Modeling of Preemptive RTOS Scheduler with Priority Inheritance

Karim Abdul Khalek

A Thesis

In

The Department

of

Electrical and Computer Engineering,

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Applied Science (Electrical and Computer Engineering)

at

Concordia University

Montreal, Quebec, Canada

2013

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Karim Abdul Khalek

Entitled: "Modeling of Preemptive RTOS Scheduler with Priority Inheritance"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Zahangir Kabir

_____ Examiner, External
Dr. Nizar Bouguila To the Program

_____ Examiner
Dr. Yan Liu

_____ Supervisor
Dr. Samar Abdi

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

___November 12, 2013_____ _____
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

ABSTRACT

Modeling of Preemptive RTOS Scheduler with Priority Inheritance

Karim Abdul Khalek

This work describes an approach to generate accurate system-level model of embedded software on a targeted Real-Time Operating System (RTOS). We design a RTOS emulation layer, called RTOS_SC, on top of the SystemC kernel. The system level model can be used for software optimization in the early stage of a processor design. The model precision is obtained by integrating key features which are provided in typical RTOS schedulers. We first discuss a case study which shows the impact of the implemented features on a priority-driven scheduler. We then present the abstraction of tasks scheduling and communication mechanisms. To validate the accuracy of our model we use the tasks response time metric with industrial-size examples such as MP3, Vocoder and Jpeg encoder. The experimental results show a significant improvement compared to existing RTOS models.

Acknowledgments

I would like to take the opportunity to thank my supervisor, Dr. Samar Abdi, for being a support and a guide to me throughout the two years which I have spent on research. Dr. Samar has been more than a supervisor to me, he taught me to always be confident and think about the problems as challenges for me to develop my skills and knowledge. Most importantly, he trusts me and each of his students, which encourages us to be more productive and prove to ourselves and him that we are trustworthy. I am honored to work with him and have as my supervisor.

I would like as well to thank my father and mother, who encouraged me to keep on working hard and always look forward to more achievements. They have been supporting and believing in my ability to reach my goals.

I would also like to thank my brother Wassim for involving himself with every decision I take and always being there for me. Finally, I thank my close friends Abdulla, Kazim, Paul, Richard, Zaid, Partha, Ehsan, Ali B, Sethu, Tushar, Ali H, Ali N, Patrick, and Sanaya for all the advices they gave me.

To all

Contents

List of Figures

List of Listings

List of Tables

List of Acronyms

RTOS	Real-Time Operating System
RTOS_SC	Real-Time Operating System Emulator
VP	Virtual Platform
ISS	Instruction-Set Simulator
SW	Software
HW	Hardware
FIFO	First-In-First-Out
RR	Round-Robin
RM	Rate-Monotonic

CHAPTER 1

INTRODUCTION

Today, with the wide variety of processors and increasing development of new hardware platforms, such as OMAP, Tegra 2, Hummingbird, and Snapdragon, embedded software developers are forced to transition to such platforms in order to enhance the system performance. In addition, system designers are required to maintain fast software/hardware integration and short time-to-market. Therefore, a system validation is needed before the hardware becomes available to the designers. To realize this goal, many software implemented models of the hardware are introduced, such as virtual platforms (VPs). Software designers may profit from such models by developing the RTOS in parallel with the hardware instead of waiting for the physical hardware to be delivered.

VPs, based on instruction-set simulators (ISS), are widely used to develop and optimize embedded software before the hardware is available. The software can be compiled and executed on the VP similar to execution on the real hardware platform. To overcome the slow simulation of an interpretive ISS, VPs based on binary translation has been introduced [2-4]. This approach consists of translating the legacy code instructions from host to target. Dynamic binary translation is a commonly used technique since it enables easy translation to different hardware architecture, compared to the static binary translation [2]. With such models available early in the hardware development stage, the time for software and hardware integration is reduced.

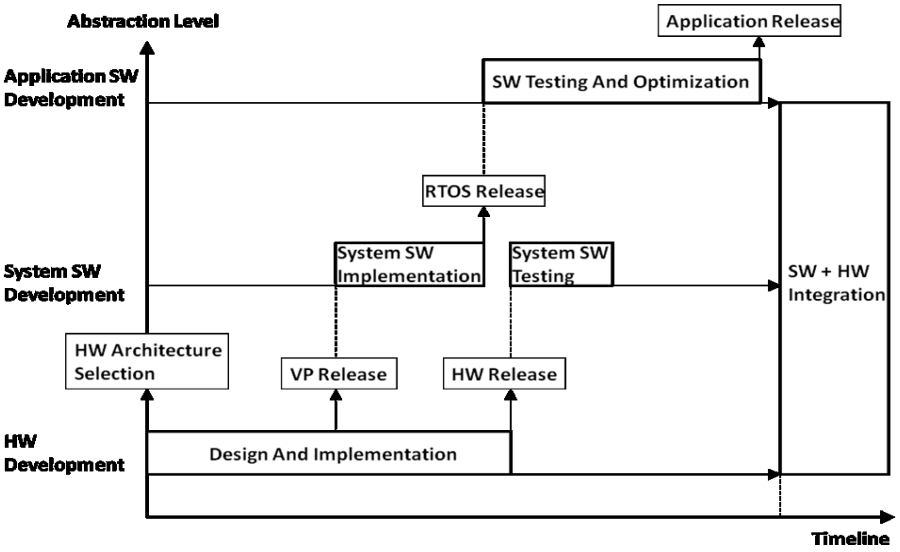


Figure 1.1 Traditional SW/HW Design flow

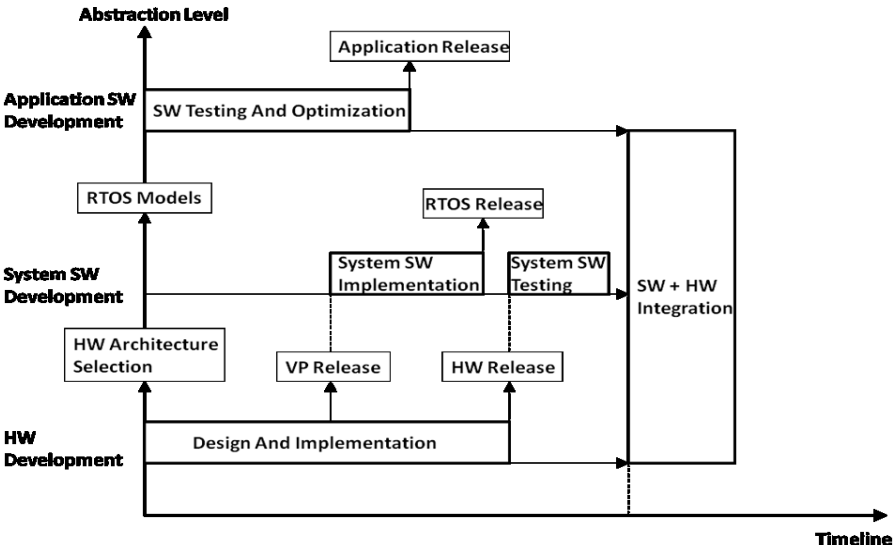


Figure 1.2 SW/HW Design flow with RTOS models

In a traditional software and hardware design flow, shown in Figure 1.1, the hardware implementation stage starts when the hardware architecture is selected. VPs may be available prior to the hardware release, which allows system software developers to start early implementation of the RTOS on the target hardware. Upon the release of the hardware silicon, RTOS testing can be done on the silicon. On the other hand, the legacy application software architecture may require optimization to take advantage of the new

HW/SW platforms. Although hardware may be modeled at different levels of abstraction for speed and accuracy tradeoff [3], application developers cannot benefit from such models without the incorporation of an RTOS model.

In a conventional design flow where RTOS models are not available, RTOS implementation must be completed on the VP, before testing and performance analysis of the applications. Application code depends on RTOS services, such as hardware resource accesses and task management. As a consequence, application testing and optimization may not start until the end of the RTOS implementation. This delay leads to an increase in the product's time-to-market and hence, the additional pressure to deliver the product on time. To avoid the time-to-market pressure and identify the optimization opportunities, host-compiled models of the software executing on the hardware, and the software's interaction with other system components must be created early in the design process.

Figure 1.2 illustrates the software and hardware design flow when RTOS models are introduced. Such models are independent from the VPs. To identify the optimization opportunities in the legacy code, an RTOS model is required to model the key concepts available in an RTOS, such as preemption and synchronization services. Furthermore, applications can be directly linked to these models and tested independently of the RTOS. Hence, software exploration and optimization can happen in parallel with the hardware and RTOS development. The time until the software and hardware integration is therefore reduced compared to the conventional design flow.

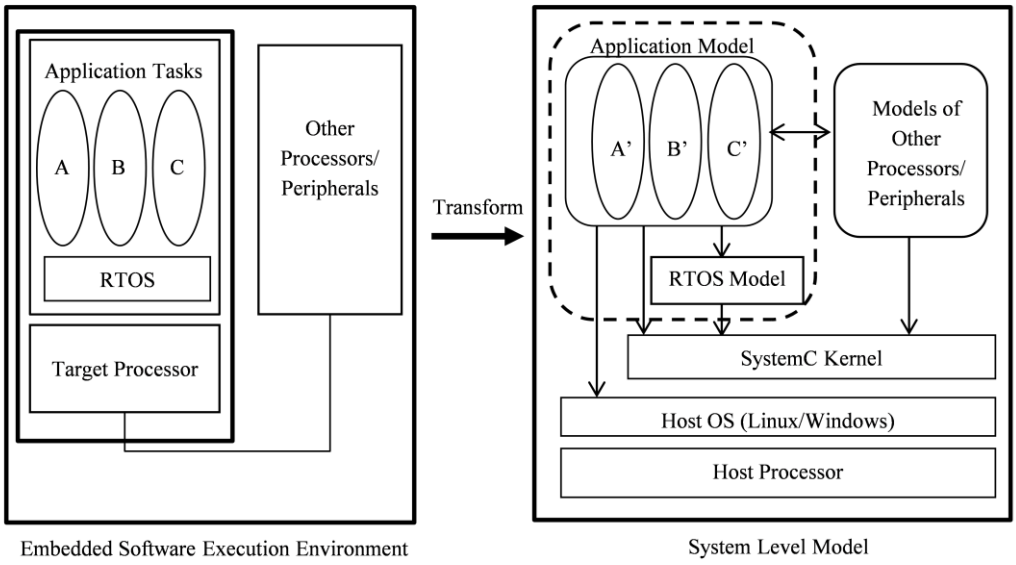


Figure 1.3 Modeling Methodology

1.1 Methodology

Figure 1.3 illustrates our modeling methodology. We start with the application software code and platform specification, shown on the LHS, and derive a functionally equivalent SystemC model, shown on the RHS. The application software is targeted to a specific RTOS, running on an embedded x86 processor. Concurrent application tasks *A, B,* and *C*, are captured as processes or threads in an RTOS. Inter-task communication is implemented using a message passing API, as in typical RTOSes. The target processor is typically part of a larger hardware platform consisting of other system components such as processors, DSPs, custom hardware accelerators, memories and I/O. It is assumed that executable C/SystemC models of other system components are available.

On the RHS, we have a derived model of the system, which can be compiled and executed on the host (typically a PC running Linux) before the hardware is available.

Here, *A′, B′,* and *C′* are functionally equivalent abstractions of *A, B,* and *C*, and are implemented as SystemC threads. Since SystemC does not natively support any RTOS primitives for scheduling and communication, an RTOS emulation layer on SystemC, named RTOS_SC, is modeled on top of the SystemC kernel.

SystemC natively supports timing and events; therefore, timers and pulses in the application can be abstracted using native SystemC constructs. This part of the model has a direct dependency on the SystemC libraries. Finally, we do not explicitly model the memory management of the target platform or the I/O needed for debugging. These services are used from the run-time system available on the host. A structured model with clear semantics is understandable, maintainable and amenable to automation.

1.2 Contribution

One of the problems a priority-based scheduler has to deal with is priority inversion, a very important issue that affects the functionality of the applications. This phenomenon may cause tasks in soft and hard real-time applications to miss their deadlines. Therefore, modern RTOSs solve this issue by implementing the following two features: (i) preemption and (ii) inter-task communication protocols. In this paper, we present an RTOS model that, unlike previous ones, detects and avoids priority-inversion in priority-based multithreaded applications. To achieve this goal, we create a model of the CPU time consumption and task communication, which incorporate features (i) and (ii). Therefore, we present a methodology to accurately model preemption in an SLDL-based RTOS model. Furthermore, we developed a model of priority inheritance protocol which avoids priority inversion during inter-task and I/O communication.

Our model can be used by application developers for early validation and optimization

of the software. We use the task's response time to measure performance and validate the functional correctness of the embedded software. This metric represents the time duration from the occurrence of the event until the time the task produces its results. Modeling features (i) and (ii) leads to an accurate modeling of task's response time, and therefore, accurate performance estimation can be obtained.

In order to identify the optimization opportunities in soft and hard real-time applications, a generic RTOS model is required. In this paper, we provide a general model of the RTOS by modeling the following RTOS features in addition: software timers, different communication mechanisms and different scheduling algorithms, such as First-In-First-Out (FIFO), Round-Robin (RR) and Rate-Monotonic (RM).

1.2.1 Preemption Modeling

During software execution, application tasks may be blocked to wait for particular resources, such as events and timers. When a resource is granted, the task is unblocked and becomes ready to execute. Consequently, the RTOS selects a ready task to execute, according to the algorithm specified by the user. The selection of the task to execute, called rescheduling, always occurs before a running task is blocked on a resource. Further, in RTOSs where a priority-driven scheduler is implemented, rescheduling also occurs during task execution to ensure that the task with the highest priority is not delayed from consuming CPU time, by a lower priority task. If a ready task has a higher priority over the one running, preemption occurs.

Preemption is the temporary interruption of the task execution to give the CPU access to a ready task with higher or equal priority. The preempted task is selected to run again if it is the one with the highest priority among the ready tasks when rescheduling occurs.

Modeling a preemptive scheduler in an RTOS model provides two essential improvements:

- Support of priority-driven scheduling at any time during simulation, as in a typical RTOS [1].
- Abstraction of asynchronous interrupts and timer pulses, which leads to accurate modeling of the software execution time.

1.2.2 Priority Inheritance Modeling

During message passing, preemption by itself may not be sufficient to maintain the priority-based scheduling, due to the priority inversion phenomenon. Priority inversion may occur when acknowledgment protocols, such as the double hand-shaking, are used. These protocols cause tasks to be blocked, waiting to receive acknowledgments of the data transmission. While these protocols are required to ensure reliable communication between tasks, they lead to unexpected priority inversion scenarios by blocking the tasks' execution during message passing. For instance, priority inversion may be encountered when the sender task suspends execution to wait for a signal that indicates, to the receiver task, a successful data transmission. During the sender's suspension, the receiver is required to process the sender's request. As a result, the sender effectively operates at the receiver's priority, which may differ from the original sender priority. Therefore, the execution time of other ready tasks may be delayed by this change in priority. This phenomenon is called priority inversion. One of the essential RTOS features to ensure accurate context switching during tasks communication is priority inheritance. It is applied by assigning the priority of the receiver task to that of the sender.

1.2.3 Software Timers

Once preemption is modeled, software timers, an important feature in RTOSes, can be modeled. The timer pulses enable the generation of interrupts on a timely basis. This technique is commonly used in drivers to access peripherals and obtain data. Therefore in our RTOS model, we implement a timer class which enables periodic tasks to be created.

1.2.4 Scheduling Policies

To meet the application requirements, such as deadlines for a job completion, software developers may optimize the performance by changing the scheduling algorithm. We provide three different policies that can be used in our RTOS model:

- First-In-First-Out (FIFO): This policy is the most commonly used in priority-based applications since it enables preemption and guarantees low CPU overhead.
- Round-Robin (RR): This policy is assigned for tasks that may cause starvation due to their high computation time. A time-slice is given to each task to complete its job. If the job is not completed within this time, rescheduling occurs to allow preemption.
- Rate-Monotonic (RM): This policy is assigned to periodic tasks which have deadlines. The priority of the task depends on the period of the task. The task with the shortest period is assigned the highest priority.

1.2.5 Communication mechanisms

We increase the optimization opportunities in an application by modeling the common communication mechanisms implemented in RTOSes:

- Channels: Model reliable inter-task communication by transmitting data through

channels, to which different protocols can be modeled.

- Queues: Model message passing with less context switching but less reliability.
- Global Variables: Model I/O communication, with synchronization services such as mutual exclusion, semaphores, conditional variables and barriers.

1.3 Related Work

To allow software designers to perform design space exploration and early validation of real-time constraints, such as deadlines, a lot of work has been done to model the RTOS executing on the target platform. The techniques used are divided into three categories: (i) RTOS models based on specific simulation engines, (ii) clock-based RTOS models and (iii) event-based RTOS models.

1.3.1 RTOS models based on specific simulation engines

Some RTOS models have been designed to target a specific RTOS [5-7]. With such executable models, the application performance on a single RTOS may be accurately measured. Therefore, the supported features and the scheduling algorithm can typically be implemented as the target RTOS. However, these models can only adapt to the selected RTOS, which limits the design space exploration. To solve this problem, generic RTOS models based on specific simulation engines have been provided in [8-9]. These models, however, prohibit the integration of such models with other hardware models, which may themselves use other simulation engines.

1.3.2 Clock-based RTOS models

To avoid restrictions to a specific RTOS and complexity in SW/HW integration, generic RTOS models based on SLDL have been introduced [10-20]. Based on these

languages, some of the proposed techniques use a clock-based approach to model an RTOS. The simulation clock in such models is advanced in fixed micro steps for which the user determines the interval. The accuracy of preemption modeling in these approaches depends on the clock step value, and hence, performance estimation can be done at a coarse-grained level. In [10], threads sharing the same resources, such as CPU and memory accesses, were given a static time-slice to complete their execution. Depending on the task priority, preemption may occur at the end of a time-slice. Using this technique, application tasks that run the round-robin scheduling algorithm are accurately modeled. However, hardware interrupts and events are not instantly handled since preemption happens only at the end of the time-slice.

1.3.3 Event-based RTOS models

To model preemption with better accuracy, a technique has been proposed to decrease the time interval between the clock micro-steps [11]. By reducing the interval, however, the simulation time of the application increases. Therefore, another technique has been proposed to dynamically change the clock- step value Δ to the event arrival time Tin, if the current simulation time $C + \Delta > Tin$ [12]. Consequently, the remaining time $(C + \Delta)$ - Tin is added to the new value of Δ, to obtain the original clock-step value. An accurate estimate of Tin is obtained by assigning it to the earliest time at which an event may be sent to the corresponding task. Furthermore, to obtain an accurate estimation of Tin, the time interval during which the task generating the event is preempted is added to the estimated time of the next input event. The drawback to this technique is that the additional time delay caused by asynchronous communication such as shared memory accesses is not taken into consideration. Moreover, the time for the interrupt occurrence

must be known and specified by the application developer in the specification model, adding a limitation to the model.

In order to avoid the complexity of selecting the appropriate clock-step value that minimizes preemption latency in a clock-based approach, previous techniques based on a finer-grained time annotation have been introduced for RTOS modeling [13-20]. This approach consists of dividing the application user code into segments in each of which a "wait" statement is introduced to model the time it consumes. Techniques that follow this approach provide more control over task management and hence accurate preemption modeling.

In [13], a technique has been proposed to model a generic RTOS with a preemptive, priority-based scheduler and task communication through channels. Based on the specification model, a dynamic scheduling step is performed to refine the behavioral model into an architectural model. A different approach was presented by Hessel for software and hardware synthesis [16]. Some work has been done to obtain more accurate results of the application execution time by modeling the RTOS overhead, which is caused by scheduling and context switching [14-16]. Some of these models provide a power estimation of the application running on the selected processor. Moreover, they implement different scheduling policies and communication mechanisms. However, the proposed methods do not accurately model preemption, since they depend on the timing annotation granularity. If an interrupt occurs during time consumption, it is processed at the end of the required time consumption and therefore delays preemption. Moreover, these models do not take priority inversion, which affects the functionality of priority-based schedulers, into consideration.

To improve the interrupt response time modeling, time modeling using a dynamic time annotation technique has been proposed [18-19]. In [18], Posadas presents a technique to enable preemption at the interrupt arrival time by changing the length of the code segment during the software run-time. The segment scope ends either when a call to a kernel function is reached or a timer interrupt occurs. Interrupts with unpredictable arrival times, are handled at the end of the segment by dividing the annotations into two steps: the time until the interrupt arrives and the time remaining for the task to complete its initial time required. Although interrupts are accurately modeled in the proposed methods, a dynamic estimation of the interrupt arrival time must be performed during simulation, adding complexity to the model. In [1], a methodology has been suggested to model preemption by interrupting the "wait" statements. Consequently, no calculation is needed for the interrupt arrival time. However, none of these methods take into account priority inheritance, which avoids priority inversion during communication.

A different solution to preemption modeling has been presented in [20]. When an interrupt occurs during time consumption of a task, the task generating the interrupt - the preemptor - is scheduled without interrupting the current task from its "wait" statement. Upon completion of the preemptor's time consumption, the preemption time interval of the preempted task is modeled by waiting for the time interval which was consumed by the preemptor. In case the preempted task completes its time execution before the preemptor does, the preempted task suspends until its preemption time is known. Compared to the immediate preemption approach in [18], this approach increases simulation speed by reducing the number of calls for "wait" statements in the model. However, the corrective measures taken during simulation to model the tasks' preemption

time produce an inaccurate representation of the internal states of the processes. Due to this impact, the internal states are hidden in the model. As a result, it becomes harder for software designers to debug their application, providing less optimization opportunities. Furthermore, the RTOS model may not be feasible when software complexity increases, due to the excessive amount of computation produced.

In our model, we provide an accurate preemptive RTOS model; based on time annotation granularity. We avoid priority inversion during communication by implementing the priority inheritance protocol, which is available in modern RTOSes. Furthermore, we improve the preemption modeling presented in [1] by providing a solution to model multiple interrupts that can occur at the same logical time in an SLDL-based RTOS model. Consequently, it is possible to obtain functional verification of soft and hard real-time applications running on a target processor.

1.4 Thesis Organization

The thesis is organized as follows: In Chapter 2, we present our technique of modeling a basic preemptive RTOS model. Further in this chapter, we discuss the modeling of interrupts based on timer pulses. We provide an example that shows the effect of priority inheritance on the accuracy of the RTOS model in measuring application performance. We explain in Chapter 3, our methodology to accurately model preemptive RTOSs, with our proposed methodology of modeling priority inheritance to solve priority inversion during inter-task communication. In Chapter 4, we show the accuracy of our preemptive RTOS model with priority inheritance through experimental results. We also demonstrate how the model can be used for early, fast and accurate software validation and optimization.

CHAPTER 2

PREEMPTIVE RTOS MODELING

Since SystemC does not natively support any RTOS primitives for scheduling and communication, a model of an RTOS Emulation layer on top of the SystemC kernel is needed. A classical preemptive RTOS model supports priority-based scheduling and inter-task communication which are available in a typical RTOS. On the other hand, SystemC provides a support for timing and events. Therefore, SystemC constructs can be used in an application to abstract timers and pulses [19]. In a preemptive RTOS model, the tasks are modeled as SC_THREADs and dynamically created using the SystemC kernel function. Further, since task priorities are not supported by the SystemC kernel, a *Task_SC* class is created. Task_SC has a task ID, a priority, an activation event, a state and a pointer to the SystemC process handle.

When a software application is linked against the RTOS model, two main aspects must be provided: correct event trace and accurate execution time. To meet these conditions, the RTOS scheduler should always select the thread with the highest priority to execute before the lower priority threads. However, the thread selection must be among threads which are not blocked on a resource.

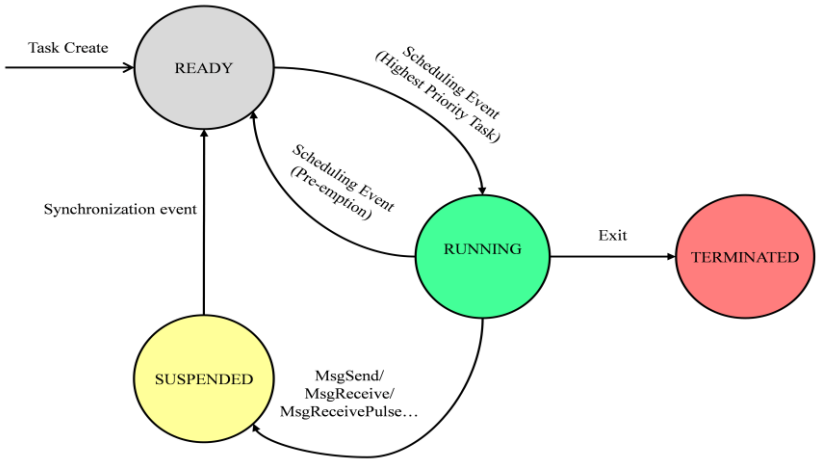


Figure 2.1 Task states in classical SystemC RTOS Model

To differentiate between blocked and unblocked threads, a preemptive RTOS model must have four different states defined for each thread as shown in Figure 2.1: *RUNNING*, *READY*, *SUSPENDED* and *TERMINATED*. When the thread is created, it is moved to a READY state. Among the ready tasks, the one with the highest priority is selected to execute and hence moved to a RUNNING state. A thread is moved from a SUSPENDED state to a READY state as soon as its access to the resource is granted. A thread moves from a RUNNING state to a SUSPENDED state when it waits for a specific event. It is finally moved to a TERMINATED state when it accomplishes its task. It is also possible that a thread is moved from a RUNNING state to a READY state if preemption occurs.

```
void RTOS::RunningToReady ()
1:  RTOS_task *t = ACTIVE;
2:  ACTIVE = GetHighestReady();
3:  t→State = READY;
4:  if (ACTIVE != t) {
5:     ACTIVE→Activation.notify();
6:     wait (t→Activation);
7:  } // end if
```

Listing 2.1 Transition from running to ready

2.1 Scheduler Modeling in A Preemptive RTOS Model

2.1.1 State Transition

To implement the transitions states shown in Figure 2.1, private functions must be defined in the scheduler class to obtain a preemptive RTOS model. Listing 2.1shows the pseudo-code for transitioning from running to ready state. ACTIVE indicates the currently running task and is stored in task pointer t. The ACTIVE task pointer is changed to the highest priority ready task and set to RUNNING state by calling the *GetHighestReady* function.

If the READY list is not empty and the returned READY task is different from the caller, its *Activation* event is notified to enable its execution. The caller task's state is changed from RUNNING to READY and waits on its own *Activation* event.

```
void RTOS::RunningToSuspended ()
 1: ACTIVE→State = SUSPENDED;
 2: ACTIVE = GetHighestReady();
 3: if (ACTIVE != NULL) {
 4:     ACTIVE→Activation.notify();
 5: } // end if
```

Listing 2.2 Transition from running to suspended

Listing 2.2 shows the pseudo-code for the transition from running to suspended state. The calling thread moves to SUSPENDED, and the highest READY priority is assigned to ACTIVE. If a READY task exists, its' activation event is notified. This function only updates the task's state. In order to block the thread, the SystemC *wait* statements will be used after the return from this function.

```
Void RTOS:: SuspendedToReady ()
1:  RTOS_task *t = GetTask (sc_process_handle());
2:  t→State = READY;
3:  if (ACTIVE == NULL)
4:      ACTIVE = GetHighestReady(); //run caller
5:  else
6:      wait (t→Activation);
```

Listing 2.3 Transition from suspended to ready

Listing 2.3 shows the pseudo-code of the implementation of the *SuspendedToReady* function. When the thread receives the resource that blocks its execution, the state of the thread becomes READY. If the calling thread is the only one in the READY state, the ACTIVE variable would point to NULL. In this case, the state of the current task is set to RUNNING by the *GetHighestReady* function. Otherwise, the thread waits on its activation event.

2.1.2 CPU Time Consumption

The system level model must contain delay annotations in the tasks to model their CPU time consumption. Modeling the time consumption of a given code segment on a hardware platform is an inherently difficult problem. The problem has been actively researched and there are some well known methods for predicting software execution time based on a model of the hardware. Typically, a prototype board with the processor core is available. Therefore, in order to obtain accurate delays, we measure the execution time of the computation blocks between the kernel calls on the processor. The delays are then back annotated to the model.

On the other hand, the time delays supported by the SystemC wait statements are not sufficient to model the CPU time consumption on a processor since they only allow concurrent delay consumption, whereas the tasks follow an interleaved execution on a processor. Therefore, we use the *consume* function to model the time required for

computational blocks to execute on a RTOS.

```
void RTOS::Consume (sc_time t)
1: sc_time TimeRemaining = t;
2: sc_time Start, Delta;
3: ScheduleEvent.notify();
4: while (true){
5:     Start = sc_time_stamp();
6:     wait (TimeRemaining, ScheduleEvent);
7:     ACTIVE = GetTask (sc_process_handle());
8:     Delta = sc_time_stamp() - Start;
9:     TotalBusyTime += Delta;
10:    if (Delta == TimeRemaining){
11:        break;
12:    }//end if
13:    TimeRemaining -= Delta;
14:    Running2Ready();
15:} // end while
16: Running2Ready();
```

Listing 2.4 Pseudo-code for CPU time consumption

Listing 2.4 shows the pseudo-code of the implementation of the consume function, for the FIFO scheduling policy. The *TimeRemaining* is the amount of time left to be consumed by the task if it is preempted. The *TotalBusyTime* variable is used to obtain the total CPU time consumption. This metric is essential to study the performance of the application on the modeled RTOS. The SystemC event, called *ScheduleEvent,* is used to allow preemption by rescheduling the active task. Due to interrupts and timer pulses, a task may call the consume function while another has already issued a consume call. To ensure that the user tasks do not share the CPU time consumption, ScheduleEvent is notified after each new consume call.

Initially, the time requested to be consumed by the calling task is stored in *TimeRemaining*. The variable *Start* stores the time before the SystemC wait on *TimeRemaining* and the *ScheduleEvent* is called. After returning from the wait statement,

the ACTIVE task pointer is updated to the current RUNNING task and the SystemC timestamp is subtracted from the *Start* value (lines 8-9). The subtraction result is stored in *Delta* to obtain the consumed time. Delta is then added to the *TotalBusyTime*. If Delta is equal to *TimeRemaining*, the required amount of time in variable t is consumed. Therefore, a break statement is called to break from the loop. However, if the *ScheduleEvent* is notified prior to the end of the required time consumption, *TimeRemaining* is subtracted by Delta and Running2ready is called. The task keeps iterating until the requested time is consumed.

If the *ScheduleEvent* is notified at the same logical time as the end of the required time consumption, the Running2Ready function is called before exiting the consume function. As a result, RTOS_SC's scheduler preempts the current task and selects the new task with the highest priority to execute.

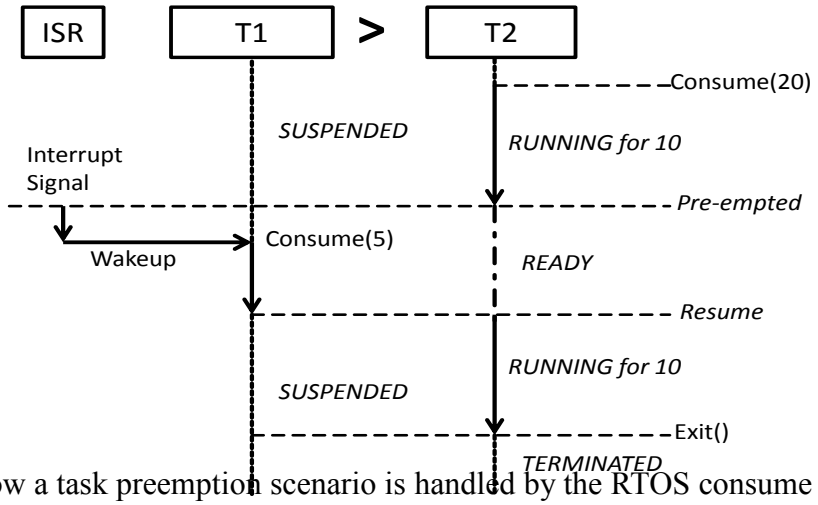


Figure 2.2 Preemption modeling in *Consume*

Figure 2.2 illustrates how a task preemption scenario is handled by the RTOS consume function. The example contains a thread *T1* having a higher priority than thread *T2*. We assume that T1 is initially blocked until a timer pulse is received. Therefore, T2 executes and calls the consume function to model a CPU time consumption of 20 time units.

However, after 10 time units, an ISR is triggered and hence *ScheduleEvent* is notified. Consequently, the wait statement in the consume method is interrupted and T2 is preempted by calling the RunningToReady function (Listing 4, lines 7-17). At this time, ISR with the highest priority models its time consumption and sends a pulse to T1. Therefore, T1 moves to a READY state. Since T1 has a higher priority than T2, T1 is selected to execute. After T1 consumes its 5 time units, it moves to a SUSPENDED state and waits for a pulse from ISR again. At this time, T2 is the only READY task. Therefore it resumes the remaining 10 time units and terminates the job required.

2.3 Modeling Communication in A Preemptive RTOS Model

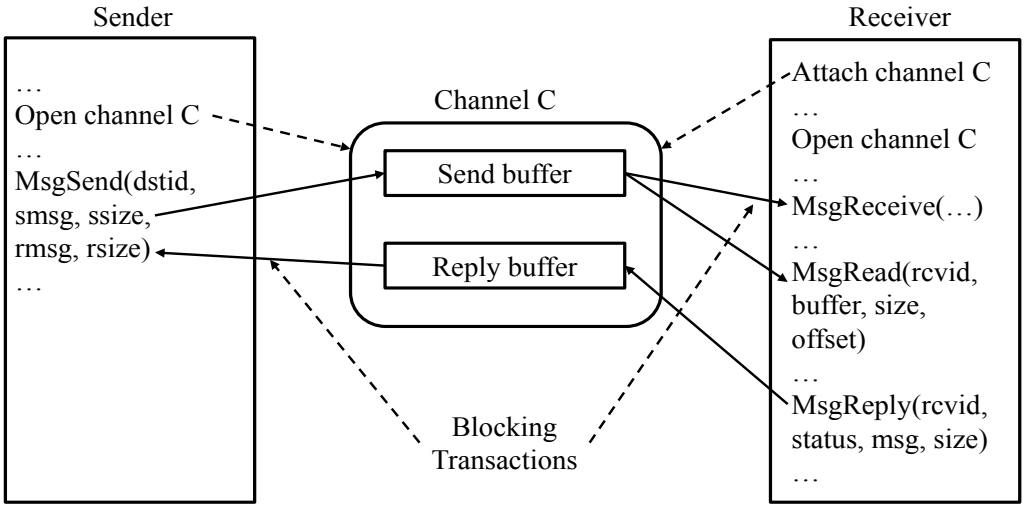


Figure 2.3 Channel-based communication in RTOS

Figure 2.3 illustrates the double-handshake semantics of channel communication in an RTOS. In order to buffer the transmitted data, the RTOS model requires an implementation of channels. As seen in this figure, the receiver initially creates the channel and opens it to receive data. The sender is also required to open the channel before the beginning of the transactions. When *MsgSend* function is called the message is sent

through the common channel between the sender and the receiver tasks. The execution of the calling thread is then suspended until a reply is sent from the receiver through the same channel. On the other hand, the *MsgReceive* function is called to receive a message and may suspend the calling task if no message was sent. The *MsgReply* method is used to send a reply from the receiver to the sender task.

In a preemptive RTOS model, the message passing communication of an RTOS can be modeled using a separate class defined on top of SystemC. To model the double handshake synchronization, a boolean flag (*SendFlag*) and events, (*SendEvent* and *ReplyEvent*) are defined in this class.

```
void RTOS:: MsgSend (int chid, …)
1: Copy data into send buffer
2: CH[chid].SendFlag = true;
3: CH[chid].SendEvent.notify();
4: RunningToSuspended();
5: wait CH[chid]→ReplyEvent;
6: SuspendedToReady();
7: Copy data from reply buffer;
```

Listing 2.5 Pseudo-code for sending message on channel

Listing 2.5 illustrates the MsgSend implementation in a preemptive RTOS model. The array CH refers to the pool of channels, indexed by variable *chid*. MsgSend copies the data pointed to by the sender into the send buffer. It then synchronizes with the receiver by setting SendFlag to true and notifying SendEvent to indicate that the receiver can now read from the send buffer. The sender waits for the reply by moving itself to the suspended state and waiting on ReplyEvent. Once the receiver has written to the reply buffer and notified ReplyEvent, the sender returns to the ready state and eventually copies over data from the reply buffer.

```
void RTOS:: MsgReceive (int chid, …)
1: if (!CH[chid].SendFlag) {
2:       RunningToSuspended();
3:       wait (CH[chid].SendEvent);
4:       SuspendedToReady();
5: }
6: CH[chid].SendFlag = false;
```

Listing 2.7 Pseudo-code for receiving message on channel

Listing 2.7 illustrates the MsgReceive implementation in a preemptive RTOS model. If SendFlag is true, the receiver knows that the send buffer has already been written. Therefore, it simply resets SendFlag and proceeds to read the data. Otherwise, the receiver waits on SendEvent until the send buffer is written. The receiver puts itself in the suspended state before the wait and returns to the ready state after the wait.

```
1 : sigevent pulse;
2 : timer_t timer;
3 : itimerspec tspec;
4: SIGEV_PULSE_INIT (&pulse, channel,…);
5 : timer_create(CLOCK_REALTIME, &pulse, &timer);
6 : tspec.it_value.tv_sec = 0;
7 : tspec.it_value.tv_nsec = 10 * 1e6; //10 ms
8 : tspec.it_interval.tv_sec = 0;
9 : tspec.it_interval.tv_nsec = 20 * 1e6; //20 ms
10:timer_settime(timerid, 0, &tspec, NULL);
```

Listing 2.6 RTOS Timers and Pulses

2.4 Timer and Pulse Modeling

In an RTOS, a timer can be set up to periodically send pulses to a user task at given time intervals. Timers are a common feature of real-time embedded software, so their modeling is highly pertinent. Listing 2.6 illustrates the key aspects of timer creation and setup in an RTOS. Timers use system events (*sigevent*) as pulses that are sent over a channel (lines 1-5). A timer specification (*itimerspec*) consists of an initial wait time (*it_value*) for the first pulse and an interval wait time (*it_interval*) for subsequent periodic

pulses (lines 6-10). The timer initialized in Listing 2.6 sends pulses over channel at times 10ms, 30ms, 50ms, 70ms and so on, after *timer_settime* is called.

```
1 : void timer(timer_id, it_value, it_interval){
2 :     wait(it_value);//initial wait values
3 :     while(true){
4 :         Timers[timer_id]→pulse.notify();
5 :         ScheduleEvent.notify(0);
6 :         wait(it_interval);//periodic wait values
7 :     }// end while
8 : }
9 : void RTOS::Wait4TimerPulse (int timer_id){
10:     Running2Suspended();
11:     wait(Timers[timer_id]→pulse);
12:     Suspended2Ready();
13: }
```

Listing 2.8 Timer emulation in SystemC

In order to emulate a timer in the SystemC model, we define a *RTOS_timer* class and a corresponding SystemC thread (distinct from application tasks), whose functionality is shown in the timer method in Listing 2.8. Similar to tasks, mutexes and channels, we define a pool of RTOS_timer objects. A timer pulse is modeled as an event in the RTOS_timer class. Corresponding to a timer creation in the RTOS, the SystemC model allocates a timer object from the timer pool. The timer_settime method corresponds to dynamic creation of the timer thread using *sc_spawn*.

The timer thread operation is fairly straightforward as seen in Listing 2.8. The timer waits for the initial wait time as defined in the timer specification (*it_value*), followed by an infinite while loop (lines 2-3). Inside the loop, the pulse event for the specific timer is notified to wake up the task sensitive to the timer pulse. This is followed by a delta cycle delayed notification of the scheduling event. The timer thread then waits for the interval period (*it_interval*) until the next periodic pulse. The task sensitive to the timer calls the Wait4TimerPulse method defined in the preemptive RTOS model, as shown in Listing 2.8 (lines 9-13). The waiting task is suspended while waiting for the timer pulse event. The ScheduleEvent notification (line 5) in the timer thread is delta cycle delayed to allow the waiting task to update its state to READY before the active task is moved to the ready state (Listing 2.8, line 10) and forces a rescheduling of tasks.

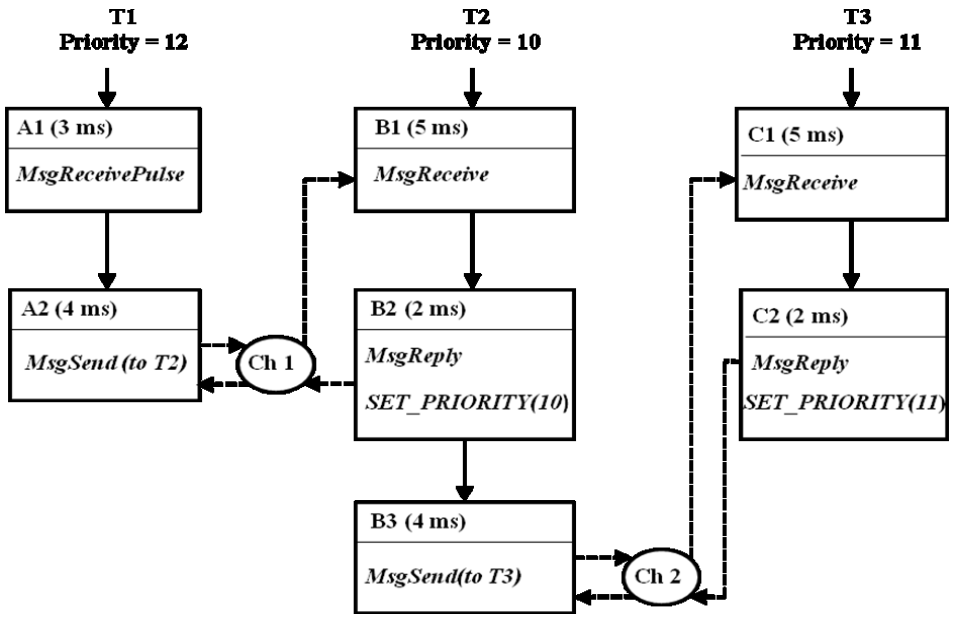


Figure 2.4 Message passing scenario

2.5 Impact of Preemption and priority inheritance

In this section, we study the impact of preemption and priority inheritance on the

accuracy of an RTOS model. For this purpose, models A and B are created. Model A implements a priority-driven scheduler in which the CPU time consumption and message passing are modeled without preemption and priority inheritance. On the other hand, model B implements a preemptive priority-driven scheduler without priority inheritance, as described in Chapter 3.

In order to validate the accuracy of the models, we trace the events generated from the execution of a multithreaded application on top of an RTOS kernel. The same application is then tested on models A and B to compare the output of each model with the one produced on the RTOS. We consider the QNX RTOS as our target system [16] and the example illustrated in Figure 2.4 as the common application.

Figure 2.4 illustrates a message passing scenario of three tasks T1, T2 and T3, with each executing at different priority. The blocks, labeled A1, B1, etc, indicate computations that are executed between the message passing functions. The timestamps specified in milliseconds, represent the intervals of time consumed by the computational blocks on QNX. These time intervals are measured on QNX and back annotated to the models. In addition to the message passing functions described in section III, the following functions are implemented:

- *MsgReceivePulse*: This function suspends the calling task if no pulse is sent. In this example, we set a timer pulse to be sent to T1 every 2ms.
- *SetPriority*: The receiver does not need to maintain the inherited priority when the message transmission is completed. Therefore, after each reply, threads' priorities are set back to the original ones by calling the *SET_PRIORITY* function.

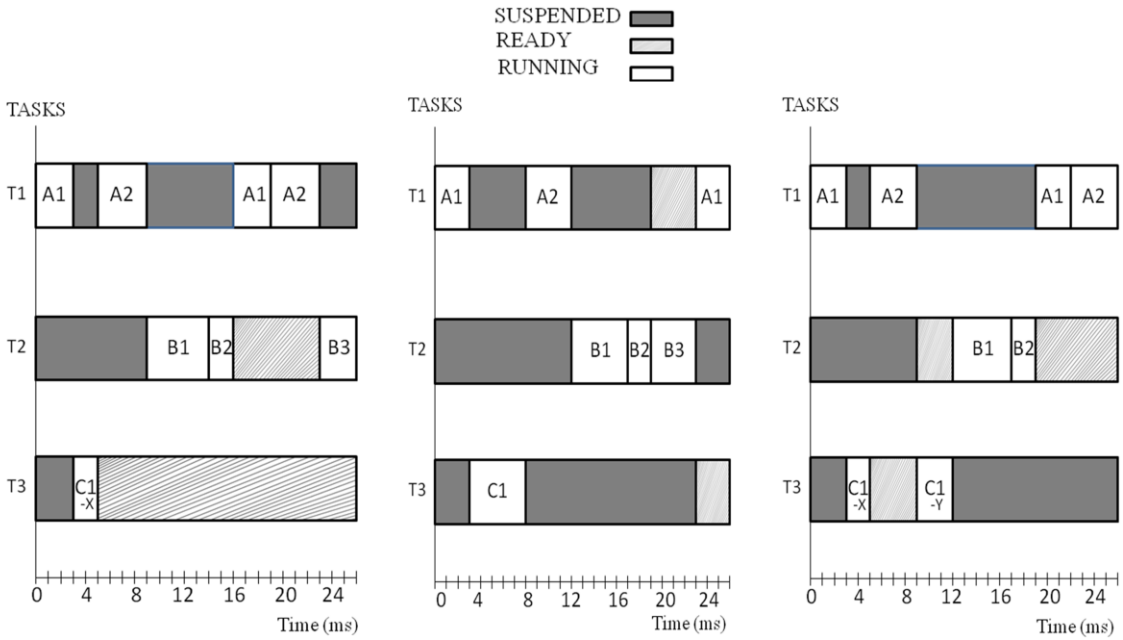


Figure 2.5 Tasks execution on (i) QNX (ii) Model A (iii) Model B

Figures 2.5(i), 2.5(ii) and 2.5(iii) illustrate a portion of the tasks scheduling that occurs when the scenario illustrated in Figure 2.4 is executed on QNX, model A and model B respectively. The letters in these figures refer to the block names shown in Figure 2.4. To indicate the task preemption during the execution of a block, this block is divided into two: block-x and block-y. Furthermore, to study the response time of task T1, the timeline is shown along with the transition states, illustrated in Figure 2.1. Initially, Figures 2.5(i) and 2.5(ii) are compared. The following events are common to both models:

1) At 0ms, the schedulers in QNX and model A select T1 to start execution since it is the READY thread with the highest priority.

2) At 3ms, T1 suspends to wait for the 2ms timer pulse. At this time, T3 becomes the highest priority READY task and therefore, is selected to execute.

3) At 5 ms, T1 receives the timer pulse.

Consequently, the QNX kernel preempts T3 and schedules T1, with the highest priority, to resume execution. On the other hand, the scheduler in model A does not instantly interrupt T3's execution since preemption is not implemented in this model. T3's execution is then interrupted only to wait for a message to be sent through Ch2. As a result, the time for block C1 is consumed before T1 executes. The time consumption causes a delay of 2ms in the response time of task T1 compared to T1's response time on QNX. The absence of preemption in model A causes T1 to be delayed again from 19ms to 23ms. Therefore, for an execution time of 25ms, task T1 with the highest priority is delayed for 6ms in model A with respect to QNX.

To avoid the thread being delayed by a lower priority, preemption is added to model B that implements the same fixed priority scheduler of A. The accuracy of B is measured by taking the same scenario illustrated in Figure 2.4. By looking at Figures 2.5(i) and 2.5(iii), it can be noticed that from 0ms to 9ms, the sequence of events generated by model B matches the one obtained on QNX. However, a mismatch occurs at 9ms when T1 sends a message to T2. On QNX, T2 inherits T1's priority which is higher than T3's priority. Therefore, T2 is selected to execute immediately after the message is sent from T1. However, because of the absence of priority inheritance in model B, T3's priority is always greater than T2's priority. Therefore, scheduling in model B happens as follows:

1) At 9ms, T3 is selected to resume execution before T2 receives the message.
2) At 12ms, T3 completes the 5ms time consumption for C1 and is moved to a SUSPENDED state to wait for a message from Ch 2.
3) At 12ms, T2 is the only READY thread. Therefore, it runs and consumes the

annotated time for block B1 before receiving T1's message.

4) At 17ms, the time for block B2 is consumed and a reply is sent to T1.

5) At 19ms, T2 is preempted and T1 is selected to run.

Compared to QNX, Model B shows an additional delay of 4ms for the highest priority thread. Therefore, even with models that implement preemptive schedulers, tasks behaviors are different from the ones observed on the targeted RTOS. In this scenario, the priority of T1 was inverted during message passing since T1 was dependent on T2, which operates at a lower priority. To avoid such delay in the tasks response time, we create a model, named RTOS_SC, in which preemption and priority inheritance are implemented.

CHAPTER 3

RTOS MODELING

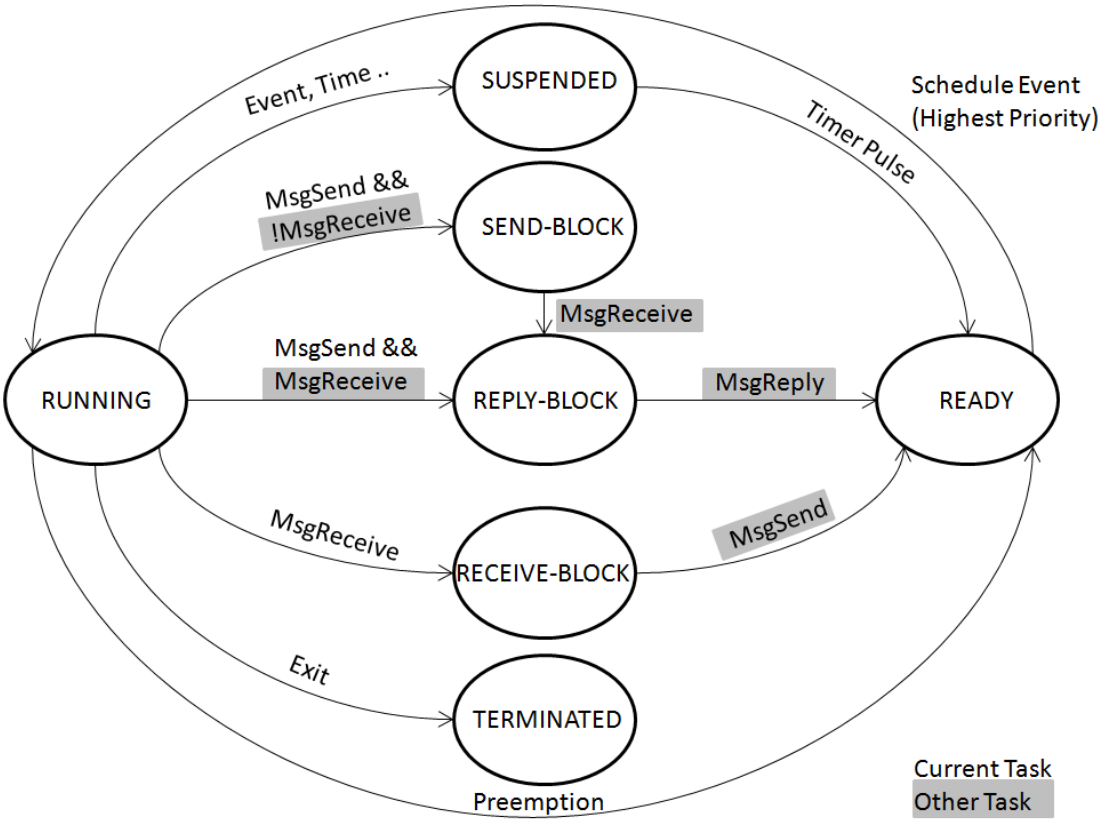


Figure 3.1 Tasks state in RTOS Model

3.1 Scheduler Modeling

To obtain a preemptive RTOS model in RTOS_SC, we implement the following states as shown in Figure 3.1: *RUNNING*, *READY*, *SUSPENDED* and *TERMINATED*. In addition, RTOS_SC has three states that represent blocking transactions during task communication: *SEND-BLOCK*, *RECEIVE-BLOCK* and *REPLY-BLOCK*. Differentiating

these states is essential to keep the priority-driven aspect when multiple tasks access the same channel. If the sender task attempts to write to the send buffer of the receiver's channel by calling the *MsgSend* function, two cases arise.

1) MsgSend is issued before the MsgReceive function is called by the receiver task.
2) The receiver issues a call to the MsgReceive function before the MsgSend function is called by the sender.

If the first scenario occurs, the sender is moved from RUNNING state to SEND-BLOCK state. After the message is received, the sender automatically moves from SEND-BLOCK state to a REPLY-BLOCK state and is suspended until a reply is sent from the receiver through the MsgReply method. The replied data are sent through the reply buffer of the same channel.

In the second case, the receiver is moved from RUNNING state to RECEIVE-BLOCK state until a message is sent from the sender end. Further, if the sender task calls MsgSend when the receiver is in a RECEIVE-BLOCK state, the sender does not need to wait in the SEND-BLOCK state. Therefore, it automatically moves from RUNNING state to REPLY-BLOCK state. Finally, after the reply, the sender moves to a READY state by calling the SuspendedToReady function.

RTOS_SC defines private functions in the scheduler class which are similar to the ones explained in Chapter 2. However slight modifications, which will be explained in this section, are applied to some of these functions to obtain more accurate preemption modeling.

```
void RTOS_SC::RunningToReady ()
1:  RTOS_task *t = ACTIVE;
2:  ACTIVE = GetHighestReady();
3:  t→State = READY;
4:  RTOS_task *t' = GetHighestConsume();
5:  if (ACTIVE != t) {
6:     ACTIVE→Activation.notify();
7:     wait (t→Activation);
8:     t' = GetHighestConsume();
9:  } // end if
10: while(ACTIVE != t') {
11:    t = ACTIVE;
12:    ACTIVE = NULL;
13:    t→State = READY;
14:    wait (t→Activation);
15:    t' = GetHighestConsume();
16 : } // end while
```

Listing 3.1 Transition from running to ready in the RTOS model

In Chapter 2, the implementation of the RunningToReady function shows that the calling threads exits this function and resumes the time consumption if the thread has the highest priority among tasks in the READY state. However, if an interrupt is generated while all tasks are suspended, the task which is blocked on this interrupt wakes up and calls the consume function. Similarly, other tasks may call the consume function due to the generation of interrupts at the same logical time. Therefore, it is possible that multiple tasks accumulate in the *consume* function during the program execution. In this case, the highest priority task should be the first to consume the CPU time.

Listing 3.1 shows the pseudo-code for transitioning from running to ready state in RTOS_SC. Among possible tasks in the *consume* function, the task with the highest priority is returned by the *GetHighestConsume* function and stored in task pointer t'(lines 3,8). If t' is different than the ACTIVE task, ACTIVE is stored in t. The ACTIVE task then points to NULL until a different thread is set to ACTIVE by the *consume* function (lines 11-12). Since t is not the highest in consume, it waits on its *Activation* event. After each wait statement t' points to the highest priority task in the *consume* function in order to compare the ACTIVE task to the most recent consuming tasks (line 11). The comparison of ACTIVE to t' repeats as long as ACTIVE is different than t'.

```
void RTOS:: SuspendedToReady ()
1:  RTOS_task *t = GetTask (sc_process_handle());
2:  t→State = READY;
3:  if (ACTIVE == NULL)
4:      ACTIVE = GetHighestReady(); //run caller
5:  else
6:      wait (t→Activation);
7:  RTOS_task *t' = GetHighestConsume();
8:  while(t' != NULL){
9:      if(t'→getPriority() > ACTIVE→getPriority()){
10:         t→State = READY;
11:         wait (t→Activation);
12:     } //end if
13:     else break;
14: } //end while
```

Listing 3.2 Transition from suspended to ready in RTOS model

Listing 3.2 shows the pseudo-code of the implementation of the *SuspendedToReady* function in RTOS_SC. After the thread's activation event is notified, the priority of the

ACTIVE thread has to be compared to the priority of the consuming threads (line 9). As long as the priority of ACTIVE is not the highest, the thread waits again on its activation event (lines 8-11). Otherwise, the active thread breaks out of the loop and resumes execution (line 13).

3.2 Modeling Different Scheduling Algorithms

To obtain a generic RTOS model which enables design space exploration, different scheduling policies are modeled: First-In-First-out, Round-Robin and rate-monotonic.

3.2.1. First-In-First-Out Scheduling Policy (FIFO SP)

The FIFO algorithm is commonly used in real-time applications, since it ensures a minimum RTOS overhead by minimizing the context switching, compared to Round-Robin policy. In a pre-emptive, priority-based scheduler, this algorithm ensures that the highest priority thread is always the one selected for execution, at any time during simulation. To select the highest priority task, the READY tasks are searched from the oldest one in the READY list, to the newest one. In RTOS_SC, we model a pre-emptive, priority-based FIFO SP, by accurately modeling pre-emption and priority inheritance, as will be explained in this chapter.

3.2.2 Round-Robin Scheduling Policy (RR SP)

Round-Robin scheduling is modeled by giving a time slice, which is specified in the specification model, for each thread running this policy. When the time slice elapses, rescheduling occurs to allow a READY thread with greater or equal priority to run. However, in our RTOS model, as in modern RTOSs, preemption does not only occur at the end of the time slice, but at any time a higher priority task becomes ready for execution. Compared to the pre-emptive scheduling with fixed priority, Round Robin

policy guarantees the execution of all threads with equal priorities. The preemptive scheduling policy is not optimal for applications containing tasks with equal priorities. When a running task takes large computation time, and a task at equal priority becomes READY, the READY task may not be selected for execution when the scheduler runs the FIFO scheduling policy. This process, called *starvation*, happens since the FIFO algorithm allows preemption only for higher priority tasks.

In RR, if a READY thread exists at an equal priority to the one running, the running thread is pushed to the end of the READY list at the end of the time-slice. Hence, the ready threads with equal priorities to the last one running, get the chance to run. However, consecutive calls to the *consume* function may occur in a thread. If thread suspension occurs between two *consume* calls, its time-slice is reset to zero ms. On the other hand, if the thread isn't blocked between two consume calls, the time slice is subtracted from the remaining time slice, which is calculated in the previous consume call.

When RR SP is selected, preemption is modeled in the *consume* function, similar to the methodology explained in Listing 2.4. However, the required time consumption, (CT), is compared to the time slice (TS) assigned to the thread. If the time to be consumed is less than TS, the thread waits for the time remaining from CT, and subtracts the time slice after the consumption time elapses. However, if CT is greater than TS, the thread waits on the TS to elapse in order to allow READY threads with equal priority to execute. Otherwise, if there are no READY threads with equal or higher priority, TS is subtracted from CT and stored in the remaining time. The thread then waits for either TS or CT, depending on whether the remaining time is greater than TS or not. This procedure is

repeated until the thread finishes time consumption. Figure 3.2 illustrates an example of the task execution in RR.

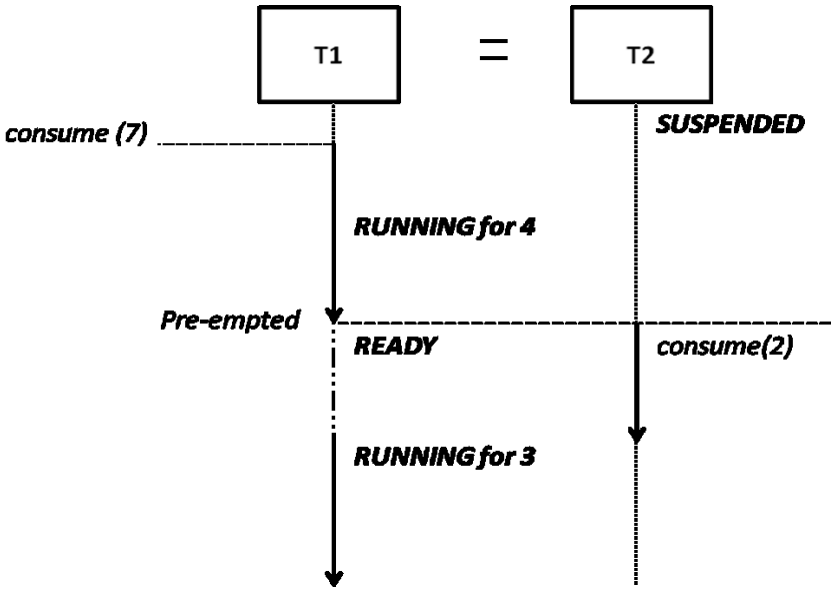


Figure 3.2 Preemption modeling in Round-Robin scheduling policy

Figure 3.2 shows two threads, T1 and T2 with equal priorities. Initially, T1 and T2 are READY to execute. T1, the first thread pushed to the READY list, starts execution. T1 is then required to consume 7 time units (CT = 7). If the TS is 4 time units, rescheduling occurs at this time since TS is less than CT. Therefore, task T2, runs and consume its 2 time units. When T2 suspends, T1's CT becomes CT – TS, that is 3 time units. Since the new CT is less than TS, T1 consumes CT time units. Therefore, T2 is not delayed for the large computation time required by T1, but only for the time-slice.

3.2.3 Rate-Monotonic Scheduling Policy (RM SP)

This policy is used for applications containing periodic tasks. In such applications, each task has a deadline to complete its job. To ensure that all deadlines are met, the priority of the task is inversely proportional to the period. The shorter the period is, the

higher the task priority would be. The priorities are statically assigned when this policy is selected. Therefore, the following constraints must be satisfied in an application in order to assign the rate-monotonic scheduling policy:

- All tasks are periodic.
- Deadlines are equal to the period.
- The overhead of context switching is negligible.

Furthermore, resource sharing, such as mutexes and queues, are not allowed when RMSP is selected in order to prevent priority inversion. However, in our RTOS model, as in modern RTOSs, RM SP is modified to avoid priority inversion by modeling the priority inheritance protocol.

```
void RTOS_SC::MsgSend (int chid …)
1: int sid,  sp;
2: if (!CH[chid].RFlag){ //Send-Block
3:     RunningToSendBlock();
4:     while(true){
5:         wait(CH[chid].RecEvent);
6:         sid = GetThreadId();
7:         if(sid == CH[chid].HighestSender)
8:             break;
9:     } //end while
10: } //end if
11: SFlag = true;
12: Copy data into send buffer
13: sp = GetSenderPriority(); // Priority Inheritance
14: if(sp > GetRp())
15:     SetReceiverPriority(sp);
16: CH[chid].SEvent.notify();
17: SendBlockToReplyBlock(); // Reply-Block
18: wait(CH[chid].RepEvent);
19: ReplyBlock2Ready();
20: Copy data from reply buffer;
```

Listing 3.3 Pseudo-code for sending message on channel in RTOS model

3.3 Modeling Inter-Task Communication with Priority Inheritance

3.3.1 Channels

Listing 3.3 shows the pseudo-code for the MsgSend function. The variable *Chid* is used as an index to select the corresponding channel among the pool of channels that are available in the *CH* array. The variable *RFlag* is set to true by the receiver thread to

indicate that it is in RECEIVE-BLOCK state.The methods *GetSenderPriority* and *GetReceiverPriority* are used to obtain the priorities of the receiver and sender. To allow the receiver to inherit the sender's priority, the function *SetReceiverPriority* is used. The method *GetThreadId* is used to get the ID that is assigned to the calling thread. The SystemC event *RecEvent* is notified when the receiver calls MsgReceive to allow a SEND-BLOCK thread to send the message through the associated channel. RepEvent is notified when the reply is sent from the receiver thread.

If *RFlag*'s value is true, the sender sends the message immediately (line 14). Otherwise, it should wait in SEND-BLOCK state until *REvent* is notified. However, it is possible that multiple threads attempt to send through the same channel before MsgReceive is called on the receiver end. Therefore, before waiting for *REvent*'s notification, the sender is pushed to a queue that represents the threads in SEND-BLOCK state by calling the RunningToSendBlock function (lines 3-5).

Further, after *REvent* is notified, all the SEND-BLOCK threads that require access to the receiver's channel become ready to be selected by the SystemC kernel. In order to ensure that the sender with the highest priority is the first one to send, the ID of this sender is selected in MsgReceive. As long as the sender ID referred by *sid* is different from *HighestSender,* the sender thread blocks in the wait statement (lines 4-5). If the sender's ID matches *HighestSender,* the calling thread returns from the wait statement and breaks out of the loop (lines 7-8).

If the sender's priority is greater than that of the receiver, the receiver inherits the sender's priority when the message is sent (lines 14, 15). The change in priority happens before the receiver receives the message to avoid the priority inversion case shown by model B in Chapter 2. After priority inheritance, the sender thread calls the SendBlockToReplyBlock method to move to REPLY-BLOCK state until a reply is sent from the receiver thread (line 18). Finally, when the reply is sent, the data is copied from the reply buffer (line 20).

```
void RTOS_SC:: MsgReceive (int channel_id, …)
 1:  int sp, rp;
 2:  CH[chid].HighestSender =  GetSid();
 3:  CH[chid].RecEvent.notify();
 4:   if(CH[chid].HighestSender == -1)
 5:         CH[chid].RFlag = true;
 6:   RunningToReceiveBlock(); // Receive-Block
 7:   wait (CH[chid].SEvent);
 8:   ReceiveBlockToReady();
 9:   CH[chid].RFlag = false;
 10: CH[chid].HighestSender = -1;
 11: sp = GetHighestSp(); // Priority Inheritance
 12: SetReceiverPriority(sp);
 13: CH[chid].HighestSender = -1;
```

Listing 3.4 Pseudo-code for receiving message on channel in RTOS model

Listing 3.4 shows how the MsgReceive function is implemented in model RTOS_SC. The method *GetSid* selects the SEND-BLOCK thread with the highest priority by returning its assigned ID. The returned value is stored in channel member variable *HighestSender*. In case no attempts were made to send data through the receiver's channel, the function *GetSid* returns -1 to indicate that the receiver is in RECEIVE-

BLOCK state (line 2). Further, the calling thread notifies the RecEvent to allow the thread with the selected ID to send the message. If *HighestSender* is equal to -1, the *RFlag* is set to true (lines 4-5). The calling thread is then suspended until a message is sent (line 6). After receiving the message, the RFlag is reset to false and the SenderId is assigned to -1 to ensure that threads in SEND-BLOCK state are not selected until the next time the MsgReceive function is called (lines 9-10).

The *GetHighestSp* function is used to return the highest priority thread among the threads in SEND-BLOCK and REPLY-BLOCK states. The returned priority is stored in the *sp* variable and the priority of the receiver is set to sp (line 12). The implementation of priority inheritance at the receiver side is explained by the scenario shown in Figures Figure 3.3 and Figure 3.4.

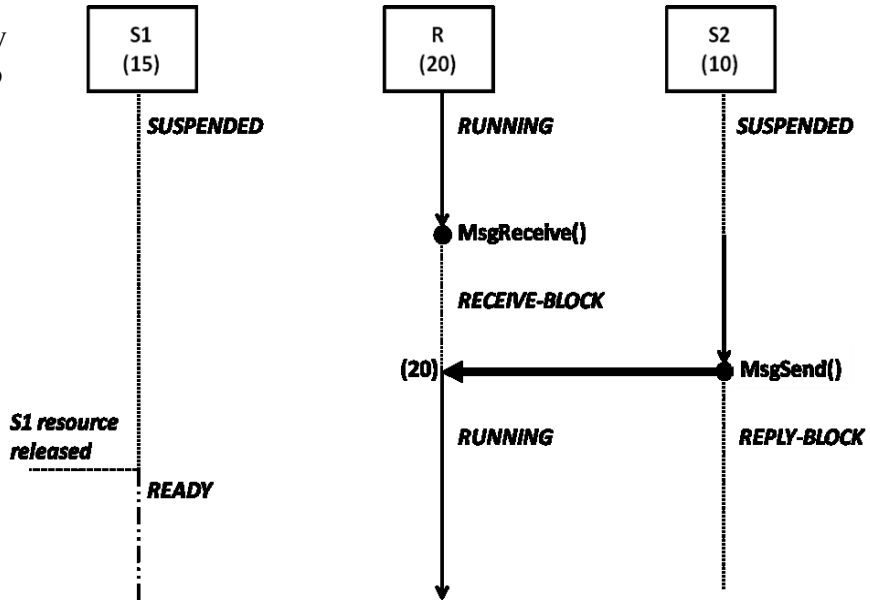


Figure 3.3 Priority inversion scenario without Priority Inheritance at the receiver end

This application contains three threads S1, S2 and R, with S1 having a higher priority than S2 and R having the highest priority. The task priorities are indicated inside the parentheses. The receiver R starts execution and moves to a RECEIVE-BLOCK state

when MsgReceive is called. Since the sender S1 is suspended on a resource, sender S2 is the only READY thread at this time and hence, it is selected to execute. During S2's execution, it sends a message to the receiver task and waits for a reply. However, while R is processing S2's request, S1 receives its resource. In this case, if priority inheritance is not modeled at the receiver end, S1 will not be able to execute because the receiver is running at a higher priority. Consequently, S2 which has a lower priority than S1, prevents S1 from executing by sending data to another thread with an even higher priority. This form of priority inversion happens when the priority of the receiver is greater than the sender's priority. In order to solve this issue, the receiver inherits the sender's priority when the message is received (Listing 12, line 12). Further, the inheritance happens at the receiver end since only then the receiver starts computing data for the sender.

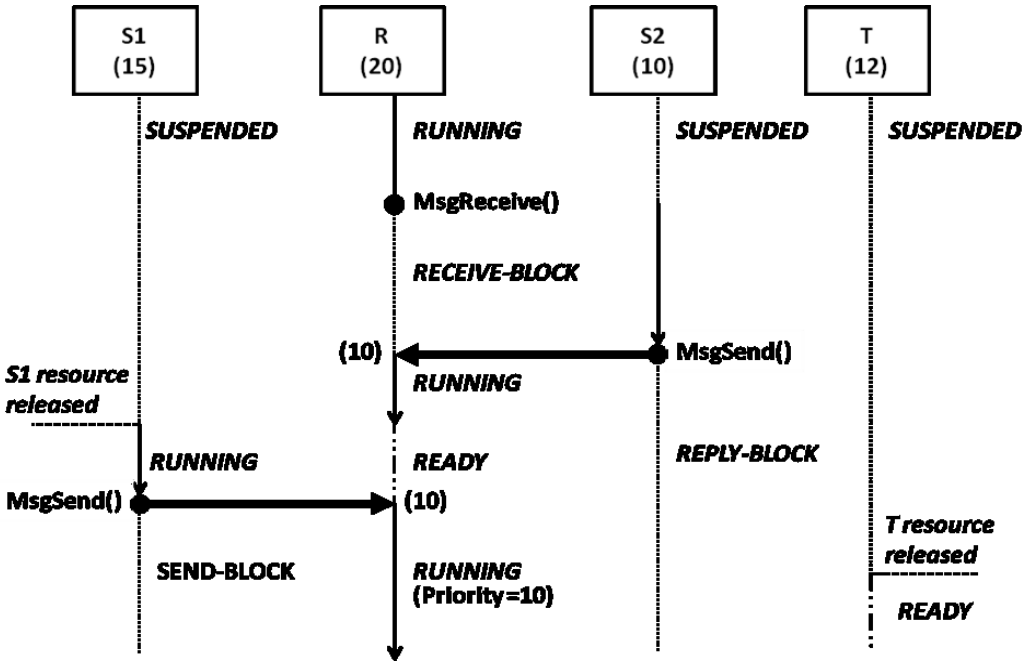


Figure 3.4 Priority inversion scenario with Priority Inheritance of REPLY-BLOCK threads at the receiver end

Figure 3.4 shows another case of priority inversion if the receiver inherits the senders which are in a REPLY-BLOCK state. After the message from S2 is received, R moves from a RECEIVE-BLOCK state to a RUNNING state and operates at S2's priority. When S1 is READY, R is preempted and S1 is selected to run. During its execution, S1 sends a message to R and moves to SEND-BLOCK state. At this time, S2 is in a REPLY-BLOCK state and hence, two threads exist in the MsgSend function. Since the receiver only inherits senders in REPLY-BLOCK state, R's priority would remain the same as S2's priority. However, another thread, say T, may exist with a priority of 12, which is greater than the current priority of R and less than S1's priority. If T obtains its resource before R replies to S2, R would be preempted by T, and hence, S1becomes again a victim of priority inversion. To avoid this problem, the receiver inherits the sender with the highest priority among tasks in the SEND-BLOCK and REPLY-BLOCK states (Listing 12, lines 11-12).

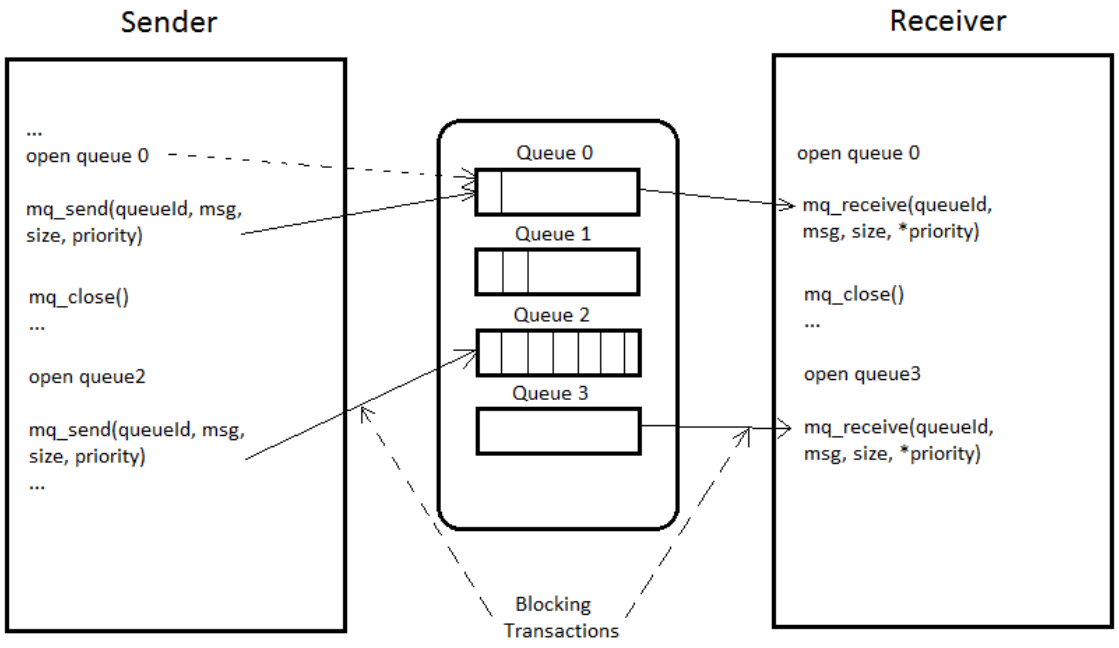


Figure 3.5 Queue-based communication in RTOS model

3.3.2 Queues

Figure 3.5 illustrates asynchronous communication using queues for message passing. The following implementation is used to minimize context switching, which adds overhead to the task response time, when multiple threads communicate using the same channel. Since multiple messages may be sent in a queue, the data transmitted are not overwritten when queues are used for message passing. Therefore, synchronization services such as mutexes are not required. Furthermore, communication protocols, such as double hand-shake, are not modeled in queues to decrease the number of context switches. Consequently, a non-blocking task communication is obtained, which increases performance and allows queues to be used in the following cases:

- The data transmission is guaranteed during message passing
- The reliability in inter-task communications does not have a severe impact on the application functionality.

The only place where threads may block, however, is when the queue is full or empty. To enable a priority-based message transmission, a protocol is built to ensure the receipt of message according to the priority it has.

Queue class, available in the RTOS_SC library, implements the mq_open, as seen in Figure 3.5. It opens an existing queue or create one if the queue is opened for the first time. It inputs the queue name, its attributes and policies and returns a reference to the specified queue. For a given number of queues in the pool, the *Queue* instances are created statically. The global AVAIL_QUEUES variable is an array of Queue instances.

Each queue has its own properties specified by the user. The properties includes the maximum messages a queue can have, the maximum size of each message and if message passing on the specified queue is *blocking* or *non-blocking*. In addition, a queue can have a read only, write only or read and write policy. The type of signals, such as priority pulses and events, is used to notify a thread of the queue status and is also declared by the user. The mentioned parameters are specified in *struct mqd_t* and *mq_attr*. Having the flexibility to specify the queue properties gives more control over the data, and therefore increases the performance for a variety of communication systems.

The mq_send function is used to send a message to a queue, and the *mq_receive* fuction, to receive the message according to the message priority. The methods *mq_unlink*

```
void Queue::mq_send (mqd_t mq_destination, char*msg,  int sbytes, int  msg_priority)
1: if(policy == WONLY || policy ==RDWT){
2:     if(getCurrQueueSize() > getMaxMsgs() && geFlags() == 1){
3:         Running2Suspended();
4:         While(msg_priority < getHighestMsg())
5:             wait(mq_event);
6:         Suspended2Ready();
7:     }
8:    if(getMsgSize() > sbytes && getMaxMsgs() > getCurrQueueSize()){
9:       push data and priority into specified queue
10:     setNotifiedId();
11:  }
12:  else
13:      return -1;
14: }
15: else
16:   return –1;
```

Listing 3.5 Pseudo-code for sending message on queue

and *mq_close()* are used to delete a queue. If a queue is opened by more than one thread, *mq_unlink* postpone the deletion until *mq_close* is called. *mq_close* always forces the queue deletion even if the specified queue is used by other threads. Attempts to use a deleted queue set error signals.

mq_send(), shown in Listing 3.5, is used to push the data sent to the specified queue. The inputs for this function are the queue from which the message is required to be sent, the actual message, the number of bytes the message contains and the message priority. Since multiple queues can be instantiated, the *QueueID* variable is used to indicate the ID of the current queue in use. The *getCurrQueueSize* function returns the number of messages in the selected queue. The method *getMaxMsgs* is used to return the queue size. *getFlags* function returns 1 if the blocking property is set, and 0 if the non-blocking property is set. *getHighestMsg* is a function that returns the highest priority message among messages that are waiting to be sent to the selected queue. The size of a message is indicated by the *getMsgSize* function.

The function first checks if the opened queue has a *write only* (WONLY) or *read and write* policy (RDWT), in order to allow writing the message to the queue. If any of these conditions are met, the calling thread enters an *if* statement to check if the queue is full and has the *blocking* property. If both conditions are true, *mqsend* blocks the calling thread on the *mq_event* variable until a message is received (line 4). However, if multiple threads call *mq_send* when the selected queue is full, the highest priority messages are sent as messages are received (lines 4-5).

On the other hand, if the queue is not empty and the message size does not exceed the maximum size allowed, the data are transmitted to the specified queue (line 9). Furthermore, the *setNotifiedId* function is called to ensure that threads that are blocked, at the receiver end, are unblocked in a FIFO order. Consequently the first thread ID in the queue is returned by *setNotifedId*. If the queue is full, and a *non-blocking* property is set, the function returns -1, to indicate that the message has not been sent to the queue.

Similarly, if the policy of the specified queue is set to *read only*, the function returns – 1 and no data is transferred to the queue.

```
Void Queue::mq_receive (mqd_t mq_destination, char *msg, int qbytes,int
*msg_priority)
1: if(policy == RDONLY || policy ==RDWT){
2:   if(geFlags() == 1 && queue.empty()){
3:       Running2Suspended();
4:       while(getCurrThreadId() != getNotifiedId())
5:           wait(qsend);
6:       Suspended2Ready();
7:    }
8:    else if(geFlags() == 0 && queue.empty())
9:       return -1;
10:  copy the data and priority to the receiver
11:  delete message from the queue
12:  resetNotifiedId();
13:  return 0;
14: }
15: else
16:  return -1;
```

Listing 3.6 Pseudo-code for receiving message on queue

Listing 3.6 shows the pseudo-code of the *mq_receive* method. *mq_receive* is used to retrieve data from the selected queue. The *getCurrThreadId* method returns the calling thread ID, and the *getNotifiedId* method returns the receiver thread ID, which is selected in *mq_send*. *mq_receive* first checks first if the queue policy is read only (RDONLY) or read and write (RDWT) in order to receive the messages in the queue (line 1). If any of these conditions is true, the second *if* statement is executed to verify if the queue is empty, and the blocking property is set for the specified queue (line 2). If both conditions are true, the calling thread waits for a message to be sent to the empty queue, as long as the current thread ID is different than the one returned by the *getNotifiedId* function (lines 4-5).

If, the selected queue is empty and has the non-blocking property, the function returns -1 to indicate that the no message has been received (line 8). Otherwise, if the queue is not empty, the oldest message with the highest priority will be received first and removed from the queue (lines 11-12). Further, the *resetNotifedId* function is called to reset the selected receiver ID. In case the policy is set to WONLY, the function returns -1 to indicate that reading from the selected queue is not allowed.

Other global functions are implemented in RTOS_SC, such as *mq_notify*. This function can be used to register a notification for the calling thread, when the queue transitions from empty to non-empty. However, when the thread requires a notification, no other threads can be notified of the transition until the first calling thread gets the notification.

3.4 Modeling Different Synchronization Services

In our model, we increase the design space exploration by modeling different synchronization services, which may be used during communication. We model barriers and condition variables in RTOS_SC since they are commonly used for inter-task communication.

3.4.1 Barriers

One of the synchronization services which may be used for inter-task communication is barriers. Any thread must stop at the point where it reaches a barrier and cannot resume execution until a number of threads, specified by the *pthread_barrier_init* function, is reached.

The function *pthread_barrier_destroy*, is used to delete the barrier initialized by *pthread_barrier_init*. It takes as input an object of struct pthread_barrier_.

The struct pthread_barrier_t is available on Linux kernels and is, therefore, renamed to

```
void barriers::bar_wait()
1: currentBlockedThreads++;
2: if(currentBlockedThreads == maximum){
3:     barEvent.notify();
4: }
5: else {
6:     running2suspended();
7:     wait(barEvent);
8:     suspended2ready();
9: }
10: return 0;
```

Listing 3.7 Pseudo-code for waiting on a barrier

pthread_barrier_ in RTOS_SC.

Listing 3.7 shows the pseudo-code of the implementation of the *bar_wait* function. *bar_wait* is used to make a number of threads wait on a specified barrier. The number of threads is stored in the *curentBlockedThread* and incremented as a thread reaches the barrier (line 1). If the number of threads, specified in *maximum*, is attained, the threads waiting on the selected barrier are notified and allowed to resume execution (line 3). The last thread reaching the barrier shouldn't then wait on the event but only notify the threads, waiting on the *barEvent* variable. Since, these threads are all notified at the same logical time, the function *suspended2Ready* is responsible of scheduling those threads to resume execution from higher to lower priority.

3.4.2 Condition Variables

To increase the control over user threads, condition variables can be used. A thread can wait on a variable for which the attributes are initialized by creating an object of the struct *pthread_cond_* and assigning an ID or a name for it. Condition variables are always used with a mutex to avoid undefined behavior: due to preemption, access to the shared region may alter the value of the condition variable and produce unexpected results.
The struct name *pthread_cond_t* is used in Linux kernels and therefore cannot be used with the same name in RTOS_SC. Hence, the struct initializing the condition variable is renamed to *pthread_cond_*.

```
int pthread_cond_wait(pthread_cond_ * cond, pthread_mutex_ * mutex)
 1: mutexUnlock();
 2: if(cond->ID <= 0 || mutex->ID <= 0)
 3:    return -1;
 4: storeThread();
 5: running2suspended();
 6: thread_wait();
 7: mutexLock();
 8: suspended2ready();
 9: return 0;
```

Listing 3.8 Pseudo-code for waiting on a condition variable

Listing 3.8 shows the pseudo-code of the *pthread_cond_wait* method. The *pthread_cond_wait* method is used to allow the user to suspend a thread on a condition variable until it receives a signal or a broadcast. Before being suspended, the associated mutex is unlocked, by calling the mutexUnlock function (line 1). The function *storeThread* is called to store the condition variable ID in an array, named *condArray*, for which the thread ID is the index (line 4). If the ID of either the condition variable or the mutex is not positive, *pthread_cond_wait* returns -1, indicating that one of them has not been initialized. The function *thread_wait* makes the calling thread wait on the sc_event

condEvent variable. After receiving a signal or a broadcast, the thread returns from the wait and locks the thread again with the function *mutexLock*.

```
int pthread_cond_signal(pthread_cond_* cond)
1: int max = -1, thread = -1;
2:  for(int i = 0; i < MAX_TASKS; i++){
3:        if(getCondId(i) == cond->cid && getCondPriority(i) > max){
4:           max = getCondPriority(i);
5:           thread = getCondId(i);
6:        }
7:  }
8: if(thread == -1 || max == -1)
9:    return -1;
10: notifyWaitingCond(cond->cid, thread);
11: return 0;
```

Listing 3.9 Pseudo-code for notifying one of the waiting threads

Listing 3.9shows the pseudo-code of the function *pthread_cond_signal*. This function is used to notify the thread with the highest priority between all threads waiting on the specified condition variable. The *max* variable is used to calculate the highest priority thread, waiting on the input condition variable, *cond*. The *thread* variable stores the thread ID of the thread waiting on *cond*. Both variables are initially assigned to -1. Each thread in the scheduler has a variable named *condID*, which stores the ID of the condition variable, if the thread is blocked on a condition variable. Otherwise, the value of *condID* would be -1. Since *pthread_cond_signal* is a global function, the *getCondId* function is used to return the *condID* of the thread, by specifying its index as input. Similarly, the method *getCondPriority* is used to return the priority of the indexed thread. Threads' priorities are compared to *max* as long as *getCondId* matches the ID, named *cid*, of the input variable *cond* (line 3). Each time a higher priority than *max* is found, the priority and thread ID are stored in *max* and *thread* respectively (lines 4-5). This process is repeated for the total number of tasks, that is *MAX_TASKS* (line 2). If any of *max* and

thread remains -1, the function returns -1 indicating that no threads are waiting on *cond*. Finally, the function *notifyWaitingCond* is called to notify the thread, which has the ID equal to *thread*, and condition variable ID equal to *cid*.

```
int pthread_cond_broadcast(pthread_cond_* cond)
1: int thread = -1;
2:  for(int i = 0; i < MAX_TASKS; i++){
3:        if(getCondId(i) == cond->ID)
4:         notifyWaitingCond(cond->ID, thread);
5:         thread = getThreadId(i);
6:  }
7:  if(thread == -1)
8:     return -1;
9: return 0;
```

Listing 3.10 Pseudo-code for notifying all waiting threads

Listing 3.10 shows the pseudo-coder for the implementation of the *pthread_cond_broadcast* function. It is used to notify all the waiting threads on the specified condition variable. The function *supended2ready* in *pthread_cond_wait* allows then, the one with the highest priority to execute first.

3.5 Modeling Semaphores And Mutexes for Input/Output Communication With Priority Inheritance

In our model, we provide means for communication between the tasks created by the scheduler and the ones which are linked to other peripherals, such as hardware models. To enable data transmission in input/output communication, we use shared variables. Accesses to such variables are protected by modeling mutexes and semaphores.

The user is allowed to control access of multiple threads, accessing a shared region, using semaphores. The semaphore is initialized by the user with *sem_init()* which takes as

input a pointer to the *struct sem_t* to initialize the semaphore, an integer specifying if the semaphore is shared only by threads or by processes and the initial value of the semaphore. The second parameter is neglected since only threads are created in RTOS_SC.

The semaphore exists until the function *sem_destroy()* is called by the user. It takes as input *sem_t* object. The function selects the convenient semaphore and deletes it.

```
void semaphore::sem_wait()
1: if(!INIT)
2:     return -1;
3: task * t = getCurrThread();
4: if( currentLockedThreads > maximum ) {
5:     pushToWaitQ();
6:     running2suspended();
7:     wait(t->semEvent);
8:     suspended2ready();
9: }
10: currentLockedThreads++;
11: deleteWaitQ();
12: return 0;
```

Listing 3.11 Pseudo-code for semaphore wait

The pseudo-code for the *sem_wait* function is shown in Listing 3.11. This function allows a specific number of threads, which is stored in the *maximum* variable, to enter into a critical region. If the semaphore has not been initialized by *sem_init,* the value of the INIT variable would be 0, and hence, the *sem_wait* function exits by returning -1 (lines 1-2). Otherwise, the current number of locked threads, stored in the *currentLockedThreads* variable, is compared to *maximum* (line 4). If *currentLockedThreads* is greater than *maximum*, the function *pushToWaitQ* is called to push the calling thread to a vector, in which the waiting threads are stored (line 5). The *t*

variable, in line 7, is an object of the *task* structure, which contains the *semEvent* variable. This structure is defined to allow semaphores to be used, not only by tasks that belong to the RTOS_SC scheduler, but also to be accessed by tasks that may belong to other hardware models.

The calling thread, if it is a task defined by the scheduler, moves to a SUSPENDED state and waits on the *semEvent* variable (lines 6-7). Only when a locked thread is unlocked by calling the *sem_post* function, that the waiting thread will be allowed to enter, in a priority-based order, the critical region. Upon return from the wait statement, *currentLockedThreads* is incremented and the function *deleteWaitQ* is called to remove the locked thread from the queue, if it has been pushed to it (lines 10-11).

```
void semaphore::sem_post()
1: if(!INIT)
2:     return - 1;
3: if(currentLockedThreads > 0)
4:     currentLockedThreads--;
5: task * t ;
6: if( QSize() != 0){
7:      t= getHighestInQ();
8:      t->semEvent.notify();
9: }
10: return 0;
```

Listing 3.12 Pseudo-code for semaphore post

The pseudo code for the *sem_post* function is shown in Listing 3.12. This function is used to unlock a thread before it exits the critical region. The number of locked threads represented by *currentLockedThreads* will be decremented to allow the waiting thread to enter the protected region (line 3). However, if there are threads waiting to lock the semaphore, the thread with the highest priority should be notified. Therefore, if the queue in which the waiting threads are stored, is not empty (line 6), the getHighestInQ function

selects the thread at the highest priority from this queue (line 7). The *semEvent* variable for the selected thread is then notified to allow it to lock the semaphore (line 8).

Mutexes are also modeled in RTOS_SC. They have the same implementation as semaphores, except that the maximum number of locked threads is 1, in a mutex. However, in both semaphores and mutexes, priority inversion may occur if pre-emption happens during the access to shared region. The scenario shown in Figure 11, describes this case.

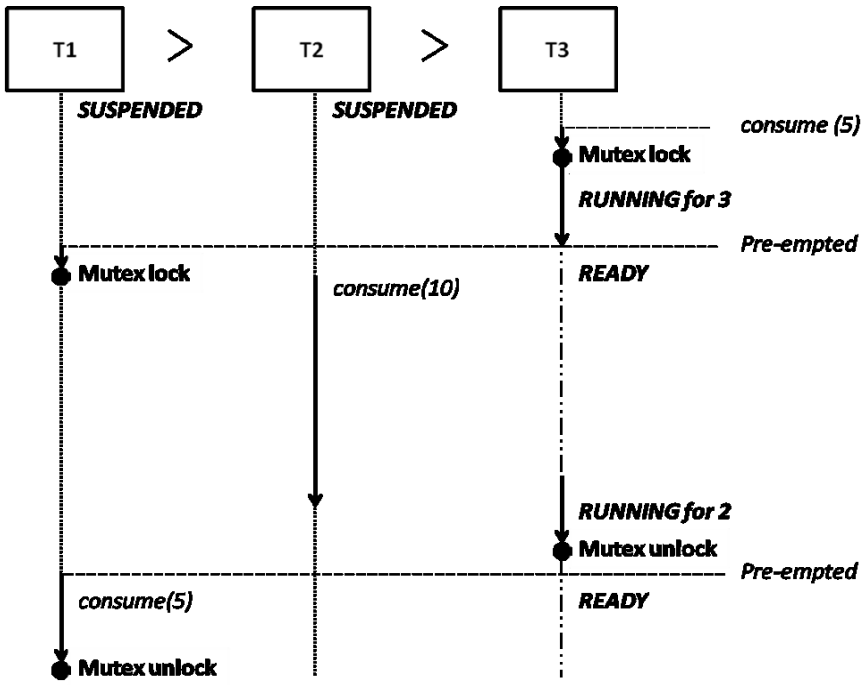


Figure 3.6 Priority Inversion scenario caused by mutex, in a basic RTOS model

Figure 3.6 shows a priority inversion scenario caused by the mutex. Initially, threads T1 and T2 are suspended, on a timer pulse. Therefore, thread T3, with the lowest priority, runs and access a shared region. Since it is the first thread accessing this region, the mutex locks T3 and allows it to resume execution. The computation time needed by T3 to

exit the shared region takes 5 time units. However, the highest priority thread T3, receives an interrupt from the timer at the third time unit. Therefore, a basic preemptive scheduler selects T1 to execute. During T1's execution, T2 with an intermediate priority, receives the timer pulses. In this case, the problem happens if T1 tries to access the same shared region locked by T3, as seen in Figure 11. T1 is then blocked by the mutex until T3 unlocks it. T2 is therefore, selected to execute and consume the 10 time units required before it suspends. At the 11^{th} time unit, T3 awakes from pre-emption and unlock the mutex, allowing T1 to lock it and resume execution. Consequently, T3 has caused T1 with the highest priority to be delayed for 10 time units by a lower priority thread, that is T2.

In RTOS_SC, we avoid this form of priority inversion by applying, in the *consume* function, the priority inheritance protocol, since pre-emption can only occur when this function is called. If a higher priority thread, whether created by the RTOS_SC scheduler or not, is blocked by the mutex/semaphore, the locked thread inherits the priority of the one blocked by the same mutex/semaphore. In case multiple threads are blocked on a semaphore, the locked thread inherits the highest priority blocked thread.

In the scenario explained above, when T1 tries to access the same mutex, which is locked by T3, the RTOS_SC scheduler assigns the priority of T3 to that of T1. Therefore, as opposed to the event trace shown in Figure 11, T3 would execute after T1 is blocked on the mutex. T3 completes then the 5 time units and unlocks the mutex. Consequently, T3 restores its original priority, allowing T1 to pre-empt T3 and access the shared region. When T1 suspends, T2 is selected to execute and consume its 10 time units. T1 was not then delayed for 10 time units by a lower priority thread.

CHAPTER 4

EXPERIMENTAL RESULTS

In this section, we validate the efficiency of the RTOS_SC model by using two Smartphone use cases. In order to validate the accuracy of our RTOS model in determining the application performance, we combine the Jpeg encoder, MP3 playback and voice encoding/decoding (Vocoder) [1] in the first experiment. In the second experiment, we only combine picture encoding with the MP3 playback to be able to show how the application can be optimized. The target platform is the QNX RTOS, running on a 2.8GHz Intel x86 embedded processor. The generated simulation models are running on a 3,2 GHz host Intel i3 Xeon Processor, with linux as the host OS.

4.1 Accuracy of RTOS model in a multithreaded application

The functional accuracy of RTOS_SC is validated by comparing the sequence of the message passing events with that obtained on the target. We use the tasks' response time to measure the timing accuracy. This metric represents the time duration from the occurrence of the event until the time the task produces its results.

Table 4.1 Accuracy of events trace with respect to QNX (in %)

Time (in ms)	RTOS_SC	Model A	Model B
45	100	92.43	71.43

Initially, we would like to show the accuracy of the events trace produced by the scenario illustrated in Figure 2.4, on each of the three models. The execution time of this example is 45ms which is the time for all threads to complete one iteration on the QNX RTOS. As shown in Table 4.1, a significant percentage of error exists in the expected sequence of events when priority inheritance and preemption are not implemented (model A). When only preemption is modeled (model B), an increase in the accuracy is obtained. However, since RTOS_SC implements both features, the application produces the same order of events on RTOS_SC and QNX. The accuracy is calculated by dividing the number of events that occur in the correct sequence (compared to QNX), by the total number of events that occur from the beginning of the program until the specified time.

In our RTOS model, we facilitate debugging of the user code by automatically generating a graph of the software trace of events with the timeline as seen in Figure 4.2. Application developers may then easily validate their software functionality and performance through an illustrated trace of the tasks' states and events during simulation. We use the gnuplot software to graphically generate the events' trace.

Using the QNX Momentics software, we similarly generate illustration of the trace of events obtained by running the application on the QNX target RTOS, as shown in Figure 4.1. The trace of events generated by our model is then compared to that generate on QNX to prove the accuracy of our model.

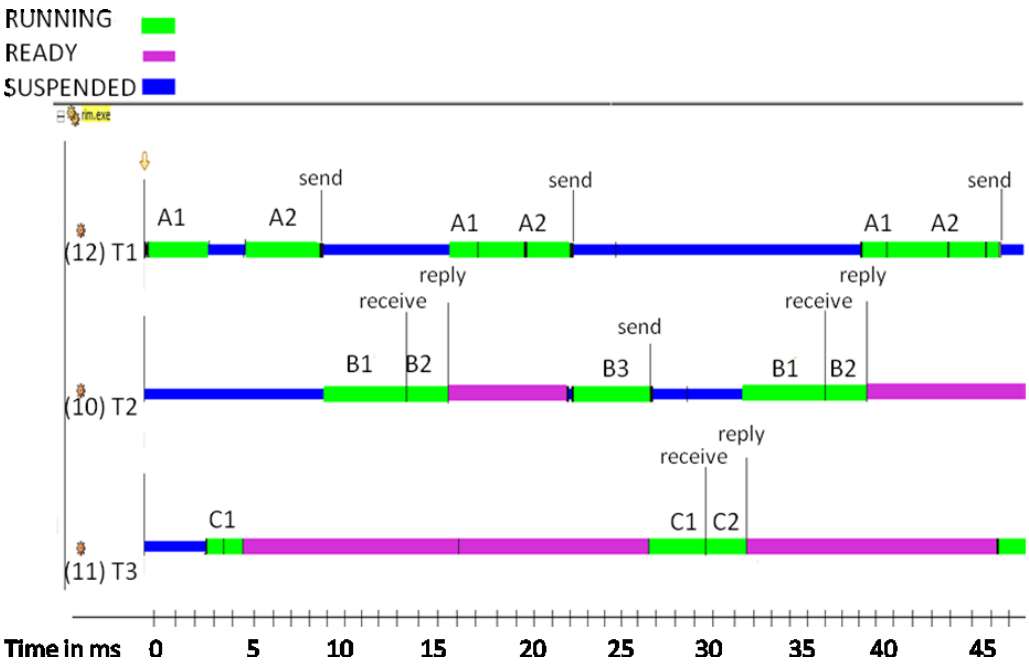


Figure 4.1 Qnx Momentics events' trace

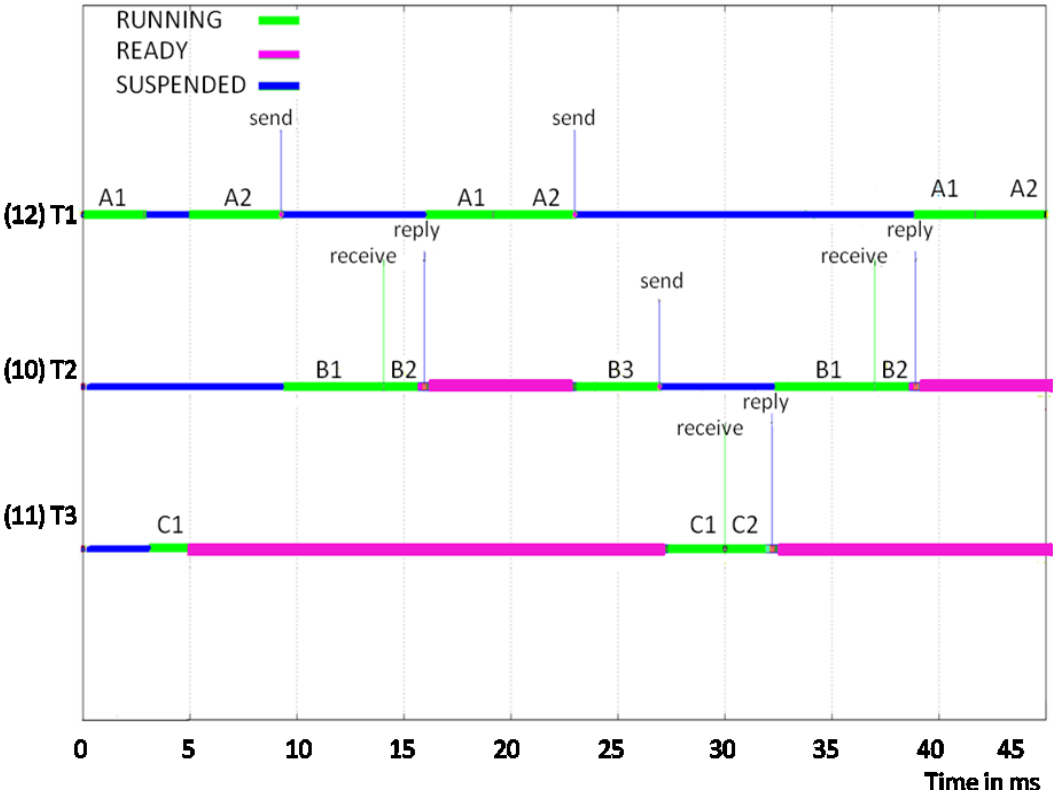


Figure 4.2 RTOS model events'trace in Gnuplot

In both Figures 4.1 and 4.2, *send* signifies that a message has been sent, whereas *receive* indicates that the message is received. *reply* signifies that a reply is sent from the receiver ask to the sender. The transition from RUNNING (shown in green) to READY (shown in purple) indicates preemption. By looking at Figures 4.1 and 4.2, we observe that, by running the example provided in Chapter 2 on QNX RTOS and RTOS_SC, we obtain the same timing and order of tasks' execution.

4.2 Trace of Events

In order to appreciate the impact of accurate trace of events on the application's performance, the same models are tested by combining the MP3+Vocoder and the jpeg encoder. Figures 4.1(i), 4.1(ii), and 4.1(iii) illustrate a scenario where the *capture* task interrupts the execution of the MP3 application and sends a signal to the *jpeg* thread to produce an image every 30ms. This time interval represents the average latency of an image acquisition in modern cameras.

Figures 4.3(i) and 4.3 (ii) illustrate our Smartphone use case of MP3 playback concurrent to a voice call. We assume that the caller wants to play an MP3 clip for the callee, while hearing it himself. The audio from the MP3 file must be decoded and mixed with the audio from the phone call at both ends, so that they can sing along or make comments to each other while the music is playing. Hence, we have 4 audio streams on the caller's phone:

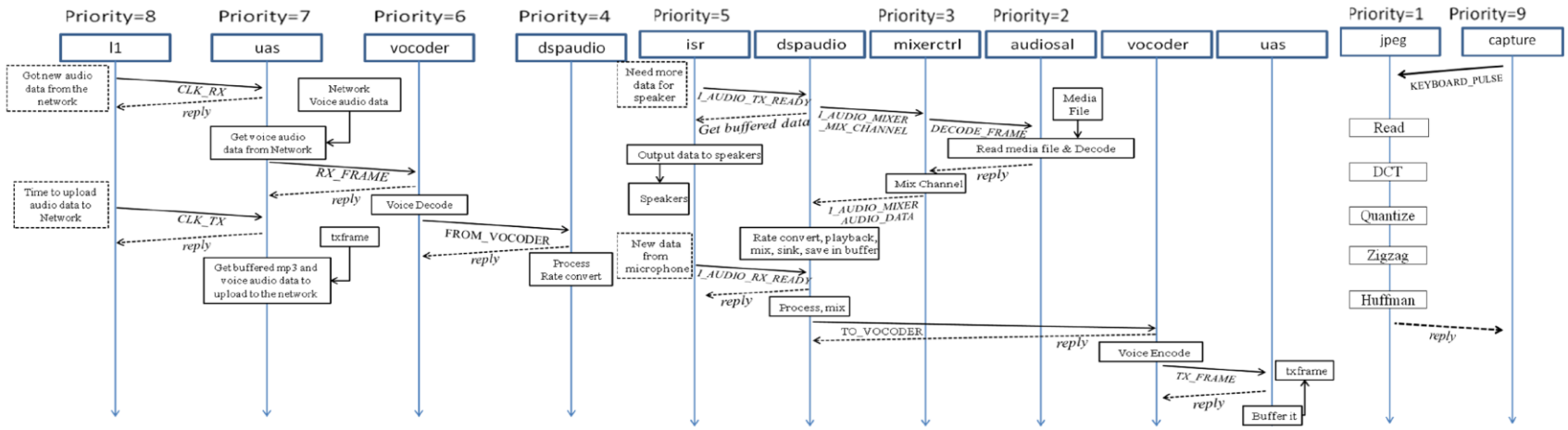


Figure 4.3 (i) Uploading to and Downloading from Network (ii) MP3 Decoding and voice encoding/Decoding (iii) Jpeg Encoding

- Uplink audio: audio being transmitted to the network (including caller's speech mixed with MP3 audio);
- Downlink audio: audio received from the network (only the callee's speech);
- Speaker audio: audio being sent to the phone's speaker (includes callee's speech mixed with MP3 audio);
- Microphone audio: audio coming from the phone's microphone (only the caller's speech);

For each stream there is a message:

- CLK_TX: Interrupt to transmit uplink audio to the network;
- CLK_RX: Interrupt to tell the system that downlink audio has just been received from the network;
- I_AUDIO_TX_READY: Interrupt indicating that the D/A converter needs more data to transmit to the speaker;
- I_AUDIO_RX_READY: Interrupt indicating that the A/D converter received new data from the microphone.

To validate the accuracy of the application, which runs the Mp3 playback concurrently to a voice call, the tasks' order of execution on QNX RTOS and RTOS_SC is shown, for one iteration, in Figures 4.4 and 4.5.

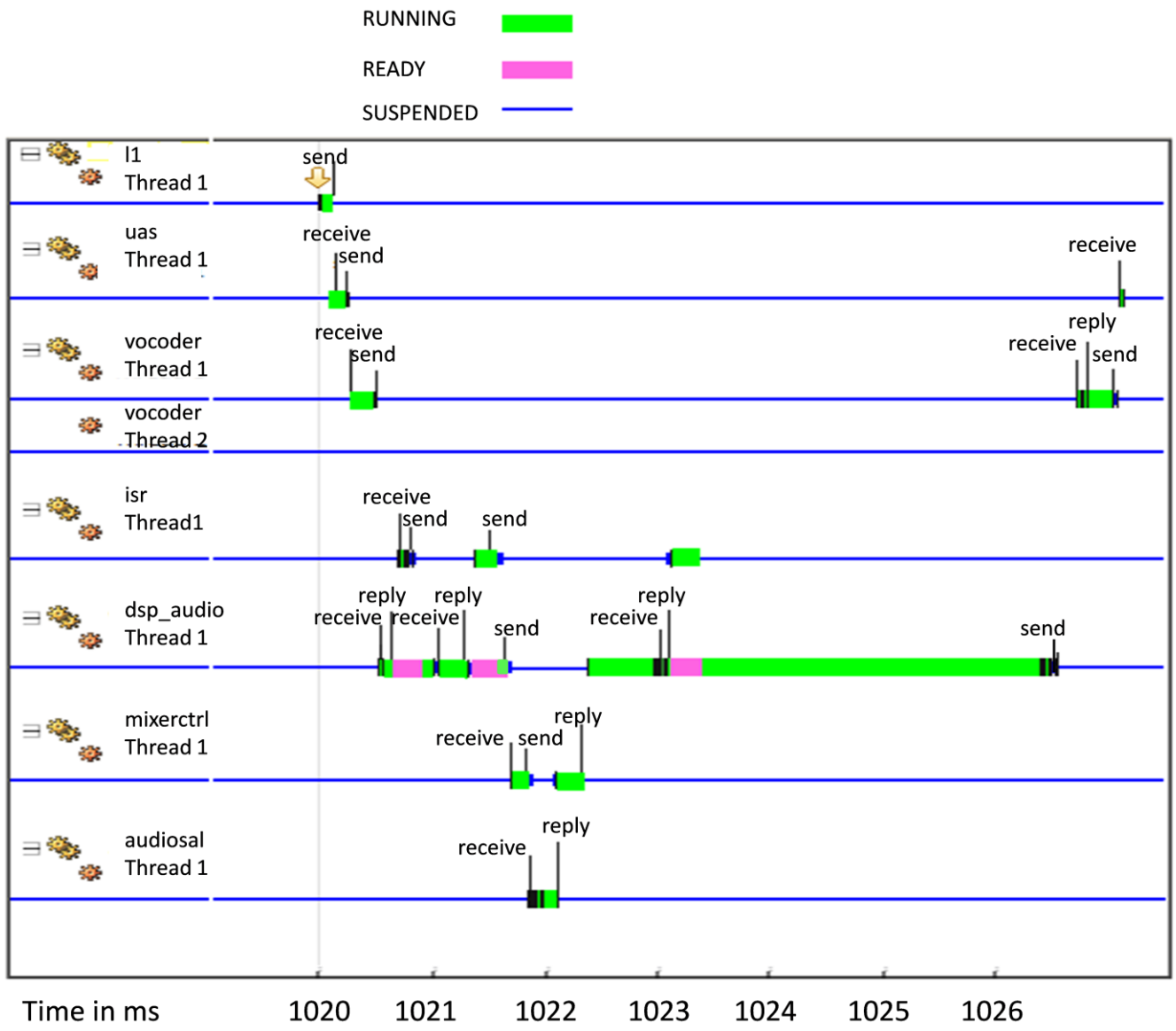


Figure 4.4 Trace of events of Mp3 with voice call on QNX

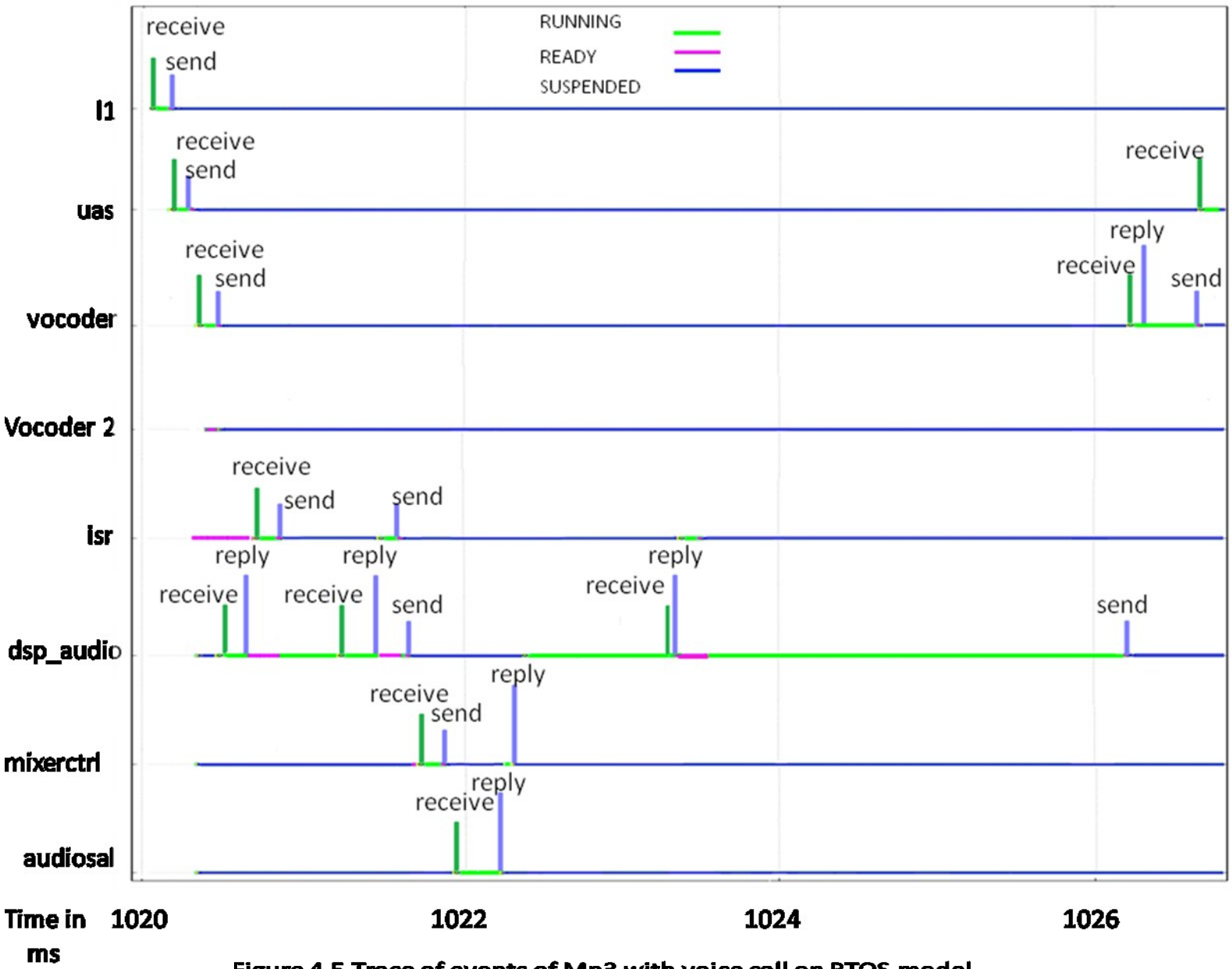


Figure 4.5 Trace of events of Mp3 with voice call on RTOS model

The embedded software consists of seven tasks: l1, uas, Vocoder, dspaudio, isr, mixerctrl, and audiosal. Task l1 implements the uplink and downlink events (CLK_TX and CLK_RX). Task isr is the interrupt handler that notifies the decoding task if more data is needed by the buffer at the speaker (I_AUDIO_TX_READY) or if new data is available at the microphone buffer (I_AUDIO_RX_READY).

As shown in Figures 4.4 and 4.5, the *dsp_audio* task starts execution prior to *isr* task, due to priority inheritance: *vocoder*, with higher priority than both *isr* and *dsp_audio*, sends a message to *dsp_audio* at time 1020. The scheduler therefore assigns the priority of *dsp_audio* to that of *vocoder*. As for preemption it appears at time 1020 and 1023. At 1020, when *dsp_audio* replies to *vocoder*, the original priority of *dsp_audio* is restored to 4, which is less than *isr*'s priority. Therefore, the ready task *isr*, preempts *dsp_audio* and starts execution. Similarly, preemption happens at time 1023, when *dsp_audio* receives the last message sent by *isr*. After receiving and replying to *isr*'s message, *dsp_audio* restores its original priority. Therefore, isr, with higher priority than each of dsp_audio, mixerctrl and audiosal priorities, preempts dsp_audio and resumes execution until it is blocked to wait for the next timer pulse to arrive. As shown by comparing these Figures, a very accurate timing and order of execution is obtained in RTOS_SC. On the other hand, running this application on top of Models A and B, show errors at time 1020 and 1023 for Model A, and at time 1020 for Model B.

Such errors, however, may affect the application functionality if tasks with high computation time, such as image encoding, are integrated into the application, as will be shown in section 4.4.

To encode a jpeg image, the following operations are applied, as shown in Figure 4.3(iii):

1) Read: To read an image in blocks of 8 by 8 pixels

2) DCT: To perform a Discrete Cosine Transform

3) Quantize: To transform from continuous to discrete domain

4) Zigzag: To group similar frequencies

5) Huffman: To apply an entropy encoding algorithm with lossless data compression

In mobile phones, the jpeg image encoder usually has a lower priority than the MP3 decoder since a delay in the display of a picture is more tolerated than a sound delay. For this reason, the jpeg priority is set to the lowest priority among all threads. However, if we consider a device connected to a high speed vision camera, the jpeg encoder would be prioritized over the sound when the camera button is pressed. Therefore, when this application is running, *capture*'s priority is set to 9, which is the highest priority. Consequently, a signal is sent to *jpeg* to operate at the highest priority. For this application, we first select the FIFO scheduling policy and then the Round-Robin policy in order to evaluate its' performance with each policy. The assigned time-slice is 4ms, and it is obtained by multiplying the clock speed on the target platform by 100 (100* 0.04=4).

Table 4.2Accuracy of the trace of event with respect to QNX using the MP3+Vocoder with Jpeg example (in %)

Execution Time(in ms)	Model A	Model B	RTOS_SC
1040	81.97	86.89	98.37
1060	79.61	85.44	99.77
1120	77.73	83.83	99.77
3260	77.01	83.83	98.28
4108	77.01	85.65	98.43

Table 4.2 shows the accuracy of the trace of events generated for each of the models A, B, and RTOS_SC in comparison with QNX. The event trace is calculated for five seconds, running the FIFO scheduling algorithm. As shown in this table, integrating jpeg encoding into MP3+Vocoder leads to a significant error percentage when preemption or priority inheritance are not taken into account in the RTOS model. As a result, it is highly possible that inaccurate timing of events generation causes a significant impact on the efficiency of the application. The average response time of *jpeg* is measured for the same scenario to show the impact of the event inaccuracy on real life applications.

Table 4.3 Response time (in us) of *capture* thread

Time	QNX	Model A	Model B	RTOS_SC
1040 1e3	2001.940	7801.821	8269.856	2002.000
1060 1e3	2001.977	6640.578	6911.366	2002.000
1120 1e3	2002.132	6747.026	7095.797	2002.000
3260 1e3	2002.060	6671.656	6938.491	2002.000
4108 1e3	2.002.138	6667.299	6946.139	2002.000

4.3 Impact of Accurate Trace of Events on Response Time

Table 4.3 shows the time at which the *capture* thread receives a pulse (which means that the camera button is pressed), and the response time of the *capture* task in each of the

QNX and the three RTOS models. The response time obtained by each model should reflect the latency of the picture encoding on the targeted RTOS when this task is required to operate at the highest priority. Since priority inheritance is not taken into consideration in models A and B, the *jpeg* thread remains with the lowest priority and hence, will only execute if all other threads are suspended. As a consequence, a late response time is obtained in these models. However, modeling preemption and priority inheritance in RTOS_SC allows the sender thread (*capture)*, to change *jpeg*'s priority to the highest and therefore, execute with a very accurate response time (approx. 2001us). As a result, the efficiency of the application executing on QNX is correctly interpreted by RTOS_SC.

Task	QNX	Model A		Model B		RTOS_SC	
	Response Time	Response Time	Error %	Response Time	Error %	Response Time	Error %
capture	2002.027	6905.676	245.106	7232.330	261.430	2002.000	-0.001
l1	3133.829	4194.826	33.856	6139.053	95.896	3021.429	-3.586
uas	7284.599	5267.557	-27.689	5268.971	-27.669	6923.527	-4.956
vocoder	9879.735	5101.148	-48.367	4300.476	-56.471	10002.951	1.247
isr	7468.294	4134.918	-44.633	3169.534	-57.560	7408.510	-0.800
dsp	9033.735	2654.239	-70.6185	3676.083	-59.307	9968.693	10.349
mixer	3914.741	9821.942	150.896	2365.556	-39.573	3505.102	-10.464
audiosal	3999.235	7670.799	91.806	2481.873	-37.941	3559.594	-10.993

Table 4.4 Average response time (in us) and error percentage of the *MP3+AUD3* with JPEG tasks (in us), using FIFO policy

4.3.1 Average Response Time, using the First-In-First-Out policy

Table 4.4 shows the average response time and error percentage of each thread of the application illustrated in Figures 4.3 (i), 4.3 (ii), and 4.3(iii), when it is executed on each of

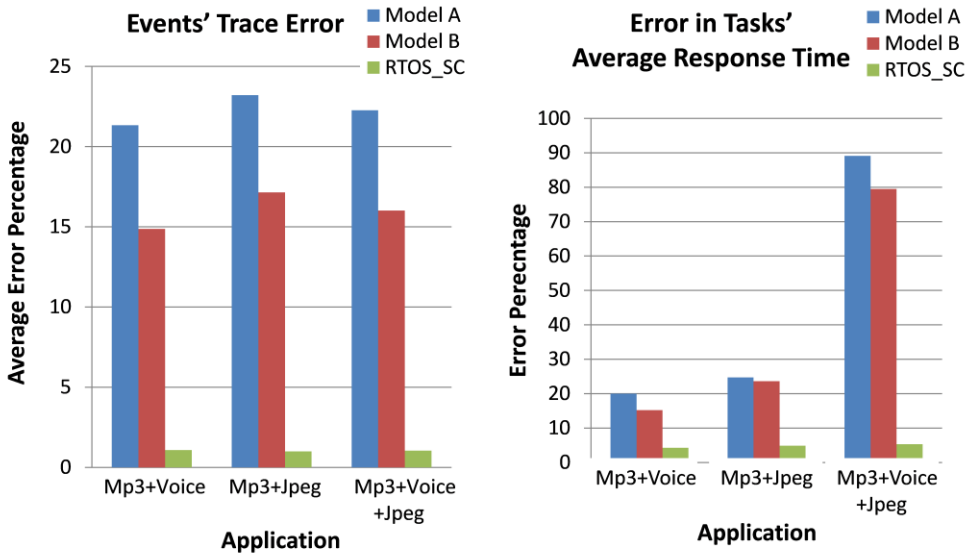
the three models and QNX. In RTOS_SC, the averages of the tasks' response time are approximately equal to the ones obtained on QNX. On the other hand, in models A and B, the modeled response time does not reflect the tasks' response time on QNX, due to the absence of preemption and priority inheritance. To confirm the accuracy of RTOS_SC compared to models A and B, the error percentage of the obtained response times is shown.

To obtain the error percentage we use the following formula: ((model's response time – QNX response time) / QNX response time) * 100. The results show a very small error percentage in the RTOS_SC model, with an average of 5%. This error happens since the time annotated to our model represents the average time consumed by each basic block in the RTOS. Consequently, the annotated times may slightly differ from the time consumed by each basic block obtained on the RTOS. However, models A and B show a significant error, which demonstrates that they are unreliable for software optimization. Furthermore, In the preemptive model (Model B), the percentage of error in the average response time is higher, for some tasks, than the ones obtained in Model A. This result is due to the high dependency of this application on message passing. Due to priority inheritance, priorities are dynamically changed in the QNX RTOS, which affects the preemption time interval, and consequently, the response time of the tasks.

4.3.2 Average Response Time, using the Round-Robin policy

The tasks' response time on QNX and our model are approximately the same when FIFO and Round-Robin policy are selected (Table 4.4), since the time consumed by the jobs in each task do not exceed the time-slice of 4ms. Hence, preemption occurs at the interrupt arrival only when the Round-Robin policy is selected, just as it occurs when the

FIFO policy is set. However, the time required for some tasks to complete a specific job may be greater than the time-slice, which affects the response time of the tasks, as will be seen in section 4.4 of this Chapter.



(a) (b)

Figure 4.6 Application's average (a) error percentage in events's trace, and(b) response time

Figure 4.6 (a) and (b) summarizes the results for the errors percentage, measured on each application with the FIFO policy. Both the error in the events' trace and in the average response time error of an application, must be measured to determine the accuracy of the RTOS model compared to target RTOS. As seen in Figure RTOS_SC has a negligible error percentage, of less than 5%, in the events trace (Figure 4.6(a)) and the tasks' response time (Figure 4.6(b)) for the three applications, which are executed for a period of 5 seconds. However, the error in each of Models A and B is at least 15%.

4.4 Software Validation and Optimization

The results from the previous examples show that the integration of preemption and priority inheritance in RTOS_SC allows for accurate modeling of tasks response time. This implies that this metric may be used in our model to identify the optimization opportunities of software architectures. For instance, task latency is a known issue which may lead to *starvation* in multithreaded systems. The starvation phenomenon happens when READY tasks with low priorities do not execute because of the continuous presence of higher priority tasks in READY state. In RTOS_SC, we use the response time of each task to detect such significant delays. When these delays are detected, the application may be optimized, as will be shown in this section.

Table 4.5 Average response time (in ms) of the *capture* thread with *capture*'s Priority between 1 and 9 using FIFO policy

capture's priority	QNX	Model A	Model B	RTOS_SC
1	39.598	33.394	38.185	38.086
2	39.594	33.236	38.154	38.042
3	35.996	33.342	38.002	36.824
4	34.996	33.592	37.957	35.594
5	32.994	33.600	37.981	33.002
6	32.994	33.559	38.111	33.000
7	33.000	33.775	38.125	33.000
8	33.000	33.482	38.098	33.000
9	33.000	33.564	37.987	33.000

4.4.1 Functional validation using FIFO policy

To show how the starvation is detected and avoided in our model, we use the example of the MP3 and the jpeg tasks (Figures 4.3(i) and 4.3(ii)). We assume that, while MP3 is running, a button is pressed to successively download images that need to be compressed

using jpeg encoder before they are stored in memory. In such a scenario, the sound is prioritized over the jpeg image since a delay in sound is less tolerated than latency in a picture display. On the other hand, the latency for jpeg encoding is desired to be within an interval of 33 to 40ms [23]. Therefore, the goal is to optimize the Jpeg application while preserving the Mp3 and Jpeg's functionalities. We first study the effect of the change in *capture*'s priority on *jpeg*'s efficiency when the FIFO policy is selected.

In Table 4.5, the response time of the *capture* task is measured for one iteration on QNX and compared to the one obtained by each of the three RTOS models. The iteration shown corresponds to the one with the greatest tasks' response time in RTOS_SC. In model A, the change in *capture*'s priority does not affect the response time. Moreover, these response times do not reflect the ones obtained on QNX when capture runs at a lower priority than *isr*. On the other hand, since model B takes preemption into account, an additional delay to the *capture* task is shown (approximately 37ms). However, the delay remains approximately the same in model B even when the priority of *capture* is greater than that of the *isr* task. This is due to the absence of priority inheritance, which makes the *jpeg* task run with the lowest priority and cause the *capture* task to operate at the lowest priority (*jpeg*'s priority). In RTOS_SC, the response times obtained match the ones shown by QNX. The results on QNX and RTOS_SC demonstrate that when *capture*'s priority is greater or equal to *isr*'s priority, the *jpeg* encoding is the fastest. However, to choose which priority is optimal for the *capture* task, we study the effect of the change in *capture*'s priority on Mp3's functionality.

In *isr*, when the MP3 tasks complete their jobs, the mixed data is written to a serial buffer. It is then read and processed by the speakers to generate the Mp3 music as shown

in Figure 4.3(ii). To model the speaker part, we create a task in a separate model, which reads from *isr*'s serial buffer. Consequently, *isr*'s functionality is preserved as long as data is available to be read, and hence, no underflow occurs in the serial buffer. Table 4.6 shows the state of this buffer when the priority of *capture* is changed.

Table 4.6 Average response time (in ms) of the *isr* thread with *capture*'s priority between 1 and 9, using FIFO policy

capture's priority	QNX	Model A	Model B	RTOS_SC
1	4.990	5.000	5.306	5.120
2	5.000	5.117	5.220	5.035
3	4.996	underflow	5.572	5.230
4	4.979	underflow	5.774	5.232
5	underflow	underflow	5.238	underflow
6	underflow	underflow	5.291	underflow
7	underflow	underflow	4.676	underflow
8	underflow	underflow	5.279	underflow
9	underflow	underflow	4.928	underflow

As shown by Table 4.6, the response times for the *isr* task in models A and B do not consistently match the ones obtained by QNX. On the other hand, RTOS_SC accurately models the response time produced by QNX. The response time of *isr* in RTOS_SC and QNX is approximately 5ms when *capture*'s priority is less than or equal 4. Due to priority inheritance, *isr*'s response time is not affected by the change in *capture*'s priority when it varies between 1 and 4 due to priority inheritance. Each of the the Mp3 tasks inherit isr's priority, that is 5, and hence, prohibit the jpeg task with priority 4 from receiving capture's message. However, the *isr* task does not complete execution when *capture*'s priority is greater than 4, as shown by *underflow* in Table 4.6. The starvation of *isr* is

caused by the multiple execution of the *jpeg* task, with the highest priority. As a result of *isr*'s delay, the *speaker* task reads from the serial buffer faster than the rate at which *isr* can write into the same buffer. Therefore, the serial buffer suffers from an underflow of data. Consequently, the underflow causes a halt in the music display which affects MP3's functionality.

The results shown by RTOS_SC in Tables 4.5 and 4.6 demonstrate that the optimal priority for jpeg encoding on QNX is 4, when the FIFO policy is set and the MP3 tasks execute with the priorities defined in Figure 4.3(ii). By assigning a priority of 4 to the *capture* task, MP3+Jpeg application ensures a fast execution of the jpeg encoding process while maintaining a correct functionality in both MP3 and Jpeg applications.

To explain how priority inheritance and preemption affects the performance and functionality of the Mp3+Jpeg application, we illustrate the tasks' different states and events on QNX and the RTOS model using the QNX Momentics and Gnuplot software respectively. Figures 4.7 and 4.8 show the tasks' execution when *capture*'s priority is assigned to 4.

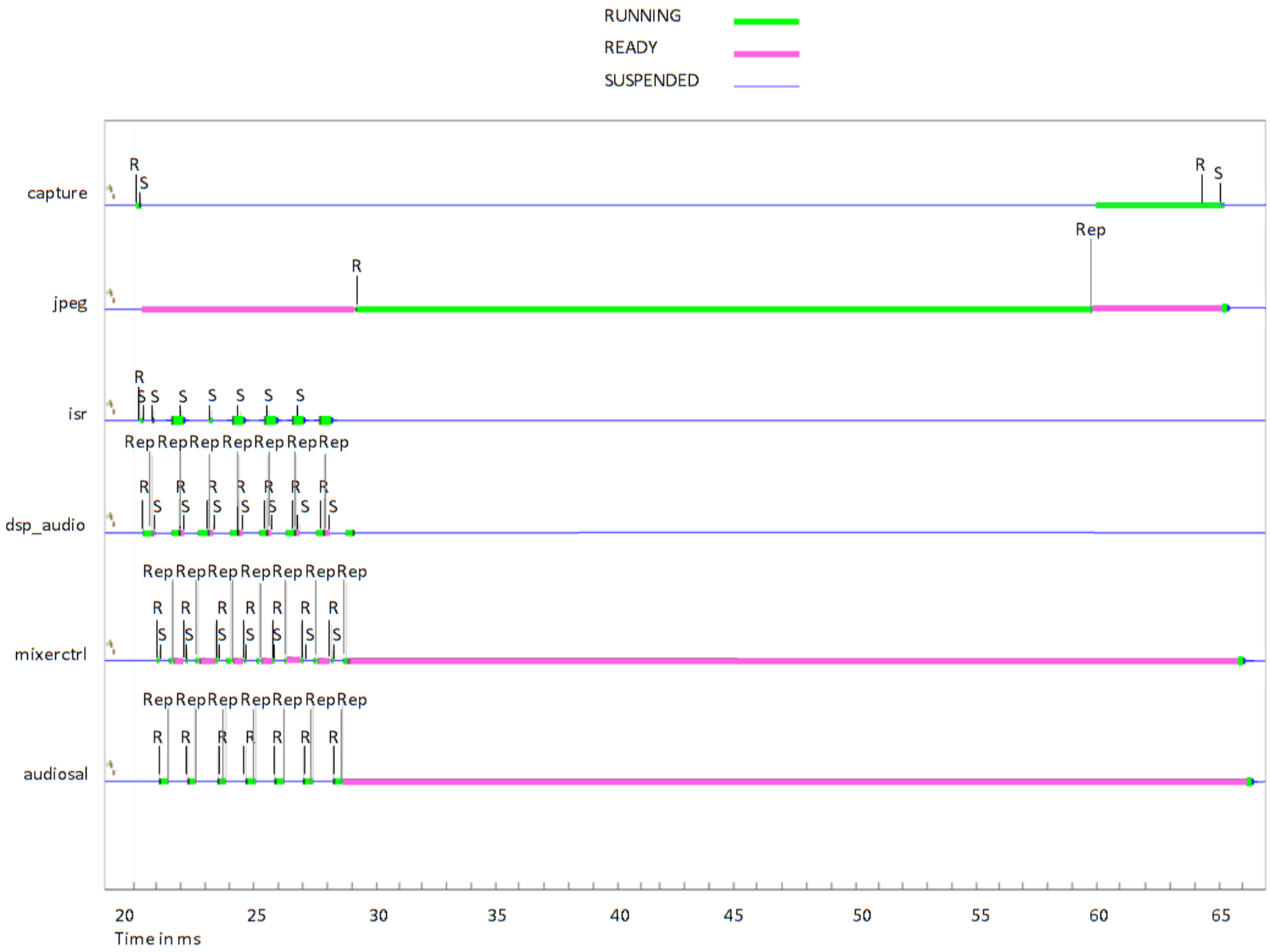


Figure 4.7 Tasks execution of Mp3+Jpeg on QNX with FIFO policy.

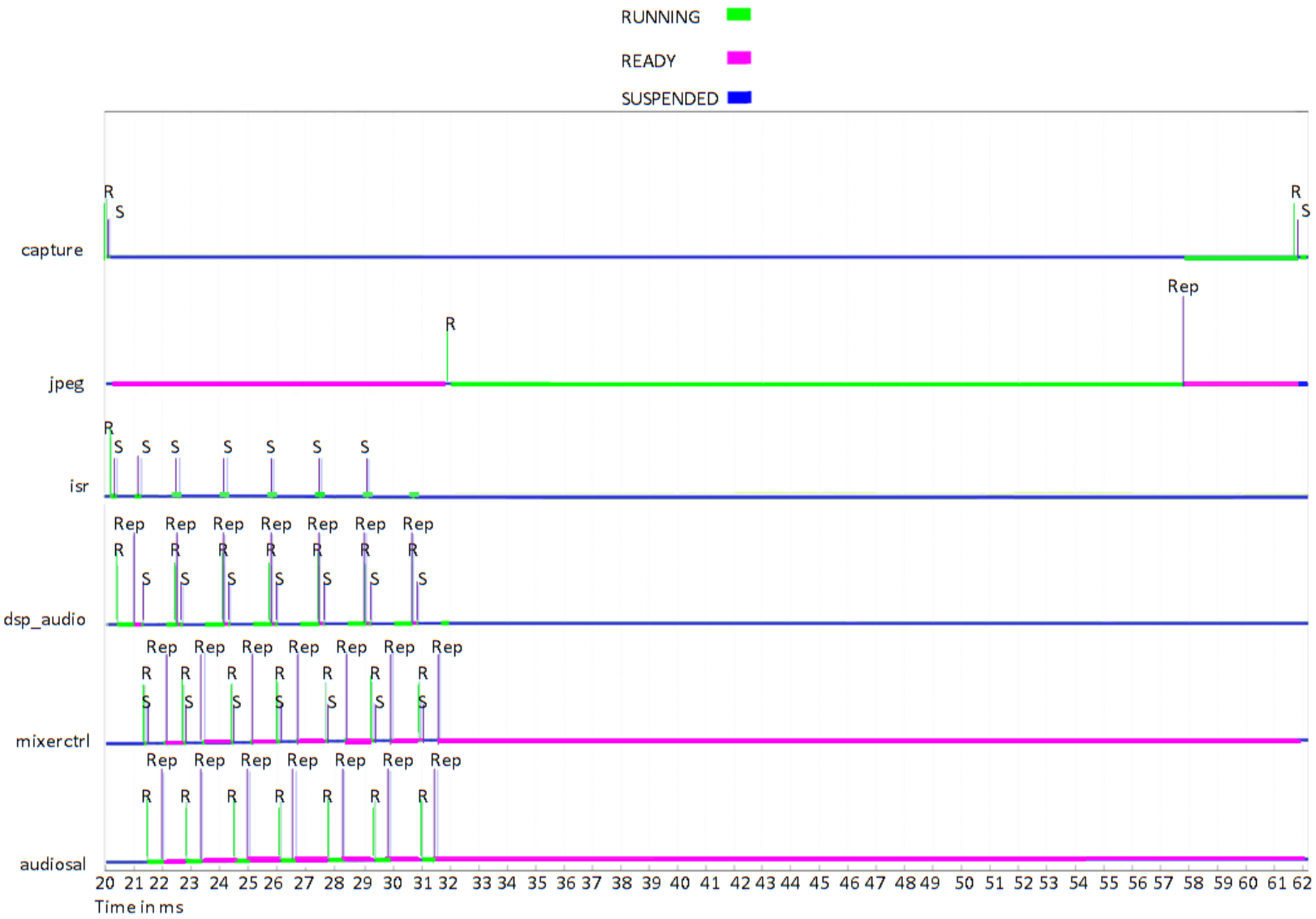


Figure 4.8 Tasks execution of Mp3+Jpeg in RTOS model with FIFO policy.

In this application, we use S to indicate that a message is sent, whereas R is used to indicate that a message is received, and Rep to indicate that a reply has been sent. As shown in Figures 4.7 and 4.8, the pulses for capture and isr are received at the same time (approximately 20ms). However, since the priority of *capture* task is 4, the *isr* task will be able to complete writing to its buffer before the *jpeg* task starts execution. This can be seen in Figures 4.7 and 4.8: the last time *isr* executes (in RUNNING state), it does not send a message to the *dsp_audio* task, which means that isr has completed its job before the execution of *jpeg*. The *isr* task, which has a higher priority that the *capture* task, allows the *dsp_audio*, *mixerctrl* and *audiosal* tasks to inherit its priority, and hence, run at a priority of 5. Therefore, the Mp3 tasks run with higher priorities than the Jpeg tasks (with priority 4). Consequently the following events, which occur in both figures from 20ms until 23ms, are repeated until the completion of *isr*'s job (at 30 ms approximately in both figures):

1) *isr* recives the timer pulse and sends a message to *dsp_audio*
2) *dsp_audio* inherits *isr*'s priority, receives the message from *isr* and sends a reply. After the reply is sent, dsp_audio restores back its original priority of 4.
3) *dsp_audio* is then preempted by isr, with the highest priority, and is therefore moved to READY state.
4) *isr* resumes execution and sends a message to *dsp_audio*
5) *dsp_audio*, inherits *isr*'s priority, and before *dsp_audio* receives *isr*'s message again, *dsp_audio* sends a message to mixerctrl.
6) *mixerctrl* receives then *dsp_audi*o's message with priority 5 and sends a message to audiosal.

7) *audiosal* receives *mixerctrl*'s message with priority 5, and replies to *mixerctrl*. After the reply, *audiosal* restores back its original priority.

8) *mixerctrl,* is then the READY task with the highest priority. It preempts *audiosal*, resumes execution and replies to dsp_audio. After the reply, *mixerctrl* restores back its original priority.

9) *dsp_audio,* with priority 5, preempts *mixerctrl*, runs and replies to *isr* before restoring its original priority.

10) *isr* preempts *dsp_audio* and resumes execution.

After completion of *isr*'s job, *jpeg*, being the only READY task, receives the message from *capture* while hardware task start is concurrently reading the data written to the serial buffer. On the other hand, assigning *capture*'s priority to 5 allows the scheduler to select the jpeg task to start execution before the Mp3 tasks (when capture sends a message to the *jpeg* task, at 20ms). Consequently, the isr task is delayed for 33ms, which is the time taken by *jpeg* to produce its results. Due to the concurrent reading of the hardware task from the serial buffer, an underflow occur in the serial buffer in both QNX and our RTOS model as shown in table 4.6.

Table 4.7 Average response time (in ms) of the *capture* thread with *capture*'s Priority between 1 and 9 using RR policy

capture's priority	QNX	Model A	Model B	RTOS_SC
1	38.956	38.394	38.185	38.320
2	38.793	38.236	38.154	38.272
3	39.011	38.342	38.002	38.073
4	37.996	38.592	37.957	38.363
5	35.981	38.600	37.981	36.770
6	32.966	38.559	38.111	33.000
7	33.002	38.775	38.125	33.000
8	33.000	38.482	38.098	33.000
9	33.000	38.564	37.987	33.000

4.4.2 Functional validation using RR policy

In this section, we take the same application illustrated in Figures 4.3(ii) and 4.3(iii). However, we select Round-Robin as the scheduling algorithm. We would like to study the effect of this algorithm on the application functionality.

Compared to Table 4.5 where the FIFO scheduling algorithm was running, Table 4.7 shows, in both QNX and RTOS_SC model, a slightly higher response time for the *capture* task when its priority is between 1 and 4. This delay is due to rescheduling at the end of the time-slice which allows the MP3 tasks to execute, and preempt the *jpeg* task. However, when the priority of *capture* is greater than any priority of the MP3 tasks (> 5), the *capture* task's response time is within the desired range (approximately 33ms). To conclude which policy provides better results for this application, we validate *isr's* functionality, as shown in Table 4.8.

Table 4.8 Average response time (in ms) of the *isr* thread with *capture*'s priority between 1 and 9 using RR policy

capture's priority	QNX	Model A	Model B	RTOS_SC
1	4.427	8.176	5.190	4.359
2	4.052	7.958	4.220	4.971
3	3.726	8.802	5.618	4.650
4	3.802	8.572	4.972	5.012
5	102.986	8.992	5.066	103.372
6	underflow	8.080	4.620	underflow
7	underflow	9.337	3.930	underflow
8	underflow	8.916	5.563	underflow
9	underflow	7.857	4.876	underflow

Table 4.8 shows *isr*'s response time when the RR scheduling algorithm is selected. Models A and B do not accurately model the response time obtained at the target QNX RTOS. As for RTOS_SC, the response time is accurately modeled. Although *isr*'s response time increases in both QNX and RTOS_SC model when *capture* runs at an equal priority to *isr* (*capture'*s priority = 5), *isr'*s buffer does not experience an underflow at this priority. The increased latency in the *isr*'s response time when capture priority is set to 5 is not important, as long as there is no underflow in the *isr*'s buffer.

Since the time consumption required for *jpeg* is greater than the time slice, rescheduling occurs at the end of each time-slice, which causes preemption of the *jpeg* task. Consequently, using the Round-Robin policy, *isr* and *capture* may run at equal priorities while ensuring correct functionality of the MP3+JPEG application. When the FIFO policy is selected, however, having *isr* and *capture* running at an equal priority caused the starvation of *isr* (Table 4.6). The underflow in *isr*'s buffer cannot be avoided in

both policies if the priority of *capture* is greater than that of the *isr* task, since the *capture* task is continuously selected for execution.

To obtain the best performance for this application while maintain correct functionality, the priority of *capture* must be 4 if the FIFO policy is selected, and 5 if the Round-Robin policy is selected. Since the sound is more prioritized over jpeg encoding in this application, selecting any of the scheduling policies ensure correct functionality. In applications where the sound display is equal in priority to jpeg encoding, the round-robin policy must be selected, to guarantee a minimum delay in the jpeg encoding and continuous display of the MP3 sound clip. However, the jpeg encoding many not be prioritized over the Mp3 display, if the main tasks have close deadlines to meet and high computation time, as shown in our Mp3+Jpeg application. By accurately modeling preemption and priority inheritance, we proved therefore, that our RTOS model allows the application software designers to identify the optimization opportunities of applications running on a target processor through accurate evaluation of the application performance.

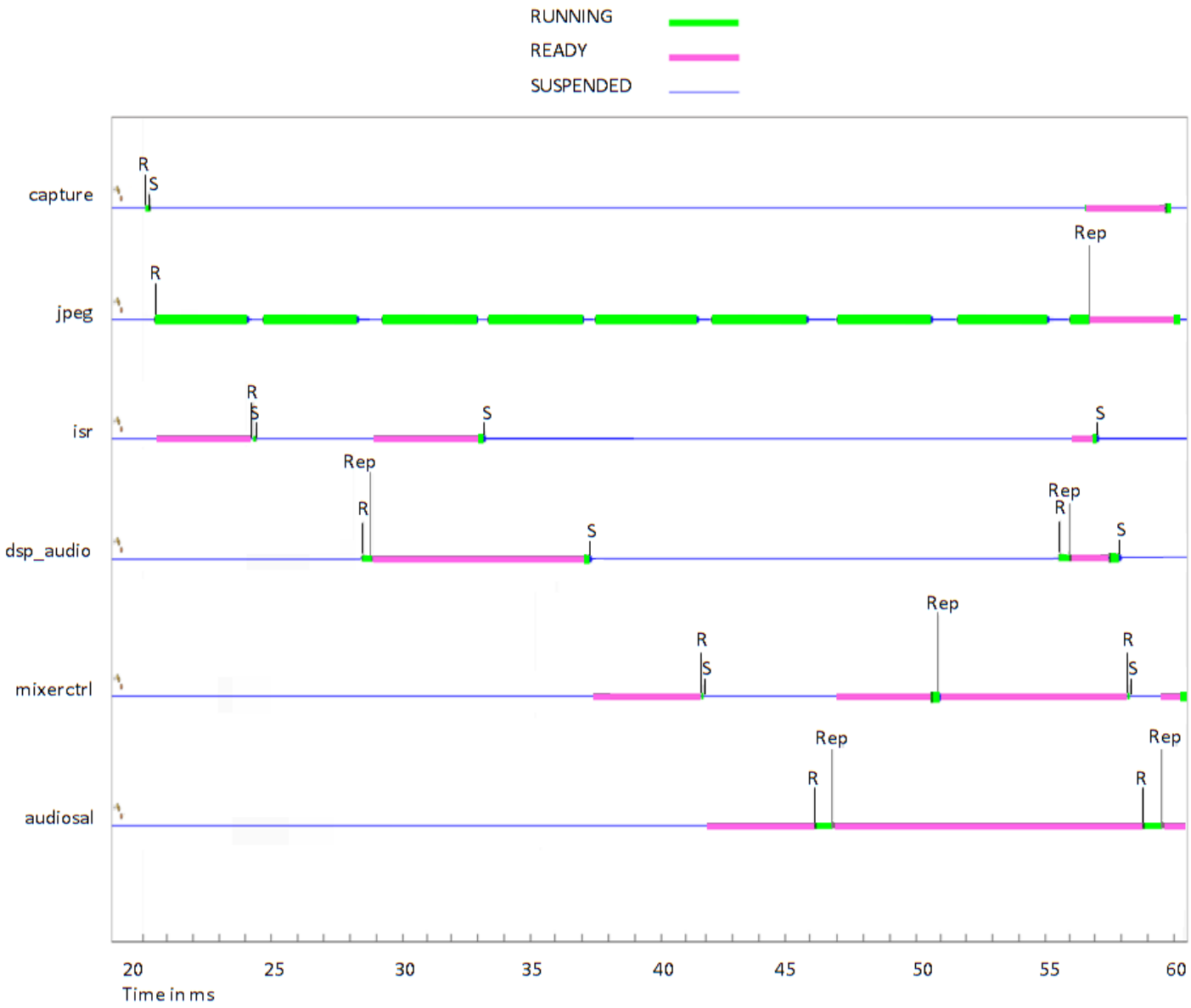


Figure 4.9 Tasks execution of Mp3+Jpeg on QNX with Round-Robin policy.

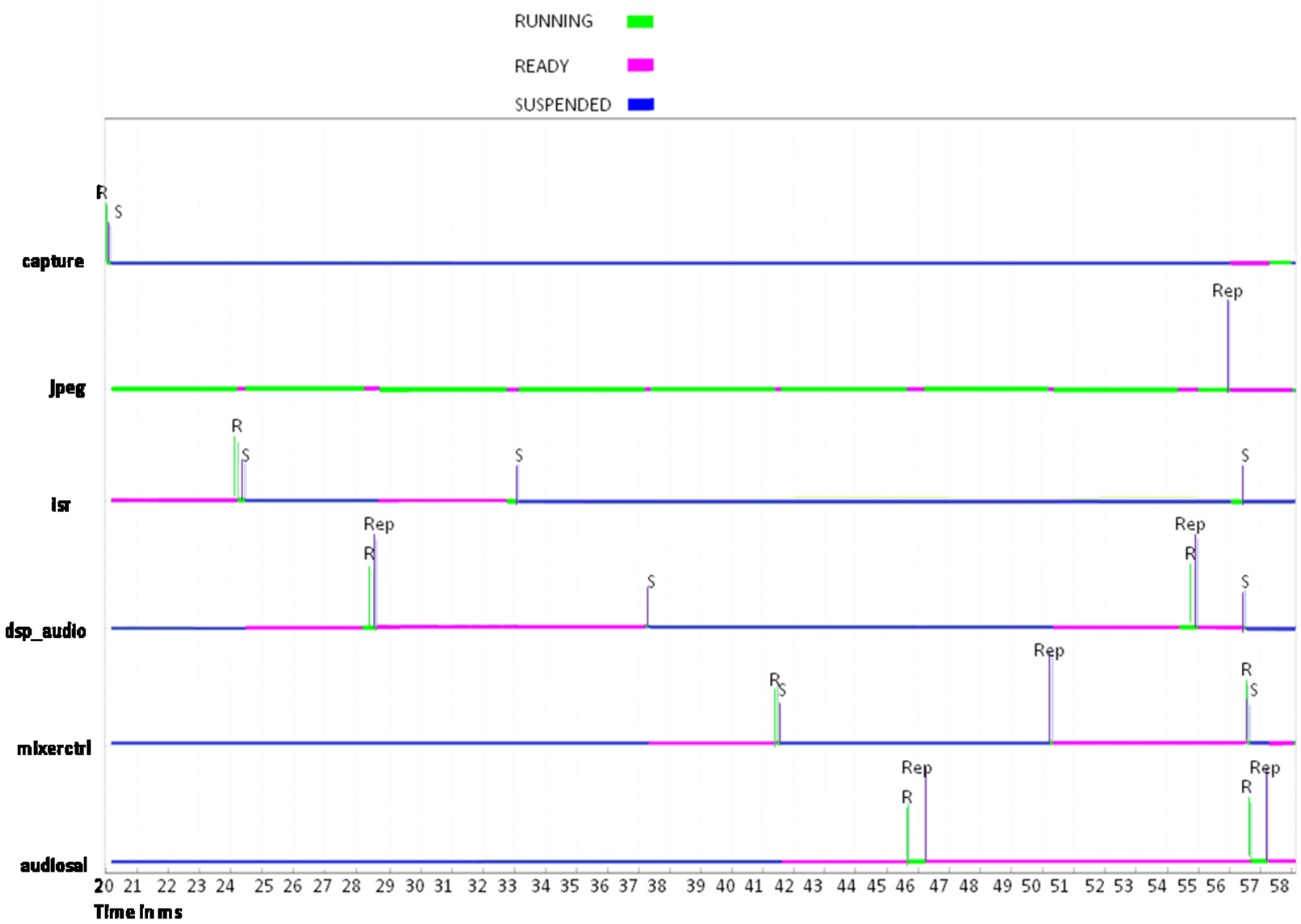


Figure 4.10 Tasks execution of Mp3+Jpeg in RTOS model with Round-Robin policy.

Figures 4.9 and 4.10 illustrate the tasks states and events on QNX and our RTOS model during the execution of Mp3+Jpeg. For the clarity purpose, we show the tasks execution until the first time the *isr* task writes the processed data into the the buffer. The same scenario is then repeated until *isr* completes writing to the buffer. For this trace, the Round-Robin policy is selected and *capture*'s priority is set to 5. As seen in Figures 4.9 and 4.10, the Round-Robin scheduling algorithm allows preemption to occur at the end of every time slice (with a period of 4ms). Therefore, the *isr* task, gets the chance to write into the serial buffer at a faster rate than the reading of the hardware task from the same buffer. Consequenly, the serial buffer does not experience an underflow, as shown in table 4.8. The delay in *isr*'s response time, however, is due to the alternative execution between the Mp3 tasks and the Jpeg tasks. Since the events and the timing of the events' occurences in our RTOS model are accuarate compared to the QNX timing, we chose Figure 4.10 to explain the tasks' order of execution in both QNX and RTOS_SC:

1) At 20ms, the *capture* task sends a message to jpeg to start encoding.
2) At 20ms, *jpeg* task receives *capture*'s message and starts the encoding process, which takes 33ms. However, since 33ms is greater than the assigned time slice of 4ms, *jpeg* executes until the end of the time slice (at 24ms).
3) At 24ms, *isr* receives a timer pulse and sends a message to *dsp_audio*
4) At 24ms, *dsp_audio* inherits *isr*'s priority and becomes with priority 5. However, the *jpeg* task, with equal priority to that of *dsp_audio*, is the older task in READY state. Therefore, *jpeg* resumes execution until 28ms.
5) At 28ms, *dsp_audio* receives the message from *isr* and sends a reply. After the reply

is sent, dsp_audio restores back its original priority of 4.

6) *dsp_audio* is then preempted by *jpeg*, since it is the oldest READY task with the highest priority. *dsp_audio* is therefore moved to READY state and jpeg resumes execution until 32ms.
7) At 33ms, *isr* resumes execution and sends a message to *dsp_audio*
8) *dsp_audio*, inherits *isr*'s priority and moves to READY state. However, jpeg task is the oldest task with priority 5 in the READY state. Therefore, *jpeg* resumes execution for 4ms.
9) At 37ms, before *dsp_audio* receives *isr*'s message again, *dsp_audio* sends a message to mixerctrl. *jpeg* task then preempts *mixerctrl*, being the older task in the READY state. It then resumes an additional execution of 4ms.
10) At 41ms, *mixerctrl* receives then *dsp_audi*o's message with priority 5 and sends a message to audiosal. *jpeg* then resume execution for 4ms.
11) At 45ms, *audiosal* receives *mixerctrl*'s message with priority 5, and replies to *mixerctrl* at 46ms. After the reply, *audiosal* restores back its original priority. Further, *jpeg* resumes execution of 4ms.
12) At 50ms, *mixerctrl ,* is then the READY task with the highest priority. It preempts *audiosal*, resumes execution and replies to dsp_audio. After the reply, *mixerctrl* restores back its original priority. *jpeg* then resumes execution of 4ms.
13) At 54ms, *dsp_audio,* with priority 5, preempts *mixerctrl*, runs and replies, at 55ms, to *isr* before restoring its original priority. At 55ms, *jpeg* resumes execution for 1ms since the remaining time from 33ms is 1, which is less than the time slice period of 4ms.

14) At 56ms, jpeg sends a reply to *capture* task and therefore, capture is moved to ready state. However, *isr* is the older task with the highest priority in the READY state. *isr* then runs and sends a message to *dsp_audio*.

15) At 56ms, *dsp_audio* sends a message to *mixerctrl*, which sends a new message to *audiosal*.

16) At 57ms, when the *audiosal* sends a reply to *mixerctrl*, *capture* task resumes execution since it is the older READY task with priority 5.

4.5 Model Execution Speed

The execution speed of our SystemC-based RTOS model is an important quality metric of the modeling methodology. It is well known that ISS-based simulation is typically a few orders of magnitude slower than real-time execution of software on the target. Our host-compiled SystemC model, on the other hand, was found to be much faster than the real-time software execution on target. The SystemC model was executed on an Intel i3 processor running at 3.20GHz.

Number of Iterations	Execution/Simulation Time (H:MM:SS.ms)			
	MP3		MP3 + Vocoder	
	QNX	SystemC	QNX	SystemC
1	0:00:01.198	0:00:00.01	0:00:01.033	0:00:00.01
2	0:00:01.368	0:00:00.01	0:00:01.062	0:00:00.01
5	0:00:01.920	0:00:00.07	0:00:01.118	0:00:00.01
10	0:00:02.840	0:00:00.13	0:00:01.218	0:00:00.03
100	0:00:19.400	0:00:01.45	0:00:03.016	0:00:00.48
1000	0:03:05.0	0:00:20.29	0:00:21.018	0:00:04.93
10000	0:30:41.0	0:02:34.07	0:03:21.018	0:00:50.74
100000	5:06:41.0	0:30:45.58	0:33:21.018	0:08:41.90
1000000	51:06:41.0	4:36:01.44	5:33:21.018	1:28:07.38

Table 4.9 Speed comparison of on-target software vs. model

Table 4.9 compares the timing of the application software execution on QNX target system versus the SystemC execution on host. Several numbers of iterations were measured, ranging from one to a million. The time format in the table is *Hours:Minutes:Seconds.milliseconds*. For a few iterations (<1000), there is no perceptible difference, although the SystemC model is much faster. However, as the number of iterations increase, we find a significant advantage to using the SystemC model. For a million iterations, the MP3 model (Figure 4.3(ii)) is over 11X faster, and the MP3+Vocoder (Figures 4.3(i) and 4.3(ii)) model is 3.8X faster than the respective real-time software.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we have presented an RTOS model that can be used for accurate and early system validation. The advantage of our methodology over other proposed techniques is that it accurately models preemption and priority inheritance, which makes it feasible to incorporate the average response time metric into our model and use it as a tool for software optimization. We validated our model using industrial size examples. Our results show that our RTOS model is significantly faster (up to 11X) than real-time software execution on target platform, and the estimated performances are very accurate, with an average error of 5%, as compared to the target platform. Furthermore, we proved that our modeling of the priority inheritance protocol during I/O and inter-task communication, leads to an accurate evaluation of the application functionality. Therefore, our model can be used by software designers to early determine the optimization opportunities.

In the future, we expect to expand our RTOS model to include tasks running on multi-core target platforms. To achieve this goal, modifications have to be done in the modeled RTOS scheduler in order to take into consideration multiple tasks executing in parallel. Furthermore, we expect to incorporate power measurement into our RTOS model, and therefore obtain an early and accurate evaluate of the application performance in terms of execution time and power.

References

[1]. Lee, R., K. Abdel-Khalek, and S. Abdi. Early System Level Modeling of Real-Time Applications on Embedded Platforms. *Quality Electronic Design,* ISQED, March 2013.

[2]. Hadjiyiannis, G., S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. *Design Automation Conference*, 1997.

[3]. Gligor, M., N. Fournel, and F. Petrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. *Hardware/software codesign and system synthesis*, CODES+ISSS, France, October 2009.

[4]. Hassan, M.A., K. Sakanushi, Y. Takeuchi, and M. Imai. RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC. In *Proceedings of the Design, Automation and Test Conference*, IEEE, 2005.

[5]. Yoo, S., I Bacivarov, A. Bouchhima, Y. Paviot, and A.A. Jerraya., TIMA Laboratory, FR. Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer. *Design, Automation and Test in Europe Conference and Exhibition,* DATE, 2003.

[6]. Desmet, D., D. Verkerst, and H. De Man. Operating System Based Software Generation for System On Chip. In *Proceedings of DAC*, June 2000.

[7]. Honda, S., T. Wakabayashi, H. Tomiyama, and H. Takada. RTOS-centric HW/SW Cosimulator for Embedded System Design. *Hardware/Software Codesign and System Synthesis,* CODES+ISSS, 2004.

[8]. Pasquier O., and J-P. Calvez. An object-base executable model for simulation of real-time Hw/Sw systems. In *Proceedings of DATE 99*, Munich, March 1999.

[9]. Lavagno, L., C. Passerone, V. Shah, and Y. Watanabe. A time slice based scheduler model for system level design. In *Proceedings of Design, Automation and Test in Europe.* March, 2005.

[10]. Yoo, S., G. Nicolescu, L. Gauthier, and A.A. Jerraya. Automatic Generation of Fast Timed Simulation Models for Operating Systems. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, IEEE, 2002.

[11]. He Z., A. Mok, and C. Peng. Timed RTOS Modeling for Embedded System Design. In *RTAS*, San Francisco, 2005.

[12]. Gerstlauer, A., H. Yu, and D. Gajski. RTOS Modeling for System-Level Design. In A.A. Jerraya, S. Yoo, D. Verkest, and N. When (eds.), *Embedded Software for SoC*. Springer, 2003.

[13]. Le Moigne, R., O. Pasquier, and J. P. Calvez. A generic RTOS model for real-time systems simulation with systemC. In *proceedings of Design, Automation and Test in Europe Conference and Exhibition,* February 2004.

[14]. Hessel, F., V.M da Rosa, I.M. Reis, R. Planner, C.A.M. Marcon, and A.A. Susin. Abstract RTOS modeling for embedded systems. In *proceedings of Rapid System Prototyping,* IEEE, 2004.

[15]. Tomiyama, H., Y. Cao, and K. Murakami. Modeling Fixed-Priority Preemptive Multi-Task Systems in SpecC. In *Proceedings of the 10th Workshop on System And System Integration of Mixed Technologies* (SASIMI'01), IEEE, 2001.

[16]. Shaout, A., K. Mattar, and A. Elkateeb. An ideal API for RTOS modeling at the system abstraction level. *Mechatronics and Its Applications,* ISMA, May 2008.

[17]. Posadas, H., J.A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in

SystemC for real-time embedded SW simulation: A POSIX model. *Design Automation for Embedded Systems*, December 2005.

[18]. Posadas, H., E. Villar, and F. Blasco. Real-Time Operating System Modeling in SystemC for HW/SW co-simulation. In *Proceedings of DCIS,* IST, Lisbon, 2005.

[19]. Schirner, G., and R. Domer. Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling. *Design, Automation and Test in Europe, 2008. DATE,* March 2008

[20]. QNX Realtime Operating Systems (RTOS) Software, available http://www.qnx.com

[21]. OSCI TLM-2.0 user manual, Open SystemC Initiative, 2008.

[22]. Perry West. High Speed, Real-Time Machine Vision. http://imagenation.com/pdf/highspeed