Starterware 工程与自建工程默认 ARM Mode 的区别由来

3-22-2016 Tony Tang

针对有不少用户对 TI 提供的 starterware 工程，以及自建的 CCS 工程里的 ARM 模式存在疑问，觉得有必要做个简单总结，以理清思路，节省大家的时间为目的。

#1. ARMV4 以上的内核有 7 个不同的模式，其状态可以通过 CPSR 寄存器确定。（至于为什么要分这么多种模式，不是本小结的讨论内容）

（1） User Mode：用户模式。操作系统的 Task 一般以这种模式执行。User Mode 是 ARM 唯一的非特权模式，这表示如果 CPU 处于这种模式下，很多指令将不能够执行，因此操作系统的资源得以保护。

（2） System Mode：这是 V4 及其以上版本所引入的特权模式。、

（3） IRQ Mode：中断模式。中断（不包括软中断）处理函数在这种模式下执行。

（4） FIQ Mode：快速中断模式。除了多了几个寄存器外，其他同 IRQ 一样。

（5） Supervisor Mode：管理模式。软中断（SWI)处理函数在这种模式下执行。

（6） Abort Mode：所有同内存保护相关的异常均在这种模式下执行。

（7） Undefined Mode：处理无效指令的异常处理函数在这种模式下执行

#2. ARM 在不同的模式下有不同的权限，以 OMAP-L138 为例，修改系统寄存器需要在 supervisor 或 system mode 这种 privilege 模式下才行，比如配置 PLL，PINMUX 等。

#3. 大家发现用 starterware 提供的工程，一切都是那么顺利，似乎这些模式与自己无关，所以从未关心过。但是一旦自己用 CCS 新建一个工程，就什么也配置不了了，一看 CPSR 原来是在用户模式，但怎么进入 privilege 模式呢~~~

#4. 百度~~~ARM 在 user mode 下要通过 SWI 才能进入 supervisor 模式，spnu151L 文档里也提供了 call_swi（）函数接口，还可以带个参数，可是为什么一调用就跑飞了呢？还有这个参数是啥意思？

#5. 不管查什么资料，基本上都说 ARM 上电启动后是 supervisor 模式，而且上电后连上仿真器，也可以看到 CPSR 显示的是 supervisor 模式。可是运行到 main 就变成 user mode 了，那么在 main 之前干了啥~~~

#6. C 工程从来都不是从 main 开始运行的，注意到 map 文件没，里面的 entry_point 是 _c_int00, 如果不知道它是干啥的，看一下 spnu151l 的第 6 章，简而言之，就是 C 环境初始化，(写汇编可从来没这个说话). 还是看看_c_int00 这个函数做了啥吧（这个函数在 RTS 库里，源文件在 CCS 的编译器安装目录下 boot.asm）。

```
;*-------------------------------------------------------
    ;* SET TO USER MODE
    ;*-----------------------------------------------------
```

```
MRS     r0, cpsr
BIC     r0, r0, #0x1F  ; CLEAR MODES
ORR     r0, r0, #0x10  ; SET USER MODE
MSR     cpsr_cf, r0
```

简单来看，就在这里切换到了 user mode，然后初始化了 user mode 的堆栈，再是对初始化段做初始化（__TI_auto_init），再就跳到我们的 main 了

```
;*----------------------------------------------------
;* INITIALIZE THE USER MODE STACK
;*----------------------------------------------------
.if __TI_AVOID_EMBEDDED_CONSTANTS
MOVW    sp, __stack
MOVT    sp, __stack
MOVW    r0, __STACK_SIZE
MOVT    r0, __STACK_SIZE
.else
LDR     sp, c_stack
LDR     r0, c_STACK_SIZE
.endif
ADD     sp, sp, r0

;*----------------------------------------------------
;* ALIGN THE STACK TO 64-BITS IF EABI.
;*----------------------------------------------------
.if __TI_EABI_ASSEMBLER
BIC     sp, sp, #0x07  ; Clear upper 3 bits for 64-bit alignment.
.endif

;*----------------------------------------------------
;* SAVE CURRENT STACK POINTER FOR SDP ANALYSIS
;*----------------------------------------------------
```

```
.if __TI_AVOID_EMBEDDED_CONSTANTS
MOVW    r0, MAIN_FUNC_SP
MOVT    r0, MAIN_FUNC_SP
.else
LDR     r0, c_mf_sp
.endif
STR     sp, [r0]

;*----------------------------------------------------
;* Perform all the required initilizations:
;*   - Process BINIT Table
;*   - Perform C auto initialization
;*   - Call global constructors
;*----------------------------------------------------
BL      __TI_auto_init

;*----------------------------------------------------
;* CALL APPLICATION
;*----------------------------------------------------
BL      ARGS_MAIN_RTN
```

#6. 怎么办，最简单的办法就是把这个文件加到工程里去，然后把切换模式这段代码删掉就行了（因为在工程里源文件与库文件存在同样的 lable，会优先调用源文件的），但是后面想切换到 user，又想从 user 切换到 supervisor 怎么办呢，再说吧~~~。

#7. 现在再来看看 starterware 的工程，注意一下 cmd 文件，里面都加了一句-e Entry，也就是说 starterware 的工程不是从_c_int00 开始执行的，所以编译输出信息里也相应多了一句警告，这没关系，只要在 main 之前把该做的做了就行，_c_int00 只是提供了一个简便的现成的函数，因为通常都是做同样的事件，没必要大家每次自己写一遍嘛。

#8. 对比一下 Entry 与_c_int00 的区别吧。Entry 在 starterware 的 system_config\armv5\下对应的编译器类型目录下的 init.asm 文件，

```
Entry:
```

```
;
; Set up the Stack for Undefined mode
;
        LDR   r0, _stackptr                  ; Read and align the stack
pointer
        SUB   r0, r0, #8
        BIC   r0, r0, #7
        MSR   cpsr_c, #MODE_UND|I_F_BIT      ; switch to undef  mode
        MOV   sp,r0                          ; write the stack pointer
        SUB   r0, r0, #UND_STACK_SIZE        ; give stack space
;
; Set up the Stack for abort mode
;
        MSR   cpsr_c, #MODE_ABT|I_F_BIT      ; Change to abort mode
        MOV   sp, r0                         ; write the stack pointer
        SUB   r0,r0, #ABT_STACK_SIZE         ; give stack space
;
; Set up the Stack for FIQ mode
;
        MSR   cpsr_c, #MODE_FIQ|I_F_BIT      ; change to FIQ mode
        MOV   sp,r0                          ; write the stack pointer
        SUB   r0,r0, #FIQ_STACK_SIZE         ; give stack space
;
; Set up the Stack for IRQ mode
;
        MSR   cpsr_c, #MODE_IRQ|I_F_BIT      ; change to IRQ mode
        MOV   sp,r0                          ; write the stack pointer
        SUB   r0,r0, #IRQ_STACK_SIZE         ; give stack space
;
; Set up the Stack for SVC mode
;
        MSR   cpsr_c, #MODE_SVC|I_F_BIT      ; change to SVC mode
        MOV   sp,r0                          ; write the stack pointer
        SUB   r0,r0, #SVC_STACK_SIZE         ; give stack space
;
; Set up the Stack for USer/System mode
;
        MSR   cpsr_c, #MODE_SYS|I_F_BIT      ; change to system mode
        MOV   sp,r0                          ; write the stack pointer
```

原来在这里初始化了各模式下的堆栈，顺便在最后设定在了 system mode。再往下看：

```
;
; Clear the BSS section here
;
Clear_Bss_Section:

        LDR   r0, _bss_start                ; Start address of BSS
        LDR   r1, _bss_end                  ; End address of BSS
        SUB   r1,r1,#4
        MOV   r2, #0
Loop:
        STR   r2, [r0], #4                   ; Clear one word in BSS
```

```
        CMP    r0, r1
        BLE    Loop                            ; Clear till BSS end

        BL     __TI_auto_init                  ; Call TI auto init

;
; Enter the start_boot function. The execution still happens in system mode
;
        LDR    r10, _start_boot                 ; Get the address of
start_boot
        MOV    lr,pc                           ; Dummy return
        BX     r10                             ; Branch to start_boot
        SUB    pc, pc, #0x08                   ; looping

;        MSR    cpsr_c, #MODE_SVC|I_F_BIT       ; change to SVC mode
;        BX     lr
;
  ; End of the file
```

上面的_TI_auto_init 还是一样调用的 RTS 库的，然后调用了 start_boot（startup.c），

```
unsigned int start_boot(void)
{

    /* Enable write-protection for registers of SYSCFG module. */
    SysCfgRegistersLock();

    /* Disable write-protection for registers of SYSCFG module. */
    SysCfgRegistersUnlock();

    PSCModuleControl(SOC_PSC_1_REGS, HW_PSC_UART2, 0, PSC_MDCTL_NEXT_ENABLE);

    PSCModuleControl(SOC_PSC_0_REGS, HW_PSC_AINTC, 0, PSC_MDCTL_NEXT_ENABLE);
    /* Initialize the vector table with opcodes */
    CopyVectorTable();

   /* Calling the main */
    main();

    while(1);
  }
```

上面在调用 main 之前这一步很关键了。将异常向量表搬到了 0xFFFF0000 开始处（在 L138 上规定 ARM 的异常向量表只能放这，可以看一下 L138 TRM 手册的 2.4 节）。

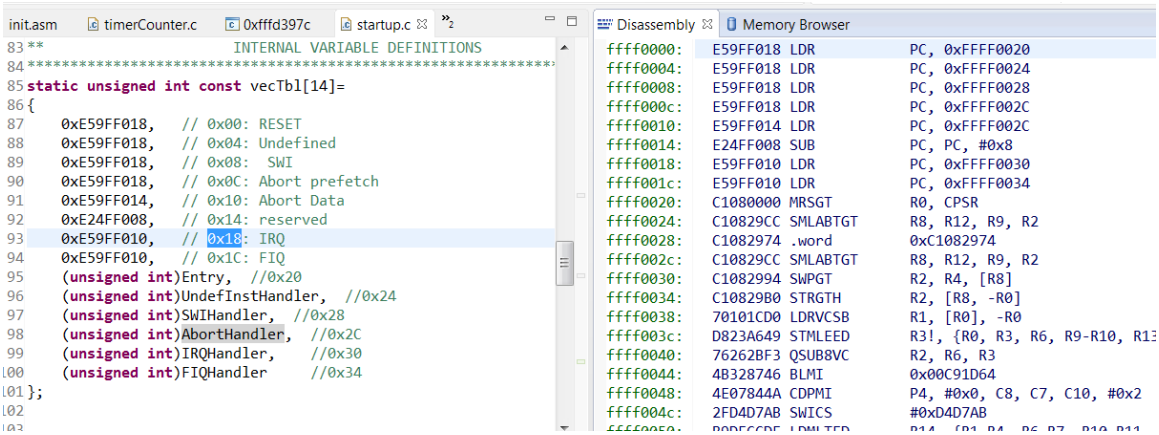
```
static unsigned int const vecTbl[14]=
{
    0xE59FF018,   // 0x00: RESET
    0xE59FF018,   // 0x04: Undefined
    0xE59FF018,   // 0x08:  SWI
    0xE59FF018,   // 0x0C: Abort prefetch
    0xE59FF014,   // 0x10: Abort Data
    0xE24FF008,   // 0x14: reserved
    0xE59FF010,   // 0x18: IRQ
```

```
    0xE59FF010,   // 0x1C: FIQ
    (unsigned int)Entry,              //0x20
    (unsigned int)UndefInstHandler,   //0x24
    (unsigned int)SWIHandler,         //0x28
    (unsigned int)AbortHandler,       //0x2C
    (unsigned int)IRQHandler,         //0x30
    (unsigned int)FIQHandler          //0x34
  };

static void CopyVectorTable(void)
{
    unsigned int *dest = (unsigned int *)0xFFFF0000;
    unsigned int *src =  (unsigned int *)vecTbl;
    unsigned int count;

    for(count = 0; count < sizeof(vecTbl)/sizeof(vecTbl[0]); count++)
    {
        dest[count] = src[count];
    }
  }
```

#9. 至于上面那个数组的 trick，在内存上显示其代表的汇编就明白了，它巧妙的在向量表上加上了跳转到各异常向量的处理 handler（这要通过对指定格式的了解才能整出这么个数值来）：



后面带着 system mode 进入 main 后，就由用户自由发挥了。但是这里跟前面那个改 boot.asm 进入 main 后的区别是什么呢？前面我也提到如果想进切换模式怎么办。

现在的基于 starterware 的工程可以调用 cpu.c 文件里的函数进行模式切换，但是前面那种修改的工程是不行了，一调用就跑飞。原因在于，在调用 swi 时，程序会跳转到 SWI 位置即 0xFFFF0008, 然后跳转到 SWIHandle（见 exceptionhandler.asm），

```
SWIHandler:
        STMFD    r13!, {r0-r1, r14}       ; Save context in SVC stack
        LDR      r0, [r14, #-4]           ; R0 points to SWI instruction
        BIC      r0, r0, #MASK_SWI_NUM    ; Get the SWI number
        CMP      r0, #458752
        MRSEQ    r1, spsr                 ; Copy SPSR
        ORREQ    r1, r1, #0x1F            ; Change the mode to System
```

```
MSREQ    spsr_cf, r1              ; Restore SPSR
LDMFD    r13!, {r0-r1, pc}^       ; Restore registers from IRQ stack
```

这里就进入了 supervisor mode，而且对传进来的参数做了对比确认，理论上可以做一长串的参数对比，做不同分支的处理。

为什么自己建的 CCS 工程不行呢，因为还没有做异常向量表的初始化，调用 SWI 时，系统自动跳转到 0xFFFF0008，谁知道这时候这地方是一条什么指令呢，所以在调用之前必需要先初始化异常向量表等等，有兴趣自己实现吧。

附上我基于 starterware 简化过来的工程。