Support for Nested Preemption in nxtOSEK

Real Time Computing Systems Project Report
http://www4.ncsu.edu/ sgupta20/pts.html

Saransh Gupta
sgupta20@ncsu.edu

Jimit Doshi
jdoshi@ncsu.edu

1. INTRODUCTION

In real-time computing scenarios, we are concerned with logical as well as temporal accuracy of the results. While developing algorithms or performing schedulability analysis, preemption overheads are usually ignored even, though it's an operation that consumes finite amount of time. Preemption allows higher priority tasks to halt the lower priority tasks, though this need not be necessarily required to meet task deadlines. Our concern is not to get the results as soon as possible but soon enough to just meet the deadlines of all the tasks. Algorithms which meet deadlines and minimize preemptions in order to decrease overheads have been developed.

The nxtOSEK is a Real-Time Operating System for the AT91SAM7S256 ARM7TDMI controller, with capabilities of ANSI-C/C++ programmability. The nxtOSEK kernel cannot be used to demonstrate any such algorithm, since it does not support nested preemption. The task scheduling in the nxtOSEK kernel is governed by a 1ms ISR routine. When a higher priority task preempts a lower priority task, it has been observed that this 1ms ISR(and hence the task scheduler) is suppressed for the duration of execution of the higher priority task. This causes an equivalent shift in the release times of all the subsequent tasks and the entire timing behaviour of the nxtOSEK kernel is skewed.

The nxtOSEK kernel has been purposefully designed to prevent successive instances of preemption i.e. the kernel does not support a case where after one preemption event, while the preempting task is yet to finish its execution, it is preempted in turn by an even higher priority task. In this project, we attempted to resolve this by modifying the current kernel behavior. The article has been divided into five sections, Background, that explains the current kernel behavior and highlights the problem in it, Design, that encompasses the suggested modifications in the kernel, Implementation, depicting the changes done to achieve the goal, Validation, defines the test-cases used to test the modified kernel and Results,presenting the output of the modification.

2. BACKGROUND

We describe below some of the relevant implementation details regarding nxtOSEK:

1. *runtsk* : The currently running task's task ID is stored in runtsk.
2. *schedtsk* : This is the task that the scheduler evaluates as the highest priority task.
3. File structure :
   a. *irq.s* - Has the assembly routine for IRQ interrupt handler.
   b. *cpu_support.s* - Assembly routines to dispatch and preempt tasks.
   c. *init.s* - SWI ISR handler added here.

We present below a high level discussion of nxtOSEK's current algorithm to deal with preemption, it's limitations and our approach to resolve them. Let us first consider the default behavior of the nxtOSEK kernel and understand how it doesn't allow nested preemption. The nxtOSEK's kernel scheduler runs from a 1ms timer ISR. Every 1 ms the currently running task is interrupted and the scheduler runs to look for ready tasks based on the period of each task (for a typical periodic system). Accordingly, the highest priority task amongst all the ready tasks and the currently running task is evaluated and updated as schedtsk. Thereafter the kernel compares the values of schedtsk and runtsk. If they are the same, it implies that the currently running task is the highest priority task and no preemption is required. However, if they are different, then it is a case of preemption as a higher priority task than the current runtsk is now ready. Consider the simpler case of non-preemption first. If schedtsk and runtsk are equal, then scheduler has to return back to the interrupted task from the ISR directly and resume task execution from the point where it was interrupted. However in case of preemption, the higher priority task is run from the ISR itself i.e. the ISR is not 'returned' as long as the higher priority task's execution is not completed. Thereafter, the execution control returns back to the pending ISR. From here, the ISR returns back to the preempted task

to resume it's execution. Evidently, since the preempting higher priority task is run from the ISR itself, it leads to the following while it is being executed:

1. No kernel/scheduler calls while the higher priority task is running. This is because the current ISR routine (which runs the scheduler) has not 'returned' yet.

2. Nested preemption is not possible, as no new higher priority task shall be evaluated for readiness since the kernel itself is no longer running every 1 ms (assuming the more general case of the higher priority task having an execution time greater than 1ms).

3. Skewed (delayed) release times of all subsequent tasks as the scheduler is suspended for the execution time of the higher priority task.

3. SYSTEM DESIGN

We attempted to modify the nxtOSEK kernel so that it supports nested preemption. For every instance of the 1ms ISR when the kernel is executed, the basic kernel activities such as updating the list of ready tasks and selecting the highest priority schedtsk remains unchaged. So does the execution flow in case schedtsk is evaluated to be the same as runtsk i.e. a case of no preemption. However, in case of preemption, we propose the following algorithm:

1. Identify that preemption needs to be done based on the comparison between schedtsk and runtsk.

2. Save the LR (link register) value of ISR which represents the address from which the low priority task's execution should resume. Begin execution of the higher priority task by 'returning' from the 1ms ISR to the higher priority task i.e. while the lower priority task was originally interrupted by the 1ms ISR for kernel execution, the ISR actually 'returns' to a higher priority task. This serves two purposes:

    a. 1ms ISR execution is completed before the high priority (preempting) task's execution is started. As a result, while the high priority task is running, the 1ms ISR can reoccur periodically and the kernel is no longer suspended unlike the case in current implementation of nxtOSEK (as detailed above).

    b. Preemption of the lower priority task by higher priority task i.e. transfer of execution control to the higher priority task.

3. Whenever the low priority task is scheduled again, it's execution should resume from the LR value of the ISR saved previously in step 2 above.

4. IMPLEMENTATION

Here we discuss the specifics of code implementation for our suggested approach. Wherever necessary, we have compared it against the current nxtOSEK implementation and also provided relevant details of the ARM architecture.

1. ARM processor has several operating modes such as IRQ, FIQ, System, Supervisor etc. which are invoked by a program's execution status (exceptions, interrupts, undefined memory reference, etc). All of these modes have certain privileges and banked copy of SPSR (Saved Program Status Word), SP (stack pointer) and LR (link register) registers. System and User mode share exactly the same registers but System mode has additional privileges.

2. At any given point of time during a task's execution in the User mode, the CPSR (Current Program Status Register) register maintains details such as the processor mode, interrupt enable/disable status and flags corresponding to the most recent ALU operation (negative, overflow, zero, etc). Whenever an interrupt/exception occurs, a corresponding mode change is performed by the ARM core (for e.g. in case of a 1ms timer interrupt, the processor mode changes from the User mode to the IRQ mode). At this instance, the current value of CPSR of the User mode is copied into the SPSR value of the new mode. Also, the address of the next instruction in the task that was interrupted in the User mode is copied to the banked LR register of the new mode by the core. Moreover, it is the banked versions of the SP and LR registers (for e.g. sp_irq and lr_irq instead of sp and lr of the User mode) that are effectively used in the new mode now. This sequence helps to restore the context back to the User mode whenever required, as the CPSR for the User mode can be set back to the saved SPSR value of the current mode and address of the instruction to be returned to in the User mode is recovered from the LR of the new mode.

3. We consider the execution flow from the instant the 1ms timer ISR occurs. Consider a certain low priority task Task_LP is being executed when the ISR occurs. The ARM core changes the mode from User mode (tasks are run in the User mode) to the IRQ mode. The CPSR value at the instance of interruption for Task_LP is saved in IRQ_spsr and IRQ_lr has the address of instruction in Task_LR from where it's execution should be resumed after the ISR returns. The interrupts are also disabled by the core.

4. To support nested interrupts, as depicted in Fig-

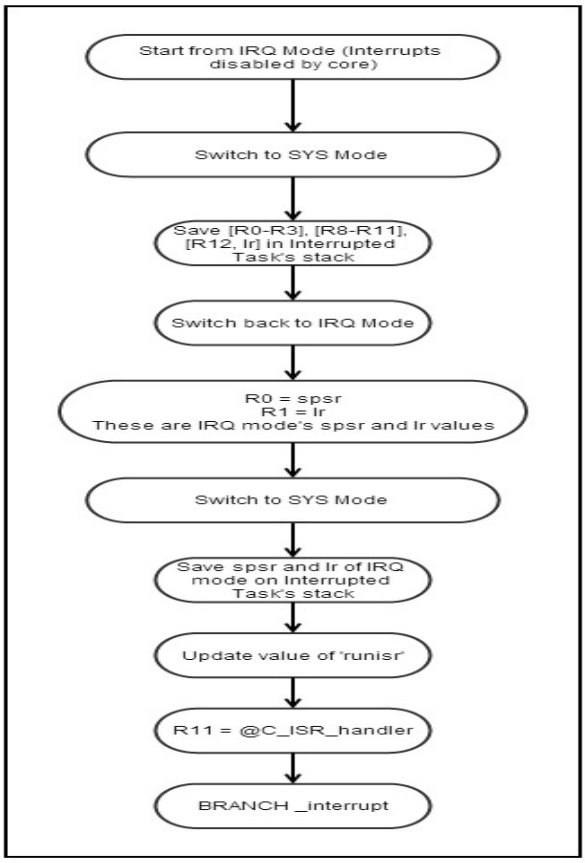


Figure 1: irq_s: Switch between IRQ and SYSTEM Mode

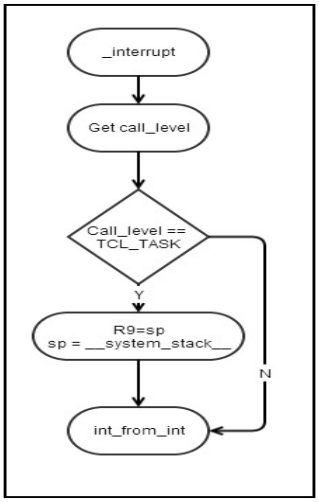


Figure 2: _interrupt

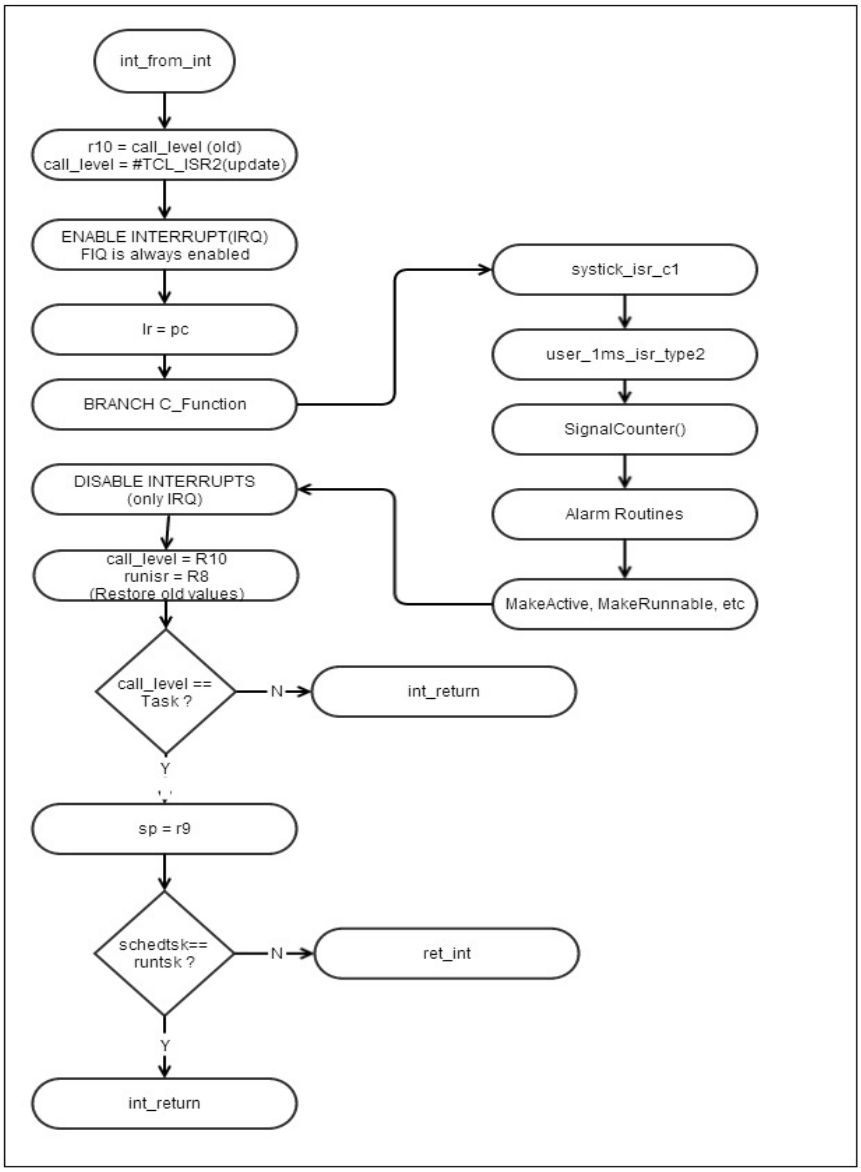


Figure 3: int_from_int

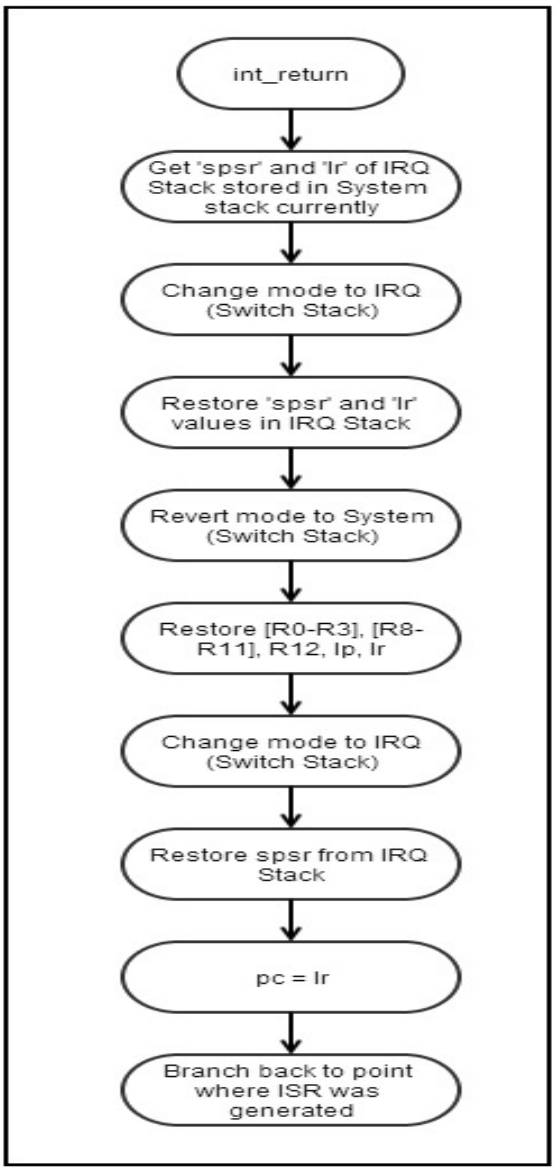


Figure 4: int_return

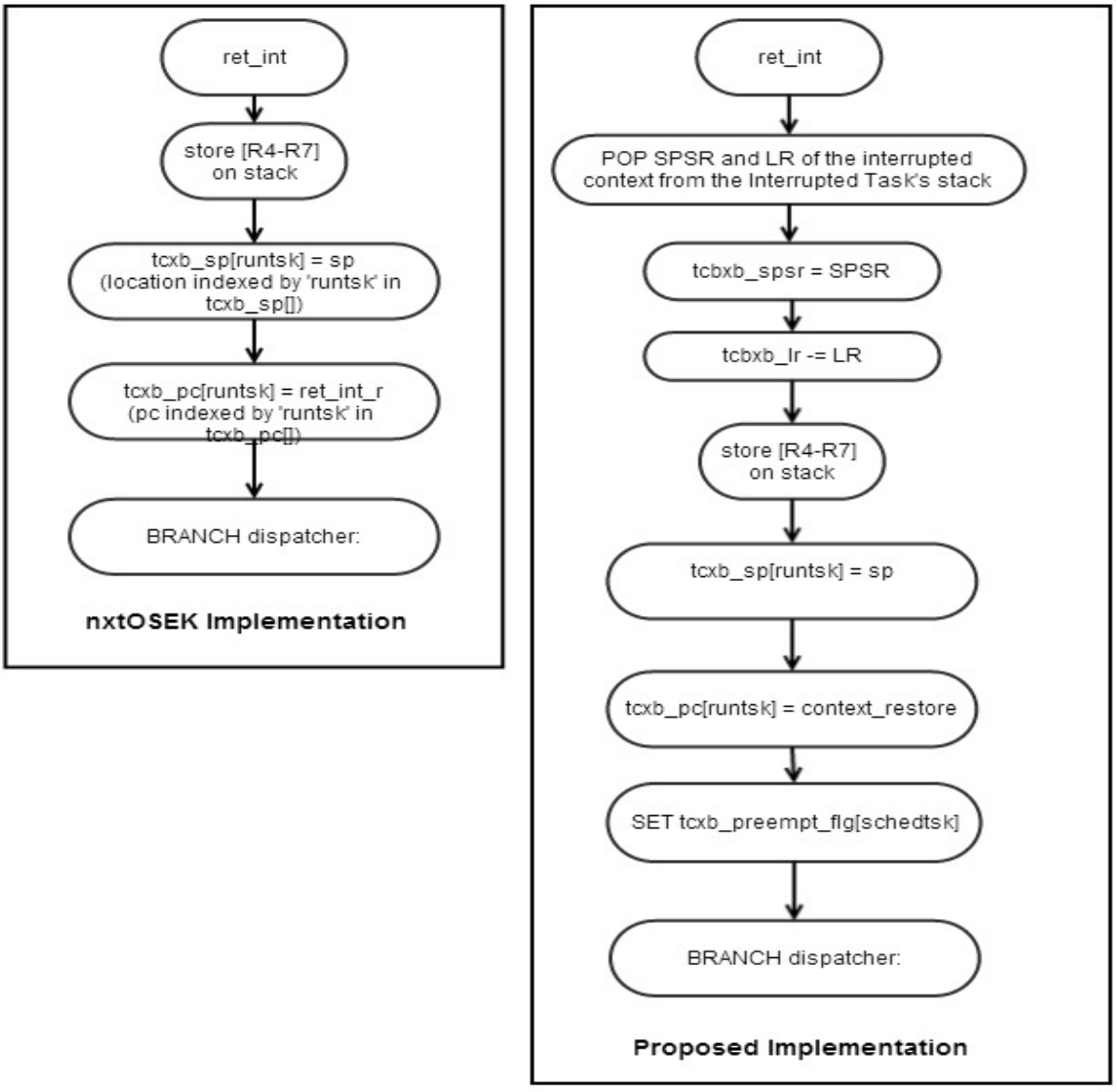


Figure 5: ret_int

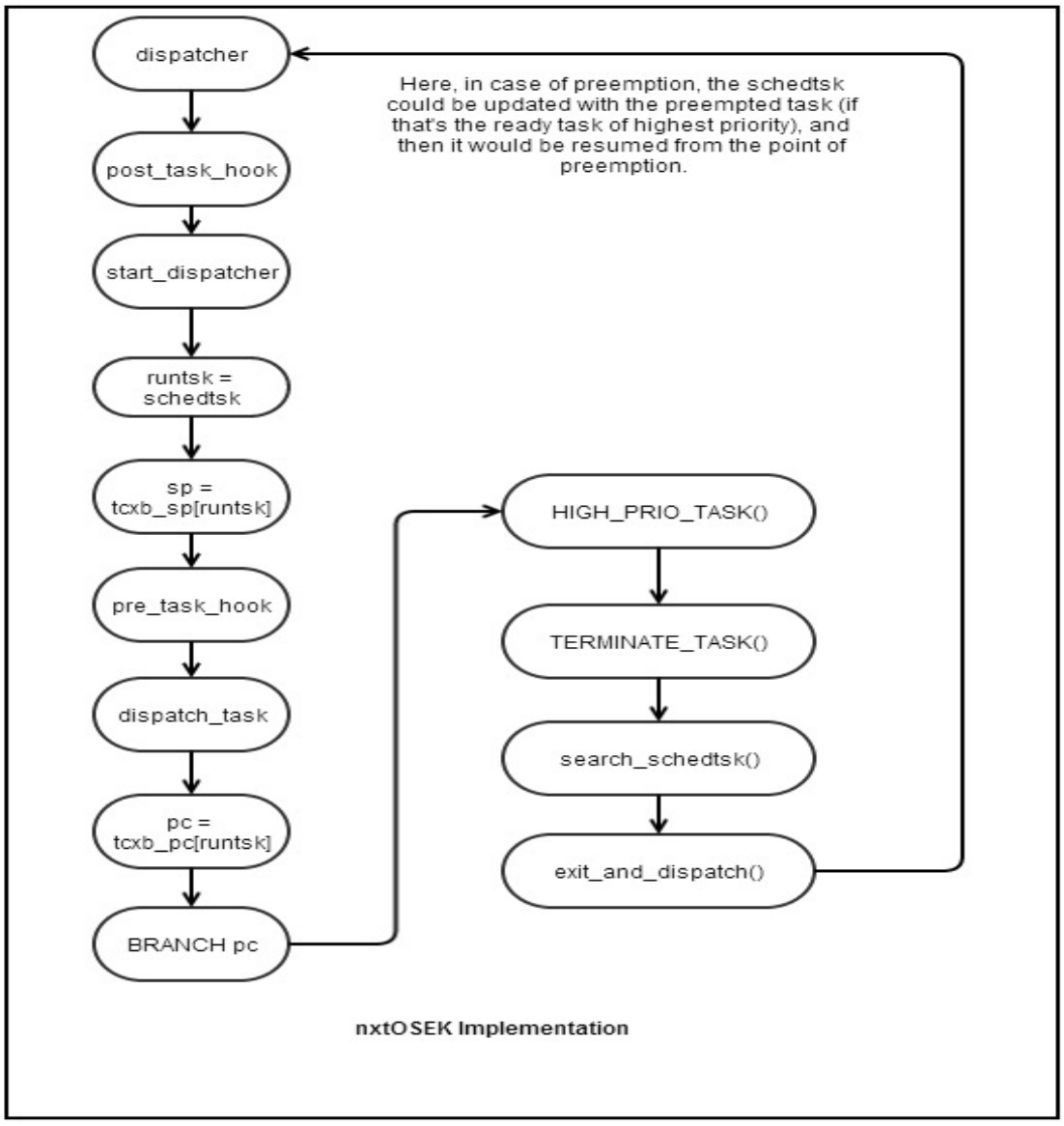


Figure 6: dispatcher nxtOSEK

ure 1, the nxtOSEK kernel follows the standard strategy of saving the spsr_irq and lr_irq on stack memory and enabling interrupts thereafter so that these values can be retrieved in spite of a new interrupt. However, the stack used by nxtOSEK is the interrupted task's stack itself and not the IRQ stack. This is done by manually switching the mode to System mode from the IRQ mode, then saving all the registers for the interrupted context, and thereafter saving the spsr_irq and lr_irq values on the (User) stack. Also, all the subsequent processing is done in System mode (and not IRQ mode).

5. Next, as we can see in Figure 2 and 3, the stack pointer is made to point to _system_stack_ and the C routine for servicing the ISR (user_1ms_isr_type2) is called. This routine in turn invokes the alarm routine SignalCounter() which updates the list of ready tasks based on their periods and also the value of 'schedtsk' to the task id of the highest priority ready task. As seen in Figure 2, once the final value of 'schedtsk' is generated after parsing through all the ready task, the control returns back from the C function to the assembly ISR routine where 'runtsk' and 'schedtsk' are compared. If they turn out to be equal, it's a trivial case of non-preemption as all that needs to be done is restore the saved context, and 'return' from the ISR to the interrupted Task_LP. This is done in the assembly section 'int_return' (Refer to Figure 4)of the code in which after restoring all the context from the task's stack, as also the irq_lr and irq_spsr values, the mode is changed from System mode to IRQ mode. The program control is actually reverted back to the interrupted task by directly modifying program counter's (pc) value to irq_lr.

6. In case of preemption, the values of 'schedtsk' and 'runtsk' won't match(Refer to Figure 5) and as such the program flow jumps to 'ret_int' instead of 'int_return'. We propose a change in the implementation of the 'ret_int' block of assembly code to update the newly introduced data structures tcxb_lr and tcxb_spsr which shall hold the LR and SPSR values respectively of the interrupted task's context (as popped from the task's stack). These shall be used at a later point of time, whenever the Task_LP is rescheduled again, to continue it's execution from the point of preemption. We also a need a new data structure called 'tcxb_preempt_flg[]' to be SET as successive program would depend on whether currently it's an instance of preemption that is being executed. Also note that in our proposed implementation that tcxb_pc[runtsk] is no longer set to ret_int_r as it is no longer required to return the interrupt after finishing the higher priority task. Rather,in our proposed implementation, the ISR shall be 'returned' as a part of preempting to the higher priority task itself.

7. We also need to modify the 'dispatcher' routine as in case of preemption, in our implementation(Refer Figure 6 and 7), we want the ISR to 'return' to the higher priority task. This is in contrast to the current nxtOSEK implementation where the preempting task is run while still the ISR has not yet returned, which was the main reason why nested preemptions were not possible with that approach.

8. Finally, whenever the preempted low priority task is rescheduled again, it's pc shall be picked from the data structure(Refer to Figure 8) tcxb_pc[] which was set to 'context_restore' before preempting. Here, we also make use of the SWI instruction to invoke Supervisor mode as we need access to a separate stack (apart from the User Mode) to restore all the context registers and also branch out of the low priority task.

9. When the SWI instruction is executed, the control jumps to the 'swi_handler' ISR . Here, depending upon the argument passed with the SWI instruction, tcxb_lr[] and tcxb_pc[] values are either pushed or popped from the stack, to resume the execution of the preempted task.

10. The SWI instruction thus helps to achieve the dual objectives of restoring the previously interrupted task's context and yet use stack based operations to jump to that task. It's nor possible to do so without an access to alternate stack and stack pointer, which is essentially what SWI ISR provides us with.

5. VALIDATION

In Figure 9, the following test cases are discussed:

I. First is the case of preemption with two tasks where the low priority task(red) is preempted by the high priority(yellow) task at instant 5.

II. Second case depicts nested preemption where the medium priority(yellow) task preempts the low priority task(red) at instant 10 and is preempted by the high priority task(green) at instant 15.

6. RESULTS AND OPEN POINTS

I. The scheduler is no longer suppressed during the execution of high priority task. This is because the ISR is completed before the high priority task begins execution. Hence, even when the high priority task is executing the scheduler executes periodically at 1ms.

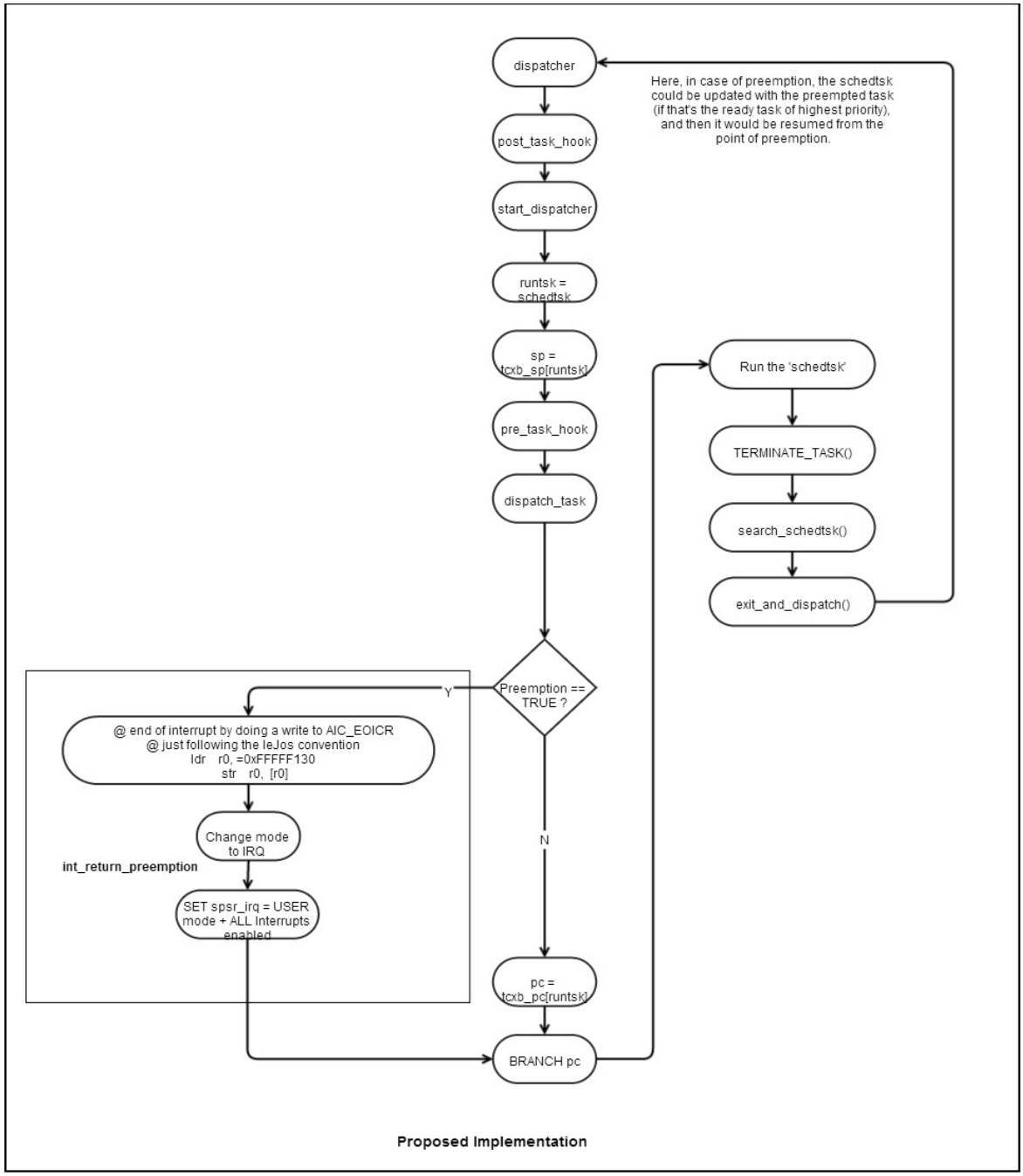
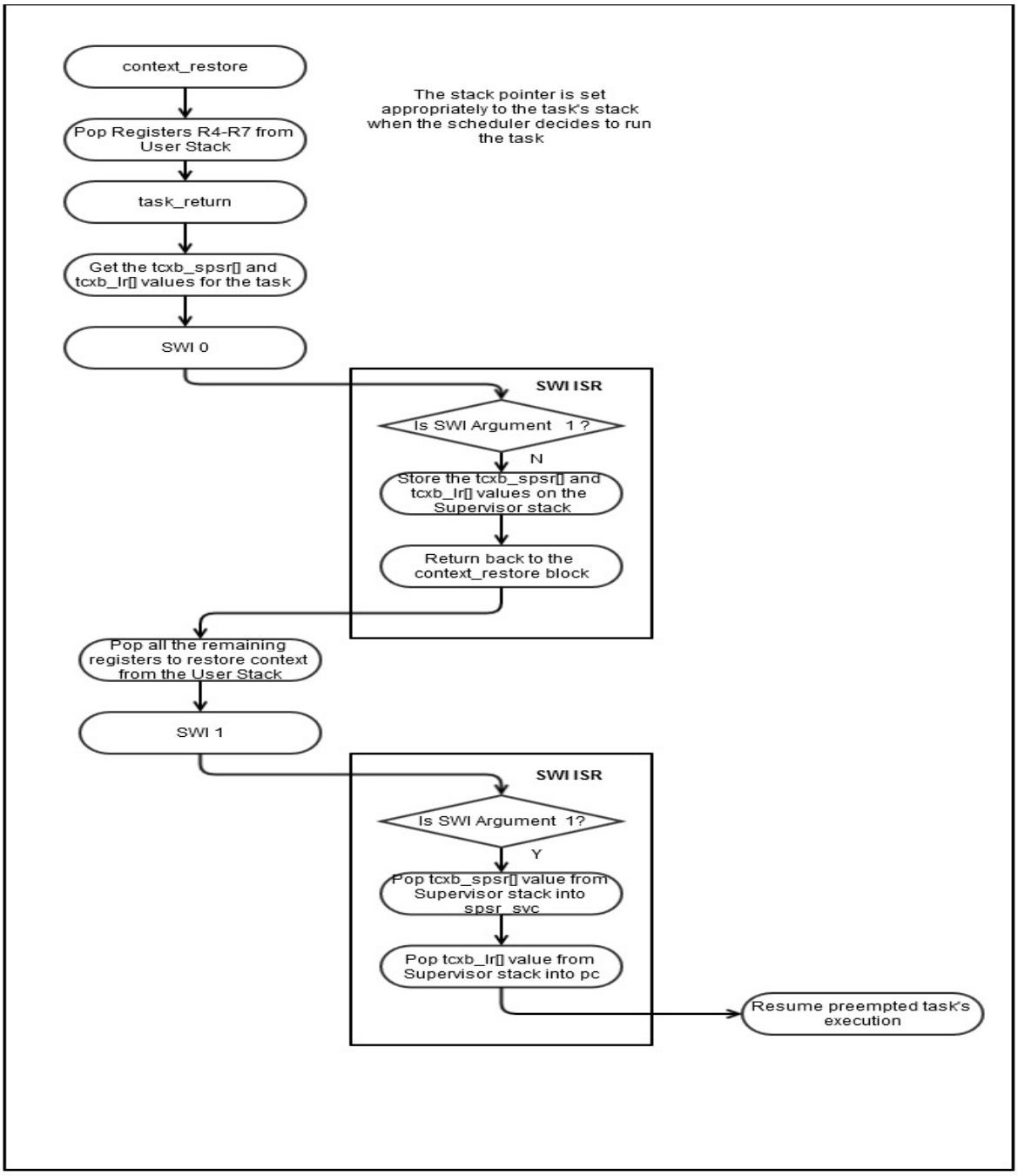


Figure 7: dispatcher Suggested


context_restore
Pop Registers R4-R7 from User Stack
task_return
Get the tcxb_spsr[] and tcxb_lr[] values for the task
SWI 0
The stack pointer is set appropriately to the task's stack when the scheduler decides to run the task
SWI ISR
Is SWI Argument 1 ?
N
Store the tcxb_spsr[] and tcxb_lr[] values on the Supervisor stack
Return back to the context_restore block
Pop all the remaining registers to restore context from the User Stack
SWI 1
SWI ISR
Is SWI Argument 1?
Y
Pop tcxb_spsr[] value from Supervisor stack into spsr_svc
Pop tcxb_lr[] value from Supervisor stack into pc
Resume preempted task's execution

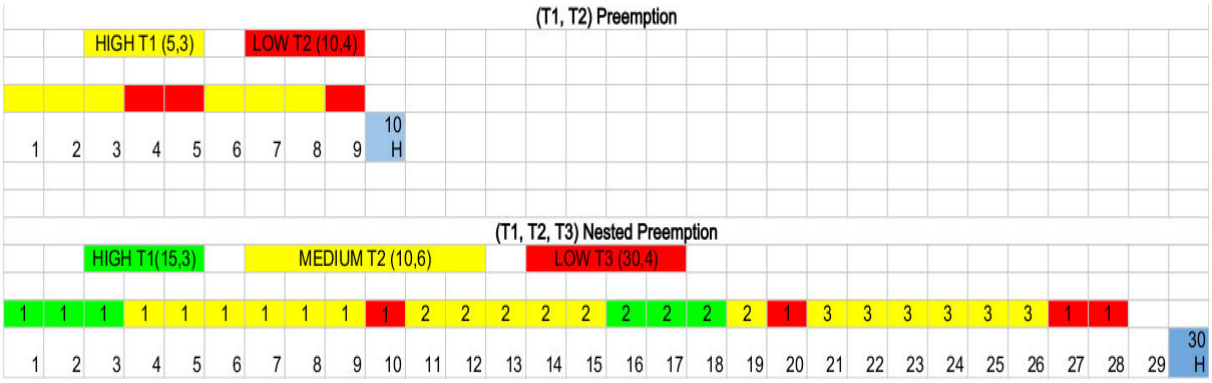


Figure 9: Test Case

II. We weren't able to demonstrate successful return to the preempted task after the high priority task finishes its execution. Instead, we observed the controller getting reset after the completion of the higher priority task. We couldn't resolve this due to limited debug/simulation capabilities.

7. ACKNOWLEDGMENT

We would like to sincerely thank Dr. Frank Mueller for initiating the idea and continuously discussing our design. The suggestions provided by Dr. Mueller turned out to be fruitful in understanding the issue and designing its solution.

8. REFERENCES

[1] Andrew N. Sloss, Dominic Symes, Chris Wright and John Rayfield, ARM System DeveloperâĂŹs Guide.

[2] SWI Instruction Handling. *http://www.heyrick.co.uk/assembler/swi.html.*

[3] SWI Instruction Handling. *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Cacdfeci.html.*

[4] ARM Assembly. *http://www.coranac.com/tonc/text/asm.htm.*

[5] ARM Interrupt Exception Handling. *http://www.iti.uni-stuttgart.de/ radetzki/Seminar06/08*$_r$*eport.pdf*.

[6] The nxtOSEK. *http://lejos-osek.sourceforge.net/*.

[7] The OSEK specs. *http://portal.osek-vdx.org/files/pdf/specs/*.