HLA Compliant Real-Time Operating System Simulation

By

Rui Mao

A Thesis Submitted to the Faculty of Graduate Studies and Research
In Partial Fulfillment of the Requirements for the Degree of

M.Sc in Information and Systems Science

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario

© August 2002

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79793-7

Canada

The undersigned recommend to the
Faculty of Graduate Studies and Research

Acceptance of the thesis:

**HLA Compliant
Real-Time Operating System Simulation**

submitted by

Rui Mao

in partial fulfillment of the requirements

for the degree of

M.Sc in Information and Systems Science

A handwritten signature in black ink, consisting of a series of loops and a long horizontal stroke, positioned above a horizontal line.

Professor Trevor Pearce, Thesis Supervisor

A handwritten signature in black ink, featuring a large 'S' and 'C' followed by a long horizontal stroke, positioned above a horizontal line.

Chair, Department of Systems and Computer Engineering

Carleton University

© August 2002

Abstract

With the increasing complexity in embedded hardware and software areas, model-based embedded system design has become increasingly difficult. Developers have been increasingly relying on simulation-based design. Since simulation is developed on a host workstation, it enables the parallel development of hardware and software; moreover, it allows system errors to be detected early in a stable host environment.

As embedded application demands grow rapidly, it is becoming increasingly important to support component interoperability in the early stages of the design process. This thesis presents a component-based RTOS simulation system built on the HLA. The HLA compliant RTOS simulation is developed for a uniprocessor system in which simulated tasks use the simulated RTOS kernel services such as periodic static priority-based preemptive scheduling, and semaphores. Moreover, shared memory is also simulated in the HLA compliant RTOS simulation. The scheduler implemented in the HLA compliant RTOS simulator is a rate monotonic scheduler in the case that task priorities are assigned in rate monotonic order and tasks do not communicate via inter-task communication mechanisms.

Acknowledgements

I would like to express my gratitude to my supervisor Trevor Pearce for his guidance, encouragement, technical expertise, and valuable time that he so generously shared with me during my research and writing of this thesis.

I would like to thank John Knox and Danny Lemay for their support and assistance in the lab. I also want to thank Shikharesh Majumdar, Judy Bowman, Prosenjit Bose, and Linda Pfeiffer for allowing me to study in Systems and Computer Engineering Department.

Finally, I would like to thank my husband, Tingjun Wen, who supported me wholeheartedly during my graduate studies.

Table of Contents

List of Figures.....	VII
List of Tables	VIII
List of Tables	VIII
List of Abbreviations.....	IX
List of Abbreviations.....	IX
Chapter 1 Introduction	1
1.1 Embedded Systems	1
1.2 Motivation.....	5
1.3 Outline	7
Chapter 2 Background.....	9
2.1 Real-Time System	10
2.2 Task Management.....	11
2.2.1 Static Table-Driven Approaches	12
2.2.2 Static Priority-Driven Preemptive Approaches.....	13
2.3 Inter-Task Communication	14
2.3.1 Semaphores	15
2.3.2 Shared Memory	16
2.3.3 Priority Inversion	17
2.4 Timing Tools	17
2.5 HLA	19
2.5.1 The HLA Rules.....	21
2.5.2 The HLA Interface Specification.....	23
2.5.2.1 RTI Components	23
2.5.2.2 Management Areas.....	25
2.5.3 The HLA Object Model Template.....	28
2.6 Related Work.....	29
Chapter 3 Problem Statement and Contributions	31
3.1 Problem Statement.....	31
3.2 Problem Solution.....	32
3.3 Research Scope.....	32
3.4 Contributions.....	34
Chapter 4 System Design for The RTOS Simulation	36
4.1 Host Workstation.....	38
4.2 System Design for Task Scheduling.....	38
4.2.1 Dispatching of Task Execution	40
4.2.2 Task Queues in the RTOS Kernel Federate	40
4.2.3 Task State Diagram	41
4.2.4 Simulation Time Management in the RTOS Kernel Federate	44
4.2.5 Schedulability Analysis	46

4.3 Generating the Simulation Application Program.....	47
4.4 System Design in The HLA.....	49
4.4.1 Shared Data Modeled in The HLA.....	49
4.4.2 Federation Management.....	51
4.4.3 Declaration Management.....	53
4.4.4 Object Management.....	55
4.4.5 Ownership Management.....	57
4.4.6 Time Management.....	60
4.4.7 Callback Delivery Strategies.....	60
4.4.8 The FOM.....	61
Chapter 5 System Implementation for The RTOS Simulation	64
5.1 Linux Process Control and Thread Control	65
5.2 Monitoring the RTOS Simulation.....	66
5.3 Conversion Process for the RTOS Simulation.....	67
5.4 Generating the RTOS Simulation Executable.....	70
5.5 Semaphore	71
5.5.1 Semaphore Limits.....	72
5.5.2 Semaphore Initialization	72
5.5.3 Semaphore Locking and Unlocking.....	74
5.6 Task Management.....	75
5.6.1 Task Control Block and Semaphore Control Block.....	76
5.6.2 Implementation of Task Queues in the RTOS Kernel Federate	77
5.7 Shared Memory.....	78
Chapter 6 Example Application.....	79
6.1 Characteristics of Tasks	80
6.2 The Real-Time Application Program.....	81
6.3 The Simulation Application Program	85
Chapter 7 Conclusion and Future Work	92
7.1 Conclusion	92
7.2 Future Work.....	94
7.2.1 Automatic Code Transformation	94
7.2.2 Extended RTOS Functionality	94
7.2.2.1 Scheduling.....	95
7.2.2.2 Solutions to the Unbounded Priority Inversion Problem	95
7.2.3. Integration of the I/O Simulation	96
7.2.4 RTOS Simulation of Multiprocessor Real-Time Systems	97
Bibliography.....	98

List of Figures

Figure 1: RTI Components.....	24
Figure 2: The HLA Interface between RTI and Federates.....	25
Figure 3: Two Federates Interact with Each Other.....	27
Figure 4: System Design for the RTOS Simulation	37
Figure 5: State Diagram	42
Figure 6: Scheduling of Two Tasks: T1 and T2.....	45
Figure 7: Generating the Simulation Application Program	48
Figure 8: Federation Management in the RTOS Federation.....	53
Figure 9: Ownership Pull Model in the RTOS Federation - Intrusive.....	58
Figure 10: Object Class Hierarchy.....	62
Figure 11: Interaction Class Hierarchy.....	63
Figure 12: Linux Process Model for the RTOS Simulation	65
Figure 13: Semaphore Locking Operation in the RTOS Kernel Federate.....	75
Figure 14: Semaphore Unlocking Operation in the RTOS Kernel Federate.....	75
Figure 15: The Future Scheduled Queue	78

List of Tables

Table 1: State of Task T1 and Task T2 on the Simulated Time Line.....	46
Table 2: Declaration Management in the RTOS Federation	53
Table 3: Some Information in the TCBs of Three Tasks.....	77

List of Abbreviations

RTOS	Real-Time Operating System
API	Application Programming Interface
POSIX	Portable Operating System Interface
I/O	Input/Output
ITC	Inter-Task Communication
HLA	High Level Architecture
RTI	Runtime Infrastructure
LRC	Local RTI Component
WCET	Worst Case Execution Time
LCM	Least Common Multiple
RM	Rate Monotonic
EDF	Earliest-Deadline-First
TCB	Task Control Block
DMSO	Defense Modeling and Simulation Office
IEEE	Institute of Electrical and Electronic Engineers
OMT	Object Model Template
FOM	Federation Object Model
SOM	Simulation Object Model
RtiExec	RTI Executive
FedExec	Federation Executive
RID	RTI Initialization Data
FED	Federation Executive Data
TSO	Time-Stamp Ordered
DDM	Data Distribution Management
eCos	Embedded Configurable Operating System
IA	Implementation Animator
MOM	Management Object Model
ET	Embedded Tk
PIP	Priority Inheritance Protocol
PCP	Priority Ceiling Protocol

Chapter 1

Introduction

An embedded system is often comprised of hardware and software components developed simultaneously. During the development of an embedded system, models are often used to abstract ideas, and host-based simulations are often conducted to allow dynamic execution of models. Real-time operating system (RTOS) simulation plays an important role in embedded software development in that it could reduce cost, time to market, and software development risk.

1.1 Embedded Systems

An embedded system is a set of pre-defined, specific functions to be performed, and the resources (e.g. memory, power, processor speed, computational functionality) available are constrained [24]. Commonly, an embedded system includes hardware and software components, and the functions defined in the system are frequently subject to timing constraints. Owing to many constraints related to an embedded system, an embedded system design can be much more complex than a general-purpose computer system design.

A widely accepted trend for developing an embedded system is hardware/software codesign [39] in which system development can be idealized as three phases: system decomposition phase, hardware and software development phase, and system integration phase. In the first phase, requirements and constraints are defined to form a specification, and the system is decomposed into hardware and software components with identified functionalities. In the second phase, hardware and software are designed and built simultaneously. In the final phase, the developed software and hardware components are integrated, verified and validated. Since separately developed modules are pulled together and run as a whole system in the final phase, each phase's development should be compatible with its subsequent phase.

A popular method of hardware/software codesign is model-based codesign [40]. Models are abstract ideas ignoring specific details, and they have been used extensively in the development of embedded systems for their advantages: Models formalize ideas, thus facilitate communication among developers; models provide documentation artifacts, and promote component reuse; and mathematical based models allow analysis to be carried out to reason about product properties. However, models are often based on some mathematical theories, and mathematical formalisms are often used to represent the real world. Languages commonly used to model systems have been analyzed formally, and many useful properties have been shown to be undecidable [43]. As a result, models can be complex and hard to understand.

When a mathematical analysis model is not practical, developers often develop a computer simulation to encode a model allowing dynamic analysis of the execution of the model. By building and executing a simulation, developers can see detailed system interactions with respect to a specific sequence of events. Moreover, because of the stability characteristic, a host-based simulation can provide a very stable environment for software development and testing. Therefore, in order to ease the development of an embedded system, a simulation can be an intermediate step between an analytic model and a product implementation.

A typical software engineering method for an embedded system is presented in a hierarchy of work products at three level of abstraction: requirement (most abstract), design, and implementation (least abstract). The requirements specify the I/O relationships and required timing constraints. The design specifies software components and their interactions. The implementation realizes the designed components by programming. Ideally, the work products can be constructed in a “top down” fashion with the requirements carried out before the design, and the design completed prior to the implementation. Parnas and Clements [22] recommended that software developers should produce the documentation by faking the idealized “top-down” method although the reality of a software development process is a tortured discovery process. In practical development scenarios, we will never find a process that allows us to design software in a perfectly rational way. For example, requirements may not be known in exact detail initially, requirements may change over time, implementation decision may be made in advance (e.g. a product may be developed by reusing or modifying existing sub-

components), technology advancements may lead to changes in some components, and human errors may result in flaws requiring correction.

The software in an embedded system is often organized as a set of cooperating tasks (real-time applications) managed by the RTOS at runtime. This enables the system to respond quickly to external events (interrupts) by immediately activating specific tasks [23]. Tasks are required to maintain time-constrained relationships while handling hardware-level inputs and outputs. Tasks are often implemented using a procedural language such as C/C++. The RTOS provides support services for tasks, such as task scheduling, task synchronization and communication, and time management, through RTOS APIs. In this thesis, the term “task” is used to refer to application entities, while the term “process” and the term “thread” are used to refer to simulation implementation entities.

A standard RTOS API could accelerate embedded software development. Consequently, many standard RTOS APIs have arisen. Currently, POSIX 1003.1b [9] is the largest effort toward standard RTOS APIs, and RTOS vendors have increasingly adopted it. However, the POSIX standard may not be capable of meeting all the functional requirements of an embedded system. As a result, in addition to adopting some POSIX APIs, many RTOS vendors embellish their RTOS products with additional functionalities via proprietary RTOS APIs. In other words, the RTOS may have its own personality. For example, VxWorks [38] is a commercial RTOS, which provides some POSIX APIs as well as many VxWorks proprietary APIs.

1.2 Motivation

Embedded real-time systems are growing rapidly in both hardware and software areas, which has led to continuous increments in system complexity. Embedded application manufactures and developers are facing many challenges such as keeping costs low, offering appealing and innovative products, and reducing time to market for future products. With rising embedded application demands, embedded systems are being built from larger scale components (subsystems), which results in increasing attention to component interoperability. Due to greater complexity of work products used in development, it is becoming more difficult to analyze model-based design. As a consequence, developers are largely relying on simulation while exploring development decisions and implications.

RTOS simulation plays an important role in achieving system development goals in that it enables software designers to develop and test embedded real-time applications on their PCs or workstations without requiring access to actual target hardware. In most cases, debugging and testing software in an actual target hardware environment is extremely expensive and time-consuming due to several reasons [1, 2]:

- Early system testing starts on non-stable targets, and faults are difficult to locate because they may be in hardware, software, or both.

- Development and testing environments for embedded systems are usually less powerful than workstation environments.
- Interdependent hardware and software development can cause massive delay, particularly when a component must be physically available to allow another activity to begin.

Since an RTOS simulation is developed on a host computer other than a target computer, it enables the parallel development of hardware and software; and it also allows partial application testing to be carried out detecting software errors in a stable environment. Many RTOS vendors integrate an RTOS simulator into their products. For example, the VxSim RTOS simulator provides a stable environment in which prototype VxWorks application programs can be exercised using the Tornado development tools [41].

In the development of real-time software, there are intimate relationships between the RTOS kernel and tasks. Each task must exhibit predictable deterministic performance to meet its timing constraints. The RTOS kernel provides task management, memory management, Inter-Task Communication (ITC), and other services to support tasks to meet their requirements. Since an RTOS simulation is developed to emulate software execution on a target machine, it must provide RTOS functionalities that will be available on the target machine.

In this thesis, I will present a multitasking RTOS simulator developed under Linux 6.2, which is based on the High Level Architecture (HLA). The HLA is the software architecture for creating computer simulations out of component simulations [15]. Due to its characteristics of reuse and interoperability, the HLA can be used as the glue to combine existing simulations and accommodate new ones. Currently, the RTOS simulator presented in this thesis supports periodic static priority-based preemptive scheduling for a uniprocessor system, and it can be used to analyze the schedulability of the hard real-time system under the worst case execution time. In addition to the real-time scheduler, shared memory and semaphores are also simulated to provide inter-task communication and synchronization.

1.3 Outline

Chapter 2 describes the background information about task requirements in real-time systems, the RTOS kernel services, timing tools related to an RTOS simulation, the HLA, and related work. Chapter 3 gives this thesis's problem statement and solution, research scope, and contributions. Chapter 4 presents how the RTOS simulation system is constructed, how tasks are scheduled in the RTOS simulation, and how the HLA Runtime Infrastructure (RTI) services are used in the RTOS simulation. Chapter 5 describes how the RTOS simulation is implemented on the Linux platform, how to convert a real-time application program to a simulation application program, and how the RTOS

functionalities are implemented. Chapter 6 gives an example application of the RTOS simulation. Chapter 7 summarizes this thesis, and gives a few future research directions.

Chapter 2

Background

This chapter discusses the requirements of tasks, describes some services provided by the RTOS kernel, gives an overview of timing tools relating to estimate the worst case execution time (WCET) of each task, introduces the HLA, and discusses related work. The significant difference between real-time systems and other computer systems is that each task is required to meet its timing constraints. In a real-time system, the RTOS kernel manages the execution of tasks using task management, ITC, memory management, and other mechanisms. In a hard real-time system, since missing a deadline may cause disastrous consequence, the schedulability of tasks is often analyzed to guarantee deterministic timing behavior of tasks. Schedulability analysis methods often assume that the WCET of each task is known. This means timing tools must be available to estimate the WCET of each task. In this thesis, in order to simulate the interoperability of tasks and the RTOS kernel of a real-time system, the HLA is used as the simulation tool to construct RTOS simulation.

2.1 Real-Time System

A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced [21]. If the timing constraints of the system are not met, system failure is said to have occurred. One important characteristic of a real-time system is timeliness, that is, tasks are scheduled to meet their deadlines [42].

In general, the timing constraints for a task include the arrival time specifying when the execution begins, the execution time indicating how long it executes, and the deadline for completion. The task arrivals may be periodic with a constant interval between successive invocations of the tasks, or may be aperiodic. The task's execution time may be fixed, or may be variable. The execution itself may be preemptible or non-preemptible. The deadline is usually some fixed duration after the task's arrival, but the implication of missing a deadline may vary. In a hard real-time system, a task must complete its execution by its deadline (called hard deadline); otherwise, missing the hard deadline may result in catastrophic failure of the system. In a soft real-time system, a task may have a soft deadline, which just specifies a recommendation or preference for the completion of the execution. In this thesis, I focus on the hard real-time system simulation.

In a real-time system, the RTOS kernel offers varieties of services to tasks such as task management, ITC, and memory management. I will describe some RTOS kernel services subsequently.

2.2 Task Management

Tasks are schedulable entities with precedence relationships, resource requirements, fault-tolerance requirements, importance levels, and timing constraints [7]. Task management includes actions such as creating, destroying, blocking, unblocking, setting priorities, and scheduling of concurrent tasks [8]. Each task has a related Task Control Block (TCB) in the RTOS kernel. A task's TCB is initialized when it is created, and it includes information such as the task's current logical state (e.g. running, blocked), priority, and so on.

Scheduling of tasks is one of the major task management responsibilities, and because a real-time system has timing constraints, scheduling all the tasks to meet their deadlines is of utmost importance. Scheduling involves the allocation of resources and time to tasks in such a way that the tasks meet their timing constraints. One of the primary issues in scheduling is schedulability analysis. In other words, tasks can be verified to adhere predictably to their critical timing constraints prior to their use [26]. Based on whether a system performs schedulability analysis and whether the analysis is done statically or dynamically, scheduling algorithms can be classified into four categories [3]: static table-

driven approaches, static priority-driven preemptive approaches, dynamic planning-based approaches, and dynamic best effort approaches. It should be noted that although four classes of scheduling algorithms have been identified, some scheduling techniques possess characteristics that span more than one of these classes. Since dynamic algorithms do not know the arrival times of tasks a priori, they cannot guarantee deterministic performance. As a result, dynamic algorithms are not suitable for hard real-time systems. In this thesis, I only introduce static table-driven approaches and static priority-driven preemptive approaches.

2.2.1 Static Table-Driven Approaches

Static table-driven approaches perform static schedulability analysis and the resulting scheduling (also called a table or time line) is used at runtime to decide when a task should be scheduled. These approaches are applicable to periodic tasks. Because the static table-driven approach is highly predictable, it is suitable for hard real-time systems. In the case that periodic tasks do not communicate or synchronize with each other, there exists a feasible schedule if and only if there exists a feasible schedule for the Least Common Multiple of the periods (LCM) [35]. That is, since the set of tasks is repeatedly executed every LCM unit of time at runtime, the schedulability of a set of tasks can be determined in LCM unit of time. However, besides periodicity constraint, tasks may have resource requirements and can possess precedence, exclusion, communication, as well as replication constraints. In these cases, the resulting scheduling problem is NP-complete [4]. Although the static table-driven approach is highly predictable, it suffers from

inflexibility since any change to the tasks and their characteristics may require a complete overhaul of the table.

2.2.2 Static Priority-Driven Preemptive Approaches

Static priority-driven preemptive approaches perform static schedulability analysis, but unlike static table-driven approaches, no explicit schedule is constructed. Each task in a real-time system is assigned with a fixed priority or a dynamically changed priority that is related to its timing constraints. At runtime, tasks are scheduled in order of decreasing priority. When a low priority task is in execution and a higher priority task arrives, the former is preempted and the processor is given to the new arrival task. Liu and Layland [5] were the first to formally study priority-driven algorithms. They focus on the periodic task scheduling on a single processor and proposed two preemptive algorithms. The first algorithm, called the Rate Monotonic (RM) algorithm, assigns static priorities to tasks based on their periods. It assigns higher priorities to tasks with shorter periods. The clearest application of the RM algorithm is for hard real-time systems [6]. Another algorithm studied by Liu and Layland is the Earliest-Deadline-First (EDF), a dynamic priority algorithm. In this algorithm, the closer a task to its deadline, the higher is its priority.

The RM algorithm is an optimal static priority scheme, in the sense that whenever the RM algorithm fails to find a feasible priority ordering, so does any other static priority ordering. The RM algorithm is studied in an idealized situation in which all tasks are

periodic with worst case execution times known, do not synchronize with one another, do not suspend themselves during execution, can be instantly preempted by higher priority tasks with zero context switch time, and have deadlines at the end of their periods. In a periodic task scheduling, each task has ready times for the task instantiations separated by a fixed period, and each instantiation or individually scheduled execution of a task is referred as a job.

The RM algorithm is attractive due to the following factors:

- It uses static priority assignment in which a task's priority is assigned once it arrives, and the task's priority does not have to be reevaluated as time progresses.
- It provides a mechanism for predicting real-time performance.
- It gives insight for uncovering subtle performance problems in real-time systems.

2.3 Inter-Task Communication

Multitasking systems often require mechanisms for ITC. ITC involves synchronization, mutual exclusion, and/or data exchange among co-operating tasks. Due to different application needs, different ITC mechanisms have evolved, such as semaphores, mailboxes, message queues, monitors, rendezvous, and so on. There are many standards

defining ITC mechanisms. For example, the POSIX standard real-time extension [9], which is supported by many commercial RTOS, includes three types of ITC: POSIX message queues, POSIX semaphores, and POSIX shared memory. In this section, I will describe semaphores and shared memory.

2.3.1 Semaphores

Semaphores can be described as counters allowing tasks to co-ordinate their activities to share resources. If a semaphore is *on* (semaphore count greater than zero), a resource is available. Otherwise, if a semaphore is *off*, a resource is not available. When a task tries to access an unavailable resource protected by a semaphore (called a locked semaphore), the task is said to be blocked on the semaphore. At some time later, this blocked task may be unblocked because another task releases the resource.

There are two types of semaphores, namely binary semaphores and counting semaphores, which are used for mutual exclusion and synchronization. When a semaphore is used as a mutex for mutual exclusion (exclude tasks from simultaneous use of a shared resource), it is called a binary semaphore, and its count is initialized to 1. On the other hand, a semaphore can be used as a resource counter for synchronization among tasks, and its initial count can be 0 or any positive integer.

There are four basic semaphore operations, which are listed as follows:

- **Creation:** When a semaphore is created, its value is set to a non-negative integer, and no task is blocked on the semaphore.
- **Wait:** When a task calls semaphore wait, a semaphore lock operation will be performed in the RTOS kernel. This means that if the semaphore's count is greater than 0, it is decremented; otherwise, this task is blocked on the semaphore.
- **Post:** When a task calls semaphore post, a semaphore unlock operation will be performed in the RTOS kernel. This means that if there are blocked tasks on the semaphore, a task at the head of the semaphore's block queue is unblocked; otherwise, the semaphore's count is incremented.
- **Destroy:** When a semaphore is destroyed, tasks blocked on the semaphore are released.

2.3.2 Shared Memory

Shared memory can provide fast data exchange among tasks. However, when using shared memory as an ITC mechanism, one must be careful to prevent race conditions in critical sections. Commonly, we call a section of code which reads/writes shared data a critical section, and we call the potential for interleaved execution of a critical section by multiple tasks a race condition [27].

Mutual exclusion mechanisms can be used to ensure uninterrupted execution of critical sections, and to co-ordinate tasks accessing shared memory. For example, we can use a binary semaphore to prevent interference in a critical section: a task calls semaphore wait before entering the critical section, and calls semaphore post after finishing execution of the critical section. Through the use of the binary semaphore, simultaneous accessing of shared data is excluded; as a result, a race condition can be prevented.

2.3.3 Priority Inversion

When synchronization mechanisms are applied with priority-driven preemptive scheduling mechanisms, priority inversion might happen. When a high priority task needs a resource held by a low priority task, priority inversion occurs in that the priority of the high priority task is inverted with respect to the priority of the low priority task. In the case that a medium priority task preempts the low priority task, the duration of the priority inversion might be longer than the duration of executing the low priority task's critical section. This situation is often called unbounded priority inversion.

2.4 Timing Tools

Tasks in a real-time system are subject to timing constraints, which requires a real-time programming language that has the capacity to express and enforce timing constraints. However, most high-level programming languages, like C/C++, do not include time

explicitly. So if we want to use one of these high-level programming languages for developing real-time software, we must use a timing tool to predict the timing behavior of each task.

In many hard real-time scheduling theories (e.g. the RM algorithm), it is often assumed that the WCET of each task is constant and is known a priori. Correspondingly, many researches have been carried out to estimate or measure the WCET.

In a non-pipelined processor without cache memory, the WCET of instructions can be calculated assuming that the execution time of each instruction is constant. For example, Mok and his students [29] presented a timing tool ignoring control costs, which analyzes an assembly language stream by a graph method to find a worst-case path and computes the execution time by simulating the hardware. Park and Shaw [30] presented a source-level timing scheme including control costs for a subset of C, which followed four steps: first, a program is decomposed into atomic blocks; second, the implementation of each atomic block is predicted; third, the execution time of each basic block is determined; finally, the WCET is computed by applying formulas.

Approaches that take into account the effects of pipelines and caches have been widely studied. For example, Zhang et al. [31] presented a timing tool based on a mathematical model of the pipelined Intel 80C188 processor, which takes into account the overlap between instruction execution and instruction fetching within each basic block. Arnold et al. [32] presented a method called static cache simulation to analyze a program's control

flow to statically categorize the caching behavior of each instruction. Lim et al. [33] proposed an extended timing schema, which allows reasoning about the execution time variation of a program construct by surrounding program constructs due to pipelined execution and cache memories of RISC processors.

2.5 HLA

The High Level Architecture (HLA) is a common architecture for constructing simulations. The RTOS simulation presented in this thesis uses the HLA as the framework for simulating the RTOS kernel and tasks.

The HLA was developed by the Defense Modeling and Simulation Office (DMSO) to facilitate interoperability among simulations and to promote reuse of simulations and their components. In September 2000, the HLA was approved as an open standard through the Institute of Electrical and Electronic Engineers (IEEE) - IEEE Standard 1516.

The HLA is defined by three interrelated components:

- The HLA Rules [10] define the principles of the HLA in terms of responsibilities that federates (simulations, supporting utilities, or interfaces to live systems) and federations (sets of federates working together) must uphold.

- The HLA Interface Specification [11] identifies how federates will interact with the federation and, ultimately, with one another. It describes a generic communication interface that allows simulations to be connected and coordinated; thus, it addresses interoperability.
- The Object Model Template (OMT) [12] provides a template for documenting HLA-relevant information about classes of simulation or federation objects and their attributes and interactions. The primary object of the HLA OMT is to facilitate interoperability among simulations and reuse of simulation components.

In the HLA, data exchange can be model as objects or interactions. Commonly, we use objects to represent simulated entities that endure, and we use interactions to represent events that may occur at a point in simulated time. The OMT defines object classes with a set of attributes, and interaction classes with a set of parameters. During a federation execution, a federate creates an instance of an object class by registering the object instance, and evolves the state of the object instance by updating the values of some instance attributes; on the contrary, a federate is informed of the existence of a new instance of an object class by discovering the object instance, and is advised of a change in the state of the object instance by reflecting the new values of some instance attributes. At any time, only one federate is responsible for updating the value of an instance attribute to other federates, and this federate is said to own the instance attribute. Updating and reflecting instance attribute values of an object class can be a means of

communication among federates. Sending and receiving interactions of an interaction class can be another means of communication among federates.

2.5.1 The HLA Rules

The HLA rules include five federation rules and five federate rules. For federations, the rules address the requirement for a federation object model (FOM), object ownership and representation, and data exchange. For federates, the rules require a simulation object model (SOM), time management in accordance with the HLA RTI time management services, and certain required functionality and constraints on attribute ownership and updates [14].

Federation Rules [10]:

- (1) Federations shall have an HLA FOM, documented in accordance with the HLA OMT.
- (2) In a federation, all simulation-associated object instance representation shall be in the federates, not in the RTI.
- (3) During a federation execution, all exchange of FOM data among federates shall occur via the RTI.

(4) During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification.

(5) During a federation execution, an instance attribute shall be owned by at most one federate at any given time.

Federate Rules [10]:

(6) Federates shall have an HLA SOM, documented in accordance with the HLA OMT.

(7) Federates shall be able to update and/or reflect any attributes and send and/or receive interactions, as specified in their SOMs.

(8) Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOMs.

(9) Federates shall be able to vary the conditions (e.g., thresholds) under which they provides updates of attributes, as specified in their SOMs.

(10) Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

2.5.2 The HLA Interface Specification

In the HLA, federates interact with the runtime infrastructure (RTI) to establish and maintain a federation and to support efficient information exchange among federates. The RTI is software that provides common interface services during an HLA federation execution for synchronization and data exchange. The HLA interface services are divided into six management areas: federation management, declaration management, object management, ownership management, time management, and data distribution management [13].

2.5.2.1 RTI Components

As illustrated in Figure 1 [13], RTI software is comprised of the RTI Executive process (RtiExec), the Federation Executive process (FedExec) and the libRTI library.

RtiExec is a globally known process. It manages multiple federation executions. That is, it manages the creation and destruction of FedExecs, directs joining federates to the appropriate federation execution, and ensures that each FedExec has a unique name. Each application communicates with RtiExec to initialize RTI components. When an RTI is initialized, an RID (RTI Initialization Data) file can be read to provide control parameters. Each parameter in the RID file has a default setting, which is used in the case that the parameter value is not specified in the RID file or no RID file is supplied.

Each executing federation is characterized by a single, global FedExec. FedExec manages multiple federates within the federation execution. It allows federates to join and to resign, and facilitate data exchange between participating federates. A FedExec process is created by the first federate to successfully invoke the “createFederationExecution” service for a given federation execution name. Each federation execution needs a FED (Federation Execution Data that is derived from the FOM), which is bound with the federation execution name when the federation is created.

RTI services specified in the HLA Interface Specification are provided by libRTI. Within libRTI, the class RTIambassador bundles the services provided by the RTI, and the abstract class FederateAmbassador identifies the callback functions that each federate is obliged to provide. As shown in Figure 2 [15], RTIambassador offers federate-initiated services, and FederateAmbassador provides RTI-initiated services.

Figure 1: RTI Components

Figure 2: The HLA Interface between RTI and Federates

2.5.2.2 Management Areas

Federation Management includes such tasks as creating federations, joining federates to federations, observing federation-wide synchronization points, effecting federation-wide saves and restores, resigning federates from federations, and destroying federations.

Declaration Management includes publication, subscription, and supporting control functions. During a federation execution, federates that produce objects instances (or object attributes) or produce interactions use declaration management services to declare exactly what they are able to publish; and federates that consume object instances (or object attributes) or consume interactions use declaration management services to declare their subscription interests.

Object Management includes instance registration and instance updates on the object producer side and instance discovery and reflection on the object consumer side. It also

includes interaction sending and receiving, instance updates upon consumer requests, and other miscellaneous support functions [13].

Federation management, declaration management, and object management comprise the basic services that federates need to exchange information with each other. Ownership management, time management, and data distribution management provide advanced services that may be critical in some federation executions and unnecessary in others.

Figure 3 shows an example with two federates exchanging information. In this example, federation management, declaration management, and object management are used. As for the first federate, it first creates and joins a federation execution by using federation management; second, it publishes object class attributes and interaction classes by using declaration management; third, it registers an instance of an object class, and then it updates the instance attribute values and sends an interaction by using object management; finally, it resigns the federation execution by using federation management. As for the second federate, it first joins the federation execution by using federation management after the first federate joins the federation; second, it discovers an object instance created by the first federate by using declaration management; third, it reflects the instance attribute values and receives an interaction by using object management; finally, it resigns and destroys the federation execution by using federation management.

Figure 3: Two Federates Interact with Each Other

Ownership Management includes ownership transfer of instance attributes and ownership transfer of the `privilegeToDeleteObject` instance attribute. During a federation execution, the privilege to update the value for an instance attribute is uniquely held by a single federate at any given time. When a federate registers an object instance, all instance attributes of the object instance for which the corresponding class attributes are currently published by the registering federate are set as owned by the registering federate [11]. In addition, only one federate has the privilege to delete an object instance at any given time. The federate that registers an object instance automatically owns the `privilegeToDeleteObject` instance attribute for that instance.

Time Management is concerned with the mechanisms for controlling the advancement of each federate along the federation time axis. In general, time advances shall be coordinated with object management services so that information is delivered to federates

in a causally correct and ordered fashion. A federate may be “regulating”, “constrained”, “regulating and constrained”, or “neither regulating nor constrained”. A regulating federate may associate some of its events (such as updating instance attributes and sending interactions) with points on the federation time axis (called time-stamp). A constrained federate is interested in receiving Time-Stamp Ordered (TSO) events (such as reflecting instance attribute value and receiving interactions) from regulating federates. In other words, regulating federates regulate the progress of constrained federates. By default, federates are neither regulating nor constrained.

Data Distribution Management (DDM) provides a flexible and extensive mechanism for further isolating publication and subscription interests to reduce both the transmission and the reception of irrelevant data. Producers of data may employ DDM services to assert properties of their data in terms of user-defined spaces. Consumers of data may employ DDM services to specify their data requirements in terms of the same spaces. The RTI distributes data from producers to consumers based on matches between these properties and requirements.

2.5.3 The HLA Object Model Template

The OMT provides a common frame of reference for describing object models in the HLA, and it can be used to create a FOM or a SOM. A FOM is concerned with inter-federate issues, and every federation has a FOM. A FOM includes an enumeration of all object and interaction classes pertinent to the federation along with a specification of the

attributes or parameters that characterize these classes [12]. An SOM describes salient characteristics of a federate to aid in its reuse and focus on the intrinsic capabilities that an individual federate can provide to HLA federations as well as an individual federate can receive from other federates in HLA federations. The standard format in which SOMs are expressed facilitates determination of the suitability of federates for participation in a federation.

2.6 Related Work

A simulation system is usually constructed using available simulation tools. For example, in [34], two kinds of RTOS simulations, eCos (Embedded Configurable Operating System) simulation and RTLinux simulation, have been constructed using the CarbonKernel simulation tool. In [16, 17], the RTOS simulation has been built in the Implementation Animator (IA) simulation tool. Although all the above-mentioned RTOS simulations are based on event driven simulation techniques, the simulation tools used have different focus. CarbonKernel is a real-time operating system simulation tool whose intent is to provide a common framework for building specific RTOS personalities on top of it. The IA is a simulation tool whose intent is to visualize and test embedded real-time software in a workstation environment. The HLA is a simulation tool whose intent is to facilitate component interoperability and reuse.

Schedulability analysis is an important aspect in developing a hard real-time system. For example, a schedulability analyzer was built in the real-time Euclid [26], and Scheduler 1-2-3, an interactive schedulability analyzer [36], was developed for real-time systems. The real-time Euclid schedulability analyzer consists of the front end, which is a language dependent timing tool, and the back end, which is a schedulability analyzer for periodic tasks. The approach of Scheduler 1-2-3 assumes that a timing tool to estimate the WCET is available prior to the schedulability analysis.

In terms of the RTOS kernel services, a great deal of work has focused on developing an RTOS simulation that is compliant with POSIX standards. For example, the RTLinux simulation [28] and the IA RTOS simulation mentioned above both follow the POSIX Threads Extension [37] in which a task can be executed as a thread. In the POSIX Thread Extension, a thread is associated with a scheduling policy (*SCHED_FIFO*, or *SCHED_RR*, or *SCHED_OTHER*) specifying the scheduling algorithm, and a scheduling parameter structure specifying the scheduling priority. In the RTLinux simulation, a thread can be made to execute periodically, but this thread cannot specify required execution time within each period. The schedulability of a set of periodic tasks cannot be analyzed in the RTLinux simulation. A periodic task in this thesis uses the same process model as a periodic task in KURT [25]. That is, in the code for a periodic task, the task first performs initialization including establishing the task's scheduling parameters (priority, execution time in each period, and period), and then the task is switched to the real-time periodic execution mode. Currently, all KURT periodic tasks should have the same period. However, periodic tasks in this thesis could have different period.

Chapter 3

Problem Statement and Contributions

This chapter presents the problem statement and solution, gives the research scope, and states contributions made in this thesis.

3.1 Problem Statement

With the increasing of embedded application demands, embedded systems are being built from increasingly larger scale components. As embedded systems become more complex, it is becoming increasingly difficult to analyze model-based design. Consequently, there is a need to develop component-based simulation to allow dynamic interaction of components. RTOS simulation plays an important role in developing embedded software. However, existing RTOS simulations are not designed to support component-based simulation.

3.2 Problem Solution

In this thesis, I will present a multitasking RTOS simulator built on the HLA. The HLA compliant RTOS simulation consists of many simulation components. Currently, the RTOS simulator supports periodic static priority-based preemptive scheduling for a uniprocessor system, and it can be used to analyze the schedulability of a hard real-time system under the worst case execution time. In addition to the real-time scheduler, shared memory and semaphores are also simulated to provide inter-task communication.

3.3 Research Scope

In this thesis, a real-time application consisting of many tasks to be run on a target workstation is referred to as a real-time application program; a real-time simulation application consisting of many tasks to be run on a host workstation is referred to as a simulation application program. A simulation application program is derived from a real-time application program.

In this thesis, a uniprocessor multitasking RTOS simulation system was implemented in C++ under Linux 6.2. The RTOS simulation system was developed on the HLA whose RTI services provide transparent communication among simulation components. The RTOS simulator is a discrete event simulator, and it implements periodic static priority-

based preemptive scheduling, semaphores, and shared memory for a uniprocessor system based on the following assumptions:

- Each task in the task set must be periodic, and each task's period must be a constant integer (a scheduling parameter).
- The deadline of each task is taken to be the end of the period for that task.
- Each task has a fixed priority (a scheduling parameter).
- A higher priority task can preempt a lower priority task.
- Context switch time is zero.
- Interrupt handling is not simulated.
- Tasks can communicate using shared memory.
- A task can synchronize with other tasks using semaphores. If a task uses semaphore calls, these calls must be done within each period.
- Each task's execution time needed out of each period must be known as a constant integer (a scheduling parameter). Within a task's period, the WCET

needed until every semaphore call (including the semaphore call time) and until the end of the period must be known as a constant integer.

- Initially, all the tasks are ready at the same time.

A real-time application program is assumed to include the code to declare shared variables and semaphores stored in shared memory (optional), the code to initialize semaphores (optional), the code to initialize shared variables (optional), the code to set each task's scheduling parameters, and the code to execute each task in real-time. Since a real-time application program does not include the WCET needed code for each task, a timing tool for measuring the WCET of each task is assumed available in this thesis.

In this thesis, the priority inversion problem is not considered, and the performance of the RTOS simulation is not a main concern.

3.4 Contributions

Contributions of this thesis include:

- **System architecture for the RTOS simulation:** the RTOS simulation is mapped into the HLA architecture in which federates including an RTOS kernel federate,

a semaphore initialization federate, and task federates cooperate by using RTI services (chapter 4).

- **Conversion Process for the RTOS simulation:** a real-time application program can be converted to a simulation application program (chapter 4 and chapter 5).
- **Implementation of the RTOS simulation:** a general, reusable RTOS kernel is implemented to provide periodic static priority-based preemptive scheduling, and semaphores services to tasks. Moreover, shared memory is also simulated in the RTOS simulation (chapter 5). By running a simulation application program on the simulated RTOS kernel, a real-time application program can be developed and tested early on a host workstation without the presence of target hardware.
- **Schedulability analysis and RM analysis:** The scheduler implemented in the RTOS simulator allows experimenting and analyzing hard real-time system schedulability in the simulator other than in the target system (chapter 5). Moreover, if task priorities are assigned in rate monotonic order and tasks do not communicate using ITC mechanisms, the scheduler is a rate monotonic scheduler.

Chapter 4

System Design for The RTOS Simulation

This chapter describes the HLA compliant RTOS simulation system architecture. As illustrated in Figure 4, the RTOS simulation is built on top of a host workstation whose operating system is Linux 6.2. The RTOS simulation system consists of a simulation application program and an RTOS framework in the HLA. The simulation application program includes the code for each task, and the code for semaphore initialization (if there are semaphores to be used). The RTOS framework in the HLA is comprised of an RTOS kernel federate, task deputies (one for each application task), and a semaphore initialization deputy (if there are semaphores to be used). The simulation application program and the RTOS framework in the HLA are coupled together by the RTOS APIs.

Since the HLA is a component architecture [15], the RTOS simulation is modeled as an RTOS federation in the HLA consisting of an RTOS kernel federate, task federates (one for each application task), and a semaphore initialization federate (if there are semaphores to be used).

Note: Sema4_Init: semaphore initialization

Figure 4: System Design for the RTOS Simulation

The semaphore initialization federate is only required in applications that use semaphores. To simplify subsequent discussion of the framework, it will be assumed that the semaphore initialization federate is required unless stated otherwise.

4.1 Host Workstation

Since the target RTOS is simulated in a host workstation, it is important to choose a suitable simulation platform. The HLA compliant RTOS simulation implemented in this thesis was built on the Linux platform for two reasons: Linux is a widespread and freely distributed Unix-type operating system; Linux supports both multiple processes and multiple threads in the programming model.

In Linux, each process is started with a single thread called a primary thread, which can subsequently create other threads. Each process has its own address space with all threads within the same process space sharing the same global memory.

4.2 System Design for Task Scheduling

In the RTOS simulation, simulation time is different from real-time: simulation time is the time that is controlled by the RTOS kernel federate, and real-time is the wall-clock time for the execution of the RTOS simulation. The kernel federate schedules tasks in simulation time. However, a current limitation in the RTOS simulation design is that when a task is executing, it cannot be preempted.

To overcome this preemption problem, a task pre-execution model is designed in this thesis:

- When the RTOS kernel federate schedules a task to run, it must first find out how much time the task needs. The kernel federate instructs the task to execute until a synchronization point is reached or the end of the task execution for this period. When a task is executing, simulation time does not advance. Since semaphores are used as the synchronization mechanism in this thesis, synchronization points are those points where a task calls semaphore wait or semaphore post. In this thesis, I will use the term “control point” to represent a task’s synchronization points plus the end of the task execution in the current period.
- It’s assumed that the WCET needed by a task to reach each control point is available. When a task reaches a control point, the task stops execution and its corresponding task deputy sends the RTOS kernel federate a message containing the WCET needed by the task to reach the control point.
- Upon receiving the WCET message, the RTOS kernel federate starts to schedule the task on the simulated time line until the task has been allocated the WCET needed.
- When the simulated time reaches the scheduling time limit for the simulation run or the RTOS kernel federate detects that a task has missed its deadline, the RTOS simulation stops.

4.2.1 Dispatching of Task Execution

In the RTOS simulation, each task has a task id assigned by the RTOS kernel federate. The RTOS kernel federate controls each task's execution by sending messages with the form "*Run*" + "*Task id*" to task deputies. When a task deputy receives such a message, it retrieves the task id in the message to compare with its own task id. If the task's id is the same as the task id in the message, the task deputy will signal the task to execute. At some time later, when the task reaches a control point, its corresponding task deputy will send a WCET message to the RTOS kernel federate. On the other hand, if a task is not instructed to run by the RTOS kernel federate, this task's corresponding deputy continues to wait for other messages.

4.2.2 Task Queues in the RTOS Kernel Federate

In the RTOS simulation, the periodic static priority-based preemptive scheduler is based on the discrete event-driven simulation model in which the control of the simulated time depends on the nearest future event's scheduled time. In order to manage the simulation time advancement, the RTOS kernel federate manages a time-ordered task queue called the future scheduled task queue. In the future scheduled task queue, each task has an entry that indicates the time of the task's next period.

The future scheduled task queue is managed in an increasing order of task occurrence time. In the case that some future events happen at the same time, tasks with higher

priority are arranged before tasks with lower priority, and tasks with the same priority are managed in the FIFO order. When a task starts a new period's execution, a future event, which will happen at the task's next period, is enqueued to the future scheduled task queue. On the other hand, when the RTOS kernel federate selects a running task from the head of the future scheduled task queue, the selected task is dequeued from the future scheduled task queue.

In the RTOS kernel federate, there are some priority-managed queues: a ready queue, and a wait queue for each semaphore. In a priority-managed queue, tasks with higher priority are arranged before tasks with lower priority, and tasks with the same priority are managed in the FIFO order. When a task is ready to run but it is not running because its low priority, the task is enqueued to the ready queue. When the running task is preempted by a higher priority task, the current running task is enqueued to the ready queue. When the RTOS kernel federate selects a running task from the head of the ready queue, the selected task is dequeued from the ready queue. When a task tries to access a locked semaphore, the task is enqueued to the semaphore's wait queue. When a task is unblocked from a semaphore, the task is dequeued from the semaphore's wait queue.

4.2.3 Task State Diagram

In the RTOS simulation, there are four task states: *wait*, *ready*, *run*, and *blocked*. When a task is in the *wait* state, its execution is suspended until its next period to become ready to run again. When a task is in the *ready* state, it is in the ready queue, and it is ready to run

but not running. When a task is in the *run* state, it is running. When a task is in the *blocked* state, it is blocked on a semaphore waiting for a resource, and it will be unblocked when the resource is available for the task.

Figure 5: State Diagram

The state diagram for the RTOS simulation is shown in Figure 5, which describes all the possible legal state transitions as follows:

1. State change: *ready* to *run*. When the running task needs to be selected from the ready queue (e.g. at the beginning of RTOS scheduling), the task at the head of the ready queue changes its state from *ready* to *run*.
2. State change: *wait* to *run*. When there is no running task, the task at the head of the future scheduled task queue will change its state from *wait* to *run* at its next

period. Otherwise, when a high priority task preempts the running task at the high priority task's period, the high priority task changes its state from *wait* to *run*.

3. State change: *wait* to *ready*. At a task's period, this task is eligible to run but its priority is lower than the running task's priority or it is not the highest priority task eligible to run, then this task changes its state from *wait* to *ready*.
4. State change: *run* to *ready*. When preemption happens, the running task (that is, the preempted task) changes its state from *run* to *ready*.
5. State change: *run* to *blocked*. When the running task tries to access a locked semaphore, it changes its state from *run* to *blocked*.
6. State change: *blocked* to *ready*. When the running task unblocks a task at the head of a semaphore's wait queue, the unblocked task changes its state from *blocked* to *ready*.
7. State change: *run* to *wait*. When the running task finishes its current period's execution, it changes state from *run* to *wait*.

All other state transitions not shown in Figure 5 are impossible, or illegal in which case the set of tasks are not schedulable. Impossible state transitions include *ready* to *blocked*, *ready* to *wait*, *blocked* to *wait*, *wait* to *blocked*, *run* to *run*, *wait* to *wait*, and *blocked* to

blocked. Illegal state transitions occur when a task can not finish its last period's execution at the time of the task's new period. For example, at a task's period, the task's state should be changed to either *run* or *ready* for the new period's execution. If the task's current state at that time is *blocked* or *run* or *ready*, then the task has not yet finished its last period execution.

4.2.4 Simulation Time Management in the RTOS Kernel

Federate

Figure 6 shows an example of how to schedule two tasks named T1 and T2. Task T1 and task T2 use a semaphore S to synchronize with each other. Task T1 has higher priority, requires 3 execution time units, and has a 5 time unit period. Task T2 has lower priority, requires 5 execution time units, and has a 15 time unit period. In the periodic execution code of task T1, there is a semaphore wait call, a semaphore post call, and a call indicating the end of task T1's period. In the periodic execution code of task T2, there is a semaphore wait call, a semaphore post call, and a call indicating the end of task T2's period. It is assumed that task T1 and task T2 have been analyzed by a timing tool, and the WCET's needed to reach each control point are shown before the control points in Figure 6.

Task T1:

Measured WCET needed = 1;

Semaphore wait call on S;

Task T2:

Measured WCET needed = 2;

Semaphore wait call on S;

Measured WCET needed = 1;	Measured WCET needed = 1;
Semaphore post call on S;	Semaphore post call on S;
Measured WCET needed = 1;	Measured WCET needed = 2;
End of T1's period;	End of T2's period;

Note:
W: semaphore wait call on S.
P: semaphore post call on S.
E: a call indicating the end of a period

Figure 6: Scheduling of Two Tasks: T1 and T2

When scheduling tasks, the RTOS kernel federate dispatches task T1 to execute at simulation time 0. At some time later, task T1 calls semaphore wait on S, where task T1's deputy sends the RTOS kernel federate a WCET message of 1 time unit needed by task T1. Upon receiving this WCET message from task T1's deputy, the RTOS kernel federate begins to schedule task T1 for 1 simulation time unit. Since T1 is the highest priority task, it will not be preempted during the scheduled time, and the simulation time is advanced to 1. At simulation time 1, the RTOS kernel federate dispatches task T1 to

execute again until task T1 calls semaphore post on S. As messages are sent between task deputies and the RTOS kernel federate, the RTOS simulation continues. Table 1 lists the states of task T1 and task T2 from simulation time 0 to simulation time 14.

In the RTOS kernel federate, preemption happens when a higher priority task become ready to run while a lower priority task is running. In the case of preemption, the lower priority task changes its state from *run* to *ready* (e.g. at simulation time 5, 7, 10).

Simulation Time	0	1	2	3	5	6	7	8	9	10	11	12	13	14
State of Task T1	R	R	R	W	R	B	R	R	W	R	R	R	W	W
State of Task T2	RD	RD	RD	R	RD	R	RD	RD	R	RD	RD	RD	R	W

Note: R: Run. RD: Ready. W: Wait. B: Blocked.

Table 1: State of Task T1 and Task T2 on the Simulated Time Line

4.2.5 Schedulability Analysis

In the RTOS simulation, the periodic static priority-based preemptive scheduler can be used to analyze the schedulability of a set of tasks. A set of tasks is schedulable within a specified period of time only if all the tasks have legal state transitions. On the other hand, if a task in the task set misses its deadline within the specified period of time, the RTOS kernel federate will send a *Stop* message to all task deputies to stop the RTOS simulation.

When all the tasks do not use semaphores, given the least common multiple of task periods, the schedulability of the task set can be drawn. Furthermore, if the priorities of tasks are assigned in rate monotonic order and tasks do not communicate using ITC mechanisms, the scheduler is a rate monotonic scheduler. In the case that some tasks need to communicate using ITC mechanisms, the schedulability of a task set in the given scheduling time duration can be derived.

4.3 Generating the Simulation Application Program

A real-time application program is assumed to have many tasks: a primary task for semaphore initialization, and periodic tasks created by the primary thread. These periodic tasks may use shared memory to store shared variables and semaphores. For this research, each periodic task in a real-time application program is modeled after a periodic task in KURT [25]. That is, the code for each periodic task includes the code to perform task initialization including establishing the task's scheduling parameters, the code to switch to real-time periodic execution mode, and the code to execute the task periodically.

Figure 7: Generating the Simulation Application Program

As illustrated in Figure 7, a real-time application program designed for a target platform can be converted to a simulation application program to run on a host workstation. In the real-time application program, each task's real-time execution code must be first analyzed by a timing tool to add in the WCET before each control point; and then the real-time application program embedded with the WCET is transformed to a simulation application program with a specified scheduling time limit. Once the simulation application program is available, it can be compiled and linked with the RTOS framework in the HLA to generate an executable file that can be run on the host workstation.

Since the focus of this thesis is to build an HLA compliant RTOS simulator, a timing tool to estimate the WCET on the target machine is not implemented here. The transformation of a real-time application program embedded with the WCET to a simulation application program is discussed in detail in chapter 5.

4.4 System Design in The HLA

In the RTOS system, the RTOS kernel provides services to tasks. In addition, tasks interact with each other through semaphores and shared memory. In order to simulate the information exchange among tasks and the RTOS kernel, the HLA is used as the framework for the RTOS simulation.

The RTOS simulation is modeled as an RTOS federation in the HLA. In the RTOS federation, federates interact with each other through RTI services from five of the management areas, namely, federation management, declaration management, object management, ownership management, and time management.

4.4.1 Shared Data Modeled in The HLA

As introduced in chapter 2, data exchange can be modeled in the HLA as objects and/or interactions. Commonly, we use objects to represent simulated entities that endure, and we use interactions to represent events that may occur at a point in simulated time. However, objects and interactions can be used interchangeably to some extent. For example, we can model the RTOS federation entirely only using interactions since we can always achieve the effect of attribute updates by sending interactions. On the other hand, we can also model the RTOS federation entirely only using objects.

In the RTOS simulation, there is one object class: *Tcb*; and there are two interaction classes: *Communication* and *CommToRtos*. As shown in Table 2, the object class *Tcb* has four attributes representing the tasks' persistent characteristics: *Tid* for task id, *Priority* giving task execution precedence, *Period* specifying periodic task's period, and *ExeTime* indicating each period's execution time. The interaction class *Communication* has one parameter: *Message*, and it is used for general purpose message passing among federates. For example, *Communication* interaction instances are sent to simulate semaphore function invocations, to simulate shared memory access, and to dispatch tasks to execute. The interaction class *CommToRtos* has one parameter: *MsgToRtos*, and it is used by task deputies to send interactions (e.g. the WCET needed) to the RTOS kernel federate.

In the RTOS simulation, shared memory contains shared variables and semaphores. Accessing shared variables needs to be protected by semaphores. Shared memory is simulated by replicating shared memory contents in federates. Whenever shared memory content changes, all the local copies of shared memory content must be updated by exchanging information among federates. In this thesis, information exchange regarding shared memory content update is modeled as interactions other than objects because this avoids transferring ownership frequently among federates. As introduced in the chapter 2 about ownership management in the HLA, the privilege to update the value for an instance attribute is uniquely held by a single federate at any given time during a federation execution. Therefore, if shared memory is modeled as an object class, ownership transfer must be done whenever a federate updates an un-owned attribute.

4.4.2 Federation Management

Federation management is used to manage the execution of the RTOS federation, which includes creating the RTOS federation, joining federates to the RTOS federation, resigning federates from the RTOS federation, and destroying the RTOS federation.

Since a simulated RTOS kernel provides services to simulated tasks, the RTOS kernel federate should be the first federate in the RTOS federation, and it creates the RTOS FedExec process by calling the RTIambassador method *createFederationExecution()*. When *createFederationExecution()* is called, a federation name and a FED are supplied as two arguments.

After the RTOS FedExec completes its initialization and informs the RtiExec process of its existence, the RTOS kernel federate will join the RTOS federation by calling the RTIambassador method *joinFederationExecution()*.

If there are semaphores to be used in the RTOS simulation, the semaphore initialization federate will join the RTOS federation after the RTOS kernel federate's joining. When semaphore initialization completes successfully or a semaphore initialization fails, the semaphore initialization federate will resign the RTOS federation by calling the RTIambassador method *resignFederationExecution()*; moreover, the semaphore initialization federate will try to destroy the RTOS federation by calling the RTIambassador method *destroyFederationExecution()*. In the case that the semaphore

initialization federate is the last federate existing in the RTOS federation, the RTOS federation will be destroyed and the RTOS FedExec will be shut down. If semaphore initialization completes successfully, task federates will join the RTOS federation.

If there isn't any semaphore to be used in the RTOS simulation, task federates will join the RTOS federation after the RTOS kernel federate's joining. After all task federates join the RTOS federation, the simulated RTOS kernel will be able to provide services to tasks. At some point, the RTOS simulation will stop, and then task federates and the RTOS kernel federate will resign the RTOS federation, and they will try to destroy the RTOS federation. However, only the last federate existing in the RTOS federation will succeed with its *destroyFederationExecution()* call.

Figure 8 illustrates federation management in the RTOS federation: in this simulation, the RTOS kernel federate creates and joins the RTOS federation; and then the semaphore initialization federate joins the RTOS federation. After semaphore initialization successfully completes, the semaphore initialization federate resigns the RTOS federation, and task federates start to join the RTOS federation. Some time later, the RTOS simulation stops, and the RTOS kernel federate and task federates resign the RTOS federation. Finally, the RTOS kernel federate, which is the last federate in this RTOS federation, successfully destroys the RTOS federation.

Figure 8: Federation Management in the RTOS Federation

4.4.3 Declaration Management

Declaration management is used to declare federates' publish and subscribe interests in information exchange in the RTOS federation (Table 2).

Federate Name	Communication: Message	CommToRtos: MsgToRtos	Tcb: Tid, Priority, Period, ExeTime
RTOS Kernel Federate	Publish Subscribe	Subscribe	Publish All The Attributes Subscribe Priority, Period, ExeTime
Sema4_Init Federate	Publish Subscribe		
Task Federate	Publish Subscribe	Publish	Publish All the Attributes Subscribe Attribute Tid

Table 2: Declaration Management in the RTOS Federation

RTI services used in the RTOS federation are:

- RTIambassador method *publishObjectClass()* conveys the intention of a federate to begin acquiring and updating instances of a set of attributes of a specified class [19]. In the RTOS federation, the RTOS kernel federate and each task federate publish all the attributes of the object class *Tcb*. The RTOS kernel federate manages the execution of tasks and keeps information about tasks by using *Tcbs* (Task Control Blocks). Initially, each task federate needs to pass the task's scheduling parameters *Priority*, *Period*, and *ExeTime* to the RTOS kernel federate; and then the RTOS kernel federate initializes the task's *Tcb*. After the RTOS kernel finishes initializing a task's *Tcb*, it will assign a *Tid* to send back to the task.
- RTIambassador method *subscribeObjectClassAttributes()* declares a federate's interest in receiving reflections for updates of a specified set of attributes [19]. In the RTOS federation, the RTOS kernel federate subscribes the object class *Tcb* attributes *Priority*, *Period*, and *ExeTime*; and every task federate subscribes the object class *Tcb* attribute *Tid*.
- RTIambassador method *publishInteractionClass()* conveys the intention of a federate to begin generating interactions of a specified class [19]. In the RTOS federation, every federate publishes the interaction class *Communication*, and every task federate publishes the interaction class *CommToRtos*.

- RTIambassador method *subscribeInteractionClass()* declares a federate's interest in receiving a specified class of interactions. In the RTOS federation, every federate subscribes the interaction class *Communication*, and the RTOS kernel federate subscribes the interaction class *CommToRtos*.

4.4.4 Object Management

Object management is used to register and discover object instances, to remove object instances, to update and reflect object instance attribute values, and to send and receive interactions in the RTOS federation. In the RTOS federation, some RTIambassador methods are used, and some FederateAmbassador callback functions are implemented.

RTIambassador methods called in the RTOS federation are:

- *registerObjectInstance()* creates a new object instance of the specified object class, and it returns a federation unique handle to identify the object instance. The object instance handle is not interpreted by the RTI, but is only used for comparison purposes. In the RTOS federation, each task federate registers an object instance of the object class *Tcb*.
- *updateAttributeValues()* notifies the federation of a change in state of an object instance. A federate can only update its owned instance attributes. In the RTOS

federation, each task federate updates its *Tcb* object instance's *Priority* attribute value, *Period* attribute value, and *ExeTime* attribute value. Upon acquiring the ownership of a task's *Tcb* object instance's *Tid* attribute, the RTOS kernel federate updates the task's *Tid* attribute value.

- *sendInteraction()* is used by the federate to communicate an interaction to the federation. In the RTOS federation, the semaphore initialization federate uses interaction sending to initialize semaphores in the RTOS kernel federate, and the RTOS kernel federate sends interactions about each semaphore's initialization status in response to the semaphore initialization federate. The RTOS kernel federate instructs each task to execute by sending interactions. Task federates send interactions about the WCET needed to the RTOS kernel federate. For shared memory, whenever its content is changed, interactions are sent.

FederateAmbassador callback functions implemented in the RTOS federation are:

- *discoverObjectInstance()* informs a federate of the existence of an object instance in the federation that is relevant to the federate's current subscription interest. In the RTOS federation, when the RTOS kernel federate discovers a *Tcb* object instance of a task federate, it must keep a local copy of the discovered object instance for the task scheduling purpose.

- *reflectAttributeValues()* provides a federate with updated values of a set of instance-attributes relevant to the federate's current subscription interest. In the RTOS federation, the RTOS kernel federate uses this callback function to get each task's *Tcb* object instance's *Priority* attribute value, *Period* attribute value, and *ExeTime* attribute value, and each task federate uses this callback function to get the task's *Tid* attribute value.
- *receiveInteraction()* notifies a federate of an interaction in the federation relevant to the federate's current subscription interest.
- *removeObjectInstance()* informs a federate that the specified object instance has been deleted from the federation. In the RTOS federation, when a federate resigns, it uses DELETE_OBJECT_AND_RELEASE_ATTRIBUTES policy. As a result, *removeObjectInstance()* callback is invoked at remote federates.

4.4.5 Ownership Management

Ownership management is used to transfer object instance attribute ownership. In the RTOS federation, each task federate creates an object instance of the object class *Tcb*, and it owns that object instance's attributes (*Tid*, *Priority*, *Period*, and *ExeTime*). In order to allow the RTOS kernel federate to update an instance attribute *Tid*'s value, the ownership of the instance attribute *Tid* must be transferred to the RTOS kernel federate.

To exchange an object attribute ownership among federates in a federation execution, a “push” model or a “pull” model can be used. In the “push” model, a federate can try to give away responsibility for one or more attributes of an object instance. In the “pull” model, a federate can try to acquire responsibility for one or more attributes of an object instance [13]. In the RTOS federation, the RTOS kernel federate uses the intrusive “pull” model to acquire an object attribute *Tid*’s ownership. Figure 9 shows the ownership “pull” model used between the RTOS kernel federate and a task federate in which two RTIambassador methods and two FederateAmbassador callback functions are used.

Figure 9: Ownership Pull Model in the RTOS Federation - Intrusive

RTIambassador methods called in the RTOS federation are:

- *attributeOwnershipAcquisition()* initiates a request to transfer ownership of the specified set of instance attributes of the specified object instance to a federate. In the RTOS federation, the RTOS kernel federate calls

attributeOwnershipAcquisition() to initiate a request to a task federate for the object attribute *Tid*'s ownership transfer.

- *attributeOwnershipReleaseResponse()* releases ownership of the set of instance attributes for the specified object instance. In the RTOS federation, upon receipt of *requestAttributeOwnershipRelease()* callback, a task federate uses *attributeOwnershipReleaseResponse()* to provide a positive response to the RTOS kernel federate's *attributeOwnershipAcquisition()* request.

FederateAmbassador callback functions implemented in the RTOS federation are:

- *requestAttributeOwnershipRelease()* informs a federate that the specified set of instance attributes are the subject of an intrusive acquisition request made by a remote federate. This means that *requestAttributeOwnershipRelease()* is only invoked as the result of *attributeOwnershipAcquisition()* called by another federate. In the RTOS federation, each task receives the RTOS kernel federate's request for the ownership of its *Tcb* object instance's *Tid* attribute through the RTI via *requestAttributeOwnershipRelease()* callback.
- *attributeOwnershipAcquisitionNotification()* notifies a federate that the federate has acquired ownership of the specified set of instance attributes of the specified object instance. In the RTOS federation, this callback is invoked as the final result

of the RTOS kernel federate's *attributeOwnershipAcquisition()* request for the ownership of a task federate's *Tcb* object instance's *Tid* attribute.

4.4.6 Time Management

In the RTOS federation, time management is only used to advance simulation time in the RTOS kernel federate. Since simulation time is managed in the RTOS kernel federate to schedule the execution of tasks and all other federates have no concept of time, all the federates in the RTOS federation are neither regulating nor constrained. As a result, when the RTOS kernel federate requests time advancement, it is always granted to the requested time.

In the RTOS federation, the RTOS kernel federate uses RTIambassador method *timeAdvanceRequest()* to request an advance of its logical time to the specified federation time; and FederateAmbassador callback function *timeAdvanceGrant()* is implemented to inform the RTOS kernel federate of the completion of its previous time advance request.

4.4.7 Callback Delivery Strategies

In the HLA, there are two process control strategies that can be used by the RTI to obtain processing cycles and support callbacks [13]: polling process model and asynchronous I/O process model. The polling strategy uses a single thread of execution shared between the RTI and the federate, and it requires that the federate provide sufficient *tick()*

invocations to transfer process control to the Local RTI Component (LRC) and allow the RTI to perform work. If *tick()* is not called appropriately, the polling strategy can starve the RTI. On the other hand, the asynchronous I/O process model uses an internal thread within the RTI to avoid starvation. This thread will periodically wake up and determine if it can perform any internal RTI work. By using the asynchronous I/O strategy, the federate only needs to invoke *tick()* when it is prepared to handle callbacks. The process model selection is defined in the RTI.rid file via the RTI.ProcessModel.StrategyToUse parameter. In the RTOS simulation, the default strategy asynchronous I/O process model is used.

4.4.8 The FOM

In the RTOS federation, there is a FOM, which defines data (object classes and interaction classes) exchanged through the RTI. In the FOM, object classes form a single-inheritance hierarchy, which has a root class called *ObjectRoot* (Figure 10); and interaction classes form a single-inheritance hierarchy, which has a root class called *InteractionRoot* (Figure 11).

In the RTOS federation, The *ObjectRoot* class has two required subclasses: *RTIprivate* and *Manager*; moreover, it has one user defined subclass: *Tcb*. The object class *RTIprivate* is used by RTI implementers, which cannot have extended subclasses. The object class *Manager* is the root of the object portion of the MOM (Management Object Model), and all subclasses of the object class *Manager* are omitted in Figure 10.

In the RTOS federation, The *InteractionRoot* class has two required subclasses: *RTIprivate* and *Manager*; moreover, it has two user defined subclasses: *Communication* and *CommToRtos*. The interaction class *RTIprivate* is used by RTI implementers, which cannot have extended subclasses. The interaction class *Manager* is the root of the interaction portion of the MOM, and all subclasses of the interaction class *Manager* are omitted in Figure 11.

In the HLA, the MOM is a general facility for managing HLA federations, furnishing information about the federation and the means to affect its state [15]. The MOM itself as an object model consists of a standard part defined by the HLA Interface Specification, and user extensions designed by federation developers. In a federation execution, the FOM must contain at least the standard part of the MOM, and federates interact with the MOM through the FOM. In the RTOS federation, the MOM defined in the FOM is the standard part of the MOM, and the RTI is responsible for animating the standard MOM according to the HLA Interface Specification.

Figure 10: Object Class Hierarchy

Figure 11: Interaction Class Hierarchy

Chapter 5

System Implementation for The RTOS Simulation

This chapter discusses the RTOS simulation implementation on top of the Linux platform. As illustrated in Figure 12 about the Linux process model for the RTOS simulation, the RTOS kernel federate is implemented as a Linux parent process with a single thread of execution. If there are semaphores to be initialized before the execution of tasks, the semaphore initialization federate is implemented as a Linux child process spawned by the parent process. The Linux child process for the semaphore initialization federate has two threads of execution: a primary thread for the semaphore initialization deputy, and another thread for executing the semaphore initialization code. Finally, each task federate is implemented as a Linux child process spawned by the parent process. Each Linux child process for the task federate has two threads of execution: a primary thread for the task deputy, and another thread for executing the task.

Although the RTOS simulation is built on Linux, the simulation application program does not use Linux kernel services but uses simulated RTOS kernel services such as task management, and semaphores.

Figure 12: Linux Process Model for the RTOS Simulation

5.1 Linux Process Control and Thread Control

In the Linux process model for the RTOS simulation, the Linux parent process calls the *fork()* function to create each Linux child process, and each Linux child process executes by calling the *execlp()* function. Before the Linux parent process terminates, it calls *waitpid()* to wait for all the Linux child processes to terminate.

In each Linux child process of the RTOS simulation, the primary thread creates another thread by calling *pthread_create()*, and these two threads synchronize with each other by using a pthread condition variable globally shared within this Linux child process.

By using Linux multi-process and multi-thread programming, the execution of the RTOS simulation can be controlled in the correct order. For example, the semaphore initialization federate will not join the RTOS federation before the RTOS federation is created; task federates will not join the RTOS federation before semaphores are initialized, the scheduling of tasks will not start before the RTOS kernel federate finishes initializing each task's TCB and sends back each task's Tid.

5.2 Monitoring the RTOS Simulation

In order to monitor the RTOS simulation at runtime, each Linux process has an associated monitoring window. That is, there is a monitoring window for the RTOS kernel federate, a monitoring window for the semaphore initialization, and a monitoring window for each task. In addition to runtime monitoring, each Linux process has an associated log file to save the RTOS simulation results.

The C++ programming language is fast and well structured, but it is time-consuming and error-prone to program a graphical user interface [20]. It is easy to use Tcl/Tk script to write a user interface. However, Tcl/Tk is not suitable for writing well-structured, efficient, and compute-intensive applications. In order to obtain the advantages of both C/C++ and Tcl/Tk, the Embedded Tk system, called "ET" [20], is used in the RTOS simulation for runtime monitoring purpose.

5.3 Conversion Process for the RTOS Simulation

A real-time application program embedded with the WCET is assumed to have several parts: header files included, the declaration and definition for shared variables and semaphores, methods for tasks, and the main program. In a method for a task, there is some code to perform task initialization including establishing the task's scheduling parameters, some code to switch to real-time periodic execution mode, and some code to execute the task periodically. In the code for each task, the WCET is specified before each control point. In the main program, there is some code to initialize semaphores, and some code to create threads for executing tasks.

A real-time application program embedded with the WCET is converted to a simulation application program (consisting of the *sharedObject.hh* file, the *TaskNames.hh* file, the *TaskNames.cpp* file, and the *application.cpp* file) according to the following steps:

1. In the simulation application program, if there are shared variables (also called shared data) and semaphores, shared variables and semaphores are declared in a structure called *shared* in a header file called *sharedObject.hh*. On the other hand, if there are no shared variables and no semaphores, the header file *sharedObject.hh* must be empty.

2. In the simulation application program, the number of task is defined in a header file called *TaskNames.hh*; and task names are stored in the *TaskNames* array in a file called *TaskNames.cpp*.
3. In the simulation application program, the *application.cpp* file includes some header files of the corresponding real-time application program, and some other header files specific to the RTOS simulation.
4. In the simulation application program, if there are shared variables and semaphores, a shared object is defined as *struct shared* type, and the shared object's size is defined as *integer* type in the *application.cpp* file. In the Linux process model for the RTOS simulation, each task and its corresponding task deputy shares a variable named *timeNeed* to pass the WCET to the RTOS kernel federate; moreover, a log file descriptor and a file buffer are declared in the *application.cpp* file as shared variables within each Linux process.
5. In the simulation application program, the semaphore initialization code is in a method called *Sema4_Init()* in the *application.cpp* file. At the end of the *Sema4_Init()* method, the *sema4_init_done()* function will be called to tell the RTOS kernel federate that semaphore initialization is done.
6. In the simulation application program, each task is associated with a method in the *application.cpp* file. The code for a task in the simulation application program

is different from the code for this task in the real-time application program embedded with the WCET as follows: a task's scheduling parameters are set in the method for the task in the real-time application program, while they are not set in the method for the task in the simulation application program embedded with the WCET. At the end of a task's period, the *suspend_until_next_period()* function is always called in the real-time application program embedded with the WCET, while this function is called in the simulation application program when the WCET at the end of the task's period is positive. Whenever the shared object is written, the *write_SO()* function must be called after it in the simulation application program, while this function does not exist in the real-time application program.

7. In the RTOS simulation, each Linux process's execution starts with the *application.cpp* file's main program. The main program includes log file creation code, embedded Tk initialization and termination code, shared variables (in shared memory) initialization code, and *argv[1]* dependent code. If the main program's *argv[1]* is "RTOS", the Linux process for the RTOS kernel federate, which includes the code to call the RTOS kernel federate, will run. When the RTOS kernel federate is called, the scheduling time for tasks is passed as an argument. If the main program's *argv[1]* is "Sema4_Init", the Linux process for the semaphore initialization federate, which includes the code to create a thread for initializing semaphores and the code to call the semaphore initialization deputy, will run. If the main program's *argv[1]* is a task's name, the Linux

process for the task federate, which includes the code to create a thread for executing the task and the code to call the task deputy, will run. When a task deputy is called, the task name and the structure for the task's scheduling parameters are passed as two arguments.

5.4 Generating the RTOS Simulation Executable

Assuming that the RTOS simulation root directory is called the *RTOS* directory, the *Makefile* file under the *RTOS* directory gives the code generation rules. Under the *RTOS* directory, there are three subdirectories: an *src* directory for simulation source files, a *include* directory for simulation include files, an *et* directory for an *et.o* file (this file is needed for ET), and a *data* directory for an *RTOS.fed* file, an RTOS simulation executable file, and simulation log files.

To generate an RTOS simulation executable file *application* under the directory *RTOS/data*, the ET preprocessing command “*make preprocess*” must be first entered in the *RTOS* directory to translate ET source code into genuine C code, and then the command “*make*” must be entered to compile and link files. To run the RTOS executable file, we can simply enter the command “*./application RTOS*” in the directory *RTOS/data*.

5.5 Semaphore

In the RTOS simulation, Posix unnamed semaphores are implemented, which can be used for synchronization and mutual exclusion. Datatype *sem_t* is used to represent semaphores, and it is defined in the header file *semaphore.hh*:

```
#define sem_t int
```

Datatype *sem_t* is defined as integer in that it represents the semaphore id used in the RTOS kernel federate.

Currently, three semaphore functions have been implemented, and they are:

```
#include "semaphore.hh"

int sem_init(sem_t *sem, int shared, unsigned int value); //shared >= 0

int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);
```

Since all tasks terminate at the same time, semaphore destroy function needs not to be implemented.

5.5.1 Semaphore Limits

In the RTOS simulation, two semaphore limits are defined:

`SEM_NSEMS_MAX` the maximum number of semaphores that can be created in the RTOS simulation. Posix requires that this number is at least 256.

`SEM_VALUE_MAX` the maximum value of a semaphore. Posix requires that this value is at least 32767.

In the RTOS simulation, `SEM_NSEMS_MAX` is defined in the header file *Sema.hh*. And `SEM_VALUE_MAX` is defined in the header file *semaphore.hh*.

5.5.2 Semaphore Initialization

The *sem_init()* function can be called in the semaphore initialization thread to initialize the unnamed semaphore referred to by *sem*. The count of the initialized semaphore is *value*. Upon successful completion of the *sem_init()* call, the semaphore *sem* is created in the RTOS kernel federate. Otherwise, if the *sem_init()* call fails, it returns `-1` for one of the following reasons:

The value argument exceeds `SEM_VALUE_MAX`.

The limit on semaphores, SEM_NSEMS_MAX, has been reached.

The argument shared is less than zero.

When the *sem_init()* function is called in the semaphore initialization thread, a message “SemaCreate ” + “sema4Count” will be generated. Then the semaphore initialization deputy will send this message to the RTOS kernel federate. Upon receiving the semaphore initialization message, the RTOS kernel checks whether SEM_NSEMS_MAX has reached. If yes, a semaphore initialization unsuccessful message “Stop” is generated, and this message is sent back to the semaphore initialization deputy. Otherwise, a semaphore is successfully initialized in the RTOS kernel federate, and the RTOS kernel federate sends a semaphore id to the semaphore initialization deputy. In the RTOS kernel federate, each semaphore’s information is maintained as a semaphore control block that includes semaphore id, semaphore count, and semaphore wait queue.

When all the semaphores are initialized, the *sema4_init_done()* function will be called in the method for semaphore initialization to allow the semaphore initialization deputy to send a “Done” message to the RTOS kernel federate. Upon receiving the “Done” message, the RTOS kernel federate creates Linux child processes for tasks. After the semaphore initialization deputy sends the “Done” message, the semaphore initialization federate resigns the RTOS federation.

5.5.3 Semaphore Locking and Unlocking

The *sem_wait()* function can be called by a task, which will result in the semaphore lock operation on the semaphore referred to by *sem*. On the contrary, the *sem_post()* function can be called by a task, which will result in the semaphore unlock operation on the semaphore referred to by *sem*.

When the function *sem_wait()* or the function *sem_post()* is called by a task, a message is formed as “*semaphore id*” + “*semaphore call type*” + “*timeNeeded*”. Then the corresponding task deputy will send this message to the RTOS kernel federate. Upon receiving the semaphore locking or unblocking message, the RTOS kernel federate parses the message and performs correspondingly.

When a semaphore locking operation is performed in the RTOS kernel federate (Figure 13), the semaphore count is decreased first. If the semaphore count is less than zero, the task will be blocked on the semaphore’s wait queue (that is, priority enqueue the task into the semaphore wait queue.) Otherwise, the task will be allowed to pass the semaphore.

Figure 13: Semaphore Locking Operation in the RTOS Kernel Federate

When a semaphore unlocking operation is performed in the RTOS kernel federate (Figure 14), the semaphore count is increased first. If the semaphore value is less than one, the task at the head of the semaphore wait queue will be unblocked.

Figure 14: Semaphore Unlocking Operation in the RTOS Kernel Federate

5.6 Task Management

In the RTOS simulation, the RTOS kernel federate manages the creation of tasks and the destroying of tasks. When creating a task, the RTOS kernel federate initializes a TCB for

the task. When the RTOS simulation stops because the scheduling time limit is exceeded or a task misses its deadline, the RTOS kernel federate sends a “*Stop*” message to each task federate, and then tasks are destroyed.

5.6.1 Task Control Block and Semaphore Control Block

In the RTOS kernel federate, each task’s information is maintained in a TCB, which has attributes: *Tid*, *Priority*, *Period*, *ExeTime*, *TimeLeft* (unscheduled time for the current period), *CurTimeNeeded* (current time needed until the next synchronization point or the end of the period), *CurEventType* (current semaphore call type or the end of the period), *RunTimes* (the number of periods the task has run), *State* (current state of the task), *NST* (next scheduled time of the task), *NSP* (next scheduled task id), *NRP* (next ready task id), and *NBP* (next blocked task id).

In the RTOS kernel federate, each semaphore’s information is maintained in a semaphore control block, which has attributes: *SemaId* (semaphore id), *SemaCount* (semaphore count), and *SemaWaitQ* (task id blocked at the head of the semaphore’s wait queue).

5.6.2 Implementation of Task Queues in the RTOS Kernel

Federate

As mentioned in chapter 4, the RTOS kernel federate manages a time-ordered task queue (future scheduled task queue) and some priority-managed queues (a ready queue, and a wait queue for each semaphore).

In the RTOS kernel federate, a global variable called *NSPQueue* stores the nearest future scheduled task id. In each task's TCB, the *NSP* value is the next task id in the future scheduled task queue, and the *NST* value is the current task's next period scheduled time. The nearest future event's scheduled time is the *NST* value (multiple of the task period) in the nearest future scheduled task's TCB. By using the *NSPQueue* value, the *Tid* value in each task's TCB, the *NSP* value in each task's TCB, and the *NST* value in each task's TCB, the future scheduled task queue can be tracked. For example, at some time in the RTOS simulation, the value of the *NSPQueue* is 1, and three tasks have the following information in their TCBs (Table 3):

Tid	NSP	NST
1	2	10
2	3	15
3	0	30

Table 3: Some Information in the TCBs of Three Tasks

The future scheduled task queue for the example is shown in Figure 15. The NST value in the third task's TCB is 0, and this indicates this task is at the end of the future scheduled task queue.

Figure 15: The Future Scheduled Queue

In the RTOS kernel federate, a global variable called *readyQueue* stores the task id at the head of the ready queue. In each task's TCB, the *NRP* value is the next ready task id. By using the *readyQueue* value, task TCBs' *Tid* values, and task TCBs' *NRP* values, the ready queue can be tracked in a similar way to track the future scheduled task queue.

In the RTOS kernel federate, each semaphore's *waitQ* value in the semaphore control block is the task id blocked at the head of the semaphore's wait queue. . In each task's TCB, the *NBP* value is the next blocked task id. By using a semaphore's *waitQ* value, task TCBs' *Tid* values, and task TCBs' *NBP* values, the semaphore's wait queue can be tracked in a similar way to track the future scheduled task queue.

5.7 Shared Memory

As mentioned in chapter 4, shared memory content is replicated. Whenever a task performs a write operation on the shared data, it notifies other tasks to change their local copies of the shared data by calling the function *write_SO()*.

Chapter 6

Example Application

A producer-consumer example is given in this chapter to illustrate the conversion process for the RTOS simulation described in chapter 5. This example is modeled after the example given in [18]. There is one producer and one consumer using a shared buffer as a circular buffer: The producer fills the buffer from the entry `buff[0]` to `buff[NBUFF-1]`, then it circulates back to `buff[0]` again. The consumer circulates the buffer in the same way, but it reads the buffer instead of filling the buffer. The producer and the consumer use three semaphores for synchronization and mutual exclusion purposes:

1. A binary semaphore named *mutex* protects the critical sections: inserting a data item into the buffer by the producer and removing a data item from the buffer by the consumer. This semaphore is initialized to 1.
2. A counting semaphore named *nempty* counts the number of empty slots in the buffer. This semaphore is initialized to `NBUFF` (the number of slots in the buffer). When the producer inserts a data item into the buffer, *nempty* is decremented by 1. When the consumer removes a data item from the buffer, *nempty* is incremented by 1.

3. A counting semaphore named *nstored* counts the number of filled slots in the buffer. This semaphore is initialized to 0. When the producer inserts a data item into the buffer, *nstored* is incremented by 1. When the consumer removes a data item from the buffer, *nstored* is decremented by 1.

6.1 Characteristics of Tasks

In the real-time application program or the simulation application program, each task's characteristics are specified in an *rtSched_param* structure. The structure is as follows:

```
struct rtSched_param {  
    int rt_tid;  
    int rt_priority;  
    int rt_exeTime;  
    int rt_period;  
};
```

The members of the *rtSched_param* structure are described as follows: the *rt_tid* parameter of a task refers to the task id assigned by the RTOS kernel; the *rt_priority* parameter of a task refers to the task's priority; the *rt_exeTime* parameter of a task refers to the execution time needed in each period of the task; the *rt_period* parameter of a task refers to the task's period.

In the producer-consumer example, the *rt_priority* parameter of the producer is specified to 1 (high priority); the *rt_exeTime* parameter of the producer is specified to 6; the *rt_period* parameter of the producer is specified to 12; the *rt_priority* parameter of the consumer is specified to 2 (low priority); the *rt_exeTime* parameter of the consumer is specified to 6; the *rt_period* parameter of the consumer is specified to 16.

6.2 The Real-Time Application Program

It is assumed that the real-time application program is as follows:

```
..... /* Header files are omitted here.*/

#define NBUFF 3

struct shared {

    int buff[NBUFF];

    sem_t mutex, nempty, nstored;

};

struct shared SO;

void * Producer(void * arg)

{
```

```

    }
}

void main()
{
    pthread_t ProducerThread, ConsumerThread;

    /* Code to initialize semaphores */
    sem_init(&SO.nempty, 0, 3);
    sem_init(&SO.mutex, 0, 1);
    sem_init(&SO.nstored, 0, 0);

    /* Create threads for tasks */
    pthread_create(&ProducerThread, NULL, Producer, NULL);
    pthread_create(&ConsumerThread, NULL, Consumer, NULL);

    return ;
}

```

In a method for a task, there is some code to perform task initialization including establishing the task's scheduling parameters (part A), some code to switch to real-time periodic execution mode (part B), and some code to execute the task periodically (part C). At the end of a task's period, the *suspend_until_next_period* function is called.

6.3 The Simulation Application Program

It is assumed that a timing tool is available to allow the WCET to be added into each task's real-time execution code in the real-time application program (refer to Figure 7 in chapter 4). Then by specifying a scheduling time limit, the real-time application program is transformed to the simulation application program consisting of the *sharedObject.hh* file, the *TaskNames.hh* file, the *TaskNames.cpp* file, and the *application.cpp* file (refer to chapter 5).

In the simulation application program, shared data (an array of integers) and semaphores are declared in the structure *shared* in the header file *sharedObject.hh* as follows:

```
#ifndef SHARED_OBJECT
#define SHARED_OBJECT
#include "semaphore.hh"
#define NBUFF 3
struct shared {
    int buff[NBUFF];
    sem_t nempty, mutex, nstored;
};
#endif
```

In the simulation application program, the header file *TaskNames.hh* defines the number of tasks, and declares an array for task names. The *TaskNames.hh* file is defined as follows:

```
#ifndef TaskNames_HEADER
#define TaskNames_HEADER

#define NUMBER_OF_TASKS 2;

extern char * TaskNames[NUMBER_OF_TASKS];

#endif
```

In the simulation application program, the *TaskNames.cpp* file defines an array storing task names, and it is coded as follows:

```
#include "TaskNames.hh"

char * TaskNames[NUMBER_OF_TASKS] = {

    "Producer",

    "Consumer"

};
```

In the simulation application program, the *application.cpp* file includes some header files, the declaration for some Linux process scope global variables, a method for initializing semaphores, a method for executing each task, and a main program. Each task includes

some code indicating the WCET estimated by a timing tool. The *application.cpp* file is coded as follows:

```
..... /* Header files are omitted here. */

/* Global shared variables within each Linux process. */

#ifdef SHARED_OBJECT

struct shared SO;

int SO_size = sizeof(struct shared);

#endif

extern int timeNeed;

int fd; /* Log file descriptor */

char fileBuf[2000]; /* File buffer */

void * Sema4_Init(void * arg)

{

    ..... /* Code to initialize semaphores */

    sema4_init_done();

}

void * Producer(void * arg)

{

    int i = 0;

    switch_to_rt();
```

```

while (1)
{
    timeNeed = 1; /* The WCET needed */

    sem_wait(&SO.nempty);

    timeNeed = 1;

    sem_wait(&SO.mutex);

    SO.buff[i%NBUFF] = i;

    write_SO(); /* Write shared object notification */

    timeNeed = 2;

    sem_post(&SO.mutex);

    timeNeed = 1;

    sem_post(&SO.nstored);

    i++;

    timeNeed = 1;

    suspend_until_next_period();

}
}

```

```

void * Consumer(void * arg)
{
    int i = 0;

    switch_to_rt();

    while (1)

```

```

    {
        timeNeed = 1;

        sem_wait(&SO.nstored);

        timeNeed = 1;

        sem_wait(&SO.mutex);

        ..... /* Read an item from the buffer, and write it into the
                * consumer task's log file and display window. */

        timeNeed = 2;

        sem_post(&SO.mutex);

        timeNeed = 1;

        sem_post(&SO.nempty);

        i++;

        timeNeed = 1;

        suspend_until_next_period();

    }
}

```

```

int main (int argc, char ** argv)
{
    ..... /* Exception handling, log file Creation,
            * embedded Tk initialization*/

    if ( strcmp(argv[1], "RTOS") == 0 )
    {

```

```

        /* Call the RTOS kernel federate */

        OS_Federate(300);    /* Scheduling time limit = 300 */

        ..... /* Wait for Linux child processes die first */

    }

    else if ( strcmp(argv[1], "Sema4_Init") == 0 )

    {

        ..... /* Create a thread for executing Sema4_Init( ) */

        /* Call the semaphore initialization deputy */

        Sema4Init_Deputy( );

    }

    else if ( strcmp(argv[1], TaskNames[0]) == 0 )    /* Producer */

    {

        ..... /* Create a thread for executing Producer( ) */

        ..... /* Specify this task's scheduling parameters */

        /* Call this task deputy */

        Task_Deputy(argv[1], &schedParam);

    }

    else if ..... /* Consumer */

    ..... /* Embedded Tk termination */

    return 0;

}

```

In the RTOS simulation, the scheduled time limit is specified to 300 time units when the RTOS kernel *federate* is called. At simulated time 168, the producer should change its state to *ready* to be eligible for its 14th period's execution while its current state is *run* indicating its unfinished 13th period's execution. As a result, the producer misses its deadline, and the RTOS simulation stops.

Chapter 7

Conclusion and Future Work

This chapter concludes the work presented in this thesis, and gives a few future work directions.

7.1 Conclusion

Simulation plays an important role during the development of an embedded system in that it could reduce cost, time to market, and system development risk. With the increasing of embedded application demands, embedded systems are being built from increasingly larger scale components. This leads to increasing attention to component interoperability. Motivated by the need for component-based simulations, this thesis has conducted research on an HLA compliant RTOS simulation.

In this thesis, a multitasking RTOS simulation system has been built on the HLA. The HLA compliant RTOS simulation is comprised of an RTOS kernel federate, a semaphore initialization federate, and task federates. By using the HLA RTI services, federates communicate transparently.

A real-time application program designed for a target platform can be transformed to a simulation application program to run on a host workstation. In the transformation, each task's real-time execution code in the real-time application program must be first analyzed by an available timing tool to add in the WCET before each control point. Then the real-time application program embedded with the WCET is transformed to the simulation application program.

A general, reusable RTOS kernel has been implemented to simulate periodic static priority-based preemptive scheduling, and semaphores; moreover, shared memory is also simulated in the HLA compliant RTOS simulation. By using the simulated RTOS kernel services, a simulation application program can run on a host workstation. This allows embedded software to be developed and tested without the presence of target hardware.

The scheduler implemented in the HLA compliant RTOS simulator allows experimenting and analyzing the schedulability of a set of tasks in a hard real-time system. In the case that task priorities are assigned in rate monotonic order and tasks do not communicate via ITC mechanisms, the scheduler is a rate monotonic scheduler.

A producer-consumer example was given to demonstrate the transformation from a real-time application program to a simulation application program.

7.2 Future Work

Future work for the HLA compliant RTOS simulation includes automatic code transformation, extended RTOS functionality, integration of the I/O simulation, and RTOS simulation of multiprocessor real-time systems.

7.2.1 Automatic Code Transformation

In this thesis, a real-time application program is transformed to a simulation program manually. In the future, a transformation tool can be developed to allow automatic transformation from a real-time application program to a simulation application program.

7.2.2 Extended RTOS Functionality

The simulated RTOS kernel can be changed to support scheduling algorithms different from the one in this thesis, to provide solutions to the unbounded priority inversion problem, and to support message queues and mailboxes.

7.2.2.1 Scheduling

The simulated RTOS kernel could be changed to schedule periodic tasks whose deadlines are different from their periods, and to schedule non-periodic tasks. At the application level, a periodic task whose deadline is different from its period has four scheduling parameters: priority, execution time in each period, period, and deadline. A non-periodic task has one scheduling parameter priority, and events associated with the task can be read from a file.

7.2.2.2 Solutions to the Unbounded Priority Inversion Problem

As mentioned in chapter 2, when synchronization mechanisms are applied with priority-driven preemptive scheduling mechanisms, unbounded priority inversion might happen. The Priority Inheritance Protocol (PIP) provides a solution for unbounded priority inversion. In the PIP scheme, when a high priority task is trying to access a resource that is held by a low priority task, the low priority task's priority will be boosted up to the priority of the high priority task. By doing so, a medium priority task cannot preempt the low priority task that is holding the resource requested by the high priority task, and the high priority task only has to wait until the low priority task releases the resource and reverts back to its original priority. The PIP solves the priority inversion problem. However, there are still two potential problems: chain blocking problem and deadlock problem. Chain blocking happens when a high priority task shares resources with a number of low priority tasks and becomes blocked whenever it attempts to access a

critical section. As a result, chain blocking may result in substantial blocking time. Deadlock occurs when all tasks are blocked waiting for another.

The Priority Ceiling Protocol (PCP) provides another solution for unbounded priority inversion. In the PCP scheme, each critical section has a priority ceiling, which is the priority of the highest priority task that will enter the critical section. A task cannot enter a critical section unless its priority is higher than the ceiling of all other currently occupied critical sections. When a low priority task blocks a high priority task, priority inheritance is applied. With the PCP, deadline cannot happen because if a task enters a critical section, this critical section must not be currently occupied by another task; and chain blocking cannot happen because a task can be blocked for at most the duration of one critical section. However, the PCP requires that each critical section's ceiling priority is known a priori. Furthermore, it not only introduces push-through blocking of the PIP (a medium priority task can be blocked by a low priority task with boosted high priority), but also introduces priority-ceiling blocking. As a result, the PCP is very complex and costly to implement in reality.

7.2.3. Integration of the I/O Simulation

I/O simulation could be integrated into the RTOS simulation by implementing an HLA compliant device driver simulation component at the application level or at the operating system level.

7.2.4 RTOS Simulation of Multiprocessor Real-Time Systems

In a multiprocessor system, multiple RTOS kernels manage tasks by sharing RTOS kernel data structures.

Bibliography

- [1] Manthos A. Tsoukarellas, Vasilis C. Gerogiannis, Kostis D. Economides. *"Systematically Testing a Real-Time Operating System"*, IEEE Micro, Vol. 15, No. 5, pp. 50-60, October 1995.
- [2] Gianluigi Castelli, Gianluca Ragazzini. *"EOS: A Real-Time Operating System Adapts to Application Architectures"*, IEEE Micro, Vol. 15, No. 5, pp. 41-49, October 1995.
- [3] Krithi Ramamritham, John A. Stankovic. *"Scheduling Algorithms and Operating Systems Support for Real-Time Systems"*, Proceedings of the IEEE, Vol. 82, No. 1, pp. 55-67, January 1994.
- [4] Dennis W. Leinbaugh. *"Guaranteed Response Time in a Hard-Real-Time Environment"*, IEEE Transactions on Software Engineering, Vol. 6, No. 1, pp. 85-91, January 1980.
- [5] C. L. Liu, J. W. Layland. *"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"*, Journal of the ACM, Vol. 20, No. 1, pp. 46-61, 1973.
- [6] Duenas, J.C.; Leon, G. *"Simulation of Real Time Scheduling at Design Levels"*, 7th IEEE Euromicro Workshop on Real-Time Systems, pp. 140 –146, March 1995.
- [7] Stankovic, J.A.; Ramamritham, K. *"The Spring Kernel: A New Paradigm for Real-Time Systems"*, IEEE Software, Vol. 8, No. 3, pp. 62 –72, May 1991.
- [8] Stewart, D.B.; Schmitz, D.E.; Khosla, P.K. *"The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications"*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 22, No. 6, pp. 1282 –1295, November/December 1992.
- [9] IEEE Standard 1003.1b-1993 POSIX Part1: System API- Amend. 1: Realtime Extension [C Language].
- [10] IEEE Standard 1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules.

- [11] IEEE Standard 1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification.
- [12] IEEE Standard 1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template.
- [13] Department of Defense, Defense Modeling and Simulation Office. “*High Level Architecture Run-Time Infrastructure: RTI 1.3-Next Generation Programmer’s Guide Version 4*”, online at <https://www.dmsomil/public/transition/hla/>.
- [14] The HLA Working Group. “*HLA Standards Development*”, online at <http://www.sisostds.org/stdsdev/hla/>.
- [15] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. “*Creating Computer Simulation Systems: An Introduction to the High Level Architecture*”, Prentice Hall PTR, 2000.
- [16] Leskela, J.; Salmela, M.; Hyvarinen, J.; Heikkinen, M. “*Visualised Real-Time Operating System Simulation in a Heterogeneous Prototyping Environment*”, 8th IEEE Euromicro Workshop on Real-Time Systems, pp. 237 –242, June 1996.
- [17] Leskela, J.; Salmela, M.; Heikkinen, M.; Hyvarinen, J. “*Visualisation of Real-Time Software in Host-Based Simulation Environment*”, 7th IEEE Euromicro Workshop on Real-Time Systems, pp. 42 –48, March 1995.
- [18] Rendon, A.; Duenas, J.C.; de Miguel, M.A.; Leskela, J.; de la Puente, J.A.; Leon, G.; Alonso, A. “*Animation of Heterogeneous Prototypes of Real-Time Systems*”, 1st IEEE International Conference on Engineering of Complex Computer Systems, pp. 47 –54, November 1995.
- [19] Department of Defense, Defense Modeling and Simulation Office. “*High Level Architecture Run-Time Infrastructure: RTI 1.3-Next Generation Programmer’s Guide Version 4, Appendix A*”, online at <https://www.dmsomil/public/transition/hla/>.
- [20] D. Richard Hipp. “*An Introduction To Embedded Tk*”, online at <http://www.comnets.rwth-aachen.de/doc/ET/ET.html>.
- [21] Stankovic, J.A. “*Misconceptions about Real-Time Computing: a Serious Problem for Next-Generation Systems*”, IEEE Computer, Vol. 21, No. 10, pp. 10 –19, October 1988.

- [22] David L. Parnas, and Paul C. Clements. "*A Rational Design Process: How and Why to Fake it*", IEEE Transactions on Software Engineering. Vol. SE-12. No.2, pp. 251-257, February 1986.
- [23] Gerard Fianen, and David Kalinsky. "*Selecting a development platform for RTOS*", OSE Systems, ECN Magazine, 3/1/2002.
- [24] David Stepner, Nagarajan Rajan, and David Hui. "*Embedded Application Design Using a Real-Time OS*", DAC 99, New Orleans LA, 1999.
- [25] Robert Hill, Balaji Srinivasan, Shyam Pather, and Douglas Niehaus. "*Temporal Resolution and Real-Time Extensions to Linux*", Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, Department of Electrical Engineering and Computer Science, University of Kansas, June 1998.
- [26] Alexander D. Stoyenko, V. Carl Hamacher, and Richard C. Holt. "*Analyzing Hard Real-Time Programs for Guaranteed Schedulability*", IEEE Transactions on Software Engineering. Vol. 17, No. 8, August 1991.
- [27] Brian Goetz. "*Threading lightly, Part 1: Synchronization is not the enemy*", online at <http://www-106.ibm.com/developerworks/java/library/j-threads1.html>, July 2001.
- [28] "*FSM: The RTLinux[®] Company RTLinux FAQ*", online at <http://www.fsmlabs.com/developers/faq/>, May 2001.
- [29] A. K. Mok. "*Evaluating Tight Execution Time Bounds of Programs by Annotations*", 6th IEEE Workshop on Real-Time Operating Systems and Software, pp. 74-80, May 1989.
- [30] ChangYun Park, and Alan C. Shaw. "*Experiments with a Program Timing Tool Based on Source-Level Timing Schema*", IEEE Computer, Vol. 24, No. 5, pp. 48 -57, May 1991.
- [31] N. Zhang, A. Burns, and M. Nicholson. "*Pipelined Processors and Worst Case Execution Times*", Journal of Real-Time Systems, Vol. 5, No. 4, October 1993.
- [32] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon, "*Bounding worst-case instruction cache performance*", Proceedings of the 15th IEEE Real-Time Systems Symposium, pp. 172-181, December 1994.

- [33] S. S. Lim et al. "*An Accurate Worst Case Timing Analysis for RISC Processors*", IEEE Transactions on Software Engineering, Vol. 21, No. 7, pp. 593-628, July 1995.
- [34] "*CarbonKernel Real-time Operating System Simulator Version 1.3 User Manual*", online at <http://www.carbonkernel.org/>.
- [35] E. L. Lawler and C. U. Martel. "*Scheduling Periodically Occurring Tasks on Multiple Processors*", Information Processing Letters, Vol. 12, No. 1, February 1981.
- [36] Tokuda, H.; Kotera, M. "*Scheduler 1-2-3: an interactive schedulability analyzer for real-time systems*", Proceedings of the 12th IEEE Computer Software and Applications Conference, pp. 211–219, 1988.
- [37] IEEE. Threads Extension for Portable Operating Systems to POSIX. Draft IEEE Standard P1003.1c., April 1993.
- [38] "*VxWorks Programmers Guide*", Wind River Systems Inc., 1993
- [39] M. Lajolo, C. Passerone, F. Bellifemine, A. Bonomo, P. Civera, G. Ghigo, A. Sangiovanni-Vincentelli. "*Hardware/Software Co-Design for Image Processing*", Proceedings of International Conference on Signal Processing and Communications, February 1998.
- [40] S. Schulz. "*A Transformation for System Level Design Model Specifications into Implementation Descriptions*", XXI International Conference of the Chilean Computer Science Society (SCCC '01), pp. 247-255, November 2001.
- [41] Wind River. "*VxSim*", online at <http://www.windriver.com/products/html/vxsim.html>.
- [42] Martin T. "*Definition Real-Time Systems*", online at <http://www.realtime-info.be/encyc/techno/terms/defini/def.htm#realtime>.
- [43] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. "*Design of Embedded Systems: Formal Models, Validation and Synthesis*", online at <http://www-cad.eecs.berkeley.edu/~polis/paper/1997/proceed97.pdf>.