Efficient Event-Triggered Tasks in an RTOS

Kaj Hänninen[1,2] , John Lundbäck[2], Kurt-Lennart Lundbäck[2], Jukka Mäki-Turja[1], Mikael Nolin[1]
[1]Mälardalen Real-Time Research Centre, Västerås Sweden
[2]Arcticus-Systems, Järfälla Sweden

Abstract - In this paper, we add predictable and resource efficient event-triggered tasks in an RTOS. This is done by introducing an execution model suitable for example control software and component-based software. The execution model, denoted single-shot execution (SSX), can be realized with very simple and resource efficient run-time mechanisms and is highly predictable, hence suitable for use in resource constrained real-time systems.
In an evaluation, we show that significant memory reductions can be obtained by using the SSX model.

Keywords: RTOS, execution model, stack usage.

1. Introduction

When designing software for embedded systems, resource consumption is often a major concern. Software consumes resources primarily in two domains: the time domain (execution time), and the memory domain (e.g., RAM and flash memory). For systems without real-time requirements, the resource consumption in the time domain may be of less importance. However, most embedded systems are either used to control or monitor some physical process, or used interactively by a human operator. In both of these cases, it is often required that the system responds within fixed time limits. Hence, methods for development of embedded systems need to allow design of both memory and time efficient systems.

Moreover, predictable use of the resources are required. Predictions of the amount of resources needed to execute the system are used to dimension the system resources (e.g., selecting CPU and amount of memory).

The current trend in development of embedded systems is towards using high-level design tools with a model-based approach. Models are described in tools like Rational Rose, Rhapsody, Simulink, etc. From these models, whole applications or application templates are generated. However, this system generation seldom considers resource consumption. The resulting systems become overly resource consuming and even worse; they exhibit unpredictable resource consumption at run-time.

In this paper, we describe and evaluate the integration of a resource efficient and predictable execution model, denoted single shot execution model (SSX), in a commercial real-time operating system. The execution model facilitates stack sharing to reduce memory consumption and priority scheduling to allow timing predictions.

The paper is organized as follows. In section 2, we describe the properties of the SSX model and the prerequisites needed to utilize the model. In section 3, we describe our target platform, the Rubus RTOS. In section 4, we describe the integration of SSX in Rubus and in section 5, we evaluate the stack usage under the SSX model in different execution scenarios. In section 6, we conclude the integration of SSX in Rubus.

2. The Single Shot Execution Model (SSX)

Throughout the years, research in real-time scheduling and real-time operating systems has resulted in a vast number of different execution models, e.g.,[3][4][7][9][12], one of them being the single shot execution model in which tasks are considered to terminate at the end of each invocation, i.e., execute to completion (as opposed to indefinitely looping tasks).

Baker [3] and Davis et al. [4] shows that the single shot execution model, with an immediate priority ceiling protocol, enables possibilities for efficient resource usage by stack sharing among several tasks. Stack sharing in the SSX model is feasible because higher priority tasks are allowed to pre-empt lower priority tasks and execute to completion (i.e., terminate) before lower priority tasks are allowed to resume their execution. However, the fact that a task must execute to completion (terminate) before any lower priority task is allowed to execute, puts some restrictions on suspensions of tasks in the SSX model:

- To guarantee correct stack access, self-scheduling of SSX tasks, i.e., calling timed sleep or delay functions may not be used in the application code of SSX tasks.
- Task synchronization should be done using the Immediate Priority Ceiling Protocol (IPCP). This ensures that a task will never be allowed to start executing, before it is guaranteed to have access to all resources it needs. Hence, calls for accessing shared resources, such as semaphores, will never result in blocking due to locked resources. Any possible blocking will occur before the task is allowed to start execute.

However, these design restriction also facilitate predictability since the administration of the tasks is left entirely to the operating system. Moreover, it is known that the immediate priority ceiling protocol is deadlock free and exhibits an upper bound on blocking times for tasks sharing resources. This implies that an analysis technique such as response-time analysis [2] enables analysis of temporal properties of an SSX system.

The SSX model is conceptually very simple, at run-time a task can be in one of three states: terminated, ready, or executing. The main difference, from a developers view, is that a conventional RTOS often uses so called self-scheduled tasks. This means that a task is activated once, typically at system start-up, and eventually, after some possible initialization code, ends up in an infinite loop where it self-schedules itself, e.g., using delay calls.

An SSX task, on the other hand, when activated by the OS, executes with no delay calls, and terminates upon completion. This means that such tasks have to be re-scheduled by the OS in order to provide a continuous service. Figure 1 illustrates the structural difference between a conventional task and an SSX task.

```
taskEntryFunc(){          taskEntryFunc(){
 while(1){
  //Task code               //Task code
  sleep(time)             }
 }
}
```

Figure 1. Looping task (left). Typical SSX task (right)

In this paper, we present an integration of the SSX model in the Rubus RTOS. We also present a quantitative evaluation of stack usage under different execution scenarios.

3. The Rubus Operating System

Rubus is a real-time operating system developed by Arcticus Systems [1]. Rubus is targeted towards systems that typically require handling of both safety critical functions as well as less critical functions. The emphasis of Rubus is placed on satisfying reliability, safety and temporal verification of applications. It can be seen as a hybrid operating system in the sense that it supports both statically and dynamically scheduled tasks. The key features of Rubus RTOS are:

- Guaranteed real-time service for safety critical applications
- Best-effort service for non-safety critical applications
- Support for time- and even-triggered execution of tasks
- Support for component based applications

Rubus consist of three separate kernels (Figure 2). Each kernel supports a specific type of execution model.

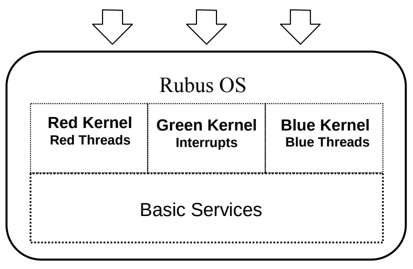

Figure 2. Rubus RTOS architecture

The *Red kernel* supports time driven execution of static scheduled 'Red tasks', mainly to be utilized for applications with fixed hard real-time requirements. The static schedule is created off-line by the Rubus Configuration Compiler. Synchronizations of shared resources are handled by time separation in the static schedule. All tasks executed under the Red kernel share a common stack. A Red task is implemented by a C function, and the task is completed when the function returns.

The *Blue kernel* administrates event driven execution of dynamic scheduled 'Blue tasks', mainly intended for applications having soft real-time requirements. Task handled by the Blue kernel are scheduled on-line by a fixed priority pre-emptive scheduler. Synchronizations among Blue tasks are managed by a Priority Ceiling Protocol (PCP)[11]. As opposed to the Red execution model, the Blue execution model does not support stack sharing among Blue tasks. Blue tasks are commonly used as indefinitely looping tasks (see Figure 1) periodically reactivated by system calls, e.g., *blueSleep*, that suspends the execution of Blue tasks for a specified time interval.

The *Green kernel* handles external interrupts. The 'Green tasks' are scheduled on-line with a priority based scheduling algorithm dependent of the application hardware, i.e., microprocessor. The Rubus off-line scheduler is guaranteed to generate a static schedule (see Red kernel above) with sufficient slack available to handle interrupts [10]. When a Green task is executed, it may utilize the stack of the currently active Red or Blue task, implying that the active task may need to supply stack space for interrupt handling.

Dispatch priorities of the tasks executing under the different kernels are illustrated in Figure 3. Tasks managed by the Green kernel have highest priority, and tasks managed by the Blue kernel have lowest priority.

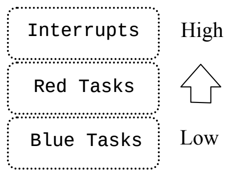

Figure 3. Task priorities in Rubus

Rubus supports the possibility to utilize software components for application development. The computational part of the supported software components is realized either by a Green, Red or by a Blue task.

4. Integration of SSX in Rubus

Introducing a new execution model in an operating system for resource constrained embedded real-time systems, require careful design to minimize the overhead of the new model and effects (temporal and spatial) on existing models. On one hand, we could minimize the memory overhead imposed by the new execution model, by sharing administrative code in the kernel between the existing execution models and the new execution model. In doing so, we would impose additional timing overhead on the existing models wherever a kernel needs to be able to separate the different models, e.g., at sorting, queuing and error handling etc. On the other hand, we could avoid imposing timing effects on the existing models by separating the models, i.e., modularize, and allow the kernel to administrate the new SSX model in isolation from the existing execution models. This approach would increase the number of administrative functions, thus requiring more memory.

In this implementation, we choose to share administrative OS code between the SSX model and the Blue model since the timing overhead imposed by the SSX model, on the Blue model, is very low.

A new execution model may be introduced to a system by changing the current scheduling policy or existing task model. In our case, we retain the same scheduling policy for the SSX model as for the Blue model (fixed priority scheduling). However, a new task model is introduced to support the SSX model. Each task in Rubus is defined by its: basic attributes, Task Control Block (TCB), stack/heap memory area and application code. By adding periodicity and deadline attributes to the existing task model, we are able to share all fundamental task structures between SSX tasks and the existing Blue tasks. Administration of SSX tasks is handled entirely by the Blue kernel (Figure 4).

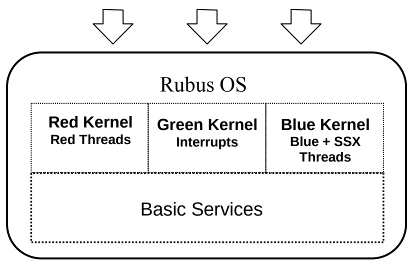

Figure 4. Rubus RTOS architecture with SSX model

The resulting relation of task priorities in Rubus, including SSX, is illustrated in Figure 5. The priority assignments and the fact that the administration of SSX tasks are handled entirely by the Blue kernel, makes the temporal attributes of tasks using the SSX model fully analyzable. In systems consisting solely of SSX tasks, the analysis can be performed with [2]. The SSX tasks can also be analyzed in hybrid systems consisting of Interrupts, Red tasks and SSX tasks with [8].

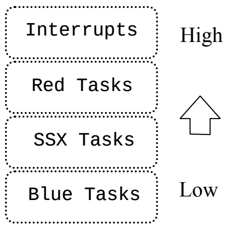

Figure 5. Task priorities in Rubus, including SSX

All tasks executing under the SSX model share a common stack (in fact, there is nothing that prevents stack sharing also between Red and SSX tasks). The common stack pointer, for SSX tasks, is globally accessible, hence it does not have to be stored in the TCBs.

To support resource sharing in the SSX model, the immediate priority ceiling protocol was implemented.

The following is a summary of all major changes made in Rubus to support the SSX execution model:

- Separation of tasks administrated by the Blue kernel and executed under different models
- Modification of administrative functions to support SSX tasks
- Error detection for SSX tasks
- Activation functionality for the SSX tasks
- Introduction of the immediate priority ceiling protocol

The integration of SSX in Rubus allows the execution model to be directly applicable for the Rubus component model. Hence, the possibility to utilize software component for applications has been extended to include four execution models, the Green, the Red, the Blue and the SSX model.

The shared stack in the SSX model can be safely dimensioned, as shown below, by summing the maximum stack usage of all tasks in each priority level, and adding stack-space for interrupts. SSX tasks with equal priorities cannot pre-empt each other in Rubus, hence it suffice to take the maximum stack usage in each priority level.

$$su_{ssx} = su_{\text{int}} + \sum_{\forall i \in P} \max_{\forall \tau_j \in P_i} su_j$$

su$_{ssx}$, denotes maximum stack usage of all SSX tasks. *su*$_{int}$, denotes interrupt stack usage. P_i denotes the set of tasks with priority *i*. *P* denotes the set of all priority levels. τ_i denotes task *i*. $su(\tau_i)$ denotes stack usage, including context switch overhead, of task *i*.

A more accurate dimensioning approach would be to examine possible pre-emptions, and identify the pre-emption(s) resulting in maximum stack usage. Identification of possible pre-emptions in a fixed priority based system is considered in [5].

5. Evaluation of SSX in Rubus

Stack sharing allows for an efficient memory usage, which may avoid or at least postpone the need for additional RAM in evolving systems. To illustrate how a shared stack affects memory usage, we simulate different execution scenarios where the total stack usage varies a lot depending on the execution model in use.

We simulate three different execution scenarios using two different execution models, the Blue and the SSX model, for each scenario. The first scenario is obtained from a flyer promoting the SSX5 RTOS [9]. The second scenario models a traditional control application, where a sequence of tasks is used to sense, calculate control parameters, and actuate. These tasks are executed in sequence, hence they do not pre-empt each other. The third scenario illustrates a system with full pre-emption depth, i.e., all tasks are pre-empted. The scenario can be seen as an example where the benefit of SSX is less, e.g., SSX as interrupt handling tasks in systems with multiple interrupt levels.

5.1 Evaluation Method

The evaluations are performed under a Rubus OS simulator running on a PC. We calculate the stack usage as the maximum number of data pushed onto the stack, from the dispatching of a task until it has finished execution. In doing so, we are able to include the concealed pushes of stack frames, i.e., stack usage occurring before execution of the actual task code, in the calculations.

All stacks are initially filled with a pre determined data pattern. We then calculate the number of overwritten data patterns, i.e., stack usage, by examining the content of the stacks at termination of the system. It is assumed that tasks do not push frames identical to the pre determined data pattern at run time.

5.2 Application description

In each of the following execution scenarios, timer interrupts are generated at a frequency of 100Hz. Each timer interrupt activates the Blue kernel task, responsible for time supervision and dispatching of the tasks in the system. The service routine for timer interrupts, having highest priority in the system, execute on the stack of an active task. However, in the following scenarios, the worst-case execution times of the tasks are very short, resulting in all tasks finishing their executions before any consecutive timer interrupt hits the system.

The run time model in each of the following scenarios is fixed priority, pre-emptive scheduling. We denote the priority of a task with π, and its period, or in the case of sporadic tasks, its minimum interarrival time with T.

Scenario 1

The task set in the following scenario, obtained from a flyer evaluating the overheads of SSX5 [9], consists of; seven periodic tasks with periodicities ranging from 10ms to 80ms, and three interrupt handling tasks with a minimum interarrival time of 20ms (see Table 1).

Table 1. Task set, Scenario 1

Task	π	T (ms)	Stack usage (bytes) SSX / Blue
τ_{KERNEL}	15	10	144 / 132
τ_1	5	10	72 / 152
τ_2	5	10	72 / 152
τ_3	4	20	72 / 152
τ_4	4	20	72 / 152
τ_5	3	40	72 / 152
τ_6	2	80	72 / 152
τ_7	2	80	72 / 152
τ_8	5	$\geq$20	72 / 72
τ_9	5	$\geq$20	72 / 72
τ_{10}	5	$\geq$20	72 / 72

Running the system under the SSX model (with one shared stack for tasks $\tau_1 - \tau_{10}$), results in a total stack usage of 316 bytes. With the Blue kernel stack included, the total stack usage is 460 bytes.

Yet again, we evaluate scenario 1 but with the difference that tasks $\tau_1 - \tau_7$ are executed as Blue tasks (Blue execution model), achieving a pseudo periodic behavior by a call to a sleep function. According to the Rubus OS reference [1], the suspension (sleep) of the tasks requires two additional local variables, and besides the sleep call, an additional call to a function that converts the suspension time into timer ticks, resulting in increased stack usage (from 72 bytes to approximately 152 bytes) for a Blue task. This results in a total stack usage of 1480 bytes for tasks $\tau_1 - \tau_{10}$. With the Blue kernel stack included, the total stack usage is 1612 bytes. We noticed that the kernel uses 12 bytes less stack under the Blue model, than under the SSX model. This is due to Blue tasks scheduling themselves, instead of being assigned an activation time by the kernel.

Table 2 shows the resulting stack usage for scenario 1.

Table 2. Stack usage, Scenario 1

Exec.Model	Total stack usage (bytes)
SSX	460
Blue	1612
Savings	≈ 71%

Scenario 2

The following scenario consists of pure periodic tasks with harmonic period times (see Table 3). The scenario can be seen as a simplification of a typical vehicular control system, e.g., as described in [6].

Table 3. Task set, Scenario 2

Task	π	T (ms)	Stack usage (bytes) SSX / Blue
τ_{KERNEL}	15	10	144 / 132
τ_1	5	10	72 / 152
τ_2	5	10	72 / 152
τ_3	4	20	72 / 152
τ_4	4	20	72 / 152
τ_5	3	40	72 / 152
τ_6	2	80	72 / 152
τ_7	2	80	72 / 152

Table 4 shows the resulting stack usage for scenario 2 under the SSX and Blue execution models.

Table 4. Stack usage, Scenario 2

Exec.Model	Total stack usage (bytes)
SSX	216
Blue	1196
Savings	≈ 82%

Scenario 3

The previous scenario shows an ideal situation for introducing SSX tasks. However, in applications where most tasks are asynchronous and pre-emptions appear randomly, the gains of SSX tasks is less, Thus, this scenario is prepared to show that the total stack usage, in certain situations, is nearly identical between the SSX and Blue execution model.

The task set in this scenario consists of one periodic task τ_4 and three event-triggered tasks $\tau_1 - \tau_3$ (see Table 5). The execution of the task set is prepared to exhibit full pre-emption depth meaning that if a task can be pre-empted it will be so. Each task is assigned a unique priority, thus enabling pre-emption between each pair of tasks.

Table 5. Task set, Scenario 3

Task	π	T (ms)	Stack usage (bytes) SSX / Blue
τ_{KERNEL}	15	10	144 / 132
τ_1	5	-	72 / 72
τ_2	4	-	72 / 72
τ_3	3	-	72 / 72
τ_4	2	80	72 / 152

Table 6 shows the resulting stack usage for scenario 3 under the SSX and Blue execution models.

Table 6. Stack usage, Scenario 3

Exec.Model	Total stack usage (bytes)
SSX	612
Blue	708
Savings	≈ 14%

5.3 Results

Simulations have shown that stack memory usage in Rubus OS varies when comparing systems executed under the SSX model and systems executed under the Blue model. The differences in stack usage are mainly dependent on the type of application being realized. The fact that each Blue task is allocated its own stack makes them less memory efficient in all scenarios.

In an example system of 7 non-pre-emptable tasks, the difference in stack memory usage is as much as 82% less for SSX tasks than for Blue tasks. Another system derived from a flyer on SSX5, results in a difference of 71% less stack usage for the SSX tasks than for the Blue tasks.

However, less difference in stack usage is observed in situations of deeply nested pre-emptions. As the pre-emption depth increases, the difference in stack usage typically decreases. This is shown by our simulations of a system with full pre-emption depth where the difference in stack usage between the SSX model and the Blue model, is relatively low.

Hence, the SSX model is specifically suitable for applications where jobs (or transactions) of dependent tasks are modeled without pre-emptions within the jobs e.g., control systems. On the contrary, the SSX model is less beneficial for applications experiencing large pre-emption depths. However, in any type of application, the SSX model is at least as resource efficient, with respect to stack usage, as the Blue model. This makes the SSX model an attractive choice when developing systems.

6. Conclusion and Future Work

In this paper, we presented the integration of a resource efficient and predictable single shot execution model in the Rubus RTOS. The model allows for efficient stack usage and predictability of temporal attributes. These facts make the model attractive for development of resource constrained real-time systems. The integration has shown that the model can be integrated with very simple run-time mechanisms.

As future work, we are planning to include support for development and analysis (temporal and spatial) of SSX in Rubus Visual Studio (VS), which is an integrated environment for design, simulation and analyzing of embedded real-time applications.

7. References

[1] Arcticus Systems AB, http://www.arcticus.se

[2] N. Audsley, A. Burns, R. Davis, K. Tindell, A. Wellings, "Fixed Priority Pre-Emptive Scheduling: An Historical Perspective", Real-Time Systems, 8(2/3), 1995

[3] T.P. Baker, "A Stack Based Resource Allocation Policy for Real-Time Processes", In Proceedings of the 11th IEEE Real-Time Systems Symposium, 1990

[4] R. Davis, N. Merriam, and N. Tracey, "How Embedded Applications using an RTOS can stay within On-chip Memory Limits", In Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems, June 2000

[5] R.Dobrin, G.Fohler, "Reducing the number of preemptions in fixed priority scheduling", In Proceedings of the 16th Euromicro Conference on Real-Time Systems, July 2004

[6] K. Hänninen, T. Riutta, "Optimal Design", Masters thesis, Mälardalens Högskola, Dept of Computer Science and Engineering, 2003.

[7] C.D. Locke, "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives", Journal of Real-Time Systems, 4, 1992

[8] J. Mäki-Turja, K. Hänninen, M. Sjödin, "Efficient Development of Real-Time Systems using Hybrid Scheduling", In Proceedings of the International Conference on Embedded Systems and Applications (ESA), June 2005

[9] Northern Real-Time Applications, SSX5 true RTOS, 1999, http://www.ssx5.com/NRTAHome.htm

[10] K. Sandström, C. Eriksson, G. Fohler, "Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System", In Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications, 1998

[11] L. Sha, R. Rajkumar, JP. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Volume: 39, Issue 9, September 1990

[12] J. Xu, D.L Parnas, "Priority Scheduling Versus Pre-Run-Time Scheduling", International Journal of Time-Critical Computing Systems, 18, 2000