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

Chun-Chiao Wang Da-Wei Chang Ruei-Chuan Chang
Department of Computer and Information Science,
National Chiao Tung University, HsinChu, Taiwan, R.O.C.
is87074@cis.nctu.edu.tw david@os.nctu.edu.tw rc@cc.nctu.edu.tw

***Abstract**- Traditional embedded operating systems usually address two issues: limited hardware resources and real-time support. However, due to the popularity of Internet and rapid development of network technologies, Internet access capability is also becoming a necessarily for many embedded systems. As a result, modern embedded operating systems should satisfy the requirements of running on 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 develop an open source, embedded real-time operating system with Internet access capability. The kernel is named Seed. It is small, flexible, and portable. And, the kernel services have deterministic or even constant timing behavior so that it can satisfy the real-time needs. Finally, it enables Internet access by integrating a tiny and open source TCP/IP protocol stack, lwIP.

Seed is currently run on top of the Samsung SNDS100 (ARM7TDMI based) evaluation board. The size of the kernel image is about 75K bytes with lwIP, or 21K bytes without lwIP. According to the performance results, Seed is suitable for real-time embedded network appliances.

Keywords: RTOS, Embedded Systems, Network Appliances.

1. Introduction

Embedded systems play a significant role in modern daily life. They can be found everywhere, such as watches, VCD/DVD players, digital cameras, mobile phones, missile systems, flight control systems, and etc. Traditional embedded operating systems usually address 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. For example, a 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 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 an open source, real-time embedded operating system for network appliances. The operating system, named Seed, contains an OS kernel designed for time-critical embedded applications. Besides the basic kernel services, we also ported a small TCP/IP stack called lwIP [7] to Seed so as to make it become Internet-enabled.

The kernel has the following design goals. First, it is designed to be flexible for supporting various kinds of applications. Second, it provides real-time support. For example, it provides preemptive multitasking and deterministic (or constant) timing services. Third, it is designed for high performance and tiny size. And fourth, Seed is extremely portable. It is easy to port Seed to other hardware platforms by replacing the code under the Hardware Abstraction Layer (HAL).

Seed is currently implemented on Samsung SNDS100 evaluation board. The kernel supports 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, which is small enough for resource-limited systems. And, according to the performance results, Seed is suitable for real-time embedded network appliances.

The rest of the paper is organized as follows. Section 2 describes the previous research related to real-time embedded kernels. Section 3 presents the

details of the Seed kernel and the lwIP porting. The experiment results are shown in Section 4. Finally, Section 5 gives conclusions and future work.

2. Related Work

In this section, 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 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, 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 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 straightforward 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 handling procedure 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 to 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[22], QNX[16], Lynx[11] and etc. 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.

3. Design and Implementation

Before describing the components of the Seed kernel, we present its features first. Seed kernel has the following features:

Flexibility. Seed kernel divides its code into several components for flexibility. Each component can be replaced, removed and modified independently. In addition, we implement a Seed component as flexible as possible. For example, when creating a task, the task management component allows the user to specify the time-slice, whether or not the task can be preempted, and etc. Changing these values at run-time is also allowed.

Deterministic timing. All the Seed kernel services have deterministic or even constant timing behavior. With this, it is possible to analyze the worst case performance of the real-time applications.

Portability. The hardware-dependent code is hidden below the Hardware Abstraction Layer (HAL). If we want to port Seed to another hardware platform, all we have to do is to modify the code below the HAL. Other components do not need to be changed at all.

High performance. Seed chooses single protection mode (i.e., kernel mode) for performance consideration. Traditional operating systems such as Linux adopt 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 study [4], single protection mode can save the time of domain switching.

In the following sections, we will describe the components of the Seed kernel. In addition, we will also present the effort of porting lwIP to Seed.

3.1 Task Management

Seed kernel supports multi-tasking. Each task is associated with a priority, which ranges from 0 to 511 (0 is the highest priority). Seed always schedules the highest priority task to run. If two tasks have the same priority, they will be executed in a round-robin manner. Besides, Seed supports both preemptive and non-preemptive scheduling.

A unique feature of Seed is that it can achieve constant-time scheduling for 512 task priorities. μC/OS-II can also achieve constant-time scheduling. However, it only supports 64 priorities. It is proved that a RTOS should have at least 256 priorities to eliminate most of the unpredictability of the run-time behavior of systems.

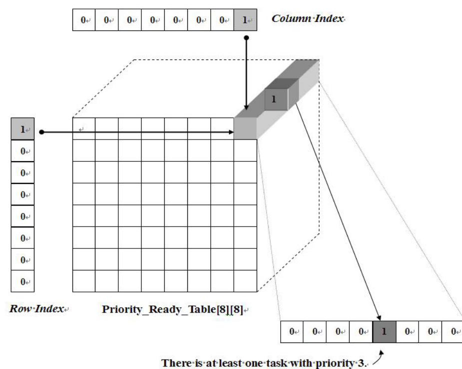

Figure 1. Finding the Highest Priority Task

We extend the μC/OS-II scheduler [10] to achieve constant-time scheduling for 512 task priorities. As shown in Figure 1, we represent 512 task priorities with an 8×8×8 cube (i.e., Priority_Ready_Table). The cube is made up of an 8×8 array, where each element is an 8-bit bitmap. Each 'set' bit indicates the existence of one or more ready tasks with the corresponding priority. For example, the binary value 00001000 in Priority_Ready_Table[0][0] means that there is at least one ready task with priority 3.

The array is referenced by two indexes, row index (*ri*) and column index (*ci*). Each of them is an 8-bit bitmap and each bit corresponds to a priority group. For example, if the bit 0 of ri and the bit 0 of ci are both set, there is at least one task (with its priority between 0 to 7) ready for execution. This is because Priority_Ready_Table [0][0] corresponds to priority 0 through 7.

Therefore, the highest priority task can be found in the following way:

1. Find the least significant bit set in ri → r_{lsbs}
2. Find the least significant bit set in ci → c_{lsbs}
3. Find the least significant bit set in Priority_Ready_Table [r_{lsbs}][c_{lsbs}] → n
4. P := $64 \times r_{lsbs} + 8 \times c_{lsbs} + n$
5. Detach the first task with priority P in the task ready queue

The above procedure requires finding the least significant bit that is set in an 8-bit bitmap. To do this in a constant time, we use a table-lookup approach, which is exactly the same as the approach used in μC/OS-II. As a result, the cost of the task scheduling is fixed no matter how many tasks are in the system.

3.2 Interrupt Management

Seed allows a component such as a device driver to register/un-register an ISR for an IRQ number (interrupt request number) dynamically. When an interrupt occurs, the HAL will recognize the IRQ, save the CPU context, execute the ISR, and finally restore the context.

We usually disable interrupts during the execution of an ISR. However, it is not desirable to disable interrupts for a long time in a real-time system. Therefore, Seed adopts a 2-stage interrupt handling scheme, which is also adopted by some other real-time kernels (e.g., eCos [18]). In this scheme, 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. Just like a task, each DISR has its own stack and control block, and hence it can temporarily be blocked for synchronization or mutual exclusion purpose. Therefore, interrupts will not be disabled for a long time.

The eCos kernel also supports DISR. However, their DISRs do not have priorities, and hence they 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, DISR scheduling only requires a small constant time.

3.3 Timer Management

This component provides all the timing facilities in Seed, including the 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.

We classify the timers into two types according to their usage, the application timers and the task timers. The former can be created, deleted, enabled, and disabled dynamically by the applications. These timers execute user-provided routines when they are

expired. The routines are specified while creating the timers. For the latter, each task has a built-in task timer, which allows a task to suspend for a specified time. When the timer expires, the task will be resumed.

3.4 Memory Management

To avoid the fragmentation problem and achieve constant-time allocation/de-allocation, Seed provides a partition-based memory management mechanism. This mechanism is also adopted by other real-time kernels such as μC/OS-II and Nucleus. It allows the applications to obtain fix-sized memory blocks from a partition, which is made up of a contiguous memory area. All memory blocks in a partition are of the same size.

Figure 2 shows the partition control block that is used to manage a partition. It 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 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.

Each block has a four-byte overhead (i.e., block header) that contains a pointer to the partition control block. This reverse pointer keeps away the application from specifying the partition control block when it de-allocates a block. It is useful since the system may crash if the application returns the block to a wrong partition.

The memory partition implementation in Seed is more robust than μC/OS-II since the former provides a reverse pointer to avoid system crash. On the other hand, a Seed memory partition incurs less space overhead than Nucleus. The latter includes the free list pointers into the block headers so that the pointers cannot be used to store data.

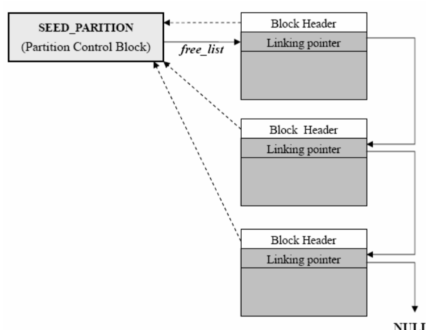

Figure 2. Partition Control Block and Free List

3.5 Message Queue

Message queue is used for inter-task communication. When a task sends a message, the message will be copied into the message queue. Then the receiving task will be able to copy the message out of the queue. To avoid large data copying, we suggest that applications just send *pointers* to the receivers. The pointer can be initialized to point to some application's data structure that will actually be referenced by the receivers.

In addition to unicast communication, it is allowed to broadcast a message to all the waiting tasks in a message queue.

3.6 Semaphore

Seed provides counting semaphores. The value of each semaphore ranges from 0 to 2^{32} -1. 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 the semaphore is available.

Seed supports priority-inheritance protocol [21] for semaphores in order to solve the problem of priority inversion. Note that μC/OS-II kernel does not implement a general priority-inheritance protocol since it can not allow two tasks to share the same priority. Hence, it requires the users to reserve some priority levels for priority-inheritance usage. This decreases the available priorities. Seed kernel does not have such limitation, so that it can implement the general priority-inheritance protocol.

3.7 Kernel Implementation Status

Seed is currently implemented on Samsung SNDS100 evaluation board, which is based on the S3C4510B/KS32C50100 microcontroller [19]. S3C4510B is a 32-bit ARM7TDMI-based [9][20] microcontroller that integrates an Ethernet MAC. And, the maximum processor frequency is 50MHz. Besides the microcontroller, the board also consists of boot EEPROM, DRAM module, SDRAM, serial ports, and Ethernet interface.

Seed works correctly on the SNDS100 board. The Seed HAL is responsible for managing the board. In addition to the HAL, other kernel components such as task management, interrupt management (ISR and DISR), memory management, timer, message queue, and semaphore are also implemented and tested. And, we have implemented two drivers (i.e., Ethernet and UART) for the board. The source code is available at http://rt.openfoundry.org/Foundry/Project/index.html ?Queue=157.

3.8 LWIP Integration

In order to provide Internet access capability, we ported a small TCP/IP stack called lwIP (i.e., lightweight IP) [5][7] to the Seed kernel. The design goal of lwIP is to reduce the memory usage and the code size, making it suitable for embedded systems. lwIP provides an interface called OS emulation layer for connecting it with the underlying OS kernel. To

port lwIP to Seed kernel, we only have to implement this interface. This interface requires the functionalities such as multi-tasking, memory management, timer, semaphore and message queue. These functionalities are fully supported by Seed. Table 1 shows the function mapping between the OS emulation layer and the Seed kernel. Each function in the OS emulation layer is mainly implemented by a single Seed kernel function.

Currently, the following network applications can be run on lwIP/Seed: TCP Echo server, UDP Echo server, HTTP daemon, and telnet daemon.

Table 1. Function mapping between lwIP OS Emulation Layer and Seed kernel

OS Emulation Layer Functions	Seed Kernel Functions
sys_thread_new	Create_Task
sys_mbox_new	Create_Message_Queue
sys_mbox_free	Delete_Message_Queue
sys_mbox_post	Send_Message_To_Queue
sys_arch_mbox_fetch	Receive_Message_From_Queue
sys_sem_new	Create_Semaphore
sys_sem_free	Delete_Semaphore
sys_arch_sem_wait	Obtain_Semaphore
sys_sem_signal	Release_Semaphore

4. Performance Evaluation

4.1 Code Size

The code was compiled for ARM7TDMI using ARM Developer Suite 1.2 [2]. The size of the compiled code is shown in Table 2. The *Code size* column shows the size of the compiled object code, and the *Data size* column shows the size of data used by the object code. The total code size is about 16K bytes and the total data size is about 35K bytes. After linking, the kernel image size is about 21K bytes. Hence, Seed kernel is very small and is suitable for embedded systems.

Table 2. Code Size of Seed Kernel

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

4.2 Kernel Performance

Table 3. Performance of Seed Primary Functions

Function	Time(us)	Cycles
Task_Scheduler	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 3 shows the execution time of the primary functions in Seed. These results can be treated as a reference while creating applications on Seed.

Table 4 gives the performance of interrupt handling. Interrupt handling can be divided into three parts. The first part is *interrupt latency*, which is defined as the time that a system takes to start running the interrupt handling code. The second part is the time to save the CPU context of the current task and branch to the ISR. The third part is *interrupt recovery*. It is the time to determinate if a higher priority task is ready and the time to restore the CPU context. The total latency is about 90us.

Table 4. Performance of Interrupt Handling

Function	Time (us)	Cycles
Interrupt Latency	34.695	1819
Save CPU Context	20.409	1070
Interrupt Recovery	35.667	1870

4.3 Network System Performance

In this section, we measure the performance of lwIP on the Seed kernel. First, the throughput is measured. 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 the TTCP tool to measure the TCP throughput. We configure TTCP to send 8M bytes of data from one device to the other. The result is shown in Table 5. From the table we can see that, Rx is slower since lwIP involves multiple tasks for receiving packets. This leads to more context switches and degrades the performance.

Table 5. Throughput of lwIP Running on Seed

	Throughput
lwIP Rx	115.93 KB/Sec
lwIP Tx	190.54 KB/Sec

Besides the throughput, we also measure the round-trip time. The measurement was taken by using the ping program. We send 1000 64-byte packets to the SNDS100 board and the average round-trip time is 0.991 ms.

The above results are comparable with that reported by the previous study [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 runs on lwIP/Seed. 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 file in ten minutes. Table 6 shows the result, which is acceptable for small embedded devices.

Table 6. Web Server Performance

Connection Rate	39.05 Conn./sec
Throughput	147.20 Kbytes/sec
Ave. Resp. Time	25.59 ms

5. Conclusions and Future Work

In this paper, we describe the internal of Seed, a real-time embedded kernel that has Internet access capability. It supports network appliances that have real-time and embedded requirements. The kernel services have deterministic timing behavior, so it is suitable for real-time applications. Moreover, a small TCP/IP stack named lwIP has been ported to Seed to enable the Internet access capability. Finally, the kernel is flexible and has a hardware abstraction layer to ease the porting effort.

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 75 Kbytes with lwIP, or 21 Kbytes without lwIP. And the interrupt handling latency is about 90 us for a 50 MHz processor. Besides, the network throughput of lwIP/Seed can reach to 190.54 Kbytes/sec. These results show that Seed is suitable for non-high speed embedded network appliances that require real-time support.

In the future, we plan to add some real-time scheduling algorithms such as RM and EDF to Seed. Besides, we will build up embedded file systems and graphic systems on Seed. With these systems, Seed will be suitable for embedded devices that are equipped with storage or display.

6. References

[1] Accelerated technology Inc., “Nucleus RTOS - Nucleus Plus”, 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/, May 2004.

[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 Neutrino RTOS”, available at http://www.qnx.com/download/ download/8483/QNX_Neutrino_RTOS_Brochure.pdf, May 2004.

[17] Real Time Linux Foundation Inc., Real Time Linux Foundation homepage, available at http://www.realtimelinux foundation.org/, May 2004.

[18] Redhat Inc., “eCos RTOS Reference Manual,” available at http://ecos. sourceware.org/docs-latest/ref/ecos-ref.html, Apr. 2004.

[19] 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.

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

[21] 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.

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