

uC/OS-II 在 SkyEye 上的移植分析

李明 lmcs00@mails.tsinghua.edu.cn

SkyEye 仿真调试器是基于 ARM7TDMI 核的，因此移植 uC/OS-II 到 SkyEye 上可以借鉴网上已有的例如 Samsung S3C3410X 的移植代码，这在 uC/OS-II 的主页上很容易找到。当然自己动手做移植也是对 ARM 体系结构和汇编语言的进一步熟悉，同时对于 uC/OS-II 内核的调度机制会有更深的认识。

整个移植工作可以分为两个方面，一部分是和 ARM 相关，一部分是和移植原理相关。在开始实际的移植工作前，需要对这两部分有一定的背景知识，尤其是和侧重于和移植工作相关的概念和原理。下面分别做一些介绍：

一、ARM 的体系结构

ARM (Advanced RISC Machines) 是目前在嵌入式领域里应用最广泛的 RISC 微处理器结构，以其低成本、低功耗、高性能的特点占据了嵌入式系统应用领域的领先地位。ARM 系列的处理器当前有 ARM7、ARM9、ARM9E、ARM10 等多个产品，此外 ARM 公司合作伙伴，例如 Intel 也提供基于 XScale 微体系结构的相关处理器产品。所有的 ARM 处理器都共享 ARM 通用的基础体系结构，所以开发者在不同的 ARM 处理器上做操作系统移植时，可以将节省相当多的工作量，这无疑将大大降低软件开发成本。

要详细完整的了解 ARM 的体系结构，当然是去读 ARM Architectur Reference Manual，这是一个 13M 的 pdf 文档，有 800 多页，可以从 ARM 的网站下载，也可以到阿卡嵌入式兴趣小组的 FTP 服务器(<ftp://159.226.40.150>)上找到。北航出的一本《ARM 嵌入式处理器结构与应用基础》基本上翻译了这个 pdf 中大部分重要的内容，可以作为入门的中文教材。这里我们仅仅对其中和移植工作密切相关的概念做简要介绍，

1、处理器模式：(cpu mode)

ARM 的处理器可以工作在 7 种模式，如下图所示：

Table 2-1 ARM version 4 processor modes

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks (ARM architecture version 4 and above)

这里除 usr 模式以外的其他模式都叫做特权模式，除 usr 和 sys 外的其他 5 种模式叫

做异常模式。在 `usr` 模式下对系统资源的访问是受限制的，也无法主动地改变处理器模式。异常模式通常都是和硬件相关的，例如中断或者是试图执行未定义指令等。这里需要强调的是和移植相关的两种处理器模式：`svc` 态和 `irq` 态，分别指操作系统的保护模式和通用中断处理模式。这两种模式之间的转换可以通过硬件的方式，也可以通过软件的方式。`uC/OS-II` 内核在执行过程中，大部分时间都是工作在 `svc` 态，当有硬件中断，例如时钟中断到来时，`cpu` 硬件上会自动完成从 `svc` 态进入 `irq` 态，在中断处理程序的结束处，则需要通过编程的方法使得 `cpu` 从 `irq` 态恢复到 `svc` 态，这个在移植代码中可以找到。

2、程序状态寄存器：（PSR：Program status register）

在任何一种处理器模式中，都使用同一个寄存器来标识当前处理器的工作模式：这个寄存器叫做 CPSR（Current Program Status Register），它的 [0-4] 位用来表示 `cpu mode`：

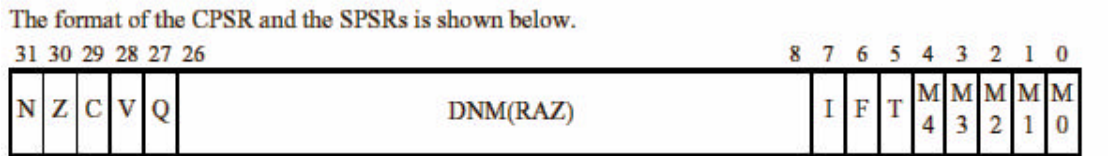


Table 2-2 The mode bits

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

每一种处理器异常模式，都有一个对应的 SPSR（Saved Program Status Register）寄存器，用来保存进入异常模式前的 CPSR。SPSR 的作用就是当从异常模式退出时，可以通过一条简单的汇编指令就能够恢复进入异常模式前的 CPSR，而这个值都是保存在当前异常模式的 SPSR 中的。例如：当从 `usr` 态进入中断 `irq` 态时，原先的 `CPSR_all` 将被保存在当前的 `SPSR_irq` 中，类似的异常模式下的 SPSR 还有 `SPSR_fiq`、`SPSR_svc`、`SPSR_abt`、`SPSR_und`。非异常模式的 `usr` 和 `sys` 模式下没有 SPSR，只有 CPSR。不能显式的指定把 CPSR 保存到某个异常模式下的 SPSR，比如 `SPSR_irq`，而必须是变更到 `irq` 态之后 `cpu` 自动完成的，不能在其他态下硬性赋值，因为 `SPSR_irq` 是其他状态下不可见的。

3、ARM 寄存器：（register）

ARM 处理器一共有 37 个寄存器，其中 31 个是通用寄存器，包括一个程序计数器 PC。另外 6 个就是上面提到的程序状态寄存器。

a）通用寄存器：

- i. R0 - R7：与所有处理器模式无关的寄存器，可以用作任何用途。
- ii. R8 - R14：与处理器模式有关的寄存器，在不同的模式下，对应到不同的物理寄存器。其中 R13 又叫做 `sp`，一般用于堆栈指针。R14 又叫做 `lr`，一般用于保存返回地址。这两个寄存器在每种异常模式下都对应到不同的

物理寄存器上，例如 lr_irq、lr_svc、lr_fiq 等。


iii. R15：又叫做程序计数器，即 pc，所有的模式下都使用同一个 pc。

b) 状态寄存器：

- i. CPSR：当前程序状态寄存器，所有的模式下都使用同一个 CPSR。
- ii. SPSR：保存的程序状态寄存器，每种异常模式下都有自己的 SPSR，一共有 5 种 SPSR，即 SPSR_irq、SPSR_fiq、SPSR_svc、SPSR_abt、SPSR_und。usr 和 sys 态下没有 SPSR。

所有的 ARM 寄存器的命名和含义，可以用下面的这张表来说明，其中相同命名的都是同一个物理寄存器，不同命名的寄存器都对应不同的物理寄存器。

Modes						
Privileged modes						
Exception modes						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

 Indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

二、uC/OS-II 移植工作介绍

uC/OS-II 实际上可以简单地看作是一个多任务的调度器，在这个任务调度器之上完善并添加了和多任务操作系统相关的一些系统服务，如信号量、邮箱等。它的 90%的代码都是用 C 语言写的，因此只要有相应的 C 语言编译器，基本上就可以直接移植到特定处理器上，这也是 uC/OS-II 具有良好的可移植性的原因。移植工作的绝大部分都集中在多任务切换的实现上，因为这部分代码主要是用来保存和恢复处理器现场（即相关寄存器），因此不能用 C 语言，只能使用特定的处理器汇编语言完成。

uC/OS-II 的全部源代码量大约是 6000 - 7000 行，一共有 15 个文件。将 uC/OS-II 移植到 ARM 处理器上，需要完成的工作也非常简单，只需要修改三个和 ARM 体系结构相关的文件，代码量大约是 500 行。以下分别介绍这三个文件的移植工作：

1、OS_CPU.H 文件

▪ 数据类型定义

这部分的修改是和所用的编译器相关的,不同的编译器会使用不同的字节长度来表示同一数据类型,比如 int,同样在 x86 平台上,如果用 GNU 的 gcc 编译器,则编译为 4 bytes,而使用 MS VC++ 则编译为 2 bytes。我们这里使用的是 GNU 的 arm-elf-gcc,这是一个免费并且开放源码的编译器。相关的数据类型的定义如下:

```
/*
*****
*
*                               DATA TYPES
*                               (Compiler Specific)
*****
*/

typedef unsigned char  BOOLEAN;

typedef unsigned char  INT8U;           /* Unsigned  8 bit quantity */
typedef signed   char  INT8S;           /* Signed    8 bit quantity */

typedef unsigned short INT16U;          /* Unsigned 16 bit quantity */
typedef signed   short INT16S;          /* Signed   16 bit quantity */

typedef unsigned long  INT32U;          /* Unsigned 32 bit quantity */
typedef signed   long  INT32S;          /* Signed   32 bit quantity */
typedef float         FP32;             /* Single precision floating point */
typedef double        FP64;             /* Double precision floating point */
```

▪ 堆栈单位

因为处理器现场的寄存器在任务切换时都将会保存在当前运行任务的堆栈中,所以 OS_STK 数据类型应该是和处理器的寄存器长度一致的。

```
typedef unsigned int  OS_STK;           /* Each stack entry is 32-bit wide */
```

▪ 堆栈增长方向

堆栈由高地址向低地址增长,这个也是和编译器有关的,当进行函数调用时,入口参数和返回地址一般都会保存在当前任务的堆栈中,编译器的编译选项和由此生成的堆栈指令就会决定堆栈的增长方向。

```
/* stack stuff */
#define OS_STK_GROWTH    1
```

▪ 宏定义

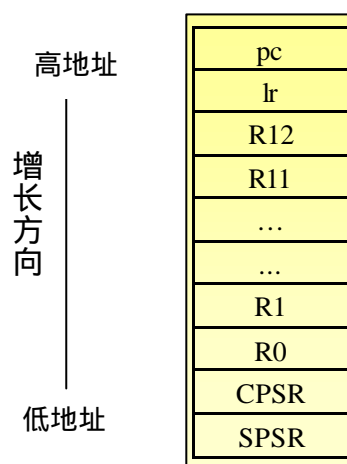
包括开关中断的宏定义,以及进行任务切换的宏定义。

```
#define OS_ENTER_CRITICAL()  ARMDisableInt()
#define OS_EXIT_CRITICAL()   ARMEEnableInt()
#define OS_TASK_SW()         OSCtxSw()
```

2、OS_CPU_C.C 文件

▪ 任务堆栈初始化

这里涉及到任务初始化时的一个堆栈设计,也就是在堆栈增长方向上如何定义每个需要保存的寄存器位置,在 ARM 体系结构下,任务堆栈空间由高至低依次将保存着 pc、lr、r12、r11、r10、...r1、r0、CPSR、SPSR。



```

void *OSTaskStkInit (void (*task) (void *pd), void *pdata, void *ptos, INT16U opt)
{
    unsigned int *stk;

    opt    = opt;
    stk    = (unsigned int *)ptos;          /* 'opt' is not used, prevent warning
                                           /* Load stack pointer

    /* build a context for the new task */
    *--stk = (unsigned int) task;          /* pc */
    *--stk = (unsigned int) task;          /* lr */
    *--stk = 0;                            /* r12 */
    *--stk = 0;                            /* r11 */
    *--stk = 0;                            /* r10 */
    *--stk = 0;                            /* r9 */
    *--stk = 0;                            /* r8 */
    *--stk = 0;                            /* r7 */
    *--stk = 0;                            /* r6 */
    *--stk = 0;                            /* r5 */
    *--stk = 0;                            /* r4 */
    *--stk = 0;                            /* r3 */
    *--stk = 0;                            /* r2 */
    *--stk = 0;                            /* r1 */
    *--stk = (unsigned int) pdata;          /* r0 */
    *--stk = (SVC32MODE|0x0);              /* cpsr  IRQ, FIQ disable*/
    *--stk = (SVC32MODE|0x0);              /* spsr  IRQ, FIQ disable */

    return ((void *)stk);
}

```

这里需要说明两点，一是当前任务堆栈初始化完成后，OSTaskStkInit 返回新的堆栈指针 stk，在 OSTaskCreate () 执行时将会调用 OSTaskStkInit 的初始化过程，然后通过 OSTCBInit () 函数调用将返回的 sp 指针保存到该任务的 TCB 块中。二是初始状态的堆栈其实是模拟了一次中断发生后的堆栈结构，因为任务被创建后并不是直接就获得执行的，而是通过 OSSched () 函数进行调度分配，满足执行条件后才能获得执行的。为了使这个调度简单一致，就预先将该任务的 pc 指针和返回地址 lr 都指向函数入口，以便被调度时从堆栈中恢复刚开始运行时的处理器现场。

■ 系统 hook 函数

此外，在这个文件里面还需要实现几个操作系统规定的 hook 函数，如下：

```

OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()

```


如果没有特殊需求，则只需要简单地将它们都实现为空函数就可以了。

3、OS_CPU_A.S 文件

▪ OSStartHighRdy ()

此函数是在 OSStart () 多任务启动之后，负责从最高优先级任务的 TCB 控制块中获得该任务的堆栈指针 sp，通过 sp 依次将 cpu 现场恢复，这时系统就将控制权交给用户创建的该任务进程，直到该任务被阻塞或者被其他更高优先级的任务抢占 cpu。该函数仅仅在多任务启动时被执行一次，用来启动第一个，也就是最高优先级的任务执行，之后多任务的调度和切换就是由下面的函数来实现。

▪ OSCtxSw ()

任务级的上下文切换，它是当任务因为被阻塞而主动请求 cpu 调度时被执行，由于此时的任务切换都是在非异常模式下进行的，因此区别于中断级别的任务切换。它的工作是先将当前任务的 cpu 现场保存到该任务堆栈中，然后获得最高优先级任务的堆栈指针，从该堆栈中恢复此任务的 cpu 现场，使之继续执行。这样就完成了一次任务切换。

▪ OSIntCtxSw ()

中断级的任务切换，它是在时钟中断 ISR (中断服务例程) 中发现有高优先级任务等待的时钟信号到来，则需要在中断退出后并不返回被中断任务，而是直接调度就绪的高优先级任务执行。这样做的目的主要是能够尽快地让高优先级的任务得到响应，保证系统的实时性能。它的原理基本上与任务级的切换相同，但是由于进入中断时已经保存过了被中断任务的 cpu 现场，因此这里就不用再进行类似的操作，只需要对堆栈指针做相应的调整，原因是函数的嵌套。

▪ OSTickISR ()

时钟中断处理函数，它的主要任务是负责处理时钟中断，调用系统实现的 OSTimeTick 函数，如果有等待时钟信号的高优先级任务，则需要在中断级别上调度其执行。其他相关的两个函数是 OSIntEnter () 和 OSIntExit ()，都需要在 ISR 中执行。

▪ ARMDisableInt () & ARMEnableInt ()

分别是退出临界区和进入临界区的宏指令实现。主要用于在进入临界区之前关闭中断，在退出临界区的时候恢复原来的中断状态。它的实现比较简单，可以采用方法 1 直接开关中断来实现，也可以采用方法 2 通过保存关闭/恢复中断屏蔽位来实现。

三、我的移植体会

移植 uC/OS-II 的绝大部分工作都集中在 os_cpu_a.S 文件的移植，这个文件的实现集中体现了所要移植到处理器的体系结构和 uC/OS-II 的移植原理；在这个文件里，最困难的工作又集中体现在 OSIntCtxSw 和 OSTickISR 这两个函数的实现上。这是因为这两个函数的实现是和移植者的移植思路以及相关硬件定时器、中断寄存器的设置有关。在实际的移植工作中，这两个地方也是比较容易出错的地方。

OSIntCtxSw 最重要的作用就是它完成了在中断 ISR 中直接进行任务切换，从而提高了实时响应的速度。它发生的时机是在 ISR 执行到 OSIntExit 时，如果发现有高优先级的任务因为等待的 time tick 到来获得了执行的条件，这样就可以马上被调度执行，而不用返回被中断的那个任务之后再进行任务切换，因为那样的话就不够实时了。

实现 OSIntCtxSw 的方法大致也有两种情况：一种是通过调整 sp 堆栈指针的方法，根据所用的编译器对于函数嵌套的处理，通过精确计算出所需要调整的 sp 位置来使得进入中断时所做的保存现场的工作可以被重用。这种方法的好处是直接在函数嵌套内部发生任务切

换,使得高优先级的任务能够最快的被调度执行。但是这个办法需要和具体的编译器以及编译参数的设置相关,需要较多技巧。

另一种是设置需要切换标志位的方法,在 OSIntCtxSw 里面不发生切换,而是设置一个需要切换的标志,等函数嵌套从进入 OSIntExit => OS_ENTER_CRITICAL() => OSIntCtxSw() => OS_EXIT_CRITICAL() => OSIntExit 退出后,再根据标志位来判断是否需要进行中断级的任务切换。这种方法的好处是不需要考虑编译器的因素,也不用做计算,但是从实时响应上不是最快,不过刚开始学习这种方法比较容易理解,实现起来也简单。SkyEye 目前的移植就是基于第二种方法的。

在中断态下进行任务切换,需要特别说明的一个问题是如何获得被中断任务的 lr_svc。因为进入中断态后,lr 变成了 lr_irq,原来任务的 lr_svc 无法在中断态下获得,这样要得到 lr_svc,就必须在中断 ISR 里面进行一次 cpu mode 强制转换,即对 CPSR 赋值为 0x000000d3,只有返回到 svc 态之后才能得到原来任务的 lr,这个对于任务切换很重要。还有一个需要留意的问题是在强制 CPSR 变成 svc 态之后,SPSR 也会相应地变成 SPSR_irq,这样就需要在强制转变之前保存 SPSR,也就是被中断任务中断前的 CPSR。

全部移植代码在 SkyEye 仿真器上调试通过,在 SkyEye 的主页上可以下载获得。欢迎大家访问我们的主页。【<http://hplab.cs.tsinghua.edu.cn/~skyeeye/>】另外在阿卡嵌入式兴趣小组的论坛上【<http://www.akaembed.org/Inetpub/forum/index.php>】,有关于 SkyEye 进展的最新讨论,也希望大家对我们的工作提出建议和批评,更希望有越来越多的人关注和参与进来。

四、总结

移植 uC/OS-II 到 SkyEye 上,既是对 uC/OS-II 的学习和实验,同时也是对 SkyEye 仿真器的验证和实践。uC/OS-II 作为一个优秀的实时操作系统已经被移植到各种体系结构的微处理器上,也是目前较为常用的公开源码的实时内核。从这里入手学习嵌入式系统开发的基本概念,以及在 SkyEye 里构造一个可以运行的 RTOS,能够使我们更深入地了解嵌入式开发的流程,在没有硬件的条件下也能对 ARM 的体系结构有个初步的认识。

在移植 uC/OS-II 到 SkyEye 之后,我得到了一块 Samsung S3C4510 的 ARM 评估板,在调通了板子上一些相关硬件(例如串口输出和定时器)的驱动后,仅仅花了不到一天时间就将 SkyEye 下的 uC/OS-II 移植到了真实的开发板上,这也说明在 SkyEye 上所做的移植工作是非常有意义和帮助的,完全可以作为嵌入式开发的入门捷径。

如果大家移植过程中遇到什么问题,欢迎发 email 和我讨论。

今天是 3 月 5 日,永远怀念周总理!

Liming

lmcs00@mails.tsinghua.edu.cn

2003-3-5