The theory and implementation of 128 tasks expansion in μC/OS-II

Li Dan and Sui Zhengwen

School of Information and Computer Engineering, Northeast Forestry University, Harbin, China

ABSTRACT

μC/OS-II is an open source kernel and has a preemptive scheduling strategy based priority, can only create 64 tasks. With the increasing development of embedded system, the current problem which must be solved is the more complex tasking support. So from studying of the μC/OS-II kernel structure and scheduling policy, a new task extended method is proposed in this paper, and form rewritten μC/OS-II kernel, this paper increases number of tasks to 128. It increases the application areas of μC/OS-II and improves its applicability.

Key words: RTOS; μC/OS-II; scheduling; expansion task

INTRODUCTION

μC/OS II is a kind of RTOS which is portable, as a general real-time operate system, it is cabinet, source code opened, real-time, easy to be planted, multitasking and has the preemptive schedule based on priority. It’s used for microprocessor, micro controller, Digital signal processor, as a RTOS which is passed by the standard certification of the federal aviation administration (FAA) for commercial aircraft and meet ‘DO178B’ from RTCA, and it is better than most other real-time embedded operator systems in the security and stability. The goal of μC/OS II is a Real Time Kernel which is based on priority scheduling, and it provides basic systematic service upon the kernel, such as semaphore, postage, message queue, memory manager, interruption manager, and so on [1-3]. A real-time system not only requires that the computing result is correct, but also the correct result must be completed within a predetermined time [4]. For real time systems, the time to accomplish tasks can be predicted in the design of the application. And in this context, μ C/OS II can manage 64 tasks. μ C/OS II can provide 56 tasks to users, because the 4 highest priority tasks and the 4 lowest priority are reserved for itself, and four tasks of the highest and the lowest priorities will be retained in system. The higher the task priority is, the lower the number of priority reflection will be. The task priority level can be used as the identifier of task. So whether in teaching or in the practical engineering application process, it is easily to meet the bottleneck problems when doing the embedded programming designing. For solving the problem, Combined with the author's teaching and the experience of engineering practice, this paper proposed a method to expand the μ C/OS II for more tasks limited. We can expand the priorities of it to provide more time and room for the embedded programming designing [5].
Fig. 1: The task status and switching functions

The maximum number of tasks is defined as a constant (OS_MAX_TASKS) that have been specified in OS_CFG.H and determines the number of OS_TCBs allocated by μC/OS-II for your application. You can reduce the amount of RAM needed by setting OS_MAX_TASKS to the number of actual tasks needed in your application. All OS_TCBs are placed in OSTCBTbl[] as shown in fig 2. When μC/OS-II is initialized, all OS_TCBs in this table are linked in a singly linked list of free OS_TCBs. When a task is created, the OS_TCB pointed to by OSTCBFreeList is assigned to the task and, OSTCBFreeList is adjusted to point to the next OS_TCB in the chain. When a task is deleted, its OS_TCB is returned to the list of free OS_TCBs. μC/OS-II always executes the highest priority task ready to run. The determination of which task has the highest priority and thus, which task will be next to run is determined by the scheduler. Task level scheduling is performed by OSSched(). ISR level scheduling is handled by OSIntExit()[6].

OSSched() verifies that the highest priority task is not the current task. Note that μC/OS used to obtain OSTCBHighRdy and compared it with OSTCBCur. On 8 and some 16-bit processors, as shown in fig 2, this operation was relatively slow because comparison was made on pointers instead of 8-bit integers as it is now done in μC/OS-II. Also, there is no point of looking up OSTCBHighRdy in OSTCBPrioTbl[] unless we actually need to do a context switch. The combination of comparing 8-bit values instead of pointers and looking up OSTCBHighRdy only when needed should make μC/OS-II faster than μC/OS on 8-bit and some 16-bit processors. The task code executes every second and basically determines how much CPU time is actually consumed by all the application tasks. CPU utilization is stored in the variable OSCPU Usage and is computed as follows [7-8]:

Wait status

Dormant status

Ready status

Running status

ISR status

OSTaskCreate()  
OSTaskCreateExt()  
OSTaskDel()  
OSTaskDel()  
OSTaskResume()  
......

OSTaskDel()  
OSMBoxPend()  
OSQPost()  
OSSemPost()  
OSTaskSuspend()  
......

OSTaskDel()  
OSMBoxPost()  
OSQPost()  
OSSemPost()  
OSIntExit()  
OSMBoxPost()  
OSQPost()  
OSSemPost()  
OSIntExit()  
OSMBoxPost()  
OSQPost()  
OSSemPost()  
OSIntExit()  
OSMBoxPost()  
OSQPost()  
OSSemPost()  
OSIntExit()  
OSMBoxPost()  
OSQPost()  
OSSemPost()  
OSIntExit()
OSSched() verifies that the highest priority task is not the current task. Note that µC/OS used to obtain OSTCBHighRdy and compared it with OSTCBCur. On 8 and some 16-bit processors, as shown in fig 2, this operation was relatively slow because comparison was made on pointers instead of 8-bit integers as it is now done in µC/OS-II. Also, there is no point of looking up OSTCBHighRdy in OSTCBPrioTbl[] unless we actually need to do a context switch. The combination of comparing 8-bit values instead of pointers and looking up OSTCBHighRdy only when needed should make µC/OS-II faster than µC/OS on 8-bit and some 16-bit processors. The task code executes every second and basically determines how much CPU time is actually consumed by all the application tasks. CPU utilization is stored in the variable OSCPU Usage and is computed as follows:

\[
\text{OSCPU Usage}(\%) = 100 \times \left(1 - \frac{\text{OSIdleCtr}}{\text{OSIdleCtrMax}}\right)
\]  

**Fig. 2: The structure of task ready table**

**The Achievement of Task Expanded Mechanism**

In the view of that the management methods and systematic architecture shows that the task scheduling mechanism is based on task ready table, the aspects should be taken into considered as follows:

1. To keeping the original systematic architecture and developing the reuse and portability, we should avoid modifying the code plenty as less as we can.
2. The tasks’ schedule of OS is very frequent, so we should use the less time and room to realize the more functions as possible as we can.
3. Considering the characters of ARM, it can take turns range left bits to right bits before the operands coming into the ALU, this enhance the flexibility of data processing significantly. Pretreatment or shift occurred in the instruction cycle, 8-bit and 16-bit data should be expand into 32-bit before they are loaded into the ARM registers, this means that the less loading data does not save the time of instructions, the ARM core is efficient with the functions of conditional executive instructions.
If we want to expand the tasks’ priorities under keeping the original OS architecture, we should modify the OSRdyTbl[] list properly to support 128 tasks’ priorities, meanwhile we also should adjust the bits of OSRdyGrp, OSMapTbl[], and OSUnMapTbl[] which are increased linearly, it only has the two status from the Ready Group list, it will find the highest priority when starting scheduling the tasks, it usually find the bit which is one from up to down and from right to left, so there should be $2^n$ numbers int OSUnMapTbl[] table, it’s not feasible apparently, so what we should do is making the program better, to develop the system performance. The UC/OS II kernel is written by Jean J.Labrosse, according to the characters of ARM, at first, we should change the bits of the variable digits, that is to say we should adjust the OSRdyTbl[] to 16 bits, and the OSMapTbl[] also should change to 16 bits, for maintaining the system architecture as possible as much, the other variable digits which related to them are also should be changed into 16 bits, the system can afford well, when the tasks are initializing the OSTCBCur. OSTCBX and OSTCBBY are calculate at one time, when it is calculating the highest priority, it’s no need to changing the OSUnMapTbl[] only shifting the correct bits the priority’s lower 4 bits stands for the column of the ready task, and the higher 4 bits stands for the row of the ready task, so we only need modify the original shift rules, as following changes are based on above.

```c
INT16U const OSMapTbl[] = {
    0x0001, 0x0002, 0x0004, 0x0008,
    0x0010, 0x0020, 0x0040, 0x0080,
    0x0100, 0x0200, 0x0400, 0x0800,
    0x1000, 0x2000, 0x4000, 0x8000};
```

```c
if((OSRdyTbl[OSIntExitY] & 0x00FF)==0)  //OSIntExit (void)
    x = OSUnMapTbl[OSRdyTbl[OSIntExitY] >> 8] + 8;
else
    x = OSUnMapTbl[OSRdyTbl[OSIntExitY] & 0x00FF];
OSPrioHighRdy = (INT8U)((OSIntExitY << 4) + x);
```

In the OSStart(void) and OS_Sched(void), doing the same modify, then in OS_TCBInit(), we should do following changes as ptcb->OSTCBY = prio >> 4; ptcb->OSTCBX = prio & 0x0F;

Do the modify in uCOS_ILH as follows

```c
#define OS_EVENT_TBL_SIZE ((OS_LOWEST_PRIO) / 16 + 1)
#define OS_RDY_TBL_SIZE ((OS_LOWEST_PRIO) / 16 + 1)
INT16U OSTCBByX, INT16U OSTCBBnY;

OS_EXT INT16U OSDtyTbl[OS_Rdy_TBL_SIZE];
extern INT16U const OSMapTbl[];
#if OS_MAX_TASKS > 127
    #error "OS_CFG.H, OS_MAX_TASKS must be <= 127"
#endif
```

It’s so necessary to change that the variabilities of *psrc and *pdest which are related to OSEventTbl[] into INT16. At last, we need modifying the OS_CFGH so as to the highest priority and the lowest priority.

```c
#define OS_MAX_TASKS   100
#define OS_LOWEST_PRIO 113
```

Because the lowest two priorities is used by the idle Rand the statistical task, what’s more, the system also will keep the highest few priorities to provide for itself to use, so the number of priorities we can use is not as many as 128.

**THE ACTUAL SIMULATION OF PC**

The simulation process is based in BC31 under in 80X86 with windows OS. Firstly, we should configure the environment variability which the bcc.exe’s path before using the actual application simulation simplified BC31, then release the source code of μ C/OS II in the path of C:\software\, there are the source codes in the document which is named Source and embedded codes in the document of Ix86L. Secondly, we should create the directory which is named of TEST.c, then create the documents named by SOURSE and TEST, there are header files, configuration files, links, files and Source codes in the document of SOURSE, and we also should create the property documents of MAK files and BAT files, the make out the executive files using by BAT files. At last, run the TEST.EXE file to accomplish the actual simulation. The main function defines 65 tasks, the priorities which is ranging from 0 to 64, and when we define the stacks’ room we should pay attention that the length of OS_STK is 16 bits, that is 64KB, so we can define the stacks’ room as 128 because we should assign many consecutive stacks’ room, but we must focus the overflow of room. Every task is output in every one second using the command of windows, the scheduling simulation Result is shown in fig 3.
CONCLUSION

As a general RTOS, with the characteristics of cabinet, source code opened, real-time, easy to be planted, multitasking, preemptive schedule based on priority. μC/OS-II is more and more common in the teaching process and industrial control. For improving the system's performance, μC/OS-II is used as the research object in this paper, the paper mainly studied three aspects as follows, firstly we researched data structure and scheduling kernel in μC/OS-II, Secondly we proposed an improved method of the task expansion from 64 to 128. At last we realized the extended algorithm form editing the task control blocks (TCB), the task communication and synchronization function. In keeping with the original system under the principle of maximum compatibility, we transformed and upgraded the μC/OS-II kernel. The improved version of the kernel in the application verification, the system is stable and reliable. The improved μC/OS-II real-time kernel not only keeps the real-time kernel’s characteristics, but also expands the application range, improves its applicability.

Acknowledgments
This work has been supported by the Fundamental Research Funds for the Central Universities Nos. 2572014CB24 and Heilongjiang Natural science fund in China Nos.F201116.

REFERENCES