

# 嵌入式系统实验报告



实验名称: GPIO 与系统状态

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

- 通过 FSM4 实验板了解实验的软硬件环境，熟悉 MDK 开发环境的使用。
- 学习查阅文档和数据手册，获取需要的信息。
- 学会使用 C 语言直接控制 IO 寄存器完成功能。
- 掌握基本的软件编写与调试方式。
- 学会 STM32 GPIO 的基本操作方式。

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

# 3 实验要求

- 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。
- 拓展要求：
  - ◆ 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

## 4 实验原理

### ● STM32 GPIO 的配置：

LED 灯的亮灭、蜂鸣器的鸣响、按键电平的读入都需要使用 STM32 芯片的 I/O 引脚。

STM32 芯片上，I/O 引脚可以被软件设置成各种不同的功能，如输入或输出，所以被称为 GPIO (General-purpose I/O)。而 GPIO 引脚又被分为 GPIOA、GPIOB...GPIOG 不同的组，每组端口分为 0~15，共 16 个不同的引脚。

如何使用，需要参考板子原理图、芯片的 datasheet 和 reference manual。

具体首先需要从原理图找到板上 LED、按键对应的 GPIO。

在实验板使用 D6 查找地底板原理图，可得：

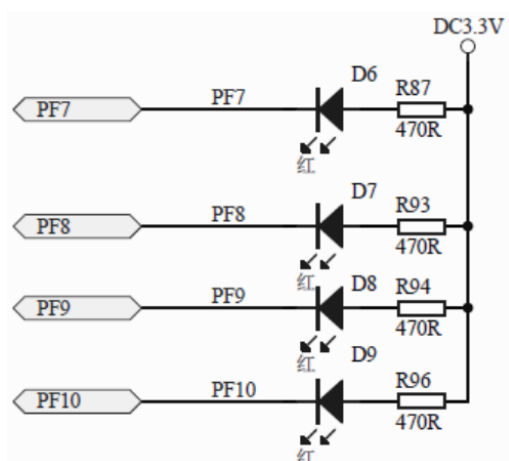
D6——PF7

D7——PF8

D8——PF9

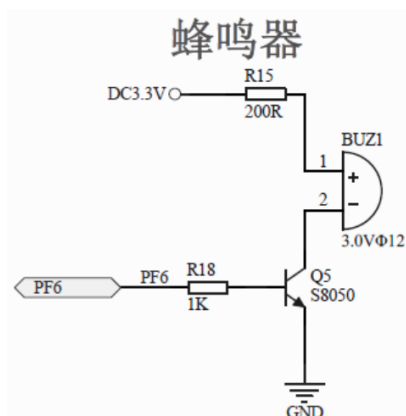
D9——PF10

控制 GPIO 的高低电平来控制 LED 的状态。0—亮，1—灭。



如使用蜂鸣器，则是通过 PF6 控制。1—响，0—灭

BUZ1——PF6



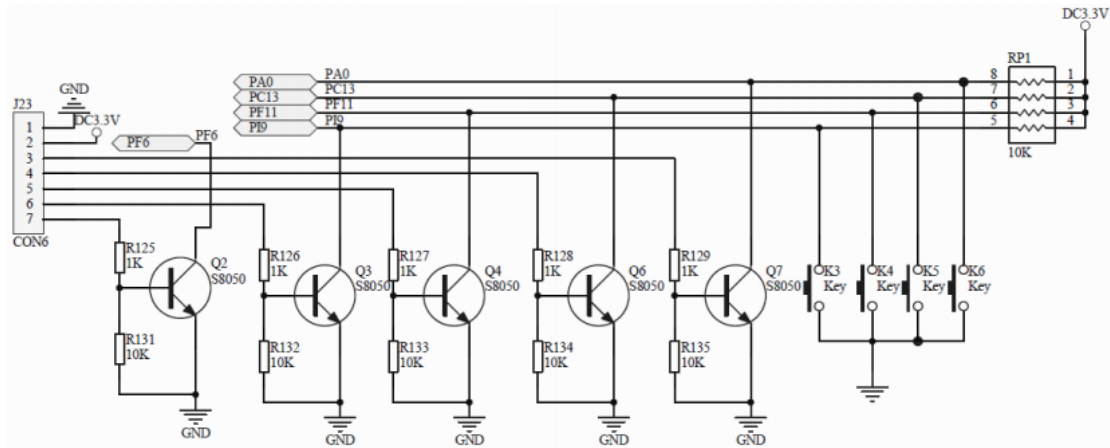
4 个按键：0—按下，1—断开

K3——PI9

K4——PF11

K5——PC13

K6——PA0



## ● LED 和按键：

LED：

- LD1 COM：LD1默认状态是红色。LD1变成绿色指示PC和ST-LINK/V2之间的通讯在进行中
- LD2 PWR：红色LED指示板子已供电
- 用户LD3：橙色LED是用户的LED，连接到STM32F407VGT6的PD13脚
- 用户LD4：绿色LED是用户的LED，连接到STM32F407VGT6的PD12引脚
- 用户LD5：红色LED是用户的LED，连接到STM32F407VGT6的PD14引脚
- 用户LD6：蓝色LED是用户的LED，连接到STM32F407VGT6的PD15引脚
- USB LD7：当VBUS在CN5上时，绿色LED指示，连接到STM32F407VGT6的PA9引脚
- USB LD8：红色LED指示CN5的VBUS的过流，连接到STM32F407VGT6的PD5引脚

按键：

- B1用户：用户和唤醒按键，连接到STM32F407VGT6的PA0
- B2复位：按键连接到NRST，用于复位STM32F407VGT6

## ● 程序控制 GPIO 的方法：

包括初始化和主逻辑 2 部分。

STM32 GPIO 的初始化需要 2 步：

1. 开启 GPIO 时钟
2. 配置好 GPIO 的寄存器

如果是复用 GPIO 管脚的外设，则一般需要 4 步：

1. 开启复用管脚的 GPIO 时钟
2. 配置好复用管脚的 GPIO 寄存器
3. 开启外设时钟

#### 4. 配置好外设的寄存器

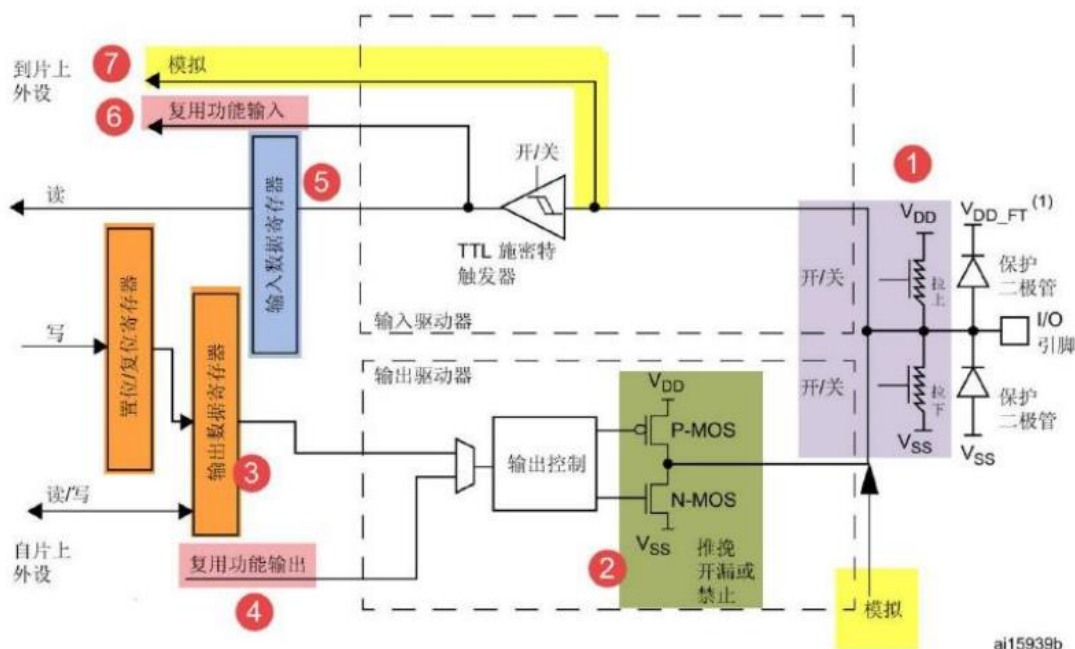
主逻辑部分主要是完成程序功能，如本实验的摩斯码编码输出。

一般主逻辑可分为 2 部分：

1. 与底层外设无关的逻辑部分。如摩斯码的编解码。
2. 需要通过外设与外部交互的驱动部分。如果摩斯码的滴答到 LED/蜂鸣器的映射。

GPIO 寄存器在手册《STM32F4xx 中文参考手册.pdf》第 7 章。

首先，需要从输入输出 2 个方向了解其原理。



然后根据具体管脚的需求设置寄存器。

7.4 GPIO 寄存器
7.4.1 GPIO 端口模式寄存器 (GPIOx_MODER) (x = A..I)
7.4.2 GPIO 端口输出类型寄存器 (GPIOx_OTYPER) (x = A..I)
7.4.3 GPIO 端口输出速度寄存器 (GPIOx_OSPEEDR) (x = A..I)
7.4.4 GPIO 端口上拉/下拉寄存器 (GPIOx_PUPDR) (x = A..I)
7.4.5 GPIO 端口输入数据寄存器 (GPIOx_IDR) (x = A..I)
7.4.6 GPIO 端口输出数据寄存器 (GPIOx_ODR) (x = A..I)
7.4.7 GPIO 端口置位/复位寄存器 (GPIOx_BSRR) (x = A..I)
7.4.8 GPIO 端口配置锁定寄存器 (GPIOx_LCKR) (x = A..I)
7.4.9 GPIO 复用功能低位寄存器 (GPIOx_AFR1) (x = A..I)
7.4.10 GPIO 复用功能高位寄存器 (GPIOx_AFR2) (x = A..I)
7.4.11 GPIO 寄存器映射

### 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

### 7.4.2 GPIO 端口输出类型寄存器 (GPIOx\_OTYPER) (x = A..I)

GPIO port output type register

偏移地址: 0x04

复位值: 0x0000 0000

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
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

位 31:16 保留, 必须保持复位值。

位 15:0 OTy[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 端口的输出类型。

- 0: 输出推挽 (复位状态)
- 1: 输出开漏



#### 7.4.4 GPIO 端口上拉/下拉寄存器 **GPIOx\_PUPDR** (x = A..I)

GPIO port pull-up/pull-down register

偏移地址: **0x0C**

复位值:

- 0x6400 0000 (端口 A)
- 0x0000 0100 (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
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

位 2y:2y+1 **PUPDRy[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 上拉或下拉。

**00**: 无上拉或下拉

**01**: 上拉

**10**: 下拉

**11**: 保留

#### 7.4.5 GPIO 端口**输入**数据寄存器 **GPIOx\_IDR** (x = A..I)

GPIO port input data register

偏移地址: **0x10**

复位值: 0x0000 XXXX (其中 X 表示未定义)

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留, 必须保持复位值。

位 15:0 **IDRy[15:0]**: 端口输入数据 (Port input data) (y = 0..15)

这些位为只读形式, 只能在字模式下访问。它们包含相应 I/O 端口的输入值。

#### 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

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

- 摩斯编码:

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

具体的编码方式如下表：

### 国际摩尔斯电码

1. 一点的长度是一个单位.
2. 一划是三个单位.
3. 在一个字母中点划之间的间隔是一点.
4. 两个字母之间的间隔是三点（一划）.
5. 两个单词之间的间隔是七点.

A	· —	U	· · —
B	— · · ·	V	· · — —
C	— · — ·	W	· — — —
D	— · ·	X	— · · —
E	·	Y	— · — —
F	· · — ·	Z	— — · ·
G	— — ·		
H	· · · ·		
I	· ·		
J	· — — —		
K	— · — —		
L	· — · ·	1	· — — — —
M	— —	2	· · — — —
N	— ·	3	· · — — —
O	— — —	4	· · · — —
P	· — — ·	5	· · · ·
Q	— — · —	6	— · · ·
R	· — ·	7	— — · ·
S	· · ·	8	— — — ·
T	—	9	— — — — ·
		0	— — — — —

FSM 实验板上有 4GPIO 控制的个 LED 灯：PF7（D6）、PF8（D7）、PF9（D8）、PF10（D9）。选择其中一个即可。

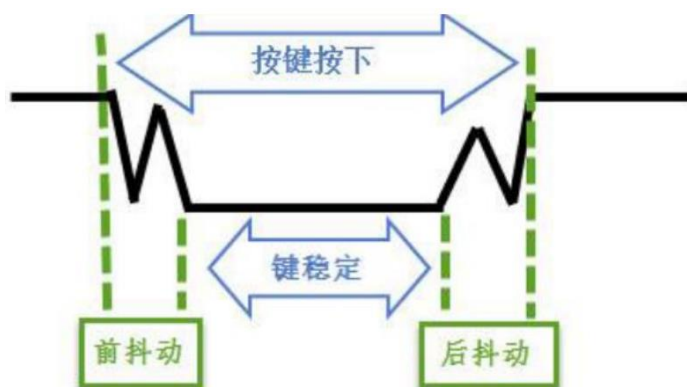
也可用蜂鸣器输出编码。

#### ● 按键扫描与消抖：

由于机械触点的弹性作用，按键开关在闭合时不会马上稳定的接通，而是有一段时间的抖动，在断开时也不会立即断开。



抖动时间由按键的机械特性所决定，一般为 5ms~20ms。如果不作处理这个抖动会给系统带来一些不稳定的因素，甚至是错误的结果，所以在做按键检测时都要加一个消抖的过程。




按键消抖可以是硬件电路消抖 or 软件消抖。

从开发板原理图上看并没有硬件消抖电路，所以需要软件消抖。

软件消抖可以有多种方案，本实验的示例采用了持续采样+延迟重采样的方法。首先识别按键跳变，然后在一段时期内重复采样，直到判定按键稳定。

## 5 实验步骤

### 5.1 基础部分

1. 下载并打开老师给的代码 `emlab2020-lab2.rar`  `emlab2020-lab2.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_b_main()`，设置断点，`build` 程序，然后先执行程序，观察程序的执行过程和现象。
4. 对照实验指导书，初始化蜂鸣器和 `led` 需要将端口模式改为通用输出模式，也就是将 `2y:2y+1` 位设置为 01（其中 `y` 为蜂鸣器或所用 `led` 灯的端口号），即可实现蜂鸣器和 `led` 的初始化。

### 5.2 附加部分

1. 打开项目中的 `lab2_b.c`，分析代码，发现实现低功耗模式、系统复位，`led` 点亮，`led` 按数量点亮的代码已经写好，主要是要正确处理 `key_process` 的逻辑以及它和这些函数之间的接口。
2. 通过分析 `key_process` 中老师给的第一个示例，按键 K3 弹起，记录 K3 从按下到抬起的过程所需时间，跟示例中已经设定好的 `PRESS_TIME_MS_SHORT` 和

PRESS\_TIME\_MS\_LONG 进行比较,判定是否为短按,然后决定是否进入低功耗模式

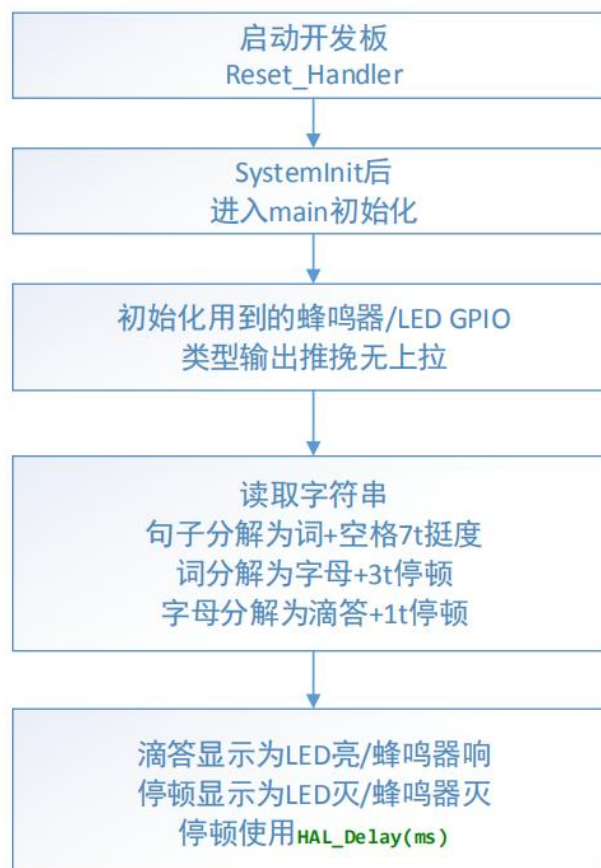
3. 以此类推,如果 K4 弹起,并且按键时间符合长按的限制,则系统复位;如果 K5 弹起,则判断是否短按两次,如果是,则进入一个 while(1)循环,实现 led 闪烁,当再次按下 K5,会跳出死循环,停止闪烁;如果 K6 按下,则开始计时,不断刷新已经按下的时间,根据时间长短调用 HAL\_Led\_write 函数时的 howmany 参数会发生变化,最多点亮 4 盏,当 K6 弹起时跳出循环。

4. 按照 3 中的逻辑在 key\_process 中编写相应代码即可。

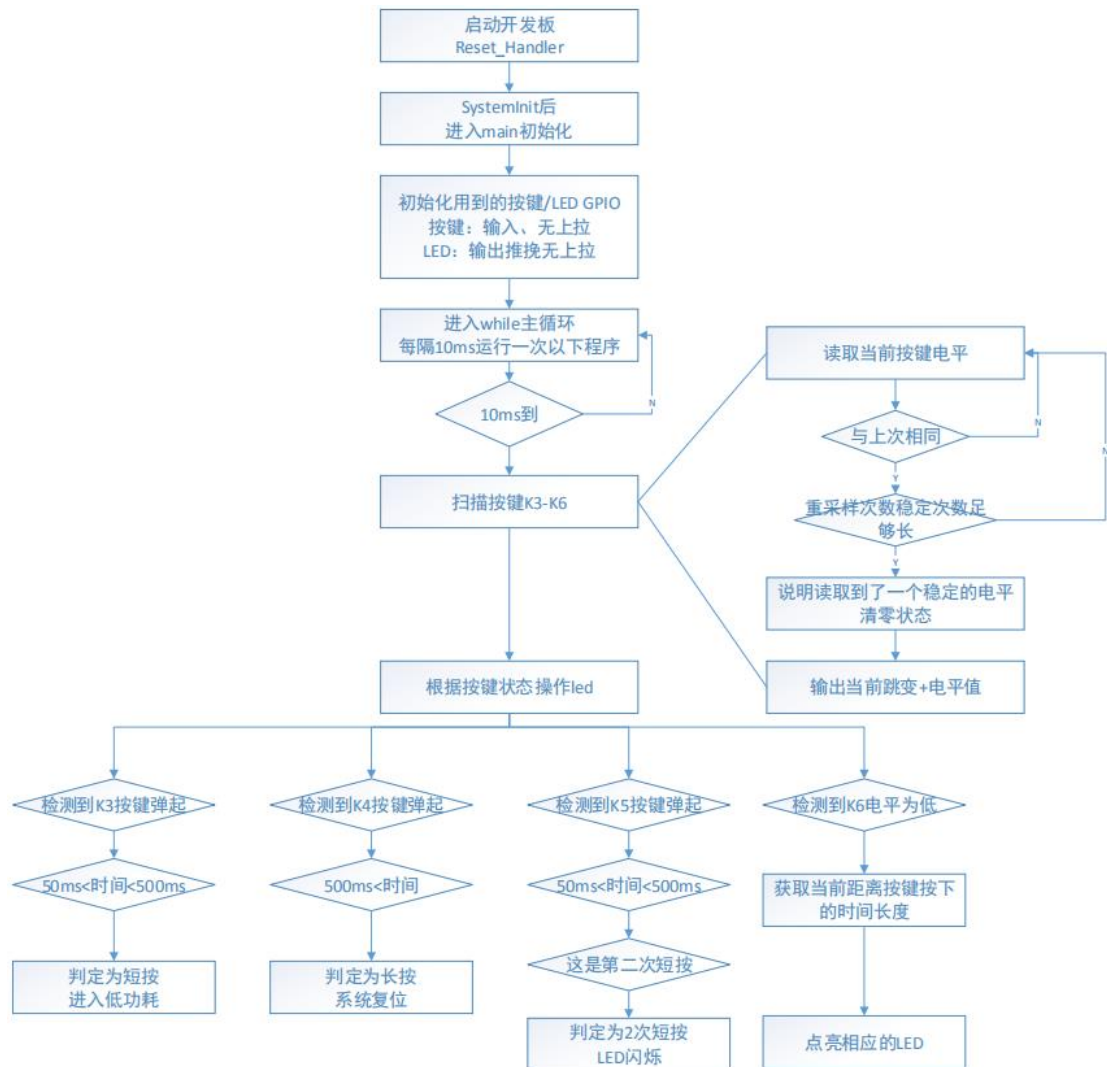
## 6 实验方案与实现

### 6.1 软件结构

- LED/蜂鸣器输出摩斯码的程序流程:



- 按键控制 LED 的程序流程：



## 6.2 源代码

- Lab2\_a:

```

void buz1_init(void)
{
    GPIOx_MODER(BUZ1_PORT) |= 1<<(BUZ1_PIN*2);
    GPIOx_MODER(BUZ1_PORT) &= ~(1<<(BUZ1_PIN*2+1));
}
  
```

```

void led6_init(void)
{
    GPIOx_MODER(LED6_PORT) |= 1<<(LED6_PIN*2);
    GPIOx_MODER(LED6_PORT) &= ~(1<<(LED6_PIN*2+1));
}

```

● Lab2\_b:

```

void key_process() {
    //detect signal and handle by state
    if (key_states[K3].output_trig_up) {
        struct key_state *ks = &key_states[K3];
        unsigned int presstime = ks->trig_up_ms - ks->trig_down_ms;
        //short push
        if ((PRESS_TIME_MS_SHORT < presstime)
            & (presstime < PRESS_TIME_MS_LONG)) {
            HAL_CPU_Sleep();
        }
    }

    else if (key_states[K4].output_trig_up) {
        struct key_state *ks = &key_states[K4];
        unsigned int presstime = ks->trig_up_ms - ks->trig_down_ms;
        //long push
        if ( (presstime > PRESS_TIME_MS_LONG)) {
            HAL_CPU_Reset();
        }
    }

    else if (key_states[K5].output_trig_up) {
        struct key_state *ks = &key_states[K5];
        unsigned int presstime = ks->trig_up_ms - ks->trig_down_ms;
        //short push
        if ((PRESS_TIME_MS_SHORT < presstime)
            && (presstime < PRESS_TIME_MS_LONG)) {
            count++;
            if (count ==2)
            {
                while(1)
                {
                    HAL_Led_flash();

```

```

        key_states[K5].output_trig_up
    }
    count=0;
}

}

}

else if (key_states[K6].output_trig_down) {
    struct key_state *ks = &key_states[K5];
    unsigned int presstime = ks->trig_up_ms - ks->trig_down_ms;
    HAL_Led_write(presstime,0);
}

}

```

## 7 实验结果与分析

基础部分的实验结果是正确初始化蜂鸣器和 led 后，会正确输出“Hello Cortex-M4”的摩斯码,蜂鸣器根据摩斯码发出正确的声响。

附加部分的实验结果是：按下 K3 之后系统会进入低功耗模式；长按 K4 系统复位，led 灯的状态复位；按键 K5 双击，led 灯开始闪烁，再次点击 K5，闪烁会停止；长按按键 K6，随着按动时长，4 个 led 灯会依次点亮。

## 8 实验总结

实验的关键是控制蜂鸣器的响与静。找到蜂鸣器的端口号，BUZ1\_PORT、BUZ1\_PIN。分别为 5 和 6。蜂鸣器开将寄存器 ODR 的端口 5 设置为 1，关将端口号 5 置为 0。

初始化 MODER 寄存器为通用输出模式：01

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

实验心得：不能空想，要结合上下文仔细观察，弄明白相关函数的意义。比如这里，自己写将寄存器的某一位设为 0 或 1 有些许困难，但联系上下的函数，发现 buz1\_on 里有将某一位置 1 的操作，buz1-off 里有将某一位置 0 的操作，因此将 2y: 2y+1 位设为 01 的问题就迎刃而解。

```

void buz1_init(void)
{
    // #error "UNIMPLEMENTED!"
    GPIOx_MODER(BUZ1_PORT) |= 1 << BUZ1_PIN*2; //mode: 01 input
    GPIOx_MODER(BUZ1_PORT) &= ~(1 << (BUZ1_PIN*2+1));
}

void buz1_on(void){
    GPIOx_ODR(BUZ1_PORT) |= 1 << BUZ1_PIN;
}

void buz1_off(void){
    GPIOx_ODR(BUZ1_PORT) &= ~(1 << BUZ1_PIN);
}

```

另外，在附加实验中，K5 和 K6 按钮功能的实现耗费了很多时间。最开始没有将 flash 函数放在一个循环里，发现并不能实现闪烁，放到 while(1) 循环里又苦于不能停止；根据程序流程图，K6 的判定条件和其他三个不同，是在按钮按下时就进入 if，刚开始不太懂得这样设定的意义，在一次次的修改和最终成功实现中，回头看就明白了为什么会有这种差别。因为其他三个的判定与按键时间有关，只有在按键弹起时才能获得按下到弹起的时间，因此是 trig\_up；而 K6 需要在按下的过程中不断刷新已经按下的时间，如果也是 trig\_up 只能得到根据总的按下时间点亮灯，而没有随着时间变长灯变多的现象。