International Journal of Smart Home
Vol. 9, No. 2 (2015), pp. 111-122
http://dx.doi.org/10.14257/ijsh.2015.9.2.10

Evaluations of Hardware and Software-Based Context Switching Methods in Cortex-M3 for Embedded Applications

Hayeon Choi and Sangsoo Park

Dept. of Computer Science and Engineering
Ewha Womans University
Seoul, Republic of Korea
hayeon.choi@ewhain.net, sangsoo.park@ewha.ac.kr

Abstract

In contrast to the previous ARM microprocessor, the ARM Cortex-M3 processor provides a method for accelerating context switching, which is supported by dedicated hardware logic vis a software interrupt (or trap). In general, it is expected that retaining the context of a task using hardware will reduce the context switching time, but it is also known that software interrupts or traps incur their own overheads. In this study, we propose an algorithm for analyzing the performance of context switching methods in uC/OS-II the Cortex-M3. According to our experimental analysis, we obtained the same results using the algorithm in an ideal state and in a real application. We expect that the algorithms and experimental results described in this study may help embedded system designers by providing quantitative measures in the context switching time of Cortex-M3 using a real-world application.

Keywords: *Context Switching, Cortex-M3, Embedded System, PendSV, Software Trap, uC/OS-II*

1. Introduction

In many embedded systems, real-time operation is required to handle time-sensitive tasks. In these systems, the scheduling of complete tasks must satisfy timing constraints, and it is better to reduce the scheduling overheads of the context switching time in real-time systems [1, 2]. Furthermore, in many embedded applications, context switching time is one of the most important factors that affect the quality of target applications [3, 4].

Unlike existing microprocessors, the ARM Cortex-M3 processor achieves a relatively short context switching time by performing a partial context switching in hardware [5]. However, this function must use a software interrupt or trap, which incurs overheads [6, 7].

The Cortex-M3 processor produced by ARM is a 32-bit embedded microprocessor for deterministic real-time applications in time-sensitive systems, such as control systems and networking systems. To facilitate better multi-tasking performance, hardware support for task-switching interrupts, i.e., PendSV, is provided in the Cortex-M3 microprocessor. PendSV is a software trap that can be initiated by assembly code, which can manually force context switching, in the Cortex-M3 [8, 9].

Context switching requires the storage of processor registers from one task, which is suspended, before restoring the registrations from another task that needs to be executed. In traditional ARM microprocessors, context switching was performed solely by software. By contrast, the software-trap handler issued by PendSV allows the Cortex-M3 to save half of the processor registers automatically and it then restores the registers when exiting the handler to achieve context switching between two tasks. Thus, it is expected that the context switching method based on PendSV should minimize the context switching time.

ISSN: 1975-4094 IJSH
Copyright ⓒ 2015 SERSC

However, it is well known that an interrupt or a software trap does not occur immediately and it inevitably incurs its own timing delay. In our previous study [10], we performed a quantitative evaluation of the context switching time with different mechanisms, which showed that the time required for the PendSV-based metchod was approximately 1.5% longer than that for the software-based method. Our experimental results showed that the delay incurred by the hardware-support for PendSV was slightly higher than the reduction in the time required for saving and restoring registers.

In our previous study [10], the experimental environment was ideal because a task only switched to another task and only the PendSV software trap was allowed. However, real applications require mixed interrupts and software traps, as well as application code. To address this limitation, we then applied context switching methods to one of the most popular embedded applications: VoIP. Our evaluation obtained consistent experimental results, which showed that the software-based context switching method obtained better performance than the PendSV-based mechanism [11].

In the present study, we describe an algorithm for Context Switching Methods in uC/OS-II, and we support the design of embedded systems based on a quantitative evaluation of the existing microprocessor by measuring the context switching time in the Cortex-M3.

The remainder of this paper is organized as follows. Section 2 compares software-and PendSV-based context switching methods. Section 3 describes the algorithm for enforcing the context switching algorithm in uC/OS-II. Section 4 presents a case study and the results of our performance evaluation. Finally, we give our conclusions in Section 5.

2. Comparison of SW- and HW-based Context Switching Methods

In the following, we refer to software-based context switching as the SW-based context switching and PendSV-based context switching is referred to as HW-based context switching.

2.1. SW-based Context Switching Method

General embedded microprocessors perform context switching by following a procedure [12]. First, all registers are stored in a stack using software. Second, the next task to be performed is selected and the context is changed for the task, including the stack data structure. Finally, all register are loaded in the stack operation by software code [10].

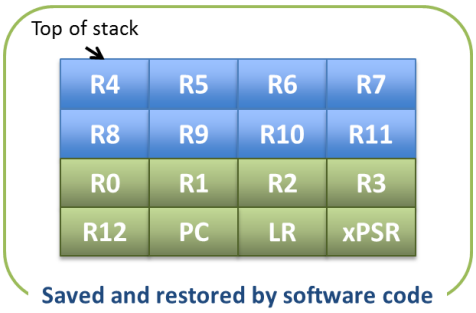

Figure 1. Register Set Saved and Restored by Software Code

2.2. HW-based Context Switching Method

The stack comprises two registers in the CortexM3: the process stack used by the task and the main stack used by the operating system. The context that needs to be saved and restored for task switching in the Cortex-M3 microprocessor includes

 Copyright ⓒ 2015 SERSC

general purpose registers, which range from R0 to R12, program counter (PC), a link register (LR), and a control and status register (xPSR).

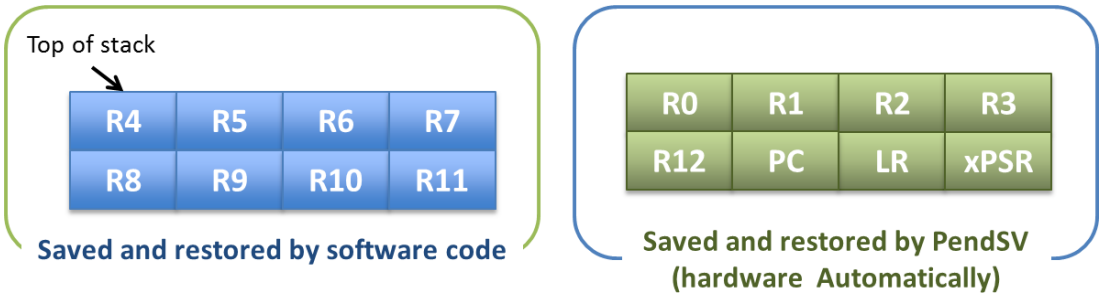

Figure 2. Register Sets Saved and Restored by Software Code and by the Hardware Logic Issued by the PendSV Software Trap

As shown in Figs 1 and 2, the number of registers in the pure SW-based mechanism is twice that in the HW-based mechanism, but the architecture of the pure SW-based method is much more straightforward compared with PendSV [11, 13].

Half of the entire register is stored and recalled automatically by the hardware, but software interrupt-based context switching must be used in PendSV to drive the Cortex-M3.

As mentioned earlier, the ARM Cortex-m3 processor achieves a relatively short context switching time by performing partial context switching in hardware according to the following procedure. First, it generates a PendSV. Second, the registers are stored in the stack automatically, except for those between R4 and R11, although these exceptions are still saved in the stack in the same manner. Third, the next task is selected for performance and context is changed for the task, including the stack data structure. Fourth, after loading the task in R4 to R11 using software code, the PendSV is returned. Finally, the registers are loaded except, from R4 to R11, in the stack operation where this task is performed in hardware [10].

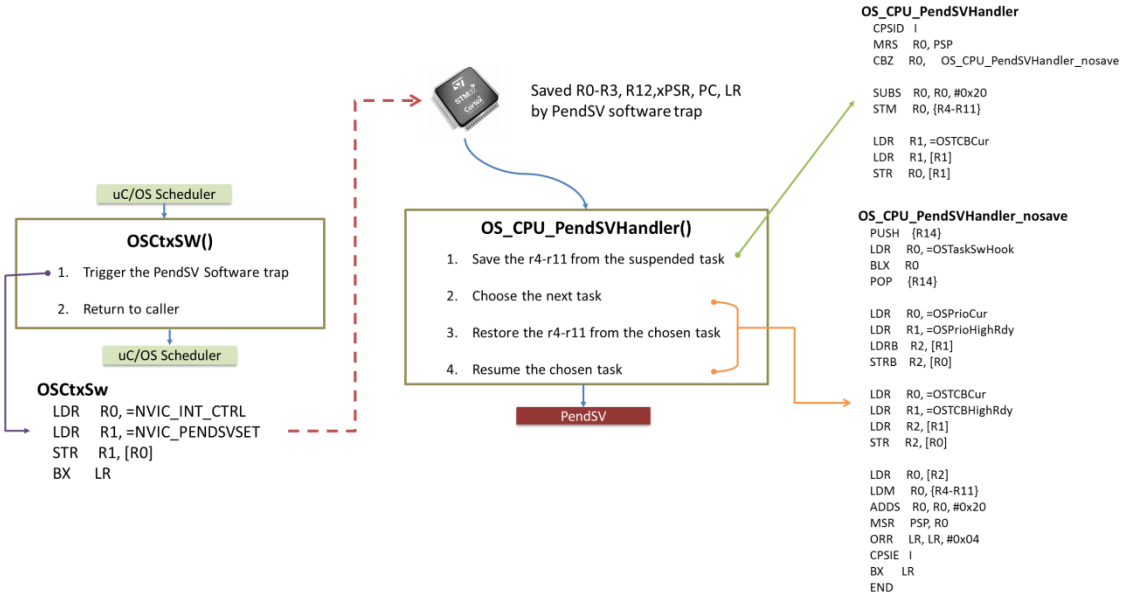

Figure 3. Architecture of the HW-based Context Switching Method

Copyright ⓒ 2015 SERSC

Figure 3 shows that it is easy to use the assembly code for these procedures, as follows. First, Get the process SP and save the remaining registers R4 - R11 in the process stack. Next, save the process SP in its TCB, OSTCBCur → OSTCBStkPtr = SP, and call OSTaskSwHook(). Second, get the current ready thread TCB, OSTCBCur = OSTCBHighRdy, and get the new process SP from TCB, SP = OSTCBHighRdy → OSTCBStkPtr. Third, restore R4 - R11 from the new process stack and perform an exception return, which will restore the context. In Cortex-M3, half of the registers are saved and restored automatically by hardware in the context switching method based on PendSV.

2.3. Implementation of SW-based Context Switching Methods and Measuring the Context Switching Time

Cortex-M3 has not been implemented in existing SW-based context switching methods. Therefore, using ARM's White Paper [14] and Board Support Package, we implemented the code for the SW-based mechanism [10], which is shown in Fig. 4.

As shown in Fig. 4, the SW-based context switching methods in the Cortex-M3 can be used to write the code, which saves/restores via software and the code is registered to save/restore via the existing hardware. When the call is OSCtxSw(), context switching is performed in the same manner as HW-based context switching methods. However, the STMFD command is used instead of hardware when saving a register. All registers are restored at the end of context switching. The LDMFD command is used instead of hardware when restoring a register.

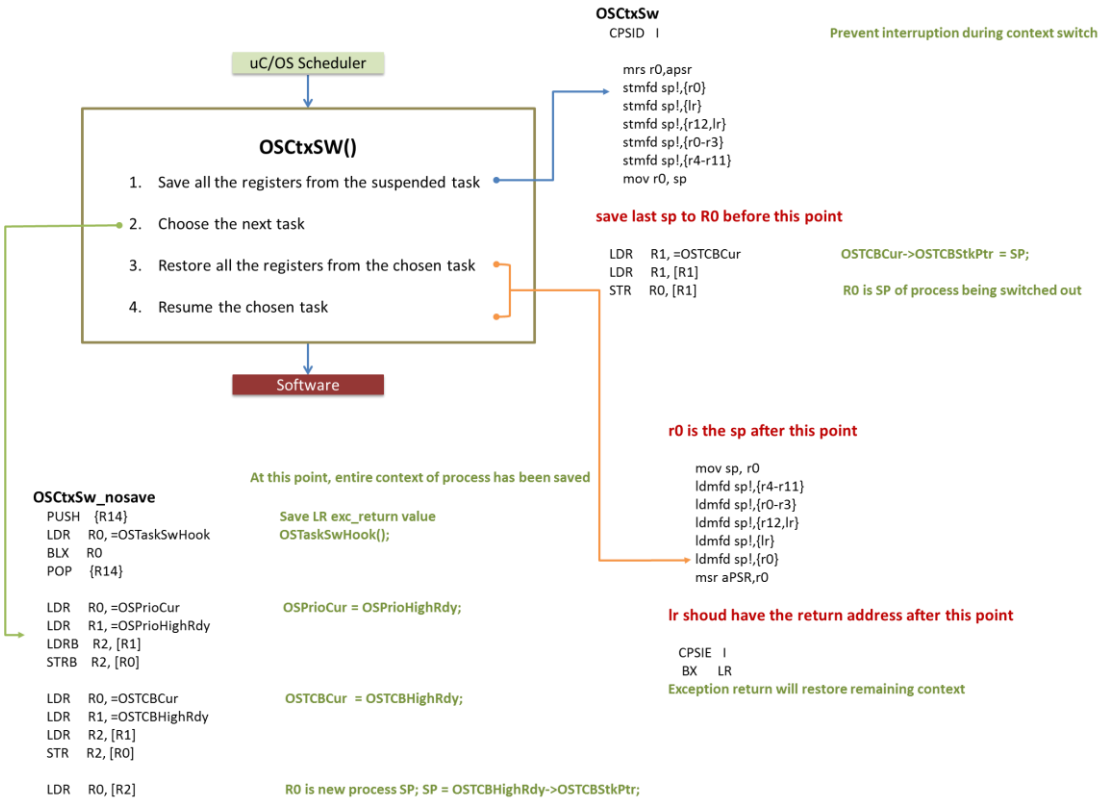

Figure 4. Architecture for SW-Based Context Switching Methods

Different methods can be used to measure the context switching time. In our previous study, we simply measured the time required to execute the OSTaskChangePrio() API, but to measure the time more accurately and precisely, it is necessary define the beginning

Copyright ⓒ 2015 SERSC

and the end of the context exchange via OSTaskSwHook(), which is called always during context switching.

After considering these points, the startup time was measured before the OSCtxSw () call and the end time was defined as the time when OSTaskSwHook() finished running. However, the time cannot be measured simply by changing the start and end of a context switch. Thus, we also needed to consider how we obtain the context switching time because the context exchange is executed many times repeatedly, and the condition that the end time is greater than the start time is not always satisfied. Therefore, we developed the method shown in Fig. 5.

Initialize $in\_time = Start\ of\ Context\ Switching$
$out\_time = End\ of\ Context\ Switching$
$context\_switching\_time = Context\ Switching\ Time$
$MAX = The\ maximum\ value\ of\ the\ time\ variable\ to\ measure$
$MIN = The\ minimum\ value\ of\ the\ time\ variable\ to\ measure$

if $out\_time > in\_time$ **then**
$context\_switching\_time \leftarrow out\_time - in\_time$
else
$context\_switching\_time \leftarrow (MAX - in\_time) + (out\_time - MIN)$
endif
return $context\_switching\_time$

Figure 5. Measuring the Context Switching Time

3. Algorithm for Enforcing Context Switching (Task Yielding) in uC/OS-II

uC/OS-II is a real-time priority-based operating system where the priority of each specific task does not have the yield function provided by time-sharing scheduling-based operating systems such as Linux. Thus, in order to determine the context switching time, it is necessary to schedule the forcing between any two tasks [10].

uC/OS-II provides an OSTaskChangePrio() API that allows the priority of any task to be changed run-time [15].

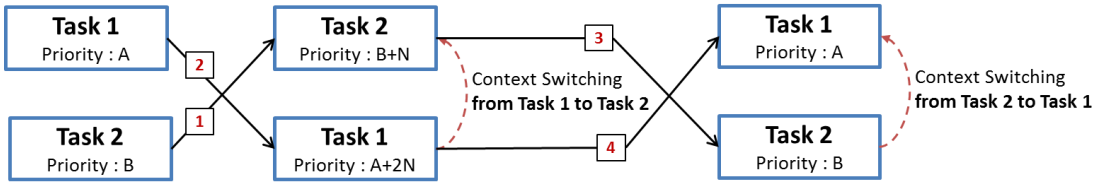

Figure 6. Flowchart Showing Context Switching between Two Tasks in a Priority-based Scheduler

Figure 6 shows the flowchart for a task-yielding algorithm with two tasks on a priority-based scheduler. Task 1 has the highest priority, which is designated as A, and Task 2 has the lowest priority, denoted as B. N is the number of tasks, i.e., two in Fig. 6. The numbers specified in the small boxes indicate the priority in the procedure.

The order of priority setting is important when measuring the context exchange time between any two tasks. From step 1 to step 2, the change in the priority of Task 1 is A+2N, and the remaining priority of Task 2 is B immediately after performing step 2. Therefore, step 1 cannot be performed from Task 1 to Task 2 immediately after step 2 (with priority-based scheduling). However, the current priority is lower and the priority of Task 2 does not change the state of Task 1, which has higher priority [10].

Copyright ⓒ 2015 SERSC

The software code created based on a consideration of these points is shown in Fig. 7.

```
Void Task1(void *pdata){
        while(1) {
                … //Context Switching Task2
                        OSTaskChangePrio(B, B+N);
                        OSTaskChangePrio(A, A+2N);
                }
}

Void Task1(void *pdata){
        while(1) {
                … //Context Switching Task1
                        OSTaskChangePrio(B+N, B);
                        OSTaskChangePrio(A+2N, A);
                }
}
```

Figure 7. Software Code for Enforced Context Switching between Two Tasks

3.1. Enforcing Context Switching Among N Tasks

uC/OS-II can manage 56 independent tasks [15]. In our previous study, we proposed a mechanism for task yielding between two tasks in the scheduler based on priorities [10, 11]. In the present study, we extended the scheduling mechanism to support an arbitrary number of tasks, where we developed a task-yielding algorithm to enforce context switching among N tasks in uC/OS-II. The assumptions of the algorithm are as follows.

· Task 1 has the highest priority, called as A.

· Task N has the lowest priority, called as B.

· OSTaskChangePrio() is called $2N(N-1)$ times during context switching.

(Regardless of the number of Tasks)

· Task 1 called the OSTaskChanePrio() N times.

· Other Task called the OSTaskChangePrio() 2N times.

When context switching is called for OSTaskChangePrio(), the order is the same as that shown in Fig. 8, where N is the number of tasks and the numbers specified in the small rectangular boxes indicate the priority assignments.

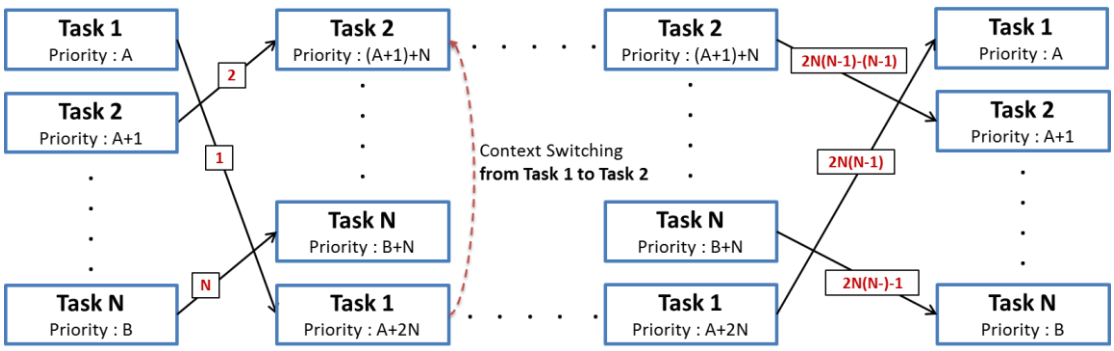

Figure 8. Flowchart Showing Task Yielding in a Priority-based Scheduler

Copyright ⓒ 2015 SERSC

The flowchart for enforced context switching is similar to the flowchart for scheduling two tasks in a priority-based scheduler. However, as mentioned earlier, the order of the priority assignments is important and the algorithm for context-switching methods in uC/OS-II (Fig. 9) is based on Fig. 8.

```
Initialize   i = Number of times called OSTaskChangePrio()
             N = The number of Maximum task
             prio = Priority of initial

for i ← 1 to i ← 2 × N step 1 do
    if i ≤ N × (N − 1) then
       prio ← prio + N
    else
       prio ← prio − N
    endif
endfor
```

Figure 9. Algorithm for Task N

```
Initialize   i = Number of task
             j = Number of times called OSTaskChangePrio()
                 grobal variable
             N = The number of Maximum task
             prio = Priority of initial

calcPrio(i, j)
    for a ← 1 to a ← N × (N − 1) step 1 do
        if j ≤ i × N then
           prio ← prio + 2 × N
        else
           prio ← prio + N
        endif
    endfor

    for b ← N × (N − 1) + 1 to b ← 2 × N × (N − 1) step 1 do
        if j ≤ 2 × N × (N − 1) then
           prio ← prio − 2 × N
        else
           prio ← prio − N
        endif
    endfor
  return prio
```

Figure 10. Algorithm for Other Tasks

The task-yielding algorithms are shown in Figs 9 and 10. A priority-based scheduler is used a variable number of times, which employs the algorithm because it is important to specify the correct sequence of calls, i.e., OSTaskChangePrio(). The number of times the function is called is the same as the order in which they are called, and the order of execution is the same as that shown in Fig. 6, where the number of tasks is specified simply by increasing the number N to two. When context switching is performed up to Task N, tasks other than Task N have rules that are applied after 2N has been added, and N is then added N-2 times. Context switching is then performed again up to Task 1 after performing Task N, and tasks other than Task N have rules that are applied when 2N is subtracted, after which N is then subtracted N-2 times. We can deduce the following equations (1) from these rules [11].

Copyright ⓒ 2015 SERSC

$$Priority\ existing + 2N + N(N-2) = Priority\ when\ Context\ Switching\ up\ to\ Task\ N \quad (1)$$

3.3. Maximum Number of Tasks Allowed by the Algorithm

According to Section 3.2, the context-switching methods in uC/OS-II, and the characteristics of uC/OS-II, the available priority range is 55. Therefore, the maximum number of tasks can be obtained using Equation (2).

$$\begin{aligned} (N-1) + 2N + N(N-2) &= B - A \\ N\text{^}2 + N - 56 &= 0 \\ (N+8)(N-7) &= 0 \\ \therefore N &= 7, 8 \end{aligned} \quad (2)$$

4. Performance Evaluation

We ported uC/OS-II to the Cortex-M3 evaluation board, MCBSTM32EXL from Keil [16]. We then performed experiments using the algorithm developed in the present study. We increased the number of context switches from 100 to 10000 and we measured the context-switching times. Figure 11 shows the experimental results obtained using the SW- and HW-based methods, which demonstrates that the SW-based method obtained better results than the HW-based method, where the context-switching time and the time difference increased with the number of context switches.

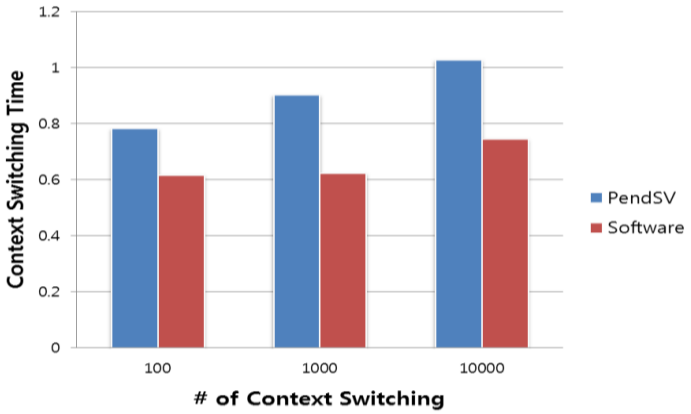

Figure 11. Experimental Results Obtained Relative to the Number of Context Switches

We measured the context-switching time until the maximum number of tasks issued calls. Figure 12 shows that the SW-based method obtained better results than the HW-based method, where the context-switching time and the time difference increased with the number of tasks.

Copyright ⓒ 2015 SERSC

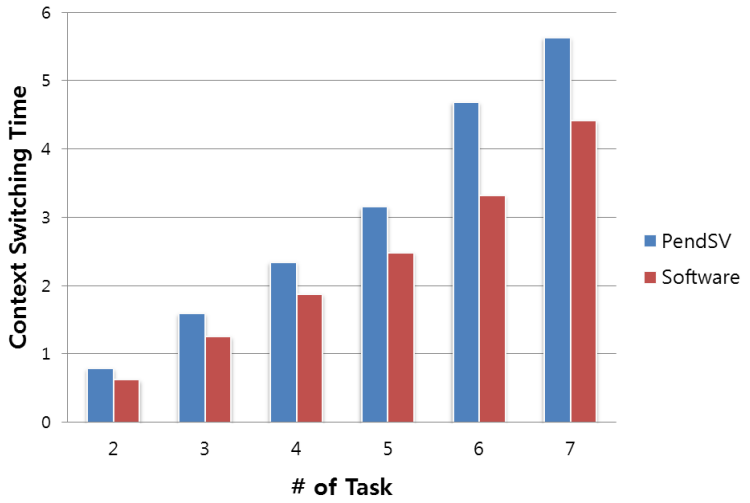

Figure 12. Experimental Results Relative to the Number of Tasks

The experiments reported in the previous section were performed in ideal conditions because one task performed one function before switching to another task and only the PendSV software trap was allowed. However, real applications require mixed interrupts and software traps, as well as application codes. To address this limitation, we applied the context-switching methods to one of the most popular embedded applications: VoIP. We modified the software code for the context switching functions, i.e., OSCtxSW() and OS _CPU_PendSVHandler(), in the source file, os_cpu_a.s, to support both context-switching mechanisms. An embedded system for VoIP application requires WiFi and an audio device for playing and recording voice communication. However, we used a timer interrupt to emulate receiving VoIP packets from a communication device because the evaluation board lacked these devices. Other features, such as sending VoIP packets and playing and recording voice communications, were replaced by memory I/Os because these are often implemented as programmed or polling I/Os.

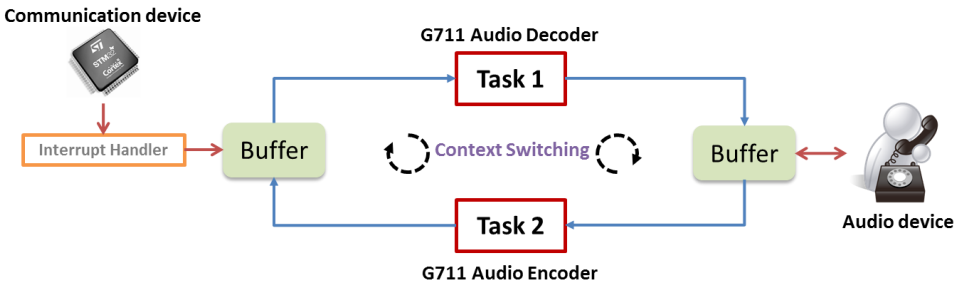

Figure 13. Embedded Software for the VoIP Application

Figure 13 shows an overview of the embedded software for VoIP [17, 18] used in the present study. The timer interrupt was set as periodic to fill the buffer for the G711 audio decoder [19]. To determine a realistic time frame for the interrupt period, we used the calculation described in [20]. In the experiment, raw audio data were encoded based on the G711 audio codec at 64 kbps with a 20-ms sampling period and the communication device required 95.2 kbps bandwidth for each VoIP connection. The increased bandwidth was attributable to the packet payload in the communication layers. In the application, one task ran on the audio decoder and the other task ran on the audio encoder to support two-way conversations. Task 1 was pending on the semaphore, where the timer interrupt handled posts when it was invoked [11].

Copyright ⓒ 2015 SERSC

To evaluate the performance, we designed the experiments to increase the number of VoIP connections up to a limit of 1000, where the microprocessor handled the decoder and encoder during the next invocation of the timer interrupt. In addition, application tasks were created to support concurrent VoIP clients up to 27 connections. Our experimental results are shown in Fig. 14, where the y-axis values obtained using the PendSV software based on the VoIP test context-switching times were longer than those obtained in the case study. According to our experimental results, the SW-based mechanism performed consistently better than the HW-based mechanism and the performance gain increased slightly with the numbers of connections.

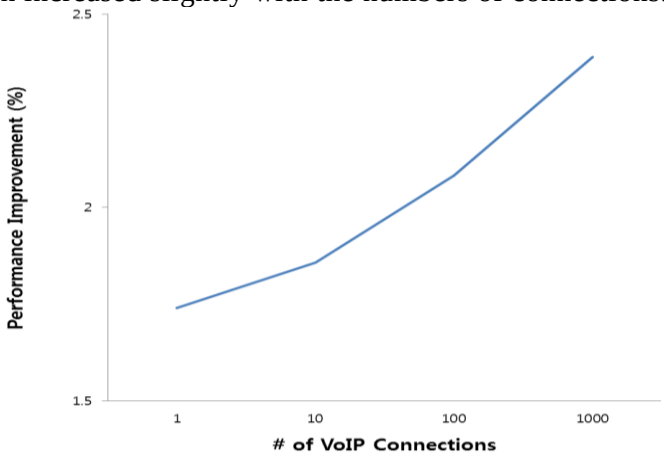

Figure 14. Experimental Results Obtained Using the VoIP Application

5. Conclusions

In contrast to existing microprocessors, the ARM Cortex-M3 processor achieves a relatively short context-switching time by using a software interrupt (Trap) in hardware for context switching. However, software interrupts (traps) are known to incur their own overheads. Thus, we developed an algorithm for context switching method in uC/OS-II. We also used the algorithm to analyze the time required for context switching with different numbers of tasks. Our experiments showed that the same results were obtained using the algorithm in an ideal state and in a real application. In future research, we will test whether the same results can be obtained with different repetitions of context switching.

Acknowledgement

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education, Science, and Technology (2011-0013422). Sangsoo Park is the corresponding author.

References

[1] J. W. S. Lio, “Real-Time Systems”, Prentice Hall, **(2000)**.
[2] N. I. Rafla and D. Gauba, “Hardware implementation of context switching for hard real-time operating systems”, IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS), **(2011)**.
[3] A. Paul and A. Pillai, “Reducing the number of context switches in real time systems,” in Process Automation, International Conference on Control and Computing (PACC), **(2011)**.
[4] P. R. Nuth and W. J. Dally, “A mechanism for efficient context switching”, ICCD, Proceedings, IEEE International Conference on Computer Design: VLSI in Computers and Processors, **(1991)**. [5] B. Nagel, “Advantages of the Cortex-M3”, Micrium, **(2008)**.
[6] K. K. Sandstrom, “Handling interrupts with static scheduling in an automotive vehicle control system”, Proc. of RTCSA, **(2008)**.

 Copyright ⓒ 2015 SERSC

[7] G. Ugurel and C. F. Bazlamacci, "Context switching time and memory footprint comparison of Xilkernel and μC/OS-II on MicroBlaze", 7th International Conference on Electrical and Electronics Engineering (ELECO), **(2011)**.
[8] T. Gilbert, "Make the most out of cortex-M3's preemptive context switches", EET India, **(2012)**.
[9] J. Yiu and A. Frame, "ARM Cortex-M3 Processor Software Development for ARM7TDMI Processor Programmers", **(2009)**.
[10] H. Choi and S. Park, "A quantitative evaluation of SW/HW-based context switch time for ARM cortex-M3", Proceedings of the Fall Conference of the KIPS, **(2013)**.
[11] H. Choi and S. Park, "Performance evaluation of context switching methods for VoIP applications in Cortex-M3", Proceedings Conference of the MITA, **(2014)**.
[12] J. Mische, S. Uhrig, F. Kluge and T. Ungerer, "Using SMT to hide context switch times of large real-time tasksets", IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), **(2010)**.
[13] Z. Qian, C. Zhiyu and S. Yibin, "Transplant method and research of muC/OS_II on ARM Cortex-M3," ICICTA: Proceedings of the Fourth International Conference on Intelligent Computation Technology and Automation, **(2011)**.
[14] S. Sadasican, "An Introduction to the ARM Cortex-M3 Processor", **(2006)**.
[15] J. J. Labrosse, "MicroC OS II: The Real Time Kernel", CMP Books, **(2002)**.
[16] Keil, "Mcbstm32exl arm cortex-m3 evaluation board (http://www.keil.com/)".
[17] W. Zhai and J. Wang, "An application of VoIP communication on embedded system", International Conference on Computer Application and System Modeling (ICCASM), **(2010)**.
[18] A. Faroudja, N. Izeboudjen, S. Titri, L. Sahli, F. Louiz and D. Lazib, "Hardware/Software development of a System on Chip platform for VoIP application", International Conference on Microelectronics (ICM), **(2009)**.
[19] N. Harada, Y. Kamamoto and T. Moriya, "Lossless Compression of Mapped Domain Linear Prediction Residual for ITU-T Recommendation G.711.0", Data Compression Conference (DCC), **(2010)**.
[20] Newport Networks Ltd, "VoIP bandwidth calculation".

Authors

Hayeon Choi. She received her BS degree from Ewha Womans University, Seoul, Korea, in 2013. Currently, she is an MS candidate in the Embedded Software Laboratory in the Department of Computer Science at Ewha Womans University.

Sangsoo Park. He received his BS degree from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1998, and his MS and PhD degrees from Seoul National University, Seoul, Korea, in 2000 and 2006, respectively. Currently, he is an assistant professor in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. His research interests include real-time embedded systems and system software.

Copyright ⓒ 2015 SERSC

Copyright ⓒ 2015 SERSC