CHALMERS

Estimating the Influence on Execution Times from Shared Resources on a Dual-Core Processor

Master of Science Thesis in the Networks and Distributed Systems

PEYMAN BARAZANDEH

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, November 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Estimating the Influence on Execution Times from Shared Resources on a Dual-Core Processor

PEYMAN BARAZANDEH

© PEYMAN BARAZANDEH, March 2016.

Examiner: PER STENSTRÖM

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 March 2016

Acknowledgements

I would like to extend my deepest appreciation for Anders Svensson, from Volvo Group Technology in Gothenburg, who provided me with careful supervision and technical support throughout the entire project. I am more than thankful to him for giving me the excellent opportunities to learn and dive into the areas that I had not touched before.

I also appreciate professor Per Stenström from the department of Computer Science and Engineering at Chalmers University of Technology, who guided me to write and organize my thesis report using his invaluable feedback.

I'm truly thankful to my parents and my lovely wife who relentlessly supported me all the way to do this study, stood by me with their everlasting love and endowed me with their passionate support and encouragement.

Dedication

To my wife Bahar and my lovely son Artin!

Abstract

The goal of this Master's thesis is to estimate the influence of shared hardware resources on the worst-case execution time (WCET) of real-time tasks on a dual-core processor. The studied hardware is MPC5517E Microcontroller which is a dual-core 32-bit microcontroller unit (MCU), manufactured by Freescale Semiconductor. The invented measurement technique scrutinizes the impact of the secondary/slave core on the execution time of a task running on the primary/master core when both cores access a shared resource. First, the slave core is disabled (is not started), and the execution process takes place for a code fragment running on the master core, and the execution time is measured. Then, the slave core is enabled, and the cores run concurrently to access the shared system resources, principally the shared SRAM. The execution time of the task running on the master core is measured for the concurrent utilization of both cores to inspect the impact of the slave core on the master core.

The executable binaries are generated using Freescale CodeWarrior Development Studio for MPC55XX/MPC56XX microcontrollers and the measurement is conducted using Freescale FreeMASTER. To have a quantitative criterion, in order to investigate the influence of the slave core on the master core, we define a factor known as the Slowdown Factor (SDF). The SDF reveals the extent at which the master core is influenced by the slave core. We try to find a upper band for this factor through developing some codes and carrying out a couple of measurements. In our survey, programs run either in the processor's internal flash or RAM. In addition, the primary core runs the same piece of code in all measurement scenarios, whereas the secondary core runs a varying code fragment to expose its impact on the primary core. One remarkable finding is that for a given piece of code, the SDF is not constant and it depends if the code runs in internal flash or RAM. This means that the master core can experience less or more impact from the slave core, depending on the type and target of the code running on the cores.

Contents

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 The Goal of the Project and Contribution 1

2 Background and Related Work 3

2.1 Embedded Real-Time Systems 3

2.2 The Problem Statement and Motivation 5

2.3 Major Issues in WCET Analysis 7

2.4 Classification of WCET Estimation Techniques 8

2.5 Distribution of Execution Time and WCET 9

3 The Investigated Hardware Platform 11

3.1 The e200 Cores 11

3.1.1 Variable Length Encoding (VLE) 13

3.1.2 Memory Management Unit 13

3.2 The MPC5510 Family of Microcontrollers 14

3.2.1 Z1 Core vs. Z0 Core 15

3.3 The Used Development Tools in the Project 17

3.4 The MPC5510 Freescale Evaluation Board 18

4 The Experimental WCET Methodology 21

4.1 Measuring Execution Time in MPC5510 Microcontrollers 21

4.2 The Basics of Execution Time Measurement Technique 22

4.3 Measuring Execution Time using Freescale FreeMASTER 24

4.4 The Slowdown Factor 25

5 The Experimental Results 26

5.1 Exploring the Scenario Leading to the Largest Slowdown Factor.. 26

5.2 The Observed Bugs in CodeWarrior Development Studio 33

6 Conclusions and Future Work 36

6.1 Conclusions 36

6.2 Future Work 37

Appendices 38

A Integrating FreeMASTER with CodeWarrior 39

B Listing of Codes 44

C List of Abbreviations 48

References 49

List of Tables

5.1 Examining the FreeMASTER impact on measurements 31

5.2 The measurements for the surveyed scenarios to find the largest SDF 32

5.3 The measurements scenario to approximate the largest SDF 34

List of Figures

2.1 A real-time task model (image exactly taken from [2]) 5

2.2 Timing analysis of systems (image exactly adopted from [6]) 9

3.1 The Freescale e200 core devices (image exactly taken from [9]) 12

3.2 High-level system architecture of dual-core MPC5510 processors 15

3.3 MPC5510 Evaluation Board (EVB) (image exactly taken from [13]).... 19

3.4 MPC5510 daughter cards (image exactly taken from [13]) 19

3.5 P&E Microsystems USB Multilink hardware interface 20

A.1 CodeWarrior IDE integrated with embedded-side FreeMASTER........ 41

A.2 Real-time display using PC-side FreeMASTER 42

A.3 Displaying a recorded variable using PC-side FreeMASTER................ 43

B.1 Code listing for the basics of execution time measurement technique ... 45

B.2 Code listing to measure execution times using Freescale FreeMASTER 47

1 Introduction

Multi-core processors are becoming rampant these days in a wide variety of applications and embedded systems are not an exception to this trend. In automotive industry, the competition among manufacturers is cutthroat, and there exists an ever-increasing desire to add novel functionalities and features to vehicles at reasonable costs in order to be able to strive in the competitive market. These functionalities range from entertaining applications, like the infotainment, to safety-critical functions, such as airbag and Anti-lock braking system (ABS). Multi-core processors enjoy high performance and are efficient in terms of cost and power consumption; hence they are very appropriate candidates to provide the sophisticated automotive software tools with their demanded underlying hardware requirements. Most of the applications in the automotive domain fall into the category of hard real-time embedded systems, in a sense that the precise functioning of the system is contingent upon meeting the timing constraints, namely the tasks deadlines.

1.1 The Goal of the Project and Contribution

The goal is to estimate the influence of shared hardware resources on the execution times, or more accurately saying, on the WCET of tasks, running on a dual-core processor. The approach is a measurement-based technique that involves developing a program to run some task and measuring its execution time. The target processor has two cores, known as the master and slave cores. The invented measurement technique, first executes the code, in isolation on the master core, meaning that the slave core performs no activity in the beginning. Then, different kinds of workload are put on the slave core to exploit both cores simultaneously, and the execution times for the task running on the master core are measured while both cores are operating concurrently. The objective is to investigate the impact of the stress imposed on common hardware resources, mainly the shared memory, due to concurrent access of both cores to such resources. Freescale MPC5510 contains a set of single- and dual-core 32-bit MCUs based on the Power Architecture. MPC5517E from this family is evaluated in this project. This processor functions at speeds up to 80 MHz and is used for vehicle central body and gateway applications in automotive embedded systems [20].

The rest of the report is organized as follows. Chapter 2 explains the main concerns associated with WCET analysis. Issues like why it is not a straightforward problem, and how the WCET estimation techniques are classified, are explained in this chapter. Chapter 3 describes the features of the investigated hardware platform in this project. Chapter 4 presents our methodology and the developed measurement approach. Chapter 5

explains the carried out experiments and their obtained results using the devised measurement technique in Chapter 4. Chapter 6 concludes the report and presents the possibilities for the potential ongoing work. The report has three appendices as well. Appendix A gives some details on how to make use of FreeMASTER application. Appendix B provides the source codes that were developed in the investigated scenarios, and Appendix C expresses the used abbreviations in the report.

2 Background and Related Work

The status quo of WCET study is not satisfactory since industry suffers from the lack of competent WCET analysis tools [15]. The advanced architecture of modern processors such as cache hierarchies and pipelining has resulted in considerable performance increase in computing. However, these features have caused the instructions of these processors have time-varying properties which make their timing analysis a cumbersome task. The reason is that caching or pipelining retains an execution-dependent internal state within the processor that can be figured out as the execution history. Any calculation made based on the previously-executed instruction streams in state-of-the-art processors is typically inaccurate, and as a result WCET analysis is challenging in such processors. Another issue is that WCET analysis cannot rely on the data sheets presented by manufacturers of processors because timing of instructions available on these documents are generally a rough estimate of real situations [14].

2.1 Embedded Real-Time Systems

Embedded systems play a key role in our today's world and we have been surrounded by a myriad of them. There are a plenty of gadgets around us that their operations are partly or wholly dependent on computer-controlled systems. The computer systems applied in these devices, are usually invisible to us because they themselves are the embedded computing components of a bigger machine. Consequently, we as the users of these machines are usually unaware of the fact that there is a hidden computer system contributing to the machine's operations. Home appliances (washing machines, dish washers, microwave ovens), transportation vehicles (cars, trucks, airplanes), submarines, satellites, mobile phones, industrial and space exploration robots (the Mars rovers), biomedical instruments (ultrasound imaging equipment, patient monitoring systems), office machines (printers, faxes, copiers, calculators) and consumer electronics gadgets (TVs, DVD players, security tokens, digital cameras) are all the ubiquitous cases of machines equipped with embedded systems. It is evident that our life has become heavily dependent on these so-called embedded systems, and this reliance will continue with an increasing drift in future. Microcontrollers have become pervasive in the automotive industry due to wide-ranging application of ECUs in a vehicle. They are exploited in a variety of applications including,

- body
- gateway
- chassis and safety
- dashboard

- engine management
- advanced driver assistance
- controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay and other peripherals
- entry-level
- cluster applications
- powertrain
- electric power steering

Many of the services, offered by the systems mentioned above, are demonstrated by real-time systems. In [1] we have,

A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated.

In a real-time system, a task should complete its execution within an allowed time frame. The temporal properties, mainly responsiveness and periodicity, are the strict timing constraints of a real-time system [2]. Responsiveness indicates that the system should be able to complete execution of tasks before their deadlines, and periodicity designates the system's sampling rate, i.e. the period at which tasks arrive (if tasks arrive periodically). If the real-time system fails to deliver its response within the permitted time frame, the computation results, although logically correct, might be useless. Depending on the type of timing constraints (hard or soft), and also the nature of system's operation in terms of safety (non-critical, critical and safety-critical), a timing failure can have different consequences. For example, in a car, airbag system and ABS are safety-critical systems, since the deferred release of airbag after a crash in high speeds or malfunctioning of ABS in a slippery and icy road can endanger passengers and cause serious injury or loss of life.

When it comes to real-time applications compared to other domains, different criteria come into the picture to evaluate the correctness of delivered output. As a case in point, in High Performance Computing (HPC), the ultimate objective is to maximize the average throughput. Nevertheless, in real-time systems the average throughput cannot be regarded as a criterion to judge about the correctness of generated results by the system. Depending on the type of application domain, a real-time system is typically optimized with regard to a definite criterion for the provided functionality or service. Dynamic control systems and multimedia applications are two application examples. In dynamic control systems, maintaining system robustness is the desirable objective and in multimedia applications the goal is to have comfort in the provided service.

2.2 The Problem Statement and Motivation

Any development process contains a number of stages or phases. The development process of a real-time system involves the following three major phases,

- specification
- implementation
- verification

It is absolutely essential to verify the system's capability in meeting its temporal characteristics for critical and safety-critical real-time systems. The verification should be carried out before the system is put into the actual mission. Selecting the suitable scheduling algorithm for the real-time tasks is an integral part of system design. The scheduling algorithms require having the tasks model. A task has temporal behaviors and these behaviors can be either static or dynamic [2]. Static parameters are derived within the specification or implementation stages and are not dependent on other tasks. Dynamic parameters involve those parameters that reflect the impacts imposed on one task by other tasks during the execution of tasks set.

Figure 2.1 illustrates a real-time task model. Only four static parameters are presented in a task model. These parameters are C_i, D_i, T_i and O_i, and are described as,

- C_i: task's worst-case execution time (WCET)
- D_i: task's relative deadline, indicates the responsiveness timing constraint
- T_i: task's period, indicates how frequently the task is released
- O_i: task's time offset, indicates the first instance of task arrival, e.g., the earliest time at which the task executes

WCET is defined as the longest possible undisturbed execution time for an iteration of the task [2]. Among the four static parameters in the task model, determining C_i or WCET is not trivial and it is a challenging factor to identify this parameter in developing a real-time system. Execution time of a task depends on several factors including [2],

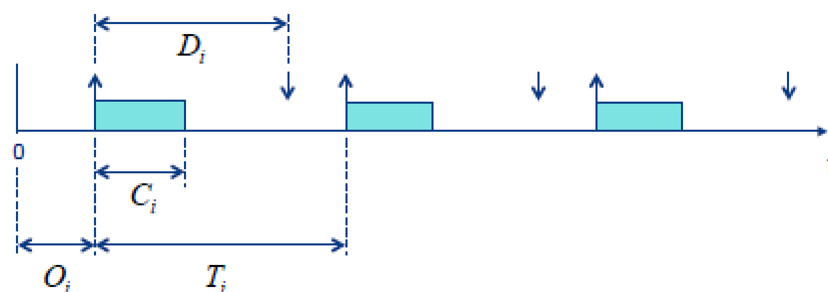

Figure 2.1: A real-time task model ([2])

- structure of the program and its input data, e.g., loop iterations, branching and conditional statements
- system properties and architectural factors, i.e. hardware as well as operating system, e.g., caches, memory hierarchies, out-of-order/speculative instructions execution and pipelining
- initial system state
- internal and external events of the system, e.g., system interrupts, exceptions handling and context switching

It is essential to have an estimation of WCET to allocate a reasonable percentage of CPU utilization to the execution of a periodic task, and construct a *feasible* schedule which is verifiable within the verification phase of system development.

A hard real-time system should complete the task execution within its allowable timing constraints (deadlines must be met) and WCET is a parameter of great importance in the schedulability analysis of such systems. In [4] it has been stated that,

The purpose of worst-case execution time (WCET) analysis is to provide a ***priori*** *information about the worst possible execution time of a piece of code before using the code in an actual system.*

There exist three different types of tasks in a real-time system [2],

- *periodic task*, task is released with a specific and constant period of T_i
- *aperiodic task*, task is released at time intervals greater than a specific period, i.e. release time $\geq T_i$
- *sporadic task*, task does not have any guaranteed minimum bound between its two consecutive releases

A WCET analysis method aims at deriving an upper bound for the time that it takes a piece of code to execute on a specific platform [6]. It is not a trivial attempt to define the real WCET and it is typically sufficient to have an estimation of the execution times for a specific task, hence the following relation applies,

$$0 \leq \text{Estimated WCET} - \text{Real WCET} < \varepsilon \qquad (2.1)$$

Regarding hard real-time systems, the method must satisfy two requirements [2],

- must be pessimistic to guarantee that the timing constraints considered in the schedulability analysis are also valid at run time

- must be tight to avoid unnecessary allocation of hardware resources, mainly CPU and memory, to the scheduled tasks set

A particular system consists of,

- organizational platform, i.e. the underlying hardware as well as the running system software
- structure of the program executing on the platform

Given the particular system and the formerly-mentioned factors affecting the execution time,

WCET problem deals with finding the worst-case execution time for all likely input data vectors, initial system states and internal and external system events for a particular platform.

The above statement formulates WCET problem.

A real-time system is specified, implemented and verified assuming that the tasks set WCET executing on the target hardware is known to a relatively accurate extent. The estimated WCET suffices to perform schedulability analysis for a set of real-time tasks on single-core systems under the available scheduling algorithms such as Rate Monotonic/Deadline Monotonic (RM/DM) or Earliest Deadline First (EDF) algorithms [2].

2.3 Major Issues in WCET Analysis

Relation 2.1 specifies the two necessary constraints that any approach dealing with WCET estimation for a particular system should meet: pessimistic and tight. Regarding these two constraints, there exist issues in analyzing the possible execution paths of a given program. How to pessimistically place a limit on WCET and how to remove the false paths, i.e. the paths that are not taken at runtime, are the two major concerns in finding an effective analysis model. Moreover, there are challenging issues in specifying the timing behavior of a system. Hardware properties of the system, such as cache memories and pipelining, make it a big deal to define an efficient and accurate temporal model for intended platform. Another important aspect is taking the impact of system events into account. Interrupts, context switching and exceptions are the examples of system events.

Modern processors, irrespective of being single- or multi-core, have a variety of advanced attributes. Cache hierarchies, pipelining, branch predictions, out-of-order and speculative execution of instructions are such attributes. As for pipelining, structural

conflicts, data conflicts and branch conflicts are the causes of inaccuracy in timing analysis. Cache misses are also a substantial source of timing variations [2]. It is hard indeed to precisely model these influencing factors in WCET analysis and define an exact timing model for such processors. In order to have a quite exact and competent WCET analysis model, any factor or parameter affecting the temporal behavior of the platform must be pessimistically modeled. Besides, practicable and feasible assumptions should be made in constructing a timing model. In the context of multi-core embedded processors the situation is more intricate, because the current techniques to measure/estimate the tasks execution times on single-core processors are not fully applicable on multi-core processors. Although multi-core processors offer better performance than single-core processors, they are very complicated to analyze [7]. The reason is that inter-core interferences arise among executing tasks, while the tasks are accessing a shared hardware resource. Such interferences disallow the available single-core processor models to provide a reliable timing analysis for multi-core processors. System bus, Level 1 (L1) and Level 2 (L2) instruction and data caches are the cases of shared resources.

2.4 Classification of WCET Estimation Techniques

In general, there are two categories of techniques in the timing analysis of real-time systems concerning WCET estimation [6],

- static
- measurement-based

In static methods, the program does not execute on real hardware target or on a simulator. Instead, the code implementing the task is taken into account to derive all possible control flow of execution paths. The code may be annotated in this step. Then the derived control flow is associated with some (abstract) model of the hardware platform to determine the upper execution bounds for the resultant association.

Static WCET analysis has some shortcomings. One noticeable problem is that it is a rather time-consuming technique because deriving a relatively exact model for a complex processor requires considerable amount of time. Further, the technique is processor-dependent and cannot be extended to other processors, even with slight architectural differences. Another important challenge is the inaccuracy of instructions timing written down on a processor technical documents presented by its vendor. Such timings are usually some approximate figures for the execution times of processor instructions and are not exact; therefore any approximation for execution times of an annotated piece of code is inaccurate and cannot be trusted in real-time applications.

Measurement-based techniques execute the real task or some parts of a partitioned task on the real hardware target, or on a simulator, for some vector or some set of input data to

measure execution times. The WCET is specified by the measured minimum and maximum execution times for the whole task, their distribution, or combined measured times for execution times of code snippets of the partitioned task. This is an effective WCET analysis approach, but the major concern is that it is almost infeasible to carry out exhaustive end-to-end measurements for the programs with large size of code. To enjoy the best of both worlds, the well-liked trend is to apply the hybrid approach, i.e. to combine measurement-based techniques with static methods. There are some WCET analysis methods available using the hybrid methodology, but they are still immature [14].

Path analysis is a common technique in static WCET analysis and it is being replaced by test data generation method in measurement-based approach [14]. In WCET analysis using test data generation, the user manually generates various sets/vectors of synthetic data or simply generates random data sets. The adopted approach in this project is a measurement-based technique and will be described in detail in Chapter 4.

2.5 Distribution of Execution Time and WCET

A real-time system comprises some tasks and each task implements some part of the overall functionality of the system. As it was described before in Section 2.2, the execution time of a task is a varying parameter and depends on several factors. The varying nature of execution time results in having a set of execution times, as opposed to just a single value, and allows us to define some real-time pertinent properties for a particular task. Figure 2.2 [6] illustrates the involving fundamental terms in timing analysis of real-time systems. The lower curve points out a subpart of the measured

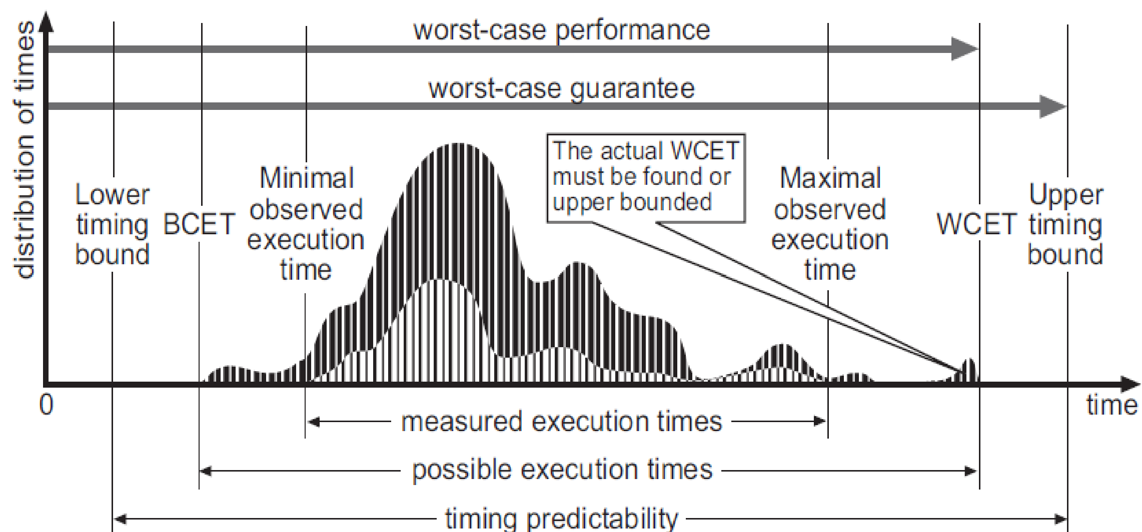

Figure 2.2: Timing analysis of systems ([6])

execution times in which there are minimum and maximum values for the measured executions. The upper curve indicates the superset of all possible execution times and its lower and upper bound characterize the best-case execution time (BCET) and worst-case execution time (WCET) of the task, respectively.

3 The Investigated Hardware Platform

Microcontrollers have become prevalent in the automotive industry thanks to broadly usage of ECUs in a vehicle. The MPC5510, presented by Freescale Semiconductor, is the first product family of 32-bit low-power microcontrollers based on the Power Architecture technology, designed for automotive body and gateway applications. All processors of the family have program flash and SRAM (with different sizes depending on the processor) and can support a range of advanced peripherals. The MCUs of this family are capable of accessing the flash memory and peripherals in one clock cycle. Different vendors active in the automotive embedded systems provide the MPC5510 with extensive support through offering quality hardware and software development tools built on the Power Architecture technology.

3.1 The e200 Cores

Freescale Semiconductor offers a series of dual-core 32-bit microcontrollers utilizing two e200 cores. The cores employ the Power Architecture presented by PowerPC [9]. Power Architecture has demonstrated its efficiency in a wide range of embedded applications including, but not limited to, automotive industry, robotics, signal processing, compact networking, industrial control and health equipment. Currently, Freescale presents four flavors of e200 cores: e200z0, e200z1, e200z3 and e200z6. Freescale's e200 architecture provides cost-sensitive, embedded real-time applications with substantial performance needs. Figure 3.1, exactly adopted from [9], illustrates the fundamental characteristics of the available e200 cores.

The Z0 is the simplest one among the four cores and the Z6 is the most complex core offering uppermost performance in the family. All cores exploit the Power Instruction Set Architecture (ISA), version 2.03, and have support for Variable Length Encoding (VLE). Besides, all cores, excluding Z0, provide full implementation for 32-bit Power architecture Book E instruction set. The dual-core processors of the family sets have two dissimilar cores that are called the master and slave cores throughout this text.

With regard to the dual-core Freescale Semiconductor MCUs, both cores can thoroughly act independent of each other; therefore they can either execute completely isolated tasks or execute partially or fully dependent tasks. The subject of dividing the code between the master and slave cores have been extensively discussed among embedded applications developers. Depending on the type of applications, there are various options available to efficiently trade off the running loads between the two cores. Some well-known scenarios are [11],

- dividing the code between cores meticulously to relieve the burden on the master core
- dedicating the slave core to interrupt-handling routines
- utilizing the slave core to accomplish an isolated functionality such as implementing gateway operations between CAN and LIN
- allocating the slave core to implement a computationally-intensive task and making calls to the task from the master core when it is needed
- allocating the slave core to carry out the error checking of the tasks being executed by the master core

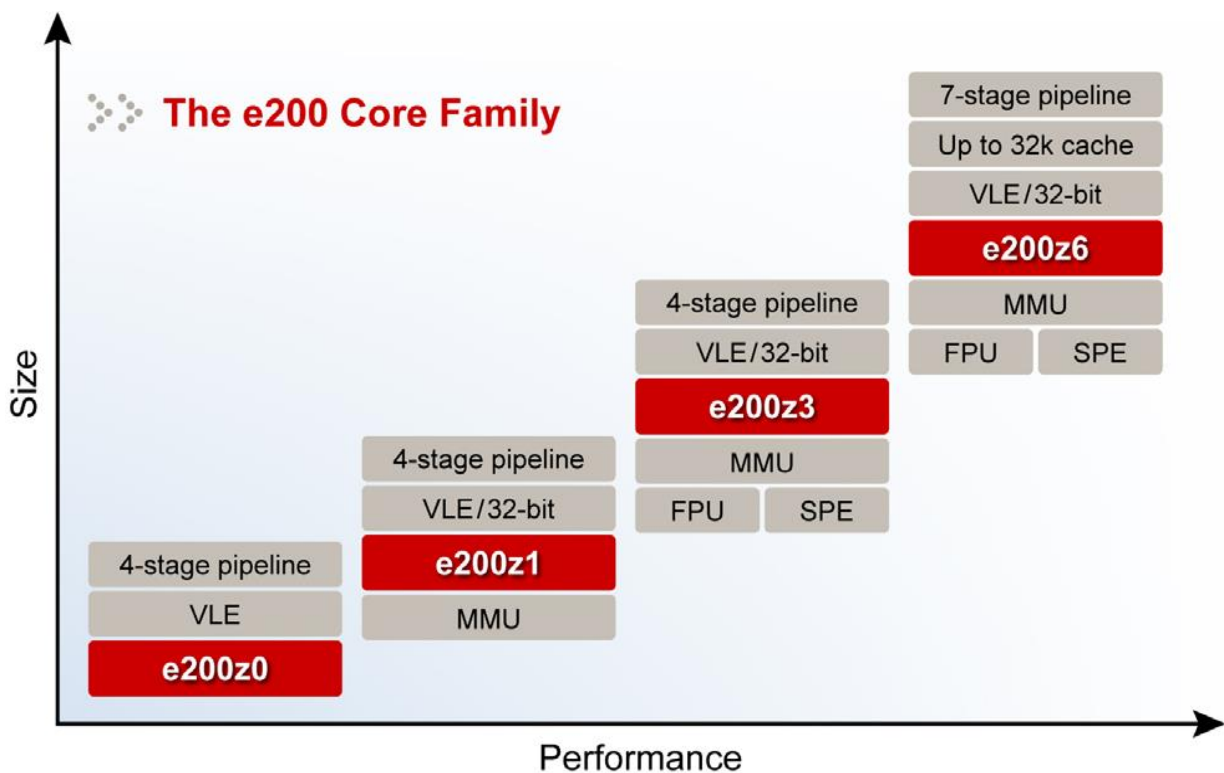

Figure 3.1: The Freescale e200 core devices ([9])

The Memory Management Unit (MMU), present on the Z1, Z3 and Z6 cores, is suitable mechanism for the applications requiring full Operating System (OS) support. The Signal Processing Engine (SPE) and the Floating Point Unit (FPU) in the Z3 and Z6 deliver signal processing capabilities for the applications performing signal processing computations. These two units typically remove the need for providing the microcontroller with a supplementary Digital Signal processor (DSP). The Z6 core enjoys seven-stage pipelining and is endowed with all the features of the Z3 core as well as an on-chip cash memory [9].

The investigated processor in this project is MPC5517E microcontroller, belonging to the MPC5510 family of microcontrollers. The processor is equipped with the Z1 and Z0 cores; hence the two important functionalities provided by this processor are VLE and MMU. The following, describes these advanced hardware-level capabilities in brief.

3.1.1 Variable Length Encoding (VLE)

VLE is a technique to re-encode the Power instruction set with the help of 16- and 32-bit formatting. Freescale developed VLE to optimize code density through encoding 32-bit PowerPC instructions into mingled 16- and 32-bit instructions in order to decrease the code footprint. It is an enhancement to the POWER (Instruction Set Architecture) ISA. It is possible to achieve high performance and code density through inter-mixing the code pages utilizing VLE formatting or non-VLE formatting. This performance improvement is yielded by exploiting the 16-bit space-efficient binary illustrations of Power instructions versus the more lengthy 32-bit instructions. VLE employs both 16- and 32-bit instructions; hence it is possible to achieve considerable code density at the expense of negligible or even no performance loss [11]. Instructions are prefixed by "se_" and "e_" to designate 16- and 32-bit VLE instructions correspondingly. For instance, the `add` instruction in the VLE format is,

- `se_addi`, is 16-bit VLE "add immediate" instruction
- `e_add16i`, is 32-bit VLE "add immediate" instruction

It is possible for the compiler to choose whether to use Power Architecture Book E or VLE by the help of a switch at compile time.

As for the embedded software applications, the achieved density in code can lead to less system cost, and also to some lesser extent, improved performance. It is possible to condense the code up to 30% using this feature via free intermingling of 16- and 32-bit instructions [9]. All four types of e200 cores back up VLE and the feature is almost offered by all development toolchains. The feature has been accepted by power.org and Power ISA (version 2.03) supports it. The Freescale's comprehensive manual [10] describes VLE in detail.

3.1.2 Memory Management Unit (MMU)

All e200 cores, except the Z0, possess an MMU. This unit is the same in the Z1, Z3 and Z6 cores, in a sense that it provides equal functionality, identical user interface and compatibility in cross-core program code. The applications requiring full OS capabilities are very appropriate candidates to be deployed on the MMU. The MMU outlines different memory sections. One of the fundamental parameters being set by the MMU is to

determine whether the memory section should contain Power Architecture Book E or VLE code. This way, the Z1 core can decide if it should use Power Architecture Book E encoding or VLE formatting. The significant benefit is that there is no need to transform the current libraries developed in Power Architecture Book E encoding scheme to VLE formatting scheme. The Z0 core is only compliant to VLE scheme; consequently it lacks an MMU [11].

The characteristics of MMU (adopted from [9]) include,

- translation from 32-bit effective (virtual) to 32-bit real (physical) addresses:
    - 32-entry MMU in Z6
    - 16-entry MMU in Z3
    - 8-entry MMU in Z1
- support for nine page sizes (4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 64 MB, and
  256 MB)
- accesses qualified by:
    - Address spaces: 2 data and 2 instruction
    - 8-bit process identifier (supervisor accessible or global resource)
- selectable access privileges:
    - User Read/Write/Execute (UR/UW/UX)
    - Supervisor Read/Write/Execute (SR/SW/SX)

Furthermore, it falls to the MMU to select the endianness of memory section for each core. The Z1 core is capable of using either big endian or little endian, while the Z0 core can only use big endian.

3.2 The MPC5510 Family of Microcontrollers

The MPC5510 family is an extensively-used set of 32-bit microcontrollers in the automotive embedded applications, based on e200 cores family. The family set includes a number of single- and dual-core cost-efficient microcontroller units (MCUs) capable of operating with outstanding performance at low power consumption modes. The dual-core members of this family are endowed with e200z1 (Z1) and e200z0 (Z0) cores. The Z1 and Z0 cores are also known as the primary/master and secondary/slave cores, respectively. Both MPC5510 single- and dual-core processors provide support for Power Architecture Book E instruction set. A majority of the family members have a second core (Z0), intended to act as the Input/Output Processor (IOP). The secondary/slave core provides the processor with a number of powerful features. This core is brought out of reset by the primary/master core, and just the once it is in the functional mode it can

operate autonomously of the primary/master core. The secondary/slave core can carry out a variety of tasks such as [11],

- handling I/O processing
- executing gateway functionalities between communication networks
- creating virtual peripherals
- offloading the burden running on the master core via performing some of necessary system tasks

Figure 3.2 [12], shows the high-level system units of a dual-core processor belonging to the MPC5510 devices.

3.2.1 Z1 Core vs. Z0 Core

Figure 3.1 illustrates that the e200 cores have some structural differences leading to different performance levels and functionality for each core. Thus, the applied dual-core organization in MPC5510 devices results in an asymmetric implementation. On the other

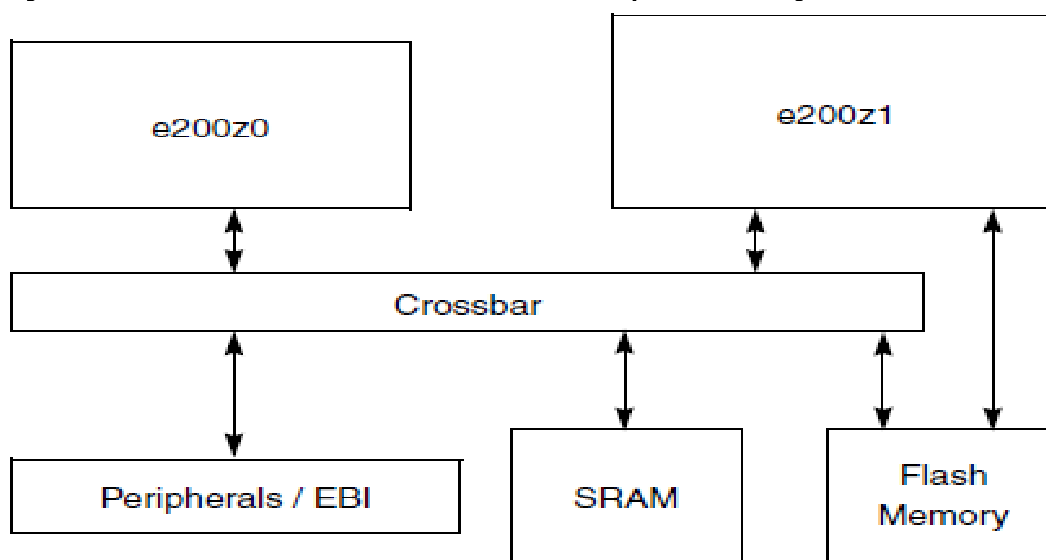

Figure 3.2: High-level system architecture of dual-core MPC5510 processors [12]

hand, there is symmetric multi-core architecture in which all cores have almost the same structural design and implementation, usually with identical functionalities present on each core. Most of the times, there is an OS running on the platform (symmetric or asymmetric multi-core systems) which dynamically decides how the execution of tasks should be allocated to the cores. This decision is made by the OS based on the available system resources and also according to the time-related necessities of the running

application. Typically, it is not needed to run an OS on the dual-core processors of MPC5510 family [12]. The master core generally executes the main control loop and the slave core is launched to accomplish certain jobs, like gathering data from various external devices.

When it comes to the dual-core processors of MPC5510 family, such as MPC5517E, the Z1 and Z0 cores are involved. Subsequently, it would be beneficial to focus on the substantial differences between these two cores. The most important dissimilarities in Z1 and Z0 are,

- the supported instruction set by each core
- the hardware-wise type of access to shared resources
- the core registers

The Supported Instruction Set by each Core

As stated earlier, one of the major differences between the master core (Z1) and the slave core (Z0) is the type of the instruction sets that they can execute. Z1 can exploit both Power Architecture Book E and the extended VLE instruction sets whereas Z0 executes only VLE instruction set.

The Hardware-wise Type of Access to Shared Resources

The Z1 core utilizes the Harvard architecture, indicating that it has separate instruction and data buses, while the Z0 core makes use of Von Neumann architecture, specifying that it has an individual, unified instruction/data bus. There are also variants of the Z0 core that utilize Harvard architecture, belonging to non MPC5510 family set. As demonstrated on Figure 3.2, the crossbar switch is in charge of managing and arbitrating access to the shared resources. The supplementary direct link between the Z1 core and the flash memory provides this core with the immediate access to the flash memory through bypassing the crossbar switch. This extra link decreases the access time and also facilitates the simultaneous access of both cores to different locations of flash memory [12]. Cores have their own individual interrupts set and run at the same speed supplied by a shared clock source which is configured in the System Integration Unit (SIU).

The Core Registers

The master core contains all the slave core's registers as well as a few additional registers which enable it to implement specific functionalities. In [12] the differences have been listed as following,

- the Z0 core lacks the timing registers available on Z1, including the two 32-bit Time-Base (TBL and TBU) and the decrementer registers
- the Z1 core has eight General Special Purpose (SPRG0 to SPRG7) and one User Special Purpose (USPRG0) registers, whereas Z0 core has only two Special Purpose registers (SPRG0 and SPRG1)
- the Z1 core has the branch target buffer to accelerate the execution of loops, but the Z0 core lacks this feature

In this project, the basic principle in the measurement of execution times is using the Time-Base registers on the Z1 core.

3.3 The Used Development Tools in the Project

There are several development tools presented by different vendors to provide embedded software developers with their needed tools to generate, compile and debug their program code or model/simulate MPC5510 MCUs. The following, taken from [9], lists the available tools. The software and hardware tools denoted with a check mark were used in this thesis work. Freescale FreeMASTER does not exist in the tools list provided by [9].

- **Compilers**
  - ✓ CodeWarrior Development Studio
  - – Green Hills
  - – WindRiver
  - – GNU
- **Debuggers**
  - – Green Hills
  - – Lauterbach
  - – iSystem
  - ✓ P&E Microcomputer Systems USB Multilink
  - ✓ Freescale FreeMASTER
- **Simulators**
  - – CodeWarrior (Core only)
  - – Green Hills (Core only)
- **Hardware**
  - ✓ Freescale Evaluation Board (EVB)
  - – Green Hills (Nexus Class 1 and Class 2+)
  - – Lauterbach (Class 1 and Class 2+)
  - – iSystem (Class 1 & Class 2+)
  - ✓ P&E Microcomputer Systems USB Multilink (Nexus Class 1)
- **Initialization tools**

- RAppID Init

- **Modeling/Code generation**
  - dSpace TargetLink
  - MathWorks Simulink

3.4 The Freescale MPC5510 Evaluation Board

Freescale Semiconductor presents an Evaluation Board (EVB) to ease hardware and software development processes for MPC5510 family of microcontrollers and also provide customers with an an easy-to-use piece of hardware to assess the capabilities of the processors belonging to this family of microcontrollers. The EVB is planned to be used in bench/laboratory applications under normal ambient tempratures [13]. Figure 3.3 shows this EVB.

The main board lacks an MCU and the MCU is mounted on an MCU daughter card. The EVB has a modular organization in the sense that different MCU packages can be installed on the supporting daughter card; hence the EVB can enjoy full flexibility and simplicity. At present, there are three packages to be used with the MPC5510 processors and by the EVB as well. These package types are 208BGA, 176QFP and 144QFP. This mechanism allows the MPC5510 microcontrollers to be exploited with the same EVB but with various package types and MCU derivatives. The daughter board is interfaced with the EVB using high density connectors [13]. The switches, jumpers and user connector on the EVB make it possible to provide the hardware with a couple of different configurations and settings. Power supply configurations, clock sources selection, reset control, debug configuration, external memory configuration, CAN, RS232, LIN and Flexray configurations, the LED dot matrix and termination resistor control are the various features that can be flexibly configured according to the needs. Figure 3.4 displays the available three daughter cards supporting the available three MCU packages.

The explored processor in this project (MPC5517E) uses 144OFP daughter board and the target hardware includes the EVB and daughter card existing in the Qorivva MPC5510 kit. The kit includes an EVB equipped with a Nexus connector through which

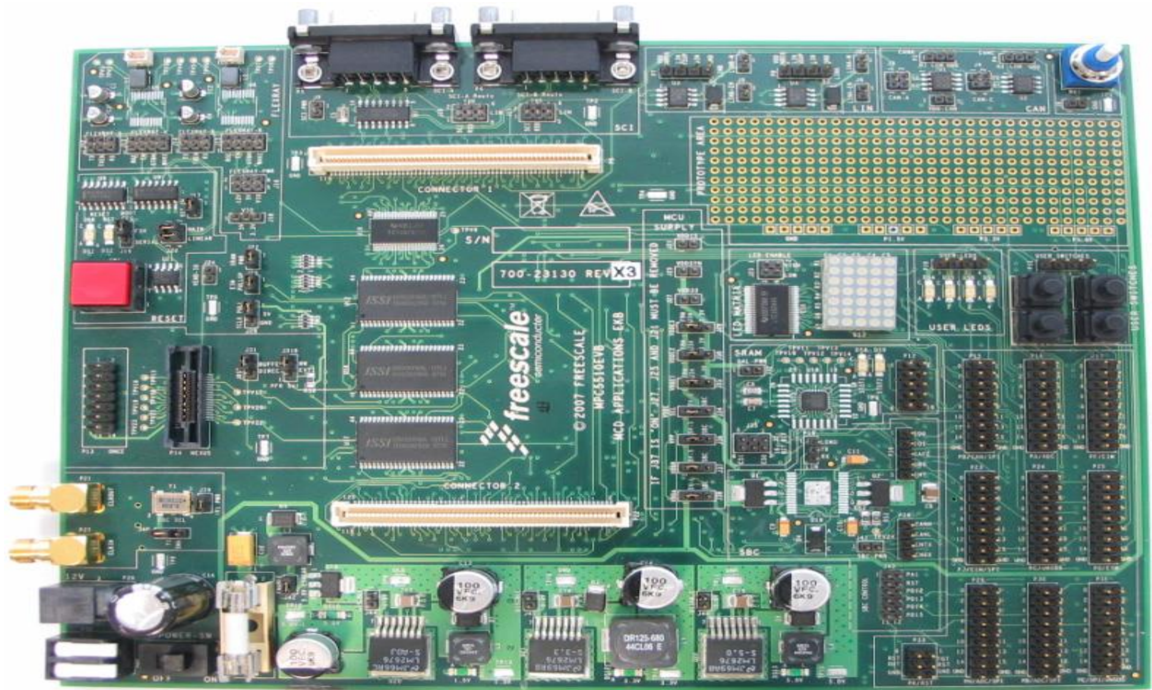

Figure 3.3: MPC5510 Evaluation Board (EVB) ([13])

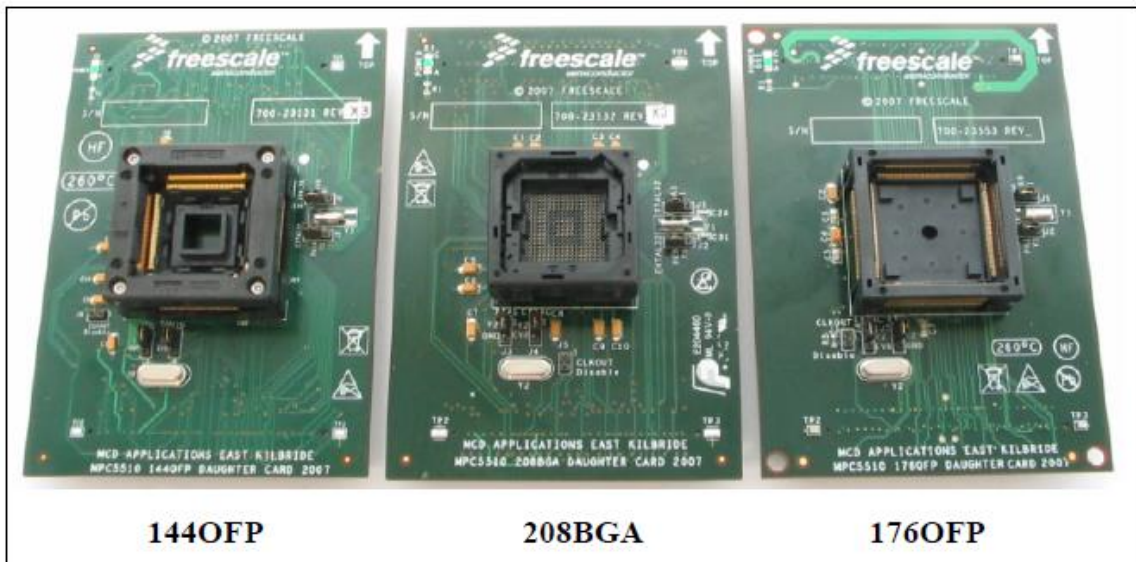

Figure 3.4: MPC5510 daughter cards ([13])

it is possible to have full debug access to the MCU. The codes are compiled using the free license of CodeWarrior Development Studio that limits the code size up to 128 KB. To upload and debug the created elf, P&E Microcomputer Systems USB Multilink interface

that comes with the Qorivva MPC5510 kit has been used. The MPC5517E has been plugged into the daughter card. This processor has the following fundamental features,

- one e200z1 (the master core) and one e200z0 core (the slave core)
- 1.5 MB internal program flash, 80 KB on-chip SRAM

Figure 3.5: P&E Microsystems USB Multilink hardware interface

The P&E USB Multilink is the hardware interface, placed between PC and the EVB, in order to transfer the generated executable binary to the processor. Figure 3.5 shows the interface. The executable binary can be placed into flash memory or SRAM, depending on the code size and the size of available SRAM.

4 The Experimental WCET Methodology

Our intention is to develop a measurement-based mechanism to estimate the influence on execution times of the task running on the master core of MPC5517E, when both master and slave cores access the shared resources. As stated earlier, in Chapter 2 (Section 2.4), measurement-based WCET techniques for multi-core processors are more promising than static WCET analysis methods. The developed approach in this work utilizes the features that are available on the investigated hardware, as well as a real-time debugging tool, to measure the execution times of a piece of code running on the master core. In the beginning, only the master core starts executing a code snippet as the baseline for measurement and the slave core does not start, i.e. it stays in reset mode. Then the slave core is brought out of reset by the master core, and starts executing a number of background loads. Using the real-time debugger, the impacts on the execution time of the piece of code running on the master core, due to concurrent operation of both cores, are inspected.

4.1 Measuring Execution Time in MPC5510 Microcontrollers

The master core in MPC5510 microcontrollers has two 32-bit registers, identified as Time Base (TB) registers [16]. These registers belong to the category of special purpose registers (spr) in the MPC5510 MCUs and they both together form a 64-bit TB register. Special purpose registers are not memory-mapped and they can be accessed indirectly through a general purpose register (gpr). Concerning an spr, there are only two instructions available: `mtspr` (move to spr) and `mfspr` (move from spr). The lower and upper 32 bits are denoted as TBL and TBU, correspondingly. Using the TB registers, it is possible to count the number of system clocks involved in the execution of a code fragment. The MPC5510 features different memory types including, internal flash, on-chip SRAM, external flash and off-chip SRAM. The TB regiters are capable of counting the number of system clocks for the code executing in any of these memory types.

Before the TB registers start counting the number of elapsed system clocks, they must be enabled. There is a special puspose register in the primary core, named as Hardware Implementation Register 0 (spr HID0) having a bit known as Time Base Enable (TBE).
To enable the TB registers, it is enough to set this bit. Once being enabled, the TB registers start counting the number of clock cycles after an initial delay owing to the pipelining feature. The primary core (e200z1) has a 4-stage pipelining [9]. Qorivva Simple Cookbook [16] has a simple example and further information to use the TB registers. The default system clock frequency for MPC551x processors is 16 MHz and at this operating frequency, it takes around 4.5 minutes that the TBL register overflows to

the TBU register [16]. The tasks that we are dealing with in our work have execution times in the range of several milli seconds. The system clock in our codes for all projects is set to 64 MHz, and at this clock rate it takes around 67.5 seconds for the TBL to overflow to the TBU. This is a rather long time and it would not be required ro take care of the overflowing issue. Thus, we only read the values of the TBL register in our measurements.

4.2 The Basics of Execution Time Measurement Technique

We need to come up with a method to measure the execution times of tasks running on the primary core (Z1) of MPC5510 dual-core devices. Assume the piece of code that we are interested in its execution time, is placed into a function named `Z1_routine()`. The TB registers allow us to count the number of clock cycles before and after making the call to `Z1_routine()`. As the processor's primary core continues running, the number of elapsed clock cycles are stored into the TB registers. The values of the TB registers, before and after making a call to `Z1_routine()`, are stored to measure the execution time. The execution time for `Z1_routine()` can be simlpy calculated using the following relation,

$$\text{execution_time} = \frac{\text{end - start}}{\text{PROCESSOR_CLOCK}} \qquad (4.1)$$

Where we have,

- end: The read value of TB register after making a call to the code snippet
- start: The read value of TB register before making a call to the code snippet
- PROCESSOR_CLOCK: The processor's operating frequency

The default clock for MPC5510 devices is 16 MHz and MPC5517E can be clocked up to 80 MHz [16]. The estimated time is in microsecond since PROCESSOR_CLOCK is set in MHz in our measurements.

As the starting point, we create a single-core project, targeted at MPC5517E, using the CodeWarrior IDE. Figure B.1 in Appendix B shows the code listed in the `main_master.c` file (running on the master core, i.e. Z1) with a simple body for the user-defined `Z1_routine()` to illustrate the basics of our measurement technique.

To enable the TBL register and read its value, some source files from a formerly created project in Arctic Studio [17] have been integrated with the created project in the CodeWarrior IDE. Arctic Studio, presenetd by ARC CORE, is an open source Eclipse-based [19] embedded software package to address AUTOSAR [8] standard. The software

provides a platform that includes real-time operating system, facilities to support communication services like CAN and LIN, memory services and also drivers for a number of microcontrollers used in the automotive ECUs, such as MPC5510 MCUs. The code listing mentioned above includes a header file named Cpu.h. This header file is borrowed from the created project for MPC5517E in Arctic Studio and then is added to the source files of our project. In the code, there is a routine as well as some defined macros in Cpu.h to utilize the TB registers as described below,

- `SPR_TBL_R`, the macro to symbolize the lower TB register to read its contents
- `SPR_HID0`, the macro to symbolize Hardware Implementation Register 0 (spr HID0)
- `get_spr(spr_nr)`, the function to read the contents of special purpose registers

The `initSysclk()` routine makes it possible to change the default operating frequency, i.e. 16 MHz, to our desired clock frequency. All the measurements for both single- and dual-core projects throughout this work are carried out at 64 MHz. `Z1_routine()` declares a volatile 32-bit integer in the SRAM and implements an empty nested loop which means the function performs no useful work. To achieve a relatively accurate measurement, the code fragment of our interest (`Z1_routine()` in our measurements) is called more than once. The `NUMBER_OF_EXECUTIONS` macro specifies how many times the piece of code executes. The project is compiled and linked, and the generated executable binary (elf) is transferred to the EVB using P&E USB Multilink hardware interface. At this stage, we find the value of execution time through watching `execution_time` variable using the existing debugger. For the implemented code, we obtained two execution times: 35937 and 35938 micro seconds, and the larger value is considered as the WCET for `Z1_routine()`. It should be noted that the code runs in the internal Flash memory. If it runs in the SRAM, the execution times, as we will see in Section 4.5, are larger. Freescale provides FreeMASTER [18] which is a powerful free and open source real-time debugger. In the rest of the work, this tool has been applied to facilitate our measurements.

4.3 Measuring Execution Time using Freescale FreeMASTER

Three steps are taken to be able to use FreeMASTER in our application. First, its source files are added to our project code in the CodeWarrior IDE, then one of the eSCI ports, available on the EVB, is initialized, and finally, FreeMASTER user interface is installed on PC to interact with it through a GUI. FreeMASTER has been implemented as an open source communication protocol in C, therefore it is added to the embedded application project as a number of `.c` and `.h` files. The tool consists of two applications: the PC-side application and the embedded-side application. The PC-side application provides a user interface to watch and record the variables in real time and the embedded-side application acts as a server to the PC-side application to fulfill the actual real-time debugging process. The EVB has two serial communication interfaces known as eSCI_A and eSCI_B. The communication between the PC-side and embedded-side applications in our project is conducted through eSCI_A serial port. The routine called `ESCI_A_Init()` is added to the `main_master.c` on the master core, and it is called after clock initialization process to manage the communication between the processor and the GUI through the eSCI port.

To use FreeMASTER in our application, the `main_master.c` file is modified with making calls to the serial communication port (`ESCI_A_Init()`) and FreeMASTER (`FMSTR_Init()`) initialization routines. Figure B.2, in Appendix B, lists the new version of our code to demonstrate the measurement approach using FreeMASTER. The code inside the `main()` function consists of two parts. The first part includes the real-time measurement using a `for` loop and the second part comprises an infinite `while` loop to provide FreeMASTER with an array for off-line display of the recorded variable(s). Here, the basic notion is that we log a variable in real time and then display its value off line.

FreeMASTER GUI lets us create oscillscope and recorder mechanism in our project to inspect and log up to eight variables in an application. To achieve a relatively precise measured value, we execute the code fragment more than once. In the code listing in Appendix B, `Z1_routine()` is called `NUMBER_OF_EXECUTIONS` times and each measured value is stored in an array with the size equal to `NUMBER_OF_EXECUTIONS`. When the first part, dealing with real-time debugging (the `for` loop) terminates, and the measurement process is over, the two FreeMASTER routines associated with the real-time debugging, i.e. `FMSTR_Record()` and `FMSTR_Poll()` are called again infinitely in the second part (the `while` loop). The second part actually supplies the FreeMASTER oscilloscope and recorder tools with their required data. In fact, the endless `while` loop in the recording part enables FreeMASTER client to visualize the measured variables (register values) on screen forever until the system (EVB) is interrupted by user.

FreeMASTER can be used in the interrupt- and poll-driven modes [18]. We use the poll-driven mode in all the carried out measurements in this project. Appendix A describes how to integrate and configure FreeMASTER in the projects that are built using the CodeWarrior IDE.

4.4 The Slowdown Factor

The execution time for a specific piece of code running on the master core, when the slave core is in reset mode, is considered as the baseline. Once the secondary core is started, the execution time of the task running on the primary core is affected, even though the secondary core does not execute any useful work. The slave core's impact on master core is embodied as the execution time of the task running on the master core is stretched.

To come up with a quantitative criterion in order to evaluate the extent at which the execution time is influenced, we define a factor known as the Slowdown Factor (SDF). The factor is described as,

$$\text{Slowdown Factor} = \frac{\text{T2}}{\text{T1}} \qquad (4.2)$$

In which,

T_1: Execution time of task running standalone on the master core (slave core is inactive)

T_2: Execution time of task running on the master core while the slave core is running

In the relation above, the numerator denotes the execution time of some task running on the master core while both primary and secondary cores are operating concurrently, whereas the denominator specifies the measured execution time when the same task runs on the master core as the second core is in reset. To explore the influence of the slave core on the SDF, the master core task is kept constant (Z1 runs the same piece of code in all measurements) and the slave core runs an increasing load, functioning as the background load. The background load is gradually increased and the execution time of the task runnig on the master core is measured. In the CodeWarrior IDE `main()` and `main_p1()` are Z1's and Z0's main functions respectively. The simplest scenario is when the secondary core starts with a bodiless main function, i.e. an empty `main_p1()` routine.

5 The Experimental Results

In this chapter we will explaine how the devised measurement technique, described in Chapter 4, is employed with the aim of conducting a number of experiments to estimate the influence on master core's WCET from shared resources in MPC5517E.

5.1 Exploring the Scenario Leading to the Largest Slowdown Factor

The slowdown factor is greater than or equal to 1.0, and it increases as the background load increases, however there is an upper band for this factor in such a way that the following relation holds,

$$1.0 \leq \text{SDF} \leq \text{Upper bound} \quad (5.1)$$

To estimate the upper band for SDF, a dual-core project in the CodeWarrior IDE for MPC5517E device is created, and three different scenarios, described fully subsequently, will be explored. In the dual-core projects, each core has its own main function placed in a separate source file. As for the CodeWarrior IDE, these files are the `main_master.c` for Z1 and the `main_slave.c` for Z0.

Scenario 1

- Z1 runs an unvarying load
- Z0 defines and manipulates a variable in the SRAM, Z1 does not access this variable

Scenario 2

- Z1 runs an unvarying load and also it defines a variable in the SRAM which is accessible from Z0
- Z0 accesses the variable defined in Z1

Scenario 3

- Z1 runs an unvarying load and also it defines a variable in the SRAM which is accessible from Z0
- Z0 accesses the variable defined in Z1, and also it defines and manipulates a variable that is only accessible to Z0

In these scenarios, Z1 defines a variable in the SRAM and allows Z0 to access it through making calls to global functions defined in Z1. The variable in Z1 is declared as a 32-bit integer using the `int` type and is preceded with the `volatile` keyword to ensure that it is stored in the SRAM. Z0 runs the background load and we will investigate its impact on Z1 by measuring the execution times of the unvarying load running on Z1. The largest

obtained values for execution times are taken into account, since we are interested in the WCET.

Implementation for Scenario 1

The only difference between the codes placed in the Z1's `main_master.c` file in the Section 4.3 and this scenario is that in this scenario Z1's `main()` makes a call to a routine, named `__start_p1()`, to trigger the secondary core (Z0). The created project here is a dual-core project to activate Z0. At startup process, Z0 is in reset and shall be started by Z1. The CodeWarrior stationery embeds `__start_p1()` routine (written in the PowerPC Assembly) with the project, and once this function is called it writes to the `CRP.Z0VEC.R` to initiate the Z0 core.

Z0 declares a variable, which is inaccessible by Z1, and increments it infinitely. The code is placed into a function, named `Z0_routine()`, and the function is called from `main_p1()`.

```
void Z0_routine(void)
{
    volatile int a1 = 0;

    while (1)
    {
        a1++;
    }
}

int main_p1(void)
{
    Z0_routine();
}
```

Implementation for Scenario 2

A variable of integer type along with the two routines responsible for manipulating and reading this variable are added to the code listing of Section 4.3, i.e. to `main_master.c` file. The variable is named `Z1_var_accessible_to_Z0` and is initialized with 0. Z1 also defines two routines to make it possible for Z0 to access and manipulate `Z1_var_accessible_to_Z0`. The calls are made within the Z0's main function in an infinite loop. These two routines are,

`increment_Z1_var_accessible_to_Z0()`

`read_Z1_var_accessible_to_Z0()`.

Following, are the contents of the `main_master.c` file (running on Z1) to implement this scenario.

```
#include "MPC5517E.h"
#include "Cpu.h"
#include "freemaster.h"

#define PROCESSOR_CLOCK 64 /* Core operating frequency in MHz */
#define NUMBER_OF_EXECUTIONS 50
#define Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS 100000
#define Z1_NUMBER_OF_INNER_LOOP_ITERATIONS 1

/* Prototype for PRC1 startup */
extern void __start_p1();

/* Measurement variables */
int enable, start, end;

/* FreeMASTER variables */
float execution_time = 0, samples[NUMBER_OF_EXECUTIONS] = {0};
int sampling_index = 0;

/* The shared variable between the two cores */
volatile int Z1_var_accessible_to_Z0 = 0;

void ESCI_A_Init(void)
{
   ESCI_A.CR2.R = 0x2000;
   ESCI_A.CR1.B.TE = 1;
   ESCI_A.CR1.B.RE = 1;
   ESCI_A.CR1.B.PT = 0;
   ESCI_A.CR1.B.PE = 0;
   ESCI_A.CR1.B.SBR = 34; /* Baud rate = 115200 */
}

void initSysclk(void)
{
   /* Initialize PLL and sysclk to 64 MHz */
   FMPLL.ESYNCR2.R = 0x00000007;
   FMPLL.ESYNCR1.R = 0xF0000020;
   CRP.CLKSRC.B.XOSCEN = 1;
   while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
   FMPLL.ESYNCR2.R = 0x00000005;
   SIU.SYSCLK.B.SYSCLKSEL = 2;
}
void Z1_routine(void)
{
   volatile int i, j;
   for (i = 0; i < Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS; i++)
   {
       for (j = 0; j < Z1_NUMBER_OF_INNER_LOOP_ITERATIONS;
             j++)
       {
       }
   }
```

```
}

void increment_Z1_var_accessible_to_Z0(void)
{
   Z1_var_accessible_to_Z0++;
}

int read_Z1_var_accessible_to_Z0(void)
{
   return Z1_var_accessible_to_Z0;
}

int main(void)
{
   initSysclk();
   ESCI_A_Init();

   /* Initialize FreeMASTER driver */
   FMSTR_Init();

   /* Start the other core by writing CRP.Z0VEC.R */
   CRP.Z0VEC.R = (unsigned long)__start_p1;

   /* Enable Time Base register (TBE) */
   enable = get_spr(SPR_HID0);
   end = 0x4000;
   enable = enable | end ; /* according to the manual */
   set_spr(SPR_HID0, enable);

   for(sampling_index=0; sampling_index < UMBER_OF_EXECUTIONS;
       sampling_index++)
   {
      start = 0;
      end = 0;

      /* Start measuring execution time */
      start = get_spr(SPR_TBL_R);
      Z1_routine();

      /* Stop measuring execution time */
      end = get_spr(SPR_TBL_R);

      /* Measure execution time using the processor clock */
      execution_time=(float)((end - start) / PROCESSOR_CLOCK);

      /* Store the measured times in an array being used by
         FreeMASTER Recorder Mode */
      samples[sampling_index] = execution_time;
      FMSTR_Recorder();
      FMSTR_Poll();
   }

   /* Record the samples using FreeMASTER Recorder Mode */
```

```
    sampling_index = 0;
    while(1)
    {
       FMSTR_Recorder();
       FMSTR_Poll();
       execution_time = samples[sampling_index];
       sampling_index++;

        if(sampling_index >= NUMBER_OF_EXECUTIONS)
           sampling_index = 0;
    }
}
```

The code placed in the `main_slave.c` file (running on Z0) is,

```
extern int read_Z1_var_accessible_to_Z0(void);
extern void increment_Z1_var_accessible_to_Z0(void);

int main_p1(void)
{
    while(1)
    {
       increment_Z1_var_accessible_to_Z0();
       read_Z1_var_accessible_to_Z0();
    }
}
```

Implementation for Scenario 3

In the implementation for Scenario 3, the code executed by Z1 is exactly the same code that was developed for the Scenario 2. The Z0 core increments and reads a variable which is defined in Z1 and is accessible to this core. Z0 also manipulates a variable which is local to it and is not accessible from Z1. The contents of `main_slave.c` file for this scenario are,

```
extern int read_Z1_var_accessible_to_Z0(void);
extern void increment_Z1_var_accessible_to_Z0(void);

void Z0_routine(void)
{
     volatile int a1 = 0;
     while (1)
     {
         increment_Z1_var_accessible_to_Z0();
         read_Z1_var_accessible_to_Z0();
         a1++;
     }
}


int main_p1(void)
{
```

```
    Z0_routine();
}
```

To make sure that FreeMASTER does not inflict any overhead on the code fragment running on Z1, the measurements are performed with and without FreeMASTER for a very simple occasion. Z0 calls an empty `main_p1()` function. FreeMASTER runs in the poll-driven mode and Z1 runs `Z1_routine()` with the following body,

```
void Z1_routine(void)
{
   volatile int i;
   for (i = 0; i < Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS; i++)
   {
       for (j = 0; j < Z1_NUMBER_OF_INNER_LOOP_ITERATIONS;
            j++)
       {
       }
   }
}
```

Table 5.1 shows that both measured values using the native debugger of Code Warrior Development Studio and FreeMASTER tool are exactly the same. The outcome of this simple test is very substantial as it made us confident that FreeMASTER does not cause any overhead on the execution times after it is hooked to the program. Table 5.2 presents the results of measurements for the three surveyed scenarios. To calculate the SDF for each case, we need a baseline. The baseline is defined as the execution time for `Z1_routine()` when the slave core has not started yet. As it is obviously evident in Table 5.2, the largest SDF occurs in the Scenario 1, where the slave core infinitely increments an integer declared in the shared SRAM. Making calls from the slave core to a variable which has been explicitly shared between the two cores (scenarios 2 and 3) does not necessarily yield the largest execution time.

No.	FreeMASTER is used	Execution Time (µs)
1	No	35938
2	Yes	35938

Table 5.1: Examining the FreeMASTER impact on measurements

Concerning the scenarios 2 and 3, the inter-core shared variable has been declared as a 32-bit scalar integer. This scalar variable was also declared as a static array with large sizes, e.g., 10000 (`volatile int Z1_var_accessible_to_Z0[10000] = {0}`), but we observed no significant effect on the SDF.

We focus on Scenario 1 to achieve a setting that can potentially lead to the possible largest execution time and therefore to the largest SDF. The background load running on Z0 is gradually increased by declaring more private variables in `Z0_routine()` to increase the rate at which Z0 accesses the shared memory space. First, `a2` is added, the measurement is done and the SDF is calculated using the baseline. Then, `a3` is added, the measurement procedure is repeated and so on. The code fragment running as the reference and the unvarying load on Z1 are the same `Z1_routine()` that we had before. We found it out that the largest SDF is gained for the arranged setting with nine defined variables in Z0.

Scenario	Measured Execution Time (µs)	SDF
Baseline	35937	1
1	38542	1.0725
2	37022	1.0302
3	37500	1.0435

Table 5.2: The Measurements for the surveyed scenarios to find the largest SDF

We are extending the Scenario 1, therefore these variables are not accessible from the master core. The body of `Z0_routine()` is,

```
void Z0_routine(void)
{
    volatile int a1=0, a2=0, a3=0, a4=0, a5=0, a6=0, a7=0,
                 a8=0, a9=0, a10=0;
    while (1)
    {
        a1++;
        a2++;
        a3++;
        a4++;
        a5++;
```

```
        a6++;
        a7++;
        a8++;
        a9++;
        a10++;
    }
}
```

Table 5.3 lists the obtained results for a number of declared variables in `Z0_routine()`. We start with one variable and increase the number of variables, one by one, to explore the influence of the increasing load on the execution time of the task running on Z1. The measurements are carried out for both the internal flash and RAM as the targets of deployed executable binary. The largest SDF when the code runs in the internal flash is 1.1033 and it occurs when 9 variables are declared in `Z0_routine()`.The maximum SDF while the code runs in the RAM is 1.3187 with 3 declared variables in `Z0_routine()`. The Largest figures for the SDF values (marked in red) reveal that the influence of the slave core on the execution time of the task running concurrently on the master core can be significant. In the explained scenario, when the task runs in RAM, there is a relatively noticeable impact on the WCET. The SDF is dependent on the pattern of accessing the shared memory and also the location of the running code, i.e. whether the code runs in internal flash or RAM. This implies that the execution times have to be measured for both cases: when the code runs in internal flash and when the code runs in RAM. This point should be taken into deliberate account by the embedded application designer at the design and verification phases of critical and safety critical applications. The reason is that the designer has to specify which target for application's code (internal flash or RAM) results in the largest SDF, and hence more impact on the cores.

5.2 The Observed Bugs in CodeWarrior Development Studio

During carrying out the experiments, two bugs, one in the compiler and one in the IDE, were found in the CodeWarrior Development Studio tool. The glitches were reported to the Freescale support team; they confirmed them and provided us with the solution to resolve them.

The first bug was a fault in the compiler, and it dealt with the dual-core projects when the target device was chosen as MPC5517E. For this processor, the code did not execute on the secondary core (Z0) when a function call was made in the Z0's main routine, i.e. `main_p1()`, as if the Z0 core was still in reset. However, if the project was created for MPC5516E as the target device, there would be no problem with

Target for Executable Binary	Internal Flash		RAM	
Number of the Declared Variables in Z0	**Execution Time (µs)**	**SDF**	**Execution Time (µs)**	**SDF**
Z0 does not start (baseline)	35938	1	50000	1
1	38542	1.0725	64583	1.2917
2	38566	1.0731	65104	1.3021
3	38945	1.0837	65937	**1.3187**
4	39063	1.0870	65024	1.3005
5	39063	1.0870	64844	1.2969
6	39313	1.0939	65533	1.3107
7	39353	1.0950	65278	1.3057
8	39535	1.1000	65625	1.3125
9	39649	**1.1033**	65625	1.3125
10	39489	1.0988	65625	1.3125

Table 5.3: The measurement scenarios to approximate the largest SDF

making calls to functions from the Z0's main routine. We contacted the Freescale support team, and they confirmed that the issue was certainly a bug in their tool. It was fixed in the version 2.1 of the compiler.

The second bug was related to the generated code by the IDE for dual-core projects. We started with a very simple scenario in which the Z1 core ran only a finite `for` loop and the Z0 core ran only a bodiless main function that does not return. In the beginning, to determine the baseline, the slave core was not enabled and the execution time of the finite `for` loop was measured as $T_{1.}$ Then, the slave core was enabled and the execution time was measured as $T_{2.}$ It is evident that T_2 must be greater than, or at its best, equal to T_1. But, amazingly enough, we observed that T_2 was less than $T_{1,}$ as it would be there is some kind of unexpected speedup, opposed to facing an anticipated slowdown. The support team explained that the reason behind this issue is the pre-fetching feature. By default, the pre-fetching is enabled by the CodeWarrior IDE stationery, despite the fact that it should be disabled according to [22]. At the startup process, there are some initializing macros that affect the device behavior through writing specific values to different registers and memory locations of the processor. One of the macros is `FLASH_DATA`, defined in `__pc_eabi_init.c`. This macro is written to the PFCRP0 register during the startup process and enables pre-fetching. To disable pre-fetching, the assigned value for this macro must be changed from `0x00016B55` to `0x00006B05`. At the time of writing this report, the bug has not been fixed in the latest CodeWarrior IDE (version 5.9.0). To avoid the issue, the set value for the macro explained above needs to be manually changed to the right value to disable pre-fetching.

6 Conclusions and Future Work

6.1 Conclusions

The model of a real-time task (Figure 2.1) points out four static parameters: offset, deadline, period and execution time. Among these parameters, the execution time is the one that requires a demanding effort to determine, since it is contingent upon a number of influencing factors. These factors are organization of program code, hardware and software properties of system, initial system state and system events. It is an intricate process to specify the exact time that it takes for a piece of code to execute on a specific platform due to the mentioned affecting factors. Hence, it suffices to estimate the execution time of a real-time task instead of dealing with its exact value. Regarding sceduling algoritms for real-time systems, we always deal with WCET which is defined as the probable uninterrupted execution time for one iteration of the task.

In this thesis, we presented a measurement method to estimate the WCET for a task running on the master core of a dual-core processor. Then, we investigated the impact of the slave core on the execution time of tasks runing on the master core, using our proposed measurement technique. We defined a term known as SDF to characterize the influence of the second core on the execution times. Depending on the codes running on each core, this influence can be conspicuous and can increase the task execution time. Consequently, system designer should reflect on the WCET for the dual-core projects with respect to this slowing down factor, especially for critical and safety critical functionalities in real-time systems. One interesting finding in this project is the need for estimating the largest SDF for all memory types (such as internal flash, external flash, on-chip SRAM, off-chip SRAM) that are supposed to host a real-time application. This stipulates that the execution time of a real-time application should be measured (estimated) for all available type of memories that are supposed to host executable binaries, and that would be wrong to apply the measured execution time for one specific type of memory to the other memory types.

6.2 Future Work

As for all the embedded projects that were implemented for the explained scenarios in this thesis work, the executable binaries run either in the internal flash or in the on-chip SRAM. However, the explored processor can also support external flash and off-chip SRAM, and one further work can be evaluating the SDF when the codes run in the external flash or off-chip RAM. Moreover, it will be very impressive to do some analytical analysis to explain the different SDF values, for example, based on the number of references to shared resources per time units. Of course this is not something trivial to achieve.

Another future work can be centered around AUTOSAR to develop a tool to automate the process of extracting all the required ports data of a Software Components (SWCs) [8] out of their XML specifications, in order to have an automatic execution estimation technique for SWCs runnables. Currently, to estimate the WCET or investigate how the primary core is affected by the secondary core, we have to manually take out the ports information and supply SWC's input ports with their allowed values that are specified in their XML specification files. A tool can be developed to automate the entire process.

Appendices

A Integrating FreeMASTER with CodeWarrior

FreeMASTER is a powerful real-time debugger presented as a free and open source tool by Freescale Semiconductor. The tool involves PC-side and embedded-side applications, where the PC-side provides a GUI to visualize and record variables in real time and the embedded-side is a server to provide the PC-side application with real-time debugging capability. To use the tool it is required to pursue three steps,

- integrate its source files with an embedded application project
- initialize a communication interface to communicate with the tool
- install its user interface on PC

The embedded-side application is actually an open source communication protocol developed as a number of `.c` and `.h` files in C that are added to an exixting embedded application project to provide real-time debugging feature.

The PC-side application has an HTML-based GUI to graphically watch the variables of our interest on PC side. The GUI is installed on Microsoft Windows operating systems, and allows the user to create oscilloscope or recorder in their FreeMASTER project and monitor or record the variables in real time.

The transferred executable binary running on the target hardware is capable of communicating with the PC-side application through a communication interface. In our measurements, we use the serial communication port available on the EVB to interact with the embedded-side FreeMASTER. The EVB has two eSCI ports: eSCI_A and eSCI_B. It is required to apply the correct jumper settings on the EVB to utilize the communication interfaces. For the proper jumper settings one should consult the EVB reference manual [13].

At 64 MHz, the default baud rate is 9600 (`ESCI_A.CR1.B.SBR = 417`). With this default value, the communication between the PC- and embedded-side applications was not handled properly and it was impossible to watch the variables designated in the FreeMASTER GUI. We found it out that the baud rate should be set as to the highest value as possible to avoid facing issues in the communication. When the processor runs at 64 MHz, 115000 is a suitable value for the baud rate. To achieve this baud rate the corresponding register is initialized with the correct matching value

(ESCI_A.CR1.B.SBR = 34) . The following routine initializes the communication between the EVB and the PC via eSCI_A serial port.

```
void ESCI_A_Init(void)
{
    ESCI_A.CR2.R = 0x2000;
    ESCI_A.CR1.B.TE = 1;
    ESCI_A.CR1.B.RE = 1;
    ESCI_A.CR1.B.PT = 0;
    ESCI_A.CR1.B.PE = 0;
    ESCI_A.CR1.B.SBR = 34; /* Baud rate = 115200 */
}
```

The above routine as well as the FreeMASTER source files are added to the project to provide our application with the real-time debugging capability. To come up with a tidy source code, the FreeMASTER files are placed in a folder with this name and a group is created with the same name in the CodeWarrior IDE to organize the related files under this group. Figure A.1 displays the project tree. To start FreeMASTER in the application, `FMSTR_Init()` routine is called and it falls to `FMSTR_Record()` and `FMSTR_Poll()` routines to provide real-time montitoring and recording features. To be able to make calls to the FreeMASTER functions, `freemaster.h` is the only header file needed to be included in the `main_master.c` file. In brife, the routines in below are added to the `main_master.c` file,

- `ESCI_A_Init()`, to initilize communication through the eSCI_A port
- `FMSTR_Init()`, to start embedded-side application
- `FMSTR_Record()` and `FMSTR_Poll()`, to implement real-time debugging

The serial communication driver file, `freemaster_cfg.h` defines the macros to set different configurations in the embedded-side application. One important configuration is

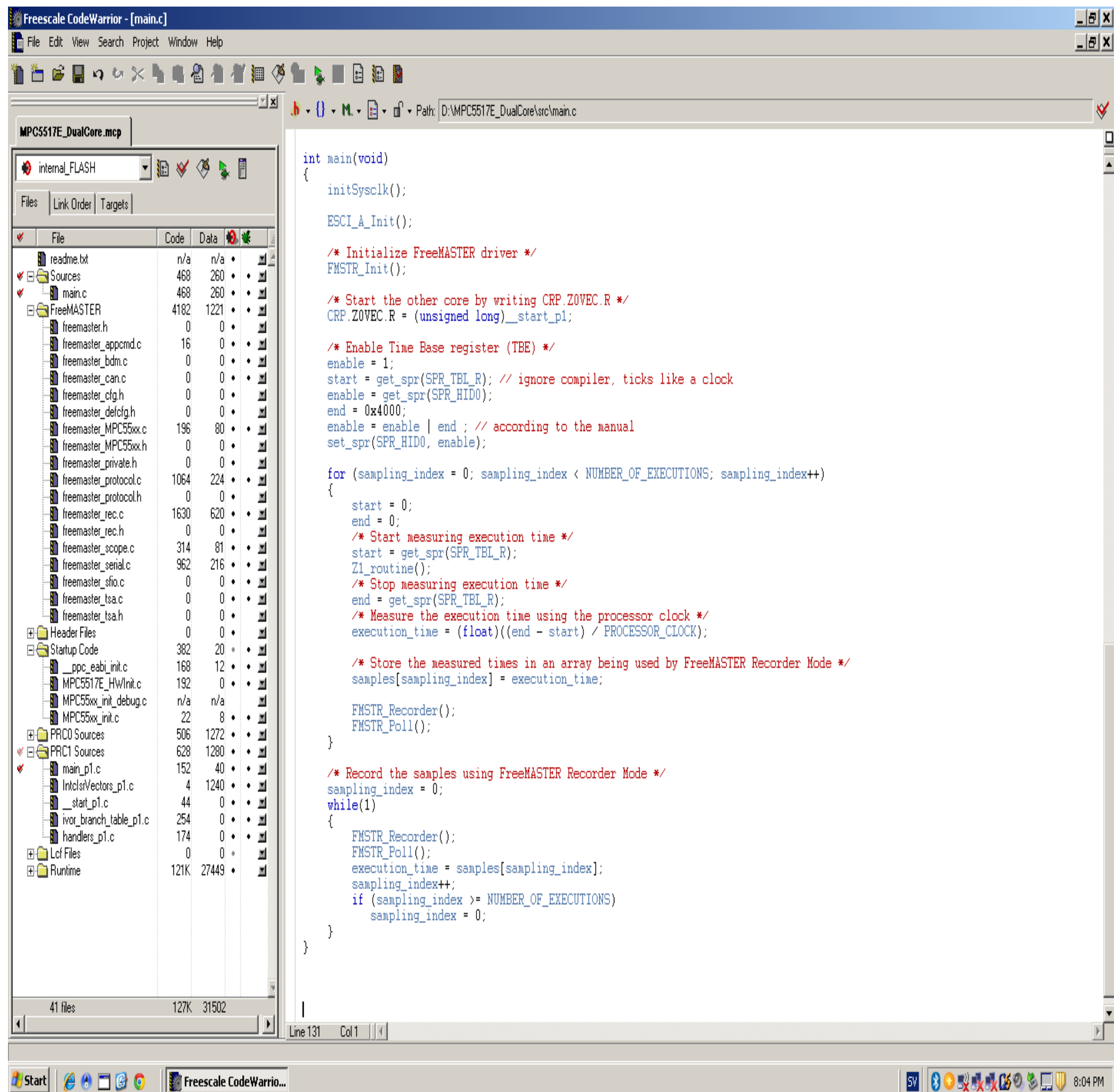

Figure A.1: CodeWarrior IDE integrated with embedded-side FreeMASTER

the operating mode. There are three modes available,

- short interrupt
- long interrupt
- poll-driven

The following variables defined in `freemaster_cfg.h` should be set properly to determine the operating mode,

```
FMSTR_LONG_INTR 0  /* complete message processing in interrupt */
FMSTR_SHORT_INTR 0 /* SCI FIFO-queuing done in interrupt */
FMSTR_POLL_DRIVEN 1 /* no interrupt needed, polling only */
```

To determine the operating mode, the corresponding macro is set to 1 and the other two are set to 0. We use the poll-driven mode so the corresponding variable is set to 1 and the other two variables related to interrupt-driven modes are set to 0. Another important macro is FMSTR_REC_BUFF_SIZE that specifies the built-in buffer size in the memory space of the target hardware. This macro determines the size of buffer in bytes and is set to an appropriate value to be able to record the desired number of samples through the recorder interface created in the PC-side GUI.

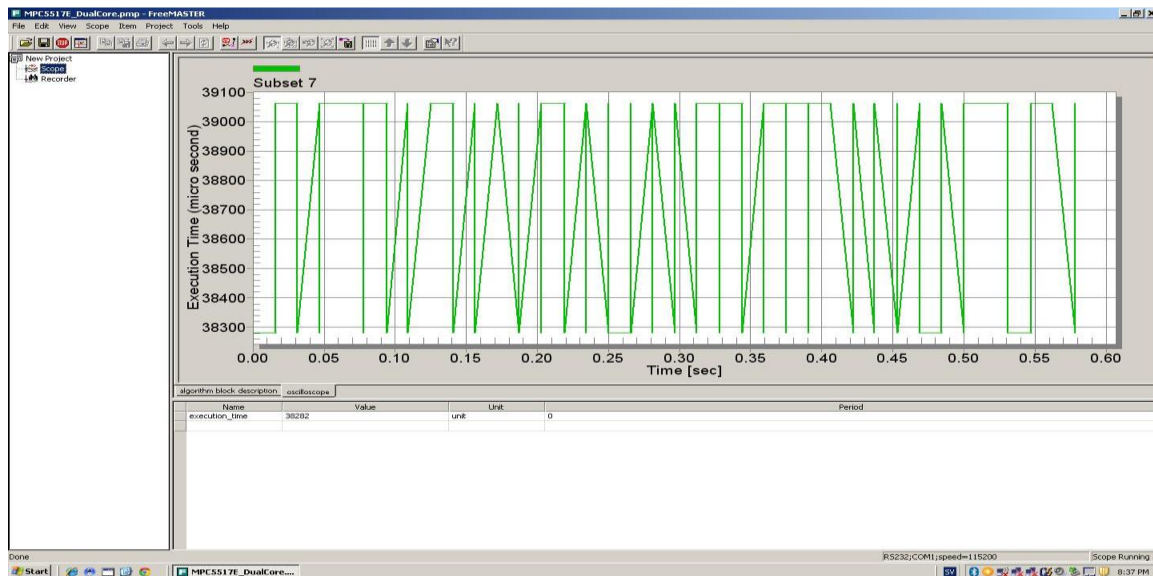

Figure A.2: Real-time display using PC-side FreeMASTER

The GUI running on PC allows watching up to eight variables and also creating oscilloscope and recorder to observe the variations of watched variables graphically. Besides, it is capable of storing the watched data on file, manually or automatically. Figure A.2 shows start, end and execution_time variables in the debugging pane for the code listed in Chapter 4, Section 4.5 and plots the variations of execution_time using the oscilloscope application. Figure A.3 displays execution_time for 50 recorded samples.

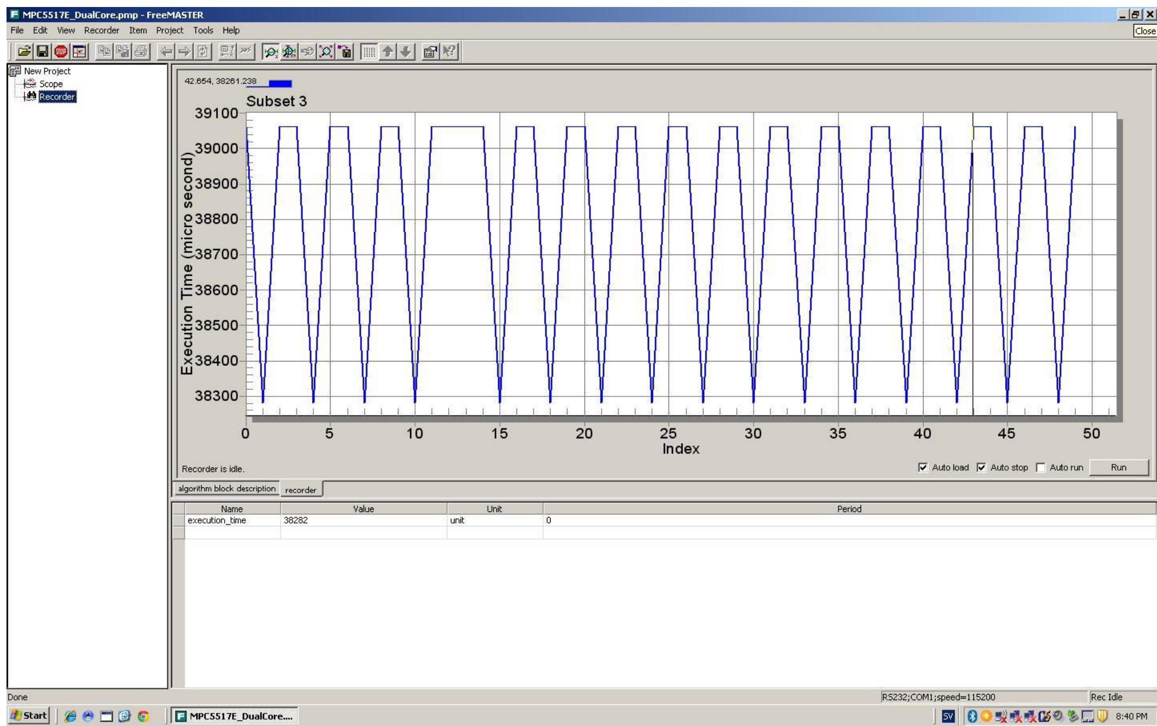

Figure A.3: Displaying a recorded variable using PC-side FreeMASTER

B Listing of Codes

```
#include "MPC5517E.h"
#include "Cpu.h"

#define PROCESSOR_CLOCK 64 /* Core operating frequency in MHz */
#define NUMBER_OF_EXECUTIONS 50
#define Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS 100000
#define Z1_NUMBER_OF_INNER_LOOP_ITERATIONS 1

/* Measurement variables */
int enable, start, end;
float execution_time = 0;

void initSysclk(void)
{
   /* Initialize PLL and sysclk to 64 MHz */
   FMPLL.ESYNCR2.R = 0x00000007;
   FMPLL.ESYNCR1.R = 0xF0000020;
   CRP.CLKSRC.B.XOSCEN = 1;
   while (FMPLL.SYNSR.B.LOCK != 1){}; /*Wait for PLL to LOCK*/
   FMPLL.ESYNCR2.R = 0x00000005;
   SIU.SYSCLK.B.SYSCLKSEL = 2;
}

void Z1_routine(void)
{
   volatile int i, j;

   for (i = 0; i < Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS; i++)
   {
       for (j = 0;j < Z1_NUMBER_OF_INNER_LOOP_ITERATIONS; j++)
       {
       }
   }
}

int main(void)
{
   volatile int i;

   initSysclk();

   /* Enable Time Base register (TBE) */
   enable = get_spr(SPR_HID0);
   end = 0x4000;
   enable = enable | end ; /* according to the manual */
   set_spr(SPR_HID0, enable);
   for (i = 0; i < NUMBER_OF_EXECUTIONS; i++)
   {
       start = 0;
```

```
        end = 0;

        /* Start measuring execution time */
        start = get_spr(SPR_TBL_R);
        Z1_routine();
        /* Stop measuring execution time */
        end = get_spr(SPR_TBL_R);

        /* Measure execution time using the processor clock */
        execution_time = (float)((end-start)/PROCESSOR_CLOCK);
    }
}
```

Figure B.1: Code listing for the basics of execution time measurement technique (Section 4.2)

```
#include "MPC5517E.h"
#include "Cpu.h"
#include "freemaster.h"
#define PROCESSOR_CLOCK 64 /* Core operating frequency in MHz */
#define NUMBER_OF_EXECUTIONS 50
#define Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS 100000
#define Z1_NUMBER_OF_INNER_LOOP_ITERATIONS 1

/* Measurement variables */
int enable, start, end;

/* FreeMASTER variables */
float execution_time = 0, samples[NUMBER_OF_EXECUTIONS] = {0};
int sampling_index = 0;

void ESCI_A_Init(void)
{
   ESCI_A.CR2.R = 0x2000;
   ESCI_A.CR1.B.TE = 1;
   ESCI_A.CR1.B.RE = 1;
   ESCI_A.CR1.B.PT = 0;
   ESCI_A.CR1.B.PE = 0;
   ESCI_A.CR1.B.SBR = 34; /* Baud rate = 115200 */
}

void initSysclk(void)
{
   /* Initialize PLL and sysclk to 64 MHz */
   FMPLL.ESYNCR2.R = 0x00000007;
   FMPLL.ESYNCR1.R = 0xF0000020;
   CRP.CLKSRC.B.XOSCEN = 1;
   while (FMPLL.SYNSR.B.LOCK != 1){};/*Wait for PLL to LOCK */
   FMPLL.ESYNCR2.R = 0x00000005;
   SIU.SYSCLK.B.SYSCLKSEL = 2;
}

void Z1_routine(void)
{
```

```
   volatile int i, j;

   for (i = 0; i < Z1_NUMBER_OF_OUTER_LOOP_ITERATIONS; i++)
   {
       for (j = 0;j < Z1_NUMBER_OF_INNER_LOOP_ITERATIONS; j++)
       {
       }
   }
}

int main(void)
{
   initSysclk();
   ESCI_A_Init();

   /* Initialize FreeMASTER driver */
   FMSTR_Init();

   /* Enable Time Base register (TBE) */
   enable = get_spr(SPR_HID0);
   end = 0x4000;
   enable = enable | end ; /* according to the manual */
   set_spr(SPR_HID0, enable);

   for(sampling_index=0; sampling_index <NUMBER_OF_EXECUTIONS;
        sampling_index++)
   {
      start = 0;
      end = 0;

      /* Start measuring execution time */
      start = get_spr(SPR_TBL_R);
      Z1_routine();
      /* Stop measuring execution time */
      end = get_spr(SPR_TBL_R);

      /* Measure execution time using the processor clock */
      execution_time = (float)((end-start)/PROCESSOR_CLOCK);

      /* Store the measured times in an array being used by
        FreeMASTER Recorder Mode */
      samples[sampling_index] = execution_time;

      FMSTR_Recorder();
      FMSTR_Poll();
   }

   /* Record the samples using FreeMASTER Recorder Mode */
   sampling_index = 0;
   while(1)
   {
      FMSTR_Recorder();
      FMSTR_Poll();
```

```
        execution_time = samples[sampling_index];
        sampling_index++;

        if(sampling_index >= NUMBER_OF_EXECUTIONS)
            sampling_index = 0;
    }
}
```

Figure B.2: Code listing to measure execution time using Freescale FreeMASTER (Section 4.3)

C List of Abbreviations

ABS	Anti-lock braking system
AUTOSAR	AUTOmotive Open System ARchitecture
BCET	Best-case execution time
CAN	Controller Area Network
DM	Deadline Monotonic
ECU	Electronic Control Unit
EDF	Earliest Deadline First
eSCI	enhanced Serial Communication Interface
EVB	Evaluation Board
GUI	Graphical User Interface
IDE	Integrated Development Environment
LIN	Local Interconnect Network
MCU	Microcontroller Unit
MMU	Memory Management Unit
OEM	Originally Equipment Manufacturer
RM	Rate Monotonic
RTE	Runtime Environment
SRAM	Static Random Access Memory
SWC	Software Component
TB	Time Base
TBL	Time Base Lower
TBU	Time Base Upper
VLE	Variable Length Encoding
WCET	Worst-case execution time

References

[1] J. Stankovic. *Misconceptions of Real-Time Computing*, 1988.

[2] J. Jonsson. *Real-Time Systems (EDA222)*. Master's level 7.5 ECTS course, Lectures slides, Chalmers University of Technology, 2012, http://www.cse.chalmers.se/edu/course/EDA222/.

[3] A. Bets. *Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs*. PhD thesis, University of York, 2010.

[4] J. Engblom, A. Ermedhal, M. Sjödin, J. Gustafsson, H. Hansson. *Worst-case Execution time Analysis for Embedded Real-Time Systems*.

[5] *MPC5510 Microcontroller Family Data Sheet*. Rev. 3, Freescale Semiconductor, 2009.

[6] R. Wilhelm et al. *The worst-case execution-time Problem—Overview of Methods and Survey of Tools*, ACM Transactions on Embedded Computing Systems, Vol. 7, Issue 3, Article No. 36, 2008.

[7] M. Paolieri et al. *Hardware Support for WCET Analysis of Hard real-Time Multicore Systems*, ACM SIGARCH Computer Architecture News, Vol. 37, Issue 3, 55-68, 2009.

[8] *AUTOSAR*, www.autosar.org.

[9] *e200 core family*: Freescale power Architecture™ IP, Product Brochure, IPextreme, 2007.

[10] *Variable-Length Encoding (VLE) Programming Environments Manual*: A Supplement to the EREF, VLEPEM, Rev. 0, Freescale Semiconductor, 2007.

[11] *Designing Code for the MPC5510 Z0 Core*, Document Number: AN3614 Rev. 0, Freescale Semiconductor, 2008.

[12] *Basic Multicore Initialization For the MPC5516G/E and MPC5514G/E Devices*, Document Number: AN3627 Rev. 0, Freescale Semiconductor, 2008.

[13] *MPC5510EVB User Manual*, Revision 1.0, Freescale Semiconductor, 2007.

[14] R.Kirner, P. Puschner, I. Wenzel. *Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation*, Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05). Seatlle, Washington, 7–10, 2005.

[15] P. Puschner. *Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures*. Proc. *2nd Euromicro International Workshop on WCET Analysis*, Technical Report, York YO10 5DD,
United Kingdom, Department of Computer Science, University of York, 2002.
[16] *Qorivva Simple Cookbook "Hello World" Programs to Exercise Common Features on MPC5500 & MPC5600 Microcontrollers*, Document Number: AN2865, Rev. 4, Freescale Semiconductor, 2010.
[17] *Arctic Studio*, www.arccore.com.
[18] *FreeMASTER for Embedded Applications, User Guide,* FMSTERUG Rev 2.1, Freescale Semiconductor, 2011.
[19] *Eclipse*, www.eclipse.org.
[20] *MPC5510 Microcontroller Family Reference Manual*, Document Number: MPC5510RM, Rev. 1, Freescale Semiconductor, 2008.
[21] *Introducing AUTOSAR ,* presentation slides, Volvo Group Trucks, 2008.
[22] *Mask Set Errata for Mask REVA*, MPC5510ACE, Rev. 08, Freescale Semiconductor, 2011.