

GPIO/中断-按键扫描和数码管综合实验报告

吴立文 23061925

1. 实验要求

针对按键扫描和数码管实验，撰写实验报告，至少包含以下内容：

- 1) 总结描述寄存器方式编程、库函数方式编程的区别，可以通过文字加代码截图的方式描述
- 2) 引入中断后，对于按键扫描和数码管显示的编程改变
- 3) 进入调试模式下，先全速运行，再停止运行，此时通过 Keil 的外设窗口，查看按键或数码管引脚的配置值，给出一个截图（CRL 或 CRH），并说明数值含义，证明正确性。
- 4) 在 watch 窗口添加一个变量，并查看数值（例如按下某个按键，此时扫描得到的数值），说明正确性。
- 5) 总结实验心得。
- 6) 上传引入按键中断和定时器中断后的代码压缩包和实验报告。

2. 实验设计

2.1. 寄存器编程与库函数编程

STM32 芯片有众多配置项：包括最基础的 CPU 频率、中断配置、引脚输入输出模式等等，加上引脚的输入输出值，都需要用户在程序中编程完成。实现上，STM32 通过把这些配置项与输入输出映射到固定的内存地址上，并通过手册约定了写入某值时，表示设定了什么内容，例如将 GPIOA 的第 5 引脚配置为推挽输出模式，只需要向 GPIOA_MODER 的第 10、11 两个比特写入 01；之后再通过 GPIOA_ODR 寄存器对应位置写 1 或 0，就能控制该引脚输出高电平或低电平。而 GPIOA_MODER 与 GPIOA_ODR 寄存器可以通过手册查询出它对应的具体内存地址。

由此产生了两种不同的编程方法：寄存器编程与库函数编程。

顾名思义，寄存器编程根据需要，每次向特定内存地址写入所需要的比特或读出数据来完成配置修改或输入输出。如以下为单个引脚初始化代码所示：

```
#define RCC_APB2ENR      (*(volatile unsigned int*)0x40021018))  
    // APB2 外设时钟使能寄存器  
#define GPIOB_CRL       (*(volatile unsigned int*)0x40010C00))  
    // GPIOB 端口配置低寄存器 (配置 PB0~PB7)  
#define GPIOB_BSRR      (*(volatile unsigned int*)0x40010C10))  
    // GPIOB 端口位设置/复位寄存器  
  
void KEY_Init(void)  
{  
    RCC_APB2ENR |= 1 << 3;      // 使能 GPIOB 时钟 (IOPBEN = 1)  
    GPIOB_CRL &= ~(0xF << 24); // 清除 PB6 的配置位 (第 24~27 位)  
    GPIOB_CRL |= (0x08 << 24); // 将 PB6 设置为上拉输入模式: CNF=10, MODE=00  
    GPIOB_BSRR = (1 << 6);     // 向 PB6 写入 1: 设置为上拉 (BSRR 的 Set 部分会写  
ODR)  
}
```

这样子做的话好处是实际运行效率高，每次操作都只涉及到简单位运算与访存，但坏处也很明显：若没有注释，整个代码充满了“magic number”，想要修改代码必须仔细阅读手册，即使一位数字的差异也会导致功能完全失效。阅读困难，调试困难。

与之对应，库函数编程通过为寄存器抽象，定义方法并调用方法完成寄存器的修改，如下代码完成了完全相同的功能：

```
#include "stm32f10x.h"  
  
void KEY_Init(void)  
{  
    GPIO_InitTypeDef GPIO_InitStruct;  
  
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // 使能 GPIOB 时钟  
  
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6;           // 选择 PB6  
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IPU;        // 上拉输入模式  
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz; // 输入速度  
    GPIO_Init(GPIOB, &GPIO_InitStruct); // 配置 PB6  
  
    GPIO_SetBits(GPIOB, GPIO_Pin_6); // 置高 PB6  
}
```

`GPIO_InitTypeDef` 结构体中包括了初始化一个引脚的全部内容：引脚编号、输入输出模式、输出输出速度。而该结构体中的各个成员也有预设的枚举

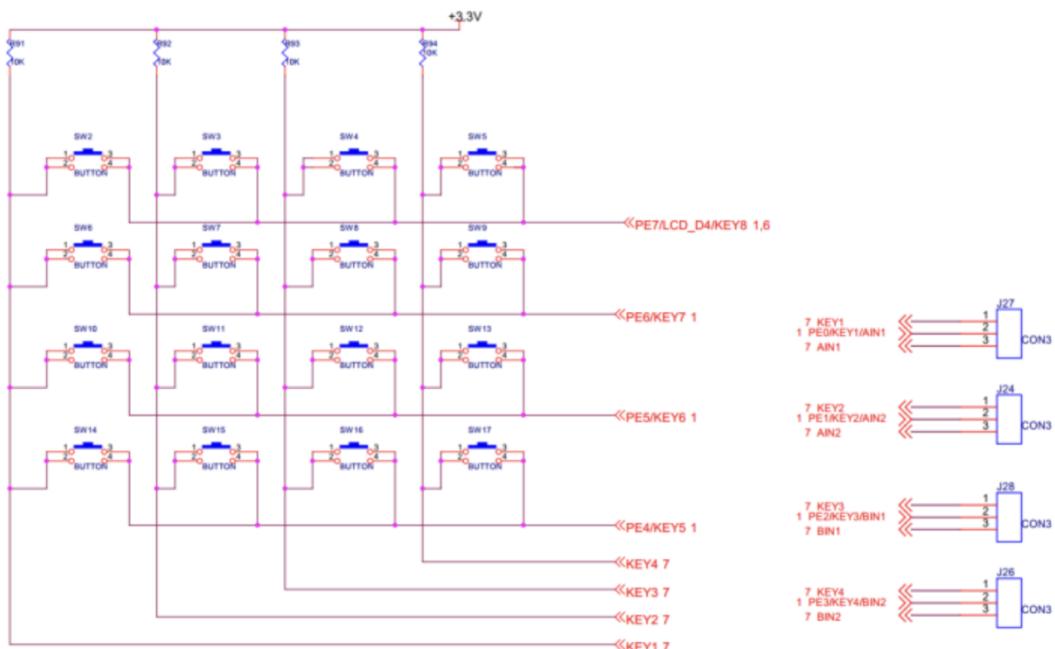
类型，只需要掌握库函数的命名规范，便可轻松完成赋值。**GPIO_SetBits** 函数也通过预先定义的 **GPIOB** 与 **GPIO_Pin_6**，方便完成了引脚的拉高。

库函数编程的好处是代码编写方便，阅读直观。只需了解各类接口，掌握命名规范，便可以轻松完成各类操作。而缺点是库函数编程为了实现方法的通用性，牺牲了部分性能。且引入库文件可能会增加编译后需要写入芯片的数据量，这也是不可忽略的。

2.2. 非中断方法的按键扫描与数码管显示

2.2.1. 按键扫描

实验板拥有 4*4 矩阵键盘，电路图如下：



按键扫描的原理为：我们将 PE0~3 设为输入，PE4~7 设为输出。先将 PE4~7 置 0，若无按键按下，按键电路不导通，PE0~3 能得到 3.3v 输入的高电平，即输入为 1。当有按键按下，按键电路导通，对应列的电路会被按键电路导通至 PE4~7 输出的 0，电位下降。对应列的输入将会变为 0，我们便能识别有按键按下。

仅知道有按键被按下是不够的，我们还要需要知道具体是哪个按键。我们通过逐行扫描的方式来找到这个被按下的按键：我们将 PE4~7 全部拉高，然后逐一拉低。假设此时我们拉低 PE7（第 1 行），其余行拉高：

- 若按下的按键不在第一行当中，则导通的按键电路也不能使该列的电位下降，因为此时这行输出的是高电位，无法导通。所以 PE0~3 都将输入 1。
- 若按下的按键在第一行，此时按键电路导通仍能拉低对应列的电位，我们会在 PE0~3 中读到一个唯一低电平（一次按下一个按键）。此时我们知道按键的行列编号，便唯一锁定了这个按键。

具体实现时，PE0~3 需要设置为上拉输入，默认保持高电平保证稳定性。PE4~7 需要设置为推挽输出，输出稳定可靠的高（低）电平。考虑每次扫描全部行与列还是存在一定性能开销的，我们采用五次扫描的方法：第一次对应前文提到的输入检测，如果没有输入就跳过后续扫描。第二至五次才逐行扫描。

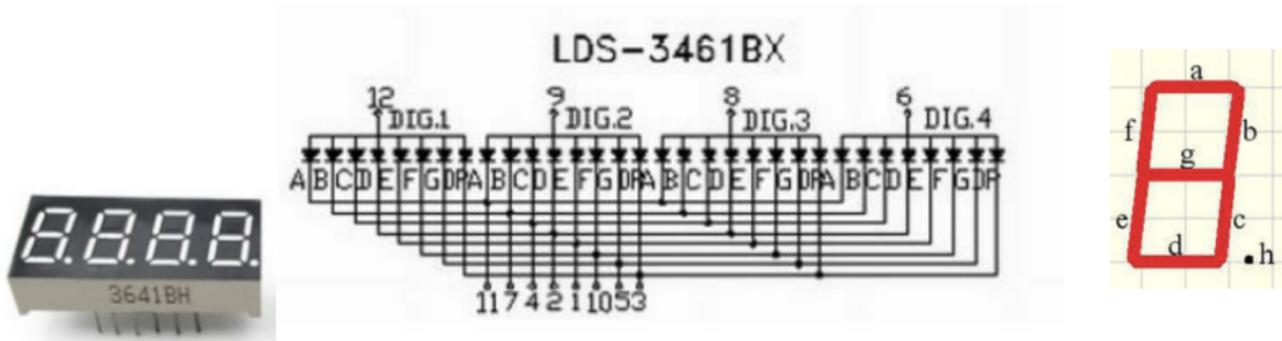
定位按键的方法也可以对电平信号编码，如左上角（SW2）对应 01111110（PE7~PE0），可以更方便我们判断。核心代码节选如下：

```
if (KEYBOARD_cnt > 0){  
    KEYBOARD_cnt--;  
    return -1;  
}  
  
GPIO_ResetBits(GPIOE, GPIO_Pin_4 | GPIO_Pin_5 |  
                GPIO_Pin_6 | GPIO_Pin_7); // 0x0000  
delay_ms(1);  
PE0_PE3_state = GPIO_ReadInputData(GPIOE)  
    & (GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3); // 读寄存器  
if (PE0_PE3_state == 0xf)  
    return -1; // 无输入  
  
KEYBOARD_cnt = 50000; // 一定时间内不再扫描  
  
GPIO_SetBits(GPIOE, GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7); // 0x1110  
delay_ms(1);  
PE0_PE3_state = GPIO_ReadInputData(GPIOE)  
    & (GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3); // 读寄存器  
if (PE0_PE3_state != 0xf) // 有输入  
    return which_key(14, PE0_PE3_state);
```

需要说明的是：代码中引入了 KEYBOARD_cnt，用于控制在一定时间内不会重复输入，避免长时间按住按键导致奇怪的现象发生。在每一次设置 PE4~

7 输出后 delay 1ms，这是因为在完成寄存器赋值到引脚实际输出电平是需要时间的，延时一会儿能确保输出稳定后再处理可以避免发生奇怪的现象，称之为“消抖”。

2.2.2. 数码管显示



实验室上的数码管电路结构如图所示。可以看到四个数位的每一个笔画都是共阴级连接。通过 6、8、9、12 输入高电平来控制某一个数位显示与否。不难发现当我们需要显示不同内容时，是不能一次点亮多个数位的，因为每个数位只要点亮，显示的内容是一定一样的。我们通过每次只点亮一个数位，根据这个数位点亮所需要的笔画。我们循环依次点亮每一个数位，只要刷新的速度够快，一个数位在一秒内就会被点亮多次，只需大于一定次数，根据人眼的视觉暂留现象便能看到四个数位全部被点亮且显示不同内容。

清楚原理编程就比较简单，我们只需要先设计好显示某个数字时需要点亮哪几个笔画，设计一个函数，可以传入数位位置与显示内容，实现只在对应位置上显示内容。我们只需要循环修改数位位置并调用函数即可。特别的，6、8、9、12 需要设置为下拉输出，默认输出低电平；1、2、3、4、5、7、10、11 需要设置为上拉输出，默认输出高电平。核心代码如下：

```
void DIGIT_display(int digit,int position){
    GPIO_SetBits(GPIOC,
        GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13); // 关闭所有数位
    GPIO_Write(GPIOB, digit_map[digit == -1 ? 16 : digit] & 0xFF); //输出段码
    GPIO_ResetBits(GPIOC, 1 << (10 + position)); // 点亮对应数位
}
```

`digit_map` 中存储的便是具体的笔画（段码）显示编码。

2.2.3. 综合实现

核心代码如下：

```
while (1){  
    tmp = KEYBOARD_Scan();  
    if (tmp != 0xff){  
        digit[3] = digit[2];  
        digit[2] = digit[1];  
        digit[1] = digit[0];  
        digit[0] = tmp;  
    }  
    if ((++cnt) > 3){  
        pos = (pos + 1) % 4;  
        cnt = 0;  
    }  
    DIGIT_display(digit[pos], pos);  
}
```

我们通过 `cnt` 计数来控制数码管数位的切换时间。

这里存在的问题是：`KEYBOARD_Scan` 函数每次都至少执行一次按键扫描（检查有无输入），效率较低。且如果有按键按下 `KEYBOARD_Scan` 函数会额外执行逐行扫描，此时通过 `cnt` 来控制时间会变得不稳定，因为每一次 `cnt++` 的间隔发生了变化，进而导致数码管显示存在明暗变化，显示效果不理想。

2.3. 中断方法的按键扫描与数码管显示

中断为我们提供了仅在特定事件发生时，才执行某段程序的方法。“特定时间”可以是某个电平变化，也可以是一定 `CPU Clock` 后。这为我们提供了新的解决方案。

针对按键扫描，原理不变，只是我们不再需要每次扫描检查有无按键被按下。我们为每一列都设置中断，只有读取到下降沿，触发中断，我们才开始逐行扫描。虽然逐行扫描效率不变，但是我们不再需要每次都判断是否需要逐行扫描。核心代码如下：

```
void EXTI0_IRQHandler(void)  
{  
    if (EXTI_GetITStatus(EXTI_Line0) != RESET)  
    {  
  
        EXTI_Keyboard_Scan(GPIO_Pin_0); //执行逐行扫描，参数为触发中断的列◆  
        EXTI_ClearITPendingBit(EXTI_Line0);  
    }  
}
```

这样做顺带解决了先前困扰我们的一个问题：因为只有下降沿才会触发中断。显示意义为：一个按键按下的全过程只有按下的那个瞬间会触发中断，即按按键这个行为全程只会触发一次按键扫描。恰好解决了之前长时间一个按键按下导致的问题。

数码管显示我们通过计时器中断来控制数位显示切换的间隔。计时器中断具体实现为：我们设置一个计数值，CPU 内部每拍会给这个计数值减 1，当递减至 0 时触发计时器中断，同时重置这个计数值，循环往复。结合 CPU 频率与计数值大小，我们可以控制数码管的刷新频率。核心代码如下：

```
void digit_display_switch(void)
{
    digit_pos = (digit_pos + 1) % 4;
    DIGIT_display(digit[digit_pos], digit_pos);
}

void SysTick_Handler(void) // 中断函数
{
    digit_display_switch();
}

#define digit_display_fps 60

void systick_init(void) // 初始化
{
    RCC_ClocksTypeDef RCC_Clocks;
    RCC_GetClocksFreq(&RCC_Clocks); // 获取当前时钟频率
    SysTick_Config(RCC_Clocks.HCLK_Frequency / digit_display_fps / 4);
    // 控制的是单个数码管的频率，所以还要除以 4
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);
    // 选择计时器时钟源为 HCLK
}
```

修改后的主函数核心部分为：

```
while (1)
{
    if (input_key_flag){
        digit[3] = digit[2];
        digit[2] = digit[1];
        digit[1] = digit[0];
        digit[0] = input_key;
        input_key_flag = 0;
    }
}
```

可以看到此时主函数部分已经变得非常简洁，实际的运行的效率也大大提高，因为我们做到了“只在需要时执行必要的事”。

3. 实验结果

启动调试模式，并让代码运行一段时间，并暂停。

我们观察 GPIOE 的 CRL 寄存器，结果如下：

Property	Value
CRL	0x33338888
CRH	0x44443333
IDR	0x0000CF0F
ODR	0x00000F0F
BSRR	0
BRR	0
LCKR	0

根据编码规则，一个引脚占用四位（对应图中一个十六进制位），分为 CNF[1:0] MODE[1:0]。通过 CNF 与 MODE 的编码组合设定引脚。对 CRL 寄存器解读如下：

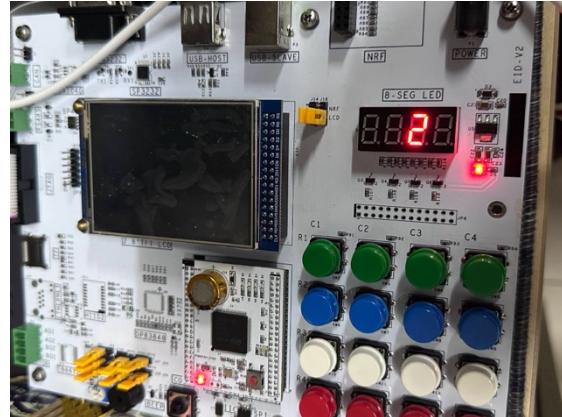
0~3 位的 8 对应 CNF=10, MODE=00, 查阅手册知表示上拉（或下拉）输入。此时要知道具体是上拉还是下拉需要查看 ODR 寄存器。此时 ODR 寄存器低 4 位为全 1（十六进制下为 F）说明是上拉模式。对应引脚为 PE0~3。

4~7 位的 3 对应 CNF=00, MODE=11, 查阅手册知表示 50MHz 推挽输出。对应引脚为 PE4~7。

结合前文我们知道 PE0~7 端口是用于控制矩阵键盘的，CRL 与 ODR 寄存器的值符合预期。

我们再将主函数中 digit_pos 与 digit 数组加入 watch, watch 列表与板卡显示如下图。数码管是扫描显示的，暂停时应当只显示一个数字。watch 表中 digit_pos 为 2，说明显示编号为 2 的数码管，即从左往右第三个数码管。digit[2]的值为 2，说明应该显示 2。观察板卡显示，符合我们的预期。

Name	Value	Type
digit_pos	0x00000002	int
digit	0x2000001...	int[4]
[0]	0x00000000	int
[1]	0x00000001	int
[2]	0x00000002	int
[3]	0x00000003	int
<Enter expression>		



4. 心得与体会

通过本次实验，我学习到了如何实现逐行扫描按键输入与数码管扫描显示。在具体编程过程中，我使用了寄存器编译与库函数编程两种不同方式，并体会到他们的差异，体会到了他们的优点与缺点，掌握在具体情况下如何选择。

更进一步，我从手动判断按键输入与数码管扫描间隔控制，进化到了使用中断来实现这两个功能。我体会到了中断在异步输入与计时上的妙用。

在编程中我遇到了各种问题，例如按键抖动、中断不触发等等。我通过查阅资料、分步调试定位并解决了这些问题。在这次实践中我的嵌入式编程能力得到了进一步提升。

5. 源代码

见附件压缩包。本实践项目同时开源于：

<https://github.com/Justin-Nickel-Wu/Embedded-Systems-Course-Project>