
嵌入式系统实验报告



实验名称: CPU 异常处理与上下文切换

姓 名: 陈姝仪

学 号: 2018211507

学 院(系): 计算机学院

专 业: 网络工程

指导教师: 刘健培

2020 年 12 月 31 日

1 实验目的

- 了解开发环境的使用。
- 学会通过查阅文档和数据手册获取信息。
- 掌握 cortex-M4 体系结构中寄存器和异常的使用方式。
- 掌握基本的软件编写与调试方式。
- 理解处理器异常上下文切换的实现方式。
- 理解多任务的实现方式。

2 实验环境

- FS-STM32F407 开发平台
- ST-Link 仿真器
- RealView MDK5.23 集成开发软件
- 串口调试工具
- PC 机 Window7/8/10 (64bit)

3 实验要求

- 基本要求
 - ◆ 在主程序中用 svc 指令触发 SVC 异常
 - ◆ 编写 SVCcall 异常处理程序，打印异常发生前的处理器现场状态（即寄存器 R0-R15、XPSR）以及异常发生后发生变化的寄存器（R13/SP、R14/LR/EXC_RETURN、R15/PC、xPSR、CONTROL），据此分析异常发生前后处理器分别处于哪种模式（handler or thread）、使用哪种栈（MSP or PSP）、特权等级（特权与非特权）
- 扩展要求
 - ◆ 使用 svc 异常模拟系统调用，实现函数间上下文切换功能
 - ◆ 如：func1->context_switch->func2->context_switch->func1

4 实验原理

- 使用 svc 系统调用指令出发 SVCcall 异常
在 eclipse 的 gcc 编译环境下，可用 C 嵌入汇编如下：
#define SVC_CALL() asm volatile ("svc 0")

指令格式:

Assembler syntax

`SVC<C><Q> #<imm>`

where:

`<C><Q>` See [Standard assembler syntax fields on page A7-177](#).

`<imm>` Specifies an 8-bit immediate constant.

The pre-UAL syntax `SWI<C>` is equivalent to `SVC<C>`.

● Cortex-M4 处理异常的流程

异常发生后，需要软件和硬件协作处理。

进入异常时，硬件会切换模式，并保存必要的寄存器。然后异常处理程序（软件）再根据需保存需要的额外寄存器。

保护好现场后，就可以进行额外的处理。此时可以进一步准备好调用 `c` 函数需要的堆栈，然后调用 `c` 函数进行进一步处理。

退出异常时，流程基本与进入异常时相反，首先软件需要恢复部分寄存器，然后通过机器指令让硬件恢复之前硬件保存的寄存器，并跳转到异常发生时的地址继续执行（如果需要）。

● “现场”的含义及需要保存的内容

一般而言，现场指的是处理器的核内寄存器。

因为当发生异常时，处理器需要运行另外一段程序，这段程序也需要使用处理器，所以必须先把异常前处理器中已有的寄存器中的数据保护起来，相当于创建一个“还原点”，然后在处理完后，在将之前保护的数据恢复到处理器中，从而从“还原点”继续往下执行原程序。

为获取异常发生前的处理器现场状态，需要知道 `arm` 在进入异常时，保存了哪些寄存器（意味着这些寄存器是我们在异常里写代码时可用的），保存在了哪个位置（据此才能获取 `arm` 保存的值）。然后在我们的代码破坏 `arm` 未保存的值之前，先把相关的寄存器的内容保存下来，然后准备好 `c` 的调用环境，就可以调用 `c` 函数进行处理了。

● 进入异常时，硬件保存的寄存器

The `PushStack()` and `ExceptionTaken()` pseudo-functions are defined as follows:

```
// PushStack()
// =====

PushStack()
  if CONTROL<1> == '1' AND CurrentMode == Mode_Thread then
    frameptralign = SP_process<2> AND CCR.STKALIGN;
    SP_process = (SP_process - 0x20) AND NOT(ZeroExtend(CCR.STKALIGN:'00',32));
    frameptr = SP_process;
  else
    frameptralign = SP_main<2> AND CCR.STKALIGN;
```

```

    SP_main = (SP_main - 0x20) AND NOT(ZeroExtend(CCR.STKALIGN:'00',32));
    frameptr = SP_main;
    /* only the stack locations, not the store order, are architected */
    MemA[frameptr,4] = R[0];
    MemA[frameptr+0x4,4] = R[1];
    MemA[frameptr+0x8,4] = R[2];
    MemA[frameptr+0xC,4] = R[3];
    MemA[frameptr+0x10,4] = R[12];
    MemA[frameptr+0x14,4] = LR;
    MemA[frameptr+0x18,4] = ReturnAddress();
    MemA[frameptr+0x1C,4] = (xPSR<31:10>:frameptralign:xPSR<8:0>);
    // see ReturnAddress() in-line note for information on xPSR.IT bits
    if CurrentMode==Mode_Handler then
        LR = 0xFFFFFFF1;
    else
        if CONTROL<1> == '0' then
            LR = 0xFFFFFFF9;
        else
            LR = 0xFFFFFFFD;
    return;

// ExceptionTaken()
// =====

ExceptionTaken(bits(9) ExceptionNumber)

    bit tbit;
    bits(32) tmp;

```

● 准备好调用 C 的堆栈环境

Table 6.1: Table 2, Core registers and AAPCS usage

Regis-ter	Syn-onym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

一般，编译器处理器 CPU 中的寄存器时有 3 类主要的用法，1 类编译器不会使用的，主要是一些特殊寄存器（如调试寄存器、特殊功能寄存器等），第 2 类是需要调用者保存的——即 callee 认为 caller 会保存从而 callee 可以不保存而使用，如 R0-R3 R12 这些 scratch 寄存器，第三类是被调用者 caller 会保存的，如 R4-R11 这些变量寄存器和 LR 等。

所以，我们作为 caller，需要保存 **R0-R3 R12**。

● 退出异常时发生的过程

```
// PopStack()
// =====

PopStack(bits(32) frameptr) /* only stack locations, not the load order, are architected */
{
    R[0] = MemA[frameptr,4];
    R[1] = MemA[frameptr+0x4,4];
    R[2] = MemA[frameptr+0x8,4];
    R[3] = MemA[frameptr+0xC,4];
    R[12] = MemA[frameptr+0x10,4];
    LR = MemA[frameptr+0x14,4];
    PC = MemA[frameptr+0x18,4]; // UNPREDICTABLE if the new PC not halfword aligned
    psr = MemA[frameptr+0x1C,4];
    case EXC_RETURN<3:0> of
    when '0001' // returning to Handler
        SP_main = (SP_main + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
    when '1001' // returning to Thread using Main stack
        SP_main = (SP_main + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
    when '1101' // returning to Thread using Process stack
        SP_process = (SP_process + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
    APSR<31:27> = psr<31:27>; // valid APSR bits loaded from memory
    IPSR<8:0> = psr<8:0>; // valid IPSR bits loaded from memory
    EPSR<26:24,15:10> = psr<26:24,15:10>; // valid EPSR bits loaded from memory
    return;
}
```

● 得知异常前后处理器的模式、栈、特权等级

CONTROL 寄存器保存了线程模式的特权等级：

表 4.3 CONTROL 寄存器中的位域

位	功 能
nPRIV(第 0 位)	定义线程模式中的特权等级： 当该位为 0 时(默认)，处理器会处于线程模式中的特权等级；而当其为 1 时，则处于线程模式中的非特权等级

EXC_RETURN 寄存器保存了异常发生前的模式和栈类型：

表 8.1 EXC_RETURN 的位域

位	描述	数 值
31:28	EXC_RETURN 指示	0xF
27:5	保留(全为 1)	0xEFFFFFF(23 位都是 1)
4	栈帧类型	1(8 字)或 0(26 字)。当浮点单元不可用时总是为 1，在进入异常处理时，其会被置为 CONTROL 寄存器的 FPCA 位
3	返回模式	1(返回线程)或 0(返回处理)
2	返回栈	1(返回线程栈)或 0(返回主栈)
1	保留	0
0	保留	1

表 8.2 EXC_RETURN 的合法值

	浮点单元在中断前使用(FPCA=1)	浮点单元未在中断前使用(FPCA=0)
返回处理模式(总是使用主栈)	0xFFFFFFE1	0xFFFFFFF1
返回线程模式并在返回后使用主栈	0xFFFFFFE9	0xFFFFFFF9
返回处理模式并在返回后使用进程栈	0xFFFFFED	0xFFFFFDD

异常发生后，处理器状态是固定的：特权等级、主栈 MSP、处理/handler 模式。

● 通过 SVC 系统调用实现函数上下文切换

资料《ARM Cortex-M3Cortex-M4 权威指南.pdf》中 10.5 节使用 pendSV 实现了任务的上下文切换，使用 svc 指令是基本类似的：

从哪儿可以得知异常前后处理器的模式、栈、特权等级？

CONTROL 寄存器保存了线程模式的特权等级：

表 4.3 CONTROL 寄存器中的位域

位	功 能
nPRIV(第 0 位)	定义线程模式中的特权等级： 当该位为 0 时(默认)，处理器会处于线程模式中的特权等级；而当其为 1 时，则处于线程模式中的非特权等级

EXC_RETURN 寄存器保存了异常发生前的模式和栈类型：

表 8.1 EXC_RETURN 的位域

位	描述	数 值
31:28	EXC_RETURN 指示	0xF
27:5	保留(全为 1)	0xEFFFFFF(23 位都是 1)
4	栈帧类型	1(8 字)或 0(26 字)。当浮点单元不可用时总是为 1，在进入异常处理时，其会被置为 CONTROL 寄存器的 FPCA 位
3	返回模式	1(返回线程)或 0(返回处理)
2	返回栈	1(返回线程栈)或 0(返回主栈)
1	保留	0
0	保留	1

表 8.2 EXC_RETURN 的合法值


	浮点单元在中断前使用(FPCA=1)	浮点单元未在中断前使用(FPCA=0)
返回处理模式(总是使用主栈)	0xFFFFFE1	0xFFFFFFF1
返回线程模式并在返回后使用主栈	0xFFFFFE9	0xFFFFFFF9
返回处理模式并在返回后使用进程栈	0xFFFFFED	0xFFFFFFFD

异常发生后，处理器状态是固定的：特权等级、主栈 MSP、处理/handler 模式。

5 实验步骤

5.1 基本部分

● 分析函数调用的现场保护和恢复

1. 下载并打开老师给的代码 emlab2020-lab1.rar  emlab2020-lab1.rar

2. 连接好实验板的相关线路，打开电源，打开串口工具并打开响应串口。

3. 在 `lab_main.c` 中找到实验入口 `lab1_a_main()`，设置断点，`build` 程序，然后用 `debug` 模式运行程序，从这个函数开始一步步跟着程序走，观察程序的执行过程。

4. 对照实验指导书和 `SVC_Handler` 函数分析函数调用和退出时是如何保存和恢复现场的，保存了哪些寄存器，调用前后哪些寄存器发生了变化。

对照实验指导书，知道了 `CONTROL` 寄存器第 0 位保存了线程模式的特权等级，`EXC_RETURN` 寄存器第 3、第 2 位分别保存了异常发生前的模式和栈类型。从 `frame` 中取，若 `CONTROL` 寄存器第 0 位为 0，则线程模式是特权等级，为 1 则为非特权等级；若 `EXC_RETURN` 寄存器第 3 位为 1 则处于 `thread` 模式，为 0 则处于 `handler` 模式；若 `EXC_RETURN` 寄存器第 2 位为 1 则为 `PSP`，为 0 则为 `MSP`。

5.2 附加部分

1. 打开项目中的 `lab1_b.c` 和 `task_co.c`，分析代码，分析任务之间通过主动调用 `svc` 现场切换系统调用让渡 `cpu` 的过程。

2. 通过分析 `take_change_state_to` 函数，发现任务一共有 4 个状态：`TASK_DEAD`、`TASK_RUNNING`、`TASK_READY`、`TASK_BLOCKED`。根据 `blocked` 状态中，向 `ready` 和 `dead` 状态转移都需要将任务从 `wait_queue` 中取出来，`running` 状态转到 `ready` 状态要把任务加到 `ready_queue` 中，转到 `blocked` 状态要把任务加到 `wait_queue` 中，因此对应的判断，`ready` 状态转到 `blocked` 状态也要把任务加到 `wait_queue` 中，`ready` 状态转到其他状态要把任务从 `ready_queue` 中移除。

3. 根据分析填充 `task_change_state_to` 函数中 `TASK_READY` 情况的代码即可。

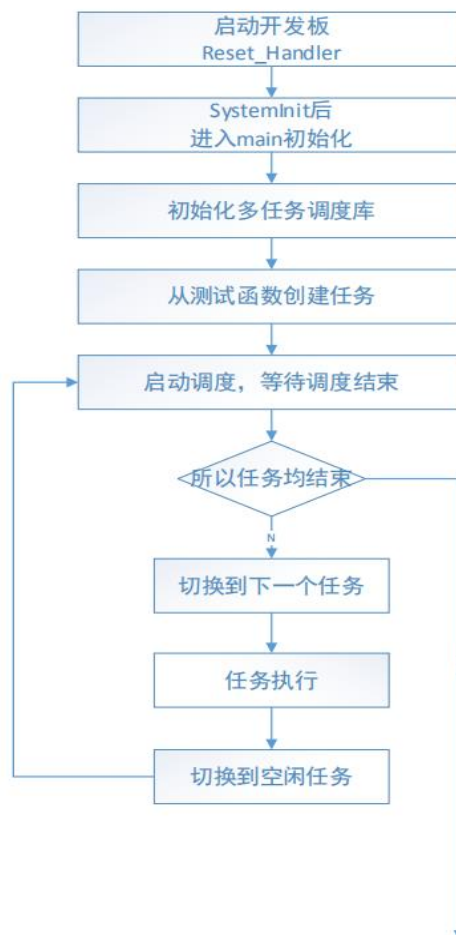
6 实验方案与实现

6.1 软件结构

- SVC 异常处理的程序流程如下图所示：



- SVC 上下文切换的程序流程如下图所示：



6.2 源代码

Lab1_a:

```
static void cpu_state(svc_stack_frame_t *frame) {
    //error NOT implemented!
    if(frame->control == 0){
        trace_printf("PRIV = P\r\n");
    }else if(frame->control == 1){
        trace_printf("PRIV = NP\r\n");
    }
    if(frame->exc_return & 0x4){
        trace_printf("Return 线程栈\r\n");
    }else{
        trace_printf("Return 主栈\r\n");
    }
    if(frame->exc_return & 0x8){
        trace_printf("Return Mode Thread\r\n");
    }else{
        trace_printf("Return Mode Handler\r\n");
    }
}
```

Lab1_b:

```
case TASK_READY:
    //error NOT implemented!
    if(newstate == TASK_DEAD) { //READY -> TERMINATED
        TAILQ_REMOVE(&ready_queue, t, stateq_node);
    }

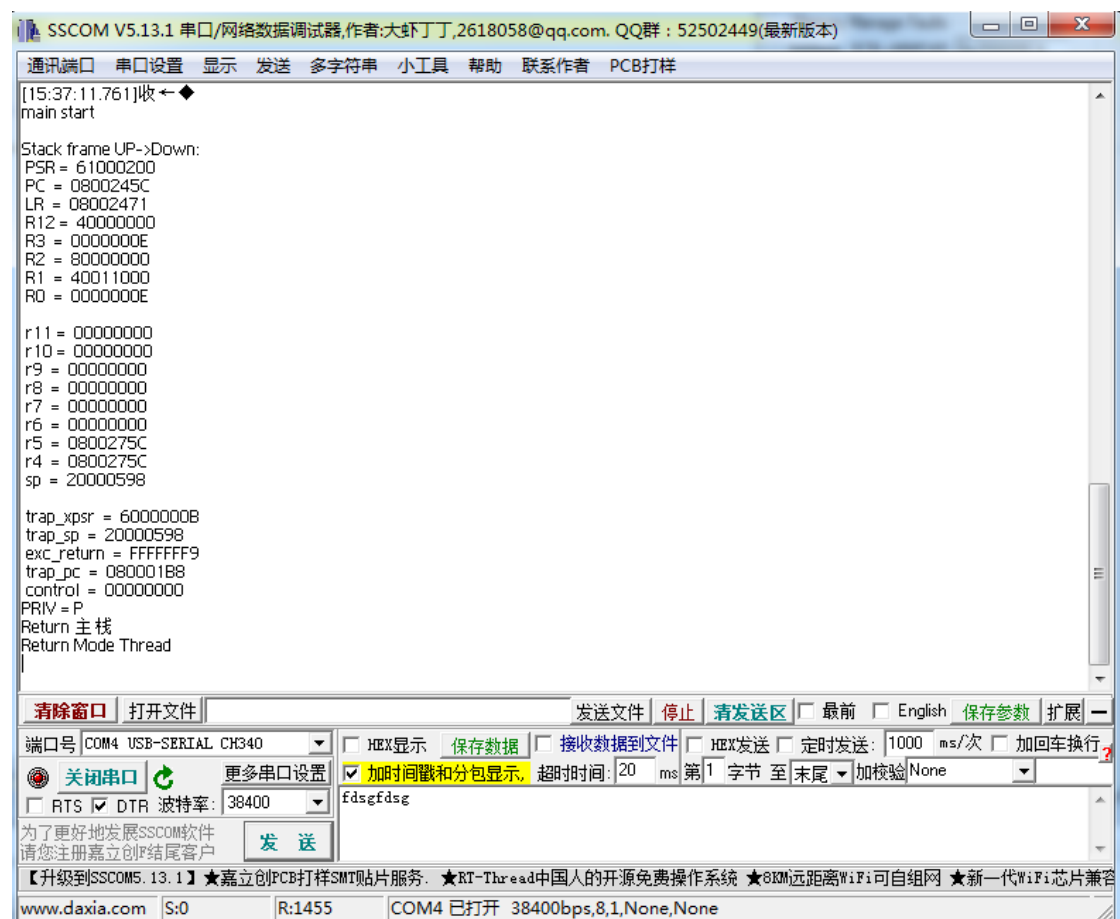
    else if(newstate == TASK_RUNNING) { //READY -> RUNNING
        TAILQ_REMOVE(&ready_queue, t, stateq_node);
    }

    else if(newstate == TASK_READY) { //READY -> READY
    }

    else if(newstate == TASK_BLOCKED) { //READY -> SWAP WAITING
        TAILQ_REMOVE(&ready_queue, t, stateq_node);
        TAILQ_INSERT_TAIL(&wait_queue, t, stateq_node);
    }
    break;
```

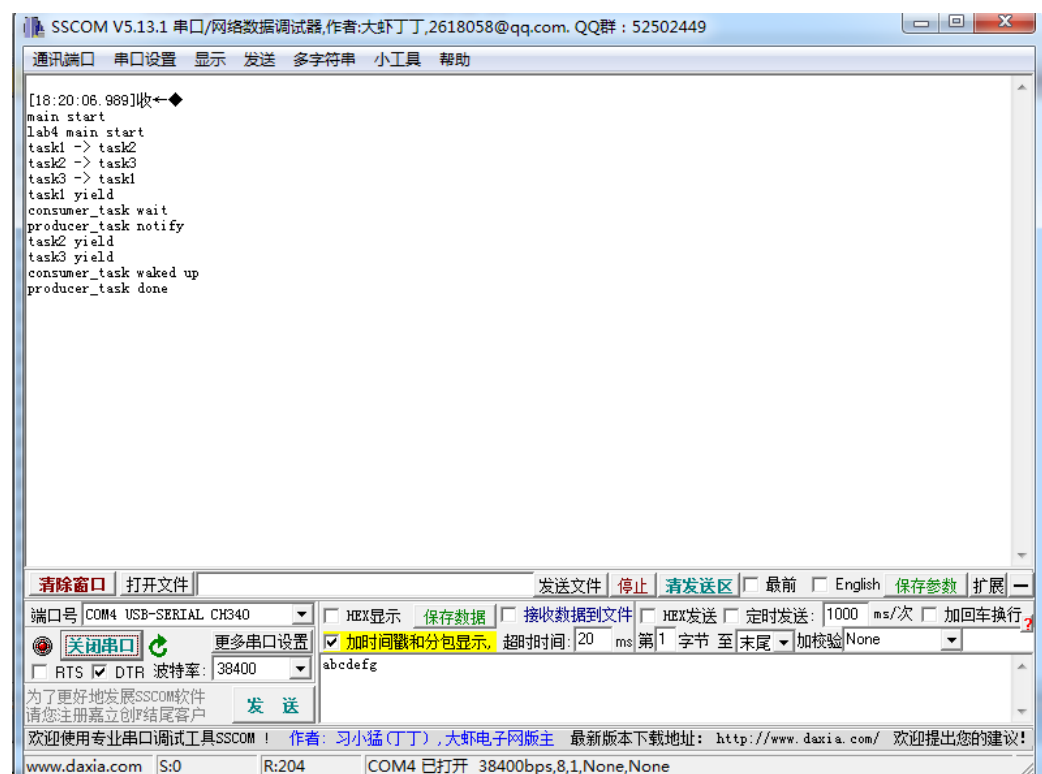
7 实验结果与分析

Lab1_a:



从图中可知，特权等级为特权，栈为MSP，模式为线程模式

Lab1_b:



8 实验总结

第一次做嵌入式实验，有点无从下手，后来认真看了实验指导书后有了些眉目。实验的关键在于根据实验要求，明白 `cpu_state` 函数的作用，以及学会查找指导手册获取自己需要的信息，并据此获得对应的信息用于判断处理器分别处于何种模式（`handler or thread`）、使用何种栈（`MSP or PSP`）和何种特权等级（特权与非特权）。

在 b 部分的实验中，发现嵌入式系统的状态切换和我们在《操作系统》课上学的不太一样。`Ready` 状态之后，每个状态都有可能出现，而不是像操作系统的状态转换图一样 `ready` 后只有 `running`。