國立交通大學

資訊科學系

碩 士 論 文

Seed： 一個適合網路設備之嵌入式即時作業系統核心

Seed: an Embedded Real-time Operating System Kernel for Network Appliances

研 究 生：王俊喬

指導教授：張瑞川　教授

中華民國 九十三 年六月

Seed: 一個適合網路設備之嵌入式即時作業系統核心

研究生：王俊喬　　　　　指導教授：張瑞川教授

國立交通大學資訊科學所

論　文　摘　要

嵌入式系統在現代日常生活中到處可見，並且扮演了一個相當重要的角色。嵌入式系統的特點在於它是一個應用程式特定的系統，而且通常硬體資源較為不足。除此之外，有愈來愈多的嵌入式應用程式要求能夠即時地完成自己的工作，並且希望能有與其他系統利用網際網路溝通的能力。

為了能夠滿足以上的系統需求，我們發展了一個支援網際網路的即時嵌入式作業系統，名叫 Seed。Seed 是一個小尺寸、富有彈性、高效能、具可攜性的嵌入式作業系統核心。此外，Seed 核心為了支援即時系統，所提供的服務都是可預測或者固定時間的行為模式。最後，為了能讓系統更進一步能夠有網際網路溝通的能力，我們也移植了一個名叫 lwIP 的小型 TCP/IP 網路溝通協定模組在 Seed 作業系統上。

Seed 作業系統核心目前支援先佔式多工、執行緒彼此間溝通和同步的機制、以及記憶體、定時器、中斷的管理。核心影像檔若包含 lwIP 大小為 75 Kbytes，若是不包含 lwIP 大小是 21 Kbytes。核心的尺寸相當地小因而適合嵌入式系統。然後我們把 Seed 的效能測量出來，結果顯示出我們的系統相當適合小型的即時嵌入式網路設備。

Seed: an Embedded Real-Time Operating System Kernel for Network Appliances

Student：Chun-Chiao Wang　　　　Advisor：Prof. Ruei-Chuan Chang

Institute of Computer and Information Science

National Chiao-Tung University

Abstract

Embedded systems are ubiquitous and play a significant role in modern daily life. The characteristics of embedded systems are application-specific and scarce hardware resources. Besides, more and more applications in embedded system care not only real-time to complete their works, but also want to own Internet-access capability which allows the devices to communicate with other systems.

To achieve these requirements, we developed an Internet-supported embedded real-time operating system called Seed. The Seed kernel is small, flexible, high performance, and portable for embedded system. Besides Seed have deterministic or constant timing behavior to support real-time system. Finally, to enable the Internet-access capability, we ported a small TCP/IP stack, lwIP, to Seed.

Seed kernel currently provides preemptive multitasking, task synchronization / communication, and management of memory, timers and interrupts. The size of the kernel image is about 75Kbytes with lwIP, or 21Kbytes without lwIP. It is quite small and suitable to embedded system. The performance results show that Seed is quite suitable for a small real-time embedded network appliance.

Acknowledgements

For accomplishing this thesis, I would like to thank my respected advisor Prof. Ruei-Chuan Chang for instructing me in research. Additionally, I deeply appreciated David Chang's assistance for giving me much advice and opinion on writing this thesis. Thanks to each member of the computer system laboratory for their encouragement and kindly help

Besides, I would like to thank my parent and my girlfriend, too, for their encouragement and unlimited love. Finally, I want to thank all my friends for all the joyous things that inspire my life.

Chun-Chiao Wang

Institute of Computer and Information Science

National Chiao-Tung University

2004/6

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Embedded systems are ubiquitous and play a significant role in modern daily life. They can be found everywhere, such as watches, VCD/DVD players, digital cameras, mobile phones, PDAs, missile systems, flight control systems and etc. Traditional embedded operating system usually addresses two issues: limited hardware resources and real-time support. Therefore, an embedded operating system must be able to run on top of limited resources as well as provide real-time support to its applications.

With the popularity of Internet and rapid development of network technologies, Internet-access capability is becoming a necessarily for many embedded systems. Such network appliances can not only communicate with each other, but also enable many creative applications on them such as remote control functionality. For example, an user can control an in-home VCD/DVD recorder to record his favorite TV programs when he is working at office.

Therefore, modern embedded operating systems should satisfy the requirements of running on the top of limited hardware resources, supporting real-time applications, and providing Internet-access capability. Many commercial real-time operating systems do satisfy the above requirements. However, they are usually expensive and not open source. On the other hand, non-commercial kernels often have limitations for fulfilling the requirements. This motivates us to design and implement a real-time embedded operating system, named Seed, for network appliances. Seed contains an OS kernel designed for time-critical embedded applications. Besides the basic kernel services, we also ported a small TCP/IP protocol stack called lwIP [7] to Seed. This makes systems based on Seed be Internet-enabled.

The kernel has the following four design goals. First, it is designed to be flexible for supporting various applications in embedded systems. Second, Seed supports real-time applications. For example, it provides preemptive multitasking and deterministic (or constant) timing services. Third, Seed is designed for high performance and small kernel size. And fourth, Seed is an extremely portable kernel. It is easy to port Seed to other hardware platforms by replacing the hardware abstraction layer.

Seed is currently implemented on Samsung SNDS100 evaluation board. The kernel supports preemptive multitasking, task synchronization/communication, and management of memory, timers and interrupts. As we mentioned above, the TCP/IP stack is also ported. The size of the kernel image is about 75Kbytes with lwIP, or 21Kbytes without lwIP, which is small enough for resource-limited systems.

1.2 Thesis organization

The rest of the thesis is organized as follows. The following chapter describes previous research related to real-time embedded kernels. Chapter 3 presents the design and implementation details of Seed kernel. Besides, we introduce lwIP and the porting status in this chapter. The experiment results are shown in Chapter 4. Finally, Chapter 5 gives conclusions.

Chapter 2

Related Works

In this chapter, we describe some of the related real-time embedded kernels.

2.1 Linux & RTLinux

Linux is a famous open source operating system. Many vendors such as MontaVista [15] and Metrowerks [12] have put efforts on making Linux an Embedded RTOS. The techniques include shrinking the kernel and the libraries, reducing the timer interrupt intervals, inserting preemption points in the kernel, and etc. However, Linux kernel is inherently designed for general-purpose and non-real-time systems [3]. The techniques can not transform Linux to a true real-time kernel.

Therefore, a Real-Time Linux (RTLinux) [8][17] was developed for real-time applications. In RTLinux, a real-time extension co-exists along with the original Linux kernel. And, each application is divided into the real-time part and the non-real-time part. The former runs directly on the real-time extension, while the latter runs on the original Linux kernel. However, the cooperation between the RT and non-RT parts not only consumes extra computing and memory resources but also make the application development complicated.

Seed is a pure real-time embedded kernel. Developing real-time applications on Seed is easy and instinctive without extra overheads.

2.2 eCos

The eCos kernel [18] is a flexible, configurable, and real-time embedded kernel. It has a hardware abstraction layer for increasing portability. Similar to Seed, eCos divides the interrupt handing into two parts: Interrupt Service Routine (ISR) and Deferred Service Routine (DSR). However, the DSR of eCos has no priority levels. By contrast, Seed has eight priority levels and supports constant time DSR scheduling. Moreover, eCos only supports 32 priority levels for constant time task scheduling, while Seed kernel supports 512 priority levels.

2.3 μC/OS-II

μC/OS-II [10] is also a preemptive, real-time, multi-tasking kernel. However, Seed is more flexible and powerful than μC/OS-II. For example, μC/OS-II supports only 64 task priorities. Moreover, different tasks must be associated with different priorities. This prevents the using of Round-Robin scheduling. Finally, μC/OS-II adopts only preemptive multitasking without the possibility of non-preemptive multitasking.

By contrast, Seed supports 512 task priorities and allows more than one tasks share the same priority. Round-Robin scheduling, preemptive or non-preemptive multitasking are all allowed in the Seed kernel.

2.4 Commercial RTOSes

There are many commercial real-time embedded kernels in the market, such as WindowsCE[13], Nucleus[1], vxWORKS[23], QNX[16], Lynx[11] and etc. They support real-time applications and are suitable for embedded systems. However, all of them are proprietary. Some of them even do not open their source code. Seed is an open source project, so it is royalty and buyout free.

Chapter 3

Design and Implementation

In this chapter, we will describe the design goals and actual implementation of the Seed kernel. In Section 3.1, we first give an overview of the kernel. Then, we describe each Seed component from Section 3.2 to Section 3.8. Finally, we describe the status of implementation in Section 3.9.

3.1 Kernel Overview

Seed OS kernel is designed for embedded systems and real-time systems. Due to the limited memory and CPU resources of embedded system and the timing requirements of real-time systems, Seed has following features:

- **Flexibility**

  Since embedded systems are application-specific, it is important to keep the kernel as flexible as possible. Seed kernel divides its code into several components for flexibility. Each component can be replaced, removed and modified without totally rewriting the kernel. The interfaces and files of each kernel component are explicitly defined. In addition to a component-based kernel, we implement a Seed component as flexible and simple as we can. For example, when we create a task, we can specify its time-slice value, option of preemptive or non-preemptive, and etc. Furthermore, changing these values at run-time is allowed by the exported interfaces of Seed.

- **Deterministic Timing (Real-Time support)**

  Real time systems care not only the correctness of the computation, but also when the computation is completed. Therefore, a key requirement of a real-time kernel is *deterministic* timing. This means that the kernel services should consume only expected amounts of time. In non-real-time kernels, their services may inject random delay into the application, and thus cause the unexpected response time. On the other hand, the real-time kernels (including Seed) have deterministic timing behaviors. Furthermore, real-time kernels should offer constant (load-independent) timing. In other words, a service consumes the same time to complete the job irrespective of the workload. The constant timing is always considered when we develop Seed kernel. With constant or deterministic timing, it is possible to analyze the worst-case performance of the real-time software.

- **Portability**

  Seed explicitly divides the kernel source code into hardware-dependent part and hardware-independent part. The former is called Hardware Abstraction Layer (HAL). The HAL abstracts the underlying hardware, hence makes Seed portable. If we want to port Seed to another hardware platform, all we have to do is modify the HAL. All other components do not need to be changed at all.

- **High performance**

  Since application is an embedded system should cooperate with the kernel, there is little need to implement multiple protection modes. Thus Seed selects single protection mode (i.e., kernel mode) for good performance. Traditional OS, such as Linux, adopts a dual-mode scheme (i.e. user mode and kernel mode) for kernel protection. Under this scheme, additional code is needed for changing protection domains. According to the previous research [4], single protection mode can save the time of system calls. Besides, for the sake of better performance, the Seed kernel is implemented in C language rather than other object-oriented languages such as C++ and JAVA.

Figure 3.1 shows the architecture of the Seed system. As shown in the figure, the applications run on top of the OS, and the hardware is under the control of the OS. Typical components in an OS are TCP/IP stack, file systems, window systems, and etc. However, the kernel (e.g., Seed kernel) is the real nucleus of the whole operating system. The kernel is the system resource manager that allocates resource (such as CPU time, memory and I/O devices) to the tasks. As shown in the right part of Figure 3.1, Seed has following kernel components to manage the system:

- Task management
- Interrupt management
- Memory management
- Timer management
- Message queue management
- Semaphore management
- Hardware Abstraction Layer (HAL)

The features of these components are described from Section 3.2 to Section 3.8.

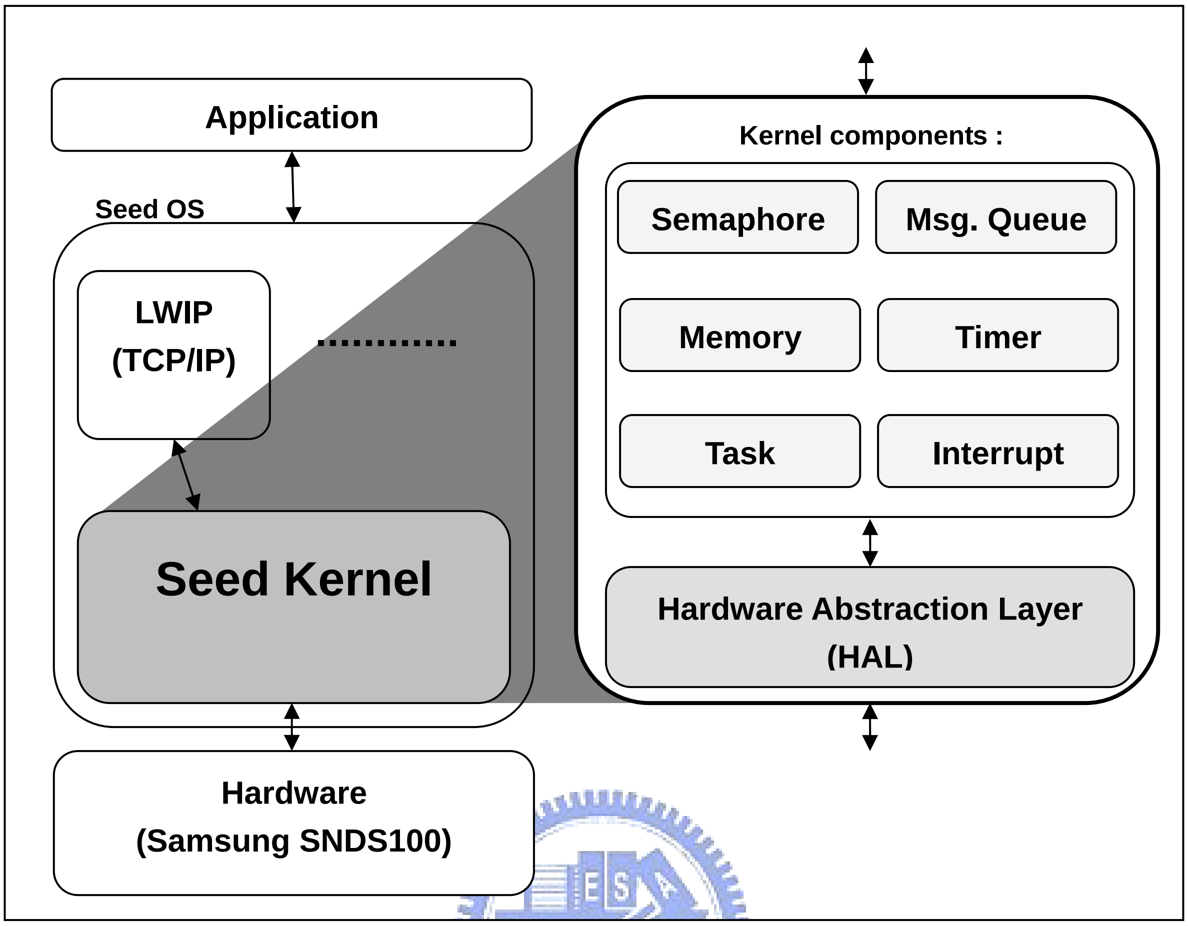


Figure 3.1 Seed Kernel Architecture

3.1 Task Management

3.1.1 Design

A task (also called a process or a thread) is an instance of program in execution. An application may divide its work into tasks, each of which is responsible for a portion of the whole job. Each task has a Task Control Block (TCB), which contains CPU registers, stack, and etc. Seed kernel provides the following features on task management.

- **Multi-tasking**

  Multi-tasking is the ability to support multiple concurrent tasks running on the same CPU. It creates pseudo parallelism and maximizes the use of the CPU. Besides, multi-tasking provides a modular construction mechanism for applications, which allows the application programs to be designed and maintained in an easier way.

- **Multiple priorities**

  Each task can be assigned a priority when it is created by the application designer. The priority ranges from 0 to 511, where 0 is the highest priority and 511 is the lowest priority. Seed always schedules the task with the highest priority to run.

- **Preemptive**

  Preemptive multi-tasking means that the running task can be interrupted at any time by another higher priority task. Oppositely, in the case of non-preemptive multi-tasking, the scheduling happens only when a task completes, or it explicitly releases the CPU. Seed kernel supports both kinds of multi-tasking. If we don't want a task to be preempted, we can specify the task as non-preemptive. In a real-time system, it is prefer to select preemptive multi-tasking for fast system responsiveness.

- **Constant time scheduling**

  Seed always selects the highest priority task to run. In non-real-time kernels, the time spent by a scheduler for choosing the next task to run is usually non-deterministic. Some real-time kernels, including Seed, allow the task scheduler to find out the task that should be run next in a short constant time.(i.e., O(1) time) We will explain the details of the task scheduling mechanism in Section 3.2.2.

- **Time-Slicing (Round-Robin scheduling)**

  Seed allows two or more tasks have the same priority. Each task runs for a determined amount of time of time (called *quantum*), and then the scheduler selects another task with the same priority to run. The time quantum can be assigned while a task is created, or be changed at run-time. Note that time-slicing is disabled if the task is non-preemptive.

At any given time, a Seed task is always in one of the following states: *create*, *running*, *ready*, *suspend*, and *terminate*. As shown in Figure 3.2, a task enters the *create* state when it is created. When the task is inserted into the ready queue[1] and waiting for execution, it is in the *ready* state. Once the scheduler selects the task to execute, the task goes to the *running* state. When the task is suspended and waiting for certain system resources, it will go into the *suspend* state. The task will be resumed and enter into the *ready* state while the resource is available. Finally, the task goes to the *terminate* state when it has been killed or its job is completed.

[1] The tasks that are ready for execution are kept on a list called *ready queue*.

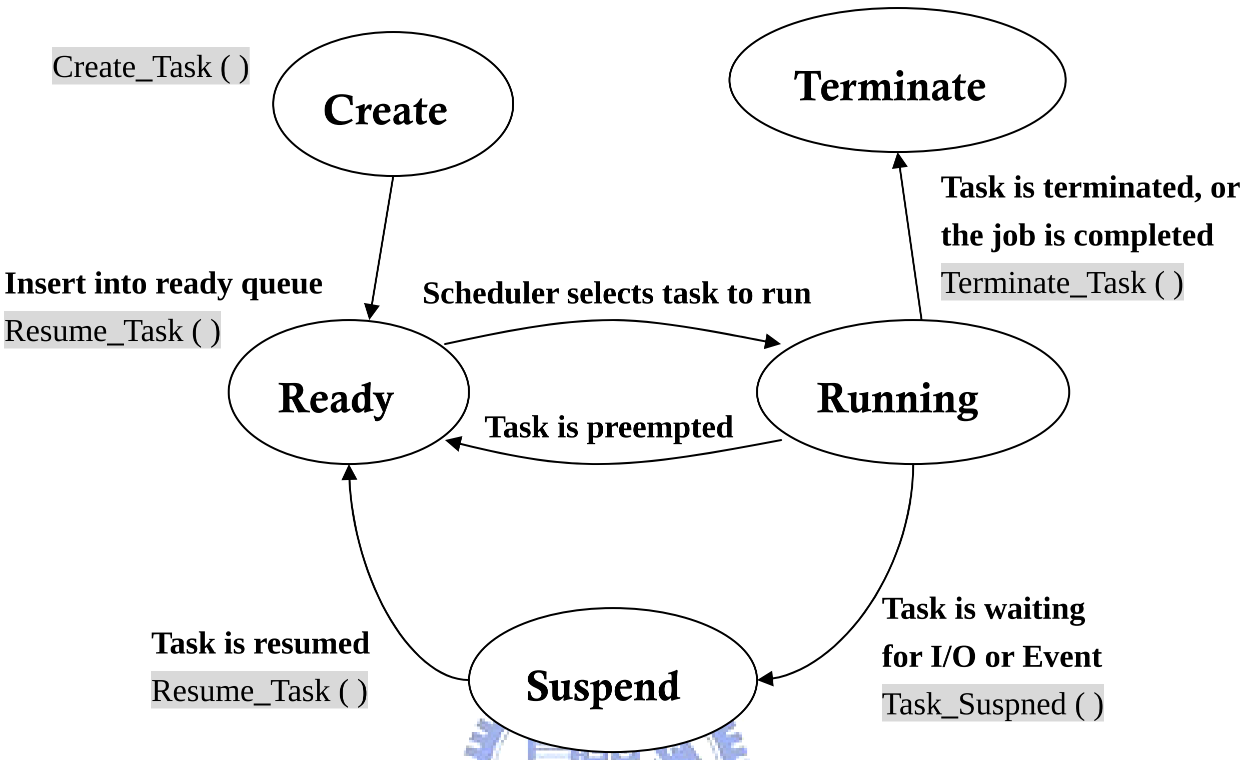
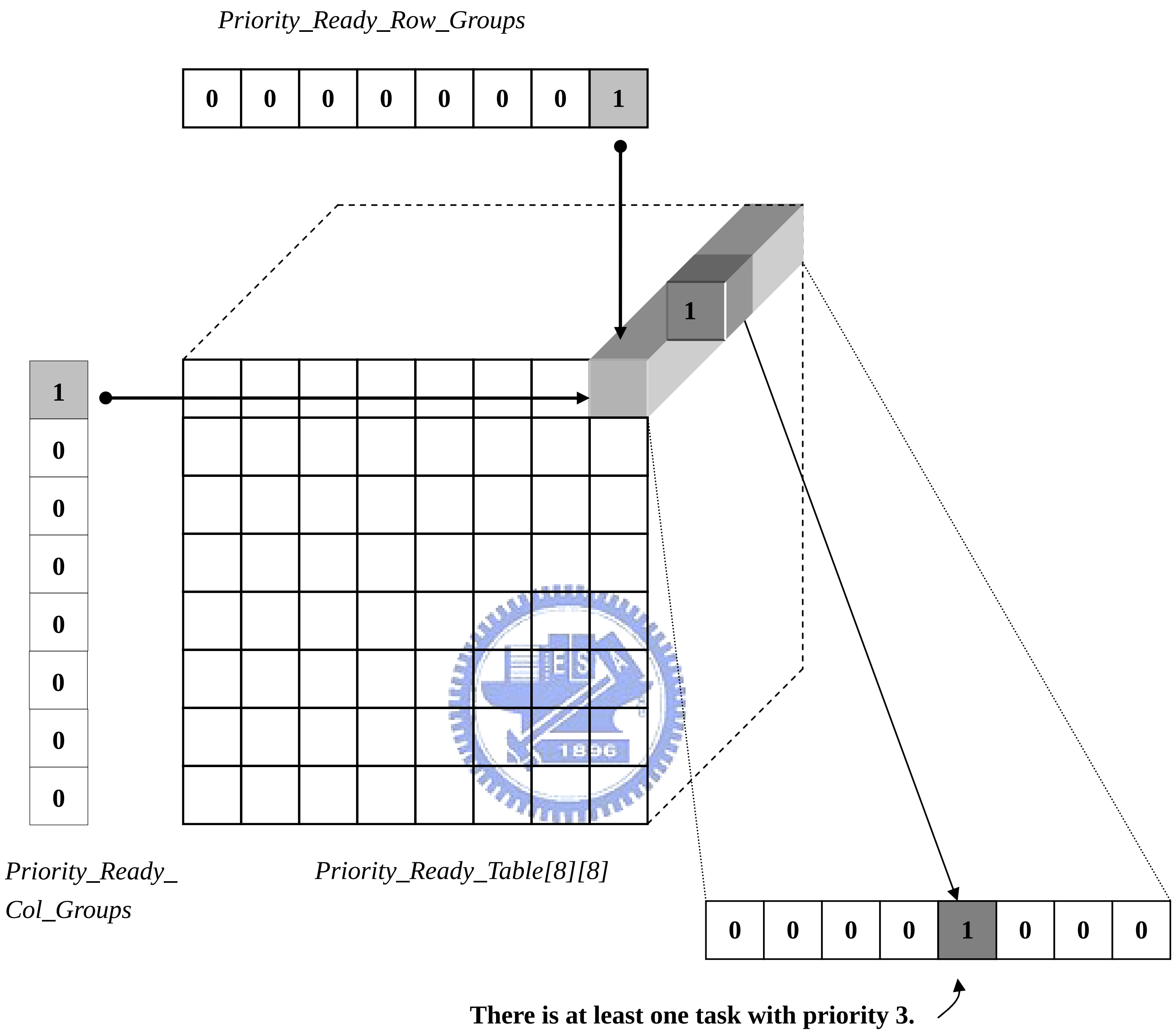


Figure 3.2 Task States

3.1.2 Implementation

In this section, we describe the implementation of Seed scheduler and the task ready queue.

We implemented Seed scheduler in a fashion similar to the μ C/OS-II scheduler[10] . However, we extended it to support more priorities (i.e., 512 priorities) and keep the scheduling job in a constant time. As shown in Figure 3.3, we represent 512 task priorities in an 8 × 8 × 8 cube data structure, *Priority_Ready_Table*. The *Priority_Ready_Table* is an array of 64 elements, where each element is a 8-bit bitmap. Each bit is used to indicate the existence of tasks with the corresponding priority. For example, in *Priority_Ready_Table* [0][0], the binary value 00001000 means that there is at least one ready task with priority 3. To determinate which task to run, the scheduler will select the lowest priority number that has its bit set in the *Priority_Ready_Table*. For the sake of efficiency, we use two data structure as the indexes of this array, *Priority_Ready_Row_Groups* and *Priority_Ready_Col_Groups*. Each of them is an 8-bit bitmap and each bit corresponds to a priority group. *Priority_Ready_Row_Groups* is the row index of this array, and *Priority_Ready_Col_Groups* is the column index. For example, if the bit 0 of *Priority_Ready_Row_Groups* and the bit 0 of *Priority_Ready_Col_Groups* are set, there is at least one task, with its priority between 0 to 7, ready for execution. This is because the two indexes point to the element 0 of the array (i.e., *Priority_Ready_Table* [0][0]) , which has the bitmap that stands for priority 0 through 7.



Priority_Ready_Row_Groups
0
0
0
0
0
0
0
1
1
1
0
0
0
0
0
0
0
Priority_Ready_
Col_Groups
Priority_Ready_Table[8][8]
0
0
0
0
1
0
0
0
There is at least one task with priority 3.

Figure 3.3 Data Structures for Task Scheduling

Using the data structures described above to find out the highest priority task, we use a table-lookup approach. Figure 3.4 shows a mapping table with 256 (2^8) values that is used for finding the highest priority task. In fact, it is a priority resolution table. Given an index, the corresponding value in the table stands for the lowest set bit of that index. This is used to determine the highest task priority represented by the previously mentioned bitmaps. For example, if the element of *Priority_Ready_Table* [0][0] is 8 (i.e., 1000b), we look up the value of *Mapping_Table[8]* , and the value 3. It means that the lowest bit of 8 is bit 3, and hence the highest task priority is 3. By using the *Mapping_Table*, we can find the highest task priority via three times of table-lookup, which is shown in Figure 3.5. First, we look up the lowest bit of *Priority_Ready_Row_Groups* and *Priority_Ready_Col_Groups*. With these two bits, we can find out the corresponding bitmap of the *Priority_Ready_Table*. Finally we look up the lowest bit of this bitmap.

```
UNSIGNED_CHAR   Mapping_Table [256] = {
        0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
        };
```

Figure 3.4 Mapping Table for Finding the Highest Priority Task

```
Row= Mapping_Table [Priority_Ready_Row_Groups];
Col  = Mapping_Table [Priority_Ready_Col_Groups];
highest_ready_priority = (UNSIGNED) ( (Row << 6) + (Col <<
3) + Mapping_Table [Priority_Ready_Table[Row][Col]] );
```

Figure 3.5 Pseudo Code for Finding the Highest Priority Task

No matter how many tasks are in the system, the cost of task scheduling in Seed is fixed. However, when the number of tasks is quite small, the Seed scheduling time may be slower than some non-real-time kernels. This is due to that a non-real-time kernel usually adopts non-deterministic scheduling, which may find out the highest priority rapidly when there are very few tasks. But the term *real-time* does not mean *as fast as possible*. Instead, it requires consistent, repeatable, known timing performance. Therefore, in order to achieve deterministic timing, the small and fixed computation overhead of Seed scheduling is worthy.

After finding out the highest priority task, Seed will de-queue a task control block from the ready queue. As shown in Figure 3.6, the *Priority_Ready_Task_List* is an array of *SEED_TASK* (task control block) pointers. Each pointer stands for a single priority, and points to a list of ready tasks (specifically, TCBs) with that priority. The TCB list is a doubly-linked list so that we can insert and remove a TCB in a constant time.

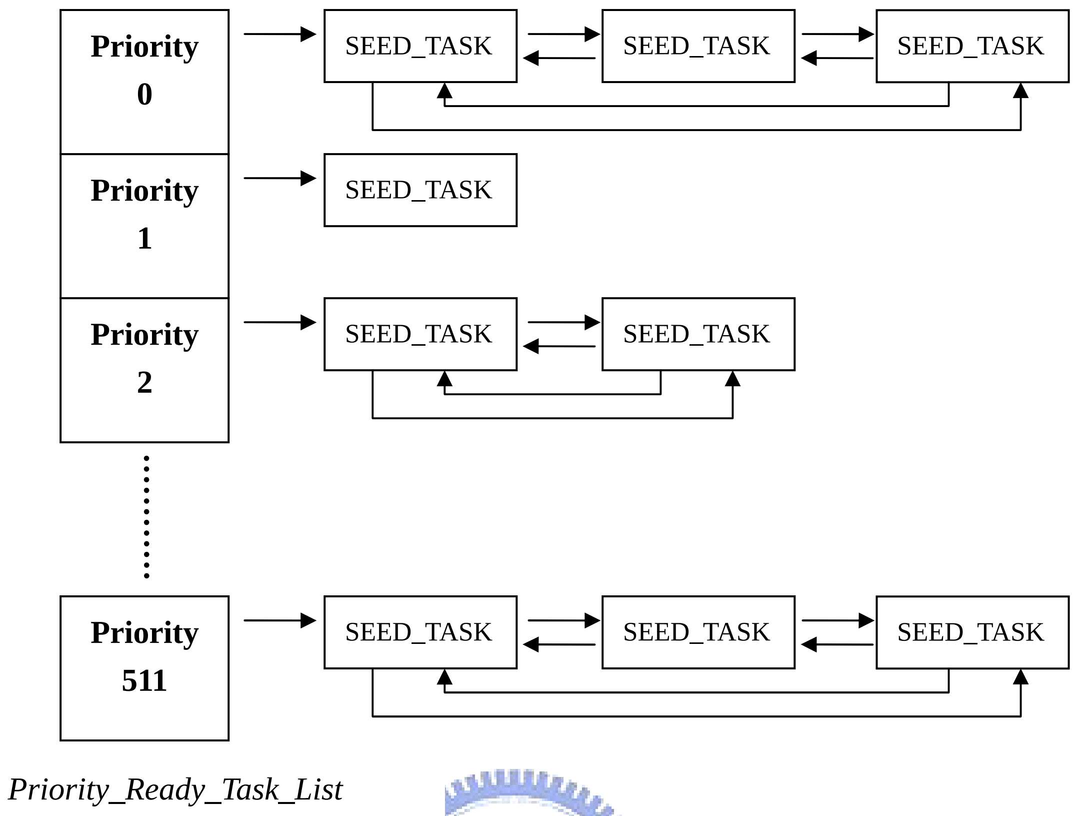


Figure 3.6 Task Priority Ready Queues with Priorities

3.1.3 Interface

The interface exported by the task management component is as follows:

1. **Seed_Create_Task**: This function creates a new task. The user can specify the time-slice value, preemptive or non-preemptive, priority and so on.
2. **Seed_Resume_Task**: This function resumes a previously suspended or created task. It will call the scheduler to check if a reschedule is needed.
3. **Seed_Suspend_Task**: This function suspends the specified task. If it is the current running task, the function will invoke the scheduler to selects next ready task to run.
4. **Seed_Terminate_Task**: This function terminates the task we specified.
5. **Seed_Relinquish_Task**: This function will yield the control of CPU to next same-priority task, and put the task to the end of the corresponding ready

TCB list.

6. **Seed_Task_Sleep**: This function suspends the calling task for the specified number of timer ticks (1 timer tick = 10ms).
7. **Seed_Change_Task_Priority**: This function changes the priority of the specified task to the new priority value. This function will call the scheduler to check if Seed needs to preempt the executing task with new priority task.
8. **Seed_Change_Task_Preemption**: This function changes the preemption state of currently executing task. If the preemption value is changed from non-preemptive to preemptive, it will call the scheduler to check if a preemption is needed.
9. **Seed_Change_Time_Slice**: This function changes the time slice of the specified task to the specified value. If the new time slice value is zero, the time slicing of the task is disabled.

The interface routines for internal use are as follows:

1. **Task_Initialize**: This function is called by *Seed_Initialize* (i.e., the system initialization function). It is responsible for setting the initial value of the internal variables and global data structures in task component.
2. **Task_Start**: This function will be invoked when the task is going to execute at the first time. It will call the task entry function with the parameters of the task.
3. **Task_Scheduler**: This function implements the task scheduling algorithm. It is responsible for finding out the highest priority task, and checks the preemption state of the executing task to see if a task context-switch is needed.
4. **Task_Context_Switch**: This function is invoked to perform a task context

switch. The context (i.e., CPU registers) of the original task is saved into memory, and the context of the resumed task is loaded into the CPU.

5. **Spinlock_Lock**: This function is called to lock a spinlock that protect critical system resources (e.g., kernel data structures) from simultaneous access. If other task has already held this spinlock, the calling task will perform context switch and give the control of CPU to the task that hold the spinlock.
6. **Spinlock_Unlock**: This function is called to unlock the spinlock. The code between *Spinlock_lock* function and *Spinlock_Unlock* function will become a critical section that is mutual exclusive.
7. **Task_Time_Slice**: This function is called when the time slice of a task is run out. It is responsible for moving the task to the end of the corresponding TCB list.
8. **Task_Timeout**: This function is called to process the task suspension timeout condition. It will resume the task from the suspend state.
9. **Lock_Scheduler**: This function is used to prevent task scheduling. The scheduler is temporarily stopped after calling this function.
10. **Unlock_Scheduler**: This function is the counterpart of *Lock_Scheduler* function. It is used to continue task scheduling.

3.3 Interrupt Management

3.3.1 Design

Interrupt is a mechanism for providing immediate response to an external hardware event. When an interrupt occurs, the CPU suspends the current path of execution and transfers control to the appropriate ISR (Interrupt Service Routine). Seed allows a component such as a device driver to register an ISR, un-register an ISR with for an IRQ number (interrupt request number) dynamically. The HAL interrupt component will recognize the IRQ, save the CPU context, execute to the ISR, and restore the context of CPU. The details will be described in section 3.8.

In order to protect the internal data structures from simultaneous access, we usually disable the interrupts when we are serving an interrupt. However, it is not desirable to disable the interrupts for a long time in a real-time system. Therefore, Seed adopts 2-stage interrupt handling scheme, which is also adopted by other real-time kernels, e.g. the eCos RTOS [18]. The interrupt handling is separated into two stages, ISR stage and DISR (Deferred Interrupt Service Routine) stage.

In the ISR stage, a normal ISR is executed with interrupts disabled. During the execution, the ISR may activate a DISR to complete the service later. When the ISR is finished, the DISR starts. A DISR is allowed to be run with interrupts enabled. Each DISR has its own stack and control block, and hence it can temporarily be blocked for synchronization or mutual exclusion purpose. In other words, a DISR is just like a task except that it is activated by an ISR. Under this 2-stage interrupt handling mechanism, the interrupts won't be disabled for a long time.

The eCos kernel also supports DISR. However, the DISRs do not have priorities, and hence that are executed in FIFO order. This might cause problems when a DISR activated by a higher priority ISR is blocked by another one that is activated by a lower priority ISR. By contrast, there are eight priority levels available for Seed DISRs. If a higher priority DISR (i.e., activated by a higher priority ISR) becomes ready, the lower priority DISR is preempted. And, DISRs with the same priority are executed in the order they are activated. The same as the task scheduling time, the time of scheduling a DISR is a small constant time.

3.3.2 Implementation

The implementation of the interrupt system is divided into two parts, namely the ISR and the DISR components.

- **ISR component**

  We define an array called *IRQ_Handlers*. Each element is a function pointer to an ISR, and the IRQ number is used for indexing the array. Therefore, an ISR can be registered and un-registered with this array dynamically. When an interrupt occurs, the interrupt part of the HAL component will get the IRQ number from the hardware register, and invoke the corresponding ISR.

- **DISR component**

  The data structures for implementing the DISR component are similar with the Seed tasks. It has an 8-bit bitmap, named *Active_DISR_Priority*, for the priority status. Since there are only eight priorities for DISRs, the *Active_DISR_Priority* is enough to represent the priority status. Besides, there is ready queue called *Active_DISR_First* with eight elements for queuing DISRs. Each element contains a doubly-linked list of DISR control blocks (i.e., *SEED_DISR*). Figure 3.7 shows an example. In this figure, there are DISRs with priority 0 and 7. Therefore, the *Active_DISR_Priority* bitmap is 129 (i.e., 10000001b). Similar to the approach used in the task management system, we also take advantage of the mapping table to find out the highest priority DISR in a constant time period. It is worth to note that the scheduler always selects the DISRs to run before running the tasks in order to completing the interrupt service as fast as possible.

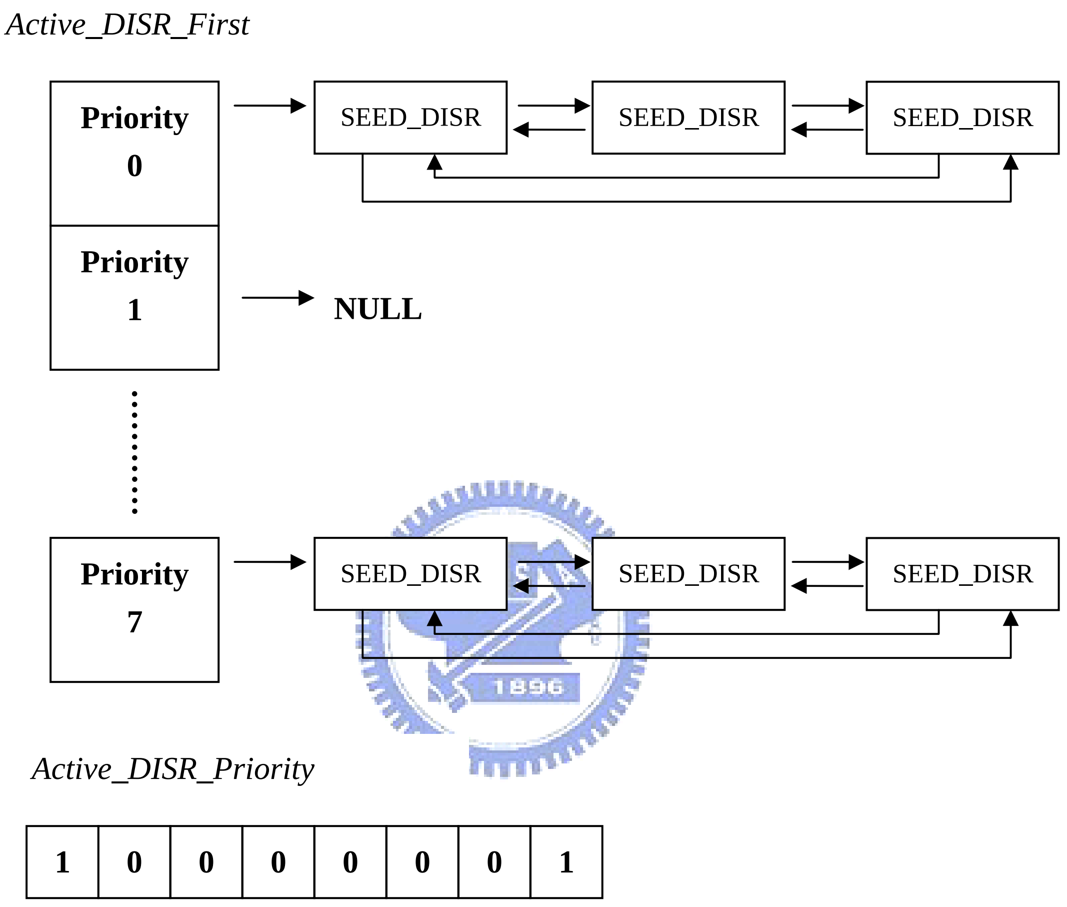


Figure 3.7 DISR Data Structures

3.3.3 Interface

The interface exported by the interrupt system is shown in the following.

1. **Seed_Register_ISR**: This function registers the ISR for the specified IRQ number. If the IRQ number has been registered, it will replace the old one.
2. **Seed_Unregister_ISR**: This function un-register the interrupt service routine.

3. **Seed_Create_DISR**: This function creates a DISR and initializes its control block.
4. **Seed_Activate_DISR**: This function activates the specified DISR. It will insert the control block of the DISR into the corresponding ready queue. This function is usually called by an ISR.

The interface routines for internal use are as follows.

1. **Interrupt_Initialize**: This function is invoked by *Seed_Initialize* (i.e., the system initialization function). It is responsible for setting the initial value of the internal variables and global data structures used by the interrupt system.
2. **Forward_ISR**: This function is called by the interrupt part of Seed HAL. It calls the corresponding ISR for the given IRQ number.
3. **DISR_Start**: This function is a wrapper to the real DISR function. It will be invoked at the first time. It calls the DISR entry function. In addition, it updates the related global data structures of the DISR after the DISR has finished its job.

3.4 Timer Management

3.4.1 Design

This component provides all timing facilities in Seed, including timer ISR, time-slicing and the timer service. The timer service is used frequently by other kernel components (e.g., task management) and time-sensitive applications. The basic time unit is *tick*, which is the time between two successive hardware timer interrupts. In our current implementation, a tick is equal to 10 ms. We classify the timers into two types according to their usage, the application timers and the task timers.

The application timers can be created, deleted, enabled, and disabled dynamically by the applications. These timers execute user-supplied routines when they are expired. The user-supplied routine is specified while creating the timer, and can be treated as a useful tool for application programming. The second kind of timer is the task timer. Every Seed task has a built-in task timer, which allows a task to suspend for a specified time period.

3.4.2 Implementation

The implementation of the timing system can be separated into the following two parts:

Timer Interrupt Handling

Periodic timer interrupt is the base of whole timing system. The HAL of Seed will initialize the hardware timer and interrupt controller (see Section 3.8). Each time a timer interrupt happens, the time ISR (i.e., *Timer_ISR* function) increases the system clock (i.e., jiffies), and decreases the time-slice value of the running task and the remaining-time value of the running timer. If the time-slice or the remaining-time value reaches zero, the *Timer_ISR* will activate the *Timer_DISR*, which processes the quantum expiration of the task (i.e., call the *Task_Time_Slice* function), and handles the timer expiration. The timer expiration either resumes the suspended task for the timeout of task timer, or calls the user-specified expiration function of application timer.

Timer maintenance

The internal data structure for implementing a task timer or an application timer is timer control block called *BASIC_TIMER*. As shown in Figure 3.8, the active (i.e., enabled) timers are maintained in a doubly-linked list, and the *Active_Timer_List_First* pointer references to the first timer (i.e. the running timer) of the list. The timer list is maintained in the order of expiration time. Each timer control block contains a remaining-time field, which represents the timing difference between the expiration time of the timer and the expiration time of its previous timer. When the field becomes zero, the timer expires. This approach is used in order to avoid adjusting the entire list on every timer interrupt.

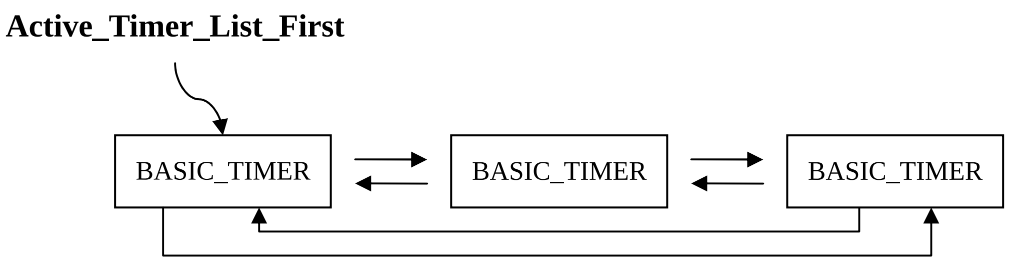


Figure 3.8 Active Timer List

3.4.3 Interface

The interface exported by the timer system is shown in the following.

1. **Seed_Create_Timer**: This function creates an application timer. The user-specified expiration function is executed when the specified timeout value runs out.
2. **Seed_Control_Timer**: This function is used to enable and disable the specified application timer.
3. **Seed_Retrieve_System_Clock**: This function returns the current value of the system clock, which is the number of ticks since the system boots up. The value increases by one every timer interrupt.
4. **Seed_Set_System_Clock**: This function sets the system clock to the new specified value.

The interface routines for internal use are as follows.

1. **Timer_Initialize**: This function is called by *Seed_Initialize* (i.e., the system initialization function). It is responsible for setting the initial value of the internal variables and global data structures in timer component.
2. **Run_Timer**: This function is used to enable an application or task timer. The timers will be inserted into the active timer list.
3. **Stop_Timer**: This function is used to disable an application or task timer. The timers will be removed the active timer list.
4. **Timer_ISR**: This function is the ISR of timer interrupt. It increases the system clock and check the timeout situation.
5. **Timer_DISR**: This function is the DISR of timer interrupt. It will be activated by *Timer_ISR* when the running timer is timeout or the current task's time-slice runs out.

6. **Timer_Expiration**: This function is invoked by *Timer_DISR*. If a task timer is expired, it will call the *Task_Timeout* function to resume the suspended task. If an application timer is expired, it will call the user-specified expiration function.

3.5 Memory Management

3.5.1 Design

Dynamic memory management is traditionally implemented in the *malloc* and *free* functions. However, the execution time of *malloc* and *free* is generally non-deterministic due to the memory management algorithms. In addition, this mechanism is easily to suffer from the fragmentation problem.

For this reason, in addition to the traditional mechanism, Seed provides an alternative memory management mechanism which is also adopted by other real-time kernels, such as μC/OS-II and Nucleus. This alternative mechanism allows the applications to obtain fixed-sized memory blocks from a partition, which is made of a contiguous memory area. All memory blocks are the same size in a partition. The allocation and de-allocation of memory blocks is done in a small constant time. In addition, this type of memory management also avoids the fragmentation problem. The limitation of this approach is its inflexibility. If an application needs memory blocks with different sizes, it has to create many partitions. This makes application programming a little inconvenient. However, in order to achieve constant timing and avoid fragmentation, it is still worthy to create partitions.

It deserves to be mentioned that Seed provides options for suspension when allocating memory. When a task attempts to allocate a memory block from an empty partition, it can suspend and wait for available block. The task can be resumed when a block is returned to the partition. There are three kinds of suspension.

- *Always suspend*: The task always suspends and waits for a block. It is only resumed until a block is available.
- *Suspend with a timeout*: The task suspends for a block. However, it only waits for a specified timeout ticks.

- *No suspend*: If there is no available block in the partition, the task just get an error message without suspension.

Besides, if multiple tasks suspend on a single memory partition, the tasks are resumed in one of the following, which can be specified when the memory partition was created.

- *FIFO (First-In-First-Out) order*: Tasks are resumed in the order they were suspended.
- *Priority order*: Tasks are resumed in the order of their priorities.

This suspension mechanism is similar to Nucleus and μC/OS-II. However, μC/OS-II does not support FIFO suspending order, and it does not provide suspension options when allocating a block from an empty memory partition.

.3.5.2 Implementation

In this section, we describe how a memory partition is implemented. The internal data structure of a memory partition is a partition control block called *SEED_PARITION*. As shown in Figure 3.9, the partition control block manages a contiguous memory area. It divides this area into fix-sized blocks, and contains a pointer (i.e., *free_list*) that points to the first free block. The free blocks are linked as a list by using the linking pointers, the first four bytes of the block data space. Block allocation and de-allocation involves only the head of the list. Therefore, the time is constant. Note that the linking pointer can used to store data when the block is allocated. Each block has a four-byte overhead (i.e., block header) that contains a pointer to the partition control block. This reverse pointer is useful when de-allocating a block. It allows constant time de-allocation under the condition that the application does not specify the corresponding partition control block when it de-allocates a block. We prevent application from specifying the partition control block for the following reason. If the application returns the block to a wrong partition control block, the system may crash. As we mentioned above, the drawback of using this pointer is that it adds four-byte overhead to each block. This makes the available blocks in a partition become fewer. For example, there will be only four 20-byte blocks available in a 100-byte partition.

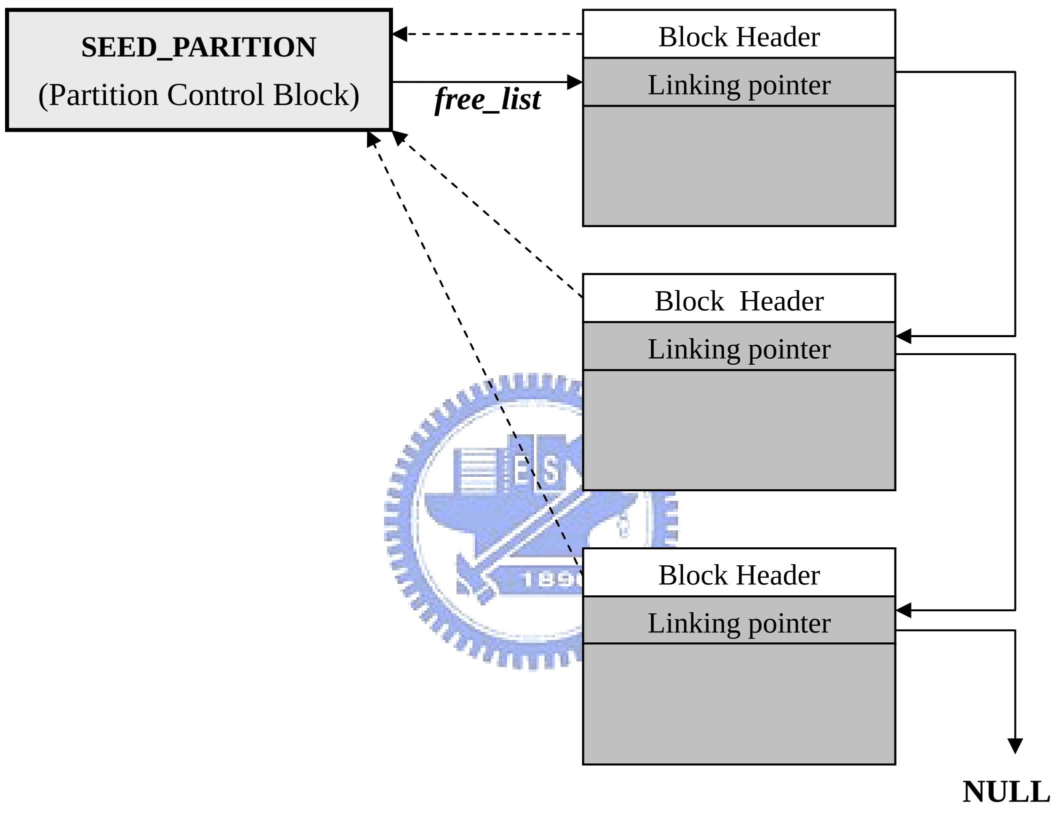


Figure 3.9 Memory Partition Data Structure

The memory partition implementation of Seed is more robust than μC/OS-II. Specifically, a Seed memory partition provides a reverse pointer to avoid system crash, which is not available in μC/OS-II. On the other side, a Seed memory partition uses less space overhead than Nucleus. The latter includes the linking pointers into the block headers so that the linking pointers cannot be used to store data.

As for the implementation of suspension, every partition control block has a waiting list for suspended tasks that are sorted in FIFO or Priority order. If another task returns a block back to the empty partition, Seed will resume the first waiting task in the list and give the block directly to the resumed task.

3.5.3 Interface

The interface exported by the memory partition is shown in the following.

1. **Seed_Create_Memory_Partition**: This function creates a memory partition with fixed-size blocks. The partition size, the block size, and the task resuming order can be specified by the caller.
2. **Seed_Delete_Memory_Partition**: This function deletes a previous created memory partition. It resumes all the waiting on that partition.
3. **Seed_Allocate_Memory_Block**: This function allocates a memory block from the partition. The suspension type is specified by the caller. Note that the size of the block is already defined when the partition is created.
4. **Seed_Free_Memory_Block**: This function returns a memory block back to the partition. If there are tasks waiting on this partition, this function will resume the first task and give the block to it. Note that the caller does not need to specify the corresponding partition.

The interface routines for internal use are as follows.

1. **Memory_Partition_Initialize**: This function is called by *Seed_Initialize* (i.e., the system initialization function). It is responsible for setting the initial value of the internal variables and global data structures in memory partition component.
2. **Memory_Partition_Cleanup**: This function is invoked by *Terminate_Task*

or *Task_Timeout*. It will remove the task from the waiting list of the memory partition.

3.6 Message Queue

3.6.1 Design

Message queue is a mechanism for tasks to communicate between each other. Each message queue is capable of holding multiple messages. When a task sends a message, it will be copied into the message queue. Then the receiving task will copy the message out of the message queue. A message can be placed at the front of the queue or at the end of the queue. A message consists of one or more bytes, which can be specified while creating the queue. However, we suggest that applications set the message size to the pointer size, that is to say, each message is a pointer. The pointer can be initialized to point to some application's data structure that will actually be referenced by the receiver. By this way, copying a large message can be avoided. In our implementation, the processing time of sending and receiving a message is deterministic, and it is relative to the message size.

Seed provides a suspension service for message queue as the memory partition does in Section 3.5.1. Here are two reasons for suspension. First, when a task is attempting to send a message to a full message queue, it is suspended and then waits for available space of the queue. The task is resumed when other task gets a message from the queue. Second, when a task tries to receive a message from an empty queue, it is suspended to wait for an incoming message. The task will be resumed when other task sends a message to the queue. The suspension options (i.e., *Always suspend*, *Suspend with a timeout*, and *No suspend*) and the suspension orders (i.e., *FIFO order* and *Priority order*) are all the same with the memory partition in Section 3.5.1. Besides, it is allowed to broadcast a message to all the waiting tasks with Seed message queue.

3.6.2 Implementation

The internal of the message queue is a circular buffer, as shown in Figure 3.10. The implementation of Seed is similar to other real-time kernels. The *read* pointer points to the first message of the queue, and the *write* pointer points to the location where the next message will be inserted. However, if we send a message at the head of the queue, the message will be inserted into the preceding entry of the *read* pointer. The *start* pointer indicates the starting address of the queue area, and the *end* pointer points to the ending address of the queue area. The *read* and *write* pointers can wrap the queue to implement circular buffer.

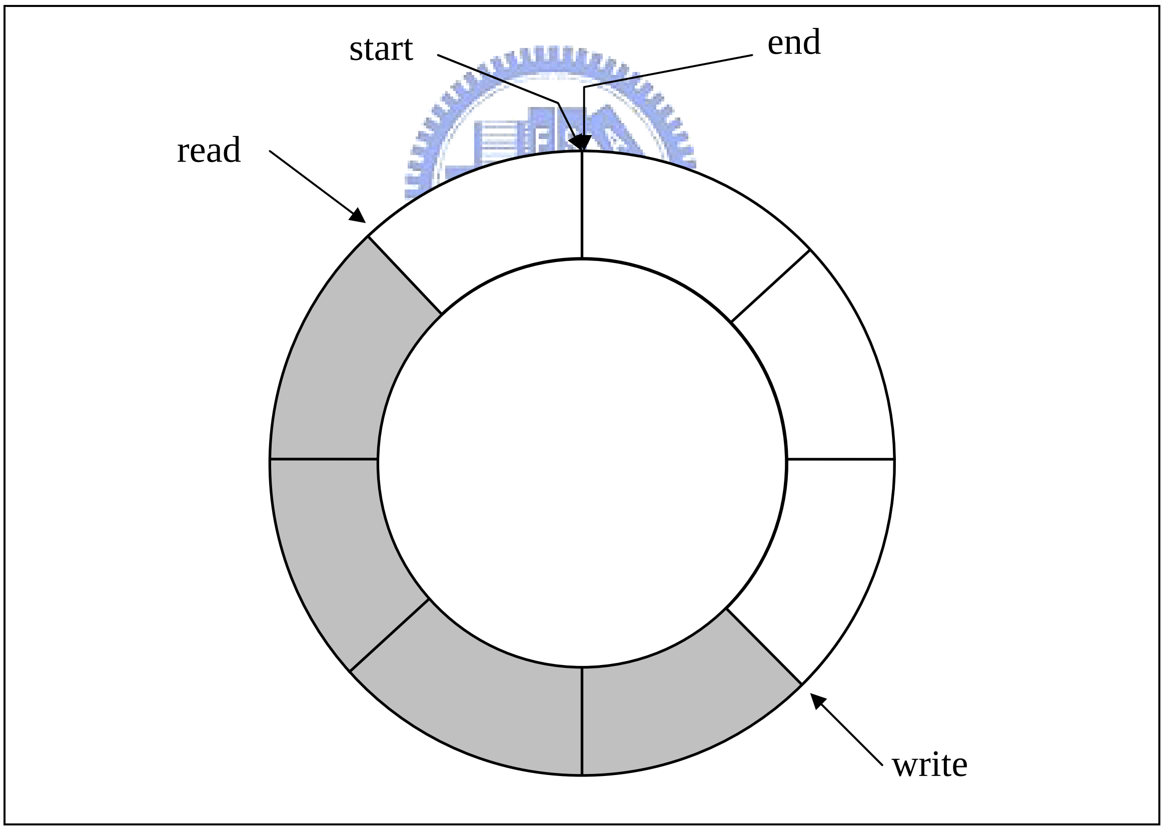


Figure 3.10 Message Queue Data Structure

The implementation of suspension is similar to memory partition. There is also a waiting list for suspending tasks. Note that if other tasks want to send a message into the empty queue, Seed will resume the waiting task and directly copy this message to the resumed task without copying into the queue. This mechanism saves message copying time.

3.6.3 Interface

The interface exported by the message queue is shown in the following.

1. **Seed_Create_Message_Queue**: This function creates a message queue from a memory area specified by the caller. The queue size, message size, and suspension order are specified by caller.
2. **Seed_Delete_Message_Queue**: This function deletes a created message queue. It resumes all the tasks waiting on that message queue.
3. **Seed_Send_Message_To_Queue**: This function places a message at the tail of the specified queue. If there is not enough queue space, a suspension is allowed.
4. **Seed_Send_Message_To_Front_Of_Queue**: This function places a message at the head of the specified queue. If there is no enough space in the queue, a suspension is allowed.
5. **Seed_Receive_Message_From_Queue**: This function receives a message from specified queue. If there is no message in the queue, a suspension is allowed.
6. **Seed_Boradcast_Message_To_Queue**: This function broadcasts a message to all tasks that are waiting for an incoming message with the specified message queue.

The interface routines for internal use are shown as follows:

1. **Message_Queue_Initialize**: This function is called by *Seed_Initialize* (i.e.,

the system initialization function). It is responsible for setting the initial value of the internal variables and global data structures in the message queue component.

2. **Message_Queue_Cleanup**: This function is invoked by *Terminate_Task* or *Task_Timeout*. It will remove the suspending task from the waiting list of the message queue.

3.7 Semaphore

3.7.1 Design

Semaphore is a mechanism for synchronizing the execution of the critical section in an application. Seed provides counting semaphores. The value of each semaphore ranges from 0 to 2^{32} -1. If the semaphore value is 1, it is a binary semaphore for mutual exclusion. Two basic operations of a semaphore are *obtain* and *release*. When a task wants to get into the critical section (e.g., access the shared resource), it tries to obtain the semaphore. If the semaphore is obtained successfully, the counter of the semaphore will be decreased by one. Inversely, when a task leaves the critical section, the task will release the semaphore and increases the counter by one.

However, if a task fails to obtain the semaphore (i.e., the counter of semaphore is zero), the task may suspend on the waiting list until it is available. The suspension options and suspension orders are the same with the memory partition and the message queue (Section 3.5.1 and Section 3.6.1). These options and orders are free for applications to choose. The optional timeout on obtain-semaphore suspension can be used to recover from a *deadlock*.

Priority inversion is a common problem in real-time kernels. It occurs when a higher priority task is suspended on a semaphore that a lower priority has. This problem can be solved via the *priority-inheritance protocol* [22]. In this protocol, kernel increases the priority of the lower priority task to the priority of the higher priority task (i.e., inherit the priority of higher priority task). When the priority inherited task release the semaphore, the priority is reverted to its original value. Seed supports priority-inheritance for semaphores in order to help the higher priority task obtaining the semaphore as soon as possible.

Note that the μC/OS-II kernel does not support a general priority-inheritance protocol. It requires the users to reserve some priority levels for priority-inheritance usage since it can not allow two tasks to share the same priority. Seed kernel does not have such limitation, so that it can implement the general priority-inheritance protocol.

3.7.2 Implementation

The internal structure of a semaphore is a 32-bit integer. Besides, there is also a waiting list for the suspending tasks. If other tasks release a semaphore, Seed will resume the waiting task and pass the semaphore directly to the resumed task.

When an application creates a semaphore, it can specify whether priority-inheritance is enabled or not. If priority-inheritance is enabled and a task is suspended on an un-available semaphore, the semaphore will check whether the priority of owner task is lower than the suspending task. If so, the owner task will inherit the highest priority among the suspending tasks and save the original priority (i.e., invoking *Inherit_Priority* function). After the owner task releases the semaphore, the priority of the owner task is reverted to the original priority (i.e., invoking *Disinherit_Priority* function).

3.7.3 Interface

The interface exported by the semaphore is shown in the following.

1. **Seed_Create_Semaphore**: This function creates a semaphore. The counting value and the suspension order are specified by caller.
2. **Seed_Delete_Semaphore**: This function deletes a created semaphore. It resumes all the tasks waiting on that semaphore.
3. **Seed_Obtain_Semaphore**: This function tries to obtain an instance of semaphore. If the counter of semaphore is zero, a suspension is allowed.

4. **Seed_Release_Semaphore**: This function releases an instance of semaphore. It will resume and give the semaphore to the suspended task if it is waiting.

The interface routines for internal use are shown as follows:

1. **Semaphore_Initialize**: This function is called by *Seed_Initialize* (i.e., the system initialization function). It is responsible for setting the initial value of the internal variables and global data structures in semaphore component.
2. **Semaphore_Cleanup**: This function is invoked by *Terminate_Task* or *Task_Timeout*. It will remove the suspending task from the waiting list of the semaphore.
3. **Inherit_Priority**: This function is used to implement the priority inheritance protocol. It helps the owner task to inherit higher priority of the waiting task.
4. **Disinherit_Priority**: This function is used to implement the priority inheritance protocol. It helps the owner task to restore the original priority.

3.8 Hardware Abstract Layer (HAL)

In this section, we introduce the Hardware Abstraction Layer (HAL). The HAL is the hardware dependent portion of the kernel. We can port Seed kernel to another hardware platform more efficiently by only replacing or modifying the HAL. Currently, Seed is running on SNDS100 board. And, we take our implementation as an example to introduce the HAL. The HAL are divided into four parts: *Boot*, *Interrupt*, *CPU Context* and *Devices*. Each part of HAL is described in the following subsections.

3.8.1 Boot

As shown in Figure 3.11, the default boot loader of SNDS100 loads the Seed kernel image into the memory, and jumps to the boot part of the HAL (i.e., the entry point of Seed kernel). The boot part is responsible for initializing the hardware devices. In other words, it contains the low-level initialization functions. In SNDS100 board, it does following jobs:

- *Vector table setting*: It sets the vector table which corresponds to the ARM exception modes [9][21]. Then, it handles the branch of an exception (i.e., save CPU contexts, branch exception handlers, and restore CPU contexts).
- *System map initialization*: It sets up the system memory mapping of the board.
- *Stack assignment*: It assigns stack space for each CPU operation modes.
- *Hardware devices initialization*: It initializes hardware devices on the board (e.g., hardware timer, interrupt controller, terminal driver and etc.).

After hardware initialization is finished, the boot part of HAL will invoke the *Seed_Initialize* function. This function is hardware-independent and is used to call the initial functions of other Seed components. It creates an idle task and application tasks, and invokes the scheduler to start these tasks.

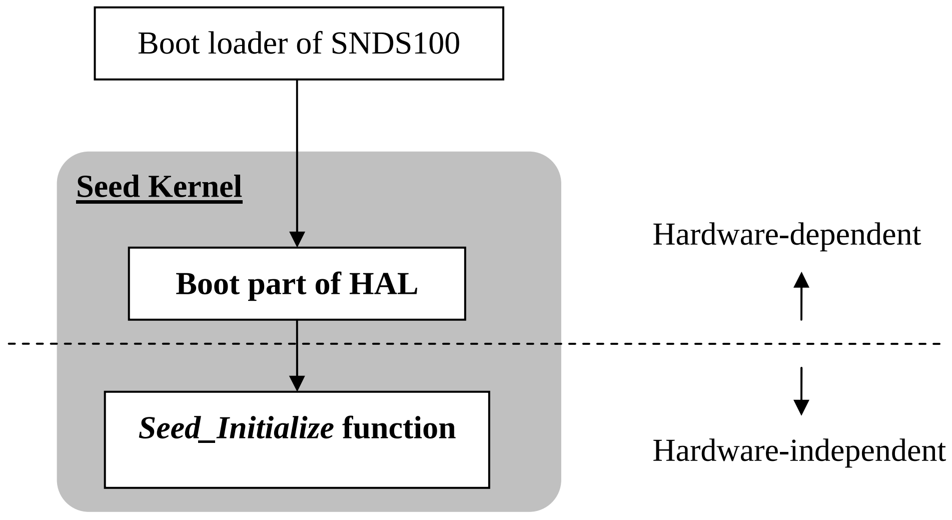


Figure 3.11 Seed Boot Flow

3.8.2 Interrupt

The interrupt part of HAL manages the hardware interrupt system. The *HAL_IRQ_Init* function is used to initialize the interrupt controller at the beginning. Then it enables the interrupts that we want to handle, and registers the Interrupt Service Routines (ISRs) with corresponding IRQ numbers.

In addition, the interrupt part of HAL helps kernel to handle interrupts. When a hardware interrupt occurs, it takes following steps:

1. It saves the CPU contexts into the memory buffer which is a member of task control block (i.e., *jmpbuf*). Because an interrupt may occur at any time, it must save all of the CPU registers.
2. It recognizes the IRQ number by reading the hardware register. Then it invokes the corresponding ISR.
3. After the ISR is finished, the interrupt part of HAL will check whether there is a ready DISR. If there is a ready DISR, its previous saved context

will be loaded.

4. If there is not a ready DISR and the interrupted task is preemptive, the interrupt part of HAL will check whether there is a higher priority task ready to run. If there is a higher priority task, its previous saved context will be loaded. Otherwise, the kernel loads the context of the interrupted task.

Besides, the interrupt part of HAL provides the interfaces to enable or disable interrupts, namely *HAL_Enable_IRQ* and *HAL_Disable_IRQ*, respectively.

3.8.3 CPU Context

This part of HAL provides a set of CPU primitives. Specifically, it provides interfaces for saving and restoring the CPU context. When a task A is switched to another task B, Seed invokes *HAL_CS_Save_Context* function to save the context of A and *HAL_CS_Restore_Context* function to restore the context of B. However, there is still another interface called *HAL_Interrupt_Restore_Context* for context restoration. This function is used to restore the context that was saved by interrupt part of HAL. The difference between these two restoring functions is the number of restored CPU registers. The *HAL_Interrupt_Restore_Context* function restore more registers than the *HAL_CS_Restore_Context* function. Because an interrupt may occur at any time, all of the CPU registers are saved. However, the normal context switch only occurs at the *Task_Context_Switch* function, which does not care the content of argument registers (i.e., a1 to a4 in ARM CPU).

3.8.4 Devices

This part of HAL is responsible for controlling the other devices on the platform.

For example, the hardware timer and UART driver of SNDS100 are initialized by the *HAL_Timer_Init* and *HAL_UART_Init* functions, respectively. The *HAL_Timer_Init* initializes the timer controller and assigns the timer interrupt period as 10ms. The *HAL_UART_Init* initializes the UART device and sets a proper baud rate for that device.

In addition, the data types of Seed are defined in a single header file for the case of porting. For example, the type *UNSIGNED* means a 32-bit unsigned long integer in Seed. Every Seed component and applications include this header file. Therefore, if the type is different at other hardware platforms, we only have to modify this file.

3.9 Implementation Status

In this section, we describe the status of the implementation.

3.9.1 Platform

Seed is currently implemented on Samsung SNDS100 Evaluation Board, which is based on the S3C4510B/KS32C50100 microcontroller [19][20]. Samsung S3C4510B is a 16/32-bit RISC microcontroller built around the ARM7TDMI RISC processor. It integrates an Ethernet controller. Thus, it is a good choice for Ethernet-based embedded systems. The S3C4510B can operate in a frequency up to 50MHz. Besides the S3C4510B, the SNDS100 Board also consists of boot EEPROM, DRAM module, SDRAM, serial ports for console, and Ethernet interface. Figure 4.1 illustrates the SNDS100 block diagram.

Seed works correctly with the SNDS100 board. The Seed HAL is responsible for managing the SNDS100 board. The Ethernet and UART drivers are well ported and operating correctly.

3.9.2 Implemented Kernel Components

The implemented and tested kernel components are shown as follow:

- Task management
- Interrupt management (ISR and DISR)
- Memory partition
- Timer management
- Message queue management
- Semaphore management
- Hardware Abstraction Layer (HAL)

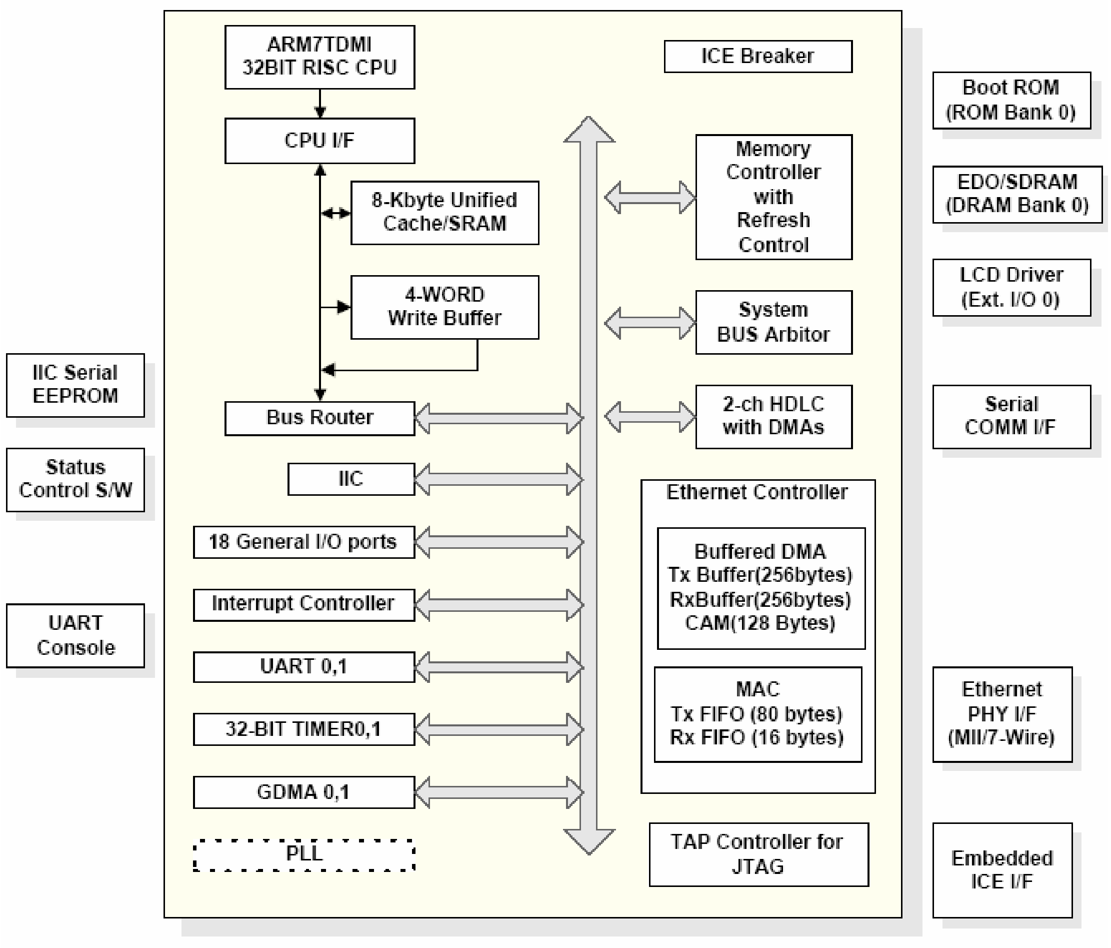


Figure 4.1 SNDS100 Block Diagram[2]

3.9.3 Implemented Drivers

The implemented and tested drivers on SNDS100 board are shown as follow:

- UART driver
- Timer driver
- Ethernet driver

3.9.4 LWIP Integration

In order to provide the Internet-access capability, we ported a small TCP/IP stack

[2] This figure is obtained from the application note of S3C4510B [19].

called *lwIP* (i.e., lightweight IP) [5][7] to the Seed kernel. lwIP is an open source implementation of TCP/IP stack. Its design goal is to reduce memory usage and code size, making it suitable for use in an embedded system. lwIP defines a common interface called *OS emulation layer* between its code and the underlying OS kernel. To port lwIP to Seed kernel, we only have to modify this layer. This layer requires the functionalities such as multi-tasking, memory management, timer, semaphore and message queue. These functionalities are fully supported by Seed kernel services. Table 3.1 shows the function mapping between OS emulation layer and the Seed kernel. Each function in the OS emulation layer is mainly implemented by a single Seed kernel function with argument adjustment.

Functions in OS Emulation Layer	**Seed Kernel Functions**
sys_thread_new	Seed_Create_Task
sys_mbox_new	Seed_Create_Message_Queue
sys_mbox_free	Seed_Delete_Message_Queue
sys_mbox_post	Seed_Send_Message_To_Queue
sys_arch_mbox_fetch	Seed_Receive_Message_From_Queue
sys_sem_new	Seed_Create_Semaphore
sys_sem_free	Seed_Delete_Semaphore
sys_arch_sem_wait	Seed_Obtain_Semaphore
sys_sem_signal	Seed_Release_Semaphore

Table 3.1 Function Mapping between lwIP OS Emulation Layer and Seed kernel

Besides, we also ported on Ethernet driver to the SNDS100 board in order to support lwIP. The network applications that currently run on lwIP / Seed are as following:

- TCP Echo server
- UDP Echo server
- Simple HTTP server
- Shell program (i.e., a telnet server).

Chapter 4

Experimental Results

In this chapter, we present the experiment results of Seed kernel. The code size and primitive performance measurement are presented in Section 4.1. We measure the network performance in Section 4.2.

4.1 Basic Measurement

4.1.1 Code Size

The code was compiled for ARM7TDMI using ARM Developer Suite 1.2 [2]. The resulting size of the compiled code is shown in Table 4.1. The *Code size* column shows the size of the compiled executable object code, and the *Data size* column shows the data size used by the object code. The total code size of Seed OS is about 16K bytes and the total data size is about 35K. The libraries are some primitive libraries for applications, such as string library, standard library (stdlib) and etc. The kernel image generated by the linker is about 21K bytes. In conclusion, Seed kernel is very small and is suitable for embedded systems whose hardware resources are scarce.

Function	**Code size (bytes)**	**Data size (bytes)**
HAL	2356	27388
Task Management	3164	2500
Interrupt Management	1036	257
Timer Management	1404	1256
Memory Partition	664	0
Message Queue	2004	0

Semaphore	776	0
Other kernel services	956	3441
Libraries	4248	308
Total	**16608**	**35150**

Table 4.1 Code Size of Seed Kernel

4.1.2 Primitive Performance Measurement

In this section, we measure the performance of the primary functions in Seed. The performance result is shown in Table 4.2. Note that interrupts are disabled during the performance measurement. These results can be treated as a reference while creating applications on Seed kernel.

Function	**Time (μ -sec)**	**Cycles**
Task_Schduler	16.079	843
Task_Context_Switch	18.081	948
Create_Task	47.207	2475
Resume_Task	8.545	448
Suspend_Task	14.763	774
Create_Message_Queue	10.147	532
Send_Message_To_Queue	16.479	864
Receive_Message_From_Queue	16.193	849
Create_Semaphore	4.101	215
Obtain_Semaphore	4.120	216
Release_Semaphore	7.706	404
Create_Memory_Partition	18.959	994
Allocate_Memory_Block	4.005	210
Free_Memory_Block	4.520	237
Create_Timer	21.954	1151

Table 4.2 Performance of Seed Kernel Functions

In addition, we measure the performance of interrupt handling. Interrupt handling can be divided into three portions. The first portion is *interrupt latency* which is defined as the time that a system takes to start running the interrupt code. In other words, it is the time of the interrupt disabled period plus the time for branching to the exception handler. The second portion is the time to save the CPU context of running task and branch to the ISR. The third portion is *interrupt recovery*. It is the time to determinate if a higher priority task is ready and the time to restore the CPU context. Table 4.3 illustrates the latency of each interrupt portion.

Function	**Time (μ-sec)**	**Cycles**
Interrupt Latency	34.695	1819
Time to save CPU context	20.409	1070
Interrupt Recovery	35.667	1870

Table 4.3 Latency of Seed Interrupt Handling

4.2 Performance Measurement of the Network System

In this section, we measure the performance of lwIP on Seed kernel. At the beginning, we measure the throughput of our system. We connect an 800 MHz Pentium III notebook (IBM Thinkpad X22) running Linux 2.4.18 to the SNDS100 board which runs Seed kernel and lwIP with a 10Mbits/Sec Ethernet link. Besides, we use a widely-used benchmarking tool called *Test TCP (TTCP)* to measure the TCP throughput. We configure *TTCP* to send 8M bytes of data from one device to the other. The testing result is shown in Table 4.4. When the SNDS100 board acts as the receiver, the throughput is 115.93 KB/second. When it acts as the sender, the throughput can reach to 190.54 KB/second. The throughput is lower when it acts as the receiver. The reason is lwIP needs another task responsible for receiving Ethernet packets. This will involves more task context switches, thus there is an unavoidable performance degradation for lwIP receiving packets.

	Throughput (KBytes/Sec)
lwIP Rx	115.93 KB/Sec
lwIP Tx	190.54 KB/Sec

Table 4.4 Throughput of LWIP Running on Seed

Besides the throughput, we also measure the round-trip time of lwIP running on Seed. This measurement was taken using the *ping* program. The average round-trip time is 0.991 ms when we send 1000 packets of 64-byte to the SNDS100 board. The performance results shown in this section are comparable with previous ones [6]. However, we do not perform precise comparison since the platforms are different.

At last, we measure the performance of a simple web server that running on lwIP. The performance is measured by using the WebStone [14] benchmark version 2.5. We configure the profile as that a client continuously requests a single homepage file in ten minutes. The testing result is shown in Table 4.5. The result shows that the performance of the small HTTP server is acceptable for small embedded devices.

Server Connection Rate	39.05 Connections/ Sec
Server Throughput	147.20 KBytes/ Sec
Average Response Time	25.59 ms

Table 4.5 Performance of Simple Web Server Running on lwIP

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we describe the architecture and internal of Seed, a real-tome embedded kernel with Internet-access capability. The design goal of Seed is to support small network appliances which may also have real-time and embedded requirements. The kernel is flexible and high performance. In addition, it has a hardware abstraction layer which eases the effort of porting the kernel to different hardware platforms. These features make Seed suitable for embedded systems. Moreover, the Seed kernel services have deterministic timing behavior, so it is also suitable for the real-time applications. Finally, a small TCP/IP stack named lwIP was ported to Seed to enable the Internet-access capability.

Seed is currently implemented on Samsung SNDS100 evaluation board. It provides preemptive multitasking, task synchronization/communication, and management of memory, timers and interrupts. The size of the kernel image is about 75Kbytes with lwIP, or 21Kbytes without lwIP. And the interrupt handling latency is about 90 μs for a 50 MHz processor. Besides, the network throughput of lwIP/Seed can reach to 190.54 KB/Sec. These results show that Seed is suitable for non-high speed embedded network appliance that requires real-time support.

5.2 Future work

In the future, we want to implement Earliest Deadline First (EDF) scheduling algorithm on Seed. This is much easier to accomplish because the kernel services are deterministic or constant, and the priority can be dynamically changed at run-time.

Besides, we want to build up embedded file systems and embedded graphic systems on Seed. With these systems, Seed will be more suitable for the embedded devices which are equipped with storage or display. Finally, we would like to port Seed to more hardware platforms to demonstrate its portability.

Reference

[1] Accelerated technology Inc., *Nucleus homepage*, available at http://www.acceleratedtechnology.com/embedded/plus.php, May 2004.

[2] ARM Co. Ltd., *ARM Developer Suite*, available at http://www.arm.com/products /DevTools/ADS.html, May 2004.

[3] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel (2nd Edition)," *O'Reilly*, Dec. 2002.

[4] L. Deller and G. Heiser, "Linking Programs in a Single Address Space," *In Proceedings of 3rd Symposium on Operating Systems Design and Implementation, USENIX, pp. 283-294*, Feb. 1999.

[5] A. Dunkels, "Design and Implementation of the LWIP TCP/IP Stack," *Technical Report*, Feb. 2001.

[6] A. Dunkels, "Full TCP/IP for 8-Bit Architectures," *In Proceedings of the first international conference on mobile applications, systems and services, pp. 85-98*, May 2003.

[7] A. Dunkels, "lwIP – a lightweight TCP/IP stack," available at http://www.sics.se/ ~adam/lwip/, May 2004.

[8] Finite State Machine Labs Inc., *RTLinux RTOS*, available at http://www.rtlinux.org/.

[9] S. B. Furber, "ARM System-on-Chip Architecture (2nd Edition)," *Addison-Wesley*, Aug. 2000.

[10] J. J. Labrosse, "MicroC/OS II: The Real Time Kernel," *CMP Books*, June 2002.

[11] Lynuxworks Inc., *LynxOS homepage*, available at http://www.lynuxworks.com/rtos/lynxos.php3, May 2004.

[12] Metrowerks Inc., "Linux Solutions", available at http://www.metrowerks.com/MW/Develop/Embedded/Linux/default.htm, May 2004.

[13] Microsoft Inc., *Windows CE homepage*, available at http://www.microsoft.com/embedded/, May 2004.

[14] Mindcraft Inc., "Webstone: The Benchmark for Web Servers", available at http://www.mindcraft.com/benchmarks/webstone/, May 2004.

[15] MontaVista Software Inc., "MontaVista Linux", available at http://www.mvista.com/, May 2004.

[16] QNX Software System Inc., *QNX homepage*, available at http://www.qnx.com/, May 2004.

[17] Real Time Linux Foundation Inc., *Real Time Linux Foundation homepage*, available at http://www.realtimelinuxfoundation.org/.

[18] Redhat Inc., "eCos RTOS," available at http://sources.redhat.com/ecos/, Apr.

2004.

[19] Samsung Electronics Co. Ltd., “Application Note for S3C4510B,” available at http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/an_s3c4510b.pdf, Oct. 2001.

[20] Samsung Electronics Co. Ltd., “User’s Manual Rev. 1.0 for S3C4510B,” available at http://www.samsung.com/Products/Semiconductor/0020SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/ums3c4510b_rev1.pdf, Oct. 2001.

[21] D. Seal, “ARM Architecture Reference Manual (2nd Edition),“ *Addison-Wesley*, Dec. 2001.

[22] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *In IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175-1185*, Sep. 1990.

[23] Windriver Inc, *VxWORKS homepage*, available at http://www.windriver.com/products/device_technologies/os/vxworks5/, May 2004.