We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

7

Operating System Kernel Coprocessor for Embedded Applications

Domen Verber and Matjaž Colnarič
University of Maribor, Faculty of Electrical Engineering and Computer Science
Slovenia

1. Introduction

The silicon evolution yields advances in contemporary processor architecture. As a result of the ever-increasing number of components in a chip, multi-core solutions have emerged. In general computing systems, their goal is to accommodate the parallel execution of processes, tasks or threads. Apart from general computing, the parallel execution of tasks is characteristic of asynchronous and dynamic embedded applications like automotive systems, process control, multimedia processing, security systems, etc., for which, in recent times, multi-core architecture has also raised interest [Lee (2010)].

However, in the case of processors for embedded systems and their ultimate requirement being predictability of temporal behaviour, the implementation of traditional multiprocessing is not straightforward. Their advanced architecture features (pipelines, cache, etc.), which are devised to improve average computing speed, may introduce severe sources of non determinism and unpredictability.

Instead of symmetrical multiprocessing, it is more adequate to employ multi-core processors for specialised operations. One of these is the execution of operating system services with a goal to deal with the nondeterministic and unpredictable time delays caused by the very nature of asynchronous events by separating the execution of process control tasks from real time operating system (RTOS) kernel routines. This approach is similar to the idea of math coprocessors, graphical accelerators, intelligent peripherals, etc. These specialized units are able to perform operations much faster than general processors that implement them as software programs.

The idea of migrating scheduling out of the main processor is already old [Halang (1988); Cooling (1993); Lindh et. all. (1998), etc.] However, with the advent of multi-core processors on one hand, and programmable hardware devices for prototyping on the other, its implementation has become much more feasible and realistic. In this contribution we are presenting a prototype of a separate Application specific integrated circuits (ASIC) implemented coprocessor performing operating system kernel functionalities.

First, some background regarding the real-time properties of embedded systems is given, and some of the most characteristic solutions of real-time operating systems which jeopardise predictability are pointed out. Then, an architectural solution to the problem is proposed and validated with the prototype.

2. Real-time properties of the embedded system

An embedded system is a special-purpose computer system designed to control or support the operation of a larger technical system which usually has mechanical and electrical components in which the embedded system is encapsulated. Unlike a general-purpose computer, it only performs a few specific, more or less complex pre-defined tasks. It is expected to function without human interaction and therefore, it usually has sensors and actuators, but no peripheral interfaces like keyboards or monitors, except if the latter are required to operate the embedding system. Often, it functions under real-time constraints, which means that service requests must be handled within pre-defined time intervals.

Embedded systems are composed of hardware and corresponding software parts. The complexity of the hardware ranges from very simple programmable chips (like field programmable gate arrays or FPGAs) over single micro-controller boards to complex distributed computer systems. In simpler cases, the software consists of a single program running in a loop, which is started on power-on, and which responds to certain events in the environment. In more complex cases, operating systems are employed. The application for the embedded system (and the others) usually consists of several processes or tasks that must be executed more or less simultaneously. The operating system (OS) provides features like multitasking and scheduling to allocate the active tasks to limited processing resources by means of different scheduling policies. The OS also provides task synchronisation, resource management, etc. [Silberschatz et. all. (2009)].

The main focus of this paper is the scheduling of processes operating under hard real-time constraints as a basis for other operating system kernel services (event management, synchronisation, etc.). For such systems, the essential and characteristic requirement is that each task, regardless of circumstance, must finish its work prior to the predefined deadline. Here, obviously, task scheduling is the critical operation. Some functionalities of operating systems (e.g., virtual memory, mass storage device management, etc.) are rarely relevant for the embedded system and are not considered here.

Although the discipline of real-time research was established thirty years ago, even now inappropriate scheduling policies (e.g., fixed priority) are very often employed. For singleprocessor systems operating in the real time regime, theoretical aspects of task scheduling have been acknowledged at least since the well-known paper [Liu-Layland (1973)]. The advantage of the often used, although inadequate, fixed priorities-based scheduling is that in most cases, it is built into the processors themselves in the form of priority interrupt handling systems. Thus, implementation is fast and simple. However, it is difficult to assign adequate priorities to tasks, which leads to the starvation of other tasks which are waiting for blocked resources. Usually, priorities are not flexible enough and cannot adapt to the current behaviour of systems. With rate-monotonic scheduling, a set of periodic tasks is considered. In this case, the tasks are scheduled according to their periods. In the paper mentioned above, the scheduling of such a task set is proven to be feasible, however, only if the utilisation of the processor is less than approximately 70%.

It is widely accepted that in a general case, the deadline-driven scheduling policy is the most appropriate, more specifically, the Earliest Deadline First (or EDF). In this case, the priority of the task is determined by its deadline. The task with the closer deadline has higher priority than the task with the more distant one.

When the deadline-driven scheduling is employed, the actual schedule can and should be tested for feasibility during run-time (schedulability check). Each time a new task is added

to the system, a test must assess whether the deadlines of all active tasks will be met. To perform this test, the sum of the (remaining) execution times of each task and the tasks that will be executed before it must be smaller than or equal to its designated deadline. The schedulability check depends on the accurate estimation of the execution times of tasks. To calculate this properly, all aspects of the embedded systems (hardware, operating systems and application) must behave with temporal predictability.

Typical embedded systems are expected to react to events from the environment. Traditionally, this is implemented by means of interrupts that signal the main processor when a specific event occurs. The problem with this method is that the interrupts and interrupts handling also introduce sporadic delays asynchronous to the execution of the running processes, and this jeopardises the temporal predictability of the latter.

Another problem facing the real-time behaviour of embedded applications is the operating system itself. Traditionally, operating systems are software services running on the same processor along with the application software, with the goal to support the application execution on the target hardware systems. Each call of the operating system routines prolongs the execution of the application. It is usually very difficult to get adequate execution times for these routines because the calculation depends on the number of active tasks currently running.

3. Outline of the architectural solution

Embedded applications usually consist of several tasks or processes, and the OS is responsible for scheduling these for execution on devices with limited processing resources. In addition, the OS is responsible for proper task synchronisation, inter-task communication, reaction to events in the system, etc. The reason adequate OS operations for real-time systems are seldom supported is that their implementation is difficult and impractical due to their complexity and often unacceptable overhead. By implementing the scheduling in hardware operating in parallel, complexity is not an issue any more, and the overhead becomes negligible. First, the hardware implementation usually outperforms any software execution. Second, in hardware, many operations can be executed in parallel, further speeding up the execution. In addition, the ever-decreasing cost of hardware devices on one hand, and a steadily increasing degree of integration on the other, justifies the use of a hardware approach even for complex solutions.

The outlook of the hardware architecture is shown in Figure 1. The main processor, where the tasks' code is executed, accesses the operating system services via its system bus. The set of registers implemented within the coprocessor are thus addressable within its memory space, providing communication with the OS kernel functions. Instead of executing the specific function on its own, the OS system routines set the appropriate parameters in the coprocessor's registers and issue a specific command. After that, the OS responses are read from the coprocessor's registers.

Task administration is split into two parts. The internal states (contexts) of tasks are kept at the main processor. The coprocessor only maintains the statuses and the essential parameters of tasks, and determines which task must be executed next. Furthermore, the coprocessor also maintains synchronisers, shared variables, etc., and is responsible for controlled system event management.

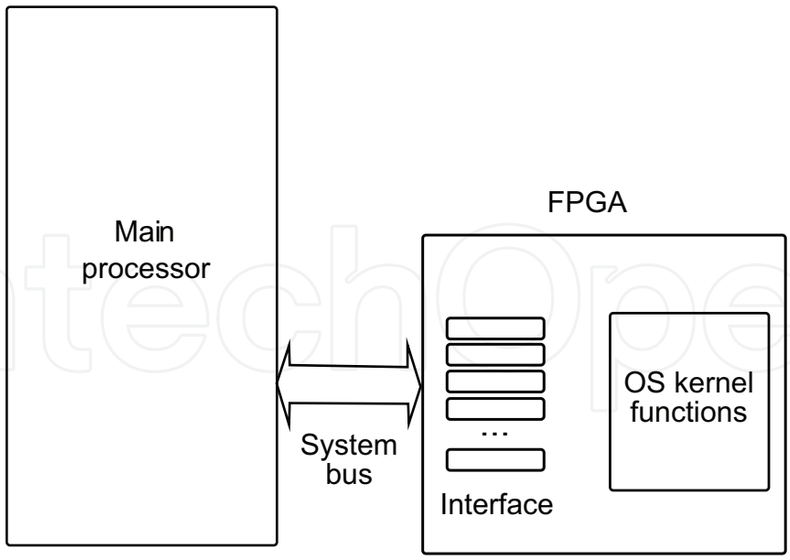


Fig. 1. General scheme of the implementation

4. Coprocessor instruction execution

The coprocessor operates by means of instructions (requests for operations) issued by the main processor (the host). Each instruction consists of an operation code and associated parameters (operands). For example, in the case of task activation, these parameters are the task identification number and the task scheduling constraints. Usually, the instruction is executed immediately after it is put into the interface registers. In addition, the coprocessor can store several instructions for future execution.

Such instructions are triggered by certain conditions that are also set by the host. When these conditions are satisfied, the instruction is issued to the instruction execution unit. There the operation code is decoded and an appropriate set of signals is generated to carry out the required operation. The process of instruction execution and its implementation is presented in Figure 2.

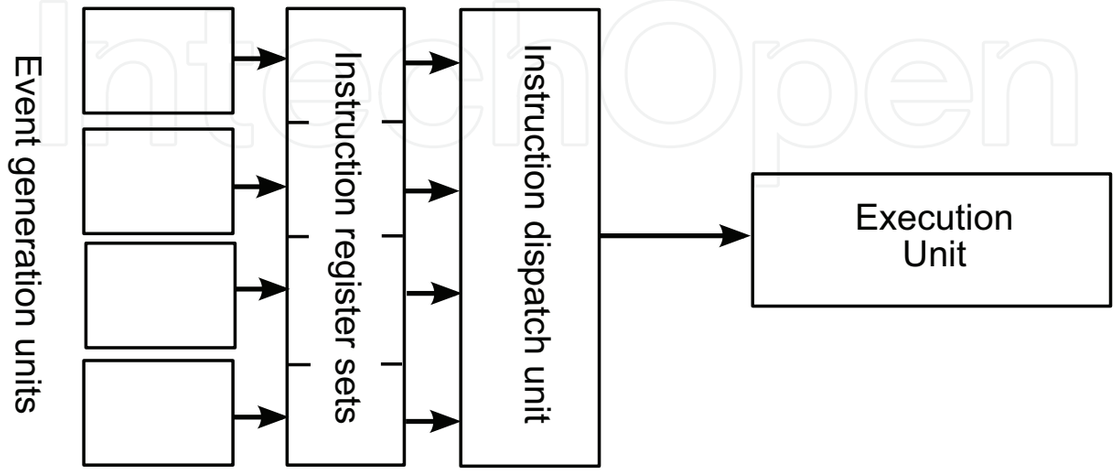


Fig. 2. The instruction execution implementation

Several instruction register sets hold the operation code and parameters of the specific instructions. The instruction dispatch unit is constantly monitoring which of the stored instructions is ready and forwards them further for execution. When the conditions of several instructions are fulfilled at the same time, the instruction register with the lowest number takes precedence.
The readiness of the instruction for execution is determined by one of the event generation units (event generators). These units generate signals when specific conditions are met, and these signals in turn may then initiate the instruction execution. Which event generation unit is connected to a specific instruction register set is determined by the application. Each register set may also be configured for immediate instruction execution. In this case, the instruction is issued immediately when the operation code is placed (i.e. the application first sets the instruction parameters and then triggers the instruction execution by writing the operation code). In this way, parameters for several instructions may be pre-set in advance. For command execution, only the operation code must be set.
There are different kinds of event generation units. The so-called external event reaction unit reacts to the events generated outside of the system. This is similar to interrupt and event handling in traditional microprocessors. The events from different external and internal sources can be combined, and some events may be masked.
Another event generation unit is the periodic event generator. This unit generates signals for periodic instruction execution. The main application may set the time of the instruction's first occurrence, the period of the repetition and the overall duration interval.
An additional unit observes the shared variable registers (described in more detail later). When a new value is written into a shared variable, a signal is generated, which can be used for instruction execution. In addition, the relevance range of the value can be associated with each shared variable separately. In this case, the signal is generated only if the value written into the register is outside of the predetermined range. In this way, different message-passing algorithms for inter-task communication may be implemented, and the system may react to some conditions which are related to the values of some parameter (e.g., temperature too high).
As will be described later, the event generators can also be used as a part of the task synchronization mechanisms. Furthermore, the event generator can be configured to generate interrupts to the main processor.

5. Operating system functions implementation

The execution unit implements specific operating system functionalities. One of the primary goals of this research was to eliminate operating system temporal interference. The main processor should read the results of the operation as soon as the operation is written into the coprocessor. When implemented in software, the execution times of most OS instructions depend on the number of tasks that are currently active. With the appropriate approach it was possible to achieve a constant execution time for each OS instruction, independent of the number of tasks (i.e., time complexity of O(1)). The consequence of this is increased silicon consumption, which is not an issue anymore. There are several groups of OS instructions that were implemented with the coprocessor.

5.1 Task scheduling

For the implementation of task scheduling, a sorted list of tasks is maintained. Each element of the list holds relevant parameters of an active task: the task ID number and the

parameters related to scheduling. For EDF scheduling, the latter would be the deadline of the task, remaining execution time, etc. For other scheduling policies, it would be the priority or period of a task. Each time the task information is added to, or removed from the list, the parameters are updated accordingly. Some parameters are also updated periodically during this time. For example, the remaining execution time of the task that is currently running must be periodically decreased. Other parameters related to task synchronization and other operations, is described in the next sections. Other parameters of tasks, not relevant to the OS coprocessor (e.g., the context of the task), are kept in the main processor.
Data in the list are sorted according to the scheduling policy. In the case of EDF scheduling, this is done based on their increasing deadlines. For priority-based systems, it would depend on priorities. For rate monotonic scheduling, the sorting criterion would be the period of tasks. The list can be observed as a set of independent cells or components with the same functionalities. This is illustrated in Figure 3.

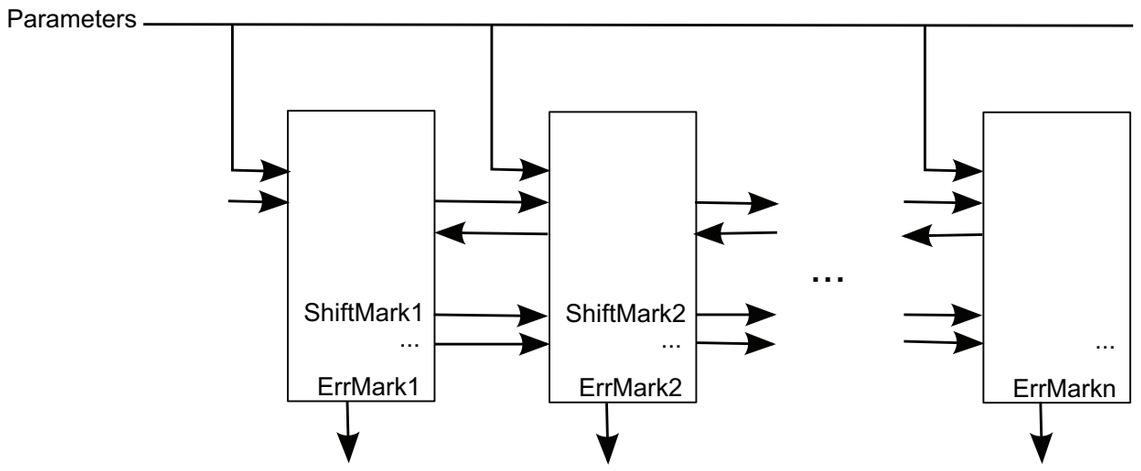


Fig. 3. Implementation of task scheduling

The parameters of the current OS instruction (such as the index of the task, its deadline and remaining execution time) are put on the common bus. Then, a series of control signals (not shown in the picture) are generated to execute different steps of the specific instruction. Each cell contains a set of registers that hold the task's parameters. Apart from the registers, each cell has two sets of inputs and outputs that are used during different phases of some OS instructions. The data from a single cell may be shifted into the next or into the previous cell. Several logical signals are used to synchronize these shift operations (ShiftMark) or to signal if there is a deadline violation or some other error (ErrMark). In addition, each cell consists of digital logic divided into several parts, which are responsible for executing different OS operations. Such division allows for parallel execution during the completion of OS instruction. One part of this digital logic is responsible for the identification of the cell by means of its task ID, the second part is responsible for the deadline comparison, and yet another part performs the arithmetic for cumulative execution time calculation, etc. In this way, it is possible to achieve the same execution time for each instruction. For example, when a new task is being added to the list, a proper position is determined: First, the deadline for the arriving task is compared with the deadlines for all tasks already in the list. Then, room is made by shifting the proper set of cells to the end of the list and finally, the

new task information is put into place. The removal of a task from the list is performed in a similar fashion. A more detailed elaboration of the procedure is given in Verber (2009). A dedicated logic within the cell also signals the current state of the task (i.e., if it is ready for execution or if it is suspended for some reason). In parallel to the list of tasks, an additional component determines which task must be executed next by detecting the first task in the list that is not in a suspended state. This variable may change every time an instruction is executed. An instruction may be executed independently from the main processors by means of instruction register sets and event generators as described above. In this case, when the new task must be dispatched for execution, an interrupt signal to the main processor is generated. The interrupt service routine may access the ID of the executing task through the coprocessor interface. This component also incorporates support for the so-called non-interruptible critical sections. A task, executing in a non-interruptible critical section, cannot be replaced by another one even if there is an active task with a shorter deadline. There are two instructions for entering and leaving such a section.
When using the EDF policy, another important part of task scheduling is the schedulability check. Each time a new task is put into the list, it must be proven that the deadlines of all active tasks will be met. To this end, each element of the list maintains the remaining execution time of the current task, as well as the cumulative execution time of the tasks to be executed prior to and including the current one. For the schedule to be successful, the latter must be smaller than or equal to the designated deadline of the current task. To maintain the sum of execution times, when a task is added to the list, its execution time is added to all cumulative execution times of the elements which come after the newly arrived one. Similarly, when the task is removed, its remaining execution time is subtracted from sums in subsequent elements of the list.

5.2 Task synchronisation

Occasionally, an active task may not be in a position to continue the execution due to the unavailability of exclusive resources, because it must wait for another task to complete its job, etc. In such cases, the OS puts the task in a suspended state. When, for example, the exclusive resource becomes available, one of the tasks waiting for it is removed from the suspended state. The main difficulty in this execution is that some sort of queue of waiting tasks must be implemented. This is easily done in the software, however, maintaining several queues in hardware would consume too many resources. Instead, each element of the sorted list contains a set of bits that represent the various synchronisers for which the current task is waiting. These bits are shifted together with every task addition/removal operation and are maintained by the synchroniser control circuits. Each sychronisation unit can be associated with the specific synchronisation bit in each cell. The task is suspended if either of these synchronisation bits in the cell is set to one. This is achieved with a simple logical operation of disjunction (or). Different synhronisation control units can be used to implement different synchronization mechanisms. In these experiments, the binary semaphore primitives Lock and Unlock have been implemented. When a Lock instruction is executed for a semaphore for a certain task, the control unit checks to see if the semaphore is already locked. If it is, a corresponding bit in the cell is set and the task becomes suspended. In other cases, if the semaphore is unlocked, the control logic marks it locked and the task remains non suspended. Upon the Unlock operation, when several tasks are waiting for the

same synchroniser, the left-most one in the list becomes ready. In the case of the EDF scheduling algorithm, this is the task with the shortest deadline. In this way, the possibility of deadline violation is minimized. The binary semaphore is implemented with simple flip-flop logic. The control logic for other synchronization mechanisms may be easily implemented.

5.3 Inter task communication

To serialize the data-dependent operation between tasks or to employ inter-task communication in general, a set of common shared variables is used. A value, written into a shared variable by one task, may be read by the others. Using the common shared variables, tasks may also be synchronised. For example, one task is waiting until another one changes a value of a variable. This can be implemented by combining the synchronization control logic with shared variable event generators. The same method is used for the implementation of traditional OS signals. Shared variables are mapped into the memory space of the main processor. The shared variables may also have a very important role in distributed embedded systems. In a previous research [Colnaric and Verber (2004)], the hardware support for transparent interprocessor communication in distributed environments was studied and implemented. In order to accomplish this, a new value's contents, when put into the shared variable, is distributed (replicated) to the other nodes in the system. In this way, inter-task communication and synchronization may be implemented between tasks running on different processors. In the current work, those mechanisms are not yet implemented.

5.4 Real-time clock

Although a typical processor may have implemented a real-time clock by other means, its integration into the kernel coprocessor may allow other operations to use and react to the same absolute time source. However, a proper real-time clock must operate even when the system is switched off. This requires battery-powered circuits. Currently, it is not possible to put part of an FPGA device into an operational state during the shutdown of the system Therefore, it is the responsibility of the main processor to set the proper time of the real-time clock at startup. Implementation of the reading and maintenance of the precise real-time clock by means of a dedicated battery-powered real-time clock chip is under development. For an even more precise clock source, the use of a GPS receiver may be considered.

5.5 Support for fault tolerance

Apart from operating within real-time constraints, the embedded systems are frequently used in situations where faults may result in large material losses or even the endangerment of human safety. There are several aspects of fault tolerance that may be incorporated into the coprocessor. For example, in the case of event generators, different self-monitoring circuits may be implemented in hardware in order to detect hardware-related faults. The event generators related to the shared variables can be used to detect abnormal values of a certain variable. The task scheduler is also capable of detecting deadline violation errors. However, for more subtle fault detection and fault management, the coprocessor is usually not adequate. If a fault is detected, a contingency plan must be employed and a new set of tasks should usually be introduced. This can only be done by the main processor.

6. Results of the experiments

To support the theoretical research, studies on an experimental hardware platform were conducted. The main processor is Texas Instruments' digital signal processor TMS320C6771 running at 150 Mhz [Texas Instruments (2010)]. The coprocessor is implemented with Xilinx FPGA device Spartan2E xc2s300e running at 50 Mhz [Xilinx (2010)]. This device consists of 1536 so-called Configurable Logic Blocks (CLBs). Each CLB is capable of performing simple logical functions and/or to be used as a memory element. This is a relatively low performance and low-cost device. In the experiments, four event generators, four synchronisers, eight shared variables and eight task scheduling cells were implemented. By this method, approximately half of the available silicon resources were used. Another half, it is planned, will be used in future work. The newest FPGA devices and dedicated ASIC chips may have hundreds of times more silicon resources and are much faster. On the main processor the artificial tasks were used for the test bed. The tasks were created with a proprietary realtime operating system on the evaluation board. Nevertheless, the OS operations were issued through the coprocessor. The task IDs and the operation codes are one byte in size. All other parameters require 16 bits. All temporal values and constraints are represented in a relative fashion (i.e., as a number of basic clock cycles relative to the current moment in time).

Each instruction is executed in four basic clock cycles. This is 80 ns at 50 Mhz. Some instructions could be executed in fewer clock cycles, however, we found that it is much easier to implement the instruction execution unit if the same four cycles are used every time. For simple instructions during some execution cycles, the instruction execution unit is idle. In any case, the execution time of a single instruction is shorter than the memory access time of the main processor. I.e., the main processor may read the results as soon as the operation code is provided.

7. Conclusion

With the ever-increasing density of silicon chips, it is possible to dedicate some areas on the chip to the implementation of operating system functionality. The situation is similar to that of the early 1990s. In the beginning, floating-point operations were implemented in software, then math coprocessors replaced software routines and execution times shrunk to only a small portion of their original. Later on, with miniaturization, the coprocessors were integrated into the processor cores. In the research described here, it was shown that the same scenario may be applied to the implementation of OS functionalities. If consumption of silicon is not an issue, the functionalities of the coprocessor may be executed in constant time (i.e., with O(1) time complexity). Hardware implementation of the operating system's functionalities has little impact on application development. Within traditional development tools, the OS support is usually considered on the application programming interface (API) level. If only the inner parts of the API to OS routines are modified, no change in the development tools is required.

Although the number of components in the experiment were limited, the proposed implementation is modular enough to be easily expanded and modified to manage a different number of tasks, synchronisers, shared variables, etc.

The main focus of our research is real-time systems. However, the principle of the OS coprocessor can be effectively used with any kind of operating system. For example,

priority-based scheduling can be implemented much easier than EDF. In this case, the number of parameters in each element of the task list is greatly reduced and there is no need for the feasibility check of the schedules. The synchronisation mechanisms, shared variables and other elements of the coprocessor may remain the same.

8. References

Edward Lee (2010). Design Challenges for Cyber Physical Systems, In: *Strategies for Embedded Computing Research*, Eutema, Vienna, March 2010.

Halang,W. A. (1988). *Parallel Administration of Events in Real-Time Systems*, *Microprocessing and Microprogramming*, Vol. 24, Jan 1988, pp. 678 – 692, North–Holland.

Cooling, J. (1993). Task Scheduler for Hard Real-Time Embedded Systems, *Proceedings of Int'l Workshop on Systems Engineering for Real-Time Applications*, pp. 196–201, Cirencester, 1993, IEE, London.

Lindh, L., Strner, J., Furuns, J., Adomat, J. and Shobaki M. E. (1998), Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems, *In Seventh Swedish Workshop on Computer Systems Architecture*, Sweden, 1998.

Silberschatz, A., Galvin, P. B., Gagne, G. (2009). Operating System Concepts. Wiley, 2009.

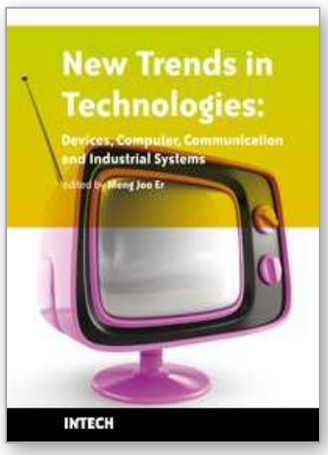
Liu, C.L. and Layland, J.W. (1973). *Scheduling algorithms for multiprogramming in a hard real-time environment*, *Journal of the ACM*, Vol. 20, No. 1, 1973, pp. 46–61, IEE, London.

Verber, D. (2009). Attaining EDF Task Scheduling with O(1) Time Complexity *Proceedings of 7th Workshop on Advanced Control and Diagnosis*, November 2009, Zielona G´ ora, Poland.

Colnarič, M., Verber, D. (2004). Communication infrastructure for IFATIS distributed embedded control application *PRTN 2004 : proceedings of 3rd intl. workshop on real-time networks*, pp. 7-10. June 2004, University of Catania, Italy.

Texas Instruments (2010). *http://www.ti.com*.

Xilinx (2010). *http://www.xilinx.com.*



New Trends in Technologies: Devices, Computer, Communication and Industrial Systems
Edited by Meng Joo Er

ISBN 978-953-307-212-8
Hard cover, 444 pages
Publisher Sciyo
Published online 02, November, 2010
Published in print edition November, 2010

The grandest accomplishments of engineering took place in the twentieth century. The widespread development and distribution of electricity and clean water, automobiles and airplanes, radio and television, spacecraft and lasers, antibiotics and medical imaging, computers and the Internet are just some of the highlights from a century in which engineering revolutionized and improved virtually every aspect of human life. In this book, the authors provide a glimpse of new trends in technologies pertaining to devices, computers, communications and industrial systems.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Domen Verber and Matjaž Colnarič (2010). Operating System Kernel Coprocessor for Embedded Applications, New Trends in Technologies: Devices, Computer, Communication and Industrial Systems, Meng Joo Er (Ed.), ISBN: 978-953-307-212-8, InTech, Available from: http://www.intechopen.com/books/new-trends-in-technologies--devices--computer--communication-and-industrial-systems/operating-system-kernel-coprocessor-for-embedded-applications

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen