上海交通大學

《工程实践与科技创新Ⅱ-A》课程

学生实验报告

实验名称：ARM 实验一 时钟选择与 GPIO 实验

姓　　名：彭俊杰

学　　号：521021910404

班　　级：ICE0501-5

任课老师：朱兰娟、邵群

2023 年 5 月 14 日

目录

1 实验目的

（1）熟练掌握 ARM 的集成开发环境 KEIL uVision5，能够自行建立一个实验用工程项目。
（2）理解 CPU 的时钟信号，了解不同时钟对电源消耗的不同。
（3）掌握 GPIO 的工作原理，能够结合 GPIO 的输入与输出功能进行实验。

2 实验原理

2.1 硬件原理分析

2.1.1 系统控制之时钟控制（clock control）

TM4C1294NCPDT MCU 中有多个时钟源可以作为系统时钟使用，包括：高精度内部振荡器（PIOSC），主振荡器（MOSC），低精度内部振荡器（LFIOSC）休眠模块 RTC 振荡器时钟源（RTCOSC）。系统时钟(SysClk)可以来源于四个时钟源中的任意一个，或者由 PLL（Phase Locked Loop，锁相环）产生。

最终我们使用高精度内部振荡器（PIOSC），频率 16MHz。

2.1.2 通用输入/输入端口 GPIO

TM4C1294 微控制器有 15 个物理 GPIO（General Purpose Input/Output）模块，即端口 A、B、C、D、E、F、G、H、J、K、L、M、N、P、Q。每个端口最多 8 个引脚，共 90 个 GPIO 引脚，可以独立配置。下表是 GPIO 可配置的工作模式。

GPIO	TM4C129x Series	
General Purpose Input（数字输入）	Floating（浮空）	
	Pull-Up（上拉）	(GPIO_PIN_TYPE_STD_WPU)
	Pull-Down（下拉）	(GPIO_PIN_TYPE_STD_WPD)
	WAKE（弱电阻）	(GPIO_PIN_TYPE_WAKE_HIGH) (GPIO_PIN_TYPE_WAKE_LOW)
General Purpose Output（数字输出）	Push-Pull（推挽）	(GPIO_PIN_TYPE_STD)
	Open-Drain（开漏）	(GPIO_PIN_TYPE_OD)
	Push-Pull+Pull-Up	(GPIO_PIN_TYPE_STD_WPU)
	Push-Pull+Pull-Down	(GPIO_PIN_TYPE_STD_WPD)
Analog（模拟输入）	Analog	(GPIO_PIN_TYPE_ANALOG)
Alternate Function Output（复用功能）	Push-Pull	(GPIO_PIN_TYPE_STD)
	Open-Drain	(GPIO_PIN_TYPE_OD)
	Push-Pull+Pull-Up	(GPIO_PIN_TYPE_STD_WPU)
	Push-Pull+Pull-Down	(GPIO_PIN_TYPE_STD_WPD)

图 1 GPIO 可配置的工作模式

GPIO A-H,J 端口可经由 AHB 和 APB 两种总线访问，K-N,P,Q 口接 AHB。其中，AHB (Advanced High-Performance Bus) 能提供更好的访问性能；APB (Advanced Peripheral Bus)是外围总线。

GPIO 的工作模式主要有以下几种：

输入模式（Input Mode）：在这种模式下，GPIO 引脚可用作数字输入，接收其他设备（如按钮、开关或传感器）的信号。电路原理是，输入设备改变电平状态（高电平或低电平），GPIO 引脚读取并识别这种电平变化。

输出模式（Output Mode）：在这种模式下，GPIO 引脚可以产生数字输出，驱动其他设备（如 LED 灯、继电器等）。电路原理是，GPIO 引脚根据程序控制输出高电平或低电平，以驱动外部设备工作。

PWM（Pulse Width Modulation）模式：这是一种特殊的输出模式，通过控制脉冲的宽度（即脉冲在高电平状态持续的时间）来模拟模拟信号。这种模式常用于控制设备的功率输出，如调整 LED 的亮度或控制电机的转速。

中断模式（Interrupt Mode）：在这种模式下，GPIO 引脚可以被配置为在输入信号发生变化时产生中断，从而立即响应外部事件。这对于需要实时响应的系统非常有用。

除此之外，还有一些 GPIO 可以工作的特殊模式，如 I2C、SPI、UART 等，这些都是串行通信协议，可以让 GPIO 引脚实现更复杂的数据交换。

对于不同的工作模式，电路原理主要是通过内部的电子开关和电压比较器来实现的。例如，在输入模式下，GPIO 引脚通过内部的电压比较器来识别输入电平的高低；在输出模式下，GPIO 引脚通过内部的电子开关来控制输出电平的高低；在 PWM 模式下，通过内部的定时器和电子开关来控制脉冲的宽度；在中断模式下，通过电压比较器和中断控制器来实现对输入电平变化的实时响应。

2.1.3 按键消抖

机械特性决定按键存在抖动，正常情况按键持续时间 50ms~200ms，抖动时间 10ms 以内。

我们可通过相邻两次读取按键状态，前高后低为按下，前低后高为弹起。要求 T2<T1<T3（T1-读取间隔时间，T2-抖动时间，T3-按键持续时间）

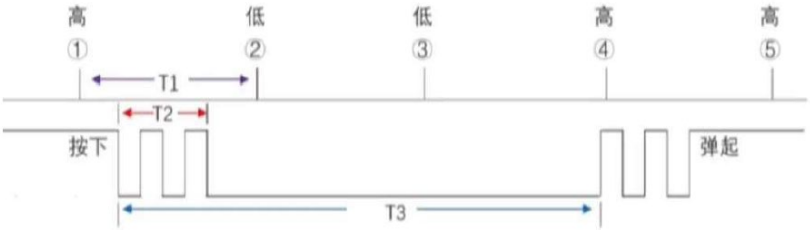

图 2 按键消抖流程示意图

如下所示，隔 20ms 相邻两次读取按键状态，判断按键按下后即可进行后续操作。

```
1.   last_key_value = key_value;          //记录上一次按键值
2.          SysCtlDelay( ui32SysClock / 150);    //过 20ms 后再次读取按键状
    态，实现按键消抖
3.
4.          key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0);     //读取
    此时按键值
5.          delay_time = SLOWFLASHTIME;
6.          if (key_value == 0 && last_key_value == 1)  //判断按键按下
7.  ……
```

2.1.4 S800 实验板上部分管脚功能以及分析

■ 红板

名称	对应管脚	说明
RESET	RESET	TM4C1294NCPDT芯片复位按键，低有效
WAKE	WAKE	从睡眠模式唤醒按键，低有效
USR_SW1	PJ0	用户输入按键，低有效
USR_SW2	PJ1	用户输入按键，低有效
D0		3.3V电源指示，绿LED，高有效
D1	PN1	用户控制绿LED，高有效
D2	PN0	用户控制绿LED，高有效
D3	PF4	用户控制绿LED，高有效
D4	PF0	用户控制绿LED，高有效

■ 蓝板

名称	对应管脚	说明
SW1-SW8	TCA6424-P01~P08	TCA6424 I2C展GPIO芯片P0口，低有效
LED1-LED8	PCA9557-P0-P7	PCA9557 I2C展GPIO芯片P0口，低有效
LED_M0	PF0	用户控制LED，高有效
LED_M1	PF1	用户控制LED，高有效
LED_M2	PF2	用户控制LED，高有效
LED_M3	PF3	用户控制LED，高有效
D10		3.3V电源指示，红LED，高有效

图 3　S800 实验板上部分管脚功能

部分管脚分析：

红板上的按键 SW1、SW2 对应 PJ0、PJ1，低电平有效，设置为输入。这两个引脚没有外接上拉电阻，必须配置为内部弱上拉，才能清楚地区分未按下与按下状态。

红板上 LED3 和 4 对应 PF4 和 PF0，LED1 和 2 对应 PN0 和 1，设置为输出，高电平时点亮，低电平时熄灭。

2.1.5 步进电机工作原理

步进电机是一种电气驱动设备，它将电脉冲信号转化为角位移或线位移。步进电机的工作原理基于磁场和电流的相互作用。下面是步进电机的基本工作原理：

步进电机主要由定子和转子两部分组成。定子具有多个极，通常是电磁铁，而转子通常是永磁材料。

控制器会按照预定的顺序和频率给步进电机的定子线圈供电，这会在定子上产生磁场。这些磁场会使转子转动到一个位置，使其与定子磁场对齐。这个位置就是一个"步进"。

控制器接下来会改变定子线圈的电流，改变磁场的方向。这会使转子转动到下一个位置，对齐新的磁场。这是下一个"步进"。

通过重复这个过程，控制器可以精确地控制转子的位置和速度。每个"步进"对应的角位移是固定的，所以通过控制步进的数量和速度，可以精确地控制转子的旋转。

步进电机的主要优点是它可以提供精确的定位和速度控制，而不需要复杂的反馈控制系统。这使得步进电机在很多需要精确控制的应用中，如打印机、CNC 机床和硬盘驱动器等，都有广泛的应用。

2.2 软件算法分析（含程序流程图和源代码）

相关程序的流程图如下：

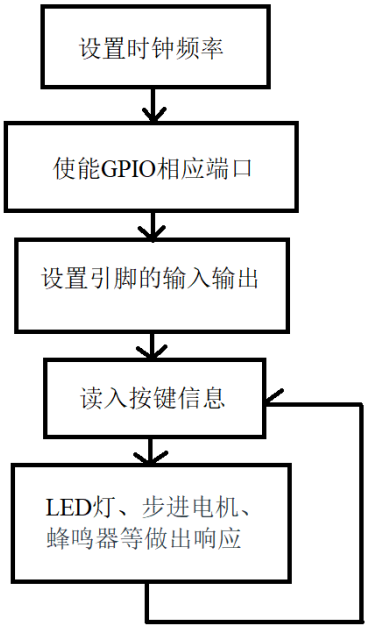

图 3 软件算法分析程序流程图

2.2.1 设置时钟频率

```
1.  void S800_Clock_Init(void)
2.  {
3.      //use internal 16M oscillator, HSI
4.    ui32SysClock = SysCtlClockFreqSet((SYSCTL_OSC_INT |SYSCTL_USE_OSC),
     16000000);
5.
6.      //use extern 25M crystal
7.      //ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |SYSCTL_OS
    C_MAIN |SYSCTL_USE_OSC), 25000000);
8.
9.      //use PLL
10.     //ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |SYSCTL_OS
    C_MAIN |SYSCTL_USE_PLL |SYSCTL_CFG_VCO_480), 60000000);
11. }
```

2.2.2 使能 GPIO 相应端口

SysCtlPeripheralEnable()实现。

```
1.   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    //Enable PortF
```

2.2.3 设置引脚的输入输出

GPIOPinTypeGPIOIntput()设置输入

```
1.  GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0);
    //Set PJ0 as input pin
```

GPIOPinTypeGPIOOutput()设置输出

```
2.  GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0);
      //Set PF0 as Output pin
```

2.2.4 读入按键信息

```
1.  key_value_PJ0 = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0) ;
       //read the PJ0 key value
2.  key_value_PJ1 = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_1) ;
       //read the PJ1 key value
```

2.2.5 LED 灯、步进电机、蜂鸣器等做出响应

以 LED 灯亮灭为例：

```
1.  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
        // Turn on the LED_M0.
2.  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
       // Turn off the LED_M0.
```

3 实验内容和步骤

3.1 实验要求 2

3.1.1 任务要求：

当按下 USR_SW1 键时，点亮 LED_M0，放开时，熄灭 LED_M0
当按下 USR_SW2 键时，点亮 LED_M1，放开时，熄灭 LED_M1

3.1.2 任务实现

在示例程序 exp1-1 的基础上，修改 main 函数中的 while 循环.在 while 循环中，分别读取两个按键的电平信息，分别存入 key_value_PJ0、key_value_PJ1 内。之后分别根据两者的值对相对应的 led 发出高低电平命令。采用 if-else 语句实现分类判断。由于 while 循环中语句简单，执行速度很快，所以可以认为是同时判断了两按键的电平高低并输出了指令信号。

代码如下：

```
3.  while(1)
4.    {
5.          key_value_PJ0 = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0) ;
               //read the PJ0 key value
6.          key_value_PJ1 = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_1) ;
               //read the PJ1 key value
7.
8.          if (key_value_PJ0   == 0)             //USR_SW1-PJ0 pressed
9.              GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
           // Turn on the LED_M0.
10.         else
11.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
                   // Turn off the LED_M0.
12.         if (key_value_PJ1   == 0)           //USR_SW1-PJ1 pressed
13.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
           // Turn on the LED_M1.
14.         else
15.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
           // Turn off the LED_M1.
16.   }
```

同时，在函数 S800_GPIO_Init(void)中进行相应修改。9、11 行添加了对 PF1 的设置。14、16 行添加了对 PJ1 的设置。除了端口有变化外其他设置与 PF0 和 PJ0 相同。新增加一行 17 行，用来设置 PJ1 按键属性。

代码如下：

```
3.  void S800_GPIO_Init(void)
4.  {
5.      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
       //Enable PortF
6.      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF));
     //Wait for the GPIO moduleF ready
7.
8.      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
       //Enable PortJ
9.      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOJ));
     //Wait for the GPIO moduleJ ready
10.
11.     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0);
     //Set PF0 as Output pin
12.
13.     GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0);
     //Set PJ0 as input pin
14.     GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
    GPIO_PIN_TYPE_STD_WPU);
```

```
15.
16.     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);
    //Set PF1 as Output pin
17.
18.     GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_1);
    //Set PJ1 as input pin
19.     GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_1, GPIO_STRENGTH_2MA,
  GPIO_PIN_TYPE_STD_WPU);
20. }
```

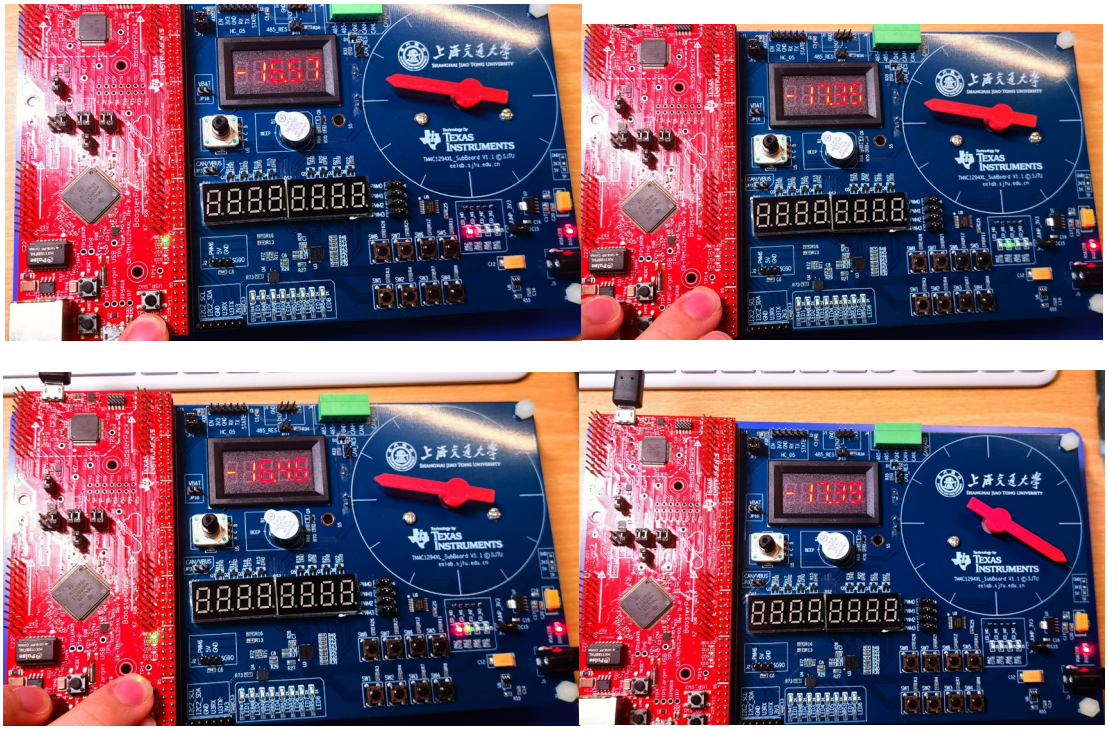

图 4 实验任务 2 效果图

实验任务 2 效果图如上所示，分别对应按下 USR_SW1 键、按下 USR_SW2 键、同时按下 USR_SW1 和 USR_SW2 键、放开四种情况。

以下实验涉及 LED 灯闪烁、蜂鸣器鸣叫等，故不在报告中体现。

3.2 实验要求 3

3.2.1 任务要求

当第一次短按 USR_SW1 键时，闪烁 LED_M0
当第二次短按 USR_SW1 键时，熄灭 LED_M0
当第三次短按 USR_SW1 键时，闪烁 LED_M1
当第四次短按 USR_SW1 键时，熄灭 LED_M1

3.2.2 任务实现

先设置一个全局变量 state，记录按键次数。在 while 循环中，先记录上一次按键状态，过 20ms 后再次读取按键状态，实现按键消抖。

若按键按下则为低电平，state 加 1。当 state 为 5 时回到 1，实现同余 4 的效果。之后分四类情况讨论。50 行和 64 行分别设置了 delay，目的是创造延时以避免因为 while 循环一次执行太快导致一次按键经历了多次 while 循环，从而产生的多次读取问题。

代码如下：

```
8.  while(1)
9.      {
10.         last_key_value = key_value;          //记录上一次按键值
11.         SysCtlDelay( ui32SysClock / 150);    //过 20ms 后再次读取按键状
    态，实现按键消抖
12.
13.         key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0);     //读取
    此时按键值
14.         delay_time = SLOWFLASHTIME;
15.         if (key_value == 0 && last_key_value == 1)  //当判断按键按下后
    state 状态更新
16.         {
17.             state = (state+1)%4;         //state 记录按键次数
18.         }
19.
20.         if (state%2 == 0)        //state = 0 or 2 熄灭两灯
21.         {
22.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
23.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
24.         }
25.         else if (state == 1)     //state = 1，闪烁 LED_M0
26.         {
27.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
28.             Delay(delay_time);
29.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
30.             Delay(delay_time);
31.         }
32.         else if (state == 3)     //state = 3，闪烁 LED_M1
33.         {
34.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
35.             Delay(delay_time);
36.             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
37.             Delay(delay_time);
38.         }
39.     }
```

程序的缺点在于，若程序恰好在 delay 中，按下按键将无响应。经过尝试，这个问题可以通过改写 delay 函数获得部分改善，即 delay 循环到一定次数后开始检测按键的电平，若低电平则用 break 跳出 delay。当然，由于 delay 的 for 循环内加入了语句，使循环一次的时间变长，因此要适当调小输入 delay 的数值来获得相近的延时时间。但该法不能彻底解决问题。该问题的完全解决需要用到后面实验的中断。

3.3 实验要求 4

3.3.1 任务要求

采用双四拍或八拍方式，依次给 PF0~PF3（或者 PF3~PF0）对应的各个相输入高电平信号，使红色指针顺时针一圈，暂停 3 秒，逆时针一圈，暂停 3 秒，循环往复。每转完一圈蜂鸣器发声。

3.3.2 任务实现

这里采用了双四拍的方法，顺时针和逆时针是通过双四拍顺序的颠倒实现的，其中，PF3~0 顺时针转，PF0~3 逆时针转。已经知道电机需要 512 个循环周期转动一周，因此在 while 循环中设立两个 for 循环，每个 for 循环执行 512 次，代表完成逆时针或者顺时针完成一圈。然后执行 S800_Beep()函数使蜂鸣器鸣叫，再通过 SysCtlDelay(ui32SysClock)延时 3 秒。顺逆时针的延时时长均为 5ms，可减小该值，加快时针转速。

代码如下：

```
1.  while(1)
2.    {
3.              int i;
4.              for (i=0; i<512; i++){ //顺时针
5.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
6.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
7.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
8.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0x0);
9.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
10.
11.                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
12.                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
13.                SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
14.                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x0);
15.                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0x0);
16.
```

```
17.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
18.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
19.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
20.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
21.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x0);
22.
23.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
24.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
25.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
26.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
27.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
28.             }
29.             S800_Beep();
30.             SysCtlDelay(ui32SysClock);
31.
32.
33.             for (i=0; i<512; i++){   //逆时针
34.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
35.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
36.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
37.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
38.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
39.
40.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
41.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
42.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
43.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
44.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x0);
45.
46.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
47.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
48.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
49.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x0);
50.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0x0);
```

```
51.
52.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
53.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
54.                 SysCtlDelay(5*ui32SysClock/3000); //延时 5ms
55.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0x0);
56.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
57.             }
58.             S800_Beep();
59.             SysCtlDelay(3000*ui32SysClock/3000);
60.
61.
62.     }
```

蜂鸣器设置如下：

```
1.  void S800_Beep(void)
2.  {
3.      int k;
4.      for (k=0; k<600;k++) {
5.          GPIOPinWrite(GPIO_PORTK_BASE,GPIO_PIN_5, ~GPIOPinRead(GPIO_PO
    RTK_BASE, GPIO_PIN_5));
6.          SysCtlDelay(ui32SysClock/(1048*3));
7.      }
8.  }
```

步进电机运行、暂停效果图如下，左图处在运行中，右图处在暂停状态。

图 5 步进电机运行、暂停效果图

3.4 实验要求 5

3.4.1 任务要求

使用 GPIO 中断方式实现实验要求 3

3.4.2 任务实现

主函数部分与实验要求 3 大同小异。区别仅仅在于用 switch 替换了 if。按键次数的增加依赖于中断。其中 `IntMasterEnable()`实现开中断。

代码如下：

```
1.  int main(void)
2.  {
3.      S800_Clock_Init();
4.      S800_GPIO_Init();
5.
6.      IntMasterEnable();  //开中断
7.
8.      while (1)
9.      {
10.         switch(btn_cnt){
11.             case 1:
12.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0)
    ;
13.                 SysCtlDelay(ui32SysClock/10);
14.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
15.                 SysCtlDelay(ui32SysClock/10);
16.                 break;
17.             case 2:
18.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
19.                 break;
20.             case 3:
21.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1)
    ;
22.                 SysCtlDelay(ui32SysClock/10);
23.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
24.                 SysCtlDelay(ui32SysClock/10);
25.                 break;
26.             case 4:
27.                 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
28.                 break;
29.         }
30.     }
31. }
```

中断方面。首先，需要在主循环前对内核级进行使能操作，即 `IntMasterEnable()`。之后，需要在常规的端口使能函数后添加对按键的中断类型和触发方式设置和使能函数，即下述代码 16-18 行。这里是对源级和 NVIC 级进行使能操作。

然后是重写中断服务程序。首先读取中断状态确定中断源，接着清除中断请求。最后对按键次数进行操作。

按下按键有时会发生抖动，这样会导致一次按键多次中断。虽然可以通过“短按”避免大部分抖动的出现，但程序对按键也进行了去抖操作。在获取中断信号之前，如下述代码 25、26 行所示，利用 IntDisable(INT_GPIOJ)和 SysCtlDelay(100*ui32SysClock/3000)，实现延时消抖。

代码如下：

```
1.  void S800_GPIO_Init(void)
2.  {
3.      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
       //Enable PortF
4.      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF));
        //Wait for the GPIO moduleF ready
5.
6.      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
       //Enable PortJ
7.      while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOJ));
       //Wait for the GPIO moduleJ ready
8.
9.    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0 | GPIO_PIN_1);
       //Set PF0,PF1 as Output pin
10.
11.     GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0);
        //Set the PJ0 as input pin
12.     GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
    GPIO_PIN_TYPE_STD_WPU);
13.
14.     //add more codes...
15.
16.     GPIOIntTypeSet(GPIO_PORTJ_BASE, GPIO_PIN_0, GPIO_FALLING_EDGE);
17.     GPIOIntEnable(GPIO_PORTJ_BASE, GPIO_PIN_0);
18.     IntEnable(INT_GPIOJ);
19.
20. }
21.
22. void GPIOJ_Handler(void)    //PortJ 中断处理
23. {
24.     unsigned long intStatus;
25.     IntDisable(INT_GPIOJ);
26.     SysCtlDelay(100*ui32SysClock/3000);
27.
28.     intStatus = GPIOIntStatus(GPIO_PORTJ_BASE, true); //获取中断信号
29.     GPIOIntClear(GPIO_PORTJ_BASE, intStatus );  //清除中断请求信号
30.
31.     if (intStatus & GPIO_PIN_0) {   //PJ0?
```

```
32.             btn_cnt = btn_cnt % 4 + 1;
33.     }
34.
35.     IntEnable(INT_GPIOJ);
36. }
```

3.5 实验要求 6

3.5.1 任务要求

使用 SysTick 定时器来控制 LED_M0 和 LED_M1 灯的亮灭和闪烁：
当第一次短按 USR_SW1 键时，LED_M0 以 1s 的周期闪烁
当第二次短按 USR_SW1 键时，熄灭 LED_M0
当第三次短按 USR_SW1 键时，LED_M1 以 2s 的周期闪烁
当第四次短按 USR_SW1 键时，熄灭 LED_M1

3.5.2 任务实现

先看 systick 的初始化和中断服务程序的重写。初始化中，先设置好定时周期为 0.5 秒，这样一亮一灭周期恰好是 1 秒，然后使能中断。在中断服务程序中，可以看到每隔 0.5 秒系统就会输出 systick_500ms_status 置 1 的命令。接下来使用“软件定时”，利用局部静态变量 k 实现“分频”，将两个 0.5 秒计时周期叠加形成一个 1 秒计时，每隔 1 秒输出 systick_1s_status 置 1 的命令，这样就可以实现 LED_M1 的 2 秒亮暗周期。

代码如下：

```
1.  void S800_SysTick_Init(void)
2.  {
3.      SysTickPeriodSet(ui32SysClock/2); //延时 0.5s
4.      SysTickEnable();
5.      SysTickIntEnable();
6.  }
7.
8.  /*
9.      Corresponding to the startup_TM4C129.s vector table systick inter
    rupt program name
10. */
11. void SysTick_Handler(void)
12. {
13.     static uint8_t k = 0;
14.
15.     systick_500ms_status = 1;
16.
17.     if (k) systick_1s_status = 1;
18.
```

```
    k = 1 - k;

}
```

在 while 循环前照例写上内核级中断初始化。GPIOJ 中断服务程序与实验要求 5 相同。按键数为 2 和 4 时写低电平是必然。当按键数是 1 时，结合 systick 中断服务程序分析时序可知，程序进入 case1，切换 led 的亮暗状态，之后迅速清除 systick_500ms_status，之后的 while 循环虽然 btn_cnt 为 1，但由于 systick_500ms_status 是 0，while 循环将不做任何操作，直到 systick 在一个定时周期（0.5s）后重新把 systick_500ms_status 置 1，while 循环才会重新执行 case1 语句。btn_cnt 为 3 时同理。由此可见，程序巧妙地实现了每隔一个周期出现切换 led 亮暗状态的信号。

代码如下：

```
int main(void)
{
    IntMasterDisable(); //关中断

    S800_Clock_Init();
    S800_GPIO_Init();
    S800_SysTick_Init();

    IntMasterEnable();  //开中断

    while (1)
    {
        switch(btn_cnt){
            case 1:
                if (systick_500ms_status)
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, ~GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_0));
                systick_500ms_status = 0;
                break;
            case 2:
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);
                break;
            case 3:
                if (systick_1s_status)
                    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, ~GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_1));
                systick_1s_status = 0;
                break;
            case 4:
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x0);
                break;
        }
```

```
31.     }
32.
33. }
```

程序的一个缺点是，第一次灯亮时常会短于 0.5 秒，之后一切正常，原因如下。在从 btn_cnt 为 4 回到 1，以及从 btn_cnt 为 2 到 3 时存在衔接问题。以 btn_cnt 是 4 时为例，由于与 btn_cnt 为 1 之间已经隔了三次按键的时差，systick_500ms_status 一般已经是 1，这时按下按键将进入第一个 case，灯亮并将 systick_500ms_status 置 0。然而，由于 systick 一直在后台运行，此时 systick 计时往往还剩不足 0.5 秒，导致不足 0.5 秒后 systick_500ms_status 又将变 1 导致灯灭，从而第一次灯亮时常会短于 0.5 秒。当然，之后一切正常。

为了弥补该缺陷，笔者尝试在 btn_cnt 为 2 和 4 时添加 SysTickDisable()取消 systick 使能，在 1 和 3 时添加 SysTickEnable()为 systick 使能，结果有了改善，但是实验现象偶尔有不稳定，原因有待研究。由于这项改进尚不成熟，故暂时没有采用该代码。

4. 讨论题解答

（1）人为修改内部时钟或外部时钟，如将内部时钟改为 8M，或将外部时钟改为 30M，会有什么结果？

答：内部时钟改为 8M，程序运行会变慢；外部时钟改为 30M，程序运行会变快。

（2）在使用 PIOSC 及 MOSC 时，能否生成非晶振频率？如 1M 或 10M？

答：可以利用分频生成。

（3） 能否将 PLL 时钟调整到外部时钟的频率以下？如将 25M 外部时钟用 PLL 后调整为 20M？

答：单纯地只使用 PLL 不能，只能配置为 320M 或 480M，但可以借助系统分频器，使用 PLL 分频降低频率。480M/20M =24 不是 2 的幂次，无法被分频。

（4）在使用 PLL 时，系统频率最小值及最大值分别为多少？

答：最小为 320M/1024=0.3125M；系统最大频率 120M。

（5）将 PLL 后的时钟调整为最大值 120M，LED 闪烁会有什么变化？为什么？

答：闪烁明显变快，若适当调小 delay 的循环次数，则肉眼难以分清闪烁，似乎在持续发光。

（6）分析语句 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0)此函数中，每个函数项的意义。如果将第三个参数项改为 1 或 2，分别会有什么现象？

答：意义分别是端口基地址、引脚屏蔽位、写入的数据。改为 1 时，正常写入，灯发光；改

为 2 时，写入被屏蔽，灯不发光。

（7）结合硬件说明 GPIOPadConfigSet 语句行的作用。如果此行注释，在 WATCH 窗口中观察 key_value 值会有什么变化。

答：配置引脚的驱动强度和类型。设置驱动强度和引脚类型。驱动强度是指电流强度，有些外设需要较强电流才能驱动。引脚类型如弱上拉，与电路设计有关。注释后由于缺少配置，按键输入无效，key_value 始终为 0。

5. 感想和建议

熟练掌握和使用集成开发环境是进行嵌入式系统开发的基础。在实验中，我通过自行建立一个实验用工程项目，学会了如何在 KEIL uVision5 中创建工程、添加源文件和头文件、进行编译和下载等操作。由此我迈出了嵌入式系统开发的第一步。

通过实验，我在实验中对 GPIO 的工作原理进行了深入的了解，并通过实验对 GPIO 的输入和输出功能进行了验证，理解了 TM4C1294NCPDT MCU 的时钟信号。

实验要求 3 中，我发现自己的现有方案存在局限性，但实验要求 3 的问题在实验要求 5 中的中断得以解决，让我直观地、循序渐进地掌握嵌入式开发的知识，收获很大。

最后，感谢《工程实践与科技创新Ⅱ-A》课程组准备的高质量课程和嵌入式开发条件，感谢朱兰娟老师为本门课程的精心准备与悉心讲解，深入浅出的为我们讲解嵌入式开发的诸多知识，将复杂的嵌入式知识网络按照对应模块分割开来，逐一细解，帮助我们拓宽视野，增长见识。感谢所有为我答疑解惑，帮助过我的老师、助教和同学。