speedup is calculated using Equation 5.1. Having the results for different workloads on the single-core and dualcore scenarios. Figure 5.5 illustrates the results. In both scenarios, the serial portion of the software is constant for all values of different workloads. However, in the dual core scenario, the serial portion of the program is bigger since an extra communication, (inter-core communication

```
void DivideRunnable(){


  Rte_IWrite_Sender1Runnable_sender2Port_dataElem_1(task1Counter);
  Rte_IWrite_Sender1Runnable_senderPort_dataElem_1(task2Counter);
}
void CollectRunnable(){

  UInt8 val1 = 99;
  UInt8 val2 = 99;
  val1 = Rte_IRead_Sender2Runnable_R1senderPort_dataElem_1();
  val2 = Rte_IRead_Sender2Runnable_R2senderPort_dataElem_1();

  if(val1 == task1Counter)
  {
    task1Counter = 1;
    task1_done = 1;
  }

  if(val2 == task2Counter)
  {
    task2Counter = 2;
    task2_done = 1;
  }

  if((task1_done==1)&&(task2_done==1))
  {
  /* Measure time, Increase workload by increasing taskcounter,
     repeat the scenario*/
  }
}
void Runnable1(){
  static UInt8 val = 11;
  taskCounter = Rte_IRead_DivideRunnable_senderPort1_dataElem_1();
  int dummy = 0;
  for(int i = 0; i < taskCounter; i++){
    dummy++;  /*workload simulation*/
  }
  Rte_IWrite_CollectRunnable_senderPort_dataElem_1(taskCounter);
}
void Runnable2() {
  static UInt8 val = 33;
  taskCounter = Rte_IRead_DivideRunnable_senderPort2_dataElem_1();
  int dummy = 0;
  for(int i = 0; i < taskCounter; i++){
    dummy++;  /*workload simulation*/
  }
  Rte_IWrite_CollectRunnable_senderPort_dataElem_1(taskCounter);
}
```

Listing 5.2: *Listing of runnables. CollectRunnable is called when a data is written on its port. After checking that the data written is correct, it will set the relative task to done. By contradiction to Listing 5.1, it checks that both tasks are done and the first time CollectRunnable is called, only one task is set to done. Once, the second data is written, it will set the relative task to done too and the scenario will be repeated.*

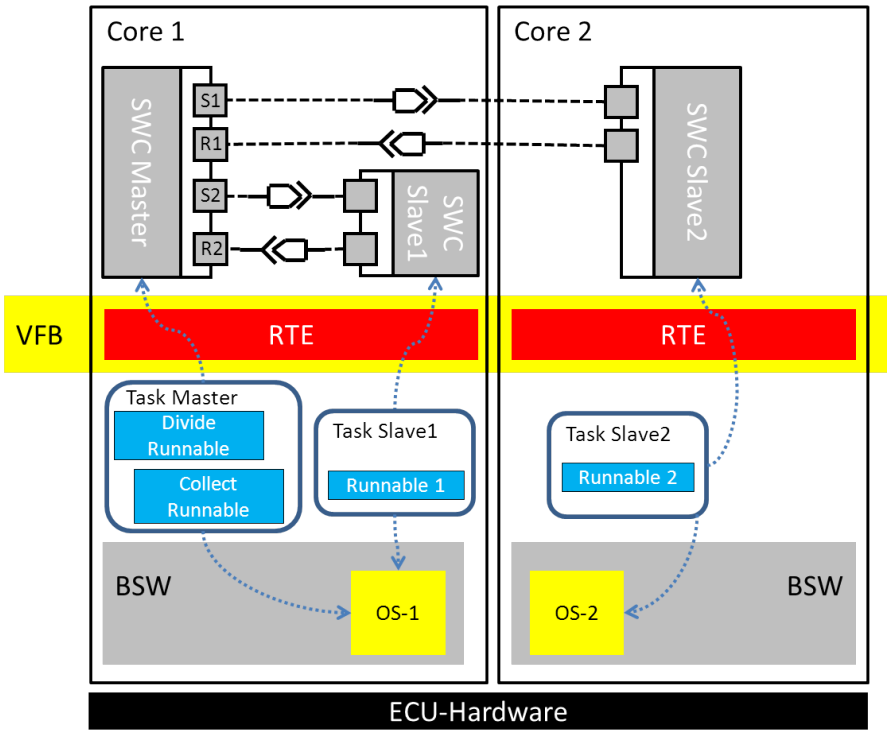

Figure 5.3: *High level illustration of implemented software for dual core scenario configuration on the AUTOSAR stack. In this scenario, two cores are used and workload fragmentation is done through an inter-core communication. The total workload is carried out by Task Slave1 and Task Slave2, which execute in parallel. The total workload is carried out by Task Slave1 and Task Slave2, which execute in parallel.*

between Master and Slave2) occurs. Therefore, if the speedup gained by workload fragmentation is smaller than the extra cross-core communication overhead, the speedup is actually less than 1, in other words the performance has been reduced! Therefore, from Figure 5.5 it is observed how big should the parallel portion of the software be so that the speedup will exceed 1. The Figure shows that for total workload of approximately $E \approx 345\mu s$ it is worth having a workload fragmentation strategy having such an specific configuration. In this experience the workload was divided *equally* among the two cores. Given that the two cores have different processing powers, it is expected to achieve a higher speedup by having a better fragmentation strategy(i.e the core with higher processing power, should carry out a bigger workload.). An 8th degree exponential fit shows a lean nor more than 45% speedup on the MPC551x. Ideally, according to Amdahl's law, in a homogeneous dual-core system, with a relatively small serial portion and a big parallel portion of the software, the speedup lean should be 2.

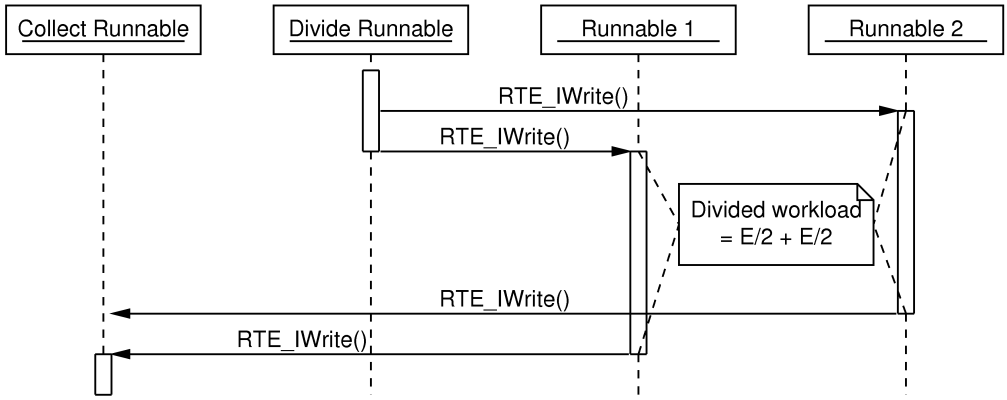

Figure 5.4: *Sequence diagram of dual core scenario illustrating workload propagation between the Master task and Task Slave1 and Slave2. The workload is equally fragmented among the two tasks.*

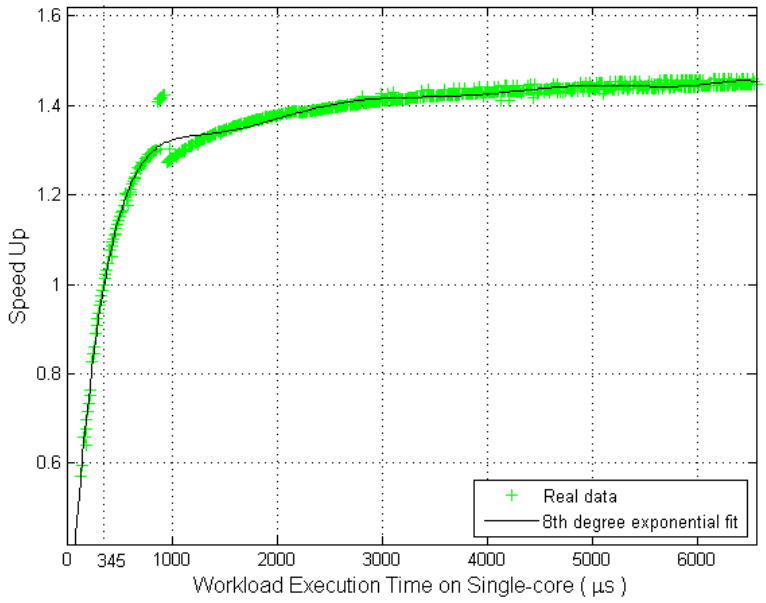

Figure 5.5: *Speedup measurements with equal workload division with 700 samples for increasing amount of workload. For tasks with a lower execution time than 345μs, it is not worth to fragment the task among two parallel tasks. An 8th degree exponential fit shows that maximum speedup gained would be less than 45 percent.*

5.3 Extending Amdahl's law to a heterogeneous multicore AUTOSAR system

One might question that having a task with a known execution time, would it be wise to fragment the task to two separated tasks on two cores, in order to achieve a lower execution time? If so, what speedup will be gained? This is a crucial question when migrating a software from a single core, to a multicore platform, with the intention of reducing execution time and gaining speedup. In this section, a theoretical model on speedup gain on workload division(task fragmentation) is developed, based on the heterogeneous nature of the MPC551x and the communication mechanism among SWCs on multiple cores in AUTOSAR 4.0. The goal is to extend Amdahl's law to the multicore AUTOSAR architecture, such that the result will provide a more precise picture of the possible speedup gain.

5.3.1 Earlier studies

Over the past four decades, many researchers have developed various extension of the Amdahl's law defining upper and lower bounds, and extending the original work to various hardware architecture factors such as pipeline, branching, out of order execution and etc. Hill and Marty in [21] investigated the Amdahl's law in symmetric, asymmetric, and dynamic multicore hardware and applied a hardware model to Amdahl's law and introduced a power factor for cores. By this, they differentiated between the processing power of different cores in asymmetric systems. However, Xian-He Sun and Yong Chen in [22] argued the conclusion by Hill and Marty were pessimistic and evaluated their work. E. Yao *et al* in [23] carried on with Hill and Marty, and did a theoretical analysis on their work. Today, the trend in multi core systems is a hot area, however, some may argue it is the repetition of history, since the arguments are very similar to the discussions when multi-processors were coming to the picture.

5.3.2 Model overview and Assumptions

In this section, a so called extended version of Amdahl's law is developed that would consider a coefficient of performance for each core in an asymmetric multicore system. Moreover, the extended model considers an overhead cost for each added core. This model will predict the possible speedup gain in multicore asymmetric systems more precisely than the original Amdahl's law.

The theoretical model for speedup gain prediction is based on the workload division strategy as described in Section 5.2. As Hill and Marty have shown, adding extra cores on a system, will introduce an execution overhead to the the program. This overhead, which can be interpreted as the serial portion of the program, may be caused by both hardware, and software properties. For example, some hardware design attributes that could affect this overhead, are memory access,

pipelines, profiling and etc[1]. From a software point of view, additional work might be needed for having extra cores, and this could introduce the naming overhead. The limit of Equation 5.2 (Amdahl's law) with N towards infinity, is a *strict ascending* curve, leaning to a certain number(depending on N and P). But, by introducing an overhead for each added core, after a certain number of cores, the speedup will actually decrease[2]. In other words, one cannot just increase the number of cores, threads or processes on it's software and expect it to run faster. In many cases, the program will actually become slower, and so, it is necessary to understand the design principles behind any software *and* hardware. The following explains how the overhead for having an additional core is introduced within the AUTOSAR context.

Considering the workload division strategy, it is obvious that by adding one core, two inter-core communications are needed and hence, more instruction operations are executed. Figure 5.6 shows how the extra overhead for the added core is introduced according to the workload division strategy. In the upper graph of Figure 5.6, M_1, E and M_2 represent "DivideRunnable", "Runnable1" and "CollectRunnable" in Figure 5.1, respectively. Similarly, in the lower graph, M_1, E_1, E_2 and M_2 represent "DivideRunnable", "Runnable1", "Runnable2" and "CollectRunnable" in Figure 5.3, respectively. In Figure 5.6, the upper graph shows that M_1 activates E(i.e. the complete workload), and E activates M_2 after its completion. In the lower graph in Figure 5.6, the second core is added, and Task E, which is the completely parallelizable, is divided between the two cores in two smaller tasks, E_1 and E_2 and so E should have a shorter execution time in overall. However, M_1 has to activate E_2 *in addition* to E_1, and so an overhead will be introduced (i.e. O_1). Similarly, E_2 and E_1 have to activate M_2 together, and so an overhead introduced (i.e. O_2). Hence, by introducing the second core, two additional overheads, O_1 and O_2, are introduced. The following describes how the extended speedup considers the naming overhead.

Assume task E is divided equally to E_1 and E_2. We have:

T_1 is the execution time of task E on one core.

T_2 is the execution time of tasks E_1 and E_2 on two cores.

$$S_2 = \frac{T_1}{T_2} \tag{5.3}$$

$$E_1 = E_2 = \frac{E}{2}\ ,\ T_2 = \frac{T_1}{2} \Longrightarrow S_2 = \frac{T_1}{\frac{T_1}{2}} = \frac{E}{\frac{E}{2}} \tag{5.4}$$

Constant O is defined as the overhead of introducing the second core such that $O = O_1 + O_2$.

[1]For example, in the MPC551x microcontroller, one core has direct access to flash memory, while the other needs to access the crossbar for accessing the flash memory. This may cause a difference in the processing power of each core.

[2]See [21] for example graphs on this matter.

$Speedup_{EXTENDED,2}$ yields:

$$Speedup_{EXTENDED,2}(E,O) = \frac{E}{\frac{E}{2} + O} \tag{5.5}$$

In an asymmetric dualcore system, where the cores have different processing powers, it is reasonable to divide the workload *unequally* among the cores. However, in Equation 5.5, it is assumed that both cores are identical and the workload is divided equally among the two cores. Therefore, Equation 5.5 is enhanced so that an unequal task division would be possible.

If task E is divided unequally among two cores as E_1 and E_2, and the two tasks start their execution simultaneously, it is unlikely that both E_1 and E_2 are finished at the same time. Therefore, the total workload (i.e. E) is finished when both divided tasks are finished. Hence, the execution time of E when it is divided to two tasks, is the execution time of the task(E_1 or E_2) with the longer execution time. Therefore, a coefficient should determine what percentage of task E should run one core, so that the remainder would run on the other core. We have:

Coefficient α is defined as the workload division coefficient and $0 \leqslant \alpha \leqslant 1$. Hence:

$$\frac{E}{2} \longleftarrow^{3} Max(\underbrace{\alpha E}_{E_1}, \underbrace{(1-\alpha)E}_{E_2}) \tag{5.6}$$

Where:

$$E_1 = \alpha E \quad and \quad E_2 = (1-\alpha)E$$

$$(5.5), (5.6) \Longrightarrow Speedup_{EXTENDED,2}(E,O,\alpha) = \frac{E}{Max(\alpha E, (1-\alpha)E) + O} \tag{5.7}$$

Having an asymmetric system with two cores, one core will have higher processing power than the other. In other words, a given task will have a longer execution time on the slower core. The following enhances Equation 5.7 such that this distinction between the cores is taken into account:

For any two cores, $Core_i$ and $Core_j$, having a known task T_i with a known execution time of e_i on $Core_i$ and e_j on $Core_j$, β is determined as the processing power coefficient such that:

$$\beta = \frac{e_i}{e_j} \tag{5.8}$$

Assume that in Equation 5.6, E_1 is executed on $Core_1$, and E_2 is executed on $Core_2$. If E_2 has an execution time of e on $Core_1$, E_2 will have an execution time of e/β on $Core_2$. We have:

[3] $x \leftarrow y$ expresses that x is replaced by y.

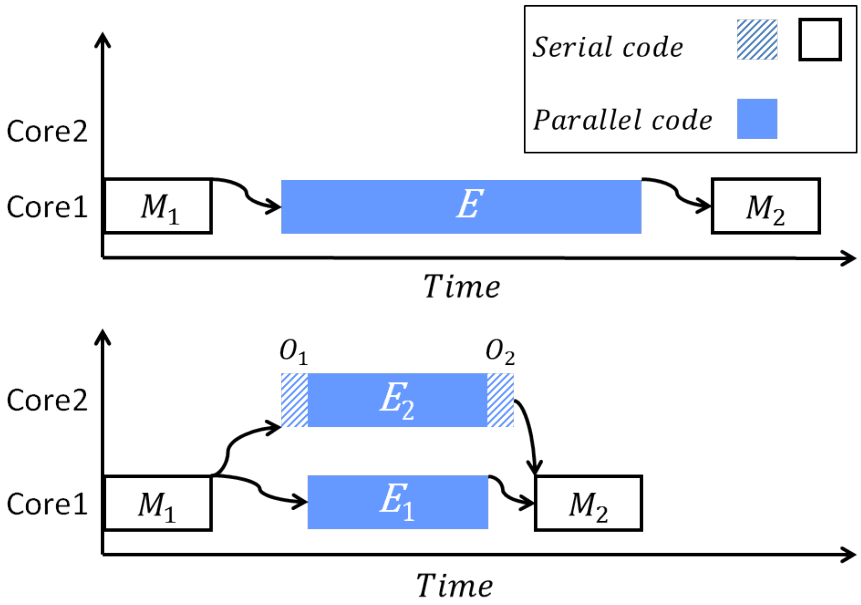

Figure 5.6: *The upper graph shows task E, which is parallelizable, executing in a single core system. The lower graphs shows task E being fragmented to two sub tasks, E_1 and E_2. By adding $Core_2$, an overhead is introduced because of the communication among the two cores. The overhead is non-parallelizable.*

$$(5.6),(5.8) \Longrightarrow E_2 \leftarrow \frac{E_2}{\beta} \tag{5.9}$$

This means, since E_2 is being executed on $Core_2$, it will take a longer time for it to finish.

Hence,by considering that the system is heterogeneous and introducing β:

$$(5.6),(5.7),(5.9) \Longrightarrow Speedup_{EXTENDED,2}(E,O,\alpha,\beta) = \frac{E}{Max(\alpha E, \frac{(1-\alpha)E}{\beta}) + O} \tag{5.10}$$

The final argument is how to optimize Equation 5.10 to achieve maximum speedup? In symmetric systems, which all cores are identical, it is reasonable to divide the tasks equally among cores. By contradiction, in heterogeneous systems the task division should be relevant to each core's processing power. In other words, task division should be such that the total execution time would be minimized. For example, in Figure 5.6 instead of dividing E equally to E_1 and E_2, task division should be such that E_1 and E_2 finish their execution at the same time. This means that α should be determined based on β. Hence, we have:

Having:

$E_1 = \alpha E \quad and \quad E_2 = \frac{(1-\alpha)E}{\beta}$*:*

$$E_1 = E_2 \Rightarrow \alpha E = \frac{(1-\alpha)E}{\beta} \Rightarrow \alpha = \frac{1-\alpha}{\beta} \Rightarrow \alpha = \frac{1}{1+\beta} \tag{5.11}$$

Therefore, for maximum speedup, Equation 5.10 is enhanced by an extra condition of defining α according to Equation 5.11. Calculating speedup analysis by using Equation 5.10 in practice, requires determining the values of the overhead, power processing coefficient and the execution time of the task in single-core case. The next section explains how these values are measured.

5.3.3 Using and evaluating the extended speedup model

In this section, the extended speedup model (i.e. Equation 5.10) is applied to a task in order to predict the speedup of the naming task, when it is migrated from a single-core architecture to a dualcore architecture. Then, the naming task is fragmented and migrated to a dualcore architecture in order to measure the speedup by experience(similar to the speedup measurement scenario in Section 5.2.2). Finally, the extended speedup model is evaluated by comparing the results in the theoretical model and the speedup achieved in the experiments.

In order to determine the speedup of a given task, according to the workload fragmentation strategy, and by using the extended speedup model(Equation 5.10), first, the values of the following variables should be determined:

- β, the processing power coefficient.
- O, the overhead of introducing an additional core.
- E, the execution time of the given task on single-core.

Determining β's value requires measuring the pure processing power of $Core_1$ and $Core_2$, independent of each other. According to Equation 5.8, this is simply doable by measuring the execution time of any given task once on $Core_1$, and then on $Core_2$. Hence, the dualcore configuration test case is done with the values of $\alpha = 1$, in which the task is executed completely on $Core_1$, and $\alpha = 0$ in which the task is executed completely on $Core_2$. By comparing the results, β's value is determined for the MPC551x microcontroller and under the given configuration[4]. In these two test cases, both cores are active meaning that message passing among the cores occurs and the software configuration is as described in the dualcore scenario in Section 5.2.2.

[4]Various configuration settings on MPC551x could alter the results. For example, one core may have priority for data fetching or accessing the crossbar. In this study, different configuration settings were studied and examined. A configuration was chosen that would provide best results in our experiments. Generally, one should fine tune hardware settings according to its software in order to achieve best performance. See chapters 10 and 11 in [17].

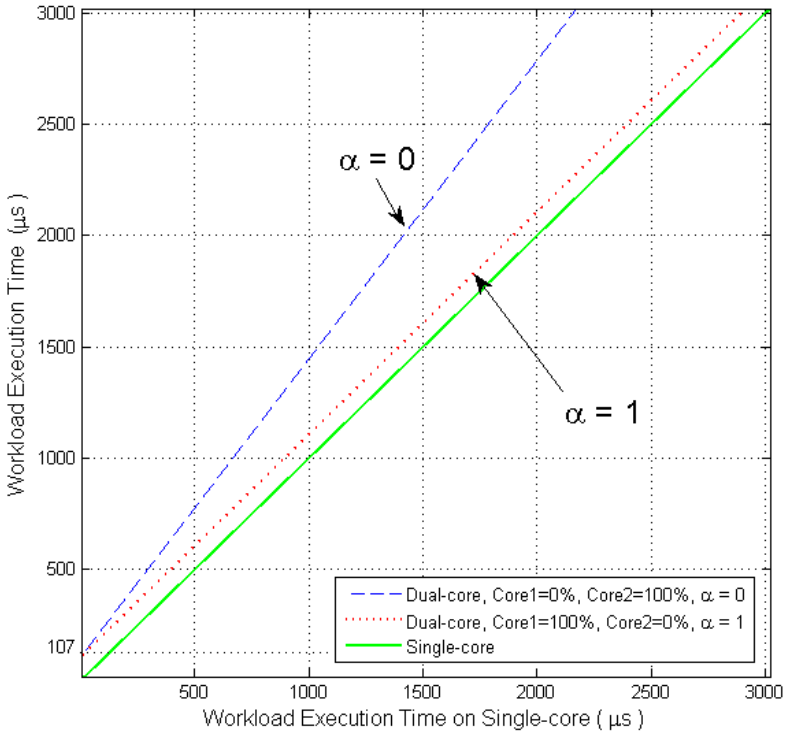

Figure 5.7: *Comparison of Core1's pure computational capacity compared to Core2's computational capacity. The execution time for a same workload is shown if executed on Core1($\alpha = 1$) or on Core2($\alpha = 0$). The higher slope of $\alpha = 0$ shows less processing power for Core2. The y-intercept for $\alpha = 1$ shows the cross-core communication overhead for Dual-core over Single-core.*

Figure 5.7 shows the results of dualcore scenarios, having $\alpha = 1$ and $\alpha = 0$. According to the definition of processing power coefficient in Equation 5.8, β can be calculated by dividing the y values of any two given points with the same x value in Figure 5.7. This will provide β's value based on only one experiment of one single task. This reasoning is prone to errors since a single execution of a task could occur in a WCET, BCET or ACET[5]. Therefore, β is calculated by dividing the slope coefficient of $(\alpha = 1)$ over $(\alpha = 0)$. By this, a more realistic and less erroneous value is measured.

$$slope\ coefficient = m = \frac{y_2 - y_1}{x_2 - x_1} \tag{5.12}$$

$$\beta = \frac{e_{\alpha=1}}{e_{\alpha=0}} = \frac{m_{\alpha=1}}{m_{\alpha=0}} \approx \frac{1}{1.35} \approx 0.74 \tag{5.13}$$

[5]WCET: Worst case execution time. ACET: Average case execution time. BCET:Best case execution time.

In order to define the overhead cost for introducing a second core, a task is executed first on $Core_1$ (i.e. the upper graph in Figure 5.6). Afterwards, the same task should be executed on two cores, having $\alpha = 1$(i.e. the lower graph in Figure 5.6). Hence, in the latter experiment, the execution time of E_2, is 0 and so the execution time of the given task (i.e. E) is increased *only* by $O_1 + O_2$. Therefore, by comparing the execution time of such experiments, the overhead of adding a second core is measured. The following describes how O is derived from the experiments and the results as shown in Figure 5.7. The Single-core line in Figure 5.7 is actually $y = x$. Thus, the additional core's overhead is the y-intercept of any dualcore test case with an arbitrary α. However, having $\alpha = 1$ shows that the overhead is easily seen as the difference between the two lines of Single-core and $\alpha = 1$. The y-intercept for this case, and any other dualcore test case, yields:

$$y - intercept \approx 107 \Rightarrow O \approx 107\mu s \tag{5.14}$$

The trustworthy of Equation 5.10 is verified by carrying out several test cases with arbitrary values of α, and matching the experience with the equation's curve. Hence, dualcore test cases with different fractions of workload on Core1 and Core2(i.e. variations of α) are carried and the results are compared to the theoretical model. The upper graph in Figure 5.8 illustrates the results for the theoretical model and the dualcore scenario for $\alpha = 0.5$, having $\beta = 0.74$ and $O = 107$. The lower graph in Figure 5.8 shows the error percentage between the theoretical model and experimental results. As it is seen in the lower graph, the extended speedup model prediction is reliable with an error percentage up to a $\pm 10\%$. As execution time increases, the extended speedup model is less erroneous. However, an abnormality is seen in the lower graph close to where the execution time is almost 1000 μs. This abnormality is due to a abnormal behavior in the software and could be a place for further investigation. It is easier to see that in Figure 5.5 the speedup curve has an abnormal raise and fall just before $1000\mu s$. This unpredicted behavior causes the theoretical model to be more erroneous. In Appendix B, additional experiments are carried out with arbitrary fractions of workload division in order to more investigate and evaluate the extended speedup theoretical model. Throughout all the experiences shown in Appendix B, with considering abnormal behaviors(ACET, BCET and WCET), the theoretical model shows a percentage error less than $\pm 10\%$ and by waving the abnormalities the percentage error is approximately $\pm 6\%$.

While Figure 5.8 shows the speedup with equal workload fragmentation(i.e. $\alpha = 0.50$), Figure 5.9 shows the theoretical model speedup according to Equation 5.10 having $\beta = 0.74$ where α is determined according to Equation 5.11. By determining α based on Equation 5.11, the speedup curve leans to 1.74 in infinity and the execution time when speedup equals to unity is $252\mu s$. This means that for tasks with an execution time less than $252\mu s$ on single-core, it is not worth fragmenting the task. In final, we recall that the speedup extended model is a preliminary version. Yet, with an eye on assumptions and limitations, and carefully determining correct values of the processing power coefficient and additional core's overhead, we believe the model is applicable and trustful.

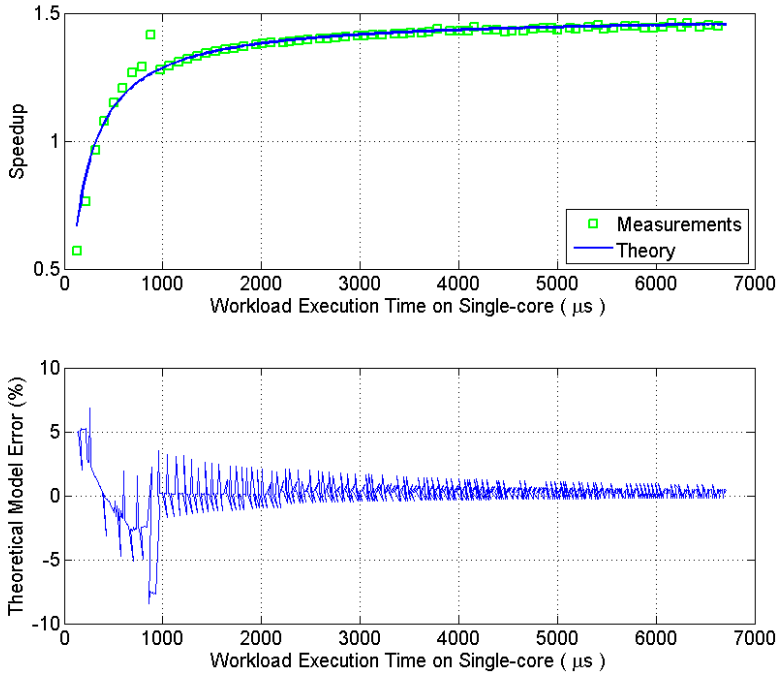

Figure 5.8: *Matching theoretical speedup analysis on real data measurements having* $\alpha = 0.50$*. Real data measurements were down-sampled by a step of 10 for ease of readability. The erroneous of the theoretical model's compared to real measurements are presented in speedup percentage in the lower graph.*

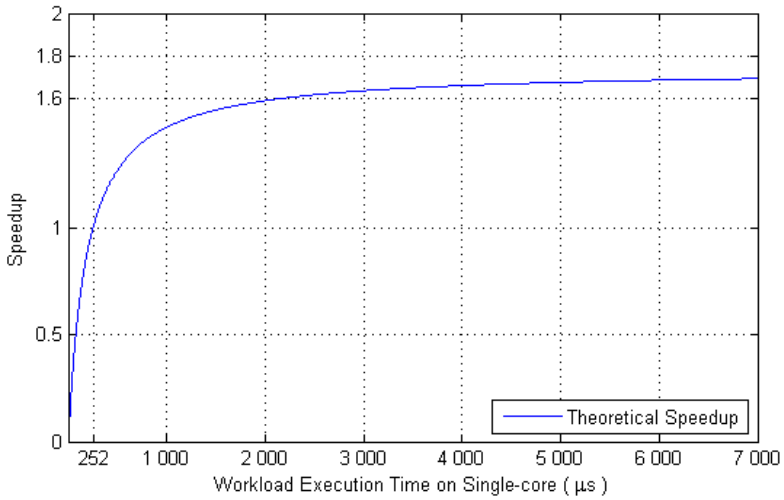

Figure 5.9: *Theoretical modeling of speedup gain with optimized workload fragmentation. The workload fragmentation factor's value(*$\alpha = 0.57$*) is defined relevant to the processing power factor for each core. The curve shows that the speedup exceeds 1 for execution times greater than* $252\mu s$*.*

6

Conclusions

The thesis has studied aspects of multicore support in AUTOSAR 4.0, while evaluating the performance of the multicore architecture. To have a better understanding of the multicore architecture, the new IOC module was investigated in detail, and a set of experimental scenarios were designed and implemented.

The communication mechanism among SW-Cs was studied as it is a key element in the performance of any AUTOSAR software. Test scenarios between software components in dualcore and single-core architectures were carried out and they have shown that the cross-core communication time has an overhead of 54% compared to a single-core communication time.

The study investigated the performance of the multicore architecture on an MPC551x processor which is a dualcore heterogeneous system. In a scenario of task fragmentation among cores, we developed a theoretical speedup model by extending Amdahl's law and adapting heterogeneous systems and unequal task fragmentation among cores. Also, the model addressed the cross-core communication overhead. Afterwards, by studying the MPC551x processor and determining its properties by carrying out several test scenarios, the extended speedup model was adapted to the MPC551x processor. As a result, the speedup curve of a dualcore AUTOSAR software with task fragmentation on this specific target hardware was developed. The speedup curve shows the system performance increase in the dualcore system compared to the single core system for any given execution time. The results showed the maximum speedup for long execution times(infinity) would be 74% and execution time when speedup equals to unity is at most $28\mu s$. This means for a task with an execution time less than $28\mu s$, it is not worth fragmenting the task. The model was verified by comparing the results with several test scenarios and the results showed that the theoretical model is more than 90% accurate.

Bibliography

[1] J. D. T. K. K. K. J. K. N. M. D. P. K. S. J. W. D. W. K. Y. Krste Asanovic, Rastislav Bodik, A View of the Parallel Computing Landscape, Communications of the ACM 52 (10) (2009) 56–67.
URL http://dl.acm.org/citation.cfm?doid=1562764.1562783

[2] AUTOSAR, AUTOSAR Basics.
URL http://autosar.org/index.php?p=1&up=1&uup=0&uuup=0&uuuup=0&uuuuup=0

[3] AUTOSAR, AUTOSAR Members.
URL http://www.autosar.org/index.php?p=2&up=1&uup=0

[4] AUTOSAR, AUTOSAR FAQ.
URL http://www.autosar.org/index.php?p=1&up=6&uup=0

[5] AUTOSAR, Specification of the RTE Software (2006).
URL http://www.autosar.org/download/R2.0/AUTOSAR_SWS_RTE.pdf

[6] AUTOSAR, Specification of the Virtual Function Bus (2008).
URL http://www.autosar.org/download/AUTOSAR_SWS_VFB.pdf

[7] M. Kunz, OSEK OS, Tech. rep., Chemnitz University of Technology (2009).
URL http://osg.informatik.tu-chemnitz.de/lehre/old/ws0809/sem/online/OSEK.pdf

[8] AUTOSAR, Feature Specification of the BSW Architecture and the RTE (2011).
URL http://www.autosar.org/download/R4.0/AUTOSAR_RS_BSWAndRTEFeatures.pdf

[9] AUTOSAR, Specification of Operating System (2011).
URL http://www.autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf

[10] R. R. Karthik Lakshmanan, Gaurav Bhatia, AUTOSAR Extensions for Predictable Task Synchronization in MultiCore ECUs, SAE 2011 World Congress

Exhibition.
URL http://papers.sae.org/2011-01-0456

[11] ArcticStuido, ArcCore AB, Göteborg, Sweden, Version 1.3.
URL http://www.arccore.com/products/arctic-studio/

[12] CodeWarrior Development Studio for MPC55xx/MPC56xx (Classic IDE), Freescale Semiconductor Inc, Texas, USA, version 5.9.
URL http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW-MPC55XX_56XX

[13] Freescale Semiconductor Inc, Texas, USA.
URL http://www.freescale.com/

[14] S. A. L. R. W.-F. W. Qin Zhao, Rodric Rabbah, How to do a million watchpoints: efficient debugging using dynamic instrumentation, Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction (2008) 147–162.
URL http://dl.acm.org/citation.cfm?id=1788388

[15] FreeMASTER Run-Time Debugging Tool, Freescale Semiconductor Inc, Texas, USA, Version 1.3.
URL http://www.freescale.com/Freemaster

[16] MPC5510EVB User Manual (2007).
URL http://cache.freescale.com/files/dsp/doc/ref_manual/MPC5510EVBUM.pdf

[17] MPC5510 Microcontroller Family Reference Manual (2008).
URL http://cache.freescale.com/files/32bit/doc/ref_manual/MPC5510RM.pdf

[18] Basic Multicore Initialization For the MPC5516G/E and MPC5514G/E Devices (2008).
URL http://cache.freescale.com/files/microcontrollers/doc/app_note/AN3627.pdf

[19] MPC5500 and MPC5600 Simple Cookbook (2010).
URL http://cache.freescale.com/files/32bit/doc/app_note/AN2865.pdf

[20] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, In Proceedings of the April 18-20, 1967, spring joint computer conference (1967) 483–485.
URL http://dx.doi.org/10.1145/1465482.1465560

[21] M. Hill, M. Marty, Amdahl's law in the multicore era, Computer 41 (7) (2008) 33–38.

URL http://dl.acm.org/citation.cfm?id=1449387&CFID=163581176&CFTOKEN=33586685

[22] X. Sun, Y. Chen, Reevaluating amdahl's law in the multicore era, Journal of Parallel and Distributed Computing 70 (2) (2010) 183–188.
URL http://www.sciencedirect.com/science/article/pii/S0743731509000884

[23] E. Yao, Y. Bao, G. Tan, M. Chen, Extending amdahl's law in the multicore era, ACM SIGMETRICS Performance Evaluation Review 37 (2) (2009) 24–26.
URL http://dl.acm.org/citation.cfm?id=1639571

[24] A. Monot, N. Navet, B. B. Françoise Simonot, Multicore scheduling in automotive ECUs, ERTSS.
URL http://nicolas.navet.eu/publi/ertss_2010.pdf

[25] B. B. F. S.-L. Aurélien Monot, Nicolas Navet, Multi-source software on multicore automotive ecus - combining runnable sequencing with task scheduling, ERTSS.
URL http://nicolas.navet.eu/publi/ECU_TIE_2012.pdf

[26] W. S.-P. Niko Böhm, Daniel Lohmann, A comparison of pragmatic multi-core adaptations of the autosar system, OSPERT.
URL http://www.seas.gwu.edu/~gparmer/ospert11/ospert11_proceedings.pdf

[27] W. P.-Y. Z.-M. Z. Rongshen Long, Hong Li, An approach to optimize intra-ecu communication based on mapping of autosar runnable entities, ICESS.
URL http://www.computer.org/portal/web/csdl/doi/10.1109/ICESS.2009.63

[28] R. R. Jörn Schneider, Michael Bohn, Migration of automotive real-time software to multicore systems: First steps towards an automated solution, ICESS.
URL http://www.fh-trier.de/fileadmin/groups/12/ProSyMig/pub/paper/Migration_ECRTS2010_WIP.pdf

[29] K. S. G. Rajeshwari Hegde, Load balancing in multi ecu configuration, ARTcom.
URL http://dl.acm.org/citation.cfm?id=1673450

[30] T. Gribb, Simplyfing multicore migration, EECatalog.
URL http://eecatalog.com/dsp/2011/10/18/simplifying-multicore-migration/

[31] W. D. Huang Bo, Dong Hui, Z. Guifan, Basic concepts on autosar development, ICICTA 1 (2010) 871 – 873.

URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5522844

[32] M. N.-K. R.-M. J. Simon Schliecker, Jonas Rox, R. Ernst, System level performance analysis for real-time automotive multicore and network architectures, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28 (2009) 979–992.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5075823

[33] U. K.-C. F. Peter Gliwa, Jens Harnisch, From single-core to multi-core platforms - systematic migration of hard real-time software in autosar, Embedded World 28 (2011) 979–992.
URL http://www.symtavision.com/downloads/success-stories/MulticoreSchedulingAnalysis_Gliwa_Infineon_Symtavision_01.pdf

[34] FreeMASTER Serial Communication Diver User Guide, find in Driver Package (2011).

[35] FreeMASTER for Embedded Applications User Guide (2011).
URL http://www.freescale.com/files/microcontrollers/doc/user_guide/FreeMasterUG.pdf?fsrch=1

[36] Free Serial Port Monitor, HHD Software, London, UK, Version 3.31.
URL http://www.serial-port-monitor.com

Appendices

A

How to use FreeMASTER

This appendix is a quick guide on how to use FreeMASTER. The reader is encouraged to read section 3.1.3 before continuing on this appendix. The purpose of this tool is achieving the ability of input and output monitoring and modification in real-time on Freescale embedded devices. The following of this appendix describes who should use FreeMASTER, how to use FreeMASTER and reflections are given on how to use FreeMASTER for best performance.

A.1 Introduction to FreeMASTER

One of the difficulties in parallel real-time software development, is detecting runtime bugs that may or may not appear in runtime due to the realtime and parallel nature of software and cause an abnormal behaviour in software. Such an abnormality is extremely difficult to detect using static debuggers simply because by *stopping* and *starting* the software, the software behavior changes and the bug will not appear. Moreover, in real-time applications, sometimes it is necessary to see the system behavior in runtime, which again is not possible with static debuggers. The problem is solved by logging(which in essence modifies the software) or by using realtime debuggers. FreeMASTER[15] is an open-source, run-time debugging tool, provided by Freescale, which makes it possible to fetch and modify variables in realtime. To fulfill this goal, a protocol is implemented that communicates through the RS232 channel or the CAN bus[1] with the target hardware using an on target driver. Through the communication with target, the desired variables are read, sampled and modified in a user-defined rate, and downloaded to the PC. FreeMASTER also provides a graphical user interface for data visualization. Through this project, FreeMASTER is used both for debugging and monitoring purposes to analyze test scenarios. In easy words, FreeMASTER is seen as a digital oscilloscope which can probe, modify, and plot desired variables on PC. Generally, FreeMASTER has two parts:

[1] In the latest version, these are the only supported methods for the MPC551x micro-controller.

- The PC based FreeMASTER client running on PC.
- The on target driver and communication software.

The remainder of this section describes some features of FreeMASTER.

A.1.1 Memory read and write

Using this feature the PC host can read and write any memory location on the target hardware. Also, by providing the *.elf file to the PC software, it is possible to read and modify any variable in the target software. The feature could be used for software control by implementing a software that would behave differently based on the value of a variable.

A.1.2 Oscilloscope

As the name implies, the oscilloscope is an scope that could probe variables and illustrates the value of the variables on the scope. It is also possible to modify the value of a variable. An oscilloscope can probe at maximum 8 variables at the same time. The scope can plot a variable's value in real-time up to a resolution based on the communication speed and sampling period. In general, for μs resolution, the oscilloscope is not recommended.

A.1.3 Recorder

Similar to the oscilloscope the recorder can record the value of up to 8 variables at the same time. The difference between the oscilloscope and the recorder is that, in the recorder mode the variables are sampled on the internal device memory. The samples are then downloaded to PC and thus, the sampling rate is much faster compared to the oscilloscope. However, the recorder does not provide the information in real-time. The recorder provides a sampling up to $27\mu s$ resolution[35].

A.2 FreeMASTER on target driver

The FreeMASTER driver itself does not initialize the communication channel in any sort of way. It is the user's responsibility to manage the communication channel whether it is serial communication or CAN bus. Hence, the fist thing to do is to establish an RS232 communication between the hardware and the PC.

A.2.1 Establishing a serial communication

A Serial Communicator Interface (SCI) communication is established in the same manner as described in [19] section 15. In order to verify a successful communication a software is needed to monitor the serial communication channel between the target hardware and the PC. For this, the Free Serial Port Monitor[36] software is

used. First, SCI initialization has to be done on target board. Listing A.2 shows a source code for SCI initialization, a "Transmit" function and a "Receive" function. The SCI initialization values depend on desired "baud rate" and hardware "clock frequency". The initialization values are valid for 64MHz clock frequency and a baud rate of 9600 bits per second[2]. In [19] a simple function for "System Clock Initialization" is provided. The source code is shown in Listing A.1. However, if code generators are being used, instead of using "SysClockInit()", set the clock frequency from the code generator software, for example, ArcticStudio. In ArcticStudio, this could be done in the following steps:

1. Under "ECU Configuration" go to "MCU Settings".
2. Create a new clock and set the parameters PllEprediv=0, PllEmfd=32 and PllErfd=5.
3. Set the new clock to default clock.

Figure A.1 shows the system clock settings page in ArcticStudio BSW Builder. The red box and arrows indicate the valid values for 64MHz clock frequency.

Also, jumpers J9, J10 and J11 must be set as explained in [16] on the evaluation board in order to activate and correctly configure the SCI port. Figure A.2 shows a close up of the jumpers next to the ESCI connector.

```
void initSysclk(void) {                 /* Initialize PLL and sysclk to
    64 MHz */
  FMPLL.ESYNCR2.R = 0x00000007;         /* 8MHz xtal: 0x00000007; 12MHz
      xtal: 0x00000005 */
  FMPLL.ESYNCR1.R = 0xF0000020;         /* 8MHz xtal: 0xF0000020; 12MHz
      xtal: 0xF0020030 */
  CRP.CLKSRC.B.XOSCEN = 1;              /* Enable external oscillator */
  while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK  */
  FMPLL.ESYNCR2.R = 0x00000005;         /* 8MHz xtal: 0x00000005; 12MHz
      xtal: 0x00000003 */
  SIU.SYSCLK.B.SYSCLKSEL = 2;           /* Select PLL for sysclk */
}
```

Listing A.1: *System Clock Initialization function for 64MHz frequency.*

[2]User should set SCI settings on PC accordingly.

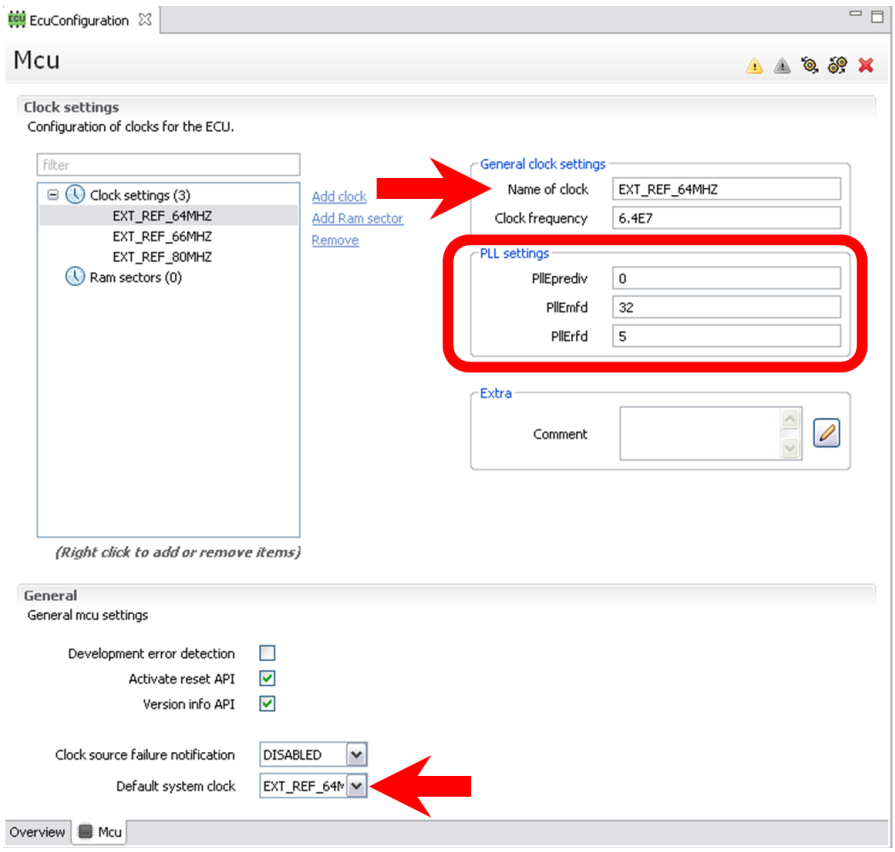

Figure A.1: *Screen shot from MCU settings page in ArcticStudio BSW Builder. The MCU is configured for a 64MHz clock frequency.*

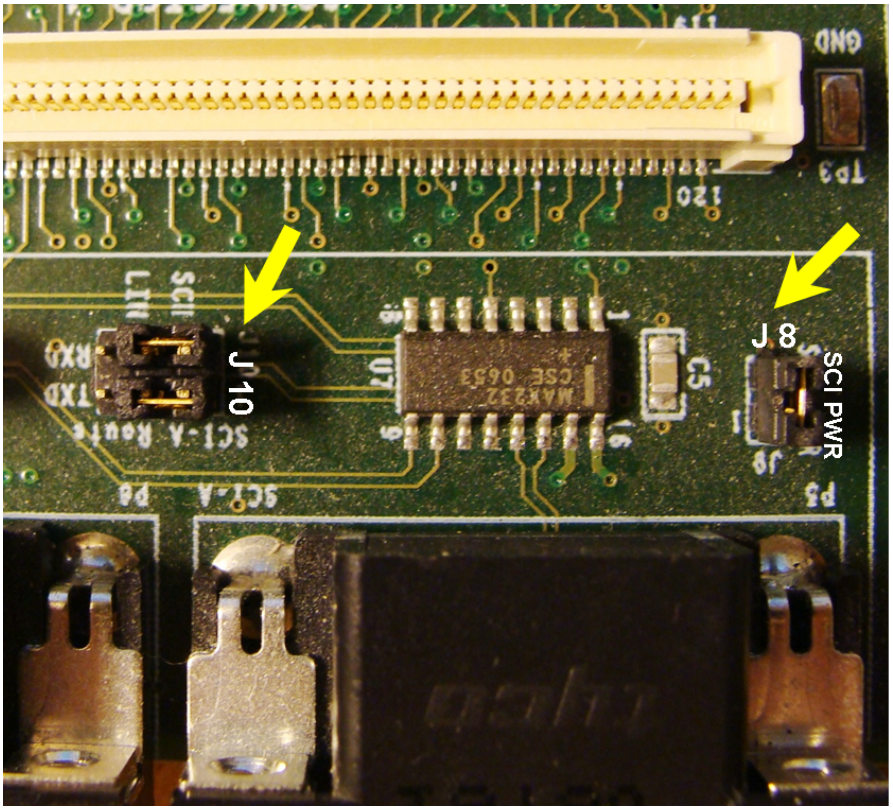

Figure A.2: *Close up view from SCI jumper settings on MPC5510EVB. J8 enables power on SCI and J10 configures Tx and Rx routing.*

```
const  uint8_t  TransData[] = {"Hello World!\n\r"}; /* Transmit
    string & CR*/
uint8_t  RecData;       /* Received byte from eSCI */

void ESCI_A_Init(void)
{
  ESCI_A.CR2.R = 0x2000;              // Enable ESCI and set all bits
      to reset value
  ESCI_A.CR1.B.TE = 1;                // receiver enable
  ESCI_A.CR1.B.RE = 1;                // transmitter enable
  ESCI_A.CR1.B.PT = 0;                // parity is even
  ESCI_A.CR1.B.PE = 0;                // parity control disable
  ESCI_A.CR1.B.SBR = 417;             // Baud rate = 9600, in case
      fipg_clock_lin = 64 MHz
  //ESCI_A.CR1.B.SBR = 34;              // Baud rate = 115200, in case
       fipg_clock_lin = 64 MHz
}

void TransmitData (void) {
  uint8_t j;                                    /* Dummy variable */
  for (j=0; j< sizeof (TransData); j++) {  /* Loop for character
      string */
    while (ESCI_A.SR.B.TDRE == 0) {}        /* Wait for transmit data
        reg empty = 1 */
    ESCI_A.SR.R = 0x80000000;               /* Clear TDRE flag */
    ESCI_A.DR.B.D = TransData[j];           /* Transmit 8 bits Data */
  }
}

void ReceiveData (void) {
while (ESCI_A.SR.B.RDRF == 0) {} /* Wait for receive data reg full =
    1 */
ESCI_A.SR.R = 0x20000000; /* Clear RDRF flag */
RecData = ESCI_A.DR.B.D; /* Read byte of Data*/
while (ESCI_A.SR.B.TDRE == 0) {} /* Wait for transmit data reg empty
    = 1 */
ESCI_A.SR.R = 0x80000000; /* Clear TDRE flag */
ESCI_A.DR.B.D = RecData; /* Echo back byte of Data read */
}
```

Listing A.2: *Simple SCI communication functions. The "MPC5517G.h" header file, or similar is needed.*

A serial communication may be established in the following steps:

1. Add the provided functions in Listings A.1 and A.2 to the source code.
2. Configure hardware clock frequency.
3. By calling "ESCI_A_Init", "TransmitData" and "ReceiveData" at a startup point(e.g. in OS startup) initialize and establish a serial communication.
4. Generate the *.elf file and upload to hardware.

5. Using the Free Serial Port Monitor, the "Hello World!" message sent to PC should be read. Also, verify that *PC to hardware* connection is working fine.

Note that it is necessary to open a SCI "session" before running the Free Serial Port Monitor software. For this, one can use the Freemaster PC side software(*.pmp). On the main screen as enable the "stop" button and then choose "keep trying".

A.2.2 Using FreeMASTER on target driver

By adding FreeMASTER source code files from the provided example to the project[3], the FreeMaSTER driver has been embedded to the on target hardware.[4]. The communication software has to be added using the driver API. The only file needed for using the driver API, is "freemaster.h". By adding this header file to the source code(e.g. main.c), all FreeMASTER functions are accessible. The remainder of this section explains FreeMASTER API and functions. The complete guide to serial communication driver may be found in [34].

A.2.2.1 Driver API

The API configuration is handled via the "freemaster_cfg.h" file. The communication between PC and the target through the SCI protocol may be implemented between the PC and the target through three different modes:

- Poll driven mode.
- Short interrupt mode.
- Long interrupt mode.

As stated in [34]: "Exactly one of the three macros must be defined non-zero, others must be defined zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver". In the poll driven mode, FMSTR_POLL() API is called in the code. The FMSTR_POLL() samples the desired variables and they are transferred to the PC via SCI. In order to reduce the FreeMASTER communication overhead at most, it is most likely to call this function in the system idle loop. In poll driven mode, a minimum rate of FMSTR_POLL() API should be called with a least minimum frequency. That is, the polling should be done as much as the SCI buffer would not become empty. This is a big drawback, since even when sampling is not desired, the FreeMASTER communication overhead has to be introduced. We interpret this as the *keepalive* time.[5] Depending on system properties and configuration(i.e. clock frequency, baud rate and SCI buffer

[3]how to address provided source code? address Example_FreeMASTER.rar

[4]FreeMASTER serial communication driver example "MPC55xx" provided by Freescale, does not work on the MPC551x microcontrollers, despite that it says MPC55xx. The reason is that the provided example is for MPC555x micro-controllers which are single core, while the MPC551x family are dual-core.

[5]See [34] section 2.4.3

size), the minimum times per second that polling should happen will differ. In complex systems with task preemption, it might be difficult to minimize, or even determine the additional overhead time. In addition to the difficulty of configuring the polling to a steady frequency rate, as our studies show, and similar to any other code snippet, FSMSTR_POLL() itself, may have different execution times. This will make the analysis even more complex, when dealing with time measurements or where execution time matters.
In short interval mode or long interval mode, the system behavior is quite the same as the poll driven mode, except that at the polling state, an interrupt is generated which will do the polling. This means that FMSTR_POLL() will execute within an ISR. Mainly, one will use the interrupt mode or poll driven mode, when the other is not possible. The interrupt has to be installed in the Interrupt Vector Offset Register (IVOR), and if FreeMASTER is used in an AUTOSAR implementation, this may cause a conflict between FreeMASTER and the AUTOSAR software. On one hand, the interrupt mode requires modification and control over the IVOR, and on the other hand, the operating system module in the BSW, *owns* the IVOR. This results as a conflict and thus, the FreeMASTER interrupts shall be installed and controlled *through* AUTOSAR-OS and interrupt handler. Because of this complexity, this approach is dropped and poll driven mode is used in this project.
Listing A.3 shows some parameters of the "freemaster_cfg.h" file that are important to be set. If one function, for example, the recorder is disabled, the corresponding API functions in the source code are generated *empty* and so, this header file provides a convenient way for disabling and enabling FreeMASTER functionalities.

```
#define FMSTR_USE_SCI           1     /* To select SCI communication
    interface */
#define FMSTR_USE_FLEXCAN       0     /* To select FlexCAN
    communication interface */

#define FMSTR_USE_RECORDER      1     /* enable/disable recorder
    support */
#define FMSTR_MAX_REC_VARS      8     /* max. number of recorder
    variables (2..8) */
#define FMSTR_REC_OWNBUFF       0    /* use user-allocated rec. buffer
    (1=yes) */

/* built-in recorder buffer (use when FMSTR_REC_OWNBUFF is 0) */
#define FMSTR_REC_BUFF_SIZE     10240 /* built-in buffer size */
```

Listing A.3: *Some parts of "freemaster_cfg.h" parameters. The "freemaster_cfg.h" file handles all configuration parameters of the communication protocol in one place.*

For establishing communication between target hardware and the PC, "FMSTR_Init()" API is called[6].

[6]Even without initialization the communication *might* work. For example, if the connection is lost, you do not need to recall FMSTR_Init() for re-establishing the communication

The “FMSTR_Recorder()” API samples the desired variables and records the values in an on-target pre-defined buffer at the point that it is called. Thus, it the sampling resolution is much higher than the oscilloscope. The samples are then send to PC upon a triggering condition, or whenever a number of samples have been taken. When implementing a communication protocol, monitoring the channel helps understanding and debugging software much easier. Therefore, it is recommended to use the Serial Port Monitoring Software to better understand the FreeMASTER protocol and for a better use of the API. The “FMSTR_Init()” information should be seen being transferred in the beginning of handshake between target hardware and PC.

A.3 Introduction to PC side software

In this section a brief explanation is given on some features of the PC side application. Figure bla shows the start up page of the application form the provided project. A complete user guide can be found in [34].

A.3.1 Communication setup in PC application

The application settings for communication can be set in “Project menu”, “Options”. Under the “Comm” tab, set “Direct RS232” as communication mechanism and the RS232 settings according to the settings used in the embedded application in the on-target driver. Under the “Map files” tab, set the corresponding binary (*.elf) file address(i.e the same *.elf file that is running on the target hardware), and set the “File format” to “Binary ELF with SWARF1 or DWARF2 dbg format”. Having the *.elf accessible to software, it is possible to easily access variables and specific memory locations from the “Grid view”.

A.3.2 Reading and Modifying variables

In the variable watch grid as shown in Figure A.3, variables can be added for monitoring and modification. By right clicking on the watch grid, under “watch properties”, new variables can be added to the watch grid. The desired variable can be selected from the “Address” combobox under variable properties. Under the “Modifying” tab, modification properties and settings for that particular variable can be set.

A.3.3 Recorder

The recorder starts downloading the recorded samples to the PC in two different modes:

1. Trigger mode.
2. Sample numbers

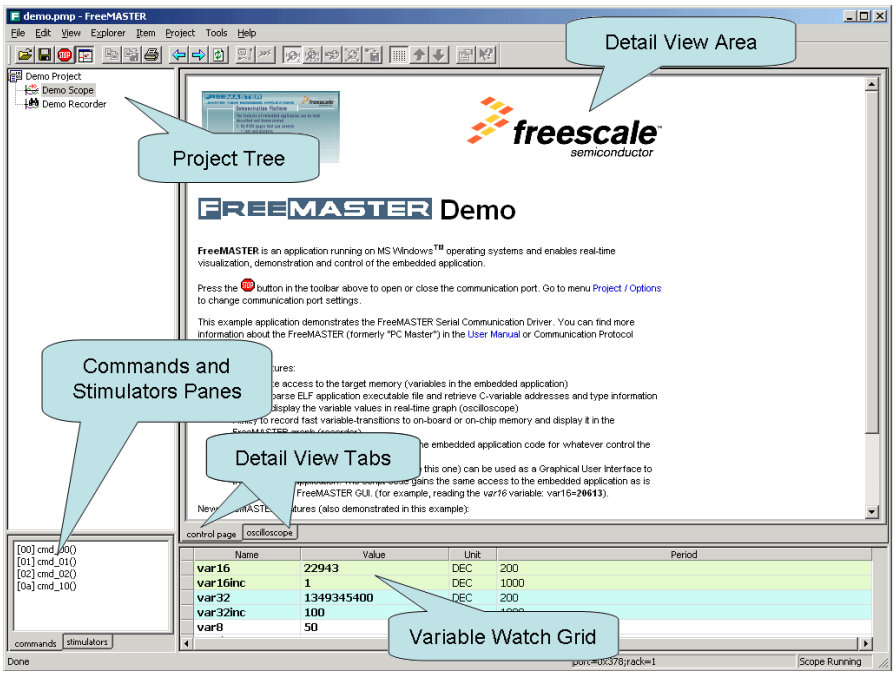

Figure A.3: *FreeMASTER PC software startup page. Built in components of the freemaster project are shown. -FreeMASTER original([34])*

In the trigger mode, the recorder starts downloading the samples on a trigger of an event. This "event" is triggered either upon a value of a particular variable exceeding a predefined threshold, or the user can trigger the event from the PC software by clicking "Run". In the Sampling numbers mode, the sampled values are downloaded once a predefined number of samples have been taken. These settings can be set in the recorder properties under the "Main" tab, and the "Trigger" tab. In the "Main" tab the size of the recorder buffer is shown. The recorder buffer size can be set in "freemaster_cfg.h".

B

Verification experiments on the Extended speedup model

This appendix shows additional experiments of speedup gain and matchings with the theoretical model as discussed and illustrated in Sections 5.3. Figures B.1 to B.4 illustrate the matching of the theoretical model over the real data measurements on the speedup. The erroneous of the theoretical model's compared to real measurements are presen in the lower graphs. This is done by subtracting the real measurements by the theoretical and the result is presented in percentage format. Also, real data measurements were down-sampled by a step of 10 for ease of readability. Figures B.1, B.2, B.3, B.4 and B.5, illustrate the results for extended speedup modeling comparison with the relative experiment, having the workload division coefficient 0.1, 0.25, 0.5, 0.75 and 0.9, respectively.

The lower graph shows a vibrating behavior through all the figures. Because of this vibrating behavior, it seems that the extended speedup model is highly erroneous(up to 20% in Figure B.1). The reason lies behind the execution time of the task in the real measurements. Lets recall how the speedup is measured. First, the task is executed on single core, and the execution time is measured, let it be e_i. Second, the same task is executed on dual core and the execution time is measured, let it be e_j. But, any task's execution time may have a *best case execution time*(BCET), an *average case execution time*(ACET), and a *worst case execution time*(WCET). The vibrations are explained by considering three different cases:

Case 1: e_i is measured in a BCET, while e_j is executed in a WCET.

Case 2: e_j is measured in a BCET, while e_i is executed in a WCET.

Case 3: e_i is measured in an ACET, and e_j is executed in an ACET.

Assumption: *If "e" is the ACET of a task, "e" in WCET is "$e + \delta$" and e in BCET would be "$e - \delta$".*

For Case 1 we have:

$$Speedup_{Case1} = \frac{e_i - \delta}{e_j + \delta} \tag{B.1}$$

For Case 2 we have :

$$Speedup_{Case2} = \frac{e_i + \delta}{e_j - \delta} \tag{B.2}$$

For Case 3 we have :

$$Speedup_{Case3} = \frac{e_i}{e_j} \tag{B.3}$$

Assume that in the real measurements in Figure B.1, for E = 500, Case1 occurs, for E = 501, Case2 occurs, and for E = 502, Case3 occurs. Assuming that "$\delta > 5$", from Equations B.2, B.3 and B.4, it is obvious that speedup will result in a vibration, starting in low edge, raising to an up edge and falling at the end. The continues occurrences of such cases, results in the vibrations in the real measurements. However, these vibration are hard to see in the upper graphs, the affect is easily seen in the lower graphs. Hence, considering such cases, the theoretical model is even more reliable, providing a percentage error less than $\pm 6\%$.

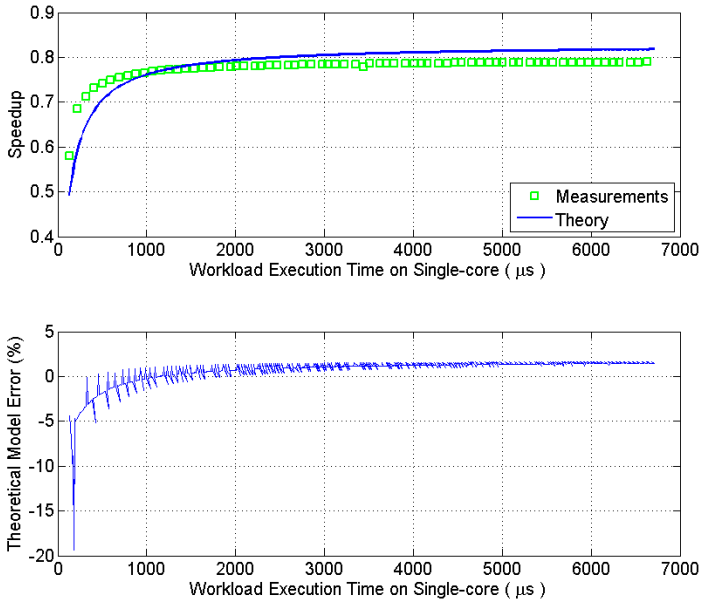

Figure B.1: *Matching theoretical speedup analysis on real data measurements having* $\alpha = 0.10$

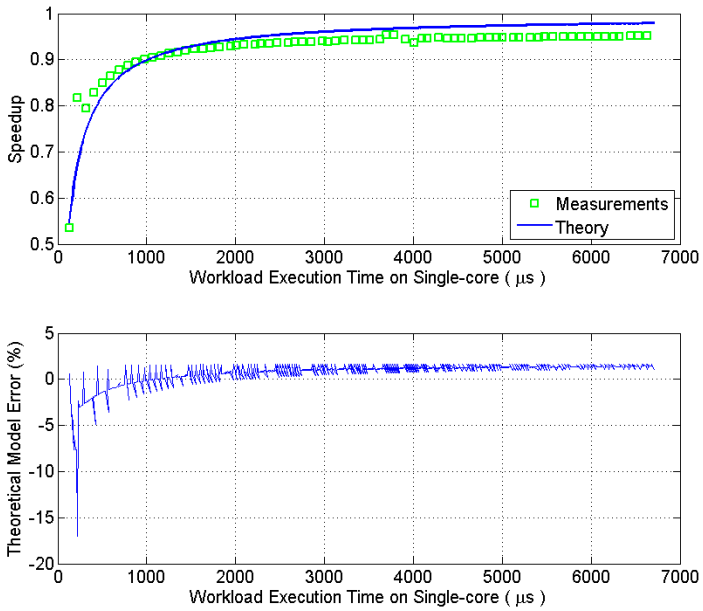

Figure B.2: *Matching theoretical speedup analysis on real data measurements having* $\alpha = 0.25$

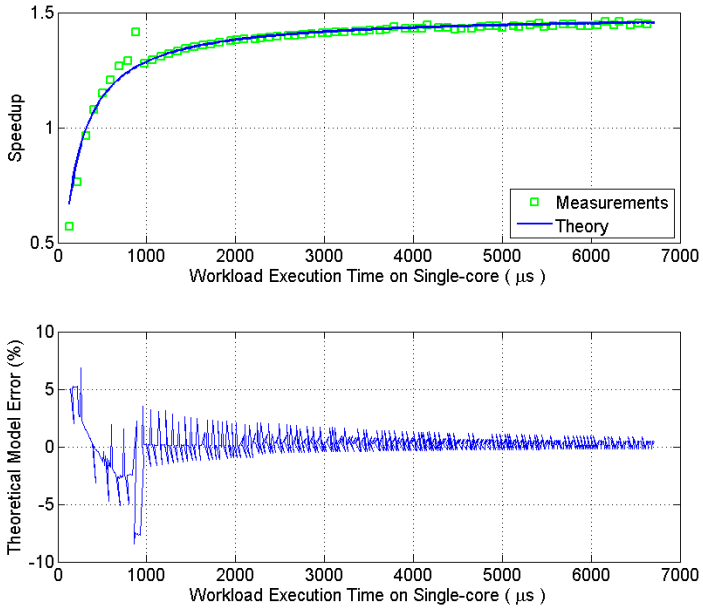

Figure B.3: *Matching theoretical speedup analysis on real data measurements having* $\alpha = 0.50$.

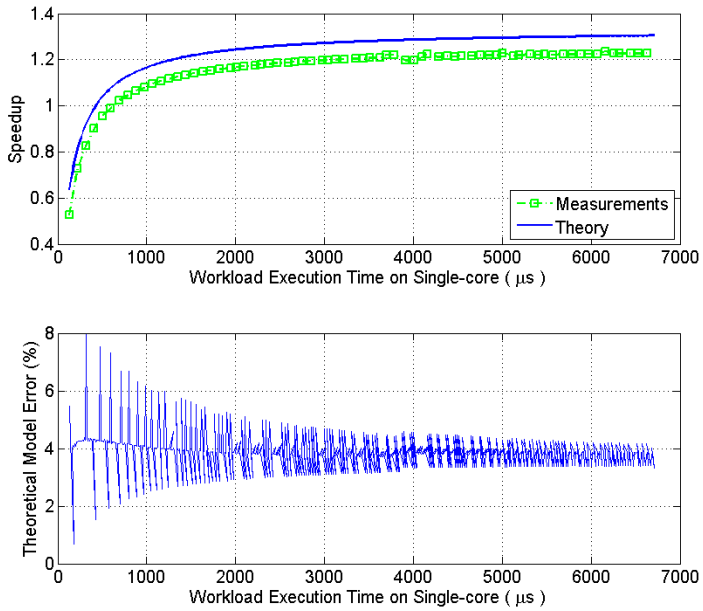

Figure B.4: *Matching theoretical speedup analysis on real data measurements having* $\alpha = 0.75$

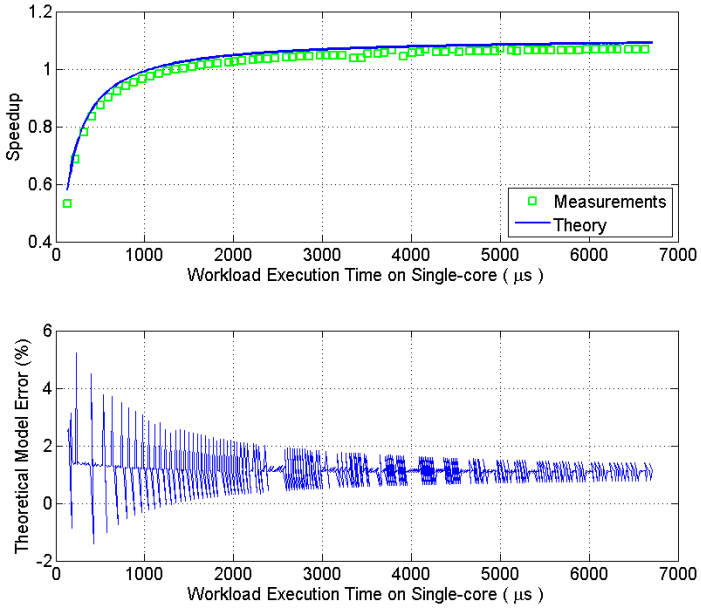

Figure B.5: *Matching theoretical speedup analysis on real data measurements having* $\alpha = 0.90$