嵌入式计算机系统与实验 | 作业 6

Shuiyuan@Noroshi

1 GIPO

1.1 理论知识

通用输入输出接口(General Purpose Input Output)是 MSP430 最主要的 I/O 接口，总共有 9 个，除了 P7 有 6 个管脚以外，其余的都有 8 个管脚，即总共 70 个管脚. 且其中的 P1~P4 支持外部中断，即共有 32 个外部中断管脚.

其中，每个接口的状态，功能等都由对应的寄存器给出，这里我们需要用到的包括:

- 输入寄存器(PxIN):

  这部分寄存器反应了对应接口所接收到的信号，每位 0 代表接收信号为低电平，1 代表接收信号为高电平，这部分寄存器是只读的

- 输出寄存器(PxOUT):

  输出芯片传递的信号，0 代表低电平 1 代表高电平. MSP 的接口还支持配置上拉/下拉电阻使能(即默认为高电平/低电平)，防止引脚悬空

- 方向寄存器(PxDIR):

  确定接口的实际接收方向，0 为 input，1 为 output

- 上拉/下拉使能(PxREN):

  (仅在输入状态下有效)确认是否启用上拉/下拉电阻，如果启用，会根据 PxOUT 的值判断上拉/下拉状态

1.2 利用按钮进行 LED 的控制

在我们使用的 MSP430F6638 实验箱中，右下角 S1~S5 按键分别于与 P4.0~P4.4 连接，右下角 LED1 LED5 指示灯分别于 P4.5，P4.6，P4.7，P5.7，P8.0 连接

我们的目标是，通过按钮来控制 led 灯的开关与数量，当我们按下 s3 按钮时，每多按一次 led 多亮一盏，如果按下 s4 按钮，则 led 灯全部熄灭，重新开始流程. 根据这个我们可以初步写出一版代码如下

```
#include <msp430f6638.h>

void update_leds(int count)
{
  if (count > 5)
  {
    count = 5;
  }
  switch (count)
  {
  case 0:
    P4OUT &= ~0x00e0;
    P5OUT &= ~BIT7;
```

```
    P8OUT &= ~BIT0;
    break;
  case 1:
    P4OUT |= BIT5;
    P5OUT &= ~BIT7;
    P8OUT &= ~BIT0;
    break;
  case 2:
    P4OUT |= 0x0060;
    P5OUT &= ~BIT7;
    P8OUT &= ~BIT0;
    break;
  case 3:
    P4OUT |= 0x00e0;
    P5OUT &= ~BIT7;
    P8OUT &= ~BIT0;
    break;
  case 4:
    P4OUT |= 0x00e0;
    P5OUT |= BIT7;
    P8OUT &= ~BIT0;
    break;
  case 5:
    P4OUT |= 0x00e0;
    P5OUT |= BIT7;
    P8OUT |= BIT0;
    break;
  }
}

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;

  P4DIR |= 0x00e0;
  P5DIR |= BIT7;
  P8DIR |= BIT0;

  P4OUT |= 0x001f;
  P4REN |= 0x001f;
  int count = 0;
  update_leds(count);
  while (1)
  {
  if (count > 5) {count = 5;}
    if (! (P4IN & BIT4))
    {
      count++;
      update_leds(count);
      flag = 0;
    }
    if (! (P4IN & BIT3))
    {
      count = 0;
      update_leds(count);
      flag = 1;
    }
  }
}
```

我们从 main 函数的初始化部分开始看，首先关闭看门狗计时器(WDTCTL = WDTPW + WDTHOLD)，防止程序运行中因为看门狗复位. 随后我们配置了 led 端口的方向，P4DIR 的 0x00e0 部分被设置为输出(即 0x11100000，也就是 P4.5~P4.7 为输出)，用于控制 led 灯. P5DIR 设置 P5.7 为输出. P8DIR 设置 P8.0 为输出. 之后为 P4 端口的低 5 位（0x001f）设置上拉/下拉电阻（P4REN |= 0x001f)，并通过 P4OUT |= 0x001f 激活内部上拉电阻，即在这里，所有按钮连接的端口都是输入状态，同时，在默认状态下为上拉电压(即按钮没有按下时这部分为高电平).

然后进入主要的程序循环，P4IN 与 BIT3 或者 BIT4 进行按位与运算，即当对应的按钮被按下时，按位与的结果为 0，再加上前面的非，也就是在对应的按钮被按下时，就会进入条件语句. 此处第一个即是会将灯光点亮数++，第二个时清零计数，即让所有的灯都熄灭，然后调用 update_led()这个点灯辅助函数. 这个函数的作用就是点亮对应数量的 led 灯.

我们可以看看效果

Figure 1: 电灯效果，完整可见视频(1_1)

但这里我们可以发现一个很严重的问题，就是因为防抖，或者因为在上面停留过长，都会导致本来只按了一次的，但实际上计数器已经加了无数次，就会全部亮起(我保证这不是程序本身的问题，程序本身设计的时是逐步点亮的). 这是因为运行频率很高，按钮按下去的时间可能已经让循环执行了无数次了，触发了条件判定(在调试时我计数发现已经溢出成负数了)，所以，还得改.

这里我们首先最简单的加入一个延迟的方法，也就是

```
if (! (P4IN & BIT4))
{
  count++;
  update_leds(count);
   __delay_cycles(1000000);
}
```

这样可以有效避免抖动和重复判定的问题.

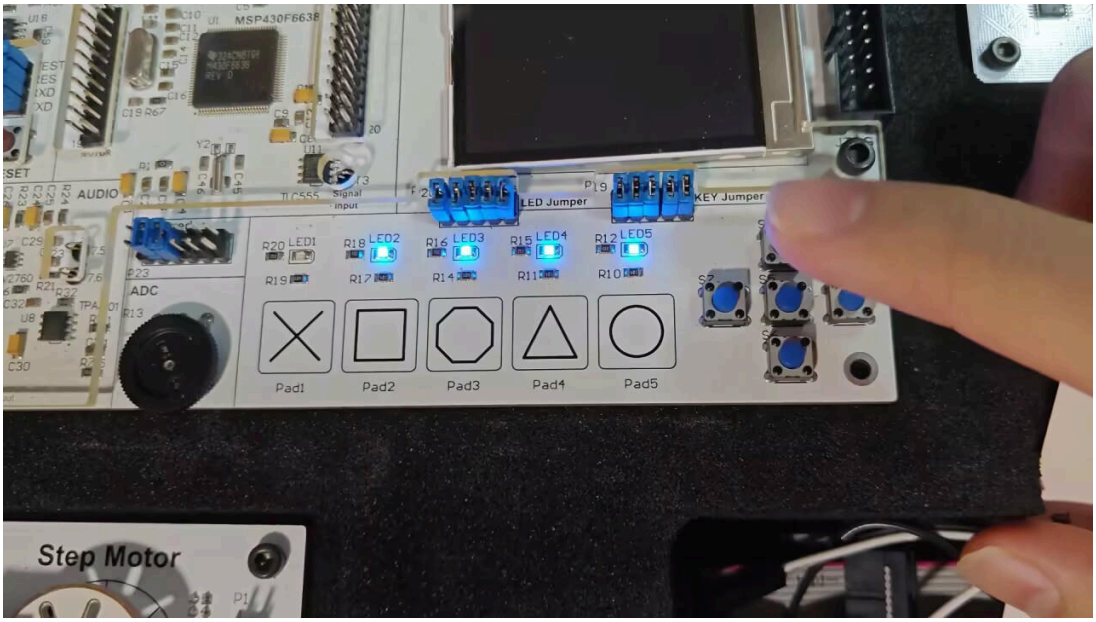

Figure 2: 改良版电灯效果，照片确实看不出来什么，还是看视频效果比较好 (1-2)

但是我们从视频中仍然可以发现，还是有问题，也就是在稍微多按一下还是会出现连续点亮的问题，在这里我们可以想办法让他在按钮被放开之前不进入下一次判定，代码实现我用了一个标记的 flag 来实现

```
if ((! (P4IN & BIT4)) && flag)
{
  count++;
  update_leds(count);
   __delay_cycles(1000000);
  flag = 0;
}

if (P4IN & BIT4)
{
  flag = 1;
}
```

这里加入的 flag 作用也就是，只在按钮被松开之后，可以触发下一次计数增加，实验之后发现效果良好.

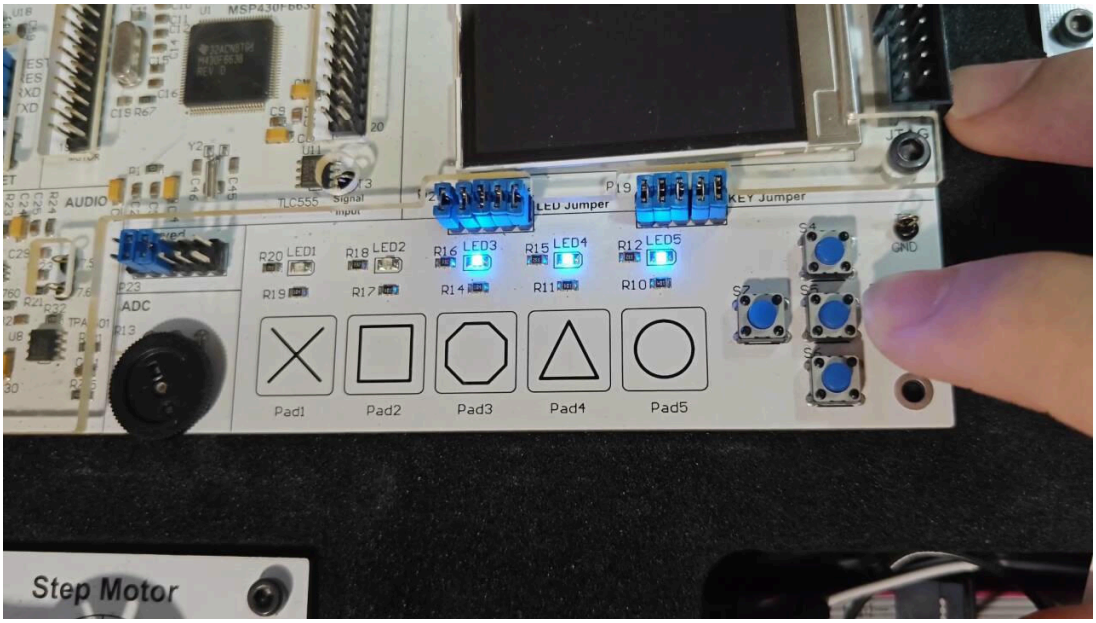

Figure 3: 算了，还是直接看视频吧(1-3)

2 段式 LCD

2.1 理论知识

从原理设计上，MSP430 的段式 LED 其实和我们之前的差别还挺大的，其驱动的管脚分为了 4 个 COM 信号和常规的 12 个管脚.

当然，在实践中其实写的代码其实和之前差别不大，就是注意 a~g 和 Px.0~Px.6 并不是一一对应关系

2.2 显示日期数字串和’SJTU’字符串

由于之前的示例实验已经提供了比较底层且详细的代码，所以数字串的显示我们可以直接调用已有的函数

```
const int date[6] = {0, 3, 0, 5, 2, 5};
while (1) // 进入程序主循环
{
  for (i = 0; i < 6; i++)
  {
    LCDSEG_SetDigit(i, date[i]);
    LCDSEG_SetSpecSymbol(4);
  }
  ...
```

这里 LCDSEG_SetDigit()就是在对应的位数显示数字，而 LCDSEG_SetSpecSymbol 就是显示小数点.

而显示字符串方面，虽然提供的驱动程序做不到，但我们也可以参考其写出一个

```
const uint8_t SJTU[4] = {0xbd, 0xe1, 0xe0, 0x6b};

void display_SJTU(int start_pos)
{
  int i = start_pos;
  int j;
  for (j = 0; j < 6; j++)
  {
    LCDSEG_SetDigit(j, -1);
    LCDSEG_ResetSpecSymbol(j);
  }
```

```
  for (j = 0; j < 4; j++)
  {
    uint8_t mem = LCDMEM[j + start_pos];
    mem = SJTU[j];
    LCDMEM[j + start_pos] = mem;
  }
}
```

首先我们写出四个字符对应的段数码， 之后我们参考之前的流程， 第一步从 `LCDMEM` 宏找到对应位的地址，然后把数据赋值给他所对应的地址.

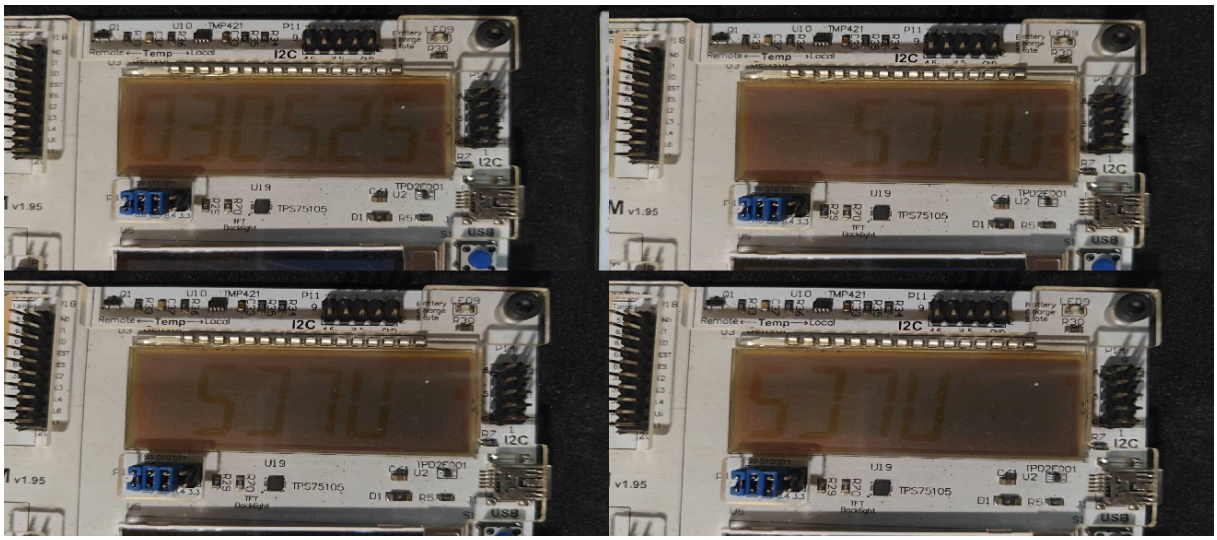

Figure 4: 显示效果，包含日期和滚动的 SJTU 字样，亦可参加视频