Design of ARM Based Embedded Operating System Micro Kernel

Bo Qu
School of Mathematics and Information Technology
Nanjing Xiaozhuang University
Nanjing, China
e-mail: Mr.QuBo@126.com

Zhaozhi Wu
School of Mathematics and Information Technology
Nanjing Xiaozhuang University
Nanjing, China
e-mail: wzz5958@126.com

***Abstract*—This paper describes the design and implementation of an ARM based embedded operating system micro kernel developed on Linux platform with GNU tool chain in technical details, including the three-layer architecture of the kernel (boot layer, core layer and task layer), multi-task schedule (priority for real-time and round-robin for time-sharing), IRQ handler, SWI handler, system calls, and inter-task communication based on which the micro-kernel architecture is constructed. On the foundation of this micro kernel, more components essential to a practical operating system, such as file system and TCP/IP processing, can be added in order to form a real and practical multi-task micro-kernel embedded operating system.**

Keywords-embedded operating system; micro kernel; ARM; multi-task schedule; inter-task communication

I. INTRODUCTION

With the rapid developments of electronic and computer technologies, operating system has already been an essential and necessary component of an embedded system. Although some famous embedded operating systems have been already developed and used, such as VxWorks, QNX, Embedded Linux, Windows CE, uC/OS, eCos, etc., due to the dedicated features, various types of embedded operating systems are needed to meet the requirements of the field [1]. It is obviously necessary and valuable to enhance the research on the design and development of embedded operating systems.

The ARM based embedded operating system micro kernel developed by the author of this paper is just a good attempt, in which the inter-task communication lays the foundation of the micro-kernel. The main contributions of this embedded operating system micro kernel are:

- Both real-time and time-sharing scheduling. As we know, real-time system is an important application field of embedded systems. While on the other hand, time-sharing tasking is also popularly required, e.g. various kinds of Web based embedded control or management systems. The micro kernel described in this paper is designed to implement both kinds of schedules. The priority schedule is used for real-time tasks and round-robin for time-sharing ones.
- Three-layer architecture. The kernel is designed as a three-layer system and all the manipulations of the kernel are elaborately designated to each of them. For example, the boot layer responds for the booting and initialization of the operating system, the core layer for the internal manipulations of the kernel, and the task layer manipulates the kernel level tasks and user level tasks. Inter-task communication routines accomplish the messages transmission among time-sharing tasks. With such architecture, it is in fact designed as a micro kernel, based on which more components can be added to form a practical micro-kernel embedded operating system.
- Beneficial for curriculum teaching and experiment. Embedded system related curriculums are already become necessary components for undergraduate computer majors. The micro kernel described in this paper is designed with GNU tool chain [7] in C and ARM based assembly. The codes count of the entire kernel is only about 3,000 lines, and simple, foundational and extensible. It can be provided to the students for learning and research, therefore is beneficial for curriculum teaching and experiment.

This paper describes the design and implementation of this micro kernel in technical details, including the architecture of the kernel (boot layer, core layer and task layer), multi-task schedule strategy (priority for real-time and round-robin for time-sharing), IRQ handler, SWI handler, system calls, and inter-task communication based on which the micro-kernel architecture is constructed.

II. ARCHITECTURE OF THE EMBEDDED OPERATING SYSTEM MICRO KERNEL

An operating system can be created by partitioning it into smaller pieces, each of which should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions [5]. Generally, the architectures of operating systems are different from each other, and the two well-known types are micro kernel and monolithic kernel.

A micro kernel can be quite small, consisting of only essential and necessary functions of an operating system. Windows and Minix are examples of system with such architectures. At the other extreme, a monolithic kernel may incorporate almost all the functions for an operating system into a very large kernel. Most versions of UNIX, as well as Linux, have this kind of kernels. For embedded operating systems, both architectures are in use.

The embedded operating system micro kernel described in this paper is just a minimum one, as simple as possible for curriculum teaching, the architecture of which is divided into three layers: boot layer, core layer and task layer. The overall architecture of the micro kernel is shown in Fig. 1.

Published by Atlantis Press, Paris, France.

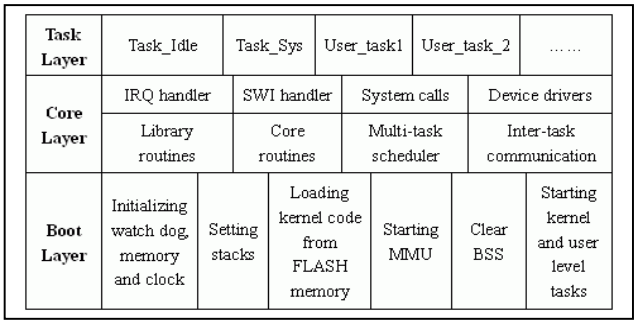

Figure 1. The overall architecture of the embedded OS micro kernel

A. *Boot Layer*

The task of the boot layer is first booting the system from a kind of secondary storage such as FLASH memory, SD card or hard disks, etc., and then provides an appropriate hardware environment to the kernel. Considering the portability, this layer is generally designed as an independent, even separated, module such as that for embedded Linux or Windows CE, etc. [2] on some embedded development boards. In order to accommodate varies kinds of hardware environments, the boot layer described in this paper is designed as a self-adapted built-in boot loader which can recognize the different hardware structure automatically, load the kernel image accordingly, and then boot the system appropriately [6].

The core of the layer includes the initialization of some hardware components such as watch dog, memory and system clock, automatic recognition of the booting address, setting of the stacks for every work mode, loading of the kernel codes, starting MMU, clearing BSS, etc.

This layer consists of two program files, one in assembly and another in C. In fact, the assembly one forms the entire program frame of the micro kernel. In order to load the kernel codes from FLASH memory, some necessary functions to access FLASH memory is designed. Since there is only 4K SDRAM space, called steppingstone [8], at the lower end of the NAND FLASH memory space, it is worthy to note that the crucial codes for booting, including the functions of loading kernel codes, must be within the address range below 4K. Due to the limit of space, it is not described in details further.

B. *Core Layer*

Just as its name implies, the layer consists of the core components of the kernel, including IRQ handler, SWI handler, system calls, multi-task scheduler, inter-task communication, and other essential functions such as initializing kernel, creating task, etc.

The task of the IRQ handler is processing the interrupt requirements of hardware. It is also the foundation of preemptive schedules. While on the other hand, SWI hander processes the software interrupts caused by the kernel, such as system calls.

Multi-task scheduler performs the preemptive schedules among tasks, including two kinds of strategies, i.e. priority for real-time and round-robin for time-sharing.

C. *Task Layer*

This layer is made up of two kinds of tasks, that is, kernel level tasks and user level tasks, in which, Task_Sys is the most important one to perform the communications among other tasks via ITC (Inter-Task Communication) mechanism. In the current stage, it is the only system task besides Task_Idle. Based on ITC, other tasks essential to an embedded operating system such as file system can be added in near future. To show the effect, some user level tasks are also designed which will be described later in this paper.

III. KEY TECHNIQUES FOR DESIGNING AN EMBEDDED OPERATING SYSTEM MICRO KERNEL

A. *Multi-Task Schedule*

For the purpose of research, two typical kinds of multi-task schedule strategies are designed, which are priority for real-time and round-robin for time-sharing.

1) *Structure of task control block (TCB)*

Task control block (TCB) represents the existence of a task. The most important field of TCB is the status of the task, named state, which can be in several statuses such as RUNNING means to be ready to run, SENDING means to send message to other tasks, or RECEIVING means to receive message from another task. For priority schedule, the field prio is used to store the priority value of the task, while for round-robin schedule, two other fields, named slice and ticks, are used to store the designated time slice and its currently remained value, respectively. Other necessary fields are also needed, e.g. that related with TCB table, inter-task communication, etc., which will be described later.

2) *Two schedule strategies*

Priority schedule is designed based on bitmap as the similar way as that in uC/OS [3] so the technical details are not described further.

In order to be compatible with the priority schedule for real-time, the round-robin scheduled tasks are all designated with the lowest priority, named OS_LO_PRIO. That means only when all the real-time tasks are blocked can the time-sharing tasks be scheduled. If no any task is ready then the idle task is scheduled, which in fact only a dummy function made up of a blank loop doing nothing [4].

There are several cases which may result in schedule, e.g. when a higher priority task is ready, a task is created, a task exits, a task is blocked when sending or receiving message, and an IRQ occurs, etc.

3) *Creation of task*

The starting of a task's life cycle is the creation of it including three parts of operations, i.e. initializing task control table (TCB), initializing task stack, and trying a new schedule for the new task.

The first part includes initializing the related fields of TCB, e.g. status and priority. For time-sharing tasks, the time slice fields, message fields and ITC related fields are also needed to be initialized.

Since each task must have its own stack and the corresponding necessary register values must be saved when

task switching occurs, the structure of the task stack is particularly important.

The second part of the task creation is just initializes the stack, the C codes of the function is shown in Fig. 2, in which get_cpsr() is a function in assembly to get the current value of register cpsr.

```
OS_STK *task_stk_init (void (*task)(), void *args,
OS_STK *sp)
{
   *(--sp) = (unsigned int) task;    /* save pc */
   *(--sp) = (unsigned int) task;    /* save lr */
   sp -= 13;
   memset(sp, sizeof(OS_STK) * 13, 0);   /* r0-r12: 0 */
   *(--sp) = (DWORD)args;         /* r0: argument */
   /* save CPSR */
   *(--sp) = (unsigned int)(SYS_Mode & get_cpsr());
   return (sp);                          /* top of stack */
}
```

Figure 2. C codes of initializing task stack

The last part invokes the task-level scheduler and the new task may be selected by the scheduler if the priority of the new task is the highest or the new task is the only ready one besides the idle task.

B. IRQ Handler

The main functions of IRQ are providing clock ticks periodically for schedule and software timer, invoking the scheduler to switch tasks. The performance of the IRQ handler will affect the quality of the entire system therefore it should be designed elaborately as far as possible. The frame of the handler is written in ARM assembly while the core routine is in C.

Since the switching between tasks is implemented by interrupt, the starting of the first task (idle task) and the schedule in the task level (e.g. task creation or task exit) is in fact a kind of simulation of interrupt therefore the codes of which are similar to an IRQ handler. By this reason, the IRQ handler described in this paper is such designed that the two functions, idle task starting and tasks switching as mentioned above, are closely combined with the IRQ handler together so that the codes are pithy and compact.

The handler invokes the interrupt handler routine in C first, the return value of which demonstrates whether task switching is needed (both real-time and time-sharing). If it is, then some related values are stored and then a C routine, named task_renew(), is invoked, the return value of which is the TCB of the new task. After the stack pointer being set to the new task's stack, the interrupt returns and the task switching is accomplished.

C. SWI Handler

SWI handler is the foundation of an embedded operating system micro kernel by which the system calls and inter-task communication can implemented.

It is worthy to note that the parameters of SWI are stored in r0, r1, r2 and r3 when the number of the parameters is not more than 4. That means in such a case the parameters can be used directly by invoked functions instead of being pushed onto stack. Since register r0 will be used to store the return value, it needs not to be saved therefore only r1-r12 and r14 are pushed onto stack to be stored.

The real processing of SWI is implemented by a function in C, named sysc_sched() which is in fact the main routine of system calls. After the function being invoked, the return value is stored in register r0.

The system calls for the micro kernel described in this paper accomplishes two functions, i.e. formatted output, named cprintf() and inter-task communication. With cprintf(), user tasks can display message via UART without worry about synchronization and exclusion.

D. Inter-Task Communication (ITC)

Inter-task communication is an important mechanism for time-sharing multitasking environment. To implements inter-task communication [5], some fields in TCB are needed.

- p_msg: a pointer pointing to the buffer of message. The buffer is provided by tasks sending or receiving message instead of the kernel, and all the related message buffers is linked one by one to form a message queue. Obviously, the length of such a queue is not limited. That means the mechanism of inter-task communication is different from that of mail box or message queue for real-time tasks.
- recv: task ID of another task from which task T waits to receive message.
- send: task ID of another task to which task T sends the message but the destination has not gotten it yet.
- q_send: if there are several tasks, say A, B, and C, sending messages to task T while the task T has not prepared to receive them, the tasks A, B, and C will form a queue as mentioned above and q_send of task T points to the first task in the queue.
- q_next: The three tasks, A, B and C as just mentioned above form the queue according to the time sequence. Assume the destination task is T, then the field q_send in the TCB of T points to A, the q_next in the TCB of A points to B, that of B points to C, and that of C points to NULL. Obviously, by the TCB field q_next of each task, a link table of tasks is formed and the field q_send of task T is the head pointer of the table through which the messages can be received sequentially.

When a task is sending or receiving message via inter-task communication, its TCB field state may be in one of the following three statuses:

- RUNNING: the task is running or ready to run;
- SENDING: the task is in the status of sending message. Because the message is yet not arrived the destination, the task is blocked;
- RECEIVING: the task is in the status of receiving message. Because the message is yet not received, the task is blocked.

The algorithms of inter-task communication for the micro kernel in this paper are described in Fig. 3 and Fig. 4.

Prepare the message M by source task A.
Invoke function msg_send() via interface function sendrecv().
Check for whether a deadlock occurs.
Check for whether the destination task B is waiting for receiving message from source task A.
If yes, the message is copied to task B and then task B leaves the blocked state to continue to run.
Otherwise, task A is blocked and added to the sending queue of task B.

Figure 3. The algorithm for task A sending message M to task B

Prepare a blank structure M to receive message by destination task B.
Invoke function msg_receive() via interface function sendrecv().
If task B is going to receive message from any task, then gets the first one from its sending queue (just mentioned above) if it exists and then copy it to M.
If task B is going to receive message from a particular task A, the first thing to do is checking for whether task A is waiting for sending message to task B. If yes, copy the message from A to message structure M.
If there is no task sending message to task B, the task B is blocked.

Figure 4. The algorithm for task B receiving message

It is worthy to note that each task maintains a message structure no matter it is sending or receiving except that the one for sending is filled with a message while the one for receiving is blank. With such a synchronization mechanism of ITC, only a task's requirement is met can it be continue to run, otherwise it will be blocked with its status being designated as SENDING or RECEIVING. This kind of communication strategy is often known as "rendezvous". Since no any extra message buffer needs to be maintained by the kernel, the implementation will be relatively simple.

IV. A Demo of Inter-Task Communication

To show the effect of inter-task communication using the micro kernel, some real-time as well as time-sharing tasks are designed, each of which periodically show a piece of message and then delay a while. The delay for the real-time tasks is implemented by the IRQ handler while that for time-sharing tasks is by a function, get_ticks(), based on inter-task communication (ITC). Fig. 5 shows the C codes of the function and Fig. 6 shows the displays on the super terminal for the running of all the tasks.

```
int get_ticks()
{
   MESSAGE msg;
   reset_msg(&msg); msg.type = GET_TICKS;
   send_recv(TASK_SYS, &msg);
   return msg.RETVAL;
}
```

Figure 5. Codes of get_ticks()

```
############################################################
     Embedded Operating System Micro-kernel on ARM
          Author: Bo Qu   <http://www.qu99.net>
------------------------------------------------------------
   PR Sched.=> (1): TaskA (2): TaskB (3): TaskC
   RR Sched.=> (4): Echo key strokes via serial port
               (5): TaskE (6): TaskF
############################################################
  TaskE[8]           Timer0[15]           TaskF[3]
                      Key: a
  TaskA[6]           TaskB[11]            TaskC[16]
```

Figure 6. Displays of the tasks

V. Conclusion

Operating system is the essential component for general computer as well as embedded systems, of which the performance will affect directly the quality of the system. By this reason, it has been being a hot topic worthy to be researched deeply.

The micro kernel described in this paper is developed on Linux platform with GNU tool chain, the purpose of which is to implement a pithy and compact ARM based embedded operating system micro kernel. Based on such a kernel, other components such as file system, network routines, etc., can be added to form a practical multi-task micro-kernel embedded operating system. Due to limitations on space, some technical details have been left out.

References

[1] A. N. Sloss, D. Symes and C. Wright, ARM System Developer's Guide: Designing and Optimizing System Software, Elsevier Inc, 2004

[2] T. Noergaard, Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers, Elsevier Inc, 2005

[3] J. J. Labrosse, Micro C/OS-II the Real-Rime Kernel, 2e, CMP Media LLC, 2002

[4] M. Barr and A. Massa, Programming Embedded Systems, Second Edition, O'Reilly Media, Inc., 2006

[5] A. S. Tanenbaum and A. S. Wookhull, Operating Systems: Design and Implementation, 3E, Prentice Hall, Inc., 2008

[6] B. Qu, Design of Built-in Boot Loader for ARM uCOS, Proceedings of 2012 IEEE International Conference on Computer Science and Automation Engineering, vol 1, pp. 429-433, 2012

[7] Stallman R M 2002 Using the GNU Compiler Collection (http://www.gnuarm.com/pdf/gcc.pdf)

[8] ARM limited 2005 ARM Architecture Reference Manual