# 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

# 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。

# 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

# 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

# 

# 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 mannual。

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：

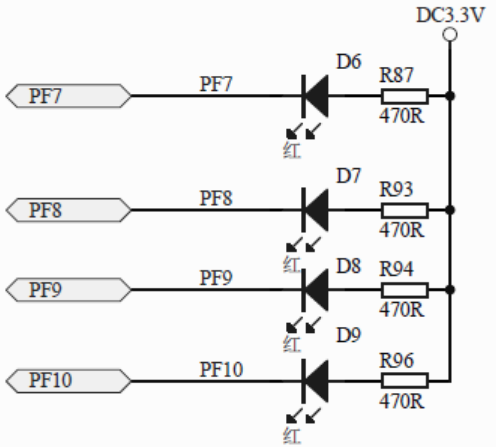
D6——PF7

D7——PF8

D8——PF9

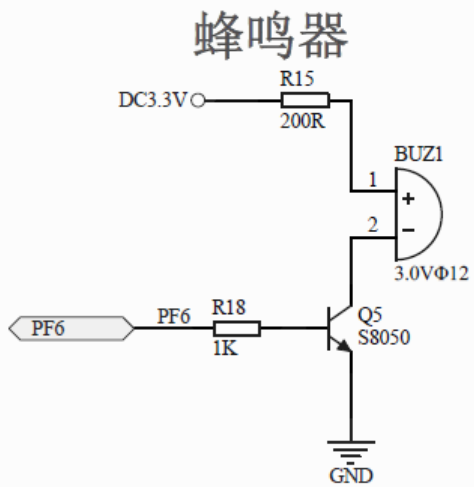
D9——PF10

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



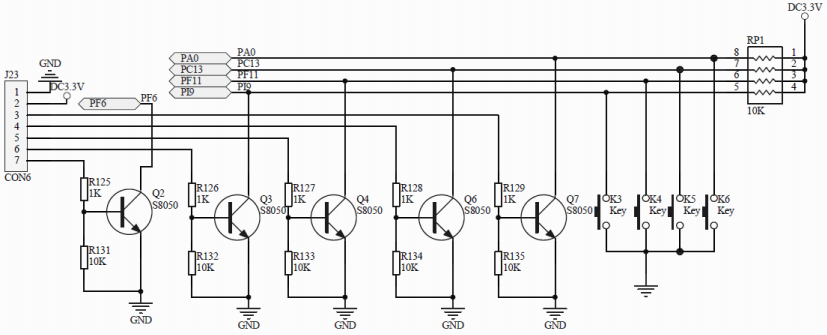
4 个按键：0—按下，1—断开

K3——PI9

K4——PF11

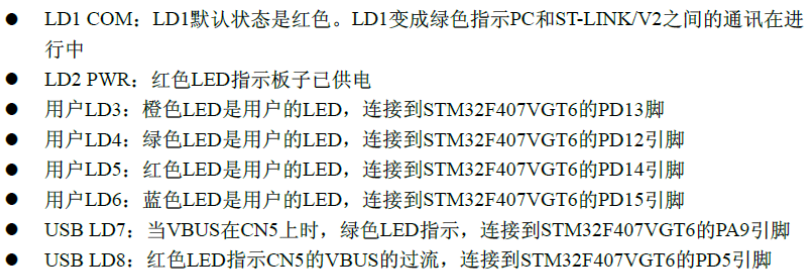
K5——PC13

K6——PA0

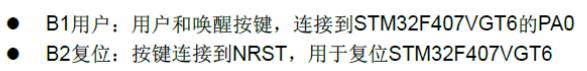


* **LED和按键：**

LED：



按键：



* **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。

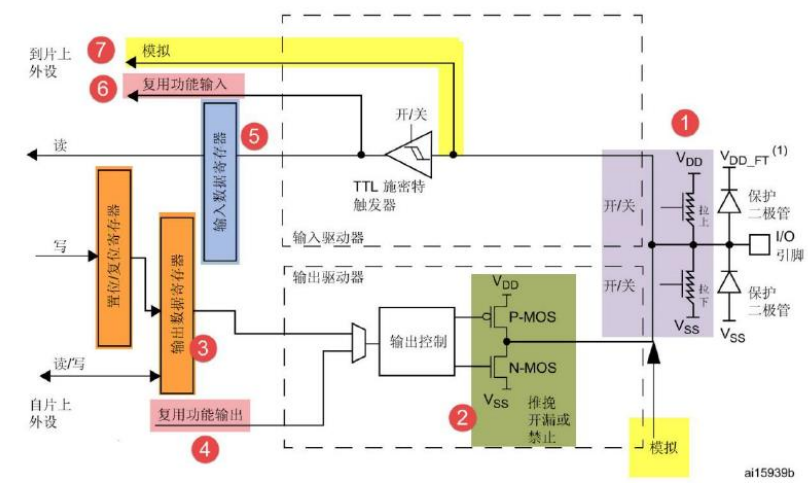
一般主逻辑可分为 2 部分：

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。

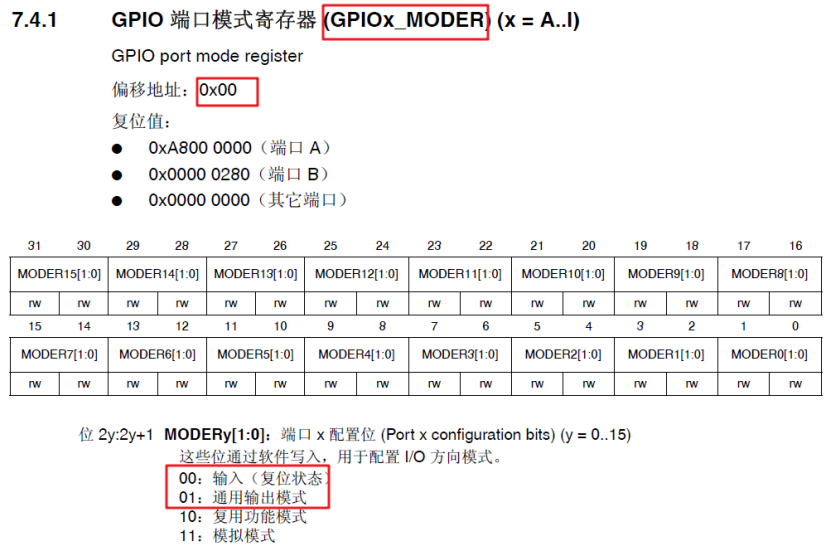
GPIO 寄存器在手册《STM32F4xx 中文参考手册.pdf》第 7 章。

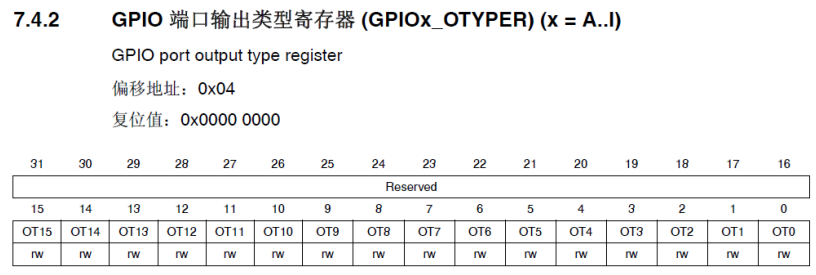
首先，需要从输入输出 2 个方向了解其原理。

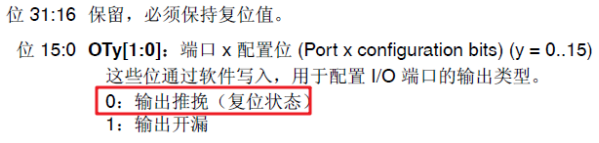


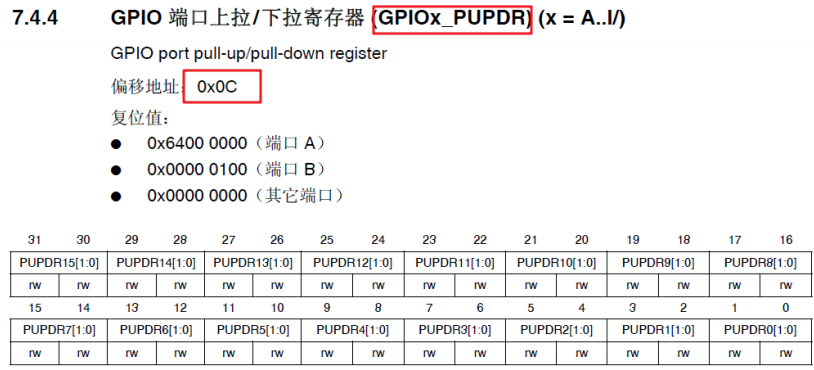
然后根据具体管脚的需求设置寄存器。

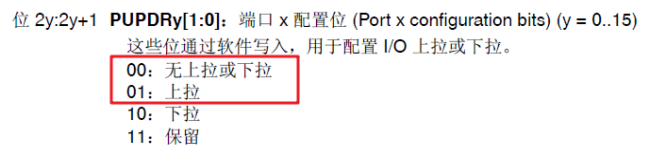


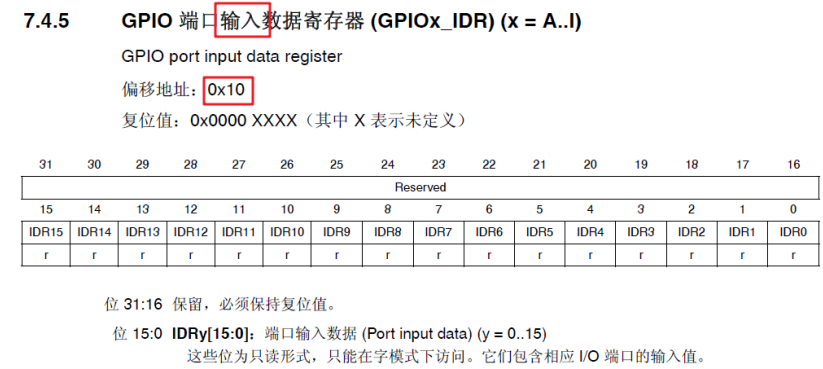


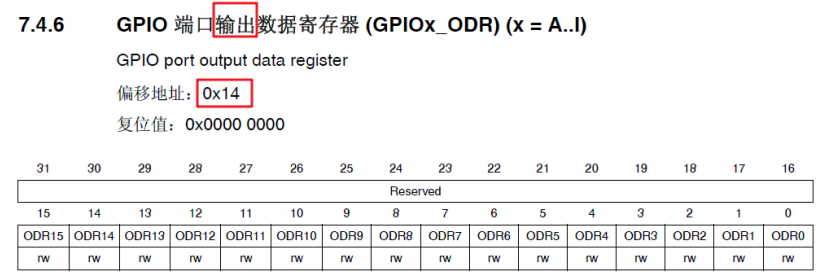










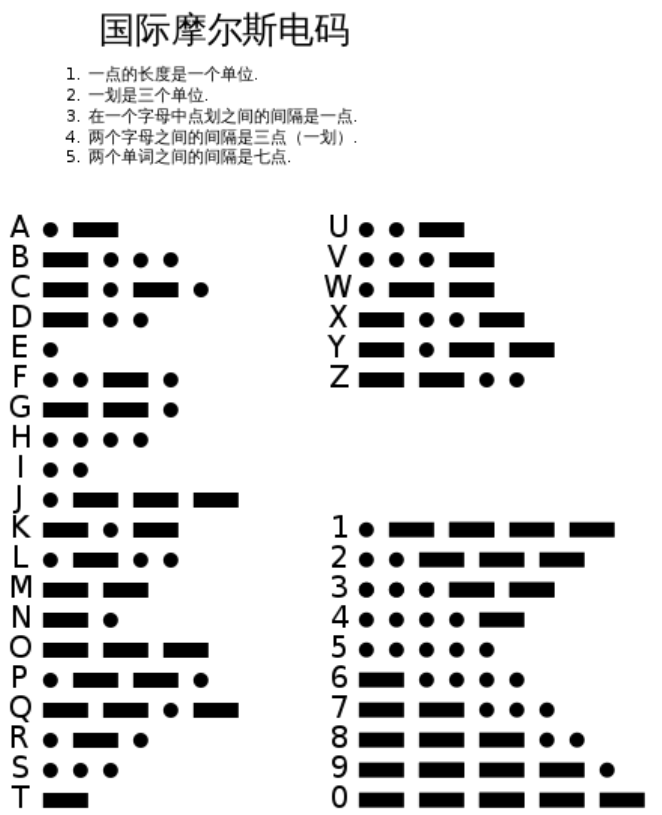
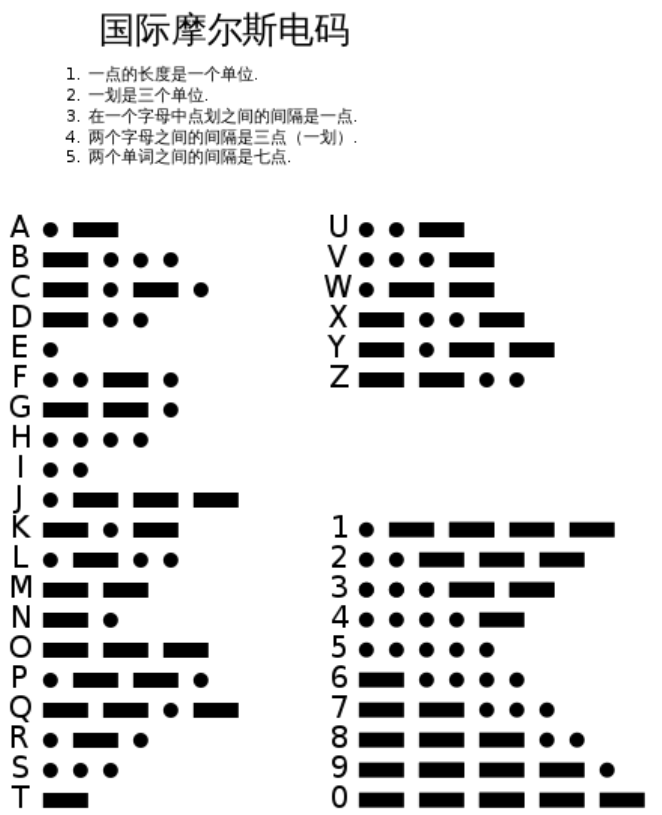


* **摩斯编码：**

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：



具体的编码方式如下表：

· 

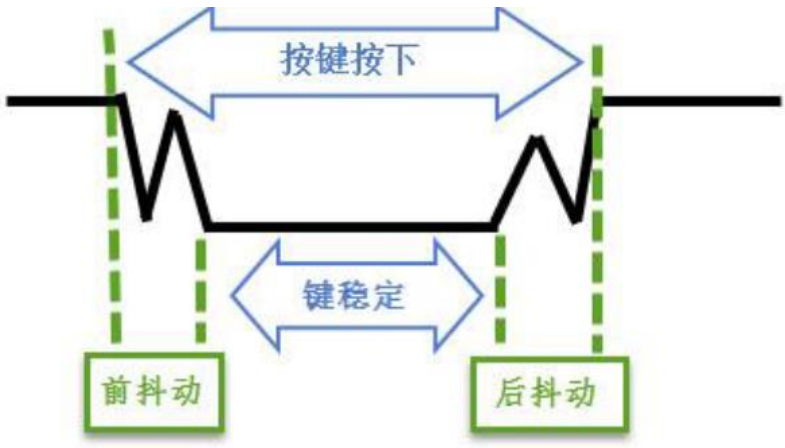
FSM 实验板上有 4GPIO 控制的个 LED 灯：PF7（D6）、PF8（D7）、PF9（D8）、PF10（D9）。选择其中一个即可。

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。

# 实验步骤

## 基础部分

1. 下载并打开老师给的代码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。

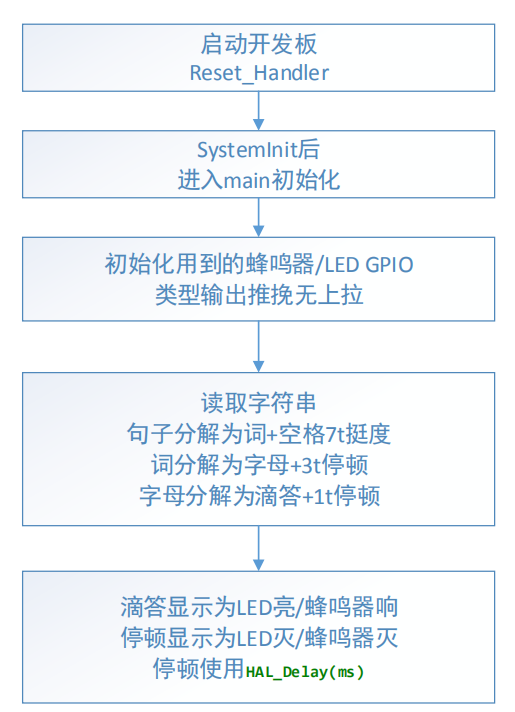
## 附加部分

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。

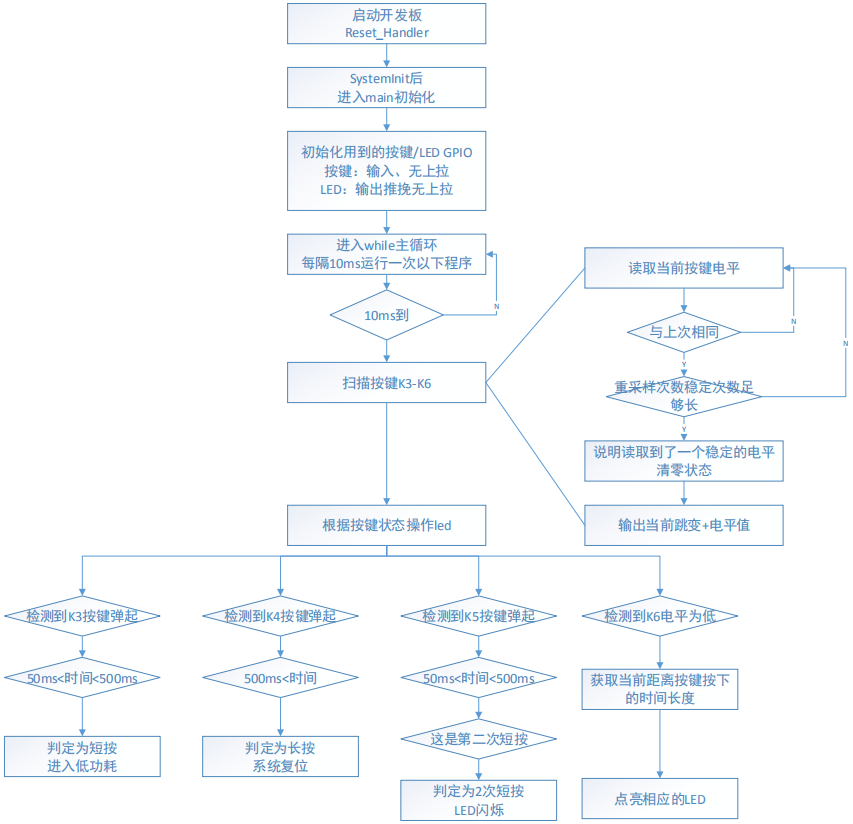
# 实验方案与实现

## 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：



## 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);

}

}

# 实验结果与分析

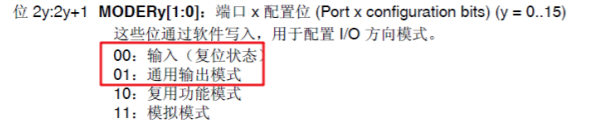
基础部分的实验结果是正确初始化蜂鸣器和led后，会正确输出“Hello Cortex-M4”的摩斯码,蜂鸣器根据摩斯码发出正确的声响。

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。

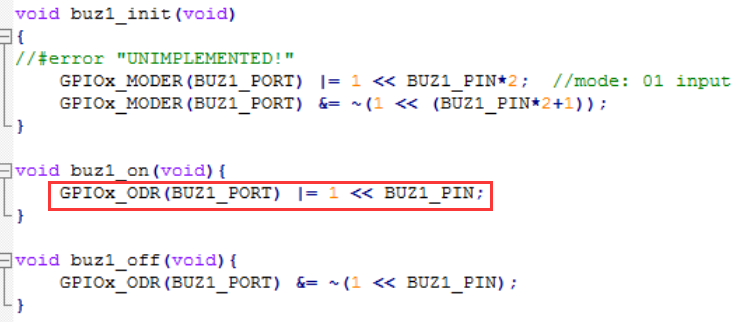
# 实验总结

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



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。



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。