Implementation of OSEK/VDX Based OS for Usage in Railway Diagnostic System

Kristian Dilov Dilov and Emil Nikolov Dimitrov

Abstract – **The paper examines the main issues during the realization of OSEK real time operating system for ARM7 cores. Analysis and synthesis on the implementation of consisting in the operating system modules is provided. The main particularities related to hardware specific platform are reviewed.**

Keywords – **OSEK, dispatching, tasks, event and alarms.**

I. Introduction

The railways' electronic diagnostic systems are characterized by complexity in terms of their performance opportunities. On the other hand there are many restrictions that they must comply:

- time critical – possibility for real time operation;
- possibility of building component based software architecture;
- effective management of system resources;
- possibility for re-use of separate software components.

This task is hard for implementation, by using the opportunities of linear programming. It is necessary to build a strategy for creation of easy configurable software.

So identified requirements, can be adequately met through the use of real-time operating system (RTOS).

The RTOS approach is quite popular in the automotive industry over the past 10 years. The OSEK/VDX standard is a result of automotive manufactures' aspiration to create a standardized software infrastructure for transport electronic systems. The main considerations arise with the fact of continued increase in electronic systems complexity, which control is possible through a well-defined modules and links between them.

Program functional blocks are separated in software components - the smallest abstract logical units (fig. A). Component is an independent software object, realizing determinate functionality. It has several inputs and outputs for data exchange, called interface.

The components, whose parameters – such as activation event, cyclic execution period - are grouped in the so-called application tasks. The OSRT realizing its manipulations over these application tasks. This is the way to achieve:

- consecutive utilization of available system resources, which leads to higher performance, without a necessity of additional hardware;
- precise tasks execution planning, implemented by the system services of the RTOS.
- possibility for jointly pseudo-parallel execution of slow, time consuming tasks, with fast time-critical tasks;
- possibility for fast system's reaction on event, realized by a different set of services for interrupt handling;
- self-diagnostic and control of tasks' execution time;
- independence of the software in terms of used hardware.
- There is an OSEK requirement - the operation system to be static. The user creates a configuration – number of application tasks, alarms for tasks' activation and so on. Based on this configuration, the source code of the operating system is generated.

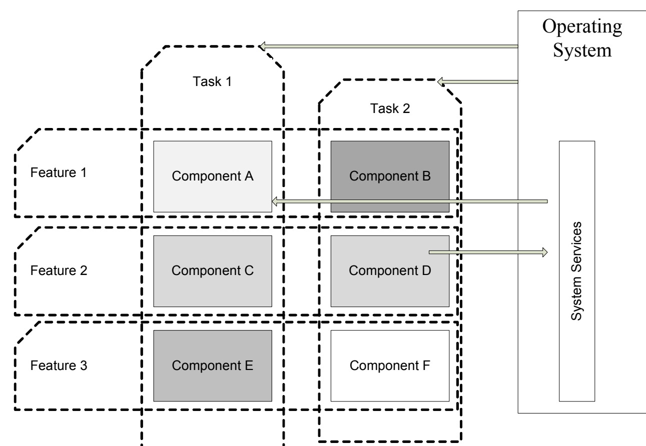

Figure 1. Functional features separation into tasks.

The chosen microcontroller for current implementation is LPC2106 by NXP Philips. It is based on ARM 7 core. The basic characteristics are:

- possible work as 16 bit word as well as 32 bit;
- 128 KB flash memory;
- 60MHz core frequency.

II. Implementation

The current implementation aims at splitting the source code in two parts – hardware independent and hardware dependant. This separation is appropriate in order to enable the code generation for the operating system, and to enable its portability to different microcontroller families.

K. Dilov is with the Department of Electronics and Electronics Technologies, Faculty of Electronic Engineering and Technologies, Technical University - Sofia, 8 Kliment Ohridski blvd., 1000 Sofia, Bulgaria, e-mail: kdilov@kdilov.com
E. Dimitrov is with the Department of Electronics and Electronics Technologies, Faculty of Electronic Engineering and Technologies, Technical University - Sofia, 8 Kliment Ohridski blvd., 1000 Sofia, Bulgaria, e-mail: edim@tu-sofia.bg

A. Dispatcher implementation

The dispatcher is responsible for process of application tasks' service. It is activated at definite moments – scheduling points:

- system time tick expiration;
- activation of an alarm;
- releasing of occupied resource, needed by higher priority task;
- event occurrence;
- explicit call to dispatcher from application tasks.

The OSEK/VDX standard defines three possibilities for scheduling policy, in accordance with tasks' states[1]. In the current implementation the mixed type is used – pre-emptive dispatching together with co-operative dispatching. The advantages of that choice are with respect to the specificity of implemented railway diagnostic system. There are time critical processes in this system – measurement of transport vehicle parameters. It is required that they are done with an precise sampling rate, and stable cyclic period. On the other hand it is not allowed to have a preemption on these measurement tasks.

Processes that have asynchronous nature, and relatively long processing time are placed in pre-emptive, low priority application task container. This could be the the diagnostic system's communication with a personal computer, used for transfer of measurement data. This approach allows to balance the load of the system, and respect to the time-critical requirements of the measurement.

An important point within the implementation of the operating system is the optimization of used resource – memory. The basic non-pre-emptive tasks[1] are started and executed in determinate, preliminarily known points of their program – beginning and end of the executable. It is not necessary to save their context temporary, during a state different than "running". This is the reason not to have a separate task's stack in the memory. Indeed they can use a shared memory, which is used by the task during its execution – "running" state.

The dispatcher groups the application tasks in accordance with their common symptom – their state. In the current implementation, three groups are realized, as linked lists:

- queue of ready for execution tasks – this is a tasks's list, arranged in descending order of their priority. The dispatcher points to the task with higher priority at list's beginning. The first task in the list will be activated, when the next scheduling point occurs.
- Queue of tasks waiting for occurrence of an event.

It is not necessary to arrange the tasks in some kind of order. After the event has come, the tasks associated with it are transferred by the dispatcher into the queue of ready for execution tasks. A particular case of such event is expiration of alarm.

- Queue of tasks, which need an access to a system resource when executed. Because of the asynchronous nature of the system, it is necessary to have procedure for resource management. This is the way to guarantee that in particular moment, only one task could occupy a resource. The tasks which required a resource occupation during their execution are grouped in a queue in descending order of their priority. When a resource is released, the highest priority task is transferred to the queue of ready for execution tasks. After it has completed its job, it releases the resource, and goes in state "suspended". This is the way, to grant resource access to the following next in terms of priority task, associated with this resource. There are two essential moments in the scheduling points, which need to be realized. The first is task context saving, in order to provide a possibility to it, to resume from the exact point, it was interrupted. The second one – activation of chosen for execution task by loading its context in the special function registers of the microcontroller.

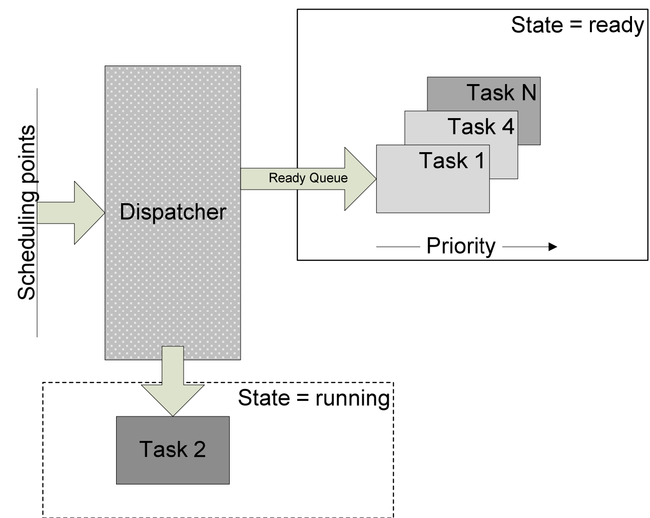

Figure 2. Tasks dispatching

These manipulations require a special access to the microcontroller's registers, available only in special mode - privileged mode. The used microcontroller has six modes. They are associated with definite events – system interrupts or exceptions. For the current implementation, the following are used:

Supervisor – with extended access to special function registers of the microcontroller – used for loading and saving tasks' context.

IRQ – this mode is entered, when an interrupt from the system timer occurs. This interrupt is used to form the basic time tick in the operating system. An access to necessary system registers is granted to the dispatcher.

User – this is the mode, which the tasks are executed, with restricted access to system registers.

The supervisor mode is used in case of explicit request for rescheduling from application task to the dispatcher. When a task is executed, the microcontroller is in user mode. It has no opportunity for manipulation over system registers, when it is necessary to reschedule the current executed task. When the task makes a request for rescheduling, the microcontroller should be in supervisor mode, realized as a software interrupt – function TerminateTask. The interrupt service routine activates the dispatcher, and the mode is supervisor because of the interrupt context. In this case, the dispatcher restores the context of next task for execution and starts it by the

function RestoreTaskContext. From microcontroller point of view, the activation of the task looks like return from interrupt. The difference is that the program counter is loaded with the first instruction of task for activation, and the stack pointer is set to the individual task's stack memory space.

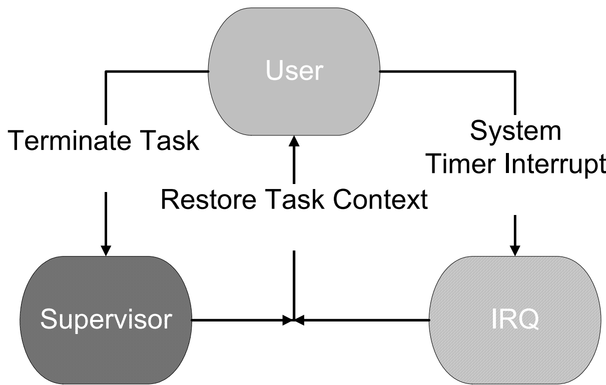

Figure 3. Transitions between modes of processor

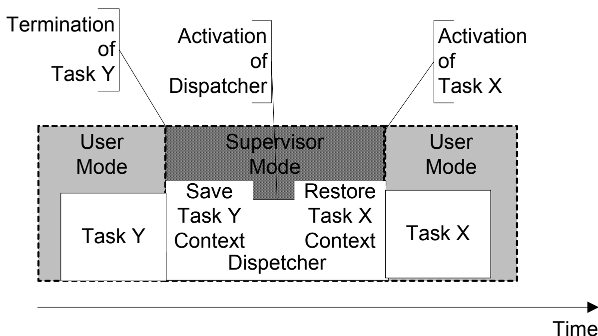

Figure 4. Tasks switching with TerminateTask.

A similar approach is used for rescheduling, when executed task is pre-emptive, and basic time tick expires – every 1ms. In that case, instead of software interrupt an interrupt from the time system is used. After the context of interrupted task is saved, then a processing of system events is performed – alarm management, ready task queue processing. As a final action within the interrupt service routine, the highest priority task is activated by loading its context into special function registers. The situation is - system interrupt suspends current task execution, manipulates the return address and stack pointer with context of highest priority task ready for execution. The returning from interrupt activates the new loaded task.

B. Realization of application tasks

The application tasks also called containers [1], are used for grouping of functional components, characterized with equal execution parameters – execution time cycle, priority and etc. The task possesses two sections in the microcontroller memory:

- Descriptor – used for task's state transitions management by the dispatcher – saving of task's context, when it is not in "running" state. Most of the fields in descriptor's structure are volatile, and this is the reason to be kept in the RAM memory.
- Runnable – this is the component's executable code, kept in the ROM memory, pointed by pTaskCode member of the descriptor.

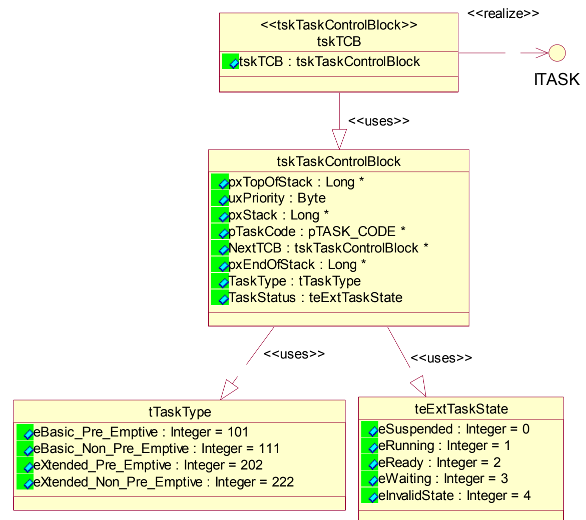

Figure 5. Task's descriptor

During execution of operating system, application tasks are changing their state[1] – field TaskType. The type of the application task[1] is kept in the descriptor member TaskType. The private stack space for the task is pointed by the pxTopOfStack. During the development phase, it is hard to estimate precisely the necessary volume of task's stack. From optimization point of view, allocation of needless large stack space is not a good idea. For this reason, the task's descriptor has a field – pxEndOfStack, which points to the physical end of the task's stack. During the initialization, this memory area is written with a pattern value, and stack's boarder with other pattern. The operating system has a chance to control the consumed stack. In case of overwriting the boarder pattern, RTOS suspends its normal execution, and marks the problem. This is a token for the user, that allocated memory space is underestimated for correct execution of application task.

C. *Alarm realization*

Alarms are events that occur after a predefined period of time [1]. Their activation could lead to task activation, alarm callback execution, or rising of a RTOS event. In the implemented operating system, there is a descriptor similar to this of application task, and used for alarm management from the OS.

The alarm is connected to a counter, which value its monitors. On the other hand, the counter increments its value on every system time tick. When a real time system is developed, based on implemented OS, there should be a strict plan for exact time for alarm start, and associated tasks' execution time. This is the way to guarantee distributed execution of application tasks in the time. Another constraint is to prevent overlapping between tasks execution. This effect will lead to out of synchronization for the time-critical, high priority tasks. For this reason there is a member in the alarm's descriptor used for starting time offset.

D. System interrupts

The system interrupts could be two types in the current OS – category 1 and category 2[1]. Category 1 are processed without operating system involvement. Because of this fact they have the lowest latent time. Within the context of this interrupt category, it is not permitted to use operating system services. Category 2 interrupts can be considered as highest priority tasks, activated with associated interrupt occurrence. When the interrupt occurs, the dispatcher is activated in T1 (fig. F). After the interrupted task context is saved, the dispatcher activates unconditionally the subprogram for interrupt service routine(ISR) – T2. At the end of ISR – T3, the dispatcher resumes the interrupted task – T4. The conclusion that the category 2 interrupt processing is similar to processing of system event is reasonable. The difference is in the obligatory resuming of interrupted task, after the ISR has completed. During the system event processing, it is possible to start a task different than the one executed before the occurrence of the event it-self.

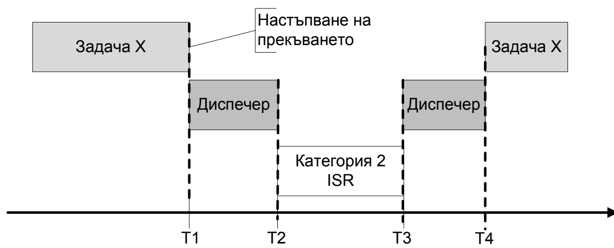

Figure 6. Category 2 interrupt processing.

E. System Generation

It is required by the OSEK standard, to have automated operating system generation based on user configuration. The implemented OS is fully static in its part for system services realization. The separate tasks, alarms and events can be presented as system tables, accessed by constant index. Therefore, their number and association do not affect the structure of the operating system. These features allow construction of a program for editing the configuration, and generating the code for the OS. After that as a next step the task containers are filled with the system interfaces of the design system components.

There are two basic points, which are dependant to hardware platform and have to be taken in account during the implementation of OS generation:

- Hardware initialization – frequency, port configuration, setting system tick interrupt. The possible solution is generating of empty functions filled by the user in accordance with his necessity; choice of standard preliminary defined configurations for supported platforms.
- Explicit request for rescheduling from task context, described in “Task dispatching”. For microcontrollers, which have a supervisor mode (*ARM, Futjitsu, NEC, Renesas*), this can be realized by software interrupt or delayed interrupt. For the other – without modes with restricted access to special function registers (MSP430, PIC). The system is generated with stubbed macro for rescheduling. For the first group it is connected to software interrupt or delayed interrupt. The respective ISR are activating only the dispatcher. For the second group, the macro is connected directly the dispatcher. This approach unifies the generation for most of the microcontroller.

III. Conclusion

The usage of real time operating system provides the user with following advantages:

- possibility for optimal organization of the software;
- effective utilization of available hardware resources;
- planning of tasks’ execution – effective system performance;
- fast and easy tuning of the system during the development phase;
- possibility for re-use of software components on different hardware platforms.

Used microcontroller in the current implementation – LPC2106 is a suitable for real time systems. The availability of different modes, allows isolation of access to critical system resources from user application tasks. Only dedicated static sections – dispatcher and system services - have access to them, which make the system more resistant to eventual problems in the user tasks.

References

[1] K. Dilov, E. Dimitrov. *Study of OSEK OS possibilities for usage in railway electronic diagnostic systems***.** In the same edition
[2] М. Луканчевски. *Системно програмиране за едночипови микрокомпютри*
[3] *OSEK/VDX - Operating System Specification 2.2.22*
[4] J.K Stankovic, J. K. Ramamritham. *The design of the Spring kernel.* Real time systems symposium San Jose 1987
[5] Shao B., R. Wang, *EMBEDDED REAL-TIME SYSTEMS TO BE APPLIED IN CONTROL SUBSYSTEMS FOR ACCELERATORS* - Tsinghua University, Beijing
[6] http://portal.osek-vdx.org/
[7] http://www.autosar.org/