Hardware-dependent Software
Principles and Practice

Wolfgang Ecker · Wolfgang Müller ·
Rainer Dömer
Editors

Hardware-dependent Software
Principles and Practice

Springer

Wolfgang Ecker
Infineon Technologies AG
Munich, Germany
E-mail: wolfgang.ecker@infineon.com

Rainer Dömer
University of California, Irvine
Henry Samueli School of Engineering
Irvine, CA, USA
E-mail: doemer@uci.edu

Wolfgang Müller
Universität Paderborn
C-LAB
Paderborn, Germany
E-mail: wolfgang@acm.org

ISBN 978-1-4020-9435-4 e-ISBN 978-1-4020-9436-1

Library of Congress Control Number: 2008938549

Cover design: Wolfgang Müller and Christof Poth

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

As the editors spend considerable time for the preparation of this book, they would like to thank their families for their support.

Wolfgang Ecker dedicates this book to his wife Monika and children Johannes, Stephanie, and Matthias.

Wolfgang Müller dedicates this book to his wife Barbara and children Philipp, Maximilian, and Tabea.

Rainer Dömer dedicates this book to his wife Julia and children Sophie, Klara, and Simon.

Contents

Preface

Hardware-dependent Software (HdS) plays a key role in desktop computers and servers for many years. Recently, the relevance of HdS in the domains of embedded systems and Systems-on-Chip (SoCs) has significantly increased, mainly due to its flexibility, the possibility of late change, and the quick adaptability.

Modern SoCs, on a single die integrated embedded systems, often contain multiple programmable cores, including general purpose processors, digital signal processors (DSPs), and/or application specific instruction set processors (ASIPs) requiring a large amount of low level software. Mobile phones and automotive control systems meanwhile come with complex boot loaders and include multiple communication protocol stacks of considerable size. Here and in many other application areas, the number and complexity of standards that need to be supported have steadily grown. For mobile phones, for instance, the set of currently expected standards includes GSM, GPRS, EDGE, UMTS, Bluetooth, TCP/IP, and IrDA, to only name a few.

In this context, HdS has become a crucial factor in embedded system design since it allows to accommodate and adapt late changes in the hardware platform as well as in the application software. Thus, even last minute changes can be quickly performed. On the other hand, changes in the HdS are often hard to track and can have a complex impact on the system with a potential for total system failure. HdS also critically influences the system performance and power management. Consequently, HdS must be carefully designed and maintained.

In contrast to its importance in the area of electronic systems design, the role of HdS is most often underestimated. Considering todays literature, we can only find very few introductory and application-oriented text books. To overcome this gap, we have brought together experts from different HdS areas in this book. By providing a comprehensive overview of general HdS principles, tools, and applications, we feel that this book provides adequate insight into the current technology and upcoming developments in the domain of HdS. The reader will find a text book with self-contained introductions to the principles of Real-Time Operating Systems (RTOS), the emerging BIOS successor

UEFI, and the Hardware Abstraction Layer (HAL). Further chapters cover industrial applications, verification, and tool environments.

This book would not have been possible without the help and contributions of many people. First of all, we would like to thank Mark de Jongh and Cindy Zitter from Springer Verlag who supported us throughout the publication process. We also thank the contributing authors for their great cooperation through the entire process. For the review of individual chapters and valuable comments and suggestions, we acknowledge the help of Stephen A. Edwards (Columbia University), Petru Eles (Linköpings Universitet), Andreas Gerstlauer (University of Texas, Austin), Grant Martin (Tensilica Inc.), and Graziano Pravadelli (Universita di Verona). Finally, we thank Christof Poth who provided us with the sparkling picture for our book cover.

Wolfgang Ecker
Infineon Technologies AG, Munich, Germany

Wolfgang Müller
Paderborn University, Paderborn, Germany

Rainer Dömer
University of California, Irvine, USA

Chapter 1

HARDWARE-DEPENDENT SOFTWARE

Introduction and Overview

Wolfgang Ecker, Wolfgang Müller and Rainer Dömer

Abstract Rapidly rising system complexity has created a growing productivity gap in the design of electronic systems. One critical component is Hardware-dependent Software (HdS), the importance of which is often underestimated. In this chapter, we introduce HdS and illustrate its role in the overall system design context. We also provide a brief overview and define a basic HdS terminology and conclude with a brief outlook over the following chapters in this book.

Keywords: Hardware-dependent Software, Systems Complexity, Productivity Gap

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

1.1 Increasing Complexity

Microelectronics are an integral part of our daily life. Electronic devices affect, support, and enable us in countless tasks and areas, including communication, transportation, work, and entertainment. Automotive applications are a classic example. A modern car or truck is filled with multiple dozens of electronic control units (ECUs) consisting of micro controllers, digital signal processors (DSPs) and application specific integrated circuits (ASICs). The control of mechanical machinery by use of electronic devices does not even stop at bicycles. For example, Shimano Inc. has recently introduced its Di2 electronic gear shifting option for high-end bicycles [Shi08].

In general, the number of new electronic devices in use is steadily increasing. For instance, the annual sales volumes of classic electronic mass products, such as personal computers (PCs) and mobile phones, are still inclining. In 2007, worldwide 271.2 million units of PCs were shipped and mobile phone sales surpassed 1.15 billion in the same year (Gartner 2007).

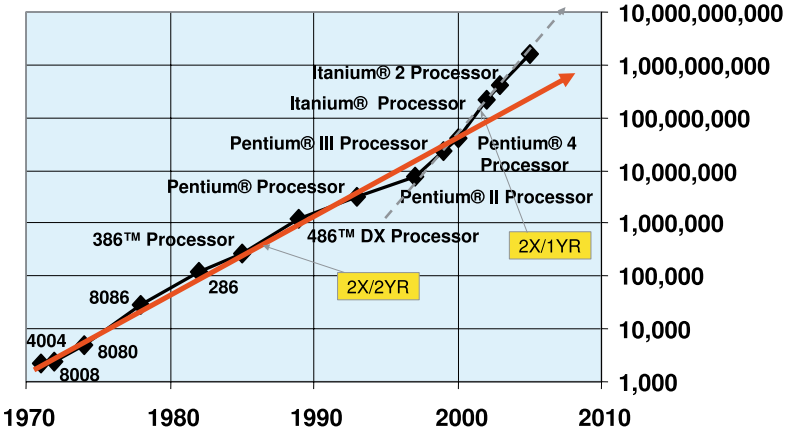

Figure 1.1. Increasing CPU complexity over time. [Source: Intel® Corp.]

At the same time, the complexity of electronic systems is growing exponentially. Moore's law still holds true, the number of transistors that can be integrated on a single chip still doubles every 18 months [Moo65]. Clearly, this development is driven by increasing requirements for larger memory sizes and higher processing speeds. Consequently, general purpose PCs and servers have reached 64-bit multicore architectures and integrate a large set of dedicated components. Figure 1.1 shows this complexity growth for the x86 family of Intel processors.

On the other hand, for many special purpose embedded applications, entire systems can be integrated on a single die, resulting in complex System-on-Chip (SoC) architectures. Such dedicated SoCs typically include several programmable cores such as general processors, DSPs, and application specific instruction set processors (ASIPs). While traditionally most embedded systems were based on 8-bit processors, simple reduced instruction set computers (RISC), or MIPS-based micro control units (MCUs), the 32-bit market today has surpassed the 8-bit market in 2007 (iSuppli 2008).

1.1.1 Growing Software Content

Naturally, the massive growth in computing power and available memory size in the hardware platforms has had a tremendous impact on the supported software. For example, the extended hardware capabilities allow the use of comprehensive software libraries and support new programming languages (Java and C#, for instance) and integrated development environments (IDEs), such as Eclipse and .NET. These, in turn, yield a significant growth in software design productivity, which again enables larger and more advanced applications.

Consequently, the complexity of software is increasing exponentially as well. In the general case, Humphrey observed that the size of software used for any given function is growing by 10x every 5 years [Wat02]. For the case of embedded applications, some sources report an even higher growth rate. We can observe that the embedded software complexity doubles in size every 10 months.[1]

For embedded systems, which traditionally are very constrained in processing power and memory size due to their embedded nature (i.e. mobile, low power, low cost), today larger applications become possible. For one, the introduction of 32-bit MCUs allows the use of more complex programming languages, such as C/C++ and dialects. This results in higher software design productivity which, in turn, allows the introduction of more innovative applications in the embedded area. Indeed, the vast majority of innovations in embedded systems is attributed to advances in microelectronics and software design. In the automotive industry, for instance, over 90% of all innovations over the last years are electronics-based.

The growing dominance of electronics and software and the associated growth of costs is not only limited to mass products in embedded systems. For example, while the F-4 fighter jet essentially came with no firmware, the current F-22 fighter jet is fully equipped with microelectronics. At the same

[1] In some applications the network became the limiting factor. According to Nielson's law [Nie98], the network bandwidth doubles only every 21 months.

time, the costs from the F-4 to the F-22 has increased drastically, from $20 million to $257 million, and *half* of these costs are attributed to the embedded software [Gan05]. In automotive systems design, in contrast, the software content currently accounts for 'only' 8% of the total cost of an average car (Frost & Sullivan 2005). However, this ratio is steadily rising.

1.1.2 Productivity Gap

It is well-known in the hardware design community that, for a number of years now, the potential in chip complexity outpaces the design capabilities. This creates the hardware design productivity gap [Sem04].

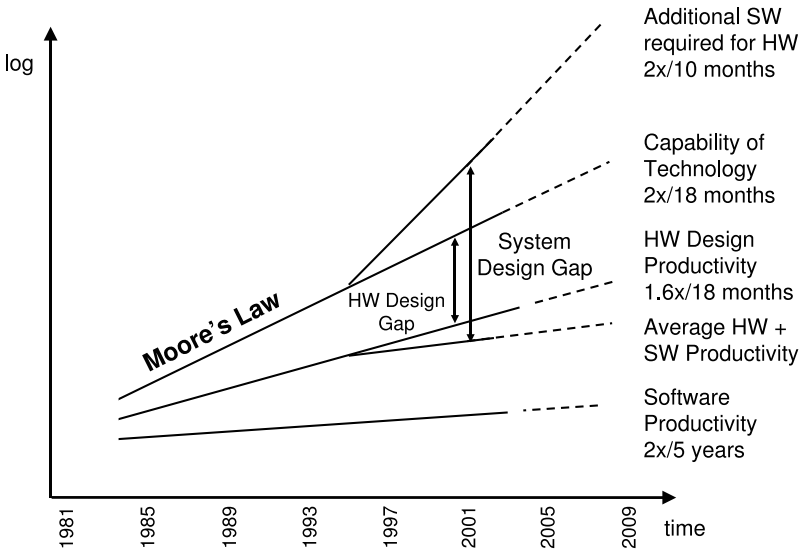

Figure 1.2. Hardware, Software, and System Design Gap. [Sources: ITRS, IFX, STM]

In particular, chip capacity doubles every 18 months according to Moore's law [Moo65], whereas the productivity growth in hardware design over the last years is estimated at 1.6x over 18 months. The latter is mostly attributed to the reuse of intellectual property (IP). Since the hardware capacity is growing faster than the productivity in its design, we face the so-called hardware design gap, as illustrated in Fig. 1.2.

Now, if we factor in the estimated software design growth, the situation for entire systems only gets worse. In particular, the productivity in software design reportedly increases by about 2x every 24 months. However, to satisfy the actual needs in embedded software complexity, we would need an estimated growth of 2x over 10 months (as discussed earlier in Sect. 1.1.1). Thus, in addi-

tion to the hardware design gap, we face a corresponding software design gap at the same time. Considering both problems together, as shown in Fig. 1.2, the two gaps combined result in a large system design gap.

We would like to emphasize that the real system design challenge is not only the addition of the two design gaps, but moreover the close interaction and tight dependency between the software and hardware domains. In other words, the necessary interfacing of software and hardware adds another layer of complexity. Thus, *Hardware-dependent Software (HdS)* is at the core of this system design challenge.

1.1.3 Design Productivity

To better understand the issues in software design productivity, we elaborate in this section briefly on the economics of software development and how its complexity can be measured and estimated.

The principal concern with increasing software size and complexity is the scalability of the development process. To reach an understanding of general scalability and provide a method for estimation of software design costs, Boehm presented an economic model, called COCOMO (Constructive Cost Model) [Boe81]. This model is based on the experience collected from multiple software projects and provides two interesting equations that estimate the amount of required effort and resources for software development. The first equation,

$$\texttt{SM} = 2.8 \cdot \texttt{KLoC}^{M} \quad \text{where } M = 1.2 \tag{1.1}$$

determines the number of staff months (SM) needed to develop a given amount of software code. Here, KLoC denotes the number of thousands of lines of code. The second equation,

$$\texttt{DT} = 2.5 \cdot \texttt{SM}^{0.32} \tag{1.2}$$

then allows to estimate the length of the needed development time (DT) in terms of chronological months.

As an example, a software module consisting of 10,000 lines of code is estimated to require about 44.4 staff months, according to the above equations, or about 5 software developers over a period of 9 months.

Both Eqs. 1.1 and 1.2 are already adjusted for the case of embedded software development (using the embedded mode of COCOMO which defines constants for medium-sized projects with tight hardware, software, and operational constraints, as they are typical for embedded software design). For better accuracy, the estimation equations can also be calibrated with additional coefficients, i.e. multiplied by a set of several cost factors C_i, like the required reliability, the product complexity, and the presence of real-time constraints. Furthermore,

whereas Boehm [Boe81] set $M = 1.2$ in Eq. 1.1, Ganssle [Gan04] proposes an M between 1.5 and 2.0 based on his experience.[2]

1.1.4 Productivity Crash

Of course, development efforts in real projects vary widely and we have to caution that estimation models, such as COCOMO in Sect. 1.1.3, can only provide a first and very rough approximation. In larger projects, we often face a *productivity crash* when we increase the number of developers beyond a certain threshold. Clearly, this drop in productivity can be attributed to the amount of interaction between the developers.

The reason for the productivity crash follows from the main dilemma that the number of interactions increases in the order of $O(n^2)$ where n is the number of involved team members. In other words, doubling the number of team members results in four times the number of distracting interactions. Some studies on software projects report roughly an optimistic 5% loss in productivity for each interaction. Studies by Joel Aron at IBM have shown that the productivity of a programmer can be significantly reduced to approx. one sixth when working in a team with a high number of interactions[3] [Bro95].

A potential solution to deal with this dilemma is to spilt a software project into completely autonomous parts, which can be independently processed by different developers [Gan05]. Given the module-based and hierarchical structure of typical multi- or many-core SoC projects, we can anticipate that this strategy may be successfully applied in such situations.

In other words, there is reasonable hope that a productivity crash can be significantly relaxed in the embedded design area when partitioning the design into independent blocks and applying *component-based design*. Following this argument, the strategy of independent development should also be applicable to networks of distributed embedded systems, such as those commonly found in automotive systems, for instance.

1.2 Hardware-dependent Software

Until a decade ago, only about 10% of the development costs of electronic systems were spent on software, whereas 90% of expenses went into the hardware. Today, this ratio has shifted significantly towards software.

[2] For $M = 1.75$, 10,000 lines of code require about 157 staff months, or about 13 developers over 1 year.

[3] He found that the productivity of a programmer largely varies between 10,000 instructions per staff-year for programmers with very few interactions to 1,500 machine instructions per staff-year for many interactions. This included design and programming and doubled when also covering system test.

1.2.1 Software Dominance

In the system design era, where hardware design has moved up to include the entire system-on-chip design task, software design and validation have gained tremendously in importance. In fact, software design starts to dominate the overall design process.

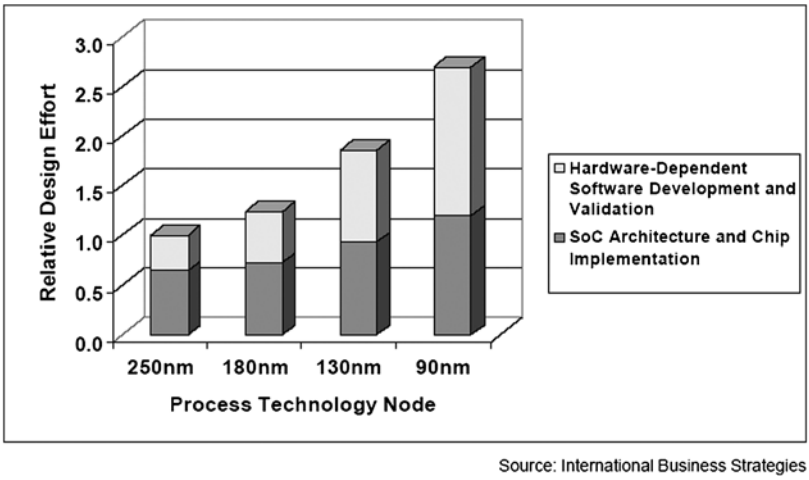

Figure 1.3. Hardware-dependent Software dominates in 90 nm designs.

The shift in focus from hardware- to software-centric design can be demonstrated in the relative design effort spent over different process technologies. Figure 1.3 illustrates the software/hardware ratios over different process technologies in 2002. We can clearly see that with older fabrication technology, i.e. in a 250 nm process, more effort is spend on hardware as on software design. However, in todays' state-of-the-art process technology, e.g. a 90 nm process, the software design effort clearly dominates over the hardware.

In addition to the constantly rising system size and complexity, there are two more drivers that increase the demand for more software content in systems, namely flexibility and configurability.

- Many systems need flexibility and feature-richness for different variations. Such flexibility can only be efficiently provided through the reuse and sharing of available hardware resources. This, in turn, requires the programming of hardware resources by use of software.

- Software configuration of standard hardware components is necessary to reduce overall system costs. This applies in particular to systems built

in mass-production, such as in the areas of telecommunication and automotive electronics. Many cars and mobile phones, for instance, support software reconfiguration, e.g. through a flash loader, after shipment.

We would like to emphasize here that both drivers, flexibility and configurability, specifically apply to the *Hardware-dependent Software (HdS)*, as opposed to the general application software. In other words, the parts of software most critical towards reconfiguration are those software modules that closely interact with the underlying hardware.

1.2.2 Importance of Hardware-dependent Software

In todays large scale interdisciplinary projects, the importance of Hds is still not understood. Therefore, the need for HdS is underestimated in the project planning phase, and sometimes even completely ignored. In contrast, HdS often becomes a crucial factor when the project progresses, and when the problem is discovered late, it usually becomes very expensive.

For instance, Ganssle [Gan04] reports costs for typical commercial firmware of about $15 to $30 per line of code, measured from the start of the project until it is shipped. However, it is not unusual for firmware costs to even reach $100 per line or more, if proper documentation and other secondary development tasks are included. As a consequence, a tiny 5 KLoC software module can easily reach a six digit budget.

In retrospective, the notion of HdS and its importance in electronic design automation (EDA) came up with the introduction of platform-based design [SVM01]. To discuss and address the issues, the Virtual Socket Interface Alliance (VSIA) initiated a development working group (DWG) on the topic of HdS in 2002. As a result of this working group, an initial taxonomy and terminology has been developed. Details can be found in [BMA05].

In the context of this chapter (and the entire book), we will define *Hardware-dependent Software (HdS)* as the software in an embedded system that closely interacts with the underlying hardware platform. More specifically,

- HdS is specifically built for a particular block of hardware, i.e. HdS is meaningless without that hardware.

- HdS and hardware together implement a systems' functionality, i.e. the hardware is meaningless without the HdS.

- HdS provides the application software with an interface to easily access the hardware features.

1.2.3 Hardware-dependent Software Architecture

Consistent with our definition of HdS in the previous section, we can identify HdS as a layer of software modules in between the application software and the underlying hardware platform. In other words, HdS can be seen as low level software.

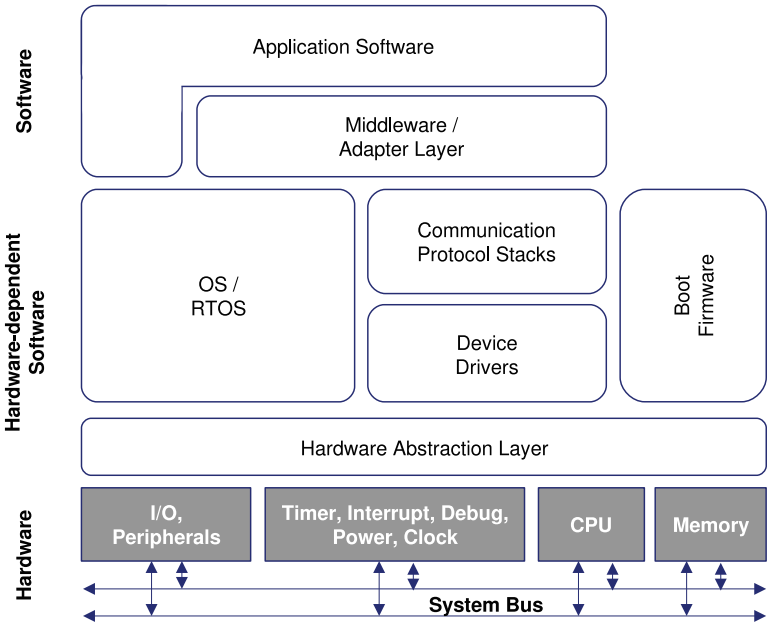

Figure 1.4. HdS in a layered software architecture.

Figure 1.4 illustrates the layering of the general HdS architecture. At the top third of the figure, we find the application software layer which is supported by a layer of various HdS modules in the middle. The HdS layer, in turn, is supported by the underlying hardware at the bottom third of Fig. 1.4.

HdS typically runs in the kernel space of an operating system, whereas middleware and application software run in user space. As such, HdS includes the software modules for boot code, device drivers, hardware-dependent portions of protocol stacks, and DSP algorithms.

More specifically, the general software stack of a typical HdS architecture, as shown in Fig. 1.4, consists of the following main components:

Application Software. One or multiple applications implement the overall functionality of the system. Application software may consist of multiple processes and/or threads. However, in most embedded systems, application software serves one single application with a dedicated purpose.

Middleware. Middleware represents a software layer that provides application-specific services. For instance, middleware can provide message-oriented communication services or SQL-oriented data access. Middleware sometimes is also referred to as an adapter layer between the application software and the operating system.

Operating System. An operating system (OS) is a software component that manages and coordinates application software tasks for sharing of available software and hardware resources. If an OS supports resource sharing under real-time constraints, it is called a real-time operating system (RTOS).

Communication Protocol Stacks. Communication protocols are typically implemented by layered software modules (more or less following a subset of the 7 layers in the ISO/OSI reference model) on top of device drivers.

Device Drivers. A device driver provides software access to a hardware resource, possibly through a hardware abstraction layer. Most device drivers provide six standard functions to initialize/reset the device, open, close, read and write data streams, and perform I/O control.

Boot Firmware. The boot firmware manages the initial boot process of a computer and typically resides in a read-only memory (ROM). It typically includes self-test routines, e.g., power-on self-test (POST), and a boot loader that initiates the actual OS. Examples of boot firmware are the basic I/O system (BIOS) and the newer unified extensible firmware interface (UEFI) used in regular personal computers (PCs).

Hardware Abstraction Layer. The hardware abstraction layer (HAL) is a software layer that provides an abstract interface to access hardware resources. The HAL is typically divided into access, register, and functional shielding.

We should emphasize that the layered software architecture discussed above is conceptual and simplified. Depending on the actual embedded system and its application, the software modules used will vary widely in size and may be even left out entirely.

1.3 Chapter Overview

In the remainder of this book, we will focus on specific aspects of HdS in more detail. Together, the following chapters provide a composition of basic principles with current and upcoming practices and tools for HdS development.

The chapters are generally self-contained, so that they can be read in the given or a different order. Therefore, all topics, the reader is already familiar with, can also be easily skipped if desired.

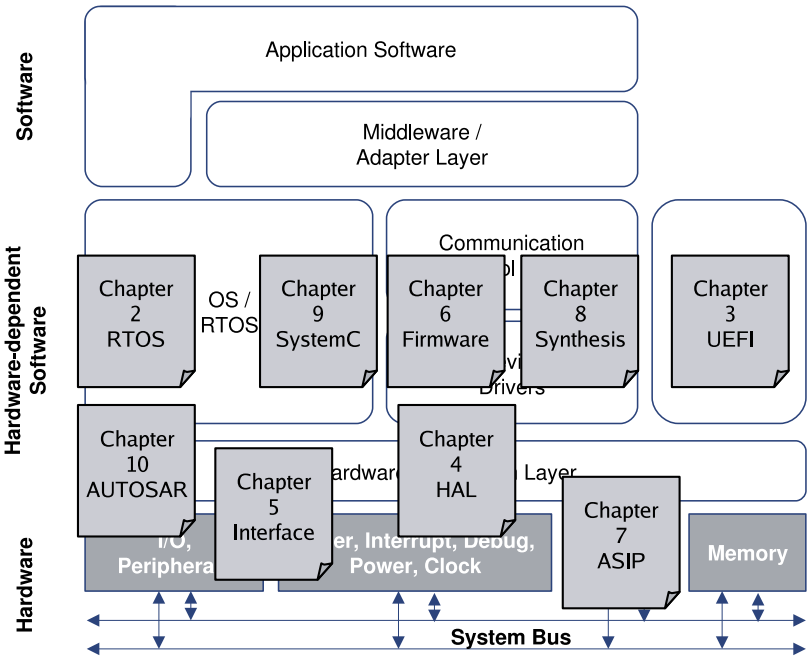

Figure 1.5. HdS coverage in the following book chapters.

The following chapters cover the various aspects of HdS quite comprehensively. As a broad overview, Fig. 1.5 illustrates the coverage of HdS topics discussed in each chapter in front of the background of the general HdS software architecture.

After this introductory chapter, the first three chapters focus on three specific components of HdS, namely RTOS, boot firmware, and HAL. In particular, Chap. 2 introduces RTOS principles and explains its basic services including process management, scheduling algorithms, and inter-task communication and synchronization. With respect to different application-specific requirements, appropriate RTOS techniques and kernel types are discussed as well.

Next, Chap. 3 presents a novel approach around the boot firmware which is needed to start any computer properly. It presents an overview of the Unified Extensible Firmware Interface (UEFI) which is expected to replace the old BIOS in the next years.

Chapter 4 then digs into the details of the layered organization of the HdS architecture and discusses in particular the hardware abstraction layer (HAL). A main topic of this chapter are the techniques that achieve flexibility as well as portability of software by use of the HAL application procedural interface (API).

Thereafter, Chap. 5 outlines the modeling and implementation of the hardware/software interface in detail. The chapter takes great care of describing the software programmers' view of the interface and provides many practical examples in form of C source code fragments.

Chapter 6 then extends the hardware/software interface discussion and presents, in a proven industrial setting, techniques that automatically derive interface descriptions for both hardware and software from a common high-level description. As such, this chapter describes the state-of-the-art in industrial telecommunication firmware development.

The second part of this book is dedicated to EDA tools and the critical tasks of verification and validation. Chapter 7 focuses on EDA tools for application-specific instruction set processors (ASIP). In particular, the issues of configurability and extensibility of an ASIP-specific tool chain are discussed, including the effects on the integrated development environment (IDE), the compiler, profiler, instruction-set simulator (ISS), and other supporting tools.

Chapter 8 describes how automatic software generation can significantly reduce the design time of HdS. The chapter proposes a system-level design flow that allows to generate HdS automatically from an abstract system specification. A system compiler is presented which generates a software implementation, including software binaries of the application, communication protocols, and operating system code.

Chapter 9 covers RTOS simulation, estimation and configuration for different scheduling strategies and task priorities. An RTOS simulation based on an abstract SystemC model is presented which supports modeling of scheduling, preemption and interrupts, in the context of a system design flow.

Finally, Chap. 10 describes verification of AUTOSAR-based automotive systems by means of SystemC simulation at transaction level. It outlines how different AUTOSAR standard entities, like basic software (BSW), virtual functional bus (VFB), and run-time environment (RTE), map to a SystemC-based design flow for timing analysis.

References

[BMA05] Brian Bailey, Grant Martin, and Thomas Anderson. *Taxonomies for the Development and Verification of Digital Systems*. Springer, Berlin, 2005.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, 1981.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Boston, anniversary edition, 1995.

[Gan04] Jack Ganssle. Firmware basics for the boss. *Embed. Syst. Design*, January 2004.

[Gan05] Jack Ganssle. Subtract software costs by adding CPUs. *EE Times*, April 2005.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[Nie98] Jakob Nielsen. Nielsen's Law of Internet Bandwidth, April 1998. www.useit.com

[Sem04] Sematech Inc. International technology roadmap for semiconductors (ITRS), 2004 update, design, 2004. www.itrs.net

[Shi08] Shimano, Inc. Shimano Turns on the Power, August 2008. bike.shimano.com

[SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test Comput.*, 18:23–33, 2001.

[Wat02] S. Humphrey Watts. The future of software engineering: Part V. *Software Engineering Institute*, First Quarter 2002.

Chapter 2

BASIC CONCEPTS OF REAL TIME OPERATING SYSTEMS

Franz Rammig, Michael Ditze, Peter Janacik, Tales Heimfarth, Timo Kerstan, Simon Oberthuer and Katharina Stahl

Abstract Real-time applications usually are executed on top of a Real-time Operating System (RTOS). Specific scheduling algorithms can be designed. When possible, static cyclic schedules are calculated off-line. If more flexibility is needed on-line techniques are applied. These algorithms are bound to priorities which can be assigned statically or dynamically. Designing a proper RTOS architecture needs some delicate decisions. The basic services like process management, inter-process communication, interrupt handling, or process synchronization have to be provided in an efficient manner making use of a very restricted resource budget. Various techniques like library-based approaches, monolithic kernels, microkernels, or virtual machines/exokernels are applied, based on specific demands. Safety critical application can be supported by separation of applications either in the time or the space domain. Multi-core architectures need special techniques for process management, memory management, and synchronization. The upcoming Wireless Sensor Networks (WSN) generate special demands for RTOS support leading to dedicated solutions. Another special area is given by multimedia applications. Very high data rates have to be supported under (soft) RT constraints. Based on the used encoding techniques (e.g. MPEG) dedicated solutions can be created.

Keywords: RTOS, Scheduling, Safety Critical Systems, Wireless Sensor Networks

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

2.1 Introduction

Most embedded systems are bound to real-time constraints. In production control the various machines have to receive their orders at the right time to ensure smooth operation of a plant and to fulfill customer orders in time. Railway switching systems obviously have to act in a timely manner. In flight control systems the situation is even more restrictive. Inside technical artifacts many operations depend on timing, e.g. the control of turbines or combustion engines. This is just a small fraction of such applications. Even augmented reality systems are real-time applications as augmenting a moving reality with outdated information is useless or even dangerous.

"Real-time" means that the IT system is no longer controlling its own time domain. Now it is the progress of time of the environment which dictates how time has to progress inside the system. This environmental time may be the real one of our physical world or it may be artificially generated by some surrounding environment as well. For the embedded system there is no difference between these options. Kopetz defines real-time systems as "A real-time computer system is a computer system in which the correctness of the system behaviour depends not only on the logical results of the computation, **but also on the physical instant at which these results are produced**" [Kop97]. This means that in strict real-time systems a late result is not just late but wrong. The meaning of "late" of course has to be defined dependent on the specific application. In case of an air-bag controller it is intuitively clear what real-time means and it is easy to understand that a late firing of the air-bag is not only late but definitely wrong.

It can be concluded that in real-time systems the program logic of application tasks has to be augmented by information about timing. Such timing information contains the earliest point of time the task may be started as well as the latest allowed finishing time. This, together with the program logic may be seen as a specification for the computing system what to do and when to do it.

Many such tasks may have to be executed concurrently on an embedded computing system. Such situations usually are handled by some kind of operating system. The same is true in case of real-time systems. But now an additional objective function is introduced, an objective function which dominates most other ones: Formulated real-time constraints have to be respected. An operating system which is capable of taking care of this is called a "Real-time Operating System (RTOS)". Of course some additional information is needed by an RTOS to manage real-time tasks. Especially the worst-case execution time (WCET) on the specific target architecture of any real-time task has to be available. Determining the WCET of a task is a demanding goal on its own. It must never be underestimated. On the other hand the potential

over-estimation has to be reduced as far as possible to allow efficient system implementations.

The above discussion indicates that we first have to discuss fundamental properties of real-time tasks. On this basis we can then introduce basic techniques used in RTOS to handle such tasks. We will concentrate on real-time scheduling and on schedulability analysis.

2.2 Characteristics of Real-Time Tasks

First of all a real-time task is a task like any other. However, there is an essential difference to other computation: the notion of *time*. *Time* means that the correctness of the system depends not only on logical results but also on the time the results are produced. In contrary to other classes of systems in a real-time system the system time (*internal time*) has to be measured with same time scale as the controlled environment (*external time*). One parameter constitutes the main difference between *real time* and *non-real-time*: the ***deadline***. Any postulated deadline has to be met under all (even the worst) circumstances. This has the consequence that real-time means predictability. It is a widespread myth that real-time systems have to be fast. Of course they have to be fast enough to enable guaranteeing the required deadlines. Most of all, however, a real-time system has to be predictable. Ensuring this predictability even may slow down a system.

Real-time systems can be characterized by the strictness of real-time restrictions.

A real-time task is called *hard* if missing its deadline may cause catastrophic consequences on the environment under control. Typical application areas can be found in the automotive domain when looking at e.g. power-train control, air-bag control, steer by wire, and brake by wire. In the aeronautics domain engine control or aerodynamic control may serve as examples.

A RT task is called *firm* if missing its deadline makes the result useless, but missing does not cause serious damage. Typical application areas are weather forecast or decisions on stock exchange orders.

A RT task is called *soft* if meeting its deadline is desirable (e.g. for performance reasons) but missing does not cause serious damage. Here typical application areas are communication systems (voice over IP), any kind of user interaction, or comfort electronics (most body electronics in cars).

Concerning timing, a real-time task J_i can be characterized by the following parameters: Arrival time a_i, WCET C_i, (absolute / relative) deadline d_i / D_i, start time s_i, finishing time f_i (see Fig. 2.1).

The *arrival time* a_i is the time J_i becomes ready for execution. Sometimes it is also called *request time* or *release time*, denoted by r_i. It is a parameter

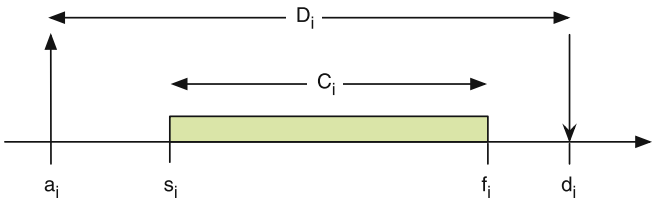

Figure 2.1. Parameters of a real-time task.

under control of the application task. A task which is not known to the RTOS obviously is also not considered by it.

Another parameter that comes with the application task is its *computation time* C_i. This is the WCET which has to be determined previously and has to be known by the RTOS. Of course it is the WCET only under the assumption that the task is not interfered by any other task. Interference can happen only when managed by the RTOS. So any influence by interference due to other tasks is known by the RTOS and has to be considered by the RTOS.

The third parameter that comes with the application task is its *deadline*. Here a distinction has to be made between an *absolute deadline*, denoted by d_i and a *relative* one, denoted by D_i. Absolute deadline means a value with respect to the global time of the entire system while relative deadline means relative to the arrival time of the respective task. In any case, it has to be guaranteed by the RTOS that the tasks will be finished not later than the deadline, independent from any circumstances, even the worst imaginable ones.

The remaining two parameters are under control of the RTOS. It is the RTOS that decides when to start an application task, i.e. to set the *start time* s_i. Of course it can never be earlier than the arrival time a_i as before this time the task is entirely unknown to the RTOS.

In a similar manner it is up to the RTOS when a task reaches its *finishing time* f_i. It can be calculated to be at least $s_i + C_i$. However, J_i may be disturbed by other tasks so that the finishing time f_i may be later. Whenever this happens, it happens under control of the RTOS. Therefore it is the responsibility of the RTOS to guarantee that f_i is not later than the respective deadline.

Orthogonally to the distinction into soft, firm, and hard real-time two main classes of tasks can be identified: *periodic* and *aperiodic* ones. Both types are generic tasks, i.e. over time a sequence of instances is generated. Usually such an instance is called *job*. All instances share the same code and therefore the same WCET and relative deadline. In case of periodic tasks these instances show up with a fixed *period*, denoted by T_i. This means that once knowing the first arrival time, all following arrival times are pre-defined. The first arrival

time, i.e. the arrival time of the first instance usually is called the phase of this (generic) task, denoted by ϕ_i.

In the case of aperiodic tasks no period is present, i.e. the next arrival time of an instance of an aperiodic task is unknown a priori and may happen at any time. Usually the assumption is made, that up to the absolute deadline of a task instance no additional instance will be issued into the system.

Periodic tasks reflect directly the "sense–execute–act" loop in control applications. They therefore represent the main workload of embedded systems. Any RTOS usually is optimized into the direction of handling such tasks in an optimized manner. Aperiodic tasks appear for initialization reasons, for setting of parameters and, most importantly, for the handling of interrupts that show up in an aperiodic manner. There is a certain style of programming embedded systems which reduces the software to a strictly event driven system. The NesC programming language used for TinyOS [CHB+01] follows this principle. So, whenever this style of programming has to be supported, the handling of aperiodic tasks becomes a major issue of the RTOS.

Tasks of a given task set may be independent or dependent. A task J_i is called dependent on task J_k if J_i cannot be started before J_k has been finished. Dependence is a transitive property. A task J_i is called direct predecessor of task J_k if there is no task J_m between them such that J_m is dependent on J_i and J_k is dependent of J_m. Dependencies can be defined using a directed acyclic graph (DAG). Obviously dependencies introduce additional constraints that need to be handled by the RTOS.

Unfortunately direct support for expressing dependencies is rarely found in modeling and programming languages. UML Sequence Charts represent dependencies, however in a rather unwieldy manner. In programming languages dependencies have to be coded in detail and therefore are hard to be identified in the program code.

Another constraint on task sets is introduced by non sharable resources. A *resource* is any object to be used by a task. In HW this may be some circuitry like an ALU or a bus, in SW it may be a certain data structure, a set of variables, or a memory area. A resource is called *private resource* if it is dedicated to a particular task, i.e. it is not used by any other one. In contrary to this a *shared resource* is to be used by more than one task. In HW a bus is a typical example of a shared resource. It is also a typical example for this class of shared resources that need most care in handling: an *exclusive resource* is a shared resource where simultaneous access from different tasks is not allowed. Coming back to the bus example, a bus is an ordinary shared resource from the point of view of components reading from this bus but an exclusive one for any writer.

Like in the case of dependencies, direct support for specifying the class of resources is lacking in both, modeling and most programming languages.

There are techniques to handle such cases. In HW design special arbiters have to be included into the circuit. Unfortunately in VHDL, e.g., they are just components like any others, i.e. they cannot be identified easily. In SW the concept of a so called *critical section* is introduced. This is a piece of code that is to be executed under mutual exclusion constraints. The management then has to be coded directly, e.g. using semaphores [Dij68]. They constitute the link to the operating system as it is the OS which provides the semaphore operations as system services. It will be shown later that the concept of semaphores needs careful rethinking when real-time systems have to be built.

To sum up, real-time tasks can be characterized by a well defined set of parameters. Fortunately in most publications the same abbreviations are used for them.

Γ set of tasks. This set may consist of aperiodic ones, periodic ones, or both.

τ_i a generic task. This means that over time many instances of this task will exist.

τ_{ij} instance j of task τ_i.

$r_{i,j}$ *release time* of $\tau_{i,j}$. The release time is an absolute value and specific for each instance.

ϕ_i phase of τ_i ($= \tau_{i,1}$, i.e. release time of first instance). It is a parameter of the entire generic task.

T_i period of τ_i (= interval between two consecutive activations).

D_i relative deadline of τ_i (relative to release time, therefore a parameter of the entire generic task).

$d_{i,j}$ absolute deadline of $\tau_{i,j}$ ($d_{i,j} = \phi_i + (j-1)T_i + D_i$). It is a property of the specific instance.

$s_{i,j}$ start time of $\tau_{i,j}$ ($s_{i,j} \geq r_{i,j}$). It is an absolute value and specific for each instance.

$f_{i,j}$ finishing time of $\tau_{i,j}$ ($f_{i,j} \leq d_{i,j}$). It is an absolute value and specific for each instance.

2.3 Real-Time Scheduling

Given is a set of n generic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a set of m processors $P = \{P_1, \ldots, P_m\}$, and a set of s resources $R = \{R_1, \ldots, R_s\}$. There may exist precedences, specified using a precedence graph (DAG) and, as we are considering real-time systems, timing constraints are associated to each task.

The goal of *real-time scheduling* is to assign processors from P and resources from R to tasks from Γ in such a way that all task instances are completed under the imposed constraints. This problem in its general form is NP-complete! Therefore relaxed situations have to be enforced and/or proper heuristics have to be applied. In principle scheduling is an on-line algorithm. Under certain assumptions large parts of scheduling can be done off-line. Generating static cyclic schedules may serve as an example. In any case all exact or heuristic algorithms should have very low complexity.

In principle scheduling algorithms may be *preemptive* or *non preemptive*. In preemptive approaches a running task instance may be stopped (preempted) at any time and restarted (resumed) at any time later. Any preemption means some delay in executing the task instance, a delay which the RTOS has to take care of as it has to guarantee respecting the deadline. In case of non preemptive scheduling a task instance once started will execute undisturbed until it finishes or is blocked due to an attempt to access an unavailable exclusive resource. Non preemptive approaches result in less context switches (replacement of one task by another one, usually a very costly operation as many processor locations have to be saved and restored). This may lead to the conclusion that non preemptive approaches should be preferable in real-time scheduling. However, not allowing preemption imposes such hard restrictions on the scheduler's freedom that for most non-static cases predictable real-time scheduling solutions with an acceptable processor utilization rate are known only if preemption is allowed. In the sequel basic preemptive real-time scheduling algorithms for periodic and aperiodic tasks will be discussed shortly.

2.3.1 Rate Monotonic Priority Assignment

All real-time scheduling algorithms strictly rely on priorities. So the basic principle is that at any point of time always this task instance τ_{ij} is executed which has the highest priority among all active task instances. A task instance τ_{ij} is active in the period between its release time $r_{i,j}$ and its finishing time $f_{i,j}$. In this section it is assumed that a task set Γ of independent tasks τ_i with no resource conflicts has to be scheduled.

Rate Monotonic Priority Assignment (RM) is a so-called static priority scheduling algorithm. In such algorithms priorities are assigned a priori and are never modified during runtime of the system. RM assigns priorities simply in accordance with its periods, i.e. the priority is as higher as shorter is the period which means as higher is the activation rate. So RM is a scheduling algorithm for periodic task sets. It is assumed that the periods of the different tasks differ, we have so called *multi-rate* systems (handling of single-rate systems is trivial). In addition it is assumed that the relative deadlines of the tasks are identical to the periods ($D_i = T_i$). RM is intrinsically preemptive as it may happen that

a task instance is running when a new instance of a lower-period, i.e. higher priority task is released. In such a case the currently running task is preempted in favor of the newly arriving one. See Fig. 2.2 for an example of a schedule produced by RM. There are two tasks, τ_1 (period $T_1 = 3$, WCET $C_1 = 1$) and τ_2 (period $T_2 = 7$, WCET $C_2 = 4$). The first two instances of τ_2 are preempted once, the third one twice due to starting new instances of the higher priority τ_1.

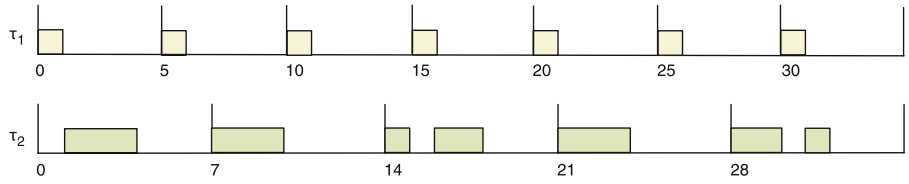

Figure 2.2. RM example schedule.

It can be shown [But04, p. 78ff] that RM is optimal among all fixed priority scheduling algorithms in the sense that if RM does not provide a feasible schedule than no other fixed priority algorithm can.

A hard real-time system cannot be started before carefully analyzing its schedulability. A specific schedule is called *feasible* if all instances τ_{ij} of all tasks τ_i can be completed according to a set of specified constraints. A set of tasks is called *schedulable* if at least one algorithm does exist that can produce a feasible schedule. When applied to RM the algorithm has already been selected. The question now is to decide a priori whether a given task set Γ is schedulable by RM.

A simple test is given by comparing the utilization of the given task set with the utilization of the worst imaginable task set which is still schedulable by RM. This constitutes a least upper bound (LUB) of utilization among all potential task sets.

Given a set Γ of aperiodic tasks the *processor utilization factor* U is the fraction of processor time spent in the execution of the tasks set. C_i/T_i is the fraction of processor time spent in executing task τ_i. The utilization U of Γ can be calculated simply by the sum

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{2.1}$$

Obviously this can be done off-line as no runtime parameters are used in this formula. It can be shown [But04, p. 87ff] that the utilization LUB, U_{lub}, i.e. the utilization of the worst case task set is given by $U_{\text{lub}} = n(2^{\frac{1}{n}} - 1)$ which converges towards $U_{\text{lub}} = \ln 2 \approx 0.69$ with increasing n. As the number n of tasks in the given task set Γ is known a priory as well, U_{lub} can be calculated

off-line and as a consequence the schedulability analysis can be performed off-line. Unfortunately $U_{\text{lub}} = n(2^{\frac{1}{n}} - 1)$ is sufficient but not necessary to guarantee the feasibility of a given task set. There may exist specific task sets which are schedulable under RM despite the fact that for their utilization U it holds that $U_{\text{lub}} < U < 1$.

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{\frac{1}{n}} - 1\right), \tag{2.2}$$

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{2.3}$$

In RM the assumption is made that the relative deadlines of the tasks are identical to the periods ($D_i = T_i$). This restriction can be relaxed easily by replacing T_i by D_i in the definition of priority assignment. The algorithm then is called *Deadline Monotonic Priority Assignment (DM)*. Even the schedulability analysis can be transferred directly resulting in the condition as in Eq. 2.2, which in this case, however is even more pessimistic than in the RM case. A crisp schedulability test for fixed priority assignment strategies like RM and DM is given by the so-called *Response Time Analysis*. In this case for each task τ_i the largest finishing time among all instances τ_{ij} with respect to its relative deadline D_i is calculated precisely. This largest finishing time is called *response time* R_i of task τ_i. If for all tasks τ_i of the given task set Γ R_i is not greater than the relative deadline D_i, schedulabilty is proven.

The response time R_i can be calculated by $R_i = C_i + I_i$ where C_i is the WCET of τ_i and I_i is the interference due to pre-emption by higher priority tasks. The question is how to calculate I_i. For this we have to sum up over all higher priority tasks τ_j, $j < i$ the number of inferences given by $\lceil R_i \,/\, T_j \rceil$ multiplied by the duration of the respective interference C_j. This results in the definition of the response time R_i of task τ_i as shown in Eq. 2.3.

Unfortunately this is a recurrent equation as the argument R_i stands on both sides of the equation. By an iterative algorithm, however we can calculate the least fixpoint of the equation. If it is less or equal to the relative deadline the test for this specific task is successful, otherwise it fails. The test is successful for the entire task set Γ if it does not fail for a single task τ_i. So this test is rather computation intense. Fortunately it can be carried out off-line as no runtime parameters have to be known.

2.3.2 Earliest Deadline First Scheduling

In contrary to RM or DM, Earliest Deadline First (EDF) scheduling is a dynamic priority assignment. Now task instances τ_{ij} always get assigned a

priority inverse proportional to their absolute deadline d_{ij} i.e. the priority is as higher as the absolute deadline is shorter (ties are broken in favor of already running task instances). This means that whenever a task instance is released the priorities have to be re-calculated and the priority of a task (i.e. of its instances) may vary during runtime. Despite this difference the handling of task instances is the same as in the case of RM or DM: At each instance of time this task instance is executed that currently has the highest priority among all active task instances. Therefore, like RM or DM, EDF is intrinsically preemptive. Figure 2.3 shows an example schedule produced by EDF for the same task set as used in Fig. 2.2. The third instance of τ_2 is preempted only once as in the case of equal absolute deadlines the already running task is preferred.

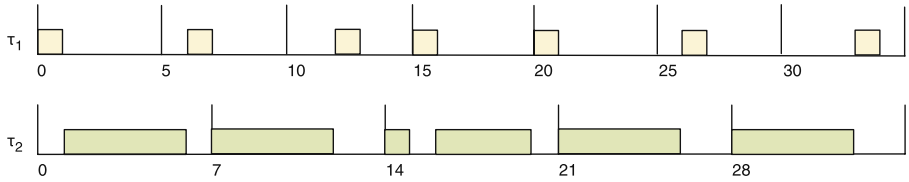

Figure 2.3. EDF example schedule.

It can be shown [But04, p. 51ff, 92] that EDF is optimal among all periodic task scheduling techniques in the sense that if EDF does not provide a feasible schedule then no other periodic task scheduling algorithm can. Another good property of EDF is that schedulability analysis is really simple for EDF. A simple utilization test can be applied where $U_{\text{lub}} = 1$, i.e. the utilization just has to be compared with the constant 1.

EDF can also be applied to aperiodic task sets. Its optimality guarantees that the maximal lateness is minimized when EDF is applied. Lateness $L_{i,j}$ of a task instance τ_{ij} is defined as the time between absolute deadline and finishing time: $L_{i,j} = f_{i,j} - d_{i,j}$.

So it seems that EDF has only advantages over fixed priority algorithms. Despite this fact those algorithms still serve as the workhorse in most RTOS systems. It is argued that EDF is more complicated to implement as at runtime it has to rearrange priorities while RM or DM do not. EDF is also considered to be extremely sensitive to overload conditions where a so-called *Domino Effect* may happen, i.e. missing a single deadline may result in missing the deadlines of all tasks of a task set. In a recent publication [But05] however, it has been shown that most of the arguments against EDF are not relevant in practical applications.

2.4 Operating System Designs

The most common Operating Systems are based on kernel designs. The kernel design has been around for almost 40 years and offers a clear separation between the operating system and the application running on top of it, as they are allocated in different memory locations. The processes can use the kernel functionality by performing system calls. System calls are software interrupts which allow switching from the application to the operating system. Therefore the kernel needs to install an interrupt handler for different modes of operation, depicted in Fig. 2.4, that can be enabled in the program status word (PSW): User mode and Supervisor mode. For this reason, protection is done in modern SoCs at peripheral side. Some registers can be changed only if the CPU signals a specific execution mode (e.g. master mode) via a set of additional HW-signals in the bus infrastructure.

Processes outside the OS are executed within user mode and are not allowed to execute instructions which are only available in supervisor mode. This means that the user mode instructions constitute a non-critical subset of the supervisor mode instructions. During runtime of a process the supervisor mode bit within the PSW is disabled and can only be enabled if an interrupt such as a system call or an external interrupt occurs. The operating system is responsible for enabling the user mode at the time a user process is activated. Typically a user process has its own virtual memory address space which separates it completely from the kernel. However this is not possible on all embedded microcontrollers as they may lack a memory management unit (MMU) enabling the use of virtual memory.

The use of virtual memory, if there is a MMU available, has to be realized without any unbound memory accesses like swapping on an external disk or replacing translation lookaside buffer (TLB) entries by searching a dynamically sized page table.

To use the functionality provided by the OS kernel it is necessary to define an interface that allows applications to use it. This interface is called the appli-

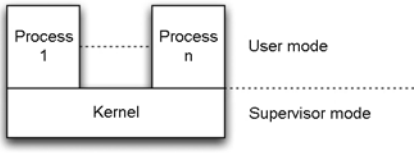

Figure 2.4. Execution modes.

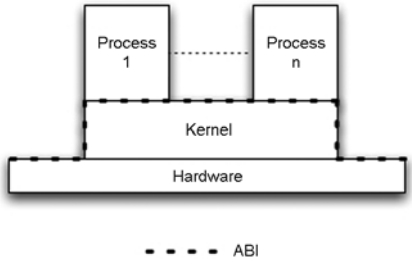

Figure 2.5. Application binary interface.

cation binary interface (ABI). The ABI defines a set of **system calls**, a **register usage convention**, a **stack layout** and enables binary compatibility whereas an application programming interface (API) enables source code compatibility through the definition of a set of function signatures providing a fixed interface to call these functions. Figure 2.5 shows the location of the ABI within an architectural schemata.

The kernel itself can be built in many ways and usually provides the following basic activities: **Process management**, **process communication**, **interrupt handling**, and **process synchronization**.

Process management is responsible for process creation, process termination, scheduling, dispatching, context switching and other related activities.

Interrupt handling in a RTOS is different to the standard implementation of an ordinary OS. In an ordinary OS interrupts can preempt running processes at any time. This can lead to unbound delays which are not acceptable in a RTOS. Therefore the handling of interrupts is integrated into the scheduling so that it can be scheduled along with the other processes and a guarantee of feasibility can be achieved even in the presence of interrupt requests.

Another important role of the kernel is to provide functionalities for the synchronization and communication of processes. The use of ordinary semaphores is not possible within a RTOS as the caller may experience unbound delays in case of a priority inversion problem. Therefore the synchronization mechanisms need to support a resource access protocol such as Priority Inheritance, Priority Ceiling or Stack Resource policy [But04, p. 191ff].

As already stated there are different ways to realize a kernel. Today the main design question is whether to us a monolithic kernel, a microkernel or a combination called hybrid kernel [Sta01, Tan01].

2.4.1 Library-Based RTOS (“Kernel-Less” Approach)

For systems without MMUs the RTOS can be built as a library which is linked together with the application. This results in one single executable which is executed in one single address space. Therefore no loader is required to dynamically load applications at run-time, by this minimizing the operating system code. Another advantage of a library-based RTOS and the execution in a single common address space is that system calls can be simply implemented as function calls. Thus no context-switches are required when calling an operating system function. This is often more efficient and less time consuming as a full context switch with address space changes when having an RTOS implemented as a kernel in a separated address space. The disadvantages of a library based RTOS running on systems with no “full MMU” is the lack of security through hardware memory separation. All application and operating system activities have to be implemented as threads in the same address space.

Bugs in one part of the system can easily affect the whole system. But on small microcontrollers on which only one application is executed this disadvantage is acceptable.

An example for a library based operating system is the operating system library DREAMS. Operating systems and run-time platforms for even heterogeneous processor architectures can be constructed from customizable components skeletons out of the DREAMS (**D**istributed **R**eal-time **E**xtensible **A**ppli cation **M**anagement **S**ystem) library [Dit99]. By creating a configuration description all desired objects of the system have to be interconnected and customized afterwards in a fine-grained manner. The primary goal of that process is to add only those components and properties that are really required by the application.

2.4.2 Monolithic Kernels

The monolithic approach of building a kernel is straightforward. All functionality provided by the OS is realized within the kernel itself. "The structure is that there is no structure" [Tan01]. The kernel consists of a set of procedures which are able to call each other without any restrictions. Figure 2.6 shows a call graph of a totally unstructured monolithic kernel versus a monolithic kernel which is separated into service functions and help functions to bring at least some structure into the kernel. The service functions are the entry points for the interrupts which are demultiplexed in the main function and delegated to the associated service function.

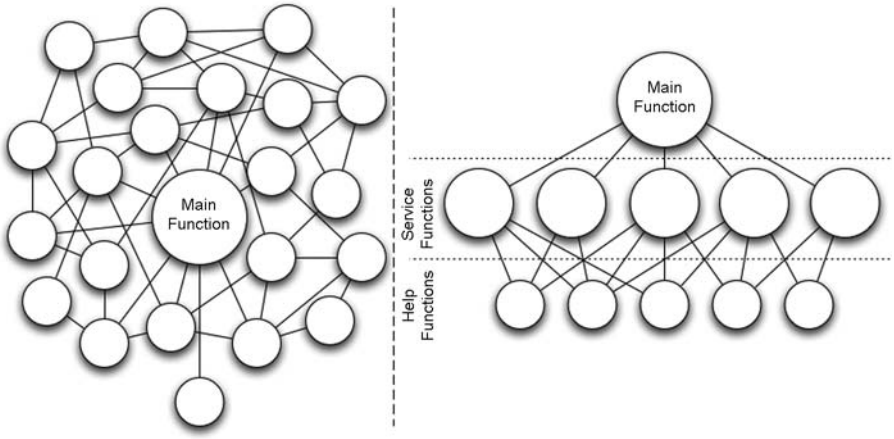

Figure 2.6. Unstructured vs. structured monolithic kernel.

The service functions can use any support function they need. The main advantage of monolithic kernels is their performance. As reaction to a sys-

tem call a context switch to the operating system has to be performed and the appropriate service functions have to be executed in the kernel space. This is pretty simple, as there are only function calls that need to be performed.

In the case of monolithic kernels it cannot be excluded that any single fault occurring within the kernel functions can lead to a total crash of the whole system. In most cases device drivers included in a monolithic kernel are very error-prone. Several studies on software dependability report fault densities of 2 to 75 bugs per 1000 lines of executable code. Drivers, which typically comprise 70% of the operating system code, have a reported error rate that is 3 to 7 times higher. A common example that can lead to a total crash is an unchecked pointer that may contain a wrong address. This results in overwriting of sensitive kernel data such as the kernel code itself [OW02, Sta01, Tan01, BP84, THB06].

2.4.3 Microkernels

To clean up the structural mess of monolithic kernels Fig. 2.7 shows microkernel design was developed. It reduces the services provided by the kernel dramatically by putting all services, which are not essentially necessary for the microkernel, into user space as isolated processes. The service processes typically behave like servers of the client-server model. To use such a service an application needs to send a message with a service request to the service which receives the request, completes the request and sends back a response message to the client application.

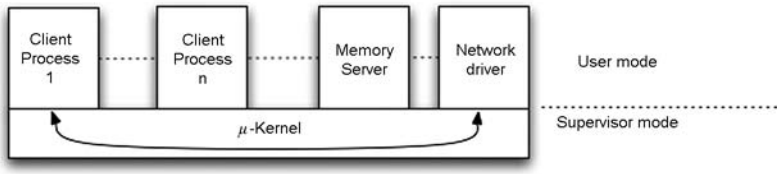

Figure 2.7. Microkernel architecture.

The big question is which services are not essentially necessary for the microkernel. The common approach puts the following services into the microkernel itself: **Dispatcher**, **Scheduler**, and **Memory Manager**.

Whether it is necessary to put the memory manager into the microkernel is a topic that has been discussed for a long time without any general agreement. However some memory management for the kernel objects itself is needed within the microkernel.

The big advantage of microkernels against monolithic kernels is the clear separation of services from the kernel itself making the kernel a very small

piece of software that provides a better fault isolation and can be maintained more easily than a monolithic kernel. The fault isolation prevents crashing the whole system. Even if e.g. a driver located in user space fails it is not possible for the driver to manipulate any kernel sensitive data like the kernel code.

The price we have to pay for the better structuring and fault isolation is that we get a high amount of interprocess communication through message passing and a high amount of context switching. The reason for this is that for every system call at least two messages have to be sent and four context switches have to be performed. This is illustrated in Fig. 2.8.

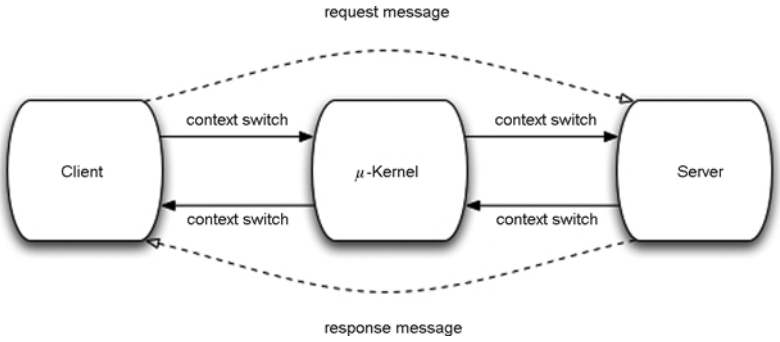

Figure 2.8. Client/Server IPC.

In contrast to monolithic kernels we also have to deal with an impact on the real-time behavior, because now system calls are not necessarily executed at the time they have been initiated. The reason is that the services behave like regular processes that have to be scheduled by the real-time scheduler. There are several approaches to deal with that problem. A very simple one is to use priority message queues for the service requests within the server and to apply priority inheritance on the server processes to guarantee that no unbound blocking time can occur [Sta01, Tan01].

2.4.4 Virtual Machines and Exokernels

The main idea of system virtual machines is to provide an exact copy of the available hardware for every virtual machine. Therefore a small control program is necessary to assign the available hardware to the virtual machines. This program is called the virtual machine monitor (VMM) or hypervisor (cf. Fig. 2.9). This program is the only code executed in supervisor mode and ensures that the virtual machines are clearly isolated from each other.

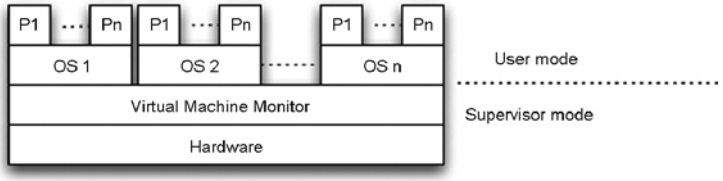

Figure 2.9. Virtual Machine Monitor.

The biggest issue to be solved is the question whether virtualization can be achieved efficiently. To answer this the instruction set architecture (ISA) plays the most important role. The ISA is divided into sensitive and innocuous instruction. Sensitive instructions interact with hardware and need to cause a trap to activate the VMM. Innocuous instruction can be executed natively if possible (provided that the ISAs of the host and the virtual machine are identical). If instructions cannot be executed natively they need to be emulated. For emulation the target code to be executed on a different host ISA needs to be transformed before it can be executed. The question whether an efficient VMM can be built is reduced to the question whether the set of sensitive instructions is a subset of the set of privileged instructions as in Fig. 2.10 [PG74].

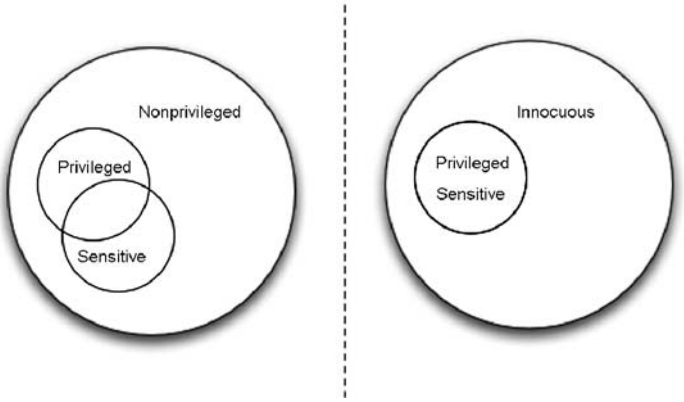

Figure 2.10. Efficiency classification of ISAs.

Exokernels are very similar to virtual machine monitors, but they differ in the way that exokernels do not provide an exact copy of the available hardware. Instead they partition the available resources and assign them to the virtual machines running on top of the exokernel. A good example is the main memory of the system. VMMs provide an exact copy of the complete main memory

to the virtual machines running on top of the VMM. The VMM needs to map the memory of every virtual machine to the real physical memory. Exokernels do not have to manage such a mapping as the different virtual machines would have only access to disjoint subsets of the available physical memory [Tan01, SN05].

2.5 RTOS for Safety Critical Systems

Computer systems that operate systems of critical responsibility are called safety-critical systems. Typically, a small deviation in the environment or the system's behavior, a failure or an error appearing within such a system can yield in hazardous situations and may cause catastrophes. Safety-critical systems therefore must not only guarantee real-time behaviour but furthermore they require absolute dependability and availability of system service. To free application developers from implementing safety and real-time mechanisms into each application, operating systems serve as the underlying platform designed towards supporting real-time and all safety-incorporating non-functional features.

Because of the critical consequences of a system failure, standards are required to specify the design and the development process. They define the methods and techniques that are required to prevent system failures and enforce a state-of-the-art quality-of-service in safety-critical applications. Two relevant standards exist: IEC 61508 and DO-178B. The title of the international standard IEC 61508 is "Functional safety of electrical/electronic/programmable electronic safety-related systems". It is a generic safety standard that forms the basis for many other—domain specific—standards. This standard defines requirements on the lifecycle of safety-related systems, from system development to its operation. It identifies measures and techniques for preventing failures and contains methods for controlling possible system failures. DO-178B is titled "Software Considerations in Airborne Systems and Equipment Certification" and specifies guidelines for the development of avionic software. It builds up a stringent application-dependent safety standard.

As recent trends are heading towards the integration of applications of different criticality levels on one single platform, operating systems for safety-critical applications face the challenge of guaranteeing the availability of the processor time as well as the availability of resources (full protection in time and in space domain). These challenges must be inherently incorporated into the RTOS architecture. The Avionics industry formulated these architectural requirements in the ARINC 653 specification to guide manufacturers of avionic application software towards maximum standardization.

2.5.1 Protection in Time Domain

Running multiple applications with different criticality levels on one processor may lead to no provision for guaranteeing processor time for critical applications. Consider the following scenario: Two applications of different criticality levels, each with one thread at the same priority run on a single system. Thread 1 is a non-critical thread whereas thread 2 is a critical one that needs at least 45% of the processor time to process its workload. As the two threads get assigned the same priority, a scheduler will assign each of the threads 50% of the processor time. In that case, the critical thread 2 will get its work done. Suppose that thread 1 spawns a new thread with the same priority. Then, the scheduler handles three threads at the same priority. As a consequence each of the threads will get only 33% of the processor time. Hence, the critical thread 2 is not able to handle its workload any more. The requirement of protection in time domain results clearly from this example.

2.5.2 Protection in Space Domain

Due to predictability reasons, many RTOS designers do not use virtual memory management. The fact that multiple applications with different criticality levels run on one single processor involves that processes share the same memory space. This implies that a process is able to corrupt the code, data or the stack of another process, intentionally or unintentionally. Furthermore, a process can also corrupt data or code of the operating system kernel which affects the safety and reliability of the system. In fact, it can lead to unexpected system behavior that infects the predictability and it can even bring down the entire system. Therefore, the protection of the memory is one key issue in RTOS for safety critical systems.

2.5.3 Secure Operating System Architecture

The answer to the requirement of protection is an architecture that defines a fully and securely partitioned real-time operating system. The partitioning is carried out also in two dimensions: Spatial Partitioning and Temporal Partitioning.

In particular, the basic design of such an operating system complies with the design of an ordinary RTOS. The fundamental difference is located above the operating system's core layer within the application layer which in fact is a construction of several separate partitions of the ordinary application layer (cf. Fig. 2.11). Each partition is assigned to an integrity level only allowing the running of applications compliant to this level. Furthermore, it consists of a small Partition Operating System that provides operating system services according to the safety features required by the safety integrity level. Further-

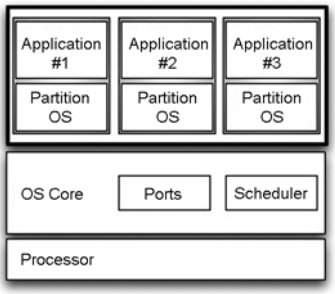

Figure 2.11. OS architecture for safety critical systems.

more, the Partition OS in fact runs the proper applications. The operating system core layer is responsible for the hardware-dependent functions, the device drivers, the scheduler, etc.

2.5.4 Providing Protection in Time Domain

The Scheduler implements temporal partitioning as it is responsible for assigning processor time to the partitions. Temporal partitioning requires an optimized two-level scheduler (cf. Fig. 2.12). The processor time for each partition is assigned statically. Within one scheduler period, also called major frame, each partition gets a guaranteed time window, a minor frame, to run its intrapartition processes. Within the minor frame, only the processes of the appendant partition can be executed. A partition is able to run more than one process. These processes have to be scheduled within the partition's processor

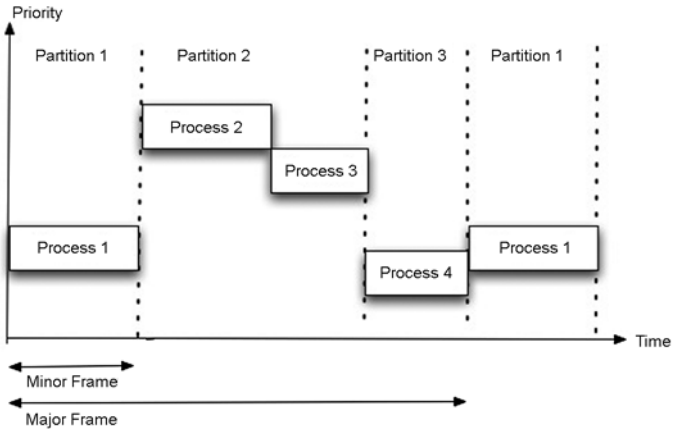

Figure 2.12. Two-level partition scheduler.

time frame. The partition remains the owner of the processor for the whole time frame, even if not all the processor time is needed for computation. Considering the given example above, a thread can only create a new thread within its own partition. Hence, the thread that creates the new thread has to share its time slice with the new thread without affecting the processor times of the other partitions.

2.5.5 Protection in Space Domain

To avoid corruption of the data of a safety-critical application individual address spaces for processes are essential. Spatial partitioning is implemented by assigning one fragment of the entire memory to each partition. The memory space can only be accessed by the processes of that partition. Such a fragmentation of the memory requires the support of an integrated Memory Management Unit (MMU). When the scheduler switches between the minor frames, a new set of logical addresses is assigned to the memory manager. Hence, each partition can only access the logical address space that is mapped by the MMU which makes careless malicious corruption across the processes of different criticality levels impossible.

2.6 Multi-Core Architectures

Multiprocessor architectures are an attempt to solve the lack of computational power in embedded systems by enabling computational concurrency. Using multiple lower-cost processors instead of cost-expensive high performance processors corresponds to the cost constraints of the embedded system market. However, multiprocessor architectures imply further challenges on the software and hence on operating systems that support these architectures [WJ04] .

Multiprocessor architectures consist of multiple processing entities (PE) connected via an interconnection network. Each one of the processing entities may represent a microprocessor (central processing unit—CPU); it also may constitute any other hardware component such as a controller, decoder etc. There are several approaches for interconnecting the PEs but the typical ones are: shared bus, crossbar and micro network (network on chip). Depending on the type of interconnection a system shows up different performance (communication collisions), costs (e.g. chip area) and reliability (e.g. single-point-of-failure).

The design of an operating system that is applied in multiprocessor systems is strongly dependent on the underlying system architecture. The software design process is strongly coupled or even an inherent part of the hardware design. Basically, the operating system architecture for multiprocessor systems extends the architecture of uniprocessor operating systems: like in uniproces-

sor operating systems it consists of the hardware abstraction layer and the core operating system. Furthermore, the operating system for multiprocessor application provides an inherent abstraction of the underlying system for the application. In fact, it abstracts design decisions of the multiprocessor architecture like:

- communication programming model: shared memory vs. distributed memory
- synchronous vs. asynchronous communication
- control strategy: centralized vs. decentralized
- redundancy mechanism
- hardware configuration
- topology: static vs. dynamic
- system architecture: homogeneity vs. heterogeneity

Beyond these design decisions that inherently incorporate into the RTOS implementation, resource management and scheduling, memory management, synchronization and interprocess communication (IPC) provide further challenges for a multi-core real time operating system.

2.6.1 Processor Management and Scheduling

The processor management and the scheduling policy strongly depend on the design decisions of control strategy and the architectural design. The initial problem of the processor management is the assignment of processes to different processors. In the case of centralized control, the scheduling algorithm deals with NP-completeness. In homogeneous systems, each process can be assigned to any processor in the system whereas in heterogeneous architectures specific tasks can only be executed on specialized task-specific/application-specific system components. This architectural decision in turn affects also the complexity of a feasibility analysis. Multiprocessor systems enable real concurrency and hence task-level parallelism. One challenge of the scheduling policy is to enable processes belonging to one single job and having strong interaction, cooperation and communication in-between these processes to be executed simultaneously. Task Concurrency Management (TCM) addresses the dynamic and concurrent task scheduling problem of multiprocessor real-time operating systems. It introduces a two-phase scheduling method: design-time scheduling and run-time scheduling. An application is represented by a set of concurrent thread frames (TF) that consist of many thread nodes (TN), which are independent sections of code belonging to a single thread of control, the thread frame. At design-time, the scheduling is applied on each identified TN and results in a set of possible solutions that include different mappings, orderings and, as two-phase scheduling is a cost-oriented approach

(cost-performance, energy-oriented etc.), performance measures. From these possible solutions the design-time scheduler generates a Pareto-optimal set. However, to guarantee hard real-time requirements the schedules generated by the design-time exploration rely on worst-case conditions. Instead of dealing with the complex problem of computing schedules at run-time, the run-time scheduler operates on the TFs by determining one configuration of the Pareto curve established at design-time. Such a exploration at design-time significantly reduces computational cost at run-time. Details of the TNs mapping are invisible for the run-time scheduler which furthermore reduces its complexity.

2.6.2 Memory Management

Programming parallel processing applications raises two main questions: how do processes on different processors share data and how do these processes coordinate themselves? The answer to these questions in the first instance depends on the memory organisation of the system. We talk about distributed memory management if the processors possess private memories and about shared memory in case of a single address space. In the case of distributed memory management, data sharing and process cooperation is realized via message-passing. In contrast to that, shared memory management offers processors one single address space to share and exchange data. Shared memory systems require synchronisation mechanisms to prevent interferences between processes while operating on shared data.

2.6.3 Synchronisation

Processors in multiprocessor real-time systems require knowledge about the overall system time/clock. Therefore, synchronization of the global system clock is essential to ensure time-dependent performance. Due to dependability reasons, distributed clock synchronization mechanisms are preferred for multiprocessor RTOS as they do not provide a single-point-of-failure. There exist some approaches to ensure the synchronization of the global system time like: Time-Triggered Protocol (TTP), TT-Ethernet and FlexRay (in the automotive industry) [Par07]. TTP is a protocol for fault-tolerant communication between distributed real-time systems. The synchronization of the clock is achieved in a masterless manner based on identifying time differences of arriving messages. To ensure a dependable communication, the communications controllers define exact time slices for sending and receiving per system node. The clock synchronization mechanism defined in FlexRay is similar to that one in TTP. Basically, the synchronization is processed by sending micro ticks between the processors. The main difference is that FlexRay enables the synchronization of heterogeneous processor clocks by identifying local deviations of receiving

micro ticks. Similar to TTP, the communication policy in FlexRay is implemented through predefined time slots.

RTEMS[1] is a known example for a multiprocessor real-time operating system. Furthermore, the automotive industry has defined OSEK-OS[2], a standard for operating systems designed to operate on the numerous controllers that are nowadays installed in cars.

2.7 Operating Systems for Wireless Sensor Networks

Given the recent advances in wireless sensor network (WSN) technology, it is possible to construct low-cost and low-power miniature sensor devices that can be spread across a geographical area in order to monitor their physical environment. Consisting of nodes equipped with a small processing unit, memory, a sensor, a battery and a wireless communication device, WSNs enable a myriad of applications ranging from human-embedded sensing to ocean data monitoring. Since each single node has only constrained processing and sensing capabilities, coordination among devices is necessary.

Due to their specific nature, sensor networks have different requirements compared to standard systems, such as self-configuration, energy-efficient operation, collaboration, in-network processing, as well as, a useful abstraction to the application developer. Given these requirements, a WSN OS must have a very small footprint and, at the same time, it must provide a limited number of common services for application developers, such as hardware management of sensors, radios, task coordination, power management, etc. (see [Sto05]). In the following section, we discuss some specific aspects relevant to OS for WSNs.

2.7.1 Aspects of Operating Systems for WSNs

We identify the following important aspects in WSNs:

Hardware Management. The OS should provide abstract services (e.g. for sensing and data delivery to neighbors). Given the lack of a memory management unit (MMU) in typical hardware, an OS library should implement this functionality (for more details see, e.g. [SRS+05]).

Task Coordination. There are two task coordination approaches:

- *Event-based Kernels*: Tasks are implemented as event handlers that run until completion. This enables concurrency without the need to elaborate mechanisms like per-thread stacks or mutual exclusion. The main advan-

[1] http://www.rtems.com
[2] http://www.osek-vdx.org

tage of this approach is its small memory footprint: because processes cannot block, just a global stack is necessary. However, a major problem occurring is the difficulty to implement applications with state-driven programming: the event-driven model is hard to manage by developers and not all problems are easily described as state machines. Further, interleaved concurrency is hard to realize in such systems.

- *Preemptive Thread Multitasking Kernels*: Preemption leads to the necessity of saving the current state of the registers to the stack. The necessity of one stack per thread leads to a relatively high memory footprint. Moreover, the context switch operation is rather time-consuming, i.e. for a task set composed mainly of IO-bound tasks or small tasks, the overhead caused by the context switch is relatively high. This problem can however be solved by assigning a static context to each process (as done e.g. in safety critical systems). In summary, given the resource constrained hardware of WSNs, the above points provide arguments against this OS paradigm. Nonetheless, preemptive multitasking supports the development of more complex, elaborate distributed applications and enables a straightforward porting of existing embedded applications.

WSNOS Architecture. Given the lack of MMUs in the typical WSN node hardware, the following OS architectures are predominantly employed (in contrast to e.g. monolithic kernel, microkernel or exokernel architectures in classical OS):

- *Library-based OS*: A set of functions implementing abstractions to facilitate the hardware management. Typically, it does not provide memory protection.
- *Component-based OS*: The OS consists of composable, self-contained components (also called "building blocks" or "modules"), which are, in contrast to library-based OS, interconnected via clear interfaces and interact with each other. They typically realize a well-defined function, such as the computation of a Cyclic Redundancy Check (CRC), and comprise code and state. Besides, the increased amount of modularity and configurability, this paradigm also suits the event-based programming approaches of WSNs. One example of a component-based OS is TinyOS, in which components are wired together explicitly using events for interaction (for details see [KW05]).

Often there are no clear borders between communication stack, OS services, and application. Cross-layer approaches are commonly used.

Power Management. Given the energy constraints of WSNs, different power management techniques have been developed (according to [DC05]):

- *Duty Cycling*: Reduces the average power utilization by cycling the power of a given subsystem.
- *Batching*: Amortizes the high cost of start-up by bundling several operations together and executing them in a burst.
- *Hierarchy Techniques*: Order the operations by their energy consumption and invoke the low-energy ones prior to the high-energy ones in a fashion similar to the short-circuit techniques used by several compilers for the evaluation of boolean expressions in various languages.
- *Redundancy reduction*: Using compression, aggregation or message suppression.

The low-power operation mode in WSNs can be addressed at various levels. In [DC05], the following levels have been recognized: sensing, communication, computation, storage, energy harvesting and reconfigurability support.

2.7.2 Examples of WSNOS

TinyOS. *TinyOS* [CHB+01] is a very efficient OS for WSNs that uses event-based task coordination in order to run on very resource-constrained nodes. The execution model is similar to a finite state machine. It consists of a set of components that are included in the applications when necessary. TinyOS addresses the main challenges of a sensor network: constrained resources, concurrent operations, robustness, and application requirement support.

Each TinyOS application consists of a scheduler and a graph of components. The components are described by their interface and internal implementation.

The concurrency model in TinyOS consists of a two-level scheduling hierarchy: events preempt tasks, but tasks do not preempt other tasks. Each task can issue commands or put other tasks to work. Events are initiated by hardware interrupts at the lowest levels. They travel from lower to higher levels and can signal events, call commands, or post tasks. Wherever a component cannot accomplish the work in a bounded amount of time, it should post a task to continue the work. This is because a non-blocking approach is implemented in TinyOS, where locks or synchronization variables do not exist. This means that components must terminate.

Mantis Operating System (MOS). The Mantis operating system (MOS) is a WSN OS designed to behave similarly to UNIX and provides a larger functionality than *TinyOS*. It is a lightweight and energy-efficient multithreaded OS for sensor nodes.

In contrast to TinyOS, the MANTIS kernel uses a priority-based thread scheduling with round-robin semantics within one priority level. To avoid race conditions within the kernel, binary and integer semaphores are supported.

The OS offers a multiprogramming model similar to that present in conventional OS, i.e., the OS complies with the traditional POSIX-based multithreading paradigm. All threads coexist in the same address space. The existence of multiple stacks (one per thread) makes MOS more resource-intensive than single-threaded OS (e.g. TinyOS).

The kernel of Mantis OS also provides device drivers and a network stack. The network stack is implemented using user-level threads and focuses on the efficient use of the limited memory.

Contiki. The *Contiki* [DGV04] operating system provides dynamic loading and unloading of programs and services during run-time. It also supports dynamic downloading of code enabling the software upgrade of already deployed nodes. All this functionality is offered at a moderate price: the system uses more memory than TinyOS but less than Mantis OS.

The main idea of *Contiki* is to combine the advantages of event-driven and preemptive multithreading in one system: the kernel of the system is event-driven, but applications desiring to use multithreading facilities can simply use an optional library module for that. A *Contiki* system is partitioned in core and loaded programs. This partition is determined at compilation time. The core comprises the kernel, program loader, run time libraries, and communication system.

2.8 Real-Time Requirements of Multimedia Application

The timing constraints for multimedia traffic originate from the requirement

- to maintain the same temporal relationship in the sequence of information on transmission from service provider to service requester
- from the necessity of preferably low offset delays between information departure and arrival
- the requisite to keep multiple types of media in sync

Consequently, each piece of information needs to be transmitted within a bound time frame and the traffic becomes real-time. Any failure to meet the timing constraints impairs the user-perceived Quality of Service (QoS) of networked multimedia applications. Different types of applications, however, have different QoS requirements. Common multimedia applications can be classified as *multimedia playback applications*, *streaming applications*, and *real-time interactive*.

- Multimedia playback applications transmit content that is pre-encoded and stored on a video server. A typical representative of this application is Video on Demand (VoD). As the video transmission is one-way and

does not involve conversational or low-latency bound elements, this type of application is tolerant to delays and delay variations.

- Streaming applications, as opposed to playback applications, require encoding video content on the fly as it is not available beforehand. This type of application does not involve conversational elements, but the latency of the transmission has a strong impact on the perceived user experience of the content. A typical candidate for a streaming application is Internet Protocol TV (IPTV) and presentable content covers live transmissions of sport events. Consequently, streaming applications have tighter requirements on delay bounds and delay variations.
- Real-time interactive applications exhibit the most challenging requirements with respect to delay and jitter. The interactive character of the applications requires conversational elements that often include speech and video. Video conferencing and interactive gaming are common representatives for this type of application. As the human perception is more sensitive to audio than it is to video it requires an undelayed synchronization between the two. Therefore, the delay bounds for this type of application are even more stringent than those for streaming applications.

QoS denotes a collective assemblage of components that (1) transform the qualitative set of user and application requirements into quantifiable performance metrics for resource allocation and (2) enforce them along the network path between a service requester and a service provider [Dit08]. Common performance metrics include the network bandwidth and acceptable bounds for packet loss, delay, and jitter. The key to QoS enforcement is to *differentiate* traffic into *isolated* transmission queues and *provide resources* on a per-flow (Intserv) or per-class (Diffserv) basis. This is accomplished by QoS traffic control and its approaches for call admission control (CAC), traffic classification, traffic shaping & policing, packet queueing, and packet scheduling. The cohesions of the individual approaches are depicted in Fig. 2.13.

The purpose of CAC is to protect traffic in a shared network by determining if an additional traffic flow's request for resources can be approved without causing interference to the resource allocation of admitted flows. It relies on a flow's traffic characterization that describes its performance metrics. A traffic classifier investigates packets for their priority level and forwards them to respective transmission queues that implement the traffic differentiation and isolation. Traffic policing and shaping ensure that flows conform to their traffic characterization, thereby defending the network against unexpected traffic bursts. The differentiated transmission queues are served by a scheduler according to a predefined scheduling policy that ensures the resource enforcement.

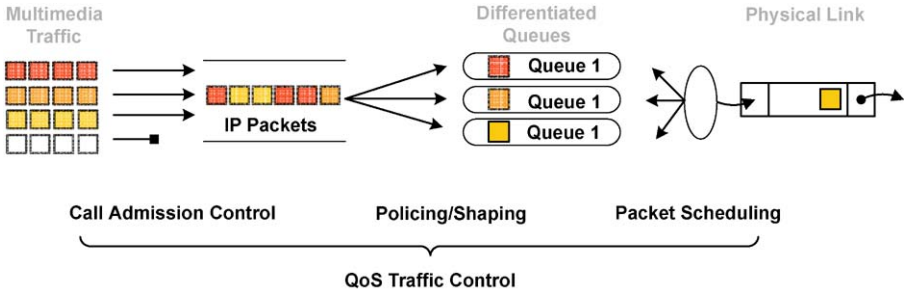

Figure 2.13. QoS Traffic Control Approaches.

The transmission of multimedia traffic across wireless networks imposes new and unique design challenges to the QoS traffic control and requires leveraging of the interaction among individual QoS approaches (1) toward lower layers of the communication model to optimize the resource provisioning of the scarce network resources and (2) in direction of the higher layers to perform content adaptation with different levels of granularity. The new challenges trace back to imperfect wireless transmission channels and the highly fluctuating traffic loads of multimedia applications. They are addressed by unidirectional cross-layer management approaches which can be partitioned into cross-layer optimization and cross-layer adaptation. Cross-layer optimization targets to improve the utilization or throughput of multimedia traffic in wireless networks by exploiting the time-varying channel characteristics. Cross-layer adaptation adjusts the content quality in respect to the traffic load toward the higher layers of the communication model. Popular approaches for multiple layer adaptation are Joint Source Channel Coding (JSCC) concepts [KYF+05].

2.9 Conclusions

Embedded applications in most cases are bound to real-time constraints and are usually executed on top of a Real-time Operating System (RTOS). Real-time tasks have to be annotated with basic timing information in order to enable the underlying RTOS to manage them properly. Such parameters include arrival time, worst case execution time (WCET) and (relative or absolute) deadline, just to mention the most important ones. Explicitly providing these information distinct real-time applications from ordinary ones where such information (usually characterized as non-functional properties) is available only in implicit manner. Having such characteristics in hand, specific scheduling algorithms can be designed. Most real-time applications show periodic behavior. When possible, static cyclic schedules are calculated off-line. If more flexibility is needed on-line techniques are applied. These algorithms are bound to priorities which can be assigned statically as in the case of Rate

Monotonic (RM) or Deadline Monotonic (DM) priority assignment, or dynamically as in the case of Earliest Deadline First (EDF). The latter one can be applied to a-periodic tasks as well. Task sets that consist of both periodic tasks and a-periodic ones, are more complicated to handle. An approach for a unified management of such situations is the introduction of so called servers. A server in this context is a periodic task that offers its processor utilization for executing a-periodic tasks.

Designing a proper RTOS architecture needs some delicate decisions. The basic services like process management, inter-process communication, interrupt handling, or process synchronization have to be provided in an efficient manner making use of a very restricted resource budget. Various techniques like library-based approaches, monolithic kernels, microkernels, or virtual machines/exokernels have been developed, each of them dedicated to specific demands. The classical approach is given by monolithic kernels. They allow efficient handling of service requests. Microkernels export as many services as possible into user space, thus reducing the risk of kernel corruption. Library-based approaches are more or less kernel-less. They can be adapted precisely to the needs of applications to be supported. Recently exokernels did gain interest. They support safety requirements in an elegant manner based on their virtualization technique.

Safety critical application can be supported by separation of applications either in the time or the space domain. Dedicated RTOS architectures preferably follow the concept of virtual machines/exokernels. By providing separated address spaces (space domain) or strictly separated time frames in scheduling (time domain) the mutual influence of tasks is substantially reduced. Multicore architectures need special techniques for process management, memory management, and synchronization. Especially scheduling needs consideration as most of the classical RT scheduling methods are proven to be optimal only for mono-processor systems. An excellent fundamental architecture for distributed real-time systems is provided by time-triggered architectures, making use of time-triggered communication protocols.

The upcoming Wireless Sensor Networks (WSN) generate special demands for RTOS support leading to dedicated solutions. The nodes of a WSN are equipped with extremely restricted resources. Due to power constraints they have to be inactive for a large fraction of time. This implies special demands concerning communication and synchronization. As a consequence of these special requirements dedicated RTOS concepts have been developed. Strictly even-based approaches (e.g. UCB's TinyOS) may serve as an example. However, a tendency towards more standard multi-threading execution models can be observed. Another special area is given by multimedia applications. Very high data rates under (soft) RT constraints have to be supported. Based on the used encoding techniques (e.g. MPEG) dedicated solutions can be created.

In such solutions the frames within an MPEG Group of Pictures (GoP) can be scheduled in such a way that the number of frames to be dropped can be reduced.

The RTOS layer in an embedded system provides interesting glue between the underlying HW and the applications to be executed. Designing a fully predictable service provider in a highly efficient manner and at the same time making use of minimal resources is really challenging. This challenge is still open despite the fact that impressive solutions have been found by the RT community.

References

[BP84] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series)*. Springer-Verlag Telos, Santa Clara, 2004.

[But05] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[CHB+01] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 114–130. Springer, Berlin, 2001.

[DC05] P.K. Dutta and D.E. Culler. System software techniques for low-power operation in wireless sensor networks. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, pages 925–932. IEEE Computer Society, Washington, 2005.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462. IEEE Computer Society, Washington, 2004.

[Dij68] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[Dit99] Carsten Ditze. *Towards Operating System Synthesis*. Phd thesis, Department of Computer Science, Paderborn University, Paderborn, Germany, 1999.

[Dit08] M. Ditze. *Coordinated Cross-Layer Management of QoS Capabilities for Transmitting Multimedia Traffic across Wireless IEEE 802.11 Networks*. To be published as University of Paderborn PhD Thesis, 2008.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic, Norwell, 1997.

[KW05] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, New York, 2005.

[KYF+05] A. Katsaggelos, Y. Eisenberg, F. Zhai, R. Berry, and T. Pappas. Advances in efficient resource allocation for packet-based real-time video transmission. *Proc. IEEE*, 93(1):288–299, 2005.

[OW02] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–64. ACM, New York, 2002.

[Par07] Dominique Paret. *Multiplexed Networks for Embedded Systems*. Wiley, New York, 2007.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, San Mateo, 2005.

[SRS+05] Brian Shucker, Jeff Rose, Anmol Sheth, James Carlson, Shah Bhatti, Hui Daia, Jing Deng, and Richard Han. Embedded operating systems for wireless microsensor nodes. In *Handbook of Sensor Network: Algorithms and Architectures*. Wiley, New York, 2005.

[Sta01] William Stallings. *Operating Systems*. Prentice Hall, Upper Saddle River, 2001.

[Sto05] Ivan Stojmenovic, editor. *Handbook of Sensor Networks: Algorithms and Architectures*. Wiley, New York, 2005.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, 2001.

[THB06] A.S. Tanenbaum, J.N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[WJ04] Wayne Wolf and Ahmed Jerraya. *Multiprocessor Systems-On-Chips*. Morgan Kaufmann, San Mateo, 2004.

Chapter 3

UEFI: FROM RESET VECTOR TO OPERATING SYSTEM

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.—T.S. Eliot

Vincent Zimmer, Michael Rothman and Robert Hale

Abstract In PCs, the firmware that sits at the reset vector is called a BIOS. The BIOS has increased in size, complexity, and extensions apace with the complexity and richness of PCs. The increases have finally reached the point that no amount of patching will fix the old architecture. The new architecture, known as the Unified Extensible Firmware Interface (UEFI) [UEFc, ZRH] and Platform Initialization (PI) [UEFb] keep the learning's of the last years but impose a modern software engineering structure that supports the basic requirements of system initialization, configuration, and abstraction of boot devices, but which is also designed to be extensible enough to address the new features of hardware to come.

Keywords: BIOS, Firmware, UEFI, EFI, Boot ROM

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

3.1 Introduction

Time was when computers were humans who did computations. Then computers became machines that filled rooms, requiring constant attention by specially trained operators. Automation all but completely replaced the human computers.

The machines in those early days were finicky and crashed often. Rebooting was a time consuming and complex process requiring the operator to flick switches and load tapes in a complex order. An untold number of discoveries, and an even larger number of hours of effort later, computers are personal, ubiquitous and, at least relatively, easy to use.

As a part of that evolution, automation has all but completely replaced the human operators. Untrained end-users now seem to do what operators trained for years to do in the past, and on systems that are arguably vastly more complex. The simple boot ROM, auto-loader, or, in the PC world, BIOS, has gone a long way to replace the operator, and, in a real sense, carried the PC revolution on its back.

Boot ROMs were at first quite small and simple, if for no other reason because ROMs were quite small and expensive. Minicomputers in the early 1980s still booted from 512 byte ROMs. In 2008 the average BIOS ROM is 512 K bytes or 1024 times as large as its minicomputer counterpart. The first BIOS for the IBM* Personal Computer was 8 K bytes. Servers can use BIOS of over well over 4 MB, four times the size of the PC's entire address space.

This chapter is about what has prompted the growth in BIOS. It is also about the next generation of BIOS, how it is architected, and some of the subtle capabilities it provides. With the specifications now well over 2000 pages in length, it cannot do so in any great detail. Instead, we try to give some idea of the underlying motivations, design decisions, and possibilities inherent in this fundamental part of modern computers.

3.2 The Ever Growing Ever Changing BIOS

The system BIOS (the BIOS on the motherboard) was initially divided into two parts, POST ("for Power On Self Test") and Run-time. POST gains control via the reset vector and is responsible for the initialization, testing, and configuration of on-board devices.

The problem of 'drivers' for add-in cards was solved most elegantly by providing 'Option ROMs', pieces of code that allow the BIOS to access the cards' devices. These option ROMs were non-volatile memory components which typically resided on the hardware of the add-in card itself. In implementation, however, Option ROMs were always second class citizens, with difficulty getting control after initialization and even accomplishing fundamental tasks like allocating memory.

The final function for POST is to load the initial bootstrap loader for the Operating System and hand control to it. The intent was then that the OS would use the BIOS as an abstraction for accessing on-board devices using the BIOS run-time interfaces.

The BIOS design was quite successful in its initialization role. The run-time abstractions were at a low enough level that they proved to be only really useful so the OS could access boot devices (video, keyboard, hard drive) before it had loaded its own drivers.

Over the years, requirements changed, initially prompting a set of marginally related BIOS extensions and finally prompting the definition of a completely new architecture.

3.2.1 The Evolution of the Power-On Self-Test

POST's goal is to provide the OS with a functioning fully configured platform. That is certainly not what the BIOS finds at the reset vector. The BIOS typically finds little other than the basic processor and ability to access its own memory functional. The rest must be initialized.

System initialization comprises three basic parts: finding the devices, configuring the devices, and testing the devices. BIOS devices traditionally cover a different set of components than an operating system: the chip that generates the frequencies used on the rest of the motherboard is a device to a BIOS but is typically invisible to an OS.

In the early days of BIOS, finding devices (enumeration) was not supported since it required additional hardware support that was too expensive at the time. As the hardware prices have decreased and end user support costs increased, device enumeration became more important. Modern buses including PCI, PCIe, ATA, SCSI, and USB have mechanisms which BIOSs use to locate and configure devices.

The BIOS is responsible for configuring the chips in the chipset. Chips are designed for basic functionality at reset. They require configuration to adapt them to each other, the board, and add-in devices and, in some cases, to work around bugs in the Silicon. Given the extremely high cost of hardware changes, BIOS workarounds are strongly preferred. This requirement, as much as any, motivated the BIOS to move from ROMs to Flash. Flash can be reprogrammed in the field, reducing recalls and improving support.

Given the high integration of modern PCs the utility of most BIOS testing has disappeared: The fact that the BIOS runs proves that most of the motherboard is working. It is more common to test connections to peripheral devices. Oddly, RAM is treated as a peripheral until it is initialized and tested.

The most complex and ever changing enumeration requirement is RAM. In early days, RAM initialization could occur within the first several hundred in-

structions and typically took less than a page of assembly code. Today's RAM must be located and its characteristics discovered (using the System management bus (SMBUS) [SMBUS]) and described to the chipset using algorithms that require thousands of lines of C code. This added complexity does allow for use of faster memory and adapting memory speeds to board and chipset.

The complexity of the BIOS routines stems from the nature of the hardware/chipset. For example, fast paged-mode Dynamic Random Access Memory (DRAM) in the mid-90's was initialized with maybe 50 lines of code. The algorithm was simple: a table had the five or six settings of the memory controller, the BIOS would attempt each setting and test to see if it "worked" (e.g., no aliasing, bits written could be read back, etc.). A modern memory controller for double data-rate DRAM, such as DDR3, may take several thousand lines of code to read serial eeprom data from the memory part, train the analog channel/compensation, and finally program the memory. So BIOS typically tracks the complexity of the hardware. And add to this complexity comprehending the requirements of various vendors who alternately provide the chipset, central processing unit (CPU), system board, clock generator chip, DRAM modules, SMBUS controllers, etc.

As time has passed, the BIOS has taken on other requirements. The BIOS is now responsible for describing the board it resides on to the operating system and applications that manage the system. Items described include board type, asset numbers, and how the board's power management features are to be accessed.

3.2.2 Run Time Evolution

The initial interfaces which abstracted peripherals were creatures of their time. They were real mode (8 or 16 bit). They mirrored the devices of their day: the maximum address space supported on a hard drive was 540 MB and the maximum memory space supported was one megabyte. In the intervening years, the industry has adopted a series of specifications, some clean and elegant, some not so, to extend the interfaces where required while retaining backwards compatibility.

3.2.3 Software Engineering

The remarkable thing about most BIOS is the amount of code that is reused. BIOS developers routinely expect to reuse over 95% of the code from platform to platform. This is a result of careful design and diligent monitoring as well as schedules that permit no alternative. A byproduct of this reuse has been a high degree of consistency and compatibility. Code is written once, tested once, and reused millions of times. Code that is known to change (chipset, processor) is isolated and cocooned with interfaces.

Traditionally, BIOS have been developed in assembly language and managed using commonly available source code control systems. Custom tools are used to manipulate the results to fit in the parts.

The BIOS run-time, while limited and archaic has been consistent enough that many generations of operating systems have booted and continue to boot on what are essentially extensions to the same interfaces. It thus may come as a surprise that there is no consistency internally between different vendors BIOS. The internal structures and architecture are almost completely incompatible. It is impossible for a company to provide a single set of code that works unmodified when integrated into all system BIOS.

3.3 Time for a Change

By around 2000, it became clear to many in the industry that the interfaces that had served us well for 20 years had become obsolete. Thirty two bit operating systems were hobbled by booting using 8 bit interfaces; 64 bit operating systems would simply not work. There was simply no way to fit the required code in the single 512 byte boot sector BIOS allowed for the first stage OS boot loader.

Again, the industry could have defined yet another patch on the existing system to allow for extensions to the 512 bytes and have lived for another year or two. But then the 64 bit operating systems would be using 8 bit interfaces to load subsequent boot stages. It became clear that something more modern was required. Existing alternatives were examined and, one by one, rejected.

A modern set of interfaces had to be defined that met the needs of the Operating System community and the system developers. There was also substantial agreement that the never quite solved problems of the option ROM vendors should be resolved by this new solution.

3.3.1 EFI and UEFI

The new set of interfaces was known as the *Extensible Firmware Interface* or EFI for short. The U (for *Unified*) was added a few years later, when an industry forum took over ownership of the specification.

The fundamental structures in EFI define extensibility, acknowledging that technology will evolve. Software engineering advances in the intervening 20 years were embraced by creating what amounts to an object architecture. This architecture was designed to be usable by all classes of systems from deeply embedded/handheld platforms to mission-critical, large, scalable servers. Basic services such as memory allocation and resource management were made a part of the core set of services. The fundamentals have the feel of an embedded non-preemptive real-time system.

Traditional interfaces, such as those for various types of peripherals, were defined using the extensible interfaces as will be defined in future years for new devices not yet thought of. Importantly, this enables option ROMs to become full members of the system. EFI embraces the idea of driver-like interfaces that exist only during boot and a much more minimal 'run-time' set of calls.

A new disk partitioning methodology was also defined which allows for greatly expanded number of larger sized partitions than what had previously been available.

The interfaces have been extended to support more devices (iSCSI for example) and more advanced features such as the Human Interface Infrastructure (HII), which supports mechanisms to support user and remote configuration of system devices with all of the localization and similar support expected of a modern system.

Backwards compatibility is a hard thing to grow out of. We do not expect to see the last of the old "legacy" BIOS interfaces disappear for many years. We are now on our way with UEFI.

3.4 UEFI and Standardization of BIOS

The BIOS evolved from crisis to crisis with small groups forming to address a need and driving industry adoption. EFI has been more encompassing and specification driven from the start. To continue that model, and to achieve input and buy-in from the industry, the Unified EFI Forum ("UEFI") was created in 2005. This organization now owns the UEFI specification, covering the interfaces between Boot Firmware, OS, and Option ROMs, and the Platform Initialization specifications, covering common interfaces between the reset vector and UEFI.

At the highest level, the UEFI Specification covers all the data one might expect to be described for launching a boot target. However, when digging into a little more detail, a reader quickly realizes that the UEFI specification covers many concepts that one might expect in a modern operating system. Ultimately, the intent of the UEFI specification is to address the issues in the pre-operating system environment that are known today, but also to provide sufficient extensibility to the described infrastructure so that the underlying architecture should be able to easily adapt to changing technology.

3.4.1 Providing Interface Extensibility

In the UEFI programming environment, the interfaces which a firmware component (i.e. UEFI Driver) would provide are commonly known as "protocols". The protocols describe the parameters and data which are exchanged when programmatically interacting with a UEFI driver. To ensure that there is a uniform interpretation of these interfaces, the specification clearly defines a

contract which associates a 128-bit globally unique identifier (GUID) with the described interface description. When bound together, a GUID and an interface (itself nothing more than a something like a C struct) form a protocol.

An example of installing an instance of the EFI_SERIAL_IO_PROTOCOL can be shown in the following example from the ISA serial driver in the open source EFI Developer Kit (EDK) [EDK]:

```
#define EFI_SERIAL_IO_PROTOCOL_GUID\
{0xBB25CF6F, 0xF1D4, 0x11D2, 0x9A, 0x0C, 0x00, 0x90, 0x27,
 0x3F, 0xC1, 0xFD}

EFI_GUID gEfiSerialIoProtocolGuid =
EFI_SERIAL_IO_PROTOCOL_GUID;
typedef struct _EFI_SERIAL_IO_PROTOCOL {
    UINT32 Revision;
    EFI_SERIAL_RESET Reset;
    EFI_SERIAL_SET_ATTRIBUTES SetAttributes;
    EFI_SERIAL_SET_CONTROL_BITS SetControl;
    EFI_SERIAL_GET_CONTROL_BITS GetControl;
    EFI_SERIAL_WRITE Write;
    EFI_SERIAL_READ Read;
    EFI_SERIAL_IO_MODE *Mode;
} EFI_SERIAL_IO_PROTOCOL;

EFI_STATUS EFIAPI SerialControllerDriverStart (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE Controller,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath
)
// Routine Description:
// Start to management the controller passed in Arguments:
//    This - pointer to the EFI_DRIVER_BINDING_PROTOCOL
//           instance.
//    Controller - handle of the controller to test.
//    RemainingDevicePath - pointer to the remaining portion
//                          of a device path.
// Returns: EFI_SUCCESS - Driver is started successfully
{
EFI_STATUS Status;
EFI_INTERFACE_DEFINITION_FOR_ISA_IO *IsaIo;
SERIAL_DEV *SerialDevice;
UINTN Index;
UART_DEVICE_PATH Node;
EFI_DEVICE_PATH_PROTOCOL *ParentDevicePath;
EFI_OPEN_PROTOCOL_INFORMATION_ENTRY *OpenInfoBuffer;
UINTN EntryCount;
EFI_SERIAL_IO_PROTOCOL *SerialIo;
SerialDevice = NULL;
```

```
43 // Get the Parent Device Path
44 Status = gBS->OpenProtocol (
45     Controller,
46     &gEfiDevicePathProtocolGuid,
47     (VOID **) &ParentDevicePath,
48     This->DriverBindingHandle,
49     Controller,
50     EFI_OPEN_PROTOCOL_BY_DRIVER
51 );
52 if (EFI_ERROR (Status) && Status != EFI_ALREADY_STARTED) {
53     return Status;
54 }
55
56 // Grab the IO abstraction we need to get any work done
57 Status = gBS->OpenProtocol (
58     Controller,
59     EFI_ISA_IO_PROTOCOL_VERSION,
60     (VOID **) &IsaIo,
61     This->DriverBindingHandle,
62     Controller,
63     EFI_OPEN_PROTOCOL_BY_DRIVER
64 );
65 if (EFI_ERROR (Status) && Status != EFI_ALREADY_STARTED) {
66     goto Error;
67 }
68
69 // Issue a reset to initialize the COM port
70 Status = SerialDevice->SerialIo.Reset(&SerialDevice->SerialIo);
71
72 // Install protocol interfaces for the serial device.
73 Status = gBS->InstallMultipleProtocolInterfaces (
74     &SerialDevice->Handle,
75     &gEfiDevicePathProtocolGuid,
76     SerialDevice->DevicePath,
77     &gEfiSerialIoProtocolGuid,
78     &SerialDevice->SerialIo,
79     NULL
80 );
```

Since the underlying UEFI environment is designed for extensibility, this means that UEFI provides mechanisms for agents to advertise UEFI protocols for use by others. For instance, participation in this level of interaction is not limited to components which were shipped with a platform. It is fully expected that add-in devices (e.g. RAID, network, etc.) when plugged into the system will have the capacity to export their own interfaces which the firmware can use.

Figure 3.1 shows a timeline of events where the UEFI subsystem will discover and launch drivers regardless of if the driver was present during platform

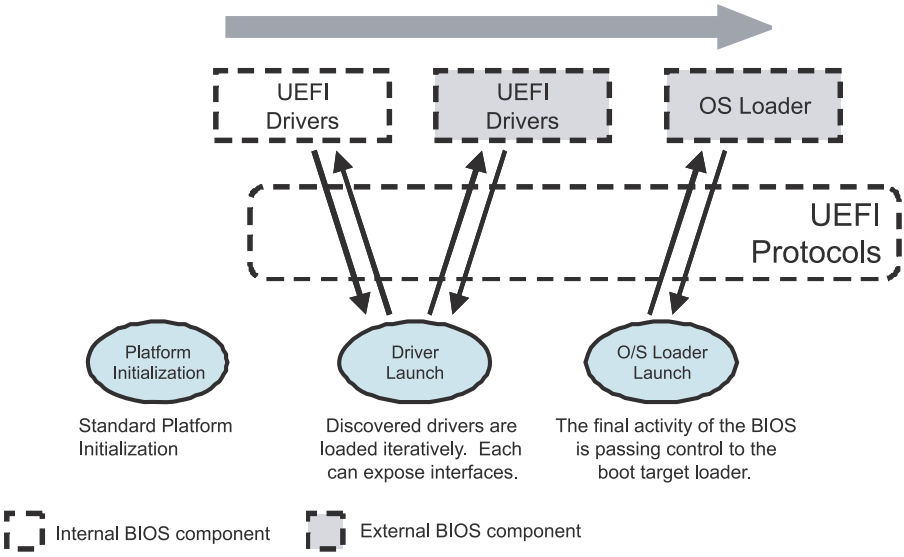

Figure 3.1. BIOS component interaction.

construction or not. One thing to note which is somewhat surprising to many is that even the operating system (O/S) loader (which is provided by the O/S) is a UEFI-compliant driver which directly uses the underlying UEFI protocols.

The drivers involved in the boot process need to ensure a substantial level of cooperation since a platform may find itself launching an add-in device's driver which exposes Block I/O abstractions to storage media. The platform may then discover an O/S vendor's O/S loader, which in-turn will use a variety of other exposed BIOS abstractions. These standard interfaces make it possible for a large contingent of BIOS components to be constructed and run in a portable fashion on various UEFI codebases.

3.4.2 The Boot Processing Logic

The UEFI specification defines a variety of Non-volatile random access memory (NVRAM) variables which are intended to indicate platform policy. These variables encompass various configuration related settings (e.g. current language to use, current console to use, etc.) as well as variables associated with what drivers to load during the initialization process.

Table 3.1 enumerates the variables commonly associated with the boot process. The listed variables can be placed into two subcategories, one which is related to being a final boot target, and the other being an auxiliary driver which is not intended to be a final boot target. The primary distinction is that

Boot####	A boot load option
BootOrder	An ordered boot load option list
BootNext	The boot option for the next boot only
Driver####	The driver load option
DriverOrder	An ordered driver load option list

Table 3.1. Variables associated with boot processing.

the Boot* oriented variables are anticipated to be the final item(s) launched by the underlying BIOS.

It should be noted that some of the variables above have some #### notation included in their name. The #### represents a unique number in printable hexadecimal representation using the digits 0–9, and the upper case versions of the characters A-F. The #### will always be four digits so small numbers will use leading zeros.

Both the Boot#### and Driver#### variables contain data which relates to "where" the driver is located. The location associated with these drivers is described using something known as a device path.

A device path is a means of describing a programmatic path to a particular device. With the aforementioned boot processing variables, sufficient information can be understood from the variable content so that all the enumerable buses can be discerned and the location of the driver can be determined.

When an operating system is installed, one of the normal processes it undertakes would be for it to add a reference to its O/S loader as a Boot#### variable. The BootOrder determines which Boot#### variables are to be executed and in what order, and BootNext is used when across the next platform reset and only on the next platform reset a particular driver needs to get launched first.

3.4.3 The UEFI System Partition

Since the O/S loader is typically a UEFI compatible driver, the ability for the underlying UEFI infrastructure to find the O/S loader is required. UEFI codebases are required to have the ability to interpret certain basic file systems (e.g. FAT32), but most modern operating systems have evolved to using other file systems which the UEFI subsystem may not be able to interpret. UEFI defines the concept of a system partition which can be used by vendors to store UEFI drivers.

In Fig. 3.2, we illustrate how a platform partition may be dedicated as a UEFI system partition, including the Logical Block Addresses (LBA) of the disk media. In addition, this partition is required to have been formatted using the FAT (File-Allocation-Table) file system so that items stored in this repository can easily be retrieved by the UEFI BIOS. Platforms with UEFI-based

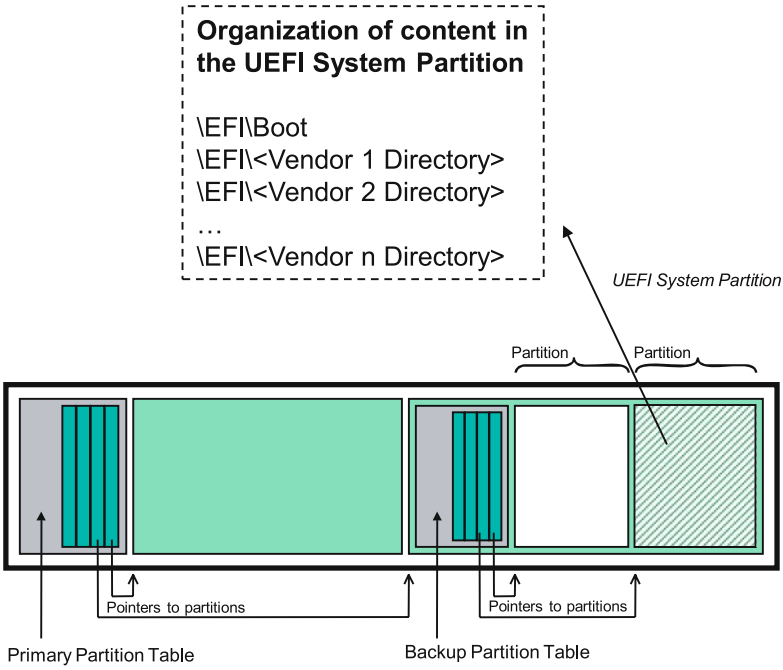

Figure 3.2. GUID Partition Table Scheme.

BIOS must support FAT12, FAT16, and FAT32 variants of the FAT file system. What variant the system partition is formatted with is defined by the size of the partition itself.

One very common usage of the system partition is for the O/S to store its O/S loader; however it is very reasonable to envision a usage where various other UEFI compatible platform utilities are placed in this same area. With various different parties vying for the usage of this common repository, it was envisioned that there might be file naming conflicts. To try to address this situation as much as possible, a registry was constructed [UEFa] so that different vendors could place material on the system partition without as much of a concern about file name collisions.

3.4.4 Advances in Configuration Infrastructure

The ability to advance interoperability of various BIOS components were not limited solely to items which are largely invisible to the end-user. Even though BIOS has largely been a user-invisible technology, which in other words could be phrased, “In almost all situations, BIOS should not be noticed by the user aside from a possible splash screen”, the expansion of the capabilities associated with the BIOS as well as a robust programming environment has

anticipated that user-visible solutions based on the underlying BIOS would come about. These solutions would encompass solutions that are pretty standard operations such as configuration of both platform and add-in devices, as well as possible other solutions which might provide user interfaces.

To facilitate the acceptance of these solutions, several areas were addressed to simplify shipping BIOS solutions in a global market. One area had to do with localization of text. The configuration infrastructure that was put in place into the most recent versions of UEFI has the support for string tokenization. This made it so that drivers could much more easily support multiple languages in a given string reference. In lieu of hard-coded references to strings a program would now simply be able to reference a string via its string number. The simplicity in this enables that there is no special software that needs to be written by users of UEFI systems to support multiple languages. As the example in Fig. 3.3 illustrates, one simply references String #4, and based on the current language setting of the UEFI system, the appropriate language would be retrieved.

String ID #4	String Representation	H	E	L	L	O		W	O	R	L	D	
	Unicode Encoding	0x0048	0x0045	0x004C	0x004C	0x004F	0x0020	0x0057	0x004F	0x0052	0x004C	0x0044	0x0000

String ID #4	String Representation	H	O	L	A		M	U	N	D	O	
	Unicode Encoding	0x0048	0x004F	0x004C	0x0041	0x0020	0x004D	0x0055	0x004E	0x0044	0x004F	0x0000

String ID #4	String Representation	你	好	世	界	
	Unicode Encoding	0x4F60	0x597D	0x4E16	0x754C	0x0000

Figure 3.3. Example of string tokenization.

In addition to strings, the ability for a platform to display characters has always been fixed by the platform vendor. This posed issues with the ability for third party vendors to provide strings which might not be displayable by the platform being executed on. The UEFI configuration infrastructure introduces a means by which character glyphs can be introduced by third parties so that these glyphs can be used to help render what would previously had been undisplayable strings.

The interaction between the BIOS and the add-in devices has always been a black box operation in that there was no programmatic interaction between the components. There was also no way for the BIOS to discern any information from the device aside from what was described by the bus that the device was plugged into. In UEFI, the configuration infrastructure enables devices to provide configuration access protocols which can facilitate a variety of interaction that was formerly impossible. In addition, since this infrastructure also demands that configurable devices contribute their content (e.g. strings, forms, etc.) in a standard form, the BIOS can now proxy user interface functional-

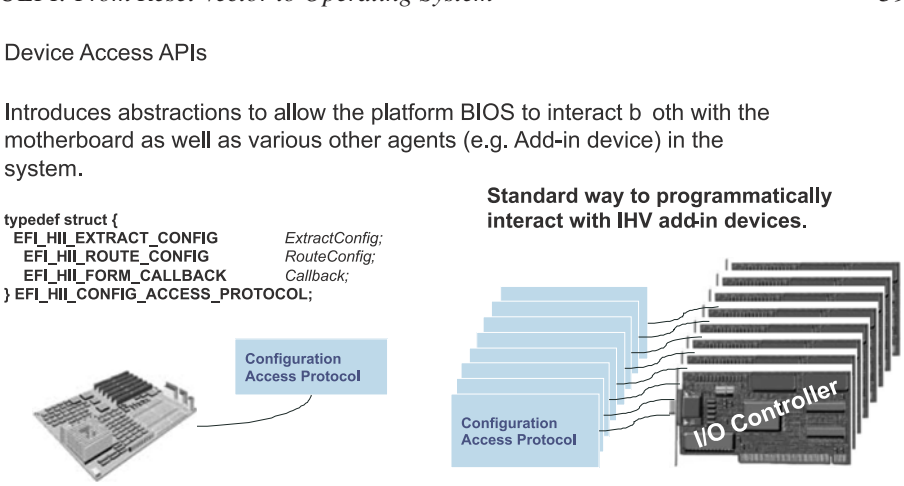

Figure 3.4. Add-in devices can now be programmatically configured.

ity for the device as well as make the content portable so that it can be used remotely, locally, or even in the O/S.

Figure 3.4 describes the EFI_HII_CONFIG_ACCESS_PROTOCOL published by each configurable I/O controller driver. Given these protocols, a single platform device manager can interact with each device and provide a consistent user interface (or remote access) to the devices.

3.5 Framework, Foundation, and Platform Initialization

EFI solved the more visible issues in the firmware. The issues of interoperability inside the BIOS architecture were no less profound but much more isolated.

Intel took the lead in defining what it hoped to be a unified architecture which is now owned by the UEFI Forum and known as the Platform Initialization (PI) specification. As its basis, PI uses the same structures and core services as found in EFI. The architecture was then defined backwards from the Operating System towards the reset vector. Phases were defined to own the reset vector, manage the system up to the point RAM was initialized, RAM-resident initialization, boot device selection, and the run-time.

To address the transition from legacy to EFI, the PI can support multiple boot modes, including a module known as the Compatibility Support Module (CSM), which allows PI to boot into legacy Operating Systems using those same interfaces defined in 1980.

3.5.1 Platform Initialization Versus UEFI

It is the purview of the UEFI Platform Initialization, or what we shall refer to as the "PI", to describe these building blocks. To that end, we will describe

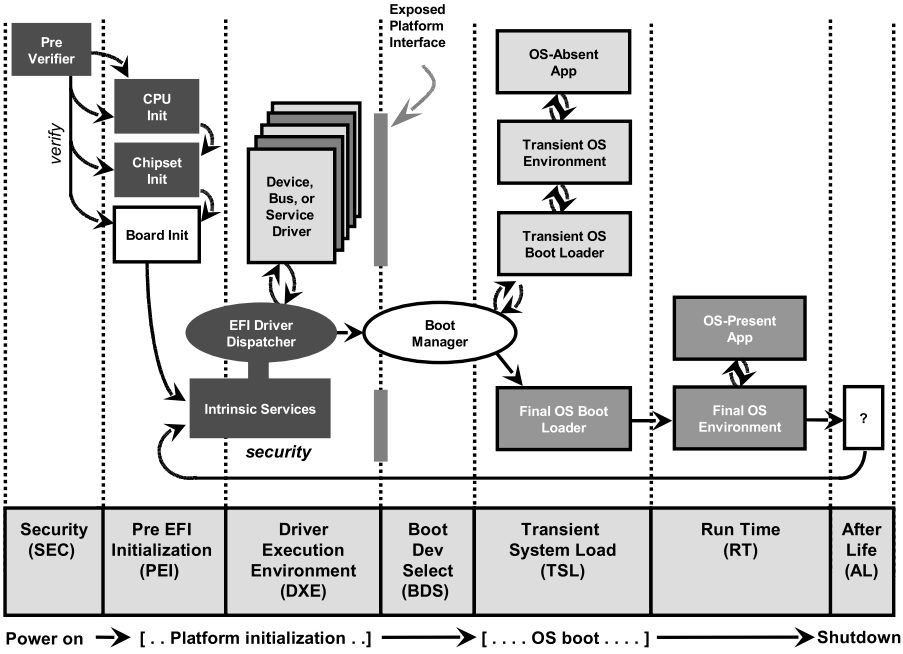

Figure 3.5. Temporal view of the system.

both a temporal and spatial view of the system. The temporal view of the PI boot is shown in Fig. 3.5. The spatial view is shown in Fig. 3.6.

What is of interest in the temporal view is that the boot flow of the machine is broken up into phases, of which there are several of interest, including the security (SEC) phase , the Pre-EFI Initialization (PEI), and the Driver Execution Environment (DXE).

3.5.2 SEC: The Security Phase

On common aspects of all processors and platforms is that they restart in a given fashion. On x86, the location is 4G – 16 bytes, for example. At this point, there is typically no initialized memory and a very rudimentary machine state. The most notable feature is the lack of memory for a heap and call-stack.

The SEC phase stands for “Security”. The intent of this moniker was to describe the first location in PI where a system root of trust in BIOS could be implemented, although without platform hardware enhancements, this is not the case in most SEC construction today.

For PI, the SEC is responsible for preparing to invoke the PEI environment & the creation of temporary memory. On Intel-based systems, in order to avoid the cost of custom SRAM or other early memory store, we configure the

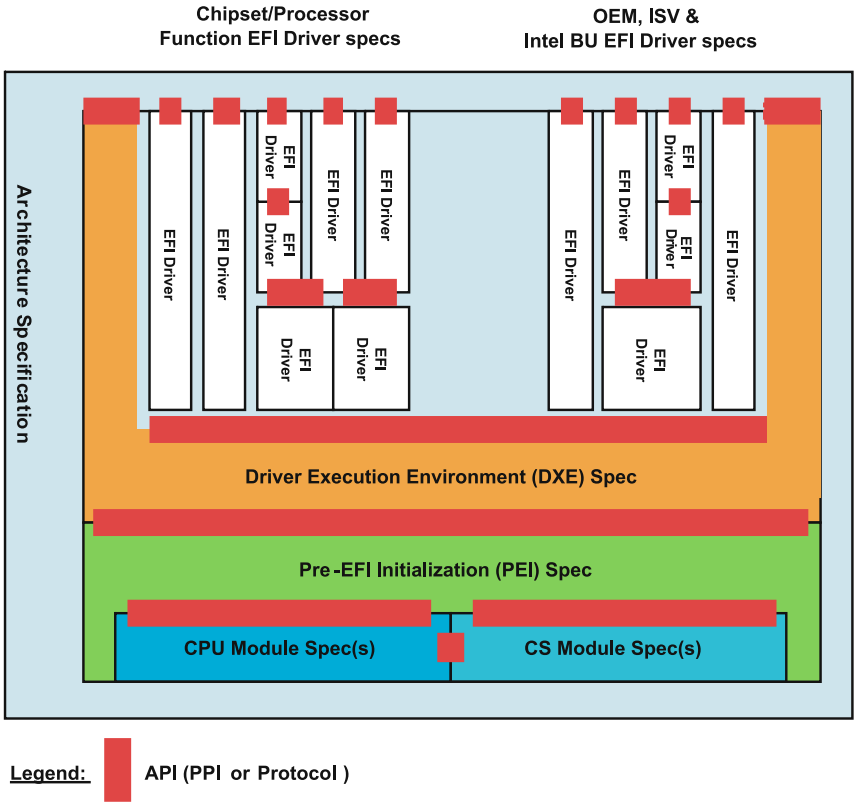

Figure 3.6. View of the PI modules.

processor cache as a temporary memory (or "cache-as-RAM" / CAR) store for the stack and heap. This small, early stack, in addition to putting the processor in the PEI-prescribed mode of execution (e.g., 32-bit protected mode on IA-32). The SEC "executes-in-place" (XIP) from the firmware store.

The SEC's ultimate goal is to put the machine in the state prescribed by the UEFI PI specification. This includes a single thread of hardware execution, some call-stack with a minimum size, and passing a possibly non-zero list of data structures into the PEI core.

The SEC is a single component of the portion of the flash part where the processor passes control upon a machine restart. For the PI bindings of x64, IA-32, and Itanium, this is near the end of the flash volume with the SEC core aligned to end at 4 Gbyte. Alternate architectures, such as ARM, instead expect to pass control to firmware at address zero. To accommodate both, there is a specific aspect of the firmware file system, namely the "Very Top File", that is required to be at either the "beginning" or "end" of the firmware store.

3.5.3 Firmware File systems, HOB, Boot Modes, and Capsules

The initial phase of execution, SEC, described how to pass off control to the second phase, namely PEI, but introduced terms such as "firmware volume" and "firmware file". In PI there are certain data structures and capabilities that will span all phases of PI execution. As such, before progressing toward the PEI and DXE descriptions, a description of these common elements is in order.

The firmware file system of PI includes volumes, files, and sections. The volume is the outermost container and is akin to a partition on disk. Within the volume there can be a plurality of firmware files. And finally, within the file can be a series of sections. The actual encoding of the file system is important for direct discovery of modules and data structures in the early, execute-in-place execution flow.

The file system can describe the literal binary encoding of the firmware volume and files in the storage, or they can be abstracted by API's in the PEI and DXE phase of execution.

Another important object is the Hand-Off-Block (HOB). The HOB is an in-memory list of data structures that are created by various PEI modules and consume by the DXE core and DXE modules. Some of the HOB's are required, such as a description of the memory resource map and the location of additional firmware volumes containing DXE drivers, or they can be vendor/domain specific data that an early PEIM needs to convey to a later DXE driver.

A capsule is a firmware volume file that is described by a particular HOB in PEI. Capsules are used to convey an update from a runtime environment back into the PI phases.

The boot-mode is a value that describes the type of machine restart, including manufacturing mode, the Advanced Configuration and Platform Interface (ACPI) [ACPI] S5 mechanical restart, or a system flash update. PEI phase typically detects and operates upon the boot mode.

3.5.4 PEI: The Pre-Initialization Phase

The Pre-EFI Initialization (PEI) phase of execution is the portion of the UEFI PI infrastructure that receives control from SEC and commences execution, like SEC, in XIP. The PEI core is the component that receives control from the SEC. The PEI core expects to have some RAM (typically from cache)-based stack that is described by the SEC hand-off, along with the location of the "Boot firmware volume", or the firmware volume that may contain other PEI modules in addition to the PEI core. The PEI Core is an executable image, such as PE/COFF or a reduced subset, that has its relocations fixed-up for XIP operation.

Given the stack and a pointer to a firmware volume, the PEI core, in turn, uses integrated read-only firmware volume and file system capability to search for PEI Modules (PEIM). A PEI module, like the PEI core, is an XIP executable image in the firmware volume. The PEI module exists in a firmware file that describes the file type as designating a "PEIM".

The PEI modules can be delivered by various business interests, including the processor manufacturer, chipset vendor, and the system board manufacturer. The PEIM's expose capabilities to other PEIM's via something referred to as a PEIM-to-PEIM Interface (PPI).

The firmware file, in addition to the PEI executable image, may also contain a firmware file system section referred to as a dependency expression. The dependency expression (DEPEX) represents a binary-encoded data structure that uses Reverse Polish Notation (RPN) to describe which PPI's are required by a given PEIM prior to its execution.

The ultimate rationale for the PI PEI phase of execution is to do the minimum amount of work in order to discover some permanent, main memory that is sufficient to pass control to subsequent phase of execution. By "permanent" we mean any physical memory initialized in PEI cannot be relocated to some other portion of the address space by a later phase of execution (e.g., DXE). This is the case because the final action of PEI is to invoke a PPI referred to as the "DXE IPL", or the "Driver Execution Environment Initial Program Loader". DXE IPL will discover the DXE core file in its volume, load, and pass control to the DXE core with the HOB list. If the memory were to "move" during DXE, the DXE core and any memory allocations in PEI that were marked as requiring preservation into the operating system runtime (e.g., AcpiNvs memory type) would be violated.

3.5.5 DXE: The Driver Execution Environment

The Driver Execution Environment, or DXE, is the phase of execution that received control from PEI. The input parameterization of PEI includes the HOB list mentioned before. A picture of the required HOB's is shown in Fig. 3.7.

The DXE core initially provides the UEFI system table and a series of memory only services since this file is ostensibly portable across any microarchi-

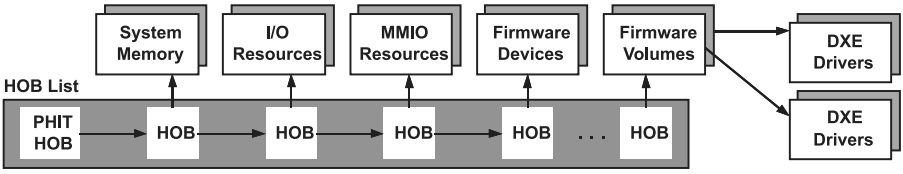

Figure 3.7. Hand-off blocks into DXE.

tecture for which it has been compiled. As such, the DXE core needs to be parameterized by details of the particular platform, including interrupt management, time keeping, UEFI variable management. This is provided by a series of DXE drivers in the firmware volume. The DXE drivers are also PE/COFF executables, but unlike most PEI modules, since the DXE phase commences with system memory, the DXE core and drivers can be decompressed and execute from main memory. Because of the performance and space-savings of this capability, the DXE drivers host higher-level, more algorithmically complex operations.

Figure 3.8 describes the UEFI system table and the associated DXE architectural protocols [UEFb] that provide the platform-specific implementation of some of the UEFI services. For example, the UEFI service SetVariable() has an associated Variable Write architectural protocol (AP) (whose instance is one of the blocks in Fig. 3.8). The use of the architectural protocols is akin to a platform hardware abstraction layer (HAL) that allows for modifying only the respective AP in response to differing platform needs.

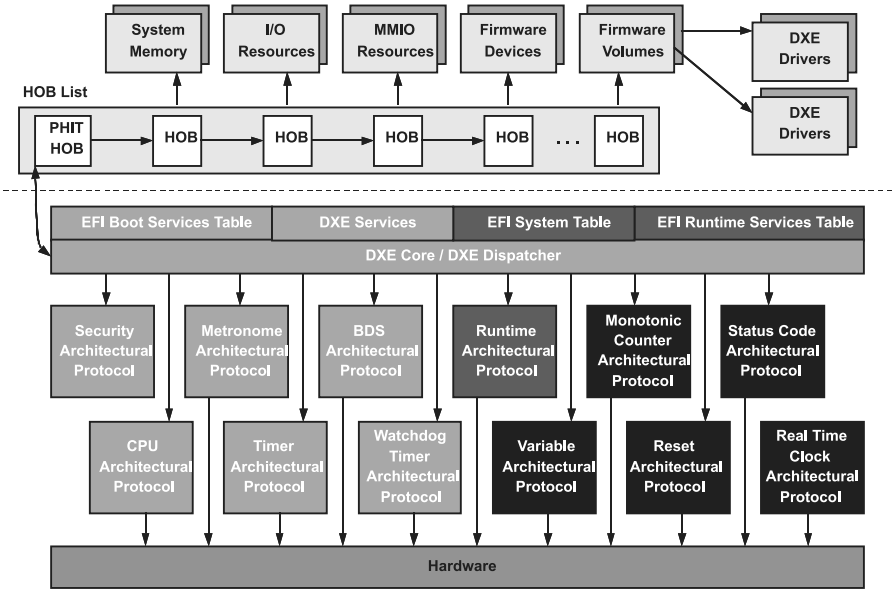

Figure 3.8. DXE interfaces.

The operations hosted in the DXE phase include initialization of the I/O buses, such as the Peripheral Component interconnect (PCI) [PCI], creation of the System Management BIOS (SMBIOS) [SMBIOS] tables, and provide the implementation of the UEFI or conventional PC/AT BIOS. The former provision of the UEFI interfaces is by having the DXE core, whether through

its memory-only services, or via a DXE call abstracted to an alternate interface, provides a fully compliant set of API's.

Unlike PEI, which refers to the interfaces it uses as PPI's, the DXE drivers expose to each other GUID'd API's that are the same as UEFI protocols. Namely an API and/or data set named by a GUID.

During DXE phase of execution, though, all UEFI interfaces (or BIOS, for that matter) are initially available. The DXE drivers uses the same binary encoding of DEPEX's as PEIM's in order to allow the DXE core to orchestrate discovery and dispatch of DXE drivers. DXE expands upon the AND, OR, and NOT opcodes of the PEI DEPEX with additional operators BEFORE, AFTER, and SOR in order to support the richer dispatch model of DXE.

Once the DXE core has dispatched all of its drivers and is ready to "boot" a subsequent pre-OS environment, whether it be the UEFI or PC/AT BIOS, the DXE core invokes the Boot Device Selection (BDS) service. The BDS implements the "boot manager" capability of the UEFI specification or provides a DXE call-down into the BIOS Boot Specification (BBS) [PhTe] capabilities. The BDS is the first opportunity to expose a user interface/splash screen. The BDS also orchestrates the behavior under various boot modes while in DXE. As such, the BDS represents the system board manufacturers specific business needs and look-and-feel. Unlike other DXE drivers that may be provided by chipset or processor vendors, the BDS is most likely heavily modified for a given manufacturer.

Again, DXE provides the subsequent pre-OS execution environment for UEFI or BIOS boots, but it is responsible for machine state construction and hand-off. DXE is typically only extensible for the system board manufacturers and doesn't admit execution of third party modules, such as PC/AT option ROM's or UEFI drivers. The interposition of any foreign content, such as a capsule firmware volume for an OS-present update utility, will typically only be executed/exposed when it is shown that the capsule was produced by the system board manufacturer (e.g. cryptographically signed).

Also, since the BDS is the last DXE component and bridges the gap into PC/AT or UEFI execution, each of which supports 3rd party adapter ROM's on cards or disk, the BDS is the last opportunity to lock down the system board resources. This includes locking the SMRAM or the block-lockable flash.

In addition to the DXE phase of execution, DXE registers components for other process modes and/or machine states. The other modes include the System Management Mode (SMM) on x64 and Platform Management Interrupt (PMI) / Machine Check Architecture (MCA) of the Itanium processor family. Each of these 2 machine phases have specific protocols that allow for loading DXE drivers into System Management RAM (SMRAM) or OS-reserved memory for SMM and PMI, respectively.

In addition, DXE provides drivers to publish data tables and other services, such as ACPI, SMBIOS, and the Itanium System Abstraction Layer (SAL) System Table (SST).

The transition to UEFI and PI from today's BIOS or proprietary boot solutions represents a seismic transition for the industry. But after the development effort and transition costs have been overcome, the "extensibility" of both UEFI and PI will offer a platform for future innovation.

References

[ACPI] ACPI. Advanced Configuration and Power Interface. www.acpi.org

[EDK] EDK. EFI Developer Kit. www.tianocore.org

[PCI] PCI. Peripheral Component Interconnect. www.pcisig.org

[SMBIOS] SMBIOS. System Management BIOS. www.smbios.org

[SMBUS] SMBUS. System Management Bus. www.smbus.org

[PhTe] Phoenix Technologies. BIOS Boot Specification. www.phoenix.com/NR/rdonlyres/56E38DE2-3E6F-4743-835F-B4A53726ABED/0/specsbbs101.pdf

[UEFa] UEFI. UEFI Registry. www.uefi.org/specs/esp_registry

[UEFb] UEFI. Unified Extensible Firmware Interface Platform Initialization Specifications, Volumes 1–5, Version 1.1. November 5, 2007. www.uefi.org

[UEFc] UEFI. Unified Extensible Firmware Interface Specification—Version 2.1. January 23, 2007. www.uefi.org

[ZRH] V. Zimmer, M. Rothman, and R. Hale. *Beyond BIOS: Implementing the Unified Extensible Firmware Interface Specification with Intel's Framework*. ISBN 0-9743649-0-8, Intel Press, September 2006. www.intel.com/intelpress/sum_efi.htm

Chapter 4

HARDWARE ABSTRACTION LAYER

Introduction and Overview

Katalin Popovici and Ahmed Jerraya

Abstract Embedded software is playing an increasing role in heterogeneous Multi-Processor System-on-Chip (MPSoC) architectures due to its high complexity. In order to reduce the long and fastidious design process, embedded software needs to be reused over several MPSoCs. Thus, software portability becomes a key challenge.

In this chapter, we present a clear separation between the hardware independent and the hardware dependent software layers, through adopting a multi-layered organization of the software stack. We introduce a component based software design flow, which allows the gradual generation and validation of the various software layers to obtain the final software stack. Then, by changing the Hardware Abstraction Layer (HAL), the software stack can be executed on different MPSoC architectures. The HAL represents the lowest software layer, which totally depends on the target architecture. The HAL abstraction, through the use of well defined HAL APIs, makes easier the software portability and enables flexibility. The paper shows that the HAL APIs allow early software development before the hardware architecture is available, but also architecture exploration. The proposed methodology is applied to design the software stack for the Motion JPEG multimedia application and to execute it on diverse processors by changing the HAL and preserving the HAL APIs.

Keywords: MPSoC, Software Design, Software Validation, HAL, HAL Abstraction

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

4.1 Introduction

Current Multi-Processor System-on-Chip (MPSoC) architectures integrate a large number of processing subsystems on the same chip [Wol06]. The processing subsystems usually contain different types of programmable processing units or CPUs, depending on the target application domain. Thus, DSP (Digital Signal Processor) is mostly used for signal processing applications; microcontrollers are more common for control-intensive applications, while ASIP (Application Specific Instruction Set Processors) represent stored-memory CPUs whose architectures are tailored for a particular set of applications.

As more and more heterogeneous processors and hardware components are integrated together, the design and validation of the software running on these complex heterogeneous architectures become a major bottleneck, because the software is even more complex [Tur05]. The key issue of the software design for MPSoC is to produce efficient software code with strong time to market constraints. Producing efficient code requires that the software takes into account the capabilities of the target architecture. This generally requires a long and fastidious software debug cycle. The classic way to accelerate the software design process relies on automatic software code generation from high level programming models that abstract the architecture, but this approach produces a huge expense in the efficiency of the generated code.

The software is generally organized into several stacks made of two layers: application and Hardware-dependent Software (HdS). The validation and debug of the Hardware dependent Software (HdS) is the main bottleneck in MPSoC design, because each processor subsystem requires specific HdS implementation to be efficient [JW05]. Current research studies proved that the HdS debug represents 78% of the global system total debugging time of an MPSoC design cycle [YYS+04]. This may be due to incorrect configuration or access to the hardware architecture, e.g. a wrong configuration of the memory mapping for the interrupt control registers.

Besides software complexity, portability becomes a major issue to decrease the overall design and validation time, because it allows software reuse over several SoCs. Portability enables execution of the same software on different hardware architectures. In terms of design reuse, the portability enables reuse of the software designed for a particular MPSoC architecture to another. Thus, portability reduces the design efforts, otherwise necessary to adapt the software for the new hardware architecture.

In order to reduce its complexity and enable easy software portability, we propose structuring the HdS into three software components: a real time operating system (RTOS) aimed to schedule the different application tasks, a specific communication library to implement the communication protocol and

the hardware abstraction layer (HAL) to access the hardware resources. The HAL represents the thin software layer that totally depends on the underlying target architecture. Structuring the HdS in these well defined layers accessible through application programming interface (API) is essential to support software flexibility and portability on different hardware platforms.

Traditional MPSoC design flow starts with the application partitioning into hardware and software tasks that are mapped on processing elements. The definition of generic HAL APIs for the target application domain makes it possible to start designing the software before the hardware is complete, thus enabling concurrent hardware and software design. In fact, the software design is structured in two main phases. The first phase is the hardware independent software design (application tasks, OS), which may start after the definition of the HAL APIs. The second phase represents the HAL design and integration into the software stack. Figure 4.1 shows these steps. Separating the hardware dependent and hardware independent software designs also makes the architecture exploration easier, since the hardware independent software can be reused over several architectures. Only the HAL must be altered in case of different architectures.

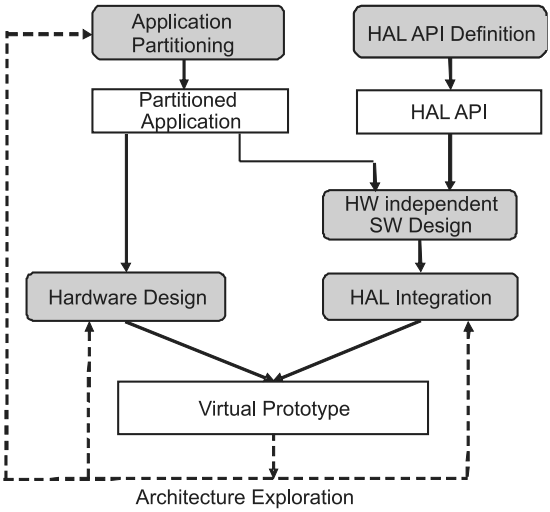

Figure 4.1. Hardware-software design flow.

In this chapter, we give the basic definitions for the different software components, and we emphasize the importance of the HAL layer in the context of MPSoC to provide software portability across different hardware platforms. We present a component based software design flow that allows generation and validation of the different software elements. In order to allow the software reuse, we define the abstraction of the HAL through APIs and the validation

of these HAL APIs. The proposed methodology is applied to design and to adapt the software stack to different processors for the Motion JPEG decoder application.

The chapter is composed of seven sections. Section 4.1 gives a short introduction to present the context of MPSoC software design. Section 4.2 defines the organization of the software stack into different components. Section 4.3 discusses the role of HAL in the software stack and explains how to achieve software portability through abstraction of the HAL. Section 4.4 enumerates several existing commercial HAL. Section 4.5 summarizes the main steps of the proposed software design and validation flow. Section 4.6 presents the HAL execution and simulation using specific software development platforms. Section 4.7 addresses the experimental results, followed by conclusion.

4.2 Software Stack

This section defines the software stack running on the different processing subsystems and presents its layered organization in several software components.

4.2.1 Software Stack Definition

The software stack represents the software running on a processing subsystem. In heterogeneous MPSoC architectures, each processing subsystem executes a software stack. The software stack is made of two layers: the application tasks code and the hardware dependent software (HdS). The HdS layer includes three software components: the Operating System (OS), specific I/O communication software and the Hardware Abstraction Layer (HAL). The HdS is responsible for providing application and architecture specific services, i.e. scheduling the application tasks, communication between the different tasks, external communication with other processing subsystems, or hardware resources management and control. The following paragraphs detail the software stack organization, including all these different components.

4.2.2 Software Components

The software stack is structured in different software layers that provide specific services. Figure 4.2 illustrates the software stack organization in two layers: application layer and HdS (Hardware-dependent Software) layer. In the first section we present the application layer. Then, the HdS layer will be defined.

Application Layer. The application layer contains the software code for applications such as multimedia (e.g. MP3, MPEG4 and JPEG 2000) or communications (e.g. protocol stack and physical layers). It may be a multi-tasking

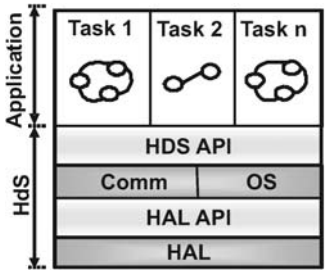

Figure 4.2. Software stack organization.

description or a single task function of the application targeted to be executed on a particular processor subsystem.

A *task* or *thread* is a lightweight process that runs sequentially and has its own program counter, register set and stack to keep track of where it is. In this chapter, the terms task and thread are used as interchangeable terms. Multiple tasks can be executed in parallel by a single CPU (single-core processor subsystem) or by multiple CPUs of the same type grouped in a single subsystem (multi-core processor subsystem). The tasks may share the same resources of the architecture, such as processors, I/O components and memories. On a single processor core node, the multithreading generally occurs by time slicing, wherein a single processor switches execution between different threads. In this case, the task processing is not literally simultaneous, as the single processor is doing only one thing at a time. On a multi-core processor subsystem, threading can be achieved via multiprocessing, wherein different threads can run literally simultaneously on different processors inside the processor node [Tan95].

The application layer consists of a set of tasks that makes use of Application Programming Interface (API) to abstract the underlying HdS software layer. These APIs correspond to the HdS APIs.

HdS Layer. The HdS layer represents the software layer which is directly in contact with, or significantly affected by, the hardware that it executes on, or can directly influence the behavior of the underlying hardware architecture [Pos03]. The HdS integrates all the software that is directly depending on the underlying hardware, such as hardware drivers or boot strategy. It also provides services for resources management and sharing, such as scheduling the application tasks on top of the available processing elements, inter-task communication, external communication, and all other kinds of resources management and control. The federative HdS term underlines the fact that, in an embedded context, we are concerned with application specific implementations of these functionalities that strongly depend on the target hardware architecture [JBP06].

To decrease the complexity of the HdS debug, the HdS is organized into three software components: operating system (OS), communication management (Comm) and hardware abstraction layer (HAL). Figure 4.2 illustrates these software components.

Operating System. The operating system (OS) is the software component that manages the sharing of the resources of the architecture. It is responsible for the initialization and management of the application tasks and communication between them. It provides services, such as tasks scheduling, context switch, synchronization and interrupt management. In the following, we define each of these basic OS services.

The tasks scheduling service of the OS usually follows a specific algorithm, called scheduling algorithm. Finding the optimal algorithm for the tasks scheduling represents a NP-complete problem [VBL05]. There are different categories of scheduling algorithms. The classic criteria are hard real-time versus soft real-time or non real-time; preemptive versus cooperative; dynamic versus static; centralized versus distributed.

Contrary to non real-time, the real-time scheduler must guarantee the execution of a task in a certain period of time. Hard real-time must guarantee that all the deadlines are met.

Preemptive scheduling allows a task to be suspended temporally by the OS, for example when a higher-priority task arrives, resuming later when no higher-priority tasks are available to run. This is associated with time-sharing between the tasks. Examples of preemptive scheduling algorithms are: round robin, shortest-remaining-time or rate-monotonic schedulers. The cooperative or non-preemptive scheduling algorithm runs each task to its completion. In this case, the OS waits for a task to surrender control. This is usually associated with event-driven operating systems. Examples of non-preemptive algorithm are the shortest-job-next or highest-response-ratio-next.

With static algorithms, the scheduling decisions (preemptive or non-preemptive) are made before execution. Contrary to static algorithms, the dynamic schedulers make their scheduling decisions during the execution.

The implementation of the scheduler may be centralized or distributed. In case of a centralized scheduler implementation, the scheduler controls all the task execution ordering and communication transactions. In case of a distributed scheduler implementation, the scheduler distributes the control decision to the local task schedulers corresponding to each processor [CYC+05].

When a task is ready for execution and it is selected by the scheduler of OS according to the scheduler algorithm, the OS is also responsible to perform the context switch between the currently running task and the new task. The context switch represents the process of storing and loading the state of the CPU which runs the tasks, in order to share the available hardware resources

between different tasks. The state of the current task, including registers, is saved, so that in case the scheduler gets back for execution the first task, it can restore its state and continue normally.

In order to ensure a correct runtime and communication order between the different tasks running on parallel, synchronization is required. The tasks can synchronize by using semaphores or by sending/receiving synchronization signals (events) each other. The mutex is a binary semaphore which ensures mutual exclusion on a shared resource, such as a buffer shared by two threads, by locking and unlocking it, whenever the resource is accessed by a task [TW97].

The interrupt handler is another OS service used for the interrupts management. There are two types of processor interrupts: hardware and software. A hardware interrupt causes the processor to save its state of execution via a context switch, and begins the execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set of the processor, which cause a context switch to an interrupt handler similar to a hardware interrupt. The interrupts represent a way to avoid wasting the processor's execution time in polling loops waiting for external events. Polling means when the processor waits and monitors a device until the device is ready for an I/O operation.

Examples of commercial OS are the eCos [eCos], FreeRTOS [FRTOS], LynxOS [LOS], VxWorks [VxW], WindowsCE [WCE] or μITRON [uIT].

Communication Software Component. The second software component of the HdS layer constitutes the communication component, which is responsible for managing the I/O operations and, more generally, the interaction with the hardware components and the other subsystems. The communication component implements the different communication primitives used inside a task to exchange data between the tasks running on the same processor or between the tasks running on different processors. It may include different communication protocols, such as FIFO (first-in-first-out) implemented in software, or communication using dedicated hardware components. If the communication requires access to the hardware resources, the communication component invokes primitives that implement this kind of low level access. These function calls are done in form of the HAL APIs.

The HAL APIs allow for the OS and communication components to access the third component of the software stack, that is the HAL layer.

Hardware Abstraction Layer. Low level details about how to access the resources are specified in the Hardware Abstraction Layer (HAL) [YJ03]. The HAL is a thin software layer which totally depends on the type of processor that will execute the software stack, but also depends on the hardware resources interacting with the processor. The HAL includes the device drivers to implement the interface for the communication with the device. This includes the

implementation of the drivers for the I/O operations or other peripherals. The HAL is also responsible for processor specific implementations, such as loading the main function executed by an OS, more precisely the boot code, or implementation of the *load* and *restore* CPU registers during a context switch between two tasks, but also software codes for configuration and access to the various hardware devices, e.g. MMU (Memory Management Unit), timer, interrupt enabling/disabling etc. More details about the HAL will be given in the following sections.

The structured representation and the organization of the software stack into several layers (application tasks, OS, communication and HAL), as previously described, have two main advantages: flexibility in terms of software components reuse by changing the OS or the communication software components, and portability to other processor subsystems by changing the HAL software layer.

The following paragraphs give the definition of the HAL software component and highlight its role in enabling software portability. Thereafter, the main steps required by the design and validation of these different software components are explained in detail.

4.3 Hardware Abstraction Layer

In this section, the definition of the HAL is given. This is followed by the HAL abstraction through well defined APIs to enable software portability across various hardware platforms.

4.3.1 Definition and Examples of HAL

The HAL is defined in [eCos] as all the software that is directly dependent on the underlying hardware. If the hardware architecture is changed, changes also have to be made to the HAL. The HAL can be implemented in the assembly language recognized by the processor or in specific C code. In fact, the HAL includes two types of software code:

- Processor specific software code, such as context switch, boot code or code for enabling and disabling the interrupt vectors.
- Device drivers, which represents the software code for configuration and access to hardware resources, such as MMU (Memory Management Unit), system timer, on-chip bus, bus bridge, I/O devices, resource management, such as tracking system resource usage (check battery status) or power management (set processor speed).

The HAL offers a set of services to the upper level OS and communication libraries that grant them access to the hardware platform. Generally, the HAL provides the following kinds of services:

- Integration with an ANSI C standard library to provide the familiar C standard library functions, such as *printf()*, *fopen()*, *fwrite()*, *exit()*, *abs()*, *atof()*, etc. An example of such a library is the newlib library, which represents an open-source implementation of the C standard library .newlib for the use on embedded systems [newl].
- Device drivers to provide access to each device of the hardware platform.
- The HAL API to provide a consistent interface to HAL services, such as device access, interrupts handling and debug facilities.
- System initialization to perform the initialization of the tasks for the processor before the execution of the *main()* function of the application.
- Device initialization to instantiate and initialize each device in the hardware platform before the execution of the *main()* function of the application.

The device drivers, that are part of the HAL, are the interface between a hardware resource and the application or OS. Usually, the drivers are hardware dependent and OS specific. Typical device drivers provide access to the following classes of hardware components:

- Character-mode devices, which represent hardware peripherals that send and/or receive characters serially, such as an UART (Universal Asynchronous Receiver/Transmitter) device.
- Timer devices, which are hardware peripherals that count clock ticks and generate periodic interrupt requests.
- File subsystems, which provide a mechanism for accessing files stored within physical devices. Depending on the internal implementation, the file subsystem driver may access the underlying devices either directly or by using a separate device driver. For example, a flash file subsystem driver may access a flash memory by using dedicated HAL APIs for the flash memory devices.
- Ethernet devices to provide access to an Ethernet connection for a networking stack, such as the NicheStack TCP/IP stack [NS].
- DMA devices that are peripherals that perform bulk data transactions from a data source to destination. Sources and destinations can be memory or another hardware device, such as an Ethernet connection.
- Flash memory devices, which are nonvolatile memory devices that use a special programming protocol to store data.

Besides the implementation of the device drivers, the HAL includes processor specific code as well, such as the implementation of the context switch or interrupt handling.

Figure 4.3 presents an example of processor specific HAL code, which performs a context switch between two application tasks running on an ARM7 processor. This example of HAL software code uses the assembly language specific to the ARM7 processor in order to access some particular processor registers (R0-R14, PC-Program Counter). The context switch needs two basic operations to be performed: store the status of the processor registers used by the current task and load the status of the registers of the new task.

```
__ctx_switch                    ; r0 old stack pointer, r1 new stack pointer
STMIA r0!,{r0-r14}              ; save the registers of current task
LDMIA r1!,{r0-r14}              ; restore the registers of new task
SUB pc,lr,#0                    ; return
END
```

Figure 4.3. HAL implementation for the context switch on the ARM7 processor.

Figure 4.4 illustrates another example of low level software code implementation that enables and disables the IRQ interrupts for the ARM7 processor. The interrupts are enabled and disabled by reading the CPSR (Current Program Status Registers) flags and updating bit 7 corresponding to bit I (IRQ Interrupt).

```
__inline void enable_IRQ(void)                          //HAL API
{
          int tmp;
          __asm
          {
                    MRS tmp, CPSR
                    BIC tmp, tmp,#0x80
                    MSR CPSR_c, tmp
          }
}

__inline void disable_IRQ(void)                         //HAL API
{
          int tmp;
          __asm
          {
                    MRS tmp, CPSR
                    ORR tmp, tmp, #0x80
                    MSR CPSR_c, tmp
          }
}
```

Figure 4.4. HAL implementation for enabling and disabling ARM interrupts.

4.3.2 Software Portability Based on HAL API

In the context of software design for MPSoC, software portability becomes a key issue. Portability enables execution of the same software on different hardware architectures. In terms of design reuse, the portability enables reuse of the software designed for a particular MPSoC architecture to another. Thus, portability reduces the design efforts, otherwise necessary to adapt the software for the new hardware architecture. The portability also eases the exchange of the software code and architecture exploration, e.g. trying different types of processors to find an optimal target processor.

As it was described in the previous paragraphs, the structural organization of the software stack is made of several layers separated by well defined APIs. The lowest level software component represents the HAL layer which is a totally hardware architecture dependent layer. The OS and communication software components make use of HAL APIs. Thus, without the implementation of the HAL APIs for the target processors, the software code still remains processor independent.

The HAL APIs gives to the operating system, communication and application software an abstraction of the hardware-dependent HAL, e.g. data types like the integer "int" data type in the standard C programming language, which has different bit size depending on the processor type. Furthermore, the HAL APIs ease OS porting on new hardware architecture. The HAL APIs can be classified in the following categories [eCos]:

- Kernel HAL APIs, such as task context management APIs (e.g. context creation, delete or context switch APIs, task initialization), stack pointer and program counter management APIs (e.g. *get/set_IP()*, *get/set_SP()*) or processor mode change APIs (e.g. *enable_kernel/ user_mode()*).

- Interrupt management APIs, e.g. APIs which enable/disable interrupt request from an interrupt source (e.g. *vector_enable/ disable(vector_id)*), configure interrupt vector (e.g. *vector_configure(vector_id, level, up)*), mask/unmask interrupt for a processor (e.g. *interrupt_enable/disable()*), the implementation of the interrupt routine services (e.g. *interrupt_attach/ detach(vector_id, isr)*) or HAL APIs that acknowledge to the interrupt source that the interrupt request has been processed (e.g. *clear_interrupt(vector_id)*).

- I/O HAL APIs, which configure the I/O devices and allows their access. For example, to configure a MMU device, the following I/O HAL APIs may be required: APIs for page management (e.g. *enable/disable_paging()*), address translation (e.g. *virtual_to_physical()*), TLB (Translation Lookaside Buffer) management, such as set TLB entry (e.g. *TLB_add()*) or get TLB entry virtual/physical page frame (e.g.

get_TLB_entry()). Other I/O HAL API examples can be considered the APIs for cache memory management, such as *Instruction/Data_Cache_Enable/Disable()*.

- Resource management APIs, such as APIs for power management (e.g. check battery status, set CPU clock frequency) or APIs to configure the timer (e.g. *set/reset_timer()*, *wait_cpu_cyle()*).

- Design time HAL APIs, which facilitates the software design process, or more precisely, the simulation. Example of such kind of API is the *consume_cpu_cyle()* to simulate the advance of the software execution time.

The HAL APIs are used by the upper software layers, like OS and communication components. Figure 4.5 shows an example of utilization of the HAL API in a fragment of code inside the OS scheduler. Thus, the OS scheduler searches for a new task in status ready for execution. If there is a new ready task, the scheduler performs a context switch, by calling the HAL API *__cxt_switch(...)*. During the context switch, the OS saves the status and registers (program counter, stack pointer, etc.) of the processor running the current task and loads those of the new task.

```
void __schedule (void){
   int old_tid = cur_tid;
   cur_tid = get_new_tid();          //get new task ready for execution

   __ctx_switch (old_tid,cur_tid);  //context switch HAL API
  …
```

Figure 4.5. Example of HAL API function call inside the OS scheduler.

4.4 Existing Commercial HAL

In the following section, we give several examples of existing commercial HAL that are used in both academic and semiconductor industry areas.

Even if the HAL represents an abstraction of the hardware architecture, since it has been mostly used by OS vendors and each OS vendor defines its own HAL, most of the existing HAL is OS dependent. In case of an OS dependent HAL, the HAL is often called board support package (BSP). In fact, the BSP implements a specific support code for a given hardware platform or board, corresponding to a given OS. The BSP also includes a boot loader, which contains a minimal device support to load the OS and device drivers for all the devices on the hardware board.

The embedded version of the Windows OS, namely Windows CE, provides BSP for many standard development platforms that support several microprocessors family (ARM, x86, MIPS) [WCE]. The BSP contains an OEM (Original Equipment Manufacturer) adaptation layer (OAL), which includes a boot loader for initializing and customizing the hardware platform, device drivers, and a corresponding set of configuration files.

The VxWorks OS offers BSP for a wide range of MPSoC architectures, which may incorporate ARM, DSP, MIPS, PowerPC, SPARC, XScale and other processors family [VxW]. In eCos, a set of well-defined HAL APIs are presented [eCos]. However, there's no clear difference between HAL and device driver. Examples of HAL APIs used by eCos are:

- Thread context initialization:
  HAL_THREAD_INIT_CONTEXT()
- Thread context switching:
  HAL_THREAD_SWITCH_CONTEXT()
- Breakpoint support:
  HAL_BREAKPOINT()
- GDB support:
  HAL_SET_GDB_REGISTERS(), HAL_GET_GDB_REGISTERS()
- Interrupt state control:
  HAL_RESTORE_INTERRUPTS(), HAL_ENABLE_INTERRUPTS(),
  HAL_DISABLE_INTERRUPTS()
- Interrupt controller management:
  HAL_INTERRUPT_MASK()
- Clock control:
  HAL_CLOCK_INITIALIZE(), HAL_CLOCK_RESET(),
  HAL_CLOCK_READ()
- Register read/write:
  HAL_READ_XXX(), HAL_READ_VECTOR_XXX(),
  HAL_WRITE_XXX(), and HAL_WRITE_VECTOR_XXX()
- Control the dimensions of the instruction and data caches:
  HAL_XCACHE_SIZE(), HAL_XCACHE_LINE_SIZE()

In the software development environment for the Nios II processor provided by Altera [HAL], the HAL serves as a device driver package, providing a consistent interface to the system peripherals, such as timers, Ethernet MAC and I/O peripherals.

In Real-Time Linux a HAL, called Real-Time HAL (RTHAL), is defined to give an abstraction of the interrupt mechanism to the Linux kernel [RTL]. It consists of three APIs for disabling and enabling interrupts and return from the interrupt.

An example of HAL that does not depend on the targeted OS is the a386 library [A386]. The a386 represents a C library which offers an abstraction of the Intel 386 processor architecture. The functions of the library correspond to privileged processor instructions and access to the hardware. The library serves as a minimal hardware abstraction layer for the OS. Later, the library is ported on ARM and SPARC processors.

4.5 Overview of the Software Design and Validation Flow

This section gives an overview of the software design and validation flow. The overall flow is illustrated in Fig. 4.6.

The software design flow has three main steps: application software generation, software stack construction, and performance and software validation through simulation on a development platform [PJ07].

The software design flow starts with a manual design step to build the high level application model that captures the grouping of the application functions

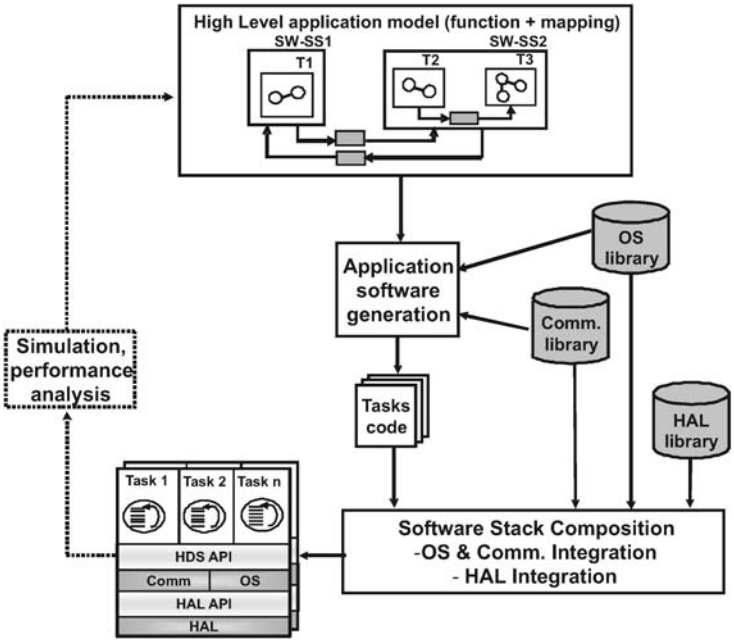

Figure 4.6. Software design and validation flow.

into tasks, and the tasks into processor subsystems. Thus, it combines the application behavior with the architecture specification, and the application mapping information onto the architecture. The result of this step represents a combined architecture/application model. This high level application model may use explicit communication units to abstract the intra-subsystem communication (communication between the different processor subsystems) and inter-subsystem communication (communication between the tasks mapped onto the same processing subsystem).

4.5.1 High Level Application Model

The high level application model represents a functional description of the application annotated with the application mapping information on the target architecture. We use Simulink environment [Math] to capture this representation. We use a specific writing style and annotation to capture the architecture details and the mapping of the communication and computation. At this level, the software is made of a set of functions grouped into tasks and the tasks grouped into software subsystems. The communication between functions, tasks and subsystems make use of abstract communication links to represent logical communication, e.g. standard Simulink links or explicit communication units that correspond to specific communication paths of the target platform. The links and units are annotated with communication mapping information. The simulation at the system architecture level allows validating the application's functionality. The hardware-software interfaces are fully abstracted. This model captures both the application and the architecture in addition to the computation and communication mapping.

4.5.2 Application Software Generation

During the application software generation, the Simulink application functions are transformed into behaviorally equivalent C code for each task. This step is similar with the code generation provided by Real Time Workshop, but the generated code uses an optimized buffer memory [Han06 $^{+}$].

The generated code is made of two parts: computation and communication. The computation part represents the C behavior of the application functions, while the communication part involves high level communication primitives, such as *send(. . .)/recv(. . .)* or *channel_write(. . .)/ channel_read(. . .)*. The implementation of these APIs relies on the underlying OS and communication libraries.

4.5.3 Software Stack Composition

During the software stack composition, the previously generated application tasks code are compiled and linked together with an OS, communication

and HAL library [GPY+07]. The OS library contains the components that implement several OS services, such as scheduling, interrupt routine services, tasks management (create/kill/exit). The communication library contains the implementation of the high level communication primitives, e.g. MPI (Message Passing Interface) primitives [MPI], the TTL communication primitives [vdW04+] or YAPI communication APIs [KSW+00]. The implementation of these communication primitives can be blocking or non-blocking. The HAL library contains the implementation of the low level hardware access primitives, e.g. context switch primitives, enable/disable interrupts, boot code or specific DMA configuration primitives. The software stack composition is performed in two main steps:

- OS and communication software components integration
- HAL integration

The result of each of these steps has to be validated in order to verify the application execution on the target hardware architecture, as it will be detailed in the next section.

4.5.4 Software Validation

The software validation allows verifying the execution of the software with explicit hardware-software interaction. Traditional software development strategies make use of the concept of software development platform to debug the software before the hardware architecture is ready.

As illustrated in Fig. 4.7, the software development and validation platform is an abstract model of the architecture in form of a run-time library or simu-

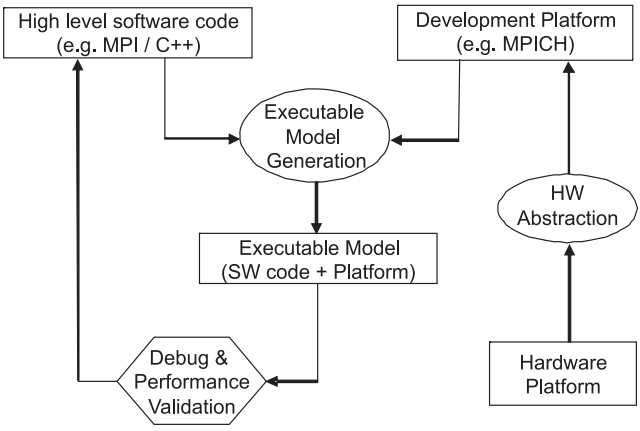

Figure 4.7. Software development and validation platform.

lator aimed to execute the software (e.g. Instruction Set Simulator). The combination of this platform with the software code produces an executable model that emulates the execution of the final system including hardware and software architecture. Generic software development platforms have been designed to fully abstract the hardware-software interfaces, i.e. MPITCH is a run-time execution environment designed to execute parallel software code written using MPI [MPI].

In this chapter, we use software development platforms implemented in SystemC TLM [GLMS02] in order to execute and debug the software code [PGR+07].

Depending on the software component to be validated (application tasks code, tasks code execution upon an OS, HAL integration in the software stack), the SystemC platform may model only a subset of hardware components, more precisely those components that are required for the software validation. The rest of the hardware components, which are not relevant for the software validation, are abstracted. For example, the debug of the application tasks code does not need explicit implementation of the synchronization protocol between the processors, such as mailboxes, semaphores or mutexes, while the debug of the integration of the tasks code with the OS requires such kind of detail in the SystemC platform.

The debug is performed using standard debuggers such as GNU debuggers or tracing SystemC waveforms during the simulation. The software validation is an iterative process because the different software components need different detail levels.

4.6 HAL Execution and Simulation Using Software Development Platforms

In order to explore and reuse the validated software components for better performance achievement, by executing them on various hardware architectures, the HAL layer plays a key role to guarantee software portability. Thus, the following sections will focus on the HAL execution on a virtual prototype using Instruction Set Simulators (ISS), and HAL APIs simulation using a transaction accurate SystemC development platform.

4.6.1 HAL Execution on Virtual Prototype

The integration of the HAL layer into the software stack needs to be validated for functional verification purpose. In order to validate such kind of HAL code, there are two possible execution techniques of HAL:

1. direct loading the software code onto the processor's program memory and execute it on a real chip or an equivalent FPGA-based emulation board;

2. using a software development platform that models the target architecture and incorporates Instruction Set Simulators (ISS) for the processors.

In this chapter, we detail the HAL execution using a SystemC development platform that combines ISS for the software execution and SystemC for the hardware simulation. This platform is also known as virtual prototype [HYL+06] and the execution model corresponds to classical hardware-software cosimulation models with ISS [Row94] [SG00].

The integration of instruction set simulators for the software execution on specific processors with hardware simulators of the architecture behavior is largely used in MPSoC domain. By using ISS, this approach allows simulating a detailed hardware-software interaction, including the HAL of the software stack. For performance verification, the timing information can be measured instead of estimated.

The execution model of the virtual prototype resides on a cosimulation between the software stack simulator and the hardware simulator [NYBJ02]. Two types of simulators are combined: ISS for simulating the programmable components running the software and SystemC for the dedicated hardware part [EPTP07].

The hardware-software simulation is driven by SystemC. The SystemC initializes the processor SystemC modules that encapsulate the ISS. During the simulation, the ISS features a simulation loop which fetches, decodes and executes instructions one after another. The ISS is developed as sequential software running on a single processor. The simulation performed at this level is cycle accurate. The simulation of the virtual prototype allows validating the HAL integration into the final software stack.

Figure 4.8 shows the execution model of an architecture made of two processors, ARM7 and XTENSA. The model contains two ISS to execute the binary

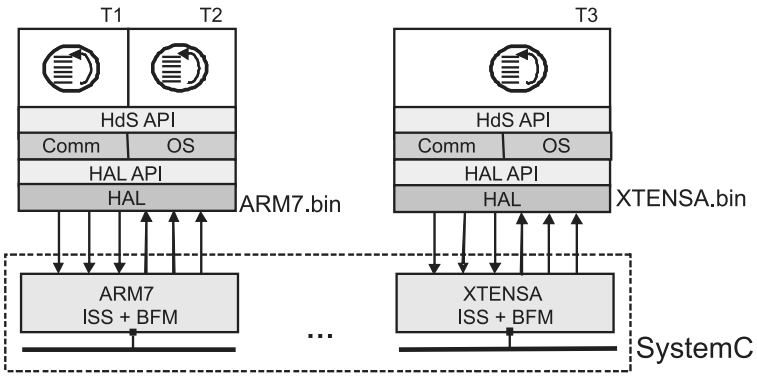

Figure 4.8. Virtual prototype execution model.

codes, corresponding to the ARM7, respectively XTENSA processors. The rest of the architecture components are cycle accurate SystemC components modeled at TLM with execution timing information. The two software stacks that are executed by the two processors include the application tasks code, communication and operating system layer and the processor specific HAL.

4.6.2 HAL Simulation on Transaction Accurate SystemC Platforms

Instead of executing the HAL on the virtual prototype, as it was previously described, the HAL can be simulated using the APIs provided by the OS running on the host machine. In this manner, the HAL APIs are executed natively on the host machine, thus providing a simulation model of the OS and the inter-processor communication scheme [BYJ04].

For example, the implementation of the *ctx_switch (old_tid, cur_tid)* HAL API, which performs a context switch between two tasks, relies on the APIs provided by the OS running on the host machine (Windows, Linux, UNIX, etc.). Figure 4.9 exemplifies the implementation of the context switch on a host machine running Linux OS, which makes use of *sigsetjmp* and *siglongjmp* APIs to save and switch the context of a task.

```
void __ctx_switch(int old_tid, int new_tid)
{
          sigjmp_buf old_buf, new_buf;

          old_buf = task[old_tid].buf;
          new_buf = task[new_tid].buf;

          if(!sigsetjmp(old_buf, 1))                          //LINUX APIs
                              siglongjmp(new_buf, 1);
}
```

Figure 4.9. Simulation of the __ctx_switch() HAL API.

Using this kind of HAL simulation model, the software stack still remains processor independent. Therefore, by abstracting the HAL through the use of HAL APIs, the application tasks code, OS and communication software components can be migrated between various processors. In this case, the only requirement is that those processors need to support the implementation of the HAL APIs, thus allowing software portability.

In order to verify the hardware-software interface, the HAL APIs are required to be executed upon a development platform with detailed hardware-software interaction. In the following, we present the execution model that allows the HAL native simulation and makes use of a transaction accurate hardware platform implemented in SystemC. The hardware platform contains

all the hardware resources that are required for the HAL APIs native execution and validation.

The combination of the transaction accurate platform with the software stack based on HAL APIs results in an executable model. The full hardware-software executable model is based on a co-simulation between SystemC for the hardware components including the abstract execution models of the processors, and the native execution of the software stacks [NYBJ02].

Each software stack is a SystemC thread which creates a Linux process for the software execution. At the beginning of the simulation, the SystemC platform launches a GNU standard debugger (gdb) Linux process for each software stack in order to start its execution. The software stack interacts with the corresponding SystemC abstract processor module through the Linux IPC layer. The hardware-software interface uses Linux shared memory (IPC Linux *shm*) for the interaction, data and synchronization exchange between the software and the hardware.

Figure 4.10 shows the execution model of two software stacks running on two processors, ARM7 and XTENSA. This represents a co-simulation between the gdb Linux processes of each software stack *gdb1* and *gdb2* (one gdb per each software stack) and one SystemC Linux process for the whole SystemC simulation of the hardware platform. The interface between the three Linux processes is performed using the Linux IPC shared memory.

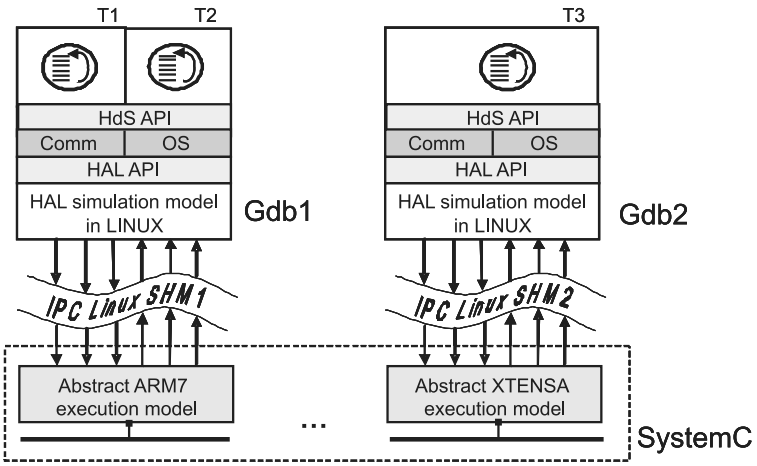

Figure 4.10. Transaction accurate SystemC execution model.

The simulation of the transaction accurate architecture allows validation of the integration of the tasks code with the OS and the communication protocol, providing a simulation model for the HAL APIs. The simulation allows debugging the software access to the hardware resources (e.g. access to the AMBA

bus, interrupt lines assignment, OS scheduling, etc.). It makes possible the debug of the access of the OS functions to the hardware resources through the HAL APIs, e.g. *read(...)/ write(...)* from/to the memory, explicit synchronization using mailboxes or the interrupt routine services. The simulation also gives more precise statistics on the communication and computation performances, such as number of exchanged bytes during the application execution or estimation of the processors cycles spent on communication.

4.7 Experiments

In this chapter, we present the HAL integration for the Motion JPEG decoder application. This application targets various hardware architectures, involving Xtensa processor [Xte], ARM processor [ARM] or Atmel DSP [mVD].

The Motion JPEG Decoder application represents an image processing multimedia application. In this chapter, the baseline Motion-JPEG decoder is used as target application example, which represents the basic JPEG decoding process supported by all the JPEG decoders [Wal91]. The JPEG decoder performs the decompression of an encoded JPEG bitstream (01011...) and renders the decoded bitmap images on a screen. The JPEG compression algorithm operates on blocks of 8×8 pixels of the image. The main functions of the Motion JPEG application, as illustrated in Fig. 4.11 are:

- Variable Length Decoding (VLD), which transforms the input binary sequence into a symbol sequence using the Huffman tables
- Differential Pulse Code Demodulation (DPCD) applied upon the DC coefficient
- Run Length Decoding (RLD) applied upon the 63 AC coefficients
- Zigzag Scan, which reconstructs the matrix of the DCT coefficients from the DC and 63 AC elements
- Inverse Quantization (IQ), which uses the quantification tables

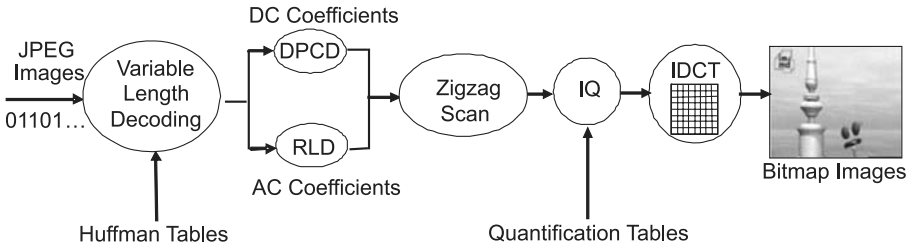

Figure 4.11. Motion JPEG decoder.

- Inverse Discrete Cosine Transformation (IDCT), which transforms the DCT coefficients from frequency domain to spatial domain.

The experimentation is carried out by using three types of processor cores. The first processor core represents the Xtensa processor [Xte]. This processor works at 350 MHz frequency and has 8 Kbytes data cache and 8 Kbytes instruction cache memories. The second core belongs to ARM9 processors family and represents the ARM926EJ-S type of core [mVD]. This runs at 200 MHz frequency and is equipped with 16 KBytes data cache and 16 KBytes instruction cache memories. The third processor represents the magicV VLIW DSP Atmel processor, running at 100 MHz [Wal91].

The Motion JPEG application aims to be executed on these different types of processors. A small OS is used to start the execution of the application and to initialize diverse hardware devices, i.e. I/O devices. The execution and portability of the application software is performed by changing the HAL component of the software stack. The processor specific application code optimization techniques are not considered during the experimentation, in order to preserve the application code hardware independent. Due to the use of the HAL APIs, the application code and OS remains unmodified, thus enabling software portability. The OS makes use of the same HAL APIs for all the hardware architectures. We illustrate three examples of HAL APIs that are identical for the different processors. These HAL APIs are the following:

- *_set_context()* HAL API, which initializes the task that will be executed by the processor, more precisely the stack
- *vector_enable()* HAL API, which enables the interrupts
- *vector_disable()* HAL API, which disables the interrupts

Figures 4.12, 4.13 and 4.14 illustrate the diverse implementations of the same HAL APIs targeting the Xtensa, ARM9, respectively DSP processor. The implementation for the ARM9 processor mainly uses assembly language. The implementations of the HAL APIs for the Xtensa processor and the DSP are based on other APIs, provided by the processors vendors.

After the compilation of the software stack, composed of the Motion JPEG decoder application, a tiny OS and the HAL specific to each CPU, the resulted memory requirements are as follows: 3072 bytes data memory and 4802 bytes of code size for the program memory in case of the Xtensa processor, 3056 bytes data memory and 5092 bytes program memory for the ARM9 processor, respectively 739 bytes data memory and 2806 bytes program memory for the DSP. Table 4.1 summarizes these values and also the code and data size of the HAL for the three different types of processors.

Xtensa

```
void _set_context(thread_context_t buf, fcall_t function, void *stack)
{
          _setjmp(buf);
          buf[JB_PC] = (int)(function);
          buf[JB_SP] = (int)(stack + STACK_SIZE);
}
```

ARM9

```
_set_context:
          stmia     r0!, {r0-r10, fp, ip}         @ we save r0-r12
          stmia     r0!,  {r2}                    @ sp
          stmia     r0!,  {r1}                    @ lr
          stmdb     sp!, {r4-r5}
          mrs       r5, cpsr                      @ we get the cpsr
          mrs       r4, spsr                      @ and the spsr
          stmia     r0!, {r4-r5}                  @ we can save them
          ldmdb     sp!, {r4-r5}
          mov       pc, lr                        @ and we branch
```

DSP

```
void _set_context(thread_context_t buf, fcall_t function, void *stack)
{
           _DBIOS_Init(function, buf, stack);
}
```

Figure 4.12. Implementation of _set_context() HAL API for different processors.

Xtensa

```
void vector_enable(int irq_type)
{
          if(intr_init==0)
                  {
                   _xtos_set_interrupt_handler(0, _irq_dispatch);
                   intr_init =1;
                  }
          _xtos_ints_on ( 1L << (0));
}
```

ARM9

```
void vector_enable(int irq_type)
{
          asm("mrs r1, cpsr");
          asm("bic r1, r1, r0");
          asm("msr cpsr_c, r1");
}
```

DSP

```
void vector_enable(int irq_type)
{
           AT91F_DSP_INTERRUPT_Enable (irq_type);
}
```

Figure 4.13. Implementation of the vector_enable() HAL API for different processors.

Xtensa

```
void vector_disable(int irq_type)
{
          _xtos_ints_off ( 1L << (0));

}
```

ARM9

```
void vector_disable(int irq_type)
{
          asm("mrs r1, cpsr");
          asm("orr r1, r1, r0");
          asm("msr cpsr_c, r1");
}
```

DSP

```
void vector_disable(int irq_type)
{
           AT91F_DSP_INTERRUPT_Disable (irq_type);
}
```

Figure 4.14. Implementation of the vector_disable() HAL API for different processors.

Processor	Application		HAL	
	Data [Bytes]	Code [Bytes]	Data [Bytes]	Code [Bytes]
Xtensa	3072	4802	112	1185
ARM9	3056	5092	14	1248
DSP	739	2806	52	296

Table 4.1. Code and data size.

Figure 4.15 illustrates the total execution cycles measured when executing the whole Motion-JPEG application on the different processors using ISS. In all the cases, the input bitstream represents a 10 frames image encoded using QVGA format, and stored in the local memory of the processor. As shown in Fig. 4.15, the number of execution cycles required to decode the 10 frames image is approximately 137 Mega cycles on the Xtensa processor, 71 Mega cycles on the ARM9 processor and 164 Mega cycles on the DSP. Table 4.2 indicates the characteristics of each of these processors, as specified by the IP vendors, in terms of speed (clock frequency), surface and corresponding power consumption. The processors are configured as shown in Table 4.2.

The performance difference between the processors is explained by the availability of the additional cache memories and improvement in number of cycles required for the load/store operations. The real time requirement of 25 frames decoded per second implies an execution per frame within 8 Mega cycles on a CPU running at 200 MHz, 4 Mega cycles on a CPU running at 100 MHz

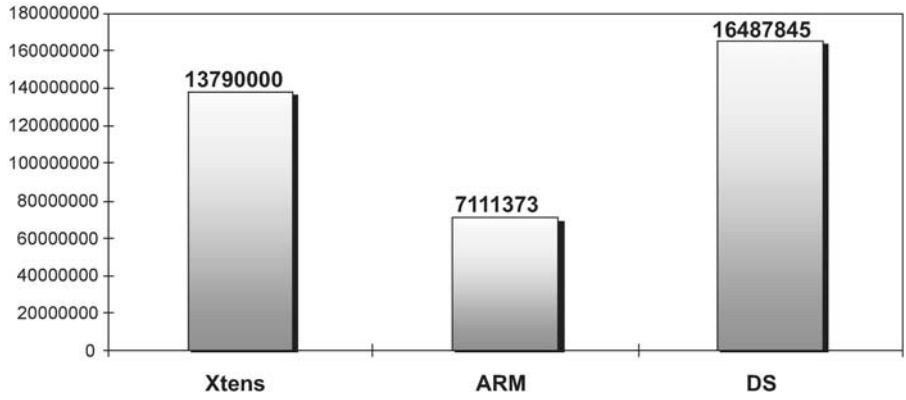

Figure 4.15. Execution clock cycles of Motion JPEG decoder QVGA.

Processor	Frequency	Surface	Power Consumption
Xtensa (core)	350 MHz	0.26 mm^2	26.25 mW
ARM9	200 MHz	2.78 mm^2	96 mW
DSP	100 MHz	13.2 mm^2	229.2 mW

Table 4.2. Frequency, surface and power consumption.

and 14 Mega cycles on a CPU running at 350 MHz. Thus, the Motion JPEG decoder can be executed in real-time by using the ARM9 processor and the Xtensa configurable processor. The surface of the hardware in case of the ARM9 processor is 2.78 mm^2 with a power consumption of 96 mW. The execution on the DSP can be improved by using DSP specific optimization features in order to speed up the critical computing part of the application. But the processor specific application optimization reduces software portability. The DSP is the biggest power consumer among the three targeted processors, and it implies a surface of 13.2 mm^2. The Xtensa core is the optimal processor in terms of surface and consumption, but it is not equipped with any extra hardware accelerators in the configuration used during the experimentation.

4.8 Conclusions

In this chapter, we presented a layered organization of the software stack into application tasks code, operating system and communication libraries, and HAL. The structured representation of the software stack separates the hardware independent and hardware dependent software layers, thus allowing easy software portability. The different software components are generated and validated gradually by using specific software development platforms. Abstracting and simulating the HAL through HAL APIs allows software reuse and flexibil-

ity. To illustrate the effectiveness of the proposed methodology, we generated the software stack for the Motion JPEG application targeting different hardware architectures. The execution of the Motion JPEG on multiple processors (Xtensa, ARM9, DSP) was possible due to the clear separation between the hardware independent software code (application tasks code, OS and communication) and the hardware dependent HAL.

References

[A386] A386. a386.nocrew.org

[ARM] ARM. www.arm.com

[BYJ04] A. Bouchima, S. Yoo, and A.A. Jerraya. Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model. In *Proceedings of ASP–DAC 2004*, January 2004, Yokohama, Japan, 2004.

[CYC+05] Y. Cho, S. Yoo, K. Choi, N.E. Zergainoh, and A.A. Jerraya. Scheduler implementation in MPSoC design. In *Proceedings of ASP–DAC 2005*, 18–21 January 2005, Shanghai, China, pages 151–156, 2005.

[eCos] eCos. www.ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html

[EPTP07] C. Erbes, A.D. Pimentel, M. Thompson, and S. Polstra. A framework for system-level modeling and simulation of embedded systems architecture. *EURASIP Journal on Embedded Systems*, Volume 2007, Article ID 82123, June 2007.

[FRTOS] FreeRTOS. www.freertos.org

[GLMS02] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic, Dordrecht, 2002.

[GPY+07] X. Guerin, K. Popovici, W. Youssef, F. Rousseau, and A. Jerraya. Flexible application software generation for heterogeneous multi-processor system-on-chip. In *Proceedings of COMPSAC 2007*, 23–27 July 2007, Beijing, China, 2007.

[HAL] HAL. www.altera.com/literature/hb/nios2/n2sw_nii5v2_02.pdf

[Han06 +] S.I. Han et al. Buffer memory optimization for video codec application modeled in simulink. In *Proceedings of DAC 2006*, San Francisco, USA, pages 689–694. IEEE Press, New York, 2006.

[HYL+06] S. Hong, S. Yoo, S. Lee, S. Lee, H.J. Nam, B.S. Yoo, J. Hwang, D. Song, J. Kim, J. Kim, H. Jin, K. Choi, J.T. Kong, and S. Eo.

Creation and utilization of a virtual platform for embedded software optimization: An industrial case study. In *Proceedings of CODES+ISSS 2006*, Seoul, Korea, 2006.

[JBP06] A. Jerraya, A. Bouchhima, and F. Petrot. Programming models and HW–SW interfaces abstraction for multi-processor SoC. In *Proceedings of DAC 2006*, San Francisco, USA, pages 280–285. IEEE Press, New York, 2006.

[JW05] A. Jerraya and W. Wolf. Hardware–software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005.

[KSW+00] E.A. de Kock, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, and G. Essink. YAPI: application modeling for signal processing systems. In *Proceedings of DAC 2000*. IEEE Press, New York, 2000.

[LOS] LynxOS. www.lynuxworks.com/rtos

[Math] The MathWorks. www.mathworks.com

[MPI] MPI. www-unix.mcs.anl.gov/mpi

[mVD] magicV VLIW DSP. www.atmel.com

[newl] newlib. sourceware.org/newlib

[NS] NicheStack. www.iniche.com/nichestack.php

[NYBJ02] G. Nicolescu, S. Yoo, A. Bouchhima, and A.A. Jerraya. Validation in a component-based design flow for multicore SoCs. In *Proceedings of ISSS'02*, 2–4 October 2002, Kyoto, Japan, 2002.

[PGR+07] K. Popovici, X. Guerin, F. Rousseau, P.S. Paolucci, and A. Jerraya. Efficient software development platforms for multimedia applications at different abstraction levels. In *Proceedings of IEEE RSP 2007*, May 2007, Porto Alegre, Brazil, pages 113–122, 2007.

[PJ07] K. Popovici and A. Jerraya. Simulink based hardware-software codesign flow for heterogeneous MPSoC. In *Proceedings of SCSC 2007*, 15–18 July 2007, San Diego, USA, pages 497–504, 2007.

[Pos03] F. Pospiech. Hardware dependent software (HdS). Multiprocessor SoC aspects—An introduction. In *Proceedings of MPSoC 2003*, 7–11 July 2003, Chamonix, France, 2003.

[Row94] J.A. Rowson. Hardware/software cosimulation. In *Proceedings of DAC 1994*, San Diego, USA, pages 439–440. IEEE Press, New York, 1994.

[RTL] RTLinux. www.fsmlabs.com

[SG00] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings of ASPDAC 2000*, Yokohama, Japan, pages 405–408, 2000.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, 1995.

[Tur05] J. Turley. Survey says: Software tools more important than chips. *Embedded Systems Design Journal*, 2005.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, Englewood Cliffs, 1997.

[uIT] uITRON4.0. www.sakamura-lab.org/tron/itron

[VBL05] N. Ventroux, F. Blanc, and D. Lavenier. A low complex scheduling algorithm for multi-processor system-on-chip. In *Proceedings of Parallel and Distributed Computing and Networks*, 15–17 February 2005, Innsbruck, Austria, 2005.

[vdW04+] P. van der Wolf et al. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proceedings of CODES+ISSS 2004*, Stockholm, Sweden, pages 206–217, 2004.

[VxW] VxWorks. windriver.com/vxworks

[Wal91] G.K. Wallace. The JPEG still picture compression standard. *Communications of the ACM, Special Issue on Digital Multimedia Systems*, 34(4):30–44, 1991.

[WCE] Windows CE. www.microsoft.com/windows/embedded

[Wol06] W. Wolf. *High Performance Embedded Computing*. Morgan Kaufmann, San Mateo, 2006.

[Xte] Xtensa. www.tensilica.com

[YJ03] S. Yoo and A. Jerraya. Introduction to hardware abstraction layers for SoC. In *Proceedings of DATE 2003*, 3–7 March 2003, Munich, Germany, pages 336–337, 2003.

[YYS+04] M.W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. Jerraya. Debugging HW/SW interface for MPSoC: Video encoder system design case study. In *Proceedings of DAC 2004*, 7–11 June 2004, San Diego, USA, pages 908–913. IEEE Press, New York, 2004.

Chapter 5

HW/SW INTERFACE

Implementation and Modeling

Wolfgang Ecker, Volkan Esen, Thomas Steininger and Michael Velten

Abstract This chapter addresses HW/SW interface implementation and modeling. As introduction, basic concepts regarding HW/SW interfaces on both HW and SW side are presented in detail. The focus is on several aspects of register and bit field read/write access, address mismatch, synchronization, and data alignment. The HW micro-architecture is outlined in block diagrams, the SW code is listed in C-code snippets. As new contributions, data flow abstraction for HW/SW models and consistently derived RTL models, TLM models, and C code by using a template approach are presented.

Keywords: Address Offset, Base Address, Bit Field, C, Low/High Level Driver, Endianness, Interrupt, Register, SystemC, Template, Volatile, XML

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

5.1 Introduction

HW/SW interfaces at the lowest level deal primarily with the transfer of data from one storage element to another. These storage elements are either registers in the CPU, registers in the HW peripheral to be accessed, or cells in a memory array. By execution of data move operations—primarily by a processor but also by specific blocks as DMA (direct memory access) units—the data is transferred from one address to another. These data move operations are either coded as assembler instructions or compiled from a higher level programming language. This seemingly simple mechanism gains high complexity in today's embedded systems:

- Thousands, even tens of thousands of registers can be found in complex SoCs. Flat or hierarchical bus systems, potentially each with an own protocol, are used to access all these registers from one or more CPUs. Alignment of data is only one issue that has to be considered in this context.
- Data transfer can be initiated by polling. Alternatively, an interrupt can be executed by a CPU or a DMA (direct memory access) device. It requires careful implementation to avoid side effects and to fulfill all time constraints of all registers to be accessed.
- The number of different kinds of registers invented by designers is almost unlimited. The simple read/write of a cell can be associated with a variety of additional functionality that either constrains the read/write access or causes side effects. The driving force here is either the limitation of available addresses or performance optimization of the HW/SW interface. To give two examples: A register is cleared after it has been read in order to show that data has been consumed already or a trigger impulse is generated and passed to the hardware core of a component in order to request the execution of some algorithm.

At the software side also a simple mechanism is executed: Data is moved from one object or address to another object or address. Complexity arises here from the wide range of interpretation of the values, the effects hidden behind these data transfers, and the interaction of the data transfer with the control flow of the rest of the software. Especially the interrupt signaling mechanism shall be mentioned here, which is another HW/SW interface to signal some request from a hardware component to the software.

In order to cope with the complexities of the HW/SW interface, formal models and specifications have been developed uniquely describing the structure and semantics of the interface. Based on the formal description, parts of the hardware side of the interface and the software side of the interface can be generated.

In this chapter, we discuss first implementation issues of the HW/SW interface from the ground up. A simplified serial interface peripheral device, in the following referred to as SIF, is used throughout the chapter as an example for the various alternatives and options of the HW/SW interface. Though being simplified, this serial interface device contains all important use cases related to a general industrial HW/SW interface. The serial interface is continuously extended and the final version is described in a data sheet like—data book oriented—style (see Sect. 5.5).

Based on the serial interface device, various aspects of the HW/SW interfaces are discussed. They include reading and writing complete data words to registers, access to single bits, synchronization between HW and SW, and register address mismatch.

Finally, modeling aspects including models and meta models are discussed. An outline of further aspects concludes the chapter.

5.2 Reading and Writing Data Words

As a first step towards the HW/SW interface, full data word read and write is introduced. For that a SIFv1 is introduced having only the `TXD_REG` and `RXD_REG`. Afterwards, the flags `data_transmitted` and `data_received` are experimentally implemented, each flag as an own register.

5.2.1 General Approach

Today's most often used HW/SW interface is a so called memory mapped HW/SW interface. Here, memory elements of hardware devices are mapped into the address space of the CPU executing the software. An address decoder takes (mostly the upper) bits of the address and converts them to select signals of the memory and the peripherals to be accessed. Potentially, additional address signals are passed to the memory and the hardware devices in order to select internal memory elements. An example is depicted in Fig. 5.1.

When the select signal is active, the memory elements are read when the read signal is active, or written when the write signal is active. In modern bus systems, address, data, enable, and read/write signals may be applied synchronously in different time windows in order to enable, for instance, pipelined access. They might also be encoded differently, for instance a read-not-write signal `R_Wbar` may replace the read `RD` and write `WR` signals in presence of a bus enable signal.

When the CPU executes a read or write operation, the signal values are set in an appropriate way in order to move one word from the memory cell or a peripheral register to the CPU register, and vice versa. This read or write CPU operation may be part of instructions that move data from and to variables of a higher level programming language. In this way, a memory-mapped HW/SW

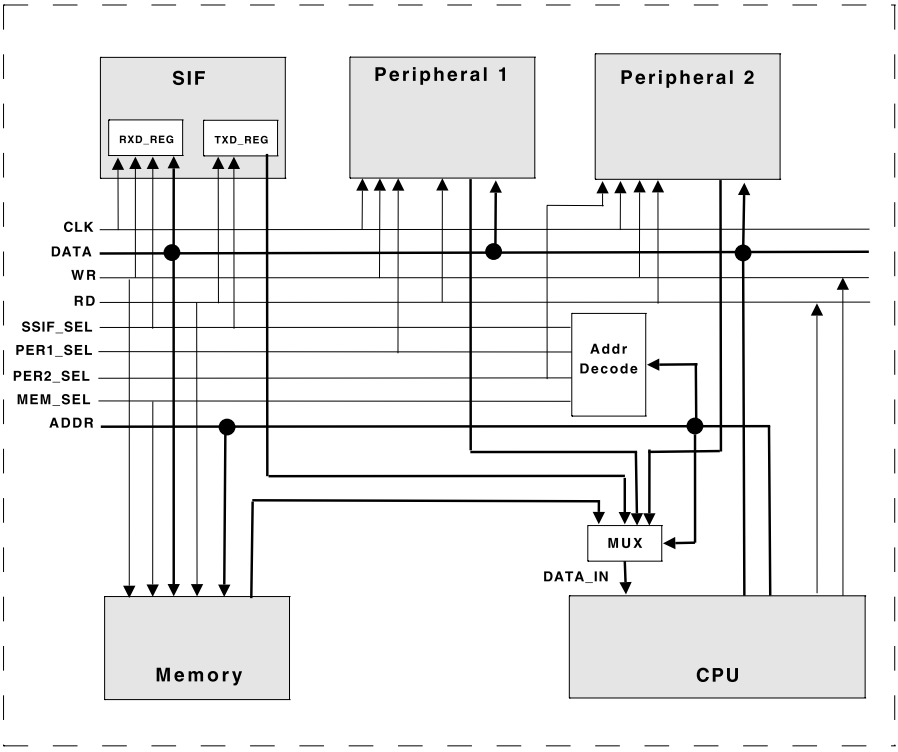

Figure 5.1. Simplified signal level connection of the SIF.

interface allows programing of a peripheral register interface using a higher level language.

As an alternative to the memory mapped HW/SW interface option, special instructions and hardware infrastructure may be provided as well. This option is often used for 8-bit CPUs, as Zilog's Z80 (see [Wik]), since the memory space is limited and shall be reserved for data memory / program memory purpose (i.e. no registers and no memory mapped I/O). As a drawback, this kind of interface has no correspondence in higher level languages, so that only assembler instructions can be used to transfer values over the interface. Higher level languages require some assembler code in-lining to support register access in this case.

In the future, also register mapped HW/SW interfaces may be introduced as a part of an I/O co-processor strategy. A first step in this direction is implemented in MIT's raw architecture (see [MIT]). Here, data can be transferred from one CPU to another by moving data to and from specific registers. Also, this kind of interface can only be programmed in assembler or with inlined assembler, since selected register access is not possible in higher level languages.

In the future, concurrent descriptions of the HW/SW interface may arise that can be used to automatically compile code for such concepts as well.

In the examples, a 32-bit CPU is assumed to be used supporting both 32 bit address size and 32 bit data size. The CPU is able to read and write bytes as well.

5.2.2 Full CPU Word Registers

As a first step, we introduce in the following two registers of the serial interface device—one can be written and one can be read. The memory space allocated by this first version is shown in Table 5.1.

To transmit data via the serial channel, this data must first be written to the `TXD` register. Data received from the serial channel can be read from the `RXD` register. This register has an offset of 0 to the base address of the peripheral as well.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32

Table 5.1. SIFv1 register overview.

Shown in Fig. 5.1 is an address decoder that computes the enable signals for all memory and peripheral devices communicating with the CPU. In the serial interface example, the address decoder takes the upper 8 bits of the address to compute 4 out of 256 possible enable signals. The byte address support of the CPU requires additional 2 address bits. So, each enabled device has available $32-8-2=22$ addresses for internal registers or memory space. So, a 16 MB memory ($4 \cdot 2^{20}$ addresses of 4 bytes each), for instance, can be enabled with one of these signals.

Assuming for the rest of this chapter, that the serial interface enable is active, when the upper 8 address bits take the value `0xFF`, the base address of the serial interface is `0xFF000000`. The memory address space reserved for the serial interface now ranges from this base address `0xFF000000` to address `0xFFFFFFFF`. In turn, the serial interface only provides memory cells for the base address `0xFF000000` and leaves the other addresses unused.

Accessing these registers from a C program can be done generally in two ways, object-based or function-based. In the first way, types and objects are declared and initialized to read and write the registers via those objects. In the second way, a function layer is introduced to allow access to the registers.

In the object-based alternative, first a pointer is declared for each register as shown in Listing 5.1.

The `volatile`-keyword in the listing gives the compiler the hint that the object may change without CPU interaction. So, an access to that object is not

```
#include <stdint.h>

volatile uint32_t *rxd_reg_ptr, *txd_reg_ptr;
```

Listing 5.1. Type and object declaration for direct register access.

removed by the optimizer of the C compiler. The type `uint32_t` is declared in `stdint.h` and specifies an unsigned 32 bit type independent from the target CPU.

The pointers are then initialized with the base address of the serial interface as shown in Listing 5.2. The distinction between the rxd-register and the txd-register is done in hardware via the read and write signal. Options herefore are discussed later in this chapter.

```
rxd_reg_ptr = (volatile uint32_t *) 0xFF000000;
txd_reg_ptr = (volatile uint32_t *) 0xFF000000;
```

Listing 5.2. Pointer initialization for direct object access.

The base address in these functions can be replaced by symbolic names—either constants or macros—, which will be shown in Listing 5.3. By doing so, the addresses can also be set via compile options.

```
#define SIF_BASE_ADDRESS 0xFF000000

rxd_reg_ptr = (uint32_t *) SIF_BASE_ADDRESS;
txd_reg_ptr = (uint32_t *) SIF_BASE_ADDRESS;
```

Listing 5.3. Pointer initialization for direct object access with symbolic values.

To preserve type consistency, the integer literal is cast to the register pointer type. Finally, transmitted data can then be accessed by dereferencing the pointers as shown in Listing 5.4. The type of the variable `rx` is assumed to be `uint32_t`. The integer literal is cast to that value.

```
/* reading a value from serial stream */
rx = *rxd_reg_ptr;

/* writing a value to the serial stream */
*txd_reg_ptr = (uint32_t) 0x12345678;
```

Listing 5.4. Accessing the SIF register.

In the function-based alternative, first types and access functions are declared, as shown in Listing 5.5. In this simple case, these functions contain

exactly the statements of the object-based access alternative. These functions can then be called—which is not shown in the code snippet—to transmit data via the SIFv1.

```
void transmit(uint32_t data) {
   *(0xFF000000) = data;
}

uint32_t receive() {
   return (uint32_t) *(0xFF000000)
}
```

Listing 5.5. Type and function declaration for data transmission.

Two further coding options are in use for accessing registers. The first one uses macros instead of functions. This is more efficient, if the compiler does not support function inlining optimizations. The other option uses classes, class variables, and class methods to access the peripheral registers. Here, the access to the registers of the serial peripheral device can be controlled more efficiently (e.g., via private and public access rights, or via additional checks), but the C++ compiler is mostly not able to produce as efficient a code as the C-compiler, since the overhead caused by the class-based approach is often not eliminated.

5.2.3 Registers Storing One Bit Each: A First Approach to Bit Fields

It is quite obvious that a two register interface so far only works correctly if the hardware read/write protocol blocks the read and write transactions until they have been successfully finished. This means in the serial interface device case, the read is blocked until a piece of data is successfully received from the serial stream, and a write is blocked until a piece of data has been correctly transmitted to the serial stream. This also blocks the CPU and prevents it of from performing other activities. This is no ideal solution!

To avoid blocking the execution of other parts of SW, the SW must be able to check if the serial interface can transmit further data, or if the serial interface has received new data that can be read. Two additional registers, each storing only zero or one, can offer this information to the software. The device is now called SIFv2.

The interface of the SIFv1 extends as shown in Fig. 5.2 and Table 5.2.

Two additional, readable registers are introduced that also require two additional address lines for distinction. A multiplexer is used to internally select the appropriate register value. For bus accesses, the SIFv2 now has one writable register and three readable registers. The ready-for-transmission register has an offset of 1, and the data-available register has an offset of 2.

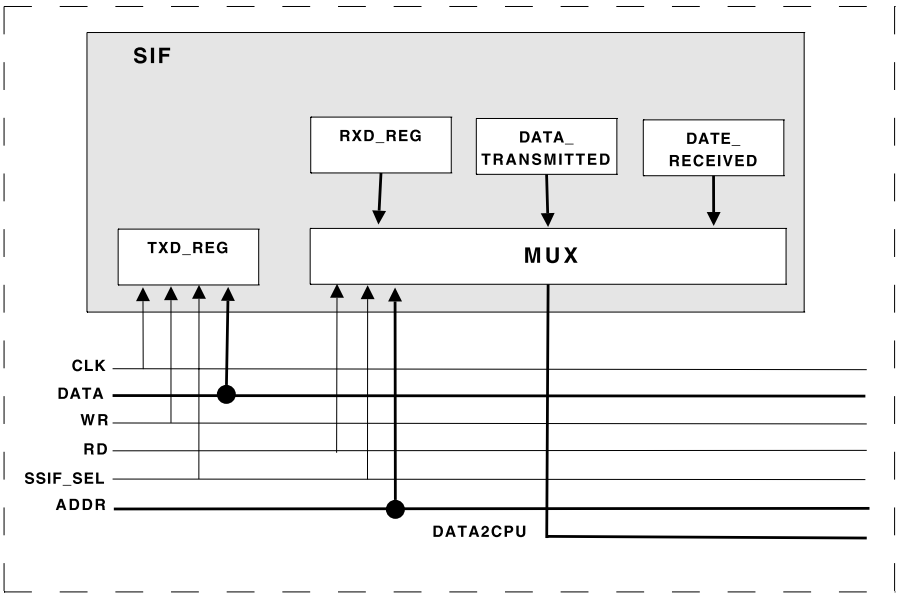

Figure 5.2. Simplified signal level register access.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_TRANSMITTED	R	1	32	32
SIF_RECEIVED	R	2	32	32

Table 5.2. SIFv2 register overview.

For this purpose, the SW interface for readable registers must be extended as well. As an initial solution, two more register pointers are declared and initialized, as shown in Listing 5.6.

The offset of the new registers is hard-coded in each pointer initialization and added here to the base address. Since the CPU supports byte access, the increment between two registers is 4.

The read access to the new status registers is similar to the received data as shown in Listing 5.4.

Listing 5.7 shows a more elegant and more frequently used option. Here, a C struct is used to describe all registers. Since each of the entries in the C struct has the size of the CPU word, the address of each entry automatically increases by 4. The explicit increment of addresses, as used in Listing 5.6, is not necessary here.

```
volatile uint32_t *rxd_reg_ptr,
                  *txd_reg_ptr,
                  *data_transmitted_ptr,
                  *data_received_ptr;

/* Initialize references to readable registers */
rxd_reg_ptr          = (uint32_t *) 0xFF000000;
data_transmitted_ptr = (uint32_t *) 0xFF000004;
data_received_ptr    = (uint32_t *) 0xFF000008;

/* Initialize references to writable registers */
txd_reg_ptr          = (uint32_t *) 0xFF000000;
```

Listing 5.6. Type and object declaration for additional flags.

```
volatile uint32_t *txd_reg_ptr;

struct r_reg_t {
   volatile uint32_t rxd_reg;
   volatile uint32_t data_transmitted;
   volatile uint32_t data_received;
} *r_reg_ptr;
```

Listing 5.7. Type and pointer declaration for data transmission.

In order to access the registers, the dereferencing mechanism of C is used, as shown in Listing 5.8. Since a pointer is used to refer to the registers, the dereferencing operator $->$ is used.

```
/* reading a value from serial stream */
rx = r_reg_ptr -> rxd_reg;

/* read flags */
ready_for_transmission = r_reg_ptr -> data_transmitted;
data_available         = r_reg_ptr -> data_received;
```

Listing 5.8. Object access to flag registers.

```
uint32_t is_ready_for_transmission() {
   return (uint32_t) *(0xFF000000)[1];
}

uint32_t is_data_ready() {
   return (uint32_t) *(0xFF000000)[2];
}
```

Listing 5.9. Function access to flag registers.

The function-based approach—to show an alternative access in Listing 5.9—makes use of the C index operator. Beginning with the base address of the serial interface peripheral, the `rxd_reg` and the transmission flags can be accessed by indices 0, 1, and 2, respectively. The compiler converts this to an address increment of the size of referenced elements in byte. In our case, the size is 32 bits or 4 bytes. So, the index of the transmission flags are addressed under `0xFF000004` or `0xFF000008`.

Also a mix of both approaches—object-based register access and method-based register access—is possible. Doing so, the data structures of the object-based access are used by the functions to implement the access. As benefit over pure object-based access, the functions provide an additional layer, which hides future HW changes from higher level software. Also additional sanity checks are possible to be embedded here.

Registers holding transmission status and other things often use just one bit, or a few bits. This is waste of address space but potentially acceptable if a sufficient number of address lines, that means a sufficiently large address space, is available. However, peripheral register accesses cause performance penalties. Merging several of those bits efficiently in one register can reduce the number of register accesses (e.g. for peripheral configuration, or check) and thus improve performance.

Further on, it is quite error prone that readable and writable registers have a different address size and information located at different addresses. It is more clear and safe to have one linear address space only, which in turn would allow having one C struct specifying all registers—the readable registers, the writeable registers and the read/writable registers—at the SW side. Both of these issues are discussed in the next section in more detail.

5.3 Bit Fields

The need for bit fields sharing one register has been introduced in the previous section. Now, different aspects of the bit field based interface are discussed. It is shown that in this application registers lose their role as memory element and take the role of a shell accessed under a specific address. In other words, registers take the role of an alias.

Bit fields—now representing the memory elements—are associated with that shell, which specifies their word access from the CPU. Internal bit offsets, as shown for the serial interface in Table 5.9, are used for bit addressing in the data word. Besides the data registers, txd-register and rxd-register, and the bit fields `data_transmitted` and `data_received` also the bit fields controlling the parity bit (`tx_enable_parity`, `tx_odd_parity`, `rx_enable_parity`, and `rx_odd_parity`) are introduced.

5.3.1 Introducing Bit Fields

In order to introduce bit fields, a third version of the serial interface SIFv3, is introduced. The registers are shown in Table 5.3.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_FLAG_REG	R	1	32	32

Table 5.3. SIFv3 register overview.

Readable Bit Fields Instead of having two separate registers storing the two flags `data_transmitted` and `data_received`, one common flag register is used. This flag register stores the `data_transmitted` flag at bit number 0, and the `data_ received` flag at bit number 1.

All other bits with numbers from 2 to 31 are unused. Since no hardware resources need to be spent to store the unused bits, the bit fields (and not the registers) specify the hardware size and properties of the required flipflops or registers. Reading from those bits cannot be avoided, since the CPU always reads a full 32 bit word. Depending on the specification, an undefined value or a constant, for instance 0 as used in the example, is returned for each of those bits. The updated diagram of the serial interface is shown in Fig. 5.3.

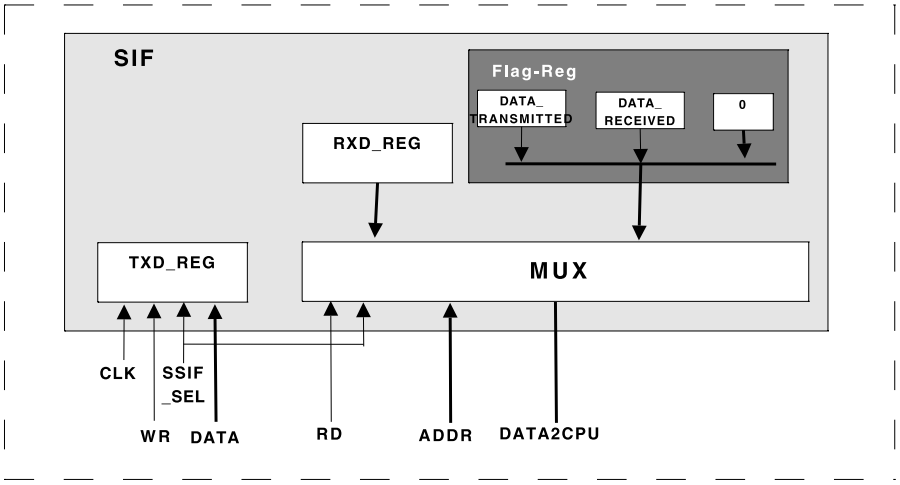

Figure 5.3. Registers with readable bit fields.

To access these flags, so-called bit fields in C are used. First, as shown in Listing 5.10, a C struct specifies the flags and unused bits of one register. Next,

a struct is defined that specifies the readable registers: the `rxd_reg_data` register and the `flag_reg` flag register.

```
volatile uint32_t *txd_reg_ptr;

struct flag_register_t {
   volatile uint32_t data_transmitted : 1;
   volatile uint32_t data_received :1;
   const    uint32_t unused :30;
}

struct r_reg_t {
   volatile uint32_t          rxd_reg;
   volatile flag_register_t   flag_reg;
} *r_reg_ptr;
```

Listing 5.10. Type and object declaration for bit field based register access.

The bit fields are then accessed with the dereferencing operator ->, as shown in Listing 5.11. The bit fields holding the flags are selected as struct elements in the corresponding register using the dot operator. The way of reading a full data word from a register remains unchanged.

```
/* reading a value from serial stream */
rx = r_reg_ptr -> rxd_reg;

/* read flags */
ready_for_transmission = r_reg_ptr -> flag_reg.data_transmitted
    ;
data_available         = r_reg_ptr -> flag_reg.data_received;
```

Listing 5.11. Register flag access via C bit fields.

A function-based coding option is shown in Listing 5.12. Instead of using a pre-initialized pointer, which would have been possible here as well, hard-coded addresses are used. The data register is read without offset, the flag register is read via base address and index offset 1. The compiler, knowing the address scheme of the CPU, translates this to the address `0xFF000004`. In case of bits not residing at bit position 0, the bit fields themselves are then extracted via masking the unused bits by performing a bitwise `and` operation with a corresponding bit mask. Potentially—as needed for the `data_ready_flag`—the bits have to be right aligned upfront by using the C shift right operation.

Writable Bit Fields In order to write configuration information to the serial interface, an additional writable register with bit fields is introduced as SIFv4 in Fig. 5.4 and Table 5.4.

```
uint32_t is_ready_for_transmission() {
   return (uint32_t) (*(0xFF000000)[1]) & 0x00000001;
}

uint32_t is_data_ready() {
   return (uint32_t) ((*(0xFF000000)[1]) >> 1) & 0x00000001;
}
```

Listing 5.12. Register flag access via C shift and logical operators.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_FLAG_REG	R	1	32	32
SIF_CFG_REG	W	1	32	32

Table 5.4. SIFv4 register overview.

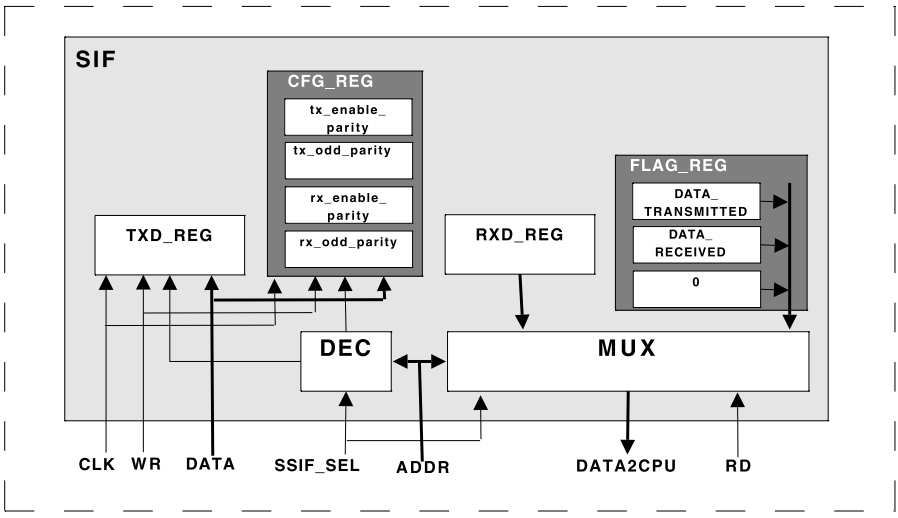

Figure 5.4. Registers with writable bit fields.

A decoder is inserted here in order to decode address values to register select signals.

None of these bit fields can be written individually, so writing to one bit field automatically modifies the other bit fields. However, the configuration register—and so the bit fields—cannot be read, since reading address 1 gets the value of the flag register.

For this purpose, as a first solution, a shadow variable is introduced that holds the last written value to the configuration register. Setting or clearing

a bit field is then first performed on the shadow variable, and afterwards the shadow variable is written to the register.

```
struct confi_reg_t {
    volatile uint32_t tx_enable_parity   : 1;
    volatile uint32_t tx_odd_paroity     : 1;
    volatile uint32_t tx_enable_parity   : 1;
    volatile uint32_t tx_odd_paroity     : 1;
    volatile uint32_t                    : 28;
}

struct w_reg_t {
    volatile uint32_t           txd_reg;
    volatile config_reg_t       config_reg;
} *w_reg_ptr;
```

Listing 5.13. Type and object declaration for writing configuration bit fields.

```
/* shadow variable for configuration register with bitfields */
static config_reg_t config_reg_shadow;

/* write flags via shadow variable */
config_reg_shadow.tx_enable_parity = 1;
w_reg_ptr -> config_reg = config_reg_shadow;

/* writing a value to serial stream */
w_reg_ptr -> txd_reg = 0x12345678 ;
```

Listing 5.14. Flag configuration via C bit fields.

As shown in Listing 5.13, structs for written bit fields are declared in the same way as for read bit fields. The use of the above mentioned shadow variable is shown in Listing 5.14. Since setting a bit field with one statement is in this case no longer possible, the use of functions for setting and clearing the bits, as shown in Listing 5.15, is the preferable solution. Besides encapsulation of the shadow register handling, additional checks may be included as well.

Listing 5.15 shows several alternatives how access functions for registers and bit fields can be built. The function `write_cfg_tx_enable_parity` sets the bit fields via bit wise `or` operation. As an alternative, the C bit fields and the structs defined for object based register access can be used as shown in function `write_cfg_ tx_odd_parity`. Without them, a bit wise `and` operation with `0xFFFFFFFE` would have been used (bit masking of the least significant bit). Finally, the function `write_cfg_tx_odd_ parity` combines two flag accesses on the shadow variable before writing it to the registers. This improves performance, since the shadow variable needs to be written to

```
/* shadow variable for configuration register with bitfields */
static config_reg_t config_reg_shadow;

void write_cfg_tx_enable_parity() {
   config_reg_shadow |=  0x00000001;
   *(0xFF000000)[1] = config_reg_shadow;
}

void write_cfg_tx_odd_parity() {
   config_reg_shadow.tx_odd_parity =  1;
   w_reg_ptr -> config_reg = config_reg_shadow;
}

void set_rx_parity(bool odd_parity) {
   config_reg_shadow.rx_odd_parity =  odd_parity;
   config_reg_shadow.rx_enable_parity = 1;
   w_reg_ptr -> config_reg = config_reg_shadow;
}

void clear_rx_parity() {
   config_reg_shadow.rx_enable_parity = 0;
   w_reg_ptr -> config_reg = config_reg_shadow;
}
```

Listing 5.15. Register flag access via C shift and logical operators.

the register only once, but semantic information about the content of the bit fields is required in this case.

Mixed Readable and Writable Bit Fields Conceptually, redundancy as existent in the configuration register and its shadow variable is a source of bugs. Furthermore, shadowing provides no appropriate solution if a bit field is also modified by peripheral internal actions. Hence, in a new version SIFv5, the writable bit fields shall be readable as well.

Therefore, the readable bit fields in the flag register are extended by the writable bit fields as shown in Listing 5.16 and Table 5.5. In order to make them read/writable, the write must be triggered by CPU write to the appropriate address, and the value must be multiplexed to the output by a CPU read.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_FLAG_REG	RW	1	32	32

Table 5.5. SIFv5 register overview.

```
struct flag_reg_t {
   volatile uint32_t data_transmitted   : 1;
   volatile uint32_t data_received      : 1;
   volatile uint32_t tx_enable_parity   : 1;
   volatile uint32_t tx_odd_paroity     : 1;
   volatile uint32_t tx_enable_parity   : 1;
   volatile uint32_t tx_odd_paroity     : 1;
   volatile uint32_t                    : 26;
}

struct r_reg_t {
   volatile uint32_t      rxd_reg;
   volatile flag_reg_t    flag_reg;
} *r_reg_ptr;

struct w_reg_t {
   volatile uint32_t           txd_reg;
} *w_reg_ptr;
```

Listing 5.16. Readable and writable register with flags.

Having this structure available, the flags can be changed consistently, as shown in Listing 5.17.

In function `write_cfg_tx_enable_parity`, first the current values of the flags are read from the status register, then aligned, and next assigned to the temporary shadow variable `config_reg_shadow`. Afterwards, the flag is set in the temporary variable. Finally, the updated flags stored in the temporary variable are assigned to the bit fields in the register. The temporary variable `config_reg_tmp` is not necessary, the flag update might have been done in one expression alone.

```
void write_cfg_tx_enable_parity() {
   config_reg_t config_reg_tmp = (config_reg_t)
      (((*0xFF000000)[1]) >> 2) & 0x0000000F;
   config_reg_tmp |=  0x00000001;
   *(0xFF000000)[1] = config_reg_tmp;
}
```

Listing 5.17. Bit field update by reading and writing a complete register.

Access control for bit fields Another alternative, SIFv6, to overcome shadow registers is the introduction of additional bits in combination with new coding as it is shown in Table 5.6.

For one flag, generally three options are possible when adding an additional bit for control:

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_CONFIG_REG	W	1	32	32

Table 5.6. SIFv6 register overview.

- The two bits in the register act as RS signals to the flipflop storing the flag. If "`00`" is assigned, then nothing changes, if "`01`" is assigned, then the flipflop, i.e. the flag, is set, and if "`10`" is assigned, then the flipflop is cleared. The assignment "`11`" is illegal.
- The two bits in the register act as JK signals to the flipflop. As expected from the JK-flipflop behavior, this has the same behavior as the RS option, except the case "`11`" is legal and forces the toggle of the flag.
- One bit in the register acts as new value for the flag and the other as enable. This alternative can also be used to enable several bits in parallel, for instance, to write a configuration value that has more than two alternatives.

It is obvious to say that control in the options above may be coded in a different way, for instance in a low active way. Having enable and data in different registers is also an option but requires three register accesses: enable–write–disable.

Listing 5.18 shows the application of the third alternative to the parity configuration bits. Each bit, marked by the name suffix `_val` is guarded by an enable bit with the name suffix `_en`.

```
struct config_reg_t {
   volatile uint32_t tx_enable_parity_val : 1;
   volatile uint32_t tx_enable_parity_en  : 1;
   volatile uint32_t tx_odd_parity_val    : 1;
   volatile uint32_t tx_odd_parity_en     : 1;
   volatile uint32_t rx_enable_parity_val : 1;
   volatile uint32_t rx_enable_parity_en  : 1;
   volatile uint32_t rx_odd_parity_val    : 1;
   volatile uint32_t rx_odd_parity_en     : 1;
   volatile uint32_t unused               : 24;
}

struct w_reg_t {
   volatile uint32_t          txd_reg;
   volatile config_reg_t      config_reg;
} *w_reg_ptr;
```

Listing 5.18. Bit field access via access control.

The flags can now be easily set or reset, as shown in Listing 5.19. Here, the `tx_ enable_bit` is set and the `tx_odd_parity` is cleared. The bit fields configuring the RX path are not changed, since the corresponding enable bits are not set.

```
/* enable parity in the transmission path */
w_reg_ptr -> config_reg = 0X00000003;

/* set odd parity in the transmission path */
w_reg_ptr -> config_reg = 0X00000008;
```

Listing 5.19. Register bit field access with guarded value access.

Unfortunately, the bit fields in C cannot be used since two of them cannot be assigned in one statement. For this reason, it is better to define the two bits—one for the value and one for the enable—in one entry of the struct. Making use of such a type definition, as shown in Listing 5.20, finally allows to set or clear the bit with one statement, as shown in Listing 5.21. Here, also the symbolic names `set_bit`, `clear_bit`, and `keep_bit` are defined as constants. Macros are also an option here.

```
constant uint32_t set_bit   : 2 = 3;
constant uint32_t clear_bit : 2 = 1;
constant uint32_t keep_bit  : 2 = 0;

struct config_reg_t {
   volatile uint32_t tx_enable_parity : 2;
   volatile uint32_t tx_odd_parity    : 2;
   volatile uint32_t rx_enable_parity : 2;
   volatile uint32_t rx_odd_parity    : 2;
   volatile uint32_t unused           : 24;
}

struct w_reg_t {
   volatile uint32_t          txd_reg;
   volatile config_reg_t      config_reg;
} *w_reg_ptr;
```

Listing 5.20. Bit field via dual-bit access control.

```
/* enable parity in the transmission path */
w_reg_ptr -> config_reg.tx_enable_parity = set_bit;

/* set odd parity in the transmission path */
w_reg_ptr -> config_reg.tx_odd_parity = clear_bit;
```

Listing 5.21. Register flag access via enabled setting of bit fields.

If the bit fields are modified by the peripheral core as well and new values have an influence on the new settings of the bit fields, then they must be made readable as well.

A structured approach to Bit Fields. All options shown above have both advantages and disadvantages. Avoiding side effects partially is not possible, if bit fields are changed by the peripheral core; bit fields have to be specified twice in the read registers and write registers, in order to allow reading and writing them; reading and writing bit fields might have to be done at different addresses or bit offsets; re-assignment of bit sets might be needed.

For this reason, we propose to reserve one address for each register, independently if it is written, read, or read and written. The read and write signal is no longer used to distinguish bit fields. Furthermore, we request, that registers that are written by software, can also be read by software.

This implies that multiplexers and decoders must be implemented here in such a way that values and bit fields can be accessed under the same address and bit position for read and write access.

The application of this methodology to an industrial style peripheral is shown after discussion of impact of bus infrastructure and a textual specification of the serial interface peripheral later on.

5.4 Register Address and Data Mismatch

In this section, advanced topics on the HW/SW interface are discussed. They deal mostly with not having a bijective mapping between CPU address space and peripheral address space. To give examples, the IP may have holes in the data and address space, multiple registers may be accessed under one address, or one register may be accessed under multiple addresses.

5.4.1 Hierarchical Bus

In order to relate the advanced topics to real architecture structures, our bus is extended in direction of a hierarchical bus (see Fig. 5.5). The bus is split in a CPU bus—mostly a high speed bus—and a peripheral bus—mostly a slower bus with potentially less address signals and data signals.

As interface between these buses, a so called bridge is introduced. In the example, the bridge is selected by the same signals as formerly the serial interface unit. Since the bridge introduces a new hierarchy in the global address map, the addressing scheme so far needs to be updated slightly. Thus, the first 8 address bits are now used to select devices connected to the CPU bus. The bridge to the peripheral bus can be considered as such a device. The remaining 24 address lines are passed to the bridge. The upper 8 lines (lines 24 to 16)

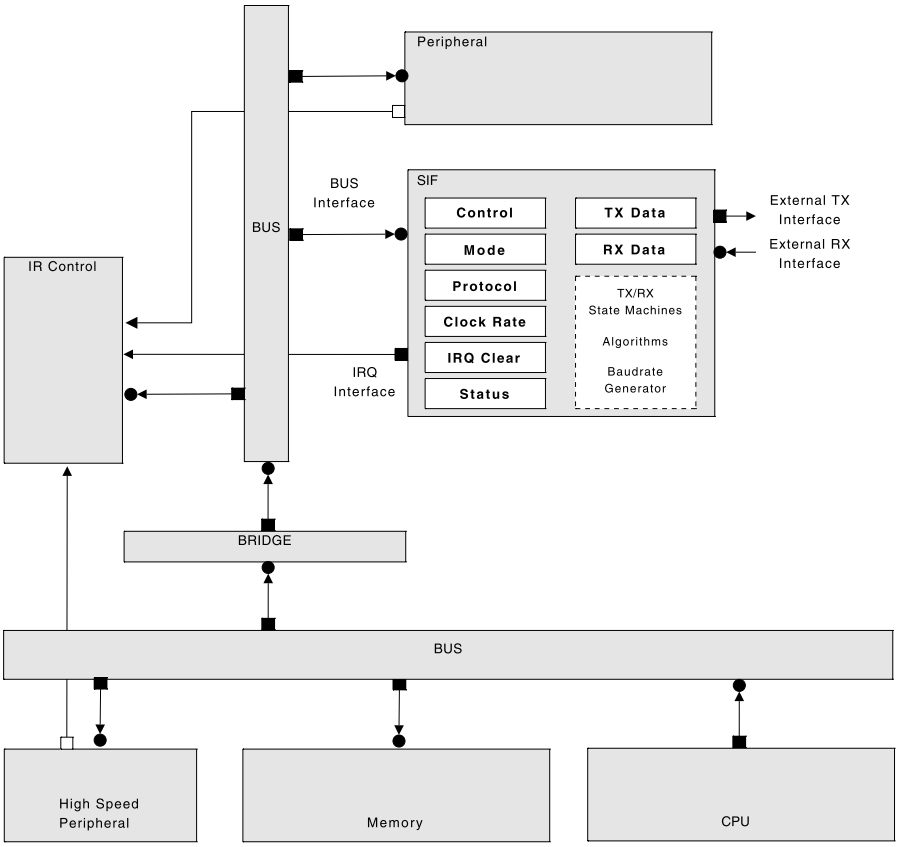

Figure 5.5. Simple hierarchical bus system.

are decoded within the bridge in order to select a specific device (i.e., peripheral) connected to the peripheral bus. With regard to these 8 address bits, the address of the serial interface unit is now assumed to be `0xFF`. Due to this adapted address map, the new base address of the serial interface unit from the CPU perspective is `0xFFFF0000`. This value is the compound of the base address of the bus bridge (i.e., `0xFF000000`) and the base address of the serial interface unit within the peripheral bus (i.e., `0xFF0000`). This yields 16 bits for addressing resources within the serial interface peripheral. Taking into account that the lower two bits are reserved for byte addressing, 14 bits remain for addressing internal registers of the serial interface which restricts the maximum number of addressable 32-bit registers of the serial interface unit to $16 \cdot 2^{10}$—still a more than sufficient number.

5.4.2 Byte Addressing

Most embedded 32 bit CPUs also support half-word and byte addressing. In this case, not only 32 bits can be accessed at once, but also just 16 bits and 8 bits. Byte addressing with 32 bit wide RXD registers and TXD registers does not provide any benefit, in fact, it makes the access worse. To show this case, a SIFv7 is introduced in Table 5.7. Here, each register has its own address. The addressable unit has changed from 32 bit to 8 bit.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	8
SIF_RXD_REG	W	1	32	8
SIF_CONFIG_REG	W	2	32	8
SIF_FLAG_REG	R	3	32	8

Table 5.7. SIFv7 register overview.

```
struct flag_register_t {
    volatile unsigned data_transmitted : 1;
    volatile unsigned data_received :1;
    volatile unsigned :6;
}
struct config_reg_t {
    volatile unsigned tx_enable_parity : 1;
    volatile unsigned tx_odd_parity    : 1;
    volatile unsigned rx_enable_parity : 1;
    volatile unsigned rx_odd_parity    : 1;
    volatile unsigned unused           : 4;
}

struct reg_t {
    volatile uint32_t          rxd_reg;
    volatile uint8_t           txd_reg[4];
    volatile config_reg_t      config_reg;
    volatile flag_register_t   flag_reg;
} *reg_ptr;
```

Listing 5.22. Struct for byte addressing.

Listing 5.22 shows a data structure definition that matches with byte addressing. The flag register and configuration register is filled up to 8 bits only. The 32 bit data is now represented by a 4 element array of the data type `uint8_t`. This `uint8_t` is also taken from the `stdint.h` include file. If 4 byte alignment of the structs is ensured—as is the case in the example—also the type `uint32_t` can be used instead of the 4 element array. In both cases, the data word can be written at a glance using CPU word access—but only by using ugly casting.

If 4 byte alignment of the structs cannot be guaranteed—as would be if the 8 bit flag register preceded the 4x 8 bit data register—the data word must be assigned byte by byte, which would cause a 4x overhead in writing the data. Both, flag register and config register can be accessed in any case by a byte access without any penalty.

If only 8 bit data registers were assumed—now in SIFv8—the access would become simpler. This is shown in Table 5.8 and Listing 5.23.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	8	8
SIF_RXD_REG	W	1	8	8
SIF_CONFIG_REG	W	2	8	8
SIF_FLAG_REG	R	3	8	8

Table 5.8. SIFv8 register overview.

```
struct flag_register_t {
    volatile uint32_t data_transmitted : 1;
    volatile uint32_t data_received :1;
    volatile uint32_t :6;
}
struct config_reg_t {
    volatile uint32_t tx_enable_parity : 1;
    volatile uint32_t tx_odd_parity    : 1;
    volatile uint32_t rx_enable_parity : 1;
    volatile uint32_t rx_odd_parity    : 1;
    volatile uint32_t unused           : 4;
}

struct reg_t {
    volatile uint8_t          rxd_reg;
    volatile uint8_t          txd_reg;
    volatile config_reg_t     config_reg;
    volatile flag_register_t  flag_reg;
} *reg_ptr;
```

Listing 5.23. Struct for byte addressing with byte data words.

As a drawback, only 8 bit values can be transmitted. Also (not shown), the hardware architecture must be changed accordingly.

5.4.3 Endianness

Mix of byte access and word access becomes challenging if different orders of bytes are implemented on the CPU side and on the serial interface side. This implementation aspect, also known as endianness, requires additional adaptation effort, either on the HW side, or on the SW side.

Best known are big endian, the numeric option, where the most significant byte comes first, and little endian, the literal option, where the most significant byte comes last. The little endian option also has the advantage of that a byte value followed by three zero byte values is read in the same way in case of byte, half-word, and word access.

Also known is a middle endian or mixed endian option (16 bit), where the most significant half word comes last and the least significant half word comes first. The advantage of this version is the seamless support of 16 bit encoded characters.

The easiest and most efficient adaptation of different endian encodings is on the hardware side. First, the peripheral can have a generic parameter, that allows to statically reconfigure a peripheral interface to big endian or little endian. This allows avoiding the remapping effort since the endianness of the CPU and the peripheral can be made identical for the cost of rewiring some signals, in other words, for free. A remapping of the bytes at the ports of the peripheral has the same effect. Also an easy and quite efficient adaptation is the use of CPUs that can be dynamically reconfigured to support big endian or little endian reading and writing of 4 byte words. This costs some hardware overhead but has the advantage that a mix of big endian and little endian peripherals can be supported as well.

Quite an overhead must be spent to adopt endianness in software. Listing 5.24 shows a possible implementation.

```
BE = ( ((LE)>>24)          | (((LE)&0xff0000)>>8) |
       (((LE)&0xff00)<<8)  | ((LE)<<24) )
```

Listing 5.24. Expression for converting little endian to big endian.

Sometimes—mainly in context of serial transmission—the term endianness is also used in conjunction with bits. In this case, the terms byte endianness and bit endianness are applied to make a distinction. Byte endianness is the classical endianness as described above. Bit endianness relates to bit orientation and defines if inside one byte, the most significant bit is first and the last significant bit is last, or vice versa.

5.4.4 Busses with Different Data Width

A hierarchical bus system also allows busses with different data widths, for instance a CPU bus size of 32 bit and a peripheral bus size and register size of 8 bit each. All presented codings shown above can be kept if the bus bridge serializes word access on the CPU side into 4 byte accesses—of course considering the right endianness.

Another option is to use the lower byte of the CPU word only and ignore the higher three bytes. In this case, a lot of unused bytes have to be filled into the struct representing the registers in software. This is shown in Listing 5.25.

```
struct flag_register_t {
   volatile uint32_t data_transmitted : 1;
   volatile uint32_t data_received :1;
   volatile uint32_t :6;
}
struct config_reg_t {
   volatile uint32_t tx_enable_parity : 1;
   volatile uint32_t tx_odd_parity    : 1;
   volatile uint32_t rx_enable_parity : 1;
   volatile uint32_t rx_odd_parity    : 1;
   volatile uint32_t unused           : 4;
}

struct unused_upper_bytes_t{
   volatile uint32_t : 24;
}

struct reg_t {
   volatile uint8_t              rxd_reg;
   volatile unused_upper_bytes   unused1;
   volatile uint8_t              txd_reg;
   volatile unused_upper_bytes   unused2;
   volatile config_reg_t         config_reg;
   volatile unused_upper_bytes   unused3;
   volatile flag_register_t      flag_reg;
   volatile unused_upper_bytes   unused4;
} *reg_ptr;
```

Listing 5.25. Struct for LSB byte addressing with byte data words.

5.4.5 Several Registers Share One Address

Not only read and write signals are used to reduce consumed address space of a peripheral. Also other techniques have been invented to extend the number of addressable registers and bit fields. These techniques are also heavily used in 8 bit CPU systems but lose their importance as more and more address space, for instance in 32 bit CPUs, becomes available.

Auto-shadow is the first technique to be discussed. Here, one register is visible at a specific address after hardware or software reset. After having written to such a register, the register hides behind another register that can be accessed instead. This is mostly applied for configuration registers, since they are mostly configured once. On the software side, these two registers share their address by being modeled as a union, as it is shown in Listing 5.26.

```
struct confi_reg_t {
    volatile uint32_t tx_enable_parity   : 1;
    volatile uint32_t tx_odd_paroity     : 1;
    volatile uint32_t tx_enable_parity   : 1;
    volatile uint32_t tx_odd_paroity     : 1;
    volatile uint32_t unused             : 28;
}

union w_reg_t {
    volatile uint32_t           txd_reg;
    volatile config_reg_t       config_reg;
} *w_reg_ptr;
```

Listing 5.26. Type and object declaration for writing hidden configuration bit fields.

Writing these registers does not distinguish between the union or struct version. The dot-operator is used in both cases. However, it must be ensured that the hidden register is accessible when needed.

Another technique is the use of indexing bit fields. This can be done by interpreting one part of a register as a value and the other as index. Also, value and index can come from different registers. Writing to a register requires in the first case to merge index and data into one word—comparable with merging bit field information as discussed above. Writing to a register in the second case requires an overhead of two write accesses, which may be acceptable if the index, that means the bit field to be written to, changes only infrequently.

5.4.6 One Register is Accessible via Several Addresses

This is exactly the opposite hardware implementation approach as described in the section above. Here, one register or bit field can be accessed under more than one address in the peripheral's address space. Since this technique allocates more addresses in the address space than absolutely necessary, it finds its application more in 32 bit CPUs.

The first reason for such a technique is compatibility with older versions. So to say, an alias to the old address is preserved. In the SW side of the interface, two registers are specified in the register struct, but using different names.

The second, and probably more important reason is the support of burst or block transfers by the bus protocol. This burst transfer moves blocks from one address to another—or the cache. To apply this protocol, for instance to send data stored in the memory via the serial interface, the txd-register must be accessible under a range of addresses. The register struct can be easily

```
struct flag_register_t {
   volatile uint32_t data_transmitted : 1;
   volatile uint32_t data_received :1;
   volatile uint32_t unused :30;
}
struct config_reg_t {
   volatile uint32_t tx_enable_parity  : 1;
   volatile uint32_t tx_odd_paroity    : 1;
   volatile uint32_t tx_enable_parity  : 1;
   volatile uint32_t tx_odd_paroity    : 1;
   volatile uint32_t unused            : 28;
}

struct reg_t {
   volatile uint32_t          unused1;
   volatile uint32_t          unused2;
   volatile config_reg_t      config_reg;
   volatile flag_register_t   flag_reg;
   volatile uint32_t          rxd_reg[32];
   volatile uint32_t          txd_reg[32];
   volatile config_reg_t      config_reg_alias;
   volatile flag_register_t   flag_reg_alias;
} *reg_ptr;
```

Listing 5.27. Type and object declaration for writing hidden configuration bit fields.

extended by putting an array—here of size 32—instead of a single element to the register struct. This is also known as mirror size 32.

Listing 5.27 shows exemplarily how alias registers and memory space for allocation of data registers have an impact on the register struct. First, two unused registers are introduced to keep the flag register and config register at their corresponding places. After the config registers follows the allocated space—implemented as arrays as described above—and finally the alias registers.

5.4.7 Multiple SIF-Peripherals in One System

If more than one serial interface has to be used in a system, several base addresses have to be served from the software point of view. This can be easily achieved by declaring and initiating two register pointers.

To access those peripherals, either the code has to be duplicated, for instance in object based access, or a register pointer has to be passed to the functions accessing the serial interfaces. The latter case is shown in Listing 5.28.

```
void write_cfg_tx_enable_parity(reg_t *reg_ptr) {
   reg_ptr -> config_reg |= 1;
}
```

Listing 5.28. Setting and clearing bit fields with reading.

To call the functions, the struct declaration of Listing 5.22 is needed. As pointers, now a `sif1_reg_ptr` and a `sif2_reg_ptr` must be declared (not shown in a listing). These pointers must be initialized with the serial interfaces' base addresses before they can be used in a call to those functions.

5.5 Textual Specification of the SIF

The following section gives an overview of the SIF peripheral. The hardware interfaces are explained, followed by the description of its registers and bit fields and their specification. The registers including their bit fields are the basic building blocks of the HW side of the HW/SW interface of the peripheral.

5.5.1 Overview

The SIF model can be used for connecting two hardware systems via a serial communication protocol.

Figure 5.6 shows the basic structure of the SIF peripheral model. The SIF contains four different hardware interfaces—the bus interface, the external interfaces, and an interface for interrupts. A detailed description of these interfaces will be given later on.

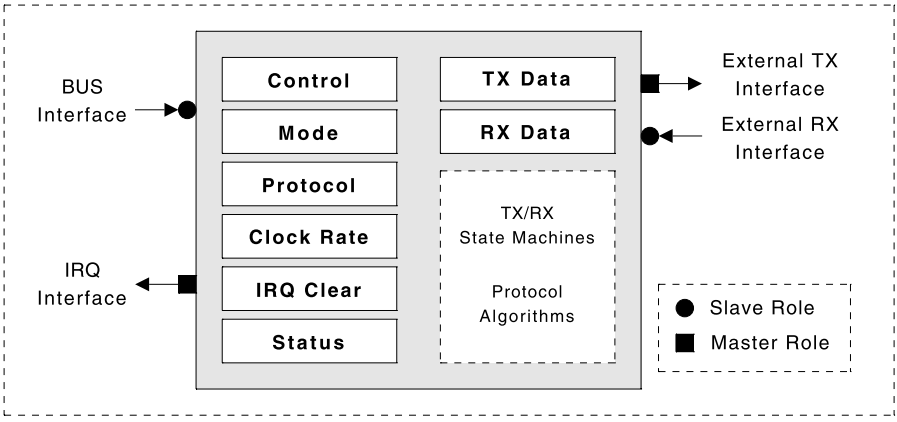

Figure 5.6. SIF structure overview.

Besides the hardware interfaces, the SIF model contains several registers as shown in Fig. 5.6. These registers are accessible from the CPU core (this denotes the CPU where the SW is executed) over the bus interface. Therefore, the registers are an essential part of the HW/SW interface. The SW can configure the behavior of the SIF by writing to the control and protocol registers. Furthermore, it can configure the baud rates for transmission and reception by writing to the clock rate register. The SW can clear interrupts by writing to the

IRQ clear register and it can query the peripheral status by reading from the status register. Data to be transmitted is written to the TX data (TXD) register, and received data can be retrieved by reading from the RX data (RXD) register.

Depending on the current protocol and control register settings, the TX and RX state machines perform different algorithms which influence the transmission and reception (e.g., heading reversal or bit inversion of transmitted/received bits). A detailed description of the registers including their bit fields is given later in this section.

5.5.2 Hardware Interfaces

Bus Interface. The SIF peripheral model can be connected to a bus through its bus interface. This bus interface acts as a slave when connected to a bus. Hence, the SIF cannot actively request a bus access. It reacts on read and write accesses from devices connected to the bus as master. The structure of the SIF bus interface is specified corresponding to the applied bus protocol—in this example a simple bus protocol.

The CPU core, which is usually connected as a bus master, can access the registers of the SIF through this bus interface. For selecting a specific register of the SIF, the core needs to set an address (in this case the sum of the SIF base address and the internal offset of the targeted register). The decoding of the register offset is implemented within the SIF. In case of a write access, the data from the bus is routed to the addressed register. In case of a read access, the content of the addressed register is made visible on the bus. Accesses to SIF registers, where applicable, trigger further actions, for instance starting of the TX state machine or clearing of an interrupt.

External TX Interface. The external TX interface of the SIF enables the data transmission to a connected external peripheral module or another SIF. Data is transmitted actively through this interface, therefore it is specified as a master interface. If data transmission is enabled, the data, which is written from the core to the TXD register, is directly transmitted to the connected external module.

External RX Interface. The module which is connected to the external RX interface of the SIF, can actively send data to the SIF. Therefore, this interface is specified as slave. The SIF can only receive data through this interface if the connected external module transmits data and the reception is enabled. If the serial reception has finished, the received data is written to the RXD register. An interrupt can be scheduled to notify the CPU core that data is available. Hence, the core can read the content of the RXD register.

Interrupt Interface. The interrupt interface of the SIF model contains all available interrupts as outgoing ports. This interface acts as a master. The in-

terrupt ports are connected to the Interrupt Control Unit (ICU) of the system. The ICU handles all peripheral interrupts and notifies the core to serve an interrupt. The core in turn initiates an interrupt service routine corresponding to the active interrupt.

5.5.3 Registers and Bit Fields

As mentioned before, registers are key elements in the context of HW/SW interfaces. Table 5.9 provides an overview of all registers of the SIF including some general register parameters. These parameters are defined as follows:

- **Offset:** Specifies the address of the register relative to the base address of the peripheral
- **Width:** Specifies the data width of the register in terms of bits
- **AddrUnit:** Specifies the addressable unit of the register in terms of bits (e.g., $AddrUnit = 8$ denotes that one address value addresses 8 bits of data)
- **MirrorSize:** Specifies the number of consecutive addresses a register can be accessed through

Register	Offset	Width	AddrUnit	MirrorSize
SIF_TXD_REG	4	32	8	1
SIF_CTRL_REG	8	32	8	1
SIF_MODE_REG	12	32	8	1
SIF_PRTC_CFG_REG	16	32	8	1
SIF_CLK_RATE_REG	20	32	8	1
SIF_IRQ_CLEAR_REG	24	32	8	1
SIF_RXD_REG	28	32	8	1
SIF_STAT_REG	32	32	8	1

Table 5.9. SIF register overview.

The TXD register and RXD register are used for data transmission and data reception. The CTRL register is used for enabling or disabling data transmission or reception of the SIF. Furthermore, mode and interrupt behavior can be configured here. The protocol behavior of the SIF can be configured by writing to the PRTC_CFG register. Activated interrupts can be cleared using the IRQ_CLEAR register. The core can retrieve the current status of the SIF by reading the content of the STAT register. The transmission and reception rate can be controlled with the CLK_RATE register.

A register can be seen as an alias for the offset address. The actual values are accessible through the bit fields it contains. The complete protocol configuration of the SIF, like parity settings, inversion, etc. happens through the

PRTC_CFG register. Therefore, this register needs to be structured into bit fields which are referring to specific protocol settings. A detailed description of the registers including their bit fields is given in the following sections.

Register SIF_TXD_REG. The TXD register is written by the core. Its content is transmitted through the external TX interface. Table 5.10 shows the bit fields of the TXD register including their parameters.

- **Offset:** specifies the offset of the bit field within the register in bits
- **Width:** specifies the width of the bit field in bits
- **AccessExt:** specifies the external access type of the bit field via the bus interface (e.g., from the core)—read only (R), write only (W), read and write (R/W)
- **AccessInt:** specifies the internal access type of the bit field from inside the SIF (e.g., from the state machines)

The TXD register contains only one bit field which covers the complete register width. It is writable and readable from the core, and only readable internally. Every time the core writes data to the register and the transmit bit field, respectively, the TXD state machine gets notified and transmits the data by accessing the content of the bit field internally.

BitField	Offset	Width	AccessExt	AccessInt
transmit	0	32	R/W	R

Table 5.10. Register SIF_TXD_REG bit fields overview.

Register SIF_CTRL_REG. The mode and control configuration of the SIF happens through the CTRL register which is structured into ten bit fields as shown in Table 5.11.

Transmission can be enabled or disabled by setting the *transmit_enable* bit field. In case the transmission of the SIF is disabled, the content available in the TXD register will not be sent. The value is stored until the transmission is enabled.

A similar behavior applies to the *receive_enable* bit field which is used to enable or disable the data reception of the SIF. In disabled state, any incoming data through the external RX interface is ignored.

Interrupts for successful or failed data transmission and reception, respectively, can be enabled or disabled by setting the *interrupt_on_** bit fields.

BitField	Offset	Width	AccessExt	AccessInt
interrupt_enable_on_transmit	0	1	R/W	R
interrupt_enable_on_transmit_error	1	1	R/W	R
transmit_enable	2	1	R/W	R
reserved0	3	1	R/W	R
interrupt_enable_on_receive	4	1	R/W	R
interrupt_enable_on_receive_error	5	1	R/W	R
receive_enable	6	1	R/W	R
reserved1	7	1	R/W	R

Table 5.11. Register SIF_CTRL_REG bit fields overview.

Register SIF_MODE_REG. The operation mode of the SIF can be configured by setting the *loop_back* and *echo_mode* bit fields within the MODE register. In loop back mode, the SIF loops the transmitted data back into the RX state machine. In echo mode, received data is transmitted instantly through the external TX interface. The bit field structure of the MODE register is shown in Table 5.12.

BitField	Offset	Width	AccessExt	AccessInt
loop_back	0	1	R/W	R
echo_mode	1	1	R/W	R

Table 5.12. Register SIF_MODE_REG bit fields overview.

Register SIF_PRTC_CFG_REG. The TX and RX protocol behavior of the SIF can be specified by writing the bit fields of the PRTC_CFG register. Table 5.13 gives an overview of the bit field specification of the PRTC_CFG register. The bit field *tx_stop_bit2* specifies whether a second stop bit should be appended to the data. Parity checking can be activated with *tx_enable_parity*, and the *tx_odd_parity* bit field specifies if an even or odd parity bit should be appended. The *tx_inversion* and *tx_heading* bit fields are used for data inversion and reversal of data heading, respectively. The SIF supports different character length values for transmission or reception—8 bit, 16 bit, and 32 bit. The TX character length can be specified with the *tx_character_length* bit field.

The same bit fields are available for the RX protocol with respective meanings.

Register SIF_CLK_RATE_REG. The settings of the CLK_RATE register define the baud rate for transmission and reception. The bit field specification

BitField	Offset	Width	AccessExt	AccessInt
tx_stop_bit2	0	1	R/W	R
tx_enable_parity	1	1	R/W	R
tx_odd_parity	2	1	R/W	R
tx_heading	3	1	R/W	R
tx_inversion	4	1	R/W	R
tx_char_length	5	6	R/W	R
reserved0	11	5	R/W	R
rx_stop_bit2	16	1	R/W	R
rx_enable_parity	17	1	R/W	R
rx_odd_parity	18	1	R/W	R
rx_heading	19	1	R/W	R
rx_inversion	20	1	R/W	R
rx_char_length	21	6	R/W	R
reserved1	27	5	R/W	R

Table 5.13. Register SIF_PRTC_CFG_ REG bit fields overview.

is shown in Table 5.14. The clock rate for the TX data path can be specified within the *tx_clock_rate* bit field and the rate for the RX path within the *rx_clock_rate* bit field. The value of these bit fields is interpreted as a multiplier to the bus clock period. The baud rate defines the speed at which the serial bits are shifted out or read in, respectively.

BitField	Offset	Width	AccessExt	AccessInt
tx_clock_rate	0	16	R/W	R
rx_clock_rate	16	16	R/W	R

Table 5.14. Register SIF_CLK_RATE_ REG bit fields overview.

Register SIF_IRQ_CLEAR_REG. Active interrupts of the SIF can be cleared by writing the corresponding interrupt ID into the *irq_clear* bit field within the IRQ_CLEAR register. The register specification is shown in Table 5.15.

BitField	Offset	Width	AccessExt	AccessInt
irq_clear	0	4	R/W	R

Table 5.15. Register SIF_IRQ_ CLEAR_REG bit fields overview.

Register SIF_RXD_REG. The bit field specification of the RXD register is shown in Table 5.16. The RXD register contains only one bit field with the

same width as the register itself. This *receive* bit field is declared as external read only. If the SIF receives data through its external RX interface, the core can read the received data by accessing this bit field.

BitField	Offset	Width	AccessExt	AccessInt
receive	0	32	R	W

Table 5.16. Register SIF_RXD_REG bit fields overview.

Register SIF_STAT_REG. The access type of all bit fields within the STAT register is specified as external read-only as it is shown in Table 5.17. The core can access status information of the current state of the SIF by reading the bit fields of this register. The bit fields *data_transmitted* and *data_received* contain information about a successful data transmission and reception, respectively. The bit fields *data_transmit_error* and *data_receive_error* contain information about a failed data transmission and reception, respectively.

BitField	Offset	Width	AccessExt	AccessInt
data_transmitted	0	1	R	R/W
data_received	1	1	R	R/W
data_transmit_error	2	1	R	R/W
data_receive_error	3	1	R	R/W

Table 5.17. Register SIF_STAT_REG bit fields overview.

5.6 Register Header File

A C header file, as explained in the beginning, is generated to enable an easy software access to the registers and bit fields of the SIF. The template based generation framework is described in Sect. 5.9. The generated header file contains a struct and a union for each register access function. A struct of the complete address space of the SIF is also generated, which is used by the access functions. The mechanism of the register access is explained in detail in the following.

5.6.1 Register Bit Field Structure

A struct is declared for each register which represents its corresponding bit field structure. Hereby the complete width of a register is divided into bit fields. Listing 5.29 shows an example of the structs for the *SIF_TXD_REG* and *SIF_*

STAT_REG registers. The *SIF_TXD_REG* register only contains the *transmit* bit field. Since this bit field has width 32 bit the struct *sSIF_TXD_REGStructure* contains only one member referring to the *transmit* bit field. The declaration of a bit field member within the register struct is shown in following rule:

["const"] *data_type* [*bitfield_name*] " : " *bitfield_width* "; "

A bit field gets declared as *const* if it is specified as externally read-only. The *data_type* refers to the *data_type* of the register. The *bitfield_name* is optional and refers to the specified name of the corresponding bit field. The *bitfield_width* refers to the width of the bit field. If no bit field is specified for a specific area of a register, this area must be marked as unused. This is done by declaring a *const data_type* without a name and the width of the unused area. The declaration order of the bit field members within the struct is given by the specified bit offset of the bit field. The *SIF_STAT_REG* register contains four bit fields which are declared as members within the struct *sSIF_STAT_REGStructure*. All of these bit fields are specified as read-only, therefore they are declared as *const*. The unused area of the bit field is declared as previously explained.

```
// SIF_TXD_REG
typedef struct {
  uint32_t transmit                    :32;
} sSIF_TXD_REGStructure;

// SIF_STAT_REG
typedef struct {
  const uint32_t data_transmitted      :1;
  const uint32_t data_received         :1;
  const uint32_t data_transmit_error   :1;
  const uint32_t data_receive_error    :1;
  const uint32_t                       :28; /* unused area */
} sSIF_STAT_REGStructure;
```

Listing 5.29. Register bit field structure.

5.6.2 Register Union

A union is generated for each register which contains an entry referring to the register value and an entry for the register structure. Listing 5.30 shows an example declaration of the *uSIF_TXD_REG* union for the *SIF_TXD_REG* register. The value of the complete register is accessible through *SIF_TXD_REG_Content* which is declared using the data type of the register. The values of the bit fields are accessible through *SIF_TXD_REG_Structure* which is declared using the previously described bit field struct *sSIF_TXD_REGStructure* as data type. The register unions are used as register types in the following described module struct.

```
// SIF_TXD_REG
typedef union {
  uint32_t SIF_TXD_REG_Content;
  sSIF_TXD_REGStructure SIF_TXD_REG_Structure;
} uSIF_TXD_REG;
```

Listing 5.30. Register union declaration.

5.6.3 Module Structure

A declaration of a struct is needed which represents the complete register address range of the module. This struct contains all registers which are ordered referring to their specified offset. Listing 5.31 shows the module structure *_sSif* of the SIF module. The previously described register unions are used as data type for each register member. If an unused address area exists where no register is specified, the area has to be marked as a reserved area. In case of the SIF, no register is specified for offset "0". Therefore, the first entry of the *_sSif* refers to a reserved area and is declared as *const* using *uint32_t* as data type and the name *reservedArea#*. The # represents a number starting with "0" which is incremented for each reserved area declaration. With the value in brackets, which represents the array size, it is specified how many sub sequential addresses should be marked as reserved. Subsequent to the struct declaration, the type definition *sSif* of the *_sSif* struct is declared. The *sSif* type is used within the following described register and bit field access functions.

```
struct _sSif {
  const uint32_t reservedArea0 [1];      // Address offset = 0x0
  uSIF_TXD_REG SIF_TXD_REG;              // Address offset = 0x4
  uSIF_CTRL_REG SIF_CTRL_REG;            // Address offset = 0x8
  uSIF_MODE_REG SIF_MODE_REG;            // Address offset = 0xc
  uSIF_PRTC_CFG_REG SIF_PRTC_CFG_REG;    // Address offset = 0x10
  uSIF_CLK_RATE_REG SIF_CLK_RATE_REG;    // Address offset = 0x14
  uSIF_IRQ_CLEAR_REG SIF_IRQ_CLEAR_REG; // Address offset = 0x18
  uSIF_RXD_REG SIF_RXD_REG;              // Address offset = 0x1c
  uSIF_STAT_REG SIF_STAT_REG;            // Address offset = 0x20
};

typedef struct _sSif sSif;
```

Listing 5.31. Component register structure.

5.6.4 Register Access Functions

The implementation of the functions for accessing the content of a complete register are described now. Depending on the external access type of a register,

a *set* (for writing) and *get* (for reading) function are generated. The external access type of a register is obtained using the bit field information. If a register contains only bit fields which are read-only from external, then the access type of the register is also read-only. In this case, only a *get* access function is generated. Listing 5.32 shows the register access functions for the *SIF_TXD_REG* register. The *transmit* bit field within this register is specified as external readable and writable. Therefore, both a *get* and a *set* function are generated as shown in the listing. The *set* function has no return value and has in its formal argument list the *sSif* pointer *_sif_* and the value which should be written to the register. The target SIF instance is specified using the *_sif_* argument which refers to the base address of the SIF. The corresponding register is accessed by using the −> operator on the *_sif_* pointer of the struct member *SIF_TXD_REG*. This means the base address plus the address offset of the register is obtained because the position of the *SIF_TXD_REG* member refers to the offset address of the register. The register union *uSIF_TXD_REG* is used as the data type for the *SIF_TXD_REG* member. Therefore, the value has to be assigned to the *SIF_TXD_REG_Content* member of *SIF_TXD_REG*. The *get* function is implemented in a similar way. It has the register value as a return value and has no value in its argument list. The implementation of the *get* function returns the value of *SIF_TXD_REG_Content*.

```
/* Set complete register SIF_TXD_REG */
static inline void setSif_SIF_TXD_REG(volatile sSif *_sif_,
    uint32_t value) {
  _sif_->SIF_TXD_REG.SIF_TXD_REG_Content = value;
}

/* Get complete register SIF_TXD_REG */
static inline uint32_t getSif_SIF_TXD_REG(volatile sSif *_sif_) {
  return _sif_->SIF_TXD_REG.SIF_TXD_REG_Content;
}
```

Listing 5.32. Register read and write access functions.

5.6.5 Bit Field Access Functions

After the generation of the access functions for the registers, the access functions for the bit fields are generated. Depending on the external access type of a bit field, a *set* and *get* function has to be implemented. Listing 5.33 shows the implementation of the access functions for bit field *transmit* within the *SIF_TXD_REG* register. The argument lists and return values are equal to the register access functions. The difference is located in the implementation of the bit field *set* and *get* functions. In case of the bit field *set* function, the value is assigned to the *transmit* member of *SIF_TXD_ REG_Structure* of register union *uSIF_TXD_REG*. Within the register union, the bit field structure

sSIF_TXD_REGStructure is used as data type for *SIF_TXD_REG_Structure*. Hence the register union enables the access to the complete register content, or to a specific bit field of the register.

```
/* Set element transmit of register SIF_TXD_REG */
static inline void setSif_SIF_TXD_REG_transmit(volatile sSif *
    _sif_, uint32_t value) {
  _sif_->SIF_TXD_REG.SIF_TXD_REG_Structure.transmit = value;
}

/* Get element transmit of register SIF_TXD_REG */
static inline uint32_t getSif_SIF_TXD_REG_transmit(volatile
    sSif *_sif_) {
  return (uint32_t)_sif_->
      SIF_TXD_REG.SIF_TXD_REG_Structure.transmit;
}
```

Listing 5.33. Bit field read and write access functions.

5.7 SIF Driver Functions

The register and bit field access functions which were described in the previous section are not directly used in a software application since they are too low-level. Driver functions have to be implemented to enable a higher-level software access to the hardware module. These driver functions make use of the lower-level functions offered by the register header file. Unlike the register header file, the driver functions cannot be generated using a meta-model that only describes interfaces but no functionality. A part of the driver header file including the most important driver functions is shown in Listing 5.34. The generated register header file *sif_reg.h* is included in the driver header file *sif_drv.h*, which is shown in the listing. The implementation of the *sifOpen*, *sifInit*, *sifWrite* and *sifRead* functions is explained in detail in the following. Other functions like *sifIOCtl*, *sifClose*, *sifSelfTest*, etc. are not explained within the context of this section. One possibility for realizing software access to the SIF module is represented by the following driver functions. Depending on the operating system and the software environment, the implementation of the drivers may be different.

5.7.1 SIF Open Function

The *sifOpen* function, which returns the *sSif* pointer of the specified SIF module, is shown in Listing 5.35. The *sifOpen* function contains in its formal argument list the unsigned integer *id*, whose value corresponds to a specific SIF module of the hardware system. In case of the example which is shown in

```
// include register header file
#include "sif_reg.h"

// open specified sif
volatile sSif* sifOpen(unsigned id);
// initialize sif
void sifInit(volatile sSif *p_sif);
// write block to sif
void sifWrite(volatile sSif *p_sif, const char *string);
// read block from sif
void sifRead(volatile sSif *p_sif, char* buffer);
...
```

Listing 5.34. SIF driver header file.

the listing, the hardware system contains two SIF modules, one with the base address 0x20000000, and another with the base address 0x30000000. Depending on the value of *id*, the base address of the corresponding SIF module is assigned to the *sSif* pointer and returned.

```
volatile sSif* sifOpen(unsigned id) {
  volatile sSif *pSIF;
  switch(id) {
    case 0:
      pSIF = (sSif*)0x20000000; break;
    case 1:
      pSIF = (sSif*)0x30000000; break;
    default:
      pSIF = 0; break;
  }
  return pSIF;
}
```

Listing 5.35. SIF open function.

5.7.2 SIF Init Function

After a specific SIF has been opened using the *sifOpen* function, it has to get initialized. This is done by calling the *sifInit* function using the returned *sSif* pointer as the argument. The implementation of the *sifInit* function is shown in Listing 5.36. The specified SIF module gets initialized with a default control and protocol configuration. The low-level bit field access functions of the register header file are used for configuring the *SIF_CTRL_REG* and *SIF_PRTC_CFG_REG* registers. First, the SIF is enabled for data transmission and reception, then the character length of both the TX and RX data are set to 32 bits. Not all configurations are shown in the listing. After the SIF module is

opened and initialized, the application can write data to the SIF and read data from it. The necessary driver functions for the data transfer are described in the following.

```
void sifInit(volatile sSif *p_sif) {
  // set default rx and tx control
  setSif_SIF_CTRL_REG_transmit_enable(p_sif, 0x1);
  setSif_SIF_CTRL_REG_receive_enable(p_sif, 0x1);
  ...
  // set default rx and tx protocol
  setSif_SIF_PRTC_CFG_REG_tx_char_length(p_sif, 0x20);
  setSif_SIF_PRTC_CFG_REG_rx_char_length(p_sif, 0x20);
  ...
}
```

Listing 5.36. SIF initialization function.

5.7.3 SIF Write Function

The *sifWriteChar* function, which is shown in Listing 5.37, is used within the *sifWrite* function to write one character to the SIF. The argument list of the *sifWriteChar* function contains the *sSif* pointer and the value to be written. The function polls the value of the *data_transmitted* bit field within the *SIF_STAT_REG* register, until it is "0", which means that the previous data has been transmitted and a new value can be written. In that case, the *set* function for the *SIF_TXD_REG* register is called and the value gets written to the SIF. The *sifWriteChar* function is not accessible by the application software, since the interaction of the application software with the SIF is realized in data blocks.

```
void sifWriteChar(volatile sSif *p_sif, uint32_t value){
  while(!getSif_SIF_STAT_REG_data_transmitted(p_sif))
  ;
  setSif_SIF_TXD_REG(p_sif, value);
}
```

Listing 5.37. SIF write data word function.

The application software uses the *sifWrite* function for data transmission which is shown in Listing 5.38. This function defines in its argument list the character pointer *string*. Hence, the application software can write complete strings to the SIF module. The implementation of the *sifWrite* function serializes the given string into single data words depending on the specified *tx_char_length*, and sends each data word to the SIF by using the previously described *sifWriteChar* function. The implementation of the *sifWrite* and the following *sifRead* functions are simplified for explanation purposes.

```
void sifWrite(volatile sSif *p_sif, const char *string) {
  uint32_t ch_length=getSif_SIF_PRTC_CFG_REG_tx_char_length(
      p_sif);
  uint32_t pkg_length, tx_data;
  while(*string) {
    pkg_length = 0;
    tx_data = 0;
    while((pkg_length < ch_length) && *string) {
      tx_data = tx_data | (*string++ << pkg_length);
      pkg_length = pkg_length + 8;
    }
    sifWriteChar(p_sif, tx_data);
  }
}
```

Listing 5.38. SIF write block function.

5.7.4 SIF Read Function

The data interface for the application software to the SIF is implemented in the *sifRead* function, as shown in Listing 5.39. This function reads a data word from the SIF, separates it into single characters depending on the *rx_char_length* value, and appends the characters to the *string* array.

```
void sifRead(volatile sSif *p_sif, char* string) {
 uint32_t ch_length=getSif_SIF_PRTC_CFG_REG_rx_char_length(
     p_sif);
 uint32_t rx_data, pkg_length;

 while(getSif_SIF_STAT_REG_data_received(p_sif)) {
    rx_data = getSif_SIF_RXD_REG(p_sif);
    pkg_length = 0;

    while(pkg_length < ch_length) {
      *string++ = (rx_data & (255 << pkg_length)) >>
          pkg_length;
      pkg_length = pkg_length + 8;
    }
  }
}
```

Listing 5.39. SIF read block function.

5.7.5 Test Software Application

Listing 5.40 shows a small test application which demonstrates the interaction of the application software with two SIF modules. This test application opens the first SIF (ID = 0), initializes it, writes a “Hello World!” string to it, and reads the string again (the TBE connected to the external interfaces of

the SIF receives all data and transmits it back to the SIF). The same is done for the second SIF (ID = 1) within a loop. This small application shows that all low-level details of the HW/SW interface are hidden from the application software. Hence a developer of application software does not need to know all hardware specific details of a specific peripheral.

```
#include "sif_drv.h"

int main() {
  unsigned id = 0;
  volatile sSif *pSIF;

  do {
    char buffer[1024];
    pSIF = sifOpen(id++);
    sifInit(pSIF);
    sifWrite(pSIF, "Hello World!");
    sifRead(pSIF, buffer);
  }
  while(id < 2);

  return 1;
}
```

Listing 5.40. SIF test application.

5.8 Synchronization

This section provides an overview on concepts which together form the synchronization of HW/SW interfaces.

5.8.1 Register-Access Synchronization Schemes

Clock domains and synchronization. At the HW level, the settings of the bus and the master and slave interfaces ensure that the accesses of the core to peripherals are synchronized in terms of clock frequencies. It is possible that a core runs with a different clock than a peripheral. In such a case, it is common to use clock divider circuits and data buffers in the HW for synchronizing the data flow appropriately. Usually, the SW needs only to take care of the configuration of these components to maximize throughput.

Blocking vs. non-blocking bus protocols. As explained in the previous sections, SW accesses the HW via pointers (i.e., addresses). An access to an address is in turn broken down in the HW to read and write transactions. The duration of each access can vary depending on the underlying bus protocol. Such accesses can be categorized in the following way:

- Non-Blocking: The duration of the access is a priori defined; the duration of read transactions may differ from write transactions, however.
- Blocking: The duration of the access can dynamically vary, depending on the current bus load and peripheral activity.

These categories need to be taken into account when modeling peripheral drivers, because they can have a huge impact on the overall performance of a system.

5.8.2 Functional Synchronization Schemes

Polling. The easiest way to functionally synchronize SW behavior to the behavior of a peripheral is to use polling. For instance, the SIF peripheral offers a status register which yields whether the SIF is ready to transmit data or respectively has data available to be fetched by the core. The SW can retrieve the status of the SIF by accessing the corresponding register. After initiating one transmission, the SW cannot know when the SIF is ready to transmit further data. Therefore, one possible way to resolve this, is to read the status register periodically. As soon as the read value indicates that the SIF is ready for another transmission, the SW can stop reading the status register and proceed with the next value to be sent. Listing 5.41 shows an example.

```
uint32_t txd_array[4] = {10,20,30,40};
for(int i = 0; i < 4; i++){
  w_reg_ptr->txd_reg = txd_array[i];
  while !(r_reg_ptr->flag_reg.data_transmitted)
  ;
}
```

Listing 5.41. Polling of SIF status.

Yet being a simple solution, polling is not very efficient with regard to performance, due to the periodic read access to the SIF status register. The next section shows how interrupts can be used in order to notify the SW by the peripheral, as soon as it is ready to transmit further data.

Interrupt Handling. Interrupt-based synchronization of SW with HW is one of the dominant schemes used in embedded systems design. Herein the basic principles of interrupt handling are explained. Most CPU cores in industrial use provide interrupt mechanisms in order to decouple SW from the state of a peripheral. The interrupt lines of each peripheral are connected to the so called Interrupt Control Unit (ICU). By setting an interrupt line, a peripheral indicates that it requests to interrupt the CPU core execution. Since it is possible that several peripherals can issue interrupt requests at the same time, the

ICU implements some sort of interrupt prioritization scheme and notifies the CPU core in turn by raising an interrupt to the CPU core. The HW-based interrupt infrastructure of a core can vary. It has to take into account when exactly to halt the current execution of a program and to save its context, in order to serve a specific interrupt request. An ICU usually contains registers for each interrupt input. A programmer can configure these registers by storing the address of a specific interrupt service routine in each of these registers. Once the CPU receives an interrupt from the ICU, it can retrieve the previously stored address to the corresponding service routine and execute it. Such a service routine, among other things, needs to take care of clearing the specific interrupt in the peripheral, which had raised it. Once the interrupt service routine has been executed by the CPU core, the execution of the previously halted program resumes. For that purpose, the CPU restores the program context automatically. In the following example, it is assumed that the ICU interrupt input of number 42 is connected to the *data_transmitted* interrupt line of the SIF. Furthermore, it is assumed that the general setup of the ICU has been taken care of. Listing 5.42 shows a simple code example, which contains an interrupt service routine. This routine is invoked whenever the *data_transmitted* interrupt is raised. It takes care of sending a further data value to the TXD register.

The function *enable_IRQ* sets up the CPU internal interrupt infrastructure. For that purpose, some assembly language code (omitted in the example) needs to be inlined. The *SIF_TXREQ* function is the actual interrupt service routine. It ensures that the corresponding ICU interrupt line is enabled again by writing value 42 to specific ICU register addressed by *ICU_REENABLE_HW_LINE*. Furthermore, the function clears the raised interrupt request in the SIF peripheral by writing to the interrupt clear register addressed by *SIF_CLEAR_REG*. Following that, one value is written to the TXD register of the SIF. Within function *send_data*, the ICU register, which is associated with port 42, is written with the address of the interrupt service routine. Hence, when line 42 shows an interrupt, this service routine will be called. Afterwards, the IRQ infrastructure is enabled. Following that, a while loop is entered, which can only be left if the service routine has been called 4 times. Note, that instead of the while loop, the SW could perform any other actions, because the data transmission is handled by the interrupt service routine.

5.9 Template Based Code Generation

Before the implementation of an HW/SW system starts, a specification has to be created which is normally provided as a textual description and not in a formal way. This often leads to implementation inconsistencies or misunderstandings of the specification. Due to the increasing complexity of HW systems formal methods for specifying interface information were developed.

```
__inline void enable_IRQ(void) {
  int tmp;
  __asm {
    // core dependent asm code
    // for enabling IRQs
  }
}

volatile int i;
uint32_t txd_array[4] = {10,20,30,40};

void __irq SIF_TXREQ(void) {
  *ICU_REENABLE_HW_LINE = 42;
  *SIF_CLEAR_REG = 0xF;
  if(4 != i) {
    w_reg_ptr->txd_reg = txd_array[i++];
  }
}

void send_data() {
  i = 0;
  // setup ICU Register for port 42
  *ICU_CBPORT_42 = (volatile unsigned)SIF_TXREQ;
  enable_IRQ();
  while(i < 4)
  ;
  printf("Data Transmitted");
}
```

Listing 5.42. Interrupt based transmission.

On the one hand, these formal descriptions can be used for IP reuse, on the other hand, they can be used for the generation of consistent hardware and HW/SW interfaces. The following sections introduce a UML based meta model and a generated API which supports easy access to the specification data, followed by a technique to import XML based textual specifications into the meta model. In connection to that, a template-based generation framework is described which enables a flexible use of the meta model for code generation.

5.9.1 UML Meta Model and its API

The basis of the generation framework is a UML based meta model which contains the information about how to model the hardware interfaces, like the bus interface and the register and bit field information referring to the HW/SW interface. The specification of each register and bit field of a hardware module would be tedious using UML. Hence, a UML meta model is developed containing all interface and top-level mapping data. Most UML tools provide

source code generation but only for a few target languages. Due to the variety of target languages (SystemC, SystemVerilog, C, etc.), a flexible UML meta model and generation framework is required. The existing standard IP-XACT [SPI], which is based on XML, targets IP reuse and IP exchange by focusing on the packaging of IP. But since we also need to incorporate more functional aspects in order to support IP generation (e.g., generate stubs for the IP development) as well, we defined our own data model instead. In order to fully leverage the benefits of IP-XACT for automating third party IP integration we developed import and export filters for IP-XACT as well.

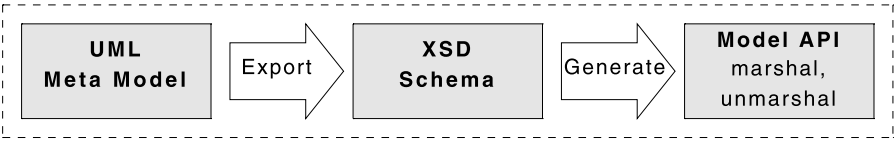

Figure 5.7. Meta model API generation.

Figure 5.7 shows the generation of the API for accessing the data of the meta model. The meta model itself is developed using a common UML tool and exported as an XSD schema. From this schema a class library is generated which offers marshaling and unmarshaling of XML meta model data and provides `set`, `get`, and `add` functions for the elements.

5.9.2 Specification Import

The previously described API for the meta model supports marshaling to create an XSD schema compliant XML file. Hence, a tool is developed which supports an easy import of textual specifications that were created compliant to company specific guidelines. Figure 5.8 gives an overview of the tool which converts textual specifications using the API. A Python [Pyt] conversion plug-in and a textual specification—written with an editor and saved as XML—are used as input while the meta model compliant XML is the output.

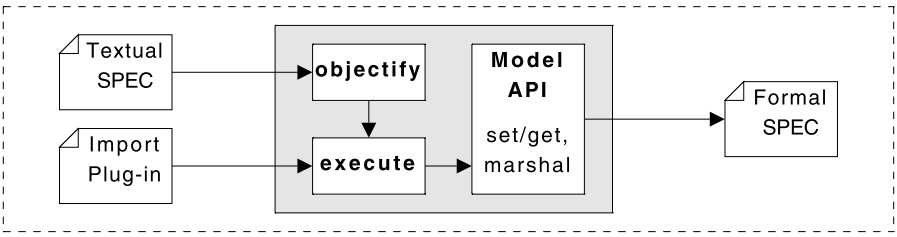

Figure 5.8. Specification import mechanism.

Within the tool the XML based textual specification is objectified using the Python Gnosis library. The obtained object structure is iterated with the conversion plug-in and its values and elements are passed to the meta model API. At the end the object created by the API is marshaled to a formal XML specification. At the moment many different XML methodologies exist within a single company which can be imported to support a company wide XML specification.

5.9.3 Template Based Code Generator

A flexible generation methodology has been developed which provides the code generation for different target applications, for instance firmware header files or the register interface of RTL or TLM models. This is achieved by linking the meta model API to a template engine to access the data of the XML from the template. Templates allow the separation of model and view, in our case the data provided by the XML and the target code to be generated. We use the Python MAKO template engine [Mak] which offers a template language for conditional branches, loops, and hierarchical templates. Furthermore, the complete Python scripting functionality can be embedded within a template. A template can be composed of hierarchical templates, hence, it is possible to reuse so called sub-templates for the generation of different target applications.

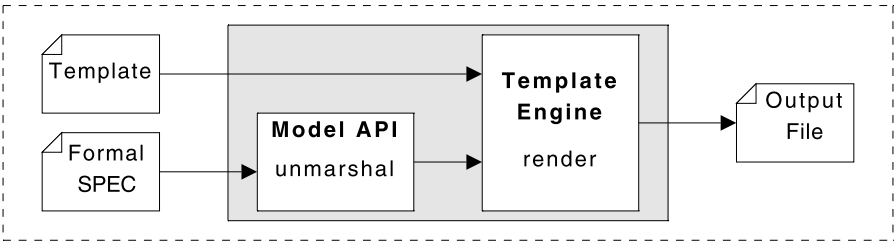

Figure 5.9. Template based code generator.

Figure 5.9 gives an overview of the template generator tool. The template gets rendered by the MAKO template engine using the data provided within the XML file.

5.9.4 HW/SW Interface Generation

The HW/SW interface is described by the registers of a hardware module and by the connection of the CPU core through the bus to the module. Hence, the meta model must contain both the register and bit field information as well as the interconnect information of the module and the CPU core. In the fol-

lowing an overview of the generation of software header files and the register interface of a module is given.

Register header files offer low-level access functions to the memory mapped registers of a hardware module. A register within a hardware module is addressable by using the base address of the module and the corresponding register offset. Each register itself is divided into one or more bit fields. A bit field is described by a bit offset, bit width and an access type which specifies if a bit field is readable, writable, or both.

A hierarchical template structure was developed for the generation of a register header file as it is shown in Fig. 5.10.

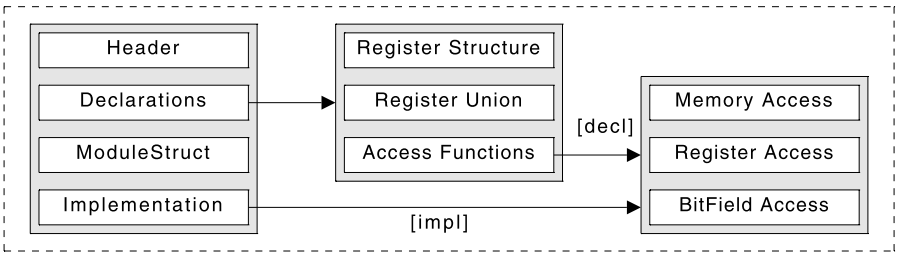

Figure 5.10. Hierarchical register header template.

The main template includes four sub-templates. The *Header* sub-template generates register header file specific information like version, name, module description, and so on. Furthermore, required type definitions are generated by this sub-template.

The *Declaration* sub-template iterates over the registers of a module and generates the register bit field structure and a register union which provides the access to the bit field structure and the value of the registers. The declaration of the register and bit field access functions is also generated here.

The *ModuleStruct* sub-template generates a representation of the address space of the hardware module using the register offsets, the register data width, and the addressable unit of the bus interface.

The *Implementation* sub-template generates the implementation of the access functions.

Following the same methodology the RTL or TLM register interfaces are generated in a consistent way.

5.10 Modeling the HW/SW Interface

The previous sections focus only on the software side of the HW/SW interface. This section describes the modeling of a abstracted hardware module of the SIF module using transaction level modeling in SystemC. First, a introduction to transaction level modeling is given. After that it is shown how the SIF

is integrated into a hardware system, including the description of its interface implementations. At the end of this section a methodology for modeling high performance simulation models is introduced, which enables complex SW tests in an early design stage.

5.10.1 Transaction Level Modeling

Transaction level modeling plays a major role in the success of the development of so called virtual prototypes (VP) which represent abstract models of HW platforms. This allows breaking down a system to a set of components or blocks (representing the actual architecture of the platform top level) comprised of concurrent processes. These blocks communicate with each other via so-called transactions. A transaction represents a high-level form of a communication protocol. All protocol-specific details are encapsulated within a transaction. Hence, the actual act of initiating a transaction results in a remote function call from within a process (parent). A designer focuses more or less on the data that has to be transported rather than the protocol specifics.

In SystemC, the most established modeling language for TLMs, transactions are modeled as functions which are defined in pure virtual interface classes and are implemented in corresponding child classes which inherit from these interfaces. The implementation details of a transaction strongly depend on the targeted abstraction level. Yet two distinctions with regard to transactions can be made:

- Blocking: A blocking transaction may suspend its parent process which means that the transaction is resumed in a later delta-cycle. This kind of transaction can be invoked in suspendable SystemC processes, only (i.e., `SC_THREAD`).

- Non-Blocking: A non-blocking transaction is atomic and may not suspend its parent process; the whole transaction is executed within the same delta-cycle it has been invoked. This kind of transaction can be called from within any SystemC process (i.e., `SC_THREAD` and `SC_METHOD`).

Invoking a transaction results in dereferencing a pointer that holds the address of the target object and in calling a member function of that object. The whole call or even several calls can happen within a single delta-cycle (e.g., with non-blocking transactions). In contrast to that, communication in RTL models is obtained via signals and hence, always consumes at least one delta-cycle due to the induced value-changes that form the protocol. In order to provide connection semantics for transactions as well, SystemC provides a port concept. Figure 5.11 shows a graphical representation.

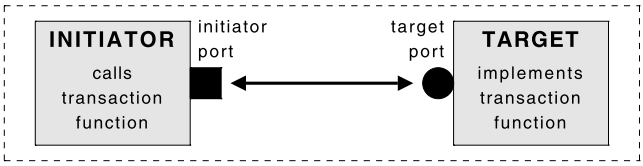

Figure 5.11. TLM interface.

Transactions are called over initiator ports. Transactions are provided by target ports. Initiator ports are modeled in SystemC using `sc_port` and target ports using `sc_export`. Both these ports expect a template argument which holds the interface definition, i.e., the pure virtual class which defines the signatures of all transactions accessible through this interface. In order to ensure easy IP reuse and interoperability, a transaction level modeling standard has been developed by the Open SystemC Initiative. This standard defines different interface classes including transaction signatures and argument types. These interfaces are organized in terms of their characterizations, i.e., into blocking or non-blocking interfaces, and the flow of data, i.e., unidirectional or bi-directional. A module which contains a target port has to provide an implementation for the transaction defined in the interface class which was given as template argument to this target port. In the following sections the so-called transport interface (see Listing 5.43) from the TLM standard is used.

```
// bidirectional blocking interfaces
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_core::sc_interface {
public:
  virtual RSP transport( const REQ & ) = 0;

  virtual void transport( const REQ &req , RSP &rsp ) {
    rsp = transport( req );
  }
};
```

Listing 5.43. Blocking transport interface class.

This interface is a bidirectional blocking interface. It defines a function called "transport" which in turn is templated with two abstract data types (`REQ`, `RSP`)—one which holds the information on a specific bus request, e.g., target address and data, and one holding the response or rather the result of the transaction. The user needs to define request and response classes and needs to provide these as template arguments to customize the transport interface. Both initiator and target port need to use the very same interface, and thus also the same classes for request and response in order to be connected.

5.10.2 SIF Transaction Level Model

Figure 5.12 shows the connections of the SIF to a bus of a hardware system. This system includes a CPU core, a RAM and the SIF, which is externally connected to a test-bench element (TBE). In the following the implementation of the transaction level bus interface and the external RX/TX interface of the SIF is explained.

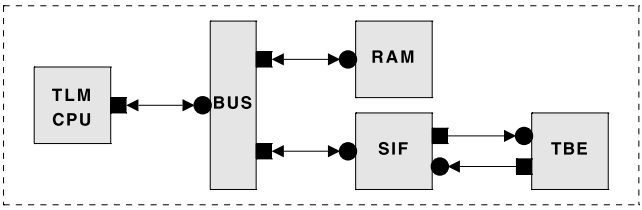

Figure 5.12. TLM system.

Bus Interface. The bus interface of the SIF is specified as a blocking transport interface of OSCI TLM 1.0 [OSC]. It is a bi-directional interface which enables read and write access to a module. The initiator port (*sc_port*) of the bus is mapped to the target port (*sc_export*) of the SIF. The target implements the *transport* interface function. Listing 5.44 shows the implementation of the bus interface of the SIF. First, the declarations of the *bus_request* and *bus_response* classes are shown. The *bus_request* class contains *addr*, *data*, and *access* members. The enumeration type *access_type*, which contains the values *READ* and *WRITE*, is used to indicate the type of the bus access. The *bus_response* class includes a *data* member which contains the requested data in case of a read access and the integer variable *status*. The port *bus_port* of type *sc_export* which is templated with the transport interface using the *bus_request* and *bus_response* classes.

SIF RX/TX Interfaces. The non-blocking put interface is used for the external RX/TX interfaces of the SIF. This interface enables a uni-directional transfer of the external payload. Listing 5.45 shows the implementation of the external interfaces. First, the external payload class *ext_payload* is declared. It contains the data, the start and stop bits, and the parity information. The external RX interface is declared as a non-blocking put *sc_export* target port. Therefore, the *nb_put* transaction function has to be implemented within the SIF. The external TX interface is implemented as a non-blocking put *sc_port* initiator port. Both non-blocking put interfaces are templated with the *ext_payload* class.

```
#include <stdint.h>

// bus protocol
enum access_type {READ, WRITE};

class bus_request {
public:
   access_type access;
   uint32_t addr;
   uint32_t data;
};

class bus_response {
public:
   int status;
   uint32_t data;
};

// SIF bus interface port
sc_export< tlm_transport_if<bus_request, bus_response> >
    bus_port;
```

Listing 5.44. Blocking transport bus interface.

```
#include <stdint.h>

// external interface payload
class ext_payload {
public:
   unit32_t data;
   unsigned char_length;
   bool start_bit, stop_bit, stop_bit2, has_stop_bit2,
        has_parity, odd_parity_bit;
};

// external RX interface
sc_export<tlm_nonblocking_put_if<ext_payload> > rxd_port;
// external TX interface
sc_port<tlm_nonblocking_put_if<ext_payload> > txd_port;
```

Listing 5.45. External non-blocking put interface.

5.10.3 Data Flow Abstraction

Fast simulation models of the hardware are required for the development of complex application software. Current transaction level simulation models mostly do not meet this requirement. Therefore, new methodologies need to be developed to close this gap. The first step in this direction is the abstraction of the data flow from the software to the hardware, and vice versa. Figure 5.13

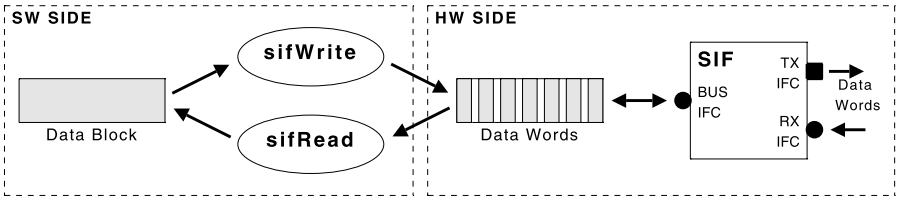

Figure 5.13. HW/SW data flow.

shows the data flow of the previously described SIF module and the application software.

As it is shown in the figure, the software side deals with data blocks, and the hardware side with single data words. In case of a write access to the SIF, the data block gets divided into sequential data words by the *sifWrite* function. In case of a read access, the data words are put together into a data block. The basic concept of the data flow abstraction is explained in the following.

Basic Concept. As it was previously explained, the software side deals with data blocks, but the hardware side with data words. The *sifWrite* and *sifRead* driver functions act like a transactor in between which converts a block to words and vice versa. Not only the conversion costs simulation performance, but also the bus accesses for each data word. The solution for this problem is the abstraction of the data flow, like it is shown in Fig. 5.14.

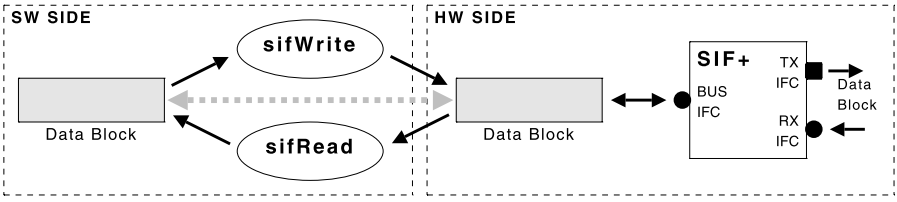

Figure 5.14. Abstracted HW/SW data flow.

As it is shown in the figure, the concept is that the data blocks are not converted by the driver function to data words, but passed directly to the extended SIF module, which is referred to as SIF+ module in the following. This means that software and hardware functionality are not considered separately, but in a common view. In this common view, both the software and the hardware side are dealing with data blocks. Some requirements need to be fulfilled before the common view of hardware and software can be realized.

Requirements. The first requirement for merging hardware and software functionality is that both the software and the hardware are executed on the CPU of the host simulation system. That means that the software cannot be instruction set simulated on the TLM CPU core, but must be emulated on the simulation host. This can be solved by replacing the TLM CPU core of the hardware system with a SystemC module, which wraps the C++ class members referring to the TLM bus interface to C, and executes the C software. As it is shown in Fig. 5.15 the TLM CPU core was replaced by the *EMUCPU* SystemC module and the *RAM* module is no longer part of the system.

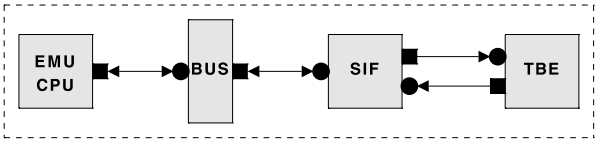

Figure 5.15. Host emulated system.

Now the implementation of the register and bit field access functions within the register header file directly accesses the TLM bus interface. The template, which generates the access function, can be reused by adding an argument which specifies whether a header file should be generated for an instruction set simulated or a host system emulated environment.

This leads to the next requirement to assure the consistency of the TLM model and the abstracted model. The template-based generator helps to achieve this requirement, since the existing templates can be reused for the generation of the abstracted model; the normal TLM interfaces are still used for the control flow.

The last requirement is that both the bus interface and the external interfaces of the SIF module need to be abstracted to support block data transfer. The abstraction of the interfaces is described in the following.

Interface Abstraction. Both, the bus interface and the external RX and TX interface need to be extended to support block transfer. In addition to the bus interface, the SIF gets extended by an abstract interface which enables read and write block transfers. The control flow of the SIF still happens through the TLM bus interface, but the abstracted data flow is realized using the abstract interface. An overview of the interfaces is given in Fig. 5.16.

As shown in the figure, the driver functions *sifRead* and *sifWrite* do not use the register and bit field access functions anymore, but they directly access the abstract interface of the SIF. The *sifInit* and *sifIOCtl* functions are still accessing the TLM bus interface of the SIF using the register and bit field access functions.

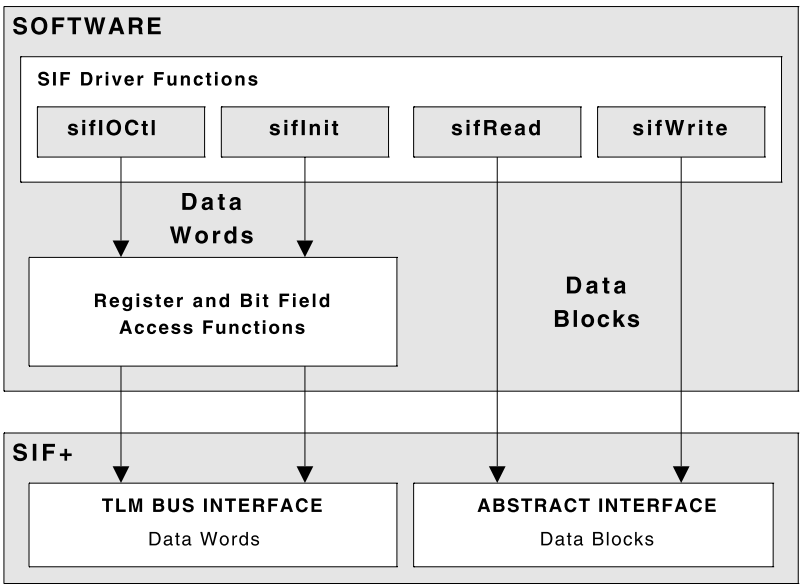

Figure 5.16. Abstracted SIF interface.

The abstraction of the external RX and TX interfaces is quite simple. It is achieved by extending the *ext_payload* class (see Listing 5.45) with a *data_block* member of data type *char** and an integer *block_size* member referring to the number of characters contained within the *data_block* array.

The *SIF+* model is fully compatible to the SIF TLM module, since it still contains the full TLM functionality. Therefore, it can be used in classical TLM simulations.

5.11 Conclusions

Within this chapter the modeling and implementation of HW/SW interfaces was explained, by high-lighting all involved areas, step by step. First some basic concepts were presented on both the HW side and the SW side explained how accessing HW through SW is accomplished. This concept was elaborated on in more detail taking also HW module internal register layouts into account. Many examples were presented illustrating alternative modeling styles which were also discussed. It was also illustrated how the internal communication infrastructure within a HW architecture is dealt with by the SW along with synchronization concepts for on-chip communication. Following these general considerations more detailed examples were provided using an example peripheral model. Based on this example the structure of low-level drivers was explained in detail.

As new contributions, it was also described how to improve the overall consistency of the HW/SW interface by using a single-source approach for obtaining its implementation. In this approach a peripheral specification is formalized in terms of its register layout and internal address map. The formal description serves as a basis for generating most parts of the HW/SW interface, including also the generation of different abstraction views on the HW as well some layers of the SW driver development. Furthermore, new concepts were introduced for raising the abstraction level for HW modeling by abstracting the data flow within the communication between HW and SW and also merging parts of HW and SW to a single abstract HW interface.

The explained concepts on modeling HW/SW interfaces show the huge diversity and the non-negligible complexity of modeling HW/SW interfaces. Using virtual prototyping, a close interaction of designers developing drivers and HW designers becomes possible at early stages of the whole design process. By getting HW and SW even closer through data and interface abstractions, a much better quality of the HW/SW interface can be achieved due to team working over different design domains namely HW and SW. Hence, virtual prototyping can also be considered as a bridge in between these domains.

References

[Mak] Mako Templates for Python. *Hyperfast and lightweight templating for the Python platform*. www.makotemplates.org

[MIT] raw Homepage. *raw Architecture Workstation*. www.cag.csail.mit.edu/raw

[OSC] OSCI TLM Working Group. *OSCI standard for SystemC TLM*. www.systemc.org

[Pyt] Python Software Foundation (PSF). *Python Programming Language*. www.python.org

[SPI] SPIRIT Consortium. *IP-XACT Standard*. www.SPIRITconsortium.org/tech/docs

[Wik] Wikipedia's Z80 Article. *Zilog Z80*. en.wikipedia.org/wiki/Zilog_Z80

Chapter 6

FIRMWARE DEVELOPMENT FOR EVOLVING DIGITAL COMMUNICATION TECHNOLOGIES

Stefan Heinen and Michael Joost

Abstract Advances in modern digital communication technologies enable higher data transmission speeds and thus more and more sophisticated services can be offered to the end user. For a semiconductor manufacturer playing in this arena, the market demand for new products supporting the latest communication standards becomes an ever growing challenge. This is owing to the tremendous complexity increase from one technology generation to its successor—in wireless communications this is roughly a factor of ten—while time-to-market has to be preserved for the sake of competitiveness.

Putting more resources on product development projects can only be part of the answer to the complexity increase. What is needed in the first place are faster and more efficient development approaches. In this chapter we present a product-proven firmware design flow that was recently successfully applied in a Universal Mobile Telecommunications System (UMTS) baseband chip development.

The described flow guides system level modeling in a consistent, reuse-oriented fashion from abstract algorithmic exploration towards a virtual prototype for firmware integration and verification. Important key characteristics are single source description and code generation of hardware / software interfaces as well as a sophisticated test bench concept comprising stimulus generation, response analysis and system state monitoring.

Keywords: Virtual Prototype, UMTS, Single Source Interface Specification, Code Generation

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

6.1 Introduction

Modern digital communication technologies have become widespread over the last years. Especially, wireless cellular standards are getting more and more powerful and meanwhile penetrate people's daily life all around the world. Consequences of the ongoing functionality enrichment are highly sophisticated new products, which require continuous advances in both the hardware and the software development process. To keep pace with the tremendous complexity growth of future products, traditional development approaches have to be reviewed, optimized and possibly replaced by new ones.

Looking from the end user perspective at the evolution of wireless cellular standards means in the first place that higher and higher data rates become available, which give rise for more advanced services. However, the developer of such systems is confronted with an exponential growth of operation modes and parameterization options. Due to the virtually infinite number of configurations, system verification becomes the major challenge. Especially for complex logic products, as e.g. baseband chips, the effort for system integration and verification can today already amount to more than 50% of the R&D costs. Therefore the development of an efficient, model driven system level design flow is an important step to meet today's and future challenges.

Though methodology indeed is one indispensable prerequisite to manage complexity, almost as important to achieve competitive development cycles and products is to choose smart system architectures that ease verification and allow to achieve a high test coverage with reasonable effort. A base band solution developed for the 3.5G Universal Mobile Telecommunications System (UMTS)—one of the most complex wireless systems in operation today—may serve as a good example of how the verification aspect impacts the system architecture as well as the firmware design and how system level methodology can provide the necessary system verification framework [HS06, BSH+07]. In this chapter we will briefly browse through the development process with special focus on firmware and system level verification.

This chapter is organized as follows. In Sect. 6.2, we have a glance at the ongoing evolution of wireless cellular communication standards and identify the consequences with respect to system complexity. To handle this ever growing complexity, we derive a set of architectural design paradigms, which proved to be highly successful in recent large scale product development projects. These architectural measures are complemented by a set of methodical innovations. An efficient reuse-oriented system level design flow is presented in Sect. 6.3. A methodology to handle the overwhelming amount of hardware / firmware interfacing registers that a chip of the considered complexity possesses is subject of Sect. 6.4. The advantages of the applied approach are consistency throughout the development process and reduced coding effort. Finally, in Sect. 6.5,

we focus on the integration and verification challenge and address the task of creating a complex system level test bench containing stimulus generators and response analyzers.

6.2 Evolution of Wireless Standards and the Consequences

As indicated above, the main driving factor for the evolution of wireless standards is the end-user's request for higher transmission data rates, e.g. to enable a faster download of Internet contents. Along with this usually comes the demand for higher flexibility in order to exploit the available bandwidth on the air interface in an efficient manner. For instance, it would be a bad idea to do web browsing over a circuit switched connection reserved for one single user, since precious resources would be occupied even if no download is taking place. Clearly, a packet-oriented connection with variable transmission rate is better suited in this case.

A good example for such an evolution is the extension of the UMTS standard (originally released in 1999) by the High Speed Downlink Packet Access (HSDPA) feature. The basic innovation of HSDPA is a more bandwidth-efficient modulation scheme called 16-point Quadrature Amplitude Modulation (16QAM), which allows to transmit double the number of bits per time unit as with the original 4-point Quadrature Phase Shift Keying (QPSK) modulation. In addition, so-called multi-code transmission is applied to bundle transmission resources for one user temporarily. Clearly, HSDPA is enriching UMTS by further operation modes and configuration parameters, which in total lead to an overwhelming flexibility and verification space. A similar evolution can be observed when looking at past, current, and future standards as shown by Table 6.1.

Standard	Gener.	Extensions	
Global System for	2G		
Mobile Comm.	2.5G	GPRS	General Packet Radio Service
(GSM)	2.75G	EDGE	Enhanced Data rates for GSM Evolution
	3G		
Universal Mobile	3.5G	HSDPA	High Speed Downlink Packet Access
Telecommunications	3.75G	HSUPA	High Speed Uplink Packet Access
System (UMTS)		MIMO	Multiple Input Multiple Output
		LTE	Long Term Evolution
...	4G		

Table 6.1. Examples of the evolution of mobile standards. Analog systems of first generation (1G) are not considered.

When mapping such a standard on a technical realization, the system architect will be confronted with a multitude of system states, lots of different

scheduling constellations, and processing paths with different timing demands, e.g. time-critical loops or throughput-dominated processing. Based on his system knowledge, the architect has to determine an appropriate hardware / software split and to select an architecture with sufficient performance to fulfill the system requirements. Here, however, the complexity of the system comes into play. Due to the virtually innumerable[1] number of system configurations, it can become extremely difficult to prove that a certain architecture fits all needs. Also, the wireless standard does not help much here, since it only specifies a very limited percentage of requirements and tests. The focus usually is on basic performance requirements to be met. These specified tests comprise some few percent ($< 5\%$ in case of UMTS) of the overall system functionality which should be tested before such a complex system can run in production. Thus, designing systems such that they are suited for efficient verification becomes a critical key task for the role of the system architect.

6.2.1 System Design Paradigms

More specifically, the system design and verification methodology must go hand-in-hand with the specification of an easily verifiable system architecture to achieve competitive development cycles and products. The considered UMTS base band solution is a good example, showing how the verification aspect has been taken care of in the system design. We comprised the most important of the applied design paradigms below.

Orthogonalization of functionality. Functionality is clustered to independently operating hardware / software blocks with small interfaces to other blocks. In this way, interference of functional blocks is minimized; a clearly structured vertical firmware architecture with few horizontal interconnects results, which is easier to implement, verify, maintain and reuse.

Orthogonalization by hardware / software split. The hardware / software interface is treated as a natural boundary for functional separation. Hardware is designed to operate widely independent, which avoids time-critical hardware / software interaction and unburdens the firmware from real-time constraints.

Restricted configuration grid. Hardware reconfiguration is only allowed at specific time instances. This simplifies the scheduling of firmware tasks considerably and helps to keep the control flow in the hardware lean.

[1] The recent evolution of wireless standards is characterized by an dramatic growth of operation modes, the number of parameters for each system component, and the number of values they can take. For instance, in UMTS we have normal mode / compressed mode, different diversity schemes, different decoders, huge amount of puncturing / repetition possibilities in the outer receiver, lots of slot formats for the inner receiver and so on. All these parameters can be combined almost arbitrarily.

6.2.2 Methodology

Furthermore, to handle the complexity of the design and verification process, an efficient methodology is required. When reviewing our traditional approaches, two innovative decisions made in this context turned out to be particularly beneficial in the course of the UMTS base band project.

System level modeling. The system is modeled in terms of a transaction level Virtual Prototype (VP). Compared to a Field Programmable Gate Array (FPGA) or the final silicon, the advantages of the VP are early availability (months before final netlist), better debugging capabilities, and flexible availability (no board limitation; "virtual hardware" can easily be distributed and updated).

Single source hardware / software interface specification. The huge amount of hardware registers of highly complex chips requires special care. Therefore, error-prone and inefficient paper specifications were replaced by a machine-readable, single-source description, which can be converted into different hardware / software interface representations, such as Register Transfer Level (RTL), SystemC[2] / VP, and HyperText Markup Language (HTML) documentation.

6.3 System Level Design Flow

To understand the development cycle of a wireless modem chip, it is important to know that standards like GSM or UMTS provide bit-exact specifications only for the transmitter side, and hence for the transmission format to be used on the air interface. The receiver side is specified indirectly in terms of performance requirements, e.g. for a given transmission scenario, an upper limit of the residual block error rate is defined. This way, the standard gives the players in the field room for differentiation by allowing them to realize their own receiver algorithms for synchronization, demodulation, and error correction.

6.3.1 Algorithmic Exploration

Consequently, the beginning of the development cycle is characterized by scientific engineering work on receiver algorithms, including their development, evaluation and selection. Simulation at this development stage takes place on a very abstract level since the focus is on algorithmic performance and high simulation speed, rather than on architectural issues. A widely applied simulation approach in this algorithmic exploration phase is the so-called Stream-Driven Simulation (SDS). Functional entities communicate in terms of data streams, such that the developer is released from control flow or timing

[2] System Description Language, IEEE 1666.

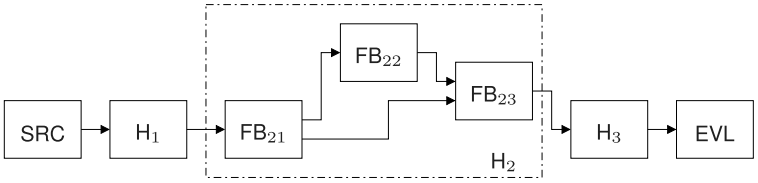

Figure 6.1. Algorithmic model based on stream driven simulation.

issues and can fully concentrate on the algorithmic performance, while the stream-driven simulator kernel takes care of buffer allocation and scheduling.

Figure 6.1 illustrates the typical structure of a stream-driven simulation. Three composite models (H_1–H_3) are embedded into a test bench consisting of a source block (SRC) and an evaluation block (EVL). Each of the composite models may consist of a multitude of functional blocks (FB_{21}–FB_{23}) as exemplified for composite model H_2.

Let us consider Fig. 6.1 as the sketch of a link level simulation of a wireless system. Then SRC may represent a base station model including the propagation channel, H_1–H_3 are models of receiver entities, such as front end, demodulator and channel decoder. Finally, EVL is a bit or block error rate counter, which allows to evaluate the reception quality.

6.3.2 Hardware / Firmware Split

The simulation environment of Fig. 6.1 is an ideal playground to develop, evaluate and optimize algorithms. As soon algorithms start to converge, a first guess about the hardware/firmware split is made. This guess is based on numerous considerations, such as chip area, power consumption, hardware / firmware interface width, and flexibility. To reflect the hardware / firmware split in the structure of our simulation model, we extract all firmware-related functionality from the functional blocks, in Fig. 6.2 explicitly shown for FB_{21}–FB_{23}, and comprise it in a dedicated FirmWare (FW_2) block. Consequently, this split lets functional blocks FB_{ij} convert into pure HardWare (HW_{ij}) models.

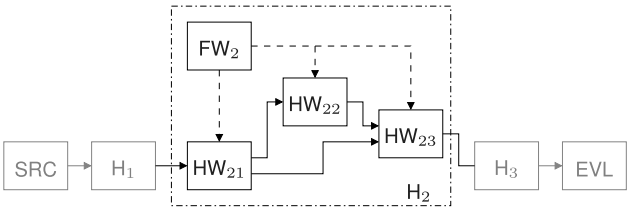

Figure 6.2. Algorithmic model structurally reflecting hardware firmware split.

Block FW_2 represents a collection of algorithmic functions and parameterization code providing the configuration for the hardware blocks HW_{21}–HW_{23}. In particular, the parameterization code, which can be seen as a predecessor of Hardware dependent Software (HdS), is still on an abstract level since the configuration of the HW models in Fig. 6.2 is expressed in terms of parameters rather than registers.

This issue is taken care of by a further refinement step that introduces a Register File Model (RFM_2) according to Fig. 6.3. The FW_2 block is now communicating via RFM_2 with the hardware models, i.e., at its execution, each hardware model is provided a snapshot of the register file from which the required parameterization information can be extracted. Thanks to our eXtensible Markup Language (XML)-based single source description of hardware / firmware interfaces, the register file model is automatically generated, as explained later in Sect. 6.4.

An important detail in this context is the interface between firmware (FW_2) and register file model RFM_2. Register entities are not accessed directly but through access methods, which are also automatically generated. Depending on the respective development stage within the flow, these access methods can have different *expansions*, but the access method *signature* (consisting of method name and parameter list) remains unchanged.

For instance, in the simulation model of Fig. 6.3, access methods are expanded to code accessing the register file model. Later on, for use in the target firmware, the access methods will expand to true register accesses. The advantage of this simple but effective methodology is evident; it allows to start HdS development very early in the development cycle with a quite realistic hardware model coming more or less for free.

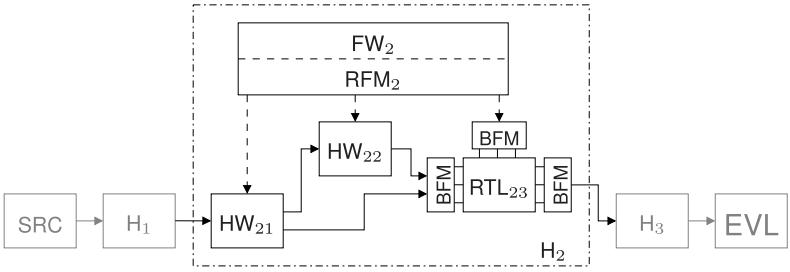

Figure 6.3. Algorithmic model as means for HdS development and as hardware verification test bench; RFM = Register File Model; BFM = Bus Functional Model.

Even though we are focusing on firmware development in this chapter, it is worth mentioning that the model in Fig. 6.3 has a double use in our overall design flow. As exemplarily depicted, hardware model HW_{23} is replaced by

its RTL counterpart RTL_{23}. Such embedding of RTL becomes possible by applying Bus Functional Models (BFM) as adapters translating streams into RTL signals and vice versa. In this way, the hardware models become the golden reference for RTL design.

6.3.3 Time-Behavior Modeling

Although the stream driven model shown in Fig. 6.3 offers already quite a good platform for low level HdS development, its limitations become apparent as soon as the true timing behavior of different functional units interacting with each other in a system has to be modeled. Naturally, timed event-driven simulation as e.g. provided by SystemC is better suited for such a purpose. However, rewriting models for another simulator would not only generate additional effort; it also bears the risk of conversion errors.

A one-to-one reuse of the "golden reference" hardware models is therefore a must in subsequent modeling steps. In our case, the reuse was greatly facilitated by the ability of the applied stream-driven simulator to be run in a "nutshell", i.e. as a sub-function of another simulation model. This allows to reuse not only isolated hardware model primitives (e.g. HW_{21} in Fig. 6.3) but also their interconnectivity.

Figure 6.4 illustrates the integration of a stream-driven model into a time-behavioral model. Easily the almost unchanged structure of model H_2 from Fig. 6.3 can be rediscovered; only the firmware part FW_2 has been stripped off. The stream-driven island (SDS) exchanges via buffered ports data and register information with the surrounding Event-Driven Simulation (EDS) domain. To model the time behavior, two essential components have been added. First, an automatically generated bus interface model (BIM_2) providing a time-aware transaction level interface for register read / write accesses as well as controllers for reset handling (RST), clocking (CLC) and interrupt generation (IRC). Second, a manually coded state machine controlling inputs / outputs and the execution of the SDS kernel as indicated by dashed arrows in Fig. 6.4.

Figure 6.4 gives a simplified example of a state machine having four states. In the idle state (Id), the model waits to be activated. Triggered by a register access or some other specific event, a transition into the load state (Ld) can be issued. Now input data and register information are collected and stored in the input buffers of the SDS. As soon as the scheduling conditions for the SDS are fulfilled, the state machine enters the process state (Pr) in which the SDS is executed and writes processed data as well as modified register information to its output buffers. Finally, in the report state (Rp), the chunk of data delivered by the SDS is transferred to the outside world with the correct timing.

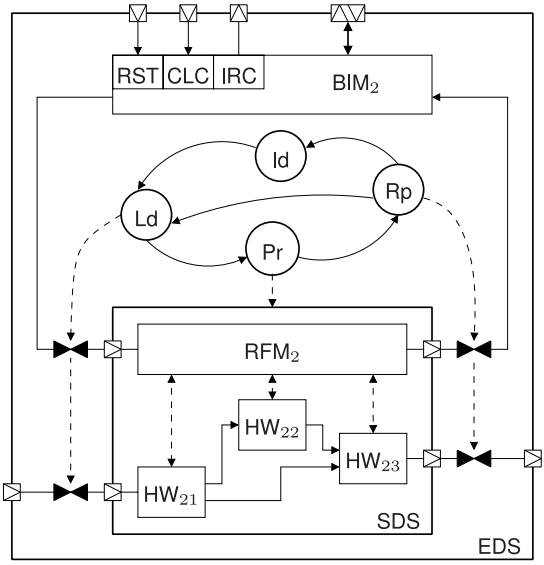

Figure 6.4. Transaction level model with reused stream driven simulation kernel; SDS = Stream Driven Simulator; EDS = Event Driven Simulator; RFM = Register File Model; BIM = Bus Interface Model.

From the above, it becomes clear that the state machine alone determines when inputs / registers are read and when outputs / registers are updated; the SDS in fact runs in zero time from the event driven simulator's point of view. On the other hand, the scheduling conditions of the SDS have impact on the granularity of timing relations between input and output. For instance, if the SDS requires a data block of say 100 input items for being scheduled, and there is one item coming in per 10 ms, 1 s simulation time elapses before the SDS can be run. So the first output item can be delivered after 1 s the earliest. Real hardware behaves differently; depending on the number of pipeline stages, it would usually output the first data item much earlier. Fortunately, for firmware development such inaccuracies in the modeling are often negligible. Timing accuracy is definitely an issue for time-critical closed hardware / firmware loops, and special care has to be taken in these cases.

In general, there is a trade-off between simulation accuracy on one hand, and simulation speed / modeling effort on the other. The key for efficient and successful modeling is therefore a careful analysis of the needed abstraction level and the required accuracy. The modeler should therefore always do his job in close cooperation with system experts, as well as hardware and software developers.

6.3.4 Virtual Prototype

In the final step of our system modeling flow, the time-behavioral models are integrated in a system architecture model consisting of the micro core (µC) and bus subsystems. Figure 6.5 displays the overall structure of this Virtual Prototype (VP). Beside the signal processing peripheral models (#1 ... #n) with their algorithmic SDS kernels, we recognize also pure EDS models (#$n+1$ ... #$n+k$). These models cover control functionalities of the chip, such as interrupt, memory or clock control.

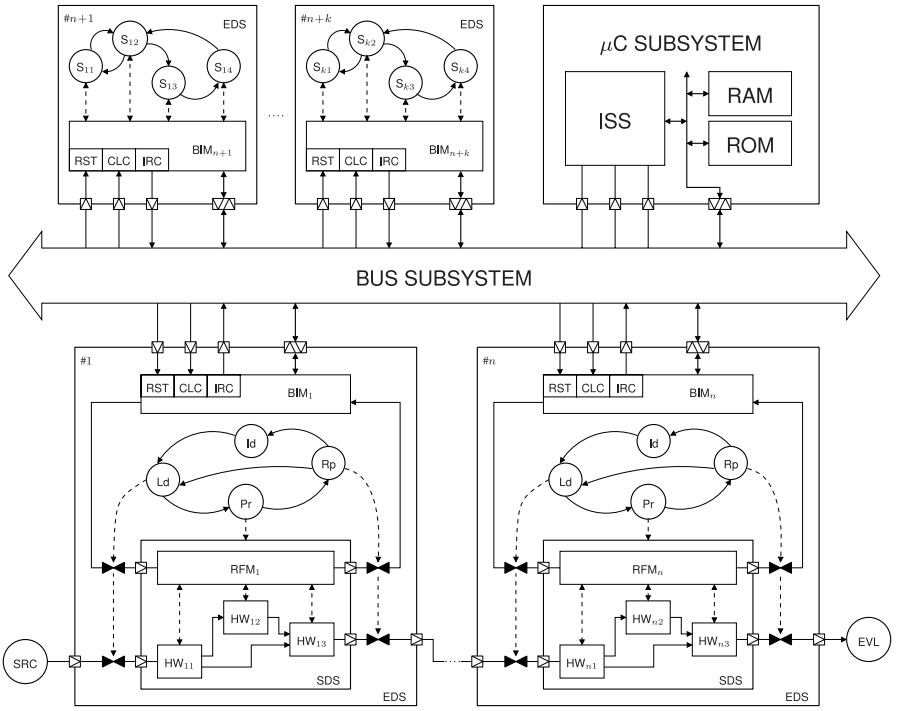

Figure 6.5. Virtual Prototype (VP) for firmware development and verification; #1 ... #n: signal processing peripheral models; #n+1 ... #n+k: control peripheral models; ISS = Instruction Set Simulator; SDS = Stream-Driven Simulator; EDS = Event-Driven Simulator; BIM = Bus Interface Model; SRC = signal generator; EVL = signal analyzer.

To establish an overall simulation time, the Instruction Set Simulator (ISS) coming with the µC subsystem and the event-driven simulator (EDS) are synchronized. From the firmware programmers point of view, the VP functionally and temporally "behaves" in the limits of chosen approximations like the real hardware. Pre-developed HdS, firmware state machines and schedulers can

now be put on the VP and incrementally, feature-by-feature can be brought up and running, typically several months before real hardware is available.

In this firmware / VP integration phase, the remarkable debugging facilities of the VP play an important role. For instance, many HdS errors can be easily found by equipping hardware models with assertions indicating faulty configurations or illegal register accesses. One further very powerful means is combined firmware / hardware debugging, which allows to step through virtual hardware and firmware code simultaneously.

Despite these advantages, it has to be noted that in fact a VP alone has only limited value. Its real strength becomes apparent if the VP is integrated into a test bench (indicated by SRC and EVL in Fig. 6.5), which can be run in a test regression. Details on this will be covered in Sect. 6.5.

To conclude this section, we give some figures about our UMTS layer 1 modem VP. It comprises 9 algorithmic peripheral models controlling in total 12 SDS models (up to 3 per peripheral) and 10 control peripheral models. It is important to mention that the peripherals in case of our UMTS modem are not just small hardware accelerators, but represent full-blown subsystems. Owing to the bit-exact signal processing modeling, the full layer 1 data path in downlink and uplink can be simulated. The simulation speed has a real-time factor of about 1/300 on a one-core 2.6 GHz PC, which is several orders of magnitude faster than a comparable RTL simulation and well-suited for testing of layer 1 procedures (100 UMTS frames take about 5 minutes simulation time).

6.4 Hardware / Firmware Interface

The hardware / firmware interface of the UMTS-Layer 1 (UMTS-L1) system-on-chip has a complexity of around one thousand registers and ten memories, grouped in several peripheral units. These are used for the exchange of configuration information, measurement data and events/interrupts between the firmware and the hardware peripherals. This complexity is typical for todays designs. In a development project characterized by a challenging time-to-market time frame, and, consequently, a large number of engineers working simultaneously on the design, the need for a consistent and flexible change management regarding the frequent interface changes becomes indispensable.

The hardware / firmware interface has therefore been specified using a formal description in XML. From this central definition all register interface dependent representations have been derived by an automation tool, including

- HTML and paper documentation
- the bus/peripheral interface in Very High Speed Integrated Circuit Hardware Description Language (VHDL) code, including the complete register file, address decoder and interrupt wiring

- low-level functions and constants for firmware drivers in programming languages C++/C (ISO/IEC 9899, ISO/IEC 14882)
- register definitions for debugger
- Transaction Level Modeling (TLM) and cycle-true interfaces to hardware models in the Virtual Prototype in SystemC
- support for verification tools and test benches

The generated VHDL- and HdS code provides just the communication means between each other entities. This code is combined with manually written code covering the application-specific high-level aspects of software drivers and hardware peripherals. The HdS is further completed by another layer that coordinates both, inter-peripheral and global aspects of the System on Chip (SoC) device (Fig. 6.6).

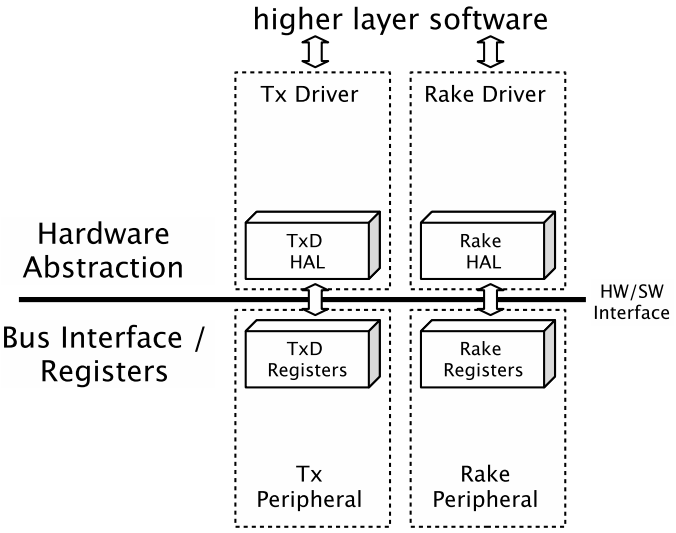

Figure 6.6. Hardware/Software interfaces.

The use of a formal specification, together with an automated generation tool, allows to provide consistency among the many parties involved in the hardware / firmware interface, even under the constraint of a 'floating', emerging interface definition. Furthermore, this implements a correct-by-construction approach, as, regarding the implementation of the register interface, only the tool needs to be verified for correct code generation, rather than each individual peripheral's instance.

There are several additional requirements on all flavors of documents and code produced by the generator:

- semantic checks on the XML input, e.g. the consistency of interrupt mask/status/set/clear registers
- equalize the user input, e.g. transform names into redundancy-free, yet unique identifiers
- the documentation conforms to corporate layout standards
- the code has to cooperate with proven libraries, e.g. VHDL code for bus interface protocols, SystemC for Virtual Prototype integration
- the code must be highly readable, and conform to in-house coding standards
- must allow to analyze the generated code with a source-level debugger, thus, non-cryptic, with full visibility
- software code should allow 'virtualization' of hardware peripherals, providing sort of a hardware abstraction layer
- last but not least, the SW code must execute highly efficient, consuming minimal cycle count under all possible combinations of different register accesses.

The consequence of these requirements is that the generator tool needs to contain quite a significant amount of logic, not just a trivial style sheet-type transformation of the XML source into some other format.

During the development and verification of HdS, the code needs to be executed in many different environments. Often these environments don't represent hardware behavior in the same way as the target system. In particular, it might not reside in the address space of the processor executing the HdS code. Furthermore, different compilers might be involved, such as the target compiler and the workstation compiler. Such environments include:

- HdS module tests without hardware, simulating hardware peripheral behavior in the test bench
- HdS running against a TLM model of the hardware (Virtual Prototype)
- HdS running against a VHDL simulator of the hardware peripheral
- HdS running on target system, with real hardware peripherals

Obviously, flexible moving among those environments has to occur without modifications on the HdS source code. The standard method of representing peripheral registers as a set of structures cannot handle these requirements adequately, although the same execution speed on the target system is required.

The code generator for the hardware / firmware interface therefore generates function-based access methods for register access, which can be individually adapted to each environment.

Conventional system design has invented a plethora of different kinds of registers, which usually implement some type of side effect, when the software accesses the register content. To support all those different register types in a design tool would cause an unlimited amount of effort. Furthermore, those side effects often disturb system debugging, as then there are three parties involved in monitoring and manipulating register contents, the hardware, the software, and the debugger.

Fortunately, these side effects can easily be avoided, to the benefit of a lean and simple communication between the HdS and the hardware peripherals. We have found, by analyzing such attempts thoroughly in many embedded system designs, that the system requirements could always be fulfilled without disadvantages using a 'canonical' set of only three register types, providing a clean and simple communication between the hardware peripheral and the software layer (Table 6.2).

Register type	Access	Behavior
non-volatile	R/W	The software can write and read the register content. The hardware peripheral uses the register content read-only, and never modifies it. Thus, the software can assume to know the value from its most recent write operation, and thus avoid lengthy read-modify-write cycles when modifying only parts of the register content. This type is used to configure the hardware peripheral.
read-only	R	The register content is set by the hardware, and read by the software. The software cannot modify the register contents. This type includes interrupt status reporting.
write-only	W	The software can write the register content, but not read it. The value is interpreted as active-1 encoded, that is, writing a '1' in a bit position invokes some action in the hardware peripheral, while with writing a '0' bit, the action is not invoked. This type includes the clearing of pending interrupts.

Table 6.2. Canonical Register Types.

For generating a bus/peripheral register interface in VHDL code, further sub-constraints on those 'architectural' types are necessary, e.g. to determine synchronization with clock signals. These details are usually not visible to the software layer.

6.5 Test Bench

The functional correctness of a software or hardware component is verified in a test bench (Fig. 6.7). A key requirement for test benches is their capability for regression testing, that is, provide an easy means to repeat all previously developed test cases at any time, in order to assure that subsequent modifications to the test subject implementation did not induce faults or unwanted side effects into already successfully tested aspects of the implementation. Execution of the test suite must be fully automatic, so that the entire test suite can run succeeding the regular nightly builds or completion of release versions.

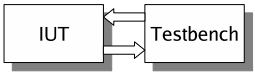

Figure 6.7. Test bench scenario.

There is a multitude of different levels of test benches involved in the development of a complex SoC like the UMTS-Layer1. Each level attempts to test at a specific scope. The next higher level integrates the pre-tested components into a test scenario at a larger scope. A monolithic test on the complete complex product alone would not lead to a stable system in acceptable time, thus there is demand for hierarchically structuring the test scenarios.

Examples of such testing scopes relating to software development are function tests at a module/unit scope, or certification tests of the entire mobile phone with the UMTS network operators worldwide. What we will discuss here is a medium-level scope including the entire SoC firmware and a realistic representation of the hardware, either a model furnished as Virtual Prototype, or the Application-Specific Integrated Circuit (ASIC) running at a reduced clock speed. This scenario allows for both, the early integration of all hardware and software components long before the ASIC is actually produced, as well as later on running the same tests against the ASIC. The Virtual Prototype variant provides full visibility on the internals of all components, and can thus support debugging even after the ASIC silicon is available.

6.5.1 Stimuli

An important part of a test bench is the generation of stimuli to drive the implementation-under-test, as well as the analysis of the output generated by the Implementation Under Test (IUT) in consequence to those stimuli. In all but trivial scenarios, these parts can form a major part of the complexity of

the test environment. Static, pre-computed stimuli (test pattern) and output expectations can only be used for rather low-complexity IUTs.

If the IUT contains multiple state machines, processes and timers interacting with each other and with the external world, hence, the IUT is serving communication protocols at one or more of its interfaces, the test must follow sort of a dialogue with the IUT. The dialogue brings the IUT into a particular state, and allows to explore the IUT's behavior from there. Such a dialogue is usually highly dynamic. The test must react at the specific time when the IUT emits a particular output, and respond with situation-dependent actualized stimuli in conformance with the communication protocol requirements.

During the development of the IUT, it frequently happens that an updated implementation of the IUT shows a minor variation in its response times compared to its predecessor version, but within the limits of the communication protocol. Such changes can cause huge changes to the timing conditions and sequence of events during the test execution. A static test case attempting to predict the exact timing and sequence of events, e.g. based on the earlier implementation, will consequently fail, although the IUT is still functionally correct with respect to the protocol definition. Thus, this flexibility renders a static test behavior useless.

The test environment must implement means to encode and decode the signals exchanged with the IUT, and also likely contain state-machines and timers, according to the communication protocol needs. Furthermore, the same or similar stimuli or analysis might be needed in a plurality of test cases, therefore, exporting these functionalities into a collection of sub-routines pays off quickly. With this partitioning, the test case is significantly simplified by operating on an abstracted level.

In the scope of a UMTS-L1 SoC, the signals exchanged at the air interface towards the base station (Node-B in UMTS terminology) are by far the most complex signals to deal with in the test bench. Effectively, the test bench must emulate the behavior of the base station in both directions, downlink (stimuli) and uplink (responses). To reduce the efforts, we have excluded the radio frequency aspects, hence, concentrated solely on the baseband behavior. Also, behavior beyond the scope of layer 1 are not modeled in this environment. Besides the generation of protocol-conform content, the downlink generator also includes filters for signal shaping[3] and a model representing signal distortions by noise and volatile signal reflections, modeling the mobile phone moving relative to the base station and buildings.

[3]UMTS defines a low-pass filter with root-raised cosine transfer function in order to reduce inter-symbol interference on the bandwidth-limited transmission channel

6.5.2 The Library Approach

The test bench for the UMTS-L1 SoC is supported by a software library containing a collection of stimuli generators and analyzers for various types of interfaces. It strictly follows a responsibility partition paradigm. The test case, as the application using the library, defines what to test. The library supports the test case in providing any low-level communication issues towards the IUT, but without actually being bound to any particular test strategy (Fig. 6.8).

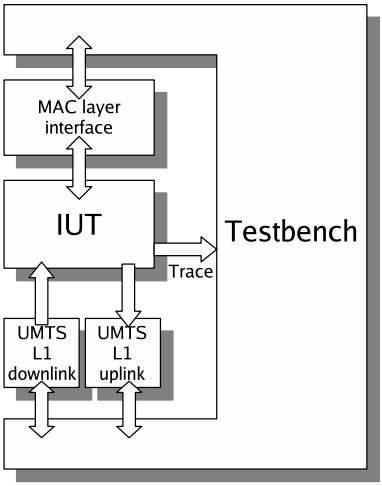

Figure 6.8. Library-supported test bench scenario.

The modular, re-usable design of the generator/analyzer library allows to easily combine its building blocks in various scenarios. Thus, it cannot only be used to test the IUT as a whole, but also, it can be used in other configurations to test sub-components of the IUT.

The collection of stimuli generators and analyzers supports many interfaces, including proprietary protocols, such as chip-internal and external bus interfaces, as well as commonly known standard protocols, such as ATM[4] and UMTS-L1, and can easily be extended to support further protocols.

The library structure used for testing the UMTS mobile system is represented in Fig. 6.9.

Monitoring the behavior of the library, as well as the IUT, during the test execution is crucial for rapid debugging. Two aspects are considered, trace and diagrams.

[4] Asynchronous Transfer Mode, protocol used in wire line communication.

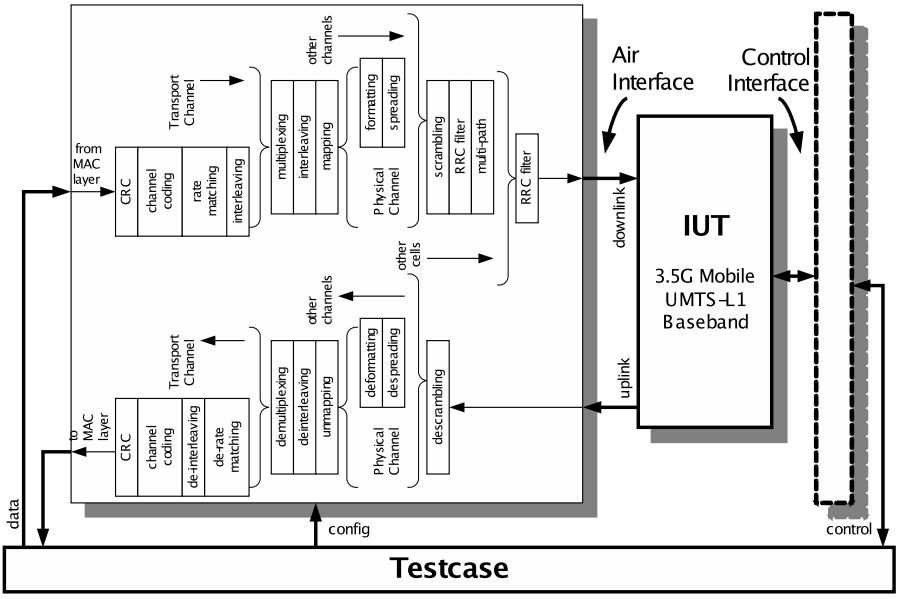

Figure 6.9. UMTS stimuli and analyzers.

Trace

In addition to providing just the bare stimuli signals, a rapid debugging of problems in the IUT or in the library requires that the library informs about its configuration and internal activities in a human-readable format. Especially in the case of UMTS-L1, the data exchanged with the IUT are (almost) random bit patterns, so debugging a problem from this information is quite impossible.

While a general-purpose debugger allows to inspect all variables in a universal but unstructured way, this is complemented by the trace capabilities built into the software, which show relevant information situation-dependent and in protocol context. This helps in particular in analyzing and understanding dynamic variations of the information, especially in large data sets. The library therefore provides extended trace capabilities, of which the example in Fig. 6.10 gives an impression.

It shows the bits received on the uplink Enhanced Dedicated Physical Control Channel (E-DPCCH), and how the dynamic physical configuration of the associated Enhanced Dedicated Physical Data Channel (E-DPDCH) is determined following the standards-defined algorithm. Obviously, reading the trace requires a significant amount of knowledge on the protocol details.

Likewise, the IUT also emits similar trace information about its internal operation. However, that trace is somewhat restricted due to the limitations imposed by the processor and the communication channel. The IUT trace gen-

```
E-DPCCH EDPCCH 0:2 tti=2ms gain=162L2560

Slot-Data
  01110100 10
CodeWord=E4735748 decoded=00000055 ETFCI=10 RSN=2 HAPPY=1
    Configure 0:2:0, TbsTable 0, TTI2ms, ETFCI=10
      trBlockSize=166, blocksPerTti=1
      bitsPerTti before RM Nej=582, Ntx1=15, PLnonmax=0.96 (558.72 bit),
      PLmax=0.33 (192.06 bit)
      Allowed Configs:  N256,N128,N64,N32,N16,N8,N4,2xN4,2xN2,2xN2+2xN4
      Set0:  N256,N128,N64,N32,N16,N8,N4,2xN4,2xN2,2xN2+2xN4
      Set1:  N8,N4,2xN4,2xN2,2xN2+2xN4
      NeDataj=960, phyConfig=N8
E-DPDCH1 frame:slot=0:0 SF=8 Ch=2 I delay=0:0:0 gain=316L2560
Slot-Data (L=320)
  01111100 00000000 01010011 01100101 10010001 01101100 01000000 10100011
  11011110 00110110 11100111 11111100 11010111 10101100 00000001 00011010
  11111000 01000001 01010011 01100111 10001110 11100010 11010111 01100001
  11000000 00000000 00000101 01100101 01011111 00101011 01100111 00000100
  10010001 01001100 01000100 01101101 11010100 01101101 00000010 11011110
```

Figure 6.10. Trace of the Enhanced Dedicated Physical Control Channel (E-DPCCH).

eration is part of the firmware, and constrained to a limited cycle count. It is communicated to the outside world using a limited bandwidth serial interface. This trace therefore concentrates more on high-level aspects. Nevertheless, a flexible management of the over 600 observation points still allows to look into selected details despite those limitations.

Ergonomic, human-readable trace messages and limited transmission bandwidth are conflictive constraints, requiring a highly effective information compression method. Consider that the above trace excerpt covers a period of 660 µsec on one communication channel, while the UMTS mobile has to cope with around 20 simultaneous channels in up- and downlink, at a total trace link capacity of only less than 180 bytes during this time period to transport all that information to the outside world.

The population of observation points is quite floating during the development phase. Therefore, this method needs to be sufficiently extensible and easy to handle, so that the software designer can add or modify trace observation points at any time without much overhead.

Still, a considerable amount of resources of the SoC, in terms of processor cycles, buffer and code memory, and communication paths implemented into the hardware, need to be reserved for the purpose of tracing. The ability to look deeply into the behavior of the SoC is crucial for success, not only during the development, but also in field- and certification tests, as well as for long-term maintenance.

The IUT trace should be consistent through all different platforms that the HdS code runs on, that is, show the same trace messages in the final product as in a unit test of a small software module. Obviously, some platforms are intended for greater visibility onto the system's internal behavior, and therefore provide better capacity, e.g. the software unit module test is not at all bandwidth-limited with respect to the trace load. These platforms therefore can display additional trace information, which cannot be implemented in the final product platform.

6.5.3 Diagrams

Further insights into the operation of the IUT and the stimuli library are gained by means of graphical diagrams. The graphical representation helps to comprehend certain contexts more quickly than analyzing a pile of textual trace messages.

A dedicated tool collects data from various observation points in the stimuli library and displays them in a number of diagrams in real-time during the test run. In the context of UMTS, especially power-over-time, polar In-Phase / Quadrature-Phase (I/Q), and code-domain diagrams of individual channels or composite signals, are useful.

The example in Fig. 6.11 shows the UMTS compressed mode gap with pre- and post-power boosts. During the gap, the transmission from the base station is stalled for a short moment, so that the mobile phone can measure the signal strength from other base stations to eventually decide for a hand-over.

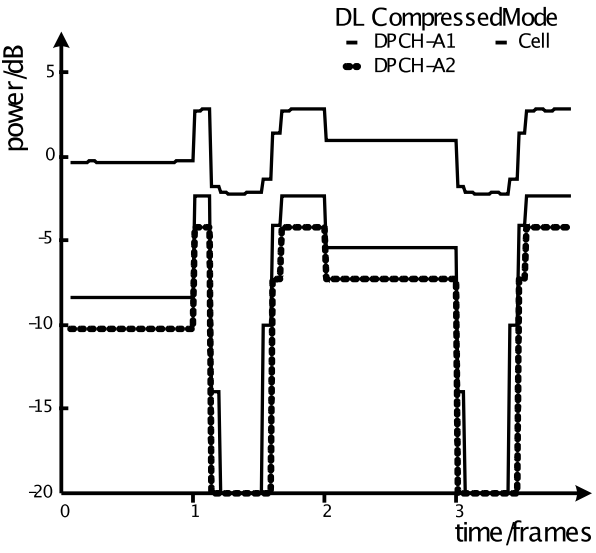

Figure 6.11. Compressed mode diagrams.

Similar graphical representation is available, real-time or offline, based on the trace information emitted by the mobile phone.

6.6 Summary

In this chapter, we consider firmware development in the context of the rapid growing complexity of modern wireless communication standards. System integration and verification becomes a major challenge in the overall development process of mobile platforms. Raising the efficiency in this area is a big lever for shorter time-to-market and reducing R&D costs.

We presented a twofold strategy to reach this goal. On one hand, we aim at controlling the system complexity by choosing appropriate hardware / firmware architectures following the design paradigms given in Sect. 6.2. On the other hand, we apply an advanced system integration and verification methodology based on three major elements, (a) an efficient reuse-oriented system level design flow, yielding a Virtual Prototype, (b) a single source hardware / software interface specification guaranteeing consistency and correctness by construction, and finally, (c) a sophisticated verification concept including software-based stimulus generation and response analysis integrated in a regression-capable test suite, which is applicable to both the Virtual Prototype and the final hardware.

The described development strategy has successfully been proven in a recent UMTS 3.5G product development and will be further applied and optimized for future products.

References

[BSH+07] M. Brandenburg, A. Schöllhorn, S. Heinen, J. Eckmüller, and T. Eckart. From algorithm to first 3.5G call in record time: a novel system design approach based on virtual prototyping and its consequences for interdisciplinary system design teams. In *Proceedings of DATE*, 2007.

[HS06] S. Heinen and M. Steinert. Virtual Prototyping for a 3G baseband chip based on VaST CoMET/Synopsys System Studio Cosimulation. In *Proceedings of SNUG Europe*, 2006.

Chapter 7

GENERATION AND USE OF AN ASIP SOFTWARE TOOL CHAIN

Sterling Augustine, Marc Gauthier, Steve Leibson, Peter Macliesh, Grant Martin, Dror Maydan, Nenad Nedeljkovic and Bob Wilson

Abstract Software-development tool chains are hardware-dependent by their nature, because compilers and assemblers targeted to specific processors must generate target-specific code. However, a processor that is both configurable and extensible, with a variable instruction set architecture (ISA) melded to a basic architecture compounds the problems of adapting the software development tools to specific processor configurations. The only tractable way to support such extensible processor ISAs is through a highly automated tool-generation flow that allows the dynamic creation and adaptation of the development-tool chain to a specific instance of the processor. To be of practical use, this process (automated tool generation) must transpire in minutes. This chapter discusses the issues of application-specific instruction set processor (ASIP) configurability and extensibility as they relate to all the elements of a software development tool chain ranging from an integrated development environment (IDE) to compilers, profilers, instruction-set simulators (ISS), operating systems, and many other development tools and middleware. In addition to drawing out the issues involved, we illustrate possible solutions to these hardware-dependent software (HdS) problems by drawing on the experience of developing Tensilica's Xtensa processor, as an example.

Keywords: Software Development Tools, Configurable and Extensible Processors, ASIP, IDE, ISS, Xtensa

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

7.1 Introduction

Software-development tool chains are of necessity hardware-dependent, in that they must compile, generate, support, and deal with software that is ultimately destined to be executed on a particular processor or family of processors (even if that software is written to be portable). Because these software-development tools can be complex, writing them to be specific to only one processor type and generation at a particular point in time is very inefficient. Open-source tools such as those developed by the GNU project are meant to run on many different types of target processors and thus must contain at least rudimentary features that enable open-source developers to re-target them with a reasonably contained effort. But the world of processors encompasses more than just a variety of fixed-ISA processor architectures. It now includes configurable and extensible processors, making it a much more complex and interesting world.

If we look at configurable and extensible processors, the need to create a dynamically adaptable software tool chain for every variant of a configurable processor adds considerable complexity to the tool-generation task. Rather than manually modify the tool chain every few months or years along with the slowly evolving architecture of a fixed-ISA processor, the engineering team for a configurable, extensible processor must provide an automated flow that can generate a new tool chain for each new configuration in minutes or at most hours. During system development, a development team may create new processor architecture variations at the rate of one every few minutes, day in and day out. You must multiply this variability over a worldwide set of design teams to approximate the actual number of processor variations being created. Perhaps most important: these processor configurations are created by system developers rather than by processor developers or the associated tool-chain developers, and the system-design community has little desire or ability to modify the development-tool chain. They merely want to use it to develop systems. They must have automation to make the task feasible.

This chapter discusses the challenges of creating software-development tool chains for configurable and extensible processor systems used for generating specific instances of application-specific instruction-set processors (ASIPs). As will be seen, this kind of hardware-dependent software is both vital and practical. It enables a viable SoC development methodology based on processor extension and the application of ASIPs to real product requirements. The existence of commercial configurable processors is proof that such methodologies are possible. That such technologies have succeeded in SOC designs over a wide application range testifies that such methodologies are practical.

7.2 Range of Processor Configurability

Several examples of configurable and extensible processors exist. A good summary is found in [IL06]. One such example is Tensilica's Xtensa processor, which has been well documented in [Gon00], [WKMR01], [RL04] and [Lei06]. Processor configuration and extension have a very wide scope: configurable and extensible Xtensa processor features include:

- Register file size
- Endianness (little or big)
- Addition of special function units such as multipliers, multiply-accumulators (MAC), floating point units, and DSP instruction units
- Local-memory interfaces for local instruction and data memory (RAM and ROM), and a generalized local-memory interface
- System-memory interfaces for attachment to global on-chip buses
- Debug, tracing, and JTAG ports
- Timers, interrupts, and exception vectors
- One or more load-store units
- Multi-operation VLIW instructions with flexible instruction encoding allowing intermixing of native 16- and 24-bit instructions with multi-operation 32- and 64-bit instructions
- Five- or seven-stage pipeline
- Definition of ports and queue interfaces of arbitrary width (as wide as 1024 bits) that connect directly into and out of the processor's datapath. Queue interfaces can be attached directly to hardware FIFOs. More than 300 such interfaces can be configured on a processor.
- Instruction extensions, defined in a Verilog-like language called TIE (Tensilica Instruction Extension) to define application-specific instructions. A TIE compiler translates these instruction-extension descriptions directly into hardware execution units that are automatically embedded in the processor's datapath. These extensions are under the control of the software tool chain. TIE allows a designer to specify input/output arguments of an instruction, storage elements and registers (existing and newly created), and the instruction syntax and semantics.

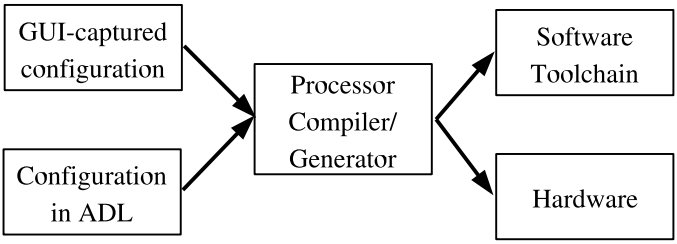

Figure 7.1. Processor generation via GUI-captured configuration and ADLs.

7.3 Models for Generating Software Development Tools

In general, you configure an extensible processor using one of two methods or a combination of both. The first method involves some kind of Graphical User Interface (GUI) that allows you to select configuration parameters using check boxes or other selection tools. Very often these parameters specify coarse-grained structural elements such as the cache sizes, the presence or absence of various interfaces, and the inclusion or exclusion of special functional units such as multipliers or floating-point units. Instruction extension often requires an Architecture Description Language (ADL, see [MD06] for a survey)—a specialized language particularly suited for describing instruction set architectures or parts thereof. For example Tensilica's TIE is an ADL—one that has been developed over a number of years and over several architectural generations.

Both ADL-based descriptions and GUI-captured configuration parameters feed into a processor compilation and generation process as illustrated in Fig. 7.1. There are two basic models for creating a software (SW) tool chain based on the generation of a configured processor. These two models can be roughly categorized as the "static compiled tools" approach, and the "dynamic run-time tools" approach.

Figure 7.2 illustrates the static compiled tools approach. In this approach, the act of processor generation also generates the source code for all software tools (compiler, ISS, linker, etc.) and target software (operating systems, libraries, etc.). This source code completely captures the particular configuration and extension properties of the particular instantiation of the configurable processor being developed. These development tools are then compiled for a specific host development platform using standard tools and they are then used by SW developers on the host platform to develop, verify, and debug code for the target processor.

In the dynamic run-time tools approach, the act of processor generation does not produce the source code for all the software tools. Instead, it produces parameter files and possibly source code for part of the software tools (for example,

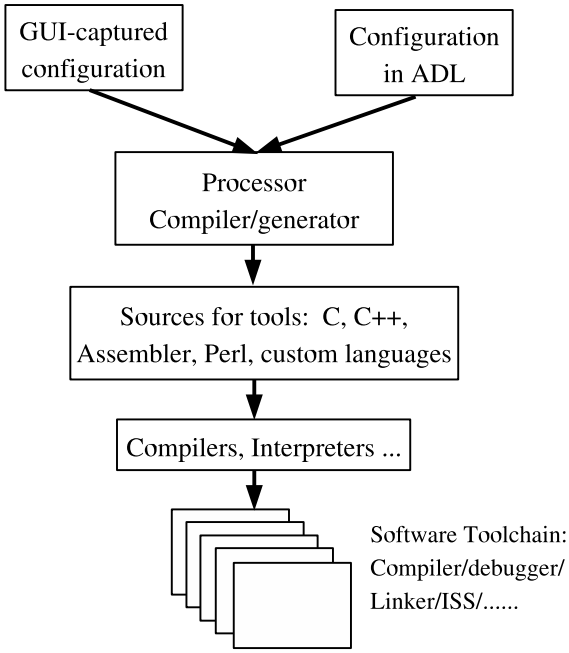

Figure 7.2. Static compiled tools generation.

simulation descriptions for new instruction semantics, which can then be compiled into dynamic libraries). These parameter and control files are loaded or accessed at run-time by parameterized software tools. For example, a compiler might access a parameter file at run time to determine the configured size of a processor's register files; an ISS might dynamically access a compiled library to find the semantics of an instruction extension. Figure 7.3 shows the dynamic run-time tools-generation approach.

Each approach has advantages and disadvantages. The static compiled tools-generation approach is simpler to implement. There is no need to separate the tools into configuration-independent and configuration-dependent portions. The static approach can produce a more optimized set of software tools (optimized for execution time, not necessarily optimized in features or quality of results). For example, a compiler will often employ many data structures with arrays indexed to the number of registers or other machine resources. Using the dynamic approach, the compiler writer must either set maximum limits on the resources or must use dynamic data structures. Either choice can result in execution-time inefficiencies. Using the static approach, the compiler writer is free to use static limits based on the actual processor configuration.

The dynamic approach makes it much easier for the system developer to create and manage multiple processor configurations. A tool chain can be very

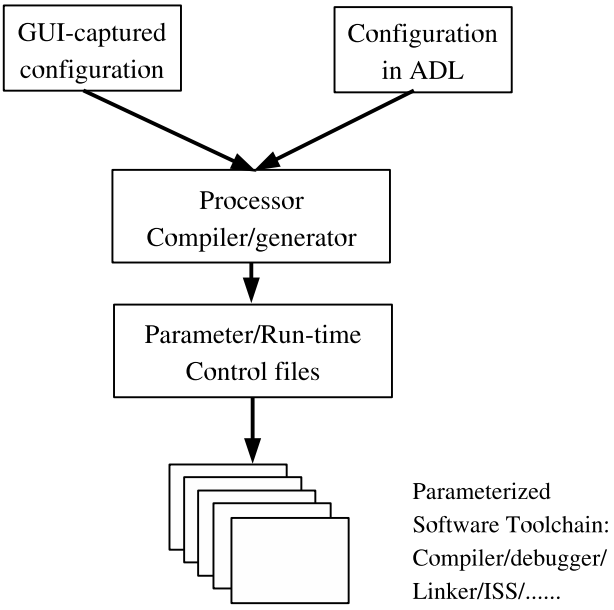

Figure 7.3. Dynamic run-time tools generation.

large while configuration files and dynamic libraries can be quite small. Building and potentially downloading large files from a vendor's server can be very time consuming. Storing many tool chains for several configured processors can also consume a lot of resources. Considerable storage space can be saved if the only variation in a software tool chain from one configured processor to the next is a small set of configuration files and dynamic libraries.

Developing processor extensions, like developing software, is an iterative process dominated by making small changes, testing, evaluating, and repeating. The difference between generating a new tool chain in one minute versus one hour makes a substantial difference, both qualitative and quantitative, that can lead to large increases in developer productivity. If tool-chain generation required a day or a week, the resulting inefficiencies would make use of configurable processors unwieldy and impractical.

The choice of a strategy can be made primarily on the choice of the processor-generation strategy. If the processor is designed by configuring and extending an existing processor, then much of the tool set remains constant through the configuration process. Excess time will be needed to re-generate an essentially similar output each time unless the dynamic approach is used. However, if the processor is designed primarily by compiling an ADL description from scratch each time instead of extending an underlying base architecture, then the static compiled tools approach may make the most sense.

The nature of the target software brings in additional considerations. A C library or an operating system can be compiled with a configuration-specific compiler or these software packages can be compiled to a base processor configuration that is common to an entire family of configured processors. Compiling to a base processor configuration allows for binary compatibility across an entire family and helps third-party vendors to distribute binaries suitable for multiple configurations. Using a configuration-specific compiler can, in some instances but not others, make dramatic performance differences. An MP3 binary targeted for an audio DSP will certainly use a compiler configured for that DSP while the C library printf function might be compiled with a base configuration.

If processor generation is not based on extending an existing base architecture, then trying to support the entire configuration space with one tool driven by configuration and extension parameters and controls can be extremely complex. The software-development tools must encompass the complete design space in their parameters and controls. This requirement may lead to a very difficult development process and a difficult maintenance problem, especially if significant evolution occurs in the processor's architecture from one generation to the next.

Separate from the decision of whether to employ static or dynamic tool generation is the decision of whether to build the processor-specific tools on a centralized server or on the client (i.e. developer) side. Building tools on a centralized server is simpler and more reliable. Build processes can be quite complex and trying to replicate a build process on the large variety of customer systems can be problematic. On the other hand, building tools on the client side is potentially faster and simpler for the system developer and is easily scaled as the number of teams generating new processors increases.

7.4 Evolution of Tool-Development Approaches

The Tensilica Xtensa processor-generation process originally employed a static compiled tool-generation approach, as discussed in [Gon00]. This choice was made predominantly for expediency; Tensilica was a new company and needed to release a product quickly. However, the need for a dynamic system soon became apparent. Extra processor-configuration options and the evolution of the TIE language including the added capabilities of increasingly powerful processor extensions had two effects. First, customers started to build and experiment with a much larger number of processor configurations. Second, the development tools also became richer and more complicated, and therefore took more time to build. Both effects increased the advantages of the dynamic approach.

With the move to a dynamic tool-generation system, Tensilica had to choose whether to build the dynamic parts on a centralized server or client-side at the developer's site. While considering these options, it became clear that processor configuration and extension had very different usage models. Extensibility requires much more interactivity. Writing an extension instruction using an ADL, just like writing an application program in a high-level language (HLL), is an interactive activity involving several iterations to fix the syntax, several more to get the semantics right, and several more to evaluate the effects of the new instructions on the targeted application program. Allowing the system developer to make a processor-configuration change and to evaluate that change in a minute tremendously improves productivity over a system that requires even 10 minutes, let alone an hour, to make such changes. In contrast, simple processor configurability requires much less interactivity. For example, when configuring an additional interrupt, actually using the new interrupt in the application code and creating devices in simulations that trigger the interrupt take a lot of time. Waiting 10 minutes or even an hour to generate the development tools in such cases will not make much difference.

While extensibility requires much more developer interactivity, configurability touches larger portions of the development-tool chain. The more things built on the client side, the longer the time required for the client-side build, blunting some of the benefit of the dynamic approach. Therefore, Tensilica chose a hybrid approach. TIE files (which add processor extensions) are compiled dynamically on the client side while almost all configurability aspects are largely built into the tools in a configuration-build process run on Tensilica's servers.

There are a few exceptions to this general principle. For example, linker scripts can all be regenerated on the client side—which is necessary because users can change system memory maps long after the chip is built, for every new board that incorporates that chip. Another example is in the ISS, which allows the user to experiment with different cache sizes at run time to support design-space exploration. Such exceptions are relatively few.

Over time, the TIE language and its usage evolved to support the description of much of the Xtensa processor's core microarchitecture and not just the extension instructions. Thus TIE moved from being a language for add-on instructions to be more of a mainstream ADL. With this evolution, the TIE compiler evolved to support some client-side software generation combined with server-based generation. More on TIE can be found in [RL04]. Reference [WKMR01] describes TIE as it was introduced. This reference comments that, as of the time of the paper (2001), "TIE is not intended to be a complete processor description language." Although TIE is still not a complete description language for Xtensa processors, it has evolved and grown considerably in its descriptive ability.

The hybrid Tensilica approach to tool generation is illustrated in Fig. 7.4. In this approach, the processor is described by three basic parts:

- Special-case processor infrastructure—parts of the Xtensa processor hardware are described in Verilog or in parameterized Verilog form rather than TIE. This portion of the processor description includes special instructions for synchronization and cache handling. To ensure that the software tools have a complete picture of the ISA, hand-coded C describes these parts. The hand-coded C is compiled into an associated library that's used for the software tools, labeled *special case isa.so*.

- The basic Xtensa processor core, configured by the system-design team including such structural parameters as number of registers, memory interfaces, etc. The basic processor core also includes the core RISC ISA built into every Tensilica processor. This description is captured in the TIE ADL and compiled into core libraries by the TIE Compiler (TC).

- Extensions to the basic Xtensa core processor that define an application-specific ASIP. These descriptions are captured in customer-written TIE, which defines new instructions, new state (registers and register files), and special interfaces into the processor's data path (TIE queue interfaces and ports). TC is used to compile customer-written TIE into user libraries.

The three sets of *isa.so* libraries (which are dynamically linked libraries or dll's on Windows hosts and shared object libraries (.so) or archives (.a) on Linux hosts) are combined to make the configuration-specific library *libisa* for the specific configured, extended Xtensa processor. This *libisa* file then serves as an input into all the various software tools in the tool chain (compiler, debugger, ISS, etc.) to give each tool a view of the configured and extended ISA for which the software is being generated. In addition, other files emerge from the compilation process, such as a core *params* file that describes additional processor attributes. These other files also feed into the software tools. They are conceptually similar to the *libisa* files discussed above. This basic software-tools architecture underpins most of the hardware-dependent tool chain that we discuss in the following sections.

Note that the official name of the *libisa* library is the *xtisa* library (i.e., the Xtensa ISA library). Colloquially within Tensilica we call it *libisa* and these terms are used interchangeably in this chapter.

Figure 7.5 illustrates the basic components of the Tensilica software toolchain. These will be discussed in much more detail in Sects. 7.5 through 7.15.

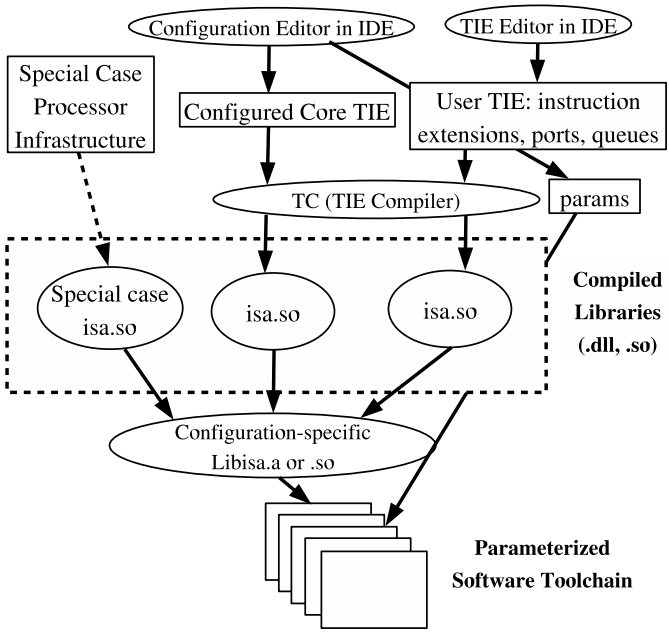

Figure 7.4. Tensilica software tools generation process.

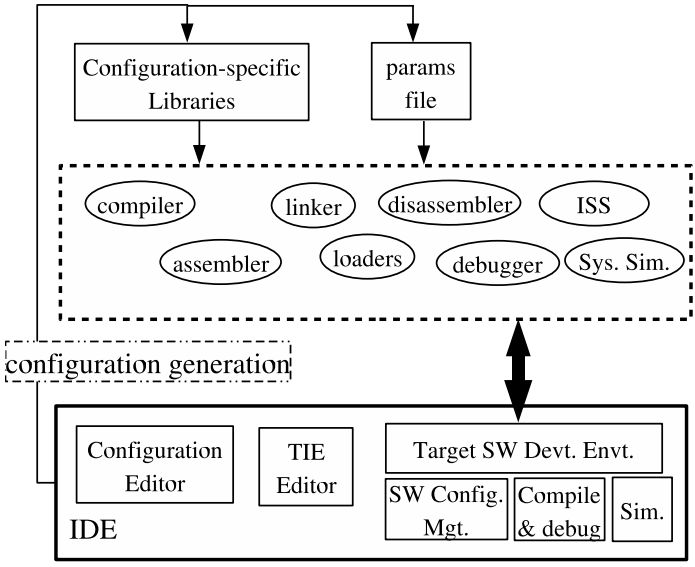

Figure 7.5. Tensilica parameterized software toolchain.

7.5 The C/C++ Compiler

Tensilica's C/C++ compiler, called XCC, is based on the Open64 compiler released as open-source by Silicon Graphics (SGI). Open64 is itself a hybrid of SGI's original proprietary MIPS compiler mated with the GNU compiler, gcc. The gcc portion serves as the compiler's front end: the compiler's language-specific part that parses and analyzes the input program. The optimizing and code-generating back end of gcc is bypassed and the modified gcc front end instead feeds into the SGI compiler's optimizer and code generator. This hybrid arrangement produces a compiler that is highly compatible with gcc yet offers the superior optimization capabilities and flexibility of the SGI compiler.

Tensilica based its compiler on Open64 for several reasons. Using the gcc front end makes XCC highly compatible with gcc, which is the most familiar, most widely used compiler among Tensilica's potential customers. At the same time, Open64 is freely available and it is based on a widely used, commercial-quality compiler, which gives XCC both accessibility and quality.

Open64 was designed to be a high-performance compiler. It generates high-quality code at standard optimization levels and has many more advanced optimizations such as interprocedural analysis, software pipelining, and feedback-directed compilation. Finally and perhaps most importantly, Open64 was designed to be flexible and easily modifiable, which are both critical for a compiler base slated to be enhanced to support processor configurability and extensibility. There are substantial enhancements in XCC that support processor configurability and extensibility.

Before describing the compiler, it is useful to understand the programming model. Processor configurability is not typically exposed to the programmer. XCC, for example, will use a hardware multiplier to implement the C/C++ "*" operator when a hardware multiplier is available. If no hardware multiplier is available in the targeted processor configuration, XCC will emulate the multiplier in software. Thus, the programmer need not be aware of the configured hardware available in a specific processor configuration for normal C/C++ multiplication.

Extension is sometimes invisible to the programmer because XCC has been enhanced with many types of inference. However, extensions must often be exposed because of their nature. Every TIE instruction is directly accessible through C or C++ intrinsic functions. In addition, the TIE language allows developers to define new C data types that are mapped to TIE register files along with appropriate instruction sequences to load and store these data types from and to memory. The C/C++ programmer can use these new data types as if they were native by declaring scalar variables, arrays, or structures of them. Data operations are described via intrinsic functions but register allocation, instruction sequences for loading and storing new data types, and addressing arith-

metic and control-flow generation are all handled automatically just as native data types are. The generated dll's contain information about the side effects and pipelining of all TIE instructions, enabling the C/C++ compiler to schedule these instructions correctly and efficiently. Programming with custom data types is usually much simpler than using C's original data types because algorithms directly map into appropriate operations using application-specific, data-type extensions. There is a substantial reduction of the mental gymnastics required of the programmer when application-specific data types are employed. The reduction in mental complexity reduces errors and improves coding productivity.

XCC additionally supports various types of TIE inference. TIE allows the creation of flexible length instruction extension (FLIX) instructions, which are a code-space-efficient variation of VLIW using variable-length instructions. The TIE developer starts by creating simple custom operations. The TIE developer then defines one or more instruction formats consisting of several operation slots. Finally, the TIE developer lists all the operations that can appear in each operation slot in each FLIX instruction format.

The C/C++ programmer writes code in terms of operations, not instructions and XCC automatically schedules and bundles these operations together into FLIX instructions. The result is a code stream that employs wide FLIX instructions only where multiple simultaneous operations can be performed. Where simultaneous operations cannot be executed, as determined by the compiler, the code stream consists of the simpler, smaller instructions.

TIE also supports SIMD, or vector, versions of either standard or extension scalar instructions. Vector instructions are marked in TIE with properties to indicate that they are vector versions of particular scalar instructions. XCC incorporates a vectorizer that analyzes loops, identifies inherent parallelism, and automatically uses SIMD instructions where possible to reduce loop execution time and code size.

TIE supports fused instructions or fusion, in which one new instruction is equivalent to a sequence of other instructions. For example, a TIE developer might write a multiply-shift instruction that multiplies two variables and shifts the result by a fixed amount. Instruction fusion speeds execution and reduces code size. XCC has a graph-based matching phase that searches for and automatically infers fused instructions from the processor's base instructions. The instructions used for fusion can be standard processor instructions or TIE-defined extension instructions.

XCC is also capable of combining the basic inference techniques. For example, XCC can infer the fusion of a SIMD instruction that resulted from the vectorizer, and it might schedule the resultant operation inside a FLIX bundle to maximize parallel operation.

The following is an example of a simple multiply-accumulate (MAC) instruction captured in TIE. This uses a register file and state variables, and accepts 24-bit values from the register file, multiplying them together and accumulating the result in the state variable. The resulting MAC instruction can be invoked in C/C++ source code via a pragma or may be inferable by the XCC compiler.

```
regfile XR 24 16 x
state ACCUM 56

operation MAC {in XR in0, in XR in1}{inout ACCUM} {
  wire [47:0] prod = in0 * in1;
  assign ACCUM = prod + ACCUM;
}

schedule mac_sched {MAC} {
  use in0 1; use in1 1;
  use ACCUM 2;
  def ACCUM 2;
}
```

Configurability and extensibility are supported as integral parts of XCC. The TIE compiler generates a shared library that describes the properties both of TIE-defined extension and standard instructions. The generated library includes descriptions of the resources used by each instruction during every execution cycle, the operands and immediate ranges used, and the FLIX bundling constraints. XCC uses this library to build resource tables for each instruction. The compiler then uses these tables to schedule operations and bundle them together into FLIX instructions. A pattern-based code generator performs the code selection using rules that are dynamically created for each configuration based on actual processor configuration parameters.

TIE data types are represented in the internal representation of the compiler as native data types. Loads and stores of these TIE data types are represented as standard load and store operations operating on these dynamically configured data types.

Other instructions are represented as intrinsic functions. There are two classes of intrinsics. The first, called intrinsic operations, are well-behaved TIE instructions that produce no side effects. Intrinsic operations are subject to the complete set of standard compiler optimizations including loop hoisting and common sub-expression elimination. TIE instructions that produce side

effects are represented as intrinsic calls. While some parts of the XCC compiler will conservatively treat these instructions as function calls, the compiler knows the side effects for each instruction in the context of different optimizations and can make appropriate decisions. For example, alias analysis knows that a TIE load can never write to memory regardless of address computation complexity, even if the load also reads or writes from arbitrary TIE states.

XCC serves as the basis for XPRES, a tool that can automatically generate TIE processor extensions based on C or C++ source-code files. XCC analyzes the input C program for vectorization, fusion, and VLIW opportunities. XPRES then performs a global search over combinations of potential TIE extensions to suggest one or more set of extensions that provide different amounts of performance improvement at different cost points (gate counts). XCC can then automatically infer and use all of the XPRES-generated TIE instructions from the same or a sufficiently similar input program. The input program might be a standard C/C++ program or might itself be a program already taking advantage of base extensions. XPRES bypasses the need to explicitly use TIE-based intrinsic functions in many cases. For example, a developer might manually write TIE to implement scalar fixed-point instructions and then rely on XPRES to create vector versions of those instructions.

7.6 The Assembler

Like assemblers for many other processors, Tensilica based its assembler on the GNU assembler and its chain of releases. The most difficult part of an assembler to create is the code to handle symbols, symbol files, and relocations. These issues were solved in the GNU tool chain a long time ago. As a result, all assembler ports including the Xtensa derivative of the GNU assembler share this proven code.

Tensilica's assembler extends the GNU assembler by replacing most of the table-driven aspects of the GNU assembler with the *libisa* mechanism discussed earlier. Instead of a hard-coded table of instructions, operands and encodings, the assembler opens a dynamic library and programmatically determines the instructions present and their representations. This approach allows the assembler to target any configuration without rebuilding the assembler itself. For example, Tensilica could change the ISA-defined bit representation of an instruction and the assembler would continue to work without rewriting or even recompiling the assembler source code.

By the admittedly imperfect lines-of-code measurement, the only GNU-based assembler close to Tensilica's in complexity is the assembler used for the Intel's IA64 architecture (Itanium). Three factors contribute to the complexity of Tensilica's assembler. First, using dynamic libraries to describe the instruction set, operands, register files, and other processor resources adds significant

complexity when compared to hard-coded tables. Second, various trade-offs in the Xtensa processor architecture have pushed complexity into the assembler. Third, configurability can increase the burden on assembly-code writers and Tensilica wished to ease that burden, not add to it.

Some things that the Tensilica assembler handles differently than other assemblers include:

- **Instruction Relaxation:** If a written instruction uses a constant that cannot be encoded as an operand, Tensilica's assembler converts that instruction into a series of instructions that match the semantics of the original instruction. However, for a configurable processor, the optimal instruction sequence could differ depending on the options selected. Tensilica's assembler therefore dynamically determines the options that are present and then chooses the optimal instruction sequence. For example, an instruction sequence could take a different amount of space on different processor configurations, which would require assembly-code writers to be conservative when they estimate the range of a branch. Because Xtensa branch instructions have relatively limited range, using the more conservative branch-instruction sequence can degrade performance. With the Xtensa assembler's sophisticated relaxation mechanism, the assembly-code writer can use the shortest-range, most-efficient branch instructions in their code and can be confident that the assembler will substitute a less efficient branch-and-jump sequence automatically, but only when necessary to reach the branch-target address. This feature relieves the assembly-code writer from the burden of tracking ranges. This feature does not derive from configurability or extensibility per se, but is definitely a side-effect of the processor architecture and thus hardware-dependent.

- **Instruction Scheduling:** Again, this feature relieves the assembly writer of much of the burden of tracking which instructions are present in a specific processor configuration and how long these instructions will take to execute. For example, the Xtensa ISA supports several different sequences for loading large immediates. The optimal sequence and its scheduling characteristics are configuration-dependent. The assembly-code writer can simply choose the simplest variant, confident that the assembler will later choose and schedule the optimal one. Furthermore, some processor configurations contain Tensilica's FLIX multi-operation instructions, where one instruction contains several individual operations, all executed at once. Processor configurations incorporating FLIX instructions often exhibit dramatic performance improvements, but it is impossible for a third-party company to write assembly code that uses an unknown FLIX scheme. Therefore, the assembler scheduler bundles

operations into FLIX instructions, which allows assembly-code writers to target any configuration, ignore FLIX, and still get the performance advantages of FLIX instructions with processor configurations that have it.

- **Instruction Alignment:** Like most modern architectures, Xtensa processors fetch instructions in power-of-two-sized words (specifically four or eight bytes per instruction fetch). However, the Xtensa ISA is unusual in that it consists of 16- and 24-bit instructions (and, for some extended configurations, 32- or 64-bit FLIX bundles). An Xtensa processor takes an additional single-cycle, taken-branch penalty if an instruction crosses a fetch boundary. The mismatch between the instruction size (2, 3, 4, or 8 bytes) and the instruction-fetch width (4 or 8 bytes) potentially makes this branch penalty common—especially combined with the other assembler transformations. To avoid this performance-robbing branch penalty whenever possible, the assembler therefore automatically aligns branch targets by:

  1. Converting 2-byte instructions into equivalent 3-byte instructions;
  2. Inserting padding in unreachable locations; and
  3. Inserting no-ops in locations where the processor would otherwise stall. This last feature is enabled by the instruction scheduler.

As above, this last characteristic does not arise from configurability or extensibility considerations, but it is definitely hardware-dependent. A code developer can turn off all these transformations to get classic assembler "What you write is what you get" behavior. In fact, some assembly-code writers new to the Xtensa architecture and Tensilica's tool-chain do so. However, most eventually turn these features back on when they see how much easier the code is to write and how much more efficient it is after the assembler optimizes it.

7.7 The Linker

A linker serves as the intermediary between object files generated by the compiler or the assembler and the executable code required to run the program on a target processor. A formal definition of a linker is

> ". . . a computer program used to link. The linker takes one or more object files, assembles them into blocks which are to fit into particular regions in memory, and resolves all external (and possibly internal) references to other segments of a program and to libraries of precompiled program units. This prepares relocatable object code for execution, thus producing a binary executable file" [Lap01], p. 280.

To elaborate on this definition, the linker is responsible for placing the code, data, and other sections of object files into an executable file. Object files contain symbolic references and the linker must resolve these references to specific addresses based on target hardware. Symbolic references in object files are identified by relocation records, often just called *relocations*, which specify the symbolic address being referenced, the position in the object file where the reference is located, and a relocation type. Each processor architecture defines a set of relocation types that corresponds to the different kinds of symbolic references supported by that architecture. The simplest relocation type is an address. This relocation type is often used in data sections. The linker computes the address value for the data and inserts it directly into the executable image at the location specified by the relocation.

Other relocation types correspond to specific processor instructions. For example, some processor architectures synthesize a 32-bit address constant with a 2-instruction sequence, where the first instruction sets the high bits of a register and the second instruction adds the low bits. Each of these instructions needs a different relocation type. For the first "set-high" relocation, the linker needs to insert the high bits of the address value into the instruction's immediate field. For the second "add-low" relocation, the low bits of the address must go into the immediate field of a different instruction. In both cases, the relocation type is tied to a particular manipulation of the address value (e.g., extracting the high or low bits) and also to a particular method for inserting the value into the executable (e.g., shifting and masking into the immediate operand field of a particular processor instruction).

Handling relocations for extensible ISAs presents a dilemma. If a new instruction, such as a conditional branch, supports symbolic references, a new relocation type must be defined to be used with that instruction. Otherwise, the linker will not know how to apply relocations on that instruction. Unfortunately many software tools handle only a fixed set of relocation types for a particular processor ISA. Generating a new and arbitrary set of relocation types for each processor configuration is not viable with such tools. The Xtensa architecture resolves this dilemma by defining a single generic relocation type for most instructions. We then rely on information from the Xtensa ISA library to provide the relocation details. To handle the FLIX multi-operation instructions discussed earlier, we actually define a set of these generic relocations, one for each operation slot in a FLIX instruction. By themselves, these generic relocations do not provide enough information to be useful. To apply one of these relocations, the linker must decode the instruction, locate the immediate operand field, evaluate an instruction-specific function to transform the address value into an immediate operand value, and then encode that value back into the instruction. All of these operations are performed via the Xtensa ISA library.

This is a powerful and flexible approach to configurable relocations but this approach also has some disadvantages. The advantage is clear: new instructions can be relocated without any changes to the linker. The disadvantages are more pragmatic. Resolving relocations requires substantial information from the Xtensa ISA library, which is implemented with shared object plug-ins and runs on the host development platform. The full set of relocations are currently not supported on target Xtensa systems. This creates a problem for various dynamic loaders that read object files and resolve their relocations on target systems; the set of such loaders includes the Linux kernel module load. There is no fundamental problem here—it would be quite possible to generate the necessary relocation information in a form that could be included in a dynamic loader running on the target. Another drawback is that applying Xtensa relocations is considerably slower than for traditional fixed relocation types because of the added complexity.

Long ago, the GNU tool chain incorporated the notion of "linker scripts" that allow developers to define memory maps for linking purposes. These maps include the locations for segments in local memories, reset vectors, interrupt vectors, and other vectors. Tensilica's processor generator automatically generates a memory map for each processor configuration during the processor-generation step. From that map, we generate linker control scripts (several different kinds, for different uses—e.g. ISS, RTL simulation, FPGA prototyping, and production target code). This default memory map is sometimes sufficient but often requires manual modification to match the actual set of local and global memories present or to reflect application-specific software partitioning of these memories.

Using GNU-style linker scripts directly can be difficult because they are very complicated. This is especially true for ROMing linker scripts for Xtensa processors because the vector locations are completely configurable and the resulting linker scripts are complex. To ease this burden, we have developed some intermediate tools that generate the correct GNU linker scripts from simpler memory maps. Another tool can be used to modify existing linker scripts based on incremental changes to the memory map, rather than requiring that a new map be generated from scratch whenever a change occurs.

7.8 The Loader

Although the linker will create a large, complete, and self-contained object file for a single processor, some cases require special loaders. The term "loader" is rather generic and several parts of Tensilica's tool chain can carry out loading functions, such as:

- GDB (the debugger), which loads code onto a target board using On-Chip Debug (OCD) capabilities

- The ISS, which loads code into a simulated target processor's simulated memory
- The Makefile sequence, which builds a ROM image from a ROMable executable
- The Linux kernel module loader
- The Linux user-side dynamic application loader, and
- The Tensilica dynamic library loader, which is used—for example in audio/video applications—where a configured processor may be capable of running many different digital codecs but only has the local instruction memory to hold one or a few of them, keeping the rest in ROM or flash memory to be loaded as needed. The library loader allows loading code without the use of an operating system.

For a normal ELF executable file, the loader loads the ELF image contents into the target processor for execution. If the executable is fully resolved by the linker, the loader uses the ELF program headers (pheaders). If the executable still has some unresolved relocations, the loader may need to resolve the relocation addresses using much the same techniques as the linker does as described in the previous section. However, the Tensilica dynamic library loader does not currently include all the relocation handling code that the linker does, because this added code would make the loader very large and much more complex. The lack of this added code limits dynamic library loading to simpler cases. For example, it is possible to create position-independent codec libraries that can be loaded anywhere in system memory. The actual loading location for these codecs is then determined at run time. This ability preserves some flexibility in using the resulting configured processor and its software in a complex SoC. A similar capability is available in the Linux user-side dynamic application loader.

7.9 The Disassembler

Very little needs to be said about the Xtensa disassembler. Its design very naturally falls out from the structure of our hardware-dependent tool chain. That is, the disassembler's design falls out from the support *libisa* libraries as depicted in Fig. 7.4. The disassembler decodes any object-file line and distinguishes the instructions and operands using API routines. If a program has been compiled and linked with the correct options to support debugging, then the debugger can display meaningful symbol names in the disassembly views instead of displaying absolute addresses and unintelligible function entry points.

7.10 The Debugger

Tensilica offers multiple debuggers with Xtensa processors to support diverse customer needs. Underlying this approach is the software library strategy depicted in Fig. 7.4. Of course, the *libisa* libraries are used to retrieve all instruction information: encodings, operands, etc. In addition, a debugger support library *(libdb)* is automatically generated along with the processor. *Libdb* contains information required by all of the debuggers—for example, information on registers and their properties: name, size, alignment, saving and restoration, etc. Both internally supported debuggers and various 3rd-party debuggers access *libdb* to obtain all needed information about registers: the fixed registers that are part of every Xtensa processor configuration and the configured registers—TIE-defined registers created as part of the processor's instruction extensions.

Tensilica provides a variant of the GNU gdb debugger. It is called *xt-gdb* and it supports basic debugging. The *xt-gdb* debugger also supports sophisticated single-processor and multiprocessor debugging within the Xtensa Integrated Development Environment (IDE), called Xtensa Xplorer. (The Xtensa Xplorer debug environment is discussed in more detail in Sect. 7.15.)

7.11 Other Software-Development Tools

The assembler, linker, and other software-development tools and utilities such as the *objdump* program and *size* utility, came from the GNU *binutils* project. We modified these tools as appropriate for the Xtensa configurable and extensible processor family and to allow them to deal with configuration and extension issues (for example, we created *xt-objdump*). We donated these changes back to the GNU project.

7.12 Operating Systems and Other System Software

We support the porting of operating systems and other software to Xtensa processors via a hardware-dependent software layer called HAL. Normally, the term "HAL" refers to a "Hardware Abstraction (or Adaptation) Layer". To be more precise, the Xtensa HAL is really a processor description, a PAL (Processor Abstraction Layer), which describes those aspects of a processor configuration that are visible to programmers. These are of particular importance to operating systems. Thus the HAL reflects both relevant software and hardware aspects of the processor configuration. The use of a HAL allows the operating-system writer to create low-level system code that conditionally depends on processor configuration parameters. For example, the operating system should initialize a cache only if that cache is present in the processor configuration. In other words, operating systems need to be made aware of certain processor configuration parameters for proper OS operation.

The primary form of the Xtensa HAL is a small set of source-level header files. With the header-file approach, the OS source code must be rebuilt for each processor configuration. If a binary OS port and distribution is required, the OS binary (usually in relocatable object form) is kept independent of processor configuration by adding an intermediate layer built using the generated header files. A generic binary library version of the Xtensa HAL is provided for this purpose, which may be augmented with a thin set of OS specific files distributed in source form for certain performance critical code sequences sensitive to processor configuration. The HAL header-file strategy is more reasonable today than in times past because many contemporary operating systems such as Mentor Graphics Nucleus, expressLogic ThreadX, and Linux are distributed in source code rather than binary form. Therefore, configuring the OS requires nothing more than the inclusion of the correct HAL configuration header file and OS recompilation.

The HAL header files describe many aspects of the processor configuration-some that an operating system cares about and some that it may not. For example, specific options that describe exception handling and indicating that the configuration includes a zero-overhead loop instruction can maximize the efficiency of an RTOS. The HAL header files also describe the processor's registers and their properties because operating systems generally need to know what registers to save and restore during interrupts and context switches.

The HAL layer is then used as part of an overall RTOS porting strategy for Xtensa processors. An RTOS-porting layer is written once, incorporating the HAL for the configuration and adapting the OS to the processor. This approach saves time when porting subsequent RTOS generations to a particular processor configuration. We have used this approach successfully for several generations of the Nucleus and ThreadX RTOSes. Some OS vendors create their own Xtensa ports by using the generated HAL files and relevant documentation; others involve Tensilica for assistance or for specific questions.

The HAL-based strategy is not used much for fixed-ISA processors, which results in manual recoding of each OS for each new processor generation. The only reason that hand-coded OS porting works at all for fixed-ISA processors is because OS porting occurs relatively infrequently for these processors.

Tensilica provides a very simple runtime called XTOS, which works with customer-specific configurations and supports some OS basics such as interrupts and exception-handling. However, XTOS does not support multithreading or multitasking nor does it include device drivers beyond simple character I/O. A number of Tensilica's customers use XTOS for single-threaded applications when they do not need a full set of RTOS features.

7.13 The Instruction Set Simulator (ISS)

As discussed earlier in this chapter, the initial Xtensa ISS was regenerated with each new processor configuration. We later changed this approach so that the ISS now has a configuration-independent part and configuration-dependent ISS libraries that are created during processor generation and within the TIE-compilation development loop. There is a fairly strong correlation between the ISS configuration-independent and -dependent portions and the processor hardware structure. That is, much of the processor hardware is described in TIE form and the corresponding Verilog processor description is generated by the TIE compiler (TC); the rest of the processor hardware is described in hand-coded Verilog.

The hand-coded-RTL part of the processor tends to describe the processor's infrastructure including the instruction-fetch engine, local-memory interfaces, the load-store unit(s), cache-memory interfaces, write buffers, store buffers, and the PIF (main bus) interface logic. Software analogs of these hardware components constitute the configuration-independent core part of the ISS. Although these parts are "configuration-independent," the processor hardware and the ISS are parameterized. Parameters include, for example, the presence or absence of certain local-memory interfaces (for instruction RAM, instruction ROM, data RAM, and data ROM) and the bit width of these local-memory interfaces. The configuration-dependent part of the ISS tends to correspond to the TIE description that is used to generate the rest of the processor hardware including the instruction semantics (for the processor's base ISA and all TIE instruction extensions), registers and register files, other TIE state, and exception semantics.

The Xtensa ISS is cycle-accurate and directly models the processor pipeline. ISS instruction processing consists of three steps: stall computation, instruction issue, and semantic instructions for the activities occurring within each pipeline stage. The stall computation is cycle-accurate and models pipeline interlocks. The instruction-issue step defines use of registers within each stage and sets up the computation for the stall functions of subsequent instructions. The TIE compiler generates the configuration-dependent parts of the ISS during the processor's server-based configuration build or within the local (client-side) TDK (TIE Development Kit) development loop. The associated *xtensa params* files provide additional configuration-specific parameters and the core and configuration-specific parts are connected to create the full ISS. The ISS uses function pointers and dynamically loaded libraries for efficiency.

Interfaces and signals provide communication between the core and configuration-dependent parts of the ISS. For example, when an instruction wishes to load memory, the ISS has an interface called *mem data in* that is called via the semantic functions generated by the TIE compiler. This function, which re-

sides in the configuration-independent part of the ISS, then determines whether the memory reference refers to local memory, to a cache, or if it needs to be sent out via an interface to a system memory based on the address mapping defined in the processor's configuration.

The ISS uses core signals to communicate between its configuration-dependent TIE part and its hard-coded, configuration-independent part, which represents the HDL-coded part of the processor. These core signals are used for special states, such as the *DBREAK* state. *DBREAK* allows the ISS user to set a watch on various memory locations so that a breakpoint occurs when the watched locations are accessed.

Instruction extensions written in TIE are modeled in the configuration-dependent part of the ISS; much of the fixed ISA is also written in TIE and is therefore modeled in this part of the ISS. Some instructions are partially modeled in TIE for instruction decoding but the semantics are left blank and are modeled via the hard-coded ISS core. Special signals are used to signal execution of such instructions to the ISS. For example, synchronization instructions (ISYNC, DSYNC, and ESYNC) or cache-access operations such as dirty-line replacement tend to be hardwired in the ISS using this communication mechanism because they are very tightly bound to the processor infrastructure. (Consequently, users cannot provide their own cache model.)

The Xtensa processor's configurable nature combined with the dynamic nature of the ISS results in some operation inefficiencies. For example, when an interrupt or memory access occurs, the ISS must dynamically check the settings of various configuration parameters (in this example, the interrupt settings and the cache parameters) to determine which part of the ISS logic to execute. Given a wide range of configuration parameters, the ISS code must check a lot of Boolean and integer variables, resulting in a lot of branching code. A well-written ISS for a fully fixed-ISA processor or a completely regenerated, configuration-specific ISS would not have this overhead. However, we feel we've made a reasonable tradeoff to create a flexible software system that matches and complements the flexibility in the Xtensa processor architecture. Of course, many fixed-ISA processors also have some variability that their ISSes must handle—checking cache parameters for example. In theory we could move more of the overhead into the configuration-specific part of the ISS but this choice would make the ISS harder to maintain and would increase the amount of time taken to build new processor configurations. Although it is hard to estimate the cost of this overhead, one reasonable estimate is that the cycle-accurate mode of the Xtensa ISS runs at about half the speed that it would if it was fully regenerated for each new processor configuration. We believe this represents a good engineering compromise between speed and flexibility.

However, software developers always want more speed so we have recently expanded our ISS strategy by adding a fast, functional, instruction-accurate

simulation capability called TurboXim, which uses just-in-time compiled-code techniques to significantly speed simulation. (The speed improvement varies widely depending on target code and processor configuration but, as a rule of thumb, 20–50X speed improvements or more seem realistic.) Because TurboXim performs just-in-time code generation and compilation for specific target code on a specific processor configuration, it knows exactly what code to generate and avoids a lot of conditional processing. The instruction-accurate (as opposed to cycle-accurate) TurboXim allows software and firmware developers to trade off speed for accuracy. Of course, developers would prefer to have the speed while retaining 100% cycle accuracy, but this is not practical.

TurboXim reports the instructions executed but makes no attempt to predict the actual executed cycles that the target code might use while executing on the target processor. It is possible to add the ability to predict cycle counts using table-lookup functions in a kind of "cycle-approximate" mode and this feature remains a future possibility for this technology. Meanwhile, it is possible to switch between cycle- and instruction-accurate simulations dynamically during a simulation run. This ability enables a kind of hybrid simulation where a sufficient amount of execution in cycle-accurate mode can be used to predict the overall cycle count of the complete simulation. Used in this manner, a hybrid simulation might run 99% in instruction-accurate mode and only 1% in cycle-accurate mode. Of course, some combination of both these methods would be possible.

7.14 System Simulation

The ISS is provided as a standalone tool for single-processor software development and for use within two system-simulation environments: XTMP, which uses a proprietary C-based modeling API, and XTSC, which is based on SystemC. These system-simulation environments allow the easy use of ISS models for one or more configurations of the Xtensa processor and allow the creation of generic models that can be adapted to configuration characteristics with little or no model source-code modification. This is supported by two mechanisms: transaction level modeling (TLM) approaches that provide generic methods for classes of processor interfaces, and introspection, which is supported by a variety of model-query methods that allow all the relevant configuration characteristics of a particular configuration to be determined.

Examples of the TLM interfaces include TIE queue interfaces supported by standard methods that work for all queue sizes, local-memory interfaces with standard access methods that work for all memory widths (these local memory interfaces are very similar to the system PIF memory interfaces), and TIE lookups, which support a common set of access methods for all lookup instances. Examples of the introspection routines include APIs that allow the

simulator to determine configuration parameters, the presence of all configured interfaces, and access methods for the ports provided for these interfaces. When these configuration-specific interface methods and ports are combined with a number of generic devices (such as memory devices, connectors, routers, arbiters), then system-simulation models for multi-core systems can be developed, and also linked to third party models and system simulation tools.

7.15 The IDE (Integrated Development Environment)

Tensilica's integrated development environment (IDE), called Xtensa Xplorer, is based on the increasingly popular, open-source Eclipse development environment. Xtensa Xplorer provides two essential capabilities:

1. Access to the Xtensa Processor Generator (XPG), which allows a developer to define a particular Xtensa processor configuration by selecting architectural parameters and by adding TIE descriptions of instruction extensions (as discussed earlier).

2. A software-development IDE that allows the developer to manage target-software sources and compiled-object files, to target and migrate software to different processor configurations, to manage projects and libraries, to launch software compilations and ISS simulation runs, to debug single-processor and multiprocessor configurations, to launch the XTMP and XTSC multiprocessor simulation models, to profile software execution, to model energy consumption, etc.

The IDE for Xtensa processors must be both very generic and very configuration-aware. It must also have the capacity to allow selective addition of configuration specifics to generic functions as it evolves. By basing our IDE on Eclipse rather than developing an IDE from scratch, we have benefited from the substantial body of work created by the Eclipse open-source community. By doing so, however, we accepted a less-than-ideal foundation for Xtensa Xplorer.

Several years ago, the Xtensa configuration tool, which provided access to the XPG, was completely separate from the IDE. All processor configurations were created using a Web-based tool and, after a specific processor configuration was built, the developer downloaded the freshly generated processor RTL and the relevant software-development tools to a local workstation. The world of processor configuration and software development, which took place on a client workstation, was completely separate from the world of processor generation, which occurred in Tensilica's server farm.

With the move to the Xtensa Xplorer IDE, there was an opportunity to link these two worlds more closely, thus providing a faster development flow. We did this in a limited and strictly controlled way to limit our own develop-

ment time and risk. Thus, the Xtensa Xplorer IDE still maintains the concept of two relatively separate spaces: the configuration space and the classical IDE develop–compile–build–run–debug space for target-software development. The IDE space has only a limited view of the configuration space—just enough for it to obtain the configuration information it needs to be an effective development environment. In hindsight, this approach has been very beneficial. It permits more independence between these parts of the tool chain and allows the parts to evolve independently, with less development effort than if they evolved in lockstep.

Being based on Eclipse, the Xtensa Xplorer environment could incorporate capabilities of the Eclipse C Development Toolset (CDT) to serve the needs of Xtensa developers. The CDT was built as a generic debug environment for an open-source processor and debugger, which actually served as a good architecture for a configurable IDE. The CDT's operating principles are based on an API that queries the specific debugger about relevant processor configuration parameters, such as the register set (size, names, and types). Then the debugger's user interface renders the register set on screen.

The underlying Xtensa Xplorer Eclipse debugger is *xt-gdb*, which we also offer as a standalone debugging tool. This debugger provides configuration-specific information via its text command/response API to the Xtensa Xplorer debugger. Because *xt-gdb* allows the user to set breakpoints, to modify register values, and offers other features while running, there is in fact a continuous two-way flow of information between the Eclipse debugger and *xt-gdb*. If we were dealing with a fixed-ISA processor with known configuration information, then it would be possible to make this communication link a little more efficient. On the other hand, the existing debugger architecture is easier to evolve and the latency involved in these communication operations is relatively low when running on modern workstations, despite the large amount of dialogue between processes that produces reasonably high socket traffic.

CDT's design proved useful for our purposes because it was architected to support many different back ends. CDT assumes that the tools have reasonable introspection so it provides a higher level of control. Thus we feel that CDT achieves a reasonable balance between performance and hardware independence. Other IDEs tend to blend their capabilities together in such a way that it is difficult to see where the debugger starts or ends and where the rest of the IDE starts or ends, which might potentially be nice in a closed world but not very useful in a configurable one.

Many Eclipse-based C IDEs employ *gdb* or a variation of *gdb* as their debugger; other IDEs from other vendors, including Mentor Graphics (EDGE) and Nokia, use proprietary debuggers. Some companies, Wind River for example, avoid the CDT for debugging completely and build directly on the Eclipse debug framework. With command-line-driven tools such as *gdb*, com-

mands might need many seconds to execute and can produce thousands of lines of "user-readable" output, which is not very amenable to IDE integration. Thus the MI layer in *gdb* was created to support UI-based IDEs. Although MI defines many commands, only about 70–80% of them are implemented (a byproduct of being an open source product) and thus there are some debugging capabilities that must use communication approaches apart from the MI layer.

The Xtensa configurator part of the IDE is very heavily hardware-dependent. Here, a complex set of relationships between the parameters and their ranges or settings define the processor's configuration space. For backwards compatibility, the configurator knows about previous architectural generations. (We support more than five Xtensa processor generations with our development tools.) To make the management of the configuration space tractable, we model the options in Xtensa Xplorer rather than in the output range of the configuration hardware or in processor features. For example, the configurator knows that the floating-point (FP) option requires the Boolean-registers option, so we don't model the actual output of the FP option in terms of instructions and operands. This is a way of abstracting the basic configuration set and space into something simpler.

The configurator's model of the configuration space thus becomes a collection of rules and valid parametric selections or values (or a value range). These rules are organized into cascading value chains so that a selection of one parameter propagates into associated rules. These rules are expressed in Java. They could be written in some kind of HLL and compiled into the configurator but it seems reasonable to keep the rules in Java because that is Eclipse's native language. There are some generic rules, such as a value dependence on other values being set, but there are also very specific rules such as dependencies between parameters in specifying interfaces to local instruction or data RAMs.

The most complex rules deal with address specifications and dependencies. For example, our exception handlers require that there is enough space to load an address at the point of a branch. The required amount of space might be 12, 16, or 24 bytes depending on the actual instruction. This rule is expressed using a complex set of rule code. The UI both predicts and constrains what the downstream development tools can work with.

Because the exact instruction generation for an exception handler is not necessarily known when the configuration rules are being written, there is a negotiation process to ensure that the configurator stays in the feasible region of the configuration space. The configurator must also be aware of potential future software upgrades so that the generated configurations are as compatible as possible with future upgrades.

One key point that results from defining a configuration space in the configurator is that the configuration space of possible processors is actually derived

from the characteristics of the XPG build process. This space can become bigger than can reasonably be built into a rules-driven configurator. That is, there may be feasible configurations that are extremely difficult to abstract into a configurator. Although this aspect of configuration may seem like a limitation, it has the beneficial side effect of constraining the range of what can be expressed, thus keeping configurations safe, predictable, and sane. Another way of stating this concept is that the XPG process defines a model or space for possible configurations that is strictly larger than the model or space defined by the IDE configurator. Although this restriction may appear to be limiting, it has not proven to be a problem in practice.

The current configurator rules, readers, writers, and generators require about 50,000–60,000 lines of Java code. This code is organized into a sequence (actually, a tree or network) of abstract models of the configuration space that reflects the architectural evolution of the Xtensa processor. Because we need to understand rules and options from many processor generations, the configuration rule base grows monotonically and becomes more complex over time. It becomes an increasingly deep class hierarchy of rules, values, legal restrictions, and choices. This growth clearly increases the maintenance burden with each new architectural generation, but such is the price of change.

One fundamental choice made in developing the IDE-based configurator, the front end of the development process, was whether to drive it with the same file types, formats, and files as the back end of the process, which generates hardware. We made a conscious decision to keep these front- and back-end files, file types, and file formats separate, which suits some parts of the process better than others. The basic issue is that there are different concerns at different stages of the hardware-generation process that are best expressed with different views and files. This aspect of the overall problem argues for separation rather than trying to have one omnibus master database.

For example, the configurator has a relatively simple view of some options. The software tool chain generates dlls and shared object files that represent the outcome of the configuration process: for example what the registers, new instructions, latencies, and interlocks are and not what they could have been or how they were specified. CAD-flow engineers care about interactions among different components (which are not necessarily instruction blocks). In fact, the configuration space of arbitrary hardware as described with an HDL is far bigger than the TIE configuration space. Consequently, RTL developers must test their resulting designs far more than is required of the relatively restricted design space encompassed by the specification capabilities of the Xtensa processor configurator and TIE. Thus processor generation based on TIE gives a more robust system and a much lower risk of error.

An RTL-centric hardware team requires a much bigger set of rules and tradeoff space that allow them to generate and test combinations and configu-

rations that are impossible to specify via the IDE-based configurator. Having the same rules and constraints across the whole engineering community and tool flow would be unnecessarily limiting and cause additional work (to enforce the rules given the different development histories) and might actually reduce quality by reducing the scope of testing.

Much of the configurator guards against situations that could produce unreasonable results. These rules are enforced by the TIE compiler. The IDE TIE editor has an integrated rules checker. Around 99% of the TIE grammar is defined in the editor, which deals with low-level syntactic checks and options. In addition, the TIE compiler has an internal mode that can generate a model of the processor. The IDE uses this mode, for example, to show dependencies between stages of TIE instructions and state variables and registers and other semantic analysis. This use of the TIE compiler occurs in the TIE editor's background mode. Background syntax checks run continuously for quick feedback but semantic analysis is performed only on demand because it can be quite time consuming.

7.16 Conclusions and Futures

Using a configurable and extensible processor demands a sophisticated set of software development and verification tools. Such tools must support both the process of determining the optimal configuration and programming the resulting configured processor. Allowing an almost infinite range of processor options makes the tool-creation process intractable unless it is highly automated.

In this chapter we have outlined some of the options open to creators of this class of hardware-dependent software, illustrated with practical examples based Tensilica's Xtensa processors. Architectural growth and improvements drove considerable evolution in our tool-development strategy. The options available and choices made should be useful to anyone contemplating the development and support of flexible processor architectures.

References

[Gon00] Ricardo E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[IL06] Paolo Ienne and Rainer Leupers, editors. *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann, San Francisco, 2006.

[Lap01] Philip A. Laplante. *Dictionary of Computer Science, Engineering and Technology*. CRC Press, Boca Raton, 2001.

[Lei06] Steve Leibson. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*. Morgan Kaufmann, San Francisco, 2006.

[MD06] Prabhat Mishra and Nikil Dutt. Processor modeling and design tools. In Luciano Lavagno Louis Scheffer and Grant Martin, editors, *EDA for IC System Design, Verification and Testing*, volume I of *Electronic Design Automation for Integrated Circuits Handbook*. CRC Press/Taylor and Francis, Boca Raton, 2006.

[RL04] Chris Rowen and Steve Leibson. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. Prentice Hall, Upper Saddle River, 2004.

[WKMR01] Albert Wang, Earl Killian, Dror E. Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Design Automation Conference*, pages 184–188. Assoc. Comput. Mach., New York, 2001.

Chapter 8

HIGH-LEVEL DEVELOPMENT, MODELING AND AUTOMATIC GENERATION OF HARDWARE-DEPENDENT SOFTWARE

Gunar Schirner, Rainer Dömer and Andreas Gerstlauer

Abstract With the increasing software content in modern embedded systems, software development clearly dominates the design cost. The development of Hardware-dependent Software (HdS) is especially challenging due to its tight coupling with the underlying hardware. Therefore, automatic generation of all embedded software including the HdS is highly desirable to meet today's shortened time-to-market demands.

In this chapter, we describe a system-level design approach that offers a seamless solution for generating embedded software, starting from an abstract specification and going to an implementation. In our high-level development environment, the application is developed in a platform-agnostic format that hides most implementation detail. The target platform and the mapping of the application to the platform are described separately. A system compiler then automatically generates a system model at the transaction level for performance analysis and development. The same system model later serves as an input to a software generation process, which generates the final binaries for all processors in the system. These binaries include the application, device drivers, and operating system code.

Using a design flow with automatic software generation offers significant productivity gains. At the same time, it allows the designer to focus on the algorithms without being burdened by implementation-level detail.

Keywords: System-level Design, Development Environment, Firmware, Software Generation

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

8.1 Introduction

Software development starts dominating the design cost of modern complex Multi-Processor System-on-Chip (MPSoC). The software content is increasing since it allows to flexibly implement complex features and to quickly react to customer demands. In this context, Hardware-dependent Software (HdS) is especially challenging, due to its tight coupling with the underlying hardware (HW). Traditional approaches of manually implementing HdS become very time consuming. With a large amount of implementation detail, a manual implementation is tedious and error prone. Additionally, validating and debugging software executing on real hardware delays this important process until the availability of the final hardware platform. This hinders a parallel development of hardware and software and may result in missing the tight time-to-market constraints. On the other hand, a validation using low-level instruction set simulation suffers from a slow simulation, especially in a multi-processor context.

To increase productivity, we envision an integrated design flow that eliminates the need for low-level programming. In this chapter, we propose high-level HdS development that hides HW dependencies from designers and allows focusing on algorithms without being burdened by driver-level details.

In our high-level environment, as outlined in Fig. 8.1, the application is developed in a platform-agnostic specification written in a System-Level Design Language (SLDL). The specification model consists of a hierarchical process graph containing sequential C code in each process. In the hierarchy, processes are composed in a parallel-sequential fashion. Communication between processes is captured in abstract communication channels and shared variables, independent of their later implementation.

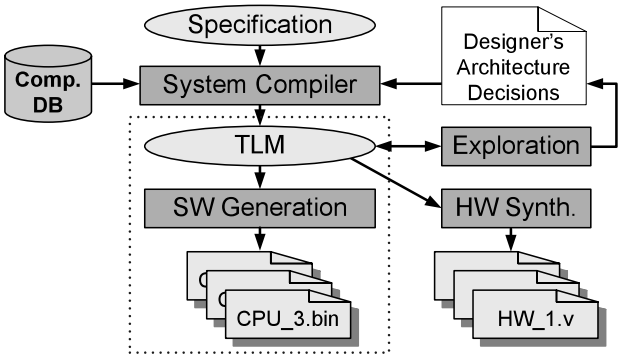

Figure 8.1. System design flow overview.

The targeted hardware platform is specified separately, containing processor and hardware allocation, mapping of processes to processors and hardware blocks, and the definition of the communication topology and its parameters. While mapping the specification to the platform, the designer also specifies important software aspects, such as task mapping, the definition of task priorities, and selection of the scheduling policy for each processor.

Based on application and platform specification, our system compiler automatically maps the application down to a set of processors and busses, creating a set of tasks for each processor, and generating the communication drivers between processes depending on their HW/SW mapping. The application-specific hardware-dependent code is generated by the system compiler. As one output, it generates a system model at selectable abstraction level (with different amount of detail).

The abstract system model is valuable for virtual prototyping, early performance estimation, and validation of the feasibility of the HW/SW mapping. It also enables functional validation of the application over the given platform. Furthermore, it exposes the effects of dynamic scheduling for each processor, allowing optimization of priority mapping and guiding static load balancing. Altogether, the system model is a convenient virtual debugging platform that is usable before HW availability.

Most importantly, the system model serves as an input to the back-end SW generation, which generates and cross-compiles the C code. In particular, it generates the firmware, drivers and interrupt handlers, which implement the external communication of the processor. It also adjusts the application code to execute on top of the selected Real-Time Operating System (RTOS). Finally, the linker creates the final software binary for each processor. For early validation of those binaries, a system model with integrated Instruction Set Simulators (ISSs) can be used.

We informally distinguish between software synthesis and software generation. Both produce an implementation out of an abstract input model by adding implementation level detail. In contrast to generation, synthesis includes in addition an automatic optimization for a given objective or cost function. In our work described in this chapter, we describe a pure generation-based approach that does not include an optimization.

The rest of this chapter is organized as follows. We first discuss the context of software generation and survey current approaches. Then, Sect. 8.2 describes in detail the envisioned HdS development based on a platform-agnostic input and abstract system models. Section 8.3 provides an overview on SW generation and Sect. 8.4 focuses on the generation of HdS. Section 8.5 discusses application examples and demonstrates the approach for six real-life applications. Section 8.6 summarizes and concludes the chapter.

8.1.1 Context and Related Work

Designing a modern complex MPSoC is challenging both in terms of hardware and software. The current manufacturing capabilities offer tremendous integration capabilities and a high degree of implementation freedom. For optimization, a vast exploration space has to be explored and analyzed in the design process. At the same time, the market demands a shorter time-to-market to yield competitive products. Hence, the challenge is to design increasingly complex embedded systems in a shorter period of time.

System-level design is accepted as the main approach to address the complexity challenges. It uses a unified approach to design hardware and software concurrently. System-level design uses higher levels of abstraction to describe a system. Ideally, this allows to describe a system solely as a composition of algorithms, so that the designer can maintain the system overview, while not being burdened by the vast amount of implementation details.

To capture systems jointly with hardware and software, System Level Design Languages (SLDLs) have been developed, such as UML, graphical input, Esterel and C-based languages. In this chapter, we focus on C-based SLDLs. Examples of C-based SLDLs are SystemC [GLMS02], which is widely used in academia and industry, and SpecC [GZD+00]. These languages are based on C++ and ANSI-C, respectively, and have been extended to also capture system and hardware aspects, such as parallelism, pipelining, signals, and bit-vectors to just name a few added concepts.

Abstract models for system-level design are often described as Transaction Level Models (TLMs) [GLMS02], which abstract away the details of pins and wires [CG03]. By omitting implementation-level detail, TLMs execute dramatically faster than bit-accurate models. Therefore, they are widely used for design space exploration and early development.

Today, TLMs are typically written manually [HYL+06] and are moreover rarely used for generation of a complete final implementation. Specialized partial solutions are already very successful, e.g. for generating the interface description between RTL hardware and software (see Chap. 5). To increase productivity, we envision a design flow that spans from an abstract, untimed, and platform-agnostic specification down to an actual implementation on real hardware, as we will describe in this chapter.

Traditionally, SW generation has been addressed from very specific input models and with a limited target architecture support. Some examples are POLIS [BCG+97], DESCARTES [RPZM93], and Cortadella et al. [CKL+00]. The POLIS [BCG+97] approach uses a Co-design Finite State Machine (CFSM) model, where each FSM represents a component in the system. Software generation is performed by transforming the input model into an S-Graph, and subsequent C code generation. This work focuses on reactive systems and

is not designed for general applications. DESCARTES [RPZM93] uses a data flow description (Asynchronous Data Flow (ADF) and an extended Synchronous Data Flow (SDF)) as an input and supports heterogeneous systems. With the specific input choice, these solutions favor a particular application type. In contrast, a flexible generic C-programming model is desirable over these specific input models to cater to the needs of a broader programming audience and to capture a wider range of application domains.

Abstract models, based on SLDLs with a generic C-programming model, have been used for modeling software (SW) and its execution in abstract form [KKW+06, GYNJ01]. Additionally, ISSs have been integrated into abstract system models to create system co-simulation environments [BBB+05, CoWa]. Such, virtual platforms allow for a detailed analysis of the system before availability of real hardware, often revealing details not available on the target [HYL+06]. While these approaches focus on simulation and validation, they do not offer an integrated solution to generate the final implementation.

Some early approaches show solutions to use an abstract model, which contains the common description of HW and SW, as a source for generating the embedded software. Herrara et al. [HPSV03] describe SW generation from a SystemC model. With SystemC being a library extension of C++, they propose to overload SystemC library elements for execution on the target system. This has the advantage of reusing the same model for specification and target execution. However, the approach partly replicates the simulation engine.

Krause et al. [KBR05] generate source code from SystemC and adjust the application to execute on top of an RTOS. To flexibly target different RTOS vendors, they capture the API in an XML format for a customized generation. This approach, however, does not describe in detail the generation of communication and synchronization code and the creation of the final target binary.

Gauthier et al. [GYJ01] describe a method for generating application-specific operating systems and the corresponding application SW. Their work focuses on the OS portion and does not address external HW. Our solution, on the other hand, explicitly includes heterogeneous external HW. Yu et al. [YDG04] show generation of application C code from an SLDL, however without showing the final target binary. Our approach includes generation of communication drivers, multi-task adaptation, and the generation of the final binary image.

The Phantom Serializing Compiler [NG05] translates multi-tasking POSIX C code input into flat C code by grouping blocks to Atomic Execution Blocks and custom scheduling them. This approach is oriented toward a pure SW solution. In contrast, we address SW generation in a system context, specifically taking HdS and external communication into account.

8.2 Software-enabled System Design Flow

Electronic System Level (ESL) design addresses the complexity challenges of designing a modern embedded system. One such flow is outlined in Fig. 8.1 and uses a two step design approach. This ESL flow, implemented in [DGP+08], generates first a system TLM for detailed performance estimation and early MPSoC development. In a second step, the TLM is used as an input to automatically generate SW binaries for the processors in the target platform.

The input to the system design flow is the *specification model*. It describes the algorithms of the system and their dependencies. The specification model is captured in an untimed and platform-agnostic form using a C-based SLDL. For the experiments reported in this chapter, we use the SpecC SLDL [GZD+00]. The concepts shown, however, are equally applicable to other C-based SLDLs, such as SystemC, as well.

Important for a flexible and analyzable input specification is the separation of computation and communication. This separation enables automatic refinement of communication and mapping of computation to separate processing elements. The computation is grouped in behaviors (or modules / processes), and communication is expressed in channels. The upper portion of Fig. 8.2 shows a graphical representation of a simple system specification. The boxes with rounded corners symbolize behaviors. The actual C code inside the behaviors (e.g. *B2* and *B3*) is omitted for brevity.

The behaviors communicate via direct point-to-point channels. For an easier generation, these channels are selected from a feature-rich set of standardized channel types. They allow for a wide range of communication types, such as synchronous and asynchronous communication, blocking and non-blocking communication (e.g. FIFO), as well as for synchronization only (e.g. semaphore, mutex, barrier). Basically, these channels are similar to standard communication primitives offered by middleware or an operating system.

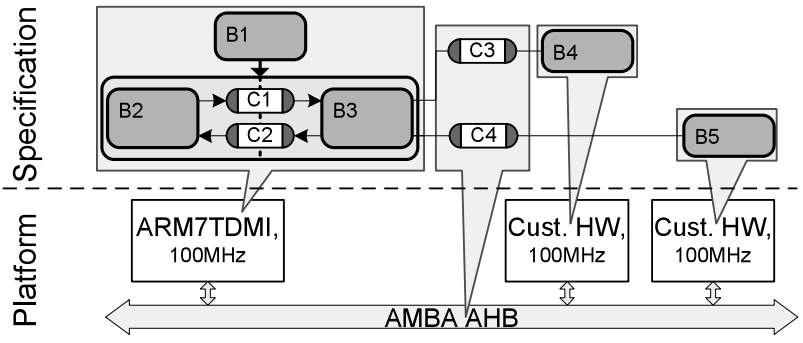

Figure 8.2. Example specification with architecture mapping.

Behaviors can be composed hierarchically to allow complex structures. They can be arranged to execute in any order, such as sequential, parallel, pipelined, or state machine controlled. In the example, behaviors *B2* and *B3* execute in parallel. They communicate through channels *C1* and *C2*. These channels are of type "double handshake", which implies blocking, synchronous communication that is not buffered. The channels *C3* and *C4*, for communication between *B3*, *B4* and *B5*, are finite depth FIFO channels. Using these standard channels allows for a very intuitive programming approach, that is independent of any hardware selection and application distribution.

A second input to our system design flow contains the architecture decisions which describe the platform, as visualized in the bottom portion of Fig. 8.2. The designer enters these decisions using an interactive Graphical User Interface (GUI).

Architecture decisions include the allocation of processing elements (PEs) (e.g. processors, HW components). In the example, an ARM7TMI processor and two custom hardware components are allocated. PE-specific parameters, such as clock frequency, are chosen during allocation. Additionally, the user defines the mapping of behaviors to PEs, deciding which PE will execute the computation inside each behavior. Behaviors, that are assigned to execute on a processor, are wrapped into tasks. The user can then define important task parameters, such as priority and stack size.

Besides dealing with the computation, the designer also controls the allocation and mapping of communication protocols. The example mapping decisions are illustrated in the bottom portion of Fig. 8.2. Here, a bus system of type AMBA AHB [AMBA] is allocated. The call-out boxes symbolize mapping the channels to that bus. For each channel, the user can also define essential communication parameters. For one, the user can select the synchronization scheme, such as polling or interrupt-based synchronization. Additionally, a bus address, that identifies the channel on the communication medium, can be selected.

Based on this these inputs, our system compiler [DGP+08] automatically generates a system TLM that reflects the architecture decisions. For this model refinement, components out of the component data base (compare Fig. 8.1) are instantiated and connected. The communication between processing elements is refined from the standardized abstract channels down to communication based on the selected medium (here the AMBA AHB). The TLM, see example in Fig. 8.4, allows for system exploration, performance analysis and debugging. The TLM simulates significantly faster than a traditional ISS-based model [SGD07].

Once the designer is satisfied with the performance and quality of the system, the same TLM serves then as input for the back-end HW synthesis and SW generation. The SW generation produces the final SW binaries that are

executable on a set of processors composing the platform. It generates the application code, and all drivers for communication in a heterogeneous system. The SW application executes on an off-the-shelf RTOS, or by using an interrupt-driven system for small applications.

8.3 Software Generation Overview

The SW generation, as shown in Fig. 8.3, uses the TLM as an input. As described before, the TLM reflects all architecture decisions. Computation is mapped to processing elements. Computation within each processor is grouped to tasks, all essential task parameters are captured, and the tasks are executed on top of an abstract RTOS (the concepts of RTOS modeling are also described in Chap. 9). The external communication has been refined according to an ISO/OSI layered approach. It is mapped to a set of busses and protocols using bus primitives. External synchronization is implemented (e.g. polling or interrupt) based on the designer's choice. Furthermore, the model contains all structural information to implement the communication decisions. Therefore, the input TLM contains all functional and structural information needed for the target implementation. Please see [DGP+08] for a more detailed description of the TLM generation.

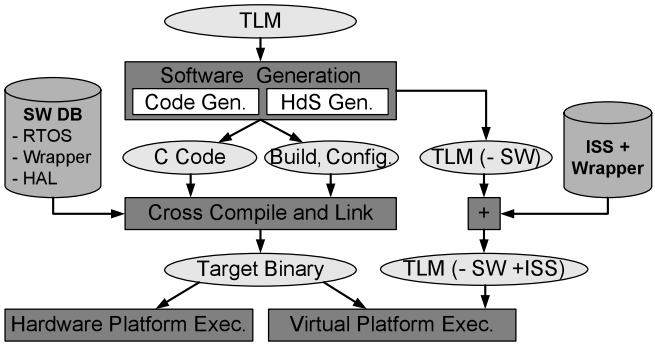

Figure 8.3. Software generation flow [SGD08].

Our *software generation* is divided into C code generation and HdS generation. The C code generation [YDG04], generates flat C code out of the hierarchical model captured in the SpecC SLDL. It converts behavior hierarchies into a set of C functions. Instance-specific variables are translated into a set of data structure instances. Additionally, the channel connectivity between behaviors is resolved into flat C code. In other words, the C code generation solves similar issues as early C++ to C compilers that translated a class hierarchy into flat C code.

The second portion, the HdS generation, generates code for processor internal and external communication, including drivers and synchronization (polling or interrupt). It also generates code to execute multiple tasks on the same processor. To create the complete binary SW image, it finally generates configuration and build files (e.g. Makefile) which select and configure database components. As such, a particular RTOS is chosen, properly adapted/ported to the selected processor. A hardware abstraction layer (HAL) is included based on the target platform, consisting of low-level drivers for the timer, the programmable interrupt controller (PIC), and the bus accesses.

Using a cross compiler, the final target binary (or binaries) is created, which can execute on the target processor(s), or alternatively on a virtual platform. A virtual platform allows validation and development of the final software binaries already before the availability of real hardware. To generate a virtual platform, our SW generation removes the model of the SW running on each processor from the TLM and replaces it with an ISS that is wrapped for integration into the system model. Each ISS instance then executes one SW binary.

8.4 Hardware-dependent Software Generation

The HdS generation uses the system TLM as an input (see example in Fig. 8.4), which was generated by the system compiler based on the designer's architecture decisions. Following the mapping definitions, illustrated in Fig. 8.2, the behaviors *B1*, *B2* and *B3* execute on the processor. The behaviors *B4* and *B5* are each mapped to an own HW accelerator. The TLM contains hierarchical behaviors, channels, and additional HW to properly reflect the platform characteristics. For example, it contains a model of a PIC that maps multiple external interrupts to the available CPU interrupts, and a timer module for periodic interrupts.

The HdS generation parses the input TLM into an abstract syntax tree and then operates on this tree for code generation. For explanation, we distinguish three generation aspects: communication generation, multi-task generation and generation of the final target image. The following sections describe each aspect individually.

8.4.1 Communication Generation

The communication generation deals with processor internal and external communication. In particular, it creates the driver code for communication between the software and external HW. It also generates code for synchronization, for which it inserts stubs into the application code, and generates interrupt handlers and/or polling code.

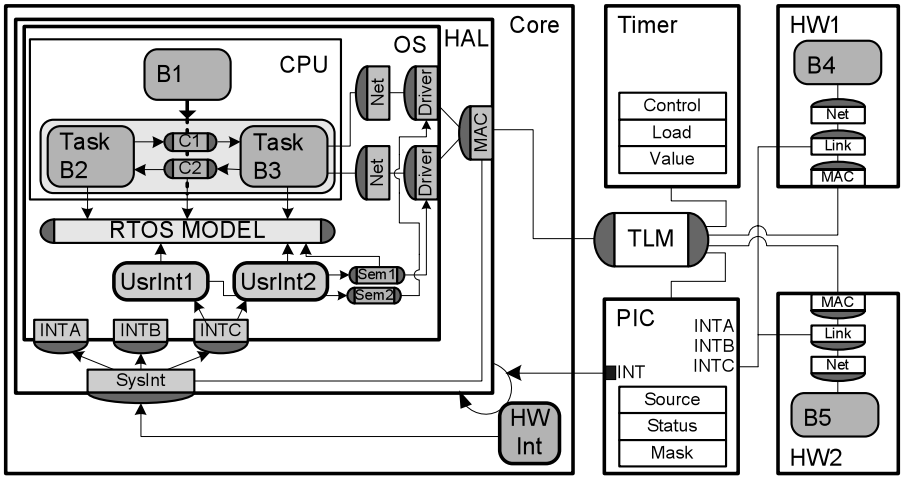

Figure 8.4. Processor and application TLM.

Internal Communication. Internal communication takes place between tasks on the same processor. In the example shown in Fig. 8.4, the channels *C1*, *C2*, *Sem1* and *Sem2* are used for internal communication. These are instances of our standard channels as also used in the specification. To provide the particular communication on the target system, the abstract standard channels are replaced with a target-specific implementation that uses the primitives of an underlying RTOS (or an emulation thereof, in case an RTOS is avoided). Note, that this implementation does not recreate the simulation environment on the target. Instead, a target-specific implementation is used that recreates the same interface and semantics as the abstract channels. For example, a blocking synchronous communication channel is implemented on an RTOS-based system with a semaphore, two events, and a *memcpy* using the services of our RTOS Abstraction Layer (RAL), which we insert for independence of the actual RTOS (for details, please refer to the later section about multi-task generation).

External Communication. To support heterogeneous systems, we follow the ISO/OSI layering model [ISO94] to implement external communication. Examples of external communication are the channels *C3* and *C4* of the initial specification (see Fig. 8.2). According to the mapping information, these channels capture communication between different processing elements (e.g. processor and custom hardware). These channels no longer appear directly in the system TLM in Fig. 8.4. Our system compiler has refined the abstract channels into stacks of half channels (namely *Net*, *Driver*, and *MAC*),

which are inserted into the processor model. A matching stack of half channels is inserted into each HW component (*HW1* and *HW2*) as well.

At the top of the stack, the typed user data is marshalled into a flat untyped data stream. This untyped stream provides a common representation that can be interpreted among different processing elements regardless of bitwidth, endianness and padding rules. This common representation for example allows that a little endian processor can read and interpret the data stream of a big endian processor.

The communication generation has access to the abstract syntax tree representing the application code. Therefore, it can extract the necessary type information from the application code and generate application-specific marshalling code that uses standard conversion functions to create the untyped data stream. For example, the user may define structure *tReq* that contains three elements *startTime*, *coeff1* and *base*, as shown in Listing 8.1.

Based on the information of the channel *Net* (see Fig. 8.4), the communication generation produces marshalling code that serializes the structure data into a flat byte stream as shown in Listing 8.2. Note that, in contrast to using fixed bitwidth types already in the specification, as discussed in Chap. 5 and Chap. 6, our system-level approach contains platform-agnostic types (e.g. plain *int*) in the initial specification model. The marshalling process here therefore is necessary in order to create the platform-specific types.

Data from the input structure (pointer *pD*) is converted into the buffer (pointer *This−>buf*). The marshalling code uses standard conversion functions for each basic data type (e.g. *uhonlong()*). Later in the generation process, a processor-optimized implementation of the marshaling function is selected from the database.

The next half channel, the *Driver*, contains information about the channel's system-wide addressing. It maps the end-to-end channel, which connects two behaviors, to a set of point-to-point links. In a platform with many busses, an end-to-end link may connect processing elements on different busses. Then, multiple point-to-point links create the connection across the busses, which are connected via communication elements (e.g. bridge or transducer). Note that, in comparison to the Chapter 5 and Chapter 6, our system-level approach

```
1 typedef struct stReq {
2   long           startTime;
3   short          coeff1;
4   unsigned short base;
5 } tReq;
```

Listing 8.1. User type definition in the specification model.

```
1  void c_pre_req_CPU_send(/* ...*/ *This, struct tReq *pD){
2    unsigned char *pB = This->buf;
3    htonlong(pB,   pD->startTime);
4    pB += 4;
5    htonshort(pB,  pD->coeff1);
6    pB += 2;
7    htonushort(pB, pD->base);
8    pB += 2;
9    c_link_CPU__CAN_CTRL_DLink_send(/*...*/This->buf, 8);
10 }
```

Listing 8.2. Generated code for marshalling of user data.

generates a custom register addressing here on-the-fly, based on an available system-wide view of the components and their address space.

The slave in our example is connected to the processor bus. Therefore, direct communication is possible and no additional communication elements are necessary. However, complex communication schemes spanning multiple bus hierarchies are possible. Then, user messages are packetized to minimize buffer requirements of intermediate communication partners. Depending on the information in the *Driver* channel, the corresponding source code is generated.

The driver also implements a channel-specific synchronization mechanism, which will be explained in the next section. Finally, the *Driver* transfers the data using the Media Access Control (MAC) layer, which implements the low-level access to the communication media. This layer provides services to transport an arbitrary sized contiguous block of bytes to an address in the system. According to the platform definition, the HdS generation selects later a processor-specific MAC implementation. In a simple case of a processor's primary bus, the MAC may use the processor's memory interface.

Synchronization. For a typical master/slave bus, external synchronization is required for a slave to indicate it being ready for a data transfer (e.g. required data being available). The designer choses the type of synchronization for each channel, selecting between polling or interrupt-based synchronization. Furthermore, the designer may choose to share interrupts between sources to reduce the overall number of interrupt pins. These choices are reflected in the generated system TLM.

If *polling* was chosen, polling code is generated as part of the driver code. An example is outlined in Listing 8.3. The CPU accesses the slave's polling flag to check whether the slave is ready for the communication. This access is performed using the MAC services analogous to the external communication (see the call to function *Ahb_masterMemRead()* in Line 5). If the slave is not

```
1  void c_link_CPU__HW_DLink_send(/* ... */ *This,
2    const void *pData, int len) {
3      unsigned char flag;
4      do { /* poll slave if ready */
5           Ahb_masterMemRead(/* ... */,
6           HW1_DLink_0_FLAG_ADDR, &flag, sizeof(flag));
7           if(flag) { /* break if ready */
8                break;
9           }
10          /* delay for poll. period */
11          TaskDelay(HW1_DLink_0_POLL_DELAY);
12     } while(1);
13     /* successfully synch'ed, transfer data now */
14     Ahb_masterMemWrite(/* ... */,
15          HW1_DLink_0_DATA_ADDR, pData, len);
16 }
```

Listing 8.3. Polling synchronization example.

ready, the polling code uses RTOS services to delay execution for the polling period (see function call *TaskDelay()* in Line 11), and repeats polling. Once determined that the slave is ready, the polling loop terminates (Line 8) and transfers the data (Line 14).

In case of *interrupt* synchronization, the TLM contains a model of the interrupt chain. In Fig. 8.4, for example, the chain consists of the *PIC*, the system interrupt handler *SysInt*, the application-specific interrupt handler *INTC*, the user interrupt handler *UsrInt1* and *UsrInt2*. Finally, semaphore channels (*Sem1*, *Sem2*) connect each interrupt handler with the driver code, so that the (short) interrupt handler can start the (long) driver to handle the communication. To implement interrupt-based synchronization, our HdS generation produces a chain of correlated code. The next paragraphs describe the interrupt-based synchronization code, following the event sequence when sending a message from *B5*, which is mapped to a hardware component, to *B2*, which is mapped to the processor. The event sequence is illustrated in Fig. 8.5.

At t_0, the behavior *B2* expects a message. With the message not being available, *B2* waits on the semaphore *Sem1* and yields execution to the next lower priority task *B3*. At t_1, behavior *B5*, that is mapped to *HW2*, reaches the code to send the expected message and signals via interrupt *INTC* the availability of the message to the processor core. On the way, the PIC sets the processor interrupt *Int*. This in turn triggers the interrupt chain on the processor, which we have labeled 1 through 4.

1. The low-level assembly interrupt handler preempts the currently running task *B3*. It stores the current context on the stack and then calls the sys-

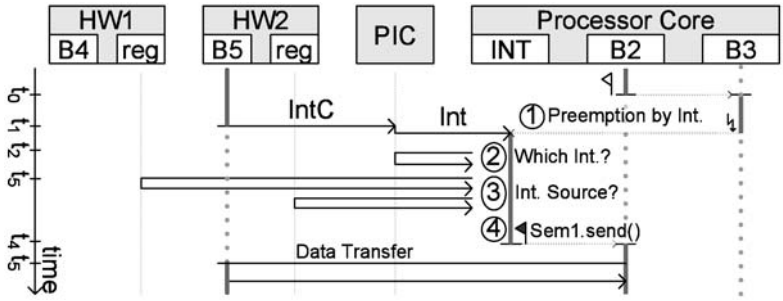

Figure 8.5. Events in external communication.

tem interrupt handler. The low-level assembly interrupt handler, which is part of the RTOS port is inserted from the software database.

2. The system interrupt handler (see half channel *SysInt* in Fig. 8.4) communicates with the PIC. It determines through memory mapped I/O the highest priority pending interrupt. It then invokes the application-specific interrupt handler (see half channel *INTC* in the TLM in Fig. 8.4). The *SysInt* code is one element of the Hardware Abstraction Layer (HAL) stored in the database.

3. Since the interrupt in this example is shared between *HW1* and *HW2*, the actual source of the interrupt is determined next. The application-specific interrupt handler *INTC* determines the source of the interrupt by reading the status registers in *HW1* and *HW2*. Subsequently, *INTC* then calls the corresponding User Interrupt Handler (in this case *UsrInt2* of Fig. 8.4).

4. Finally, *UsrInt2* calls the semaphore *Sem1* to release the driver code that executes in the behavior *B2*. The semaphore channel uses the earlier described internal communication services.

After releasing semaphore *Sem1*, the interrupt handler terminates. Subsequently, the task for *B2* becomes ready and is scheduled. Finally, after the context switch, *B2* reads the data from *HW2*.

For HdS generation, we implement this chain on the processor. The code falls into two distinct portions. The first part is application-independent, and therefore can be stored in the software database. The second portion is application-specific and has to be generated out of the system TLM. The code for steps 1 and 2 belongs to the first portion that is application-independent, and their code is taken from the database. The code for steps 3 and 4, on the other hand, is application-specific, and is generated (step 3 based on *INTC*, and step 4 based on *UsrInt2*).

```
1  void ARM7TDMI_INTC_body(/* ... */ *This) {
2      unsigned char flag;
3      Ahb_masterMemRead(/* ... */,
4          HW1_DLink_0_FLAG_ADDR, &flag, sizeof(flag));
5      if(flag) {
6          c_os_semaphore_release(/* ... */ This->sem1);
7      }
8      Ahb_masterMemRead(/* ... */,
9          HW2_DLink_1_FLAG_ADDR, &flag, sizeof(flag));
10     if(flag) {
11         c_os_semaphore_release(/* ... */ This->sem2);
12     }
13 }
14
15 void ARM7TDMI_OS_CPU_main(/* ... */ *This){
16     /* ... */
17     c_os_semaphore__init(/* ... */ This->sem1);
18     c_os_semaphore__init(/* ... */ This->sem2);
19     BSP_UserIrqRegister(INTNR_int1handler,
20         ARM7TDMI_INTC_body, /* ... */);
21     /* ... */
22 }
```

Listing 8.4. Interrupt handler outline for shared interrupt.

Listing 8.4 outlines the generated code for an application specific interrupt handler (as described for step 3) that is shared between two interrupt sources. The handler sequentially checks the interrupt sources using the MAC communication services (e.g. Line 3). Once the handler finds the interrupt initiating hardware, it releases the associated user task that executes the driver code (see call to *c_os_semaphore_release()* in Line 6).

In addition, startup code is necessary to setup the interrupt chain on the processor side. For one, the application-specific interrupt handler needs to be registered to the system interrupt handler, so that it executes upon receiving of the associated interrupt. In this example, our HdS generator produces startup code that registers application-specific interrupt handler *INTC* to the system interrupt handler for execution upon receiving *INTC* on the *PIC* (see Listing 8.4, Line 19). To gather the necessary information, it traverses the connectivity and architectural information stored in the TLM. It also generates code to instantiate the semaphore channel and inserts appropriate calls into the driver code.

8.4.2 Multi-Task Generation

When multiple tasks are mapped to the same processor, they have to be dynamically scheduled to alternate their execution. Our *multi-task genera-*

tion produces code that uses an underlying multi-task engine in order to control tasks and schedule them. We support two different approaches for multi-tasking. First, we mainly focus on a traditional execution on top of an off-the-shelf RTOS. Furthermore, we provide an alternative of interrupt-based multi-tasking that can execute on a "naked" processor without any operating system.

RTOS-based Multi-Tasking Our main focus rests on targeting an off-the-shelf RTOS. This ensures using a reliable, well-tested operating system that offers great flexibility and often comes with significant tool support from the RTOS vendor. Operating systems are available in a wide range and focus. Often, they are highly configurable to tailor the OS to the application needs. By configuration, the memory footprint can be minimized to fit the needs of the embedded system under design.

Our *multi-task generation* makes use of a canonical OS interface, which we call the RTOS Abstraction Layer (RAL), see Fig. 8.6 (left). The very thin RAL (few hundred lines of (mostly inlined) code), abstracts from the particular OS's function names and parameters. We have chosen the RAL approach to limit the interdependency between our generation and the actual target RTOS. To ensure a generic API, we investigated different RTOS APIs (uCOS-II, vxWorks, eCos, ITRON, POSIX) and chose common primitives for task scheduling, communication and synchronization.

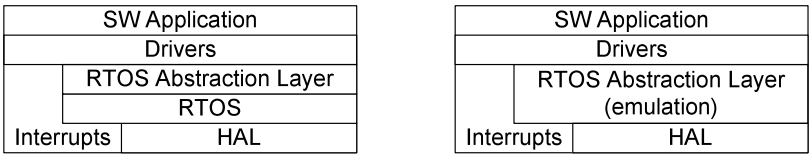

Figure 8.6. Software stack RTOS-based (*left*), interrupt-based (*right*).

Although the investigated RTOS APIs provided all necessary interfaces, this may not be the case for other RTOS APIs. In such cases, the RAL implements an emulation of the required functions that is constructed out of the available primitives. This approach guarantees that always an identical API, the RAL, is available to the generated SW generation, regardless of the particular RTOS implementation.

The input TLM contains mapping of behaviors to tasks (*Task B2*, *Task B3*) and their scheduling parameters. For RTOS-based multi-tasking, our HdS generation extracts the task control information from the TLM and generates task creation calls to the RAL. It also initializes the task's parameter set of the TLM (e.g. priority, stack size) on the target. From SLDL statements, which describe parallel execution of behaviors, our HdS generation produces code that calls

the RAL for task creation and release, and furthermore inserts code to join the multiple threads of execution after their completion.

To give an example, Listing 8.5 shows a partial specification following the system definition already shown in Fig. 8.2. It instantiates the three behaviors; *B1*, *B2* and *B3*. It executes first *B1* (Line 8) followed by a parallel execution of *B2* and *B3* (Lines 9 through 12).

Listing 8.6 outlines the generated C-code. The sequentially executing *B1* is directly called in the parent's main function (see call *TB1_main()* in Line 5). The parallel executing behaviors *B2* and *B3* are spawned using the RAL API function *TaskCreate()* (see Line 6 and Line 7). Note that *TaskCreate()* both creates a task and releases it for immediate execution. After spawning the tasks, the parent task waits until the created tasks have terminated (Lines 9 and 10).

In addition to the task control, processor internal communication is translated to RTOS-based communication. For that, the standardized communication channels (as described for the input) are implemented on top of the RAL. Our *multi-task generation* instantiates the target implementation and connects the channels according to the TLM connectivity information.

Interrupt-based Multi-Tasking In the second case, targeting a "naked" processor, concurrent software execution is performed without any RTOS. Instead, interrupts are utilized to provide multiple flows of execution. We support this alternative for systems where RTOS execution is not desirable. This may be the case, when the system consist of only very few tasks, the code is targeted to execute on a DSP, or when strict memory footprint limitations rule out utilizing an RTOS. We describe a motivating example for an interrupt-based

```
1  behavior B0(/*...*/) {
2      /* ... */
3      TB1  B1(/*...*/); /* instantiate behavior B1 */
4      TB2  B2(/*...*/); /* instantiate behavior B2 */
5      TB3  B3(/*...*/); /* instantiate behavior B3 */
6
7      void main(void) {
8          B1.main();
9          par {
10             B2.main();
11             B3.main();
12         }
13     }
14 };
```

Listing 8.5. Specification of behaviors.

```
1  void TB0_main(/* ... */){
2      os_task_handle B2_thdl;
3      os_task_handle B3_thdl;
4      /* ... */
5      TB1_main(/* ... */);
6      B2_thdl = TaskCreate(TB2_main, /* ... */);
7      B3_thdl = TaskCreate(TB3_main, /* ... */);
8
9      TaskJoin(B2_thdl);
10     TaskJoin(B3_thdl);
11 }
```

Listing 8.6. Generated RTOS-based multi-tasking code outline.

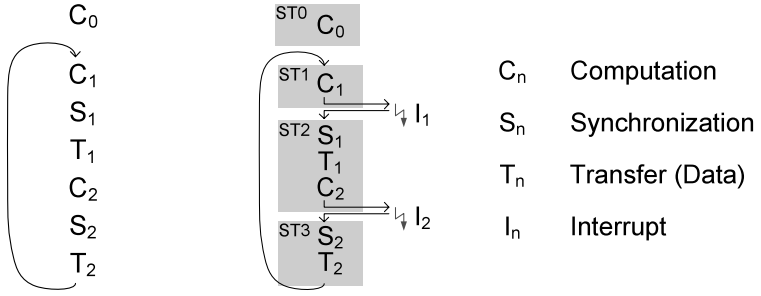

Figure 8.7. Reactive task template input (*left*) and output (*right*).

solution in Sect. 8.5. This case implements a GSM speech codec on a DSP with only two reactive tasks.

For our interrupt-based multi-tasking alternative, the RAL (see Fig. 8.6 (right)) implements a (very thin) RTOS emulation. It provides a subset of the RTOS services needed for software execution (e.g. events, processor suspension, and interrupt registration). To give an intuitive explanation, the *multi-task generation* converts the lowest priority task to execute in the processor main function, and all other tasks are converted to execute in a state machine fashion, in the context of their interrupt handlers.

More formally, we assume that each task is composed of a sequence of computation (C), synchronization (S), and data transfers (T). Figure 8.7 (left) shows an example sequence for one task. As described before, the driver code for communicating with external hardware contains both synchronization and communication. If only interrupts are used for synchronization, then the task main function can be transformed into a state machine, as shown in Fig. 8.7 (right).

In the state machine, each synchronization point starts a new state. For example, state *ST2* was created due to synchronization point S_1, and *ST3* due

```
1  void intHandler_I1() {
2      release(S1);       /* set S1 ready */
3      executeTask0();    /* task state machine */
4  }
5  void executeTask0() {
6    do {
7      switch(State) {
8                  /* ... */
9              case ST1: C1(...);
10                 State = ST2;
11             case ST2: if(attempt(S1)) {
12                   T1_receive(...);
13                 } else {
14                   break;
15                 }
16                 C2(...);
17                 State = ST3;
18             case ST3: /* ... */
19                 State = ST1;
20     }
21   } while (State == ST1);
22 }
```

Listing 8.7. Interrupt-based multi-tasking excerpt.

to S_2. The state machine transitions to the next state upon successful synchronization. For example, upon receiving of interrupt I_1, the state machine would transition from *ST1* to *ST2*. Additional states are inserted to implement conditional execution and loops. For example, the separation between the states *ST0* and *ST1* has been introduced to accommodate the one-time execution of the initialization code in C_0.

The created task's state machine is then executed in the interrupt handlers, which were initially chosen for synchronization of that task (in this example, the handlers of I_1 and I_2). In order to preserve the task priorities, the interrupts have to be chosen accordingly. A higher priority task has to exclusively use higher priority interrupts than a lower priority task. Consequently, the lowest priority task executes in the main task (T_{main}), the startup task of the processor.

Each local variable of a task's main function is integrated into a global data structure. Hence, the task execution no longer relies on an own stack, and may be executed in separate calls to the task's state machine.

Listing 8.7 outlines the generated C implementation. Please assume for explanation that the task's state machine is currently executing in the interrupt handler for I_1, *ST1* is the current state, and that computation C_1 has just finished. Next, the synchronization S_1 is checked (line 11). In case the synchronization has not yet occurred, the state machine terminates (line 14). Conse-

quently, the do-while loop, the function *executeTask0*, as well as the interrupt handler, all terminate. Thus, the processor can then serve a lower priority interrupt, or the main function.

Upon receiving the next interrupt I_1, the system interrupt handler calls the registered user interrupt handler *intHandler_I1* (see line 1). In line 2, the handler signals that S_1 is ready and then calls the state machine again (line 3). The current state is *ST2*, therefore the condition in line 11 is tested again. It now passes, since the synchronization has occurred, receives the data (line 12), and subsequently executes the computation C_2 in line 16.

The switch-case statement (lines 7 to 20) is surrounded by a do-while-loop, which is required to implement loops between states. In this example, the loop is necessary to transition from state *ST3* back to *ST1* without terminating the interrupt handler.

8.4.3 Binary Image Generation

The final aspect of HdS generation is the generation of a complete target binary. Our generation uses a cross-compiler tool chain (gcc) that is specific to the target processor and binary format. It generates configuration and makefiles for the binary image creation, which select components from the software database, configure these components, and in addition control the compilation and linking of generated code. This process is illustrated in Fig. 8.8.

An important aspect for establishing a flexible generation flow, with a wide variety of configurations with many processor and hardware combinations, is an effective design of the database. It is essential to identify the dependencies

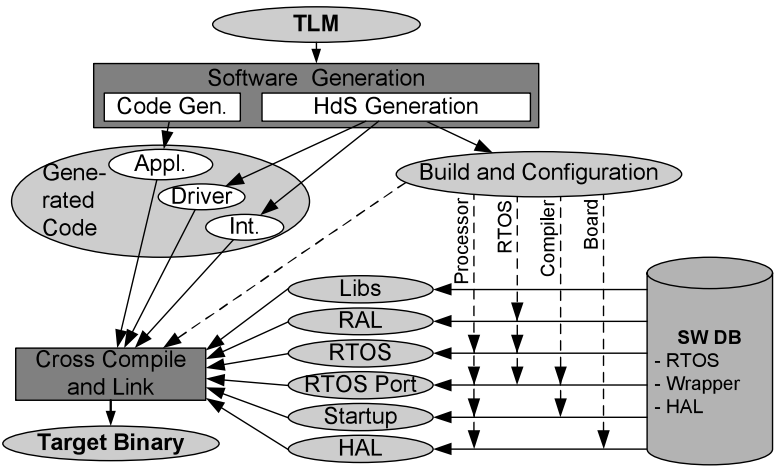

Figure 8.8. Generation of target binary.

of each database component with respect to the selected hardware/software configuration, e.g. the selected processor, RTOS, cross compiler, and board components. Capturing all dependencies is necessary for correctly selecting a component. On the other hand, overly specializing a component would lead to code duplication within the database, and yield a code bloat.

The matrix of arrows in Fig. 8.8 symbolizes the dependencies when selecting a component. Usually the most specific element is the RTOS port, since it depends on the RTOS type, the processor, and the cross-compiler (for example, for the call frame layout and the stack layout needed for the task creation). Our software generation also produces a customized Makefile, which selects the components according to the architecture information in the TLM, and then uses the cross-compiler to generate the target binary. Automating this step has the advantage, that the TLM serves as the sole input to the binary generation, avoids duplication of the system configuration (i.e. in the Makefile), and further minimizes the user effort.

8.5 Experimental Results

In this section, we describe some practical applications of your approach. We have applied it to a set of real-life examples. Two examples are covered in more detail. The first is a telecommunication example, the second uses an application from the automotive domain. Following that, we describe our generation results for several applications to more quantitatively compare the results.

8.5.1 Interrupt-based Implementation Example

We start by showing a specific example of an interrupt-based multi-tasking implementation. We implemented a GSM 06.60 [ETSI96] encoder and decoder on a Motorola DSP 56600 platform. As shown in Fig. 8.9, the DSP is assisted by a HW accelerator and four HW blocks that deal with input and output. The HW accelerator is dedicated to the computation-intensive codebook search of the encoding process.

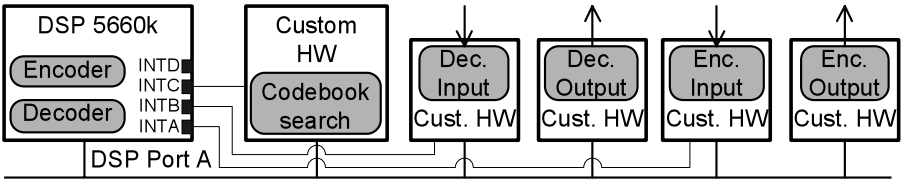

Figure 8.9. Media example of GSM transcoding.

In our application, the DSP only executes two reactive tasks (encoding and decoding). Also, an RTOS port for this particular DSP was not easily available. Therefore, we applied our interrupt-based multi-tasking approach to this example. Following a shortest-job-first scheduling policy, the longer executing encoder is assigned the lower priority of the two tasks. Hence, the encoder will execute in T_{main}. The higher priority (shorter) decoder task is transformed into a state machine. According to the architecture decisions, the decoder uses *IntB* for synchronization. Hence, the generated decoder's state machine will execute in the interrupt handler of *IntB*.

Figure 8.10 shows the state machine for the decoder task, which consists of 4 states. The states *ST1* and *ST2* have been created due to synchronization (S_1, S_2). The interrupt *IntB* is used for both synchronization points. A GSM speech frame consists of four sub-frames. Accordingly, *ST2* is repeated four times. The states *ST0* and *ST3*, respectively, are inserted to accommodate initialization, which executes only at the beginning, and post processing, which executes once per frame.

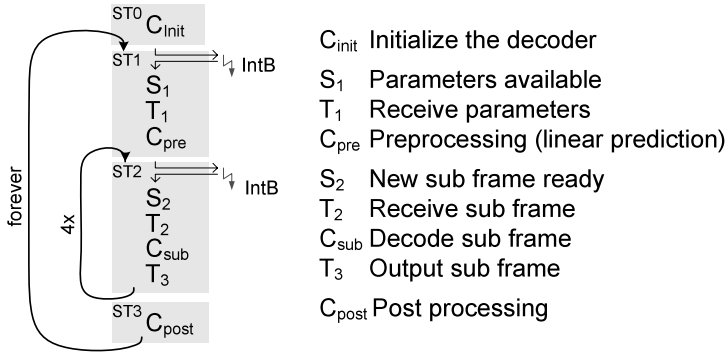

Figure 8.10. State machine for GSM decoder.

The input data is read by T_1 and T_2, which receive the initial parameters and the compressed sub-frame data, respectively. The decoded speech samples are transferred by T_3 without any additional synchronization into the output HW block. This particular transfer is performed without a preceding synchronization, since the receiving I/O HW is always ready.

Figure 8.11 shows the time line for transcoding one sub-frame after the initialization has already passed. The processor is suspended at the start of the time-line and waits for input data. At t_1, *IntA* signals availability of input data, and the registered interrupt handler is executed. The handler triggers event *e1* which the main task, T_{main} is waiting on. Hence, after termination of the interrupt handler T_{main} is resumed. After some processing, the encoder feeds

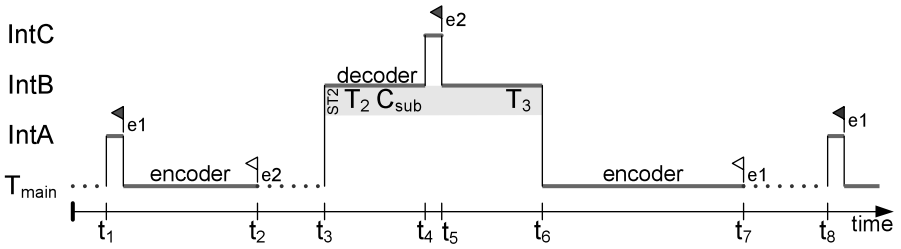

Figure 8.11. GSM transcoding execution.

the codebook accelerator. The encoder then suspends on event *e2* waiting for results from the accelerator. Again, the processor is suspended.

Later at t_3, *IntB* signals the availability of sub-frame data for decoding. The decoder state machine, which currently is in state *ST2*, is executed in the *IntB* handler. It reads the input data (T_2), decodes the sub-frame (C_{sub}), and transfers in T_3 the decoded speech samples to the output HW. Again, the latter needs no synchronization, since the output HW in the architecture is always available. At t_4, while decoding (in C_{sub}), the decoder is preempted by the higher priority *IntC*, which announces that the codebook search has finished. Subsequently, the interrupt handler releases the event *e2*. After the decoder interrupt handler has finished, the encoder resumes at t_6 and finishes at t_7. The same cycle repeats at t_8 with the next sub-frame. Throughout the execution of our testbench, 3451 interrupts are triggered. More results are later available in Table 8.2.

8.5.2 Exploration Example

We use an automotive example to illustrate the exploration capabilities with respect to comparing the two multi-tasking approaches. We model an Electronic Control Unit (ECU) containing an ARM7TDMI processor [ARM7]. The processor executes three tasks; anti-lock break control, RPM computation, and engine fan controller. Six sensors and actuators are connected to the ECU via two CAN busses (Fig. 8.12). Three further sensors are integrated in the ECU and are attached directly to the processor bus.

We have generated code for both approaches, first toward execution on top of the RTOS μCOS-II [Lab02], and second for interrupt-based execution. μCOS-II is a small, highly configurable RTOS that is mostly implemented in ANSI C. Ports of this RTOS are available for a wide range of processors, which dramatically simplified the integration.

Table 8.1 compares the generated RTOS-based and interrupt-based multi-tasking implementations. For the latter case, we mapped the lowest priority task, the fan control, to T_{main}, while the other two tasks were converted to state machines for execution in interrupt handlers.

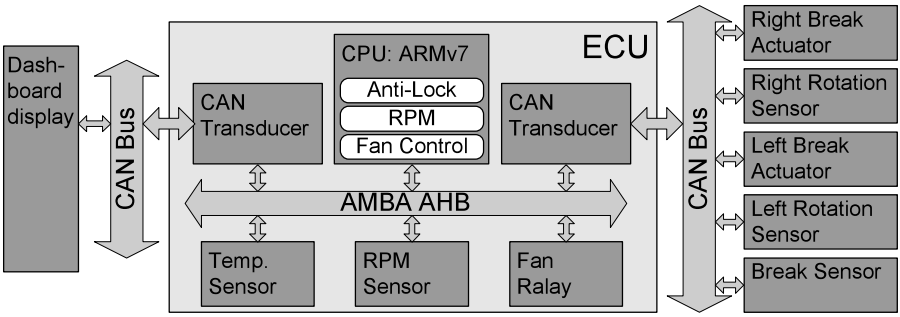

Figure 8.12. Automotive example application.

Multi-tasking	RTOS-based	Interrupt-based
Footprint	36224 Bytes	21052 Bytes
Alloc. Stacks	4096 Bytes	1024 Bytes
CPU Busy Cycles	6.706 MCycles	5.106 MCycles
# Interrupts	1478	1027

Table 8.1. Automotive example results.

As the results in Table 8.1 show, the automotive example profits from the interrupt-based solution. Avoiding the RTOS code yields a smaller memory footprint, since a simpler, more specific code is used instead. The footprint reflects the size of the ROM-able image and includes data, text and BSS segment. Neither solution uses dynamic memory allocation.

The interrupt-based multi-tasking results also in a smaller stack size, since all tasks share the same stack. Additionally, the interrupt-based solution shows a lower CPU consumption. The CPU busy cycles drop from 6.7 MCycles to 5.1 MCycles. This drop is due to the simpler implementation. The RTOS startup is avoided and fewer cycles are needed for the OS functionality (e.g. for event handling and context switching) due to simplicity.

To give an inside view of the system's performance, we analyze the interrupt latency. For the purpose of our measurements, we focus on the delay from the RPM sensor triggering the interrupt wire (to the PIC) to the first bus transaction appearing on the bus to read the RPM sensor.

In the interrupt-based approach, the latency until reading the RPM sensor is shorter (1001 cycles instead of 1794 cycles). This significant reduction is due to the execution in the interrupt handler itself. To compare, in the RTOS-based solution, the sensor is read in the task context, which results in an additional event communication and a context switch.

Also, we counted the number of occurring interrupts, which drops from 1478 to 1027. The interrupt-based solution does not use the timer for keeping

the system time, which explains the lower number of interrupts. On the other hand, the number of interrupts for data synchronization remains constant in both solutions.

Our automotive example clearly shows the benefits of the interrupt-based execution. We position it, where applicable, as an effective alternative in special cases (very few tasks, strict optimization requirements, or unavailability of an RTOS). Since either implementation can be generated automatically, a comparative exploration becomes easily possible.

8.5.3 Generation Results

To show the benefits of an automatic HdS generation, we have applied our HdS generation to a range of six target applications. The first two applications are the already described GSM transcoder and the car ECU. In addition, we examined a JPEG encoder, an MP3 decoder implemented in software, an MP3 decoder with 3 hardware accelerators, and a combined system with MP3 decoding and JPEG encoding.

Table 8.2 summarizes our generation results. The top section quantifies each target applications' complexity. It ranges from the simple JPEG with 2 I/O blocks to the combined application *Mp3 HW + JPEG*, which uses 6 I/O blocks, 3 HW accelerators, and 4 busses.

Example	GSM	Car	JPEG	Mp3 SW	Mp3 HW	Mp3 HW + JPEG
Complexity						
IO/ HW/ Bus	4/ 1/ 1	9/ 2/ 3	2/ 0 /1	2/ 0/ 1	2/ 3/ 4	6/ 3/ 4
SW Behaviors	112	10	34	55	54	90
Channels	18	23	11	10	26	47
Tasks/ ISRs	2/ 3	3/ 5	1/ 2	1/ 3	1/ 8	3/ 14
Lines of Code, RTOS-based						
Application	–	153	818	13914	12548	13480
HdS	–	649	210	299	763	1186
Lines of Code, Interrupt-based						
Application	5921	210	797	13558	12218	–
HdS	377	575	187	256	660	–
Execution, RTOS-based						
CPU Cycles	–	6.7 M	127.7 M	185.8 M	44.5 M	174.6 M
CPU Load	–	0.9%	100.0%	100.0%	30.9%	86.6%
Interrupts	–	1478	805	4195	1144	1914
Execution, Interrupt-based						
CPU Cycles	42.0 M	5.1 M	126.7 M	182.3 M	43.3 M	–
CPU Load	42.5%	0.7%	100.0%	100.0%	30.5%	–
Interrupts	3451	1027	726	4078	1054	–

Table 8.2. SW generation and execution results.

Next, the table shows the number of generated lines of code for application and HdS, each for the RTOS-based and the interrupt-based multi-tasking. As described earlier, we have not implemented the GSM in an RTOS-based solution, since we had no RTOS port available for the DSP. Also, we have not realized the *Mp3 HW + JPEG* example in the interrupt-based form, since it uses services we do not intend to replicate with interrupts. In the examples with HW acceleration, the HdS code is larger due to the extra effort in communication. Overall, a significant amount of code is generated (e.g. 1186 lines for *Mp3 HW + JPEG*).

Automatically generating the software binaries yields a significant gain in productivity. In all examples, our HdS generation completes in less than a second. On the other hand, manually writing the HdS would take days. Thus, the code generation in our approach has a significant impact on reducing the overall design time of embedded systems with HdS context.

To validate the correctness of the generated code, we executed each synthesized target binary on a virtual platform. For that, we integrated a Motorola proprietary instruction set simulator (ISS) for the DSP, and the SWARM ISS [Dal00] for the ARM7TMDI.

Each application executes functionally correct, yielding an output matching the specification. Table 8.2 shows the execution statistics of the ISS cosimulation. As in the car example, fewer CPU cycles (busy cycles only) are consumed in the interrupt-based solution. However, with an increasing computation complexity, the relative improvement becomes marginal. Similar to before, avoiding the OS timer tick reduces the number of processed interrupts.

8.6 Conclusions

Embedded software generation is an essential aspect of implementing todays SoC. It avoids the tedious and error prone manual implementation. In this chapter, we have presented a systematic approach for generating the final target binaries from an abstract specification model. We have shown software generation as an integral part of an ESL flow. Beginning from an abstract model containing the application specification, our flow automatically generates a system TLM based on the designer's architecture decisions. From the generated TLM, the software generation then automatically generates the binaries for each processor in the system. Together, this completes the ESL flow for the software, offering a seamless solution from an abstract system model down to an implementation on embedded processors.

The presented HdS generation addresses three parts: communication generation, multi-task generation, and binary image generation. It generates communication drivers, interrupt handlers, and adjusts for the target multi-tasking. Our approach supports targeting toward an existing RTOS. Furthermore, it of-

fers an alternative to use interrupts for multi-tasking if an RTOS-based execution is undesirable.

We have demonstrated automatic generation using six real-life target applications: different media applications and a control system. The ESL flow with integrated software generation addresses a wide range of target processors, platforms and applications.

Automating the tedious and error-prone process of manual firmware development results in significant gains in productivity. Not only is the automatic generation much faster than a manual implementation, it also allows the designer to focus on the essential algorithms, without the burden of implementation details. Further, with the automatic generation, alternative solutions can be quickly and easily generated. This allows for a rapid exploration of the embedded software design space, e.g. when investigating alternative mapping solutions.

Acknowledgments

The authors thank the SCE research team at the Center for Embedded Computer Systems at UC Irvine for their technical support. The authors also thank the editors and reviewers of this book for their valuable feedback in improving this chapter.

References

[AMBA] Advanced RISC Machines Ltd (ARM). AMBA Specification (Rev. 2.0), ARM IHI 0011A.

[ARM7] Advanced RISC Machines Ltd. (ARM). ARM7TDMI (Rev. 4) Technical Reference Manual, 2001.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Oliver. MPARM: exploring the multi-processor SoC design space with SystemC, *VLSI Signal Process.*, 41:169–182, 2005.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic, Dordrecht, 1997.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, October 2003.

[CKL+00] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Marc Massot, Sandra Moral, Claudio Passerone, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference (DAC)*, Los Angeles, CA, June 2000.

[CoWa] CoWare. Virtual Platform Designer. www.coware.com.

[Dal00] Michael Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, November 2000. www.cl.cam.ac.uk/~mwd24/phd/swarm.html.

[DGP+08] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel Gajski. System-on-Chip Environment: A SpecC-based framework for heterogeneous MPSoC design. *EURASIP J. Embed. Syst.*, 2008.

[ETSI96] European Telecommunication Standards Institute (ETSI). *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, 1996. GSM 06.60.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic, Dordrecht, 2002.

[GYJ01] Lovic Gauthier, Sungjoo Yo, and Ahmed A. Jerraya. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 20(11), November 2001.

[GYNJ01] Patrice Gerin, Sungjoo Yoo, Gabriela Nicolescu, and Ahmed A. Jerraya. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2001.

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic, Dordrecht, 2000.

[HPSV03] F. Herrera, H. Posadas, P. Sánchez, and E. Villar. Systematic embedded software generation from SystemC. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[HYL+06] Sungpack Hong, Sungjoo Yoo, Sheayun Lee, Sangwoo Lee, Hye-Jeong Nam, Bum-Seok Yoo, Jaehyung Hwang, Donghyun Song, Janghwan Kim, Jeongeun Kim, HoonSang Jin, Kyu-Myung Choi, Jeong-Taek Kong, and SooKwan Eo. Creation and utilization of

a virtual platform for embedded software optimization: an industrial case study. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Seoul, South Korea, October 2006.

[ISO94] International Organization for Standardization (ISO). *Reference Model of Open System Interconnection (OSI)*, second edition, 1994. ISO/IEC 7498 Standard.

[KBR05] Matthias Krause, Oliver Bringmann, and Wolfgang Rosenstiel. Target Software generation: An approach for automatic mapping of SystemC specifications onto real-time operating systems. *Des. Autom. Embed. Syst.*, 10(4):229–251, 2005.

[KKW+06] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2006.

[Lab02] Jean J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, Gilroy, 2002.

[NG05] Andre Nacul and Tony Givargis. Lightweight multitasking support for embedded systems using the phantom serializing compiler. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2005.

[RPZM93] Sebastian Ritz, Matthias Pankert, Vojin Zivojnvic, and Heinrich Meyr. High-level software synthesis for the design of communication systems. *IEEE J. Select. Areas Commun.*, April 1993.

[SGD07] Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer. Multi-faceted modeling of embedded processors for system level design, Abstract. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2007.

[SGD08] Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer. Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seoul, Korea, January 2008.

[YDG04] Haobo Yu, Rainer Dömer, and Daniel Gajski. Embedded software generation from system level design languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Yokohama, Japan, January 2004.

Chapter 9

ACCURATE RTOS MODELING AND ANALYSIS WITH SYSTEMC

Henning Zabel, Wolfgang Müller and Andreas Gerstlauer

Abstract Today, mobile and embedded real-time systems have to cope with the migration and allocation of multiple software tasks running on top of a real-time operating system (RTOS) residing on one or several system processors. Each RTOS has to be configured towards the individual needs of the application and environment. For this, different scheduling strategies and task priorities have to be evaluated in order to keep execution and response times for a given task set. Abstract RTOS simulation is applied to analyze different parameters in early design phases. This chapter presents a SystemC RTOS library for abstract yet accurate RTOS simulation, supporting modeling of preemption in the presence of prioritized and nested interrupts. After introducing basic principles of abstract RTOS simulation, we present our SystemC library in detail. Thereafter, we discuss related approaches and close with applications in electronic automotive systems design and some evaluations.

Keywords: RTOS Modelling, RTOS Simulation, SystemC, Task Scheduling, Interrupt Analysis

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

9.1 Introduction

Real-Time Operating Systems (RTOS) are required to provide a predictable platform for the execution of multiple programs, i.e., software tasks, on single microprocessors. As such, operating system and task scheduling effects largely determine the overall software execution behavior, timing and quality metrics, such as performance or power. Each RTOS has to be accurately configured for the individual needs of the application and communication network. An RTOS can be configured, for instance, by different task scheduling algorithms and parameter settings like preemption, execution times, execution deadlines, and periods.

In addition to scheduling and multi-tasking effects, however, the early consideration of software and hardware exceptions, i.e., interrupts, are of utmost importance. Exceptions have significant impact on the predictability of deadlines in the execution of tasks and their response times. Interrupt requests (IRQs) issued by hardware events, such as timers and I/O operations, may trigger the immediate execution of an Interrupt Service Routine (ISR). ISRs can be nested and may have different priorities. They may either immediately execute simple instructions or activate a high priority software task in the case of more complex operations. Hence, interrupts and interrupt handling in the OS and software can have a large, non-negligible influence on and contribution to overall task execution behavior on the processor.

9.1.1 RTOS Analysis and Simulation

Today, RTOS timing analysis is mainly performed by Worst Case Execution Time (WCET) and Worst Case Response Time (WCRT) analysis [PB00, ESHR08], or by Instruction Set Simulation (ISS) [MAF91, NBS+02]. Additionally, logic analyzers, tracing hardware, and specialized trace boxes may come into application.

WCET and WCRT usually employ a static, formal timing analysis based on a graph representation extracted from source or machine code. They determine safe and theoretical worst-case upper bounds for execution and response of software tasks for a specific processor and operating system. Estimation of caching, pipelining, and branch prediction effects is still the subject of ongoing research but a few existing tools, like aiT (AbsInt), already aim to cover some of those areas. In general, microprocessors with more complex pipelines and caching are often avoided for real-time applications so that WCET and WCRT analysis can give better estimations. However, the unpredictable occurrence of hardware and software exceptions remains a problem for worst-case considerations.

Traditionally, ISS is mainly used for functional and performance analysis based on a specific target processor and operating system. Instruction set sim-

ulators and debuggers usually come with the integrated development environment (IDE) of a specific microprocessor or -controller. ISS can be cycle-based or timing-based. Timing results can be used for general feedback and analysis or for back annotation into a cycle-accurate system simulation. ISS mainly performs the emulation of assembly code at the machine code level covering the execution of the complete application plus operating system by accurately representing and simulating values of single variables and all registers. As such, ISS requires the complete software executable including the real RTOS to be available in binary form.

Though modern instruction set simulators already reach considerable speeds, simulation times are typically still insufficient for early estimation and rapid prototyping, especially for multi-processor simulation. Moreover, ISS is hardly applicable for early design phases since it requires the complete target binary to be already available. It is also limited to a specific processor and operating system, imposing too many constraints on later design steps. On the other hand, a purely functional, native simulation is fast but may not be sufficiently accurate due to its mismatch in concurrency models. A functionally correct implementation is useless when it cannot guarantee execution within the deadlines imposed by a specific application.

9.1.2 Simulation Speed Trade-offs

In order to efficiently explore the design space, designers need models of the embedded software running in its execution environment, providing rapid and early feedback about effects of design decisions, such as the chosen scheduling strategy. As outlined previously, traditional ISS is increasingly becoming infeasible due to low speeds (especially in a multi-processor context) and the need to have the final target binary readily available. Therefore, alternative models at higher levels of abstraction with fast simulation speeds yet enough accuracy are needed.

There have been several approaches aiming to develop high-level models of embedded software and its runtime environment including the RTOS itself. Depending on the level of detail, typical trade-offs between speed and accuracy of models can be observed. For example, in [SGD07] a model of software processors is proposed that is gradually constructed out of successive layers of features and functionality (Fig. 9.1). At the highest level, the *Application* is running natively on the simulation host. At the *Task* level, an abstract RTOS model is introduced, and processes and communication channels are refined into tasks and Inter Process Communication (IPC) primitives running on top of the RTOS model. In the processor models at the *Firmware (FW)* and *Transaction-Level Modeling (TLM)* refinement steps, hardware abstraction (HAL) and processor hardware layers are introduced. As such, FW

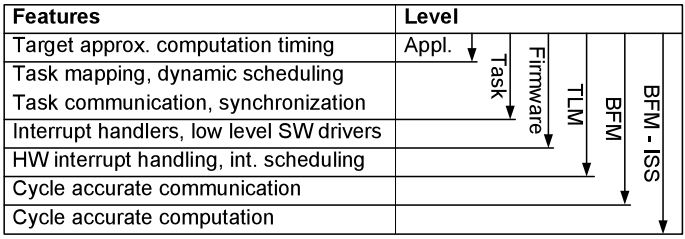

Figure 9.1. Processor modeling layers [SGD07].

and TLM levels add models of the external bus communication (drivers and bus interfaces) and the interrupt handling chain (interrupt handlers and interrupt logic including processor suspension) on the software and hardware side, respectively. Finally, a *Bus-Functional Model (BFM)* of the processor includes pin- and cycle-accurate models of the bus interfaces and the protocol state machines driving and sampling the external wires.

Figure 9.2 shows results for comparing processor models at different feature levels against a traditional, cycle-accurate ISS. The graphs show simulation speeds and model accuracy for an example of a Motorola DSP running voice encoding and decoding tasks as part of a mobile phone baseband application. Tasks are executed on the DSP on top of an RTOS kernel. Accuracy is measured as the average error in frame encoding and decoding delays.

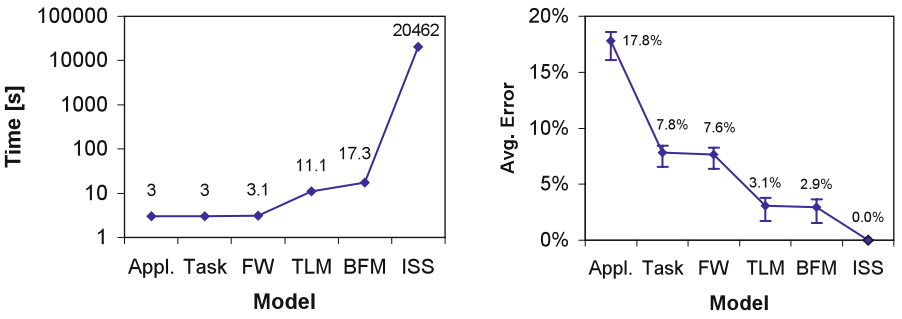

Figure 9.2. Speed and accuracy at different levels [SGD07].

These results confirm that significant speedups can be obtained with high-level processor models. The application level is equivalent to a native execution of the task C code on the simulation host. As such, the full performance of the simulation machine is available. However, due to the mismatch in concurrency models, accuracy is low. Introducing a high-level RTOS model at the task level accurately reflects dynamic scheduling effects, significantly reducing the model error. As expected, OS and scheduling parameters typically

have a large influence on system metrics, such as timing or performance. On the other hand, RTOS modeling adds little to no overhead and the task model still executes at native speeds. Interestingly, another increase in accuracy is achieved by introducing accurate models of the processor's hardware interrupt logic. This confirms that interrupt handling including processor suspension and interrupt priorities can contribute significantly to overall software delays.

In the end, a processor model at the TLM level can achieve significant speedups with an error of less than a few percent compared to traditional cycle-accurate ISS models. Note that speed results are for sustained simulation performance in a single-processor system as presented here. In specific cases and for peak performance, speedups can be even higher and simulation speeds of more than 2 GHz can be achieved. As such, high-level processor models can form a viable alternative to traditional ISS-based software validation. As pointed out above, however, a crucial component of any processor model is an efficient and effective model of the RTOS including both dynamic scheduling and interrupt handling effects.

9.1.3 RTOS Modeling

As previous studies have shown, fast simulation speeds at the task level are reachable by means of an abstract RTOS model that simulates dynamic scheduling effects. High-level processor and RTOS models are typically described and implemented by means of a C-based system-level design language (SLDL) like SpecC or SystemC, where all models are compiled and running natively on the simulation host. At the application level (Fig. 9.3, left), processes and tasks are executed directly on top of the underlying SLDL kernel. Again, simulations run at native speeds yet concurrency models do not match reality. On the other hand, in traditional ISS models (Fig. 9.3, right), the actual target binary consisting of cross-compiled application linked against the ported RTOS kernel and libraries is running in an instruction set simulator, which in turn potentially sits on top of the SLDL kernel for co-simulation with the rest of the system. ISS models can reach full cycle accuracy at the expense of slow simulation speeds.

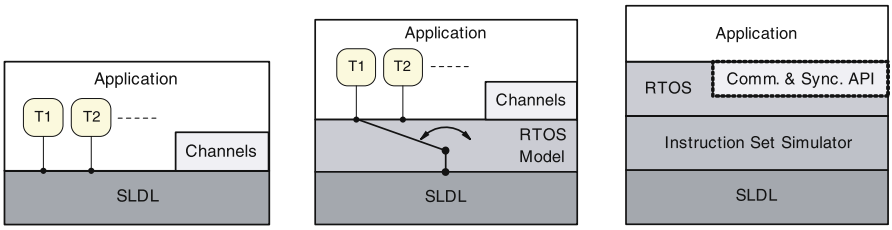

Figure 9.3. Application models (*left*), task models (*middle*) and ISS models (*right*) [GYG03].

The goal of high-level RTOS modeling is to provide the speed of native application execution with the accuracy of an ISS model. Instead of running the real operating system with all its associated overhead, an RTOS model abstracts away unnecessary implementation details and only focuses on modeling the key aspects. At the task level, the RTOS model is inserted as an additional layer that sits between application and underlying SLDL kernel (Fig. 9.3, middle). It replaces the event handling and wraps around corresponding SLDL primitives in order to ensure that at any given time only one task is active at the SLDL interface. Internally, the RTOS model blocks all but the active task, and it selects and dispatches tasks based on a model of the desired scheduling algorithm. In the process, all necessary RTOS concepts such as multi-tasking, dynamic scheduling, interrupt handling, preemption, inter-process communication (IPC) and task synchronization are modeled. As such, an RTOS model provides an abstract implementation of an RTOS API, e.g., POSIX or μ-Itron. When an abstract RTOS model aims to cover task switching and scheduling of many existing RTOS standards or pseudo-standards using a single, fixed API, we denote it as a canonical RTOS model [GYG03].

9.1.4 Abstract RTOS Simulation

Abstract RTOS simulation at the task level is typically based on partitioning of the application into hardware components and software tasks, including Interrupt Service Routines (ISR). Tasks and ISRs are then further divided into a sequence of time-annotated software segments. Timing information can be back-annotated into each segment from performance measurements or timing estimations obtained through ISS or WCET analysis, respectively. In order to accurately capture data-dependent delays, segments are usually defined at the basic block level, though it is possible to partition into more coarse-grain segments.

Figure 9.4 shows an accurate ISS sequence with atomic blocks, i.e., single instructions, compared against a simulation sequence of two time annotated

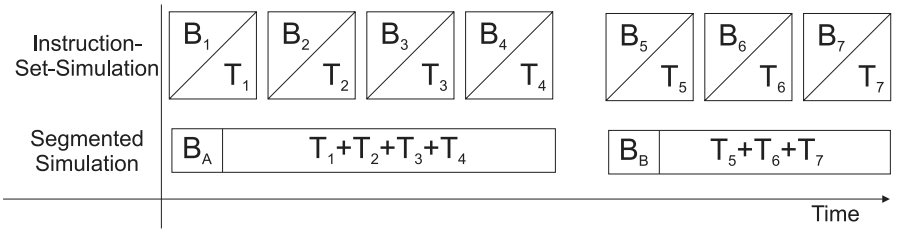

Figure 9.4. Instruction set simulation with time annotations vs. task level simulation with time annotations.

segments, i.e., compositions of multiple atomic blocks. Here, the ISS executes atomic blocks B_1 to B_7 annotated with their execution times T_1 to T_7. As on the final hardware platform, ISRs can immediately execute after each atomic block. At the task level, on the other hand, software blocks B_A and B_B each combine atomic blocks B_1–B_4 with time annotations $T_1 + T_2 + T_3 + T_4$ and B_5–B_7 with time annotations $T_5 + T_6 + T_7$, respectively. In a C-based SLDL simulation at the task level, occurrence of an interrupt after T_1, for example, will delay execution of the corresponding ISR until the end of $T_1+T_2+T_3+T_4$, i.e., the ISR is executed and analyzed with a delay of $T_2 + T_3 + T_4$.

Thus, abstracting the time $T_1 + T_2 + T_3 + T_4$ into one atomic segment can lead to significant timing errors in simulation, in particular when multiple interrupts occur within this interval. In addition, in case of multiple interrupts within a segment, the sequence of the occurrences has to be considered for a functionally correct simulation. As scheduling decisions for ISRs are based on priorities, the sequence information is potentially lost when scheduling an ISR after $T_1 + T_2 + T_3 + T_4$. In general, high-level abstract simulations of segmented time at the task level can result in timing and functional inaccuracy, depending on the chosen segment granularity. However, inaccuracies can be avoided when the time interval is interruptible during simulation. If the language semantics support preemption of time wait statements, the occurrence of an IRQ allows a time interval to be divided into two parts where the second part can be considered after the ISR. This allows scheduling decisions to be made right in time, avoiding the previously mentioned inaccuracies due to a delayed execution.

In the context of C-based SLDL simulation, each segment is typically composed of a section of regular application code followed by a statement that models the execution time of the code block and synchronizes with the RTOS model and the simulation kernel. The execution order of individual segments of different tasks is then determined by the scheduler in the RTOS model based on scheduling strategies such as Earliest Deadline First (EDF), Rate Monotonic (RMS), or Round-Robin (RR) scheduling. Again, with the decomposition of the software into segments, the trade-off between simulation speed and accuracy depending on the granularity of segments has to be considered. With an increasing number of segments, the number of interactions/synchronizations with the RTOS scheduler and simulation kernel increases, resulting in a larger number of simulated events and hence longer simulation times. On the other hand, with a decreasing number of segments, segment sizes increase, which in turn may result in inaccurate simulation results, especially in the case of frequent interrupt occurrences.

Speed and accuracy trade-offs motivate the need for an accurate management of time annotated segments in the RTOS model, as presented in the remainder of this chapter. Inaccurate simulation in C-based SLDLs results from

their non-preemptive simulation kernels and is addressed by the implementation of our abstract SystemC RTOS model in the next section. Thereafter, we discuss related work before we sketch an application using an automotive system design example and close with some evaluation results.

9.2 SystemC RTOS Model

As background information, we first give an overview of the SystemC simulation principles before outlining basic concepts of our RTOS library followed by technical details.

9.2.1 SystemC Simulation

SystemC is a C++ library with language extensions defined as macros and special classes. It implements an event-driven simulation kernel, which coordinates the execution of SystemC processes, i.e., *SC_THREADs* and *SC_METHODs*. Processes communicate via channels and shared objects. Synchronization between processes is implemented via explicit and implicit events generated by the execution of explicit and implicit *wait* statements. When executing a wait statement, the calling process is suspended. After all processes are suspended, the simulation kernel starts processing events collected during process execution. The simulation kernel sets processes ready-to-run when they have received at least one event they are sensitive to during the current simulation cycle. All processes which are ready-to-run are finally invoked in arbitrary sequential order. If there is no process available, the simulation time is advanced to the next available time event. The simulation of time is modeled by wait statements, which generate an event for the respective process in a future point in time. One sequence of process execution and event processing in the simulation kernel is denoted as a simulation cycle. Any simulation cycle which does not advance the simulation time is denoted as a delta-cycle.

It can easily be seen that, comparable to an RTOS, SystemC implements a pseudo-parallel execution of sequential processes with cooperative multitasking where suspension and invocation are controlled by the simulation kernel. However, switching between two processes is explicitly defined by means of wait statements. In multi-tasking operating systems, switching between two tasks and ISRs is denoted as a context switch. We therefore have to distinguish context switches of RTOS tasks and context switches of SystemC threads. In contrast to SystemC simulation, a context switch in a preemptive multi-tasking OS may occur after each instruction of a task and is not limited to explicit context switches as in SystemC or in cooperative multitasking operating systems.

For RTOS task level simulation in SystemC, we model each task and ISR by a *SC_THREAD*. The sequential part of each task and ISR is divided into timed segments, where the time annotation of a segment is defined by the

CONSUME_CPU_TIME function. As each SystemC thread has its own, independent execution time line, we have to explicitly map task and ISR segments into a sequential execution order with respect to a single time line associated with a single thread of control inside a processor. The RTOS model has to implement the mapping of task context switches to SystemC context switches and the actual RTOS task scheduling for sequentialization of software segments. In our implementation, this is realized by a so-called RTOS context. The RTOS context synchronizes the execution of segments through additional internal SystemC events: an event that invokes the scheduler for selection of the next runnable task (*schedule_event*), an event for triggering the execution of one segment of a specific task (*wakeup_event*), and a preemption event to handle interrupts. These events trigger SystemC context switches to model the actual task context switches. The RTOS context in combination with an adequate partitioning of tasks and ISRs into timed segments enables a functionally correct and accurate timed simulation of tasks and ISRs on top of SystemC without the need for any modifications of the SystemC kernel.

9.2.2 Basic Principles

Our SystemC abstract RTOS library (*aRTOS*) implements an abstract canonical RTOS model that is based on SystemC V2.1 [IEEE06] and provides basic functions for task and ISR synchronization, context switching, and scheduling (Fig. 9.5).

RTOS Context. The RTOS API is defined and implemented by the class *sc_rtos_context*. Each instance of the RTOS context represents a separate execution unit or processor core. It provides functions to register/deregister tasks and ISRs as well as managing their synchronization and performing RTOS context switches. Each RTOS context holds pointers to the task scheduler and the ISR scheduler, which select the next runnable task or ISR for sequential execution. The two schedulers are coordinated by a main *schedule* function of the RTOS context, which is sensitive to the *schedule_event*. The separation into two schedulers is necessary since scheduling strategies for tasks and interrupts may differ. Unlike software tasks, which are scheduled by the RTOS, ISRs are preemptively scheduled by the processor following the priorities of individual IRQs. Tasks and ISRs are implemented as *SC_THREADs*, which are spawned by the *SC_RTOS_CTOR* of an RTOS module. An RTOS module extends the native *SC_MODULE*. It is defined as an instance of the *sc_rtos_module* and declared by the *SC_RTOS_MODULE* macro.

Task Control Block. Registration of a task or ISR is defined by means of the *task_start* or *isr_start* function. They create an individual task control block (TCB) *sc_rtos_context_tcb* in the RTOS context at each registration. The regis-

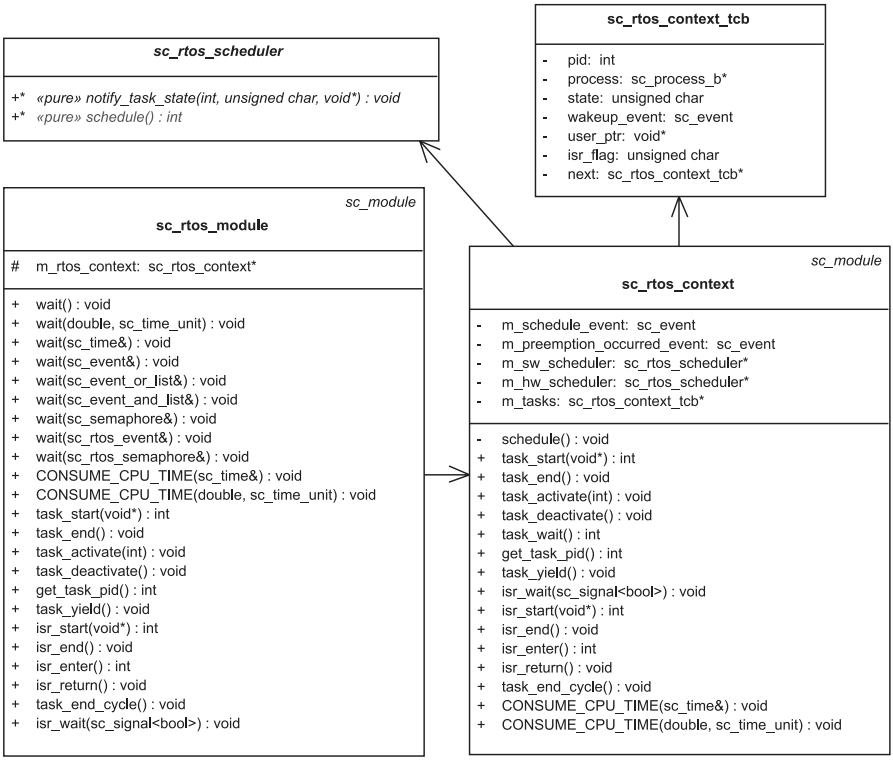

Figure 9.5. UML class diagram of the SystemC RTOS library.

tration function accepts an optional pointer to additional task specific scheduling information like deadlines and sampling rates. Correspondingly, *task_end* and *isr_end* deregister a task or ISR by deleting its TCB from the context so that it can no longer be accessed. The TCB basically holds individual task and ISR information such as respective scheduling information, an unique *pid* as a pointer to the corresponding SystemC thread and its current *state* $\in$ {*created*, *waiting*, *ready*, *running*, *dead*}.

RTOS State Model. Tasks and ISRs follow a state model that is shown in Fig. 9.6. State transitions are always forwarded to the scheduler that processes the information in its implementation.

After registration, a task or ISR becomes *created*. Tasks further switch to *ready* whereas ISRs become *waiting* in order to wait for an IRQ. *ready* tasks and ISRs are considered by the scheduler as ready for execution, which finally sets them to *running*. Herewith, the scheduler has to guarantee that only one task or ISR becomes *running* at one time. At a preemption or in the case of a

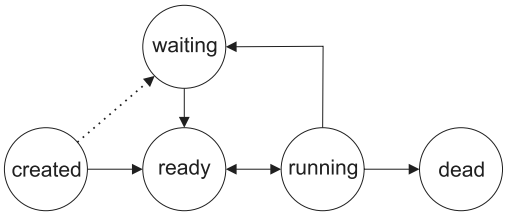

Figure 9.6. States of a task.

CPU release, a *running* task is reassigned to *ready*. When waiting for a SystemC event —implemented through *wakeup_event* synchronization— or in the case of an explicit deactivation, the state is set to *waiting*. Tasks and ISRs in state *waiting* are waiting for internal synchronization and are not considered by the scheduler. State transitions to *ready* can be indirectly triggered by other tasks or ISRs. Alternatively, they can be directly triggered by the receipt of a specific event, like an IRQ. After termination, a task/ISR becomes *dead* and the corresponding TCB is removed by executing a final wrap-up so that the corresponding SystemC thread can no longer be identified as a task or ISR.

Segments. For RTOS simulation, the code of each RTOS task and ISR is divided into sequential segments that are annotated with an interruptible specification of their execution time by means of a *CONSUME_CPU_TIME*() statement. *CONSUME_CPU_TIME*() basically implements an interruptible statement to advance simulation time. It synchronizes with the RTOS context by means of an explicit and dedicated *wakeup_event* inside a *task_yield* call. The RTOS context in turn notifies the individual task for invocation through this event (see Fig. 9.7). Additionally, the *wakeup_event* is used in *task_deactivate* for explicit synchronization as well as in the overloaded RTOS module wait statement in *task_activate*.

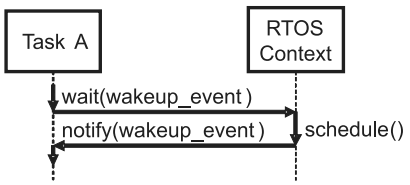

Figure 9.7. Task-context synchronization.

The following example gives a basic idea of a task definition in the RTOS constructor of an RTOS module:

```
SC_RTOS_MODULE(Task)
{
public:
    SC_RTOS_CTOR(Task)
    {
        SC_THREAD(run);
    }
    // my methods ..
};
. . .
void run ()
{
    task_start (<ptr>);
    // infinite loop
    while (true)
    {
        // Execute native C or SystemC code
        my_code();

        // Specify execution time
        CONSUME_CPU_TIME (4,SC_MS);
        ....
        // Execute overloaded wait
        wait (<event>);
        ....
    }
    task_end ();
}
```

The actual software task is defined as the *SC_THREAD run* that is registered and deregistered via *task_start* and *task_end*. The task behavior is implemented by an infinite loop with native C or SystemC code (*mycode*()), time specifications, and optional overloaded wait statements for inter-task communication. ISRs are defined similarly and further outlined at the end of this section.

9.2.3 Time Specification and Scheduler Synchronization

Recall that the body of each task and ISR is specified by a sequence of software code segments followed by an interruptible timing specification. Furthermore, RTOS modules issue overloaded *wait* statements for inter-task communication. To synchronize with the RTOS context, the implementation of the overloaded *wait* wraps the native SystemC *wait* enclosed in calls to context methods as follows:

```
void sc_rtos_module::wait (sc_event &e)
{
    int pid = m_rtos_context->task_wait ();
```

```
4       sc_module::wait (e);
5       m_rtos_context->task_activate (pid);
6   }
```

In this implementation, the overloaded *wait* first notifies the context that the task or ISR changes to *waiting*, where *task_wait* triggers a rescheduling for the next delta-cycle. After the return of the function, the actual context switch is executed in the SystemC kernel through the SystemC *wait* statement. After the invocation, *task_activate* notifies the context about the receipt of the event and that the task or ISR is ready for execution. *task_activate* reschedules the task to become *ready* after which it waits for *wait_event*, i.e., *task_activate* blocks the task or ISR until being notified by the scheduler.

Figure 9.8 shows a simple example with two tasks and their interaction with the RTOS context. Task A executes an overloaded wait statement. This statement first sets the task to *waiting* in the RTOS context. It then executes a native wait statement, which gives control to the RTOS context. The schedule function may thereafter set Task B to *ready* and hence start its execution. Meanwhile, the SystemC thread of Task A is invoked in one of the next delta-cycles and sets itself to *ready* by executing *task_activate*. After the return of Task B, the scheduler recognizes that Task A is ready and subsequently invokes it again.

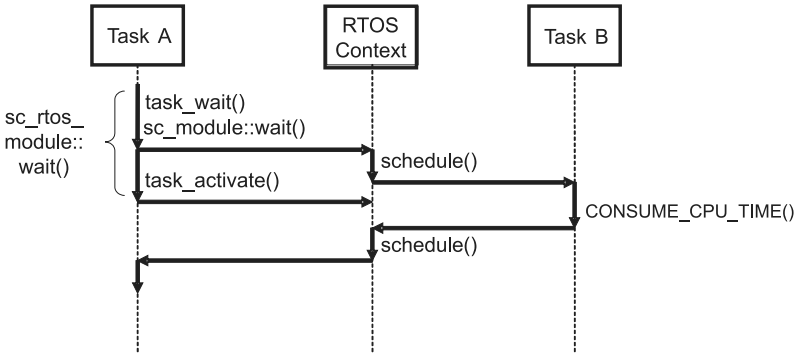

Figure 9.8. Overloaded wait statement.

The interruptible time specification via *CONSUME_CPU_TIME* is, furthermore, defined by the following code:

```
1  void sc_rtos_context::CONSUME_CPU_TIME (sc_time &d)
2  {
3       sc_event continue_event;
4       sc_time  wait_time (d);
5       sc_time  wait_start;
6
7       // time to wait ?
```

```
8       while (wait_time > SC_ZERO_TIME)
9       {
10          // notice current time
11          // and start timer
12          wait_start = sc_time_stamp ();
13          continue_event.notify (wait_time);
14
15          // wait on timer or reschedule
16          wait (continue_event | m_preemption_
17          occurred_event);
18          continue_event.cancel ();
19
20          // compute real waiting time
21          sc_time now = sc_time_stamp ();
22          wait_time -= now - wait_start;
23
24          // release processor and suspend until
25          wait_event
26          task_yield ();
27      }
28  }
```

The function realizes an interruptible time wait by first notifying an internal *continue_event* at a specific future point in time. Thereafter, it suspends itself until the time point is reached or an external event (*m_preemption_occurred_event*) occurs that triggers a rescheduling in case of a preemption, e.g., due to a high priority task or an interrupt activation. This event is always sent one simulation cycle before the execution of the scheduler. On each reactivation upon the receipt of this event, the *wait* is aborted and the time that is elapsed since its start is deducted from the total waiting time. The control is finally returned to the RTOS context through *task_yield()*.

9.2.4 Scheduling Tasks and ISRs

The ISR scheduler and the task scheduler are both derived from class *sc_rtos_scheduler*. They each overwrite and reimplement the *schedule* function of the abstract base class *sc_rtos_scheduler*. This function implements the individual RTOS and ISR scheduling algorithms and returns the identifier of the next selected task or ISR. The scheduler is invoked by the main scheduler function *schedule()* of the RTOS context class whenever a scheduling event is triggered. Each scheduler can create and manage its own state transition table through the function *notify_task_state()*. This function notifies the schedulers about state changes of ISRs and tasks including registration and deregistration. Alternatively, it is possible to use the state table of the RTOS model itself. In any case, the function *schedule()* selects the next runnable task based on this information.

Tasks and ISRs are sensitive to the *wakeup_event* of their TCB and block themselves until they receive the event from the RTOS context via *task_yield* and *task_deactivate*. Based on the return value of the scheduler, the context notifies the particular task or ISR through a *notify*. This mechanism guarantees that only one task is active at any time.

The *schedule()* function of the RTOS context is sensitive to *schedule_event*, which is generated in the delta-cycle after a task's state change. In the same simulation cycle, the *m_preemption_occurred_event* is sent to the context to guarantee that an already running task returns its control to the context. The *schedule* function first checks for ISRs that are in state *ready*. In the case of at least one ready ISR, the ISR scheduler is invoked and evaluates the next ISR to run. If there is no ready ISR or if the ISR scheduler does not return an ISR identifier, the task scheduler is executed.

Note that the scheduling interface also supports the implementation of nested and non-maskable interrupts (NMIs). The latter, for instance, can be implemented by returning an empty identifier or the identifier of the currently running task.

Our aRTOS library currently comes with a Round-Robin and priority-based scheduler with and without preemption. However, the previously introduced interface supports the implementation of arbitrary scheduling strategies.

9.2.5 Switching Context

Recall that the RTOS context interface supports the explicit definition of task state changes for context switches using a state model with five different states: *created*, *ready*, *running*, *waiting* and *dead* (cf. Fig. 9.6). Explicit state transitions performing context switches can be executed by the following functions:

task_wait changes the task/ISR state to *waiting* and triggers a call of the scheduler for the next delta-cycle. This function does not block and is typically followed by a SystemC *wait* that realizes the actual context switch. The function notifies the RTOS context about the following wait statement and gives other tasks the opportunity to get invoked. The implementation of the function is as follows:

```
1 int task_wait ()
2 {
3     int task = get_tcb ();
4     // set task to waiting
5     set_task_state (task, WAITING);
6     // enforce rescheduling
7     schedule_event.notify (SC_ZERO_TIME);
8     return task->pid;
9 }
```

task_deactivate deactivates a task. It is used for the implementation of events and semaphores through the RTOS context. Like the previous function, the current state changes to *waiting* and a call of the scheduler is triggered for the next delta-cycle. Then, *task_deactivate* waits for the *wakeup_event* of its TCB until another task reassigns the state of this task to *ready*. After being invoked by the scheduler, the task/ISR state changes to *running*. The function is implemented as follows:

```
1  int task_deactivate ()
2  {
3      task = get_tcb ();
4      // change state to waiting
5      set_task_state (task, WAITING);
6      // suspend
7      schedule_event.notify (SC_ZERO_TIME);
8      wait (task->wakeup_event);
9      // change state to running
10     set_task_state (task, RUNNING);
11 }
```

task_activate changes the state to *ready*. The function can be used for two different purposes. First, any calling task can set the state of another task given as a parameter to *ready*. Second, the function should be applied after each SystemC wait statement when the task is ready for activation. The underlying SystemC thread can execute but the RTOS scheduler has to finally release it. As with the previous function, the scheduler is triggered in the next delta-cycle. After waiting for *wakeup_event*, the state changes to *running*. The function implementation is as follows:

```
1  void task_activate (int pid)
2  {
3      task = get_tcb ();
4
5      // if the task is invoked
6      if ( task->pid == pid )
7      {
8          // change state back to ready
9          set_task_state (task, READY);
10
11         // suspend and wait on invocation
12         schedule_event.notify (SC_ZERO_TIME);
13         wait (task->wakeup_event);
14
15         // change state to ready
16         set_task_state (task, RUNNING);
17
18     } else
```

```
19      {
20          // Wake up another thread
21          task = get_tcb (pid);
22
23          // change state to ready
24          set_task_state (task, READY);
25      }
26  }
```

For explicit event management, the aRTOS library provides the class *sc_rtos_event* with methods *wait* and *notify*. This class implements an event mechanism directly by the RTOS model. The class and methods manage a list of tasks that are waiting on the event. By calling the member function *wait*, the calling task is entered into the task list for the event. Additionally, it is explicitly suspended by invocation of *task_deactivate*. Waiting tasks can then be reassigned to *ready* by means of the *sc_rtos_event* member function *notify*.

9.2.6 Periodic Tasks

Periodic tasks are frequently used in implementations of embedded real-time systems. A periodic task can be seen as a function that is periodically executed. The following code shows how to implement periodic tasks by means of our library:

```
1  void run ()
2  {
3      task_start (<user_callback_ptr>);
4      while (true) // endless loop
5      {
6          // wait on trigger
7          wait ( <trigger> );
8
9          // execute native code
10         my_code();
11
12         // consume time
13         CONSUME_CPU_TIME (<X>,SC_MS);
14
15         // notify about end of cycle
16         task_end_cycle ();
17     }
18     task_end ();
19 }
```

The loop first waits for the invocation by a timer or another event, after which the actual code is executed followed by a time specification. The task then explicitly executes *task_end_cycle*, which notifies the scheduler about the end of each period. Note that this enables the possibility to support the im-

plementation of periodic task scheduling. The actual scheduling has still to be implemented by a software scheduler.

9.2.7 Interrupt Service Routines (ISRs)

As described previously, the functions *isr_start* and *isr_end* register a SystemC task of an RTOS module as an ISR. In addition, our aRTOS library provides three functions for waiting for an IRQ, as well as for entering and returning from the ISR execution.

```
1  void run ()
2  {
3      isr_start (<ptr>);
4      while (true)
5      {
6          // wait on irq request
7          isr_wait (<event>);
8
9          // notify about entering ISR
10         isr_enter ();
11
12         // execute native code
13         my_isr_code();
14
15         // timing specification
16         CONSUME_CPU_TIME (..);
17
18         // notify about return from ISR
19         isr_return ();
20     }
21     isr_end ();
22 }
```

As given in the previous code, the pattern for an ISR is similar to a periodic task. The example defines the SystemC thread *run* as an ISR encapsulated by calls to *isr_start* and *isr_end*. The body of the ISR runs an infinite loop. In each iteration, the body of the loop first waits for an IRQ. The actual ISR code and the specification of its execution time are enclosed by *isr_enter* and *isr_return* in order to explicitly notify the scheduler about begin and end of one invocation. Note that, since *CONSUME_CPU_TIME* is interruptible, this implementation also covers cases where a higher prioritized IRQ interrupts the execution of a low priority ISR. Details of the ISR functions are as follows:

isr_wait waits for an IRQ that is defined as a *signal<bool>*. If the signal equals *true*, the IRQ is present and absent otherwise. During the time between the occurrence of the interrupt and the ISR execution it is possible that an IRQ returns to *false*. Our ISR event management considers this case since it (i) waits on the *wakeup_event*, i.e., on the activation by

the scheduler, and (ii) is sensitive to the falling edge of the IRQ. In case the IRQ returns to *false* before, the ISR is reset to *waiting*. The complete management is given by the following code:

```
1  void sc_rtos_context::isr_wait (sc_signal<bool> &irq)
2  {
3      task = get_tcb ();
4
5      for (;;)
6      {
7          // set task to waiting
8          set_task_state (task, WAITING);
9
10         // reschedule
11         schedule_event.notify (SC_ZERO_TIME);
12
13         // wait on high signal
14         if (irq.read () == false)
15         {
16             wait (irq.posedge_event ());
17         }
18
19         // set task to ready
20         set_task_state (task, READY);
21
22         // fire scheduler
23         schedule_event.notify (SC_ZERO_TIME);
24
25         // wait on scheduler or irq going low
26         wait ( (task->wakeup_event)
27                     | irq.negedge_event ());
28
29         // irq remains active
30         if (signal.read () == true)
31             break;
32
33         // IRQ has been canceled
34     }
35
36     // Task is running
37     set_task_state (task, RUNNING);
38 }
```

The function first changes the ISR state to *waiting* while waiting for an IRQ. Upon the occurrence of an IRQ, the state changes to *ready* and the ISR waits for the invocation by the ISR scheduler. Before *isr_wait* returns, the ISR state changes to *running* with the previously indicated exception in case the IRQ has been deactivated in the meanwhile.

isr_enter defines the beginning of the ISR code.

isr_return defines the end of an ISR code. Since execution time of ISRs has to be specified by *CONSUME_CPU_TIME*, which gives control to the corresponding scheduler and additionally may be interrupted by higher prioritized interrupts, the end of the ISR cannot be decided by just considering its state. Therefore, the end of the ISR code has to be explicitly marked by a call to *isr_return*.

9.3 Related Approaches

RTOS simulation with time annotated segments is either based on an abstract canonical RTOS or on a standard RTOS API. Our approach implements a canonical RTOS and is based in most parts on the work of Gerstlauer et al. [GYG03]. More details of that SpecC library were outlined by Yu [Yu05], who also introduced an approach for SoC software development and evaluation of different scheduling algorithms and their impact on HW/SW partitioning in early design phases. Communication between tasks, including interrupts, is implemented by means of events. ISRs are modeled as tasks. As task scheduling is implemented on top of the non-preemptive SpecC simulation kernel, simulations may give inaccurate results, which has most recently been resolved by Schirner and Doemer [SD08]. However, interrupts are still modeled as high priority tasks and have to apply the same scheduling algorithm as the software scheduler.

We can find multiple alternative approaches for RTOS simulation in SystemC.

Early work by Hastano et al. [HKH04] outlines a simple RTOS simulation in SystemC, where specific schedulers can be in principle derived from a basic class. They model processes by a 1-safe Petri-Net with atomic transitions annotated by time and power consumption. Individual state transitions are triggered by the μ-ITRON-OS based RTOS kernel via round-robin scheduling, and I/O operations call hardware operations via a bus functional model. They do not consider interrupt management.

Huss and Klaus [HK07] introduces a SystemC RTOS API for estimation of scheduling strategies by means of a Gumbel distribution to approximate execution times. Their API is in some parts comparable to our implementation. However, they do not consider interrupt management.

Desmet et al. [DVD00] and Yoo et al. [SGGA02] are early approaches for RTOS generation from SystemC. Later, [KBR06] introduced a tool-based approach for system refinement with RTOS generation. Step-wise refinement covers abstraction levels from CP (Communicating Processes) to CA (Cycle Accurate) models. In the context of the PORTOS (POrting to RTOS) tool, they introduce the mapping of individual SystemC primitives to RTOS functions.

Mapping to different target architectures is implemented by a macro definition. PORTOS is configured by a XML specification characterizing the individual target platforms. Krause et al. [MGR08] introduce SCAS (SystemC Abstract Scheduler) as an abstract SystemC library for timed functional models. SCAS basically provides additional classes for FIFO communication (`scas_fifo`) and software tasks (`scas_module`) with states and state changes. A SCAS task overloads the eight different variants of the SystemC wait statement to better reflect the preemptive nature of software tasks. However, interrupt management is not explicitly considered.

Hessel et al. [HRR+06] apply SystemC for abstract RTOS simulation within an approach for multi-processor system-on-a-chip (MPSoC) development and refinement of TLM SystemC models into individual software task and RTOS assignments to different processors. Similar to Yu, they support basic descriptions of task changes and task synchronization. Wait statements and blocking read/writes are replaced by RTOS calls for refinement.

Destro et al. [DFP07] introduce a refinement for multi-processor architectures in SystemC with a clear mapping from SystemC primitives to POSIX function calls. Starting from functional SystemC, first processor allocation and then HW/SW partitioning are performed. A final step maps SystemC to a co-simulation of hardware in SystemC and software running on top of an RTOS. After the mapping hardware threads are executed by a specific SystemC compliant hardware scheduler. Mapping is basically done by the exchange of SystemC macros and definitions, i.e., replacing `systemc.h` by a proprietary `sc2os.h`.

Posadas et al. [PAV+05] have published several articles on RTOS simulation. They introduce concepts of their freely available SystemC RTOS library PERFidiX, which covers approximately 70% of the POSIX standard. Software threads are partitioned into segments, which are annotated by timing information and controlled by a time manager. Time is estimated at runtime by means of overloaded operators of C primitives. PERFidiX supports interruptible wait statements. For timing analysis, the simulation time of the software task is suspended in the case of an interrupt and resumed after the execution of the ISR. For functionally correct simulation of the access sequence to global variables and communication, system calls and accesses to global variables are separated in one segment. Interrupts are separately managed as predictable (e.g., timeouts) and non-predictable interrupts. They report a gain in simulation speed w.r.t. ISS of more than 142 times in one of their first publications, including a 2x overhead in speed due to their operator overloading. [QPV+06] further extends the approach for TLM, where interrupts are received from HW components via the TLM bus interface. A separate SystemC thread monitors IRQs and creates a POSIX thread for the corresponding ISR simulation on demand.

As it can be seen, only very few approaches consider interrupts and thus address the problem of non-interruptible wait statements. In SystemC implementations, PERFidiX seems to provide an adequate solution of interruptible simulation of software tasks. However, in contrast to our implementation, PERFidiX is limited to the application of one scheduling strategy for software tasks and ISR scheduling. Additionally, it is based on POSIX and cannot be applied in earlier design phases when the decision on a specific operating system is still open.

9.4 Applications

To bring our library into the perspective of realistic applications, we briefly outline the integration into an automotive electronics system design flow and present some evaluation results of a simulation study.

Industrial automotive system development flows apply Integrated Development Environments (IDEs) that typically include editors, code generators, instruction set simulators, and debuggers for MIL (Model-in-the-Loop), SIL (Software-in-the-Loop), and HIL (Hardware-in-the-Loop) testing. For software development and analysis, current IDEs, like the AUTOSAR-compatible SystemDesk (dSPACE), combine software components (SWCs) to tasks, which are finally configured and assigned to an individual microcontroller, the so-called Electronic Control Unit (ECU). For simulation, the individual configuration and assignment can be combined into a DLL, which can be dynamically loaded for analysis where the actual operating system is typically not included in the simulation. Such an analysis currently covers a timeless simulation considering the activation points of tasks rather than their duration. Our approach additionally provides an analysis taking full account to all timing effects including task duration for detailed scheduling analysis.

Some automotive IDEs include code generation tools, like TargetLink (dSPACE), which automatically insert checkpoints as C-macros into the SWC code for code coverage analysis. In TargetLink, those checkpoints correspond to markers of basic blocks in the C code so that they can provide excellent means to interface with our timed RTOS simulation. Therefore, without applying any changes to the SWC code itself, checkpoint macros can be easily redefined for our SystemC task annotation, i.e., to retrieve the estimated worst case execution time of a software segment from a timing graph and inserting a call to the *CONSUME_TIME* function. Furthermore, by the use of SystemC, hardware components like I/O blocks or buses can be easily combined into one simulation model.

Figure 9.9 depicts the integration of our abstract SystemC RTOS simulation into an automotive systems IDE. Our system takes checkpoint-segmented software components (SWCs) from the IDE and links them to the executable

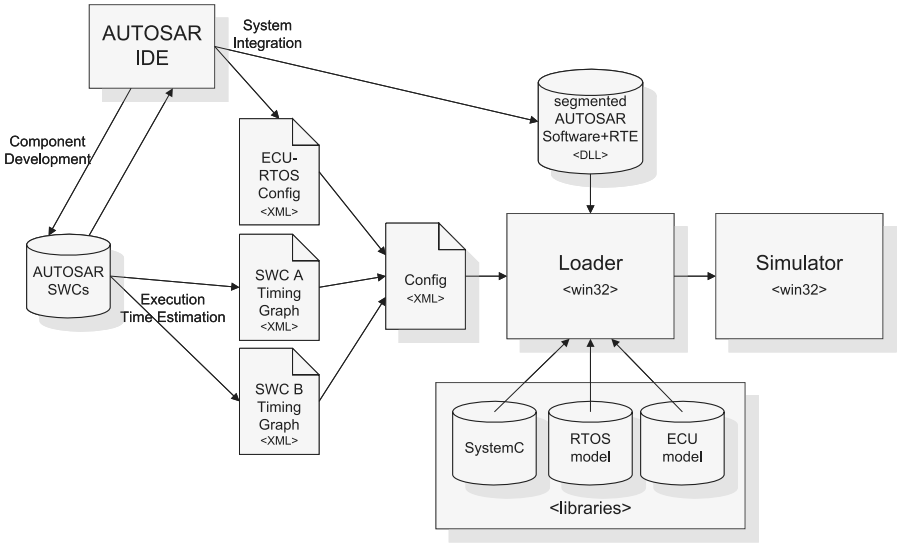

Figure 9.9. Configurator for simulations of automotive networks [Bec08].

SystemC simulator. Additionally, each SWC has to be analyzed and a timing graph with worst-case execution times generated. A final configuration file describes the ECU architecture, the allocation of SWCs to ECUs, and their association to the different timing graphs. Finally, a loader links the SWCs, ECUs, our RTOS library, and the SystemC simulation kernel to create the final executable of the simulator.

In our framework, the configuration is defined by an XML file where the following example gives some fragments of an injection control configuration.

```
<ecu>
  <id>FUELSYS_ECU</id>
  <class>sc_conf_ecu</class>
  ...
  <ecu_task_scheduler>
    <class>sc_conf_ecu_scheduler<class>
    <library>sc_conf_ecu_style.dll<library>
    <has_context>FUELSYS_ECU<has_context>
  </ecu_task_scheduler>

  <ecu_isr_scheduler>
  ...
  </ecu_isr_scheduler>

  <ecu_task><id>MAIN</id>
    <class>sc_conf_ecu_task</class>
    <library>sc_conf_ecu_style.dll</library>
```

```
18      <has_context>FUELSYS_ECU</has_context>
19      <has_scheduler>TASK_SCHEDULER</has_scheduler>
20      <wrapping_function>
21          fuelsys::MainApplication
22      </wrapping_function>
23      <priority>3</priority>
24      <autostart>true</autostart>
25    </ecu_task>
26
27    <ecu_signal>
28      <identifier>ENGINE_MODEL_FUEL_RATE<identifier>
29      <class>sc_conf_ecu_signal<class>
30      <library>sc_conf_ecu_style.dll<library>
31      <datatype>uint16<datatype>
32      <has_tracer>VCD_SIGNAL_TRACER<has_tracer>
33      <wrapping_signal>fuelsys::RteSignal1<wrapping_signal>
34      <address>0x00ad518c<address>
35    </ecu_signal>
36     ...
37  </ecu>
```

The configuration specification starts with a main identifier. Thereafter, references to C++ classes of a specific DLL implementing the task and ECU scheduler functions are defined. Finally, a list of different SystemC tasks and signals are given, each with its class identifier and the corresponding DLL.

In this context, we also performed several case studies for simulation performance evaluation. The following results apply to parts of a motor management. The evaluated application focused on the electrical differential drive of an automatically guided vehicle and compares to principles of a real engine management system with injection control, crank shaft hall sensors etc. The vehicle subsystem is composed of two controllers for the wheels and a module to communicate with the controlling PC via a serial interface. Tasks and ISRs of our system are composed of WCET-annotated software segments. They implement several drivers for hardware I/O in order to measure (1) the rotation speed of each wheel by means of a hall encoder, (2) the electrical current and (3) the voltage supply for the two drives. The software implements 5 ISRs and 11 tasks, where communication between tasks and ISRs is realized through events. Simulation results validated that the task scheduler performs correctly with a 250 Hz sampling rate of the controllers and a guaranteed estimated wheel rotation of up to 300 rpm (rotations per minute). Each hall encoder has two lines with a pulsed signal and a 90 degree phase shift. The rotation speed is evaluated by measuring the time between two pulses where a sensor generates 180 pulses for each rotation. The rotation direction can be determined by sensing the second signal at the rising edge of the first one. To identify the correct direction, the second signal has to be read as soon as possible after detecting a rising edge of the first. In the implementation, this means that the ISR

Task/ISR	WCET	WCRT	Deadline
TIMER0_COMPARE_ISR	54	214	16000
ENCODER0_ISR	74	601	4438
ENCODER1_ISR	74	679	4438
USART_RX_ISR	50	740	1527
USART_UDRE_ISR	53	239	1527
handle_clock_1KHz	7	743	16000
handle_clock_250Hz	9	752	64000
handle_command	497	1232	1527
signal_comm_tx_empty	324	590	1527
controller_left	468	633	64000
controller_right	468	1627	64000
analog_update	127	1761	16000
transmit_rpm	232	2424	64000
transmit_ticks	274	1940	64000
transmit_supply	259	1925	64000
transmit_throttle	256	1922	64000

Table 9.1. Worst case and actual execution times of different tasks.

that is sensitive to the first encoder signal reacts within $\frac{1}{4}$ of the minimal time between two pulses. Thus, for 180 pulses per rotation and 300 rpm, the upper bound for the response time (deadline) is supposed to be 4438 CPU cycles on a 16 MHz microcontroller.

Table 9.1 shows the simulated (worst case) execution times, the measured (worst case) response time, and the actual deadlines for the tasks and ISRs. All times are shown in CPU cycles where the ISRs are listed first. The table shows that all tasks and ISRs meet their deadlines. For task scheduling, we decided to apply fixed priorities without preemption. Here, the simulation validated that replacing the non-preemptive scheduler by a preemptive one has no significant influence on the response times. Additionally, we could show that tasks and ISRs keep their deadlines when we reduce the clock from 16 MHz to 4MHz to save power, provided the bit rate for serial communication is reduced from 115200 bps to 38400 bps at the same time.

For comparing the accuracy of our RTOS simulation library with a conventional approach, we investigated three different time annotations: (1) SystemC wait statements with accuracy of a segment of duration n, (2) n SystemC wait statements with accuracy of one cycle per statement ($wait(1)$), and (3) our interruptible wait by the means of the CONSUME_CPU_TIME function. Table 9.2 shows the measured response times of the ISR for the two wheel encoders.

We can see that the application of native SystemC wait statements over n time units gives a significant simulation error. This is because each controller

	(1) SystemC wait with segment accuracy	(2) SystemC wait with 1-cycle accuracy	(3) interruptible
ENCODER0_ISR	572 cycles	132 cycles	134 cycles
ENCODER1_ISR	650 cycles	210 cycles	212 cycles
Simulation time	3.6 s	39.0 s	4.5 s

Table 9.2. Maximal response and simulation times with different time annotations.

task contains software segments with a maximum time of 468 cycles during which they cannot be interrupted. Results also show that the invocation of *n wait*(1) statements, i.e., 1-cycle accuracy, results in precise response times. However, the simulation speed is poor due to the huge number of synchronization points within the RTOS model. Our interruptible wait achieves response times with negligible simulation error and a negligible overhead in simulation speed. Additional studies have demonstrated that our approach consistently keeps the simulation error below 2%. Our current studies have shown speed-ups between 4000x–40000x of our models with respect to ISS. However, since the applied ISS was optimized for debugging rather than for simulation speed, this comparison should be considered with reservations.

9.5 Conclusions

This chapter introduced our aRTOS SystemC library for abstract, accurate RTOS simulation. The library supports the development and evaluation of RTOS schedulers and covers all relevant timing effects including preemption with prioritized and nested interrupts. Our main studies and applications are currently derived from software developed for automotive systems that have evolved into fairly complex distributed, network-based systems over the last years. Here, CAN, LIN, MOST and—most recently—FlexRay(TM) buses come into application. Especially FlexRay(TM) with its high scalability requires early attention and thus needs new approaches with respect to analysis and simulation of the entire system. Our RTOS simulation provides fast and accurate RTOS evaluation and configuration that can be integrated with complex networks for overall system analysis. It therefore complements todays automotive integrated development environments that mainly execute a timeless simulation. Our applications have demonstrated that our approach seamlessly combines with AUTOSAR based development environments, where more details on the AUTOSAR standard can be found in the next chapter.

Acknowledgments

The work described in this chapter was partly funded by the research award 2007 of Paderborn University, the German Ministry of Education and Research

(BMBF) in the context of the ITEA2 project TIMMO (ID 01IS07002), and the ICT project COCONUT (Grant Agreement No. 217069). We also gratefully acknowledge the contributions of Markus Becker and Ulrich Kiffmeier (dSPACE) for the given application case studies.

References

[Bec08] M. Becker. Vergleichende Evaluation von Netzwerk-Simulatoren anhand einer Fallstudie. Master's thesis, Universitaet Paderborn, 2008.

[DFP07] P. Destro, F. Fummi, and G. Pravadelli. A smooth refinement flow for co-designing HW and SW threads. In *DATE'07: Proceedings of Design, Automation and Test in Europe*. IEEE Computer Society, Los Alamitos, 2007.

[DVD00] D. Desmet, D. Verkest, and H. DeMan. Operating system based software generation for systems-on-chip. In *DAC'00: Design Automation Conference*, 2000.

[ESHR08] R. Ernst, S. Schliecker, A. Hamann, and R. Racu. Formal methods for system level performance analysis and optimization. In *DVCon'08: Design and Verification Conference and Exhibitation*, San Jose, CA, 2008.

[GYG03] A. Gerstlauer, H. Yu, and D. Gajski. RTOS modeling for system level design. In *DATE'03: Design, Automation and Test in Europe*, 2003.

[HK07] S. A. Huss and S. Klaus. Assessment of real-time operating systems characteristics in embedded systems design by SystemC models of RTOS services. In *DVCon'07: Design and Verification Conference and Exhibitation*, San Jose, CA, 2007.

[HKH04] P. Hastano, S. Klaus, and S. A. Huss. An integrated systemc framework for real-time scheduling assessments on system level. In *RTSS'04: Proceedings of IEEE Int. Real-Time Systems Symposium*, 2004.

[HRR+06] F. Hessel, V. M. Da Rosa, C. Eduardo Reif, C. Marcon, and T. G. Serra Dos Santos. Scheduling refinement in abstract RTOS models. *ACM Trans. Embed. Comput. Syst.*, 5(2):342–354, 2006.

[IEEE06] IEEE. *IEEE Standard SystemC Language Reference Manual—IEEE Std 1666-2005*. IEEE Computer Society, New York, 2006.

[KBR06] M. Krause, O. Brinkmann, and W. Rosenstiel. A SystemC-based software and communication refinement framework for distributed embedded systems, 2006.

[MAF91] C. Mills, S. Ahalt, and J. Fowler. Compiled instruction set simulation. *Softw. Pract. Exp.*, 21(8):877–889, 1991.

[MGR08] M. Müller, J. Gerlach, and W. Rosenstiel. Abstrakte Modellierung von Hardware/Software-Systemen unter Berüsichtigung von RTOS-Funktionalitä. In *MBMVS'08: Proceedings of the 11th Workshop of Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, March 2008.

[NBS+02] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC'02: Proceedings of Design Automation Conference*, 2002.

[PAV+05] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *Des. Autom. Embed. Syst.*, 10(4):209–227, 2005.

[PB00] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *J. Real-Time Syst.*, 18(2/3):115–128, 2000.

[QPV+06] D. Quijano, H. Posadas, E. Villar, F. Escuder, and M. Martinez. TLM interrupt modeling for HW/SW co-simulation in SystemC. In *DCIS'06: Proceedings of the XXI Conference on Design of Circuits and Integrated Systems*, 2006.

[SD08] G. Schirner and R. Dömer. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *DATE '08: Proceedings of Design, Automation and Test in Europe*. IEEE Computer Society, Los Alamitos, 2008.

[SGD07] G. Schirner, A. Gerstlauer, and R. Dömer. Multifaceted modeling of embedded processors for system level design, Abstract. In *ASP-DAC'07: Proceedings of the 2007 Conference on Asia South Pacific Design Automation*, 2007.

[SGGA02] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic generation of fast timed simulation models for operating systems in SoC design. In *DATE'02: Proceedings of Design, Automation and Test in Europe*. IEEE Computer Society, Washington, 2002.

[Yu05] H. Yu. *Software Synthesis for System-on-Chip*. PhD thesis, University of California, Irvine, 2005.

Chapter 10

VERIFICATION OF AUTOSAR SOFTWARE BY SYSTEMC-BASED VIRTUAL PROTOTYPING

Matthias Krause, Oliver Bringmann and Wolfgang Rosenstiel

Abstract This chapter focuses on simulation-based verification of AUTOSAR software. It introduces the methodology and technical aspects behind AUTOSAR and outlines the affinities of the concepts of AUTOSAR and SystemC. It discusses in detail how SystemC supports the implementation of AUTOSAR and how SystemC can be applied with respect to the different AUTOSAR layers. Furthermore, it illustrates the different views of car makers and tier 1 suppliers onto the system and discuss how SystemC can support them. Therefore, the different layers of abstraction within TLM (**T**ransaction-**L**evel **M**odeling) space are introduced and different ways of integrating timing behavior into the entire system are shown. The article is attended by a case study of a traffic sign recognition system, which demonstrates the functional and timing evaluation of the entire system.

Keywords: Design Methodology, Embedded Software, Virtual Prototyping

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

10.1 Introduction

In recent years, the raising complexity of electronic systems has become a new challenge within the automotive industry. This is particularly true for software since most innovations in a car are made in the area of software. In future, this trend will continue and the part of software will steadily increase in the next years. Additionally, automotive electronic is a highly distributed electronic systems resulting in a further growing degree of interconnection and interaction. An increasing number of subsystems are involved in providing a specific system function and more system functions are sharing a heterogeneous network architecture. In this context, interaction becomes one of the most critical challenges of the design process. Another very automotive-specific problem is the differing view of OEM and tier 1 onto the system. Until recently, an ECU (**E**lectronic **C**ontrol **U**nit) is a single unit for the supplier. But the growing degree of interaction makes it necessary to consider the entire system. This is particularly true for the manufacturer who is responsible for the entire system and the integration of the several components. On the other hand, he considers the suppliers components as black box units. In terms of interaction, the black box consideration makes it difficult to focus the entire system.

Right now, a paradigm shift is proceeding from a ECU-centric view to an entire system view. Also well-known requirements like quality and development time are still important and result in the need of modularity, reuse and scalability of the software. It is difficult to reconcile this with highly hardware-dependent software of today's automotive systems that causes significant software modifications if changing the underlying hardware architecture. In summary, the likewise increased complexity of the systems caused many problems also in terms of stability, error-proneness, performance, reusability, modularity, processes and the like.

10.1.1 The AUTOSAR Initiative

Currently, AUTOSAR (**AUT**omotive **O**pen **S**ystem **AR**chitecture) [AS] tackles the problems. AUTOSAR is an international development partnership consisting of a multitude of car manufacturers, suppliers and tool vendors, defining concepts and workflows, how electronic automotive software-related systems can be formally specified and processed. AUTOSAR focuses on a software architecture that decouples software and hardware by offering a hardware abstraction layer and a basic software. The application software is implemented within modules. These software components communicate via well-defined interfaces. The goal is to make the application software completely independent from the underlying hardware architecture to allow an arbitrary distribution onto different ECUs. Configuration and generation processes built the final ECU software.

Lately, first standardization results of AUTOSAR have been published in several specification documents which are available at the AUTOSAR web page [AS]. Thus, car makers, suppliers and tool vendors start to transfer results into practice. However, the required engineering steps, or how-to-come from a logical to a technical architecture and implementation, are not well supported by tools, yet. For instance, one request of AUTOSAR is "the simulation of networked systems in combination with well defined timing information and constraints" [AS06b]. This is of strong interest since it helps to capture requirements and to analyze whether or not the intended system meets the defined requirements. For example, the timing behavior of a software component (i.e. the execution time) or of a dedicated communication can totally differ with respect to different system configurations. (This affects in turn the scalability and reusability of software components.) To allow the time based simulation, an appropriate timing concept is required which has not been sufficiently supported by AUTOSAR until now. AUTOSAR only offers timing assertions within the specification semantic. On the other hand, SystemC virtual prototypes provides a promising solution for simulation of distributed embedded systems since they allow to include the timing behavior the underlying platform architecture.

10.1.2 Virtual Prototyping

The introduction of virtual prototypes is also a consequent continuation of the changeover from a single ECU-centric development approach to an entire system view. They allow an early system simulation and analysis and hence the evaluation of the entire system. The abstraction level of virtual prototypes varies from an abstract and target platform independent level, that allows a very early system simulation, to a detailed system model that enables accurate timing consideration by integrating a detailed target platform description. In between, various intermediate levels exist for gradual integration of target platform aspects. Some work about virtual prototyping with SystemC is presented in several articles of the SystemC book [MRR03]. SystemC offers a comprehensive way to simulate, analyze, and verify software by means of virtual prototypes. Furthermore, it is even able to take the timing behavior of underlying hardware and communication paths into account. It provides an appropriate timing concept as well as an event-driven simulation kernel that enables the simulation of concurrent processes. Today's state-of-the-art modeling/simulation tools are limited to a single functional timing consideration (i.e. only the sequential timing order is considered). Already at a first glance, there are many similarities between SystemC and AUTOSAR with respect to the modeling structure between the both concepts.

10.1.3 Section Organization

The rest of the article is organized as follows: Sect. 10.2 gives a basic introduction to the AUTOSAR standard. Section 10.3 discusses the different system views onto distributed embedded systems with respect to the automotive industry (manufacturers an suppliers view) and shows correspondences to the abstraction levels of a SystemC based design flow. Section 10.4 points out a methodology to verify AUTOSAR software by mapping the AUTOSAR software onto SystemC virtual prototypes. Section 10.5 briefly introduces the consideration and integration of timing behavior into virtual prototypes. Section 10.6 demonstrates the introduced design methodology by an application example. Section 10.7 concludes this article followed by the acknowledgment.

10.2 Concepts of AUTOSAR

AUTOSAR is revolutionizing the art of software development in automotive application domain. Instead of the current state-of-the-art ECU-centric development approach, AUTOSAR focuses on the entire system. A fundamental feature is the separation of application and infrastructure which allows for a model-driven architecture like methodology, i.e. a platform independent software development of functionality. Applications can exist and communicate independently of a particular infrastructure and mapping onto ECUs in an environment called **V**irtual **F**unctional **B**us (VFB).

However, AUTOSAR comprises even more: it specifies methodologies and workflows on how to come from the system living in the VFB to software running onto particular ECUs and a three-layer ECU architecture. The ECU architecture consist of an application layer, a middleware layer, called **R**un**T**ime **E**nvironment (RTE), and the infrastructure layer, called **B**asic **S**oft**W**are (BSW). Assuming that the application elements of the application layer behave exactly the same like in the VFB, then RTE and BSW implement the VFB for a particular ECU. Properties of AUTOSAR applications are described with a specific language, called AUTOSAR software component template (as part of the entire AUTOSAR metamodel). In general, the AUTOSAR software component template is arranged into three parts regarding the structure, the behavior and the implementation of models.

10.2.1 AUTOSAR Technology

Regarding the wide complexity of AUTOSAR, this section can only give a short overview of the AUTOSAR technology. For further readings, please have a look on the documents published at [AS]. The documents provide detailed information about the AUTOSAR software architecture, methodologies and

tools as well as conformance tests. In addition, [Hei04] explains the methodology of the definition and generation of data exchange formats in AUTOSAR.

Software Component Template. Referring to the structure, applications encapsulate functionality within software components, whereas software components are available in two flavors: atomic software components and compositions. Atomic software components contain single threads of execution, so-called **R**unnable **E**ntities (RE), and are later-on mapped onto particular ECUs. Compositions are means to structure atomic software components and can therefore form hierarchies. It is remarkable that the top-most hierarchy level then represent the entire system. Components communicate via ports which are typed by interfaces. General communication paradigms between entities are sender-receiver or client-server communication. In this sense, interfaces are either described for sender-receiver or for client-server communication. The behavioral section contains the aforementioned Runnable Entities. Runnable Entities can either be triggered by so-called RTEEvents and will just be executed, or they can wait (inside) for an RTEEvent. In the first case, they are referred to as category 1 Runnable Entities; in the latter as category 2. Common RTEEvents are TimingEvents (a cyclic trigger), DataReceivedEvent (a trigger caused by the reception of data) or OperationInvokedEvent (a trigger caused by an request for an service). Additionally to the application software components, special software components encapsulate the dependencies of applications on sensors and actors. Such components are dependent on the ECU hardware. Beyond, there are concepts for implicit reading and writing of data, data consistency mechanisms or mode managements, and others.

Virtual Functional Bus. To allow hardware and infrastructure independent software view, AUTOSAR defines the Virtual Functional Bus. This bus virtually specifies all kind of communications between all software components. Therefore, all communications have to be implemented by well-defined ports. There are two different kind of ports: A PPort provides a specific service or data, a RPort requires a specific service or data. Ports are connected to interfaces which implement either client-server or sender-receiver communication. In a sender-receiver communication pattern, the sender distributes information to one or several receivers. On the other hand, in a client-server pattern, the server provides services for a client that initiates the communication. Figure 10.1 illustrates the virtual functional bus and the notation for ports and interfaces.

ECU Architecture. Software components, Runtime Environment and Basic Software compose the ECU architecture individually for each ECU by applying ECU configuration and ECU software generation tools. The runtime

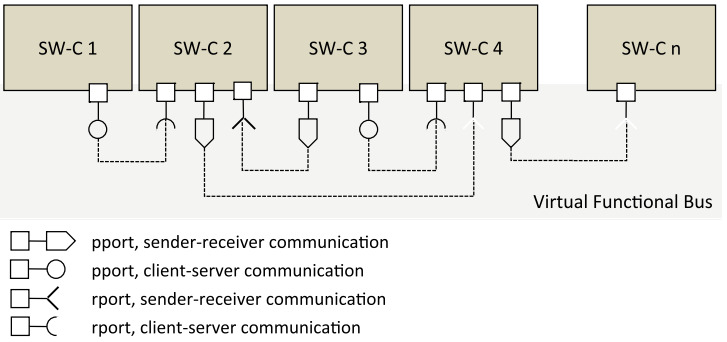

Figure 10.1. Virtual Functional Bus.

environment is an abstraction layer for all communications of software components with other software components (on the same or on other ECUs) or with the basic software. The basic software by itself is the sum of all standard software components and ECU specific components. They provide all necessary services for the application software and must be configured individually. Standard software components and ECU specific components include:

- **Operating System.** The operating system is mainly the scheduler and provides the dedicated scheduling algorithms. The scheduled objects are the Runnable Entities. The scheduler provides priority-based scheduling and protection functions at runtime. The AUTOSAR operating system bases on the OSEK OS [OSEK05] operating system.
- **Services.** The Services software components provide services for diagnostics, memory management, error management and other services (e.g. CRC calculation).
- **Communication.** The communication components are responsible for input/output operations, network management and communication access to the dedicated automotive busses (LIN [LIN], CAN [CAN], FlexRay [FlRa]).
- **ECU Abstraction.** The ECU Abstraction component provides a software interface to allow hardware abstraction. The interface enables access to the supported microcontroller drivers. This software component is ECU specific.
- **Microcontroller Abstraction.** The Microcontroller Abstraction interfaces the registers of a microcontroller and makes the higher software layers independent of the microcontroller. The microcontroller abstraction component is accessed via the ECU abstraction software components.

- **Complex Device Drivers.** Complex Device Drivers are interfacing other hardware components like sensor and actuator components as well as peripheral components. Complex Device Drivers are also ECU specific and in general used for resource critical applications. They offer a container for specific software implementation by providing the standard AUTOSAR interface for interconnection with the environment.

Figure 10.2 illustrates the ECU Software Architecture defined by AUTOSAR.

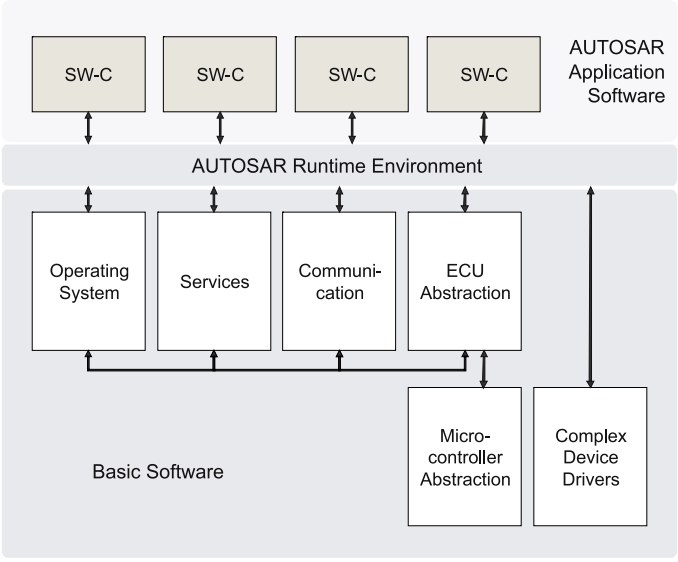

Figure 10.2. AUTOSAR ECU architecture.

Methodology and Workflow. AUTOSAR defines a design flow for a particular ECU implementation. The software components defined at VFB-level are mapped onto a dedicated ECU. As illustrated in Fig. 10.3, the workflow is divided into three categories: system configuration, ECU configuration and component implementation. Mainly, the flow bases on templates defined by an unique XML exchange format. The design process starts with the system configuration. The system configuration input contains information about software components, system constraints and ECU resources. The latter includes specification with respect to the processing unit, sensors and actuators, memory, peripherals and the like, as well. The system constrains provides the information to map software components onto ECUs and communications to busses. Hence, also topology information is included. The configuration process outputs a system configuration description including a complete communication matrix which describes the bus frames and schedules. Based on the communi-

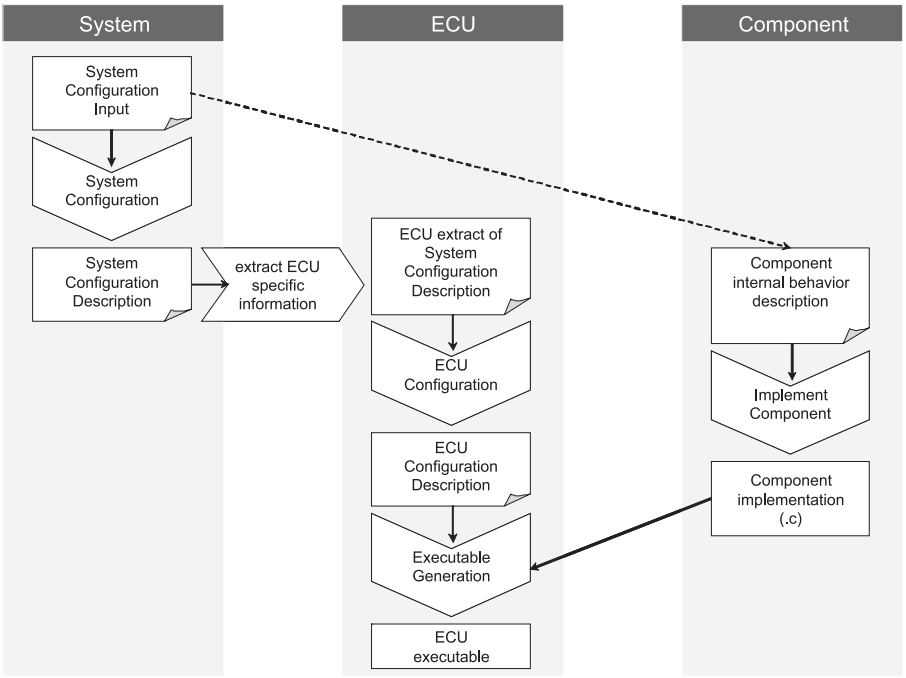

Figure 10.3. AUTOSAR methodology.

cation description, the required information for each ECU are extracted. The ECU configuration is a complex engineering process since it configures the complete basic software with respect to the extracted system configuration. The configuration concerns the runtime environment as well as all basic software components (e.g. operating system, communication). For instance, the configuration determines the task scheduling and integrates the required basic software modules. The result of ECU configuration serves for code generation of the basic software and run time environment. Software developers implement the software component with respect to the internal behavior description of the software component. Compilation is ECU specific. Finally the component object files are linked together with the RTE object files and the basic software object files.

10.2.2 Authoring, Modeling and Simulation

Since first specifications have been published, several tool vendors (e.g. dSPACE [dS], ETAS [ET], Vector Informatik [VI]) are providing tools to support the AUTOSAR design methodology. Authoring tools are supporting the workflow, i.e. the configuration and mapping process. At the cur-

rent state of the art, they serve as input for the RTE generation (e.g. Vector Informatik DaVinci Developer, dSPACE SystemDesk), BSW generation (e.g. Vector Informatik MICROSAR, ETAS RTA) as well as the code generation (e.g. dSPACE TargetLink, ETAS ASCET). Additionally, AUTOSAR specifies how to interact with behavior models [AS06b] and, in particular, how to apply Simulink [AS06a]. Also an UML metamodel for the tool Enterprise Architect [EA] has been implemented and is provided at the AUTOSAR homepage [AS]. Tool vendors are also integrating AUTOSAR into their product lines for modeling (e.g. Vector Informatik: DaVinci System Architect), simulation and test (e.g. Vector Informatik CANoe, ETAS INCA). This article contributes to simulation and verification of AUTOSAR software by applying virtual prototyping based on SystemC. Currently, this development trend is an actual object of research but of significant interest within the automotive industry, since AUTOSAR claims toward supporting timing concepts for simulation in combination with timing behavior in future.

10.3 Different System Views on Distributed Embedded Systems

In automotive industry exist different views onto the electronic system. Tier 1 supplier are mainly interested in a ECU-centric view whereby OEMs have to consider the entire system. This problem is mitigated by AUTOSAR by introducing the Virtual Functional Bus and the corresponding design methodology. Also SystemC is affording an entire system view. To support the existing AUTOSAR design flow, its innovations are picked up and consequently proceeded. Here, two points are essential and also supported by SystemC: The first one is to use abstraction levels similar to AUTOSAR, particularly at the architecture independent level (i.e. the VFB). The second one is the separation of application and infrastructure. This separation also expresses the different views of tier 1, which is mainly interested in the application, and OEM, which is mainly interested in the infrastructure.

10.3.1 Manufacturers and Suppliers View

In automotive industry, parts of the functionality are developed by the suppliers. Control functions are often developed with behavior modeling tools and then transformed into source code of a programming language, usually C/C++. Before AUTOSAR, application has almost been developed independent from the entire system architecture as illustrated in Fig. 10.4. The integration of applications into the entire system has been performed lately, in general with prototyping platforms. During design phase, basically three levels of abstraction have been passed through: specification level with no timing

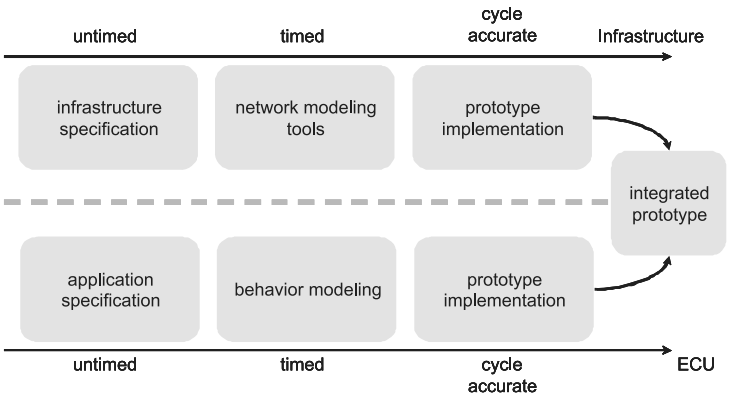

Figure 10.4. ECU-centric system view with separated consideration of infrastructure and component.

behavior, a timed modeling level whereas time has been considered by sequenced behavior and cycle accurate timing by implementation at prototyping platforms.

With AUTOSAR, the entire system is focused already in early design phases. This paves the way for an early system integration. VFB, infrastructure and component templates as well as the suggested workflow allow a much better exchange of information between infrastructure (focused by the OEM) and application (focused by tier 1) and hence, the consideration of the respective requirements and requests.

With respect to potential system views, the former independent infrastructure and application views now span the matrix of Fig. 10.5. Several combinations are possible, but not well supported until now. The different combinations reflect the different interests of OEM and tier 1. Within the design space of the matrix, the OEM would refine the infrastructure (black arrows in Fig. 10.5) while the tier 1 would refine the application (gray arrows in Fig. 10.5). But both would consider the behavior of the entire system. Nevertheless, specific timing requirements are still considered late in the design phase after implementation on real prototypes. A more early system integration is possible by introducing virtual prototyping models implemented in SystemC. Additional modeling techniques including the TLM 2.0 standard [Mon07] and the event based simulation kernel arrange the introduced matrix in more detail as explained within the next section. A more detailed discussion about this matrix including a communication and software refinement flow in SystemC is found in [KBR05].

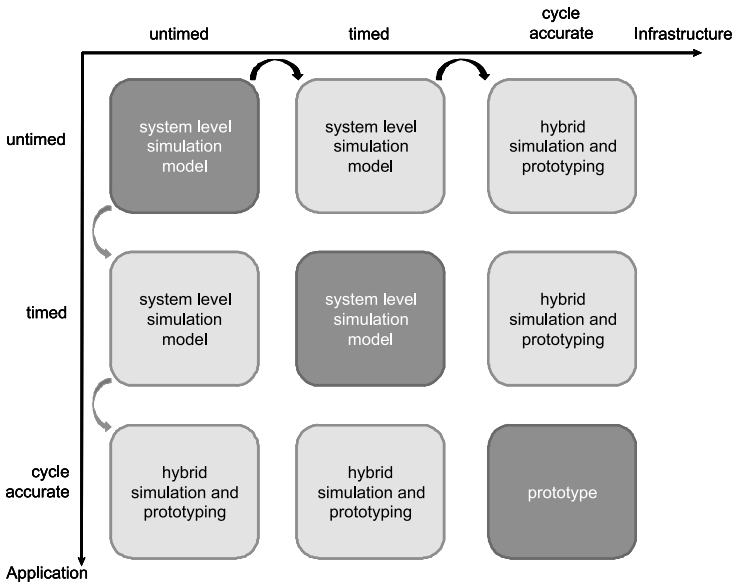

Figure 10.5. Design space matrix for a joint and entire system view.

10.3.2 SystemC Design Methodology

In a SystemC-based design methodology, different modeling styles, including transaction-level modeling [Mon07] and various levels of abstraction are used. The OSCI Transaction Level Working Group [TLM] has defined different levels of abstraction for transaction-level modeling. These levels are introduced in [Don04] and [DBR04] and are briefly presented below:

- **Communicating Processes (CP).** At this level, the behavior is partitioned into a network of parallel processes exchanging data through point-to-point connection. Hence, there is no arbitration of the data communication. Systems modeled at this level are still architecture and implementation independent.

- **Communicating Processes with Time (CPT).** This level is functionally identical to CP and annotated with high-level performance data.

- **Programmers View (PV).** The PV level is much more architecture specific. Bus or NoC models are instantiated to act as transport mechanisms between the model components. The models are sequenced but untimed. The PV level is register accurate.

- **Programmers View with Time (PVT).** As with the CPT level, a PVT level is functionally identical to the PV level but annotated with estimated multi-cycle timing information.

- **Cycle Callable (CC).** At this level, the system behavior includes cycle-true details and communication models are protocol-true.

If the design process starts at the highest level of abstraction (i.e. CP), the model can be refined stepwise down to CC level. Communication and computation are refined independently. This expands the matrix of Fig. 10.6 which is traversed during the refinement process.

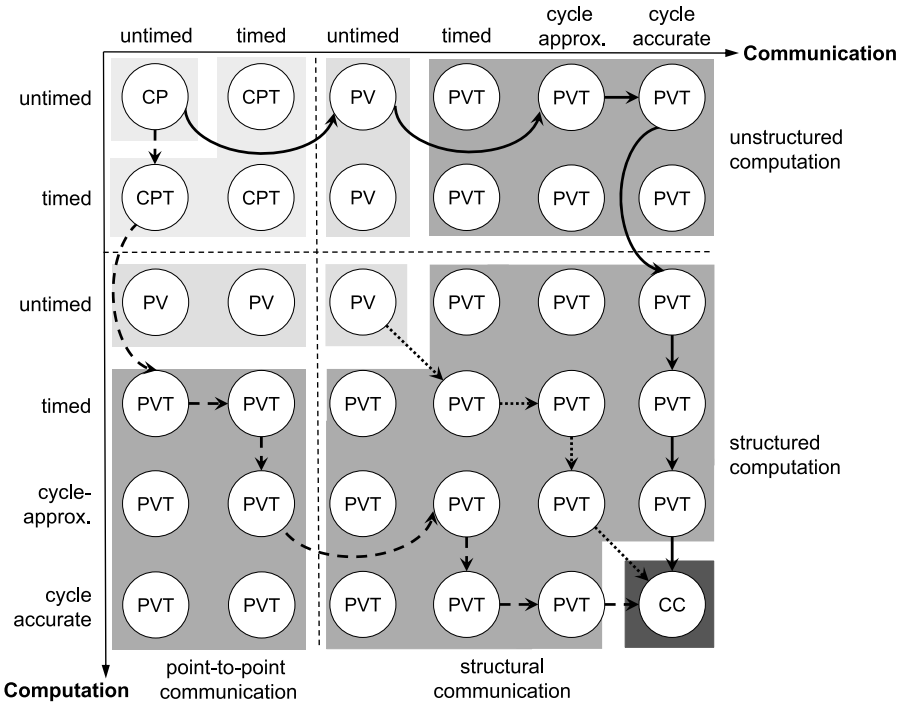

Figure 10.6. Abstraction level matrix of the refinement flow in TLM space.

The introduced view of the abstraction levels is strongly influenced by a hardware design perspective. Since the focus is on embedded software, abstraction levels have to be examined more closely. By definition, point-to-point communication is refined to an abstract bus model at PV level which means that communication takes place in a sequential scheduled order. A sequential order is also required for a software implementation of computation tasks, so the execution switches from a parallel to a sequential order at PV level. By sep-

arating the communication from the computation, we can define the following scenarios:

- Communication at PV/PVT or either CC, and computation at CP/CPT. The communication is refined to an abstract (timed or untimed), cycle approximate or cycle accurate behavior model while keeping the computation unscheduled and parallel. By refining the communication from CP/CPT to PV/PVT the computation processes are allocated to a processing element, but they are still simulated in parallel on the SystemC simulation kernel.

- Computation at PV/PVT and computation at CP/CPT. The communication is performed by point-to-point communication calls and (software-) computation is refined to a scheduled order with or without timing information (timed or untimed structured computation in Fig. 10.6). Please note that both scenarios are defined as PV/PVT in Fig. 10.6.

Basically, there are multiple possible paths through these levels. An OEM has to consider the infrastructure hence communication refinement down to a cycle accurate level of abstraction is recommended in order to obtain significant simulation results with respect to the communication architecture (continuous line in Fig. 10.6). Otherwise, tier 1 suppliers require an early refinement of computation while keeping the communication at a high level of abstraction (dashed line). Finally, a mixed level modeling for distributed systems is possible. Some of the computation may already be refined to a cycle accurate level while other computation entities are still at a time level of abstraction. This is particularly interesting for enhancements of systems application. New application features, which are initially described at an high abstraction level, are allowed for early integration into the virtual prototype which already introduces accurate timing behavior. Different configurations, also reconfiguration and remapping of existing system components, are easy to implement and simulate in comparison to a real prototyping platform.

10.4 Applying SystemC for AUTOSAR Software Verification

Since AUTOSAR has specified concepts, infrastructure and workflows, but does neither consider timing concepts nor system evaluation, simulation of the entire system is a recommended step to evaluate timing and behavior in an early design phase. Early simulation helps to find errors and bottlenecks within the design resulting in decreasing development time by preventing possible redesigns. Moreover, simulation with virtual prototypes evaluates the timing behavior of the entire target architecture that has also a strong impact on the behavior of the entire system with respect to performance and possible errors.

Such a detailed timing simulation is not able with state-of-the-art behavior modeling tools (e.g. Simulink).

SystemC enables entire system simulation. A particular value of SystemC, with respect to the design at system level, is the ability to design and model the functionality of embedded distributed systems as well as the required target architecture and infrastructure within one design language. This enables simulation and evaluation of a software application on its underlying target architecture and infrastructure respectively, both specified in SystemC. Additionally, SystemC introduces a simulation concept for the designs and provides a simulation kernel as well. This concept includes timing notations and timing behavior which is not part of AUTOSAR. In brief, SystemC offers those features that are not defined within AUTOSAR or supported by additional simulation tools until now. Therefore, sharing both methodologies will lead to an increase in value. The benefit is to enable simulation of interconnected AUTOSAR software components by integrating timing behavior already at a high level of abstraction to the communication as well as to the application, and hence to detect timing caused errors at an early design time. As a result, the simulation and evaluation of the entire system is affecting the configuration and mapping decisions respectively of the AUTOSAR design flow.

10.4.1 Affinities between AUTOSAR and SystemC

On closer examination, there are a lot of affinities between the AUTOSAR Software Component Template and the SystemC language. In terms of structure, both have entities containing behavioral elements and both can form ordered hierarchies. Therefore, AUTOSAR software components can generally be represented by SystemC modules (SC_MODULE) as shown in Fig. 10.7. Since there is no extra hierarchical element in SystemC, AUTOSAR compositions map to SC_MODULE, too.

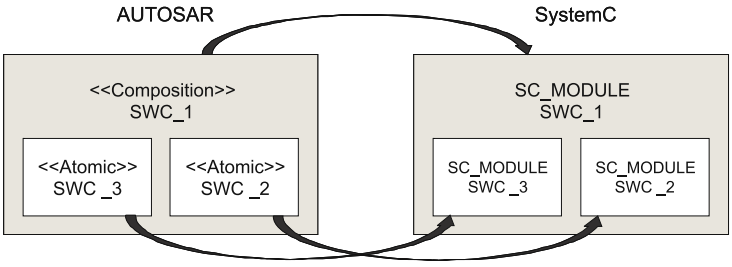

Figure 10.7. Analogy regarding AUTOSAR software-components or compositions and SystemC modules.

Figure 10.8 depicts the context of the "schedulable" ("triggerable") entities. In general, AUTOSAR Runnable Entities can directly be mapped to SystemC processes. Furthermore, since AUTOSAR and SystemC distinguish two types of schedulable entities, a one-to-one mapping of AUTOSAR Runnable Entities of category 1 to SC_METHODs and of AUTOSAR Runnable Entities of category 2 to SC_THREADs is possible. That also means inherently that both offer the opportunity of triggering schedulable entities via events and can wait for an trigger.

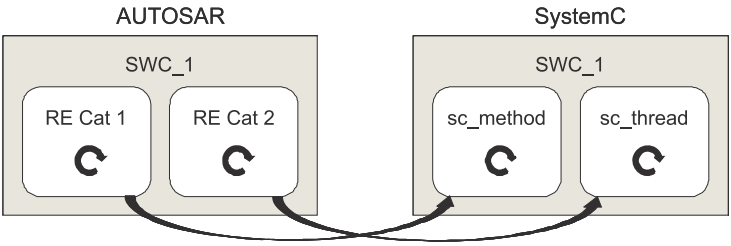

Figure 10.8. Analogy regarding AUTOSAR Runnable Entities (RE) and SystemC methods and threads.

Figure 10.9 shows analogies regarding communication. Both have the concept of ports: AUTOSAR ports, regardless their direction (provided or required), have their counterparts in sc_port. The same holds slightly true for interfaces which type ports. The marginal difference here is that AUTOSAR explicit states a specific kind of interface: sender-receiver or client-server. SystemC, however, hides this detail within the concept of channels. In both technologies, ports can be exported throughout the entire hierarchy. AUTOSAR realizes this by the concept of delegation connectors, whereas SystemC provides the specific sc_export construct.

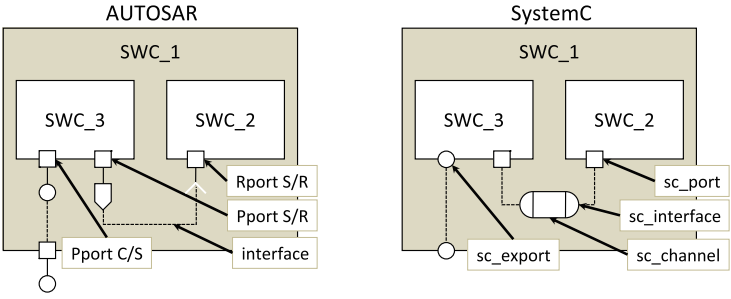

Figure 10.9. Analogy regarding AUTOSAR ports and interfaces and SystemC ports, interfaces and channels.

10.4.2 Mapping of AUTOSAR Software Components onto SystemC Virtual Prototypes

Like in the SystemC-based design methodology, also the AUTOSAR methodology has different views which can be compared to abstraction levels. Considering applications independent of a particular infrastructure and mapping onto ECUs is the highest level in AUTOSAR. Similarly, the SystemC based design approach starts at CP level. Systems modeled at CP level are still architecture and implementation independent, and there is no arbitration of the data communication. In contrast to the widespread view of SystemC as a hardware modeling language, this level of abstraction is completely independent of the partitioning into hardware and software. The communicating processes describe software processes which are later completely mapped to a target processor.

Mapping AUTOSAR VFB View to SystemC CP Level. As a consequence of the similar abstraction views, an AUTOSAR design can be transformed into an equivalent SystemC design at CP level. However, the design can be simulated by the provided SystemC simulation kernel. Since the VFB does not contain functional behavior (i.e. the application implemented by the Runnable Entities), the equivalent CP model just contains information regarding the infrastructure for the moment. The AUTOSAR application is developed independently of the configuration flow (cf. Fig. 10.3) but implemented by the guidelines of the software component template. Hence, such a component can easily be integrated into the CP model which implements the software component infrastructure and the VFB. If functional behavior of an application does not exist as yet, communication traffic can be generated by cyclic or random transmitting data of the corresponding data type. Cycle periods must be specified by the designer. This is possible because at VFB level the communication relationships and data types as well as Runnable Entities are already specified. The early integration of application into software components and the likewise involved simulation supports the design process and the decisions of the configuration and mapping process. Moreover, partial decision can be made and immediately proved by the simulation. For example, the OEM may prove the communication infrastructure by replacing the point-to-point connections by the dedicated bus. This requires configuration concerning the communication but not the operating system or ECU-configuration as well. On the other hand, the entire SystemC model can be refined starting from CP level. As already discussed in Sect. 10.3.2 (cf. Fig. 10.6), the SystemC design can be refined in different ways, communication driven or computation driven. Refining a SystemC model means gradually stepping from an abstract system specification to the desired target architecture. Each step adds new information about timing

and architectural details to the design. The hardware (e.g. CPU) as well as the bus protocol or either the timing configuration of the bus are responsible for the global timing behavior. Structural information is given by the scheduling policy, the bus access method, and the bus protocol for example. Refinement aims to make a decision which target platform should be used and to adapt the system according to that architecture. After each refinement step, the chosen target architecture can be evaluated with respect to given requirements. However, the user can refine communication and computation independently and keep the computation at a high abstraction level and firstly refine the computation or vice versa. Communication refinement is supported by encapsulating communications into channels and separates them from the computation. Thus, channels can easily be swapped, e.g. the channels implementing the point-to-point communication (that derives from the VFB) are swapped by channels implementing a specific behavior of a communication protocol. The application is left untouched if the same interface is used as before. Bus models are embedded within the hierarchy of the inserted channel. Such models can be untimed, timed, cycle approximate or cycle accurate as well. Internal refinement between the different accurate transaction level models is possible as well as refinement to a register transfer level (RTL) model. In this case transactions are replaced by signals. This refinement technique is described in [GLMS02] in detail. Computation refinement from a unstructured (i.e. parallel processes at CP level) to a structured (i.e. scheduled processes at PV level) execution order is done by introducing a scheduling policy to the implementation of the Runnable Entities which are mapped to a single processing element and which are executed in parallel at CP level. Further refinement steps introduce the complete basic software as well as the processing element models by itself. The refinement process in SystemC is comparable with the configuration and mapping process of AUTOSAR. The required information for SystemC refinement is also available within the AUTOSAR configuration files. The information is extracted to the appropriated XML configuration for SystemC. In turn, the evaluation results are used for the configuration and mapping flow of AUTOSAR. Figure 10.10 shows both design flows and the transition points from AUTOSAR to SystemC and vice versa. On the other hand, it is also possible to transform AUTOSAR into a virtual prototype after configuration and mapping (as also illustrated in Fig. 10.10). However, this requires the evaluation of transformation rules regarding the RTE.

Mapping AUTOSAR ECU View to SystemC. By applying the AUTOSAR workflow, ECU view including RTE and BSW is generated. The RTE implements the VFB, i.e. it implements the interfaces and events for triggering Runnable Entities. In contrast to the transformation of VFB to CP, the RTE transformation to SystemC PV/T requires changes of the transformation rules

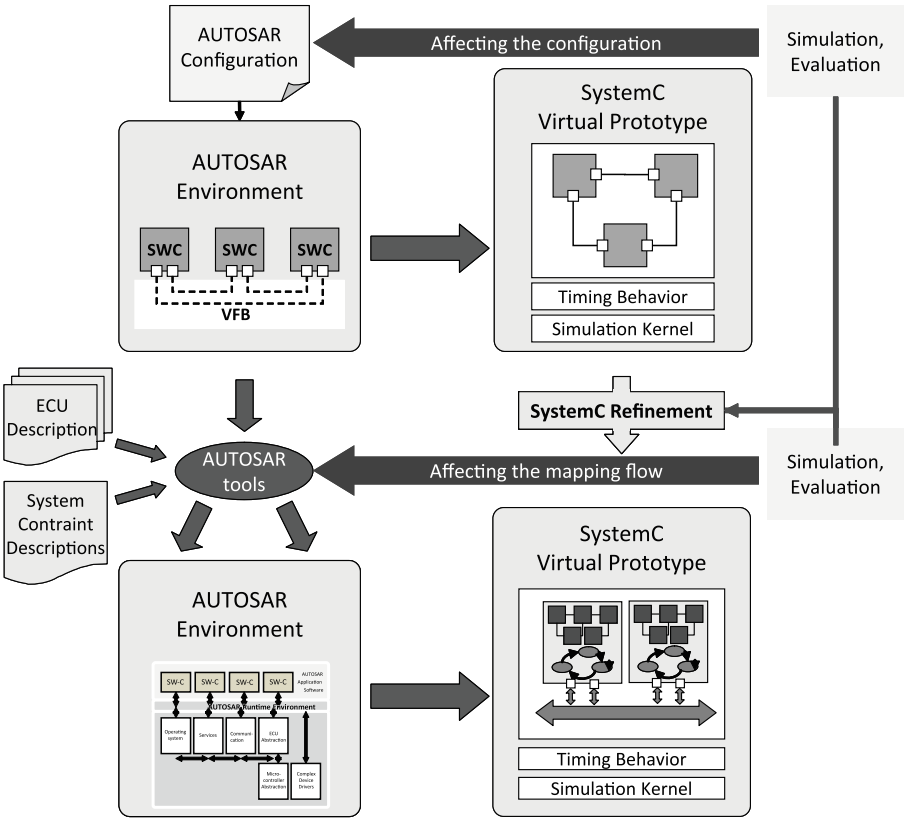

Figure 10.10. Concurrent AUTOSAR and SystemC design flow with potential transition points at VFB level and ECU level.

if also the virtual prototype has to provide the RTE.[1] SystemC PV/T simulates behavior that is running at one processing element not in parallel but in a sequenced order by providing the dedicated scheduling mechanisms. The Runnable Entities (or the communicating processes respectively) are partitioned to a specified processing element. By taking a closer look at the transformation, several abstraction levels of the SystemC design space (cf. Sect. 10.3.2 and Fig. 10.6) can be the target. Figure 10.11 illustrates these possible target levels. The ECU can be modeled by an abstract model of the OS considering only the scheduler or also other OS components. This is the untimed or timed programmers view. A cycle approximate view is implemented by using In-

[1] Basically, one can also consider the SystemC CP model as an implementation of the sum of all RTEs connected with a "minimum BSW" only consisting of point-to-point communications and no scheduling, bus, or other BSW components. Hence, neither configuration nor mapping is required.

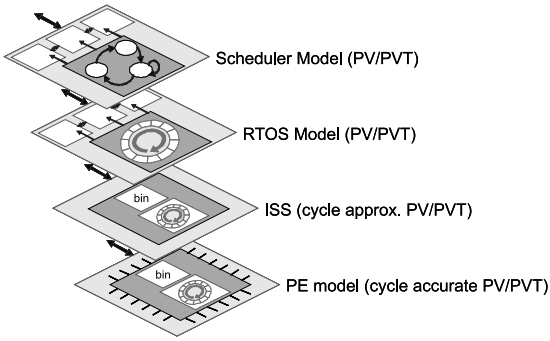

Figure 10.11. Potential SystemC target levels for transformation from ECU view.

struction Set Simulators or even more accurate **P**rocessing **E**lement (PE) models. Application, RTE and BSW do not need to be transformed into SystemC but compiled and run as binary on the models. SystemC wraps the models and acts as simulation master to connect several models with the communication infrastructure. However, the implementation is very expensive and simulation performance is very slow the more PE models are included. Therefore transformation onto an abstract RTOS model is recommended and discussed further.

The implementation of the BSW by an abstract RTOS model in SystemC entails two questions: First, how the BSW is implemented in SystemC with respect to the various configuration options. Second, dependent on the solution of the first question, how are the Runnable Entities handled within the SystemC simulation environment. The first question concerns the different modules of the BSW. A more detailed consideration of the several modules clarifies this point. Figure 10.12 shows the modular implementation of the BSW. The modularization is due to the abstraction of the hardware (i.e. the microcontroller) and the ECU as well. As already known, software components other then specialized SWCs are implemented independently of the underlying microcontroller and ECU as well. BSW modules introduces dependencies on the infrastructure, e.g. memory services (for memory configuration) and communication services (dependent on the bus network). Furthermore, the hardware abstraction layers modules are dependent on the ECU or external devices respectively. The driver modules and parts of the system services are microcontroller dependent software modules. Hence, the big question is which modules (supposed that they are used within the configuration are) are used from the generated BSW code (i.e. the source code is integrated into the SystemC model) and which modules are represented by an abstract SystemC model.

Since hardware components are not considered at this abstraction level, all hardware-dependent software modules of the BSW (i.e. complex drivers,

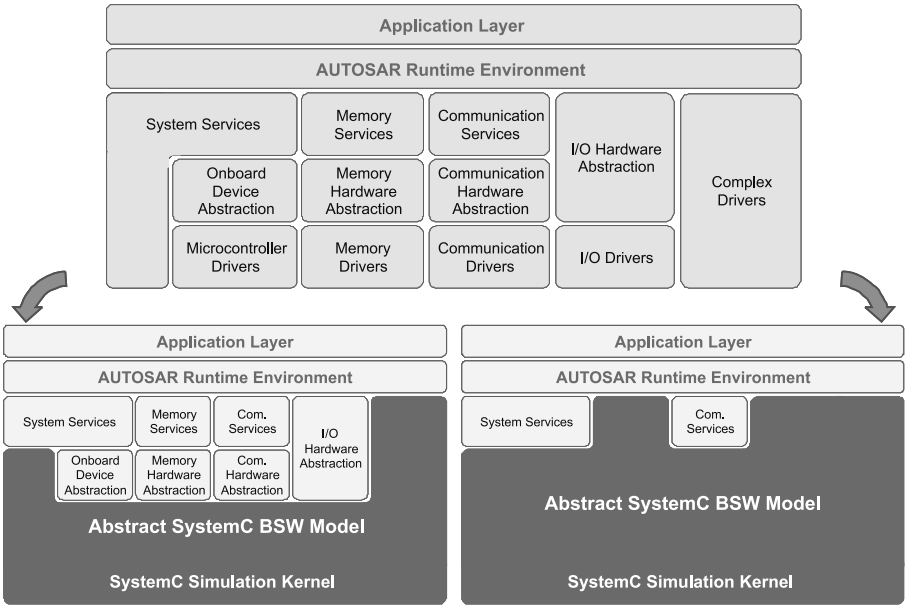

Figure 10.12. Basis software modules with different implementation alternatives for SystemC.

I/O drivers, communication drivers, microcontroller drivers and partially the system services) should be abstracted by the SystemC BSW model. Figure 10.12 shows two options of using generated software modules of the BSW. In case one, all modules that do not depend on the microcontroller are used (service and hardware abstraction modules). In the second case, only some system and communication services are used while the other modules are simulated by the abstract SystemC BSW model. In both cases, the RTE is not translated into SystemC syntax but the generated RTE configuration is used. This requires the implementation of a well defined interface between the RTE and SystemC to integrate the BSW model into the interaction. Furthermore also the interaction between source code and modeled BSW modules has to be supported by SystemC interfaces. This may be a difficult task if the model abstracts away some part of functionality. Besides, standardized interfaces between the single BSW modules are helpful to realize the implementation of BSW module models. A more difficult task is the interaction of the system services with the SystemC model. Some parts are directly connected with the underlying microcontroller and no standardized interfaces are used (i.e. no hardware abstraction is used within the system services). Such parts include access to interrupt control registers, processor status words and stack pointers. Further direct hardware dependency may include memory protection, time protection, time synchronization by a global time source and “privileged/non-

privileged modes" of the microcontroller. Hence, the hardware dependent parts of the system services have to be detached and abstracted by the BSW model.

To avoid this problems of interconnecting generated BSW modules with the SystemC BSW model it may be more practical to model the entire BSW by an abstract SystemC model. In this case the integration of the SWCs is similar to the integration at CP level that has directly been derived from the VFB. This includes also the transformation of RTE to equivalent SystemC elements. However, as already mentioned, the transformation rules for RTE elements have changed. This is due to the SystemC RTOS model since a process that derives from a Runnable Entity is not longer implemented by a thread- or method-process (SC_THREAD, SC_METHOD) but by a RTOS model task. This is the object of scheduling for modeled RTOS scheduler. Furthermore, also the special task concept of the AUTOSAR operating system has to be considered by the RTOS model. A task can include more than one Runnable Entity (of category 1). Those REs are executed sequentially within a task (basic task) or executed depending on a OS event (extended task). Figure 10.13 illustrates this behavior. The SystemC model has to implement the special task behavior. Thus, only the RTOS model itself is implemented at SystemC thread level. SystemC threads (SC_THREAD) should be used rather than SystemC methods (SC_METHOD) to allow task interruption. More details on RTOS modeling can be found in former chapters of this book. Finally, there should be no suggestion which kind of suggested implementation the user has to accomplish. It rather depends on the kind of application, e.g. how powerful is the available RTOS model or which BSW modules are even in use.

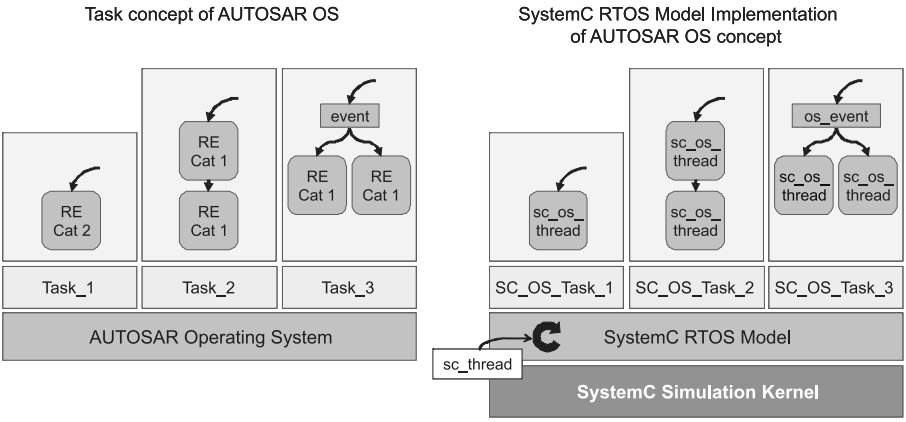

Figure 10.13. AUTOSAR task concept and its implementation in SystemC.

Formal Transformation Rules. This section finally summarizes the transformation rules discussed before. The affinities shown in Sect. 10.4 are the basis of the transformation from AUTOSAR to SystemC. Basically, the complete semantic of the Software Component Template has to be mapped to SystemC semantic. This section summarizes the formal transformation rules for the mapping from AUTOSAR to SystemC. This concerns the VFB (Table 10.1) as well as the RTE (Table 10.2) whereby SystemC CP/T and PV/T has to be distinguished.

AUTOSAR	SystemC PV/T
ComponentType	abstract class
AtomicSoftwareComponentType	SC_MODULE
SensorActuatorSoftwareComponent	SC_MODULE
CompositionType	(hierarchical) SC_MODULE
ComponentPrototype	specific instance of an SC_MODULE
PortPrototype	abstract class
PPortPrototype	sc_export for client-server-communication and sc_port for sender-receiver-communication
RPortPrototype	sc_port
PortInterface	abstract class
SenderReceiverInterface	sc_interface impl. sc_channel
ClientServerInterface	sc_interface impl. sc_channel
Datatype	abstract class
PrimitiveType	abstract class
Range	abstract class
IntegerType / e.g. uint8	sc_int, sc_uint, int, . . . / e.g. uint<8>
RealType	float, double
BoolType	bool, sc_bit
OpaqueType	sc_bv
CharType	char
StringType	sc_string
ConnectorPrototype	abstract class
AssemblyConnectorPrototype	sc_export-sc_port binding for client-server-communication and sc_port-channel binding for sender-receiver-communication
DelegationConnectorPrototype	sc_export-sc_export binding or sc_port-sc_port binding

Table 10.1. Formal transformation rules for the Virtual Functional Bus.

By using the UML 2 profile for AUTOSAR [AS06c], the transformation rules are implemented by tool-based mapping of XMI onto SystemC. How to map from UML to SystemC can be checked e.g. in [VSBR06].

AUTOSAR	SystemC	SystemC RTOS model
Internal Behavior	In SystemC an AtomicSoftwareComponent must only have one internal behavior.	
Runnable Entity—category 1	SC_METHOD	sc_os_thread
Runnable Entity—category 2	SC_THREAD	sc_os_thread
RTEEvent	abstract class	
TimingEvent	specific instance of sc_event	
DataReceivedEvent	specific instance of sc_event	
DataReceivedErrorEvent	specific instance of sc_event	
DataSendCompletedEvent	specific instance of sc_event	
OperationInvokedEvent	specific instance of sc_event	
AsynchronousServerCallReturnsEvent	specific instance of sc_event	
ModeSwitchEvent	specific instance of sc_event	
WaitPoint	wait(specific _sc_event)	
DataReadAccess	implicit data access—specific	
DataWriteAccess	implementation of semantic in SystemC	
DataReceivePoint	explicit data access—via sc_export respectively sc_port	
DataSendPoint	explicit data access—via sc_export respectively sc_port	

Table 10.2. Formal transformation rules for the Runtime Environment.

10.5 Integration of Timing Behavior into Virtual Prototypes

SystemC offers timing behavior concepts to consider the timing of a system during simulation. However, the user is responsible for reproducing the timing behavior as accurate as possible (with respect to the chosen abstraction level). This section gives a brief outline of timing integration into virtual prototypes. The first part provides an overview of components which, directly or indirectly, influence the timing of an entire system. The second part introduces options of timing integration into system models. However, for detailed information please refer to the indicated literature. A more detailed consideration would be beyond the scope of this book.

10.5.1 Time Consuming System Components

Computation has no timing behavior as long as the target platform is unknown. The timing is affected by the target platform component where the computation is executed. Also, operating system and device drivers are indirectly affecting the timing since they are executed on the target platform additionally to the application. Important components with focus on direct/indirect time consumption are:

- processing element: functional unit, pipeline, cache, register, counter.
- peripherals: memory, bus scheduling, interrupts, sensor and actuator components.
- indirect time consummation: task switch time, interrupt latency, error detection, scheduler, communication protocol.

Ideally, all these effects should be considered to calculate the timing behavior of the system. In terms of automotive distributed systems, a focus of interest is the communication timing that strongly depends on the communication architecture and the bus scheduling. Also important are sensor (actuator) components since they provide (require) data at dedicated time instants. This also applies for algorithms that consume execution time on a processing element.

10.5.2 Determination and Integration of Timing Behavior

Since a lot of components are responsible for the timing behavior of the entire system, the main question is how to incorporate the timing into the simulation model. There are basically two approaches possible if using SystemC as modeling language:

- Timing is integrated by instrumentation of the software application code with timing values using the SystemC timing statement wait(T), whereas T is a dedicated time value, i.e. the execution time of a piece of software or the execution time of a communication. The main task of this approach is the determination of the execution time by itself and the placement of the timing statement into the source code.
- Timing is incorporated by integration explicit models of the components that are responsible for the time consumed by software. Such models do reproduce the (parallel) behavior of hardware components due to the delta-cycle simulation [SC05] of the SystemC simulation kernel. The main task is to integrate and connect the IP model components with the entire system. Automatic model refinement supports this task.

The main advantage of the code instrumentation approach is a very fast simulation time since no complex hardware model components are required. On the other hand, accurate timing instrumentation requires a detailed code analysis. Considering the abstract BSW modeling discussed in the second subsection of Sect. 10.4.2, the timing instrumentation is the method that has to be applied to integrate timing information into the system model.

Determination of Application Timing Behavior for Code Instrumentation. Determining the execution time of application software may be a difficult task

if accurate timing information should be obtained. There are several methods to determine timing behavior for applications, but most of them are not very accurate since it is only approximately estimated. This is true for running the application on a dedicated processing unit, i.e. the target processor or on an instruction set simulator. A coarse mean value can be determined and annotated at the entry or exit point of a function. Although this is very inaccurate, a simulation based on end-to-end timing can give a first rough impression of the real time behavior of a running system. In contrast, also WCET/BCET (**W**orst **C**ase **E**xecution **T**ime/**B**est **C**ase **E**xecution **T**ime/) analysis may be used to determine end-to-end timing. But this approach is often too pessimistic/optimistic and does not consider dynamic aspects of execution like branch prediction or caches. The accuracy also depends on the granularity of the analysis. For instance, [KKW+06] presents an approach that instruments timing by an application profiling tool [KFK+05]. The analysis bases on source code and neither considers compiler aspects (optimization) nor specific architecture aspects (register, pipeline). The WCET/BCET-approaches have in common to use a static timing estimation to represent the actual runtime behavior during simulation. A more accurate way is the combination of static timing analysis during compile time and dynamic timing simulation during runtime. This approach analyses the cross-compiled binary code for a dedicated target processor and determines the static timing behavior using static processor models. The timing information is back-annotated into the source code at the boundary of a basic block. Additionally, data dependencies are considered during run-time of the simulation. This may cause the invocation of correction code depending on a simultaneous running model for branch prediction and caches as well. They determine at runtime whether or not a cache hit or cache miss occurs and whether or not the branch prediction fails. A detailed description of this approach is found in [SBVR08].

Determination of Application Timing Behavior by Integration of Processing Component. SystemC allows modeling and integration of processing element units as well. In particular, processing element models are offered by industry as IP components (e.g. IBM PowerPC [Ber05]) or delivered within ESL tools (e.g. CoWare [CoWa] Platform Architect, Mentor [MG] System Architect, Synopsys [Syn] DesignWare). The behavior of the target processing unit is often implemented by an ISS (**I**nstruction **S**et **S**imulator) [NBS+02] but also by complex processor models that includes the complete processor units (function units, pipelines, cache, register, counter, etc.). An ISS abstracts from a processor by simulating the instruction set. Different kind of implementations are possible: interpreting ISS and compiling ISS which is the faster solution in general. The compiling ISS may be static compiling or just-in-time compiling. The integration into the entire SystemC model may be difficult,

particularly, if the ISS has to be wrapped in SystemC. The application code runs as binary code in the instruction set simulator. The accuracy of execution time strongly depends on the accuracy of the processing element model. For instance, a register-transfer-level model is cycle accurate but requires much simulation time.

Determination of Communication Timing by Integration of Communication Components. Expressing the timing behavior of a communication model is more easy. Since the timing depends on the data transfer rate and the amount of data, an association with timing values is possible. In particular, the communication scheduling is also important. It contains the information when a communication may access the communication medium. Meanwhile, communication controller models are available and delivered by industry (e.g. Freescale FlexRay Executable Reference Model [BFA05] in SystemC). The integration process of communication model components is supported by refinement strategies that allow (semi-)automatic refinement from CP- or PV- to CC-level. This is supported by well-defined communication interfaces based on SystemC TLM 2.0 [Mon07] and the principle of separating communication and computation.

10.6 Application Example

This section presents a use case for exploration of the potential of a given system based on the aforementioned methodology. First, a traffic sign analysis system is developed using the AUTOSAR design methodology. The system is mapped onto SystemC CP level for first exploration steps. Afterwards, the system is mapped onto an already existing target platform that runs legacy code. The existing platform consists of five ECUs connected via FlexRay. Different configurations are evaluated for the extended system. The use case is kept quite simple and the results are easy to comprehend. However, a more complex scenario is more difficult to understand for humans. Hence, simulation heavily simplifies the evaluation.

10.6.1 Traffic Sign Analysis

The traffic sign analysis system consists of five components: an actuator component camera, a sensor component display for result visualization, and three software components. Each software component runs an algorithm: image decoder for data compression, traffic sign recognition, and traffic sign classification. An additionally control value is the current car speed that controls the data rate of the camera since a higher speed requires a higher data rate for recognition. The software component including their ports and communications are shown in Fig. 10.14.

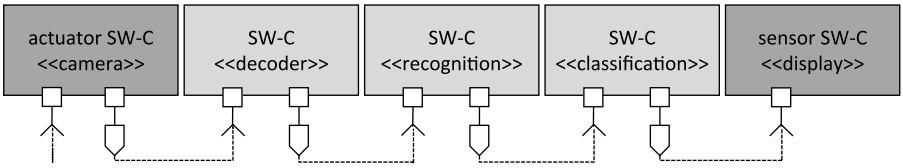

Figure 10.14. Virtual Functional Bus view of the traffic sign analysis system.

An early evaluation of the system becomes important since there are a lot of unknown aspects especially regarding the data transfer. It is influenced by the data transfer rate (i.e. how many pictures per second are analyzed), the camera type (picture resolution, color depth), the image compression algorithm (Huffman, Lempel–Ziv–Welch) and of course by the driving situation that is responsible for the number of traffic signs that are detected within a certain timing interval and also for the size of the compressed image since data compression is individual for each image. The designer has to decide the type of camera, the compression algorithm and the distribution of the software components. To allow a proper calculation, the resolution should not stay below a certain value. The data rate has also to guarantee that no traffic sign is missed.

10.6.2 VFB Transformation onto CP Level

The first evaluation step is the simulation at SystemC CP level. CP evaluation allows a functional verification of the system, e.g. the verification of the algorithms. Furthermore, the simulation gives a first view onto the communication traffic. The timing of a communication is given by the data transfer rate. With regard to the FlexRay that allows 10 MBit/s, an effective data transfer rate of 5 MBit/s has been chosen for data transfer between decoder, recognition, classification and display. For data transfer between camera and decoder a higher rate (50 MBit/s) has been chosen since it is already unrealistic to send this data via FlexRay. Figure 10.15 shows some results of the simulation. The x-axis represents the sending time of a message while the y-axis represents the sending duration (i.e. the latency) of this message. Three types of messages are drawn. The sending of the decoded image is periodic but has different latencies (depending on the data size of the decoded image). The result of the recognition and classification only occurs if traffic sign is recognized. As a result, the timing of the communications can be evaluated to map them onto the bus. The sending frequency for the decoded data is approximately one image each 50 ms while the sending of an image takes between 14 ms and 23 ms.

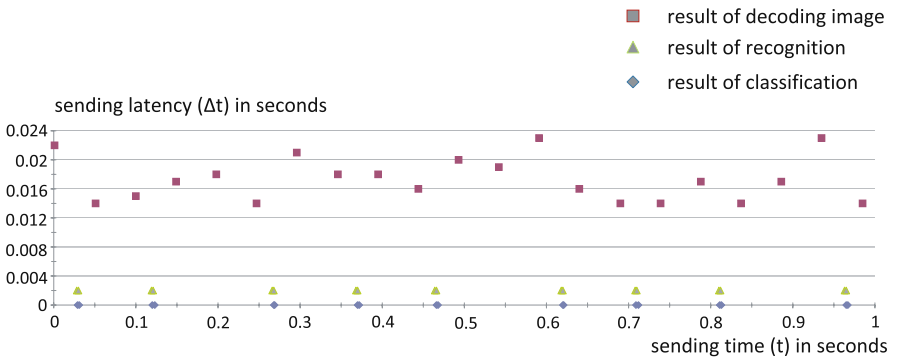

Figure 10.15. Result of CP simulation.

10.6.3 ECU Mapping and Transformation onto PVT Level

As mentioned before, the target architecture is a system already existing that consists of five ECUs connected by FlexRay. Additionally, a CAN network is connected by a gateway to obtain information from the engine control systems. The following decisions have been made: A new ECU has been added to the FlexRay network. This ECU includes the camera and also a processing element that runs the decoder algorithm. The display software component is mapped to the ECU that is responsible for all display functions. The recognition and classification software components were distributed to two additional ECUs. The mapping is illustrated in Fig. 10.16, the new bus communications are drawn by dotted lines. Assuming that the communication is the bottleneck of the system the communication is refined first. The FlexRay bus is integrated by a cycle callable transaction level model. Since the traffic sign analysis application is not safety relevant, the communications may also be mapped to the dynamic segment. The simulation environment allows for calibrating the application parameters. Based on the results of the CP simulation, the communication is distributed as follows: the decoded image data is mapped to the static segment of the FlexRay bus. This is because this data is continuously transmitted with only different data length. On the other hand the recognition and classification results are mapped to the dynamic segment of the FlexRay bus since they only occur sporadically.

The configuration of the static part of the FlexRay bus now depends on the existing communication data and the new image data. Since the maximum length of one communication cycle is limited by the existing communication that uses the static frame, the static segments has to be configured so that the latencies for both, the image data and the existing data, hold. The simulation helps to proof the latency of image data that is not fixed within the static part since the image size of a picture varies. Figure 10.17 shows two configura-

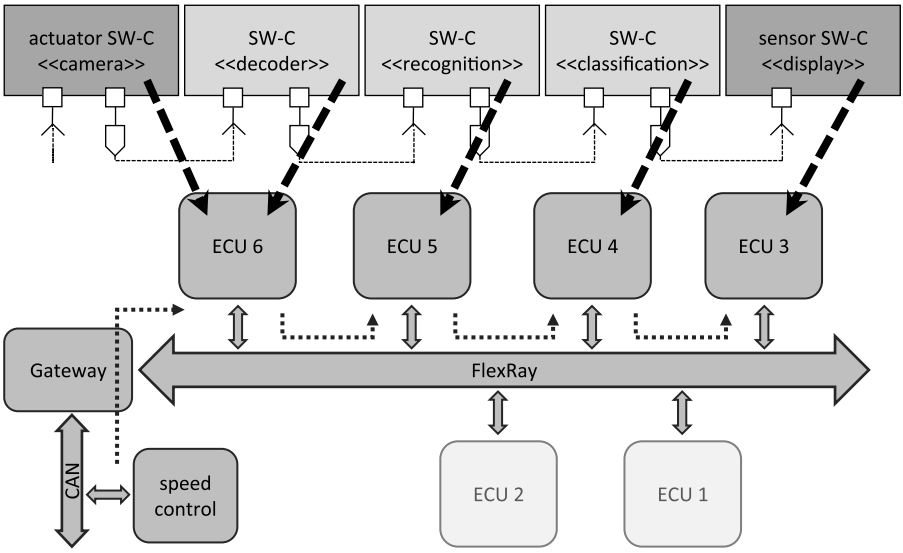

Figure 10.16. Mapping of software components onto the target architecture.

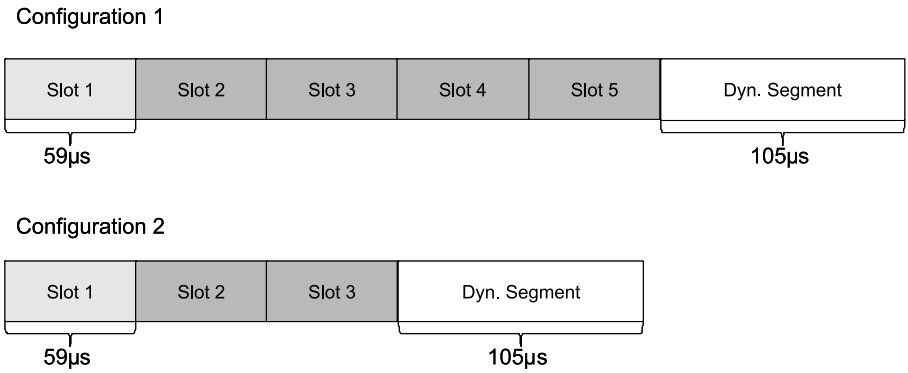

Figure 10.17. Different FlexRay configurations for two simulation scenarios.

tions of the FlexRay bus: Configuration one consumes the maximum possible length for the communication cycle and reserves four static slots (Slot 2–5) for the decoder within the static segment. Configuration two has a shorter communication cycle with only two static slots (Slot 2–3) for the decoder.

As a result, the different latencies for the decoded image are shown in Fig. 10.18. Also the latencies of the recognition and the classification result within the dynamic segment are changing since the communication cycle is

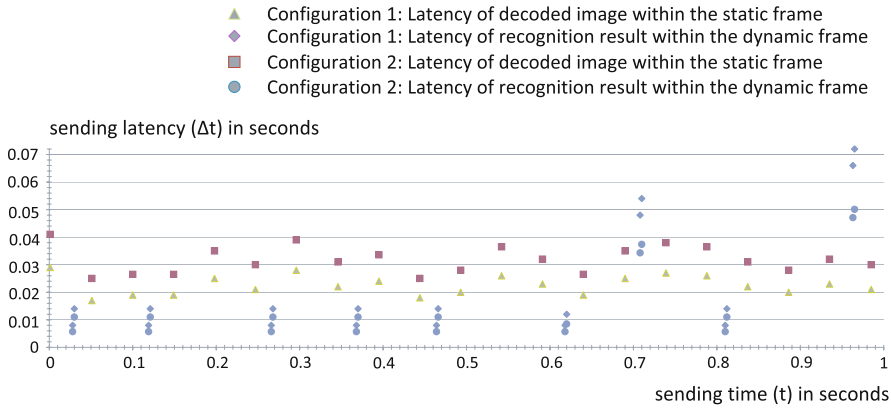

Figure 10.18. Result of PVT simulation: latencies within the static and dynamic segment.

of configuration two is shorter then configuration one. They are also heavily influenced by the existing communication traffic of the dynamic segment that has a higher priority for sending. Figure 10.18 shows the latencies for the recognition as well. The latency of the decoded image within the static segment is between 17 ms and 28 ms for configuration one while configuration two takes between 25 ms and 41 ms. On the other hand, the maximum latency of the recognition result within the dynamic segment is much lower (50 ms) for configuration two than for configuration one which takes 72 ms.

10.7 Conclusions

An early system evaluation and verification are the key to raise the quality of distributed electronic systems and to shorten development time and costs. Since virtual prototyping has been established within the SoC industry, also the automotive industry in terms of electronic development, and especially the software development, can benefit from this methodology. The applicability and the procedural method were shown in this article taking into account the AUTOSAR standard (release 2.1). Concerning hardware-dependent software, the goal of AUTOSAR is to hide the hardware specific features behind a standardized middleware layer to allow hardware independent implementation of software applications. This fact can be utilized not only for abstraction of hardware dependent software but also for easy integration into a virtual prototype to verify the system in an early design phase. At the time of going to press, the AUTOSAR has gone into phase 2, although, the introduced AUTOSAR concept will remain in the future. The introduced work is also just the beginning of a lot of ongoing work and hopefully this article encourages the reader for further research.

Acknowledgments

The authors would like to thank André Hergenhan and Gökhan Tabanoglu for their contribution to this work.

References

[AS] AUTOSAR. www.autosar.org

[AS06a] AUTOSAR. *Applying Simulink to AUTOSAR*, specification version 1.0.1 edition, June 2006.

[AS06b] AUTOSAR. *Specification of Interaction with Behavior Models*, specification version 1.0.1 edition, June 2006.

[AS06c] AUTOSAR. *UML Profile for AUTOSAR*, specification version 1.0.1 edition, June 2006.

[Ber05] R. Bergamaschi. Transaction-level models for PowerPC and CoreConnect. 11th European SystemC Users Group Meeting, 2005.

[BFA05] M. Baumeister, P. Fuhrmann, and F. Armbruster. Taking concept models from standardization to silicon. *Hanser Automotive Electronics + Systems*, FlexRay Special Edition, 2005.

[CAN] CAN. www.can.bosch.com

[CoWa] CoWare. www.coware.com

[DBR04] A. Donlin, A. Braun, and A. Rose. SystemC for the design and modeling of programmable systems. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 811–820. Springer, Berlin, 2004.

[Don04] A. Donlin. Transaction level modeling: flows and use models. In Alex Orailoglu, Pai H. Chou, Petru Eles, and Axel Jantsch, editors, *CODES+ISSS 2004*, pages 75–80. Assoc. Comput. Mach., New York, 2004.

[dS] dSPACE. www.dspace.com

[EA] Enterprise Architect. www.sparxsystems.com

[ET] ETAS. www.etas.com

[FlRa] FlexRay. www.flexray.com

[GLMS02] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic, Dordrecht, 2002.

[Hei04] H. Heinecke. Automotive open system architecture—an industry-wide initiative to manage the complexity of emerging automotive

e/e-architectures. In *Convergence International Congress and Exposition on Transportation Electronic*, Detroit, MI, USA, 2004.

[KBR05] M. Krause, O. Bringmann, and W. Rosenstiel. Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems. *Des. Autom. Embed. Syst.*, 10(4):229–251, 2005.

[KFK+05] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained application source code profiling for ASIP design. In William H. Joyner Jr., Grant Martin, and Andrew B. Kahng, editors, *DAC 2005*, pages 329–334. Assoc. Comput. Mach., New York, 2005.

[KKW+06] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. In Georges G. E. Gielen, editor, *DATE 2006*, pages 468–473. European Design and Automation Association, Leuven, 2006.

[LIN] LIN. www.lin-subbus.org

[MG] Mentor Graphics. www.mentor.com

[Mon07] M. Montoreano. Transaction Level Modeling using OSCI TLM 2.0. Technical report, Synopsys, 2007.

[MRR03] W. Mueller, W. Rosenstiel, and J. Ruf, editors. *SystemC Methodologies and Applications*. Kluwer Academic, Dordrecht, 2003.

[NBS+02] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th Design Automation Conference (DAC 2002)*, pages 22–27. Assoc. Comput. Mach., New York, 2002.

[OSEK05] OSEK/VDX. *OSEK/VDX Operating Systems*, version 2.2.3 edition, February 2005.

[SBVR08] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High performance timing simulation of embedded software. In *Proceedings of the 45th Design Automation Conference, DAC 2008*, Anaheim, CA, USA, June 8–13, 2008. Assoc. Comput. Mach., New York, 2008.

[SC05] OSCI. *SystemC Language Reference Manual*, draft standard edition, 2005.

[Syn] Synopsys. www.synopsys.com

[TLM] Transaction Level Modeling Working Group. www.systemc.org

[VI] Vector Informatik. www.vector-informatik.com

[VSBR06] A. Viehl, T. Schönwald, O. Bringmann, and W. Rosenstiel. Formal performance analysis and simulation of UML/sysML models for ESL design. In Georges G. E. Gielen, editor, *DATE 2006*, pages 242–247. European Design and Automation Association, Leuven, 2006.

Index

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,