Real-Time Operating System Modelling and Simulation Using SystemC

Ke Yu

Submitted for the degree of Doctor of Philosophy

Department of Computer Science

THE UNIVERSITY *of York*

June 2010

Abstract

Increasing system complexity and stringent time-to-market pressure bring challenges to the design productivity of real-time embedded systems. Various System-Level Design (SLD), System-Level Design Languages (SLDL) and Transaction-Level Modelling (TLM) approaches have been proposed as enabling tools for real-time embedded system specification, simulation, implementation and verification. SLDL-based Real-Time Operating System (RTOS) modelling and simulation are key methods to understand dynamic scheduling and timing issues in real-time software behavioural simulation during SLD. However, current SLDL-based RTOS simulation approaches do not support real-time software simulation adequately in terms of both functionality and accuracy, e.g., simplistic RTOS functionality or annotation-dependent software time advance.

This thesis is concerned with SystemC-based behavioural modelling and simulation of real-time embedded software, focusing upon RTOSs. The RTOS-centric simulation approach can support flexible, fast and accurate real-time software timing and functional simulation. They can help software designers to undertake real-time software prototyping at early design phases.

The contributions in this thesis are fourfold.

Firstly, we propose a mixed timing real-time software modelling and simulation approach with various timing related techniques, which are suitable for early software modelling and simulation. We show that this approach not only avoids the accuracy drawback in some existing methods but also maintains a high simulation performance.

Secondly, we propose a Live CPU Model to assist software behavioural timing modelling and simulation. It supports interruptible and accurate software timing simulation in SystemC and extends modelling capability of the mixed timing approach for HW/SW interactions.

Thirdly, we propose a RTOS-centric real-time embedded software simulation model. It provides a systematic approach for building modular software (including both application tasks and RTOS) simulation models in SystemC. It flexibly supports mixed timing application task models. The functions and timing overheads of the RTOS model are carefully designed and considered. We show that the RTOS-centric model is both convenient and accurate for real-time software simulation.

Fourthly, we integrate TLM communication interfaces in the software models, which extend the proposed RTOS-centric software simulation model for SW/HW inter-module TLM communication modelling.

As a whole, this thesis focuses on RTOS and real-time software modelling and simulation in the context of SystemC-based SLD and provides guidance to software developers about how to utilise this approach in their real-time software development. The various aspects of research work in this thesis constitute an integrated software Processing Element (PE) model, interoperable with existing TLM hardware and communication modelling.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Program Interface
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
AT	Approximately-Timed
BCET	Best-Case Execution Time
BIOS	Basic I/O System
BSP	Board Support Package
CA	Cycle-Accurate
CP	Communicating Process
CP+T	Communicating Process with Time
CPU	Central Processing Unit
DMA	Direct Memory Access
DPS	Dynamic-Priority Scheduling
DSE	Design Space Exploration
DSP	Digital Signal Processor
ECB	Event Control Block
EDA	Electronic Design Automation
EDF	Earliest Deadline First
ESL	Electronic System Level
FIFO	First-In-First-Out
FPGA	Field-Programmable Gate Array
FPS	Fixed-Priority Scheduling
GPP	General-purpose Programmable Processor

HW	Hardware
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HdS	Hardware-dependent Software
I/O	Input/Output
IC	Integrated Circuit
IMC	Interface Method Call
IP	Intellectual Property
IPC	Inter-Process Communication
IPCP	Immediate Priority Ceiling Protocol
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISCS	Instruction Set Compiled Simulation
ISR	Interrupt Service Routine
ISS	Instruction Set Simulation
ITRS	International Technology Roadmap for Semiconductors
LT	Loosely-Timed
MMU	Memory Management Unit
NRE	Non-Recurring Engineering
NRT	Non-Real-Time
OCP-IP	Open Core Protocol International Partnership
OSCI	Open SystemC Initiative
OS	Operating System
PCB	Printed Circuit Board
PCP	Priority Ceiling Protocol
PE	Processing Element
PIP	Priority Inheritance Protocol
POSIX	Portable Operating System Interface
PV	Programmers View
PVT	Programmers View Timed
RM	Rate Monotonic
ROM	Read Only Memory

ROM	Result Oriented Modelling
RR	Round-Robin
RT-CORBA	Real-Time Common Object Request Broker Architecture
RTES	Real-Time Embedded System
RTL	Register-Transfer Level
RTOS	Real-Time Operating System
RTS	Real-Time System
RTSJ	Real-Time Specification for Java
RTX	Real Time eXecutive
SHaRK	Soft Hard Real-time Kernel
SW	Software
SLDL	System-Level Design Language
SoC	Systems-on-Chip
TCB	Task Control Block
TLM	Transaction-Level Modelling
UML	Unified Modelling Language
VHDL	Very-high-speed integrated circuit Hardware Description Language
WCET	Worst-Case Execution Time
µITRON	micro Industrial The Real-time Operating system Nucleus

Acknowledgements

I am most grateful to my supervisor Dr. Neil Audsley for his constant and valuable support and guidance during my PhD study in the University of York.

I would also like to thank my assessors Professor Andy Wellings and Dr. Leandro Soares Indrusiak for their advice and help in my research.

I give all my love to my parents Yu Shiliang and Song Yipu for their endless love to me. This PhD thesis is also my sincere gift to them.

I am full of gratitude to Ms. Zhang Jing. She gave invaluable spiritual support to me during the bittersweet PhD years.

I would like to express my thanks to all colleagues and friends in Real-Time Systems Research Group. In particular, I thank Dr. Chang Yang, Dr. Shi Zheng, Dr. Gao Rui, Dr. Zhang Fengxiang, Dr. Kim Min Seong, Lin Shiyao, and Mrs Sue Helliwell for their help to me and experience shared with me. I also thank Qian Jun, Shen Jie, Yao Yining, Dr. Liu Yang, and Dr. Chen Jingxin for our friendship and cheerful lives in UK.

Declaration

The research work presented in this thesis was independently and originally undertaken by me between October 2005 and June 2010 with advice from my supervisor Dr. Neil Audsley. Three conference papers have been published:

K. Yu and N. Audsley, "A Mixed Timing System-level Embedded Software Modelling and Simulation Approach," in 6th International Conference on Embedded Software and Systems 2009, (ICESS '09), 2009. [13] This paper received the best paper award in the conference.

K. Yu and N. Audsley, "A Generic and Accurate RTOS-centric Embedded System Modelling and Simulation Framework," in 5th UK Embedded Forum 2009 (UKEF '09), 2009. [14]

K. Yu and N. Audsley, "Combining Behavioural Real-time Software Modelling with the OSCI TLM-2.0 Communication Standard," in 7th International Conference on Embedded Software and Systems 2010, (ICESS '10), 2010. [15]

Certain chapters of this thesis are based on above papers as follows:

Chapter 3 is based on [13] and [15].

Chapter 4 is based on [14].

Chapter 5 is based on [15].

Chapter 1

Introduction

1.1 General Background

No matter whether or not you are aware of the networked printer in your office, the electronic stability program in your car or the portable media player in the palm of your hand, over the past decades embedded systems have reshaped our everyday work, life and play. Embedded systems are special-purpose computer-based information processing systems performing some pre-defined tasks and often built into enclosing products [16]. They are widely integrated into various product categories, such as transportation vehicles, telecommunication devices, industrial equipment, home appliances, etc. It is estimated that embedded systems consume more than 99% of the manufactured processors in the world [17]. Besides these invisible embedded systems, consumer electronics (e.g., handheld computers, mobile internet devices, and smart phones) can be also seen as self-contained embedded systems in terms of their similar hardware (HW) components. Embedded systems are usually designed with resource-constrained hardware and low-extensible software (SW), and are optimised to work with specific requirements for dedicated applications. These characteristics make embedded systems distinct from general-purpose computer systems, for instance, personal computers, work stations and servers.

A special category of embedded systems is classified as the real-time embedded system, which can be distinguished by its requirement to respond to external environment in real time. The term “real-time” leads our attention to Real-Time Systems (RTSs), which usually occur in company with embedded systems. There are various interpretations of what a real-time system is, however “physical interactions with the real world" and "timing requirements of these interactions" are its two essential characteristics [17]. A RTS receives physical events from the real-world environment. These events are then processed inside the RTS and appropriate actions finally respond. Timing requirements mean that the corresponding output must be generated from the input within a finite and specified timing bound, giving the deterministic timing behaviour. The correctness of a RTS depends not only on the computation result, but also on the time when the result is produced. "Real-time" does not mean "as fast as possible", but emphasises "on time". Neither a too late output nor a too early output is correct. The vast majority of embedded systems have real-time requirements, and most real-time systems are embedded in products. At their intersection are Real-Time Embedded Systems (RTES). The Operating System (OS) used in a RTES is usually a Real-Time Operating System (RTOS), which supports the construction of RTSs [16]. RTESs and RTOSs are the general context for this thesis.

From the perspective of system design, an embedded system is constructed from various hardware and software components. As illustrated in Figure 1-1, they can be classified into four reference layers [18]. The architecture of an embedded system represents an abstraction model including all embedded components. It introduces relationships between abstract hardware and software elements without implementation details.

All embedded systems have a hardware layer, which contains electronics components and circuits located on a Printed Circuit Board (PCB) or on an Integrated

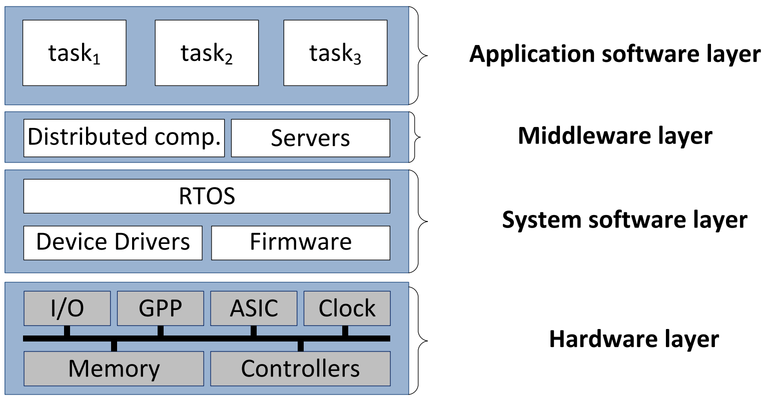


Figure 1-1. Typical layers of an embedded system

Circuit (IC). Although some time-critical or power-hungry portions of a system can be implemented with customised application-specific hardware (e.g., Application-Specific Integrated Circuits (ASICs), Field-Programmable Gate Arrays (FPGAs)), most embedded systems mainly function through software running on embedded General-purpose Programmable Processors (GPPs) (e.g., Central Processing Units (CPUs) or Digital Signal Processors (DSPs)). With the development of the microelectronics industry, Systems-on-Chips (SoCs) have emerged as the state-of-the-art implementation of embedded systems. A SoC is an integrated circuit combining multiple GPPs, customised cores, memories, peripheral interfaces, as well as communication fabric, all on a single silicon chip, which provides substantial computation capability for handling complex concurrent real-world events. Comparing the different embedded hardware solutions as indicated above, application-specific hardware offers high computing performance and low power consumption at the expense of limited programming flexibility, whilst GPPs offer higher design flexibility and lower Non-Recurring Engineering (NRE) costs, but with a relatively low computing capability [16].

In general, embedded software can be grouped into three layers: the application software layer, the middleware layer, and the system software layer. The application functions of an embedded system consist of a task or a set of tasks.

Middleware is an optional layer under application software but on top of system software. Middleware provides general services for applications, such as flexible scheduling [19], distributed computing (e.g., Real-Time Common Object Request Broker Architecture (RT-CORBA) [20]), and Java application environment (e.g., Real-Time Specification for Java (RTSJ) [21]). Using middleware technologies has strengths to reduce complexity of applications, simplify migration of applications, and ensure correct implementation of reusable functions.

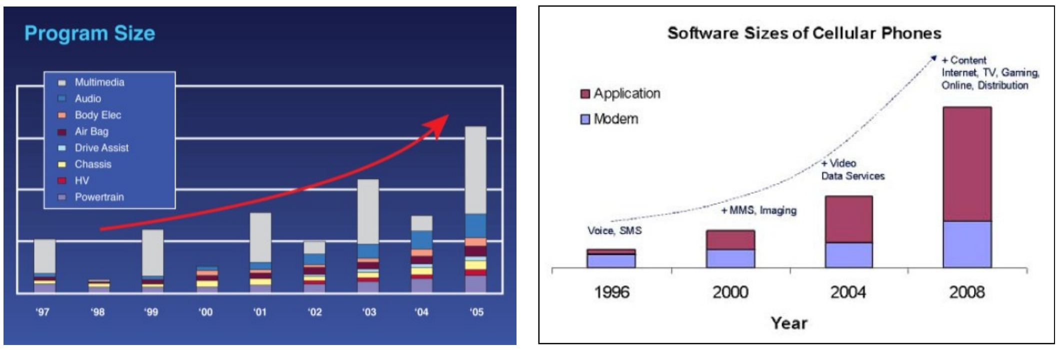
The system software layer is sandwiched between upper-level software and bottom-layer hardware. It usually contains device drivers, boot firmware and RTOS, which closely interact with the hardware platform. This kind of software is also called Hardware-dependent Software (HdS) [22]. Device drivers, e.g., a Board Support Package (BSP) for a given platform, are the interface between any software and underlying hardware. They are the software libraries that take charge of initialising hardware and managing direct access to hardware for higher layers of software [18]. Boot firmware, e.g., the Basic I/O System (BIOS), carries out the initial self-test process for an embedded system and initiates the RTOS. It is usually stored in the Read-Only Memory (ROM).

Regarding the RTOS, it is unnecessary and cost-inefficient to introduce a RTOS in some small embedded devices, where an infinite loop program with the polling policy for Input/Output (I/O) events may work well [23]. However, in order to satisfy the complex functional requirements and timing constraints for concurrent real-time software execution, the RTOS has become an essential component in most embedded systems. Here, concurrent real-time software execution refers to situations that, under the control of a RTOS, multiple tasks either share a uniprocessor in interleaving steps or execute on multiple processors in parallel. A RTOS is needed to provide convenient interfaces and comprehensive control mechanisms to let applications utilise and share hardware and software resources effectively and reliably. The kernel is the core element of a RTOS and contains the most essential functions. In most kernels, there is the notion of task priority, dynamic pre-emptive scheduling services, synchronisation primitives, timing services, and interrupt handling services [24] [25] [26]. Other OS features such as memory management, file systems, device I/O etc. are often optional in a RTOS in order to maintain its compactness and scalability. As a central part of the real-time embedded software stack, a RTOS's own timing behaviour also needs to be predictable and computable. Designers must know some important RTOS timing properties, for example, the context switch time, Worst-Case Execution Times (WCETs) of system calls, the interrupt handling latency, and the maximum interrupts disabled time, etc. Hence, they can analyse and evaluate the real-time performance of the whole system.

The research in this thesis will investigate how to model RTOS kernel functional and timing behaviours in order to support high-level real-time software simulation in a uniprocessor system.

1.2 Challenges in Embedded System Design

In recent years, the complexity of embedded software has increased rapidly. According to the International Technology Roadmap for Semiconductors (ITRS) 2007 Edition (ITRS 2007), embedded software design has emerged as "*the most critical challenge of SoC productivity*" [4]. For many products of consumer electronics, the amount of software per product is thought to be double every two years [27]. The General Motor Information Systems CTO predicts that the average car, with one million lines of software codes in 1990, will run on one hundred million lines by 2010 [28]. Figure 1-2 shows growing trends of embedded software complexity in motor and mobile phone industries.



Automobile software size increase (Toyota)

Mobile phone software size increase (Infineon)

Figure 1-2. Embedded software size increases in industry (reprint [5] [10])

In addition to the overwhelming system complexity, the time-to-market pressure is another overriding priority in contemporary embedded systems development [10] [29]. If the projected delivery date is missed, it results not only in an increase of design costs but also a decrease of market share. This pressure is even tougher for embedded software design. Since in a traditional hardware-first design

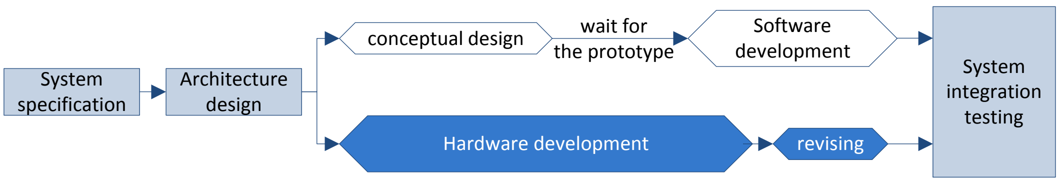


Figure 1-3. The hardware-first design process

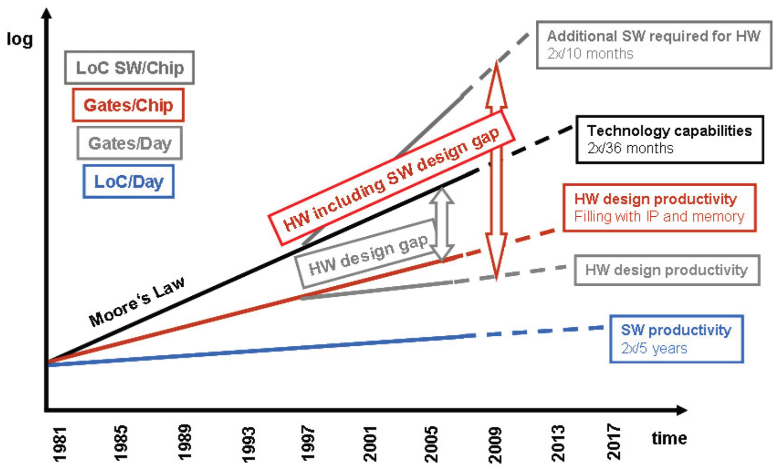


Figure 1-4. Gaps between the design complexity and productivity (reprint [4])

flow (see Figure 1-3), the software development cannot go through until the hardware prototype is available. This means that software designers often face imminent product delivery deadlines [30].

There is also a big gap between ever-growing semiconductor fabrication capability and the design productivity of embedded systems (including both HW and SW aspects) [31]. The ITRS 2007 presents a summary about hardware and software design gaps and Figure 1-4 is the pictorial illustration [4]. In Figure 1-4, regarding the HW design aspect, the cutting-edge embedded HW advancements and design methodologies, e.g., multi-core/processor components and Intellectual Property (IP) reuse, have somewhat narrowed the distance between HW design productivity and HW technology capabilities. Unfortunately, although enormous SW complexity has already been exacerbated, these HW advances further increase demand for HdS development. As what is shown in the figure, SW productivity is further behind the steeply increasing SW complexity. An industrial report even indicates that rapidly increasing software design efforts may exceed the cost of hardware development when IC technologies evolve from deep submicron-scale to nano-scale [29].

1.3 System-Level Design Methodologies

Motivated by the challenges outlined above, since the 1990s, System-Level Design (SLD), or so-called Electronic System-Level design (ESL), and corresponding System-Level Design Languages (SLDLs) have been developed as enabling tools for embedded system specification, simulation, implementation and verification [32].

In the view of Electronic Design Automation (EDA) industry, SLD is indicated at *"a new level of abstraction above the familiar register-transfer level"* [4]. This definition reflects a hardware-centric viewpoint. A more complete definition emphasises *"the concurrent hardware and software design interaction"* as a guiding concept in a SLD process [17], that is, the HW/SW codesign [33] philosophy is inherent in SLD methodologies.

1.3.1 Raising Abstraction Levels

Raising system abstraction to higher levels is a traditionally intuitive solution to cope with design complexity. In the area of digital electronic design, abstraction levels went from the transistor model in the 1970s, to the gate-level model in the 1980s, to the Register-Transfer Level (RTL) models in the 1990s, and latterly to the higher system-level models [17]. Higher-level abstractions focus on critical system-wide behaviour and ignore unnecessary low-level implementation details at early design times. System behaviours are represented by executable models. These models are continuously refined and evaluated through simulation and details are gradually added in the design process, which enables early and fast validation of the system [34]. The current RTL Hardware Description Languages (HDLs) (e.g., Verilog [35] and VHDL [36]) are believed too low and time-consuming to describe hardware at early development stages [37]. Furthermore, despite expressive features of RTL HDLs for hardware development, they fail to support description and validation of an entire system, including both hardware and embedded software, which is a key necessity in system-level design. Consequently, SLDLs (e.g., SystemC [38] and SpecC [39]) have been developed to support unified high-level HW/SW specification, modelling, simulation, verification and synthesis in recent years. In this thesis, SystemC is the research tool for software modelling and simulation.

1.3.2 Orthogonal Concepts in System-Level Design

SLD aims to separate orthogonal design concerns in order to allow independent and swift exploration of alternative solutions [40]. At a specific design stage, different design aspects may not require the same level of abstraction. Consequently, separating design issues and building independent abstract models not only save design time, but also achieve better simulation performance when various models are simulated together. The following two classical separation ideas are most often referred to in SLD:

Functionality versus *architecture* [41] (also called *Application* and *Platform Implementation* [17]): According to the definitions put forward in [40] [42], the functionality aspect refers to what basic tasks a system is supposed to do, i.e., specification; whereas the architecture aspect refers to how to do these tasks by configuring resources, i.e., implementation. In SLD, there are often a series of mapping and refinement steps between a functional specification model and the final implementation architecture. The motivation of this orthogonal separation is for design reuse and flexibility. Supposing the functionality is defined in a separate specification model, designers can explore many possible architecture implementations with different performance and cost attributes. As well, if several basic HW or SW architecture implementations can construct some generic clusters, i.e., components and platforms, then they could be reused for a variety of applications [40].

Computation versus *communication* [7]: The central idea is to develop computation and communication independently by hiding their details from each other. Computation components, either hardware or software, are modelled as modules (i.e., Processing Elements (PEs)) that contain a set of concurrent processes. Communication components such as buses or on-chip networks are modelled based on basic abstract elements, e.g., ports, channels, and interfaces. Computation modules communicate by transferring data transactions through these communication infrastructures. This separation introduces an important and widely accepted SLD approach Transaction-Level Modelling (TLM) [3]. TLM methods often define a number of intermediate computation and communication models for simulation in a design flow. At each level, models include necessary functional and timing details for a specific design stage. An important TLM research topic is the trade-off between simulation performance and the accuracy of different models. The research in this thesis is also concerned with this trade-off.

1.3.3 System-Level Design Flows

System-level design flow is a process containing multiple design steps, during which an embedded system is gradually transformed from a conceptual specification to a final product. At each design step, designers successively build, simulate and refine various abstract models in order to validate system properties early before detailed implementation [43]. There is not a generally accepted "design flow" template. The starting and ending design points also vary in different SLD theories and practices. This is because a specific design process is largely dependent on its applying domains and contexts, e.g., re-using an existing platform may shorten the design flow. There are probably as many system-level design flows as there are researchers and projects. Nevertheless, we can observe that many research works [43] [44] [45] [46] [47] generally group design activities into three top-down phases with corresponding models: the system specification phase (specification models), the architecture exploration phase (architecture models), and the architecture implementation phase (implementation models). Figure 1-5 outlines a typical system-level design flow including above three phases. The research in [48] [49] presents a different view of system-level design flow which excludes the implementation phase. This viewpoint in fact reflects the status of current system-level design community that existing SLD methodologies are still not mature enough to effectively cover all phases from system specification to implementation.

At the system specification phase, the embedded system's planned functions and requirements are clarified and written in documents or models. Natural languages are used in documents, whilst some computer specification languages (e.g., Unified Modelling Language (UML) [50], MATLAB [51], SpecC [39], Rosetta

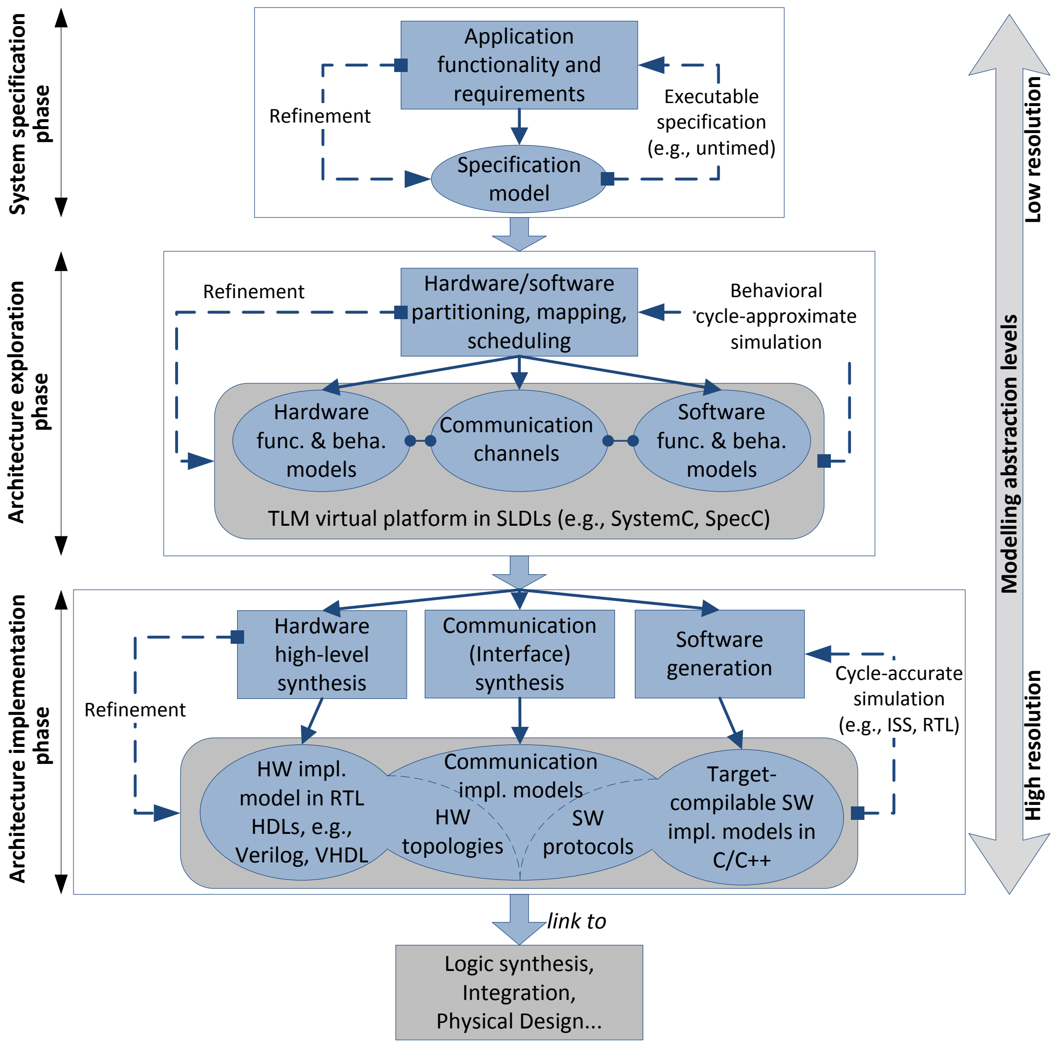

System specification phase
Architecture exploration phase
Architecture implementation phase
Application functionality and requirements
Refinement
Executable specification (e.g., untimed)
Specification model
Hardware/software partitioning, mapping, scheduling
Refinement
Behavioral cycle-approximate simulation
Hardware func. & beha. models
Communication channels
Software func. & beha. models
TLM virtual platform in SLDLs (e.g., SystemC, SpecC)
Hardware high-level synthesis
Communication (Interface) synthesis
Software generation
Refinement
Cycle-accurate simulation (e.g., ISS, RTL)
HW impl. model in RTL HDLs, e.g., Verilog, VHDL
Communication impl. models
HW topologies
SW protocols
Target-compilable SW impl. models in C/C++
link to
Logic synthesis, Integration, Physical Design...
Low resolution
Modelling abstraction levels
High resolution

Figure 1-5. A system-level design flow

[52]) can be also used to produce formal or executable models. These models can describe behaviour of a system and may become a vehicle for next-step system refinement.

The architecture exploration phase, so-called hardware/software partitioning and mapping phase, is concerned with how to distribute system functions between hardware and software, i.e., Design Space Exploration (DSE). This phase can be further divided into the pre-partitioning step, the partitioning step, and the post-partitioning step, according to a detailed design flow explanation in [32]. Usually, this design phase starts from a unified abstract TLM model, which comprises a set of PEs for computation and channels for communication. These PE models are explored to implement in either HW (i.e., application-specific hardware logics) or

SW (i.e., programs running on a GPP), and channel models are tried with various abstract communication topologies and protocols. These TLM models are successively refined, with timing information and implementation details added. Various alternatives are simulated in order to evaluate and analyse diverse system characteristics, e.g., functional correctness, scheduling decisions, real-time performance, power consumption, chip area, and communication bandwidth, etc. Once a system's functions have been partitioned and mapped onto some hardware and software elements, a golden architecture model [46] comes into being and the implementation step is ready to begin. This thesis studies RTOS and real-time software behavioural modelling and simulation, which can be seen as being after-partitioned TLM software PE computation research in the architecture exploration phase. Our research has some relevance to current SLD and TLM research, in terms of comparable abstract modelling styles, fast simulation performance, reasonable accuracy, and some interoperability with other system-level abstract hardware and communication models.

In the architecture implementation phase, previous architectural models are transformed into lower-level models in automated synthesis for final product implementation design and manufacturing. For the hardware aspect, the developing high-level synthesis (sometimes also referred to as Electronic System-Level synthesis, system synthesis, behavioural synthesis) technologies aim to synthesise HW models in the form of high-level languages (e.g., C, C++, SpecC, SystemC) into synthesisable RTL descriptions. RTL descriptions are input of the existing "RTL to Layout" design flow [32]. This automated high-level synthesis process connects system-level design with the current design flow in order to produce actual integrated circuits. Although there is a substantial body of research work in this domain, automatic high-level synthesis is still thought to be not mature [53] and has "*never gained industrial relevance*" [54]. In SLDL-based system-level design, communication synthesis (also known as interface synthesis) aims to map TLM channels or similar high-level interfaces to a set of synthesisable cycle-accurate software protocols and RTL descriptions of target communication topologies [55]. There are several approaches regarding bus-based communication synthesis [56] [57] and on-chip communication networks synthesis [58] [59].

More complete surveys on this topic can be found in [54] and [17]. In high-level software synthesis (namely target software generation), embedded software (including the applications, RTOS and other HdS) implementation models (i.e., C/C++ codes that are ready to be compiled into binaries for a target instruction set) can be generated from TLM software PE models written in SLDLs [60] [61]. Several approaches have investigated embedded software target code generation, in which SLDL functions or generic RTOS services in TLM models are mapped and translated to the Application Program Interface (API) of a specific RTOS [43] [62] [63] [64] [65].

1.4 System-Level Design Languages

The need for efficient and effective specification, modelling, simulation, verification and synthesis in SLD has led to many SLDLs. In general, SLDLs provide a collection of libraries of data types, modular components, and discrete-event kernels to model an entire HW/SW system and simulate dynamic system behaviour at a higher level of abstraction. Using SLDLs enhances system design productivity by representing a whole system in expressive programming models and presenting diverse traceable run-time information through simulation.

Inspired by the need to describe both HW and SW parts with a general programming language, C/C++ based design and specification languages (e.g., SystemC and SpecC) have been developed and used by the design community. It is attractive to extend C/C++ for hardware and communication design exploration in SLD, since they are already familiar to software designers. These C/C++ based SLDLs are equipped with built-in hardware description constructs such as signals, ports, clocks, explicit parallelisms and the structural hierarchy for system modelling.

1.4.1 SystemC

SystemC is the most commonly used C++ based SLDL. It has been in development by the association Open SystemC Initiative (OSCI) since 1999 [38]. In its early days, the initial SystemC versions 0.9 and 1.0 concentrated on describing hardware-centric RTL features with the goal to replace Verilog and VHDL as a new HDL, so as to realise high-level synthesis. From the version 2.0, its focus changed to high-level computation and communication modelling and became an effective SLDL. It was approved as an IEEE standard in 2006 [66] and is currently the de facto industry standard for ESL specification, modelling, simulation, verification and synthesis.

The syntax of SystemC is based on the standard C++ language. It is not a brand new language but a set of C++ libraries together with a discrete-event simulation kernel that is also built with C++. A mixture of software programs written with SystemC and C++ can be compiled by a standard C++ compiler (e.g., GCC or Visual C++) and linked with SystemC libraries in order to generate an executable simulation program.

A *module* (`SC_MODULE`), namely a class, is the basic SystemC language construct to describe an independent functional component. It contains a variety of elements to define behaviour and structure of a model, e.g., data variables, computation *processes*, communication *ports* and *interfaces*, etc. SystemC supports the hierarchical model structure, which means a parent module can include instantiations of other modules as member data. This characteristic is helpful to break down a large system into manageable sub-models. The main SystemC mechanisms for inter-module communications are *channels* (`sc_channel`), which can be either a simple *signal* (`sc_signal`) or a complex hierarchical structure such as the Advanced Microcontroller Bus Architecture (AMBA) bus [67]. The communication methods implemented by *channels* are named *interfaces*, which are abstract classes declaring pure virtual methods. A module accesses a channel through a port by calling interface methods. In this way, computation and communication can be explicitly separated and modelled in SystemC.

SystemC uses a discrete-event simulation kernel, which relies on a co-operative, so-called co-routine, execution model [68]. It does not support a priority assignment or pre-emption. Only one SystemC process can execute at a time. The executing process cannot be pre-empted or interrupted by either the kernel or another process. A process only yields control to the kernel by calling *wait-for-time* and *wait-for-event* functions at its own will. When two processes are ready at the same time in simulation, it is non-deterministic which process will be chosen to run by the simulation kernel. This particular characteristic is suitable for parallel hardware operations and outperforms a pre-emptive simulation kernel in terms of fast simulation speed because of less context switch overheads [69]. However, it is not applicable for concurrent real-time software simulation, which requires pre-emptive and deterministic scheduling services. This deficiency can be problematic when importing legacy real-time software into SystemC. Some research pessimistically abandoned real-time software simulation in SystemC [70]. Whereas, many researchers have presented various remedies on this problem to some extent, e.g., extending the SystemC language with process control constructs [71], revising the SystemC simulation kernel [69] [68], implementing RTOS functions on top of the SystemC library [72] [73]. This thesis presents a more complete solution in the last direction.

1.4.2 SpecC

SpecC is a system specification and description language that operates as an extension of standard C language [39]. The SpecC language and associated design methodologies were originally developed at the University of California Irvine beginning in the mid-1990s and continuing up to the present day. In contrast to SystemC, SpecC introduces new keywords to C language, so it needs a special SpecC Reference Compiler [74]. Many design concepts (e.g., separation of communication and computation) and language constructs (e.g., modular structure descriptions) of SpecC are either possessed or adopted in the development of SystemC. As well, both SpecC and SystemC can fulfil multiple level specification, verification and synthesis tasks in SLD and TLM. Their similarities and differences are introduced and compared in [44].

1.4.3 SystemVerilog

Arising from the semiconductor and electronic design industry, SystemVerilog is a hardware description and verification language based on extensions of Verilog [75]. In addition to features available in the classical Verilog, SystemVerilog provides new verification and object-oriented programming facilities, such as assertions, coverage, constrained random generation, build-in synchronisation primitives and classes. Although SystemVerilog offers both internal object-oriented software features and a direct programming interface to call external C functions, its scope is mostly constrained to hardware design, simulation and verification [76] [32].

1.5 Software Simulation in System-Level Design

In SLD, simulation approaches lie at the heart of many methodologies. Simulation techniques are traditional and useful tools for debugging, validation, and verification [32] [44] [77]. They are successively applied at each phase in the design flow. A set of simulation models is built to represent behaviours of various components or the whole system. By executing these simulation models, output values for given input patterns are generated and observed. The correctness and quality of output values are evaluated in order to ensure that specified requirements have been fulfilled in the models. These results can also help designers to explore and trade off different design alternatives through simulation experiments.

Today, most software simulation approaches in SLD can be classified into two categories: Instruction Set Simulation (ISS) and behavioural simulation. In this thesis, the real-time software modelling and simulation research falls into the latter category.

1.5.1 Instruction Set Software Simulation

In ISS, a clock cycle-accurate processor model runs on a host machine, which mimics the behaviour of a target processor by “executing” its instructions. The internal architecture of the target processor (e.g., general registers, status registers) alongside memory space (i.e., storing execution binaries for a target and local variables) are both modelled at the Instruction Set Architecture (ISA) level. Sometimes, peripheral models such as timers, interrupts, and I/O ports are also integrated into an ISS so that it can provide more complete features for software simulation.

Most commercial ISSs are based on the interpretation technique [77]. An ISS reads target instructions from its memory space and executes in an interpretive “Fetch-Decode-Dispatch-Execute” process in order to simulate behaviour of in-

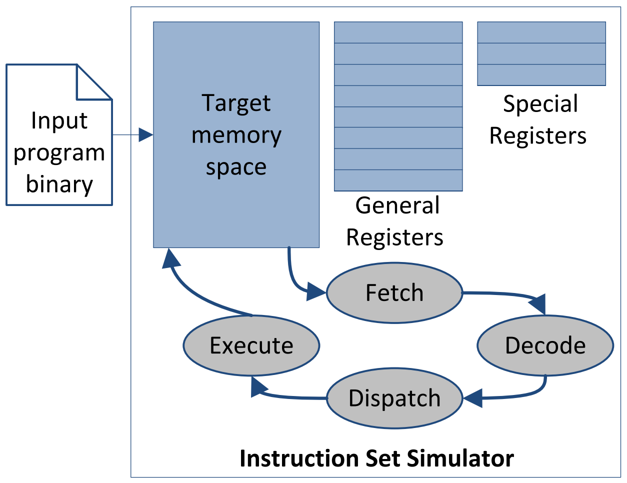


Figure 1-6. Interpretive instruction set software simulation

structions being executed on a target machine, as shown in Figure 1-6. The main advantages of ISS simulation are fine-grained functional and timing accuracy, so various ISS simulators are traditionally used by software programmers to debug cross-compiled target programs instead of using real hardware. And in system-level design, ISS simulators can be seen as references to evaluate other corresponding cycle-approximate simulators. However, simulation performance is a drawback of the ISS approach, because its interpretive simulation process incurs a large overhead. Typically, they run on the order of 100K cycles per second [78], which is not a satisfactory speed for simulating large amounts of software in system-level design [79]. Besides, an ISS simulator needs a detailed ISA-level processor simulation model, which may not be available at the desired high level of abstraction in early design stages.

The host compilation based ISS is an improved approach by addressing the performance disadvantage of traditional interpretive ISS methods [80]. The central idea of this technique is to translate target machine's instructions into host machine's at software compile time. This binary-to-binary translation avoids big run-time overheads of the interpretive process in simulation, hence resulting in a faster simulation speed. The host compilation ISS research in [80] reports a three orders of magnitude speedup compared to interpretive ISS. Unfortunately, there are also some deficiencies to this approach. This technique assumes that software does not change at run time, as a result it is not suited to self-modifying code [80]. Poor portability is another problem, because a compiled ISS is not applicable for processors with different instruction sets [77] [81]. The Instruction Set Compiled Simulation (ISCS) [81] technique combines the performance of a compilation-based approach with the flexibility of an interpretive ISS, by moving the decode step to compile time and carrying out various compile time optimisations. It claims a 70% simulation performance improvement compared with the best-known results in its domain. However, it still faces challenges in terms of both a long compilation time and a large memory usage [77]. In general, the simulation performance of ISS approaches is perceived as a bottleneck for a rapid design space exploration at the system level [79] [82].

1.5.2 Behavioural Software Simulation

In system-level design, there is always a need for fast and flexible software validation, which can be provided by behavioural software simulation. Its simulation performance is usually several orders of magnitude faster than the ISS approach, for example, one order speed-up in [83], three orders speed-up in [84], and three to five orders speed-up in [85]. Its modelling accuracy and speed are flexible in various approaches, which indeed depend on the specific modelling abstraction levels and techniques. In behavioural software simulation, high-level embedded software source code (e.g., in C/C++ or SLDL) is compiled for and natively executes on a host workstation or a PC. In many cases, behavioural software simulation is based on the support of a SLDL simulation framework. The target CPU hardware architecture model is not directly useful for native software execution, hence is often not modelled in a software PE model. This method is unlike the detailed processor model appeared in ISS simulation. Figure 1-7 shows the simulation mechanism of a typical discrete-event SLDL simulator, which includes three main steps, i.e., evaluation and schedule of a process, execution in zero-target time, and target simulation time advance.

From the perspective of abstract embedded processor and TLM communication modelling, Schirner summarises three major issues related to a fast system-level software simulation, i.e., timed native software execution, dynamic software

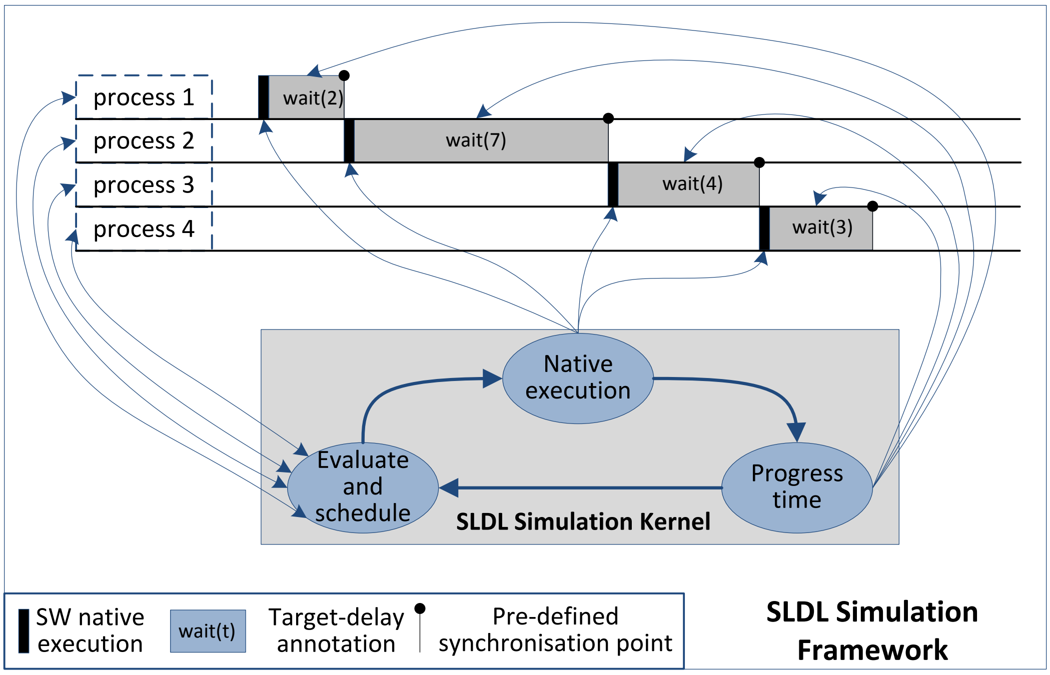


Figure 1-7. The SLDL-based behavioural software simulation

scheduling, and external TLM communication [79]. We will adapt them to reflect our software/RTOS-centric research perspective in the following section.

1.6 Research Objective and Contribution

This thesis focuses on modelling and simulating functional and timing behaviours of real-time embedded software including the RTOS. We conclude the most important issues as:

- *Timed software simulation*: this refers to timed modelling and simulating real-time software in the SLDL environment;
- *RTOS modelling*: this enlarged topic should not only provide real-time scheduling services but also support other typical RTOS services necessary for real-time software simulation;
- *Interrupt handling*: from a software simulation perspective, the Interrupt Request (IRQ) based HW/SW synchronisation [86] is the most essential external communication protocol.

1.6.1 Timed Software Simulation

As shown in Figure 1-7, in SLDL-based timed software simulation, embedded software (both applications and the RTOS) is organised (wrapped) into several concurrent processes in a SLDL simulation framework. These processes natively execute on the host under the supervision of a co-operative SLDL simulation kernel. Since the desired timing behaviour of target software execution cannot be directly represented in native software execution, estimated software execution costs (time delays) on the target are manually or automatically annotated to corresponding code segments of simulation processes. These time delays are executed by SLDL *wait(delay)* statements in order to suspend the calling process, pass control to the kernel, and advance the simulator clock. By this way, timing behaviour of real software execution on the target machine is simulated.

According to the above description, in this co-operative SLDL execution model, a number of *wait(delay)* statements are annotated into software processes when building the model. They in effect predefine synchronisation points between software processes and the SLDL kernel. Software processes can only yield the running status at these points at simulation runtime and the simulator time is progressed according to the annotated delays without an interrupt possibility. This annotation-dependent software time advance method makes it hard to model a pre-emptive real-time system. The intuitive but halfway solutions tackle this problem by using more *wait()* statements with fine-grained *delays* to advance SW time [87], or by inserting some imperative synchronisation points [3]. However, the timing accuracy is limitedly enhanced at the cost of large modelling (more annotation and synchronisation) and simulation (frequent simulation kernel context switch) overheads.

1.6.2 RTOS Modelling

A RTOS simulation model is a key point for dynamic scheduling and timing issues in behavioural real-time software simulation [72] [77]. This is because the RTOS's crucial role in embedded real-time software layers, in terms of task management, pre-emptive scheduling, inter-task communication and synchronisation, etc. Whereas, current SLDL simulation frameworks and related RTOS simulation models do not, in general, support RTOS simulation adequately. There exist some problems in this area, which affect the functional and timing accuracy of models, as well as their simulation performance.

For example, from the perspective of maximising flexibility of system-level design, designers may want to simulate multiple types of application models together. Current RTOS modelling research does not address this issue sufficiently and is incapable of integrating abstract task models (i.e., void or simple task functions with coarse-grained execution time estimates) and native-code task models (i.e., fully functional tasks with fine-grained delay annotations) in one simulator.

Besides, from the perspective of practical RTOS simulation, some RTOS models provide simplistic task management and limited synchronisation services, which are inadequate to imitate behaviour of a real multitasking RTOS.

Furthermore, the low timing accuracy is a common, yet critical, problem in some RTOS modelling approaches by lack of RTOS services' timing overhead modelling and proper time advance.

1.6.3 Interrupt Handling

As we mentioned before, the target processor, which executes software in the final implementation, is not usually modelled in SLDL-based behavioural software simulation. Because of the high abstraction level and the SLDL software simulation mechanism, multiple concurrent tasks together with a RTOS model can constitute a software PE model without the necessity of modelling low-level processor architecture. However, regarding timed HW/SW co-simulation, a software PE model should be able to handle hardware interrupts for HW/SW synchronisation. In terms of a real processor or a low-level processor model, the interrupt handling process is natural to implement because of their cycle-accurate time resolutions. However, the situation is complex when a "hardware" processor model is hidden in a high-level software behavioural simulation. From the sequential real-time software perspective, neither application tasks nor the RTOS can monitor asynchronous interrupt events (we are not talking about synchronous mechanisms such as polling) in a timely and real manner. What is more critical, it is not straightforward to interrupt a SLDL process by current SLDL kernels, since they do not support run-time process pre-emption or interruption. Consequently, it is essential to implement a HW/SW synchronisation method for SLDL-based software simulation, which behaves like an interrupt controller in a real CPU in order to monitor external events and interrupt the executing SLDL process. Besides, this mechanism should minimise the synchronisation frequency so as to reduce simulation time overhead, which is not yet achieved well in current approaches.

1.6.4 Research Hypothesis and Objectives

This thesis is motivated by current insufficient research regarding above three key issues in the domain of real-time software behavioural modelling and simulation. The research work in the thesis presents solutions to the three topics. Specifically, we aim to support SLDL-based interruptible software timing simulation with high simulation performance; we will propose a flexible and practical RTOS modelling and simulation approach that also has reasonable timing accuracy; we will support fully functional interrupt handling in high-level RTOS simulation as well.

The main goal of the research in this thesis is to support the central proposition that:

A SystemC mixed timing modelling and simulation approach can enable fast, flexible and accurate RTOS-based real-time embedded software behavioural modelling and simulation in system-level design.

To examine this hypothesis, this thesis focuses upon the investigation of timing issues in behavioural software modelling and simulation, and builds a generic RTOS model to support real-time embedded software simulation. Specifically, this thesis aims to:

1) Investigate timing issues in modelling and simulating real-time software (both applications and the RTOS) in a SystemC environment, which are closely relevant to a fast simulation performance, a flexible modelling and simulating capability and reasonable timing accuracy.
    a. Fast performance is a necessity of the proposed high-level behavioural software simulation. Simulation speed should be at the scale of several

orders of magnitude faster than traditional ISS simulations and is also better than some related behavioural software simulation methods.

   b. Flexibility is a desired benefit of software behavioural modelling and simulation for the sake of trade-off. The proposed approach can utilise varying modelling levels and degrees in different software models in terms of the functional accuracy, timing accuracy, observability of execution traces, and performance of simulation.
   c. Regarding timing accuracy of software time advance, the proposed approach should avoid the conventional "annotation-dependent" uninterruptible time advance, rather it should support interruptible time advance.
   d. Although the timing accuracy of behavioural software simulation is restricted by its high modelling level, it still should be sufficient to generate a timed software execution trace which is the same as a corresponding ISS simulation.

2) Build an abstract CPU model, which can simulate HW/SW interactions and support high-level interruptible software timing simulation.
   a. The HW/SW timing synchronisation (i.e., interrupt handling) problem must be solved, since it is related to interruptible software time advance.
   b. There is a limited abstract hardware modelling that supports hardware-dependent software service models, e.g., context switch, interrupts service, and real-time clock service.
   c. The organisation of software models and hardware models should mimic the typical structure of an embedded system, and be extensible for future development.
3) Capture essential and common RTOS features and build a generic RTOS model, in order to flexibly support early and practical simulation of real-time software in SystemC-based system-level design.
   a. The RTOS model should provide generic and standardised multi-tasking, scheduling and synchronisation services as well as other necessary OS functions.

b. In order to enhance modelling flexibility on application tasks, the RTOS simulation model should support both coarse-grain timed abstract task models and fine-grained timed native applications in a hybrid simulation.

c. The RTOS model should achieve accurate simulation in terms of both timing accuracy and functional results.

4) Incorporate limited TLM communication into software models for transaction-based inter-module communication modelling, in order to make software models interoperable with existing TLM modelling and simulation concepts and techniques.

1.6.5 Research Contributions and Methods

Corresponding to above objectives, the research work undertaken in this thesis is fourfold, with objectives 1-3 being the main focus of this thesis, i.e., software modelling and simulation.

The first part of research work contributes results related to the Objective 1, representing guidance of building specific simulation models. A mixed timing software behavioural modelling and simulation approach is proposed. It separates conventionally inter-dependent software timing modelling and simulation into two partially separate phases. It supports mixed software timing information granularities and annotation methods for performance and accuracy trade-off at the modelling phase. The mixed timing models can use both coarse-grained task timing estimates and fine-grained delay annotations in one simulation. Good software pre-emption modelling capability is achieved by the SLDL *wait-for-event* method, with a good simulation performance during the simulation phase. The proposed variable-step and fixed-step time advance methods supply varying observability of system simulation traces, and hence enable a trade-off with the simulation speed.

Regarding the Objective 2, a Live CPU Model is proposed. It represents an essential abstract hardware base in a high-level software PE model and is a proper container to include hardware related components and functions. The most crucial function of the Live CPU Model is to support interruptible time advance in mixed timing software behavioural simulation. Also, the Live CPU Model includes an interrupt controller and some virtual registers, which are actively involved in HW/SW synchronisation modelling and hardware-dependent software modelling. By this means, theoretical interrupt modelling latency and software time advance stopping latency can reach zero-time in simulation, which means an ideal resolution.

In terms of the Objective 3, the third part of research focuses on the development of a generic and accurate SystemC-based RTOS-centric real-time software simulation framework. It integrates mixed timing application models, the RTOS, and the Live CPU Model in a software PE model. The software core is the generic RTOS simulation model. It supplies a set of fundamental and practical services including multi-tasking management, scheduling services, synchronisation and inter-task communication mechanisms, clock services, context switch and software interrupt handling services, etc. These functions are summarised and abstracted from a survey on some popular RTOS standards and products. To build a predictable RTOS timing model, the timing overheads of various RTOS services are considered in models, which is an advantage over some other similar works. The dynamic execution scenarios of real-time embedded software can be exposed by tracing diverse system events and values in simulation, e.g., RTOS kernel calls, RTOS runtime overheads, task execution times, dynamic scheduling decisions, task synchronisation and communication activities, interrupt handling latencies, context switch times, and other user-concerned properties. With this RTOS-centric simulation framework, real-time embedded software designers can quickly and accurately simulate and evaluate the behaviour of both abstract and native real-time applications and the RTOS during the early design phases.

Objective 4 is fulfilled by combining the de facto OSCI TLM-2.0 [88] communication interfaces into the real-time software PE simulation model generated in the above second and third parts of research. This work also defines a SoC TLM model, which not only integrates the software PE model but also includes other typical TLM initiator, target, and interconnection models. This part of work extends the software simulation models to the TLM modelling community.

1.7 Organisation of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2 Literature Review: Transaction-Level Modelling and System-Level RTOS Simulation

This chapter will introduce current TLM research, describe the SystemC SLDL, and survey RTOS modelling and simulation research in the context of system-level design.

This chapter will start with an overview of important concepts and techniques in TLM design, including various topics such as abstraction levels, accuracy/performance trade-off, and typical simulation frameworks. After that, some important SystemC language constructs and the OSCI reference simulator will be introduced along with their relevance to real-time software simulation that is concerned by us. Finally, this chapter will survey related system-level RTOS modelling and simulation research. The existing approaches will be classified and discussed based on their modelling granularities, functional features, and application areas in system-level design flows.

Chapter 3 Mixed Timing Real-Time Embedded Software Modelling and Simulation

This chapter will propose a SLDL-based mixed timing software behavioural modelling and simulation approach and an associated Live CPU Model for fast, flexible and accurate real-time software behavioural modelling and simulation.

At first, this chapter will introduce the problematic annotation-dependent time advance method in SLDL-based software simulation and survey some remedy approaches. It will then describe the mixed timing approach, by defining two types of software models for TLM software computation modelling and discussing various issues in timing modelling and timing simulation. Afterwards, the components and operations of the Live CPU Model will be introduced in detail. Finally, evaluation metrics and experiments will also be presented in order to evaluate the research in this chapter.

Chapter 4 A Generic and Accurate RTOS-centric Software Simulation Model

This chapter will introduce a SystemC-based generic and accurate RTOS-centric real-time software simulation model. It can support flexible and practical real-time software simulation in early design phases.

Firstly, this chapter will present the research context and assumptions. An abstract embedded software stack will be defined as the research target. It will then survey common RTOS concepts and requirements as guidance of following research. Afterwards, details of the RTOS-centric real-time software simulation model will be described. This research will include three main parts, i.e., the overall structure of all simulation models, application software modelling, and RTOS modelling. RTOS modelling is the core part and will be introduced from both the functional modelling aspect and the timing modelling aspect. The chapter will afterwards explain evaluation metrics regarding simulation performance, functional accuracy and timing accuracy of the proposed RTOS-centric simulator. Accordingly, experiments will be carried out in order to demonstrate these aspects.

Chapter 5: Extending the Software PE model with TLM Communication Interfaces

This chapter will extend software simulation models with TLM communication interfaces by utilising the OSCI TLM-2.0 library. This aims to popularise our software modelling and simulation research into the promising TLM modelling domain.

It will firstly introduce related concepts of the OSCI TLM-2.0 library in brief. Then it will describe how to integrate TLM communication constructs into the Live CPU Model. Afterwards, a simple SoC TLM model will be presented in order to integrate the Live CPU Model and reveal how various typical system components are defined for co-simulation with behavioural RTOS-centric software models. Finally, an experiment will study the simulation performance of the SoC simulation model, whilst another DMA I/O experiment will demonstrate the interoperable simulation capability of the combined software and TLM models.

Chapter 6: Conclusions and Future Work

The last chapter will summarise contributions, conclude chapters, and suggest future research directions.

Chapter 2

Literature Review: Transaction-Level Modelling and System-Level RTOS Simulation

In order to help developers deal with the increasing design cost and short time-to-market of today's embedded systems industry, there is a pressing need for new design methodologies to ameliorate these problems. System-level design techniques have been proposed, that use high-level abstraction methods to design hardware and software concurrently in a unified environment. In this research domain, system-level modelling and simulation are key techniques to describe, validate, analyse and verify complex systems. In various system-level modelling and simulation approaches, the SystemC-based Transaction-Level Modelling (SystemC-TLM) has become a de facto standard. Based on the essential TLM principle "separating computation from communication", developers can divide system modelling and simulation into two main aspects, i.e., the *computation* aspect and the *communication* aspect.

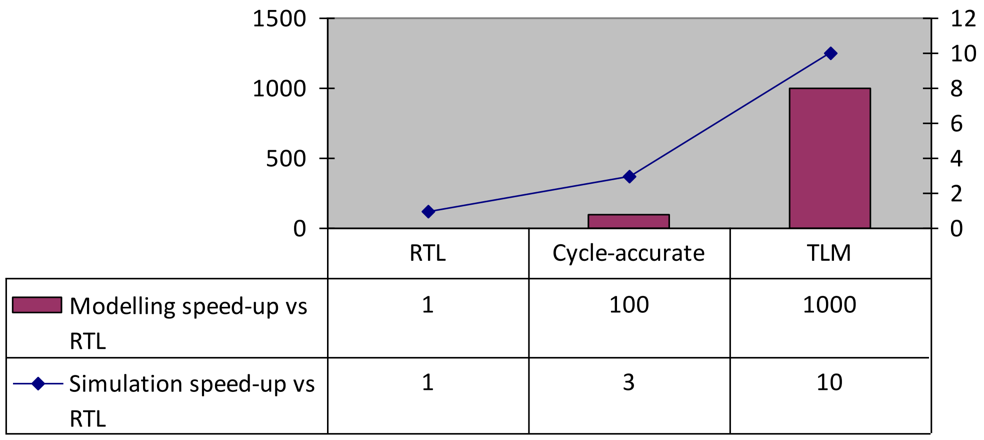
In the general context of embedded systems modelling, the *computation* can be further divided into the *software* aspect (i.e., software running on a CPU) and the *hardware* aspect (i.e., application-specific hardware logics). In this thesis, we specifically concentrate on modelling and simulating real-time software at a high level, namely the software PE model. The HW/SW timing synchronisation in the unified event-driven SystemC simulation environment is addressed, which is crucial for modelling interrupts and greatly affects both simulation timing accuracy and performance. Because of benefits of dynamic scheduling and multi-tasking execution of concurrent real-time applications, RTOS behavioural modelling has increasing relevance for both fast simulation and validation of different software implementation alternatives in the early stages of design. Various RTOS design space exploration activities (e.g., assigning task priorities, deciding scheduling strategies and designing application-specific OS services) also require an early and efficient test bed in order to be carried out. Consequently, the RTOS model is regarded as the heart of behavioural real-time software modelling and simulation research in this thesis.

This chapter starts with some basics of current TLM research and work examples in Section 2.1. As the programming language and research environment of this thesis, SystemC language constructs and the OSCI reference event-driven simulator kernel are introduced in Section 2.2, along with their relevance and inadequate ability for modelling and simulation of real-time software. In Section 2.3, an overview is presented on related RTOS modelling and simulation research in the context of system-level and TLM design. These works motivate our study in this thesis. The HW/SW timing synchronisation approaches and problems in SystemC simulation are also introduced in several paragraphs within this chapter. Section 2.4 will summarise this chapter.

2.1 Transaction-Level Modelling and Simulation

Transaction-level modelling has generally been considered as the emerging system-level modelling style for improving productivity in the design of highly integrated embedded systems which may integrate heterogeneous processors, IP cores, peripherals, memory components, and on-chip communication infrastructures. TLM models are expected to serve as interoperable references across different design teams for fast embedded systems architecture exploration, early embedded software development and functional verification [3].

From the hardware developer's point of view, TLM captures embedded systems at a range of abstraction levels higher than the traditional RTL [89]. Compared to conventional RTL modelling and simulation, TLM not only reduces the model building cost, but also speeds up the simulation performance by orders of magnitude. The literature [3] provides an example project in which the modelling effort and simulation efficiency of three different TLM, cycle-accurate and RTL models are compared. Table 2-1 shows the distinct speed-up of the TLM approach. Another benefit of the TLM approach, more interesting to software developers, is



	RTL	Cycle-accurate	TLM
▮ Modelling speed-up vs RTL	1	100	1000
—◆— Simulation speed-up vs RTL	1	3	10

Table 2-1. Modelling and simulation speed comparisons [3]

that it can support early development and validation of hardware dependent software. Developers can co-simulate software with hardware models in a single-source SLDL-based simulation framework, almost as soon as the initial architecture specification is determined [90]. In this thesis, from a software research's perspective, TLM refers to high-level interaction between different software and hardware modules. It includes behavioural software modelling/simulation, high-level hardware modelling/simulation, and transaction-based communication between them.

However, the higher abstraction levels of TLM models also indicate less modelling detail and some loss of accuracy. The accuracy of TLM simulation, in terms of both data accuracy and timing accuracy, is necessarily sacrificed to some extent due to coarse-grained data transfers and larger time-advancing steps. Of course, with the goal of rapidly describing the system architecture and validating applications, requirements are relaxed in terms of accuracy of bit-level data or cycle-accurate timing. Usually, coarse-grained and reasonably accurate assumptions are made, e.g., packet-level transmission and cycle-approximate timing. Trading accuracy issues against simulation speed [91], or preserving accuracy whilst gaining in simulation performance [92], are popular TLM research topics in terms of efficiency and flexibility. We are also concerned with them in this thesis and will present some studies in the next chapter. At this point, the term "cycle-approximate timing" (or the similar term "approximate-timed" [7]) indicates that a procedure (either a computation action or a communication transaction) in a model is assigned with timing information that spans multiple clock cycles, and that the simulation clock can be progressed with multiple clock cycles in each step. Despite the fact that this term is broadly used as a temporal resolution in the TLM taxonomy, its exact timing granularity is vague. A variety of interpretations from diverse researchers often reveal their own interest in modelling and intention of optimisation, which may make it difficult to compare the performance and accuracy of different TLM works quantitatively and horizontally.

In order to present a general idea of the existing research on TLM, three main topics will be hereby introduced:

- *Abstraction levels of TLM:* A fundamental essence of transaction-level modelling is to raise the level of abstraction by hiding low-level implementation detail. Some important concepts and popular definitions on TLM abstraction levels will be addressed.
- *Communication exploration:* A variety of transaction-based communication modelling approaches have been developed in both academia and industry to define how system components communicate. The research on communication modelling and simulation is a contributor factor to most of current TLM achievements. Here, a brief introduction on related work is presented in order to reveal this essential TLM aspect.
- *Embedded software development in TLM:* If TLM comprehends two portions "communication" and "computation", then modelling software is surely a paramount topic of the TLM computation portion.

2.1.1 Abstraction Levels and Models in TLM

A central issue in various system-level design methodologies is concerned with appropriate abstraction levels and coding styles for modelling various computation and communication activities in TLM. By a general consensus, TLM does not specifically or explicitly indicate a single abstraction level. In fact, a series of abstraction levels are classified in the general category of TLM in different TLM taxonomies. It is not practical to precisely enumerate all abstraction levels for TLM, because there are many different interpretations. However, it is still possible to indicate the range of TLM levels. Without much dispute, most researchers agree that TLM abstraction levels are relatively “higher” than the RTL used in traditional design. Also, TLM abstraction levels are considered to be “lower” than *functional* (*algorithmic*) models. Functional models are not defined as TLM models, although the abstraction level of them is sufficiently high [88]. This is because a functional model usually includes a single software thread only, e.g., in the form of a C function or a SLDL process. It does not bear two essential features of a TLM model: concurrent multitasking computation and inter-process communication [88].

Conventionally, TLM abstract models are organised with respect to some criteria, including:

- *Timing accuracy:* This is a first-class characteristic regarding the accuracy of a model. It refers to how a model is assigned with timing information, e.g., a line of code, a code block, or a task, and cares about the resolution of timing information, e.g., untimed, cycle-approximate, or cycle-by-cycle.
- *Functional accuracy:* This refers to how a model captures the function of a

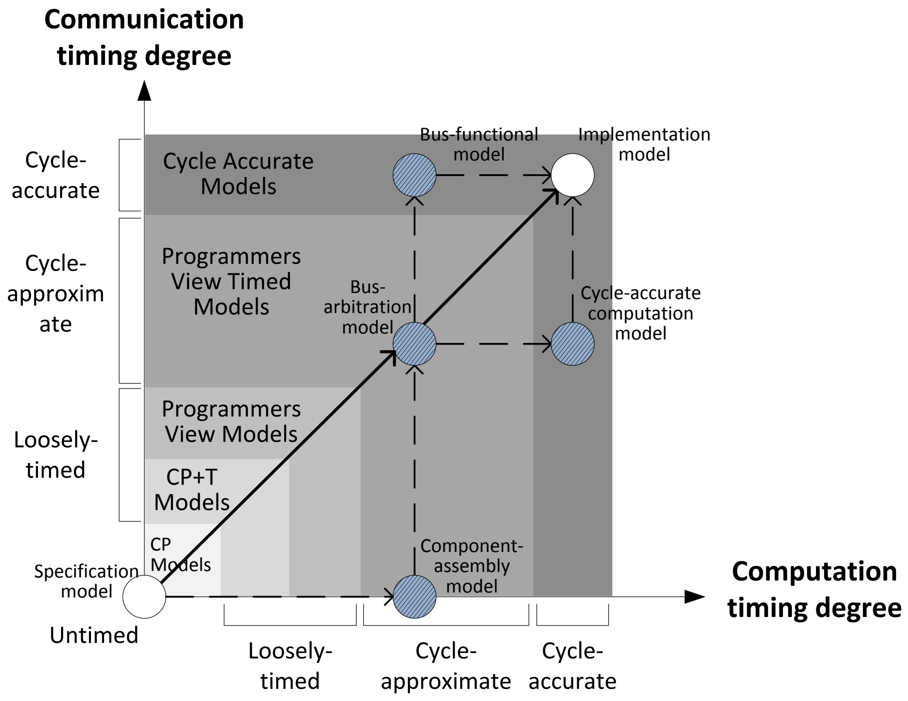


Figure 2-1. Various TLM abstraction levels (partially based on [7])

target system. For instance, some high-level simulators only abstract timing properties (e.g., execution time, period, and deadline) of a software model in order to enhance simulation speed, but without modelling its functional behaviour. The functional accuracy can be evaluated by comparing the outputs of the model with a reliable reference by giving them the same inputs.

- *Communication data granularity:* This criterion regards what data structures are transmitted through communication channels, for example, an application packet, a bus packet, or a word.

There are an number of literatures [3] [88] [7] [93] that feature definitions of TLM abstraction levels. In the following, Sections 2.1.1.1 to 2.1.1.4 will present some examples. Figure 2-1 provides a conjunctional view of these TLM abstraction taxonomies by comparing the timing accuracy of their computation aspects and communication aspects.

2.1.1.1 OSCI TLM Abstraction Levels

The most acknowledged TLM abstraction level taxonomy was proposed by the OSCI TLM working group [3] [88]. The OSCI TLM specification defines two general levels for TLM modelling: the *Programmers View* (PV) level and the *Programmers View Timed* (PVT) level (see Figure 2-1). The PV models are characterised by the *Loosely-Timed* (LT) coding style and the blocking transport interface, in which each transaction is associated with two timing points, corresponding to the start and the end of a blocking transport. It is appropriate for software programmers who require a functional virtual hardware platform with sufficient timing information in order to run an operating system and application software. A PVT model is identical to the PV level model in terms of functionality, but each PVT transaction is annotated with multiple timing points and uses the non-blocking transport interface, namely the *Approximately-timed* (AT) coding style. It enables architecture exploration and also performance analysis of the application system. This OSCI TLM abstraction level view reflects a communication-centric hardware design perspective, although some software designers, with the aim of promoting interoperable TLM modelling, are seeking its application for computation modelling [6].

2.1.1.2 Donlin's Extended TLM Abstraction Levels

In [93], Donlin introduces three TLM levels in addition to OSCI's definition above, i.e., the *Communicating Process* (CP) level, the *Communicating Process with Time* (CP+T) level, and the *Cycle-Accurate* (CA) TLM level. Referring to Figure 2-1, CP and CP+T abstraction levels are even higher than OSCI-TLM levels, where "T" means coarse timing information. CP and CP+T models are more architecture-independent and implementation-independent than PV and PVT models. System models at the two levels consist of parallel processes that exchange high-level data structures by point-to-point connections, rather than arbitrated buses. In contrast, the C*ycle-Accurate* (CA) abstraction level is lower than OSCI levels. It captures micro-architectural details and is time-accurate to the level of each clock cycle. In some TLM literatures [3] [94], CA models are sometimes not referred to as a part of the TLM space because of their limited speed-up compared to a RTL model (Table 2-1 hints at this). However, in [93], Donlin's focus is to investigate the use of CA TLM models for detailed performance analysis and verification of hard real-time software in the final design stages; consequently the drawback regarding performance is considered to be worthy of toleration.

2.1.1.3 Cai and Gajski's Orthogonal TLM Modelling Graph

Another early and classical TLM taxonomy is introduced by Cai and Gajski in [7], which concludes that *communication* and *computation* are equally important yet orthogonal aspects of TLM research. Referring to Figure 2-1, these two aspects are illustrated as two axes according to degrees of timing accuracy in a system modelling graph. They identify three timing degrees, i.e., untimed, approximate-timed (so-called cycle-approximate), and cycle-timed (so-called cycle-accurate). Moreover, the authors define six abstraction models in the graph and explore their usage in embedded system design flows, starting from the specification stage and ending at the implementation stage. Among the six models, four (the shaded circles in the figure) are classified as TLM models, i.e., the component-assembly model, the bus-arbitration model, the bus-functional model, and the cycle-accurate computation model. The solid arrows in the figure represent a typical TLM system design flow, whilst the other dotted arrows stand for some possible design routes depending on different design intentions, e.g., communication-focused or computation-focused.

2.1.1.4 Mixed-Level and Multiple-Level TLM Modelling Research

Various TLM models at different degrees of accuracy bring a potential for multiple-level or mixed-level modelling in which designers can trade off modelling accuracy and simulation performance according to different strategies.

In Chapter 2 of [3], the researchers propose a general idea for TLM mixed-level modelling by combining untimed TLM models and standalone timed TLM models. This allows for concurrently developing pure functional models (by architecture teams) and timing models (by micro-architecture teams) with dissimilar modelling purposes. Multiple timing scenarios with different resolutions can co-exist in a unified simulation model, and simulation speed can be optimised by dynamically switching untimed and timed models at runtime.

For bus communication modelling, Schirner and Dömer quantitatively analyse simulation speed and timing accuracy of three abstract communication models, e.g., the conventional TLM model, the arbitrated TLM model, and the cycle-accurate and pin-accurate bus functional model [92]. They configure them with varying data granularities and arbitration handling methods in order to trade off simulation accuracy and performance. Focusing on software computation modelling, they define five abstraction levels for processor modelling (e.g., the application level, the task scheduling level, the firmware level, the processor TLM level, the processor functional model) and quantify accuracy loss and simulation speed-up of each model [79].

For processor and communication design co-exploration, an integrated design methodology is presented in [95]. It combines multi-level processor hardware models (e.g., instruction-accurate and cycle-accurate) and communication models (TLM buses and RTL buses), by which the processor design team can co-operate with the communication team early in the design flow.

2.1.1.5 Summary

The different views of TLM abstraction levels and related models have common notions of hardware and communication modelling. Each TLM abstraction level can be seen as a limited design space for exploring and validating some functional and timing issues with corresponding models. Multiple TLM abstraction levels thus constitute a wide design space, namely a design flow, for successive model refinement through the addition of design detail.

The OSCI TLM standard is gaining a high level of popularity and sustainable development in both industry and academia. It provides two distinguishing levels (i.e., LT or AT) for communication models depending on their timing degrees and synchronisation methods. The relevance of this modelling idea will be examined to the proposed software modelling approach in Section 3.2.2. The mixed modelling idea is widely advocated for accuracy and speed trade-off in both the OSCI TLM standard and the research surveyed in Section 2.1.1.4. Specifically, it is also a guiding concept of the mixed timing software modelling approach that is to be presented in Section 3.2. The recent OSCI TLM standard Version 2.0 provides standard interfaces for creating bus communication models. Chapter 5 will investigate combining these API interfaces with the proposed software models in order to advance interoperability between TLM communication and our native-code software simulation models.

2.1.2 Communication Modelling in TLM

If we interpret the term "transaction" as an "*abstract communication operation*" [47] or as a "*high-level form of a communication protocol*" [96], then the name "transaction-level modelling" is likely to imply that *communication* is a main research topic. From a narrow viewpoint, TLM is understood as a communication-centric embedded systems modelling paradigm [97]. Early in 2002, Grötker et al. introduced the basic TLM interface-based communication style with a high simulation performance [98]. This work forwards SystemC as the most established design language vehicle for TLM approaches today. In this section, we will make a brief introduction mainly, but not limited to, SystemC based TLM communication and architecture exploration studies.

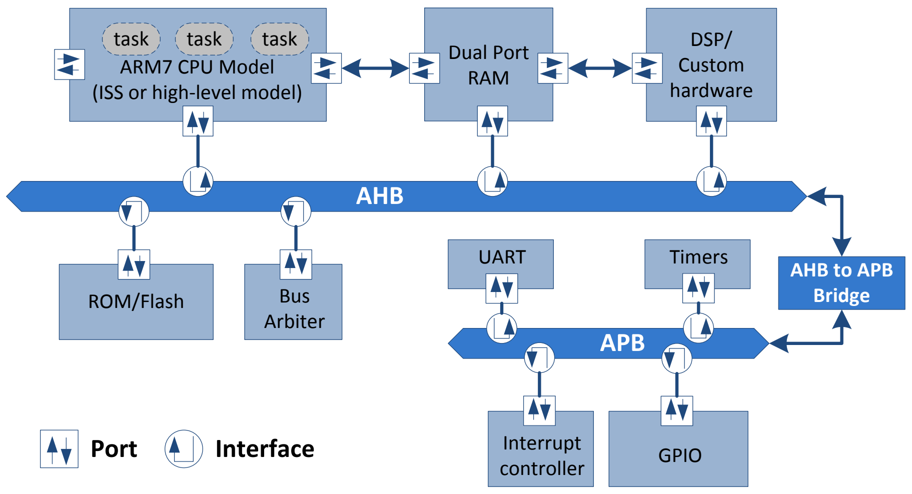


Figure 2-2. An AMBA TLM model example

In TLM, in order to build a virtual prototype that represents abstract models of an embedded system, a system is broken down to a set of computation components comprising concurrent processes to implement application functions. Computation components communicate with each other through ports and channels by sending and receiving transaction requests. Figure 2-2 shows a block diagram of an example SoC TLM model, e.g., the AMBA bus. In this model, the architecture is composed of two main computation components, i.e., an ARM microprocessor and an application-specific processor (e.g., DSP or custom logics) as initiator components in the system. Some other components including fast and slow memories, peripherals, and devices are connected to processors by direct port-to-port connections and buses, e.g., the Advanced High-performance Bus (AHB) and the Advanced Peripheral Bus (APB). From the TLM perspective, the buses are complex channels accessed by multiple modules through respective ports.

Figure 2-3 depicts the basic method of TLM communication modelling. In this example, two modules communicate through a channel. The *Process A1* in Module A can write a value to the channel by calling the method `write()` through its parent module's port *pA*, whilst the *Process B1* retrieves a value from the channel by the method `read()` via port *pB*. This Interface Method Call (IMC) scheme achieves high modularity in inter-module communication modelling, and essentially separates communication and computation details.

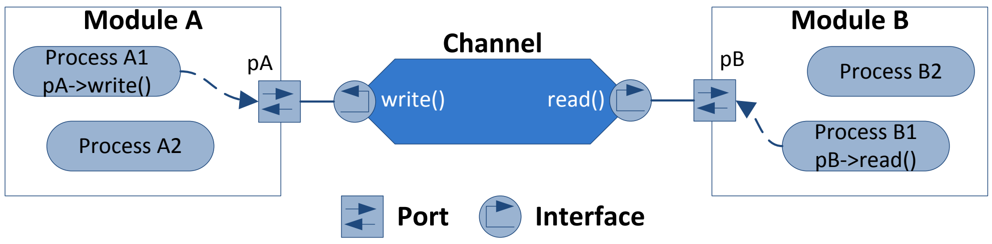


Figure 2-3. TLM Interface Method Call Communication

As the key element of the TLM IMC communication, a channel can have varying complexity across different designs. In a SystemC-TLM specification, a channel can be implemented in two styles, i.e., the primitive channel and the hierarchical channel. A primitive channel contains processes and ports and aims to provide simple and fast communication. The SystemC language reference manual [66] defines several built-in primitive channels (all derived from a base class `sc_prim_channel`), e.g., `sc_signal` (to model a simple wire carrying a digital electronic signal), `sc_fifo` (to model a first-in-first-out buffer), `sc_mutex` (to model a mutual exclusion lock) and `sc_semaphore` (to model a software semaphore), etc. Hierarchical channels are indeed hybrid modules and can contain other instances of modules, processes, ports and nested channels. They are used to model complex customised communications, such as buses or networks.

In order to advocate model interoperability between different communication modelling and architecture design communities, some standards are proposed to promote the SystemC TLM communication paradigm. The following are two predominant standards.

The OSCI TLM Working Group, which was founded in 2003, has published a series of OSCI TLM standards. The up-to-date OSCI-TLM library version 2.0 [88] [99] introduces a set of well-defined core APIs, data structures, initiators, targets, the generic payload, and the base protocol for transaction-based communications. The core interfaces support two types of transport, i.e., the *blocking* transport (a transaction can suspend its parent process) and the *non-blocking* transport (a transaction is atomic and does not suspend its parent process). The generic payload is primarily intended for modelling a typical memory-mapped bus, which is abstracted away from the details of any specific bus protocols. An extension mechanism is also offered to model specific bus protocols or non-bus protocols by users. The Open Core Protocol International Partnership (OCP-IP) consortium is another active TLM standardisation organisation. It has proposed and maintained a SystemC TLM modelling kit since 2002 [100] [101], defining a stack of communication layers including four abstraction levels, i.e., Message Layer (L-3), Cycle-approximate Transaction Layer (L-2), Cycle-accurate Transfer Layer (L-1), and the RTL Layer (L-0). Its latest version, which is built on top of OSCI-TLM v2.0, provides an interoperable standard for SystemC component models with OCP protocol features.

A number of TLM modelling and simulation approaches have been proposed for the design of complex communication systems. The following are some representative works.

Gajski's group presents examples of TLM communication research mainly based on the SpecC language. The literature [102] describes a general TLM communication modelling style for SoC design. For Network-on-Chip synthesis, they define some successive system communication abstraction layers and corresponding design models to refine abstract message-passing down to a cycle-accurate, bus-functional implementation [58]. For AMBA AHB bus modelling, they propose a Result Oriented Modelling (ROM) technique that improves accuracy drawback of conventional TLM models and gains high speed by omitting internal states and making end result correction [103].

In 2002, Pasricha pointed out the direction for using the SystemC TLM modelling approach in early architecture exploration and developed communication channels for fast simulation for embedded software development [90]. In order to bridge the gap between high-level TLM models and bus cycle-accurate models, Pasricha et al. present an intermediate TLM abstraction level "Cycle Count Accurate at Transaction Boundaries" (CCATB) for communication exploration, which improves simulation speed by keeping cycle-level timing accuracy only at transaction boundaries [104].

Kogel et al. propose a series of multiple-level SystemC-TLM co-simulation and virtual architecture mapping methodologies for architectural exploration of

NoC, SoC, and MPSoC [105] [106] [95]. Klingauf et al. describe the TRAnsaction INterchange (TRAIN) architecture for mapping abstract transaction-level communication channels onto a synthesisable MPSoC implementation by virtual transaction layers [55]. They also propose a generic interconnect fabric for TLM communication modelling that aims to support flexible buses, multiple TLM abstraction levels, and various TLM standard APIs [107].

2.1.3 Embedded Software Development with TLM

Embedded software development with TLM models is not a new topic and many studies have been conducted in this area. In this section, we introduce them depending on relationships between software modelling and TLM techniques:

- Conventional ISS software simulators utilise TLM communication for modelling SW/HW interfaces only (Section 2.1.3.1);
- System-level software modelling and simulation comply with general TLM concepts and techniques (Sections 2.1.3.2 and 2.1.3.3).

2.1.3.1 ISS SW Simulation with TLM SW/HW Interfaces

In an early TLM literature [90], Pasricha indicated the concept of developing embedded software with SystemC TLM models. This is mainly motivated by two encouraging TLM modelling results: the early availability of TLM architectural models in the SoC design lifecycle and the much higher simulation speed compared to detailed RTL models. The goal is to design and simulate embedded software on top of a virtual prototype of the target architecture instead of using traditional RTL models or the final implementation. This research uses a HW/communication-centric TLM and conventional software simulation approach.

Several efforts have been made to combine conventional cycle-accurate software simulation (e.g., an ISS) with SystemC-based abstract TLM hardware and communication models [108] [109] [95]. As shown in Figure 2-4, TLM techniques are used to model SW/HW communication interface and hardware components, which are outside the scope of software modelling. The SPACE methodology [108] encapsulates an ISS in a SystemC wrapper and connects it with rest modules of the modelling platform through TLM channels. Two types of TLM

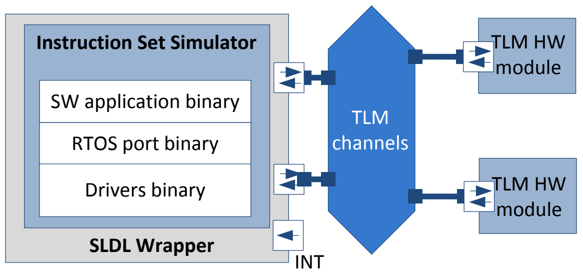


Figure 2-4. TLM technique for modelling SW/HW interfaces

communication channels (untimed and timed) are provided to support two TLM abstraction levels: untimed channels are for a faster verification of applications before partitioning, while timed channels are used for cycle-accurate modelling. Cross-compiled binary code of software application, the OS, and drivers executes in the ISS. For MPSoC design space exploration, the MPARM approach integrates multiple SystemC-based ARM processor models (ISS simulators in SystemC wrappers), the AMBA bus model, and memory models [109]. The TLM channels implement the bus communication architecture in a master-slave style.

2.1.3.2 Embedded Software Generation Using TLM Models

Recalling the fundamental TLM principle of separating the concerns of computation and communication, these two design aspects should be paid equal attention in TLM contexts. Some researchers are also concerned about applying TLM concepts and techniques to design and validation of the computation portion [9] [6]. Software is the integral and main part of many embedded systems and hence has become a major area of interest in transaction-level computation modelling.

Motivated by the goal to co-design an entire electronic system from the specification phase down to the implementation phase by using a single SLDL, some system-level design flows have been proposed to support embedded software generation and synthesis. In these studies, a series of SLDL-based specification and TLM models are simulated, refined and transformed, in order to automatically generate target embedded software C/C++ code [62] [63] [110], or to further generate final binary files, i.e., system-level software synthesis [59] [61].

Figure 2-5 shows a typical embedded software generation flow. Firstly, untimed and before-partitioned system functions are described by a set of hierarchi-

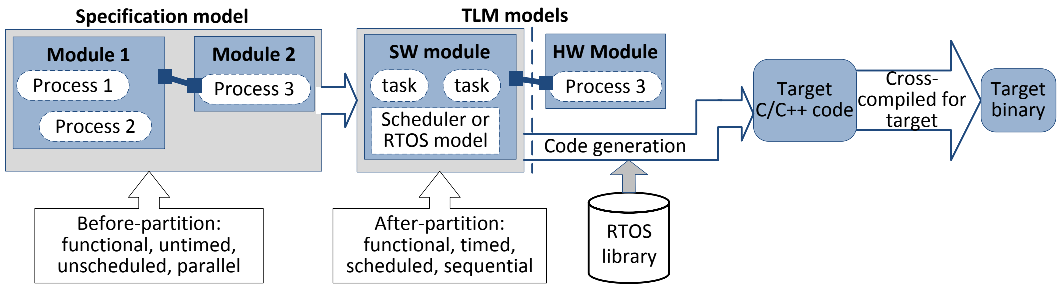


Figure 2-5. Software generation using TLM models

cal SLDL elements such as modules, processes, interfaces, channels, and ports. These processes run in parallel and communicate with each other by means of transaction style channels. Through iterative simulation and partition, untimed specification models are transformed into PV or PVT TLM models. At the TLM architecture exploration stage, a simple scheduler or a RTOS model may be integrated to assist sequential software simulation. In order to generate software implementation code towards a specific operating system, a RTOS-specific library (e.g., RTEMS [59], QNX [63]) is introduced to replace the RTOS model with behaviourally equivalent RTOS functions, and SLDL processes are mapped to real RTOS tasks. Finally, SLDL-based software code is cross-compiled into executable binary code for a target processor.

These approaches reveal a system-level design point of view and make a valuable contribution to co-design and co-synthesis flows. However, such a design flow is still not straightforward. The first obstacle resides in transforming specification models described in a SLDL into RTOS based TLM software execution models. The hardware-style channel communication mechanism used in specifications is not suitable for real-time software design, which may sacrifice the conventional software implementation productivity and legacy. Besides, it is known that the SystemC library bears the weakness of not supporting priority assignment and pre-emptive scheduling, so the built-in SystemC kernel scheduler and synchronisation primitive channels are not applicable for real-time software modelling. Consequently, the idea in [62] that simply replaces SystemC library elements with target RTOS functions may not be appropriate. A usual solution is to integrate a RTOS model on top of the SLDL in order to supply necessary dynamic real-time software services, which is also the method used in the thesis. Another problem is the increasing size of binary code, because the generated software code includes overhead from some SLDL language constructs [62] [59]. For resource-limited embedded systems, some efficient optimisation techniques may be required to reduce the interference from the SLDL library in target code.

2.1.3.3 TLM Modelling of Software Processing Element

While some research activities have been devised for software development in the overall system-level design flow, recently some methodologies and techniques have emerged that specifically focus on the need of abstract modelling a software PE (i.e., software running on a CPU) in the context of TLM [79] [111]. This topic can be seen as a mixture of two aspects: abstract processor modelling (the hardware aspect) and behavioural software simulation (the software aspect). Figure 2-6 depicts features of a TLM software PE model and some possible modelling options.

From the hardware designers' angle, the motivation is to abstract physical processor features into functional elements in order to simulate high-level software models in the execution environment and connect software models with the rest of the system. In [111], Bouchhima et al. present an abstract CPU model aiming for timed MPSoC HW/SW co-simulation. It provides a set of Hardware Abstraction Layer (HAL) APIs for upper-layer software models and an interface for connecting other system components. This CPU model captures an architectural view of a processor, which includes subsystems like an execution unit for HW

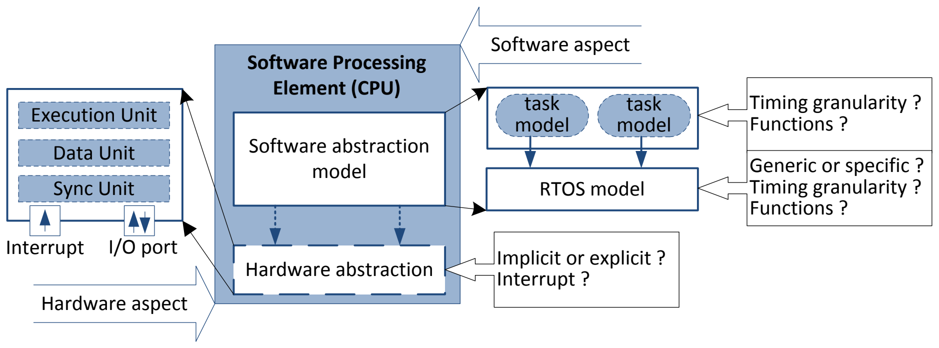


Figure 2-6. Software processing element modelling in TLM

multiprocessing, a data unit wrapping any devices and memory elements, an access unit containing address space, and a synchronisation unit behaving as an interrupt controller. In a subsequent work [6], they introduce a SW TLM communication refinement approach named “SW bus” to enable SW tasks to access logical resources of HW TLM models. In [79], Schirner et al. develop a high-level processor model to support software simulation. The abstract processor model is modelled in a layered approach including five increasing feature levels, i.e., the application layer, the OS layer, the HAL layer, the TLM hardware layer, and the bus functional hardware layer. This model enables incremental and flexible description of the software subsystem at different design stages.

If we turn to a software developers’ perspective, a software processing element model should consist of various software models at appropriate levels of abstraction for behavioural software simulation. Timed software simulation, RTOS scheduling, and interrupt handling are three key aspects to evaluate research in this area. In a large number of embedded systems, a RTOS provides a useful abstraction interface between real-time applications and processor hardware abstraction. Consequently, most software processing element modelling approaches integrate a RTOS model in order to supervise native execution of application, which is known as RTOS modelling [12, 43, 73, 87, 112, 113, 114, 115]. In respect of the research in this thesis concentrating on the RTOS modelling, a more complete survey will be given in Section 2.3. In Figure 2-6, timing granularity and functional accuracy are used as dimensions to guide and compare software models, which offer choices on abstraction levels of task models and RTOS model. Still in the figure, the hardware abstraction model is illustrated by a dotted frame, this reflects the current situation whereby some software modelling approaches do not include interrupt handling, nor consider the interoperability with hardware models, i.e., hardware abstraction is implicit in the high-level PE model.

2.2 The SystemC Language

SystemC is an open-source C++ based system-level design language that is often used for high-level system modelling and simulation. Unlike the conventional heterogeneous HDL-ISS HW/SW co-simulation, the SystemC modelling framework can provide a homogeneous programming and co-simulation environment, by which users can write both software and hardware models in a unified common language and natively compile them as a single process on the host computer. The SystemC execution model uses a discrete-event simulation kernel to schedule model processes (a set of C++ macros) so as to mimic functional behaviour and time progress of a target system.

In this section, we will start with a brief introduction to SystemC language features with regard to concerned support for software modelling. We will then take a look at the SystemC co-operative execution model which closely affects real-time software simulation. Finally, an example of a simple SW/HW system model is presented in order to illustrate the structure of a SystemC model.

2.2.1 SystemC Language Features

The SystemC class library is implemented by a set of C++ library routines, macros, type definitions, templates, and overloaded operators. Figure 2-7 shows the simplified layered structure of a SystemC application. Users can develop simulation models based on SystemC and C++ languages, and they can additionally use some SystemC libraries depending on specific design necessity, e.g., the OSCI TLM library [88].

Referring to Figure 2-7, the components of the SystemC library are briefly classified and introduced as follows. More comprehensive description can be found in the language reference manual [66].

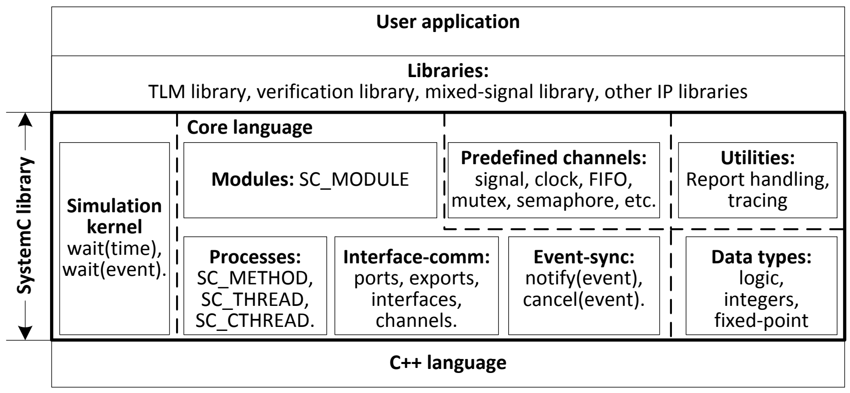

User application
Libraries:
TLM library, verification library, mixed-signal library, other IP libraries
SystemC library
Core language
Simulation kernel
wait(time), wait(event).
Modules: SC_MODULE
Predefined channels: signal, clock, FIFO, mutex, semaphore, etc.
Utilities: Report handling, tracing
Processes: SC_METHOD, SC_THREAD, SC_CTHREAD.
Interface-comm: ports, exports, interfaces, channels.
Event-sync: notify(event), cancel(event).
Data types: logic, integers, fixed-point
C++ language

Figure 2-7. SystemC language structure

The Simulation Kernel

It schedules SystemC processes in response to an event or a time delay. The exact execution mechanism will be described in the next Section 2.2.2.

Language Utilities

These utility classes provide some assisted services in terms of tracing value changes, reporting exceptions, and mathematical functions.

Data Types

In addition to supporting native C++ types, SystemC defines some data types for hardware modelling, for instance, integer types within and beyond 64-bit width (e.g., `sc_int<WIDTH>`, `sc_bigint<WIDTH>`), fixed point data types (e.g., `sc_fixed`, `sc_ufixed`, etc.) and four-valued logic types (e.g., `sc_logic`, `sc_lv<WIDTH>`, etc.). Because SystemC data types are defined in classes with inevitable overheads, it is recommended to use C++ native types or simple SystemC integer types for best performance if possible [116].

The Core Language

This category of classes provides main modelling functions regarding model hierarchy, execution units, concurrency, synchronisation and communication, etc.

- A *module* (`SC_MODULE`) is the basic SystemC building block, namely an object of a C++ class. The model of a computing system is composed of several interconnected hierarchical modules. A module is the container of a variety of modelling elements such as processes, events, ports, channels, member module instances and data members.
- A *process* is the basic SystemC execution unit (a macro) that is encapsulated in a `SC_MODULE` instance in order to perform computation of a system. There are three types of process to wrap a function: the method process (`SC_METHOD`), the thread process (`SC_THREAD`) and the clocked thread process (`SC_CTHREAD`). The main difference between them is that the method process atomically runs from beginning to end once triggered, but the thread and clocked thread processes can be suspended and resumed by directly or indirectly calling `wait()` functions that can be used to simulate time cost of a real activity. The `SC_CTHREAD` process, a variation of

`SC_THREAD`, is only statically sensitive to a single clock and mainly used in high-level synthesis [116].

- *Ports* (class `sc_port`), *exports* (class `sc_export`), *interfaces* (abstract base class `sc_interface`) and *channels* (a type definition of `SC_MODULE` and implementing one or more interfaces) are main language constructs to model inter-module communication of a system by means of the aforementioned interface method call approach.
- An *event* (class `sc_event`) is used to synchronise processes. The immediate or pending notification of an event (`event.notify()`) can trigger (resume) the process that is waiting on it immediately or at a future time point. An event can also be cancelled (`event.cancel()`) when it is at a pending notification status. Compared to the interface method call method, using an event is a lightweight synchronisation and communication method to ease modelling costs. By flexibly changing the opportunity to notify or cancel an event during simulation, users can change a process's suspending time at run-time.

Predefined channels

SystemC contains a number of predefined channels with affiliated methods and ports, which implement some straightforward communication schemes (introduced in Section 2.1.2). Note that although the mutual exclusion and the semaphore synchronisation methods are provided as predefined channels in SystemC, their characteristics differ from what they usually are in the real-time software context. We will address this issue later in Section 2.2.2.2.

2.2.2 SystemC Discrete Event Simulation Kernel

Apart from a few attempts that develop their own proprietary simulation kernels such as the synchronous data flow execution model in [68] and the POSIX thread implementation model in [69], most current SystemC simulations are driven by the built-in OSCI discrete event kernel. We now summarise some distinctive characteristics of the simulation kernel and discuss its advantages and disadvantages regarding real-time software simulation in particular.

2.2.2.1 The Co-operative Simulation Engine

The current SystemC execution model (after Version 2.1) can be implemented (compiled) using three thread libraries on different host OS platforms, i.e., the QuickThread package for UNIX-like OSs, the Fiber thread package for Windows OS and the more portable POSIX pthread library [117]. But no matter what the implementation is, the co-operative multitasking policy remains the same. Simply speaking, only one process will be dispatched by the scheduler to run at a time point. The running process cannot be pre-empted by another. In case the running process is a thread type, it transfers the control to the scheduler by calling `wait()` functions or exits; a method process only yields control when its function body finishes.

Figure 2-8 illustrates the operating cycle of the kernel. Notably, due to irrelevance to the simulation cycle, the initial *elaboration* phase (i.e., before the start of simulation), at which SystemC modules are constructed, is not included in the figure.

Initialisation: This is the first phase after a SystemC simulation starts, i.e., after calling the function `sc_start()` in the main model program. All modelling processes without a special declaration of `dont_initialize()` are put into a

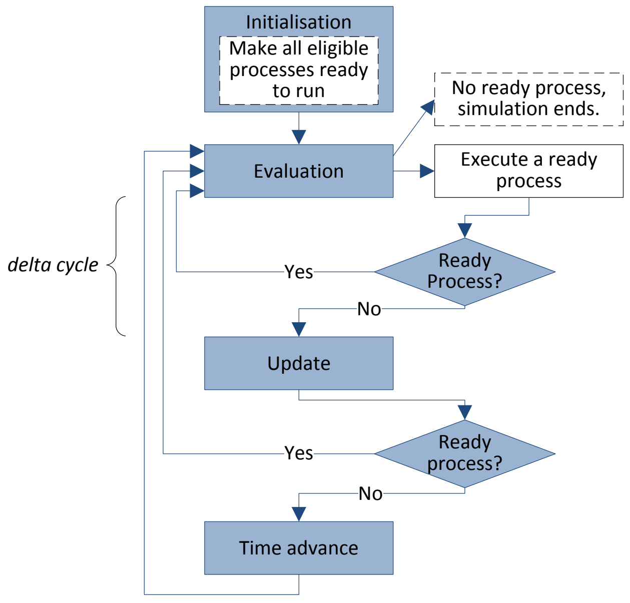


Figure 2-8. SystemC kernel working procedure

ready pool.

Evaluation: At the evaluation stage, ready processes execute sequentially, otherwise the simulation ends if there are no runnable processes. The execution order of them is unspecified in the SystemC specification. In the co-operative execution, a process quits the running state either by initiatively calling a `wait()` statement or simply finishing its function body. There are two kinds of `wait()` statements:

- The `wait(time)` function makes a process blocked for an un-interruptible time duration and will resume the process after that specified time. This will be also referred to as the *wait-for-delay* method hereafter.
- The `wait(event)` function makes a process blocked until the specified event occurs. This will be also referred to as the *wait-for-event* method hereafter.

Because processes may also notify some events immediately in execution and thus cause other processes to be ready to run at once, the evaluation stage will iterate until no process is runnable. Besides, executing a process may access primitive channels and change the signal value, which will consequently result in the updating of data at the next update phase.

Update: In order to model the phenomenon that combinational electronic signals change values instantaneously in parallel within the sequential SystemC simulation, SystemC uses an *evaluation-update* method to guarantee all signals are synchronised. At the update phase, the `update()` method of each channel that previously had requested an update before is called by the kernel to renew the signal with a new value. If this action notifies an event to wake up a process, or the kernel finds that some events are to notify blocked processes, then the kernel will enter the evaluation phase again in order for repetition to occur. This procedure, from evaluation to update and iteration, is known as a *delta cycle*, which does not advance the simulation clock because everything happens at the same time point in actual life.

Time advance: When there is no runnable process, the kernel will progress the simulation clock to the earliest time point specified by a time delay or the nearest pending event it is scheduled to notify. Some processes may thus become runnable and it is thus necessary to begin a new evaluate phase.

2.2.2.2 Advantages and Disadvantages for Real-time Software Modelling

Regarding fast TLM HW simulation and behaviour software simulation, the SystemC SLDL can supply a homogeneous environment to model SW/HW by the same C++ language description and drive their co-simulation by the same engine. The global SystemC clock can be used for both the HW part and the software part, which avoids the overhead of exchanging local clock information in a heterogeneous co-simulation environment [118] [86]. However, the HW/SW timing synchronisation problem still exists within the SystemC simulation. The uninterruptible SystemC `wait(time)` clock advance method leads to a problem whereby a process using `wait(time)` is not pre-emptible during its delay duration. The timeliness to respond to an asynchronous event depends on the length of the current time delay slice.

In the SystemC discrete event simulation, if events occur at different time points and make corresponding processes ready, the scheduler is deterministic and schedules process execution sequentially. However, if multiple processes get ready at the same time point (i.e., during the same evaluation phase or in a delta-cycle), the SystemC standard does not specify their running order [66]. This partial ordering concurrency has disadvantages for real-time software modelling which requires predictability and determinism. For example, multiple processes are blocked waiting to execute a SystemC `mutex.lock()` operation, then which process will get a chance to run is non-deterministic, depending on the order of process execution during the evaluation phase. This behaviour also happens on the SystemC semaphore synchronisation mechanism.

The SystemC co-operative execution model has a native side-effect of keeping the integrity of shared data in atomic process execution. Because a process cannot be pre-empted involuntarily, it can access shared variables exclusively in zero time. However, this feature cannot replace common software synchronisation methods for protecting shared resources, since it is necessary to guarantee the exclusive access in a period of time by using a `wait()` function in a timed software simulation. It is possible that another process may rewrite shared data in the same delta cycle before the `wait()` delay of the last accessing operation has been progressed, which is not desired simulation behaviour.

Consequently, in order to model and simulate real-time software in the SystemC environment, people should try to avoid or otherwise carefully use the aforementioned error-prone features.

2.2.2.3 Discussions on Simulation Time

SystemC uses an integer-valued absolute time model. A time object (class `sc_time`) is represented by two parts: a numeric value and a time unit. The time value is a 64-bit unsigned integer, whilst the time unit can have six granularities from the most fine-grained femtosecond (`SC_FS`) level to the most coarse-grained second (`SC_SEC`) level. The time resolution is the smallest time that can be presented in a simulation and is defined before starting simulation.

When people talk about time in SystemC modelling and simulation, there are often two terms involved:

- *Target Time* (also called *simulated time, target simulation time*): People build models in SystemC and simulate them on the host computer in order to mimic the behaviour of a target system. If models are timed, then people need to assign time delays for various operations in models, which represent the corresponding execution time on a target system. This kind of "execution time" can be called the "target time”, which relates to the *virtual clock* (also known as *virtual time*, *target clock*). In SystemC simulation, its elapse can be observed by inserting the SystemC `sc_time_stamp()` function in model code.
- *Host time* (also called *simulation time*): As a native simulation approach, SystemC models are compiled for and run on a host computer. Running a SystemC program necessarily consumes some host CPU time, just like all other software programs. People call this "host CPU time" as the "host time” or the “simulation time", and regard it as the simulation performance (speed) that indicates how fast a simulation is in the real world.

It is worth noting that there is not a simple linear relation between the simulated time and the simulation time regarding different SystemC simulations. Because of the discrete-event nature of the simulation engine, in general, the simulation speed mainly depends on how many events are involved in simulation, i.e., the more events, the lower the speed.

2.2.3 A SystemC SW/HW System Example

This section gives a simple SystemC example consisting of a software processing element and a hardware component. Figure 2-9 depicts the architecture of this example. The hardware model transmits integer data to a software process via a signal channel, and another software process is in charge of outputting the received data.

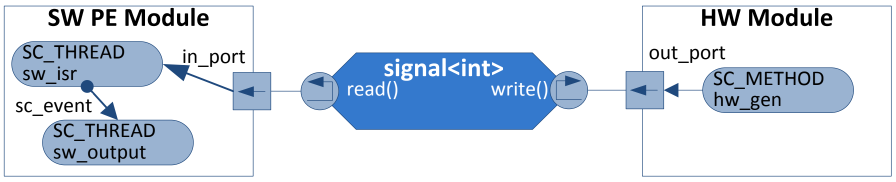


Figure 2-9. Block diagram of a SystemC example

This example covers several basic SystemC modelling issues, e.g., concurrent processes, software sequential execution, co-operative scheduling, event-based synchronisation method, interface method call communication, static sensitivity, and dynamic sensitivity, etc. The SystemC code of this example includes three parts: the hardware module in Table 2-2, the software module in Table 2-3, and the main function in Table 2-4.

```
SC_MODULE(HW)                         //Hardware component module
{
    int TXD;
    sc_out<int>   out_port;           //Data transmission port
    SC_CTOR(HW)
    {
         SC_METHOD(hw_gen);           //Process declaration
    }
    void hw_gen()
    {
         TXD = rand()%10;
         out_port->write(TXD);
         cout<<sc_time_stamp()<<" HW:"<<TXD<<endl;
         next_trigger(1+rand()%5, SC_US);    //Next run
    }
};
```

Table 2-2. SystemC code of a HW module

Referring to Table 2-2, the function of the hardware module is simply embodied in a `SC_METHOD (hw_gen)`, which executes repeatedly after a randomised interval (see line 14). In each execution, it writes a random integer `TXD` to the output port by calling the method on the port.

Referring to Table 2-3, there are two `SC_THREAD` type processes in the software processing element module. At line 12, the `sw_isr` process is sensitive to the value change of the `in_port` and then receives data from it. Once `sw_isr`

```
#001  SC_MODULE(SW)//Software PE module
#002  {
#003      sc_in<int>    in_port;          //Data receiving port
#004      sc_event      evt_sw;
#005      int           RXD;
#006      boolean       cpu_busy;         //CPU is occupied
#007      SC_HAS_PROCESS(SW);
#008      SW(sc_module_name name):sc_module(name),cpu_busy(false)
#009      {
#010           SC_THREAD(sw_isr);
#011           dont_initialize();
#012           sensitive<<in_port;        //Static sensitivity
#013           SC_THREAD(sw_output);
#014           dont_initialize();
#015           sensitive<<evt_sw;
#016      }
#017      void sw_isr()
#018      {
#019           for (;;)
#020           {
#021             if (!cpu_busy)
#022             {
#023               cpu_busy = true;
#024               cout<<sc_time_stamp()<<" sw_isr runs"<<endl;
#025               RXD = in_port->read();
#026               wait(1, SC_US);        //wait for delay
#027               cpu_busy = false;
#028               evt_sw.notify();       //Trigger sw_func()
#029             }
#030             wait();                  //Revive static sensitivity
#031           }
#032       }
#033       void sw_output()
#034       {
#035           for (;;)
#036           {
#037             if (!cpu_busy)
#038             {
#039               cpu_busy = true;
#040               cout<<sc_time_stamp()<<" sw_output data:"<<RXD<<endl;
#041               wait(2,SC_US);
#042               cpu_busy = false;
#043             }
#044             wait();
#045           }
#046       }
#047   };
```

Table 2-3. SystemC code of a SW PE module

```
#001 int sc_main(int argc, char **argv) //Main function
#002 {
#003     sc_signal<int>         sig;
#004     HW hw_i("HW_moduel");
#005     SW sw_i("SW_module");
#006     hw_i.out_port(sig);
#007     sw_i.in_port(sig);
#008     sc_start(100, SC_US);
#009     return(0);
#010 }
```

Table 2-4. SystemC code of the main function

finishes execution, it notifies the event `evt_sw` in order to make the other process `sw_output` ready (see line 28). The two processes use `wait(time)` statements to simulate their execution time cost (lines 26 and 41). Since it is assumed that there is only one conceptual SW PE, the two processes need to execute sequentially. A flag variable is used to guarantee that only one software process can be at the running state (i.e., during a delay interval) at a time.

Referring to the main function in Table 2-4, modules and channels are created and instantiated (lines 3-6). Corresponding ports on both HW and SW modules are connected by the channel object `sig` (lines 6, 7) in the elaboration phase. A call to the function `sc_start()` begins the simulation, which will continue for 100 microseconds target time in our simulation (line 8).

It should be noted that, in this example, two software processes execute according to the SystemC native co-operative scheduling policy and use the uninterruptible `wait(time)` function to advance the target clock. That is, one software process executes up to completion and one process cannot pre-empt the other. As a result, if a hardware signal arrives when a software process is executing, the

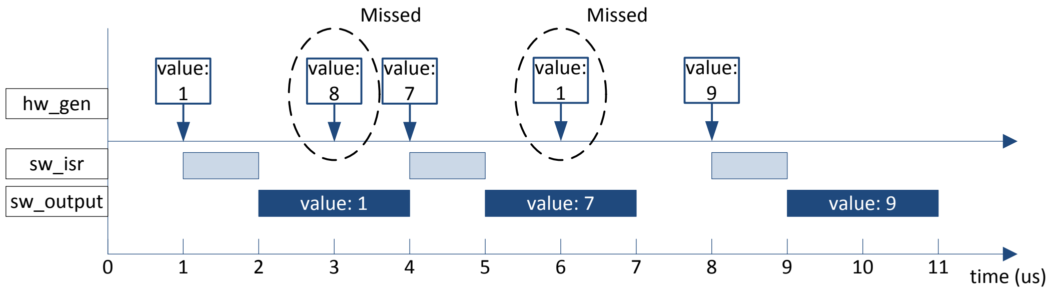


Figure 2-10. Non-pre-emptible execution

software Interrupt Service Routine (ISR) cannot serve the hardware interrupt. Figure 2-10 shows this phenomenon, in which interrupts are missed at time points 3 μs and 6 μs. In Chapter 3, we will present the solution to this problem.

2.3 RTOS Modelling and Simulation in System-level Design

In recent years, RTOS modelling and simulation have been proposed as important embedded software validation techniques in the context of embedded systems system-level design. This section surveys related SLDL-based RTOS modelling and simulation research. There are several criteria by which to classify and compare different approaches, for instance:

- *By application scope:* Various RTOS models have been developed for high-level abstract software simulation [112] [72] [113], native-code software simulation [119] [87], HW/SW co-simulation [120] [121] and system-level design flow refinement research [43].
- *By software simulation methods:* As already introduced in Section 1.5, there are two main software simulation approaches being used in system-level design: Instruction Set Simulation and behavioural simulation. Accordingly, researchers develop *ISS-based RTOS models* for complete and accurate validation of final software implementation [108] [109], whilst *behavioural RTOS simulation models* are more widely used for fast and flexible software early exploration.
- *By functional accuracy:* According to the functional accuracy of the RTOS model, RTOS simulation models are summarised into three categories in [77]: *abstract OS models* that rely on communication primitives and scheduling service by the underlying SLDL kernel, *virtual OS models* that mimic the functionalities of the final OS but with independent implementation code, and *final implementation OS models* which can be used in ISS simulation. It should be noted that the definition of *abstract OS models* seems to overlook the fact that SLDLs fail to supply enough RTOS capability natively.

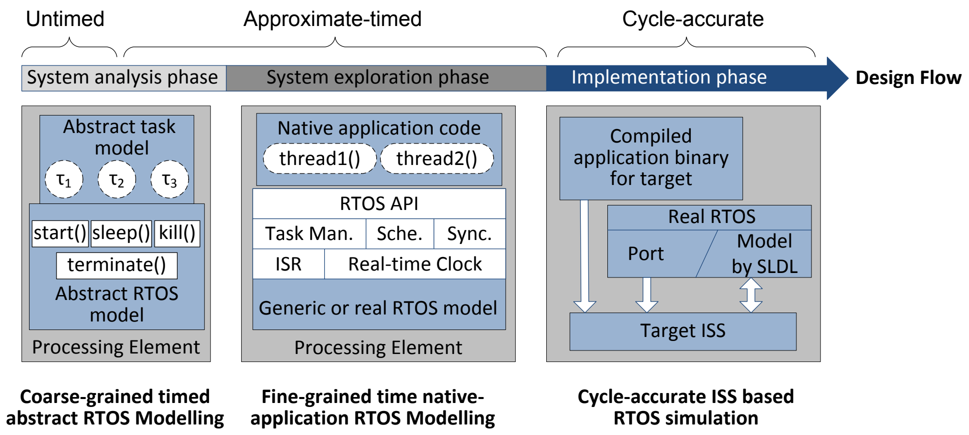


Figure 2-11. Three types of RTOS simulation models

We categorise and analyse RTOS modelling and simulation research based on their timing and functional accuracy levels, as well as their applicability stage in an embedded system design flow. Referring to Figure 2-11, most existing methods fall into three main categories: coarse-grained timed abstract RTOS modelling, fine-grained timed native-code RTOS modelling and ISS based RTOS simulation. The "coarse-grained" and "fine-grained" criteria refer to the timing accuracy level of software models (including both the RTOS aspect and SW applications), and they both belong to the domain of behavioural software simulation.

2.3.1 Coarse-Grained Timed Abstract RTOS Modelling

Abstract RTOS modelling and simulation focus on early design phases, such as system specification, system analysis and SW/HW pre-partitioning stages. At this time, the target platform is undetermined and software code has not been implemented. Also, it is not possible to presume specific RTOS API services in the system-level simulation framework before enough decisions have been made regard the system architecture. However, general structures and execution mechanisms of the RTOS model should still be not far from real RTOSs, in order to make sure that the RTOS model has a practical usability for real-time software design. Abstract RTOS modelling is supposed to provide extensible real-time system modelling capabilities and be fast to be changed in evolving simulation loops.

In this approach, software applications are normally organised as a collection of abstract tasks associated with coarse-grained temporal properties, e.g., period, deadline, offset, and execution times [112] [72]. Periodic, aperiodic, and sporadic tasks are typically explicitly defined by different timing characteristics, which include the main information obtained by the RTOS in order to handle a task. A qualified abstract RTOS model needs to at least provide priority-based preemptive scheduling services and basic primitives to control the "start" and "termination" of a task. This feature is essential for a practically usable RTOS model in order to overcome the previously-mentioned limitations of underlying SLDL bases. A task's execution cost is usually modelled by the *wait-for-delay* statement. The delay interval of every task instance (i.e., a job) is either statically annotated by estimation or dynamically randomised by some statistical theories, e.g. uniform distribution [8]. The "delay-measurement and back-annotation" timing method is also proposed in [113] [43], but it is applied at a coarse-grained timing granularity (i.e., task-level). Inter-task synchronisation for resource sharing, communication services and interrupt handling are usually not adequately considered in this kind of model. The advantage of this method is the fast simulation speed, since applications and RTOS are highly abstract models. The main drawbacks of this method are low timing accuracy (coarse time annotations for applications and inadequate modelling of RTOS timing overhead) and incomplete modelling capability of RTOS functionalities. Besides, in most existing research, there is a lack of SW/HW interaction modelling, and hardware parts of a CPU subsystem are not explicitly modelled either. This means that software application tasks and the abstract RTOS model form the software PE model by themselves.

Gerstlauer et al. present an early SpecC-based abstract RTOS model in order to integrate software scheduling support in the TLM model refinement flow [43] [122]. This RTOS model provides 16 basic primitives to support task management and scheduling. RTOS timing overheads are not mentioned sufficiently. Besides, it uses the imperfect *wait-for-delay* time advance method, so interrupt handling cannot be accurately modelled and the timing accuracy is limited by the minimal resolution of time annotations. A subsequent work [123] resolves this initial HW/SW synchronisation problem by using an improved *wait-for-delay* method named "Result Oriented Modelling". In recent, Zabel et al. [124] use the SystemC SLDL to implement an abstract RTOS model where most parts are based on the work of [43]. They solve the HW/SW timing synchronisation problem by using the SystemC *wait-for-event* method, which is also utilised in our research in this thesis.

Early work by Madsen et al. presents a SystemC-based abstract RTOS model [112], which is further extended for MPSoC simulation [8] and NoC simulation [125]. The basic idea is to decompose an embedded system model into three compact sub-models: the task graph model, the scheduler model, and the link communication model. The scheduler model provides both fixed-priority scheduling (e.g., rate-monotonic priority assignment) and dynamic-priority scheduling (e.g., EDF) services by using three primitives (i.e., run, pre-empt, and resume) to manage tasks. The task model is characterised by coarse-grained temporal information or estimates, e.g., WCET, BCET, period, deadline and offset, but without any functionality code. This RTOS model is a good basis for high-level system exploration, but it also has some limitations. Firstly, RTOS service overheads are not included in the model. Furthermore, its task state machine model is different from that usually found in a typical real-time kernel, and the task model is also too simple to mimic a real system. Finally, its link communication model heavily relies on the SystemC Master-Slave message-based communication library for both software internal and inter-module communications, whose behaviours are different from common RTOS synchronisation and communication mechanisms.

Hessel et al. describe an abstract RTOS model in SystemC SLDL for use in the embedded systems refinement flow [113]. Both the structure and implementation of this RTOS model is similar to Madsen's model; hence, it is also weak due to simplistic task modelling and incomplete RTOS service modelling.

Moigne et al. propose a generic RTOS model for real-time systems simulation [114]. This work has the advantage of considering timing overheads of three RTOS services, i.e. context-load time, context-save time and scheduling algorithm duration. Nevertheless, this work does not address task functionality modelling, interrupt handling and synchronisation modelling.

Hastono et al. use an abstract RTOS model for real-time scheduling assessments [126] and embedded software simulation [72]. The RTOS model provides basic task management services similar to the models of Gerstlauer and Madsen.

Various static and dynamic scheduling policies, e.g., event-driven, time-triggered, fixed-priority RMS, dynamic-priority EDF, etc. are integrated in order to evaluate and compare different task scheduling decisions. The functionality of a task is decomposed into non-pre-emptive atomic actions and pre-emption is assumed to happen only at boundaries of atomic actions. Consequently, this pre-emption model cannot simulate interrupts realistically.

Hartmann et al. present an abstract RTOS simulation model as a part of their SystemC-based system synthesis design flow [127]. They model software on a generic run-time system rather than directly modelling existing RTOS services, i.e., all conventional software synchronisation and inter-task communication mechanisms are modelled by the shared objects method. The intention is to inherit their previous hardware modelling work and thus allow a seamless high-level SW/HW specification environment.

2.3.2 Fine-Grained Timed Native-Code RTOS Simulation

Native-code RTOS models are used to support simulation of high-level software functional code at the system exploration phase, when the target platform and the RTOS are in the process of being selected, and application software is under development.

Its timing accuracy has been improved compared to abstract RTOS models. Software execution delays are measured and annotated in models at some finer granularities (e.g., function level, block level, and source code line level), so timing accuracy becomes a major focus in this approach. This kind of RTOS simulation model often supplies comprehensive and specific services, and contains some timing overhead information. In some research, a real RTOS is modelled [128] [87] [129] [130], whilst some other works attempt to build a generic RTOS model [131] [12] [130]. Because of its much faster simulation speed (two or three orders of magnitude faster than ISS simulation [128]) and acceptable loss of modelling accuracy, fine-grained timed native-code RTOS and software simulation is proposed as the counterpart of TLM HW and communication modelling.

Jerraya's group performs a series of studies addressing native software simulation in SoC HW/SW co-simulation, and presents two different typical software simulation methods in [121] [128] [130], respectively. In [121] [128], they build a software simulation model (including OS, application software, and a bus functional model) annotated with timing delays and run it as a host Unix process, whilst, the hardware part is modelled in SystemC SLDL. The communication between software and hardware is implemented with Unix IPC methods, such as shared memory and signal. In order to solve the HW/SW synchronisation problem, they propose a "variable timing granularity" method to simulate interrupts by trading off the simulation performance with the timing accuracy. In [130], they use a different way to model the software part, where application tasks are scheduled by an OS model by using the multi-threading functionality of the host OS, and then the whole software part is integrated into a SystemC HW/SW co-simulation framework. Both a pre-emptive FIFO based scheduler and a real eCOS RTOS are implemented in the OS model library. With the same RTL model on the HW side, compared to the cycle-accurate ISS software simulation, the co-simulation performance with native RTOS simulation is reported as three orders of magnitude faster, and the simulation accuracy achieves 86% of the ISS. In general, from the RTOS modelling aspect, this research has the advantage of considering various detailed RTOS service overheads and accurately modelling HW/SW interactions (e.g., interrupt handling and memory access). However, their models sometimes utilise the underlying host OS services, which may deteriorate the portability and negate SLDL's intent as a homogeneous modelling framework.

A SystemC-based native simulation model for a commercial Texas Instrument RTOS is presented by He et al. in [87]. It models common RTOS services such as task management, priority-base scheduling, task synchronisation, I/O, and inter-process communication with timing overheads estimated from the target processor's benchmark sheet. This simulator uses an event time-stamp prediction method for interrupt modelling, which is based on an assumption that application tasks can report happening times of their future synchronisation events to the kernel. This tight requirement requires pre-requisite analysis of the whole system and may hence restrict its usability.

A HW/SW co-simulator that includes a special-purpose μITRON 4.0 RTOS model is introduced in [129]. It natively simulates a complete μITRON RTOS model with application software on the host computer. For the HW aspect, C/C++ or HDL HW models can be included in the simulator and can communicate with the software simulator by using Windows IPC methods. This work has a drawback in that its simulated clock relies on the host OS clock, i.e., it is untimed from the perspective of target software simulation. Furthermore, host IPC methods may bring an extra and unexpected simulation overhead.

Chung et al. describe a generic SystemC-based RTOS model which is oriented for MPSoC simulation in [131]. Its generic RTOS and POSIX like API models support native application code to execute with RTL/TLM HW models. However, its RTOS task machine model is lacking in modelling real-time synchronisation mechanisms. And it also uses a polling method to check interrupt events in every clock-cycle, which may result in undesired consequences that interrupt latency depending on the length of a simulation clock cycle, i.e., it is an "annotation-dependent" HW/SW timing synchronisation approach.

Posadas et al. develop a comprehensive POSIX compliant RTOS simulation model on top of SystemC in [12] and apply a dynamic delay annotation method by assigning each C++ operator with a corresponding target-platform execution cost. In [132], they address the global variable accessing problem and propose three joint solutions. Their first method is a fine-grained annotation technique (see Section 3.1.2); the second method can guarantee a correct functional simulation result but still has the delayed interrupt handling deficiency due to its *wait-for-delay* method (see Section 3.1.1); the third method is satisfactory and similar to a method used in this thesis (see Section 3.2.3.2), but it focuses on abstract software programming models by providing a special primitive channel to protect global variables.

2.3.3 ISS-based RTOS Simulation

ISS-based RTOS simulation can be used in a HW/SW co-simulation framework when embedded software has been fully implemented. The high accuracy, low simulation performance speed and late availability are its contradictory characteristics, and therefore make it applicable for the late development phases where high reliability and high accuracy are the main focuses of simulation. Finished software source code is cross-compiled and simulated in a cycle-accurate instruction set simulator that represents the target processor's behaviour. The ISS is usually wrapped in a SLDL module. A real RTOS is often ported in the ISS to supervise software application. Other SLDL-based HW component models are connected with the ISS-wrapper model by the SLDL communication backplane to achieve a co-simulation. This co-simulation approach is similar to the traditional cycle-accurate embedded system co-simulation approach, which uses HDLs to model hardware components at RTL level and uses the ISS to execute software. Compared with the conventional approach, this unified system-level HW/SW co-simulation approach can enhance design productivity by raising the abstraction level of HW models and then gain simulation speedup to some extent. However, this may somewhat contradict the system-level design concept of raising abstraction level for more efficient design space exploration, because it does not change the software simulation method.

Chevalier et al. integrate a μC/OS-II RTOS on an ARM ISS which is wrapped by a SystemC model [108]. Their modelling framework constructs a conversion interface between SystemC API and the μC/OS-II API in order to let the RTOS schedule SystemC-based application software processes. Benini et al. build a SystemC-based multi-processor co-simulation platform [109] that uses SystemC to wrap several cycle-accurate ARM ISS simulators to run multiple cross-compiled μClinux kernels and software applications.

To trade-off simulation speed with accuracy, the approaches in [120] and [133] take a different approach by running software application on the ISS whilst building a RTOS model on top of the SLDLs. However, [120] only supplies task preemption services and considers limited RTOS timing overheads.

2.3.4 The Proposed RTOS Simulation Model

In this thesis, a SystemC-based generic RTOS modelling and simulation approach will be presented. Essentially, it falls into the native-code RTOS simulation category, but also integrates some abstract RTOS modelling features in terms of supporting abstract task models.

Compared with existing research, the proposed RTOS simulation model embodies the mixed timing software modelling idea (in Section 3.2) by supporting hybrid abstract task models and native-code task models in a single simulator, in order to enhance modelling flexibility and expand application domain.

Furthermore, the generic RTOS model's functionality is determined by surveying some popular RTOS products and standards. It aims to support more realistic software simulation than other simplistic RTOS models. Most importantly, the high simulation performance and good timing accuracy are preserved at the same time in the RTOS simulation model because of the underlying Live CPU Model. The details of this model will be described in Chapter 4.

2.4 Summary

In this chapter, some basic concepts in transaction-level modelling research have been introduced. The focus is to survey current abstraction levels, timing degrees, and communication modelling in the TLM research context, in order to inspire our research on real-time software behavioural modelling and simulation that can be seen as the TLM software computation aspect. However, we noticed that existing TLM abstraction levels and models are not appropriate and are insufficient for real-time software modelling. Thus, in the next chapter, we will define some real-time embedded software simulation models in the context of SystemC based TLM research.

Subsequently, SystemC language constructs and the co-operative simulation kernel were introduced. A SystemC-based HW/SW system example model was presented. This demonstrates how the use of uninterruptible *wait-for-delay* statements may lead to missing external interrupts in simulation, which highlights a problem to be solved.

Some state-of-the-art RTOS modelling approaches and simulation models for SLDL-based system-level design were surveyed also. They are classified into three categories depending on timing and functional accuracy levels. Among them, the abstract RTOS modelling approach and the native-code RTOS modelling approach are of concern to this thesis. We aim to propose a generic mixed timing

RTOS simulation model with improved features in terms of timing accuracy, functionality, and modelling flexibility.

Chapter 3

Mixed Timing Real-Time Embedded Software Modelling and Simulation

In previous chapters, SLDL-based software behavioural modelling and simulation have been introduced for validation of real-time embedded software (applications and RTOS) in the context of system-level and TLM design. Three objectives can be been identified as key requirements:

1) Fast simulation performance compared to ISS simulation.
2) Flexibly modelling software functions and their timing delays.
3) Accurate simulation results in terms of both functional and timing aspects if they are both modelled.

This chapter presents a SystemC-based mixed timing software behavioural modelling and simulation approach (referred to as the mixed timing approach hereafter) and the associated Live CPU Model. A basic assumption of this approach is that the proposed simulation models are applicable after HW/SW partitioning and applications can be divided into tasks. If multiple tasks need to execute concurrently and pre-emptively, then a behavioural OS model should be included in simulation.

Referring to Figure 1-7 (Page 18), a SLDL-based behavioural simulation approach includes three main steps, i.e., schedule of processes, native execution of a process in zero target time, and target delay time advance according to annotation. The first and second steps are the *functional aspect* of behavioural modelling and simulation, whereas the third step refers to the *timing aspect*. According to the OS-based task simulation model assumption, in this thesis, the functional aspect of modelling and simulation is relatively fixed as software functions are wrapped in OS tasks and the OS model controls their execution order.

In this chapter, the mixed timing approach mainly seeks answers to the above three key requirements from the timing perspective of modelling and simulation, but also considers software functional modelling. Separating timing issues in modelling and preserving high timing accuracy in simulation are two characteristics of this approach. The conventionally annotation-dependent SLDL-based software modelling and simulation is treated as two partially separated stages[1]:

1) The *timing modelling* step mainly refers to annotating target platform execution costs (time delays) and defining time advance points in software task code, when SLDL-based software task models are being built.
2) The *timing simulation* step mainly refers to advancing the target simulated clock according to those annotated time delays, when these SLDL-based software task models are dynamically simulated upon a SLDL simulation engine.

This approach allows flexibility in software timing modelling, achieves good timing accuracy in software timing simulation, and maintains a high simulation speed. It has following basic features:

- It utilises multiple-grained software timing information and variable annotation methods for software models at the modelling stage (in Section 3.2). It facilitates model builders and simulation users for using a variety of available means of timing estimation sources, and allows building mixed timing simulation models with varying timing precision for workload and accuracy trade-off.
- It preserves high hardware interrupt handling and software pre-emption timing accuracy within a certain bound at the timing simulation stage. The Live CPU Model (in Section 3.3) is introduced to supervise software timing simulation and monitor external interrupts in simulation. By excluding possible interrupt disabled cases (e.g., critical section code), the Live CPU Model can interrupt current software simulation (i.e., stop its delay time ad-

[1] It is necessary to point out that the separation of timing issues in modelling and in simulation is “partial”, because these two aspects cannot be totally decoupled in back-annotated timed software simulation.

vance) as soon as an IRQ is caught, and resume remaining time advance for the pre-empted task at the correct time point, just like real CPU execution. Compared to some conventional pre-emption simulation approaches that trade off simulation speed for accuracy, the simulation performance of the proposed approach is not sacrificed whilst timing accuracy is sustained.

- It offers varying system simulation similarity and run-time information observability. By configuring the Live CPU Simulation Engine with the variable-step and the fixed-step time advance methods, the users can make trade-offs between simulation similarity, information observability and simulation performance (in Section 3.3.4).

Figure 3-1 illustrates the mixed timing software modelling and simulation approach. In the figure, various grained delay time slices, e.g., task-level, function-level, and source code line-level, can be annotated to the same software model at the timing modelling stage. The Live CPU Model uses these different sizes of time annotation statements to progress the target simulated clock. In this mixed timing approach, the granularity of a time annotation does not interfere with the dynamic timing accuracy of HW/SW synchronisation (i.e., interrupt handling) in timing simulation. Interrupt handling does not need to wait until a delay slice has totally elapsed, i.e., reaching a delay boundary. On the contrary, an ISR can pre-empt current running software task as soon as an external interrupt happens, just like the situation at the time point t_1. After an ISR finishes execution at time t_2, the pre-empted software task is resumed and the remaining value of the previously-interrupted delay annotation slice is also continued.

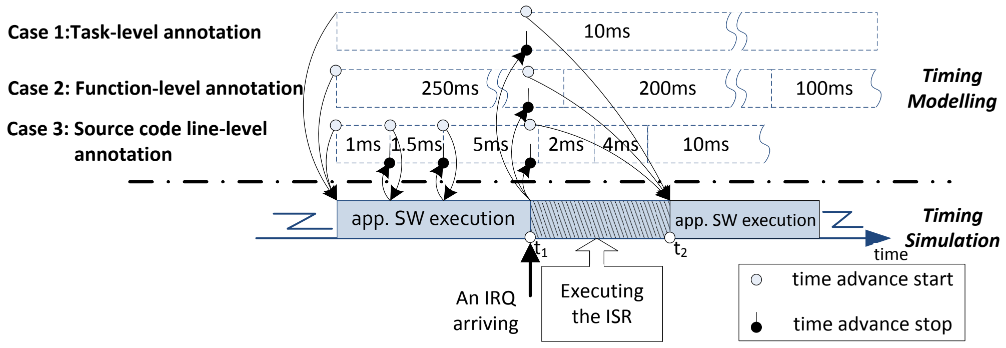


Figure 3-1. Mixed timing software modelling and simulation

In the reminder of this chapter, some problems and approaches regarding timing issues in existing SLDL-based software modelling and simulation will be surveyed (Section 3.1). Section 3.2 describes the mixed timing approach in detail, in terms of various timing techniques for software modelling and simulation. The Live CPU Model is introduced in Section 3.3, which is not only important for timing accurate pre-emptive software simulation but also meaningful for extending the software processing element model to the TLM modelling context. Finally, evaluation metrics and experiments are presented in Section 3.4 and Section 3.5 respectively, in order to demonstrate benefits of the proposed approach. Section 3.6 will summarise this chapter.

3.1 Issues in Software Timing Simulation

This section briefly surveys some timing issues in related SLDL-based behavioural software timing simulation approaches. Concerning two important simulation timing characteristics - *timing accuracy* and *simulation performance*, we will introduce their capabilities and also their deficiencies.

3.1.1 Annotation-Dependent Time Advance

In SLDL-based real-time software behavioural simulation, a software model executes its function code on the host CPU architecture, which does not have any timing correlation to its execution cost on the target CPU. Accordingly, the SLDL *wait-for-delay* function (e.g., `wait(time)` in SystemC) is used to model software timing behaviour on the target [72] [43]. On the one hand, it adds target-platform delay annotations in software models; on the other, it also progresses the simulated clock. Hence, software timing modelling (adding delays) and timing simulation (using delays) are not separated in this kind of time advance approach.

However, the uninterruptible characteristic of the *wait-for-delay* statement is problematic, with the “annotation-dependent” software time advance method becoming an issue in software simulation. Figure 3-2 shows examples regarding *wait-for-delay* statements in software simulation. There are two application tasks (i.e., *$task_1$* and *$task_2$*) and an ISR that serves an external hardware interrupt. The

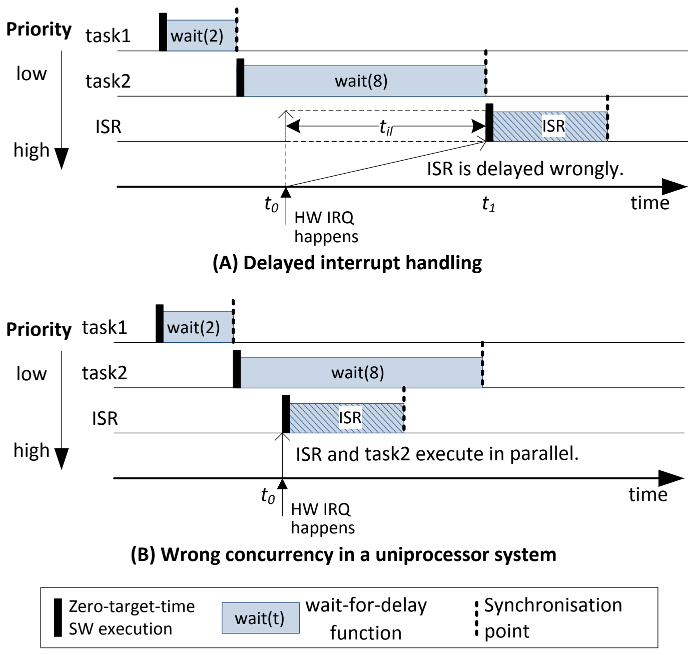


Figure 3-2. Annotation-dependent time advance method

interrupt event should be processed as soon as possible once it occurs, just like the normal situation of a real-time system.

In simulation, once a *wait-for-delay* statement is invoked, the value of software delay time will be totally consumed without a possibility of interruption. Consequently, *task*$_2$ can only execute after the *wait-for-delay* statement of *task*$_1$ is finished. In such cases, once an interrupt event is raised by a hardware module during this delay duration, e.g., at time t_0 in the example, it may lead to two problematic simulation phenomena depending on modelling methods.

Figure 3-2 (A) shows the first possible problem: "delayed interrupt handling". Because the *wait-for-delay* statement of the running *task*$_2$ cannot be interrupted, the ISR can only start when the current delay time slice finishes at time t_1. It can be observed that the ISR is wrongly postponed rather than serving the interrupt request at the expected time point. Under such circumstances, both software tick scheduling and the HW/SW synchronisation (i.e., interrupt handling) can only occur at the boundaries of delay annotations. Simulation time advance is dependent on the granularity of annotation. In simulation, both the pre-emption latency and the interrupt latency t_{il} are unrealistically restricted by length of *delays* that are defined at the modelling stage. Under the worst circumstances, the latency equals the largest time delay value. This time advance method makes it hard to model a pre-emptive real-time system or a real interrupt handling procedure.

Considering the second case in Figure 3-2 (B), the model programmer may choose to start the ISR as soon as it is raised. However, this brings a critical problem in that the ISR and the existing task execute in parallel in simulation, i.e., they are both at the RUNNING state from the perspective of CPU scheduling. Obviously, in a uniprocessor system, this situation cannot occur. For this simulation problem, programmers therefore need to correct the affected time delay in order to serialise software execution with right timing behaviour. This problem resembles the conventional optimistic co-simulation that may require time rollback and re-execution.

In the following Sections 3.1.2 - 3.1.4, three existing techniques will be introduced, which aim to remedy this annotation-dependent time advance problem. More importantly, we will present our complete solutions the "mixed timing approach" and the "Live CPU Model" in the rest of this chapter.

3.1.2 Fine-Grained Time Annotation

An intuitive means of solving the above-mentioned "delayed interrupt handling" problem is to use more *wait-for-delay* statements with finer grained delay time slices [132]. In the context of mixing untimed and timed TLM models, Ghenassia et al. propose a similar idea to define some pseudo synchronisation points in untimed TLM models where other timed TLM threads can detect interrupt more frequently but without advancing the simulated clock [3].

Figure 3-3 illustrates this fine-grained time advance approach. The software model code is divided into small segments. The granularity of *wait-for-delay* time annotations is limited to an acceptable extent at the model building stage. This means that there are more time advance points in the models. As a consequence, interrupt events can be frequently checked in order to more realistically represent the interrupt latency in the simulation. Unfortunately, the HW/SW timing synchronisation accuracy is enhanced at a cost of:

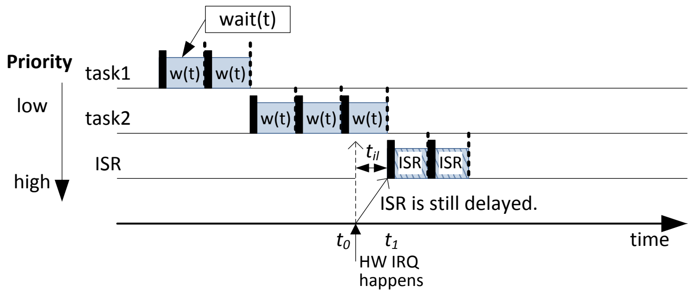


Figure 3-3. Fine-grained timing annotation

- A large quantity of time profiling work and annotation statements when building simulation models
- More *wait-for-delay* statements mean frequent SLDL simulation engine context switches and thus large overhead.
- Interrupt handling may still be delayed, although the delay time is minor because of fine-grained annotation slices.

Compared to this approach, fine-grained time annotation is also supported in the proposed mixed timing approach. However, this is not a necessary condition to ensure high timing accuracy of HW/SW synchronisation. The HW/SW synchronisation problem is tackled by the Live CPU Model in this thesis, which fully relaxes the limitation of the annotation-dependent problem. The Live CPU time advance approach can maintain the same and high timing accuracy for software pre-emption and hardware interrupt handling at simulation runtime, no matter what the time annotation granularity is. Hence, less overhead can be expected than in the fine-grained annotation approach.

3.1.3 Multiple-Grained Time Annotation

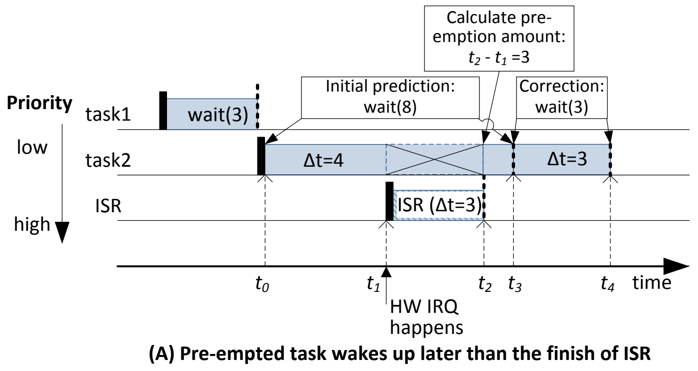
For UNIX process-based native-code software and SystemC-based hardware co-simulation, Bacivarov et al. discuss trade-offs between simulation performance and timing accuracy by adopting multiple-grain HW/SW timing synchronisation [121]. The basic idea is to reduce or increase the granularity of time annotations depending on the desired timing accuracy of interrupt handling.

The approach in [121] uses asynchronous co-simulation, in which software and hardware simulators are two separate UNIX processes. The software and hardware simulators manage their local clocks separately and exchange timing information via inter-process communication. It is known that IPC overheads may contribute a large portion of simulation time and affect the simulation performance. The HW/SW timing synchronisation in [121] can be seen as a compromise of the classic *conservative* algorithm [134]. Therefore, HW/SW timing synchronisation accuracy may not be guaranteed when using coarse-grained granularity of timing annotations.

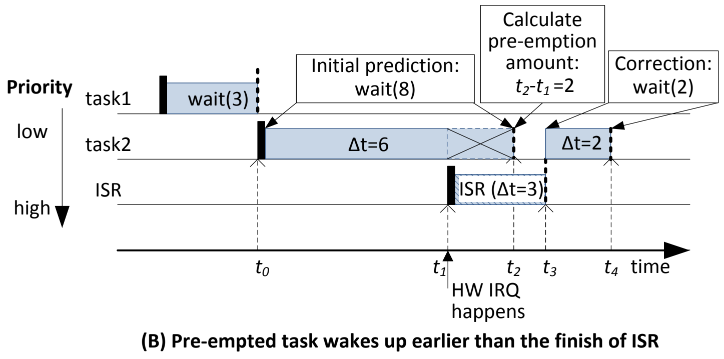
3.1.4 Result Oriented Modelling

To solve the problem in Figure 3-2 (B), Schirner et al. introduce their time correction method Result Oriented Modelling for SLDL-based pre-emptive software simulation [123]. It still uses the uninterruptible *wait-for-delay* statement for time annotation and clock progress, but it can virtually interrupt a *wait-for-delay* statement in order to enable pre-emption at any time point. In the case of an interrupt event, the ROM-based RTOS model first records pre-emption timing information. Then, after the finish of both the existing *wait-for-delay* statement and interrupt disturbance, it will finally make a new corrective *wait-for-delay* statement for the affected time advance step.

Figure 3-4 illustrates two possible interrupt handling scenarios in the ROM approach. In case (A), the application *task*$_2$ begins to run at t_0 and then calls a *wait-for-delay* statement ranging 8 time units from t_0 to t_3, so as to mimic its execution timing cost. This step is called an “initial prediction” in ROM, because it simply assumes that the *task*$_2$ can solely occupy the CPU during this *wait-for-delay* time interval. However, at t_1, a hardware interrupt request is detected. Thus, the RTOS scheduler dispatches a corresponding ISR as the new RUNNING task to pre-empt the lower-priority *task*$_2$. Herein, the RTOS model changes OS status of *task*$_2$ from RUNNING to READY, and records the pre-emption time stamp in the Task Control Block (TCB) of *task*$_2$. Afterwards, the ISR executes some functions and begins its *wait-for-delay* statement. During the time duration from t_1 to t_2, although both the ISR and *task*$_2$ are suspended by *wait-for-delay* statements, their task states are distinct in the sense of RTOS task management. When the ISR finishes at t_2, RTOS changes OS status of *task*$_2$ to RUNNING again. More importantly,



Calculate pre-emption amount: $t_2 - t_1 = 3$
Initial prediction: wait(8)
Correction: wait(3)
Priority
low
high
task1
task2
ISR
wait(3)
Δt=4
Δt=3
ISR (Δt=3)
t_0
t_1
t_2
t_3
t_4
time
HW IRQ happens
(A) Pre-empted task wakes up later than the finish of ISR



Calculate pre-emption amount: $t_2 - t_1 = 2$
Initial prediction: wait(8)
Correction: wait(2)
Priority
low
high
task1
task2
ISR
wait(3)
Δt=6
Δt=2
ISR (Δt=3)
t_0
t_1
t_2
t_3
t_4
time
HW IRQ happens
(B) Pre-empted task wakes up earlier than the finish of ISR

Figure 3-4. The Result Oriented Modelling approach

RTOS calculates how long *task*$_2$ is pre-empted as its new delay time interval, namely t_2-t_1. The initial prediction of *task*$_2$ ends at t_3 and the new corrective *wait-for-delay* statement is then issued immediately.

The scenario of Figure 3-4 (B) is slightly more complex than the previous case. In this example, the initial prediction of the pre-empted *task*$_2$ finishes at t_2 that is earlier than the ISR's *wait-for-delay* finishing time t_3. This means that *task*$_2$ will wake up and needs to be processed immediately so as not to execute its subsequent model code. The RTOS model firstly calculates the pre-emption interval of *task*$_2$ as t_2-t_1 and then indefinitely suspends *task*$_2$. The ISR finishes at t_3 while *task*$_2$ is scheduled by the RTOS to resume again. A new *wait-for-delay* statement that uses the before-calculated pre-emption interval as the delay parameter is released in order to revise time advance for *task*$_2$.

In summary, a ROM simulation procedure contains three steps: 1) Execution of an initial *wait-for-delay* statement; 2) Collection of any disturbing events and update of delay information; 3) Making a corrective *wait-for-delay* statement. By this approach, the sequential software concurrency can be realised for a uniprocessor system model. The good timing accuracy of HW/SW synchronisation and software pre-emption is successfully achieved from the perspective of virtually pre-empting *wait-for-delay* statements in SLDL-based simulation.

The "black box" simulation concept is another worthy point emphasized by ROM [135]. It prefers to only present adjust *end results* (e.g., *termination time* and *final state*) of a simulation process rather than model and reveal any internal state changes to users. For example, during a *wait-for-delay* interval of a software task, if multiple interrupts happen, the ROM will collect the disturbances together and only issue one corrective *wait-for-delay* statement. This "black box" concept has positive and negative aspects:

1) It brings the advantages of speeding up simulation performance by hiding intermediate states and maintaining timing advance accuracy by considering interference from hardware interrupts.
2) In ROM, it is difficult to maintain the similarity of middle state changes to a real execution at certain circumstances. This is an inevitable compromise. Because ROM uses the inherently uninterruptible *wait-for-delay* functions, there is no way to cancel or postpone a *wait-for-delay* statement once it begins. Hence, the timing point when a model process wakes up from a *wait-for-delay* duration is also unchangeable either. This feature may bring a defect in simulation traces, incurring an amount of simulation overheads. In ROM, the pre-empted task may wake up at unexpected time points as long as its *wait-for-delay* time period is finished. Referring to Figure 3-4 (B) for instance, $task_2$ wakes up at t_2 and calls for processing from the RTOS model. However, from the perspective of OS multitasking management, $task_2$ should not initiatively trigger the OS to process it at this time point because it has been pre-empted. This phenomenon will result in an unnecessary RTOS processing procedure, a SLDL simulation kernel context switch, and a consequential simulation overhead.
3) The ROM approach aims to collect all interrupts that happen during a *wait-for-delay* time advance interval and launches a new *wait-for-delay* statement for the affected task to correct its delay time. In the best case, only one new corrective *wait-for-delay* statement is needed to revise an affected

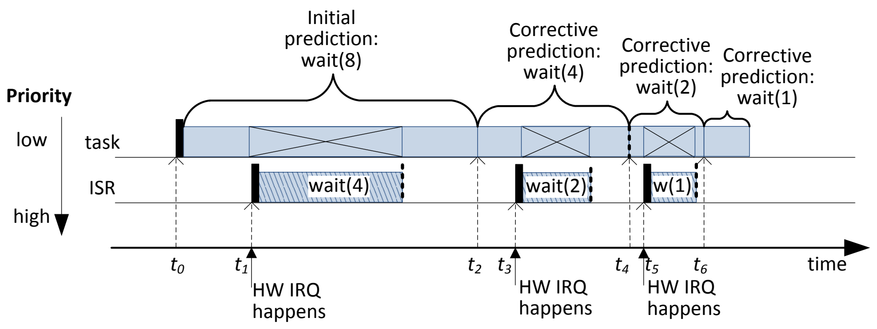


Figure 3-5. Successive corrective wait-for-delay statements

time advance step. Whereas, the possibility should be taken into account that another pre-emption event may happen during a corrective *wait-for-delay* interval. This means that one more successive corrective *wait-for-delay* statement is required. Figure 3-5 shows such an example. In fact, the exact number of *wait-for-delay* statement may vary depending on the number of pre-emption events and where they happen, which are dynamically determined in simulation. It may be very costly to correct successively interrupted time advance steps in some conditions.

3.2 The Mixed Timing Approach

In this section, a mixed timing approach is proposed. It achieves a similar timing accuracy level to the ROM approach. However, the two approaches are conceptually different because of their underlying time advance methods and in addition the proposed approach can generate a better simulation trace without the above "inappropriate wake up" problem.

Concerning the fundamental problem of handling interrupts during an ongoing time advance step, mixed timing approach uses the *wait-for-event* mechanism to ensure that a pre-empted task only wakes up upon receiving an event issued at the correct time point. Only one *wait-for-event* statement is called by a software task in a time advance step. We do not need to call a new *wait-for-event* statement for the pre-empted task. Hence, the SLDL processes (wrappers of software tasks) do not frequently change between "suspending" status and "wake-up" status. Consequently, a simulation speedup can be expected due to a fewer number of costly simulation kernel context switches.

The mixed timing approach is a general approach oriented to SLDL-based real-time software (including tasks and the RTOS) behavioural modelling and simulation. According to the aforementioned taxonomy of system-level software and RTOS simulation research in Section 2.3, it can be applied to both coarse-grained timed abstract software modelling and fine-grained timed native software modelling. In this section, this modelling and simulation approach is implemented by typical SystemC language constructs, mainly the *wait-for-event* method (see Section 2.2). Because of the similarity between SystemC and SpecC SLDL, it is promising to be generalised to the SpecC context.

3.2.1 Separating and Mixing Timing Issues

In SLDL-based behavioural software simulation, multiple-facet/level timing models can be written and simulated in the same discrete-event software simulation environment, e.g., the SystemC framework. These kinds of approaches can be divided into two parts, i.e., modelling and simulation.

- In modelling, functional and timing characteristics (time delays) of target software computation components are described by SystemC language. In this research, specifically, software applications are divided to tasks and each task is then mapped to a SystemC process. The results of this modelling process are SystemC process models for executable simulation purposes.
- In simulation, these models are compiled together with the SystemC simulation kernel and natively executed on a host computer in order to help software designers to observe behaviour of the target software system, validate different design strategies, and measure the mean or extreme simulation data for analysis.

In real-time embedded software design, timeliness is a first-class factor to determine the accuracy of modelling and simulation. The mixed timing approach puts focus on timing issues of above two aspects:

1) Timing issues in modelling: This aspect is concerned about timing issues that are statically determined at the model building stage. It relates to various jobs that add time delays for software computation models, e.g., define timing styles of models, choose sources of timing information, apply variable annotation granularities, annotate timing information into model code, and insert time advance points in models.
2) Timing issues in simulation: This refers to timing issues that are dynamically behaved at simulation runtime. It relates to jobs that use time delays for simulation time advance, e.g., simulate target timing behaviour for software models, progress the simulation clock, and process interrupts.

In the following, this mixed timing approach is explained with regard to various issues in relation to aspects of timing modelling (Sections 3.2.2 - 3.2.6) and timing simulation (Section 3.2.7). Besides, the Live CPU Model is an essential basis of this approach (Section 3.3).

3.2.2 TLM Software Computation Modelling

Before presenting any detailed timing modelling and simulation methods, we clarify general guidance for defining software timing simulation models and the relevance to existing TLM research.

In Section 2.1.1, abstraction levels in SystemC TLM modelling are reviewed, with this thesis concerned with software computation modelling in the general context of SystemC TLM research. Consequently, it is natural to explore the possibility of inheriting some common concepts from existing TLM proposals. For example, the OSCI TLM standard defines the PV and PVT abstraction levels based on criteria such as the transmission method and the timing granularity of a communication transaction. Baklouti et al. propose the application of the PV and PVT concepts to refine software communication [6]. As shown in Figure 3-6 (A), its horizon focuses on using TLM synchronous and asynchronous interfaces for abstract software inter-module (between initiators and targets) communication. In [9], Dömer proposes to define TLM computation abstraction levels based on the concept of separating functionality and timing. Referring to Figure 3-6 (B), four levels are identified in a modelling flow for software that runs on programmable

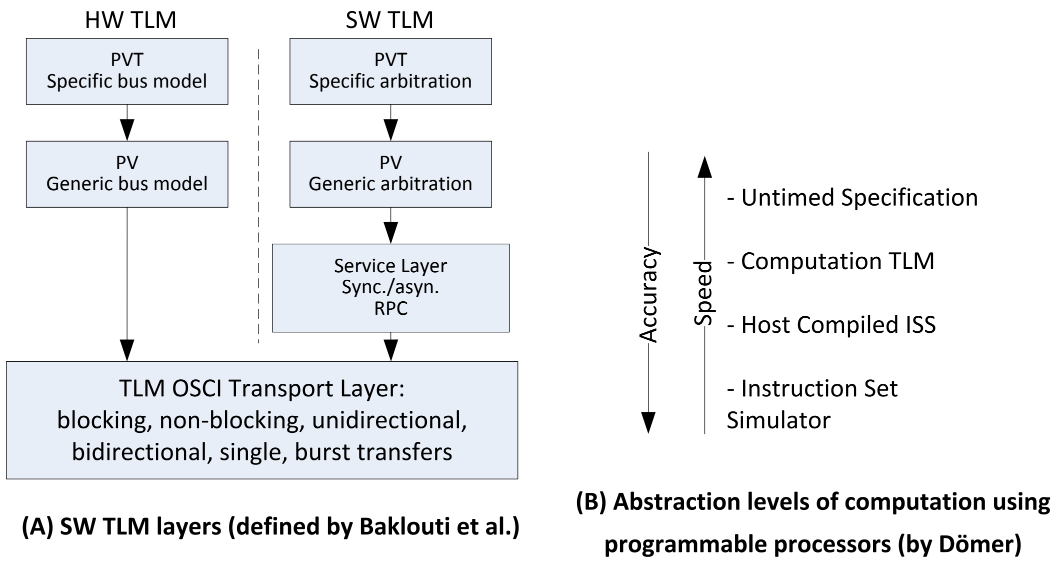

HW TLM
SW TLM
PVT Specific bus model
PVT Specific arbitration
PV Generic bus model
PV Generic arbitration
Service Layer Sync./asyn. RPC
TLM OSCI Transport Layer: blocking, non-blocking, unidirectional, bidirectional, single, burst transfers
(A) SW TLM layers (defined by Baklouti et al.)
Accuracy
Speed
- Untimed Specification
- Computation TLM
- Host Compiled ISS
- Instruction Set Simulator
(B) Abstraction levels of computation using programmable processors (by Dömer)

Figure 3-6. Related SW modelling abstraction level definitions (reprint [6] [9])

processors. However, this work does not specifically distinguish various TLM abstraction levels. In general, bearing the current status of TLM research in mind, most TLM abstraction level definitions have focused on modelling abstractions for communication and hardware design, and may not be appropriate for software modelling.

According to the basic assumption of OS-based task modelling and simulation in this thesis, it is not recommended to use TLM communication techniques in software modelling, since they are not common methods in conventional real-time software development. This idea is contrary to [6] that uses OSCI TLM communication services for joint HW and SW communication exploration.

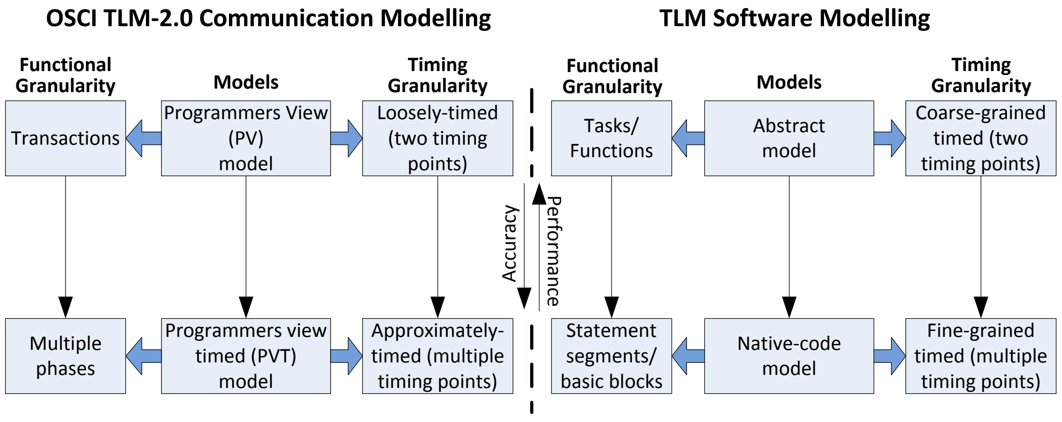
Note that it is not nontrivial to utilise existing TLM concepts directly. Here we need to define appropriate behavioural software abstraction levels/models and introduce their relationships with existing TLM modelling communication concepts.

3.2.2.1 Comparison with the OSCI TLM-2.0 Standard

Regarding the TLM communication modelling abstraction level definition, the latest OSCI TLM-2.0 modelling standard is selected as the reference [88]. It defines two coding styles for bus-based communication modelling, i.e., the LT style for PV models and the AT style for PVT models. Regarding the software modelling part, in Section 2.3, system-level software (RTOS) behavioural modelling and research is classified into two general categories depending on their timing accuracy: coarse-grained timed abstract models and fine-grained timed native-code models.

This section compares characteristics of the mixed timing software models and the OSCI communication modelling standard (see Figure 3-7):

- Both modelling approaches decompose a model's functionality into several basic entities, i.e., tasks (or finer-grained functions) for software modelling in our approach, and transactions with corresponding transport functions for TLM communication modelling. If there is a further necessity for more accurate modelling, then a basic entity can be divided into some finer-grained entities, i.e., multiple functions inside a task or multiple basic blocks inside a function, as well as corresponding multiple phases that task place during a transaction's transmission life.
- We define two comparable timing abstraction levels for models. The coarse-grained timed level and the fine-grained timed level for software modelling are comparable to the LT coding style and the AT coding style for TLM communication. We propose that the coarse-grained timed level uses two time points to represent the execution cost of a task or a function, i.e., the beginning and the end of execution. The LT coding style also defines two time points for each transaction to denote calling to and returning from the



OSCI TLM-2.0 Communication Modelling
TLM Software Modelling
Functional Granularity
Models
Timing Granularity
Transactions
Programmers View (PV) model
Loosely-timed (two timing points)
Multiple phases
Programmers view timed (PVT) model
Approximately-timed (multiple timing points)
Accuracy
Performance
Tasks/ Functions
Abstract model
Coarse-grained timed (two timing points)
Statement segments/ basic blocks
Native-code model
Fine-grained timed (multiple timing points)

Figure 3-7. OSCI TLM-2.0 models and proposed TLM software models

transmission respectively. Accordingly, the concept of the fine-grained timed level is also parallel to the OSCI AT communication coding style. This is because they both use multiple timing points inside a basic functional unit, namely, multiple annotations and timing synchronisation points.

- Besides, both the untimed timed level and the cycle-accurate timed level are not recommended in either our software modelling or the OSCI TLM standard. This is because modelling real-time software and contemporary bus communication systems apparently need a timing concept.

Based on the above comparison, our software modelling proposal has some similarity to the OSCI TLM-2.0 communication modelling standard, that is, in terms of modelling concepts about timing granularity and functional granularity. Since they are both implemented in the SystemC simulation environment, they also include similar changing trends in terms of modelling accuracy and simulation performance. This means that models at a corresponding level are "harmonious" to each other without resulting in undesired extreme behaviour in the context of TLM co-simulation. We will explain software model definitions in detail in Section 3.2.3.

In addition, each hardware computation model (e.g., a hardware peripheral device) needs to be annotated with delays to accompany with software timing models. Each TLM inter-module communication action is also to be assigned with corresponding communication delays. However, these two parts are not the focus of this thesis.

3.2.3 Defining Software Models

Let us consider two possible situations in an embedded software development flow:

1) At an early design phase, the application software, RTOS, and hardware devices may have different levels of development progress. This means that the components of a system may have incomplete source code. The precision of corresponding timing information also varies. Therefore, in many cases, it is difficult to build models at the same abstraction level.

2) Different system design teams may focus on modelling different system aspects according to their respective design circumstances. For example, modelling computation and modelling communication are two distinct working directions in the context of embedded systems modelling and simulation. As well, RTOS designers and application software programmers also pay different attention to SW modelling. It is not only infeasible but also costly to build all sub-models with the same timing accuracy level.

Therefore, in order to increase flexibility of software validation, a mixed timing approach is an efficient and practical solution. At some certain early and middle design stages, with the advance of the development and change of validating intention, software designers can build and simulate behavioural software models at various functional and timing levels in a unified SystemC framework.

There are two difficult issues in system-level software modelling and simulation: *timing accuracy* and *simulation performance*. It is well known that the granularity of annotation is a dominant factor of timing accuracy, in terms of mostly determining whether or not the execution cost of a code segment is "accurately" reflected in the model. For example, given a code segment including dynamic data-dependent loops, a single coarse-grained time annotation for the whole code segment is very likely to be less accurate than several fine-grained time annotations for each loop. On the other hand, simulation performance is also a major issue concerning simulation users in the early design phases. Simulation models need to process many annotation statements intervening between functional codes, which necessarily result in simulation overheads. Moreover, a delay annotation statement is always implemented as a *wait-for-delay* statement or associated by a *wait-for-event* statement in order to progress the simulated target clock. Such statements result in context switches between the SystemC simulation kernel and software model processes. Consequently, fine-grained time annotations may lead to more simulation overheads as a side-effect. The mixed timing approach proposes using different annotation granularities in software models, and thus enables model programmers to switch timing accuracy for simulation performance in simulations.

There are already some typical annotation granularities mentioned in existing annotation-based software simulation research, e.g., the assembly instruction level, the source line level, the basic block level, the function level, and the task level [121]. This thesis uses some of them in research and presents guidelines for using some appropriate timing annotation granularities in the two types of software behavioural models, i.e., abstract software models and native-code software models. Currently, time annotations are manually inserted into software models and automatic annotation is beyond the focus of this thesis. Research examples in this area can be found in [136] [137].

3.2.3.1 Abstract Software Models

The underlying assumption of the abstract software model is that it is usually applied at the early design phases for fast real-time software prototyping simulation and validation. At the time, the target hardware platform is undetermined and most software code has not yet been implemented. Consequently, abstract software models do not contain much implementation code or only contain limited functional specification code. Corresponding timing information of running code on a target platform cannot be obtained with high precision for these kinds of models. Rather, timing estimates and execution budgets are used for timing annotations. This kind of modelling and simulation is similar to the reservation-based timing analysis approach in real-time system timing analysis research, which advocates using timing predictions to incrementally validate timing properties of a system from its early development stages [138].

Software applications are organised as a set of SLDL process based independent tasks with coarse-grained temporal properties, e.g., period, deadline, offset, and execution times. Periodic execution should be explicitly supported by a generic RTOS model that supplies basic periodic execution services, meaning that the RTOS can schedule periodic tasks according to explicitly-defined period properties. Timing overheads of RTOS functions can be considered as rough estimates and annotated in models.

An abstract software task model contains one conceptual functional unit (i.e., a task) or several subunits (i.e., several functions). Correspondingly, task-level

```
#001 void task1(){
#002   while(1){
#003   //No code or
#004   functional_code;
#005   DELAY(fixed_value);
#006   //or
#007   DELAY(random_value);
#008   wait-for-event;
#009   }
#010 }
```

(A) Pseudo code of task-level time annotation

```
#001 void func1()
#002 {
#003   ...
#004   ...
#005   DELAY($t_1$);
#006   wait-for-event;
#007 }
```

(B) Pseudo code of function-level time annotation

Table 3-1. Abstract software models and coarse-grained time annotations

(Table 3-1 (A)) and function-level (Table 3-1 (B)) time annotation levels are proposed for abstract software models. Each annotation statement corresponds to an execution unit, i.e., a task or a function. The delay time information can either be given as a fixed value representing the WCET at the model building stage, or be randomised between a lower bound (i.e., the BCET) and an upper bound (i.e., the WCET) for each job of a task in simulation time.

An annotation value is inserted by the `DELAY()` function (e.g., line 5 in Table 3-1 (A)), which passes the delay value to the Live CPU Model and triggers it for an interruptible time advance. A *wait-for-event* statement is inserted after a delay statement (e.g., line 8 in Table 3-1 (A)), in order to yield control of the SystemC simulation kernel and let the task wait for resuming after the delay. It defines a time advance point (also referred to as a timing synchronisation point). From the multitasking OS point of view, calling the *wait-for-event* statement and returning from it mark the beginning and the end of "execution duration" of a software model along the target simulation timeline. From the perspective of SystemC simulation, a piece of "execution duration" is in fact a piece of "waiting duration" of a SystemC process.

As shown in Figure 3-8, because an abstract software model is assumed to be independent and does not access shared variables, it execution duration can be freely interrupted by higher-priority IRQs, i.e., any asynchronous interrupt events can stop its time advance step. Although a delay value is only annotated once, it can be divided into many slices due to ISRs. This models a correct timing order of execution.

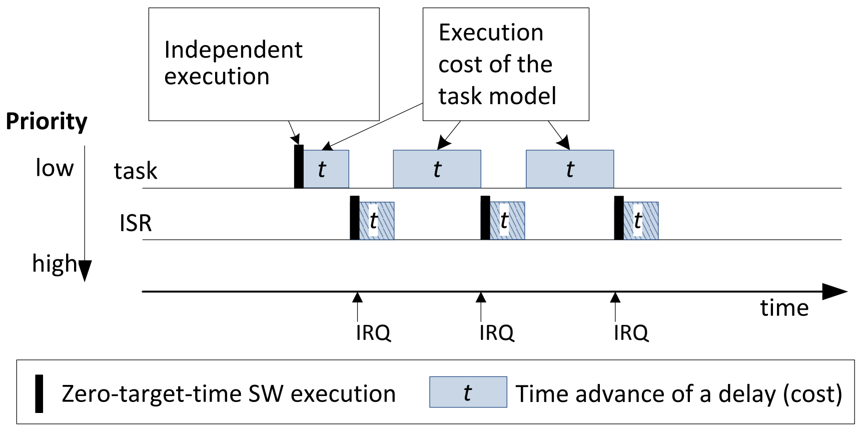

Independent execution
Execution cost of the task model
Priority
low
high
task
ISR
t
time
IRQ
IRQ
IRQ
Zero-target-time SW execution
Time advance of a delay (cost)

Figure 3-8. Execution trace of an abstract task software model

The details of the *wait-for-event* method, the interruptible time advance method, and the `DELAY()` function will be introduced in Sections 3.2.7, 3.3.4 and 4.5.8.1.

3.2.3.2 Native-Code Software Models

When a large quantity of software application code has been developed and a RTOS has been either supplied as an off-the-shelf product or developed in-house, native-code software models can be built. The available software code is wrapped in some software task models that are also implemented as SLDL processes. These task models can be further divided into statement segments or atomic basic blocks whose performance is measurable or estimable with relatively high accuracy. These native-code application software tasks can utilise the APIs of a RTOS model, which may model specific services of a real RTOS and is annotated with corresponding timing delay information.

Timing accuracy becomes a major concern in native-code software simulation. The desired target timing behaviour cannot be directly represented in native-code software execution. Hence, software execution costs (time delays) on the target platform need to be either analysed by a static analysis method or dynamically evaluated in a measurement-based method, and then be manually or automatically annotated to corresponding code statements in task models. Fine-grained *statement segment* level annotations and *basic block* level annotations are advocated to be applied in this type of software models.

```
void func1(){
  if(condition)
  {                              A compound
    ...                          statement
  }
  DELAY(t1);
  wait-for-event;

  int temp;
  temp = 100;                    Several statements
  temp++;
  DELAY(t2);
  wait-for-event;
}
```

(A) Pseudo code of statement segment level time annotation

```
void func1()
{
  DELAY(t1);  <-- annotation before code
  wait-for-event;
  int temp = 0;                  Basic block 1
  if(condition)
  {
      temp++;                    Basic block 2
      DELAY(t2);  <--
      wait-for-event;            annotation
  }                              after code
}
```

(B) Pseudo code of basic block level time annotation

Table 3-2. Native-code software models and fine-grained time annotations

In the example code sown in Table 3-2 (A), a statement segment is either a compound statement or several sequential statements. A compound statement is defined as a sequence of source statements enclosed by a pair of curly braces [139]. In modelling, several sequential assignment or number operation statements are also treated as a statement segment for convenience of annotation. However, a statement segment should not include access to an OS service, which should be treated as another segment.

A basic block is a sequence of code that has only one entry point and only one exit point [140]. In Table 3-2 (B) the annotation statement of a basic block may have two possible places, i.e., before the basic block or after the basic block. In modelling, where to place the annotation statement depends on how to "glue" the time annotation near its code block, in order to make native-code execution synchronise with corresponding target-time advance steps as much as possible.

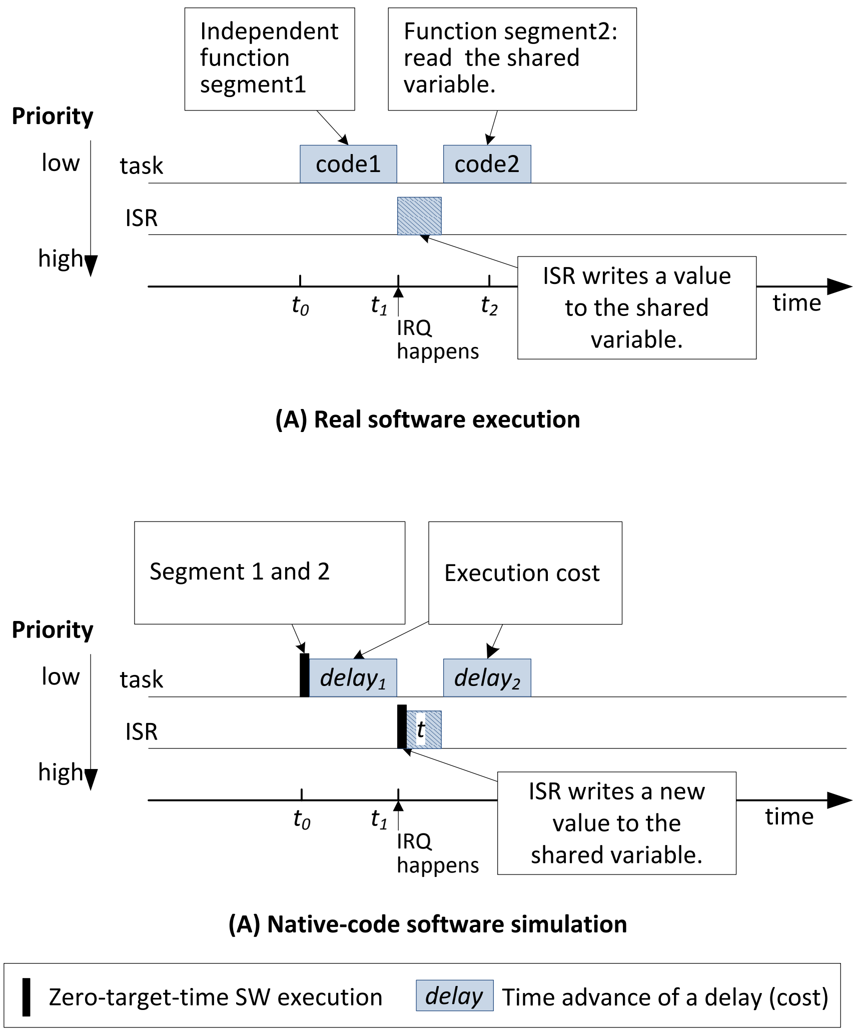
Multiple `DELAY()` functions and *wait-for-event* time advance points are inserted in native-code software models. Their respective behaviour is the same as the before-mentioned abstract software models.

In native-code models, software code segments may access global shared variables that may be affected by external interrupts. If a code segment and its annotation are defined improperly, a wrong simulation trace and a result may be generated. As shown in Figure 3-9 (A), in real software execution, a task independently executes code segment 1 from time t_0. At time t_1, an IRQ happens and pre-empts the task. An ISR writes a value to a global variable. Afterwards, the task resumes and its code segment 2 reads the global variable to obtain an updated value.

Figure 3-9 (B) shows a possible corresponding simulation trace, in which the task code segment (with its corresponding annotation) includes both code segment 1 and 2. This means that the task not only executes some independent functions but also reads the global variable at t_0, and its total delay begins accordingly. The IRQ still happens at t_1, then pre-empts the task, and writes the global variable. Although the time advance of the task can be interruptible and maintained correctly in terms of the simulation time order, the functional simulation result is possibly wrong because the software task gets an outdated value of the global variable.

The solutions to this problem are straightforward:

1) In software models, global variables should be protected by mutual exclusions in order to avoid race conditions. This is effectively a common con-



Independent function segment1
Function segment2: read the shared variable.
Priority
low
high
task
ISR
code1
code2
t_0
t_1
t_2
IRQ happens
ISR writes a value to the shared variable.
time
(A) Real software execution
Segment 1 and 2
Execution cost
Priority
low
high
task
ISR
$delay_1$
$delay_2$
t
t_0
t_1
IRQ happens
ISR writes a new value to the shared variable.
time
(A) Native-code software simulation
Zero-target-time SW execution
delay
Time advance of a delay (cost)

Figure 3-9. Unmatched real execution and simulation traces

vention in software programming.

2) In terms of native-code simulation, a code segment should not include both independent functions and an access to a global variable. In another words, an access to a global variable should be placed in a separate segment that is as short as possible. Based on the first solution, this requirement is not difficult to implement in modelling, because a global variable segment is always marked by calling to OS mutually exclusive services.

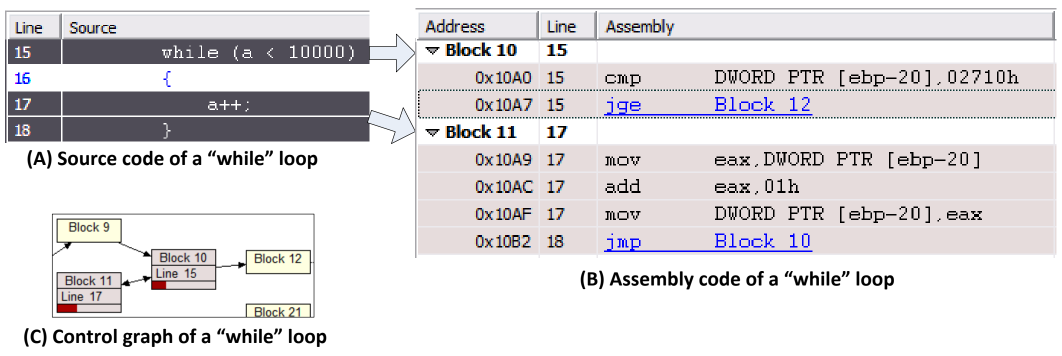
3.2.4 Techniques for Improving Simulation Performance

Fine-grained time annotations can improve timing accuracy in case there are data-dependent conditional or looping statements in code, but too many intrusive annotations not only require more modelling work but also decrease simulation speed. Similarly, defining many time advance points (so-called timing synchronisation points) can make the simulated clock be progressed smoothly. However, it also decreases simulation performance. Consequently, two techniques regarding timing annotations and time advance points are utilised in order to improve simulation performance.

3.2.4.1 Reducing the Number of Time Annotations

This first technique is to reduce the number of annotation statements by merging several sequential time annotations into one longer annotation.

Given a simple "while" loop program in Figure 3-10 (A) as an example, the Intel VTune Performance Tuning Utility [141] is used to carry out basic block


```
Line  Source
15        while (a < 10000)
16        {
17            a++;
18        }
```
(A) Source code of a "while" loop

```
Address    Line  Assembly
▽ Block 10  15
  0x10A0   15    cmp     DWORD PTR [ebp-20],02710h
  0x10A7   15    jge     Block 12
▽ Block 11  17
  0x10A9   17    mov     eax,DWORD PTR [ebp-20]
  0x10AC   17    add     eax,01h
  0x10AF   17    mov     DWORD PTR [ebp-20],eax
  0x10B2   18    jmp     Block 10
```
(B) Assembly code of a "while" loop

(C) Control graph of a "while" loop

Figure 3-10. A "while" loop example

```
#001 while (a < 10000)
#002 {
#003   DELAY(tbb10);
#004   wait-for-event;
#005   a++;
#006   DELAY(tbb11);
#007   wait-for-event;
#008 }
#009 DELAY(tbb10);
#010 wait-for-event;
```

(A) Precise basic block level time annotations

```
#001 while (a < 10000)
#002 {
#003   a++;
#004   DELAY(tbb10+tbb11);
#005   wait-for-event;
#006 }
#007 DELAY(tbb10);
#008 wait-for-event;
#009
#010
```

(B) Merging time annotation statements

Table 3-3. Reducing number of time annotations

analysis for application software. This tool can organise assembly code in basic blocks (see Figure 3-10 (B)) and generate a control flow graph (see Figure 3-10 (C)). Referring to the figure, there are two basic blocks in the program, i.e., the "Block 10" of the "while" statement and the "Block 11" of the looping body.

If this program is annotated with basic block level timing delays, then three annotation statements are needed, as shown in Table 3-3(A). Because the two basic blocks "Block 10" and "Block 11" (line 1 and line 5 of Table 3-3 (A)) execute sequentially at most times except for jumping out of the while loop, their time annotations t_{bb10} and t_{bb11} can be merged into one annotation as showed on line 4 of Table 3-3 (B).

This technique advances the annotation level from the basic block level to the statement segment level, which is a mixed timing annotation technique and widely used in our research. Normally, merging multiple annotation statements should sacrifice timing accuracy of annotations as little as possible. For instance, the `DE-LAY(`t_{bb10}`)` statement (line 9 of Table 3-3 (A)) corresponds to the "compare and jump out" execution of the while statement and is improper to be combined into the annotation statement inside the loop body. Otherwise, target time advance steps cannot match the native-code execution flow. However, if model builders intentionally make tradeoffs between accuracy and performance, it is also acceptable that some tiny one-shot annotations can be omitted.

3.2.4.2 Reducing the Number of Time Advance Points

The second technique to increase the simulation speed is to reduce the number of *wait-for-event* statements in models, i.e., reducing the number of time advance points. The basic idea is inspired by the “lazy synchronisation” method introduced by Hartmann et al. [127], in which this method is used in proprietary abstract software modelling. Here, we refine it for native-code software simulation models.

As introduced before, a time advance point refers to a timing synchronisation point where a software model process yields control to the SLDL simulation kernel in order to let it advance the simulated clock.

In discussions and figures in Section 3.2.3, the annotation statement `DELAY()` and the *wait-for-event* method are used together. A `DELAY()` function finishes two jobs, i.e., injecting an annotation value into the Live CPU Model and invoking it to advance the timing delay value at once. In fact, in the proposed mixed timing approach, a delay annotation function does not need to implement the two jobs conjunctively. And, a *wait-for-event* method does not necessarily follow each time annotation statement either.

As shown on line 5 and line 9 in Table 3-4, the lightweight `DELAY_WR()` function only processes an annotation value in terms of storing and accumulating it in a variable (see Virtual Registers in Section 3.3.2) in the Live CPU Model, but it does not invoke the Live CPU Model to progress the simulated clock immediately. It is especially appropriate for use in data-dependent loops in order to reduce time advance overheads.

The dual-function `DELAY()` and the *wait-for-event* statements are also important at specific points in model code (e.g., lines 12 and 13 in Table 3-4). Some rules are defined to indicate where time advance points are essential. In modelling, these situations include:

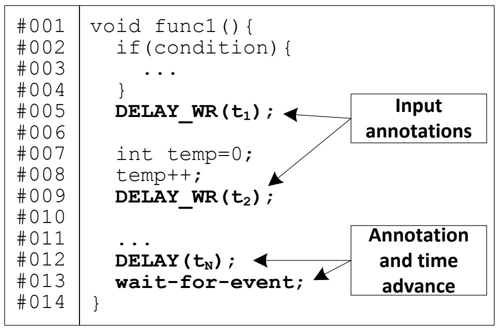

```
#001  void func1(){
#002    if(condition){
#003      ...
#004    }
#005    DELAY_WR(t1);
#006
#007    int temp=0;
#008    temp++;
#009    DELAY_WR(t2);
#010
#011    ...
#012    DELAY(tN);
#013    wait-for-event;
#014  }
```

Table 3-4. Reducing number of time advance points

1) In application tasks, time advance points are necessary before calling and returning from RTOS system functions. These points define the boundary of a task and a RTOS function, and allow switches to be made between them.
2) If the current running application task will terminate execution, then a time advance point is necessary. This point defines the boundary between different tasks.
3) In any critical sections (no matter in tasks or in RTOS functions) where interrupts are disabled, time advance points are necessary in order to progress the target clock.

This technique essentially separates annotation points from time advance points. This is a native capability of the mixed timing approach because of the underlying annotation-independent time advance method. The reduced running chances of the Live CPU Model and fewer context switches of the SystemC kernel can speed up simulation speed. At the same time, fine-grained timing annotations can still be used in order to accurately reflect the timing cost of software models' execution traces.

3.2.5 Application Software Performance Estimation

Previously, it has been noted that behavioural software modelling and simulation need timing information of software execution on the target platform. Software instrumentation and performance estimation are pre-requisites of all back annotation based behavioural simulation. This is a quite broad and non-trivial research domain, which is far beyond the focus of this thesis. Example research in this domain can be found in [84] and [142]. In Sections 3.2.5 and 3.2.6, some related performance estimation methods are introduced in brief rather than presenting in-depth research. The final modelling builders and simulation users can determine and apply appropriate time estimation methods in practice.

3.2.5.1 Static Timing Analysis Method

A typical static analysis method is the WCET analysis[2] [143]. It aims to compute an upper bound for the execution time of a piece of program by analysing the code but without actually running it. A WCET analysis includes three steps:

- The *program flow analysis* extracts possible executing sequences of a program at the basic block level. This study should try to cover all possible paths in order to generate a safe coverage.
- The *low-level analysis* calculates execution time of each basic block on a given target hardware architecture. The complexity of this study is to consider various performance-enhancing features of modern processors, such as caches, pipelines, etc.
- The *calculation step* combines paths information and low-level execution times in order to derive a WCET.

WCET results might be used as source of time annotations in our mixed timing software modelling.

For abstract software models, the assumption is that much software code has not been available; hence, specific WCET analysis cannot be implemented. For native-code models, model programmers can use conventional WCET analysis to obtain software timing information. In our consideration, now that the source code is available for simulation, we prefer to annotate statements at a fine granularity, which means that the basic-block WCET information are more useful than function-level or task-level WCET results that may be over-pessimistic. Colin et al. specifically take WCET analysis on the RTEMS RTOS with the intention to study the predictability of RTOS timing behavioural [144]. This research reveals the possibility of obtaining timing information of RTOS services by the static analysis approach.

3.2.5.2 Statistical Methods

We can use time estimates of tasks and functions to build simulation models in order to capture initial approximate timing behavioural of a system. These time

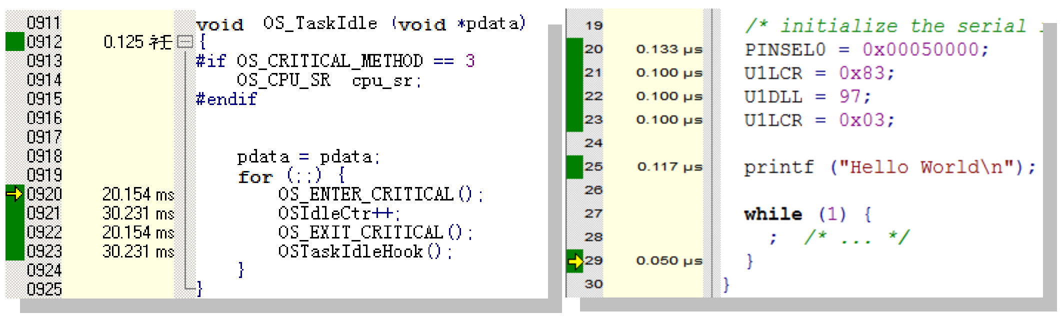
[2] The BCET analysis is a related problem to find the lower execution bound of a program.

estimates can be generated either from functional specifications or a random function. Regarding the latter technique, for simple cases that do not have a strict requirement on the approximation of generated numbers, the `rand()` pseudo-random function in the C Standard General Utilities Library (with the head file `stdlib.h`) is used. If there are some definitions on the probability densities of periods and computation times of tasks, the well-acknowledged UUNIFAST algorithm can be used to generate task sets with uniform distribution in a given space [145].

3.2.5.3 Dynamic Simulation-Based Method

In simulation-based software performance estimation methods, software source code is compiled for a given processor architecture, and is then executed on the actual target CPU or on an accurate model of the target CPU, e.g., an instruction-set simulator. Accurate performance information can be profiled after executing real software. In this thesis, the ISS-based profiling technique is used to acquire accurate timing information of both application software and the selected RTOS.

For ARM-based embedded systems, the KEIL µVision ARM development kit [146] is recommended to use, which provides various cycle-accurate instruction-set models of ARM processor and complete execution profiling functions. As shown in Figure 3-11, the µVision execution profiler can display and record execution times and calling times of each function or statement through ISS execution.



```
            void  OS_TaskIdle (void *pdata)
0.125 ҿ    {
            #if OS_CRITICAL_METHOD == 3
                OS_CPU_SR  cpu_sr;
            #endif


                pdata = pdata;
                for (;;) {
20.154 ms           OS_ENTER_CRITICAL();
30.231 ms           OSIdleCtr++;
20.154 ms           OS_EXIT_CRITICAL();
30.231 ms           OSTaskIdleHook();
                }
            }
```

Function-level profiling information

```
              /* initialize the serial 
0.133 µs      PINSEL0 = 0x00050000;
0.100 µs      U1LCR = 0x83;
0.100 µs      U1DLL = 97;
0.100 µs      U1LCR = 0x03;

0.117 µs      printf ("Hello World\n");

              while (1) {
                ;  /* ... */
0.050 µs      }
            }
```

Statement-level profiling information

Figure 3-11. µVision software profiler

3.2.6 RTOS Performance Estimation

3.2.6.1 The Scaling Parameter Method

For early and abstract modelling research in which both RTOS and the target platform are not fixed, simulation users may be interested in the relative magnitude of RTOS timing cost and compare simulation results of several different design alternatives. It is not necessary to assign precise timing estimates for every RTOS activity. RTOS system services can be annotated by a scaling parameter method in [2]. This relates execution cost for each RTOS action to a scaling parameter (*S*), which reflects relative timing magnitudes of different RTOS services depending on their typical computational complexities. Table 3-5 shows execution times of some typical RTOS services in terms of the scaling parameter S. Note that in an individual modelling case, the programmer can correct the scaling factor of a specific RTOS function depending on available timing information.

Action	***S***	**Action**	***S***	**Action**	***S***
Context switch	2	Task suspend	1	Semaphore/mutex create/delete	6
Task initiate	12	Task resume	1	Message queue create/delete	10
Task create and run	28	Semaphore/mutex post	1	Message queue available	2
Task delete	10	Semaphore/mutex wait	1	Message queue not available	1

Table 3-5. Basic RTOS actions and their relative execution times [2].

3.2.6.2 The Benchmark Method

If the software model programmer intends to model a well-documented commercial RTOS case, then RTOS benchmark results from production vendors can be used as the timing annotation source for the RTOS simulation model, which is similar to the approach in [87]. A benchmark document supplies timing costs of various RTOS services, for example: kernel entry, context entry, message passing, synchronization, timers, signals, task management, and message queues.

RTX-RTOS on LPC2138 ARM7 CPU @ 60MHz (Unit : µs)
code executed from internal flash with Memory Accelerator Module

Action	Time	Action	Time
Initialize system	34.9	Task switch	7.1 – 10.5
Create defined task, no task switch	14.3	Send semaphore (no task switch)	2.7
Create defined task, switch task	16.7	Send message (no task switch)	5.3
Delete task	9.6	Interrupt response for IRQ ISR	0.8

Table 3-6. RTX RTOS timing specification [1]

For instance, the QNX Neutrino RTOS [147] is provided with average kernel benchmark results based on different hardware platforms such as Intel Pentium4 processors, XScale processors, and TI OMAP processors. And, referring to Table 3-6, the RTX RTOS is also provided timing specifications on a specific ARM platform [1]. If benchmark documents are not available for some specific platforms and RTOS versions, development kits or benchmark suites are sometimes supplied by their vendors, in order to let users measure timing costs by themselves.

3.2.6.3 The ISS-based Measurement Method

The ISS-based simulation method is utilised to measure RTOS timing overheads. Table 3-7 shows some timing information of the µC/OS-II RTOS measured on the 48 MHz ARM KEIL ISS simulator. It is worth noting that, although an ISS simulator can produce fine-granularity timing information of real RTOS source code, only the function-level timing cost of each RTOS service is concerned. This is because this thesis proposes to build a generic RTOS model that can provide comparable functionality to a real RTOS. The implementation code of the RTOS model may not have one-to-one correspondence to actual RTOS source code. It is

µc/OS-II RTOS on Keil LPC2378 ARM7 ISS @ 48MHz (Unit : ns)
code executed from internal flash

Action	Time	Action	Time
Enter the main RTOS function	1366310	Task switch	2660
RTOS initialisation	51750	Initialise a semaphore	3170
RTOS starts multi-tasking	2770	Wait a semaphore	3930
Create a task	22500	Received a message	3160

Table 3-7. µC/OS-II RTOS timing specifications

not feasible to annotate the RTOS model at the basic block level or at the statement level.

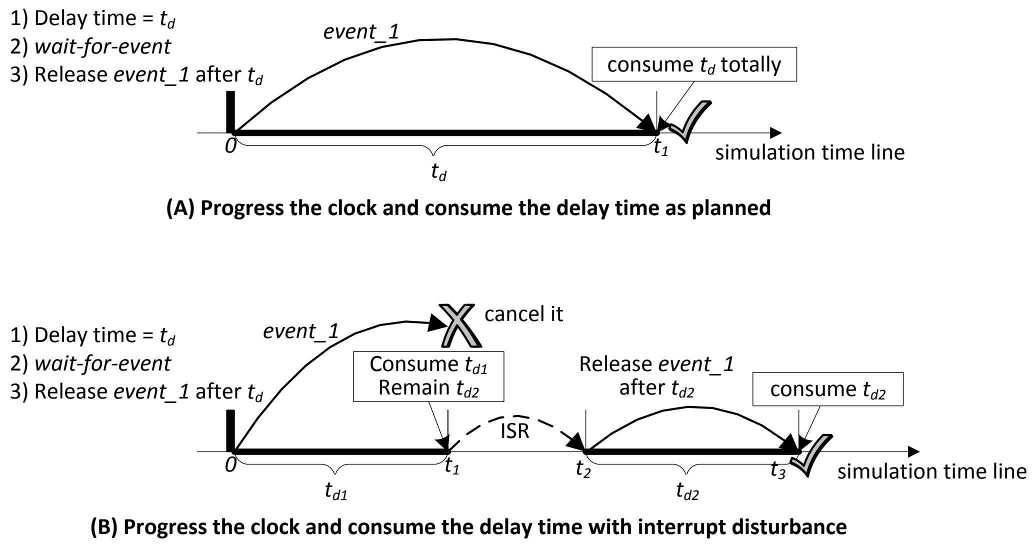
3.2.7 Timing Issues in Software Simulation

3.2.7.1 The Variable-Step Time Advance Method

In the mixed timing approach, at simulation runtime, a software model firstly executes its functional code in zero-time and then passes its corresponding delay information to the Live CPU Model. Afterwards, the Live CPU Model advances the simulated clock in order to mimic the software execution timing cost on the target platform. The specific progress step of the clock not only depends on inputted delay information, but is also affected by whether an interrupt event happens during this delay duration, which may disturb delays of low-priority tasks. It is named the "variable-step" time advance method, since the actual length of a time delay step at simulation runtime is variable rather than being restricted by the time annotation defined at the modelling stage. Figure 3-12 shows this time advance idea in two simulation cases. Note that no matter which simulation case, when an event is planned to be released (an arc in the figure) at a future time point, it is actually unknown when this event will be finally released because of possible interrupts and pre-emptions.

In Figure 3-12 (A), since there is no interrupt interference, an event is thus successfully released according to the input delay information t_d in order to resume the waiting software task. The simulation clock is also progressed with a step of t_d.

However, in Figure 3-12 (B), an external interrupt is raised at the time point t_1 that is earlier than $(0 + t_d)$. Consequently, the planned event is cancelled and the initial ending time point $(0 + t_d)$ is no longer validated for time advance of the waiting software task. The software task is pre-empted and its remaining delay value is calculated as t_{d2}. After a time interval, i.e., following the execution of the ISR in this case, the pre-empted task resumes and the rest of its delay time is advanced again until completion at t_3. This example shows the "variable-step" characteristic of the time advance method.



1) Delay time = t_d
2) wait-for-event
3) Release event_1 after t_d
event_1
consume t_d totally
0
t_1
t_d
simulation time line
(A) Progress the clock and consume the delay time as planned
1) Delay time = t_d
2) wait-for-event
3) Release event_1 after t_d
event_1
cancel it
Consume t_{d1} Remain t_{d2}
Release event_1 after t_{d2}
consume t_{d2}
ISR
0
t_1
t_2
t_3
t_{d1}
t_{d2}
simulation time line
(B) Progress the clock and consume the delay time with interrupt disturbance

Figure 3-12. The variable-step time advance method

3.2.7.2 The Fixed-Step Time Advance Method

Schirner et al. propose that it is unnecessary to mimic intermediate states in simulation, and it is only essential to generate correct results at state-changing boundaries [123]. High performance is thus the primary goal of simulation. Indeed, with the consideration for simulation performance and efficiency, all abstract and behavioural simulation bears this underlying assumption to hide intermediate simulation runtime details and only maintain similarity between the simulation trace and the real execution trace to a certain extent. The variable-step time advance method also generally accords to this point of view. It consumes software execution delays in coarse-grained steps, and aims to minimise the number of "steps" for a better simulation speed. From the perspective of maintaining simulation correctness at specified event-changing points, e.g., interrupt points or task switching points, this time advance method is satisfactory.

From the perspective of debugging real-time embedded software execution traces and observing status of system-wide variables, simulation users may not be satisfied by observing limited information only at event-changing points. Thus, the *fixed-step* time advance method is proposed as a complementary time advance method. Referring to Figure 3-13, this advances the simulated target clock over more steps, according to pre-defined periods. In the fixed-step mode, the Live

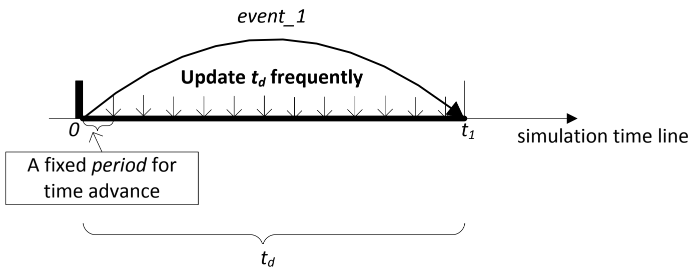


Figure 3-13. The fixed-step time advance method

CPU Simulation Engine can run periodically to update run-time changing variables, such as value of timers, software delay slices, execution budgets, etc. The increasing number of time advance steps may also increase simulation times. Hence, the Live CPU Simulation Engine can blend variable-step and fixed-step time advance methods in simulation if simulation users want to trade off simulation performance with intermediate observability.

3.2.7.3 Timing Accuracy of Simulation

In the mixed timing simulation approach, the theoretical timing accuracy of software simulation can be evaluated through three aspects, i.e., the timing of the simulated target clock, the timing of software delay advance, and the timing of software/hardware interactions:

1) Resolutions (the minimum interval of time) of progressing the simulation clock, which are dependent on timing resolutions of two basic actions:
    a. The resolution of advancing software delay duration:
        i. **General requirement**: This resolution refers to the minimum step to progress the target simulation clock. It should be as fine-grained as possible in order to be able to represent tiny delays accurately.
        ii. **Features of the proposed approach**: Since models are simulated in the SystemC environment, they are restricted by the SystemC simulation kernel's timing resolution - the default value is 1 *picosecond*. It is enough to represent software execution costs accurately. In fact, for high-level behavioural software simulation, the common timing resolution is at the microsecond (μs) level or the millisecond (ms) level in practice.

b. The resolution of stopping software delay duration:

   i. **General requirement**: It refers to the latency to stop the current target simulation clock advance step, in the case that an interrupt happens. It should be as small as possible, i.e., zero-time in theory, in order to mimic the real situation.

   ii. **Features of the proposed approach**: Because the proposed interruptible time advance method relies on the Live CPU Simulation Engine, when an interrupt happens, the simulated clock is progressed to this time point. At the same time, the consumed part of a software delay is immediately calculated and the remaining delay part is saved. Consequently, this means that the resolution of stopping software delay duration is zero-time, i.e., without incorrect latency.

2) Maintaining execution delay information of software models:

   a. **General requirement**: Every software model has some delay information representing its running cost on the target architecture. These delays must be accurately consumed in terms of the quantity and order.

   b. **Features of the proposed approach**: According to the time advance methods introduced earlier, a software model's timing delay information is securely kept on a per-task basis and correctly consumed in its time advance in simulation. In case of a pre-emption, the delay information of a task is updated, and its remaining part is able to resume in future time advance.

3) Timing accuracy of handling interrupts:

   a. **General requirement**: This is mainly revealed by the interrupt latency, which is the time from the raising of an external interrupt signal till the beginning of a software interrupt handler. The simulated interrupt latency should be similar to the real situation in terms of predictability and functionality.

   b. **Features of the approach**: The interrupt handling approach is based on a combination of the timely hardware interrupt catching model and the zero-latency software delay stopping method. The Live CPU Model can sense external interrupt requests when it consumes software delays at the same time. Since both hardware models and software models execute in the discrete-event SystemC simulation framework with a unified global clock, there is no additional HW/SW synchronisation latency that may appear in asynchronous co-simulation. Hardware-initiated interrupt handling can begin immediately and can be propagated to a software handler without delay. The theoretical minimum *interrupt latency* is zero-time in simulation, and the worst-case *interrupt latency* is bounded by the longest interrupt disabled time which is fully configured by model builders. This timing behaviour is the same as a real-time system that runs on a real CPU.

3.3 The Live CPU Model

3.3.1 The HW Part of the SW Processing Element Model

To undertake accurate system-level embedded software modelling and simulation, it is necessary to consider and model the underlying hardware architecture at an appropriate abstract level. Because many RTOS services, e.g., context switch, interrupt service, and clock service, are hardware-dependent, it could be difficult to model HW/SW interactions accurately without support from a hardware model on which software models are assumed to run. Moreover, one-sided software modelling is against the system-level HW/SW co-design principle for embedded systems. The existence of hardware models makes the simulation more likely to resemble a full embedded system. Many studies have suggested using transaction level models for high-level system modelling and simulation. In Section 2.1.3.3, the concept of the software processing element model has been introduced, which consists of two research aspects of this thesis, i.e., software modelling and hardware modelling. As shown in Figure 3-14, this software PE model can be seen as a mixture of two parts: behavioural software simulation (from the software modelling aspect) and the hardware abstraction model (from the abstract hardware modelling aspect).

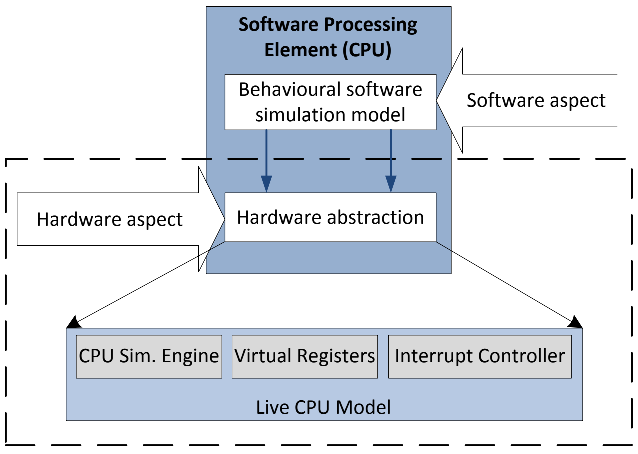


Figure 3-14. Hardware part of the software PE model

In a real embedded system, software runs on top of a CPU subsystem. In our *software processing element* modelling approach, the CPU subsystem is abstracted and encapsulated into the hardware abstraction model, namely the Live CPU Model. It provides abstract yet essential hardware controlling functionality and architecture (e.g., interrupt controller, real-time clock, and virtual registers) for modelling upper-level software systems. More importantly, it supports interruptible and pre-emptive SystemC-based behavioural software simulation by the Live CPU Simulation Engine. It plays a live role in managing software time advance in order to mimic the timing behaviour of executing software on a target platform, just like a real CPU executing software instructions.

Because of the high abstraction level and the underlying native simulation concept, our mixed timing software simulation does not need a low-level instruction-set architecture processor model with complete internal components, such as logic units, control units, memory subsystems, general-purpose registers and special-purpose registers. The Live CPU Model is composed of three essential components for software simulation:

1) The Virtual Registers are used for storing delay information and setting flag bits (in Section 3.3.2). They are internal model constructs in the proposed simulation approach.
2) The Interrupt Controller Model monitors interrupt-request lines and activates software handlers (in Section 3.3.3).

3) The Live CPU Simulation Engine takes charge of advancing software simulation time (in Section 3.3.4).

Based on these components, this abstract Live CPU Model is actively involved in high-level software simulation. In the following, they will be introduced in detail.

3.3.2 The Virtual Registers Model

In a typical real-world processor system, computer programs are stored in a three-level memory hierarchy, e.g., main memory, cache, and hardware register. The CPU directly accesses these memory components to load and store instructions and data. Memory protection, cache management, coherency and consistency are important research issues in this area. However, for concerned SLDL-based behavioural software simulation, this thesis does not model this memory subsystem, because it is not necessary to model the instruction-execution mechanism of the target processor. Instead, software natively executes on the host platform, which maintains its own memory system as a black-box for our simulation.

However, in order to support hardware-dependent software simulation, a Virtual Registers model is built inside the Live CPU Model. These Virtual Registers do not correspond to registers of a real CPU, but rather hide inside the abstract Live CPU Model and take effect in a black-box way. Model builders can tailor this virtual register set in our software simulation context. Referring to Table 3-8, Virtual Registers are divided into two categories depending on their use:

- Some Virtual Registers are related to software time advance. The prime concern of these virtual registers is to assist the Live CPU Simulation Engine to progress software simulation time. Six virtual registers store 64-bit software timing information such as delay value, deadline, start time stamp, etc. The CPU_REG[0] "Delay Register" and the CPU_REG[4] "Start-time Stamp Register" are two particularly important registers for software time advance and will be frequently referred to in description of the Live CPU Simulation Engine later. When a software task context switch is invoked, current contents of these registers are saved in the pre-empted task's TCB,

Virtual Registers			
For SW simulation time advance		**For system status and flags setting**	
Register Name	**Descriptions**	**Register Name**	**Descriptions**
CPU_REG[0]	**Delay Register**: delay value of current code block	CPSR	Current Program Status Register
CPU_REG[1]	Total delay of current task job	SPSR	Saved Program Status Register
CPU_REG[2]	Absolute deadline of current task job	ICRR	Interrupt Controller Raw Status
CPU_REG[3]	Consumed delay time	ICSR	Interrupt Controller Status Register
CPU_REG[4]	**Start-time Stamp:** start time of current delay	ICMR	Interrupt Controller Mask Register
CPU_REG[5]	Task suspension time	… …	… …

Table 3-8. Virtual Registers

and the newly dispatched task's timing information in its TCB is loaded into these registers.

- As illustrated in the right part of Table 3-8, some 8-bit Virtual Registers hold system runtime status and help the Interrupt Controller Model to handle interrupts. For example, the Current Program Status Register (CPSR) is mainly used to distinguish the execution mode of the Live CPU Model, i.e., the *normal* software simulation mode or the *interrupt request* mode. The Interrupt Controller Raw Status (ICRS), the Interrupt Controller Status Register (ICSR), and the Interrupt Controller Mask Register (ICMR) contain original interrupt request information, interrupt service information, and interrupt masking configuration, respectively.

3.3.3 The Interrupt Controller Model

It is acknowledged that the *interrupt latency*, *interrupt response time*, and *interrupt recovery time* are some concerned timing properties of a real-time embedded system. The Interrupt Controller Model provides a hardware-level foundation to model a usual HW/SW cooperative interrupt handling mechanism, which usually has three bottom-up layers: the HW interrupt controller, the RTOS interrupt handler, and application ISRs. As illustrated in Figure 3-15, the main function of the Interrupt Controller Model is encapsulated in the `cpu_ic()` `SC_METHOD` process. It monitors a set of `sc_ports`, which are further connected to various interrupt sources (e.g., peripheral devices) by IRQ lines.

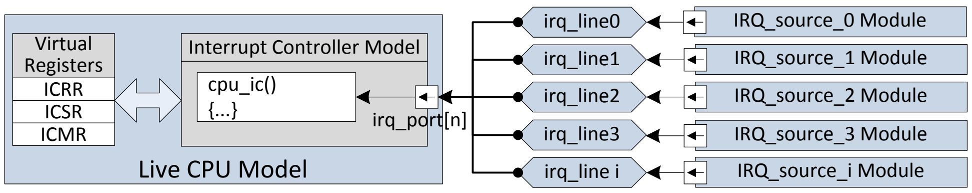


Figure 3-15. Interrupt Controller Model

In order to deal with multiple simultaneous interrupts from various devices and bound the *interrupt latency*, the Interrupt Controller Model can prioritise, mask or disable interrupt sources by setting corresponding register bits in ICRR, ICSR and ICMR. When a hardware device raises an IRQ by asserting a signal through its interrupt request line, the Interrupt Controller Model can catch the signal immediately and call a software interrupt handler, which could be either a RTOS kernel interrupt handler function or a vectored ISR depending on a specific interrupt handling scheme. This software handler will subsequently invoke the Live CPU Simulation Engine to stop the current delay process. Depending on specific implementation, a software handler can be pre-emptible or non-pre-emptible.

3.3.4 The Live CPU Simulation Engine

In the mixed-timing software modelling and simulation approach, SystemC-based software models are compiled for the host platform and then executed on it. It is necessary to model the target simulated clock in order to mimic the timing behaviour of real-time software in the target environment. As introduced before, current SLDL-based real-time software simulation approaches have some deficiencies on interrupt and pre-emption modelling. The Live CPU Simulation Engine relaxes the existing problems by controlling time advance for software models, and cooperates with the Interrupt Controller Model to handle external hardware interrupts in a timely manner. Excluding possible interrupt-disabled situations, e.g., executing a critical section, the Live CPU Simulation Engine can interrupt current software execution (stopping its delay period in practice) as soon as an interrupt event is caught by the Interrupt Controller, just like software execution on a real CPU.

```
#001 SC_METHOD(cpu_sim_engine);
#002 dont_initialize();
#003 sensitive << evt_rtos_start_call_cpu_sim_engine
#004           << evt_apps_call_cpu_sim_engine
#005           << evt_rtos_service_call_cpu_sim_engine
#006           << evt_tick_isr_2_cpu
#007           << evt_interrupt_handler_enter_2_cpu
#008           << evt_cpu_advance_total
#009 #ifdef _CPU_DYNAMIC_FIXED
#010           << m_cpu_clk.posedge_event()
#011 #endif
```

Table 3-9. Sensitivity list of the Live CPU Simulation Engine

The basic modelling idea of the Live CPU Simulation Engine is to use the SLDL *wait-for-event* mechanism instead of the uninterruptible *wait-for-delay* mechanism. The Live CPU Simulation Engine is implemented as a `SC_METHOD` process. It coordinates its execution and controls time advance of various software tasks by corresponding *events* (i.e., objects of the SystemC `sc_event` class). Table 3-9 shows the static sensitivity list of the Live CPU Simulation Engine. The events on lines 3-7 are externally called by software models to trigger execution of the Live CPU Simulation Engine, the event on line 8 is internally used by the Live CPU Simulation Engine to trigger itself for time advance, and lines 9-11 configure the running mode of the Live CPU Simulation Engine if it needs to run periodically, i.e., the fixed-step time advance method.

Referring to Figure 3-16 (A), most real CPUs execute software cycle-by-cycle

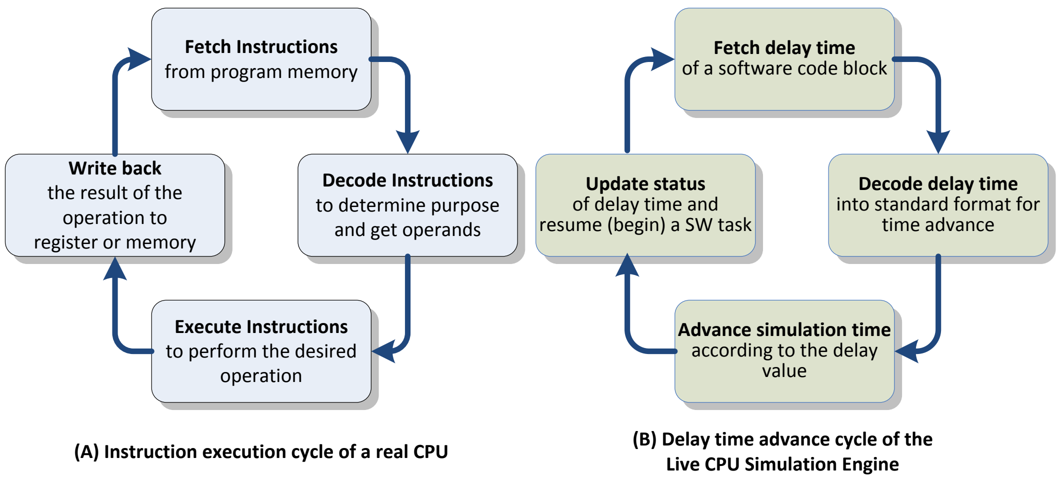

Fetch Instructions from program memory
Decode Instructions to determine purpose and get operands
Execute Instructions to perform the desired operation
Write back the result of the operation to register or memory
(A) Instruction execution cycle of a real CPU
Fetch delay time of a software code block
Decode delay time into standard format for time advance
Advance simulation time according to the delay value
Update status of delay time and resume (begin) a SW task
(B) Delay time advance cycle of the Live CPU Simulation Engine

Figure 3-16. Real CPU execution and Live CPU simulation

according to an execution mechanism that includes four fundamental stages: fetch instructions, decode instructions, execute instructions, and store (write back) results. Inspired by this classical mechanism, the Live CPU Simulation Engine instead executes software models' *delay times* over four comparable conceptual stages: fetch delay time, decode delay time, advance simulation (delay) time, and update status (see Figure 3-16 (B)).

3.3.4.1 Software Prerequisites of the Live CPU Simulation Engine

Before describing the Live CPU simulation cycle, it is necessary to indicate some assumptions and pre-requisite background knowledge of the Live CPU based software simulation approach:

1) Application software has been organised into tasks. Each task is wrapped in a SystemC `SC_THREAD` process and has a TCB storing some individual information. Each task is registered to an exclusive event, whose notification can make the task resume from a *wait-for-event* statement.
2) If there are multiple concurrent tasks in the system, basic OS software functions are needed. They include: OS scheduling functions to select a new task to run and mark it with the RUNNING state; OS interrupt handling functions to select an appropriate ISR for a relevant IRQ; and OS context switch functions to save and load task's context information between its TCB and Virtual Registers. The "context" mainly refers to timing information of a task such as, for example, the execution cost, the used execution time, the deadline and the start time.
3) The Live CPU Simulation Engine is only responsible for maintaining delay value stored in Virtual Registers and advancing the simulated target clock for the RUNNING task. It is independent from any above software OS functions. This reflects the SW/HW orthogonal and modular modelling principle.

3.3.4.2 Operation of the Live CPU Simulation Engine

Referring to Figure 3-17, the Live CPU based software time advance process can be described over five steps along the target simulation timeline. There are two possible software time advance cases, i.e., without interrupt interference (see

Figure 3-17 (A)), or with interrupt interference (see Figure 3-17 (B)). In following descriptions, Steps (A), (B), (C), and (D) of the two cases are the same, their difference residing in Step (E).

1) Step (A): Preliminary to advancing software simulation time by the Live CPU Simulation Engine, a software task is firstly loaded into the Live CPU

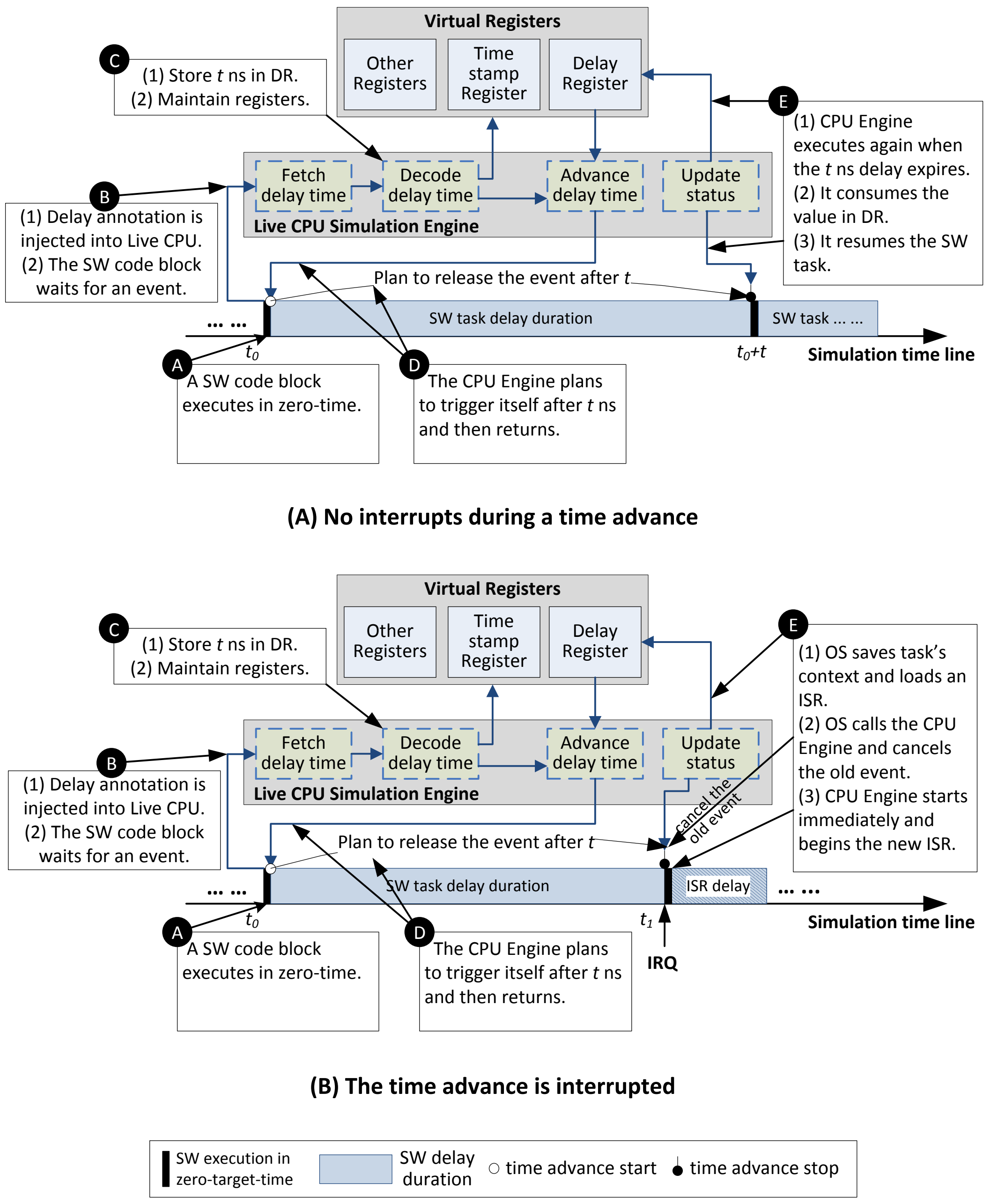

Virtual Registers
Other Registers
Time stamp Register
Delay Register
C (1) Store *t* ns in DR. (2) Maintain registers.
E (1) CPU Engine executes again when the *t* ns delay expires. (2) It consumes the value in DR. (3) It resumes the SW task.
B (1) Delay annotation is injected into Live CPU. (2) The SW code block waits for an event.
Fetch delay time
Decode delay time
Advance delay time
Update status
Live CPU Simulation Engine
Plan to release the event after *t*
SW task delay duration
SW task ... ...
... ...
t_0
t_0+t
Simulation time line
A A SW code block executes in zero-time.
D The CPU Engine plans to trigger itself after *t* ns and then returns.
(A) No interrupts during a time advance
E (1) OS saves task's context and loads an ISR. (2) OS calls the CPU Engine and cancels the old event. (3) CPU Engine starts immediately and begins the new ISR.
cancel the old event
ISR delay
t_1
IRQ
(B) The time advance is interrupted
SW execution in zero-target-time
SW delay duration
○ time advance start
● time advance stop

Figure 3-17. Operations of the Live CPU Simulation Engine

Model by an OS context switch operation. Then a software code block, which could either be a whole task, a function, a statement segment, or a basic block, executes in zero-target-time at time t_0.

2) Step (B): After the software code block finishes execution, an explicit time advance point can be reached. Here, there is a delay annotation function and a SystemC `wait(event)` statement, just as what is introduced in Section 3.2.3.
   a. The delay annotation function generates a delay value which may have different timing units (e.g., second, millisecond, microsecond, etc.) and meanings (e.g., task level delay or basic block level delay) for modelling convenience. The value is written into a temporary variable in the Live CPU Model, i.e., **delay information is fetched**, and the Live CPU Simulation Engine is triggered to be ready-to-run.
   b. The software code block then keeps waiting for its exclusive SystemC `sc_event` object that will be released by the Live CPU Simulation Engine at a future time point. This `sc_event` object represents the "address of code block to run" in our simulation. Its importance is similar to the *program counter* in a real CPU.
   c. From the perspective of the internal SystemC scheduler, the SystemC process, which the software code unit belongs to, yields control to the SystemC simulation kernel and the Live CPU Simulation Engine process will be selected to run in next. However, from the perspective of OS scheduling, this software task is still at the RUNNING state.
   d. Note that, when using the simple single-purpose annotation function DELAY_WR() in Section 3.2.4.2, only the delay value is stored for prospective time advance, but the Live CPU Simulation Engine is not triggered and there is no `wait(event)` statement. Hence, the software model will continue executing until a time advance point is reached.

3) Step (C): Because inputted delay information may have specific formats, it is necessary to transform them into standard-form data for use with time advance. The Live CPU Simulation Engine then **decodes delay informa-**

tion into a double float number with the nanosecond timing scale. The decoded result "*t ns*" is stored in the *Delay Register* (DR) that belongs to the virtual register set of the Live CPU Model. At the same time, the current time stamp t_0, which can be obtained by the SystemC function `sc_time_stamp()`, is also recorded in another virtual register.

4) Step (D): Subsequently, the Live CPU Simulation Engine starts the "**simulation (delay) time advance**" step at t_0. This stage consists of two operations: the Live CPU Simulation Engine plans to wake up itself at a future time point and then returns. The CPU Engine's sleeping duration represents execution cost of a software model. Depending on the execution mode of the Live CPU Simulation Engine, there are three possible cases:
   a. If the Live CPU Simulation Engine works in a pure variable-step time advance mode, it plans to progress the delay time *t* in the DR in a single step. It sets the internal event to trigger itself at the coming time point t_0+*t*. Then it returns control back to the simulation kernel in order to advance the simulation time by the duration of *t*.
   b. If the Live CPU Simulation Engine is set with a fixed-step time advance mode, it runs periodically in order to decrement and update the delay value in DR until the delay value is totally exhausted, whilst, the simulation clock is progressed period-by-period.
   c. If the Live CPU Simulation Engine is configured with both the variable-step and the fixed-step modes, it not only plans to wake up at the final time point, but also periodically decrements the delay value.
5) Step (E): In this stage, the Live CPU Simulation Engine **updates the simulation status** by maintaining delay time and resuming or beginning a software task. There are two possible situations depending on whether an interrupt happens:
   a. Assuming a simple case where there is no interruption or pre-emption during the *t* time duration as illustrated in Figure 3-17 (A), thus the Live CPU Simulation Engine wakes up at time t_0+*t*. It consumes the value in DR and then issues the event related to the current RUNNING

task so as to make it continue executing. Upon that, the above execution cycle is repeated.

b. A main target of the mixed timing approach is to solve the non-interruptible problem of SystemC software simulation. It is important to consider the interference from an unexpected interrupt event during ongoing software delay duration. As shown in Figure 3-17 (B), before the time advance duration t expires, an IRQ happens at t_1 that is earlier than the time point t_0+t projected in Step (D). Given that the interrupt handling mechanism of the system is not intentionally disabled, the Interrupt Controller Model thus catches the IRQ immediately and then invokes the software OS interrupt handling function to serve this IRQ, i.e., the current RUNNING task will be pre-empted by a higher-priority ISR. The OS interrupt handling function saves the remaining portion of the delay time slice and other timing information in Virtual Registers to the pre-empted task's TCB for future use. The remaining portion of the delay time is calculated as: $t_{remain} = t-(t_1-t_0)$, where t is the initial value of the DR and t_1 is the current time stamp. The OS interrupt handling function then dispatches (i.e., loads its context to Virtual Registers) an appropriate ISR as the next-to-run software task and calls the Live CPU Simulation Engine by notifying an event to replace the previously-planned wake-up event. The Live CPU Simulation Engine faces fresh values in the Virtual Registers and sends an event to allow the ISR to run immediately. Consequently, the software ISR executes its functional code and repeats the above time advance process. In this way, both software time advance and hardware interrupt handling are simulated accurately.

3.4 Evaluation Metrics

Recalling the three requirements on SLDL-based software behavioural modelling and simulation mentioned at the beginning of this chapter, the flexible modelling aspect is mainly addressed in Section 3.2.3 by supporting different software functional and timing models. The simulation performance and simulation accuracy aspects are addressed in this section in order to evaluate experiments in Section 3.5.

3.4.1 Simulation Performance Metric

In this section, the metric of simulation performance is defined as how much simulation time (i.e., host time) is used to execute a specific simulation in the host computer. A specific simulation refers to executing a software test program, which is modelled in the mixed timing approach and simulated by the Live CPU Model for a set of repeated iterations. As the referenced cycle-accurate simulator, the KEIL ARM ISS [146] executes the same test program for a same number of loops. Simulation speeds of the mixed timing simulation approach and the ISS approach are compared in order to calculate a simulation speedup, which is:

$$speedup = \frac{ISS\ simulation\ time}{Mixed\ timing\ simulation\ time}$$

Note that although the ISS simulator is also a software-based simulation approach, it executes cross-complied software binaries for a target hardware platform. In the context of high-level software simulation, functional and timing behaviours of an ISS are commonly deemed the same as real software execution on a corresponding processor.

3.4.2 Simulation Accuracy Metrics

Simulation accuracy metrics of the mixed timing approach relate to two aspects, i.e., functional accuracy and timing accuracy. In Section 3.3.4, some simple OS functions are introduced as the basis for mixed timing software simulation. However, the focus of this section is not to present a detailed OS simulation model with complete multi-tasking and concurrent execution services. Rather, this section concentrates on relationships between software models and their timing characteristics, i.e., time annotation and advance. Hence, a test program does not utilise many OS functions but needs to include data-dependent loops that require dense time annotations.

3.4.2.1 Functional Accuracy

Functional accuracy refers that, in terms of a given test program, whether behavioural simulation models can represent similar functions and generate correct results compared to real software execution. Based on the definition in Section 3.2.3.1, abstract software models do not sufficiently reflect this property if they do not aim to include enough functional code. Regarding native-code software simulation models, this property can be evaluated by compared its simulation results to those of an ISS simulation.

However, evaluating functional accuracy is not an emphasis in this chapter, because it is not difficult to guarantee that a single task model can execute correct modelling functions. Especially, a native-code task model may have the same code as a real task. Functional accuracy of concurrent multi-tasking software models will be addressed in Chapter 4, when a complete RTOS model is introduced.

3.4.2.2 Timing Accuracy

By simulating a software model in the proposed mixed timing approach, it is known how much simulated time (i.e., the target time in SystemC) is used to execute a set of repeated iterations of a given test program, which is referred to as t_{mixed}. It can also find the simulated time of the same test program and iterations in an ISS simulator, which is referred to as t_{ISS}.

Timing accuracy can be reflected by comparing t_{mixed} and t_{ISS}. If they are close, then the timing accuracy of the mixed timing approach is deemed good enough. A timing accuracy loss is computed as:

$$\frac{|t_{mixed} - t_{ISS}|}{t_{ISS}} \times 100\%$$

Inaccuracy of timing is contributed by three parts, i.e., software performance estimation, delay annotation, and time advance.

The first part is not within research focus of this thesis, so ISS-based measurement method is used (See Section 3.2.5.3). It can provide highly accurate software performance information.

The second part is addressed in definitions of software models in Section 3.2.3. It should be noticed that inaccurate annotations may be intentional choices of simulation users for the sake of fast simulation performance and ease of modelling.

The third part is a notable advantage of the mixed timing approach in terms of supporting interruptible software time advance by the Live CPU Simulation Engine. However, in this chapter, without involving many task switches and RTOS services in simulation, this aspect cannot be evaluated thoroughly.

Still, referring to Section 3.2.7.3, there are three basic features related simulation timing accuracy can be evaluated:

1) The resolution of stopping a software time advance step
2) Timing accuracy of handling interrupts
3) Maintaining execution delay information of software models

The first point can be evaluated by measuring how fast a time advance step can be stopped in the proposed simulation approach. The second point can be simplified as the interrupt latency at the moment. In fact, it refers to the same feature as the first point. The third point can be evaluated by observing whether a task's time advance can be resumed properly after it is pre-empted.

3.5 Experimental Results

All simulation tests in this section are performed with SystemC v2.2 on three x86 PCs (frequencies ranging from 1.86GHz to 2.2GHz) running Windows OSs. Tests of a single topic are always carried on the same PC in order to be comparable. Host simulation times are measured by Windows Win32 function `QueryPerformanceCounter()`, which can retrieve the value of the high-resolution hardware performance counter and provide microsecond level host execution time [148]. Target simulated times are obtained by using SystemC function `sc_time_stamp()`.

3.5.1 Performance Evaluation

3.5.1.1 Simulation Performance of Different Timing Models

In Section 3.2.3, the abstract software model and native-code software models are introduced. Because they have distinct functional and time annotation characteristics, their simulation performance necessarily differs. Furthermore, in Section 3.2.4, two techniques are introduced to improve simulation performance by adjusting time annotation and advance statements in code. This section presents some tests to evaluate simulation performance of these different models and modelling techniques. In order to concentrate on the above-mentioned aspects and eliminate the possibility that software functional complexity may dominate simulation performance, the test program includes a single task implementing a *selection sort* algorithm. This algorithm involves typical data-dependent *if* conditional operations and *for* loop operations, which require fine-grained time annotations if the timing accuracy is a concern. Although RTOS services are not called by the task, limited RTOS services (without delay annotations) are still executed in order to initialise the software simulation system.

As shown in Table 3-10, the same program is simulated in six cases:

- Two abstract software models: The first abstract software model does not implement the actual function of the sort algorithm, whilst the second abstract model does. They are both annotated one time annotation statement and one time advance point at the task level.
- Three native-code models: They all implement the sort function and have four fine-grained segment level annotation statements, which are approximately timing accurate regarding data-dependent loops.
  - The native-code 1 and 2 are both implemented by the proposed mixed timing method and the interruptible Live CPU based time advance method. Their difference is: two time advance points are defined in native-code model 1, which utilises the reduced time advanced technique in Section 3.2.4.2; whereas, four time advance points are defined in native-code model 2 and inside data-dependent loops.

	Proposed abstract model 1	**Proposed abstract model 2**	**Proposed native-code model 1**	**Interuptbible native-code model 2**	**Uninterruptible native-code model 3**	**ISS**
Functions	Without functions	With functions	With functions	With functions	With functions	Final code
Time annotation granularity	Coarse-grained function-level	Coarse-grained function-level	Fine-grained segment-level	Fine-grained segment-level	Fine-grained segment-level	Cycle-accurate ARM7TDMI-S LPC2124 @60MHz
Number of time anno. statements	1	1	4	4	4	
Time advance granularity	Coarse-grained function-level	Coarse-grained function-level	Reduced advance	Fine-grained segment-level	Fine-grained segment-level	
Number of time adva. statements	1	1	2	4	4	

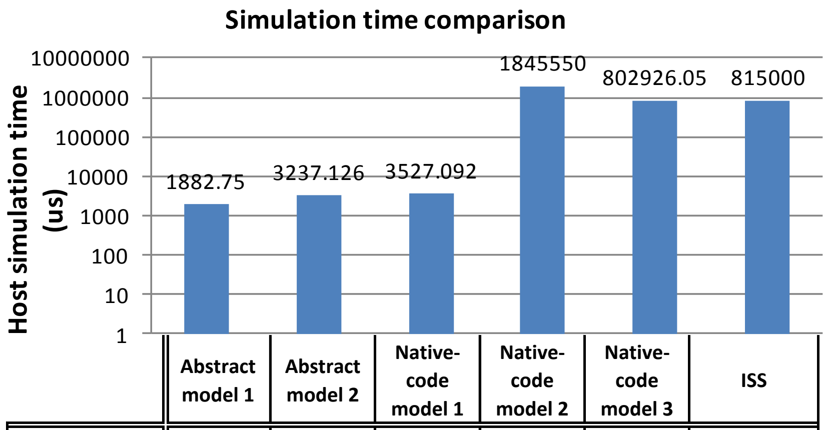
Table 3-10. Descriptions of experimental cases

- ▪ The native-code model 3 utilises the uninterruptible *wait-for-delay* time advance method. It is a conventional annotation-dependent software simulation model.

- The test program is also run on the KEIL ARM ISS without cache and OS support and its execution time costs are used to annotate above behavioural models.

Simulation results are shown in Figure 3-18. Some phenomena and conclusions can be inferred:

- The abstract model 1 is faster (over 400x speedup compared to ISS) than other models because that it does not model functionality and has the fewest execution counts of annotation and time advance in simulation. The abstract model 2 is slower than abstract model 1 due to its functional complexity. They both can be used for abstract software modelling in this thesis.
- The proposed native-code model 1 has fast simulation speed, i.e., over 200x speedup compared to ISS. It is functional accurate, i.e., with the native-code function. Its timing accuracy is also promising because of sufficient execution counts of annotation statements in simulation (see the quantification re-



	Abstract model 1	Abstract model 2	Native-code model 1	Native-code model 2	Native-code model 3	ISS
Host simulation time (us)	1882.75	3237.126	3527.092	1845550	802926.1	815000
Total execution counts of annotation	1	1	125749	125749	125749	
Total time advance steps	1	1	2	125749	125749	

Figure 3-18. Simulation time results

sult in Section 3.5.2.1). It is recommended to be used in native-code software modelling in this thesis.

- The interruptible native-code model 2 also has similar functional and timing accuracy behaviours compared to the native-code model 1. However, its slowest simulation speed is not satisfactory. Certainly, it may represent some special software simulation situations, where many time advance points are necessary (see Section 3.2.4.2). If these “uncommon” situations indeed happen frequently, the simulation speed of the proposed mixed timing simulation approach will necessarily decrease.
- The uninterruptible native-code 3 is weak in terms of its uninterruptible time advance method and slow simulation speed, i.e., using over 200x simulation time more than the proposed native-code model 1.

In addition, Figure 3-18 shows some statistics on execution counts of time annotation statements and time advance steps in simulation. Regarding the proposed mixed timing simulation approach in this thesis, two characteristics can be inferred from the perspective of this experiment and give guidance to some extent:

1) More annotation statements do not contribute too much simulation time. Comparing the native-code model 1 and the abstract model 2, 125749 times more annotation statements result in less than 10% simulation overheads.
2) Time advance steps (i.e., execution of the Live CPU Model) affect simulation performance greatly. Comparing the native-code model 2 and the native-code model 1, 62875 times more time advance steps incur 500 times more simulation time.

3.5.1.2 Simulation Performance of Varying Time Advance Methods

In the previous section, simulation performance was evaluated by varying timing modelling related aspects. This section inspects simulation performance of models by changing time advance method of the Live CPU Simulation Engine.

In Section 3.2.7, the variable-step and fixed-step time advance methods are introduced as execution mechanisms of the Live CPU Simulation Engine. By setting the two time advance methods for the Live CPU Simulation Engine, trade-offs can be made on simulation speed, observability, and time advance accuracy.

The software test program consists of eight abstract tasks (i.e., four equal-priority periodic tasks and four higher-priority ISR tasks) with randomly-generated task-level delays. A very simple OS model provides pre-emptive multi-tasking services. The OS scheduler implements fixed-priority and round-robin scheduling algorithms and is triggered by a combined time-driven and event-driven mechanism. Four interrupt sources are included in simulation and raised randomly in order to trigger ISRs. The test program runs for 1000 ms target time that allows a task to repeat at least 20 times.

The Live CPU Simulation Engine is configured in following models:

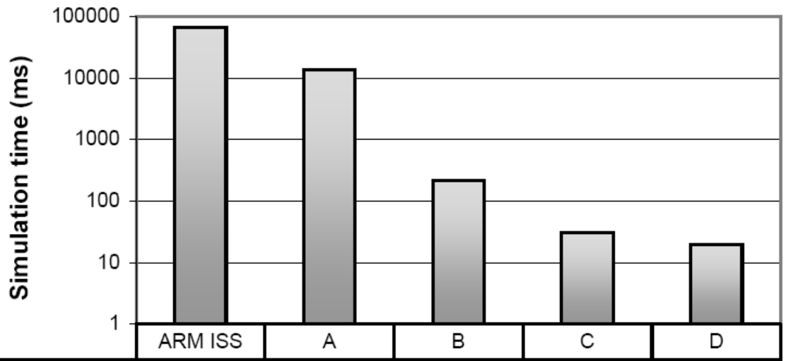
1) Model A: uses a fixed-step time advance method, which runs every 1 μs and advances the target clock by a step of 1 μs. It is similar to the fine-grained time period synchronization approach in Section 3.1.2. This achieves 1 μs time advance resolution.
2) Model B: uses a dual-grained fixed-step time advance method. It is similar to the multiple-grained time annotation method introduced in Section 3.1.3. When a software delay value is greater than 1 ms, the engine runs every

1ms to progress the target clock by a step of 1 ms. Once the delay value falls below 1 ms, then the engine runs every 1 μs to advance the target clock by a step of 1 μs. This achieves 1 μs time advance resolution.

3) Model C: uses a mixed fixed-step and variable-step time advance method. It progresses a delay slice in an interruptible variable-length step and also runs every 1 ms to advance the target clock by a step of 1 ms. The time advance resolution is only restricted by the timing resolution of SystemC simulation engine.
4) Model D: uses a variable-step time advance method. It progresses a delay slice in an interruptible variable-length step. The time advance resolution is only restricted by the timing resolution of SystemC simulation engine.

The same test program is run on KEIL ARM ISS for a same duration of 1000 ms. The target processor is a NXP LPC2378 running at 40MHZ. A μC/OS-II RTOS [149] is ported on this ISS to manage tasks.

Obtained simulation speed results are shown in Figure 3-19. Compared to ISS simulation, mixed timing models obtain drastic performance improvement in terms of the biggest speedup over 3000 times. Unsurprisingly, the variable-step approach is also faster than the fixed-step time advance approach. Model D



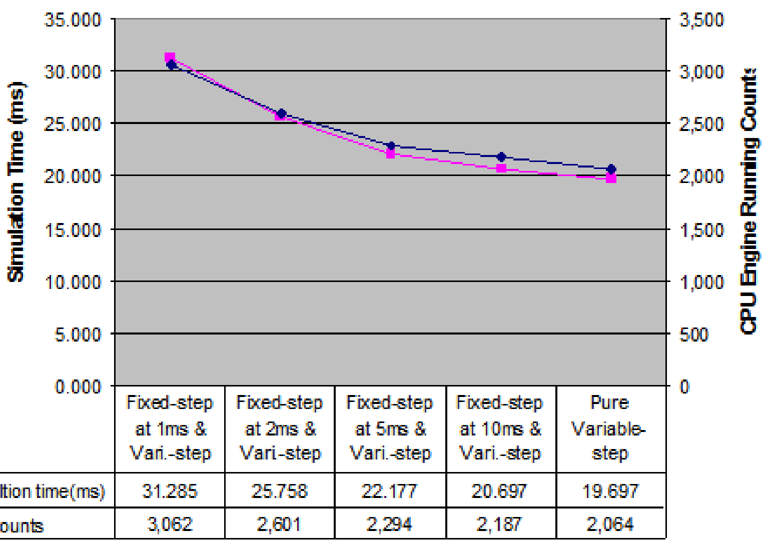
	ARM ISS	A	B	C	D
Simulation time (ms)	65892.5	13743.924	213.874	31.285	19.697
Simulation Engine Mode	Instruction Set Simulation	Fixed-step at *1 μs*	Fixed-step at *1 μs* and *1ms*	Fixed-step at *1ms* and variable-step	Variable-step
SW Time Advance Resolution	Cycle-accurate	1μs	1μs	Depending on SystemC resolution	Depending on SystemC resolution
Live CPU Engine Running Count	N/A	1,000,000	42,389	3,062	2,064

Figure 3-19. Simulation time comparison

achieves a considerable speedup (over 600 times) compared to model A. This is because the fixed-step approach progresses the target clock much more frequently than the variable-step approach, which is reflected by higher running counts of the Live CPU Simulation Engine.

The models B and C use combined time advance methods. From their simulation results, it can be inferred that finer periodic time advance steps result in more simulation overheads. In order to reveal relations between step lengths and simulation speeds of fixed-step time advance method, three additional tests are carried out with periodic steps of 2 ms, 5 ms and 10 ms, which mean the Live CPU Simulation Engine is activated to advance the target clock in every 2 ms, 5 ms, and 10 ms respectively.

Figure 3-20 shows that simulation times and Live CPU running counts steadily decrease whilst the fixed-step period is growing larger. This characteristic can be used to tune the Live CPU Simulation Engine and optimise the simulation performance and simulation observability in different situations. Besides, the periodic fixed-step time advance method can represent the behaviour of handling the periodic real-time clock interrupt of a RTOS, in which the Live CUP Simulation Engine is triggered periodically. According to simulation results, finer real-time clock interrupt periods incur extra but not excessive overheads, which can be used



	Fixed-step at 1ms & Vari.-step	Fixed-step at 2ms & Vari.-step	Fixed-step at 5ms & Vari.-step	Fixed-step at 10ms & Vari.-step	Pure Variable-step
Simualtion time(ms)	31.285	25.758	22.177	20.697	19.697
CPU counts	3,062	2,601	2,294	2,187	2,064

Figure 3-20. Comparison of varying fixed-step lengths

as a reference to determine the period of the clock interrupt in a RTOS model.

3.5.2 Accuracy Evaluation

3.5.2.1 Experimental Timing Accuracy

Experimental tests in Section 3.5.1.1 are also studied here. According to the analysis in Section 3.4.2.2, regarding a simple software model, its timing accuracy depends on its performance estimation and delay annotation granularity. Performance is measured in ISS and used for native-code software models. Timing delays are annotated at the segment level. Consequently, a good timing accuracy should be expected. As shown in Table 3-11, in terms of the same test program, native-code models consume very similar target time to the ISS simulator. This table also demonstrates that reducing time advance points does not affect timing accuracy of independent software models.

	Native-code model 1	Native-code model 2	ISS
Simulated times (μs)	6986.115	6986.115	6977.51
Accuracy loss	0.12%	0.12%	

Table 3-11. Timing accuracy of native-code models

3.5.2.2 Timing Accuracy of Basic Operations

Referring to the three basic features related simulation timing accuracy in Section 3.4.2.2, an interrupt experiment is executed in order to evaluate them in simulation.

This experiment includes five IRQs (IRQ1-5) and five associated ISRs (ISR1-5), which are assigned ascending priorities. Each IRQ randomly happens 500 times in 10 seconds simulated time. A normal task runs in the background and can be interrupted by any IRQs and pre-empted by their ISRs. The software system is configured so that interrupts are always enabled and the Live CPU Simulation Engine can stop current time advance as soon as a higher-priority interrupt happens. Therefore, at any simulation time point, interrupt latency of the highest-

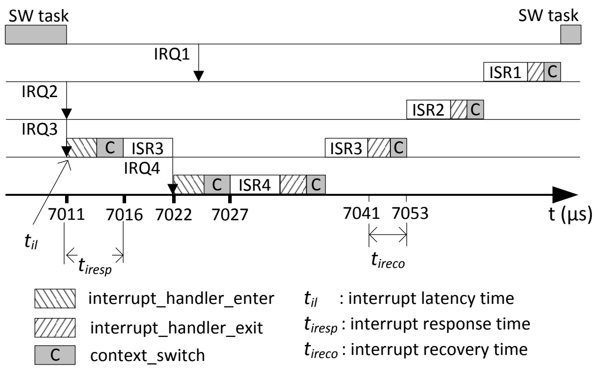


Figure 3-21. Interrupt handling experiment

priority IRQ should always be zero, and all other IRQs are only able to be post-poned by higher-priority ISRs.

Figure 3-21 shows a part of the timeline of this experiment, which is drawn according to the actual simulation log. It illustrates three concerned basic timing related features, i.e., immediate stop of time advance, resumable time advance, and zero-time interrupt latency. As well, it demonstrates some functions of the Interrupt Handler Model.

Referring to this simulation trace, at t=7011 μs, IRQ2 and IRQ3 happen simultaneously. Since the Live CPU model controls software time advance and monitors IRQ lines, the current software time advance step is stopped immediately and an IRQ is handled immediately. This interrupt latency is zero-time. Because the priority of IRQ3 is higher than IRQ2, the Interrupt Controller Model ignores IRQ2 and begins to service IRQ3. Afterwards, RTOS interrupt services and ISR3 execute sequentially. At t=7022 μs, a higher-priority IRQ4 happens and invokes nested interrupt service by pre-empting ISR3. Note that IRQ1 is raised during ISR4 execution; however, it is ignored by the Interrupt Controller Model due to its lower-priority priority. After the completion of ISR4, lower-priority ISRs are handled successively according to their priorities. Among them, ISR3 is released firstly to continue its remaining delay and finishes at t=7041 μs.

In order to quantify the interrupt latency in simulation, we measure interrupt latencies of these five IRQs in this experiment. The theoretical maximum interrupt latency of an IRQ can be computed as the sum of all higher-priority ISR time costs:

$$t_{max\ latency} = \sum\nolimits_{higher\_prio} t_{ISR}$$

Table 3-12 compares measured maximum interrupt latencies with calculated theoretical values. As expected, the highest-priority IRQ5 is always serviced without any delay. And other IRQs are also serviced with zero-time latency if there is no other higher-priority ISR in the system. In case that an IRQ is delayed by some other higher-priority ISRs, its maximum interrupt latency does not exceed the theoretical worst-case value either.

	Counts of zero-time interrupt latency	Counts of delayed Interrupt latency	ISR time cost (µs)	Theoratical maximum latency (µs)	Measured maximum latency (µs)
IRQ5	500	0	500	0	0
IRQ4	441	59	10	500	494
IRQ3	440	60	10	510	488
IRQ2	448	52	10	520	502
IRQ1	444	56	10	530	488

Table 3-12. Comparison of theoretical and measured interrupt latencies

3.6 Summary

In this chapter, a SystemC-based mixed timing software behavioural modelling and simulation approach and the Live CPU Model have been introduced.

In the context of TLM software computation modelling, two types of software timing models were proposed for use in different software modelling stages. Also, they can be mixed in simulation for modelling flexibility. By isolating the timing modelling aspect from the timing simulation aspect, various timing annotation granularities (i.e., task-level, function-level, segment-level, and basic block-level), functional accuracy levels (i.e., abstract and native-code), and time advance methods (i.e., variable-step and fixed-step) can be utilised on mixed timing software models for various sakes of fast simulation performance, modelling flexibility, simulation observability, and reasonable accuracy.

The proposed SystemC-based Live CPU Model can achieve interruptible software time advance and zero-time delayed interrupt handling latency in software simulation. The HW/SW synchronisation problem is solved without the need of fine-grained time annotation and advance. This approach avoids the annotation-dependent software time advance approach that may result in uninterruptible software timing simulation. The Live CPU model supports multiple execution modes, which could trade off simulation speed with simulation observability. The Live CPU Model also provides an essential Interrupt Controller Model, a real-time clock and some Virtual Registers to assist software simulation. In the context of a software PE model, the Live CPU Model behaves as the conceptual hardware part and is promising to be extended with SW/HW interfaces for inter-module communication.

Regarding the requirement of fast performance, a representative test program shows that the proposed mixed timing software models achieve about 200 to 3000 times speedup[3] to an ARM ISS simulator and the conventional fine-grained uninterruptible behavioural software model. The proposed abstract and native-code software models also show distinct simulation performance as expected. Various execution models of Live CPU Simulation Engine are tested in order to present their effects on simulation performance. In general, more time advance points in models inevitably incur more simulation overheads.

In this chapter, twofold timing accuracy of the simulation approach was measured in experiments. Firstly, focusing on timing accuracy of single task execution, with fine-grained segment-level annotations, the proposed native-codes only incur a 0.12% timing accuracy loss. Secondly, the basic time advance stopping latency and interrupt latency is evaluated by measuring interrupt latencies in simulation

[3] The variation in simulation speedup are mainly because of two reasons: firstly, different experiments and test settings affect the simulation speed of a specific experiment; secondly, experiments were carried out at different times when the overall functionality and complexity of the proposed software simulator were evolving, which affected simulation speeds. In general, compared to the KEIL ARM ISS, the proposed simulation approach has two or three orders of magnitude speedups in this thesis.

tests. The result accords with the theoretical value, i.e., zero-time latency. The re-sumable time advance method is demonstrated in a simulation case.

Chapter 4

A Generic and Accurate RTOS-Centric Software Simulation Model

In recent years, with embedded systems moving towards System-on-Chip platforms, the complexity of the hosted embedded software is increasing. The RTOS has become an essential software component in many real-time embedded systems, providing efficient resource sharing and controlling facilities as well as guaranteed services between upper-layer application software and underlying hardware resources. The traditional software simulation approach, which executes a real RTOS and fully developed applications in an instruction set simulator, appears to be time consuming. In order to speed up simulation performance and validate real-time embedded software early in the system-level design flow, researchers have proposed system-level SLDL-based behavioural software simulation as a new design paradigm.

RTOS behavioural modelling and simulation have been proposed as enabling techniques that simulate and evaluate different real-time embedded software design alternatives in the early design phases. They can be used to evaluate system-wide, dynamic, run-time properties of real-time software, such as scheduling policies, application execution times, and interrupt handling, etc. These methods usually build generic RTOS models that can provide various typical RTOS services or can be adapted to mimic behaviour or specific RTOSs. The RTOS model and abstract software models or native-code application software models are dynamically executed together in an SLDL environment on a host machine, in order to mimic timing and functional behaviour of a software system on a target platform.

4.1 Motivation and Contribution

Within the system-level RTOS modelling and simulation research area, there still exist some unaddressed aspects and issues for improvement. These relate to the functionality, timing accuracy, and simulation performance of simulation models. For example, from the perspective of maximising flexibility of system-level software modelling, designers may want to simulate multiple abstraction-level software models in one simulation framework. Current RTOS modelling research does not address integrating coarse-grained timed abstract task models (i.e., associated with best-case and worst-case execution times) and fine-grained timed native-code application software (i.e., associated with multiple delay annotations) in one simulator. Besides, from the perspective of practical RTOS engineering, some RTOS models provide simplistic task management and limited synchronisation services, which are inadequate to imitate the behaviour of a real multi-tasking RTOS. Furthermore, the low timing accuracy is a common, yet critical, problem borne by many RTOS modelling approaches. On the one hand, this is due to the lack of inclusion of RTOS services' timing overheads in modelling. On the other hand, some SLDL-based modelling methods rely excessively on the uninterruptible SLDL *wait-for-delay* time advance mechanism (see Section 3.1.1); consequently, task switches and HW/SW synchronisation can only happen at limited pre-defined time advance points.

In this chapter, a SystemC-based system-level RTOS-centric real-time embedded software simulation model is presented. Its objectives are fast simulation and behavioural evaluation of real-time embedded software with good flexibility and reasonable accuracy in early design phases. Dynamic execution scenarios of a modelled target system can be exposed by tracing diverse system events and values in simulation, e.g., RTOS kernel calls, RTOS runtime overheads, task execution times, dynamic scheduling decisions, task synchronisation and communication activities, interrupt handling latencies, context switch overheads, and other properties. The whole simulation framework integrates multi-tasking applications, RTOS, Live CPU and other hardware component models in a unified SystemC prototyping environment. The core is a generic RTOS simulation model, which supplies a set of fundamental and typical services including task management, scheduling services, synchronisation, inter-task communication, clock services, context switch and interrupt handling services, etc. These services refer to several commercial RTOS products and specifications in order to supply general and standard functions. With the aim of building a timing RTOS simulation model, timing overheads of various RTOS services and application tasks are also considered in the models.

All models in the simulation framework are implemented on top of the SystemC library. The basic SystemC core language and the OSCI referenced simulation kernel are used without modification.

In the remainder of this chapter, Section 4.3 introduces a typical embedded software stack and considers its inclusion within our simulation model. Section 4.4 presents background knowledge of real-time applications and the RTOS. Section 4.5 describes the RTOS-centric software modelling approach in detail. Sections 4.6 and 4.7 introduce evaluation metrics and experiments to demonstrate the simulation performance, function, and accuracy of RTOS-centric real-time software models. Finally, the chapter is summarised in Section 4.8.

4.2 Research Context and Assumptions

Referring to Figure 4-1, we have introduced software PE modelling in Section 2.1.3.3. The Live CPU Model, as described in Section 3.3, represents the hardware aspect of the software PE model. This chapter will introduce the behavioural RTOS-centric software simulation model, namely the software aspect of this software PE. The software simulation model runs on top of the Live CPU Model,

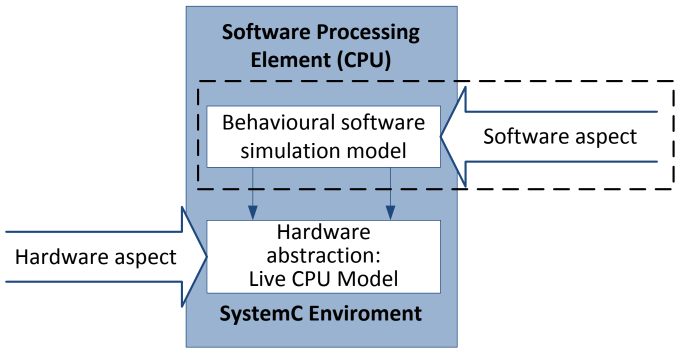

Software Processing Element (CPU)
Behavioural software simulation model
Software aspect
Hardware aspect
Hardware abstraction: Live CPU Model
SystemC Enviroment

Figure 4-1. Software part of the software PE model

so software simulation is guaranteed with reasonable timing accuracy and good HW/SW synchronisation (i.e., interrupt handling) timing accuracy. The whole software PE model is the research context, i.e., multi-tasking real-time applications and a RTOS run in a uniprocessor embedded system model.

Due to the high abstraction level of the software simulation approach in this thesis, advanced CPU architectures such as multiple-level caches and pipelines are not considered, i.e., their effects on software execution times are not explicitly modelled. However, according to the software performance estimation methods discussed in Sections 3.2.5 and 3.2.6, a KEIL ARM ISS without cache is used to measure software performance for back annotations of our software models in this thesis. In terms of other specific ISSes, caches may or may not be supported when the ISS executes software instructions, which means that caches can still affect timing accuracy of software time annotations. Hence, timing accuracy losses of software execution times - between the proposed behavioural software simulation, the referenced ISS, and the real hardware platform - are inevitable. Recalling the research intention of this thesis for fast and accurate software simulation, it is assumed that the referenced ISS is accurate enough to support and evaluate our behavioural software simulation.

As introduced in Section 3.3.1, the memory subsystem for actual software execution (e.g., RAM) is not included in the Live CPU Model because that it is not necessary for behavioural (i.e., abstract or native-code) software simulation. Hence, target software memory environments such as stack, heap, and memory protection, and RTOS memory management services such as swapping, paging, allocation, segmentation, and virtual memory, are also out of the modelling focus.

Nowadays, there are many general RTOS concepts, popular RTOS standards, and specific RTOS products. This thesis aims to present a generic RTOS model for behavioural real-time software simulation. It should be representative yet without a loss of generality. The selection and determination of functions and requirements of the RTOS model are made with reference to both some classical RTOS literature [25] [26], and some current RTOS specifications and products, including:

- The Didactical C Kernel (DICK) [25]: this is a small real-time kernel that introduces basic and important issues for designing a hard real-time kernel and hence informs our simulation model from the theoretical aspect.
- Real-Time extensions of the POSIX (Portable Operating System Interface) standard 1003.1 (referred to as RT-POSIX hereafter) [150]: this is a very broad and successful API standard particularly facilitating handling multi-threading and multiprocessing real-time applications. RT-POSIX is scalable with four subsets (namely Real-Time Profile PSE51 (minimal), PSE52 (controller), PSE53 (dedicated), and PSE54 (multi-purpose)) for different-scale systems. The RTOS model in this thesis refers to the PSE51 profile for small embedded systems.
- μITRON (micro Industrial The Real-Time Operating system Nucleus) 4.0 standard [151]: this standard is oriented to small/medium-size embedded systems. Over 40% of RTOSs used in Japan are based on this standard [129]. It inspires the task state machine in the proposed RTOS model.
- μC/OS-II [149], ThreadX [152], and Keil RTX (Real Time eXecutive) [1]: they are representative popular small-size RTOSs. Their functions and kernel structures mostly influence the proposed RTOS model from a practical engineering aspect.
- QNX Neutrino [147]: this is a RT-POSIX compliant multiprocessor-enabled high-end RTOS. Its implements basic thread and real-time services in the microkernel and can be extended to support multiple processes by adding optional components.

4.3 The Embedded Software Stack Model

The left part of Figure 4-2 depicts a typical embedded software stack. It includes three layers, i.e., the application software layer, the middleware layer, and the system software layer. According to the research context and intention of this chapter, the software stack needs to be abstracted into a model in order to accommodate software components for high-level modelling and simulation.

The right part of Figure 4-2 illustrates the abstract model of the embedded software stack. The application layer is obviously essential to be included, be-

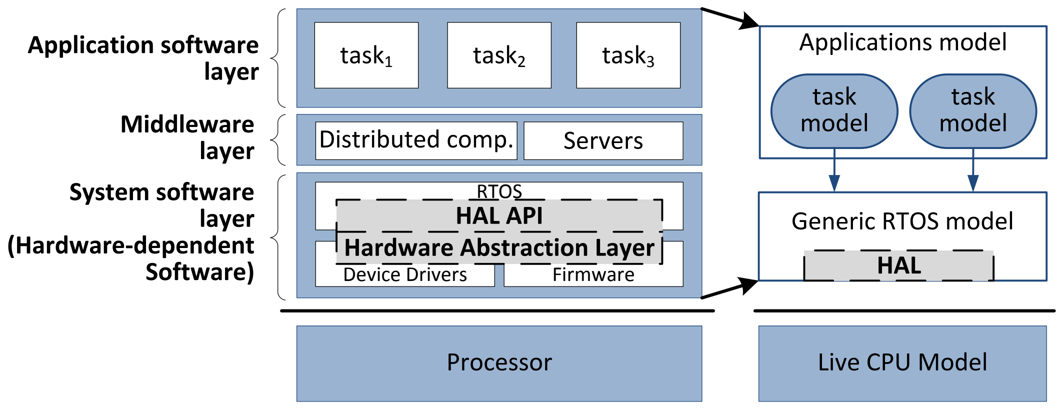


Figure 4-2. Embedded software stack and its abstract model

cause it represents some actual functions of the embedded software. It is necessarily one of the main targets of the modelling research. However, the middleware layer is not considered. This is because it is oriented to complex and distributed applications, so it is not fundamental for early-phase real-time software simulation and is beyond this research.

The system software layer of an embedded software stack always includes various HdS components such as device drivers, boot firmware, and RTOS.

Device drivers are essential in real computer software systems to provide specific services for software to accesses hardware resources, namely I/O services [153]. However, for the proposed system-level software modelling and simulation approach, these are largely outside the scope of the research. We note that development and evaluation of device drivers is usually carried out with ISS simulators or fully functional hardware prototypes at a relatively later design phase, which differs to the research assumption of this thesis. Consequently, device drivers are not addressed in the real-time software modelling and simulation approach. The same consideration applies to boot firmware as well.

In some embedded system contexts, HAL is a meaningful and abstract concept referring to the lowest system software components, which directly access hardware resources and totally depend on the target architecture [22] [154]. Figure 4-2 shows the conceptual location of the HAL inside the system software layer. The HAL components define platform-specific data types, cover hardware-dependent parts of device drivers, and especially include processor-specific software code (e.g., boot firmware, context switches, processor mode change, and interrupt configuration functions) [153]. The intention of HAL is to ease HdS porting on different hardware architectures by separating HdS into the hardware-independent part (e.g., most RTOS services) and the hardware-dependent part (e.g., HAL). Hence, in software development, the hardware-independent part can possess reusability over different architectures to some extent. Only the hardware-dependent part needs hardware-specific development. Furthermore, by means of using HAL APIs, upper-level application software can utilise abstract hardware resources early in the design flow before the hardware architecture is fixed and finished, which embodies a reuse concept.

One issue is where the HAL should appear in the embedded software stack model. Firstly, consider the hardware and processor resources available. In Section 3.3, the Live CPU Model has been introduced as the underlying hardware model for software simulation. It can provide essential hardware resource for modelling interrupt-based HW/SW interaction and clock services. In addition, in the forthcoming Chapter 5, the Live CPU Model will be extended with TLM interfaces for inter-module communication modelling. Based on these foundations, there is a necessity to provide a HAL model in the software modelling stack which can offer a set of low-level hardware-related functions. By this means, application software and RTOS models can utilise and configure the Live CPU Model for timing simulation, and can access other hardware resources. These HAL functions include context switches, interrupt handling, critical section control, and TLM transfers etc. For the purpose of simplifying model structures, the HAL model is implemented as a number of member functions inside the SystemC module of the RTOS model. The external behaviour and interface of the generic HAL model is similar to what is used in a typical embedded software stack. However, the exact functionality of some parts of the HAL model is only applicable for simulation purpose in this thesis, which means it is different to the HAL code that is finally implemented.

4.4 Common RTOS Concepts and Features

4.4.1 "Real-Time" Features of Embedded Applications

Concurrent real-time application software is divided into several tasks that are organised (scheduled) by the RTOS. In the general context of real-time systems, there are different kinds of embedded applications depending on timing stringency of tasks, which mean whether an application task must finish its execution within a time interval - the *deadline*. We can categorise real-time embedded applications into two classes:

- Non-Real-Time (NRT) applications: where tasks do not have deadlines;
- Real-Time applications: where tasks have deadlines. Moreover, this class of applications can be further distinguished as *hard*, *firm* or *soft*. *Hard* real-time applications are imperative for finishing execution within the required deadline; otherwise, catastrophic consequences may be the result. Examples can be found in sensor data acquisition and low-level control components in avionics and automotive electronics. In *soft* real-time applications, meeting deadlines is still of importance in terms of concern regarding performance. But if deadlines are occasionally missed, applications can still function correctly and do not result in serious failure. Handling input from the keyboard and displaying information on the screen are examples of soft real-time applications [25]. As a variation of soft real-time applications, *firm* real-time applications result in neither functional benefits nor a total failure from missing deadlines.

In a real-time system, the RTOS must be able to handle hard real-time applications to fulfil strict requirement of deadlines. In addition, because there are different types of applications in the real world, a RTOS may need to support a hybrid NRT, hard and soft real-time application set. In real-time systems research, some approaches have been proposed to not only guarantee timing constraints of hard real-time tasks, but also optimise the average performance of NRT and software real-time tasks. For example, the hierarchical scheduling schemes use *global* (so-called *kernel-level* or *system-level*) and *local* (so-called *user-level* or *subsystem-*

level) schedulers to schedule various applications and their inclusive tasks by different scheduling algorithms [26] [155]. There have been some attempts to consider this hybrid application problem both on top of existing RTOSs and in the design of new RTOS research kernels, e.g., hierarchical scheduling extension on top of VxWorks [156] and Soft Hard Real-time Kernel (SHaRK) [157]. However, as indicated in [26] "*most OSs schedule all applications according to the same scheduling algorithm at any given time*" – currently, most popular commercial and open source RTOSs do not have explicit special mechanisms to effectively support NRT, soft, and hard real-time applications running in the same environment.

A real-time task has some timing properties that need to be aware or considered in RTOS management and scheduling. Referring to Figure 4-3, typical timing parameters of a real-time task usually consist of [21] [25]:

- Arrival time (*a*): also called *release time*, which means the time point when a task is ready to execute.
- Offset (*O*): the time length between the arrival time and time point 0. In RTOS execution, it refers to the possibility that different tasks may not simultaneously become ready to run after the system is started up.
- Worst-Case Execution Time (WCET): the longest possible execution time of a task.
- Best-Case Execution Time (BCET): the shortest possible execution time of a task.
- Execution time (*E*): the actual execution time of a task, which is the time length between the start time (*s*) and the finish time (*f*). It should reside in

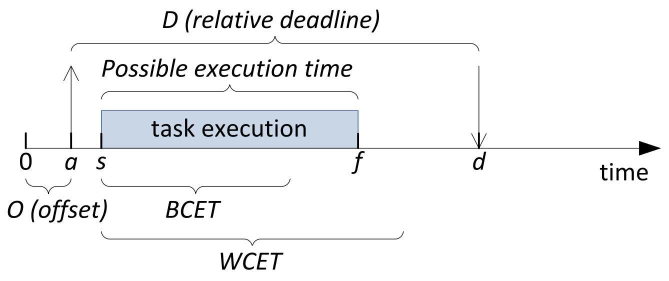


Figure 4-3. Timing parameters of a real-time task

the range of the BCET and the WCET. Note that *s* could be later than *a*, because a ready task may need to wait for other higher-priority tasks to finish.

- Absolute deadline (*d*): a time point before which a real-time task must complete its execution, otherwise undesired consequences will happen.
- Relative deadline (*D*): the time length between the arrival time and the absolute deadline, i.e., $d = a + D$.

It is very common that a real-time task will need to regularly or irregularly repeat its execution. Based on the periodic characteristic, tasks can be classified into three types:

- Periodic: a task executes once in every regular time interval, i.e., a period (*T*). Each execution is called an instance or a job. In RTOS execution, a periodic task can be triggered either by an external periodic event or by the clock tick timer.
- Aperiodic: a task may execute once or many times, but its activation rate is not constant. In RTOS execution, an aperiodic task is usually used to handling interrupt events.
- Sporadic: is an aperiodic task but includes a minimum time interval between its two executing jobs.

4.4.2 RTOS Kernel Structures

While different operating systems vary in terms of what components they contain, the *kernel* is the core part of a RTOS. A RTOS kernel must at least provide basic functions with respect to task management, interrupt handling, intertask synchronisation and communication [25]. Some large kernels may also wrap additional system software modules such as drivers and file systems, but this is not common in RTOSs. In fact, many RTOSs can actually be seen as kernels because of their limited functionality and the small size. Application tasks can access kernel functions and data through a series of source-level API functions. In some embedded systems, the kernel and application software may have their own memory address spaces for the purpose of memory protection. In real execution, a call to an API function is known as a *system call* (see Figure 4-4), which is effectively

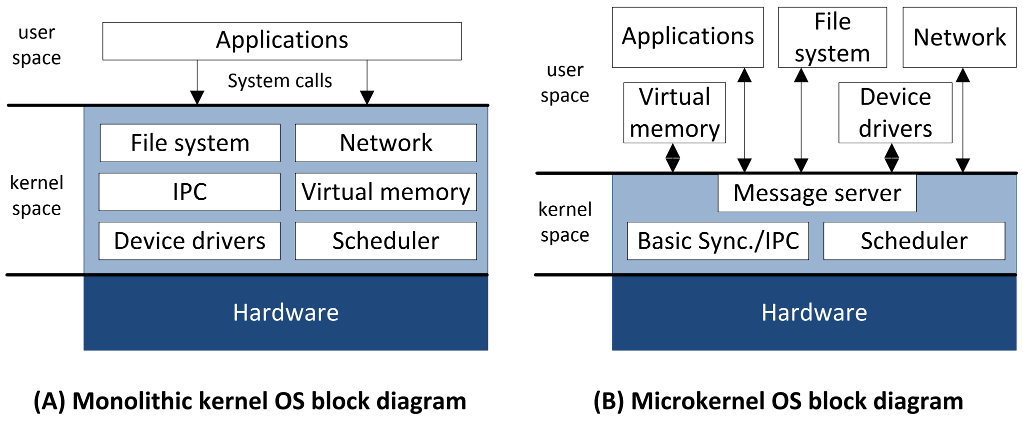


Figure 4-4. Block diagrams of two RTOS kernel approaches

a software interrupt executed by a *trap* instruction. When receiving a system call, the kernel firstly saves the calling task's context, and switches the system status from the *user* (also known as *application*) mode to the *kernel* (also known as *supervisor*) mode by changing a bit of the processor status register, then finally executing the requested OS function on behalf of the calling task. After the kernel finishes a system call, an opposite mode transition occurs. Note that some embedded processors are lacking of a Memory Management Unit (MMU) which supports virtual memory and some RTOSs do not provide memory protection mechanisms; hence the kernel and the applications exist in the same memory space and a system call is similar to a function call inside the application task [26].

As shown in Figure 4-4, depending on the internal structure, there are two traditional kernel design approaches that exist within operating system research: the *monolithic kernel* approach and the *microkernel* approach. In addition, the *nanokernel*, the *hybrid kernel*, and the *exokernel* are other common RTOS kernel architectures, but are far from the focus of this research. More complete surveys on RTOS kernel design can be found in [17] (Chapter 11), [18] (Chapter 9), and [158]. It is noticed that, in the context of discussing the small size characteristic of a RTOS core, some literature may interchangeably use "microkernel" and "kernel" [26]. Additionally, some literature explains the "microkernel" as a "slim kernel" that only supplies scheduling service [159]. However, when discussing how a kernel is realised, researchers should use "microkernel" to refer to a specific kind of kernel structure.

The monolithic kernel is a conventional RTOS design approach and is popular for small or deeply embedded applications [17]. Referring to Figure 4-4 (A), it implements all OS services (e.g., scheduler, task management, synchronisation, inter-process communication (IPC), memory management, interrupt handlers) and some system software modules (device drivers, file systems and network stacks) in the kernel space. That is to say, the monolithic kernel itself equals the entire RTOS subsystem. RTOS service functions directly call each other as they need. The main advantage of the monolithic kernel approach is straightforward usage and fast performance due to simple function calls [158]. However, the tight integration of many components in the kernel is error-prone, so that a bug in one module can bring down the whole system [160]. VxWorks [161] and μC/OS-II are often cited as monolithic kernel RTOS examples.

As shown in Figure 4-4 (B), the microkernel approach only provides a few essential OS services in the kernel space such as task management, scheduler, basic synchronisation/IPC, and a message manager [158] [162]. Other services are usually provided as normal server processes running in the user mode. A message passing system is introduced to support communication between these server processes [163] [164]. Applications request services from these servers via system calls through a client-server method. The loose-coupling modularity and clear separation between kernel services and user-level services make a microkernel RTOS more reliable and compact. However, when processing a system call, the client-server service model may bring more run-time context switches from an application's memory space to the server's memory space, resulting in intense message communication overheads. For these reasons, the microkernel approach is seen as a promising method suitable for complex and scalable RTOSs such as the QNX Neutrino RTOS.

4.4.3 RTOS Requirements and Modelling Guidance

Although diverse RTOSs vary in terms of size, functionality and application domain, they do have some common requirements and characteristics that differentiate them from general-purpose OSs. In behavioural software modelling and simulation research, we need to consider which features are necessary and how they can be embodied in the RTOS simulation model. The following subsections contain key aspects related to this problem.

4.4.3.1 Predictable and Responsive Timing Behaviour

A good RTOS should not only provide efficient OS services, but also keep its own time consumptions and response times predictable and accountable [26]. Ideally, a RTOS needs to guarantee execution time of each service as a fixed value, or at least indicate a trend with an upper bound under all system load circumstances. Given the scheduler function as an example, the μC/OS-II RTOS looks up a table to find the highest priority task and the task scheduling time is constant in spite of the number of tasks created [149]. In contrast, the Olympus real-time kernel moves tasks between two queues when it makes a scheduling decision; hence the overhead of the scheduler varies depending on the number of queue operations. Its worst-case execution time can be computed according to the number of tasks in the system [165].

In addition to predictability, "fast real-time performance", or analogous "rapid real-time response", is the top RTOS feature concerned by many real-time embedded software developers [166]. This issue reflects the real-world requirements of a real-time system in terms of promptly processing interrupt events within a bounded amount of time. Failure to respond may result in a failure of the real-time embedded system.

Two foremost timing properties (latencies) are usually used to evaluate the response capability of a RTOS, namely *interrupt latency* and *task switching latency* [167] [168]. Typically, they are in order of a few or a few tens of microseconds [26]. Figure 4-5 shows two different interrupt handling schemes (i.e., the RTOS-assisted scheme and the vector-based scheme) used by two RTOS products [149] [166]. They are good examples in terms of diverse views on interrupt latency and task switching latency:

- *Interrupt latency* is usually defined as the elapsed time between the occurrence of an interrupt to the entry (first instruction) of the corresponding software interrupt handler. In the RTOS-assisted interrupt handling scheme in Figure 4-5 (A), the interrupt handler includes two parts, i.e., the kernel

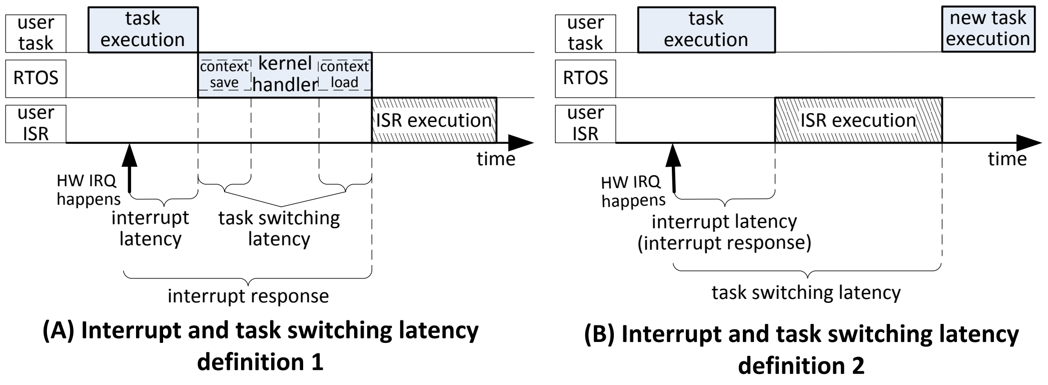


Figure 4-5. Two definitions of interrupt latency and task switching latency

handler and user ISR handler. The interrupt latency refers to the elapsed time between the interrupt event and the beginning of the kernel handler. It uses the term *interrupt response* time to define a longer elapsed time between the interrupt event and the beginning of the user ISR. In contrast, in the vector-based interrupt handling scheme in Figure 4-5 (B), the user ISR is the only interrupt handler in charge and it can be activated directly. Therefore, the interrupt latency is the same as the interrupt response time and their definition are shown in the figure.

- *Task switching latency* is sometimes interchangeably used with the term *context switching latency* [168] [149] [167]. In Figure 4-5 (A), task switching latency refers to the time of two portions, i.e., the time to save the currently executing task's context and the time to load another task's context. It is shorter than the interrupt time. However, in Figure 4-5 (B), task switching latency refers to the time elapsed from the interrupt event to the beginning of a task that is activated because of the interrupt. It is greater than the interrupt latency (response). When comparing the two definitions of task (context) switching latency, we can see that the first definition reflects a point of view of the processor context switch, since it effectively refers to the time consumed by the processor to save and load the context of registers. Whereas, the second definition reflects an OS context switch viewpoint, because it counts for the total switching time used by the RTOS to save and load tasks. To eliminate the ambiguity, this thesis uses the first definition.

4.4.3.2 Multi-Tasking Management

To support complex real-time applications in the real world, a RTOS should provide multi-tasking services [168]. For the majority of embedded software systems, applications are usually subdivided into a set of concurrent units. These concurrent units are usually described via three terms: *task*, *thread* and *process*. So far in this thesis, the term *task* is mostly used to refer to an execution unit of a software application. Before capturing the behaviour of application software entities in some different RTOSs and standards, the three terms are firstly clarified and differentiated. Notwithstanding all of them are widely used, they come from various contexts and are sometimes obscurely defined. The problem can be discussed in two contexts.

Firstly, in some complex RTOS environments, e.g., RT-POSIX compliant or UNIX/Linux-originated RTOSs, the operating system can support separate memory spaces for different execution entities. In this case, the term process refers to an executable program with its own protected memory environment that includes processor registers, I/O addresses, and memory-management information [21]. It is noticed that switching from one process to another involves much saving/loading work for heavy context information, which is time-consuming in execution. In order to reduce context switch overheads and avoid the expense of memory protection, most RTOSs support the thread concept as a smaller, semi-independent, execution unit compared to a process. Multiple threads can be created within the memory space of a process/program. Threads within a process can unrestrictedly share everything of their parent process, whilst they only keep limited private information. A thread is also named as a lightweight process, and a process can be seen as a thread container. In terms of the meaning of a task, it is described as interchangeable with either a process or a thread in different contexts. For example, a task is defined as equal to a process and can contain lower-level threads in [169]; whilst, [21] specifies that a task is interchangeable to a thread and executes within the memory context of a process. To remove ambiguity, this thesis complies with the latter definition regarding the three terms.

Secondly, for some simple RTOSs, where software execution units do not have independently protected memory spaces (i.e., they share the same memory ad-

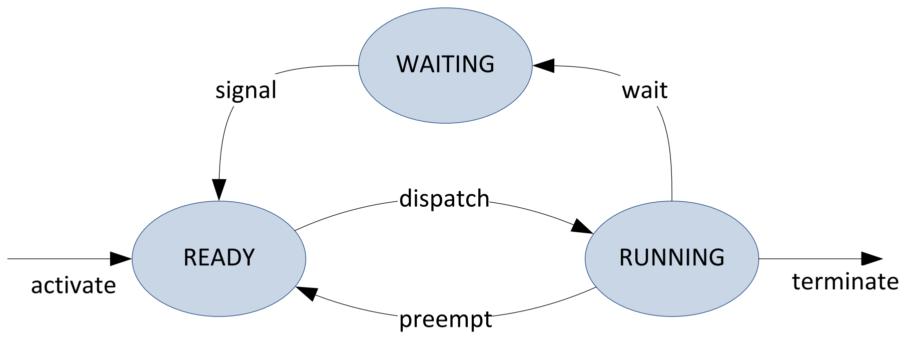
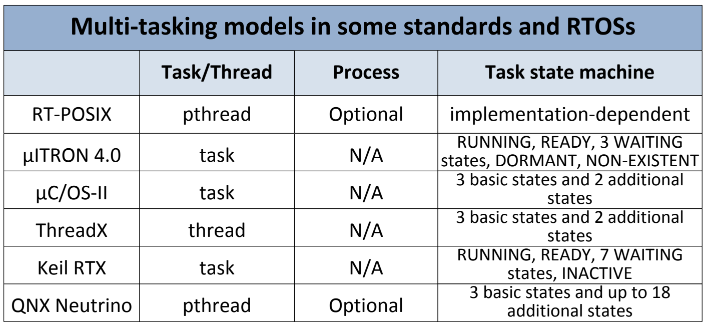


Figure 4-6. The classical three-state task state machine

dress), then there is only one kind of concurrent entity in the system. Hence, people can interchangeably use the term task or thread to refer to a basic concurrent activity, but the term process is not appropriate to use in this context.

After applications are divided into multiple tasks, the RTOS enables them to execute (namely occupy the processor resource) interchangeably in order to finish their jobs and meet their respective deadlines. Task management services are implemented inside or utilised by various higher-level scheduling, synchronisation, and RTOS initialisation services. Typical multi-tasking primitive functions include creating tasks, suspending tasks, resuming tasks, and terminating tasks, etc. These functions control state transitions of tasks during their execution. Figure 4-6 shows a classical minimal RTOS task state machine that has three basic states: RUNNING, READY, and WAITING [25]. In a RTOS execution, a task must stay at one of them:

- RUNNING: in a uniprocessor system, only one task can enter this state and execute at a time. If the RUNNING task is pre-empted, then it enters the READY state.
- READY: Tasks at this state are eligible for execution, but cannot execute immediately as another task is currently at the RUNNING state. All READY tasks are organised in a queue by the RTOS kernel. This is named the *ready queue* [25]. The scheduler regularly checks the *ready queue* and the RUNNING task (if there is one) according to various scheduling policies, in order to dispatch a new task to run when the scheduling policy permits.



Multi-tasking models in some standards and RTOSs			
	Task/Thread	Process	Task state machine
RT-POSIX	pthread	Optional	implementation-dependent
μITRON 4.0	task	N/A	RUNNING, READY, 3 WAITING states, DORMANT, NON-EXISTENT
μC/OS-II	task	N/A	3 basic states and 2 additional states
ThreadX	thread	N/A	3 basic states and 2 additional states
Keil RTX	task	N/A	RUNNING, READY, 7 WAITING states, INACTIVE
QNX Neutrino	pthread	Optional	3 basic states and up to 18 additional states

Table 4-1. Multi-tasking models in some RTOS standards and products

- WAITING: Tasks enter this state (also known as a SUSPENDED or BLOCKED state in some contexts) when they are blocked. Reasons may include waiting for a synchronisation primitive or sleeping for some time. All WAITING tasks are organised in the *waiting queue*. When the unblocking condition of a WAITING task is satisfied, the task enters the READY state.

Additional states or sub-states are always introduced to extend task state machines in different RTOSs, in order to support more task state transitions and RTOS services. Table 4-1 surveys multi-tasking/threading/processing models and task state machines in some RTOS standards and products.

4.4.3.3 Pre-Emptive and Priority-Based Scheduling

Scheduling can be triggered by other RTOS services or an interrupt. It decides which task to dispatch next according to a real-time scheduling policy. Most RTOSs support multiple scheduling policies and users can specify one or several policies for their application software. In general, pre-emptive and priority-based scheduling policies are commonly required by most RTOSs [170].

Pre-emptive scheduling policies are in contrast to non-pre-emptive policies. In pre-emptive scheduling, a READY task can pre-empt the RUNNING task, whilst, in non-pre-emptive scheduling, the RUNNING task executes until it finishes or calls the kernel to relinquish the CPU. The First-In-First-Out (FIFO) algorithm behaves as a non-pre-emptive policy if it is applied as the only scheduling policy in a system without prioritisation. According to this, tasks execute in the order that they become READY. Certainly, pre-emptive scheduling policies are preferable in RTOSs, because they can respond to external events in a timely manner.

In priority-based scheduling, all tasks are assigned priorities according to criteria such as the period (so-called *rate*) or the deadline. Priority-based scheduling is natively pre-emptive as long as a higher-priority task is allowed to pre-empt a lower-priority task. Depending on whether the priorities of tasks are assigned before execution or are dynamically assigned in execution, there are two types of priority-based scheduling policies, i.e., Fixed-Priority Scheduling (FPS) and Dynamic-Priority Scheduling (DPS). The FPS scheme is easy to implement in RTOS design because the RTOS kernel needs only to maintain a priority queue or a priority table in execution.

In FPS research, a priority assignment is an important problem and can be seen as a prerequisite of FPS. However, it is not directly related to RTOS design and implementation, because users mostly specify priorities of their application tasks. Rate Monotonic (RM) priority ordering is the most common priority assignment algorithm for periodic FPS systems. In RM, tasks are assigned fixed priority levels that are inversely proportional to their rates, i.e., the shortest period task is assigned the highest priority. The simple periodic task model assumption has been shown to be optimal in all FPS policies, which means that if it cannot schedule a task set, then no other FPS algorithms can do so either [171]. There also exist some other priority assignment policies, such as Deadline Monotonic [172] priority ordering and the Optimal Priority Assignment [173], which have been proven to be optimal with their specific assumptions. A review on this topic can be found in [174].

In the context of FPS, considering the situation whereby multiple tasks may share the same priority level in a RTOS, a FIFO algorithm can be used as an assistant policy in this case. However, this is not ideal, because a task may monopolise the CPU for a very long time while other equal-priority tasks are starving. A Round-Robin (RR) scheduling policy is proposed to tackle this problem. It allocates each task a maximum amount of executing time, so-called a *time slice* or a *quantum*. Once a RUNNING task has exhausted its quantum, then it is moved to the tail of its priority queue by the scheduler and the head task of the priority queue will be dispatched.

The Sporadic Server scheduling algorithm is introduced to improve the average response time of aperiodic tasks in fixed-priority systems [175]. It creates a high-priority server (a task) for serving aperiodic tasks. The server is allocated an amount of processor time, i.e., *execution capacity*. Aperiodic tasks execute at the priority level of the server and consume execution capacity. If substantial aperiodic task execution totally consumes execution capacity, then the server priority is decreased to a low priority level, which means aperiodic tasks then execute at a low priority without the possibility of frequently pre-empting other periodic tasks. The execution capacity can be replenished periodically according to the *replenishment period*.

In terms of DPS policies, the Earliest Deadline First (EDF) scheduling policy is probably the most notable one. In EDF, priorities are inversely proportional to absolute deadlines of tasks and are dynamically assigned to them. It has been demonstrated that EDF is optimal for uniprocessor system in terms of fully utilising the processor bandwidth [171]. EDF also has some other theoretical advantages compared to FPS in terms of less schedulability analysis complexity and lower context switch overheads [25]. However, a major disadvantage prevents EDF from implementation in common commercial RTOSs: EDF requires the RTOS kernel to track and update absolute deadlines and priorities of tasks at each job activation, which increases the complexity of the kernel and brings run-time overheads [176]. Consequently, nowadays the EDF scheduler is mostly provided in some research RTOS kernels such as SHaRK [157] and MaRTE OS [177]. SHaRK implements the EDF scheduler as an external scheduling module, which is used by its core generic kernel to schedule tasks. In order to popularise application-defined scheduling and standardisation, MaRTE OS uses RT-POSIX interfaces to introduce a user-thread level EDF scheduler. However, the literature [26] has argued that the later method may incur expensive overheads from excessive system calls of tracking system time and setting tasks' priorities.

Although there is a fair amount of research on real-time scheduling strategies today ([25] surveys many details), FPS is the most common priority-based and

Scheduling policies in some standards and RTOSs				
	Pre-emptive FPS	**FIFO**	**Round Robin**	**Sporadic Server**
RT-POSIX	Yes	Yes	Yes	Yes
μITRON 4.0	Yes	Yes	Yes	N/A
μC/OS-II	Yes	N/A	N/A	N/A
ThreadX	Yes	Yes	Yes	N/A
Keil RTX	Yes	Yes	Yes	N/A
QNX Neutrino	Yes	Yes	Yes	Yes

Table 4-2. Scheduling policies in some standards and RTOSs

pre-emptive scheduling policy implemented in most commercial RTOSs and standards due to its simplicity in engineering and low execution overheads. Table 4-2 surveys the scheduling policies defined and implemented in some standards and RTOSs.

4.4.3.4 Sufficient Priority Levels

In order to apply priority-based scheduling policies, a RTOS must have a sufficient number of priority levels [178]. Exactly, there are some points that should be considered when implementing prioritisation in a RTOS simulation model.

In order to support complex user applications that each may contain multiple tasks, sufficient priority levels should be provided to differentiate relative importance or timing requirements of tasks. In addition, a range of priority levels should be reserved for user-level interrupt handlers that usually require higher priorities than normal tasks. Additionally, NRT or soft real-time tasks need to be allocated a range of priorities that are lower than normal hard real-time tasks.

From the perspective of priority assignment theory, the most-mentioned RM priority ordering algorithm requires that tasks have distinct priorities and that there are unlimited priority levels. However, the number of priority levels cannot be infinite in practical RTOS design. Utilising limited priorities in FPS systems is a traditional topic in real-time systems research. Literature [26] has addressed this issue by answering two questions from a theoretical aspect: 1) How long is a task delayed by equal-priority tasks when using FIFO or RR scheduling policies? 2) How does the processor utilisation deteriorate when using limited priority levels in the RM algorithm? In [26], these two questions are concluded with favourable

Priority levels in some standards and RTOSs				
	Priority levels	Priority range of user tasks	Lowest priority of user tasks	Sharable priority
RT-POSIX	Each scheduling policy has at least 32 levels	User-defined	User-defined	Yes
μITRON 4.0	At least 16 levels	User-defined	1	Yes
μC/OS-II	64	4-59	59	N/A
ThreadX	32	0-31	31	Yes
Keil RTX	255	1-254	1	Yes
QNX Neutrino	256	1-63	1	Yes

Table 4-3. Priority levels in some standards and RTOSs

assurance[4]. The author deems that 256 priority levels can perform very well even, for the most complex FPS systems.

According to the survey in Table 4-3, priority levels range from 16 to 256 across different RTOS specifications and products, and there are no special restrictions on what priority levels are available to user tasks and which priority represents the lowest level.

4.4.3.5 Resource Access Control Protocols

Tasks may contend for shared resources (e.g., registers, variables, data structures, memory areas) in order to communicate or process data in execution. It is necessary to guarantee that operations on a shared resource are carried out in a consistent and protected manner, which means that a shared resource can only be used by one task at a time, i.e., achieving *mutually exclusive* access. The code segment modifying the mutual exclusive resource is called a *critical section*, and its instructions need to execute sequentially without interruption. Like general-purpose OSs, almost all RTOSs provide some conventional lock-based synchronisation mechanisms (e.g., mutexes and semaphores)[5] to implement mutually exclu-

[4] It is reported that, compared to a RM system with 100,000 priority levels, the relative schedulability of the system with 256 levels is merely reduced to 0.9986 [26].

[5] Semaphores can be divided into two classes depending on their value, i.e., the general-form counting semaphore with a non-negative integer value, and the simple-form binary semaphore with values of zero or one. The later one can be used for mutual exclusion. Mutexes can be seen as a specialized binary semaphore used for mutual exclusion only [17].

sive access to shared resource through atomic primitives *wait* and *signal* (also called *P/V* or *lock/unlock* operations).

When a mutually exclusive resource is held by a task using a lock-based synchronisation method, then other competing tasks that want to acquire the resource cannot get the resource immediately and are said to be *blocked*. The notable *priority inversion* phenomenon refers to the situation where a high-priority task is blocked on a resource that is already locked by a low-priority task. The problem becomes more severe considering the low-priority task may in turn be pre-empted by one or more intermediate-priority tasks, which is referred to as *transitive blocking*. In these cases, the high-priority task cannot enter its critical section and needs to wait for the finish of both the low-priority task and some (the number may be uncertain) intermediate-priority tasks. This means that the duration of priority inversion is unbounded; hence, the finish time of the high-priority task is unpredictable.

In order to solve the unbounded priority inversion problem, real-time systems research has proposed some resource access control protocols. The Priority Inheritance Protocol (PIP) [179], the derived Priority Ceiling Protocol (PCP), and the further-improved Immediate Priority Ceiling Protocol (IPCP) [180] are three of the most well-known protocols applied to FPS systems. They can be bracketed into the same PIP protocol family because of their close relevance. The basic idea behind them is similar: the priority of the task that incurs a blocking is temporarily changed to a higher priority that is inherited according to some algorithms; then the task can execute through its critical section without being pre-empted by a medium-priority task; and finally the task's priority is restored after it exits the critical section. However, due to their difference in protocol definitions, these three protocols possess variant features. In general, PIP suffers from a potential long blocking duration, chained blocking and deadlock, but it does not require prior knowledge about the resources shared by tasks and hence is easy to implement at the user level on top of an existing RTOS. Compared to PIP, PCP can prevent deadlock and chained blocking. However, it needs the software programmer to define a ceiling priority for each shared resource and the OS kernel needs to keep tracking ceiling values and task priorities, which means both implementa-

Resource access protocols in some standards and RTOSs		
	Priority Inheritance Protocol	Immediate Priority Ceiling Protocol
RT-POSIX	Yes	Yes
μITRON 4.0	Yes	Yes
μC/OS-II	Yes	N/A
ThreadX	Yes	N/A
Keil RTX	Yes	N/A
QNX Neutrino	Yes	N/A

Table 4-4. Resource access protocols in some standards and RTOSs

tion complexity and run-time overheads [17]. Furthermore, IPCP improves the PCP in terms of being easier to implement and with low overheads. Exact definitions, analysis and comparisons of these protocols can be found in [25] and [181].

Some RTOS standards and commercial products have implemented one or some of above protocols. Because the above-mentioned classical PIP family protocols assume that there is only one unit of each shared resource, they are naturally employed on the mutex synchronisation mechanism that provides mutual exclusion to a single-unit resource. Table 4-4 summarises the resource access control protocols utilised in some standards and RTOSs, where PIP is the most common protocol and IPCP is also provided in RT-POSIX and μITRON specifications, although the PCP does not appear in the survey.

Although binary semaphores can also be used for mutual exclusion, access control protocols are not applied to them in most OS specifications[6] [150] [152] [1]. Instead, semaphores are mainly used for event notification and thread synchronisation through an embedded counter, i.e., in the form of a counting semaphore. In addition, it is noticed that some access control protocols (derivatives of PIP and PCP [26]) can support safe access to multiple-unit resources, which means usability for counting semaphores. However, they have not attracted much interest from RTOS designers.

[6] The RTEMS RTOS [182] is an exception that it supports PIP and PCP on binary semaphores. Actually, RTEMS provides functions of the mutex mechanism through its binary semaphores.

4.4.3.6 Summary of RTOS Features in the Model

In this thesis, features of the proposed RTOS simulation model are mainly determined based on surveys in above Sections 4.4.3.1 to 4.4.3.5. In order to be practically useful for current system-level embedded software development, this research prefers to model some common characteristics and services of the surveyed RTOS standards and products, rather than invent and integrate too many proprietary features and theories. However, the two surveyed RTOS standards (i.e., RT-POSIX and μITRON) and four RTOS products (i.e., μC/OS-II, ThreadX, RTX, and QNX Neutrino) combine a wide range of RTOS services and features, which are too broad to be included in the generic RTOS simulation model. Instead, the three small-size RTOSs (i.e., μC/OS-II, ThreadX, RTX) are used a focus for RTOS modelling.

Regarding the predictable and responsive timing behaviour of a RTOS (Section 4.4.3.1), the RTOS modelling approach attempts to model common RTOS situations from two aspects.

1) Firstly, this thesis considers the timing latencies introduced in Section 4.4.3.1, providing annotations for all related RTOS services' timing overheads in simulation models. Normally, the timing overhead of a RTOS service is annotated at the service/function level or statement segment level if possible. Usually, timing accuracy of the RTOS model is sufficient if the execution time of each service can be obtained before starting simulation, namely its value is fixed and obtainable. However, if a service's timing overhead is dynamically determined in execution, then a simple calculation function could be inserted in the model to sum the aggregated timing overhead. Otherwise, a degradation of timing accuracy occurs, which may not be appropriate for the real-time systems being modelling (i.e., if there are hard deadlines). Section 4.5.9.2 will describe the general method of how a RTOS service is modelled with timing information.
2) Secondly, the thesis aims to simulate the common interrupt handling processes and other services of real RTOSs, in order to represent timing behaviour of a system in simulation accurately. In this thesis, the modelling and simulation approach supports the above-mentioned two interrupt handling

schemes (Figure 4-5). Section 4.5.7 will introduce them in detail. Note that some over-complex or proprietary RTOS functions are not implemented in the models. Consequently, their timing behaviour cannot be represented in simulation.

In terms of multi-tasking management (Section 4.4.3.2), essential multi-tasking services are implemented in the RTOS simulation model. Especially, in order to model the possible states of concurrent task execution across various RTOSs without loss of generality (see the survey in Table 4-1), a generic extensible task state machine is used. Section 4.5.4 will introduce this multi-tasking model and associated services in detail.

The state of the art analysis on RTOS scheduling in Section 4.4.3.3 gives direction for the RTOS scheduler model. The basic RTOS scheduler model is fixed-priority and pre-emptive, and FIFO and RR algorithms are supported to handle equal-priority tasks. The sporadic server algorithm is not currently considered, because it is not normally implemented in small-size RTOSs, but it could be added into the RTOS model in future development. Shi introduced a detailed implementation of adding a RT-POSIX sporadic server scheduler in a Linux-based RTOS [183], which is a good reference on this issue. Besides, although the EDF policy is rare in practical RTOSs, the RTOS model does have included some mechanisms to natively support a kernel-level EDF scheduler as an add-on feature. Section 4.5.5 will detail scheduler modelling work.

Referring to the survey in Section 4.4.3.4, the priority system of the RTOS model is flexibly defined and can be adapted to different configurations depending on the modelling target. This part of work will be presented in Section 4.5.5.1.

Regarding the resource access control protocols in Section 4.4.3.5, PIP is implemented on the mutex synchronisation service in the RTOS model. Section 4.5.6 will address this issue.

4.5 The Real-Time Embedded Software Simulation Model

4.5.1 Simulation Model Structure

Figure 4-2 illustrates the real-time embedded software stack model in a simple block diagram. Now, Figure 4-7 shows a more detailed view of the architecture of the layered real-time embedded software PE model. This software PE model is composed of three layers: the application software layer, the RTOS layer and the hardware layer. Various sub models are contained in the whole simulation model in order to embody the three layers' functional and timing behaviour. This section will generally introduce their interrelationship and corresponding SystemC-based models. Detailed modelling methods and implementations are presented later in Sections 4.5.2-4.5.9.

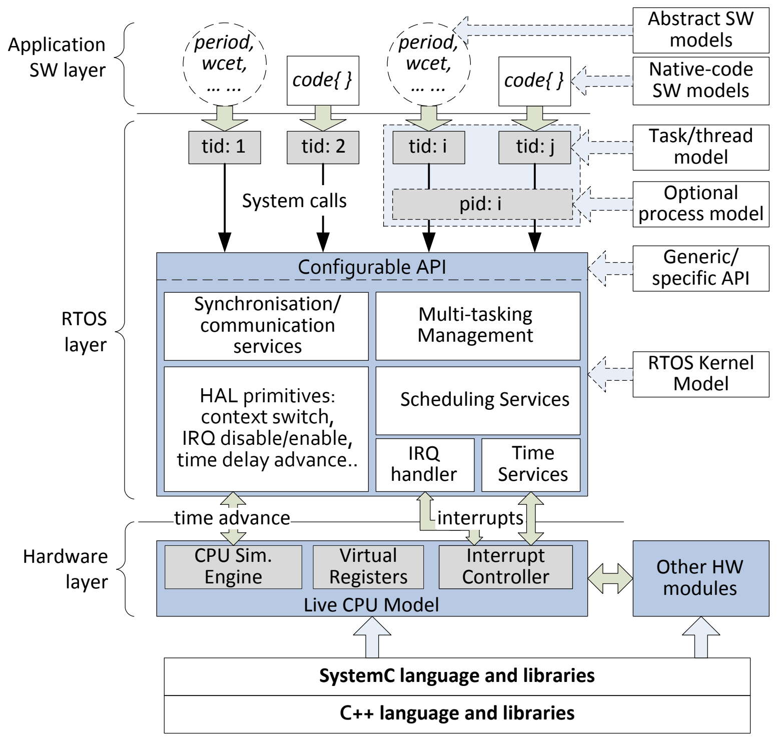


Figure 4-7. Structure of the software PE model

4.5.1.1 Software Layers

In the whole software PE simulation model, two software layers (i.e., the application software layer and the RTOS layer) constitute the software part, namely a software stack model.

From the top down, the application software is divided into several execution entities (i.e., tasks) and each entity can be modelled as an *abstract software model* or a *native-code software model*. The former focuses on quickly simulating timing properties of applications. The model is characterised by a set of timing parameter as introduced in Section 4.4.1. Whereas the latter aims to simulate functions of applications by using functional code close to actual implementation at the expense of simulation speed reduction.

No matter which way application software is modelled, each application task model is projected onto a RTOS-level task/thread model, which is runnable in the SystemC simulation environment. The task/thread abstraction is handled as the software scheduling entity in RTOS kernel multi-tasking management, which is true for most RTOSs in the research context. Process models can be optionally created in modelling, but they do not play effective roles to compete for resources.

System calls are implemented mainly by function calls to APIs of the RTOS model, i.e., application tasks call member methods of the RTOS kernel module (an object of a C++ class). This function-call feature is similar to the real situation in a RTOS. This modelling method also has a good "side effect" of protecting RTOS kernel data structures because data access is protected by the C++ object-oriented program language. It represents the native distinction between the user space and the kernel space in the real-time embedded software stack. Based on the essential services provided in the RTOS kernel model, their APIs can be partially configured to mimic different RTOS standards and products. For the reason that APIs of various proprietary RTOSs may be quite different from each other in terms of functionality and function parameters, exact compatibility to a specific RTOS is not the goal of research in this thesis. Rather, generality, ease of use and reasonable accuracy is desired for system-level behavioural software simulation.

As introduced in Section 4.4.2, the kernel structure is the first-class concept for designing and modelling a RTOS. The RTOS kernel model encapsulates all its data and functions in a single class and this model structure is akin to the monolithic approach. However, since modelling extended OS components such as device drivers, file systems, and network stacks is outside the scope of this thesis, the presented RTOS model contains fundamental services (e.g., multi-tasking management, scheduling, inter-task synchronisation, and interrupt handling) that are commonly provided by a microkernel. From this lightweight (i.e., limited and essential) service modelling perspective, the RTOS model is also similar to a "micro" kernel.

Given the high abstraction level of software and hardware simulation models, modelling real memory space management and the processor MMU is not addressed. Hence, potential advantages and disadvantages of monolithic and microkernel structures are not revealed and evaluated in behavioural RTOS modelling. This feature brings benefits in terms of modelling simplicity and fast simulation speed, but is also a defect in terms of functionality and remains for future research.

In the RTOS kernel, some HAL primitives directly interact with the Live CPU Model for advancing software simulation time and setting system states. System clock interrupts and other external hardware interrupts can invoke associated interrupt handlers in the RTOS kernel module.

4.5.1.2 Hardware Layer

In Figure 4-7, the hardware layer is represented by the Live CPU Model that was introduced in Section 3.3. It is the hardware part of the software PE model and the basis of the upper software layers. In general, its main purpose is to support and assist behavioural software simulation from two perspectives: supporting pre-emptible time advance and modelling hardware I/O. In some conventional system-level real-time software and RTOS simulation (e.g., [113] [114] [126]), the application software model and the RTOS model construct a PE, and in fact, there is not any hardware model in the PE. Unlike them, the Live CPU Model executes software delay annotations in a way that is conceptually comparable to the way a real CPU executes instructions. The Live CPU Model also monitors real-time clock and external interrupts and can start, stop, and resume a software delay time advance without any undesired latency.

If some other SystemC-based hardware modules need to be combined with the software PE model for further HW/SW co-simulation, they can be connected to the Live CPU Model by either SystemC interface method call channels or specific TLM interfaces (see Chapter 5).

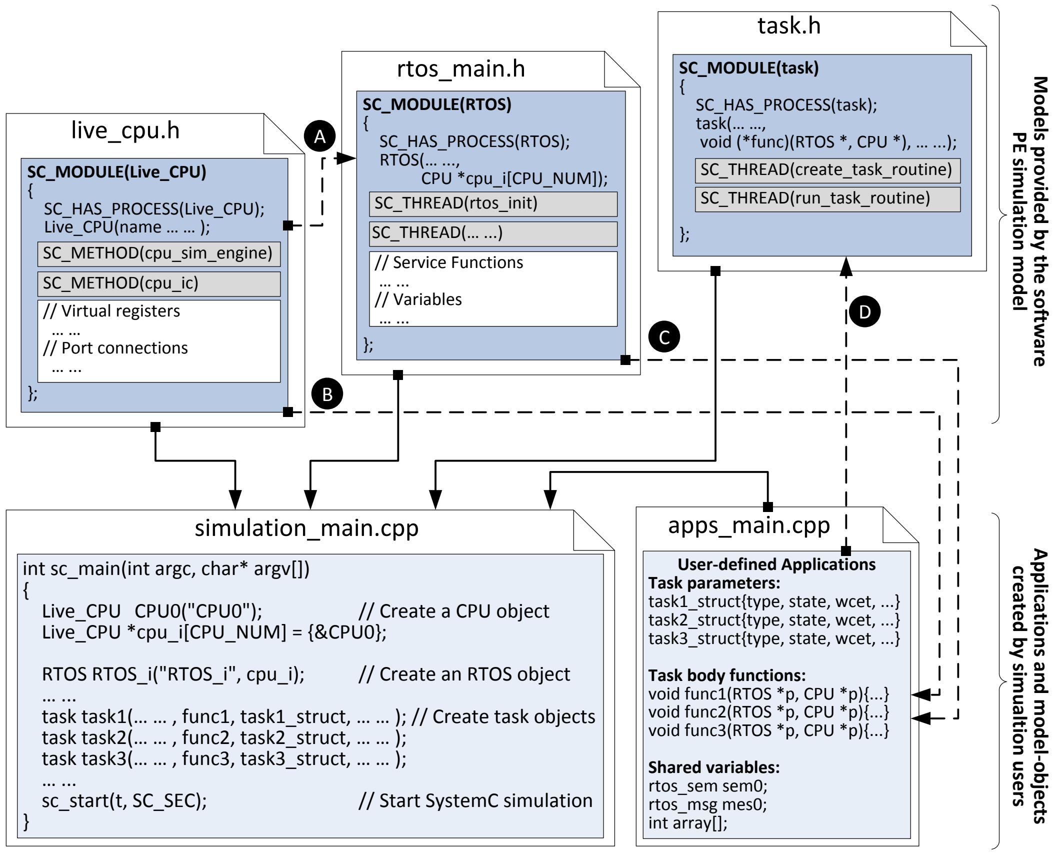
4.5.1.3 Structure of SystemC Models

As indicated by the lowest layer of Figure 4-7, all models in the SystemC-based real-time software simulation framework are implemented in SystemC and C++. Figure 4-8 illustrates how various components of the software PE simulation model are implemented and relate to each other in SystemC. Depending on their functionality and creator, they can be divided into two classes:

- *Software PE related models* (See upper half of Figure 4-8): There are inherent hardware and system software components in the software PE model, which provide standard services for simulating user applications. Simulation users can directly use these default services in their software simulation models. Additionally, users can modify them or add new models (services) depending on the necessity. In implementation, each model in this category is implemented as a SystemC `SC_MODULE` in a separate header file. There are three types of `SC_MODULEs`: the `Live_CPU` module, the `RTOS` module, and the `task` module, which represent the Live CPU Model, the RTOS kernel model, and RTOS task models.
- *User application models and simulation related programs* (See lower half of Figure 4-8): This part contains models and programs that are defined by simulation users in order to simulate specific software applications in the software PE environment. Referring to `apps_main.cpp` in Figure 4-8, an application task model is given as a segment of C/C++ code, which includes an application task body function, global variables to be shared by multiple tasks, and possible timing parameters of this application task. Referring to `simulation_main.cpp` in Figure 4-8, objects of various hardware and software models are created and connected with each other in

the `sc_main()` function so as to constitute a whole SystemC simulation program. Specifically, in the research context of “a uniprocessor system”, there should be a single `Live_CPU` object, a single `RTOS` object, several RTOS `task` objects, and several user application functions.

Models and objects are organised and invoked in a straightforward hierarchy, according to their logical relationship - namely, the RTOS runs on top of the CPU and software tasks run on top of both the RTOS and the CPU. Referring to Figure 4-8, the dotted lines, and pseudo code in `simulation_main.cpp` and `apps_main.cpp` demonstrate their interrelationship. An object of the `Live_CPU` model is created and then used as an argument to create a `RTOS` object (See the dotted line (A)). By this means, various RTOS functions can make use of CPU resources. Similarly, as illustrated by dotted lines (B) and (C), both the `Live_CPU` object and the `RTOS` object are passed to application task body



task.h
rtos_main.h
live_cpu.h
simulation_main.cpp
apps_main.cpp
A
B
C
D
Models provided by the software PE simulation model
Applications and model-objects created by simualtion users

```
SC_MODULE(Live_CPU)
{
  SC_HAS_PROCESS(Live_CPU);
  Live_CPU(name … … );
  SC_METHOD(cpu_sim_engine)
  SC_METHOD(cpu_ic)
  // Virtual registers
   … …
  // Port connections
   … …
};
```

```
SC_MODULE(RTOS)
{
  SC_HAS_PROCESS(RTOS);
  RTOS(… …,
         CPU *cpu_i[CPU_NUM]);
  SC_THREAD(rtos_init)
  SC_THREAD(… …)
  // Service Functions
   … …
  // Variables
   … …
};
```

```
SC_MODULE(task)
{
  SC_HAS_PROCESS(task);
  task(… …,
   void (*func)(RTOS *, CPU *), … …);
  SC_THREAD(create_task_routine)
  SC_THREAD(run_task_routine)
};
```

```
int sc_main(int argc, char* argv[])
{
   Live_CPU  CPU0("CPU0");                  // Create a CPU object
   Live_CPU *cpu_i[CPU_NUM] = {&CPU0};

   RTOS RTOS_i("RTOS_i", cpu_i);          // Create an RTOS object
   … …
   task task1(… … , func1, task1_struct, … … ); // Create task objects
   task task2(… … , func2, task2_struct, … … );
   task task3(… … , func3, task3_struct, … … );
   … …
   sc_start(t, SC_SEC);                     // Start SystemC simulation
}
```

```
User-defined Applications
Task parameters:
task1_struct{type, state, wcet, ...}
task2_struct{type, state, wcet, ...}
task3_struct{type, state, wcet, ...}

Task body functions:
void func1(RTOS *p, CPU *p){...}
void func2(RTOS *p, CPU *p){...}
void func3(RTOS *p, CPU *p){...}

Shared variables:
rtos_sem sem0;
rtos_msg mes0;
int array[];
```

Figure 4-8. SystemC implementation of the software PE simulation model

functions as arguments. The meaning of this is twofold: firstly, a task body function executes on the CPU; secondly, it can call services provided by the RTOS. Then, an object of the RTOS `task` module is created, because it is the scheduling entity of both the RTOS kernel and the underlying SystemC simulation kernel. Referring to the dotted line (D), a user-application task body is wrapped to a RTOS `task` object with a one-to-one correlation, in order to be involved in a RTOS-based software simulation.

The above-introduced modular structure makes the software PE simulation model simple, reusable, and extensible. The simple structure of the whole simulation model is representative yet abstract enough to represent a real-time embedded software system. The interdependency between different sub models and sub modules are reduced as low as possible through carefully and explicitly defined interfaces. The inherent software scheduling (i.e., the RTOS) and executing (i.e., the Live CPU Model) models are distinguished and independent from user-developed application task models; hence the reusability of the RTOS model and the Live CPU model is preserved to some extent. Referring to the code line `*cpu_i[CPU_NUM]` in the constructor of the `RTOS` module in Figure 4-8, the `RTOS` module can accept several CPU objects, which means that the RTOS model reserves the potential to be extended as a multi-processor RTOS model in future development.

4.5.2 Application Software Modelling

According to the mixed timing software modelling approach in Section 3.2, a simulation user can model application software as both abstract task models and native-code task models. The two types of task models can co-exist in simulation so as to increase simulation flexibility.

4.5.2.1 Abstract Task Model

The abstract task model applies to situations where application software code has not been fully finished for modelling, or where the simulation user does not have much interest in functional simulations. Such task models are primarily intended simply to simulate the timing behaviour of real-time software with the assistance of a RTOS model.

Table 4-5 shows an abstract periodic task model. Referring to lines 1-10, some user-defined task identity information (e.g., task type, initial state, etc) and timing properties of the model are stored in a data structure variable that will be used to create a RTOS TCB later during the task creation process. Note that Table 4-5 does not show all necessary user-defined items. They are shown as bold in Table 4-8. The timing behaviour of a task is characterised by a set of parameters, e.g., BCET, WCET, relative deadline, period, and offset. In simulation, BCET and WCET are used to generate an intermediate random value that serves as the execution time of a specific task instance. Otherwise, the WCET is used as the execution time of every task instance, because the worst-case behaviour is usually more concerned with real-time system simulation. The relative deadline is converted into an absolute deadline in execution, in order to facilitate deadline-driven scheduling or is used to monitor a task's status in terms of whether it misses a deadline. The period explicitly specifies how often a task should be regularly activated. The offset indicates means the initial waiting time of the first task job in execution. In simulation, the RTOS kernel model can track the period, the offset, and instance numbers of an abstract periodic task in order to support periodic task

```
#001  // Defining parameters of a task in a struct
#002  {
#003   THREAD_TYPE  task_type;
#004   THREAD_STATE task_state;
#005   unsigned __int64 bcet;
#006   unsigned __int64 wcet;
#007   unsigned __int64 relative_deadline;
#008   unsigned __int64 period;
#009   unsigned __int64 offset;
#010   … … }
#011  // Task body function
#012  void task(RTOS *rtos_i_ptr, CPU *cpu_i)
#013  {
#014    while(1){
#015     t = random_function();  // Generate random execution time of a job
#016     DELAY(t);               // Pass t to Live CPU
#017     wait(event);            // Wait for time advance
#018     (*rtos_i_ptr).task_wait_cycle(); //Yield CPU, wait for next cycle
#019     wait(event);            // Wait for next execution
#020    }
#021  }
```

Table 4-5. The abstract periodic task model

execution (see Section 4.5.4). Note that not all of the above four parameters are required for an abstract task model. For example, an aperiodic task that services an interrupt does not have a period parameter.

Lines 11-21 of Table 4-5 show the body function of an abstract periodic task model. A RTOS model object and a Live CPU Model object are passed to the function body, in order to let the task use RTOS functions and CPU resources, although the task model usually contains little or none functional code. Optionally, the simulation user can appoint a probabilistic function in order to generate a random execution time for each task job. This method is also used in similar-purpose research [8] [72]. The time advance method (i.e., lines 16 and 17) was introduced in Chapter 3 and note that the `event` object at line 17 is exclusive to this task model. Recalling it again, this time advance process is interruptible and the task model is pre-emptible.

At lines 18 and 19 of Table 4-5, a RTOS function `task_wait_cycle()` is called to notify the RTOS kernel that this periodic task reaches its end and waits for next execution cycle (period). Accordingly, the RTOS kernel will take some actions to process this request, which will be introduced later in Section 4.5.4.3.

In case of an abstract aperiodic task model, it shall call another RTOS function `task_wait_suspend()`, which will suspend the task indefinitely until the task is invoked by an interrupt again.

Note that, if a task is not independent, namely it cooperates or competes for some shared resources with other tasks, then specific RTOS synchronisation or communication services must be called in the body function. In this case, the native-code task model is more applicable.

4.5.2.2 Native-Code Task Model

If applications come with functional code and corresponding fine-grained delay annotations, then a native-code task model can be built. Table 4-6 shows its parameter definition (lines 1-7) and body function (lines 9-21). At lines 1-7, the data structure variable is still necessary to define identity information of a task, but it no longer contains timing properties about coarse grain computation time and an explicitly defined period.

```
#001 // Defining parameters of a task in a struct
#002 {
#003  THREAD_TYPE  task_type;
#004  THREAD_STATE task_state;
#005  unsigned __int64 relative_deadline;
#006  unsigned __int64 delay_time;
#007  … … }
#008
#009 // Task body function
#010 void task(RTOS *rtos_i_ptr, CPU *cpu_i)
#011 {
#012   while(1){
#013    func_block1();          // The 1st block does some functions
#014    DELAY(B1_DELAY);        // Pass B1_DELAY to Live CPU
#015    wait(event);            // Wait for time advance
#016    … …                     // The 2nd block does some functions
#017    … …                     // The 3rd block dose some functions
#018    rtos_i_ptr->sleep(500); // Call RTOS API: sleep()
#019    wait(event);            // Wait for next execution
#020   }
#021 }
```

Table 4-6. The native-code task model

Referring to Table 4-6, this *wait-for-event* time advance method is briefly repeated here. Timing delay annotation (line 14) interleaves with a code block (line 13) in the function body. The `DELAY()` function at line 14 injects a delay value `B1_DELAY` into the Live CPU Model. The granularity of a delay annotation depends on the choice of the simulation user, for example, the basic block level or statement segment level. Unlike [72] [43], delay annotation statements in the native-code task model do not define fixed pre-emption points for HW/SW synchronisation. Their main purpose is to notify the Live CPU Model how long computation time a code block needs, and then let the task wait for an event that will be released when the delay time is consumed. The `event` object at line 15 is exclusively used in this task model. Interruption and pre-emption can happen at any necessary (i.e., there is an interrupt event) and possible (i.e., system-wide interrupts are enabled) time points during a delay duration.

Compared to two task examples provided by ThreadX RTOS [152] and µC/OS-II RTOS [149] in Table 4-7, the body function of a native-code task model does not differ too much from the entry function of a real RTOS task in terms of the code structure. That is, a loop contains the C/C++ main functional code and a RTOS system function is called at the end of the loop body in order to suspend the task. The periodic execution of a task can be achieved by calling the RTOS time delay function, e.g., line 18 in Table 4-6 and lines 8 and 19 in Table 4-7. This co-

```
// An example body function of a ThreadX thread
void data_capture_process(ULONG thread_input)
{
  while(1){
   temp_memory[frame_index][0] = tx_time_get();
   temp_memory[frame_index][1] = 0x1234;
   frame_index = (frame_index +1) % MAX_TEMP_MEMORY;
   tx_thread_sleep(1);
  }
}
// An example body function of a µc/OS-II task
void TaskClk(void *pdata)
{
   char s[40];
   data = data;
   for(;;){
         PC_GetDateTime(s);
         PC_DispStr(60,23,s,DISP_FGND_BLUE+DISP_BGND_CYAN);
         OSTimeDly(OS_TICKS_PER_SEC);
   }
}
```

Table 4-7. Two task examples in ThreadX RTOS and µC/OS-II RTOS

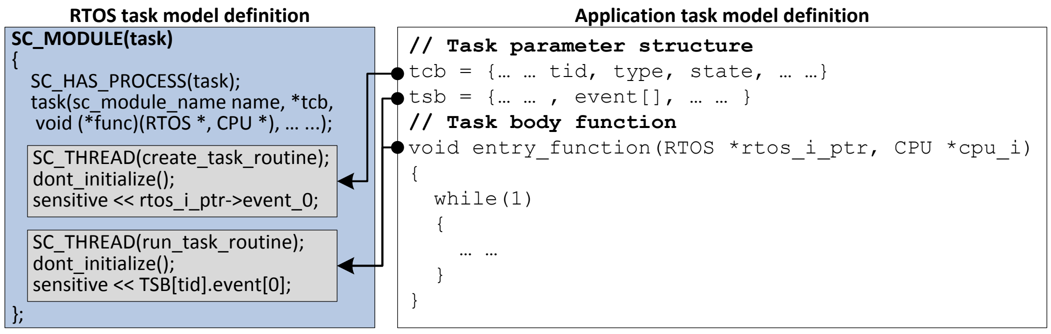
herence facilitates using the simulation model with conventional RTOS applications. The difference mainly resides within two points in the native-code model: firstly, time annotations and synchronisation points are inserted for time advances; secondly, a RTOS service should be invoked through a pointer to a RTOS model object.

4.5.3 RTOS Task/Thread and Process Modelling

4.5.3.1 Task/Thread Model

Given that application software has been divided into task body functions and that their timing parameters are provided, it is necessary to create RTOS-level task models in order to let the RTOS kernel organise these execution entities. As mentioned before, the task/thread concept is chosen as a RTOS scheduling unit. Based on the survey in Section 4.4.3.2, such a multi-tasking model is common and powerful enough to organise real-time embedded applications in various RTOS products. In the modelling approach, a RTOS task/thread model is implemented as an object of the SystemC `task` module. Figure 4-9 shows the definition of a RTOS task and its relationship to an application task model.

Note that there is a clear separation between a user task model and a RTOS task model, in terms of both the modelling concept and the SystemC implementa-



```
SC_MODULE(task)
{
  SC_HAS_PROCESS(task);
  task(sc_module_name name, *tcb,
   void (*func)(RTOS *, CPU *), … …);
    SC_THREAD(create_task_routine);
    dont_initialize();
    sensitive << rtos_i_ptr->event_0;

    SC_THREAD(run_task_routine);
    dont_initialize();
    sensitive << TSB[tid].event[0];
};
```

```
// Task parameter structure
tcb = {… … tid, type, state, … …}
tsb = {… … , event[], … … }
// Task body function
void entry_function(RTOS *rtos_i_ptr, CPU *cpu_i)
{
  while(1)
  {
    … …
  }
}
```

Figure 4-9. Defining a RTOS task model

tion in the modelling approach. The creation of a unified RTOS task object only utilises task information and the function body defined by a user in one application task model (see Section 4.5.2), which means it is not necessary to define a variety of RTOS task modules in order to accommodate different applications.

In the modelling approach, the implementation of a RTOS task involves two data structures and two operations, which are referred to as the Task Control Block, the Task Service Block (TSB), initialising a TCB, and wrapping a function body, respectively.

Definition of the Task Control Block

Every RTOS needs a TCB structure for each task in order to store task-specific properties and manage the task through the TCB during run-time. Table 4-8 describes the TCB fields within the RTOS model. Among them, the bold fields can be provided in user-defined application models (see Section 4.5.2). All TCB fields can generally be classified within three categories:

- *The task ID and status section:* Fields in this section are related to statically-assigned identifiers and dynamically-changed states;
- *The task timing information section:* This section stores some timing parameters of a real-time task as well as time advance information;
- *The pointers section:* Some pointers are provided in order to correlate message and synchronisation event control blocks to a task, and they are also used to maintain task scheduling queues.

Although the contents of the TCB are internal affairs in the design of a RTOS simulation model, a certain degree of similarity between the model's TCB and a real RTOS's TCB is still helpful in allowing for simulation users to inspect and

	Field	Description
ID & status section	**rtos_tcb_cpu_id**	The CPU which a task belongs to
	rtos_tcb_pid	Process identifier
	rtos_tcb_tid	Task identifier
	rtos_tcb_thread_type	Task type
	rtos_tcb_thread_state	Task state
	rtos_tcb_wait_flag	Sub-state of WAITING state
	rtos_tcb_base_prio	Initial (base) priority
	rtos_tcb_cur_prio	Current (effective) priority
Time info. section	**rtos_tcb_relative_deadline**	Relative deadline of a task
	rtos_tcb_period	Period of a task
	rtos_tcb_thread_bcet	BCET of a task
	rtos_tcb_thread_wcet	WCET of a task
	rtos_tcb_slice	Current time slice
	rtos_tcb_new_slice	New time-slice
	context[CONTEXT_LENGTH]	Timing context for time advance

	Field	Description
Pointers section	*rtos_tcb_ecb_ptr	Pointer to an event control block
	*rtos_tcb_msg	Pointer to a message
	*rtos_tcb_back	Pointer to the previous TCB in a sche. queue
	*rtos_tcb_next	Pointer to the next TCB in a scheduling queue

Field	Description
block_exec_time	delay slice of a code block
thread_exec_time	total delay of the task job
thread_abs_dln	absolute deadline of the task job
thread_used_time	consumed delay time
thread_cur_sta_time	start time of current delay slice
thread_sleep_length	sleeping time of the task job

Table 4-8. Task (Thread) Control Block

understand the state of tasks in simulation. Comparing the RTOS model's TCB to those of μC/OS-II and ThreadX, the task ID/status section and the pointers section of them are mostly alike. The significant differences include:

1) The RTOS model's TCB omits memory stack setting fields, which however do exist in the TCB of a real RTOS. This is because the RTOS model does not aim to model software execution memory space.
2) Regarding the timing information section, the RTOS model's TCB has some real-time task-related timing fields; whereas, a real RTOS's TCB does not normally contain them. The proposed TCB is based on the consideration that these real-time parameters are necessary for abstract task modelling and real-time system simulation.
3) The `context[CONTEXT_LENGTH]` field is essential for software time advance in our timed software simulation method. Its six sub-fields are namely the "*processor-related context*" of a task model. Their value needs to be written to and read from the virtual `CPU_REGs` of the Live CPU Model in each context switch. The context-switch process will be introduced in Section 4.5.8.2. Note that a real RTOS TCB does not need these fields, but contains the real *program counter*, *stack point*, and other *data registers* as substitutes.

Definition of the Task Service Block

The TSB is a user-defined data structure associated with each RTOS task model. Its main purpose is to store simulation-related configuration parameters and statistical information of a task that are not normally contained in a real TCB, in order to simplify the TCB structure. The most useful field of a TSB is a `sc_event` object array. Each `sc_event` object is exclusive to a task function body (as shown in Table 4-5 and Table 4-6). The Live CPU Model controls time advances of each task model via the *wait-for-event* method and these `sc_event` objects. The sequence number of task jobs and initial offset are another two notable TSB fields. They record how many instances a task has executed and the task's initial offset, which are used to calculate activation time of an abstract periodic task.

Initialising a TCB

In model implementation, a vacant TCB array (`rtos_tcb_array[]`) was defined before the TCB initialisation process. Referring to Figure 4-9, the `SC_THREAD(create_task_routine)` takes charge of initialising a TCB item in the array. A sole task ID offers a connection between the existing TCB item and this initialisation process. This `create_task_routine` uses task properties provided in the user-defined data structure (see Table 4-5 and Table 4-6) and initialises all necessary fields of a corresponding task's TCB. Note that both the `offset` in Table 4-5 and the `delay_time` in Table 4-6 correspond to the `thread_sleep_length` subfield in Table 4-8, which represent a task possibly being delayed for some time after its creation, i.e., with an offset. The `thread_abs_dln` subfield in Table 4-8 refers to the absolute deadline of a task. If necessary, it can be computed as the sum of the task creation time and its relative deadline.

Figure 4-10 illustrates the timeline of the TCB initialisation process in a real RTOS executing situation. Normally, the task creation happens just after the RTOS kernel has been initialised. The RTOS kernel initialisation necessarily consumes some simulated time and so consequently, there is a time offset from the zero time of the simulated clock to the initialisation of the first TCB. Furthermore, every task creation activity sequentially progresses the target clock. In our approach, the practical timing behaviour of this execution order is modelled by two techniques:

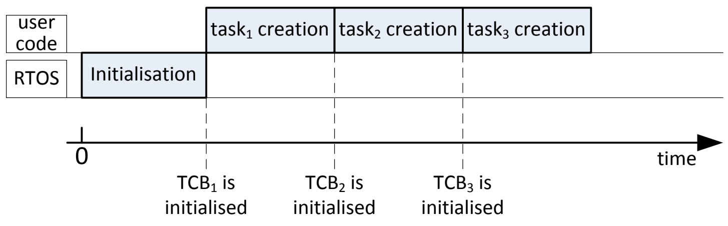


Figure 4-10. Initialising TCBs

- This `SC_THREAD` is activated to run by a `sc_event` that is released by the RTOS kernel initialization function (see Figure 4-9). This guarantees that every TCB is initialised after the initialisation of the RTOS kernel.
- All TCBs are initialised in the same SystemC delta cycle, which is not same as the real execution. In order to serialise them with delay intervals along the timeline, a global counter function and a *wait-for-delay* function are inserted at the beginning of the `SC_THREAD` function body. The counter function makes a statistic on how many task objects are in the system and how many of them need to be created with time advance at this early RTOS executing time point. Note that this RTOS modelling research requires that all task module objects are created at the SystemC elaboration stage, but a RTOS task can be "dynamically" created at simulation runtime (see Section 4.5.4.3 for further discussion). Hence, not all task creation timing overheads may need to be released in order to progress the simulated clock now. Depending on the calculation result of the counter function, the *wait-for-delay*[7] function temporarily suspends appropriate `SC_THREADs` for a time delay before it executes.

Wrapping a function body

As simple timing parameters of a task or a pure C/C++ based task entry function are not directly supported by the SystemC simulation kernel for timed execution, it is necessary to attach the user-defined application task model to a SystemC executable process in order to run it in the SystemC environment. Regarding the

[7] We assume that all tasks are created just after the RTOS kernel initialisation but before the start of the OS multi-tasking service. Interrupts are disabled at the time. Hence, the use of *wait-for-delay* is allowed here.

two kinds of SystemC processes, `SC_METHOD` and `SC_THREAD`, the latter is selected as the wrapper. Because it can be suspended and resumed in execution, this behaviour is essential for modelling task pre-emption and simulation time advance. In Figure 4-9, the `SC_THREAD(run_task_routine)` behaves as such a wrapper to encapsulate a task entry function. It is sensitive to a `sc_event` stored in its TSB.

4.5.3.2 Optional Process Model

In some complex RTOSs (e.g., QNX and other RT-POSIX compliant ones), applications are managed in both the *process* model and the *task/thread* model, where a process contains at least one thread and provides a memory space for all its containing threads. This two-level structure brings some advantages such as better modularity because of distinct process containers, less interdependency since each process has its particular definition, and more reliability because threads are protected in different memory spaces [184].

It is important to reiterate that, in this thesis the multi-tasking model is based on a single-level task/thread abstraction model and without modelling memory management functions. Modelling process is out of the research scope. However, for a consideration of preserving the extendibility of the software PE simulation model, a simple Process Control Block model is defined as well (See Table 4-9). In the modelling approach, a process can be created by modelling the RT-POSIX `spawn()` function. This creates a child process by directly specifying an executable to load and its implementation is very similar to the previously mentioned task/thread creation method. A process and its inclusive threads are related to each other according to the `pcb_child_tcb_array[]` field in the PCB and the

Field	Description
pcb_pid	Process identifier
pcb_uid	User identifier
pcb_gid	Group identifier
pcb_child_tcb_arrary[NUM]	Child-task/thread *tids'* array
pcb_process_base_priority	Initial (base) priority
pcb_process_current_priority	Current (effective) priority
start_address	Starting address of the process's memory space
end_address	Ending address of the process's memory space
*pcb_back	Pointer to the previous PCB in a sche. queue
*pcb_next	Pointer to the next PCB in a scheduling queue

Table 4-9. Process Control Block

`pid` field in the TCB.

4.5.4 Multi-Tasking Management Modelling

4.5.4.1 Task State Machine

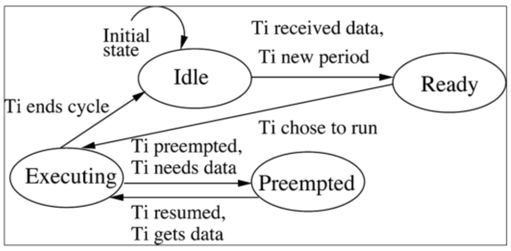
The task state machine is the basis of both multi-tasking management and scheduling services in our RTOS kernel model. The task state machines implemented in some real RTOS products were surveyed in Section 4.4.3.2.

Note that the task state machines implemented in some existing RTOS modelling research used some terms and structures that are confusing or not common in practical RTOSs. For example, as shown in Figure 4-11 (A), [8] and [11] implement a similar task state machine including four states[8]: Idle, Ready, Executing, and Pre-empted. Two points of this model are worth discussing:

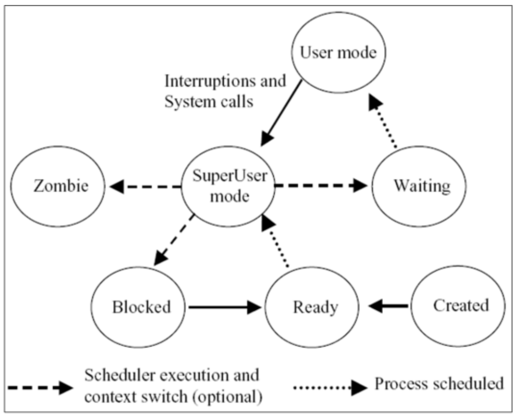
1) In a normal RTOS kernel, if a task is pre-empted, then it usually enters the "READY" state. However, in Figure 4-11 (A), a special Pre-empted state is defined as different from its Ready state, which may be unnecessary.
2) In a normal RTOS kernel, if a task is blocked due to waiting on a synchronisation method (namely a resource) or explicit self-suspension, then it usually goes to the "WAITING" state. In Figure 4-11 (A), a task enters the Pre-empted state when it is waiting on data, whereas it enters the Idle state for self-suspension. The two different states cannot simply be interpreted as synonyms of the classical "WAITING" state, because of the confusing meaning of the Pre-empted state. In the model, it reflects the function of both the "READY" state and the "WAITING" state, which are diverse in relation to classical RTOS concepts.

Figure 4-11 (B) shows a seven-state RTOS task state model presented in [12]. Just as its authors indicated, it is similar to the task state machine commonly used in UNIX systems. Hence, although it is complete and expressive enough, it may not be applicable for small-size compact RTOSs. It is noticed that:

[8] Hereafter, the first letters of the states in the referenced RTOS task state models are capitalised. Distinctively, the states in the surveyed RTOS products in Section 4.4.3.2 and in our RTOS model are spelled using capitals letters.



(A) Task state machine defined by Hessel et al. and Madsen et al. (reprint)


User mode
Interruptions and System calls
Zombie
SuperUser mode
Waiting
Blocked
Ready
Created
Scheduler execution and context switch (optional)
Process scheduled

(B) Task state machine defined by Posadas et al. (reprint)

Figure 4-11. Task state machines: reprint A [8] [11], B [12]

- The task state machine divides the classical “RUNNING” state into the User mode and Super User mode, which is not common in RTOSs.
- Based on the above feature, the task state machine has divided the classical “READY” states into two states, i.e., Ready and Waiting. This makes the RTOS state machine redundant.

Research in [114] implements the three-state (i.e., READY, RUNNING, WAITING) task state model depicted in Figure 4-6. This canonical structure is also the basis for research in this thesis. Furthermore, based on the survey in Table 4-1, a *four-state extensible* task state machine is proposed to contain more states in order to be more representative and correspond to specific kernel services of some RTOSs. Figure 4-12 shows its structure and task state transitions. The main modelling idea behind this is as follows:

1) Add a TERMINATED state, because it appears to be useful in many RTOS products. For example, it is referred to as, or similar to, the INACTIVE state in RTX [1], the DORMANT state in μITRON [151] and μC/OS-II [149], the COMPLETED and TERMINATED state in ThreadX [152]. The TERMINATED state is the *exit* of a task in the system, that is, where the task has already finished and cannot execute again.
2) Subdivide the WAITING state into seven sub-states, i.e., WAITING_SUS, WAITING_SEM, WAITING_MUT, WAITING_QUE, WAITING_EVT, WAITING_DLY, and WAITING_CYC. As shown in Figure 4-12, each

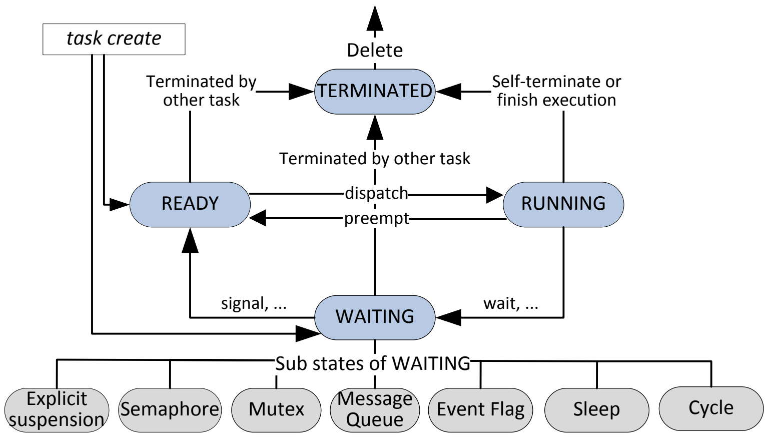


Figure 4-12. The proposed four-state extensible task state machine

sub WAITING state corresponds to a specific blocking condition. Note that the WAITING_CYC state has been specially designed so that idle periodic tasks can wait for their next execution cycle. This sub-state modelling concept is similar to the task state machine of μITRON [151]. The proposed task state machine is said to be extensible because the important and variable WAITING state can be specified into different sub-states and modelled by easily setting the `rtos_tcb_wait_flag` field (listed in Table 4-8) in the task TCB. When a simulation user wants to model a new blocking situation, it is not necessary to insert a new state in the task state machine and create an additional *waiting queue*. Just adding a sub WAITING state and redefining the flag is enough. By shrinking and extending sub WAITING states, the RTOS model can mimic behaviours of different RTOSs.

4.5.4.2 Task Queues

The RTOS normally manages tasks by organising their TCBs in several queues [25] [26]. Usually, there are two pointers in a TCB by which multiple TCBs link to each other (See `*rtos_tcb_back` and `*rtos_tcb_next` in Table 4-8). As mentioned in Section 4.4.3.2, a *ready queue* and a *waiting queue* are necessary for maintaining tasks at the READY state and WAITING state, respectively. In addition, the TERMINATED state needs a separate queue. Because there is only one RUNNING task in the uniprocessor system at any time, the RUNNING state does not need a queue and a `RTOS_RUNNING_TCB` pointer indicates the TCB of the current RUNNING task. If this RTOS model is possibly extended to a multi-processor platform in future research, then the RUNNING state can have multiple `RTOS_RUNNING_TCB` pointers.

The exact implementation method of a queue varies in different RTOSs. In μC/OS-II, the ready queue is effectively implemented as a table with two variables: an integer and an integer array [149]. Their bits represent states of tasks and task IDs, respectively. The RTOS kernel looks up the table to find the highest priority READY task and removes a task from the ready list by clearing a bit of the integer variable. Considerations of such an implementation are to save limited memory space, improve lookup speed, and keep the lookup execution time constant. However, it is not very user-friendly or well visualised. In QNX, the ready queue is implemented as 256 separated queues – each priority level having a linked list [162]. This structure is quite organised with an inserting time complexity of $O(1)$, but its implementation complexity is relatively high for modelling.

In this thesis, in order to keep a balance between implementation complexity in modelling and operating time complexity in simulation, a basic task queue is implemented as a single priority-descending[9] doubly linked list (See Figure 4-13). All tasks at the same state are inserted into the queue according to their priorities, with a time complexity of $O(n)$. Same-priority tasks are adjacent. This is similar to the ready thread list of ThreadX [152]. Basic primitives are provided to manipulate and debug a queue, for example, inserting a TCB, deleting a TCB, returning the head of the queue, reporting the number of TCBs in the queue, and printing one or all TCBs in the queue. The *ready queue, waiting queue,* and *terminated queue* all inherit this base task queue class but may derive different functions from it. For example, a simulation user can implement a specified policy regarding how same-priority TCBs are ordered in a queue, e.g., FIFO or LIFO, by overloading the inserting primitive. These derived task queues and their member functions are

[9] In order to support EDF scheduling, in modelling, a task queue can also be ascendingly ordered by tasks' absolute deadlines. Corresponding primitives have been implemented in the model.

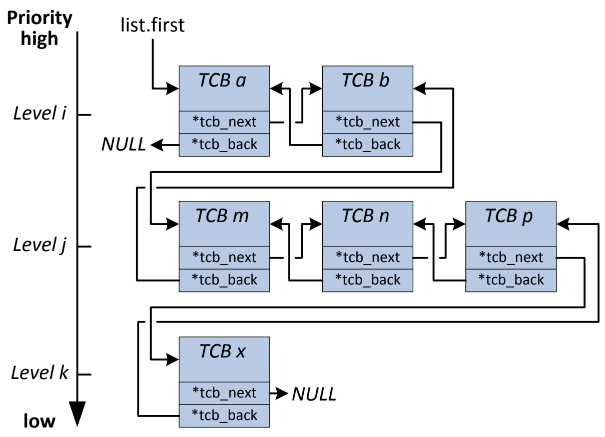


Figure 4-13. A priority-descending doubly linked task queue

involved in various services of RTOS task management, scheduling, synchronisation and interrupt handling in the proposed RTOS model.

4.5.4.3 Task Services

A RTOS usually supplies a variety of multi-tasking services for application tasks concerning their *state transitions* around the state machine and *TCB configurations*[10]. The basic modelling consideration is to comply with common services available in small-size RTOS products. ThreadX, µC/OS-II, and RTX are still used as referenced samples. Table 4-10 enumerates task management services implemented in the proposed RTOS model and corresponding functions found in three RTOS products. Comparing these task services, the RTOS model can be seen to have approximately covered 12 out of 13 services of ThreadX, 8 out of 11 services of µC/OS-II, and 10 out of 13 services of RTX. In general, the RTOS model is capable of supplying main and typical task services of a RTOS. However, memory-related services (e.g., stack check) are not included in modelling, and some proprietary or small variant services are not implemented neither. In different RTOSs, the arguments return values, and detailed internal functions of a similar-purpose service are necessarily different to some extent. Hence, in order to

[10] These two sets of services mean a narrow definition of “multi-tasking services”. Other specific task services such as scheduling and synchronisation will be introduced in the following related sections.

	Services implemented in RTOS model	ThreadX	µc/OS-II	RTX
Task state transition related services	Create a task	tx_thread_create	OSTaskCreate	os_tsk_create
	Terminate a task (to TERMINATED state)	tx_thread_terminate	OSTaskDel	os_tsk_delete/self
	Delete a task from the system	tx_thread_delete		
	Transfer from RUNNING to READY	tx_thread_relinquish		os_tsk_pass
	Transfer from WAITING_SUS to READY	tx_thread_resume	OSTaskResume	os_evt_set
	Transfer from WAITING_DLY to READY	tx_thread_wait_abort	OSTimeDlyResume	
	Transfer from RUNNING to WAITING_DLY	tx_thread_sleep	OSTimeDly	os_dly_wait
	Transfer from RUNNING to WAITING_SUS	tx_thread_suspend	OSTaskSuspend	
	Transfer from RUNNING to WAITING_CYC			os_itv_wait
TCB related services	Return a pointer to RUNNING task's TCB	tx_thread_identify		os_tsk_self
	Output information of a TCB	tx_thread_info_get	OSTaskQuery	
	Change priority of a task	tx_thread_priority_change	OSTaskChangePrio	os_tsk_prio os_tsk_prio_self
	Change time-slice of a task	tx_thread_time_slice_change		

Table 4-10. Task services in the RTOS model and some RTOSs

model a specific RTOS product, services of the RTOS model may also need to be adapted. The careful definitions of the task state machine and the TCB structure make a sound base from which to revise existing services or add new services into the RTOS model without many obstacles.

Supporting periodic execution of abstract tasks is a notable task service of the RTOS model. It is shown as the service "Transfer from RUNNING to WAIT-ING_CYC" in Table 4-10 and is implemented as the function `task_wait_cycle()` in Table 4-11. Upon being called by a task model (see the example in Section 4.5.2.1), this function firstly calculates the task's next activation time according to its first release time and number of instances that are stored in its TSB. The next activation time is then converted to a sleep value relative to the current time stamp and is set in the `thread_sleep_length` subfield of the task's TCB. Finally, the `task_wait_cycle()` function moves the task to the WAITING_CYC state to let it wait for its next activation. Afterwards, clock interrupts check whether the task should be awakened (see Section 4.5.5.3).

In terms of SystemC implementation, according to the specification in Table 4-10, task services are implemented in the `RTOS` module as normal member functions rather than separate SystemC processes (See Table 4-11). They are invoked by tasks through a pointer to the `RTOS` object. In order to be general, they require minimal input parameters. Depending on needs, a task service can output status values indicating a success or a failure, as well as other specified information. Note that, task state transition services usually result in a rescheduling action by the RTOS scheduler.

```
SC_MODULE(RTOS)
{
    SC_HAS_PROCESS(RTOS);
    RTOS(sc_module_name name, CPU *cpu_i[CPU_NUM]);
    … …
    /*Task state transition-related services*/
    unsigned int task_create(void);
    unsigned int task_terminate(unsigned int tid);
    unsigned int task_delete(unsigned int tid);
    unsigned int task_give_up_CPU(void);
    unsigned int task_resume_sus(unsigned int tid);
    unsigned int task_resume_dly(unsigned int tid);
    unsigned int task_sleep(unsigned __int64 t);
    unsigned int task_wait_suspend(void);
    unsigned int task_wait_cycle(void);
    /*TCB-related services*/
    rtos_tcb* task_tcb_get_pointer(void);
    unsigned int task_tcb_get_info(rtos_tcb *source, rtos_tcb *dest);
    unsigned int task_change_prio(unsigned int tid);
    unsigned int task_change_time_slice(rtos_tcb *tcb,
                                unsigned __int64 new_slice);
};
```

Table 4-11. Implementation of task services

Among the listed services in Table 4-11, the `task_create()` function is another interesting point worthy of discussion. Normally, as a RTOS task service function, `task_create()` could be called either from the main function of a program before the RTOS multi-tasking service starts (i.e., known as *static task creation*), or from a task body function, after the RTOS multi-tasking service starts (i.e., known as *dynamic task creation*). The former is preferable for predictability, whilst the latter bears the hallmarks of flexibility. Many RTOSs support a mixture of these, as does the RTOS model in this thesis. Note that no matter which method is used, execution of the `task_create()` function necessarily consumes some CPU time and should advance the target clock.

Recalling Section 4.5.1.3, we have introduced how `Live_CPU`, `RTOS`, and `task` model objects are created and connected in order to generate a SystemC simulation program in the `sc_main()` function. Further, in Section 4.5.3.1, the detailed creation process of a `task` model object is described. The two sections explicitly explain the modelling method whereby all tasks in the RTOS simulation environment are effectively created by static creation of objects of the `task SC_MODULE` during the SystemC elaboration phase before the start of SystemC simulation. But, does this method contradict or restrict the use of the `task_create()` function?

Explicitly, the creation of a `task` module object in the `sc_main()` function does not contradict using the `task_create()` function in task body functions. The former plays the functional role to create a task in SystemC simulation, but it cannot be used in an application task, nor can it reflect the timing overhead of a dynamic task creation at simulation runtime. The latter is a dummy in terms of its void function. However, it complies with the traditional RTOS programming method by modelling the task creation API of a specific RTOS. This is undertaken in order to support conventional real-time software simulation. In addition, in case of a dynamic creation in simulation, it can be annotated with timing consumption of a task creation service, and hence can represent its timing behaviour at a correct timing point when it is called. This dual task creation technique utilises SystemC modular modelling approach and supports native-code real-time software models.

In this thesis, the `task_create()` function can model both the *static task creation* and *dynamic task creation*, provided that all related `task` module objects have been statically created. This "pseudo" *dynamic task creation* could be seen as a limitation of the modelling method. The reason for this is that a task is created by creating a SystemC `SC_MODULE`, but the SystemC standard does not natively support *"dynamic creation or modification of the module hierarchy during simulation"* [66].

4.5.5 Scheduler Modelling

According to the survey in Section 4.4.3.3, like situations in most practical RTOSs, the RTOS model includes a priority-based pre-emptive scheduler. In terms of scheduling policies, FPS is the basic scheduler model, while FIFO and RR deal with equal-priority tasks. Furthermore, EDF scheduling is regarded as an experimental add-on algorithm. In execution, the scheduler is invoked in a combinational way by two common modes found in RTOSs, i.e., time-driven and event-driven [26].

4.5.5.1 The Priority Assignment and the FPS Scheduling Model

The Priority Assignment is the basis of scheduling in the RTOS model for FPS scheduling. Figure 4-14 depicts the priority setting of the RTOS model. This prioritisation system is fully configurable by defining some constants, as shown in the figure. In general, at least 256 levels should be available with the exact number depending on a specific configuration.

The lowest priority level 0 (i.e., the smallest number) is always assigned to the special *IDLE task*[11] . Some of the highest priority levels (i.e., the largest numbers) are currently reserved without use. In the whole priority range, all ISR priorities are higher than normal task priorities. In the RTOS model, these ISRs represent special kinds of aperiodic tasks that can be defined by users, but are not equal to user-defined normal aperiodic tasks that belong to normal real-time tasks in this model. The specific priority ordering algorithm for normal real-time tasks is dependent on the simulation user's choice and is unimportant to RTOS modelling research here (See Section 4.4.3.3 for an introduction to some classical priority ordering algorithms). If there are non-real-time tasks in the system, then they should be allocated priorities lower than all other real-time tasks.

Note that in the TCB depicted in Table 4-8, there are two priority fields, i.e.,

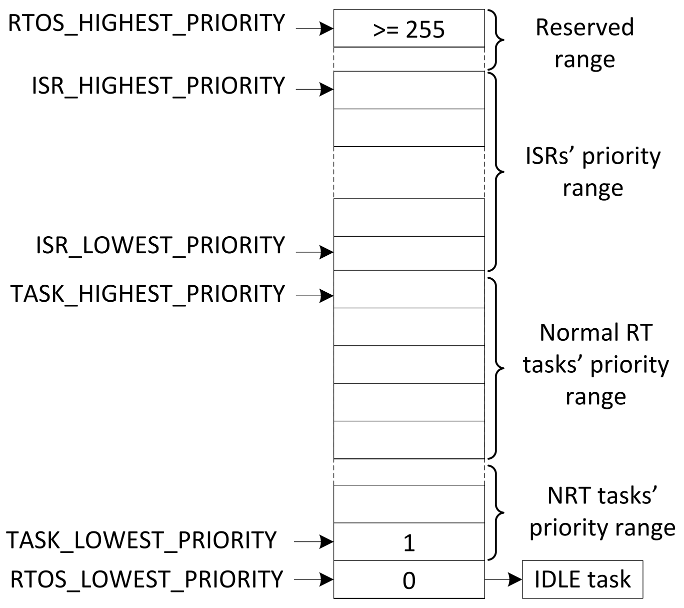


Figure 4-14. Priority setting in the RTOS task model

[11] The IDLE task is always ready to run. It is dispatched when there are not any other runnable tasks in the system, which actually means that the CPU is idle.

`rtos_tcb_base_prio` and `rtos_tcb_cur_prio`, which represent the basic (initial) priority of a task and the current (effective) priority of a task, respectively. In RTOS execution, a task's current (effective) priority is used by the scheduler because it is updated in case of a priority change operation.

The basic algorithm of FPS is to compare the current priority of the RUNNING task and the current priority of the first task in the ready queue. The result of the comparison is the basis on which to make a scheduling decision. Regarding FIFO and RR algorithms in the scheduler model, their theories and usages were introduced in Section 4.4.3.3. The RTOS model follows the classical concepts and can choose one of the two algorithms for all tasks in the system.

4.5.5.2 Implementation of the FPS Scheduler in the SystemC Model

In SystemC implementation of the RTOS model, the scheduler is implemented as a function (i.e., `scheduler()`) in the RTOS module and is called by other RTOS services. It is conceptually different from methods in [72] [113] [185] that model the scheduler as a continuously-running SystemC process to schedule multiple tasks and activate a task to run. Literature [114] has compared the two scheduler modelling techniques and proposed that the function-call modelling technique is preferable mainly because it does not incur SystemC kernel context switches between the scheduler SystemC process and task SystemC processes, namely less simulation overhead. In contrast, a dedicated SystemC process-based RTOS scheduler has the advantage of easy implementation. In addition, in this thesis, there appear another three benefits to implement a function-call-based RTOS scheduler:

1) Support of traditional usage as it complies with a normal situation of the scheduler in a RTOS kernel. Also, invoking the scheduler function in traditional real-time software code is straightforward.
2) Support of a timing model. It is also easy to model the timing behaviour of a scheduler function in software simulation because it behaves in a similar way to real execution.
3) Better modularity because it simplifies the function of the RTOS scheduler and decouples it from a combination of a RTOS scheduler, an interrupt

monitor, and a conceptual software executing engine in [72] [113] [185]. In the RTOS model in this thesis, the scheduler just finishes a reasonable software function to choose the next-to-run task and then calls the task switch service. The low-level task switch service and the Live CPU Model collectively finish the remaining work to activate the next-to-run task.

Referring to Figure 4-15, the working flow of the FPS scheduler model can be described as follows:

1) Once the scheduler is triggered, it compares the current priority of the RUNNING task and the current priority of the first task in the ready queue. There may be three results:
2) If the current RUNNING task's priority is higher, then the scheduler needs to check whether the RUNNING task is blocked by a condition. If it is blocked, then the RUNNING task is moved to the waiting queue, and the first READY task is chosen as the new next-to-run task. Otherwise, the scheduler just exits. No task switch is necessary, and the RUNNING task

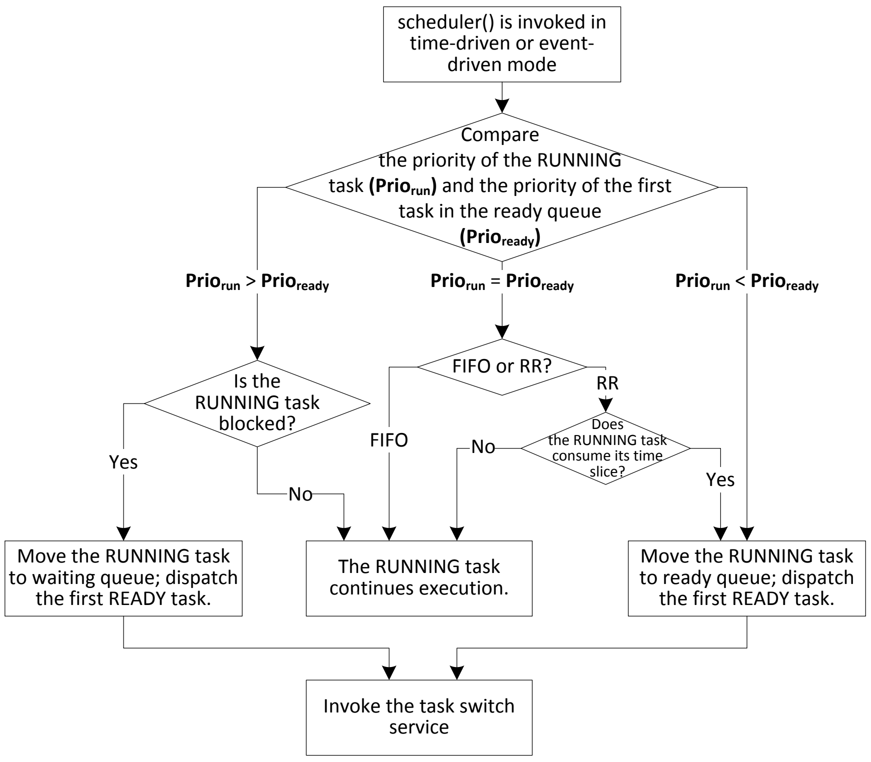

scheduler() is invoked in time-driven or event-driven mode
Compare the priority of the RUNNING task (Prio_run) and the priority of the first task in the ready queue (Prio_ready)
Prio_run > Prio_ready
Prio_run = Prio_ready
Prio_run < Prio_ready
Is the RUNNING task blocked?
Yes
No
FIFO or RR?
FIFO
RR
Does the RUNNING task consume its time slice?
No
Yes
Move the RUNNING task to waiting queue; dispatch the first READY task.
The RUNNING task continues execution.
Move the RUNNING task to ready queue; dispatch the first READY task.
Invoke the task switch service

Figure 4-15. FPS scheduler working flow

continues execution.

3) If the first READY task's priority is higher, then it is removed from the ready queue and chosen as the new next-to-run task. The old RUNNING task is inserted into the ready queue, which means it is pre-empted. Then the scheduler calls the task switch service and finishes.
4) When their priorities are equal and if the FIFO algorithm is set up, then the RUNNING task continues executing and the scheduler just exits. If the RR algorithm is chosen, the scheduler checks whether the RUNNING task's time slice is exhausted. If it is, then the first READY task is dispatched as the new RUNNING task and the old RUNNING task is inserted into the ready queue. If the RUNNING task's time slice still exists, then the scheduler just exits.

4.5.5.3 Time-Driven and Event-Driven Scheduling

Time-driven scheduling is also called *tick scheduling* or *time-based scheduling* [26]. In this mode, the scheduler is periodically triggered by *clock interrupts* to make scheduling decisions. The time interval between two clock interrupts is defined as the *time resolution* of the system, also known as a *system tick*. Some RTOSs use the system tick mechanism to delay task executions [149] [152]. However, the actual delay time may not be exactly the same as the appointed ticks, but have possible sleeping jitters due to clock interrupt handling and scheduling time. The author in [149] discussed this problem and concluded that increasing the frequency of clock interrupts may be a solution. Indeed, the length of the system tick greatly affects the responsiveness and run-time overhead of a RTOS system. A minor value of the tick could improve system responsiveness in terms of the ability to handle periodic tasks with high activation rates [25]. However, a too small tick size also means that the tick scheduling service is activated very frequently, resulting in a higher runtime overhead. The tick size used by most operating systems is 1-50 milliseconds [25] and is fully configurable in this RTOS model.

In implementation, the clock interrupt and its associated ISR are modelled by the standard interrupt handling method of the proposed modelling approach,

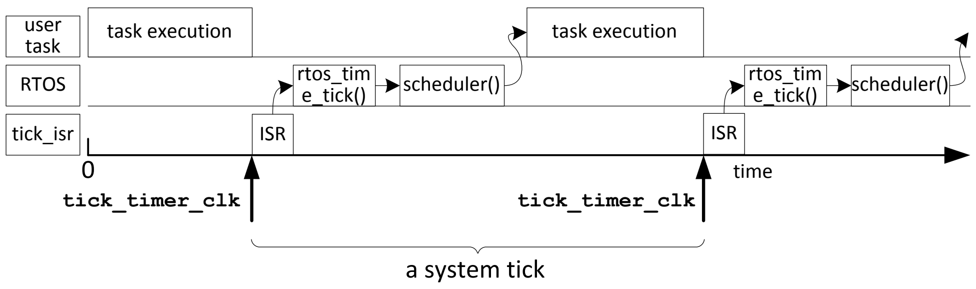


Figure 4-16. Tick scheduling model

which will be addressed in Section 4.5.7. Referring to Figure 4-16, a configurable clock interrupt `tick_timer_clk` (a `sc_clock` object) periodically triggers the tick timer ISR `tick_isr`. The ISR then calls a RTOS kernel function `rtos_time_tick()` to carry out the following actions:

- It checks and updates the status of sleeping tasks in the waiting queue (i.e., at WAITING_DLY and WAITING_CYC states). If a sleeping task expires, the task is moved to the ready queue at the time.
- It updates the execution budget of the RUNNING task if it is scheduled by the RR policy.
- It monitors the absolute deadline of the RUNNING task (if this property is available) and notifies the kernel in case it is missed.
- It finally calls the `scheduler()` function to make a scheduling decision.

In the event-driven mode, the RTOS scheduler is invoked by various events and should act immediately upon their occurrences. These scheduling events can be hardware-sourced external interrupts or internal to the software system, for example, a task is created or unblocked by RTOS services.

4.5.5.4 Supporting the Dynamic-Priority EDF Algorithm

Although not common in practical RTOSs, various theoretical issues in EDF scheduling have been thoroughly studied in real-time systems research [176]. Some abstract RTOS models have also simply mentioned an EDF scheduler [72] [11] [8]. However, according to the survey on RTOSs scheduling algorithms in Section 4.4.3.3, an EDF scheduler may not be a necessity or a desired function of a RTOS model that aims to model practical and general scheduling behaviours of some common RTOS products. In [176] [26], two implementation methods of an EDF scheduler in OSs are discussed:

1) Implementing an EDF scheduler on top of usual RTOS kernel with a limited number of priority levels: The kernel maps absolute deadlines to priorities and allows changing priority at runtime. However, this method is "not easy nor efficient" [176]. [176] shows an example situation: at execution runtime, if two task jobs have been allocated two adjacent priority levels according to their absolute deadlines, then it is not easy to allocate a priority to the third task job that has an intermediate absolute deadline. The only solution deemed in [176] is to remap the two existing jobs to new nonadjacent priority levels. Possibly, in the worst case, all jobs in the ready queue may need priority remapping and the incurred overhead could be excessive.
2) Implementing an EDF scheduler on top of a deadline-based RTOS kernel: The ready queue of the RTOS kernel orders tasks according to increasing absolute deadlines. This method is believed to be a "better alternative" [26] because it needs a relatively small modification of the kernel structure and its services. Basic queue operations such as insertion, deletion, and returning the queue head all behave similarly to those priority-based queue operations. The absolute deadline of a task actually plays as the "priority" of a task in this model. This implementation method requires that the absolute deadline of a task is calculated at each release time and recorded in its TCB.

It is noticed that the EDF scheduler in [72] is implemented in the first priority reassignment method. However, in this thesis, the EDF model follows the second method. In fact, various implementation elements of this model have already been referred to in the above paragraphs of this section.

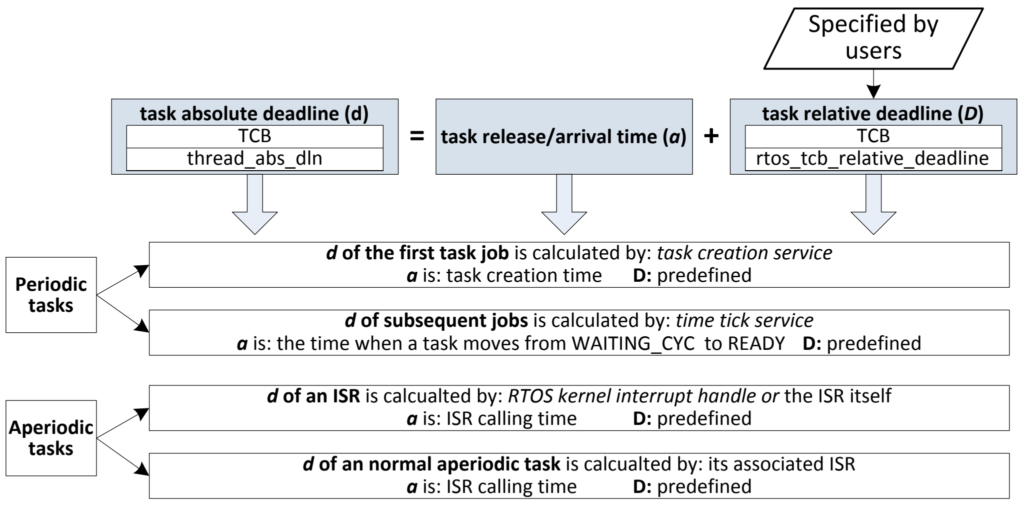
In the TCB definition in Table 4-8, the task relative deadline should be specified by the user and stored in the `rtos_tcb_relative_deadline` field, and the task absolute deadline task is stored in the `thread_abs_dln` sub field. In Section 4.5.4.2, the priority-based task queue class is introduced and it has the possibility of becoming an absolute deadline-based queue.

It is well known that: task absolute deadline (*d*) = task release time (*a*) + task relative deadline (*D*). The task relative deadlines are defined by users, i.e., they are known. The difficulty of modelling an EDF scheduler in the RTOS model is mainly dependent on how to determine task release times, by which task absolute deadlines can be calculated. Referring to Figure 4-17, the proposed implementation method is described as follows:

For periodic tasks, it is required that each of them should enter the WAITING_CYC state to wait for next activation when it finishes its current execution cycle. The method is carried out by two RTOS services:

- The task creation service (in Section 4.5.3.1) uses the task creation time (or adding an offset) as *a* of the first job of a task. Hence, *d* of the first job is obtained.
- The RTOS time tick service (in Section 4.5.5.3) takes charge of calculating *d* for subsequent jobs of a task. If the tick service moves a task from the WAITING_CYC state to the READY state, then it means that a task has entered its new cycle. This time point is deemed as the approximate *a* with a possible but acceptable sleeping time jitter.

There are two kinds of aperiodic tasks in the model, namely ISRs and normal aperiodic tasks. ISRs provide first level (i.e., early) simple software interrupt services, while normal aperiodic tasks provide second level (i.e., later) software in-



Specified by users
task absolute deadline (d)
TCB
thread_abs_dln
=
task release/arrival time (a)
+
task relative deadline (D)
TCB
rtos_tcb_relative_deadline
Periodic tasks
d of the first task job is calculated by: task creation service
a is: task creation time D: predefined
d of subsequent jobs is calculated by: time tick service
a is: the time when a task moves from WAITING_CYC to READY D: predefined
Aperiodic tasks
d of an ISR is calcualted by: RTOS kernel interrupt handle or the ISR itself
a is: ISR calling time D: predefined
d of an normal aperiodic task is calcualted by: its associated ISR
a is: ISR calling time D: predefined

Figure 4-17 Calculating absolute deadlines of tasks in simulation

terrupt services. The calculation of *d* is twofold:

1) ISRs are either directly invoked by the hardware interrupt controller or by a RTOS kernel interrupt handler. In the case of the former mode, an ISR uses its beginning time as *a in order* to calculate its *d*; in the case of the latter mode, a RTOS kernel interrupt handler will use its calling time as *a* of the ISR and calculate its *d*.
2) Normal aperiodic tasks are initiated by an associated ISR through synchronisation methods. Hence, a precedent ISR can use this calling time as the *a* of a subsequent normal aperiodic task, with *d* calculated from it.

Except for the above points, other implementation details of the EDF scheduler are similar to the FPS scheduler. Bear in mind the EDF scheduler is implemented in the RTOS kernel with some restrictive conditions on both application models and the RTOS model. Consequently, it is unable to refer to a practical RTOS. The research in this thesis does not aim to implement many RTOS functions in this EDF model.

4.5.6 Task Synchronisation and Communication Modelling

According to the survey in Section 4.4.3.5, in a multi-tasking RTOS environment, application tasks need to synchronise and share data, in order to cooperate with each other properly. Some synchronisation (e.g., semaphores, mutexes, and event flags) and communication methods (e.g., mailboxes and message queues) are used in various RTOSs as lightweight mechanisms to ensure inter-task/thread synchronisation, mutual exclusion, and communication. A general difference between a synchronisation method and a communication method is that the former is used mainly to coordinate the execution orders of involved tasks, while the latter can explicitly exchange data between tasks.

In Section 2.2.2.2, SystemC built-in synchronisation primitive channels (e.g., `sc_semaphore`, `sc_mutex`, and `sc_fifo`) were discussed, with the conclusion that they are not suitable for direct use in a RTOS model, due to their non-deterministic characteristics. Hence, the proposed RTOS model natively implements four three real-time synchronisation and communication methods, i.e., semaphores, mutexes, and message queues, with the PIP protocol being applied for mutexes in order to avoid the priority inversion problem. Their usage, for instance whether a specific synchronisation or communication function is allowed to be used in ISRs, is consistent with referenced RTOSs [149] [152] .

4.5.6.1 The Event Control Block

Referring to Table 4-12, the RTOS model uses a universal Event Control Block (ECB), in common with μC/OS-II RTOS [149], to control different synchronisation and communication entities (referred to as *event objects* hereafter) at the kernel level. These different types of event objects share some fields and primitives of the ECB method. This implementation technique brings reusability to RTOS modelling. In addition, they contain respective fields and application interface functions.

An ECB represents the various characteristics of an event object. As shown in Table 4-12, all types of event objects own an ECB ID, an ECB type property, a pointer to its respective resource, and a suspension task list. Besides this, a mutex or a semaphore event object also needs a counter field. More particularly, a mutex ECB records the original priority of its owning task for the PIP protocol, and a

Event Control Block (struct rtos_ecb)	
Field	**Description**
rtos_ecb_id	ID of this ECB
rtos_ecb_event_type	Type of this event object, e.g., mutex, semaphore, ...
rtos_ecb_counter	Value of a semaphore / a mutex
rtos_ecb_original_priority	Original priority of a mutex owner
rtos_ecb_ceiling_priority	Ceiling priority of a resource (Reserved for PIP in mutex)
*rtos_ecb_ptr	Pointer to a possible 2nd-level control block/resource
rtos_ecb_task_list	Suspension task list (based on the STL list class)

Suspension task list (struct tid_priority_block)
Task ID
Task current priority

ECB management primitives		
`sync_create()`	`sync_del()`	`sync_wait()`
`sync_timeout()`	`sync_signal()`	

Table 4-12. Event control block (ECB) and management primiitves

ceiling priority field is reserved for the PCP protocol. The suspension task list is based on the STL *list* template class [139]. An element (i.e., `struct tid_priority_block`) of the list includes two essential properties of a task, i.e., a task ID and the current priority. The task suspension list can be ordered by either FIFO or priority, which is able to model optional features provided by some RTOSs, e.g., ThreadX. By default, the highest-priority task is placed at the head of the suspension list in the RTOS model.

In Table 4-12, five basic primitive functions are implemented to manage an ECB, i.e., creating an ECB, deleting an ECB, waiting for an event object (namely a P operation), waiting for an event object with a timeout, and signalling an event object (namely a V operation). These kernel functions are called by different synchronisation and communication application functions accordingly.

In order to explain these primitives, Table 4-13 shows an example code of the "waiting for an event object" function (`sync_wait()`) and the "signalling an event object" function (`sync_signal()`). The processing sequences of the two functions are similar in terms of including three sequential steps, i.e., operating the ECB task suspension list, operating the task's TCB, and operating the RTOS

```
void RTOS::sync_wait(rtos_tcb *ptcb, rtos_ecb *pecb)
{
    tid_priority_block tmp_tpb = {ptcb->rtos_tcb_tid,
                                  ptcb->rtos_tcb_thread_cur_prio};
    pecb->rtos_ecb_thread_list.push_back(tmp_tpb);
    pecb->rtos_ecb_thread_list.sort(greater<tid_priority_block>());
    ptcb->rtos_tcb_ecb_ptr = pecb;
    RTOS_TCB_WAITING_QUEUE_0.insert_node_priority(ptcb);
}

void RTOS::sync_signal(rtos_tcb *ptcb, rtos_ecb *pecb, void *msg)
{
    rtos_tcb      *ptcb_rdy;
    ptcb_rdy = &rtos_tcb_array[(pecb->
                                    rtos_ecb_task_list.front()).tid];
    pecb->rtos_ecb_thread_list.pop_front();
    ... ...
    ptcb_rdy->rtos_tcb_wait_flag &= ~pecb->rtos_ecb_event_type;
    ptcb_rdy->rtos_tcb_ecb_ptr = NULL;
    ... ...
    if (ptcb_rdy->rtos_tcb_wait_flag == WAITING_NUL) //Check again
    {
         RTOS_TCB_WAITING_QUEUE_0.delete_node(ptcb_rdy);
         RTOS_TCB_READY_QUEUE_0.insert_node_priority(ptcb_rdy);
    }
}
```

Table 4-13. Example code of *wait* and *signal* primitives

task queues. However, their exact functions differ. The `sync_wait()` function firstly inserts a blocked task in the ECB task suspension list (lines 5, 6), then records the ECB in this task's TCB (line 7), and finally puts this task in the RTOS waiting queue (line 8). In contrast, the `sync_signal()` function firstly removes the unblocked task from the ECB task suspension list (line 16), then clears blocking information from this task's TCB (lines 18, 19), and finally moves the task from the RTOS waiting queue to the ready queue (lines 23, 24).

4.5.6.2 Modelling Semaphores

In the RTOS model, a counting semaphore includes a 32-bit counter (i.e., the `rtos_ecb_counter` field in an ECB). Its value represents how many tasks are allowed to access the protected resource. Its usage complies with normal situations in RTOSs:

- A semaphore does not have a notion of ownership, and any tasks can *wait* (i.e., P) or *post* (i.e., V) a semaphore.
- A positive counter value means resources are available, while a zero value means the resource is unavailable.
- The wait operation will decrement the counter value by one. If a counter value reaches zero, then a wait operation will block the calling task (i.e., at the WAITING_SEM state) and put into the suspension task list.
- A post operation will increment the counter by one or unblock the highest-priority task in the suspension task list.

Table 4-14 enumerates seven semaphore services supported in the RTOS model and corresponding services provided by three referenced RTOS products, which shows that the proposed RTOS model has a good coverage of typical

Semaphore services in the RTOS model	Semaphore services in some RTOSs		
	ThreadX	μc/OS-II	RTX
Initialise a semaphore	tx_semaphore_create	OSSemCreate	os_sem_init
Destroy a semaphore	tx_semaphore_delete	OSSemDel	
Wait (P) for a semaphore	tx_semaphore_get	OSSemPend	os_sem_wait
Wait for a sem. with a timeout			
Wait for a sem without blocking		OSSemAccept	
Post (V) a semaphore	tx_semaphore_put	OSSemPost	os/isr_sem_send
Get semaphore counter value	tx_semaphore_info_get	OSSemQuery	

Table 4-14. Semaphore services in the RTOS model and some RTOSs

```
SC_MODULE(RTOS)
{
    ... ...
    int  sem_init(rtos_ecb *psem, int pshared, int c_value);
    int  sem_destroy(rtos_ecb *psem);
    int  sem_wait(rtos_ecb *psem);
    int  sem_timedwait(rtos_ecb *psem, unsigned __int64 nanoseconds);
    int  sem_post(rtos_ecb *psem);
    int  sem_trywait(rtos_ecb *psem );
    int  sem_getvalue(rtos_ecb *psem, int *value);
    ... ...
};
```

Table 4-15 POSIX-like semaphore APIs in the RTOS model

semaphore functions. In fact, because of similarities on common semaphore functions across different RTOSs, the proposed RTOS model's semaphore services can be adapted to various APIs. As shown in Table 4-15, default semaphore interfaces in the proposed RTOS partially refer to the RT-POSIX standard, in terms of similar functions, arguments, and return values.

In terms of implementation, it is not only necessary to model typical semaphore services, but also needs to be aware of the RTOS model's particular characteristic, i.e., semaphore services run in the software PE simulation model in the SystemC environment.

Table 4-16 gives the example implementation code of the `sem_wait()` func-

```
int RTOS::sem_wait(rtos_ecb *psem)
{
    rtos_tcb *ptcb;
    ptcb = RTOS_RUNNING_TCB;
    ... ...
    if (psem->rtos_ecb_counter > 0)          // Semaphore is available
    {
        psem->rtos_ecb_counter--;
    }
    else                                     // Semaphore is unavailable
    {
        /* Set WAITING flag */
        RTOS_RUNNING_TCB->rtos_tcb_wait_flag |= WAITING_SEM;
        sync_wait(RTOS_RUNNING_TCB, psem);  // Call sync_wait primitive
        scheduler();                         // Call scheduler function
        /* Call Live CPU Model to run a READY task */
        m_CPU_ptr[0]->scevt_rtos_call_cpu.notify(SC_ZERO_TIME);
        /* The calling task is blocked here */
        wait(rtos_tcb_service_array[ptcb->rtos_tcb_tid].rtos_tcb_evt[0]);
        /* After waiting, the calling task is unblocked */
        ... ...
    }
}
```

Table 4-16. SystemC implementation code of the *sem_wait()* function

tion. Note that only part of the original code is displayed in the figure due to a page limit. The `sem_wait()` function is used inside a task body function when the task wants to acquire a semaphore count. In case of a positive semaphore value, the semaphore counter is simply decreased by one (line 6); in case of a zero semaphore value, the before-mentioned `sync_wait()` primitive is called (line 14) to block the calling task. Afterwards, the `scheduler()` function (introduced in Section 4.5.5.2) is invoked to make a rescheduling decision (line 15). On line 17, because the `scheduler()` function should have already selected (namely dispatched) a new task as the next-to-run task, the Live CPU Model is thus triggered by notifying a `sc_event` in order to execute the new task. Then, on line 19, the calling task is blocked by a *wait-for-event* statement. This `sc_event` will be released at a future time point when the task is unblocked.

4.5.6.3 Modelling Mutexes

In the RTOS model, a mutex is used to provide mutually exclusive access to a critical section. Its counter has a binary value stored in the ECB `rtos_ecb_counter` field. Its usage complies with normal situations in RTOSs:

- A mutex is a public object but can be owned by one task at any time and whose ownership is indicated by the ECB `*rtos_ecb_ptr` pointer.
- The *lock* (i.e., P) operation tries to acquire the mutex and decrements mutex value from one to zero if it succeeds. If a task attempts to lock a mutex, but the mutex has been already locked by another task previously, then the calling task will be blocked (i.e., at the WAITING_MUT state) and put in the suspension task list. With the PIP protocol, if a high priority $task_b$ is blocked by a mutex that is owned by a low priority $task_a$, then $task_a$ temporarily inherits the high priority of $task_b$ and the original priority of $task_a$ is stored in the ECB.
- The *unlock* (i.e., V) operation releases mutex ownership. It increments mutex value from zero to one or unblocks the highest priority blocked task in the suspension task list. With PIP, the task that calls the unlock function will revert its original priority.

Mutex services in the RTOS model	Mutex services in some RTOSs		
	ThreadX	μc/OS-II	RTX
Initialise a mutex	tx_mutex_create	OSMutexCreate	os_mut_init
Destroy a mutex	tx_mutex_delete	OSMutexDel	
Lock (P) a mutex		OSMutexPend	
Lock (P) a mutex with a timeout	tx_mutex_get		os_mut_wait
Lock a mutex without blocking		OSMutexAccept	
Unlock (V) a mutex	tx_mutex_put	OSMutexPost	os_mut_release
	tx_mutex_info_get	OSMutexQuery	

Table 4-17. Mutex services in the RTOS model and some RTOSs

```
SC_MODULE(RTOS)
{
    ... ...
    int  pthread_mutex_init(rtos_ecb *pmutex, int *attr);
    int  pthread_mutex_destroy(rtos_ecb *pmutex);
    int  pthread_mutex_lock(rtos_ecb *pmutex);
    int  pthread_mutex_timedlock(rtos_ecb *p, unsigned __int64 timeout);
    int  pthread_mutex_unlock(rtos_ecb* pmutex);
    int  pthread_mutex_trylock(rtos_ecb *pmutex);
    ... ...
};
```

Table 4-18 POSIX-like mutex APIs in the RTOS model

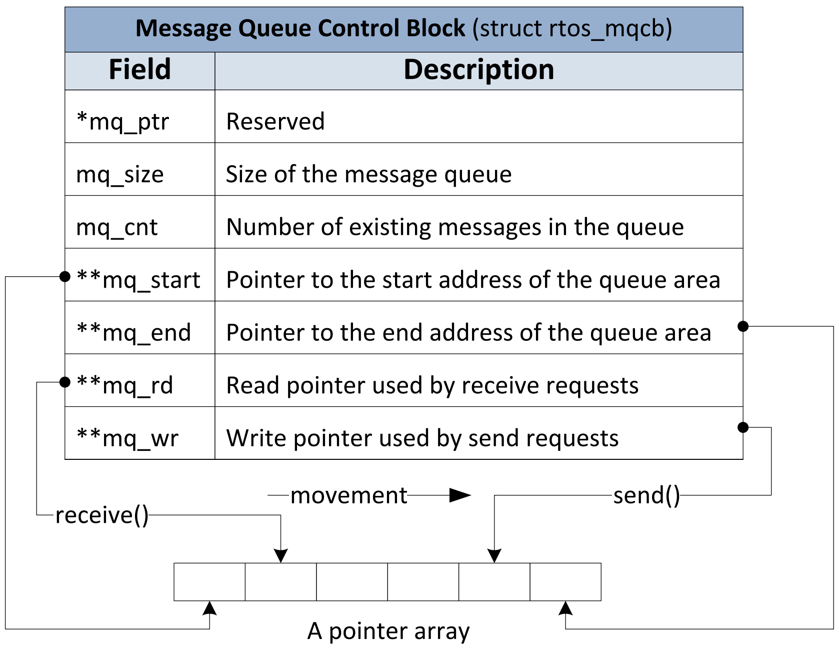
Table 4-17 enumerates six mutex services supported in the RTOS model, which also have an approximate equivalence to corresponding services provided by the three referenced RTOS products. Table 4-18 shows default mutex interfaces implemented in the proposed RTOS, which partially refer to the RT-POSIX standard. The modelling method of a specific mutex service is similar to the semaphore modelling technique in last section. Hence, it is not repeated again.

4.5.6.4 Modelling Message Queues

Currently, message queues are the main inter-task communication method in the RTOS model. A message queue is a public resource and can be connected to various sender tasks and receiver tasks (using the *receive* operation). In the model, by default, multiple messages are stored in a FIFO order queue and each message is actually a pointer to a variable (e.g., character type, unsigned integer type, integer type, float type, and double float type) that is to be communicated. The LIFO order queue and the transfer-by-copy function are not currently implemented. The usage of a message queue is as follows:

- The *send* operation inserts a message pointer into the message queue. If the queue is full, the calling task will be blocked (i.e., at the WAITING_QUE state) and put into the ECB suspension task list.
- The *receive* operation retrieves and removes a message pointer from the message queue. If the message queue is empty, the calling task will be blocked and put into the ECB suspension task list.
- The unblocking conditions of *send* and *receive* operations are similar to previously mentioned semaphore and mutex behaviours and hence are abbreviated here.

In implementation, a message queue needs a special second-level control block (i.e., `rtos_mqcb`) in addition to its ECB. Its structure partially refers to µC/OS-II RTOS [149]. As shown in Figure 4-18, a message queue control block stores various control information regarding a message queue and is involved in send and receive operations. The read and write pointers move in the same direction from the start address to the end address of the pointer array, i.e., messages are First-In-First-Out.



Message Queue Control Block (struct rtos_mqcb)	
Field	**Description**
*mq_ptr	Reserved
mq_size	Size of the message queue
mq_cnt	Number of existing messages in the queue
**mq_start	Pointer to the start address of the queue area
**mq_end	Pointer to the end address of the queue area
**mq_rd	Read pointer used by receive requests
**mq_wr	Write pointer used by send requests

Figure 4-18 Message queue control block

Message queue services in the RTOS model	Message queue services in some RTOSs		
	ThreadX	µc/OS-II	RTX
Initialise a message queue	tx_queue_create	OSQCreate	os_mbx_init
Destroy a message queue	tx_queue_delete	OSQDel	
Receive a message	tx_queue_receive	OSQPend	os_mbx_wait
Receive a msg. with a timeout			
Send a message	tx_queue_send	OSQPost	os/isr_mbx_send
Send a message with a timeout			
	tx_queue_flush	OSQFlush	
	tx_queue_front_post	OSQPostFront	
	tx_queue_info_get	OSQQuery	os/isr_mbx_check
		OSQAccept	isr_mbx_receive

Table 4-19. Message queue services in the RTOS model and some RTOSs

```
SC_MODULE(RTOS)
{
    ... ...
    int  mq_open(void **start, int size,
                 rtos_ecb *pecb, rtos_mqcb *pmqcb);
    int  mq_close(rtos_ecb *pecb);
    int  mq_receive(rtos_ecb *pecb, void* msg_ptr,
                 MQ_SIZE_T msg_len, unsigned int* msg_prio);
    int  mq_timedreceive(rtos_ecb *pecb, void *msg_ptr,
                 unsigned __int64 nanoseconds);
    int  mq_send(rtos_ecb *pecb, void *msg_ptr);
    int  mq_timedsend(rtos_ecb *pecb, void *msg_ptr, MQ_SIZE_T msg_len,
                 unsigned int msg_prio, unsigned __int64 nanoseconds);
    ... ...
};
```

Table 4-20. POSIX-like message queue APIs in the RTOS model

Table 4-19 lists message queue services in the RTOS model and referenced RTOSs. Currently, six basic functions have been included in the model and other additional RTOS-specific functions can be implemented in future work. In Table 4-20, RT-POSIX-like interfaces are utilised again as the wrapper of message queue functions in the RTOS model. Note that a standard RT-POSIX message has a priority property, whereas the proposed RTOS model does not support this feature. Hence, the priority argument in these APIs is meaningless at the time.

4.5.7 Interrupt Handling Modelling

4.5.7.1 Basic Concepts of Interrupt Handling

As mentioned in Section 4.4.3.1, interrupt handling is a crucial mission of the RTOS for servicing IRQs that are generated by external devices. In different RTOSs, there are various interrupt handling mechanisms. Focusing on handling interrupts on the ARM processor, Sloss et al. survey eight interrupt handling schemes including the simple non-nested method; complex grouped and prioritised methods; and the vector interrupt controller method [67]. Based on this survey, three notable common characteristics are extracted and should be considered in modelling:

1) Nested: A non-nested scheme handles individual interrupts sequentially. When an interrupt is being serviced, other interrupts are disabled; hence interrupt latency is substantially high. In contrast, a nested scheme allows the handling of another interrupt during the current interrupt handler. In a simple nested scheme, interrupts may not be prioritised, which means the newest interrupt can block an existing one.
2) Prioritised: Interrupts are assigned priorities that indicate their stringency. A higher-priority interrupt is serviced in precedence to a lower-priority interrupt, which also means a lower-priority interrupt is ignored if it happens during a higher-priority interrupt handling process. Depending on specific implementation, either a hardware interrupt controller or a low-level software handler (i.e., in RTOS or drivers) can achieve interrupt prioritisation.
3) Vectored: In a non-vectored interrupt handling scheme, the entry point of all software ISRs remains the same, i.e., either a RTOS kernel interrupt handling function or a similar low-level software handler, which takes charge of determining which ISR should serve the raised IRQ and then load the ISR into the program counter of the CPU for execution. In a vector-based scheme, the hardware vector interrupt controller has an array (i.e., a vector) of ISR addresses. Hence, a specific software ISR can be invoked by the hardware directly, which means a smaller interrupt latency. These two schemes are illustrated in Figure 4-5 and referred to as the RTOS-assisted scheme and the vector-based scheme, respectively.

4.5.7.2 The RTOS Interrupt Handling Model

In order to model typical interrupt schemes in this research, the RTOS model provides a modular interrupt handling model. It splits interrupt handling functions in the software PE model into several cooperative HW and SW components. Through configuration of these components, the interrupt handling model can flexibly support the above-mentioned nest, prioritisation, non-vector, and vector features.

In the software PE model, one hardware component is related to interrupt handling, i.e., the Interrupt Controller Model in the Live CPU Model (see Section 3.3.3). It is the lowest-level component in the interrupt handling stack and connected to various hardware sources by IRQ lines. Its function and structure has already been introduced in detail, so they are not repeated here. Just remember, an essential function of the Interrupt Controller Model is to invoke upper-level software interrupt handlers.

Regarding software parts in the interrupt handling stack, there contains the following components:

- **RTOS kernel-level interrupt handler functions** (i.e., `interrupt_handler_enter()` and `interrupt_handler_exit()`): Depending on a specific interrupt scheme, they are invoked by either the hardware Interrupt Controller Model or user-level ISRs, and their functions also may vary but in general can prioritise and mask interrupts and call ISRs if necessary.
- **User-defined ISRs** (also known as immediate interrupt services [26]): they are attached to corresponding interrupts and programmed by users to provide simple and non-blocking functions (e.g., post a semaphore) in order to serve an IRQ promptly. They are assigned higher priorities than are normal user tasks, as shown in Figure 4-14, among which the tick timer ISR `tick_isr` (introduced in Section 4.5.5.3) has the highest priority in the default setting. Note that a lower-priority ISR can be pre-empted by a higher-priority ISR. Depending on the configuration of the interrupt model, a user-defined ISR can be invoked by the RTOS kernel interrupt handler indirectly or the hardware Interrupt Controller Model directly.
- **User-level aperiodic tasks** (also known as scheduled interrupt services [26]): they are normal real-time tasks in the RTOS model. Because user-defined ISRs are typically too simple to include all necessary interrupt handling functions, subsequent aperiodic tasks are always necessary so as to complete interrupt handling [26]. Their priorities reside in the range of nor-

mal real-time tasks, and are usually set to be higher than periodic tasks. These aperiodic task stay at the WAITING state in normal times and are triggered by synchronisation functions that are operated by user-defined ISRs.

The RTOS model can currently support two typical interrupt handling schemes as shown in Figure 4-5, i.e., the RTOS-assisted (non-vectored) scheme and the vector-based scheme. No matter in which scheme, nested, prioritised, and maskable handling functions can all be supported.

Figure 4-19 depicts the process of the RTOS-assisted (non-vectored) interrupt handling scheme. In this scheme, the RTOS kernel-level interrupt handler `interrupt_handler_enter()` is the entry point for all ISRs. It is implemented as a SystemC `SC_THREAD`, which is sensitive to a related `sc_event` in the Live CPU Model. The handling process includes the following functions and transition steps:

1) In Step 1, the Interrupt Controller Model releases the `sc_event` when it finds an IRQ.
2) Upon being triggered, the RTOS interrupt entry handler firstly identifies the external IRQ source and masks other lower-priority IRQs (i.e., ignores their occurrence during this handling process) by setting interrupt-related virtual registers in the Live CPU Model. The entry handler then pre-empts

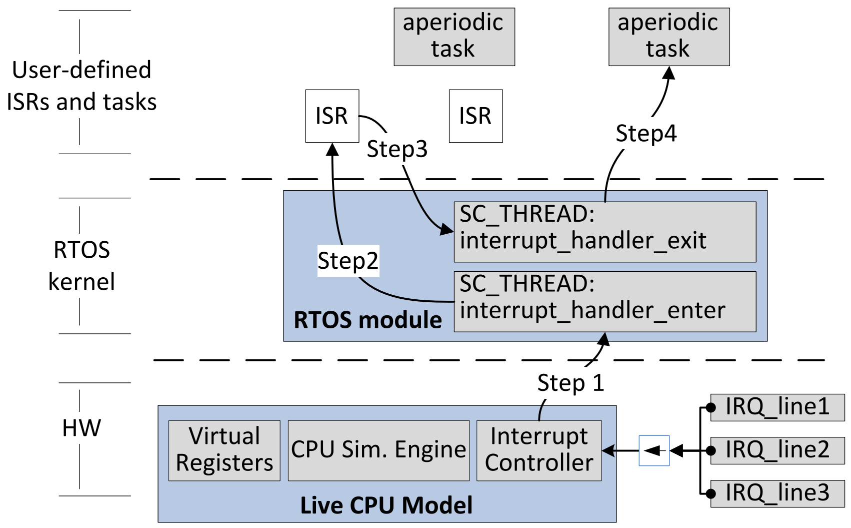


Figure 4-19 RTOS-assisted (non-vectored) interrupt handling model

the RUNNING task and inserts it into the RTOS ready queue. Possibly, the pre-empted "task" may be another lower-priority ISR, and thus the entry handler will operate an `IRQ_NEST_LIST` and a `RTOS_IRQ_NEST_COUNT` counter in order to record this nested situation for later recovery. The entry handler also notifies the Live CPU Simulation Engine in order to stop a time advance of the pre-empted task (details are introduced in Section 3.3.4.2). Finally, in Step 2, the entry handler sets the corresponding ISR as the next-to-run task, invokes a context switch, and triggers the Live CPU Simulation Engine to start. Note that this prioritised and masked interrupt handling process guarantees that the priority of new ISR is higher than both the pre-empted task and all other READY tasks in the system, consequently it is not necessary to invoke the RTOS `sched-uler()` function here.

3) Then, an ISR is driven by the Live CPU Simulation Engine to execute its function. This may unblock a WAITING aperiodic task and make it READY. When the ISR finishes, it triggers another kernel handler `in-terrupt_handler_exit()` in Step 3.
4) This exit handler checks whether there are any nested IRQs. If there are, their execution will be resumed sequentially according to their priorities.
5) Finally, in the last Step 4, the exit handler sets the highest-priority READY task to run next, calls a context switch, and activates the Live CPU Simulation Engine.

Figure 4-20 illustrates the vector-based interrupt handling model. Regarding the hardware part of this model, the Interrupt Controller Model is able to obtain both the IRQ source and determine the relevant ISR from a vector table. The vector table is defined as constants in the model, indicating the mapping between IRQ numbers and ISR's task IDs. The Interrupt Controller Model also takes charge of masking lower-priority IRQs when it identifies an IRQ.

In the software part, the `interrupt_handler_enter()` function is implemented as a normal RTOS function rather than a SystemC process, because it is no longer the interrupt service entry point and so does not need to be triggered

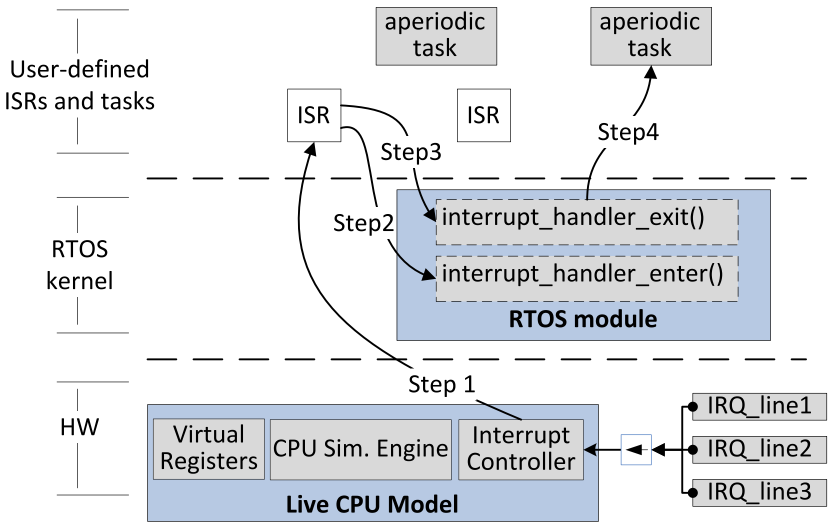


Figure 4-20. Vector-based interrupt handling model

by an external `sc_event`. The handling process includes following the functions and transition steps:

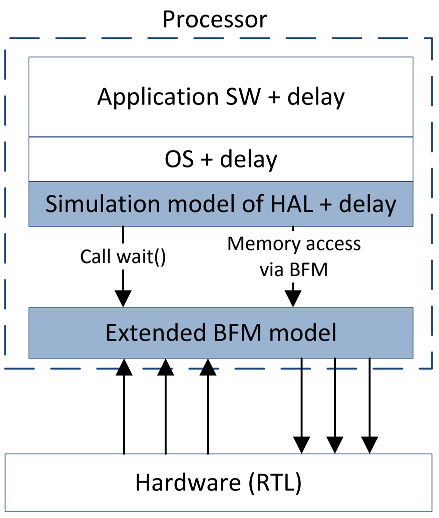
1) According to the vector table, a corresponding ISR is driven by the Live CPU Simulation Engine to execute directly (Step 1).
2) Referring to Step 2, before the ISR can carry out its main service function, it firstly calls the RTOS `interrupt_handler_enter()` function in order to pre-empt the RUNNING task. This pre-emption process is similar to that of the non-vectored model.
3) Then the ISR executes its service function and may unblock a WAITING aperiodic task. Before the ISR finishes, it calls the RTOS `inter-rupt_handler_exit()` function in Step 3.
4) In Step 4, the `interrupt_handler_exit()` function remains similar to that in the non-vectored model. After it checks possible and processes possible nested ISRs, it schedules the highest-priority READY task to execute next, calls a context switch, and activates the Live CPU Simulation Engine.

4.5.8 HAL Modelling

In Section 4.3, the concepts and functions of HAL were briefly outlined. Giving an example, researchers from the TIMA laboratory present some work on HAL modelling for native-code software simulation in SoC and MPSoC designs [153] [186] [154]. Referring to Figure 4-21, their research includes low-level implementation details of both software subsystems (e.g., assembly HAL code) and hardware subsystems (bus functional and RTL hardware models). Their simulation models apply to the later implementation phases, where HAL API functions need to be implemented for specific processors.

Compared to the detailed HAL modelling method, most conventional abstract RTOS modelling work is oriented to early system exploration phases and includes neither hardware models nor the HAL model, i.e., so-called implicit software PE modelling and inadequate interrupt handling.

Differing from the implementation-oriented HAL model and the abstract RTOS model, this thesis proposes a lightweight conceptual HAL model inside the RTOS module, which supplies some essential hardware-related functions and data structures for upper-level RTOS and application task models. These functions include both proprietary services supporting the proposed software PE simulation



(A) Timed software models and the HAL model

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

(B) HAL implementation for the context switch on the ARM7 processor

Figure 4-21. TIMA laboratory's HAL modelling work

model and conventional low-level system software primitives. This section introduces three of them, i.e., the delay information injecting service, the context switch service and the interrupt-related service. Note that transaction-based I/O communication services will be addressed in Chapter 5.

4.5.8.1 Delay Information Injecting Services

In above chapters, the `DELAY()` functions is used to make a delay annotation statement and define a time advance point. Its actual function is to inject a delay value into the Live CPU Model for a time advance. According to above definitions on annotation granularities, two main granularities of delay information are used in injecting services, namely the task level and the subtask level (including the function level, the statement segment, and the basic block level). Accordingly, two delay primitives are implemented, i.e., `write_task_delay_time ()` and `write_subtask_delay_time()`. They write different grained delay values into different Virtual Registers of Live CPU Model and activate the Live CPU Simulation Engine by releasing a `sc_event`.

Certainly, recalling the idea of separating annotations from time advance points in Section 3.2.4.2, the `DELAY_WR()` function is also implemented as two injecting primitives, i.e., `write_task_delay_time_wr ()` and `write_subtask_delay_time_wr()`. The two primitives only inject delay values but do not trigger the Live CPU Simulation Engine for a time advance. In modelling, the `DELAY_WR()` service is used more frequently than the `DELAY()` service in fact, because it can increase simulation speed by decreasing Live CPU running counts.

4.5.8.2 Context Switch Services

The context switch is an essential hardware-dependent service in an embedded software stack. In RTOS modelling, it is also valuable for modelling task switches and corresponding timing behaviour. Because of its processor-specific nature, unlike the example context-switch code of saving and restoring ARM7 processor registers in Figure 4-21, the context-switch service in this thesis maintains task time advance information (in Section 4.5.3.1) with Virtual Registers of the Live

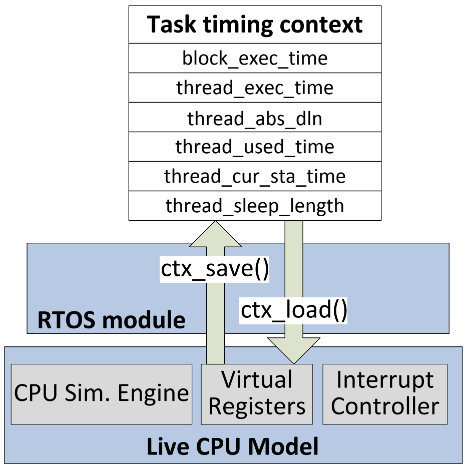


Figure 4-22. Context switch service

CPU Model (in Section 3.3.2). As shown in Figure 4-22, the service is implemented as two functions, i.e., `ctx_save()` and `ctx_load()`:

- Upon being called, the `ctx_save()` uses the values of the virtual registers to calculate how long time has elapsed since the saved task began its time advance and record the current time stamp. The updated results are utilised in a later execution of the Live CPU Simulation Engine, which has been introduced in Section 3.3.4.2. Afterwards, the `ctx_save()` saves the updated software timing context to its TCB.
- The `ctx_load()` function loads a task's timing context from its TCB into the virtual registers of the Live CPU Model.

To complete a context-switch process, the RTOS model needs to provide a method to activate the Live CPU Simulation Engine in order to let it execute software time delays. This function is implemented by releasing some appropriate `sc_events` that are included in the Live CPU module and listened to by the Live CPU Simulation Engine (in Section 3.3.4).

4.5.8.3 Interrupt Related Services

The two most important interrupt-related kernel functions (i.e., the entry handler and the exit handler) were described in Section 4.5.7.2. Also, some assistant functions are provided and detailed below:

Disabling interrupt: is an essential service in various RTOSs that protects short critical sections in kernel functions. It is implemented as a pair of functions, i.e., `enter_critical()` and `exit_critical()`. After executing the former function, all system interrupts are disabled and can be re-enabled by invoking the latter function.

Clearing an interrupt source: the `interrupt_clear()` function can be called by ISRs to clear a specific IRQ source (according to its IRQ ID number) by resetting corresponding bits of the Interrupt Controller Raw Status and Status registers in the Live CPU Model (in Section 3.3.2).

Unmasking interrupts: occurs during an ISR's execution when its equal- and lower-priority interrupts are automatically masked by the entry handler in a prioritised interrupt handling scheme. After the ISR finishes its function, it needs to call the `interrupt_unmask_equal_lower_irq()` function to unmask these affected interrupts.

4.5.9 General Modelling Methods for RTOS Services

In the above Sections 4.5.3 to 4.5.8, various functional components of the RTOS model have been described. This section concludes some general RTOS modelling ideas in the context of SystemC simulation, and addresses an untouched but important issue – modelling timing behaviour of RTOS services.

4.5.9.1 Modelling Functionality of RTOS Services

As indicated before, the presented RTOS model aims to provide services similar to those in real RTOSs, in terms of both their formation (normal C++ functions) and usage (function calls). Most services are implemented as RTOS class member methods and are called by applications task models through a pointer to their parent RTOS object. The main benefits of modelling RTOS services as normal functions are:

- It is more straightforward to input arguments and return values in a normal function, whereas a SystemC process does not easily support them.

- It is similar to real-time programming conventions and interfaces, as far as a RTOS service model can be adapted to a specific RTOS API by changing its input, output, and function if necessary.
- A normal C++ function executes much faster than a SystemC process, because it does not incur a context switch in the SystemC simulation kernel.

Regarding this point, SystemC language constructs are used as C++ constructs in modelling, and the RTOS model seems to be implemented by the C++ language in a normal OS design way. Note that normal C++ RTOS functions can execute in a SystemC simulation, but cannot represent the timing overheads of the target RTOS. This problem will be addressed in Section 4.5.9.2.

Certainly, some services and functions in the RTOS model are also implemented as SystemC processes to take advantages of the SystemC language. The selection of these is based on the following considerations:

- Some RTOS services only execute once in a predetermined cooperative order in simulation, thus it is convenient to implement them as SystemC processes and use simple *wait-for-delay* statements to advance the simulated clock. For example, the RTOS initialisation service (i.e., `SC_THREAD(rtos_init)` in model implementation) and the RTOS multi-tasking start function (i.e., `SC_THREAD(rtos_start)` in model implementation) only need to execute at RTOS startup before the beginning of pre-emptive multi-tasking execution.
- Some RTOS services are activated by other `SC_MODULEs` through static sensitivity `sc_events`; consequently they are preferred to be implemented as SystemC processes. The examples are RTOS kernel interrupt entry and exit handlers in Section 4.5.7.2.

To conclude, the internal communication methods in the RTOS model are conventional and simple in terms of real-time software programming, i.e., by function calls. The SystemC `sc_event` mechanism is mainly used for inter-module and limited SystemC process-related notifications. The Interface Method Call approach does appear inside the RTOS model, however it is used in other parts of this research: in hardware modelling (i.e., the Live CPU Model) and in inter-module communication modelling (i.e., the TLM communication model in Chapter 5).

4.5.9.2 Modelling RTOS Timing Overheads

An advantage of the proposed RTOS model, compared to some other research in the domain of abstract and generic RTOS modelling and simulation, is to consider timing overheads of various RTOS services. Building a timed simulation model for a RTOS service includes three jobs:

1) Collecting delay information;
2) Annotating this into the RTOS model;
3) Advancing the simulated target clock according to annotations.

The RTOS performance estimation methods in Section 3.2.6 have addressed the first job. Timing overheads of a RTOS product on a specific processor can be measured and collected in a corresponding ISS simulator, or can be obtained from published benchmark documents.

The second and third jobs are now considered. Because this thesis focuses on behavioural and generic RTOS modelling rather than implementation-ready ISS simulation, we observe that the implementation of a RTOS service model will not be completely functionally identical to a specific RTOS. As shown in Figure 4-23 (A), a RTOS service may invoke other RTOS internal functions and primitives.

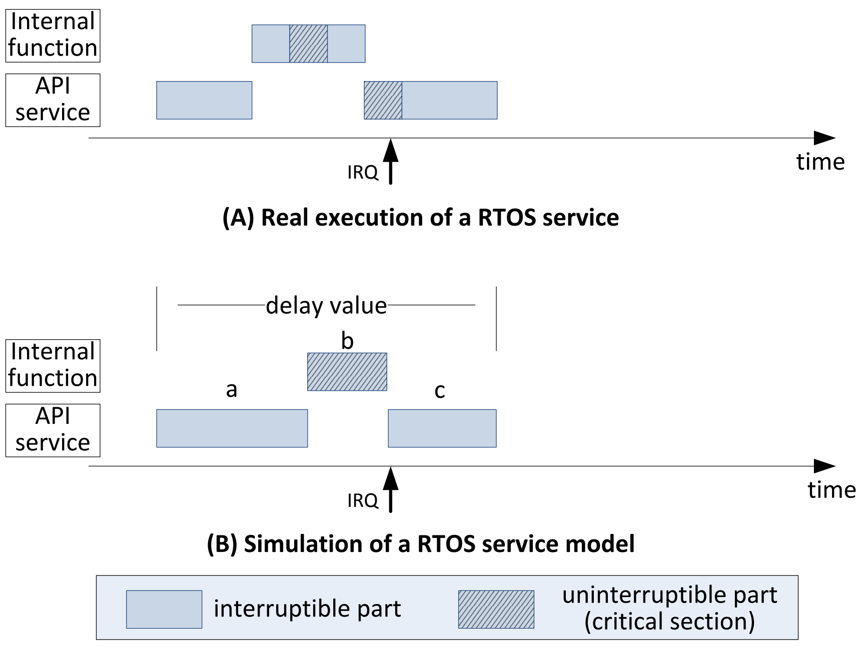

Internal function
API service
IRQ
time
(A) Real execution of a RTOS service
delay value
b
a
c
Internal function
API service
IRQ
time
(B) Simulation of a RTOS service model
interruptible part
uninterruptible part (critical section)

Figure 4-23. Unmatched RTOS service execution and simulation traces

Both RTOS services and internal functions may be fully interruptible (there is no critical section in code), fully uninterruptible (code is a critical section), or partially interruptible (with part code in one or several critical sections). Although a service in the RTOS model can generate similar results to a corresponding service in a real RTOS, the simulation trace may be quite different from the real execution trace in terms of exact included function blocks (see Figure 4-23). Hence, service-level timing annotations are sufficiently accurate for modelling RTOS service timing behaviours in this thesis. In fact, unless there is a deep enough understanding of the target RTOS code and the RTOS model is thoroughly adapted for the target RTOS, there is not an easy solution for enhancing the timing accuracy of RTOS services to a finer level.

Thus, is it possible to use the same time advance method of application tasks for RTOS services? This is not straightforward for several reasons:

1) RTOS services do not have native control blocks that can store their delay information.
2) In this thesis, RTOS services are modelled as functions rather than independent executable entities. They do not have separate SystemC process wrappers to support their execution on top of the SystemC simulation kernel.
3) Many RTOS services are re-entrant, for example, a wait-semaphore function may be invoked in several concurrent tasks and can be blocked in the middle. If a single `sc_event` object is used in a RTOS service for the *wait-for-event* time advance method, once the Live CPU Model releases this `sc_event`, then multiple execution instances of a RTOS service may be triggered at the same time. This may result in race conditions.

Within this thesis, the RTOS service time annotation and advance problems are solved by a lightweight approach after investigating common characteristics of RTOS service time annotations in the model. The service-level RTOS service annotation assumption actually means that it is difficult to implement partially interruptible time advance for a service. The RTOS model does not necessarily have target-like function blocks inside a service, nor does it support the insertion of several very accurate interruptible and un-interruptible annotations for these blocks. Thus, the RTOS service timing modelling problem is simplified, with the time advance method for RTOS services needing to cover two simulation situations:

1) Time advance in a single step, i.e., uninterruptible;
2) Time advance divided into several steps in case of interruptions, i.e., interruptible.

The approach is therefore divided into two methods, i.e., the interruptible method and the un-interruptible method.

The interruptible RTOS time advance method means that the time advance duration of a service can be interrupted and resumed later. This requires users to annotate RTOS services when they build application task models. Rather than maintaining delay information by a RTOS service itself, the delay value of a RTOS service is annotated in the calling application task. The calling task acts as an agent to progress the simulated clock for its invoking RTOS service. See Table 4-21 (A) for an example, a semaphore initialisation function executing at line 4. Subsequently, its interruptible timing overhead `SEM_INIT_FUNC_DELAY_TIME` is injected into the Live CPU Model on line 5 and then a *wait-for-event* statement is inserted on line 6 as normal.

```
#001 | void task(RTOS *rtos_i_ptr, CPU *cpu_i)
#002 | {
#003 |   ... ...
#004 |   rtos_i_ptr->sem_init(pecb0, 0, 0);
#005 |   write_subtask_delay_time(SEM_INIT_FUNC_DELAY_TIME);
#006 |   wait(event);
#007 |   ... ...
#008 | }
```

(A) Interruptible RTOS service time advance

```
#001 | unsigned RTOS::ctx_save()
#002 | {
#003 |   enter_critical();
#004 |   ... ...
#005 |   wait(CTX_SAVE_DELAY_TIME, SC_NS);
#006 |   ... ...
#007 |   exit_critical();
#008 |   ...
#009 | }
```

(B) Uninterruptible RTOS service time advance

Table 4-21. Time advance methods for RTOS services

The un-interruptible RTOS service advance method relates to RTOS critical-section services and functions during which system-wide interrupts are totally disabled. Usually, these services are internal RTOS functions, e.g., the context switch service and the scheduler service, neither of which is directly visible to user task models. Hence, their annotations need to be inserted inside the RTOS module. Since it is not necessary to worry about interruptions during the delay duration, a simple *wait-for-delay* statement is used to annotate and advance the simulated time (see Table 4-21 (B)). This method also avoids invoking the Live CPU Simulation Engine and decreases SystemC kernel engine switches. Hence, it can improve simulation speed.

We note that above methods may bring unmatched time advances for critical sections inside a RTOS service. For example, referring to Figure 4-23, a real RTOS service may include critical sections that are different from those in a RTOS service model, and these critical sections may execute at different times along the timeline. In a real execution, an IRQ happens during a critical section and may hence be ignored or delayed due to temporarily disabled system interrupts. However, in simulation, an IRQ happens at the same absolute time point, but it may be processed immediately by the system because there is not a critical section currently available. Given the previously mentioned assumption on RTOS timing overhead modelling, this limitation should be acceptable.

4.6 Evaluation Metrics

4.6.1 Simulation Performance Metrics

The simulation performance metric utilised in this chapter is similar to Section 3.4.1. A concurrent multi-tasking test program is run in both the proposed RTOS-centric software simulation models (referred to as the RTOS-centric simulator hereafter) and an ISS simulator with a comparable real RTOS product. Their host simulation times are compared to calculate a speedup.

4.6.2 Simulation Accuracy Metrics

4.6.2.1 Functional Accuracy

Since the proposed RTOS model provides a set of practical functions to support native-code real-time tasks, functional accuracy of some typical RTOS services can be represented in simulation. Both simulation traces and results can be compared to the ISS counterpart.

4.6.2.2 Timing Accuracy

Conventionally, researchers examine the timing accuracy of a behavioural simulator by running a test program and comparing with the same program executed by a more accurate standard simulator. If both simulators consume similar simulated target time (or numbers of cycles) to finish the same test program, then the timing accuracy of the behavioural simulator is believed to be enough (see Section 3.4.2.2).

In this chapter, we use a series of comparison points to evaluate the timing accuracy of the RTOS-centric simulator compared to the ISS simulator. As shown in Figure 4-24, the method used is to record values of the simulated target clock at more observation points along the simulation timeline, instead of only measuring the final accumulative number. These same observation points are also used in the

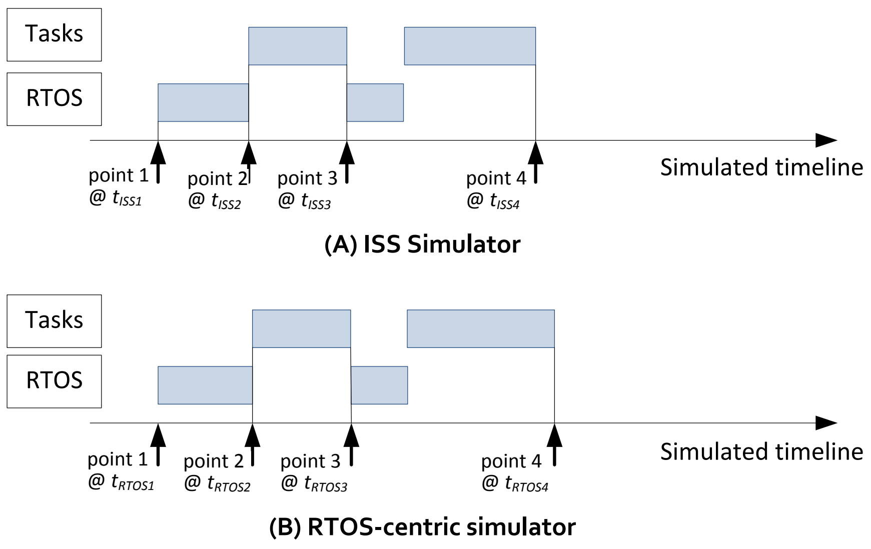


Figure 4-24. Evaluating the timing accuracy by comparing traces

ISS simulator. These observation points are chosen as important state transition points in concurrent multi-tasking execution, e.g., task switching points, RTOS service invoking points, task completion points, etc.

4.7 Experimental Results

4.7.1 Multi-Tasking Simulation with μC/OS-II RTOS

In order to demonstrate simulation performance, functional and timing accuracy of the RTOS-centric simulator, a multi-tasking A/D (Analogue-to-Digital) data collection program and the μC/OS-II RTOS are used as the modelling and simulation target. As shown in Figure 4-25, three tasks take charge of watching the keyboard (i.e., polling the I/O port), collecting A/D data (i.e., reading the A/D converter) and sending out results via the serial port. Tasks have periods of 90 ms, 100 ms and 510 ms. According to the RM algorithm, they are allocated descending priorities. The RTOS model implements fixed-priority pre-emptive scheduling and is time driven. A tick timer ISR is associated with a real-time clock IRQ to drive tick scheduling with a 5 ms tick length. A semaphore and a message queue provide synchronisation and communication services between tasks. The RTOS sleeping service is also used by tasks.

The same program and the μC/OS-II RTOS are executed in the KEIL ARM ISS, which is used as the cycle-accurate reference model in the experiment. The target processor is configured as a 48MHz NXP LPC2378 processor without using cache. Timing overheads of the RTOS model and application tasks are measured based on the μC/OS-II RTOS in this ISS and annotated at the function level

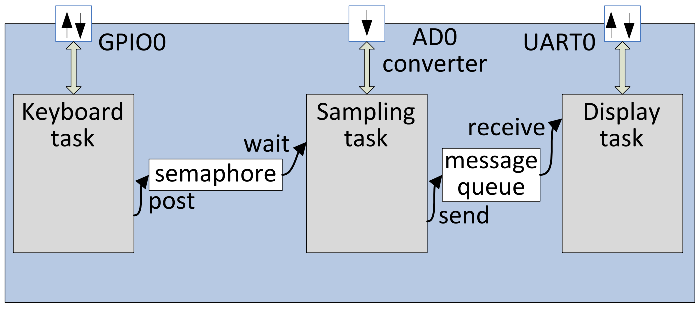
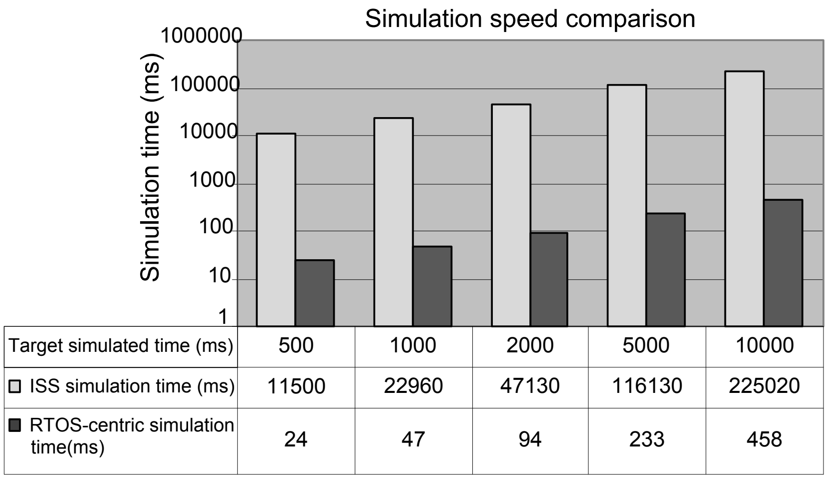


Figure 4-25. Experiment setup



Target simulated time (ms)	500	1000	2000	5000	10000
□ ISS simulation time (ms)	11500	22960	47130	116130	225020
■ RTOS-centric simulation time(ms)	24	47	94	233	458

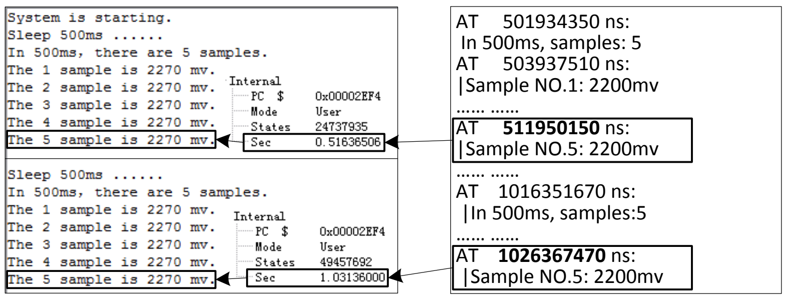
Figure 4-26. Simulation speed comparison

and segment level respectively. All tests are executed on an x86 PC at 1.86GHz.

In order to compare the speed of RTOS-centric simulator with the standard ISS simulation, we let each simulator simulate for 500 ms, 1000 ms, 2000 ms, 5000 ms and 10000 ms target time. During the longest 10000 ms simulation, the three tasks can repeat about 110, 100, and 19 iterations respectively.

Not surprisingly, as a behavioural software simulator, the RTOS-centric simulator is much faster speed than the ISS simulation. Figure 4-26 reveals the simulation performance of RTOS-centric simulation: it is nearly 500 times faster than the ISS simulator.

Regarding functional accuracy, the RTOS-centric simulator generates simulation sequences and results at the right time compared with real execution. In the experiment, we input same stimuli, i.e., keyboard signals and voltages (dummy values), into both μVision ARM ISS and the RTOS-centric simulator. We observe A/D converting results, which are generated after various multi-tasking interactions between application tasks and the RTOS. Figure 4-27 (A) shows the functional results generated at two time points in the ISS simulator. Note that the time is displayed with the unit of second. Figure 4-27 (B) shows part of the trace file of the RTOS-centric simulator. It can be observed that the RTOS-centric simulator produces similar functional results at very close time points to the ISS simulator, which demonstrates its functional correctness.



```
System is starting.
Sleep 500ms ......
In 500ms, there are 5 samples.
The 1 sample is 2270 mv.
The 2 sample is 2270 mv.     Internal
The 3 sample is 2270 mv.       PC  $      0x00002EF4
The 4 sample is 2270 mv.       Mode       User
                               States     24737935
The 5 sample is 2270 mv.       Sec        0.51636506

Sleep 500ms ......
In 500ms, there are 5 samples.
The 1 sample is 2270 mv.     Internal
The 2 sample is 2270 mv.       PC  $      0x00002EF4
The 3 sample is 2270 mv.       Mode       User
The 4 sample is 2270 mv.       States     49457692
The 5 sample is 2270 mv.       Sec        1.03136000
```

```
AT    501934350 ns:
 In 500ms, samples: 5
AT    503937510 ns:
|Sample NO.1: 2200mv
...... ......
AT    511950150 ns:
|Sample NO.5: 2200mv
...... ......
AT   1016351670 ns:
 |In 500ms, samples:5
...... ......
AT   1026367470 ns:
 |Sample NO.5: 2200mv
```

(A) KEIL ISS simulator output **(B) RTOS-centric simulator output**

Figure 4-27. Simulation output comparison

According to the method introduced in Section 4.6.2.2, Figure 4-28 shows timing accuracy comparison between the ISS simulator and the RTOS-centric simulator. The X-axis is 22 observation points (e.g., task switching points or RTOS service entry points) in simulation flows and the Y-axis is the simulated target time of each observation point, which ranges from 0 to 600 ms, i.e., including a full operation cycle of the system. In the figure, two simulator flows' curves are in close accordance, which reveals the good accuracy intuitively.

Table 4-22 shows the timing accuracy losses of the RTOS-centric simulation compared with the ISS simulation at these 22 comparison points. Results in the table show that accuracy losses of the RTOS-centric simulator are marginal in this experiment, i.e., 14 out of 22 points are less than 0.7% and all are less than 4.5%.

Referring to Table 4-22, note that there are some sudden changes of the timing accuracy in the RTOS-centric simulation timeline, where the accuracy loss

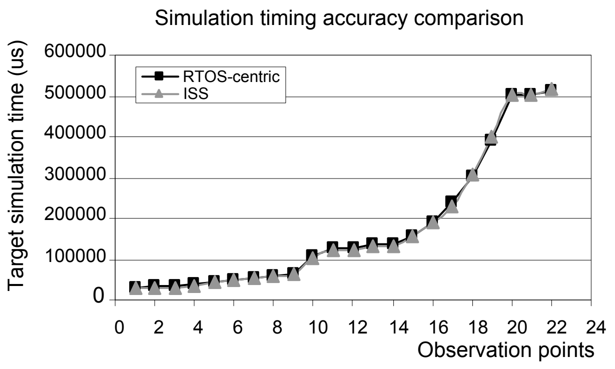


Figure 4-28. Simulation timing accuracy comparison

Comparison point	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**	**#13**	**#14**	**#15**	**#16**	**#17**	**#18**	**#19**	**#20**	**#21**	**#22**
Accuracy loss	0.34%	0.34%	0.34%	0.68%	0.64%	0.61%	0.59%	0.57%	0.56%	4.44%	3.76%	3.76%	3.50%	3.51%	0.13%	0.03%	2.20%	1.49%	2.52%	0.08%	0.09%	0.22%

Table 4-22. Accuracy loss of the RTOS-centric simulation compared with ISS

abruptly spikes or decreases within a certain degree. This phenomenon can be discussed twofold. Firstly, the generic RTOS model is not implemented the same as the real μC/OS-II RTOS in terms of its internal functions and associated timing overheads. Hence, there are differences regarding timing behaviours of various RTOS services in our simulation to ISS simulation. This inevitable inaccuracy can both contribute the timing accuracy loss and unintentionally remedy the accumulated loss. Secondly, application tasks are annotated with segment level timing costs, which also have inherent inaccuracy compared to ISS simulation. The consequence is similar to the RTOS aspect as well.

4.7.2 Interrupt Simulation with RTX RTOS

In order to demonstrate the interrupt modelling and simulation capability of the RTOS-centric simulator, we carry out an interrupt handling example with the RTX RTOS on the KEIL ARM ISS, and use the RTOS-centric simulator to simulate the same program.

As shown in Figure 4-29, this experiment includes an ISR `ext0_int()`, an aperiodic task `isr_task()`, and a periodic task `counter_task()`. The `counter_task()` increments an internal counter by 1 in every 1 ms. The ISR is associated to the ARM external interrupt 0 and can trigger `isr_task()` by a semaphore, which can then pre-empt `counter_task()` because of its higher priority. In this experiment, it is expected to observe that: firstly, the ARM EINT0

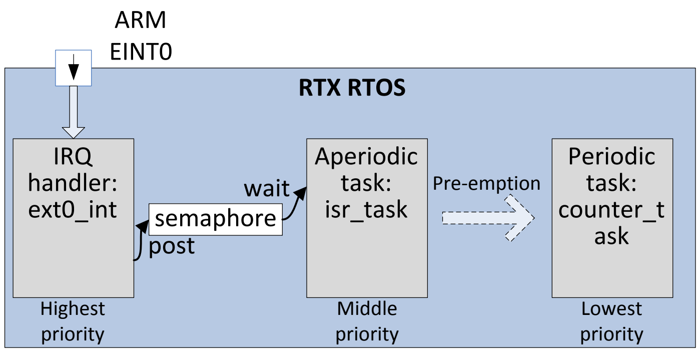


Figure 4-29. Interrupt handling experiment

IRQ can be handled immediately once happening; secondly, the three involved ISR and tasks can coordinate correctly in terms of both functionality and timing.

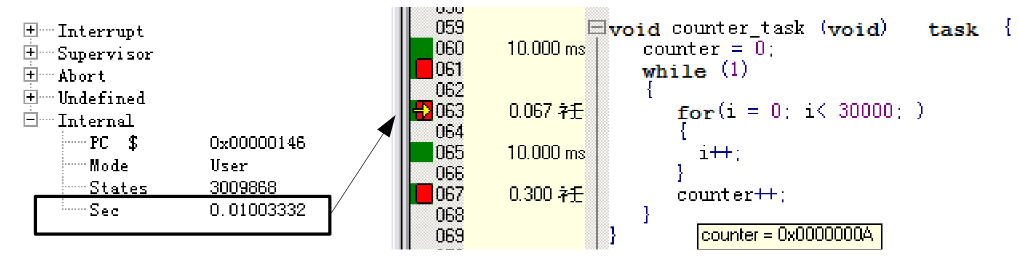
The KEIL ARM ISS simulates a 60MHz LPC2129 processor with the vectored interrupt controller. The RTOS-centric simulator is also configured with this vector-based interrupt handling mode. The task models and RTOS model are annotated with timing costs that are measured from the ISS simulator at the segment level and function level respectively.

Firstly, we run ISS and RTOS-centric simulators for 100 ms target time and repeat 10 times in order to compare their simulation performance. The results are shown in Table 4-23. Not surprisingly, the RTOS-centric simulator achieves a considerable speedup compared to the ISS simulator.

Secondly, we compare interrupt handling processes in the ISS simulator and the RTOS-centric simulator. We raise the ARM external interrupt at (almost) same target time points in both simulators (i.e., at 0.01003332 s in ISS and 0.10033290 s in RTOS-centric simulator), when the task `counter_task()` is currently executing. ISS and RTOS-centric simulation outputs are shown in Figure 4-30 and Table 4-24, respectively. The two figures show that a series of events,

	Average simulation time (µs)	Speedup
ISS	14174000	
RTOS-centric simulator	16425.88	862.9066

Table 4-23. Simulation speed comparison

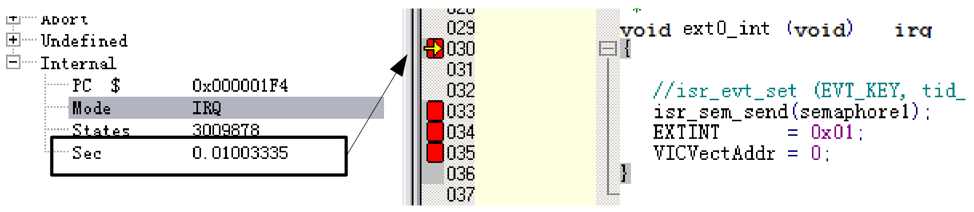


```
Interrupt
Supervisor
Abort
Undefined
Internal
    PC  $        0x00000146
    Mode         User
    States       3009868
    Sec          0.01003332
```

```
void counter_task (void)   task  {
  counter = 0;
  while (1)
  {
    for(i = 0; i< 30000; )
    {
      i++;
    }
    counter++;
  }
}
```

counter = 0x0000000A

(A) Before the IRQ event, counter_task() is executing

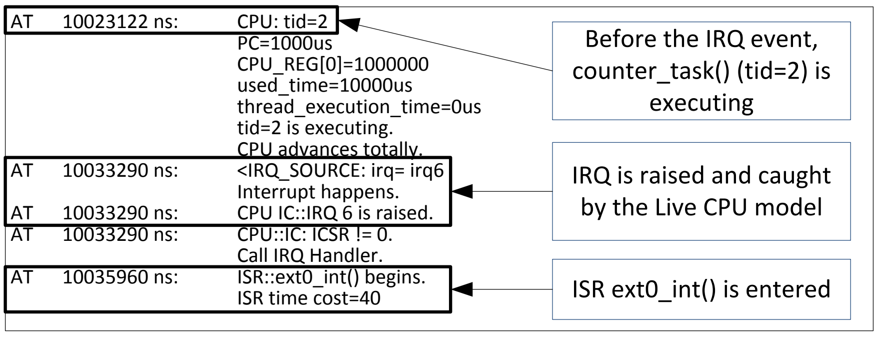


```
Abort
Undefined
Internal
    PC  $        0x000001F4
    Mode         IRQ
    States       3009878
    Sec          0.01003335
```

```
void ext0_int (void)   irq
{
  //isr_evt_set (EVT_KEY, tid_
  isr_sem_send(semaphore1);
  EXTINT       = 0x01;
  VICVectAddr  = 0;
}
```

(B) After the IRQ event, ISR ext0_int() is entered

Figure 4-30. RTX interrupt handling in the ISS



```
AT    10023122 ns:     CPU: tid=2
                       PC=1000us
                       CPU_REG[0]=1000000
                       used_time=10000us
                       thread_execution_time=0us
                       tid=2 is executing.
                       CPU advances totally.
AT    10033290 ns:     <IRQ_SOURCE: irq= irq6
                       Interrupt happens.
AT    10033290 ns:     CPU IC::IRQ 6 is raised.
AT    10033290 ns:     CPU::IC: ICSR != 0.
                       Call IRQ Handler.
AT    10035960 ns:     ISR::ext0_int() begins.
                       ISR time cost=40
```

Before the IRQ event, counter_task() (tid=2) is executing

IRQ is raised and caught by the Live CPU model

ISR ext0_int() is entered

Table 4-24. Interrupt handling in the RTOS-centric simulator

i.e., interrupt raise, CPU catch, task pre-emption, and ISR entry. They are executed and simulated in the same order in both simulators. This means that our RTOS-centric simulator can model the realistic interrupt handling method in RTX RTOS.

Thirdly, in order to evaluate the timing accuracy of our RTOS-centric simulation in this experiment, we still use the "observation points" method introduced in Section 4.6.2.2. The result is shown in Figure 4-31. The X-axis is 18 observation points in simulation flows, which represent entries of RTOS services and task job completions. The Y-axis is the simulated target clock time of each observation point. It ranges from 0 to 14 ms that includes a full interrupt cycle of the system.

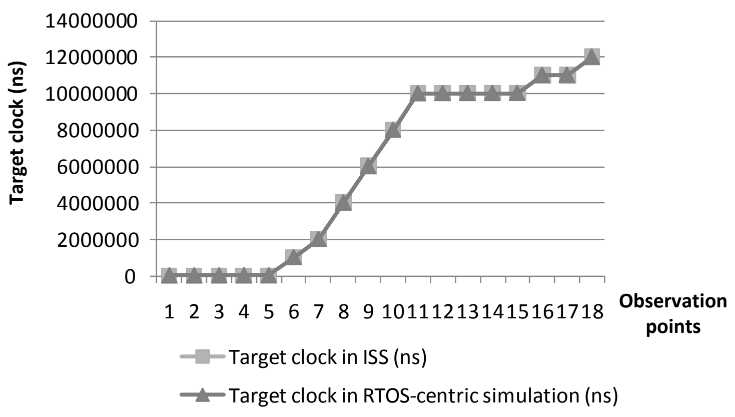


Figure 4-31. Simulation timing accuracy comparison

In the figure, two simulation curves are coincident, showing the good timing accuracy of our simulator. As shown in Table 4-25, the calculated timing accuracy losses are marginal regarding these 18 observation points in this experiment. This is mainly due to our carefully fine tuning of the RTOS simulator and relatively simple functions of the test program.

Comparison point	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**
Accuracy loss	0.00%	0.00%	4.28%	3.96%	0.59%	0.02%

Comparison point	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**
Accuracy loss	0.10%	0.09%	0.09%	0.09%	0.10%	0.00%

Comparison point	**#13**	**#14**	**#15**	**#16**	**#17**	**#18**
Accuracy loss	0.03%	0.03%	0.03%	0.03%	0.04%	0.06%

Table 4-25. Timing accuracy losses

4.8 Summary

This chapter has presented a generic RTOS-centric real-time embedded software simulation model. It allows modelling and simulating application tasks, the RTOS, and the CPU processing element in a unified SystemC-based framework. It can help designers to evaluate both functional and timing effects of the projected real-time embedded software design fast and early.

It can flexibly model application tasks by supporting hybrid abstract software models and delay-annotated native-code application task models. It improves the functionality of the RTOS model by providing various generic and practical services selected from common RTOS standards and products. It achieves reasonably accurate simulation, in terms of both functional and timing accuracy, by modelling RTOS services as their normal structures and formations and considering timing overheads of various RTOS services. The underlying Live CPU Model also enables RTOS-centric software models with interruptible time advance.

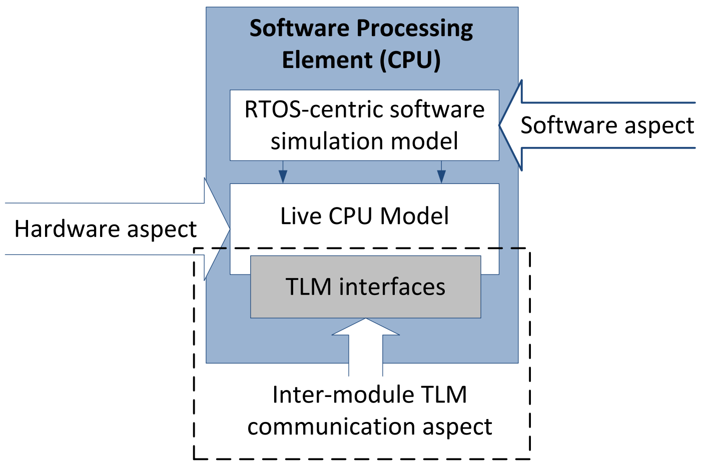
Experiments show the fast performance, sufficient function, and marginal timing accuracy loss of the RTOS-centric simulation approach compared to cycle-accurate ISS simulation of two real RTOS products. The reasons are mainly three-fold: firstly, as introduced in the chapter, the RTOS simulation model's structure is elaborate and its functional and timing behaviours are carefully modelled; secondly, the RTOS simulation model is adapted to model the two RTOSs; thirdly, delay information of both applications and RTOS services is measured on the ISS before being used in RTOS-centric simulation.

Chapter 5

Extending the Software PE Model with TLM Communication Interfaces

In a real embedded system, the software subsystem runs on top of a CPU subsystem. These software and hardware subsystems collectively constitute a software PE model. Previous chapters have investigated behavioural modelling and simulating real-time software and RTOS in the context of a software PE model, as shown in Figure 5-1.

In the SystemC-based high-level software modelling and simulation approach, the hardware aspect of the software PE model is abstracted and encapsulated into a Live CPU Model (in Section 3.3). It provides abstract yet essential hardware control functions (e.g., interrupt controller, virtual register, and real-time clock,



Software Processing Element (CPU)
RTOS-centric software simulation model
Software aspect
Hardware aspect
Live CPU Model
TLM interfaces
Inter-module TLM communication aspect

Figure 5-1. TLM communication interface of the software PE model

etc.) to upper level software. Especially, it supports interruptible SystemC-based software timed simulation through the Live CPU Simulation Engine. In Chapter 4, the RTOS-centric real-time software simulation model is described as the software aspect of the software PE model. It can supply various practical and flexible RTOS services in order to support abstract and native-code real-time application task models.

Within software modelling and simulation research, transaction-level modelling has frequently been considered. TLM is a promising system-level modelling paradigm to improve productivity in the design of integrated embedded systems, e.g., SoC. TLM models are expected to serve as interoperable references across different design teams with different aims such as fast embedded systems architecture exploration, functional verification, and as well as the interest of this thesis - early embedded software modelling and simulation. SystemC is the research tool of this thesis and also the most popular SLDL in TLM design area today [3]. Based on the essential TLM principle "separating computation from communication", TLM research can be divided into two aspects: the computation aspect and the communication aspect [7]. In this thesis, the proposed software models reside in the domain of TLM software computation aspect. In Section 3.2.2, some software TLM software computation models have been defined with inspiration from the OSCI TLM-2.0 standard [88], which is the official SystemC TLM communication modelling standard.

For the aim of extending software simulation models to the wider and encouraging TLM communication modelling world, this chapter considers the integration of existing TLM communication interfaces in the software PE model. These added interfaces and structures support SW-to-HW and HW-to-HW inter-module communication modelling with existing software models. OSCI TLM-2.0 standard interfaces are selected due to their popularity. As depicted in Figure 5-1, this TLM interface modelling work can be seen as an add-on module in terms of the whole software PE model. By this means, the software PE model can be integrated in an abstract TLM embedded system model that includes the CPU, memory, bus, and peripheral devices, which will improve functionality and extend this research. Note that this chapter does not aim to propose any new or complex TLM communication modelling methods, because the scope of this thesis is on software modelling and simulation.

5.1 Integrating OSCI TLM-2.0 Interfaces

5.1.1 The OSCI TLM-2.0 Standard

The OSCI TLM-2.0 standard consists of two aspects: coding styles defining abstract communication models and standard interfaces implementing these models. The former concepts mainly refer to LT and AT abstract models (see Sections 2.1.1.1 and 3.2.2), and the latter are to be introduced and used in next sections.

The TLM-2.0 standard defines various aspects about transaction-based communication modelling, e.g., transport interfaces, sockets, temporal decoupling methods, communication protocols, utilities, etc. However, currently, only parts of them are related to our research as follow (refer to Figure 5-2):

- Transactions: include information to be exchanged between modules and are passed by references.
- Transport interfaces: two main interfaces are utilised in research, i.e., the blocking transport interface and the non-blocking transport interface (in Sections 2.1.1.1 and 2.1.2). In the standards, they are affiliated to sockets and called by software tasks in order to transfer transactions between communicator modules.
- Sockets: there are two types of sockets, i.e., initiator sockets and target sockets. An initiator socket contains a SystemC `sc_port` for sending out transactions (so-called the forward path) by its associated interface method calls and a SystemC `sc_export` for receiving returned transactions (so-called the backward path). A target socket is oppositely defined, in which the `sc_port` is used in the backward path and the `sc_export` is used for the forward path.
- Communicator modules: are classified into three basic types, i.e., initiator modules, target modules, and interconnect modules. An initiator module (e.g., a processor) can create new transaction objects and initiate communication by calling an interface method of its included initiator socket(s). A

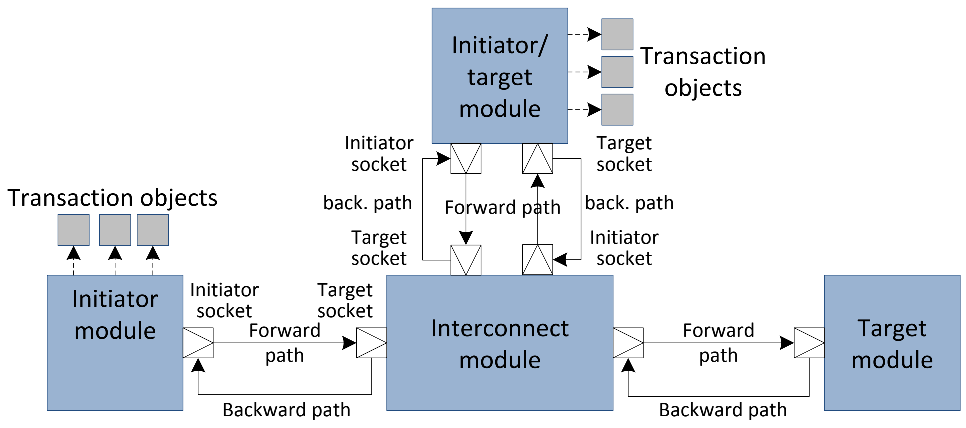


Figure 5-2. OSCI TLM-2.0 essentials

target module (e.g., a memory) is the final destination of transactions and includes at least one target sockets. Note that a module can act as both an initiator module and a target module by including both sockets. An interconnect module (e.g., a router) transmits transactions but it does not initiate a transaction or become the final destination.

- TLM communication protocols: the *generic payload* is recommended by the OSCI TLM library to achieve the interoperability of memory-mapped bus models. It provides typical characteristics of memory-mapped bus protocols, for instance command, address, data, single word transfer, and burst transfer, etc.

5.1.2 TLM Constructs in the Software PE Model

According to above definitions, the software PE model naturally is an initiator module. An important problem is: where are initiator functions and sockets placed in the model?

As explained in Section 4.5.1, the concise and extensible software PE model is constituted by three types of modules, i.e., application tasks modules, the RTOS module, and the Live CPU Model module. Software modules run on top of the Live CPU Model and utilise its conceptual computing resources. Consequently, as the root model, the Live CPU Model is the most suitable module to implement initiator's TLM communication sockets and interfaces. This is also a straightforward choice, since the CPU controls software communications with other hard-

```
class TLM_LT_COMPONENT : public SimpleLTInitiator1
{
  int LT_write(unsigned uiId, unsigned uiData);
  int LT_read(unsigned uiId);
  .......
};
class TLM_AT_COMPONENT : public SimpleATInitiator1
{
  int AT_write(unsigned int uiId, unsigned int uiData);
  int AT_read(unsigned int uiId);
  .......
};
```

(A) LT and AT TLM communication components

```
SC_MODULE(CPU)
{
  TLM_LT_COMPONENT::initiator_socket_type PE_LT_socket1;
  TLM_LT_COMPONENT *LT_initiator1;

  tlm::tlm_initiator_socket<32> PE_AT_socket1;
  TLM_AT_COMPONENT *AT_initiator1;
  .......
};
```

(B) TLM components in the Live CPU Model

```
SC_MODULE(RTOS)
{
  int rtos_LT_write(unsigned targetID, unsigned data);
  int rtos_AT_write(unsigned targetID, unsigned data);
  .......
};
```

(C) TLM communication primitives in the RTOS HAL Model

```
void task_write(RTOS *rtos_i_ptr, CPU *cpu_i)
{
  rtos_i_ptr->rtos_LT_write(0, array[i]);
  rtos_i_ptr->rtos_AT_write(0, array[i]);
  ...
}
```

(D) An application task uses TLM functions

Table 5-1. TLM implementation in the software PE model

ware components in a real design. Besides, because HAL services are included in the RTOS module (see Section 4.5.8), the RTOS model needs to provide TLM APIs for application tasks.

In order to support both LT-style blocking and AT-style non-blocking communication, two initiator classes are derived from the *simple socket* interfaces of the OSCI TLM-2.0 library. Referring to Table 5-1 (A), they both supply simple *write* and *read* functions. In the current model, a *write* function needs two arguments for a transport, i.e., a target ID indicating the destination module and a datum to be transferred; whereas a *read* function only needs a target ID argument and the obtained datum is returned by the function. Exact addresses of transports are maintained by these interfaces internally. The model also can support user-defined addresses in future refinement.

Based on above interface classes, an LT initiator component is instantiated and bound to a LT socket inside the Live CPU Model, and so is an AT initiator component and an AT socket (see Figure 5-3 and Table 5-1 (B)). Afterwards, in Table 5-1 (C), some RTOS HAL services wrap the TLM interfaces provided by the Live CPU Model. Finally, as shown in Table 5-1 (D), an application task can invoke RTOS communication services so as to transfer some data to a target.

5.1.3 The TLM System-on-Chip Model

Low-level hardware architecture modelling and complex communication exploration are out of the scope of this thesis. Consequently, for simplicity and gen-

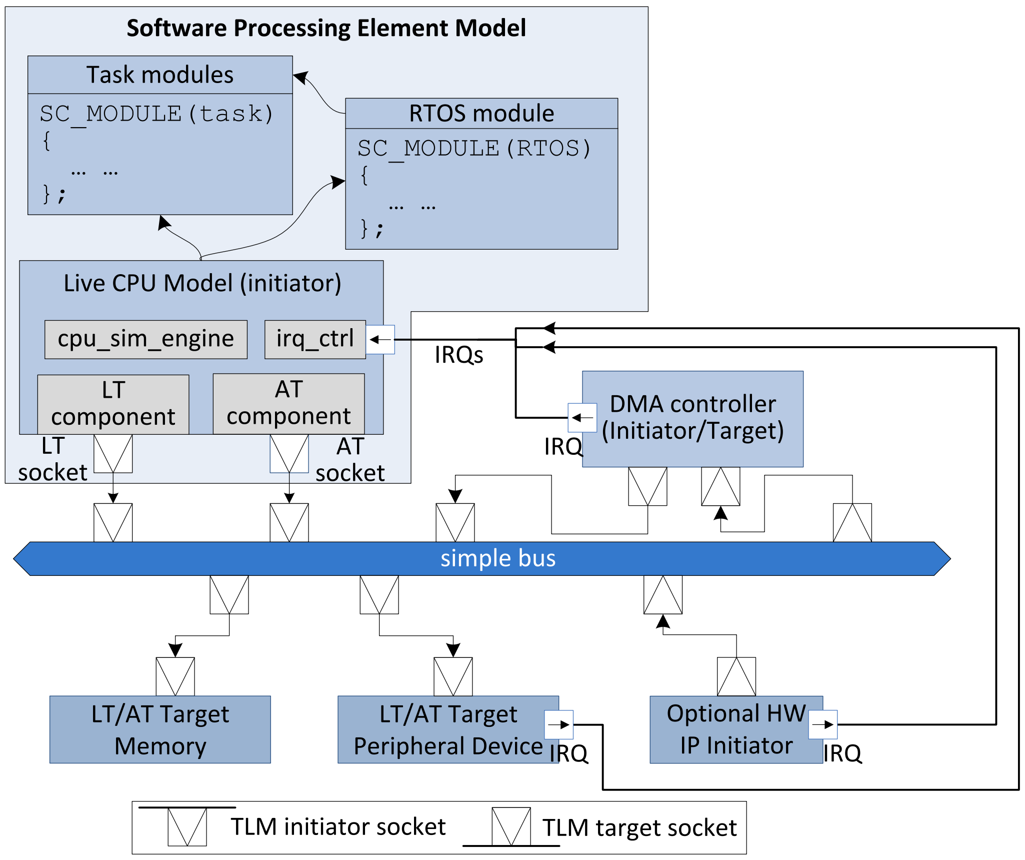


Figure 5-3. Combining software PE model with TLM interfaces and SoC models

erality, as shown in Figure 5-3 , a simple SoC topology is presented as the referenced model to extend the software PE for TLM modelling. It includes following modules.

5.1.3.1 Initiator Modules

The Live CPU Model is the main software PE initiator. In addition, an optional hardware IP module can be integrated as another initiator for customised hardware computation. It supports both LT and AT interfaces like the software PE model (See Table 5-1 (A)). This HW IP module can also be connected to the Interrupt Controller Model in the Live CPU Model by a standard SystemC primitive channel, in order to trigger a software interrupt hander in case of an interrupt event.

5.1.3.2 Target Modules

Target modules could be memory components or peripheral devices. In case there is more than one memory component, this topology can represent an embedded system with application data partitioning. The small-size memory module has a fast access speed whilst the big-size memory module is slow. Both LT and AT styles can be applied to their target interfaces and sockets, which are implemented by inheriting `SimpleLTTarget1` and `SimpleATTarget1` classes in the OSCI TLM-2.0 library (see Table 5-2). Their class member methods can receive, store, and process transactions depending on modelling configuration.

```
class SimpleLTTarget1 :    public sc_core::sc_module,
                           public virtual tlm::tlm_fw_transport_if<>
{
  target_socket_type socket;
  void b_transport(transaction_type& trans, sc_core::sc_time &t);
  .......
};
class SimpleATTarget1 : public sc_core::sc_module
{
  target_socket_type socket;
  void endRequest();
  void beginResponse();
  void endResponse();
  .......
};
```

Table 5-2. LT and AT targets

5.1.3.3 Combined Initiator/Target Module

The Direct Memory Access (DMA) controller is an example combined initiator/target module in the proposed SoC model. It allows directly moving data between memory locations and devices without intensive handling from the Live CPU Model. It plays roles as both an initiator (for reading and writing data) and a target (being programmed by a DMA requester). The structure of the DMA controller is illustrated in Figure 5-4.

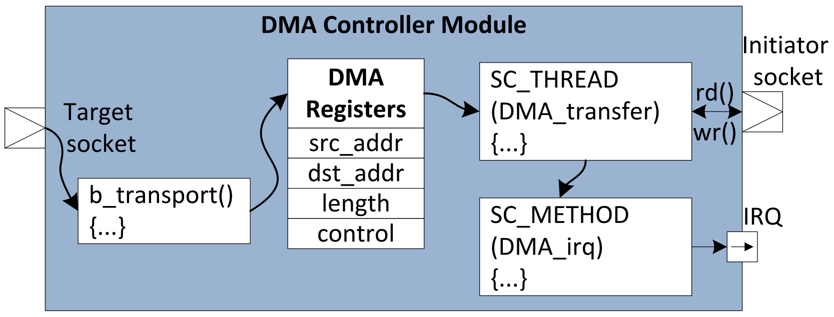


Figure 5-4. The DMA controller model

Referring to Table 5-3, two sockets, three main methods, and four virtual registers implement a model of a typical DMA mechanism. The `b_transport()` method, which is inherited from the standard TLM LT target interface, listens to

```
class DMA: public sc_core::sc_module,
        public virtual tlm::tlm_fw_transport_if<>, //From target
        public virtual tlm::tlm_bw_transport_if<>  //From initiator
{
  initiator_socket_type    DMA_initiator_socket;
  target_socket_type       DMA_target_socket;

  // DMA blocking transport interface
  void b_transport(transaction_type& trans, sc_core::sc_time &t);
  void DMA_transfer();     // DMA transfer management process
  void DMA_irq();          // DMA transfer IRQ management process

  // DMA registers
  unsigned int m_dma_src_addr;      // Source address register
  unsigned int m_dma_dst_addr;      // Destination address register
  unsigned int m_dma_length;        // Length register
  unsigned int m_dma_control;       // Control register
  .......
};
```

Table 5-3. Implementation of the DMA controller

the target socket and waits for configuration information from requestors. Upon receipt, the configuration information (i.e., source address, destination address, size of transfer, and control bits) is saved in virtual registers of the DMA controller. Then the `DMA_transfer()` function begins to read data from source locations and then writes them to destinations. When an entire DMA transfer is finished, the `DMA_irq()` method will interrupt the Live CPU Model.

5.1.3.4 Interconnection

The OSCI TLM-2.0 *simple_bus* router model is selected as the memory-mapped interconnection bus. It is implemented in the AT coding style but supports a combination of LT and AT initiators and targets. A number of target and initiator sockets can be defined by the user to connect initiators and targets. It provides a FIFO bus arbitration scheme. This bus model can be used in architectural exploration and early software development.

5.1.3.5 Communication Protocol

The aforementioned OSCI TLM-2.0 *generic payload* is directly utilised to support these memory-mapped bus models.

5.2 Experiments

In this section, some case studies are presented in order to demonstrate the performance and capability of the integrated software PE model and TLM communication models. They are based on the above introduced TLM SoC model. All experiments run on a 1.86GHz x86 PC.

5.2.1 Performance Study of TLM Models

This experiment investigates simulation performance of combined software and TLM models. The basic benchmark model consists of the software PE model as an initiator (including two software tasks and an optional RTOS model), two optional memory modules as targets (one is LT and the other is AT), and the *simple_bus* model as the interconnection. One software task implements an *insert-sort* algorithm to process an array and then writes the result to a memory module through the TLM bus, whilst another software task reads data from that memory module. The RTOS model mimics the μC/OS-II RTOS and is annotated with relevant executing overheads on a 48MHz ARM7 processor.

The benchmark model is configured and run in six scenarios as follows:

1) Scenario 1 (Pure SystemC + LT TLM): This is an original SystemC TLM model without the RTOS model. The SystemC native kernel scheduler provides co-operative scheduling without prioritisation and pre-emption. Software tasks are implemented as SystemC `SC_THREADs`. Coarse-grained time annotation and the LT style are used for software and TLM communication models, respectively. This case can represent behaviour of an original SystemC TLM simulation.
2) Scenario 2 (Pure SystemC + AT TLM): The only difference from Scenario 1 is that AT TLM communication is used in this case. It supports more timing phases in a transaction than the LT TLM model.
3) Scenario 3 (Abstract SW + LT TLM): The software PE model (including the Live CPU Model, RTOS model, and task models) and the LT style TLM model are integrated in this case. Two coarse-grained timed abstract software tasks are controlled by the RTOS model and utilise RTOS synchronisation and timing services.
4) Scenario 4 (Abstract SW + AT TLM): Being different from Scenario 3, the AT TLM communication method is used in this case.
5) Scenario 5 (Native-code SW + LT TLM): In this case, software tasks are annotated with fine-grained time delays, whose number is about 1000 times more than the abstract model. Other properties are the same as Scenario 2.
6) Scenario 6 (Native-code SW + AT TLM): This case includes both fine-grained timed software model and the AT TLM communication model.

The model of each scenario is executed ten times so as to obtain an average result. In each run, a thread repeats about ten jobs, with two thousand transactions being transferred on the bus.

The simulation results are shown in Figure 5-5. Not surprisingly, pure functional SystemC models achieve the fastest simulation speed due to simplicity. As expected, native-code software models and AT TLM models always give a worse

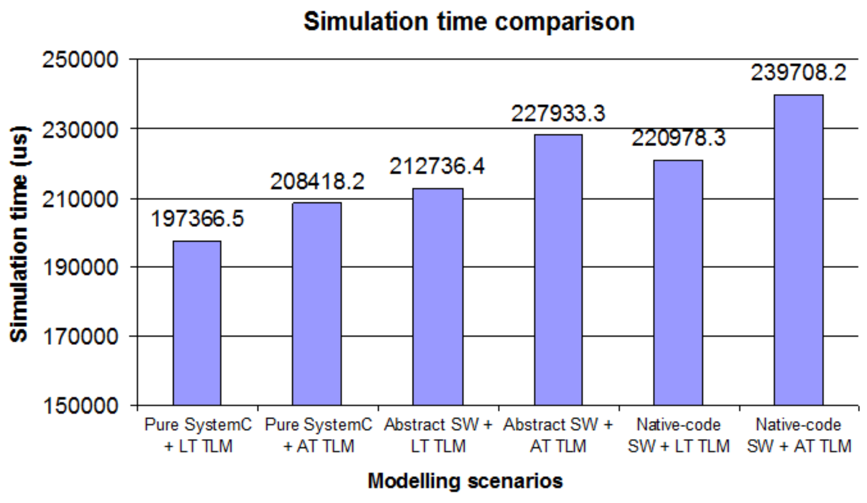


Figure 5-5. Simulation performance results

performance than corresponding abstract and LT TLM models. However, when different levels of software models and TLM models are mixed, it is not a straightforward process to predict the behaviour of their simulation speed. This is because either annotation statements in software models or TLM transaction phases may become the dominant factor in the simulation performance.

Besides, it is more interesting to us that the proposed RTOS-included software PE model incurs about 10% timing overheads more than native SystemC models (compare simulation times of the comparable scenario pair 1, 3 and pair 2, 4, because they all have coarse-grained timing annotations), given that it provides basic real-time software services and interruptible software timed simulation in this experiment. It is noticed that, in a similar-purpose study [130], introducing an OS simulation model to functional level SystemC models incurs about a 15% speed overhead, which is at the same order of magnitude as this research.

5.2.2 DMA-Based I/O Simulation

DMA is an essential component to reduce CPU Input/Output (I/O) workload in modern computers. It can especially improve CPU performance in some System-on-Chips applications where I/O functions have a high data bandwidth. In a typical ARM-based SoC, I/O functions are implemented by a combination of memory-mapped addressable peripheral registers and interrupt inputs [187]. The proposed models in this thesis have good enough capability (i.e., DMA enabled memory transfer and full-functional interrupt handling) to model the two mechanisms.

This experiment implements the RSA cryptography algorithm in the software initiator and uses DMA to transfer encrypted and decrypted messages across a memory target module and a peripheral target device. Specifically, this experiment includes following modules and components:

- The software PE initiator module includes two software tasks and the RTOS model. Hereinto, the `task_encypt()` task encrypts randomly-generated messages and saves them in a memory module; the other `task_dma_transfer()` task invokes the DMA controller to transfer ciphered messages and secret keys from memory locations to a hardware decipherer device. The two tasks are synchronised by a semaphore.
- A memory model serves as a target module and is accesses by software PE and the DMA module.
- A hardware peripheral device `RSA_IP` is a target module and acts as a decipherer.
- The DMA model is a combined initiator/target. It raises an interrupt to the Live CPU Model when it finishes transport.
- The *simple_bus* interconnection.

Various modelling characteristics of the software PE model and the simple SoC model such as software processing, DMA transfers, and CPU interruption are included in this experiment. It is expected to observe successfully recovered messages after several transfers across the TLM simple bus and low frequency of I/O related interrupts by using the DMA method.

Figure 5-6 shows some parts of the SystemC simulation trace of this experiment. They are organised in five sequential blocks in order to illustrate a working cycle of this experiment. In the 1st block, the task `task_encypt()` encrypts original messages, saves them in a target memory module, and then unblocks the task `task_dma_transfer()`. As shown in the 2nd block, the second task then uses TLM primitives (i.e., the CPU TLM interfaces) to program the DMA module to initiate transfers. After transactions are transferred, the hardware peripheral de-

```
//SW task: task_encrypt_data() encrypts:
AT  1472170 ns:   |Message to be ciphered: 9614
                  |Ciphered message: 3307
AT  2272170 ns:   |Message to be ciphered: 1454
                  |Ciphered message: 35894
AT  3072170 ns:   |Message to be ciphered: 5878
                  |Ciphered message: 2726
```

```
//SW task: task_dma_transfer() uses DMA to
//transfer encryption from memory to HW device
AT  3887545 ns:CPU::CPU0.LT_initiator2:
                    Send write request
                    A = 0x20000000, D = 0x1
AT  3887595 ns:DMA::DMAC:
                    DMA is programmed:
                    A = 0x0        D = 0x1
                    Receive DMA control request.
```

```
//HW peripheral device RSA_IP decrypts:
AT  3990020 ns:   RSA_IP::RSA1:
                  Deciphered message =9614
AT  4090020 ns:   RSA_IP::RSA1:
                  Deciphered message =1454
AT  4190020 ns:   RSA_IP::RSA1:
                  Deciphered message =5878
```

```
//HW peripheral device uses DMA to transfer
//decrypted data to memory
AT  4190070 ns:DMA::DMAC
                    DMA is called by a device.
AT  4190070 ns:DMA::DMAC: reads:
                    A = 0x10000008
AT  4190255 ns:DMA::DMAC
                    Receives OK response.
                    D = 0x258e
AT  4190255 ns:DMA::DMAC: writes:
                    A = 0x0        D = 0x258e
```

```
//DMA controller interrupts the CPU after it
//finishes transferring
AT  4191180 ns:DMA::DMAC
                    DMA transfer finishes.
AT  4191180 ns:<IRQ_SOURCE: irq= DMAC
               Interrupt happens.
//CPU IRQ_controller acknowledges and processes
//it
AT  4191180 ns:CPU IC::IRQ 6 is raised.
AT  4191180 ns:CPU::IC: ICSR != 0.
                        Call IRQ Handler.
```

Figure 5-6. The simulation log of the DMA experiment

vice model decrypts these messages correctly (in the 3rd block). Afterwards, the device initiates a DMA transfer to write decrypted data back to the memory module (in the 4th block). Finally, in the last block, the DMA controller interrupts the CPU to notify the finish of transfer, and the Interrupt Controller in the Live CPU Model recognises its IRQ number and notifies the RTOS model for software interrupt handling.

Figure 5-7 shows the simulation timeline. DMA transfers relax the software system from frequent and time-consuming context switches, where the interrupt

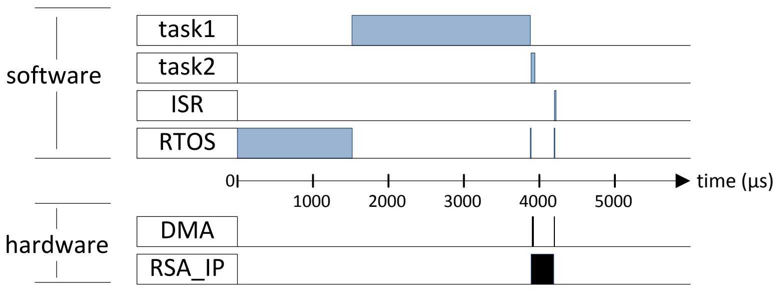


Figure 5-7. Simulation timeline

source only triggers once in each system working cycle. This not only means that the running speed (total cycles to finish a specific job) of a I/O intensive model can be improved by utilising DMA, but also infers the possibility to unitise the CPU more efficiently by implementing some more software functions during DMA transfer duration.

5.3 Summary

The software PE model has been extended with TLM communication interfaces by utilising the OSCI TLM-2.0 library and integrated in a simple SoC demonstration model including common TLM initiator and target modules. The favourable expandability of the Live CPU Model and the software PE modelling approach is also reflected in this work. One experiment shows the co-simulation performance of combined software PE and TLM models and indicates the marginal overheads of the software PE model in simple TLM simulation. Another experiment simulates a DMA I/O experiment by the proposed SoC TLM models. This demonstrates the TLM HW/SW co-simulation capability of the extended software PE model.

Because of highly abstract features and functions of the conceptual TLM models in this chapter, the timing accuracy and complete functionality of these integrated software PE and TLM communication models cannot be easily judged. Nevertheless, some academic research [6] as well as some industrial tools [188] [189] have successfully used OSCI TLM as a sound base for in-depth communication modelling, which inspire us to improve the software PE model for TLM-based HW/SW co-simulation in future research.

Chapter 6

Conclusions and Future Work

6.1 Summary of Contributions

This thesis has addressed the problem of defining and implementing real-time software simulation models in the SystemC language for software behavioural modelling and simulation in system-level design. The hypothesis was stated in Section 1.6.4 as follows:

A SystemC mixed timing modelling and simulation approach can enable fast, flexible and accurate RTOS-based real-time embedded software behavioural modelling and simulation in system-level design.

Regarding the fast characteristic, experiments show that the proposed approach can achieve two or three orders of magnitude speedups compared to the KEIL ARM ISS simulator and the uninterruptible fine-grained behavioural timing simulation approach in representative tests.

The flexible characteristic is mainly embodied by proposing multiple software models, time annotation granularities, and time advance techniques. The RTOS model and the whole simulator's modular structure also possess expandability to a certain degree.

The accuracy characteristic means twofold: functional accuracy and timing accuracy. The former is actualised by native-code simulation of the practical RTOS model. The latter is realised by both careful annotations in modelling and interruptible time advance in simulation. Experiments have demonstrated the two aspects.

This hypothesis was refined into four objectives in Section 1.6.4. Correspondingly, this thesis has contributed in four aspects:

1) The mixed timing real-time software modelling and simulation approach:
    a. It identifies the key aspects for real-time software timing modelling and simulation in the SystemC simulation environment.
    b. It defines two types of software models for early real-time software simulation, according to the granularity of function and timing, and describes their relevance to existing TLM abstract models.
    c. It proposes to use various modelling and simulation techniques for fast, flexible and reasonably accurate behavioural simulation.
2) The Live CPU Model:
    a. It proposes an abstract hardware CPU model inside a modular high-level software PE model, which ideally supports interruptible software time advance in SystemC simulation.
    b. It extends modelling capability of the mixed timing software modelling approach for HW/SW interactions.
3) The RTOS-centric Real-time software simulation model
    a. It provides a systematic approach for building and simulating real-time software (including both application tasks and RTOS) modular simulation models in SystemC, which represent the software aspect of the modular high-level software PE model.
    b. It identifies essential RTOS features that are necessary for practical RTOS modelling and implements them in a SystemC based RTOS model.
    c. This RTOS-centric approach can simulate mixed timing application task models in fast and accurate behavioural simulation, which reasonably approximate the functional and timing behaviours of a target software system.
4) Extending Software Models for TLM Communication
    a. It integrates standard TLM communication interfaces into the modular high-level software PE model.

b. This work also proposes a SoC TLM model, which not only integrates the software PE model but also defines other typical TLM initiator, target, and interconnection models.

6.2 Conclusions

6.2.1 The Mixed Timing Approach

In Chapter 3, a new SLDL-based software behavioural timing modelling and simulation method was proposed: the mixed timing approach. In this approach, in the context of TLM software computation modelling, two types of software timing models, i.e., abstract and native-code, are defined for different software modelling stages and can be mixed in simulation for modelling flexibility. Being independent from their timing annotation granularities, i.e., without the need of fine-grained time annotations, these software models are simulated by the *wait-for-event* time advance method, which can guarantee interruptible time advance and accurate software pre-emption. Various timing annotation granularities (i.e., task-level, function-level, segment-level, and basic block-level), functional accuracy levels (i.e., abstract and native-code), and time advance methods (i.e., variable-step and fixed-step) can be utilised in mixed timing software models for trade-offs between fast simulation performance, modelling flexibility, simulation observability, and accuracy.

Experiments demonstrate the fast performance of the mixed timing models, which are two or three orders of magnitude faster than ISS simulation and the conventional fine-grained uninterruptible behavioural software simulation. Timing accuracy of models is reflected from two aspects. Firstly, the basic time advance stopping latency and interrupt latency is zero-time, which means the mixed timing approach is capable to model and simulate real executions. Secondly, regarding timing accuracy of single task simulation, the proposed native-codes with fine-grained segment-level annotations incur a 0.12% marginal timing accuracy loss.

6.2.2 The Live CPU Model

In Chapter 3, we present the SystemC-based Live CPU Model as the conceptual hardware part of the modular software PE simulation model. The Live CPU Model consists of three components, i.e., the Live CPU Simulation Engine, the Interrupt Controller Model, and the Virtual Registers. It could also be extended with SW/HW interfaces for inter-module communication.

The Live CPU Simulation Engine is the basis of the *wait-for-event* time advance method for upper-level mixed timing software models. It consumes delay annotations and software models in an interruptible and resumable way, by which the target simulated clock is accurately progressed. It also supports mixed variable-step and fixed-step execution modes, which enable trade-offs between simulation speed and simulation observability.

The Interrupt Controller Model monitors external HW interrupts that are connected to the Live CPU module. It is the first-level component in the software PE model to handle interrupts and supports prioritised and maskable handling functions. Once it finds an interrupt, it can immediately notify the Live CPU Simulation Engine to stop current software time advance in order to handler the interrupt, i.e., with a zero-time interrupt latency.

Some Virtual Registers are modelled to assist software simulation in terms of task timing information context switch and flag setting.

In general, this Live CPU Model is a novel idea to introduce the conceptual hardware CPU model into generic high-level software simulation. It separates functions between software modules and hardware modules and makes whole the simulation framework more structured and extensible.

6.2.3 The RTOS-Centric Real-Time Software Simulation Model

In Chapter 4 a SystemC-based generic RTOS-centric real-time embedded software simulation model was presented. It is a native-code RTOS simulation approach, but can also flexibly support abstract task models. We describe the general embedded software stack model where our RTOS and application task models reside, together with common RTOS requirements.

This generic RTOS-centric real-time embedded software simulation model has a modular structure. It clearly separates application tasks, the RTOS, and the Live CPU Model into different modules and integrates them for simulation. The underlying Live CPU Model enables accurate time advance for RTOS-centric software models. Application tasks are modelled according to the mixed timing approach, which means that hybrid abstract task models and delay-annotated native-code task models can co-exist in one simulator, in order to enhance modelling flexibility. The RTOS model provides essential and generic services including task/process modelling, multi-tasking management, scheduling services, task synchronisation and communication, interrupt handling, and HAL services. This rich set of services can be invoked by tasks through normal function calls, which enable convenient and practical real-time software simulation at early design phases. In addition to these ample functional features, timing overheads of various RTOS services are also considered and added into simulation models.

Experimental results have shown fast performance, high functional accuracy, and small timing accuracy losses of RTOS-centric simulation, compared to cycle-accurate ISS simulation.

6.2.4 Extending Software Models for TLM Communication

In Chapter 5, OSCI TLM-2.0 communication interfaces are integrated into the software PE model. This extends the proposed RTOS-centric software simulation models for SW/HW inter-module TLM communication modelling. We describe how a software task can utilise TLM interfaces by presenting a set of refined functions in each level of software and hardware models. Furthermore, a SoC TLM model is introduced. It integrate the software PE model as the main software initiator and other common TLM modelling components such as target modules, combined initiator/target modules, and the interconnection module. This SoC model can be used for abstract HW/SW TLM co-simulation studies. In one experiment, the co-simulation performance of combined software PE and TLM models is investigated, which reveals that adding the software PE model in TLM simulation does not contribute much overheads. In another experiment, a DMA I/O simulation is carried out through the proposed SoC TLM models, which demonstrates the TLM HW/SW co-simulation capability of the extended software PE model.

6.3 Future Work

Based on research in this thesis, we realise that some topics and directions could be addressed in future work.

6.3.1 Improving Timing Modelling Techniques

In SLDL-based software behavioural timing modelling, timing estimation and annotation are two important aspects. They are the basis of software time advance in timing simulation. In current research, we mainly use the ISS simulator for accurate software performance estimation. This method consumes much execution time due to its slow speed. ISS estimation results are not directly connected with SystemC models and annotation statements are manually inserted into software models.

We consider the possibility to utilise some other software performance estimation methods for the proposed software simulation models. The static WCET timing analysis is a practical choice. Using low-level faithful static timing analysis results in high-level flexible dynamic simulation can maximise their respective advantages. It is also beneficial to develop an automatic annotation tool. This tool should help to insert annotation statements into models and support the timing techniques proposed in Section 3.2.4 in order to improve simulation performance.

6.3.2 Enriching RTOS Model Features

Comparing the RTOS model in this thesis to real RTOS products, e.g., ThreadX and μC/OS-II, memory management is not modelled or simulated in our research. As mentioned before, this is because that all our simulation models are natively executed by the SystemC simulator in the host machine and rely on memory management services provided by the C++ language and the host OS. However, even though it is not necessary to model stacks and heaps for task models, some RTOS services may also require dynamic allocation of memory spaces, e.g., message queues. We consider that lightweight memory management services could be integrated in the RTOS model. Their functions can be implemented by wrapping corresponding C++ dynamic memory operators and building memory control blocks. Their timing overheads can be annotated with the costs of memory management services in the target RTOS.

6.3.3 Multi-Processor RTOS Modelling

Given the fast development of MPSoC, multi-processor and multi-core RTOS modelling is certainly a promising research direction. In this thesis, the modular software-hardware system structure and the Live CPU Model idea have possessed some facilities to support multi-processor/core platforms in future research. However, in order to support Symmetric Multi-Processing (SMP) and Bound Multi-Processing (BMP), various system blocks and services of the current RTOS simulation model need to be revised. For example, the scheduling policies should explicitly support bounded and migrated tasks in multiprocessors, and resource sharing protocols should support sharing resources in multiprocessors. Besides, multiple cores may communicate with each other through the on-chip bus, and our approach can enable this TLM modelling capability. However, regarding high-level behavioural RTOS and software simulation, it is necessary to distinguish the relationship between RTOS/software internal communication and hardware inter-core communication. This requires a definition of software and hardware modelling abstraction levels.

Bibliography

[1] "RTX Real-Time Kernel," KEIL, http://www.keil.com/rl-arm/kernel.asp, 2009.

[2] R. Kamal, *Embedded Systems: Architecture, Programming and Design*: McGraw-Hill, 2008.

[3] F. Ghenassia, *Transaction Level Modeling with SystemC: TLM Concepts and Application for Embedded Systems*: Springer, 2005.

[4] "International Technology Roadmap for Semiconductors 2007 Edition Design," ITRS, http://public.itrs.net/, 2007.

[5] E. Frank, "VSPs Spur On-Time Delivery of Embedded Automotive Systems Software: Part 1 " VaST Systems Technology Corp., http://www.automotivedesignline.com/193400687%3Bjsessionid=4JEMQXURBKMNMQSNDLQCKH0CJUNN2JVN?printableArticle=true, 2006.

[6] M. Baklouti, A. Benzina, A. Bouchhima, and F. Petrot, "Extending Transaction Level Modeling for Embedded Software Design and Validation," in *International Conference on Design & Technology of Integrated Systems in Nanoscale Era, (DTIS 2007)*, 2007, pp. 64-69.

[7] L. Cai and D. Gajski, "Transaction level modeling: An Overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* Newport Beach, CA, USA: ACM Press, 2003.

[8] J. Madsen, K. Virk, and M. J. Gonzalez, "A SystemC-based Abstract Real-Time Operating System for Multiprocessor Systems-on-Chips," in *Multiprocessor Systems-on-Chips*, A. A. Jerraya and W. Wolf, Eds. San Francisco, CA: Morgan Kaufmann, 2005, pp. 284-311.

[9] R. Dömer, "Transaction Level Modeling of Computation," Technical Report, Center for Embedded Computer Systems, University of California, Irvine., 2006.

[10] F. J. Winters, C. Mielenz, and G. Hellestrand, "Design Process Changes Enabling Rapid Development," Convergence Transportation Electronics Association, http://www.vastsystems.com/notes/convergence20041018.pdf, 2004.

[11] F. Hessel, V. M. D. Rosa, C. E. Reif, C. Marcon, and T. G. S. d. Santos, "Scheduling Refinement in Abstract RTOS Models," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 5, pp.342-354, 2006.

[12] 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," *Design Automation for Embedded Systems,* vol. 10, pp.209-227, 2005.

[13] K. Yu and N. Audsley, "A Mixed Timing System-level Embedded Software Modelling and Simulation Approach," in *6th International Conference on Embedded Software and Systems, (ICESS '09)*, 2009.

[14] K. Yu and N. Audsley, "A Generic and Accurate RTOS-centric Embedded System Modelling and Simulation Framework," in *5th UK Embedded Forum (UKEF '09)*, Leicester, UK, 2009.

[15] K. Yu and N. Audsley, "Combining Behavioural Real-time Software Modelling with the OSCI TLM-2.0 Communication Standard," in *7th International Conference on Embedded Software and Systems, (ICESS '10)*, 2010.

[16] P. Marwedel, *Embedded System Design*, 1st ed.: Springer, 2003.

[17] R. Zurawski, *Embedded Systems Handbook*: CRC Press, 2006.

[18] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*: Newnes, 2005.

[19] A. Zerzelidis, *A Framework for Flexible Scheduling In Real-Time Middleware*, PhD Thesis, Dept. of Computer Science, University of York, 2007

[20] D. Schmidt and F. Kuhns, "An Overview of the Real-Time CORBA Specification," *Computer,* vol. 33, pp.56-63, 2000.

[21] A. Burns and A. Wellings, *Real-time Systems and Programming Languages: Ada 95, Real-time Java, and Real-time POSIX*, 3rd ed.: Addison Wesley, 2001.

[22] R. Domer, A. Gerstlauer, and W. Muller, "Introduction to Hardware-Dependent Software Design Hardware-Dependent Software for Multi- and Many-Core Embedded Systems," in *Proceedings of the 2009 Conference on Asia and South Pacific Design Automation* Yokohama, Japan: IEEE Press, 2009, pp. 290-292.

[23] S. Ball, *Embedded Microprocessor Systems: Real World Design*: Newnes, 2002.

[24] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*: CMP, 2003.

[25] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed.: Springer-Verlag New York Inc, 2004.

[26] J. W. S. Liu, *Real-Time Systems*: Prentice Hall, 2000.

[27] R. H. Bourgonjon, "Embedded systems in consumer products," in *Lecture Notes in Computer Science: Lectures on Embedded Systems: European Educational Forum School on Embedded Systems*. vol. 1494, 1998, p. 396.

[28] R. McMillan, "GM CTO predicts cars will run on 100 million lines of code," in *IDG News Service*, 21 October 2004.

[29] "Analysis of The Relationship Between EDA Expenditures and Competitive Positioning of IC Vendors for 2003," International Business Strategies, Inc., 2003.

[30] P. Dreike and J. McCoy, "Co-Simulating Software and Hardware in Embedded Systems," Sandia National Laboratories in Albuquerque, http://www.embedded.com/97/feat9706.htm, 1997.

[31] W. Wolf, *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*: Morgan Kaufmann, 2006.

[32] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: a Prescription for Electronic System-Level Methodology*: Morgan Kaufmann Publisher, 2007.

[33] G. De Michell and R. Gupta, "Hardware/Software Co-Design," *Proceedings of the IEEE,* vol. 85, pp.349-365, 1997.

[34] A. Gerstlauer and D. D. Gajski, "System-Level Abstraction Semantics," in *Proceedings of the 15th International Symposium on Systems Synthesis (ISSS '02)*, Kyoto, Japan, 2002, pp. 231-236.

[35] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, 5th ed.: Springer, 2008.

[36] P. J. Ashenden, *The Designer's Guide to VHDL*, 3rd ed.: Morgan Kaufmann, 2008.

[37] A. Habibi and S. Tahar, "A Survey on System-on-a-Chip Design Languages," in *3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2003.

[38] OSCI, "SystemC," http://www.systemc.org/.

[39] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, 1st ed.: Kluwer Academic Pub, 2000.

[40] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 19, pp.1523-1543, 2000.

[41] J. Plantin and E. Stoy, "Aspects of System-Level Design," in *Proceedings of the seventh international workshop on Hardware/software codesign*: ACM New York, NY, USA, 1999, pp. 209-210.

[42] G. Yang, A. Sangiovanni-Vincentelli, Y. Watanabe, and F. Balarin, "Separation of Concerns: Overhead in Modeling and Efficient Simulation Techniques," in *Proceedings of the 4th ACM international conference on Embedded software*: ACM New York, NY, USA, 2004, pp. 44-53.

[43] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS Modeling for System Level Design," in *Conference on Design, Automation and Test in Europe - Volume 1*: IEEE Computer Society, 2003.

[44] M. Fujita, I. Ghosh, and M. Prasad, *Verification Techniques for System-Level Design*: Morgan Kaufmann Publishers, 2008.

[45] B. Bhattacharya, L. Lavagno, and L. Vanzago, "Design Space Exploration of On-Chip Networks: A Case Study," in *Multiprocessor Systems-on-Chips*, A. A. Jerraya and W. Wolf, Eds. San Francisco, CA: Morgan Kaufmann, 2005, pp. 223-248.

[46] W. O. Cesário and A. A. Jerraya, "Component-Based Design for Multiprocessor Systems-on-Chips," in *Multiprocessor Systems-on-Chips*, A. A. Jerraya and W. Wolf, Eds. San Francisco, CA: Morgan Kaufmann, 2005, pp. 357-393.

[47] W. Klingauf, *Systematic Transaction Level Communication Modeling with SystemC*, Doktor-Ingenieur Thesis, Technische Universität Braunschweig, 2008

[48] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification Methodology and Techniques*: Kluwer Academic Publishers, 2002.

[49] K. MASSELOS and N. S. VOROS, "Design Flow for Reconfigurable Systems-on-Chip," in *System level design of reconfigurable systems-on-chip*, N. S. VOROS and K. MASSELOS, Eds.: Springer Verlag, 2005, pp. 87-105.

[50] "UML Superstructure Specification version 2.2," Object Management Group, http://www.omg.org/spec/UML/2.2/, 2009.

[51] MATLAB, MathWorks, http://www.mathworks.com, 2009.

[52] P. Alexander, *System-level Design with Rosetta*: Morgan Kaufmann Publishers, 2007.

[53] T. Mark, N. Hristo, S. Todor, D. P. Andy, E. Cagkan, P. Simon, and F. D. Ed, "A Framework for Rapid System-Level Exploration, Synthesis, and Programming of Multimedia MP-SoCs," in *5th IEEE/ACM international conference on Hardware/software codesign and system synthesis* Salzburg, Austria: ACM, 2007.

[54] T. Kogel, R. Leupers, and H. Meyr, *Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms*: Springer, 2006.

[55] W. Klingauf, H. Gädke, and R. Günzel, "TRAIN: a Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC," in *Proceedings of the conference on Design, automation and test in Europe* Munich, Germany: European Design and Automation Association, 2006.

[56] F. R. Wagner, W. Cesário, and A. A. Jerraya, "Hardware/Software IP Integration Using the ROSES Design Environment," *Trans. on Embedded Computing Sys.,* vol. 6, pp.17, 2007.

[57] N.-E. Zergainoh, A. Baghdadi, and A. Jerraya, "Hardware/Software Codesign of On-Chip Communication Architecture for Application-Specific Multiprocessor System-On-Chip," *International Journal of Embedded Systems,* vol. 1, pp.112-124, 2005.

[58] A. Gerstlauer, D. Shin, R. Dömer, and D. D. Gajski, "System-Level Communication Modeling for Network-on-Chip Synthesis," in *Proceedings of the 2005 conference on Asia South Pacific design automation* Shanghai, China: ACM Press, 2005.

[59] S. Ouadjaout and D. Houzet, "Generation of Embedded Hardware/Software from SystemC," *EURASIP Journal on Embedded Systems,* vol. 2006, pp.1-11, 2006.

[60] G. Schirner, G. Sachdeva, A. Gerstlauer, and R. Dömer, "Embedded Software Development in a System-Level Design Flow," in *Embedded System Design: Topics, Techniques and Trends*: Springer Boston, 2007, pp. 289-298.

[61] H. Yu, R. Dömer, and D. Gajski, "Embedded software generation from system level design languages," in *Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair*, 2004, pp. 463-468.

[62] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systematic Embedded Software Generation from SystemC," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*: IEEE Computer Society, 2003.

[63] M. Krause, O. Bringmann, and W. Rosenstiel, "Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems," *Design Automation for Embedded Systems,* vol. 10, pp.229-251, 2005.

[64] L. Guthier, S. Yoo, and A. Jerraya, "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software," in *Proceedings of the conference on Design, Automation, and Test in Europe (DATE 2001)*, 2001, pp. 679-685.

[65] G. Schirner, A. Gerstlauer, and R. Domer, "Automatic Generation of Hardware Dependent Software for MPSoCs from Abstract System Specifications," in *Proceedings of the 2008 conference on Asia and South Pacific design automation* Seoul, Korea: IEEE Computer Society Press, 2008.

[66] "IEEE Std 1666™-2005, IEEE Standard SystemC® Language Reference Manual," IEEE Computer Society, 2006.

[67] A. N. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*: Morgan Kaufmann, 2004.

[68] H. D. Patel and S. K. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling*: KLUWER ACADEMIC PUBLISHERS, 2005.

[69] N. Savoiu, S. K. Shukla, and R. K. Gupta, "Automated Concurrency Re-assignment in High Level System Models for Efficient System-Level Simulation," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '02)*, Paris, France, 2002.

[70] P. Destro, F. Fummi, and G. Pravadelli, "A Smooth Refinement Flow for Co-Designing HW and SW Threads," in *Proceedings of the conference on Design, automation and test in Europe*: EDA Consortium San Jose, CA, USA, 2007, pp. 105-110.

[71] B. Bhattacharya, J. Rose, and S. Swan, "Language Extensions to SystemC: Process Control Constructs," in *Proceedings of the 44th annual conference on Design automation* San Diego, California: ACM, 2007.

[72] P. Hastono, S. Klaus, and S. A. Huss, "Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems," in *The International Forum on Specification & Design Languages (FDL'04)* Lille, France, 2004, pp. 380-391.

[73] M. A. El-Salam, A. Salem, and G. Aly, "RTOS Modeling Using SystemC," in *System-on-Chip For Real-Time Applications*, W. Badawy and G. Jullien, Eds.: Kluwer Academic Publishers, 2003.

[74] "The SpecC Reference Compiler," Center for Embedded Computer Systems UC Irvine, 2006.

[75] Wikipedia, "SystemVerilog," http://en.wikipedia.org/wiki/Systemverilog.

[76] W. Rosenstiel, S. Swan, F. Ghenassia, P. Flake, and J. Srouji, "SystemC and SystemVerilog: Where do they fit? Where are they going?," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*. vol. 1, 2004.

[77] G. Nicolescu and A. A. Jerraya, *Global Specification and Validation of Embedded Systems*: Springer Netherlands, 2007.

[78] J. Jung, S. Yoo, and K. Choi, "Fast Cycle-Approximate MPSoC Simulation Based on Synchronization Time-Point Prediction," *Design Automation for Embedded Systems,* vol. 11, pp.223-247, 2007.

[79] G. Schirner, A. Gerstlauer, and R. Domer, "Abstract, Multifaceted Modeling of Embedded Processors for System Level Design," *Proceedings of the 2007 conference on Asia South Pacific design automation,* 384-389, 2007.

[80] V. Zivojnovic and H. Meyr, "Compiled HW/SW Co-Simulation," *Design Automation Conference Proceedings 1996, 33rd,* 690-695, 1996.

[81] M. Reshadi, P. Mishra, and N. Dutt, "Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 8, pp.1-27, 2009.

[82] E. Cheung, H. Hsieh, and F. Balarin, "Fast and Accurate Performance Simulation of Embedded Software for MPSoC," in *Proceedings of the 2009 Conference on Asia and South Pacific Design Automation* Yokohama, Japan: IEEE Press, 2009, pp. 552-557.

[83] 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 conference on Design, automation and test in Europe: Proceedings*, 2006, pp. 468-473.

[84] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation," in *Proceedings of the 2009 Conference on Asia and South Pacific Design Automation* Yokohama, Japan: IEEE Press, 2009.

[85] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Multiprocessor Performance Estimation Using Hybrid Simulation," in *45th ACM/IEEE Design Automation Conference (DAC 2008)*, 2008, pp. 325-330.

[86] S. Yoo and K. Choi, "Synchronization Overhead Reduction in Timed Cosimulation," in *Proc. of Int. High Level Design Validation. 6th International Workshop on Hardware/Software Co-Design, CODES/CASHE98*, 1997, pp. 157-164.

[87] Z. He, A. Mok, and C. Peng, "Timed RTOS Modeling for Embedded System Design," in *11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, 2005, p. 448.

[88] OSCI TLM Work Group, "OSCI TLM-2.0 Language Reference Manual (Software Version: TLM 2.0.1)," http://www.systemc.org/, 2009.

[89] A. Wieferink, M. Doerper, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr, "Early ISS Integration into Network-on-Chip Designs," in *Lecture Notes in Computer Science 3133, Proceedings of Third and Fourth International Workshops, SAMOS 2004*, Samos, Greece, 2004, pp. 443-452.

[90] S. Pasricha, "Transaction Level Modeling of SoC with SystemC 2.0," in *Synopsys User Group Conference (SNUG)*, 2002.

[91] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip," in *Proceedings of the conference on Design, automation and test in Europe* Munich, Germany: ACM, 2008.

[92] G. Schirner and R. Dömer, "Quantitative Analysis of the Speed/Accuracy Trade-Off in Transaction Level Modeling," *Trans. on Embedded Computing Sys.,* vol. 8, pp.1-29, 2008.

[93] A. Donlin, "Transaction Level Modeling: Flows and Use Models," in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 75-80.

[94] J. Cornet, *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip*, PhD Thesis, Mathématiques, Sciences et Technologies de l'Information, Informatique, Institut Polytechnique de Grenoble, 2008

[95] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl, "A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms," in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*: IEEE Computer Society, 2004.

[96] W. Ecker, V. Esen, T. Steininger, and M. Velten, "HW/SW Interface -- Implementation and Modeling," in *Hardware Dependent Software -- Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds.: Springer Science + Business Media B.V., 2009, pp. 95-150.

[97] F. Doucet, R. K. Shyamasundar, I. H. Krüger, S. Joshi, and R. K. Gupta, "Reactivity in SystemC Transaction-Level Models," in *Lecture Notes in Computer Science 4899, Proceedings of Third International Haifa Verification Conference (HVC 2007)*, Haifa, Israel, 2008, pp. 34-50.

[98] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*: Kluwer Academic Publishers Norwell, MA, USA, 2002.

[99] OSCI TLM Work Group, "Requirements specification for TLM 2.0," http://www.systemc.org/, 2007.

[100] A. Haverinen, M. Leclercq, N. Weyrich, and D. Wingard, "SystemC based SoC Communication Modeling for the OCP Protocol," Open Core Protocol International Partnership, 2002.

[101] T. Kogel, A. Haverinen, and J. Aldis, "OCP TLM for Architectural Modeling," Open Core Protocol International Partnership (OCP-IP), www.ocpip.org, 2005.

[102] D. Shin, L. Cai, A. Gerstlauer, R. Domer, and D. D. Gajski, "System-on-Chip Transaction-Level Modeling Style Guide," Technical Report CECS-TR-04-24, Center for Embedded Computer Systems, University of California, Irvine, 2004.

[103] G. Schirner and R. Dömer, "Fast and Accurate Transaction Level Models Using Result Oriented Modeling," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, p. 368.

[104] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration," in *Proceedings of the 41st annual Design Automation Conference* San Diego, CA, USA: ACM, 2004.

[105] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens, "A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03)* Newport Beach, CA, USA: ACM, 2003, pp. 7-12.

[106] T. Kogel, M. Doerper, T. Kempf, A. Wieferink, R. Leupers, G. Ascheid, and H. Meyr, "Virtual Architecture Mapping: A SystemC Based Methodology for Architectural Exploration of System-on-Chip Designs," in *Lecture Notes in Computer Science 3133, Proceedings of Third and Fourth International Workshops, SAMOS 2004*, Samos, Greece, 2004, pp. 138-148.

[107] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton, "GreenBus: a Generic Interconnect Fabric for Transaction Level

Modelling," in *Proceedings of the 43rd annual Design Automation Conference* San Francisco, CA, USA: ACM, 2006.

[108] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, and F.-R. Boyer, "Space: A Hardware/Software SystemC Modeling Platform Including an RTOS," in *Languages for System Specification: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specification from FDL'03*: Kluwer Academic Publishers, 2004, pp. 91-104.

[109] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *The Journal of VLSI Signal Processing,* vol. 41, pp.169-182, 2005.

[110] A. C. Nacul, M. Lajolo, and T. Givargis, "Interface-Centric Abstraction Level for Rapid Hardware/Software Integration " in *FDL'05 - Forum on Specification and Design Languages* Lausanne, Switzerland, 2005, pp. 329-340.

[111] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, and A. A. Jerraya, "Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration," in *Proceedings of the 2005 conference on Asia South Pacific design automation*, 2005, pp. 969-972.

[112] J. Madsen and M. Gonzalez, "Abstract RTOS Modelling in SystemC," in *20th IEEE NORCHIP Conference*, 2002, pp. 43-49.

[113] F. Hessel, V. M. d. Rosa, I. M. Reis, R. Planner, C. A. M. Marcon, and A. A. Susin, "Abstract RTOS Modeling for Embedded Systems," in *15th IEEE International Workshop on Rapid System Prototyping (RSP'04)*, 2004, pp. 210-216.

[114] R. L. Moigne, O. Pasquier, and J. P. Calvez, "A Generic RTOS Model for Real-time Systems Simulation with SystemC," in *Conference on Design, automation and test in Europe - Volume 3*: IEEE Computer Society, 2004.

[115] W.-T. Sun and Z. Salcic, "Modeling RTOS for Reactive Embedded Systems," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, 2007, pp. 534-539.

[116] D. C. Black and J. Donovan, *Systemc: From the Ground Up*: Springer, 2005.

[117] J. Vennin, S. Meftali, and J.-L. Dekeyser, "Understanding and Extending SystemC UserThread Package to IA64 Platform " http://www.design-reuse.com/articles/9402/understanding-and-extending-systemc-userthread-package-to-ia64-platform.html.

[118] F. Fummi, M. Loghi, G. Perbellini, and M. Poncino, "SystemC Co-Simulation for Core-Based Embedded Systems," *Design Automation for Embedded Systems,* vol. 11, pp.141-166, 2007.

[119] H. Posadas, J. Ádamez, P. Sánchez, E. Villar, and F. Blasco, "POSIX Modeling in SystemC," in *2006 conference on Asia South Pacific design automation* Yokohama, Japan: ACM Press, 2006.

[120] Y. Yi, D. Kim, and S. Ha, "Fast and Time-Accurate Cosimulation with OS Scheduler Modeling," *Design Automation for Embedded Systems,* vol. 8, pp.211-228, June, 2003.

[121] I. Bacivarov, S. Yoo, and A. A. Jerraya, "Timed HW-SW Cosimulation Using Native Execution of OS and Application SW," in *7th IEEE International High-Level Design Validation and Test Workshop*, 2002, pp. 51-56.

[122] H. Yu, A. Gerstlauer, and D. Gajski, "RTOS Scheduling in Transaction Level Models," in *1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* Newport Beach, CA, USA: ACM Press, 2003.

[123] G. Schirner and R. Domer, "Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling," *Design, Automation and Test in Europe, 2008. DATE'08,* 122-127, 2008.

[124] H. Zabel, W. Müller, and A. Gerstlauer, "Accurate RTOS Modeling and Analysis with SystemC," in *Hardware Dependent Software -- Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds.: Springer Science + Business Media B.V., 2009, pp. 233-260.

[125] J. Madsen, S. Mahadevan, K. Virk, and M. Gonzalez, "Network-on-Chip Modeling for System-Level Multiprocessor Simulation," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*: IEEE Computer Society, 2003.

[126] P. Hastono, S. Klaus, and S. A. Huss, "An Integrated SystemC Framework for Real-Time Scheduling Assessments On System Level," in *Proceedings of The 25th IEEE International Real-Time Systems Symposium (RTSS), WIP Session*, Lisbon, Portugal, 2004.

[127] P. A. Hartmann, H. Kleen, P. Reinkemeier, and W. Nebel, "Efficient Modelling and Simulation of Embedded Software Multi-Tasking Using SystemC and OSSS," in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, 2008, pp. 19-24.

[128] I. Bacivarov, A. Bouchhima, S. Yoo, and A. A. Jerraya, "ChronoSym: a New Approach for Fast and Accurate SoC Cosimulation," *International Journal of Embedded Systems* vol. 1, pp.103 - 111 2005.

[129] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada, "RTOS-Centric Hardware/Software Cosimulator for Embedded System Design," in *2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Stockholm, Sweden, 2004.

[130] A. Bouchhima, S. Yoo, and A. Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model," in *Asia and South Pacific Design Automation Conference 2004 (ASP-DAC'04)*, 2004, pp. 469-474.

[131] M.-K. Chung, S. Yang, S.-H. Lee, and C.-M. Kyung, "System-Level HW/SW Co-Simulation Framework for Multiprocessor and Multithread SoC," in *VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, 2005, pp. 177-179.

[132] H. Posadas, E. Villar, F. Blasco, R. D. Ds, P. Tecnológico, and S. Paterna, "Real-Time Operating System Modeling in SystemC for HW/SW co-simulation," in *Proceedings of Conference on Design of Circuits and Integrated Systems, IST* Lisbon, 2005.

[133] M. Krause and O. Bringmann, "Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation," in *6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*: ACM New York, NY, USA, 2008, pp. 143-148.

[134] R. Righter and J. C. Walrand, "Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE,* vol. 77, pp.99-113, 1989.

[135] G. Schirner, *Analysis and Optimization of Transaction Level Models for Multi-Processor System-on-Chip Design*, PhD Thesis, Electrical and Computer Engineering, University Of California, Irvine, 2008

[136] H. Zabel and W. Müller, "An Efficient Time Annotation Technique in Abstract RTOS Simulations for Multiprocessor Task Migration," in *Distributed Embedded Systems: Design, Middleware and Resources*. vol. 271, IFIP International Federation for Information Processing, 2008, pp. 181-190.

[137] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor," in *Conference on Design, Automation and Test in Europe*, 2008, pp. 276-279.

[138] A. Grigg and N. Audsley, "Reservation-Based Timing Analysis - a Practical Engineering Approach for Distributed Real-Time Systems," in *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, 2001, pp. 103-110.

[139] S. Lippman and J. Lajoie, *C++ Primer*: Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1998.

[140] Wikipedia, "Basic block," http://en.wikipedia.org/wiki/Basic_block.

[141] "Intel® Performance Tuning Utility," Intel Corporation, http://software.intel.com/en-us/articles/intel-performance-tuning-utility/.

[142] L. Cai, A. Gerstlauer, and D. Gajski, "Retargetable Profiling for Rapid, Early System-Level Design Space Exploration," in *Proceedings of the 41st Design Automation Conference (DAC'04)*, 2004.

[143] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson, "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems," *International Journal on Software Tools for Technology Transfer (STTT),* vol. 4, pp.437-455, 2003.

[144] A. Colin and I. Puaut, "Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System," *13th Euromicro Conference on Real-Time Systems,* 191–198, 2001.

[145] E. Bini and G. Buttazzo, "Measuring the Performance of Schedulability Tests," *Real-Time Systems,* vol. 30, pp.129-154, 2005.

[146] "µVision IDE," KEIL, http://www.keil.com/uvision/.

[147] "QNX® Neutrino® RTOS," QNX Software Systems, http://www.qnx.com.

[148] Microsoft, "How To Use QueryPerformanceCounter to Time Code," http://support.microsoft.com/?scid=kb%3Ben-us%3B172338&x=12&y=14.

[149] J. J. Labrosse, *Microc/OS-II: The Real-Time Kernel*: Cmp, 2002.

[150] "IEEE Std 1003.13-2003, IEEE Standard for Information Technology-Standardized Application Environment Profile (AEP)-POSIX Realtime and Embedded Application Support," The Institute of Electrical and Electronics Engineers., 2004.

[151] "uITRON4.0 Specification," ITRON Committee, TRON Association, http://www.assoc.tron.org/spec/itron/itron403e/mitron-403e.pdf, 2002.

[152] E. Lamie, *Real-time embedded multithreading: using ThreadX and ARM*: Cmp Books, 2005.

[153] S. Yoo and A. A. Jerraya, "Introduction to Hardware Abstraction Layers for SoC," in *Embedded Software for SoC*, A. A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, Eds.: Kluwer Academic Publishers, 2003, pp. 179-186.

[154] K. Popovici and A. Jerraya, "Hardware Abstraction Layer Introduction and Overview," in *Hardware Dependent Software -- Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds.: Springer Science + Business Media B.V., 2009, pp. 67-94.

[155] R. Davis and A. Burns, "Hierarchical Fixed Priority Pre-Emptive Scheduling," in *26th IEEE International Real-Time Systems Symposium (RTSS 2005)*, 2005, p. 10.

[156] M. Behnam, T. Nolte, I. Shin, M. Asberg, and R. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008)*, 2008, p. 63.

[157] "S.Ha.R.K.: Soft Hard Real-Time Kernel," http://shark.sssup.it/.

[158] F. Rammig, M. Ditze, P. Janacik, T. Heimfarth, T. Kerstan, S. Oberthuer, and K. Stahl, "Basic Concepts of Real Time Operating Systems," in *Hardware Dependent Software -- Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds.: Springer Science + Business Media B.V., 2009, pp. 14-45.

[159] P. A. Laplante, *Real-Time Systems Design and Analysis*, 3rd ed.: Wiley-IEEE Press, 2004.

[160] Wikipedia, "Monolithic kernel," http://en.wikipedia.org/wiki/Monolithic_kernel.

[161] "VxWorks RTOS," Wind River, www.windriver.com.

[162] QNX Software Systems, "QNX Neutrino RTOS System Architecture," 2007.

[163] Wikipedia, "Microkernel," http://en.wikipedia.org/wiki/Microkernel.

[164] A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating System Concepts*, 6th ed.: Wiley, 2002.

[165] A. Burns, K. Tindell, and A. Wellings, "Effective Analysis for Engineering Real-Time Fixed Priority Schedulers," *IEEE Transactions on Software Engineering,* vol. 21, pp.475-480, 1995.

[166] J. A. Carbone, "RTOS Real-Time Performance vs. Ease Of Use: Assessing performance needs of an application vs. other considerations," Express Logic, Inc., http://www.rtos.com/page/imgpage.php?id=208.

[167] B. Doherty, "Determining Worst-Case RTOS Response Time," Green Hills Software, http://www.rtcmagazine.com/articles/print_article/100152.

[168] S. Baskiyar and N. Meghanathan, "A Survey of Contemporary Real-time Operating Systems," *Informatica* vol. 29, pp.233-240, 2005.

[169] S. Heath, *Embedded Systems Design*, 2nd ed.: Newnes, 2003.

[170] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *Proceedings of the 36th ACM/IEEE conference on Design automation* New Orleans, Louisiana, United States: ACM Press, 1999.

[171] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM),* vol. 20, pp.46-61, 1973.

[172] J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation,* vol. 2, pp.237-250, 1982.

[173] N. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times," Technical Report YCS 164, Department of Computer Science, Univerisity of York, 1991.

[174] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective," *Real-Time Systems,* vol. 8, pp.173-198, 1995.

[175] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems," *Real-Time Systems,* vol. 1, pp.27-60, 1989.

[176] G. C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems,* vol. 29, pp.5-26, 2005.

[177] "MaRTE OS (Minimal Real-Time Operating System for Embedded Applications)," http://marte.unican.es/.

[178] "What Makes a Good RTOS," Dedicated Systems, http://www.es2.be/encyc/BuyersGuide/RTOS/Evaluations/docspreview.asp, 2001.

[179] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: an Approach to Real-Time Synchronization," *IEEE Transactions on Computers,* vol. 39, pp.1175-1185, 1990.

[180] M. H. Klein and T. Ralya, "An Analysis of Input/Output Paradigms for Real-Time Systems," Software Engineering Institute, Carnegie Mellon University, 1990.

[181] J. Chen, *On Synchronization Issues in Multiprocessor Real-Time Systems*, PhD Thesis, Department of Computer Science, University of York, York, UK, 1998

[182] "Real-Time Executive for Multiprocessor Systems (RTEMS)," OAR Corporation, http://www.rtems.com/.

[183] W. Shi, *Implementation and Performance of POSIX Sporadic Server Scheduling In RTLinux*, Master of Science Thesis, Department of Computer Science, THE FLORIDA STATE UNIVERSITY, 2001

[184] "QNX Neutrino RTOS System Architecture," QNX Software Systems, http://www.qnx.com/download/feature.html?programid=9342.

[185] S. Mahadevan, K. Virk, and J. Madsen, "ARTS: A SystemC-Based Framework for Multiprocessor Systems-on-Chip Modelling," *Design Automation for Embedded Systems,* vol. 11, pp.285-311, December, 2007.

[186] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. A. Jerraya, "Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment abstraction layer," *Design, Automation and Test in Europe Conference and Exhibition, 2003,* 550-555, 2003.

[187] S. Furber, *ARM System-on-Chip Architecture*: Addison-Wesley Professional, 2000.

[188] "CoWare Commits Support for SystemC TLM-2.0 Standard," CoWare, http://www.coware.com/news/press664.htm.

[189] "ARM's IP and OSCI TLM 2.0," ARM, http://www.nascug.org/events/8th/nascug_8_paper_5.pdf.