

# Highly Dependable Time-Triggered Operating System

## Static Scheduling Approach and Effective Run-Time Implementation

*A new generation of highly dependable fault-tolerant real-time control systems (such as automotive X-by-wire applications) is under development. Specific target area requirements lead to the new features to be supported by the system software. These requirements are best supported by a time-triggered approach. This article provides an overview of static scheduling process as well as compares possible run-time implementations of a time-triggered system. The main issues preventing effective static schedule generation are discussed and possible solutions provided. The approach described in the article is suitable for the highly dependable application domain (e.g. provides enhanced performance, predictability, error detection and fault tolerant communication protocol support).*

### INTRODUCTION

There are two basic principles of how to build real-time control system: event-triggered approach and time-triggered one. A wide range of existent Operating Systems (OS) supports event-triggered approach. In the case of event-triggered system, all system actions (including task activation and event setting) are triggered by external events and there are user-level system services similar to `ActivateTask()` and `SetEvent()` therefore. OS scheduler performs run-time task state transitions based on the scheduling criteria. Fixed priority preemptive scheduling is used in the most cases. Per contra, time-triggered approach assumes that an application timing behavior is known at design time and the OS performs task state transitions internally. Therefore there are no user-level services similar to `ActivateTask()` and `SetEvent()` in the system, but there is a statically generated Time Table instead. The Time Table is generated on system configuration phase as a result of application timing analysis and static scheduling. In the case of time-triggered system, fixed priority preemptive scheduler may be used as one of the possible implementations, but another approaches should be considered also.

### REQUIREMENTS FOR HIGHLY DEPENDABLE OS

OS targeted for highly dependable fault-tolerant real-time control systems shall satisfy the following requirements:

- guarantee task timing constraints (including task deadlines);
- support the needs of highly dependable application domain (e.g. support error detection, deadline monitoring and utilize all advantages of a target communication protocol);
- minimize run-time calculations in order to ensure safety;
- provide a programmer with a simply tested highly predictable environment (restricted set of OS ser-

vices, constant OS response time).

The most appropriate way to implement distributed highly dependable system is to use special communication hardware supporting fault tolerance and to implement operating system on the top of such hardware. As an example of suitable communication hardware, fault-tolerant communication protocols constructed around the ideas originally supported by TTP protocol [1] may be considered. Time-triggered approach is preferable in the cases where the desired level of safety cannot be reached with traditional scheduling techniques [2] and a restricted set of OS services is required therefore. Those systems often build on mechanisms that are synchronized and time-triggered. E.g. tasks perform their actions at specific times; the tasks and messages are statically scheduled. The discussion of time-triggered approach is not a subject of this article.

### FIXED PRIORITY PREEMPTIVE SCHEDULING

As it is widely known, in the case of fixed priority preemptive scheduling for periodic independent tasks when deadlines are equal to periods, Rate-Monotonic approach [3] may be used to ensure that all tasks will meet their deadlines. Here are some definitions:

- $T_i$  - task period;
- $D_i$  - task deadline;
- $C_i$  - task worst case execution time (WCET);
- $B_i$  - the worst case blocking time of the task by any lower priority task.

In according with the Rate-Monotonic Scheduling theory, all periodic tasks will meet their deadlines in the system supporting fixed priority preemptive scheduling, if the following conditions are satisfied:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_i + B_i}{T_i} \leq i \left( 2^{1/i} - 1 \right) \quad \forall i, 1 \leq i \leq n$$

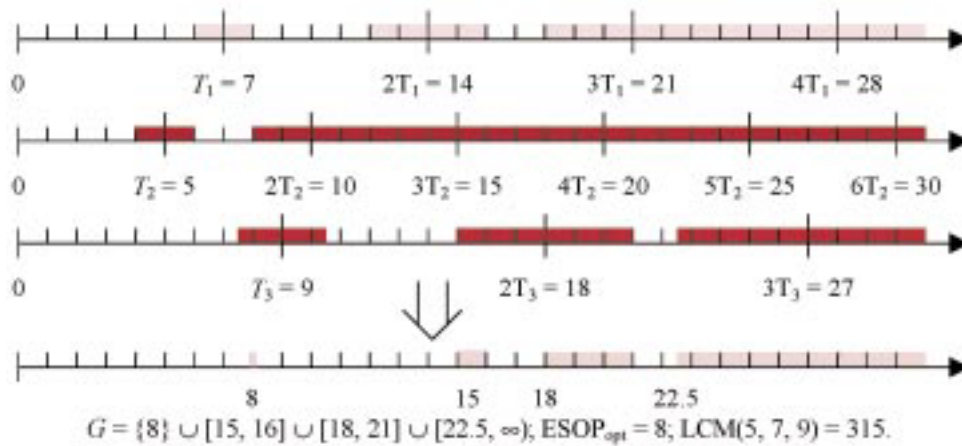


Figure 1. Example of ESOP optimization.

Hence, utilization bound for many tasks is 69%. For the details on application timing analysis and a method of task worst-case response time determination, please refer to [4].

## STATIC TASK SCHEDULING

In contrast with the event-triggered approach, in a time-triggered system all dependable communication and computation actions are triggered by the progression of time. Basically, a clock tick is the major event in such a system. All activities are assigned to certain clock ticks. When the clock reaches a value, to which an activity is assigned, the corresponding activity is initiated (e.g., task is activated). Optionally sporadic tasks (interrupt service routines) are supported to react to sporadic external events. The main entity of time-triggered operating system is the Time Table generated by offline Static Scheduling tools. Time Table defines run-time task state transitions. The Time Table is executed cyclically providing a periodic task execution scheme. For the details on time-triggered task activation scheme, please refer to [5]. There are some additional task characteristics essential for the time-triggered system:

- $\delta_i$  task period tolerance;
- $\varphi_i$  task offset from the beginning of the communication round;
- $E_i$  task best-case execution time (BCET).

In a general case, Entire System Operation Period (ESOP) is the least common multiply of the periods of the periodic tasks. As a result, the Time Table will hardly fit into the available memory if task periods are not harmonized. There are two methods on how to solve the problem mentioned above: ask the user to change individual task periods to stick to the reasonable range of basic values; introduce additional task period tolerance parameter ( $\delta_i$ ) and to let the system to optimize Time Table size internally. First approach is very simple and works in some cases. However, it will lead to the CPU utilization degradation because task periods will be decreased which will waste CPU time. Another approach is based on the assumption that we may allow the system to change task periods internally during Time Table generation. Practical data shows the viability of this approach. Task period decrease is not a problem from functional point of view except of possible performance degradation. In the most cases, nothing worst will happen if the system will increase

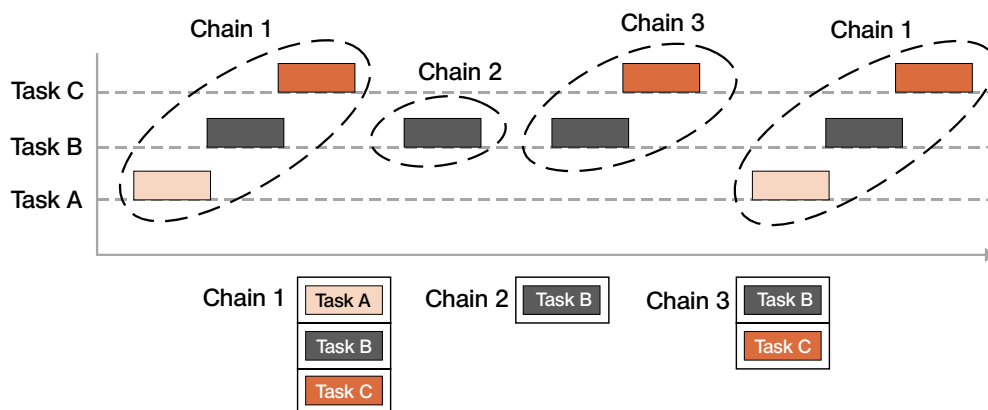


Figure 2. Task chain generation.

non-critical task period up to the bounded maximum value defined as a part of the system configuration description. Task period increase in statically scheduled system corresponds to the task entry latency in the event-driven system. Task period tolerance is defined in the percents of task period. Introducing  $\delta_i$  means that if an instance  $j$  of the task  $p_i$  is running, the next instance ( $j+1$ ) of the task  $p_i$  will be activated after  $T_i$  time where  $T_i$  satisfies the following formula:

$$T_i(1 - \delta_i) \leq T_i' \leq T_i(1 + \delta_i)$$

This approach is more effective in terms of Time Table generation because introduces flexibility in static schedule generation process. There is a range of allowed task periods for each task. Optimal ESOP may be found as an optimal value suitable for the whole task set. Please consider the following example (Figure 1). There are three periodic tasks:

$$T_1 = 7 \pm 1; T_2 = 5 \pm 1; T_3 = 9 \pm 1.5$$

In this example, ESOP was decreased by factor 40. In the real cases, introducing 5% of task period tolerance, may decrease ESOP dramatically. Moreover, having absolutely different task periods is not a common practice in embedded systems. Therefore we may fit to the available memory resources in the most cases. It is known, that there are at least two optimal algorithms for dynamic task scheduling: EDF (Earliest Deadline First) and LLF (Least Laxity First). Run-time EDF and LLF implementations are not widely used at the moment due to the significant run-time scheduling overhead and difficulties in application behavioral analysis. However these techniques may be successfully used for static scheduling. The most effective strategy in terms of runtime performance explored at the moment is EDF-based type of static scheduling. With EDF-based static scheduling, application run-time behavior will be the same as in the case of EDF run-time scheduler. Static scheduling allows us to do the most part of the scheduling job offline as well as to do a number of offline optimizations. One of the possible

optimizations (task chain generation) is shown in Figure 2.

A task chain contains a sequential list of tasks with precedence relations between them. The main purpose of the task chain concept is to reduce the operating system overhead by reducing the number of trigger interrupts invoking run-time scheduler. In the example shown above, Time Table will have four entries for the task chains (Chain1, Chain2, Chain3) instead of nine entries for separate tasks (TaskA, TaskB, TaskC). Task BCET ( $E_i$ ) configuration attribute is necessary for effective task chain generation. User-defined task chains may be effectively combined with generated task chains on system configuration stage. Resource and message access constraints may be resolved offline as well. Offline tools are able to optimize the amount of local message copies needed to ensure data consistency based on the precedence relations between the communicating tasks.

The detailed analysis of static scheduling approach is out of the scope of this article. For the discussion on static scheduling, please refer to [6]. There are different options on how to implement run-time part of the system in the case of time-triggered approach. First option is to do a partial static scheduling (e.g. just group tasks into the task chains and generate Time Table entries), assign fixed task priorities and let the fixed priority preemptive scheduler to do the rest of the job. Second option is to do a complete offline scheduling. In this case, a simplest run-time scheduler not dealing with task priorities may be implemented. This simplest run-time scheduler will just start tasks in the order defined in the Time Table. In the remaining sections we will describe the simplest stack-based run-time implementation as well as compare both approaches from system scheduability, run-time performance, and predictability points of view.

**COMPLETE STATIC SCHEDULING**

In this chapter it is assumed that EDF-like scheduling algorithm is used for the Time Table generation. E.g. the main scheduling criteria is to guarantee task deadlines. This means that if TaskA should be activated in

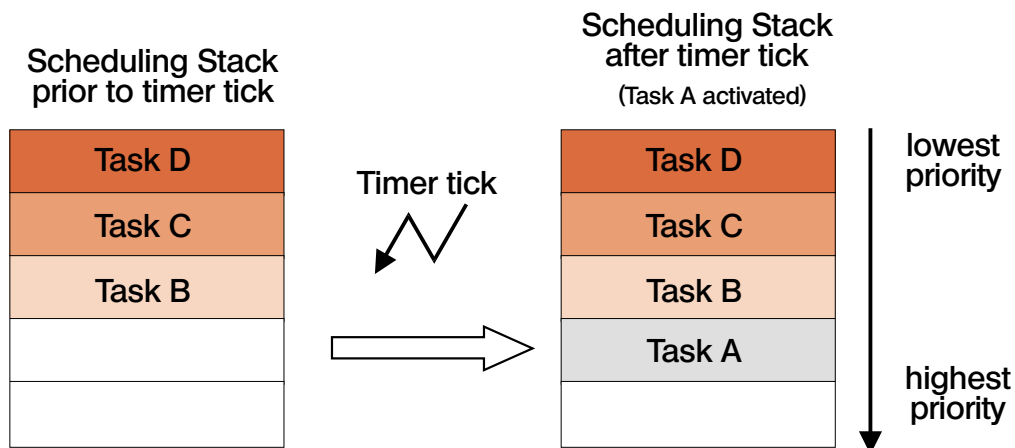


Figure 3. Scheduling Stack.

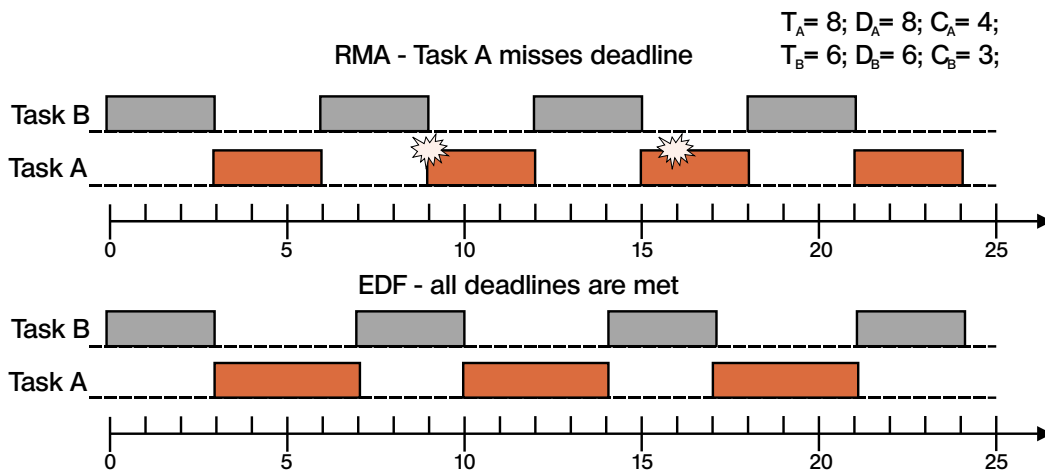


Figure 4. RMA falls, but EDF works.

accordance with the Time Table and the system state is correct, TaskA has the shortest deadline. In this case, a task may not have a constant task priority, but rather a dynamic task priority instead. OS Scheduler assigns the dynamic task priority for each task instance during task activation based on the current System State (the number of the tasks in running and ready states). Therefore if TaskA should be activated in accordance with the Time Table, the OS Scheduler will increase the dynamic highest task priority in the system and will assign this priority to TaskA to let TaskA finish prior to the other tasks in order to guarantee TaskA deadline. Therefore task TaskA may preempt task TaskB (in the full-preemptive system) if TaskA deadline is close to the expiration and vice versa based on the current System State and the timing constraints defined in the Time Table. The structure of the scheduling stack (full-preemptive system) is presented in Figure 3.

If a task is activated, the first free cell of the scheduling stack array is allocated for the task. If a task is terminated, the corresponding memory cell is de-allocated. The dynamic task priority defines the cell of the scheduling stack to be occupied by the task as a result of the timer tick. The size of the scheduling stack is defined upon system configuration. An error code indicating scheduling stack overflow may be raised during run-time. In the example shown in Figure 3 (left part), TaskB has the highest dynamic priority, while TaskD has the lowest dynamic priority. As a result of the timer tick activating TaskA (right part of the Figure 3), TaskA occupies the bottom of the scheduling stack (has the highest priority). That's all and very simple. Non-preemptive task scheduling policy is also available.

## SYSTEM SCHEDULABILITY AND CPU UTILIZATION COMPARISON

The main arguments against full static scheduling are the following:

- Unreasonable huge size of Time Table;
- The necessity to split long tasks into a number of sub-tasks;
- Performance decrease due to the event pulling;

- Delays in high priority event service.

Now we know how to optimize the size of Time Table. The remaining issues are based on the assumption that only non-preemptive type of run-time scheduling is supported and that a lot of CPU time is reserved for sporadic tasks (interrupt service routines). As shown above, full-preemptive run-time scheduling is also available. E.g. it is not necessary to split long tasks into a number of sub-tasks. If a task did not spend all CPU time defined by  $C_i$ , it may leave the remaining time to the other tasks in the ready task set. As to the time reserved for sporadic tasks, the situation is absolutely the same as we have in the case of fixed task priorities - enough time should be reserved to execute all sporadic tasks under a peak load scenario. There is a boundary - sporadic task minimal interarrival time. If task chain is generated, the time should be reserved for the whole task chain, not for the individual tasks containing in the task chain. Sporadic tasks may handle high priority events. There are a number of articles saying that full static scheduling decreases CPU utilization significantly, which is not always true. In some cases, complete static scheduling even increases CPU utilization. The main advantage of the static scheduling approach is a chance to resolve resource access constraints offline. In this case, we will eliminate  $B_i/T_i$  composed in formula (1), which will increase CPU utilization comparing with the Rate Monotonic approach. Moreover, an application may still be statically scheduled using EDF while RMA does not guarantee schedulability due to overloading utilization bound. Please see the following example (Figure 4) derived from [6].

There is common situation that some tasks are non-preemptive ones (or should be executed with disabled interrupts) in which case we have difficulties trying to analyze application schedulability via RMA and have difficulties implementing OS interrupt management services in a consistent manner [7]. In this case, complete offline scheduling may offer better solution. Time Table will guarantee that one critical task will not interrupt another. In some cases, it is not possible to achieve application constraints (e.g. to guarantee

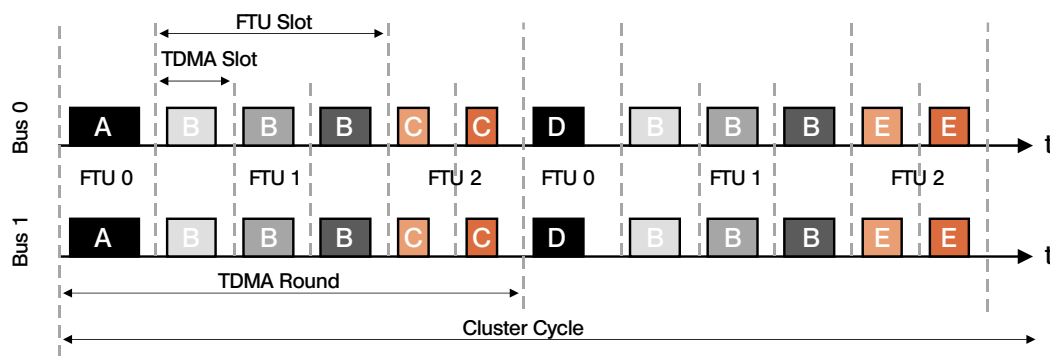


Figure 5. TDMA Bus Access Scheme.

precedence relations between tasks) in a system supporting fixed priority preemptive scheduling because task priorities derived from RMA criteria conflict with task priorities necessary to achieve a desired order of task execution. Static scheduling will help here too.

### COMPARISON FROM FUNCTIONAL POINT OF VIEW

First of all, in the case of fixed priority preemptive scheduling it is hard to guarantee a constant response time due to the complex run-time calculations (such as schedule queue processing). If fixed priority preemptive system is used, task scheduling is not performed completely by a static scheduling tool. As a result, application configuration process cannot be automated in a full volume. During the application configuration phase, application designer should know the run-time behavior of the complete system in order to harmonize task priorities with task timing characteristics (WCET, period and deadline). However, there are a number of techniques (such as RMA and DMA), which may be used to assign task priorities consistently. Additionally, fixed task priorities may prevent effective static schedule construction (one task can not preempt another due to unnecessary priority constraint). With a fixed priority implementation, generic run-time resource and message access conflict protection mechanisms (e.g. Priority Ceiling

Protocol and additional message buffers) may be required even if such conflicts may be resolved offline. Therefore fixed priority preemptive scheduling leads to a relatively complex run-time part and cannot utilize all advantages of time-triggered approach. The main advantage of a priority-based dynamic scheduling is the possibility to run the system without precise estimation of task WCET's. The simplest stack-based implementation provides a constant response time, minimizes run-time calculations and guarantees timing constraints (if offline scheduling is done correctly). Per contra, very precise task WCET estimation is required in this case. If task WCET is estimated incorrectly, run-time system behavior will be wrong. But if automatic application code generation is used (which becomes a common practice these days), task WCET may be relatively simply received as an outcome of application development process. In any case, it is

impossible to prove system safety and guarantee schedulability without having a complete set of task timing characteristics.

### CPU RESOURCE USAGE COMPARISON

Experience with the fully tested time-triggered OS implementation supporting complete static scheduling shows that at least 2.5x RAM and ROM usage reduction may be achieved on 16-bit CPU's without special RTOS hardware support in the case of stack-based implementation. The same situation is in terms of run-time performance due to the absence of run-time calculations and offline resource access conflict resolution. For 32-bit CPU's (e.g. PowerPC), OS task activation overhead will be similar to the interrupt entry latency. If acceleration hardware is required (such as FPU or multimedia co-processors), additional performance benefit may be achieved because the minimum amount of information to be saved and restored during each context switch will be calculated offline and incorporated into the Time Table. Minimal number of task context switch operations guaranteed by EDF algorithm and not reachable with RMA is a plus also.

### FAULT-TOLERANT SYSTEM SUPPORT

OS designed for highly dependable applications should support error detection, deadline monitoring as well as utilize all advantages of target fault-tolerant protocol. Pure run-time implementation of task deadline monitoring is time consuming and ineffective. Only the presence of offline static scheduling tools allows effective task deadline monitoring implementation. There are two task deadline monitoring options in the case of the static scheduling: incorporate additional task deadline monitoring entries in the Time Table exactly at the time of task deadline expiration and to group task deadline monitoring entries with existent Time Table entries. The second option is more preferable in terms of run-time performance because does not lead to additional interrupts. In the case of complete static task scheduling, additional third option comes up. The size of the stack-like scheduling structure may be checked together with a task activation, which usually does not require additional CPU resources at all. There

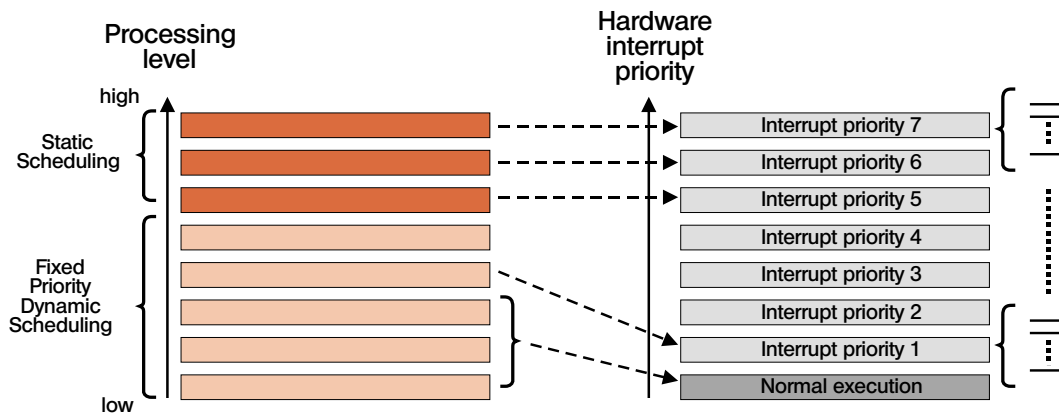


Figure 6. Combined system.

is a trend to use communication hardware supporting the Time Division Multiple Access (TDMA) bus access scheme for critical messages. Originally, TDMA bus access scheme was implemented in TTP protocol. Figure 5 describes TDMA bus access scheme. Each node is allowed to send only during a predetermined time span, called its TDMA slot. The sequence of the periodic TDMA slots is called a TDMA round. The set of periodically recurring TDMA rounds with possibly different message length and contents is called a cluster cycle.

Each communication controller schedules transmission and reception according to a statically generated message transition schedule or Message Descriptor List (MEDL). TDMA bus access scheme guarantees that there are no bus access conflicts between different communication controllers. In the case of complete static scheduling, it is guaranteed that task will be started exactly at the expected point of time. Therefore it may be ensured that there are no communication device access conflicts between different tasks. Task offset attribute ( $\varphi_i$ ) may be used by offline tools to ensure that there are no message access conflicts between the communication layer and the application code. Possibility to do a direct communication device access from the application code and the absence of bus access conflicts lead to a very effective, fast and transparent communication layer implementation. If priority-based scheduler is used, no guarantees are given and more complex communication device access scheme should be supported.

## APPLICABILITY

There are a number of requirements to be met in order to use the approach described in the article effectively. Here is the list of requirements: · Full information on application timing behavior is available; · Application design is based on a time-triggered approach; · Time Table fits to the available memory; · Application complexity is not very high (e.g. resource access conflicts may be resolved offline); · Programmable interrupt timer is available. The major requirements are discussed already. Usually, all requirements are met in the highly dependable domain. The design of such applications

assumes periodic task activation and message transition schemes. As a result, the major part of the applications may be statically scheduled.

## COMBINED SYSTEM

If not all preconditions necessary for complete static scheduling are fulfilled in the case (e.g. exact task timing parameters are not known for non-critical background tasks or resource access constraints are difficult to resolve offline), combined system presented in Figure 6 may be used.

In this case, the highest priority part of the system will support complete static scheduling and will provide highly dependable functionality as well as will interact with communication hardware. All critical tasks will be executed at interrupt level. The rest part of the system will support fixed priority preemptive scheduling technique. If separate interrupt levels are used for different parts of the system, high-level application design tools may not deal with a timing model of the low level part of the system. Interrupt logic of the controller will guarantee that the low priority part of the system will not influence critical task schedule. Information characterizing the timing behavior of the highest priority part of the system (such as the duration of gaps in the Time Table) may be passed to low level design tools in order to allow RMA analysis. In the case of necessity, the duration of gaps in the static schedule may be easily controlled in the scope of high-level application design tools. Combined system is suitable if a target communication hardware supports mixed message transmitting scheme (time driven as well as event driven message frames). However, this approach requires hardware supporting a number of interrupt levels.

## CONCLUSION

This article provides an overview of static scheduling process as well as compares possible run-time implementations of a time-triggered system. It is shown that the issues preventing effective complete static scheduling implementation may be resolved if a number of preconditions are fulfilled. It is demonstrated that static scheduling technique may increase CPU utilization comparing with the fixed priority preemptive schedul-

***AD AMAZING  
ADVERTISING***

ing. If a complete information characterizing application timing behavior is available, simplest stack-based run-time implementation wins against an implementation supporting fixed priority preemptive scheduling in terms of CPU resource usage, predictability, error detection and fault tolerant communication protocol support. Combined system may be used if complete static scheduling cannot be done for a particular application. Also it is necessary to note, that the approach described in the article is suitable for the defined application domain where time-triggered system design approach, the absence of user-requested task state transitions and relatively simple application structure are the must (highly dependable applications). The approach may be used to cover a subset of real-time applications where fixed priority preemptive scheduling technique does not work or cannot achieve specific application requirements.

## REFERENCES

- [1] S. Polenda, G. Krois. The Time-Triggered Communication Protocol TTP/C. Real-Time Magazine, April 1998.
- [2] T H. Kopetz. Real-Time Systems. Design principles for Distributed Embedded Applications. Kluwer Academic Publishers, Boston.
- [3] L. Sha, M. H. Klein, J. B. Goodenough. Rate monotonic analysis for real-time systems. In Foundations of Real-Time Computing: Scheduling and Resource Management, Kluwer Academic Publishers, 1991.
- [4] K. Tindell. Deadline Monotonic Analysis. Embedded Systems Programming Magazine. June 2000.
- [5] OSEKtime: Highly Dependable Applications - Objectives, Basics and Concepts. Proceedings of 3rd OSEK/VDX workshop. Feb 2000, VDI Verlag GmbH, Dusseldorf.
- [6] J. Xu, D. L. Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. Real-Time Systems Magazine, 2000.
- [7] A. Zahir. An integrated concept of handling pre-emptions and interrupts for automotive real-time operating systems. Real-Time Magazine, 3Q99 ■

---

*Maxim Perevozchikov is a Software Engineer at Motorola Global Software Group, received his MS degree in computer science from St. Petersburg Electrotechnical University in 1998. Since 1999 he is working in Motorola St. Petersburg Software Development Center dealing with RTOS static scheduling and configuration tools. Currently he is involved in the project related to the operating system for personal communication devices.*

*Yaroslav Domaratsky is a Senior Staff Engineer at Motorola Global Software Group, received his Ph.D. in computer science from St. Petersburg Electro-technical University in 1999. Since 1994 he is working in the relation with Motorola doing RTOS development and testing. In 1997-2000 he worked in Motorola St. Petersburg Software Development Center with OSEK compliant product line supporting HCO8, HC12, CPU32, M.CORE, MPC5xx and MPC750. Member of OSEKtime working group since 1999. Currently he is involved in the project targeted for high-end PowerPC core.*



We welcome news from your company through press-releases that can be submitted on our website at  
<http://www.dedicated-systems.com/VPR>

Contact Nico Van Wijmeersch for more information:

Phone : 32-2-520.55.77

Fax : 32-2-520.83.09

Email : [info@dedicated-systems.com](mailto:info@dedicated-systems.com)