The design of embedded Operating System for vehicle Internet of Things

Luwei Cheng¹, Yongchao Tao¹,* and Songhao Piao¹

¹ Shenzhen Academy of Aerospace Technology, Shenzhen, China

*Corresponding author e-mail: taoyongchao@chinasaat.com

Abstract. This article distinguishes the difference between dynamic operating system and static operating system from the design goal, and discusses the resulting difference in the technology used in system implementation. With this as the theoretical basis, according to the ISO17356 OS specification, using hardware abstraction layer design ideas, a simple and efficient static real-time operating system is implemented on the ARM platform. This paper adopts the idea of separating mechanism and strategy, and uses internal resources as a mechanism to implement different scheduling strategies; a task scheduling algorithm based on priority grouping is proposed, which achieves good performance in time and space.

1. Introduction

In the field of automotive electronic control system development, there are often a lot of repeated development work for different products, which not only reduces the efficiency of software development, but also affects the scalability and portability of the system. In order to solve these problems, the European automotive industry formulated an open system interface OSEK/VDX specification in the mid-1990s [1]. This specification has now become an industrial standard for embedded real-time systems and has been adopted by ISO as the ISO17356 international standard. For the convenience of description, the OSEK standard used below is equivalent to the ISO17356 standard.

The application environments of embedded operating systems are diverse, and different application environments have different requirements for the services provided by the system. In some environments, users hope that the operating system can provide as many functions as possible to meet the diverse requirements of applications [2]. In other areas of high security and reliability, users pay more attention to the predictability and real-time nature of the operating system. Therefore, there are actually two main types of embedded operating systems: dynamic operating systems and static operating systems [3].

The dynamic operating system puts flexibility and adaptability in the first place, pursuing to meet the various requirements of the application to the greatest extent [4]. In order to support this flexibility, dynamic operating systems must have the ability to dynamically allocate resources, such as dynamic creation and deletion of tasks [5], and dynamic application and release of semaphores. Typical dynamic operating systems include Vxworks, pSOS, uCLinux, etc.

The design goal of a static operating system is predictability and the pursuit of optimal performance. In order to achieve this goal, users are required to statically configure all objects before the system starts, such as the number of tasks, the number of timers, etc., which gives rise to the concept of "static operating system". At present, OSEK is a typical representative of static operating systems.
systems. Different requirements lead to different goals pursued by the system, and different goals lead to differences in system design, such as task management methods and memory management methods.

2. OSEK OS standard analysis

The OSEK standard specifies four conformance classes, one conformance class can implement a version of the operating system. In fact, these four conformance classes are upwardly compatible: any application developed for the BCCx conformance class can be ported to the ECCx operating system without modification. Any application developed for the xCC1 compliance class can also be transplanted to the xCC2 operating system without modification.

![Figure 1. Operating system conformance class](image)

(1) Task management

OSEK specification divides tasks into basic tasks and extended tasks. The basic task has three states: running state, ready state and suspended state, and the extended task has one more waiting state.

![Figure 2. Task state transition diagram](image)

(2) Resource management

When multiple tasks access shared resources, a resource management mechanism needs to be used to protect the critical section. The OSEK specification adopts the Priority Ceiling Protocol, that is, when a task occupies a resource, the priority of the process will temporarily increase to the priority of the resource; when the task releases the resource, its priority returns to the original priority of the task.

(3) Interrupt handling

The OSEK specification defines two interrupt service routines:

Type 1 interrupt: This type of interrupt program does not use the services of the operating system. After the interrupt, the processing program continues to execute from the place where the interrupt occurred, without affecting the management of the task.

Type 2 interrupt: This type of interrupt program is configured through user subroutines when generated by the system. It can call the operating system's API and affect task scheduling.

(4) Alarm mechanism

The alarm is a service mechanism provided by the OSEK specification for processing cyclic events. To some extent, it can be regarded as a timer in other operating systems. When the count value of the alarm is equal to the set value, the system calls the service program to set the event or activate the task.
3. Operating system design and implementation

3.1 Introduction to XEN virtual machine scheduling algorithm

In order to facilitate the transplantation of the system, we adopt the design idea of the hardware abstraction layer to strictly divide the hardware-related and hardware-independent parts. The overall design is divided into two parts:

1. Operating system kernel: The part that has nothing to do with hardware, is the core of the operating system, and provides users with various services of the operating system. Mainly responsible for the management of tasks, resources, events, counters, alarms and interrupt service routines.

2. Hardware abstraction layer: The hardware-related part is the basis for the operating system kernel to run on a specific hardware platform. It mainly includes system startup, task context switching, interrupt and clock bottom processing.

![Diagram of Task Management](image)

**Figure 3.** The overall structure of the operating system

3.2 Task management design

Task management is the core of the operating system, and it mainly faces two problems: the effective division of scheduling strategies; the balance of scheduling algorithms in time performance and space performance.

1. Separation of scheduling mechanism and strategy

OSEK supports four scheduling strategies: preemption, non-preemption, hybrid preemption and packet scheduling. In this regard, we adopt the idea of separating mechanism and policy: the operating system provides internal resources as a mechanism, and users use internal resources to implement various scheduling strategies. The system does not distinguish between these types of scheduling strategies. When the task starts to run, the internal resources are automatically acquired; when the task runs, the internal resources are automatically released. For the management of internal resources, follow the priority ceiling protocol, see the resource management design section for details.

- Preemptive scheduling: Direct scheduling according to the priority of the task;
- Non-preemptive scheduling: When a task is running, it has the highest priority internal resources to ensure that it is not preempted by other tasks during the running process;
- Hybrid preemptive scheduling: process tasks according to the methods in (1) and (2);
- Group scheduling: When tasks are running, they will have the highest priority internal resources of the group to ensure that they will not be preempted by tasks of the same group, but can be preempted by higher priority tasks of other groups.

2. Scheduling algorithm based on priority grouping

The task scheduling algorithm is the core algorithm of the operating system and is related to the overall performance of the system. In order to achieve the goal of simple implementation and a balance in time and space, we propose a scheduling algorithm based on priority grouping:

Suppose the highest priority of all tasks is N, and they are divided into P groups from small to large, and each group has a $\frac{N}{P}$ priority.
When an explicit scheduling is performed, the highest priority in the P group is searched from high to low, and the average cost is \( \frac{P}{2} \);

- When activating a task, compare the priority of the activated task with the highest priority of the group, and then update the highest priority of the group. The cost is 1 comparison operation;

- When terminating a task, update the highest priority of the group that the task belongs to, find the highest priority in the group from high to low, and the average cost is \( \frac{N}{2P} \);

In summary, the total cost of the system for task scheduling is \( \frac{P}{2} + 1 + \frac{N}{2P} \). Obviously, when \( P = \sqrt{N} \) is the total cost of the system is the smallest.

\( \mu \)C/OS-II is an excellent operating system in the embedded field, especially in the scheduling algorithm. We compare the performance of the above scheduling algorithm with the \( \mu \)C/OS-II scheduling algorithm. Because \( \mu \)/OS-II only supports 64 fixed priority numbers, take, in the above model; the comparison results are shown in Table 1.

| Comparison index                  | C-OSEK                     | \( \mu \)C/OS-II 调度算法 |
|-----------------------------------|-----------------------------|-----------------------------|
| Explicitly schedule the time required once | Average 4 lookups           | 2 searches                  |
| space                             | Need to use a 256 byte array for bitmap mapping | No additional space overhead |
| flexibility                       | The number of priorities can be greater than 64; the lower the highest priority of the task, the faster the scheduling | The number of priorities is fixed at 64; no matter what the highest priority of the task is, the scheduling time remains unchanged |

From Table 1, we can see that, compared to the \( \mu \)/OS-II scheduling algorithm, the C-OSEK scheduling algorithm saves a lot of space overhead, and obtains high flexibility, but the time cost is very small. Since the number of priorities used in practical applications generally does not exceed 64, the performance of the C-OSEK scheduling algorithm will be very good.

### 3.3 Resource management design

Mutual exclusive access to the critical section must be guaranteed by the operating system. A good mutual exclusion method should have four characteristics: (1) Two tasks cannot occupy the same resources at the same time; (2) Priority inversion cannot occur; (3) Deadlock cannot occur; (4) Tasks that have occupied resources cannot enter the waiting state.

The OSEK specification uses external resources as a means of providing mutual exclusion, and uses the Priority Ceiling Protocol to solve the above four problems. The specific methods are as follows:

1. When a task enters the critical area to obtain resources, its own priority is raised to the ceiling priority of the resource, so as to ensure that it is not preempted by other tasks that own the resource in the critical area;

2. When the task leaves the critical area, the resource is released, and its own priority is restored to the original priority. At this time, other tasks with the resource can enter the critical area;

The advantage of the priority ceiling protocol over semaphores and other mutexes is that it avoids deadlock and priority inversion, and converts the mutual exclusion problem into a task priority problem, thereby causing problems Simplified.
Access to the critical section not only exists between tasks, but also exists between tasks and interrupts, and between interrupts and interrupts. Therefore, we extend the priority ceiling protocol to the interrupt level, the specific method is as follows:

Suppose the resource R is owned by a total of n tasks $T_1, T_2, ..., T_n$ and at the same time by a total of m interrupts $I_1, I_2, ..., I_m$.

1. When a task $T_j (1 \leq j \leq n)$ acquires a resource R, its priority is raised to the highest priority of all tasks, and all interrupts that own the resource are prohibited;
2. After the interrupt $I_k (1 \leq k \leq m)$ acquires the resource R, all interrupts that own the resource except itself are prohibited;

3.4 Design of interrupt management

 Interruption is an important factor affecting the performance of embedded systems. Some interrupts are independent of the operating system and will not affect the behavior of the kernel. The OSEK standard calls them Type 1 interrupts; while in other interrupts, users may perform operations that affect task scheduling, such as activating tasks and setting time, etc., this type of interrupt should be system-aware (RTOS-aware), OSEK standard calls it type 2 interrupt.

Type 1 interrupts are mostly used for tasks that want to be executed quickly, without interacting with the operating system, and their overhead is minimal; Type 2 interrupts need to interact with the operating system and use the API provided by the operating system. Therefore, the system design should achieve the following two goals:

- Type 1 interrupts cannot be interrupted by Type 2 interrupts to ensure that they are executed as soon as possible, but can be interrupted by Type 1 interrupts with higher priority;
- Type 2 interrupts can be interrupted by all Type 1 interrupts, and at the same time can be interrupted by Type 2 interrupts with higher priority;

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

- The lowest priority of all Type 1 interrupts The highest priority of all Type 2 interrupts;
- Type 2 interrupts need to use EnterISR and LeaveISR at the beginning and end to inform the operating system of the arrival of the interrupt;

The saving and restoring of interrupt context is another key to the design. There are generally two methods: (1) save the interrupt context in the current task stack; (2) use a separate interrupt stack. If you save the interrupt context in the task stack, you need to increase the task stack. Since the user cannot estimate the time when the interrupt comes, memory is wasted when the interrupt frequency is not high, and stack overflow is prone to occur when the interrupt frequency is too high. Therefore, we use a separate interrupt stack to save the interrupt context, which not only ensures safety, but also improves memory utilization.

C-OSEK supports interrupt nesting. The specific method is to switch the processor to another mode and turn on the interrupt before entering the interrupt handler, as shown in Figure 4.

![Figure 4. The overall structure of the operating system](image)
4 System application verification and performance analysis

This text chooses ISO17356 OS specification 2.2.3 edition and ISO17356 OIL specification 2.5 edition, realizes C-OSEK operating system on ARM AT91SAM7X256 hardware platform, and carries on the perfect function test.

Due to the in-depth analysis of the characteristics of the static operating system, the design of the task management mechanism is very concise. The C-OSEK kernel code is less than 3000 lines, only 3.5K after compiling under the 32-bit ARM platform. The following is a comparison with the kernel sizes of some OSEK products that are currently influential, as shown in Table 2.

| Operating system name       | Kernel size |
|-----------------------------|-------------|
| Emerald-OSEK                | 5.5 KB      |
| OSEKTurbo                   | 5 KB        |
| P-OSEK(Integrated System)   | 2.8 KB      |
| SmartOSEK                   | 4.3 KB      |
| C-OSEK                      | 3.5 KB      |

The size of each object control block is an important factor in determining the amount of memory used by the system, and it is also an important indicator to measure the simplicity of the system design.

Summary: This article deeply analyzed the difference between static operating system and dynamic operating system, and designed and implemented the C-OSEK operating system based on this. C-OSEK is a real-time embedded operating system with completely independent intellectual property rights, which will certainly promote the development of my country's automobile industry. In the next step, research will be conducted on the integration of AUTOSAR standards.

5 Acknowledgments

This work was financially supported by Shenzhen Special Fund for Strategic Emerging Industry Development JCYJ20170816151055158 and Key Basic Research Projects of the Military Science and Technology Commission Basic Strengthening Plan 2019-JCJQ-ZD-097-00.

References

[1] The OSEK Group. OSEK/VDX Operating System Specification 2.2.3. 2005
[2] Joseph Lemieux. OSEK/VDX automotive electronics embedded software programming technology. Luo Kelu et al. Beijing University of Aeronautics and Astronautics Press, 2004: 33-47, 78-89
[3] Andrew S. Tanenbaum. Operating system design and implementation (third edition). Chen Yu et al. Publishing House of Electronics Industry, 2007: 13-54
[4] Zhang Baomin, Sun Xiaomin. Design of Embedded Real-time Operating System for OSEK Vehicles of Tsinghua University. Computer Engineering and Design. 2004, 25: 661-664.
[5] Chen Wei. SmartOSEK operating system design and space optimization. Degree Thesis of Zhejiang University. 2005:55-56
[6] J.J.Labrosse. Embedded real-time operating system μC/OS-II. Translated by Shao Beibei et al. Second edition. Beijing University of Aeronautics and Astronautics Press, 2005: 88-90,286-306.