# 嵌入式系统实验报告


| | |
|---|---|
| 实验名称: | 嵌入式系统综合实验 |
| 姓　　名: | 蒋雪枫 |
| 学　　号: | 2017213508 |
| 学 院(系): | 计算机学院 |
| 专　　业: | 网络工程 |
| 指导教师: | 刘健培老师、戴志涛老师 |

2019 年　12　月　13　日

# 1 实验目的

1. 学会通过查阅文档和数据手册获取信息
2. 掌握 Arm Cortex M4 系列体系结构中的寄存器和异常的使用方式
3. 学会看大型嵌入式程序的代码，研究函数之间的关系，理清运行某一功能的过程
4. 了解 FSM4 华清远见实验板的软硬件环境
5. 学会 STM32 GPIO 操作方式、STM32 USART 操作方式、STM32 中断处理方式
6. 熟悉 MDK、Eclipse(Embeded)、STM32CubeMX 开发环境的使用
7. 掌握 C 语言直接控制寄存器，写入值和查看值的方式
8. 明白本地计算机如何和开发板进行信息交互的原理
9. 加深对 C 语言不常用的一些特性的理解，如位运算
10. 理解处理器异常上下文切换的实现方式、协程的实现方式
11. 学会通过 sscom 串口软件对嵌入式代码进行调试(结合 trace_printf 功能)
12. 加深对轮询式多软件编写和操作方式，学习基本的定时多任务处理方式
13. 结合实践深刻领会嵌入式系统原理，巩固课内知识

# 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
- PC机Window7/8/10 (32/64bit)
- 串口调试工具

此外，还有自己的个人电脑，虽然平时不能连接开发板，但是还是可以在课下思考和完成编码的工作。

# 3 实验要求

本次实验我尽量去把全部实验（包括扩展实验）去研究和思考了一遍，全部实验要求如下，重点的偏实践的实验要求会在里面重点表示出来。

另外值得一提的是，其实很多复杂的功能，刘老师已经给我们写好了，省去了我们很多上层的一些思考和编码工作，而我们的工作更多聚焦于嵌入式系统本身的相关工作。

实验一 a&b：

- 基本要求
  - 在主程序中用 svc 指令触发 SVCall 异常
  - 编写 SVCall 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

实验二：

- 基本要求
  - 编写程序控制 led 灯的亮灭（或者控制板上蜂鸣器的出声），输出以字母、数字、空格组成的字符串的摩斯码（以“Hello Cortex-M4”为测试用例）。
  - 特殊要求：不能使用 CMSIS 库函数操作 led 灯（蜂鸣器），需用代码直接操作 GPIO 的寄存器。
- 扩展要求
  - 使用按键控制系统状态，LED 灯显示系统状态：
    - 按键 K3 按下：待机，系统进入低功耗模拟



    - 按键 K4 长按：系统复位
    - 按键 K5 双击：led 灯闪烁
    - 按键 K6 长按：随着按动时长，4 个 led 灯依次点亮

实验三：

- 基本要求
  - 使用 USART3 控制 4 个 led 灯的亮灭闪烁
  - 命令格式：led n on/off/flash
  - 参数：38400-8-1（波特率-数据位-停止位，其余无）
- 扩展要求
  - 实现基于 USART1 的简单 shell。支持以下功能：
    - help/? – 显示命令帮助
    - Tab – 补全或者显示可选命令
    - up/down – 切换命令历史
    - Backspace – 删除字符
    - d addr n – 在串口打印 addr 地址处的 n 个字节
    - 等

实验四：

- 基本要求
  - 使用 TIM1 中断与后台循环实现定时任务的调度
  - 自定义定时任务，譬如 LED 定期闪烁(呼吸灯)、定期读取板上传感器的值均可。
  - 定时精度 1ms
- 扩展要求
  - 使用 systick 中断（or TIM1 中断）与后台循环实现定时、异步等多种类型任务的调度
  - 支持 2 种类型定时任务：运行一次、周期运行
  - 定时精度 1ms
  - 支持多任务并发运行
    - 串口输入输出
    - Led 闪烁
    - 扫描按键

# 4 实验原理

实验原理和实验步骤一起说明，因为怎么做实验的过程是来源于实验的原理，而后我们才能开始编码，即实现方法必须依赖于实验原理。如：硬件接口的控制原理、所使用的算法的原理等。这些都需要我们先对“我们要干什么”理解足够清晰，这个设备支持我们干什么，需要用到什么函数接口，传入什么值，机器是怎么收到我们的指令的等等问题都需要先提前被我们大概明确，我们才能继续实现任务。同时，在完成报告的同时，我也去往了戴志涛老师上课所讲的 PPT 当中，结合理论知识进行阐述。**另外，我觉得，本次实验虽然显得很硬核，但让学生完成的都是一些比较基础并且没有过多涉及到该“嵌入式操作系统”本身的知识，而只是浅浅地让我们使用其中一个小机制，我可以感受到本次实验是刘老师呕心沥血做出来的，看懂这些代码对我来说也属实不易，但每看懂一部分，都能感受到其中设计的精巧之处，为之感慨。所以比起讲实验本身的实现，我觉得我们更应该关注背后的机理，我也会尽力在实验过程中解说。**

在实验报告中，学生会重点说明第一个实验，因为它更偏向于 ARM 的工作模式。后面也会尽可能详细地展示出来。

# 5 实验步骤

## 1. 实验一 a&b：

在实验 1 的基础部分，我们的任务是在主程序使用 svc 指令触发 SVCall 异常，并且分析其背后的原理。


ARM 处理器有七个基本的工作模式，分别为用户模式(大部分任务正常执行工作所处于的模式)、系统模式、普通中断模式 IRQ，快速中断模式 FIQ，管理员模式 SVC（复位或者执

行软中断进入的模式），中止模式 ABT（访存异常进入的模式，一般是虚拟存储和存储器保护）和未定义模式 UND（执行未定义的指令进入的模式）。

**Cortex-M4 处理异常的流程是怎样的？即进入异常和退出异常时处理器做了什么？**

异常发生后，需要软件和硬件协作处理。 中断一般由软件实现

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）再根据需要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”到底指什么？需要保存哪些内容？**

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。

查看源代码，我们得知我们是在 exp1_main 里面触发该中断，下面这段代码有必要说明一下：

```
//svc_trap_entry
#if   defined ( __CC_ARM )
__asm void _SVC_Handler() {
        PRESERVE8
    tst LR, #4          //which stack we were using?
    ite eq
    mrseq r0, msp
    mrsne r0, psp

            //hareware saved xpsr pc lr r12 r3-r0
    mov r3, r0    //software save r11-r4 & sp
    stmdb r0!, {r3, r4-r11}

    mrs r2, control    //software save trap changed registers
    mov r3, pc
    mov r4, lr
    mov r5, sp
    mrs r6, psr
    stmdb r0!, {r2 - r6}

    tst lr, #4    //are we on the same stack? ajust sp if yes
    it eq
    moveq sp, r0

    mov r4, lr
    bl __cpp(svc_handler_c)    //stack is fine, let's call into c
        add sp, #56
        bx r4
}
#elif defined ( __GNUC__ )

void SVC_Handler() {
    __asm volatile (
            " tst LR, #4 \n"        //which stack we were using?
```

```
    mov r4, lr
    bl __cpp(svc_handler_c)    //stack is fine, let's call into c
        add sp, #56
        bx r4
}
#elif defined ( __GNUC__ )

void SVC_Handler() {
    __asm volatile (
            " tst LR, #4 \n"        //which stack we were using?
            " ite eq \n"
            " mrseq r0, msp \n"
            " mrsne r0, psp \n"

            //hareware saved xpsr pc lr r12 r3-r0
            " mov r3, r0 \n"    //software save r11-r4 & sp
            " stmdb r0!, {r3, r4-r11} \n"

            " mrs r2, control \n"//software save trap changed regis
            " mov r3, pc \n"
            " mov r4, lr \n"
            " mov r5, sp \n"
            " mrs r6, psr \n"
            " stmdb r0!, {r2 - r6} \n"

            " tst lr, #4 \n"    //are we on the same stack? ajust s
            " it eq \n"
            " moveq sp, r0 \n"

            " mov r4, lr \n"    //reserve lr to r4
            " bl svc_handler_c \n"  //stack is fine, let's call int
            " add sp, #56 \n"       //make sp point to where R0 is
            " bx r4 \n"         //never return as lr ==exc_return
```

在很多嵌入式代码，或者是在大二学习《深入理解计算机系统》的时候，我们都见过有关的代码，这些代码是在 C 语言中潜入汇编指令，称为内联汇编。__asm__是 GCC 关键字 asm 的宏定义，__asm__或 asm 用来声明一个内联汇编表达式，所以任何一个内联汇编表达式都是以它开头的，是必不可少的。__volatile__是 GCC 关键字 volatile 的宏定义，__volatile__或 volatile 是可选的。如果用了它，则是向 GCC 声明不允许对该内联汇编优化，否则当 使用了优化选项(-O)进行编译时，GCC 将会根据自己的判断决定是否将这个内联汇编表达式中的指令优化掉。而这些个代码前面的#if defined...是给编译器看的，看这个代码是比较传统的 GNU 的还是 ARM 架构的。而这段代码就是模拟我们保存寄存器的动作。

而这段代码其实干的事情《手册》已经说得很清楚了。就是把现场“context”保存在不同任务的”PCB“中。

**如何通过 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：

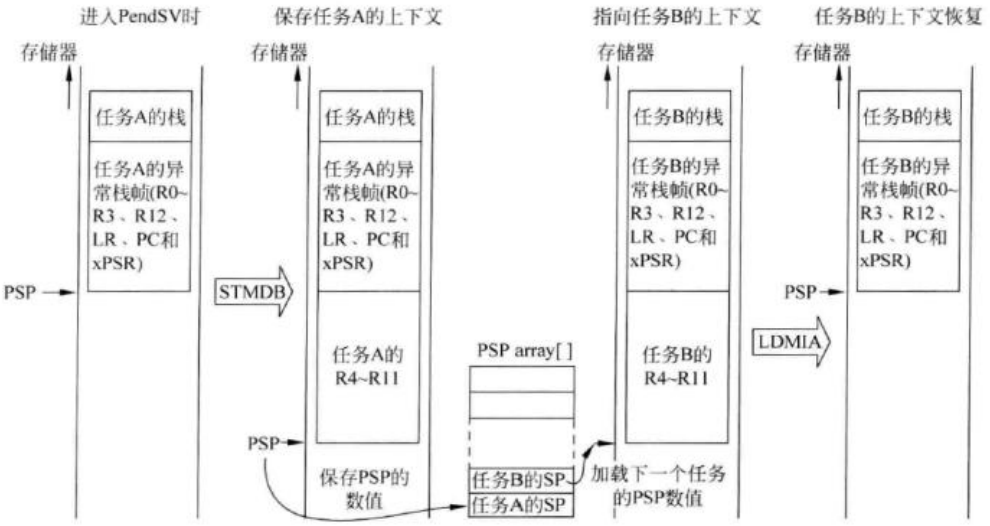
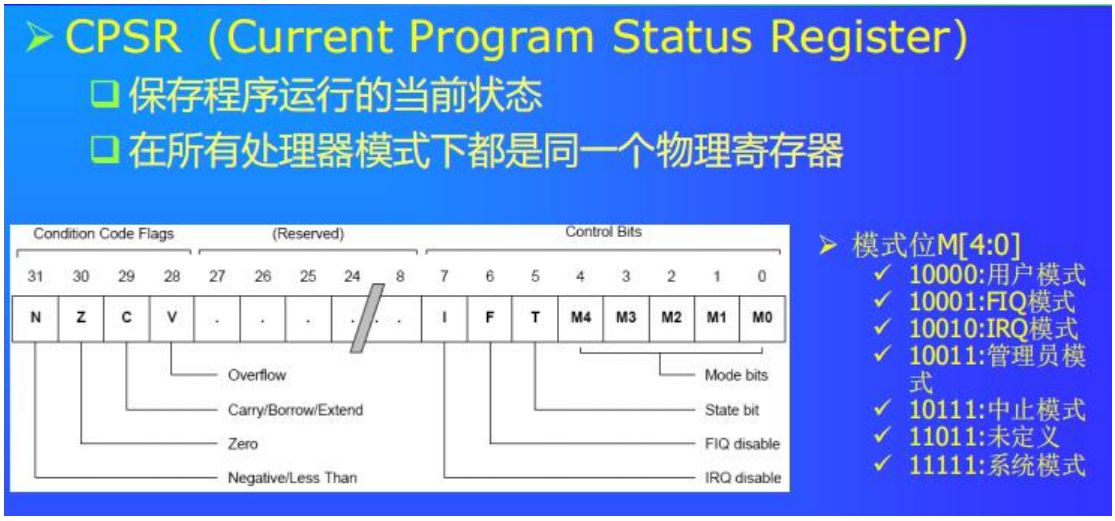



图 10.10 上下文切换

下面再简单回顾一下 ARM 的寄存器，因为我们在上面的确也用到了。


每种异常都有自己的 SPSR，共五个物理寄存器，进入异常时保存异常发生之前的 CPSR 当前值，而异常退出时恢复其 CPSR。CPSR 可以指示当前处于什么模式，也可以因为发生中断或者异常而自动改变 mode 位而进入相应的异常模式，每一种异常模式下都有一组备份寄存器，保证进入异常模式时用户模式下的寄存器不被破坏。




前面的理论知识暂时先铺垫到这里，而我们的任务是去分析触发前后的处理器模式、栈、特权等级：

**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：<br>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 |

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) | 0xEFFFFF(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 | 0xFFFFFFED | 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。

而我们设计的程序流程如下，需要我们去完成的是去分析前后的状态情况。

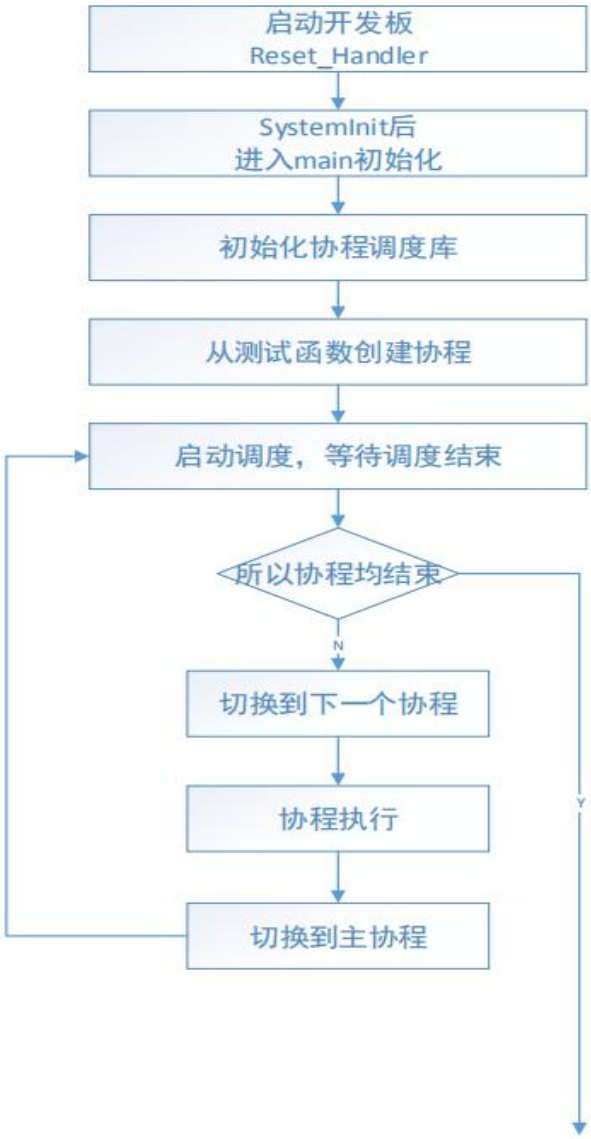

而我们通过 sscom 串口调试，可以发现打印消息是在 exp1_thread 里面，于是我们在后面加上一些位判断。于是可以写下如下核心代码：

```
//Save the swapped-out thread's stack ptr, return the to-be swapped-in thread's stack ptr
svc_stack_frame_t* thread_switch_intenal(svc_stack_frame_t *stack_save) {
    svc_nest_count++;
    if (!current_thread || !next_thread) {
        trace_printf("Stack frame on %08X:\r\n", stack_save);
        trace_printf(" xPSR     =  %08X\n", stack_save->xPSR);
        trace_printf(" PC    =  %08X\n", stack_save->PC);
        trace_printf(" LR    =  %08X\n", stack_save->LR);
        trace_printf(" R0    =  %08X\n", stack_save->R0);
        trace_printf(" EXC_RETURN =  %08X\n", stack_save->exc_return);
        trace_printf(" CONTROL    =  %08X\n", stack_save->control);
        trace_printf(" SVC NO     =  %08X\r\n", ((char*) stack_save->PC)[-2]);
        if((stack_save->control)&0x01) trace_printf("Non-Priviledged\r\n"); else trace_printf("Priviledged\r\n");
        if((stack_save->exc_return)&0x08) trace_printf("Thread mode\r\n"); else trace_printf("Handler mode\r\n");
        if((stack_save->exc_return)&0x04) trace_printf("PSP\r\n"); else trace_printf("Stack MSP\r");

        trace_printf("\r\n");
        svc_nest_count--;
        return stack_save;
    }
```

需要注意的是，我们这里没必要去改 exp1a 的代码，在用于输出调试信息的 exp1 thread 里面就行。

**在第二个实验 exp1b 中，本实验将异常调用上下文切换功能扩展到不同函数间**，并进一步通用化，完成一个基于“裸板”“协程”调度器，支持将函数创建为协程，并在协程间以 合作的方式完成调度（上下文切换）。

程序的流程图如下：




首先，我们来看看线程模型：

```
typedef struct thread {
    stack_t *context;          //pointer to current top of stack

    th_func_t func;         //user function entry
    void *arg;                //user function argument
    const char *name;          //name for debug
    unsigned int id;                    //which slot
    int run_ctr;               //statstics
```

```
    int alive;              //state: alive or dead
    int prio;       //priority. normal is zero. for interrupt backend thread
    int waiting;    //set when wait for timeout, cleared when timeout
    int delay;      //event delay, in milliseconds
    int repeat; //two type timers: oneshot and periodic. event occurs every 'repeat' millisecon
ds
    stack_t stack[STACK_SIZE];  //we use static stack
} thread_t;
```

**再说明一下：协同程序（coroutine）与多线程情况下的线程比较类似：有自己的堆栈，自己的局部变量，有自己的指令指针（IP，instruction pointer），但与其它协同程序共享全局变量等很多信息.协程，即协作式程序，其思想是，一系列互相依赖的协程间依次使用 CPU，每次只有一个协程工作，而其他协程处于休眠状态。协程实际上是在一个线程中，只不过每个协程对 CPU 进行分时.所以协程之间切换一般必须等待协程执行完.**因为协程之间必须依次使用 CPU，这样才能获得更好点调度效率。在执行完 1 之后，必须保存相应的堆栈信息

```
//user function for test
static void foo(Thread_ID *to) {
    int i = 0;
    state_num++;
    while (i++ < 1) {
        //ping()
        thread_switch_to(*to);
        //pang()
        thread_yield();
    }
}

//main entry
void exp1_b_main() {
    trace_printf("start exp1_b\n");

    threadlib_open();

    Thread_ID th1, th2, th3;
    //create thread
    th1 = thread_create((th_func_t) foo, &th2, "Function 1");
    th2 = thread_create((th_func_t) foo, &th3, "Function 2");
    th3 = thread_create((th_func_t) foo, &th1, "Function 3");

    //start thread
    thread_switch_to(th1);
//  thread_join(th1);
//  thread_join(th2);
//  thread_join(th3);

    thread_waitall();
    trace_printf("end exp1_b\r\n");
    // trace_printf("end exp1_b\n");
}
```

我们来分析一下 exp1b 的主函数，一开始，我们开启了线程池，创建了三个线程，分配给了 123 这三个函数，虽然他们都是执行 foo，然后程序切换到线程 1 进行工作，而它就是协程的入口，因为协程之间必须依次使用 CPU，才能获得更好的调度效率，并且执行完 1 之后，必须保存相应的上下文信息（他们有着自己的堆栈，自己的局部变量，自己的指令指针等），都保存在 TCB 里面。可能我们会对上面的 while(i++<1)感到疑惑，但其实他就是一个互斥锁，一旦有过程被唤醒，就修改 i 的值，while 语句一直等待唤起哪个线程去执行，while 就是一直循环调度的。协程其实是多线程的优化，它的执行效率是最高的，我们甚至可以去比较多线程的上下文切换次数和协程的切换次数，更直接一些。

说得直接一些,我们在 main 里面的 while 只是一个互斥锁,表示 Th1 和 th2 和 th3 只执行一个,而每一个 th 内我们也创建了很多”线程”,他们通过协程被维护。

同时，svc_nest_count 这个变量我也纠结了好久，但其实它的意义就是函数执行完毕后总累计切换了多少次。

具体每次 yield cpu 之后选择的协程实现如下。

```
static Thread_ID picked_rr = NULL;

#define Inc(id) id=(((id)<MAX_THREAD-1)?id+1:0)
//Schedule policy//# error "Not Implemented!"
//每个协程都有自己的堆栈，彼此之间通过主动调用 svc 现场切换系统调用让 cpu，不支持抢占，是一种合作式调度器
static Thread_ID pick_next() {
    trace_printf("pick_next alive thread count : %d\r\n", alive_thread_count);
    if(alive_thread_count == 1) {
        trace_printf("the last thread %d\r\n", current_thread->id);
        return main_thread;
    }
    unsigned int cur_id = current_thread->id;
    for (int i = cur_id + 1; i != cur_id; Inc(i)) {
        if (thread_heap[i].alive) {
            picked_rr = &thread_heap[i];
            break;
        }
    }
    trace_printf("original %d thread\t", current_thread->id);
    trace_printf("select %d thread\r\n", picked_rr->id);
    return picked_rr;
}
```

而以上代码说白了就是实现一下功能，但是是无抢占的,一旦所有协程均结束，我们要返回主线程。


## 2. 实验二 a&b:

个人觉得实验的精髓在实验 1，所以稍微说得多了些。

实验二 a 其实我觉得难点应该在 Morse code 解析部分，但老师都写好了，所以我们要做的仅仅是通过位运算来操作 GPIO 了

```
void buz1_init(void)
{
    //# error "Not Implemented!"
    GPIOx_MODER(BUZ1_PORT)|=1<<(2*BUZ1_PIN);
}

void buz1_on(void){
    //# error "Not Implemented!"
    GPIOx_ODR(BUZ1_PORT)|=1<<BUZ1_PIN;
}

void buz1_off(void){
    //# error "Not Implemented!"
    GPIOx_ODR(BUZ1_PORT)&=~(1<<BUZ1_PIN);
}
```

### 7.4.1 GPIO 端口模式寄存器 (GPIOx_MODER) (x = A..I)

GPIO port mode register

偏移地址：0x00

复位值：

- 0xA800 0000（端口 A）
- 0x0000 0280（端口 B）
- 0x0000 0000（其它端口）

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MODER15[1:0] | | MODER14[1:0] | | MODER13[1:0] | | MODER12[1:0] | | MODER11[1:0] | | MODER10[1:0] | | MODER9[1:0] | | MODER8[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MODER7[1:0] | | MODER6[1:0] | | MODER5[1:0] | | MODER4[1:0] | | MODER3[1:0] | | MODER2[1:0] | | MODER1[1:0] | | MODER0[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

位 2y:2y+1 **MODERy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 方向模式。

00：输入（复位状态）
01：通用输出模式
10：复用功能模式
11：模拟模式

### 7.4.6 GPIO 端口输出数据寄存器 (GPIOx_ODR) (x = A..I)

GPIO port output data register

偏移地址：0x14

复位值：0x0000 0000

1

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ODR15 | ODR14 | ODR13 | ODR12 | ODR11 | ODR10 | ODR9 | ODR8 | ODR7 | ODR6 | ODR5 | ODR4 | ODR3 | ODR2 | ODR1 | ODR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

实验流程：


摩尔斯码:

1. 摩尔斯电码是一种用通断表示英文字母、数字和标点符号的数字化编码方式，它由两种基本信号（点·和划-）和不同的间隔时间组成，编码包括五种：

| 编码方式 | 时间长度（t是单位时间） | 说明 |
| --- | --- | --- |
| 点（·） | 1t | 短促的点信号，读“滴”（Di） |
| 划（-） | 3t | 保持一定时间的长信号，读“嗒”（Da） |
| 滴嗒间 | 1t | 在每个点和划之间的停顿 |
| 字符间 | 3t | 每个字符间短的停顿 |
| 字间 | 7t | 每个词之间中等的停顿 |


对于 GPIO 的处理如下：

```
void buz1_init(void)
{
    //# error "Not Implemented!"
    GPIOx_MODER(BUZ1_PORT)|=1<<(2*BUZ1_PIN);
}

void buz1_on(void){
    //# error "Not Implemented!"
    GPIOx_ODR(BUZ1_PORT)|=1<<BUZ1_PIN;
}

void buz1_off(void){
    //# error "Not Implemented!"
    GPIOx_ODR(BUZ1_PORT)&=~(1<<BUZ1_PIN);
}

void buz1_play(void)
{
    buz1_on();
    HAL_Delay(1000*1);
    buz1_off();
    HAL_Delay(1000*1);
}
```

对于实验二的扩展实验：

按键处理：

```
void key_process() {
```

```
//# error "Not Implemented!"
struct key_state *ks;
for(int i=0;i<=3;i++)
{
    ks=&key_states[i];
    unsigned int interval=ks->trig_up_ms - ks->trig_down_ms;
    unsigned int ms=HAL_GetTick()%1000;
    //counter++;
    switch(i)
    {
        case 0:
            if(ks->output_trig_up&&interval >50&&interval<500)
            {
                HAL_CPU_Sleep();
            }
            break;
        case 1:
            if(ks->output_trig_up && interval>500)
            {
                led_flash=0;
                HAL_CPU_Reset();
            }
            break;
        case 2:
            if(ks->output_trig_up && interval>50 && interval <500 && k5_push)
            {
                led_flash=1;
                k5_push=0;
            }
            else if(ks->output_trig_up && interval>50&& interval<500)
            {
                k5_push=1;
            }
            break;
        case 3:
            if(ks->output_trig_down)
            {
                k6_on=1;
            }
            if(ks->output_trig_up)
            {
                k6_on=0;
            }
            break;
```

```
        default:
            break;
    }
}
}
```

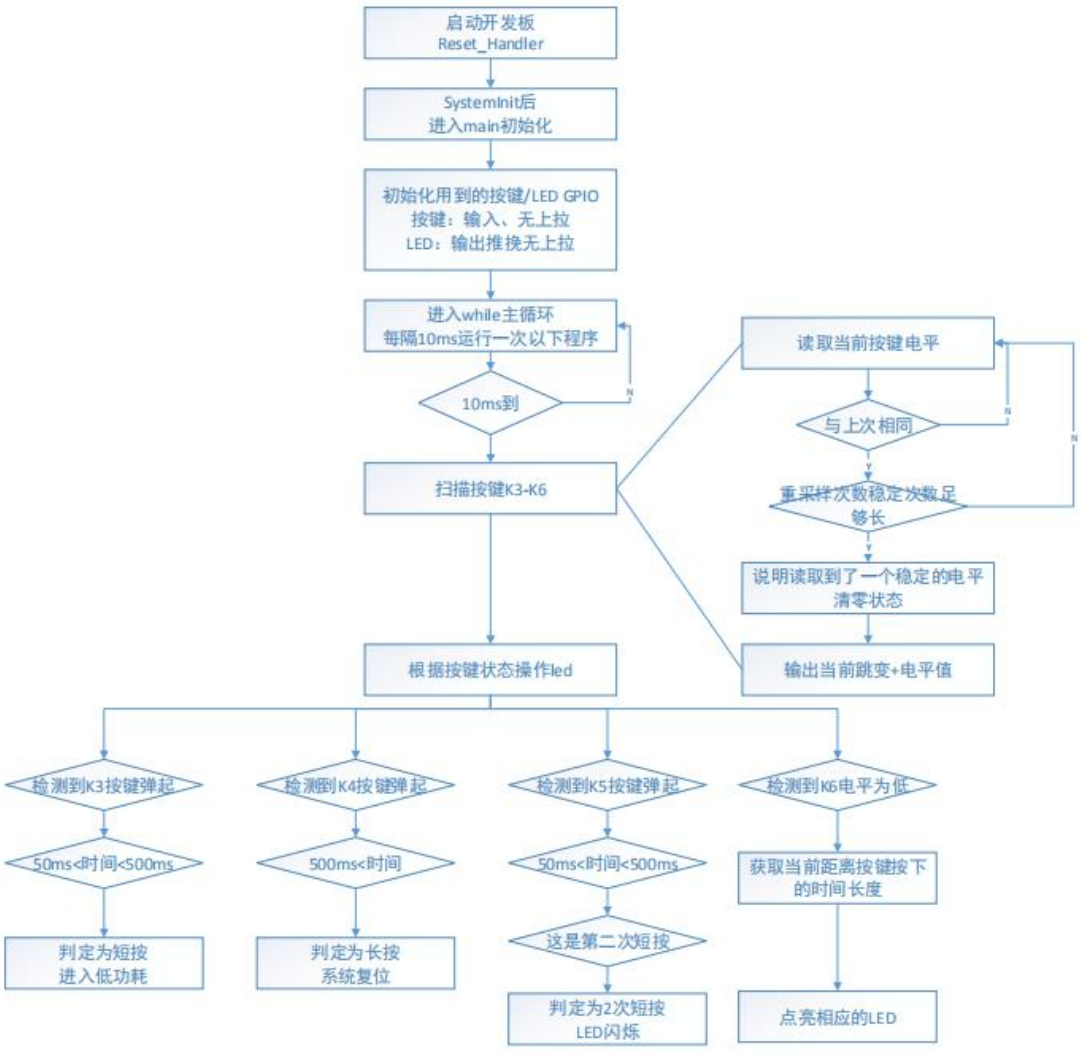
Main 函数：

```
static int timelimit=0;
static int last=0;

int stop_exp2_b;
void exp2_b_main()
{
    timelimit=HAL_GetTick();
    last=timelimit;
    exp2_b_init();

    while (!stop_exp2_b) {
        timelimit=HAL_GetTick();
        if(led_flash==1)
        {
            HAL_Led_flash();
        }
        struct key_state *ks=&key_states[3];
        if(k6_on){
            unsigned int ms=HAL_GetTick()%1000;
            unsigned int interval=ms - ks->trig_down_ms;
            switch((interval/100)%5){
                case 0:
                {HAL_Led_write(4,1);    break;}
                case 1:
                {HAL_Led_write(1,0);    break;}
                case 2:
                {HAL_Led_write(2,0);    break;}
                case 3:
                {HAL_Led_write(3,0);    break;}
                case 4:
                {HAL_Led_write(4,0);    break;}
            }
        }
        exp2_b_do();
        if((timelimit-last)>5000)
        {
            break;
        }
```

```
  }

  //exp2_b_do();
}
```

其实知道了这个问题就不大了：

```
120  void HAL_Led_write(unsigned int howmany, int on) {
121  //  trace_printf("HAL_Led_on howmany %d time %d\r\n", howmany, HAL_GetTick());
122      on &= 1;
123      if (howmany >= 1)
124          HAL_GPIO_WritePin(gpio_configs[LED6].Port, gpio_configs[LED6].Pin, (GPIO_PinState)on);
125      if (howmany >= 2)
126          HAL_GPIO_WritePin(gpio_configs[LED7].Port, gpio_configs[LED7].Pin, (GPIO_PinState)on);
127      if (howmany >= 3)
128          HAL_GPIO_WritePin(gpio_configs[LED8].Port, gpio_configs[LED8].Pin, (GPIO_PinState)on);
129      if (howmany >= 4)
130          HAL_GPIO_WritePin(gpio_configs[LED9].Port, gpio_configs[LED9].Pin, (GPIO_PinState)on);
131      //counter++;
132  }
133
```



## 3. 实验 3a&b

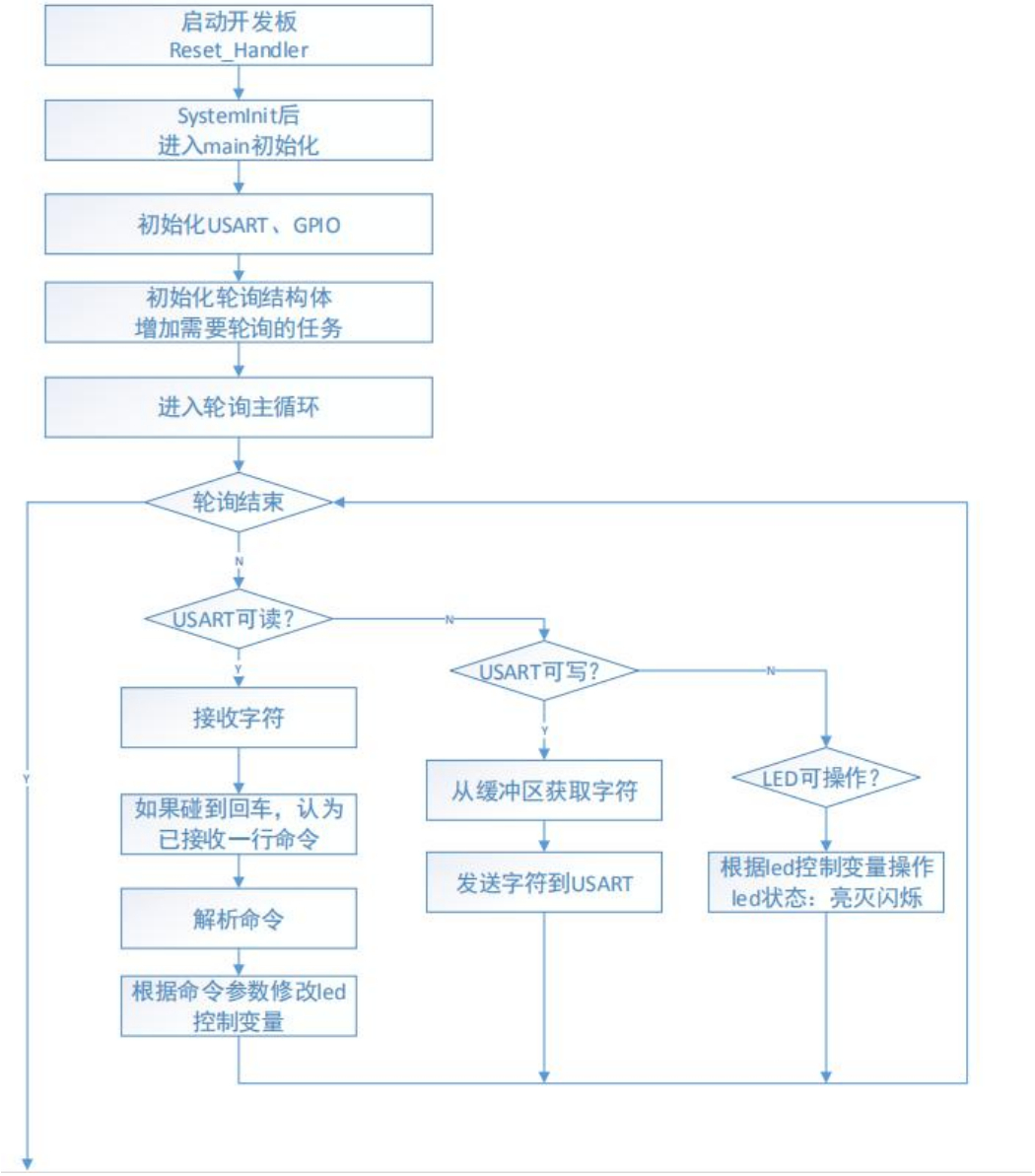
本实验我们需要完成的是通过计算机输入指令，通过 fifosend 缓冲区传给开发板，让开发板去执行我们输入的指令。而执行的实现其实就是写一个自动机。

本实验的难点有 2 处：
1. 串口不稳定，如何保证尽量不丢字符。
2. 如果使用轮询，CPU 在轮询串口状态时，如何同时控制 LED 的闪烁。

本实验的示例程序使用一种非阻塞的轮询结构。主循环将不断查询串口的收、发、led 的状态，但不死循环，如果条件不满足，立即查询下一处。通过在不同外设间来回查询，即可以做到一旦外设状态具备立即响应，也可以做到多个任务并发执行。

此外，使用一个循环缓冲来存储要发送给 usart 的字符。

程序流程：




同时我们也需要对轮询软件技术有个基本的认识：

本实验主要使用轮询式软件流程的技术：

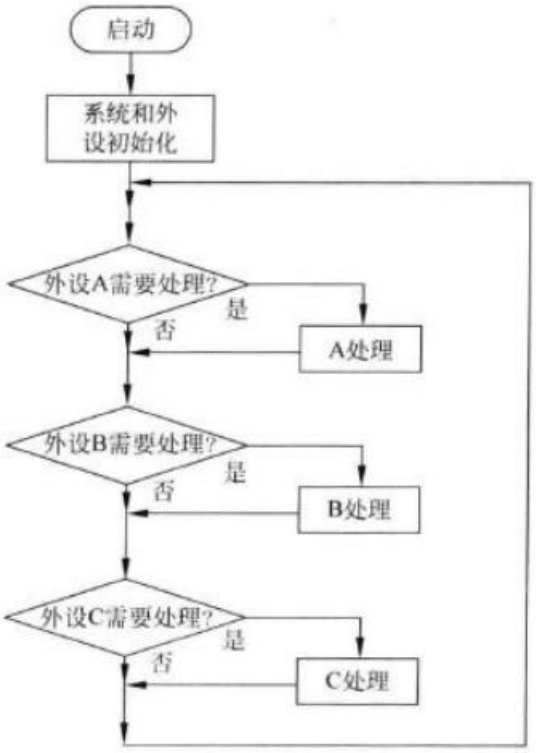



图 2.8　轮询方式的应用中存在多个需要处理的设备

在实现 3b 的时候，我们需要参考的信息如下：

**Shell 控制字：**

| | |
|---|---|
| Tab | `'\t'` |
| 回车 | `'\r'`, linux 终端`'\r'+ '\0'` |
| 上下左右 | `up key : 0x1b 0x5b 0x41`<br>`down key: 0x1b 0x5b 0x42`<br>`right key:0x1b 0x5b 0x43`<br>`left key: 0x1b 0x5b 0x44` |
| 光标后退/删除 | `'\b'` |

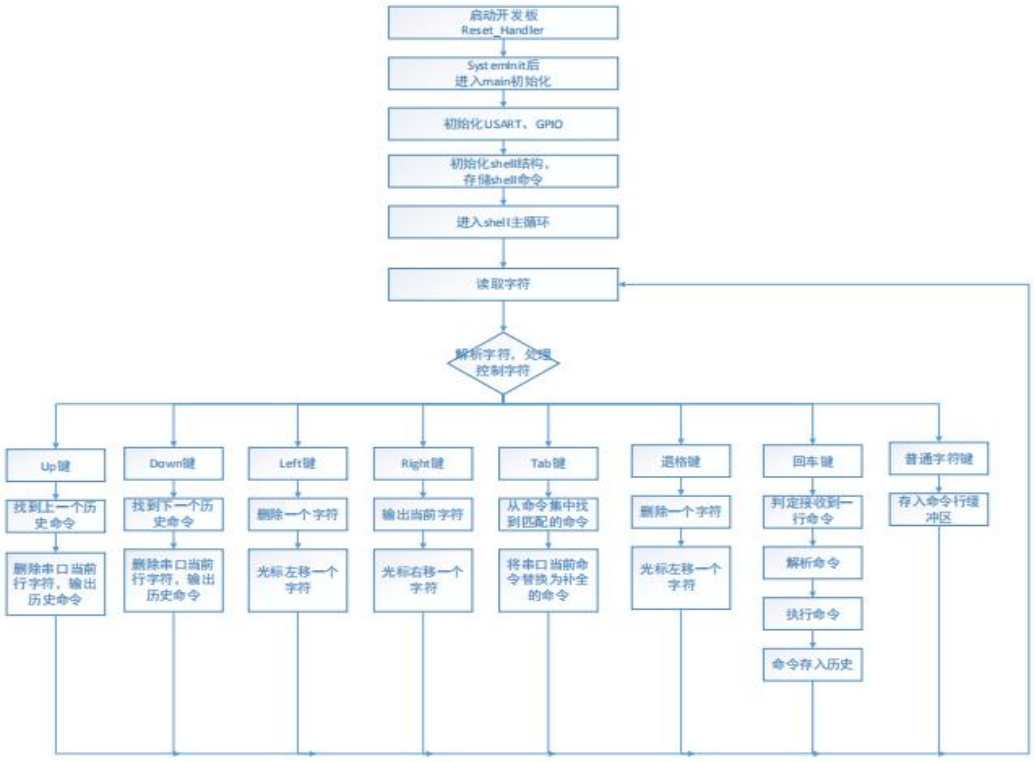
以下是 3a 的 uart_recv_process 实验实现：

```
void uart_recv_process(void) {
    //# error "Not Implemented!"
    char *instruction[5];
    char temp;
    int argc;
    if(board_getc_ready())
```

```
    {
        temp=board_getc();
        command[cmd_idx]=temp;
        cmd_idx++;
        fifo_push(&sendfifo,temp);
    }
    if(temp=='\r')
    {
        argc=split_cmdline(command,cmd_idx,instruction);
        led_cmd_exec(argc,instruction);

        int i=0;
        while(i<COMMAND_LEN_MAX)
        {
            command[i]='\0';
            i++;
        }
        cmd_idx=0;
        counter++;
    }
}
```

说一下实现的思路，首先我们等待轮询，如果键盘可以接收字符，则我们将每一次摁下的键赋值给 temp，并且连缀到我们的 command 里面，并且通过 fifopush 传值到缓冲区 sendfifo。如果输入为回车\r，则我们认为一条指令执行完毕，将指令写入“instruction”，并且记录指令由空格分开后的分词个数。

对于实验 3b，其实也是写一个自动机出来：




**Shell 控制字：**

| | |
|---|---|
| Tab | `'\t'` |
| 回车 | `'\r'`，linux 终端`'\r'+ '\0'` |
| 上下左右 | `up key : 0x1b 0x5b 0x41`<br>`down key: 0x1b 0x5b 0x42`<br>`right key:0x1b 0x5b 0x43`<br>`left key: 0x1b 0x5b 0x44` |
| 光标后退/删除 | `'\b'` |

本次实验我们在 putty 上实现，波特率为 38400。

自动机的代码如下：

```
char* shell_readline(struct uart_shell *shell) {
  int go = 1;
//
  while(go)
  {
    char c=shell->_getchar();
    if(c=='\r'||c=='\n')
    {
      shell_push_history(shell);
      go = 0;
      shell->line_position=strlen(shell->line);
```

```
            shell->line_curpos=0;
            ush_printf(shell, "\r\n");
            ush_printf(shell, "show the line:");
            ush_printf(shell, shell->line);
            ush_printf(shell, "\r\n");
            ush_printf(shell, "show the line lenth:");
            ush_printf(shell, "%d",shell->line_position);
            ush_printf(shell, "\r\n");
            break;
        }
        else if(c=='\t')
        {
            /*
            for(int i=0;i<shell->line_position;i++)
            {
                shell->_putchar('\b');
            }
            */
            ush_auto_complete(shell, shell->line);
            shell->line_position=strlen(shell->line);
            shell->line_curpos=shell->line_position;
        }
        else if(c==0x1b)
        {
            c=shell->_getchar();
            c=shell->_getchar();
            if(c==0x41)
        {
            if (shell->current_history > 0) {
                        shell->current_history--;
                        memcpy(shell->line,
                        shell->cmd_history[shell->current_history], CMD_SIZE);
                        shell_show_history(shell);
                        shell->line_curpos = shell->line_position = strlen(shell->line);
                    }
        }
        else if(c==0x42)
        {
            if (shell->current_history + 1 < shell->history_count) {
                        shell->current_history++;
                        memcpy(shell->line,
                        shell->cmd_history[shell->current_history], CMD_SIZE);
                        shell_show_history(shell);
                        shell->line_curpos = shell->line_position = strlen(shell->line);
```

```
                }
        }
        else if(c==0x44)
        {
            if(shell->line_curpos>0)
            {

            shell->line_curpos--;
            shell->_putchar('\b');
            }
        }
        else
        {
            shell->_putchar(shell->line[shell->line_curpos]);
            shell->line_curpos++;

        }
      }
        else
        {
            shell->_putchar(c);
            shell->line[shell->line_curpos++] = c;
            shell->line_position++;
        }
    }
    return shell->line;
    }
```

不过本次实验还有一些细节没有完全实现，比如我们输入一个指令，然后通过摁下左键移动光标，然后再 backspace 的时候有些不太人性化，但最后没再继续搞了。

这样，我们就实现了一个简单的 ARM SHELL 了。

## 实验四 a：

```
int stop_exp4_a;
static int ledon=0;
static int last=0;
static int counter=0;
//led10/D10 flash
static void led_flash(){
//  # error "Not Implemented!"
    int t=HAL_GetTick(); //TIM
    if(t-last>=2000)
    {
```

```
        last=t;
        if(ledon==0)
        {
            ledon=1;
            led_on(0);
            led_on(1);
            led_on(2);
            led_on(3);
            counter++;
        }
        else
        {
            ledon=0;
            led_off(0);
            led_off(1);
            led_off(2);
            led_off(3);
            counter++;
        }
    }
    //stop_exp4_a=1;
}
```

# 6 实验方案与实现

## 6.1 软件结构

在实验步骤中展现

## 6.2 源代码

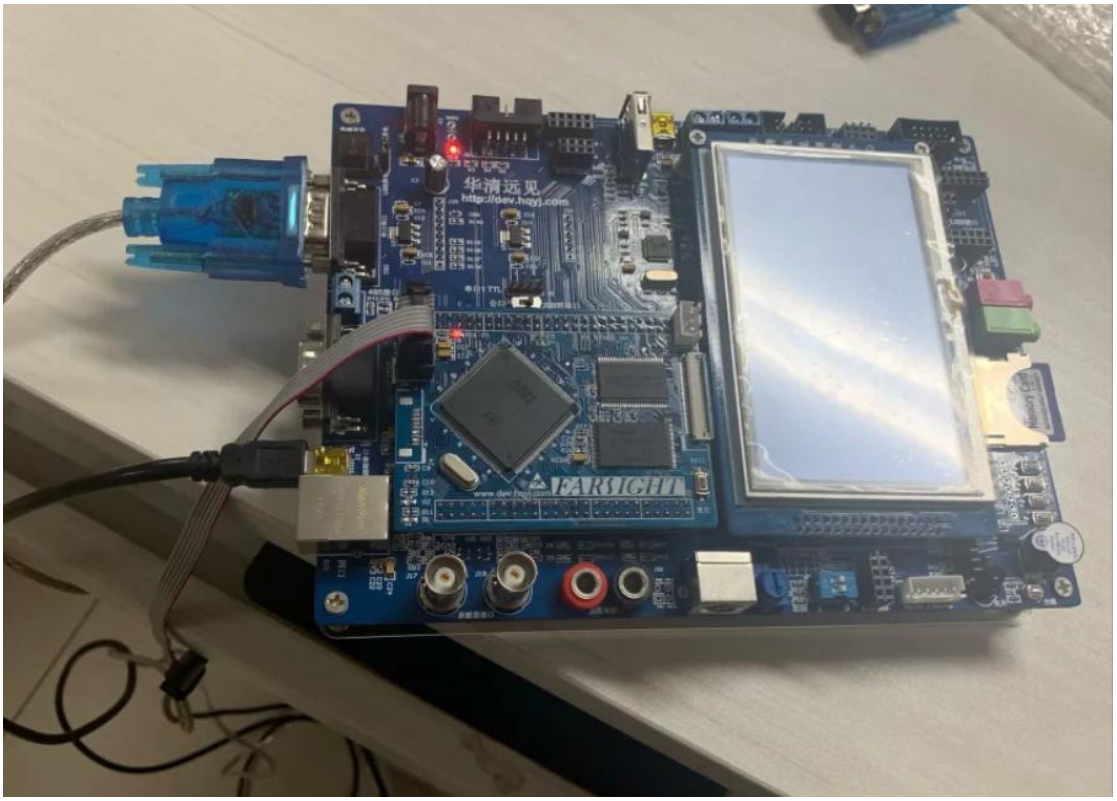
在实验步骤和代码附录中展现，也可前往

https://github.com/Sprinter1999/PlayGround 查看

# 7 实验结果与分析

实验 1ab 通过验收,实验 2ab 通过验收,实验 3ab 通过验收,实验 4a 通过验收

以下是部分实验的运行结果照片:




华清远见
http://dev.hqyj.com
FARSIGHT

# 8 实验总结

老实说，嵌入式实验是我大学至今做过最为硬核的实验。

在看懂代码，编写代码的时候，我用到了 C 语言的高级语法、操作系统有关知识、ARM Cortex M4 系列相关知识，GPIO 通用式输入输出串口、USART、计算机系统基础、轮询式软件架构等多种知识。虽然自己至今仍然对嵌入式了解不为深入，尤其是后面的 GPIO 和 USART 上课的时候就没记得太深刻，但真的算是很有收获了。在实验中，遇到了因为自己基础不扎实导致的一些代码看不太懂，然后请教老师和同学算是大概对每个实验的整体工作流有一个大致的认识，虽然自己亲自完成的代码只是这个庞大项目的冰山一角，但也借助完成实验的流程大概对嵌入式开发有了一个更加深刻的认识。虽然我们不能以行数来衡量我们思考的价值，但就这一个局部而言，我们完成的可能只有不到五百行代码，但可能整个系统的代码都超过五千行了，而且最难写的部分刘老师都给我们写好了。老师亲自完成的代码我个人认为更贴近嵌入式系统的一些精髓，这些也在验收的时候和老师大概聊了一下，不由得为老师的备课和编码感到惊异。本次实验也遇到了一些比较玄学的问题，但最后都好歹解决了。验收的时候，我也感受到了老师对于代码的严谨性有着要求能力，这一点也是学生需要后期去好好做到的一点。

本次实验 3a 是让同学给我大概讲了一下思路的，这里必须坦诚表明这一点，不过学生还是对整个实验 3 有了充足的认识并实现了一个 SHELL。学生完成全部实验的验收相对较早，4b 在之前实现了一个简单的 while 循环的轮询，但没有用到 thread 来完成，因为用到之后也遇到了一些玄学问题，不过学生没继续去深入思考了，最近课业压力也比较大，也需要好好准备复习嵌入式了。