TELKOMNIKA, Vol. 11, No. 10, October 2013, pp. 5763 ~ 5773
ISSN: 2302-4046

Optimization Design of a Real-time Embedded Operating System Based on ISO17356

Jiang Chun-Mao*[1], Qu Ming-Cheng[2], Wu Xiang-Hu Hu[2]
[1]School of Computer Science Technology and Information Engineering, Harbin Normal University
Harbin 150025, Heilongjiang, China.
[2]School of Computer Science and Technology, Harbin Institute of Technology
Harbin 150001, Heilongjiang, China
*Corresponding author, e-mail: 780319467@163.com

Abstract

This paper distinguishes the differences between dynamic and static operating system from the design target, and discusses the resulting different technologies used when system is implemented. Based on this theory, according to ISO17356 OS specification and using the hardware abstraction layer idea, a simple and efficient static real-time operating system is implemented for ARM architecture, and the kernel contains no more than 3000 lines C codes. With the idea of separating mechanisms and strategies, the internal resources are used as a mechanism to implement different scheduling policies; proposes a priority-based packet scheduling algorithm, which achieves a good performance both in time and space.

Keywords*: ISO17356 OS, Real-Time Embedded Operating System, Static*

1. Introduction

With the advancement in networking and multimedia technologies enables the distribution and sharing of multimedia content widely. Although encryption can provide multimedia content once a piece of digital content is decrypted, the dishonest customer can redistribute it arbitrarily [2, 3]. In the field of automotive electronic, there are always a lot of duplication of development work for different products, which not only reduces the efficiency of software development, but also makes the system extensibility and portability affected. To address these issues, the European automotive industry in the mid-90s of 20th century developed an open system interface OSEK / VDX specification [1]. The specification has now become an industry standard for embedded real-time systems, and was adopted by ISO as an international standard--ISO17356. To be convenient for the narrative, the following OSEK standard is equivalent to ISO17356 standard [2].

In the previous studying on the OSEK operating system, they mostly only concerned with how to implement the system, but didn't distinguish the differences between dynamic and static operating system from design idea. This paper first analyses the different requirements for embedded system in different application environments, to present the static and dynamic operating system concept; from the different design objectives of the two types, present different techniques used when implemente them. OSEK operating systems consistent with the characteristics of the static operating system, therefore, we use static design approach, discuss the design objectives of each part by top-down method, and discuss the task management, resource management and interrupt management in detail [3].

2. Dynamic Operating System and Static Operating System

Embedded operating system application environments are diverse, and different applications have different requirements for the services provided by the system. In some circumstances, users want the operating system to provide rich functionality as much possible to meet the diverse requirements of the applications; and in other high reliability and security areas, users pay more attention to the predictability and real-time of the operating system.

Received April 19, 2013; Revised July 2, 2013; Accepted July 15, 2013

Therefore, there are in fact mainly two types of embedded operating system: static operating system and dynamic operating system [4].

Dynamic operating system gives top priority to flexibility and adaptability to pursue to meet the various requirements of application as much as possible. To support this flexibility, dynamic operating system must have the ability to dynamically allocate resources, such as dynamic creation and deletion of tasks, dynamic application and releasing of semaphores and so on. Som typical dynamic operating systems are Vxworks, pSOS, uCLinux and so on [5-6].

The design objective of static operating system is predictability, and the pursuit of the optimal performance. To achieve those objectives, it requires the user to configure all objects statically before the system starts, such as the number of tasks and timers and so on, which produced a "static operating system" concept. Currently, OSEK is a typical representative of the static operating system [8-10].

2.1. System Design

In order to meet flexibility, the dynamic operating system will provide a very rich APIs for users. But the static operating system must guarantee that the APIs provided does not affect the predictability of the system, so the restrictions on the number of APIs and the ways they are used will be made.

2.1.1. Task Management

In a dynamic operating system, applications must be able to dynamically create and delete tasks, so the kernel must provide mechanism for dynamic memory management. The disadvantage of such mechanism is that: the task is not guaranteed to be created successfully. Because it is difficult to accurately estimate the total amount of required memory, when system memory runs out, the requirements of creating tasks will be delayed to meet or even ignored. Clearly, in high reliability and high security environment, this approach is unacceptable [7].

To solve this problem, the system will know in advance the creation of the tasks the user requests. Static operating system uses a method of static configurations, configuring all tasks and all resources tasks require for the system before starting. In this way, users will be able to very accurately assess the demand of applications for memory, and also ensure the request of creating tasks will not fail. At the same time, this approach also avoids the overhead of time caused by dynamic creation, and improves the system's real-time. Because the tasks in the system are implicitly created before starting, after startup they have been in existence, the static operating system only has system services of "active tasks" , and no "create a task". "Activation" is only the process of initializing the task control block based on the configuration information of tasks.

In many cases, users want to advance or delay the implementation of a task, which is required to dynamically change the priority of the tasks in the applications. Therefore, in the dynamic operating system, it must provide such APIs. However, this approach will produce uncertainty in applications, execution order of tasks and the scheduling time of takes can not be predicted. Therefore, in the static operating system, once the task priority is configured, it's not allowed to change dynamically during operation.

In many application environments, it often needs more than one task to work together, so any operating system must provide means for synchronization between tasks.

In pursuit of flexibility, the dynamic operating system provides a variety of IPC mechanisms such as semaphores, mutexes, message queues, mailboxes and so on. The common feature of these synchronization mechanisms is that when the task can not enter into the critical section, it changes to sleep state until they are awakened. Take the semaphore as an example, when a semaphore is created, any task that wants to access the critical section may apply to use the semaphore, if the semaphore is being occupied, the task changes into the sleep state. This brings two problems: (1) When more than one task is blocked because of waiting for the same semaphore, the system can not be sure that how long each task will be blocked, thus increasing the unpredictability of system behavior; (2) a high priority task may be forced to block some uncertain time to wait for a low-priority task to be completed, that is, the priority inversion problem. Clearly, in high reliability, real-time systems, thus unpredictable blocking is not acceptable.

To avoid these problems, in a static operating system, it can not use the IPC mechanism of "block - sleep - wake up". To accurately predict the time the task will be blocked,

it must know in advance the number of tasks to access the critical region. Obviously, this can only be achieved in the method of static configuring. As for the priority inversion problem, it mainly uses Priority Ceiling Protocol or Priority Inheritance Protocol to resolve, which is what OSEK standard is doing.

OSEK standard provides "resources" as a means of synchronization among tasks. Resources are defined as follows: The task gets resources before entering the critical area and releases resources after leaving the critical area, the specific mechanism is as follows: (1) through making clear the ownership of the specified resources statically before the system starts, the kernel knows which tasks will access which critical areas; (2) use Priority Ceiling Protocol to avoid the occurrence of priority inversion. When the task obtains resources, its priority will be raised to the Priority Ceiling Protocol resources priority, other tasks can not enter the critical areas naturally; when the task releases resources, its priority gets back to the original priority, then the other high priority tasks can preempts. With the guarantee of Priority Ceiling Protocol, a task in running state can always get a successful access to resources, and access to critical areas. Obviously, it can easily predict the execution order of tasks in this way.

To further enhance the ability of communication between tasks, OSEK standard also provides a "event" mechanism. With the target of not affecting the predictability of the system, the mechanism of events is designed to follow two principles: (1) it must specify the ownership of the event; (2) the task can only wait for its own event. After the owner of the event and waiting mode are specified, the waiting time of task is controllable.

In the life cycle a task has a variety of features, according to the characteristics, a task can be divided into different states. Therefore, all possible actions that the task has in the system determine the set of status of the task. In a multitasking operating system, CPU can only handle a task at any moment, so a task requires at least three states:

(1) pending: the task has not been loaded into memory, do not have any running conditions;
(2) running: the task possesses CPU, and execute its instructions;
(3) ready: In addition to CPU, the other conditions are all ready, only wait for scheduling;

Different operating systems on the basis of these three states will make the necessary expansion according to the different behaviors of tasks. From the previous section it can be seen that the dynamic operating system provides a wealth of IPC mechanisms in order to meet the various requirements of the applications, and most of these IPC mechanisms use a "block - sleep – wake up" approach. To support this tasks behavior under IPC mechanism, the operating system must add a "sleep" state for the tasks.

As discussed in the previous section, tasks in "sleep" state can not be predicted when they will be awakened, this will lead to uncertainty in system behavior. So in a static operating system, there is no "sleep" state for the tasks.

In order to support the event mechanism, OSEK standard needs to be added a "wait" state for tasks. However, this "waiting" is different from "Sleep": tasks in "wait" status can be waked up at any time, the waiting time is controllable, so it will not affect the system's predictability.

2.1.2. Memory Management

In a dynamic operating system, to support the dynamic creation and deletion of various objects, it must provide dynamic memory management mechanism. In addition, as the kernel does not know the size of the stack required for each task, it will be demanded to maintain a memory pool to meet the various needs of the task stack.

All objects of a static operating system are statically configured, the size of task stack is also static pre-allocation, so memory management is also very simple, without complex dynamic memory management mechanism.

2.2. Performance Comparison

2.2.1. Space comparison

Dynamic operating system has dynamic memory management mechanism and a large number of APIs, this makes the internal control mechanism be complicated and difficult to maintain, and directly resulting in the large quantity of kernel codes. In addition, the dynamic memory management mechanism has high memory requirements, at the same time, it will also bring about memory fragmentation.

Static operating system's internal management mechanism is very simple, the internal data structures and functions can be designed to be compact, so the number of kernel codes is very small. As a result of a static configuration, on the one hand it can store more global variables in the ROM, reducing the demand for RAM to reduce hardware costs; on the other hand, the size of each task stack can be configured according to the applications need, thus avoiding memory waste.

Static operating system obtains information of the applications to the maximum extents, it configures the kernel according to specific needs, so it can be done on the optimal use of space.

2.2.2. Real-time

Real-time performance of the system is mainly reflected in two aspects: (1) the response time of the request for resources; (2) the time of scheduling.

In a dynamic operating system, the request for resources will go through the process of dynamica allocation; when the memory is exhausted, this request will be delayed, which will decrease real-time. In the static operating system, all objects are statically configured, they exist throughout the process of system running, there is no need for the process of " allocation "; so every request for resources will be able to be immediately responded, which has high real-time.

Dynamic operating system allows applications to change priority in the process of running, which makes scheduling algorithm more complicated, longer time required for scheduling and unpredictable. In a static operating system, it doesn't dynamically change priorities, so the scheduling algorithm is simple, and scheduling time is short.

Therefore, the static operating system has higher real-time than the dynamic operating system.

2.2.3. Verifiability

Because the kernel of dynamic operating system is complex, it needs dynamic memory management mechanism, which will cause the system difficult to maintain. In application development based on dynamic operating system, all objects go through the dynamic creation and deletion, system behavior is difficult to predict, so it is very difficult to test the system completely.

In contrast, the kernel of static operating system is simpler and easier to maintain. Since all objects are statically defined before the system starts, it will be able to calculate the demand on hardware resources, it is easy to verify. Because the system has good predictability, the tests of the application can also be done through limited test cases to cover all possible situations.

In summary, the overall performance of the static operating system is better than the dynamic operating system, thi is why the article chooses OSEK standard to implement a static operating system.

3. OSEK Standard Profiles

OSEK standard specifies four conformance classes, a conformance class can be implemented as a operating system of specific version. In fact, the four conformance classes are upward compatible: any applications developed for the BCCx operating system can be transplanted without modification to the ECCx operating system, any applications developed for the xCC1 operating system can be transplanted without modification to the xCC2 operating system as Figure 1.

3.1. Task Management

OSEK specification divides tasks into basic tasks and extended tasks. Basic tasks have 3 states: running state, ready state and suspended state, extended tasks have one more waiting state, as Figure 2.

OSEK specification supports four scheduling strategies, preemptive scheduling, non-preemptive scheduling, mixed preemptive scheduling and group scheduling.

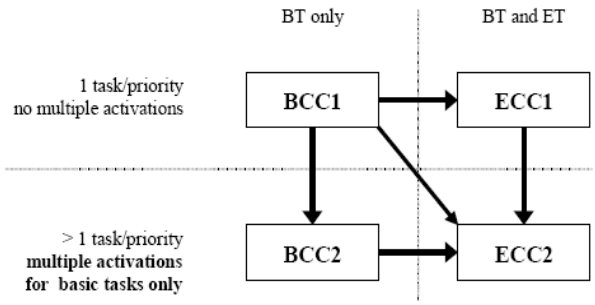

Figure 1. Restricted upward compatibility for conformance classes

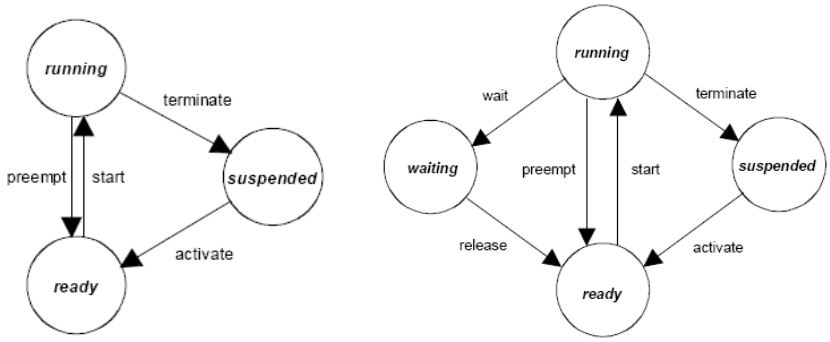

Figure 2. State transition diagram of task

3.2. Event Mechanism

Events as a means of communication among tasks, are used to ensure synchronization between different extended tasks. This mechanism means: an extended task in a wait state, when at least one event it's waiting occurs, it will enter into the ready state. Only extended tasks have events.

3.3. Resources Management

When multiple tasks access shared resources at the same time, resource management mechanisms are needed to protect critical areas. OSEK specification uses the Priority Ceiling Protocol, that is, after a task occupies a resource, the priority of the current process will temporarily be increased to the priority of the resource; when the task releases the resource, its priority goes back to the original priority of the task.

3.4. Interrupt Handling

OSEK specification defines two kinds of interrupt service routine: (1) First Class Interrupt: This interrupt routine doesn't use operating system services, after interrupt ends, the interrupt handler continues from the place where interrupt takes place, which does not affect the task management. (2) Second Class Interrupt: This interrupt routine is formed through the configuration from the user subroutine when the system is generated, it can call the operating system API, and impact task scheduling.

3.5. Alarm Mechanism

Alarm is a service mechanism provided by OSEK specification to address cycle event, to some extent it can be seen as a timer in other operating systems. When the count of the alarm value equals to the set value, the system will call the service program to set up events or activation tasks.

4. The Operating System Design

4.1. The Overall Structure of the Operating System

To facilitate the system migration, we adopted the design idea of hardware abstraction layer, the hardware-related and hardware-independent part are strictly divided. The overall design is divided into two parts:

1. Operating system kernel: hardware-independent part is the core of the operating system, which provides users with a variety of services and is responsible for tasks, resources, events, counters, alarms and the interrupt service routine management.

2. Hardware Abstraction Layer: hardware-related part is the basis for operating system kernel to run on specific hardware platform, which includes the system start, the task context switching, the bottom handling of interrupt and clock.

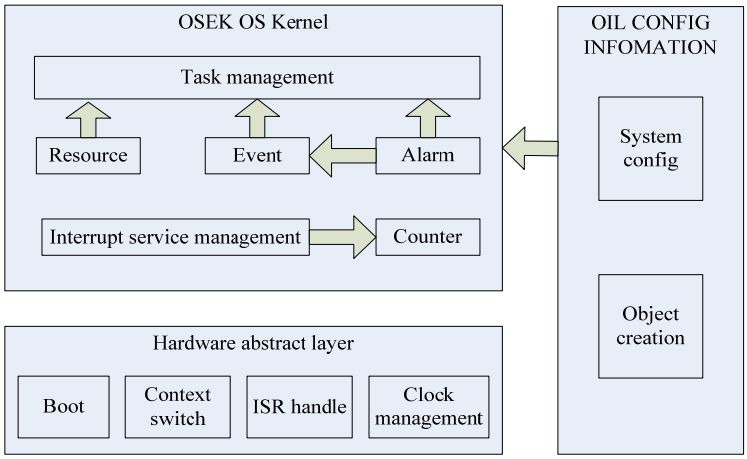

Figure 3. The overall structure of the operating system

4.2. Interface with OIL

Any operating system should have input, for OSEK operating system, its input is OIL file. OSEK specification does not define the interface with the OIL and OS, and we are in point of view of the needs of the OS input, OIL file will be divided into two parts: (1) system configuration information, including system boot parameters; (2) Object creation, used to configure the details of each object. See Figure 3.

For the system configuration information, it must contain the sum of each object, the operating system determines how much memory is needed through the use of these parameters to avoid unnecessary waste.

There are two main methods for object creation: direct and indirect creation. The advantage of direct creation: The control block objects on the operating system are directly assigned, operation is fast; Drawback is that: the coupling of the OS and OIL is too high, which does not meet the modular design. Regard to the future possible demand for expansion of OS and OIL, and OS internal data security, we use indirect methods to create: OIL stores information for each object separately, and then creates the object with OS functions, do not need to care about relationship of each object in the OS.

5. Design of Task Management

Task management is the core of the operating system, mainly faced two problems: (1) the effective division of scheduling policies; (2) balance for scheduling algorithm in time and space performance.

5.1. Separation of Scheduling Mechanisms and Policies

OSEK specification supports four scheduling strategies, preemptive scheduling, non-preemptive scheduling, mixed preemptive scheduling and group scheduling. In this regard, we use the thought of separation of mechanism and policy: the internal resources of the operating

system as a mechanism, users use internal resources to achieve a variety of scheduling strategies, within the system, it does not distinguish what kinds of scheduling policies. When the task starts running, it automatically accesses to internal resources; when the task stops, it automatically releasee internal resources. For the management of internal resources, followes the priority ceiling protocol, see the section of the design for resource management.

(1) Preemptive scheduling: directly scheduling by task priority;
(2) Non-preemptive scheduling: if task runs, it has the highest priority to the internal resources to ensure that the process is not preempted by other tasks;
(3) Mixed preemptive scheduling: According to (1) and (2) to handle tasks;
(4) Group scheduling: if task is running, it has the highest priority of the internal resources of the group to ensure that no preemption by the same group of tasks, but can be preempted by tasks in other groups with higher priority.

5.2. Scheduling Algorithm Based on Priority and Group

Task scheduling algorithm is the core algorithm of the operating system, which is related to the overall system performance. In order to achieve simply, and to achieve a balanced objective in time and space, we propose a priority-based group scheduling algorithms:

Let highest task priority be N, from small to large divide them into P groups, each group has N/P priorities

(1) For one time explicitly scheduling, find the highest priority in the P groups from high to low, the average cost is P/2;
(2) when activating a task, compare the priority of the task activated with the highest priority of the group, and then update the highest priority of the group. The cost is 1 time of compare operation;
(3) when terminating a task, update the highest priority of the group, in the group find the highest priority from high to low, the average cost is N/2P;

In summary, the total cost of system scheduling is P/2 + 1 + N/2P, when P=N1/2, the total cost is minimum.

μC/OS-II is an excellent operating system in the field of embedded systems, especially its scheduling algorithms is unique. We compare the performance of the above scheduling algorithm with the μC / OS-II scheduling algorithm. Because μC / OS-II supports only 64 fixed priorities, so in the above model, N = 64, P = 8; the results are shown in Table 1.

Table 1. Performance comparison between C-OSEK and μC / OS-II for scheduling

Categories	C-OSEK	μC/OS-II
The time for one explicit scheduling	4 times searches in average	2 times searches
Space	a array of 256 bytes is needed for bitmap mapping	No extra space is needed
Flexibility	The number of priorities can be greater than 64, and the smaller the highest priority of task is, the faster the scheduling process will be.	The number of priority is fixed to 64, and regardless of the highest priority of tasks, the scheduling time is constant

From Table 1 we can see that, compared with μC/OS-II scheduling algorithm, C-OSEK scheduling algorithm eliminates the need for a lot of space costs, and has a high flexibility, which needs very little amount of time costs. As the number of the priorities used in practical applications is generally no more than 64, the C-OSEK scheduling algorithm performance is very good.

6. Design of Resource Management

Operating system must guarantee exclusive access to the critical areas, a good exclusive method should have the four characteristics:

(1) two tasks can not occupy the same resources;
(2) priority inversion can not appear;
(3) can not deadlock;
(4) task that has possession of resources can not enter a wait state.

OSEK standard uses external resources as a means to provide mutual exclusion, it uses Priority Ceiling Protocol to address these four issues, specifically as follows:

(1) when the task enters the critical zone for resources, its priority will be raised to the ceiling priority of the resources to ensure critical areas are not seize by the other tasks that have the resources;
(2) when the task releases resources to leave the critical region, its priority goes back to the original priority, other tasks that have the resources can enter the critical area;

Priority ceiling protocol with respect to semaphores and other mutual exclusion locks has the advantages of avoiding deadlock and priority inversion, which switches the mutual exclusion problem into the task priority level problem, so that the problem be simplified.

Access to critical areas not only exists between tasks, but also between a large number of tasks and interrupts as well as between the interrupts and interrupts. Therefore, we extend Priority Ceiling Protocol to interrupt level, the specific methods are as follows:

Let resource R has been occupied by $T_1, T_2, ...T_n$,n tasks and $I_1, I_2, ...I_m$, m interrupts.

(1) when the task $T_j (1 \le j \le n)$ obtains resource R, its priority will be raised to the highest priority of all tasks,mean while banning all interrupts that have the resource;
(2) When the interrupt $I_k (1 \le k \le m)$ obtains resource R, it prohibits all other interrupts that have the resource;

7. Design of Interrupt Management

Interrupt is an important factor affecting the performance of embedded systems. Some of the interrupts are independent relatively to the operating system, which will not affect the core behavior, OSEK standard calls them First Interrupts; in some other interrupts, the user may make some impact on the operation of scheduling, such as activating task, set time, such interrupts should be perceived by the system (RTOS-aware), OSEK standard calls them Second Interrupts.

First Interrupts are mainly used for work that needs to be done quickly, which do not interact with the operating system, the cost of them is minimum; Second Interrupts need to interact with the operating system using APIs provided by the operating system. Therefore, the system should be designed to achieve two objectives:

(1) Interrupts can not be interrupted by Second Interrupts to ensure they are executed as soon as possible, but they can be interrupted by a higher priority First Interrupt;
(2) Second Interrupts can be interrupted by First Interrupts, at the same time they can be interrupted by a higher priority Second Interrupt;

Based on the above two objectives, we design as follows:

(1) The lowest priority of all First Interrupts $\ge$ the highest priority of Second Interrupts;
(2) Second Interrupts need to use EnterISR and LeaveISR at the beginning and end of interrupts to inform the operating system of the arrival of interrupt;

Interrupt context saving and restoring is another key for designing. There are two general methods: (1) store interrupt context in the current task stack; (2) use a single interrupt stack. If we store interrupt context in the current task stack, we need to increase the size of task stack. Because users can not estimate the arrival time of interrupt, when the interrupt frequency is not high the memory will be wasted, when the interrupt frequency is too high, the stack will overflow. Therefore, we use a single interrupt stack to save the interrupt context, thus not only ensuring safety, but also improving the memory utilization rate.

C-OSEK supports interrupt nesting, which makes the processor switch to another mode between the interrupt handlers and opens the interrupt, as shown in Figure 4.

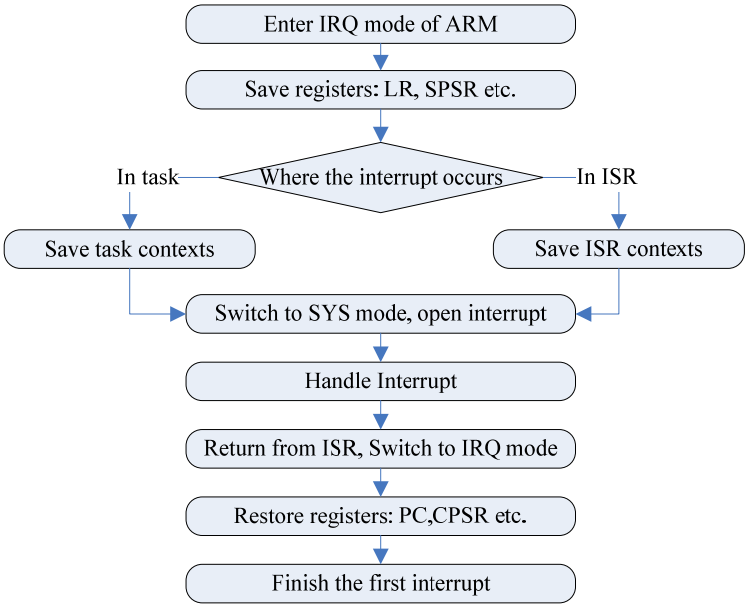

Figure 4. ARM-based interrupt handling process which can be nested

8. Other Parts

8.1. Alarm Management

Alarm depends on the counter, the counter depends on the clock interrupt. As the alarm will activate task or set the event, which will impact on task scheduling, so the clock interrupt is treated as a two types of interrupts. Every time when the clock interrupt arrives, the counting values of counter will all increase, followed by checking whether the alarm on the counter is due and performs the appropriate action. To speed up the processing of alarm, the alarm queues from small to large by triggering time. Counter is used to describe counter, ACB is used to describe alarm, as shown in Figure 5.

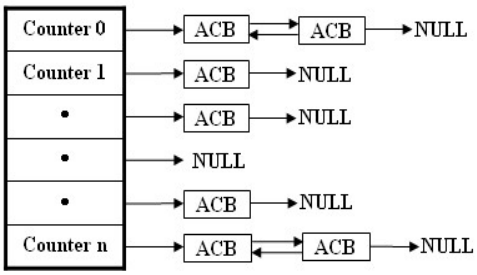

Figure 5. Alarm and counter

8.2. Event Management

C-OSEK supports for each task up to 64 events. Task control block uses three attributes allEvents, eventWaiting and eventState, to indicate all the events task has and events it's waiting for and the current state.

9. Application Verification and Performance Analysis

We choose ISO17356 OS version 2.2.3 specification and ISO17356 OIL version 2.5 specification, in the AT91SAM7X256 of ARM hardware platform we achieve a C-OSEK

operating system, and make a complete functional test. Next, the system will be applied to a color digital instrumentation system of Harbin Weidi Motor Company of China and Germany.

As in-depth analysis of the characteristics of the static operating system, the task management mechanism design is very simple. C-OSEK kernel code is less than 3000 lines, in 32-bit ARM platforms it's only 3.5K after being compiled. Here is the comparison with some of the currently influential OSEK product in the kernel size, as shown in Table 2.

Table 2. Size comparison for various OSEK kernels

Name of OSEK operating system	Kernel size
Emerald-OSEK(University of Michigan)	5.5 KB
OSEKTurbo(Metrowerks Company)	5 KB
P-OSEK(Integrated System)	2.8 KB
SmartOSEK(Zhejiang University)	4.3 KB
C-OSEK(Harbin Institute of Technology)	3.5 KB

The size of the object control block is an important factor to determine the memory usage size of the system as well as an important indicator to measure if the system design is simple. The following compares with the size of the object control block of Tsinghua OSEK, as shown in Table 3.

Table 3. Size comparision between C-OSEK and TH-OSEK for the object control block

Object	Size(Bytes) TH-OSEK	Size(Bytes) C-OSEK
Task control block	55	43
Resource control block	24	23
Counter control block	20	24
Alarm control block	36	30

From Table 3, it can be seen that C-OSEK kernel is small, the design of the object control block is more compact.

10. Conclusion

This paper analyzes deeply the static and dynamic operating system differences. We designed and implemented C-OSEK operating system based on them. C-OSEK is a real-time embedded operating system of completely independent intellectual property rights, which will play a certain role in the development of China's auto industry. Next, we will conduct study for the AUTOSAR standards integration.

Acknowledgements

This work was supported by Heilongjiang province natural science foundation F201139.

References

[1] Baynes K, et al. The performance and energy consumption of embedded real-time operating systems. Computers. *IEEE Transactions on*. 2003; 52(11): 1454-1469.
[2] Leslie, IM, et al. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 1996; 14(7): 1280-1297.
[3] John A Stankovic, R Rajkumar. Real-Time Operating Systems. *Real-Time Systems.* 2009; 28(3): 237-253.
[4] Sun Bao-Min, Sun Xiao-Min. Design of TH-OSEK embedded RTOS. *Computer Engineering and Design*. 2004; 25(5): 661-664.
[5] Zhao MD, et al. SmartOSEK: A real-time operating system for automotive electronics. *Embedded Software and Systems*. 2005; 36(5): 437-442.

[6] Steiger C, H Walder, and M Platzner.Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*. 2004; 53(11): 1393-1407.
[7] Santos RM, J Santos, and JD Orozco. Power saving and fault-tolerance in real-time critical embedded systems. *Journal of Systems Architecture*. 2009; 55(2): 90-101.
[8] Barbalace A, et al. Performance Comparison of EPICS IOC and MARTe in a Hard Real-Time Control Application. *IEEE Transactions on Nuclear Science*. 2011; 58(6): 3162-3166.
[9] Wu XD, et al. Energy-Efficient Scheduling of Real-Time Periodic Tasks in Multicore Systems, *Network and Parallel Computing*. 2010; 628(9): 344-357.
[10] Lelli J, et al. An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software*. 2012; 85(10): 2405-2416.