目录

1. 实验背景简介

本实验基于人机接口的交互操作功能（又称 Human Machine Interface，或 HMI）。人机接口是机器系统与用户之间进行交互和信息交换的媒介，实现信息的内部形式与人类可接受形式之间的转换。人机接口的使用十分广泛，传统形式的人机接口是电路箱，用户通过拨动电路箱的不同开关与按钮，来进行信息的转换与传递。现代常用的人机接口是触摸屏接口。用户只需要用特殊触屏笔或手指在触摸屏上操作，给予机器指令运作。

本次实验基于 K66 开发板为核心板进行二次扩展开发的 Cyber-Dorm 实验板，进行编程实践，实现人机交互的基本功能，Cyber-Dorm 实验板照片如图 1-1 所示。

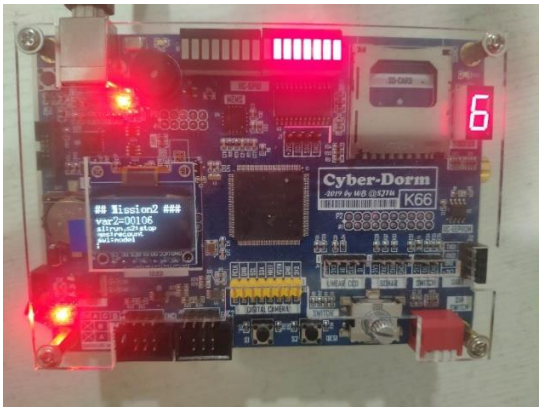

图 1-1 实验 Cyber-Dorm 开发板

2. 实验内容

本实验需要完成的任务是使用拨码开关设定步进值，使用数码开关调节变量，并将变量值通过数码管、光棒和OLED显示。

实验具体内容如下：定义两个变量Var1和Var2，设Var1初值为0，Var2初值为128。通过拨码开关SW1切换执行两个任务，如表2-1所示。

任务 1	任务 2：
程序主循环计数值显示与控制	数码开关 QES 变量显示与控制
S1 按键，使 Var1 清零并开始主循环计数，S2 按键停止计数，长按 S1 计数器清零	QES 顺时针旋转使 Var2 数值增加，Var2 最大值为十进制数 255（0xFF）
主循环计数值达到某个特定数值时蜂鸣器短鸣	QES 逆时针旋转使 Var2 减小，最小为 0
主循环计数值取 0x0F 的余数显示于数码管	按下 QES 开关使 Var2 恢复初值 128 (0x80)
合理设置延时时间来使显示字符清晰可见	将 Var2 数值显示于光柱或 OLED
测试“反应能力”：听到蜂鸣器鸣叫之后尽快按下 S2 并查看计数值	

表格 2-1 实验任务表

3. 实验原理

3.1 数码管显示控制

为实现 K66 电路板的人机接口功能，我们对 GPIO 信号进行编码，以提供一个友好的人机交互效果。使用旋转编码开关作为数字量输入，获得数码管上的数字量输出值。数码管的显示方法为：低电平点亮，高电平熄灭，多位状态同时控制。

3.2 中断控制

旋转编码开关以正交编码信号工作，如下图所示，A、B 信号相差 Phase1，目的是消除脉冲边缘振荡造成的干扰，达到良好的抗噪性能。

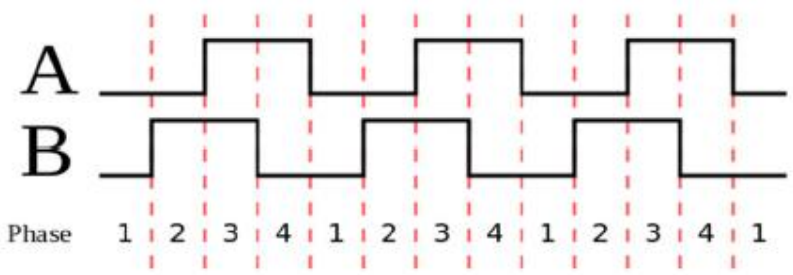

图 3-1 旋转开关原理

选择 A 或 B 信号之一作为中断信号源，配置中断条件为上升沿触发。即每出现一次中断信号的上升沿，就自动调用一次中断服务程序。我们将 A 信号作为中断信号源，那么每出现一次 A 信号上升沿，就调用中断服务程序，读取 B 信号的状态，1 表示旋转开关正转、0 表示反转，具体示例如图 3-1 所示。

3.3 IIC 总线控制

IIC 总线又称内部内置集成电路总线，旋转编码开关的输入量通过 K66 处理后经由 IIC 总线传给数出通道上，实现对 16 位光柱的点亮和熄灭控制。IIC 总线的主从关系和时序图如图 3-2 所示。

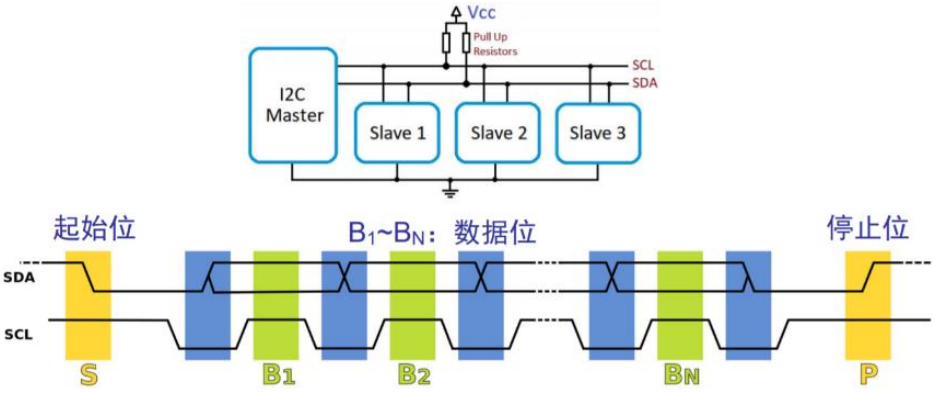

图 3-2 IIC 总线主从关系和时序图

4. 实验方案

4.1 整体框架

我们将本次实验分为两个任务，任务 1 和任务 2 分别将按键与旋转编码开关的输入转化为光柱、数码管、OLED 的输出显示。代码整体框架如下图 4-1 所示：

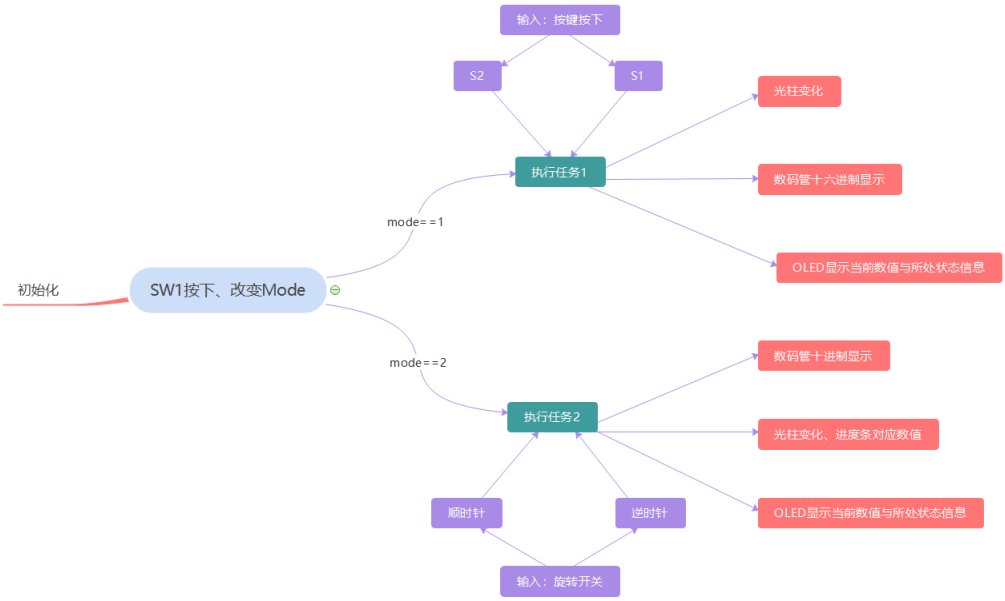

图 4-1 实验基本框图

我们小组利用基于 Ref1 示例代码文件进行开发拓展，完整实现了数码管显示计数值、数码开关的光柱显示、OLED 的任务信息显示等基本功能，并完成了按键与开关消抖的扩展任务，具体功能如下文所述。

4.2 系统中断服务

我们利用了以下三个中断用于代码的编写。

4.2.1 FTM Timer

我们首先尝试使用FTM定时器进行定时中断。FTM模块全称为FlexTimer Module，是一个有2-8个通道的定时器，内部有一个计数寄存器。每遇到一个时钟信号上升沿就给FTM寄存器加1以达到计时的功能。

我们测试了FTM定时中断与PIT定时中断后，发现两者效果差异不大。由于范例程序Ref1采用PIT中断，且FTM中断多用于硬件控制，最终我们采用PIT定时中断来实现中断功能。

FTM中断具体代码见文件ftm_timer.c与ftm_timer.h。

4.2.2 PIT Timer

我们使用 PIT 定时器来完成按键扫描与程序计时器。

PIT 全称为 Programmable Interval Timer，是一个可编程的周期中断定时器。PIT 的时钟源只有一个，即总线时钟。在 PIT 定时器是一个减法定时器，内部存在许多个寄存器，

有两个寄存器分别存储计数值与当前定时值，并在工作时会对存放的值以总线频率进行自减，而当计数值减为 0 时，PIT 就会被触发一次，并开启下一周期。

对于 K66 核心板，可以开启的 PIT 中断共有四个通道（kPIT_Chnl_0-kPIT_Chnl_3），在实验中，我们采用 kPIT_Chnl_0 通道做为计时器中断对计时器 appTick 进行加减，每 1ms 触发一次。同时，使用 kPIT_Chnl_1 通道作为按键动态扫描的中断通道（具体代码可见按键防抖部分），每 10ms 触发一次。

PIT 定时器的初始化程序和中断服务部分程序代码如下所示，完整代码可见文件 pit_timer.c 与 pit_timer.h。

```
1.  void InitPITs() {
2.      pit_config_t pitConfig;
3.      PIT_GetDefaultConfig(&pitConfig);
4.      PIT_Init(PIT, &pitConfig);
5.      /* Set timer period for channel 0: 1000us = 1ms */
6.      PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000U, APPTICK_PIT_SO
    URCE_CLOCK));
7.      /* Enable timer interrupts for channel 0 */
8.      PIT_EnableInterrupts(PIT, kPIT_Chnl_0, kPIT_TimerInterruptEnable);
9.      /* Enable at the NVIC */
10.     EnableIRQ(APPTICK_PIT_IRQn);
11.     /* Start channel 0 */
12.     PIT_StartTimer(PIT, kPIT_Chnl_0);
13.
14.     /* Set timer period for channel 1: 10000us = 10ms */
15.     PIT_SetTimerPeriod(PIT, kPIT_Chnl_1, USEC_TO_COUNT(10000U, Key_PIT_SOURC
    E_CLOCK));
16.     /* Enable timer interrupts for channel 1 */
17.     PIT_EnableInterrupts(PIT, kPIT_Chnl_1, kPIT_TimerInterruptEnable);
18.     /* Enable at the NVIC */
19.     EnableIRQ(Key_PIT_IRQn);
20.     /* Start channel 0 */
21.     PIT_StartTimer(PIT, kPIT_Chnl_1);
22.     appTick = 0;
23. }
24. //  @brief      appTicks increasing at an interval of 1ms
25. void APPTICK_PIT_HANDLER(void)
26. {
27.     /* Clear interrupt flag.*/
28.     PIT_ClearStatusFlags(PIT, kPIT_Chnl_0, kPIT_TimerFlag);
29.     appTick++;
30. #if defined __CORTEX_M && (__CORTEX_M == 4U)
31.     __DSB();
32. #endif
33. }
```

4.2.3 Systick Timer

我们使用 Systick 定时器在中断服务程序中实现精准延时，避免在主程序中使用 Delay 函数导致延时的不精确。Systick 定时器周期与系统时钟频率相关，我们根据所需的延时时间来设置 Systick 计时初值，每次进入中断服务程序程序对该计时初值进行修改操作，当计数值变为 0 时则跳出，完成精准延时的功能。

以下为 Systick 定时器的部分中断服务程序，完整代码可见 systick_timer.c 与 systick_timer.h。

```
#define systick_delay_us(time)    systick_delay(USEC_TO_COUNT(time, SYSTICK_SOURCE_CLOCK))
#define systick_delay_ns(time)  systick_delay(USEC_TO_COUNT(time,SYSTICK_SOURCE_CLOCK/1000))
//------------------------------------------------------------------------------
//  @brief      sysytick_delay assign time
//  @return     void
//  Sample usage:
//------------------------------------------------------------------------------
void systick_delay(uint32_t time)
{
    if(time == 0)   return;
    assert(SysTick_LOAD_RELOAD_Msk >= time);//time<=SysTick_LOAD_RELOAD_Msk

    SysTick->CTRL = 0x00;
    SysTick->LOAD = time;
    SysTick->VAL = 0x00;
    SysTick->CTRL = ( 0 | SysTick_CTRL_ENABLE_Msk
                    //| SysTick_CTRL_TICKINT_Msk
                      | SysTick_CTRL_CLKSOURCE_Msk
                );
    while( !(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk));
}
//------------------------------------------------------------------------------
//  @brief      sysytick_delay assign time_ms
//  @return     void
//  Sample usage:  systick_delay_ms(1000)
//------------------------------------------------------------------------------
void systick_delay_ms(uint32_t ms)
{
    while(ms--) systick_delay(SYSTICK_SOURCE_CLOCK/1000);
}
```

4.3 数码管、光柱、OLED 显示功能

4.3.1 数码管显示

本次实验使用的数码管是八段数码管，根据低电平则数码管亮、高电平则数码管暗的对应关系，我们对数码管的不同引脚进行驱动来控制数码管的显示。

数码管显示的部分代码如下，完整代码见库文件/HMI/seg.c与/HMI/seg.h。

```
1.    static uint8_t DSPTable[] = { 0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x02, 0x
   78, 0x00, 0x10, 0x08, 0x03, 0x27, 0x21, 0x06, 0x0E };
2.
3.  void ShowNumHEX(uint8_t num) {
4.      uint16_t l;
5.      l = DSPTable[num & 0x0F];
6.      // DSP: PD9~PD15
7.      GPIO_PortSet(BOARD_INITPINS_DSPa_GPIO, 0x7F << BOARD_INITPINS_DSPa_PIN);
     // turn off all DSP segments
8.      GPIO_PortClear(BOARD_INITPINS_DSPa_GPIO, ((~l)&0x7F) << BOARD_INITPINS_D
   SPa_PIN); // turn on the code segments
9.  }
10.
11. void ShowNumDEC(uint8_t num) {
12.     uint16_t l;
13.     l = DSPTable[num % 10];
14.     // DSP: PD9~PD15
15.     GPIO_PortSet(BOARD_INITPINS_DSPa_GPIO, 0x7F << BOARD_INITPINS_DSPa_PIN);
     // turn off all DSP segments
16.     GPIO_PortClear(BOARD_INITPINS_DSPa_GPIO, ((~l)&0x7F) << BOARD_INITPINS_D
   SPa_PIN); // turn on the code segments
17. }
```

4.3.2 光柱显示

我们使用IIC总线来控制光柱的显示，代码如下所示：

```
1.  void BOARD_I2C_GPIO(uint16_t ctrlValue) {
2.      uint8_t deviceAddress;
3.      g_master_txBuff[0] = ~(ctrlValue&0xFFU);
4.      g_master_txBuff[1] = ~(InverseByteBits((ctrlValue>>8)&0xFFU));
5.      deviceAddress     = 0x02U;
6.      masterXfer.slaveAddress   = I2C_CAT9555_SLAVE_ADDR_7BIT;
7.      masterXfer.direction      = kI2C_Write;
8.      masterXfer.subaddress     = (uint32_t)deviceAddress;
```

```
    masterXfer.subaddressSize = 1;
    masterXfer.data           = g_master_txBuff;
    masterXfer.dataSize       = I2C_CAT9555_DATA_LENGTH;
    masterXfer.flags          = kI2C_TransferDefaultFlag;
    I2C_MasterTransferBlocking(I2C_CAT9555_BASEADDR, &masterXfer);
}
```

4.3.3 OLED 显示

我们同样使用IIC总线对OLED屏进行控制，并对原有OLED库代码进行些许的完善与调整，部分函数代码如下所示，完整代码见/HMI/CDK66_OLED.c与/HMI/CDK66_OLED.h。

```
//-----------------------------------------------------------------------
//  @brief      OLED show int16
//  @return     void
//  Sample usage:  oled_uint16(40, 2, Var2);
//-----------------------------------------------------------------------
void oled_int16(uint8_t x, uint8_t y, int16_t num)
{
    uint8_t ch[7];
    if(num<0)   {num = -num;OLED_P6x8Str(x, y, "-");}
    else         OLED_P6x8Str(x, y, " ");
    x+=6;

    OLED_HEXACSII(num,ch);
    OLED_P6x8Str(x, y, &ch[1]);     //6*8
}
```

```
//-----------------------------------------------------------------------
//  @brief      OLED float show
//  @param      dat         float or double
//  @param      num         bit   max=10
//  @param      pointnum    Decimal display length   max=6
//  @return     void
//  Sample usage:           oled_printf_float(0,0,x,2,3);
//-----------------------------------------------------------------------
void oled_printf_float(uint16_t x,uint16_t y,double dat,uint8_t num,uint8_t pointnum)
{
    uint8_t   length;
    int8_t    buff[34];
    int8_t    start,end,point;
    if(6<pointnum)  pointnum = 6;
```

```
15.     if(10<num)      num = 10;
16.     if(0>dat)   length = printf( &buff[0],"%f",dat);//-
17.     else
18.     {
19.         length = printf( &buff[1],"%f",dat);
20.         length++;
21.     }
22.     point = length - 7;
23.     start = point - num - 1;
24.     end = point + pointnum + 1;
25.     while(0>start)
26.     {
27.         buff[end] = ' ';
28.         end++;
29.         start++;
30.     }
31.     if(0>dat)   buff[start] = '-';
32.     else        buff[start] = ' ';
33.     buff[end] = '\0';
34.     OLED_P6x8Str(x, y, buff);
35. }
```

4.4 按键与旋转开关的消抖实现

4.4.1 旋转编码开关消抖

任务2使用的是旋转编码开关，此种开关在旋转的时候容易因电平信号混乱引起较为严重的抖动，需消抖处理。

原有Ref1示例代码采用A上升沿时触发，并检测B信号来进行输入信号的判断，此方法在实际操作中效果不佳。因此我们将采用两次判断消抖的模式，将可能错误的信号滤去，以减少不必要的干扰。

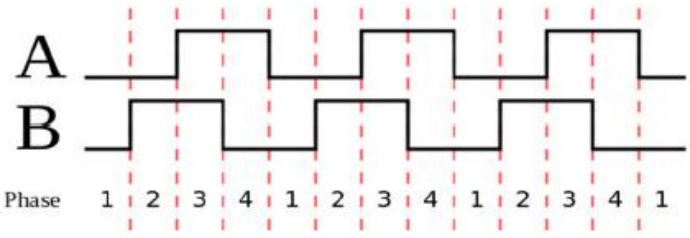

图 4-2 旋转开关电平时序图

将 A 配置为双边沿触发，当 A 触发第一次时，对 B 的电平状态进行记录，但并不直接输出控制信号，只是进行一次标志位的翻转，当 A 第二次触发时，对 B 的电平信号再次进行判断，如果与第一次判断结果相同，再进行输出，反之将信号作为抖动信号滤去，具体程序框架见下图。

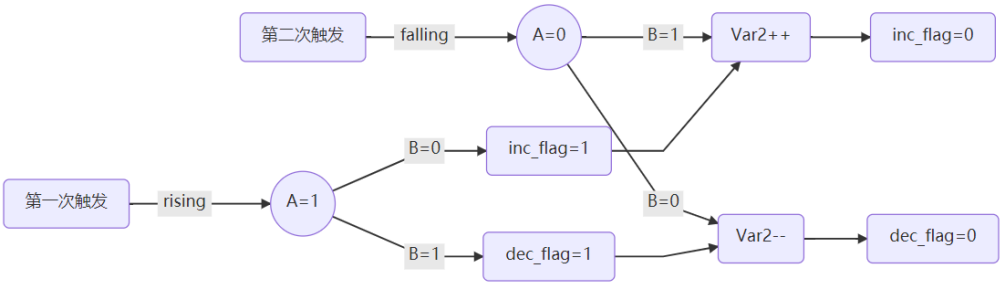

图 4-3 旋转开关防抖流程图

为了代码运行的实时性，我们将旋转编码开关的信号读取置于 PORTB_IRQn 中断处理函数里。

```
1.  void BOARD_RESa_IRQ_HANDLER(void)
2.  {
3.      GPIO_PortClearInterruptFlags(BOARD_INITPINS_QESa_GPIO, BOARD_INITPINS_QE
    Sa_GPIO_PIN_MASK);
4.      if (QESA())
5.      {
6.          if (!QESB())   QES_incflag = 1; //maybe increase
7.          if (QESB())    QES_decflag = 1; //maybe decrease
8.      }
9.      else
10.     {
11.         //Repeat judgment, elimination buffeting of keystroke of QES
12.         if (QESB() && QES_incflag == 1 && mode == 2)
13.         {
14.             Var2++;
15.         }
16.         if (!QESB() && QES_decflag == 1 && mode == 2)
17.         {
18.             Var2--;
19.         }
20.         Var2 = MINMAX(0, Var2, 255); //constrain(0,255)
21.         QES_decflag = 0;
22.         QES_incflag = 0;
23.     }
```

4.4.2 按键消抖

我们尝试了两种方式实现消抖：延时消抖和 FIFO 消抖。

a. 延时消抖

延时消抖即在进行一次按键检测后隔一段时间再进行按键检测，如果两次检测不同则说明按键已经按下，值得注意的是延时时间的选取，既要足以度过抖动期还不能太长导致执行效率变低。延时消抖具体代码如下：

```
1.  KEY_STATUS_e key_check(KEY_e key)
2.  {
3.      if(key_get(key) == KEY_DOWN)
4.      {
5.          systick_delay_ms(KEY_DOWN_DELAY_TIME);
6.          if(key_get(key) == KEY_DOWN)
7.          {
8.              return KEY_DOWN;
9.          }
10.     }
11.     return KEY_UP;
12. }
```

b. FIFO 消抖

第二种消抖方法是使用 FIFO 消抖处理，即使用队列对按键动作进行存储，通过读取队列即可获取按键的状态。

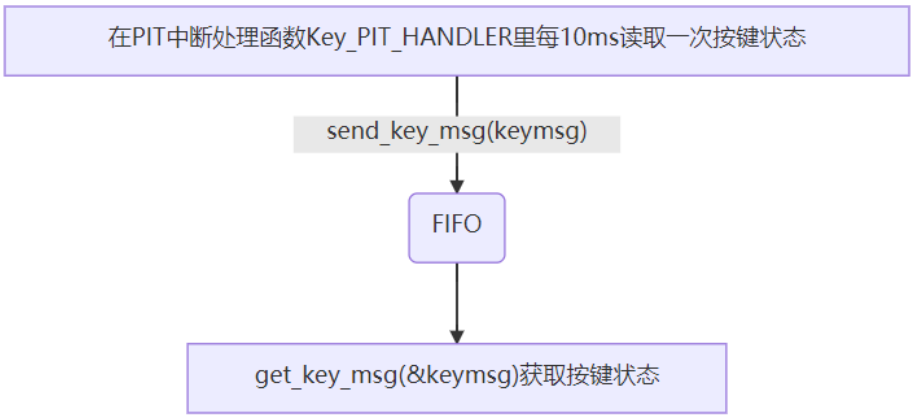

图 4-4 FIFO 消抖基本思路

使用 FIFO 进行消抖的具体实验结果可见下图，可以看到，每当 key_msg 接受到三个按键的状态改变信息，都会对队列进行修正，并存储相关信息。

```
K66_Ref1 JLink Debug (1) [GDB SEGGER Interface Debugging] K66_Ref1.axf
key_msg:1
 key1_status:1, key2_status:2, QES_status:1
 key1_count1:6, key2_count:4, QESP_count:0
key_msg:1
 key1_status:1, key2_status:2, QES_status:1
 key1_count1:6, key2_count:4, QESP_count:0
key_msg:1
 key1_status:1, key2_status:2, QES_status:1
 key1_count1:6, key2_count:4, QESP_count:0
key_msg:1
 key1_status:1, key2_status:1, QES_status:1
 key1_count1:6, key2_count:4, QESP_count:0
key_msg:0
 key1_status:0, key2_status:1, QES_status:1
 key1_count1:7, key2_count:4, QESP_count:0
key_msg:0
 key1_status:2, key2_status:1, QES_status:1
 key1_count1:7, key2_count:4, QESP_count:0
key_msg:0
 key1_status:2, key2_status:1, QES_status:1
 key1_count1:7, key2_count:4, QESP_count:0
key_msg:0
 key1_status:2, key2_status:1, QES_status:1
 key1_count1:7, key2_count:4, QESP_count:0
```

采用这一方法进行防抖有以下三个优点：

1、可以有效记录按键事件的发生，特别是系统要实现记录按键按下、松开、长按时，使用 FIFO 来实现是一种不错的选择方式。

2、系统是阻塞的，这样系统在检测到按键按下的情况，由于机械按键抖动的原因不需要在这里等待一段时间，然后在确定按键是否正常按下。

3、按键 FIFO 程序在系统定时器中定时检测按键状态，确认按键按下后将状态写入 FIFO 中，不一定在主程序中一直做检测，这样可以有效降低系统资源的消耗。

定义按键消息的结构体代码如下：

```
1.  typedef enum
2.  {
3.      KEY_DOWN  =   0,
4.      KEY_UP    =   1,
5.      KEY_HOLD  =   2,             //press
6.      KEY_LONG_HOLD  =   3,       //long press
7.  } KEY_STATUS_e;
8.
9.  typedef enum
10. {
11.     KEY1,
12.     KEY2,
13.     QESP,
14.     KEY_MAX,
15. } KEY_e;
16.
17. typedef struct
18. {
19.     KEY_e           key;
20.     KEY_STATUS_e    status[KEY_MAX];
21.     uint32_t        count[KEY_MAX];
22. } KEY_MSG_t;
```

定时扫描按键状态置于中断 PIT1_IRQn 中，并于每 10ms 触发一次，具体处理代码如下所示。

```
1.  //-------------------------------------------------------------------------
2.  //  @brief      Check key status regularly(10ms), using FIFO 1
3.  //  @data       2020/10/6
4.  //-------------------------------------------------------------------------
5.  void Key_PIT_HANDLER(void)
6.  {
7.      PIT_ClearStatusFlags(PIT, kPIT_Chnl_1, kPIT_TimerFlag);
8.
9.      KEY_e   keynum;
10.     static uint8_t keytime[KEY_MAX];
11.     for(keynum = (KEY_e)0 ; keynum < KEY_MAX; keynum ++)
12.     {
13.         if(key_get(keynum) == KEY_DOWN)
14.         {
15.             keytime[keynum]++;
16.             if(keytime[keynum] == KEY_DOWN_DELAY_TIME + 1 )  //down
17.             {
18.                 keymsg.key = keynum;
19.                 keymsg.status[keynum] = KEY_DOWN;
20.                 keymsg.count[keynum]++;
21.                 send_key_msg(keymsg);
22.             }
23.             else if(keytime[keynum] > KEY_LONG_HOLD_TIME) //long press
24.             {
25.                 keymsg.key = keynum;
26.                 keymsg.status[keynum] = KEY_LONG_HOLD;
27.                 send_key_msg(keymsg);
28.             }
29.             else if(keytime[keynum] > KEY_HOLD_TIME)  //press
30.             {
31.                 keymsg.key = keynum;
32.                 keymsg.status[keynum] = KEY_HOLD;
33.                 send_key_msg(keymsg);
34.             }
35.         }
36.         else
37.         {
38.             if(keytime[keynum] > KEY_DOWN_DELAY_TIME)  //up
39.             {
40.                 keymsg.key = keynum;
```

```
                keymsg.status[keynum] = KEY_UP;
                send_key_msg(keymsg);
            }
            keytime[keynum] = 0;
        }
    }
    if (SW1()) { premode=mode;mode=1; }
    else { premode=mode;mode=2; }
    if (keymsg.status[KEY1] == KEY_DOWN && mode==1) { loop_flag = 1;}
    if (keymsg.status[KEY2] == KEY_DOWN && mode==1) { loop_flag = 0;}

#if defined __CORTEX_M && (__CORTEX_M == 4U)
    __DSB();
#endif
}
```

FIFO 队列具体实现代码如下所示

```
//------------------------------------------------------------------------------
//  @brief          Send key message to FIFO
//  @data           2020/10/6
//  Sample usage:
//------------------------------------------------------------------------------
void send_key_msg(KEY_MSG_t keymsg)
{
    uint8_t tmp;
    if(key_msg_flag == KEY_MSG_FULL)
    {
        return ;
    }
    key_msg[key_msg_rear].key = keymsg.key;
    key_msg[key_msg_rear].status[keymsg.key] = keymsg.status[keymsg.key];
    key_msg_rear++;

    if(key_msg_rear >= KEY_MSG_FIFO_SIZE)
    {
        key_msg_rear = 0;                       //start over
    }

    tmp = key_msg_rear;
    if(tmp == key_msg_front)                    //full
    {
        key_msg_flag = KEY_MSG_FULL;
    }
```

```
    else
    {
        key_msg_flag = KEY_MSG_NORMAL;
    }
}
//------------------------------------------------------------------------
//  @brief         Get key press message from FIFO
//  @return        1-acquisition, 0-the key message is not obtained
//  @data          2020/10/6
//  Sample usage:
//------------------------------------------------------------------------
uint8_t get_key_msg(KEY_MSG_t *keymsg)
{
    uint8_t tmp;
    if(key_msg_flag == KEY_MSG_EMPTY)               //The key message FIFO i
s empty, return 0 directly
    {
        return 0;
    }
    keymsg->key = key_msg[key_msg_front].key;       //Get the key value from
 the head of the FIFO
    keymsg->status[keymsg->key] = key_msg[key_msg_front].status[keymsg->key]
; //Get the button type from the FIFO head

    key_msg_front++;     //The first pointer of the FIFO team is incremented
 by 1 to point to the next message

    if(key_msg_front >= KEY_MSG_FIFO_SIZE)          //If the FIFO pointer ov
erflows at the head of the queue, it starts counting from 0
    {
        key_msg_front = 0;                          //Start over
    }
    tmp = key_msg_rear;
    if(key_msg_front == tmp)    //Compare whether the head and tail of the t
eam are the same, the same means that the FIFO is empty
    {
        key_msg_flag = KEY_MSG_EMPTY;
    }
    else
    {
        key_msg_flag = KEY_MSG_NORMAL;
    }
    return 1;
}
```

5. 实验结果分析与结论

本次实验以人机交互实验(HMI)为基本任务展开，对实验开发板上的数码管、光棒、数码开关等进行实践。在完成两个实验任务的同时，我们完善了HMI库中按键、数码管、LED等相关库函数，并尽可能以易读的方式进行书写。

同时，在本次实验中我们通过对按键防抖这一扩展任务进行多种方法的尝试，对防抖这一任务最终选择了我们认为较优的策略，也对这一嵌入式开发的最常用代码有了进一步的了解，并进行了库的封装。

经过本次实验，我们基本熟悉示例代码Ref1的基本思路与代码实现方法，并以此代码为基础进行调整与改进，进行基本库函数的完善与封装，为后续实验积累库函数与便于参照的工程代码，后续的实验内容我们也会在此次实验代码的基础上进行开发。

最后，感谢王冰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。