

5. 1keypad 状态机事件

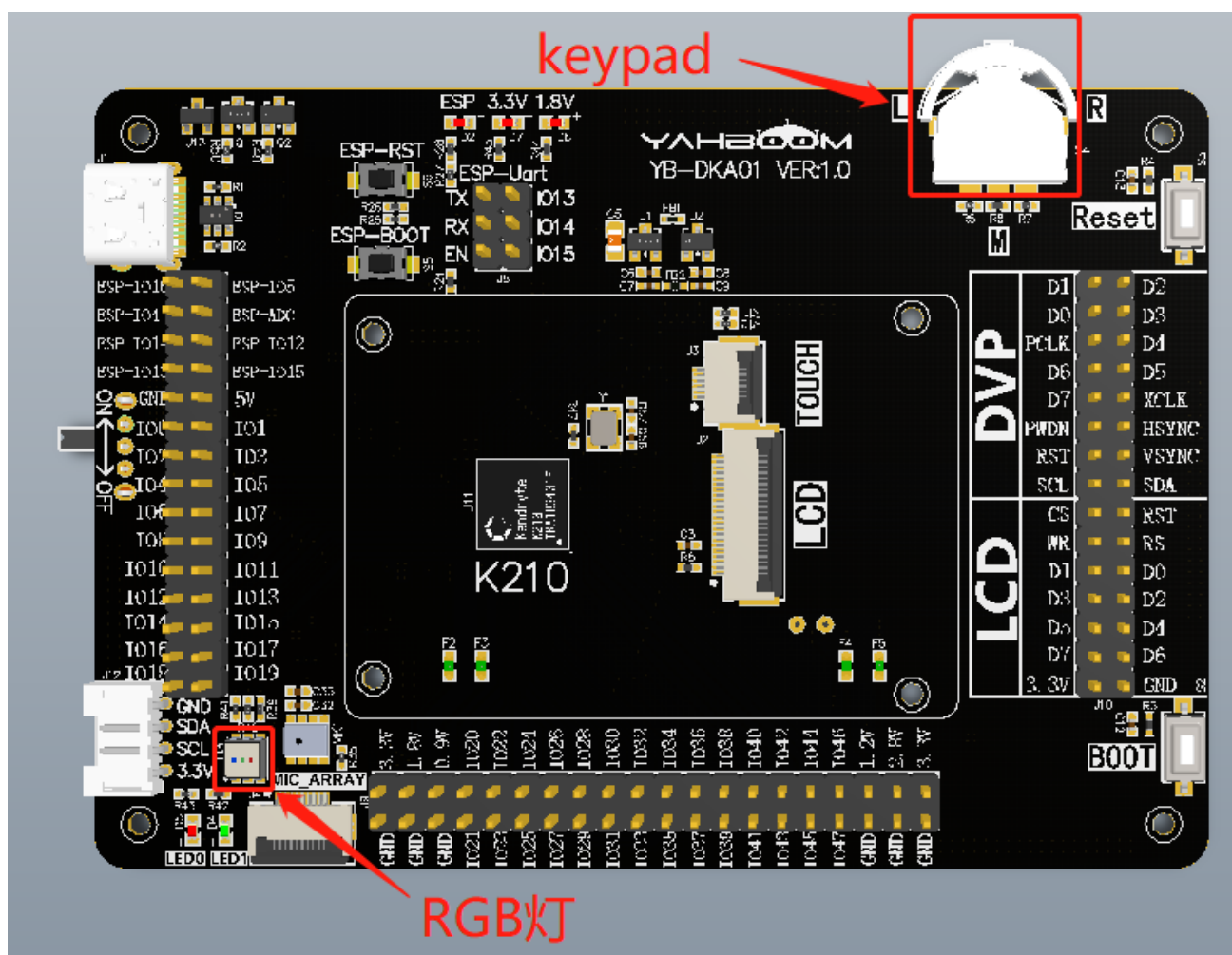
一、实验目的

本节课主要学习 K210 的拨轮开关 keypad 利用状态机的方式检测按键事件。

二、实验准备

1. 实验元件

拨轮开关 keypad、RGB 灯



三、实验原理

拨轮开关 keypad 是通过拨动开关柄使电路接通或断开，从而达到切换电路

的目的，拨轮开关的原理是经过人为的操作，控制对应电路接通，这里的作用是接通 GND，使 IO 口电平变为低电平，松开时弹簧自动复位的过程。利用定时器扫描 keypad 的状态，在定时器扫描过程中计算 keypad 状态的时间值，从而读取 keypad 三个通道的按键状态，存入 FIFO 队列机制，在通过函数回调的方式执行每个状态需要处理的事件，或者可以通过读取按键的状态，再根据状态的不同进行处理事件。

四、实验过程

1. 首先根据上面的硬件连接引脚图，K210 的硬件引脚和软件功能使用的是 FPIOA 映射关系。

```
/******HARDWARE-PIN*****  
// 硬件IO口，与原理图对应  
#define PIN_KEYPAD_LEFT      (1)  
#define PIN_KEYPAD_MIDDLE    (2)  
#define PIN_KEYPAD_RIGHT     (3)  
  
#define PIN_RGB_R            (6)  
#define PIN_RGB_G            (7)  
#define PIN_RGB_B            (8)  
  
/******SOFTWARE-GPIO*****  
// 软件GPIO口，与程序对应  
#define KEYPAD_LEFT_GPIONUM   (1)  
#define KEYPAD_MIDDLE_GPIONUM (2)  
#define KEYPAD_RIGHT_GPIONUM (3)  
  
#define RGB_R_GPIONUM         (4)  
#define RGB_G_GPIONUM         (5)  
#define RGB_B_GPIONUM         (6)  
  
/******FUNC-GPIO*****  
// GPIO口的功能，绑定到硬件IO口  
#define FUNC_KEYPAD_LEFT      (FUNC_GPIOHS0 + KEYPAD_LEFT_GPIONUM)  
#define FUNC_KEYPAD_MIDDLE    (FUNC_GPIOHS0 + KEYPAD_MIDDLE_GPIONUM)  
#define FUNC_KEYPAD_RIGHT     (FUNC_GPIOHS0 + KEYPAD_RIGHT_GPIONUM)  
  
#define FUNC_RGB_R            (FUNC_GPIOHS0 + RGB_R_GPIONUM)  
#define FUNC_RGB_G            (FUNC_GPIOHS0 + RGB_G_GPIONUM)  
#define FUNC_RGB_B            (FUNC_GPIOHS0 + RGB_B_GPIONUM)
```

```
void hardware_init(void)
{
    fpioa_set_function(PIN_KEYPAD_LEFT,  FUNC_KEYPAD_LEFT);
    fpioa_set_function(PIN_KEYPAD_MIDDLE, FUNC_KEYPAD_MIDDLE);
    fpioa_set_function(PIN_KEYPAD_RIGHT,  FUNC_KEYPAD_RIGHT);
}
```

2. 由于需要使用定时器中断，所以需要初始化系统中断和使能全局中断。

```
/* 初始化系统中断并使能全局中断 */
plic_init();
sysctl_enable_irq();
```

3. 在使用 RGB 灯前需要初始化，这里可以单独只初始化一个灯，或者全部灯一起初始化，当然我们选择的是全部灯一起初始化。

```
/* 初始化RGB灯 */
rgb_init(EN_RGB_ALL);
```

```
/* 初始化RGB灯 */
void rgb_init(rgb_color_t color)
{
    switch (color)
    {
        case EN_RGB_RED:
            fpioa_set_function(PIN_RGB_R, FUNC_RGB_R);
            gpiohs_set_drive_mode(RGB_R_GPIONUM, GPIO_DM_OUTPUT);
            gpiohs_set_pin(RGB_R_GPIONUM, GPIO_PV_HIGH);
            break;
        case EN_RGB_GREEN:
            fpioa_set_function(PIN_RGB_G, FUNC_RGB_G);
            gpiohs_set_drive_mode(RGB_G_GPIONUM, GPIO_DM_OUTPUT);
            gpiohs_set_pin(RGB_G_GPIONUM, GPIO_PV_HIGH);
            break;
        case EN_RGB_BLUE:
            fpioa_set_function(PIN_RGB_B, FUNC_RGB_B);
            gpiohs_set_drive_mode(RGB_B_GPIONUM, GPIO_DM_OUTPUT);
            gpiohs_set_pin(RGB_B_GPIONUM, GPIO_PV_HIGH);
            break;
        case EN_RGB_ALL:
            fpioa_set_function(PIN_RGB_R, FUNC_RGB_R);
            gpiohs_set_drive_mode(RGB_R_GPIONUM, GPIO_DM_OUTPUT);
            fpioa_set_function(PIN_RGB_G, FUNC_RGB_G);
            gpiohs_set_drive_mode(RGB_G_GPIONUM, GPIO_DM_OUTPUT);
            fpioa_set_function(PIN_RGB_B, FUNC_RGB_B);
            gpiohs_set_drive_mode(RGB_B_GPIONUM, GPIO_DM_OUTPUT);

            gpiohs_set_pin(RGB_R_GPIONUM, GPIO_PV_HIGH);
            gpiohs_set_pin(RGB_G_GPIONUM, GPIO_PV_HIGH);
            gpiohs_set_pin(RGB_B_GPIONUM, GPIO_PV_HIGH);
            break;
        default:
            break;
    }
}
```

4. 初始化 keypad，设置 GPIO 为上拉输入模式，把 FIFO 清零，再设置 keypad 三个通道的初始数据，最后是初始化并启动定时器。

```
/* 初始化keypad */
keypad_init();
```

```
/* 初始化keypad */
void keypad_init(void)
{
    /* 设置keypad三个通道为上拉输入 */
    gpiohs_set_drive_mode(KEYPAD_LEFT_GPIONUM, GPIO_DM_INPUT_PULL_UP);
    gpiohs_set_drive_mode(KEYPAD_MIDDLE_GPIONUM, GPIO_DM_INPUT_PULL_UP);
    gpiohs_set_drive_mode(KEYPAD_RIGHT_GPIONUM, GPIO_DM_INPUT_PULL_UP);

    /* FIFO读写指针清零 */
    keypad_fifo.read = 0;
    keypad_fifo.write = 0;
    /* keypad变量初始化 */
    for (int i = 0; i < EN_KEY_ID_MAX; i++)
    {
        keypad[i].long_time = KEY_LONG_TIME;           /* 长按时间 0 表示不检测长按键事件 */
        keypad[i].count = KEY_FILTER_TIME;             /* 计数器设置为滤波时间 */
        keypad[i].state = RELEASE;                     /* 按键缺省状态，0为未按下 */
        keypad[i].repeat_speed = KEY_REPEAT_TIME;      /* 按键连发的速度，0表示不支持连发 */
        keypad[i].repeat_count = 0;                    /* 连发计数器 */

        keypad[i].short_key_down = NULL;               /* 按键按下回调函数*/
        keypad[i].skd_arg = NULL;                      /* 按键按下回调函数参数*/
        keypad[i].short_key_up = NULL;                 /* 按键抬起回调函数*/
        keypad[i].sku_arg = NULL;                      /* 按键抬起回调函数参数*/
        keypad[i].long_key_down = NULL;                /* 按键长按回调函数*/
        keypad[i].lkd_arg = NULL;                      /* 按键长按回调函数参数*/
        keypad[i].repeat_key_down = NULL;
        keypad[i].rkd_arg = NULL;
        keypad[i].get_key_status = get_keys_state_hw;
        /* 允许上报的按键事件 */
        keypad[i].report_flag = KEY_REPORT_DOWN | KEY_REPORT_UP | KEY_REPORT_LONG | KEY_REPORT_REPEAT;
    }

    mTimer_init();
}
```

5. 定时器使用的是定时器 0 的通道 0，定时中断时间为 1 毫秒，也就是每毫秒扫描一次 keypad。

```

/* 定时器回调函数，功能是扫描keypad */
static int timer_irq_cb(void * ctx)
{
    scan_keypad();
}

/* 初始化并启动定时器0的通道0，每毫秒中断一次 */
static void mTimer_init(void)
{
    timer_init(TIMER_DEVICE_0);
    timer_set_interval(TIMER_DEVICE_0, TIMER_CHANNEL_0, 1e6);
    timer_irq_register(TIMER_DEVICE_0, TIMER_CHANNEL_0, 0, 1, timer_irq_cb, NULL);

    timer_set_enable(TIMER_DEVICE_0, TIMER_CHANNEL_0, 1);
}

```

6. 在扫描 keypad 函数中有两种方式可以选择，默认是状态机方式扫描，更快更方便，第二种是循环判断的方式。

```

/* 扫描keypad */
void scan_keypad()
{
    for (uint8_t i = 0; i < EN_KEY_ID_MAX; i++)
    {
        #if USE_STATE_MACHINE
            detect_key_state((key_id_t)i);
        #else
            detect_key((key_id_t)i);
        #endif
    }
}

```

7. 这里以状态机扫描方式来说明一下工作原理，由于 keypad 的特性，所以同一时间只能有一个通道是有效的，所以获取到对应通道的 ID，只处理有效的 ID 就可以。首先没有触发事件的是否都在 EN_KEY_NULL 这个状态中，当检测到 key_state 状态改变，也就是有效触发按键事件，就改变状态 key_state 为 EN_KEY_DOWN。

```

/* 状态机的方式检测按键,分析按键事件 */
static void detect_key_state(key_id_t key_id)
{
    keypad_t *p_key;
    p_key = &keypad[key_id];           // 指针指向按键事件结构体
    uint8_t current_key_state;         // 当前按键状态
    current_key_state = p_key->get_key_status(key_id);
    switch (p_key->key_state)
    {
    case EN_KEY_NULL:
        // 如果按键被按下
        if (current_key_state == PRESS)
        {
            p_key->key_state = EN_KEY_DOWN;
        }
        break;
    }
}

```

在每一次扫描结束时,都会把当前的状态传给 prev_key_state,表示上一次状态。

```

    p_key->prev_key_state = current_key_state;
}

```

EN_KEY_DOWN 状态下,会先判断状态是否保持,相当于消抖功能,然后上报按键按下事件,并存入 FIFO,执行回调函数,修改状态为 EN_KEY_DOWN_RECHECK。

```

case EN_KEY_DOWN:
    // 如果状态还在保持
    if (current_key_state == p_key->prev_key_state)
    {
        p_key->key_state = EN_KEY_DOWN_RECHECK;
        if(p_key->report_flag & KEY_REPORT_DOWN) //如果定义了按键按下上报功能
        {
            //存入按键按下事件
            key_in_fifo((keypad_status_t)(KEY_STATUS * key_id + 1));
        }
        if (p_key->short_key_down) //如果注册了回调函数 则执行
        {
            p_key->short_key_down(p_key->skd_arg);
        }
    }
    else
    {
        p_key->key_state = EN_KEY_NULL;
    }
    break;
}

```

EN_KEY_DOWN_RECHECK 状态主要处理长按、连发和松开，长按的功能是每次扫描加一个 KEY_TICKS 的值，这个是 1ms 扫描，所以 KEY_TICKS 的值为 1，当长按的计数 long_count 大于或等于设定的长按超时时间 long_time，则触发事件存入 FIFO，并执行回调函数。长按事件触发后，紧跟着就是连发的事件，连发的工作机制在于每次触发连发事件后执行回调函数，还会把 repeat_count 清空，等待下一次 repeat_count 溢出，再次执行任务，最后是按键松开的情况，切换到 EN_KEY_UP 状态。

```
// 长按、连发和按键松开判断
case EN_KEY_DOWN_RECHECK:
    //按键还在保持按下状态
    if(current_key_state == p_key->prev_key_state)
    {
        if(p_key->long_time > 0)
        {
            if (p_key->long_count < p_key->long_time)
            {
                if ((p_key->long_count += KEY_TICKS) >= p_key->long_time)
                {
                    if(p_key->report_flag & KEY_REPORT_LONG)
                    {
                        // 存入长按事件
                        key_in_fifo((keypad_status_t)(KEY_STATUS * key_id + 3));
                    }
                    if(p_key->long_key_down) // 回调
                    {
                        p_key->long_key_down(p_key->lkd_arg);
                    }
                }
            }
        }
    }
}
```

```

else // 连发
{
    if(p_key->repeat_speed > 0)
    {
        if ((p_key->repeat_count += KEY_TICKS) >= p_key->repeat_speed)
        {
            p_key->repeat_count = 0;
            //如果定义的连发上报
            if(p_key->report_flag & KEY_REPORT_REPEAT)
            {
                key_in_fifo((keypad_status_t)(KEY_STATUS * key_id + 4));
            }
            if(p_key->repeat_key_down)
            {
                p_key->repeat_key_down(p_key->rkd_arg);
            }
        }
    }
}
else // 按键松开
{
    p_key->key_state = EN_KEY_UP;
}
break;

```

EN_KEY_UP 状态也会先判断前后两次状态的值是否一致，如果一致就表示按键松开了，然后触发松开的回调函数，状态修改为 EN_KEY_UP_RECHECK。

```

case EN_KEY_UP:
    if (current_key_state == p_key->prev_key_state)
    {
        p_key->key_state = EN_KEY_UP_RECHECK;
        p_key->long_count = 0; //长按计数清零
        p_key->repeat_count = 0; //重复发送计数清零
        if(p_key->report_flag & KEY_REPORT_UP)
        {
            // 按键松开
            key_in_fifo((keypad_status_t)(KEY_STATUS * key_id + 2));
        }
        if (p_key->short_key_up)
        {
            p_key->short_key_up(p_key->sku_arg);
        }
    }
    else
    {
        p_key->key_state = EN_KEY_DOWN_RECHECK;
    }
    break;

```


EN_KEY_UP_RECHECK 的功能主要是确定已经松开，并且修改状态为 EN_KEY_NULL。

```
case EN_KEY_UP_RECHECK:
    if (current_key_state == p_key->prev_key_state)
    {
        p_key->key_state = EN_KEY_NULL;
    }
    else
    {
        p_key->key_state = EN_KEY_UP;
    }
    break;
default:
    break;
```

最后也就是前面讲过的把当前状态保存到 prev_key_state 中，作为下一次扫描的上一次状态。

```
p_key->prev_key_state = current_key_state;
}
```

8. 保存某个按键的状态到 FIFO 中。

```
// 将实际某个按键的状态存入FIFO中
static void key_in_fifo(keypad_status_t keypad_status)
{
    keypad_fifo.fifo_buffer[keypad_fifo.write] = keypad_status;
    if (++keypad_fifo.write >= KEY_FIFO_SIZE)
    {
        keypad_fifo.write = 0;
    }
}
```

9. 读出 keypad 的一个事件，默认为 EN_KEY_NONE，其他数字对应不同事件。

```
/* 获取keypad的状态，如果没有事件，默认为0 */
keypad_status_t get_keypad_state(void)
{
    return key_out_fifo();
}
```

```

/* 从FIFO读取一个按键事件 */
keypad_status_t key_out_fifo(void)
{
    keypad_status_t key_event;
    if (keypad_fifo.read == keypad_fifo.write)
    {
        return EN_KEY_NONE;
    }
    else
    {
        key_event = keypad_fifo.fifo_buffer[keypad_fifo.read];
        if (++keypad_fifo.read >= KEY_FIFO_SIZE)
        {
            keypad_fifo.read = 0;
        }
        return key_event;
    }
}

```

10. 如果需要修改触发的时间，可以修改以下参数。KEY_TICKS 为扫描周期，与定时器的中断周期保持一致就可以，如果不一致会导致下面的时间有差异；KEY_FILTER_TIME 为消抖时间；KEY_LONG_TIME 为长按的触发时间；KEY_REPEAT_TIME 是指长按触发后的重复触发的时间。

```

/* 修改按键触发时间 */
#define KEY_TICKS          1           // 按键扫描周期
#define KEY_FILTER_TIME    10          // 按键消抖时间
#define KEY_LONG_TIME      1000        // 长按触发时间(ms)
#define KEY_REPEAT_TIME    200         // 连发间隔(ms)

```

11. 初始化 keypad 完成后，就可以获取 keypad 的事件，总共有两种方式，第一种是通过设置回调函数的方式，第二种是通过读取状态值的方式。

第一种：设置 keypad 三个通道的事件回调函数，由于同一时间内只能有一个通道是有效的，所以把三个通道的不同状态的回调函数设置为同一个，当然也可以分别定义一个函数作为回调函数。

```

/* 设置keypad回调 */
keypad[EN_KEY_ID_LEFT].short_key_down = key_press;
keypad[EN_KEY_ID_LEFT].short_key_up = key_release;
keypad[EN_KEY_ID_LEFT].long_key_down = key_long_press;
keypad[EN_KEY_ID_LEFT].repeat_key_down = key_repeat;

keypad[EN_KEY_ID_MIDDLE].short_key_down = key_press;
keypad[EN_KEY_ID_MIDDLE].short_key_up = key_release;
keypad[EN_KEY_ID_MIDDLE].long_key_down = key_long_press;
keypad[EN_KEY_ID_MIDDLE].repeat_key_down = key_repeat;

keypad[EN_KEY_ID_RIGHT].short_key_down = key_press;
keypad[EN_KEY_ID_RIGHT].short_key_up = key_release;
keypad[EN_KEY_ID_RIGHT].long_key_down = key_long_press;
keypad[EN_KEY_ID_RIGHT].repeat_key_down = key_repeat;

```

这里为了方便，回调函数的内容比较简单：

按下事件：亮红灯，

```

void key_press(void * arg)
{
    rgb_red_state(LIGHT_ON);
}

```

松开事件：蓝灯灭，

```

void key_release(void * arg)
{
    rgb_blue_state(LIGHT_OFF);
}

```

长按事件：红灯灭，

```

void key_long_press(void * arg)
{
    rgb_red_state(LIGHT_OFF);
}

```

重复（连发）事件：蓝灯闪烁。

```

void key_repeat(void * arg)
{
    static int state = 1;
    rgb_blue_state(state = !state);
}

```

12. 第二种：通过读取状态值的方式，打印当前按键的状态值。

```
/* keypad状态值 */
keypad_status_t key_value = EN_KEY_NONE;
printf("Please control keypad to get status!\n");

while (1)
{
    /* 读取keypad的状态值，如果没有事件，默认为0 */
    key_value = get_keypad_state();
    if (key_value != 0)
    {
        switch (key_value)
        {
            case EN_KEY_LEFT_DOWN:
                printf("KEY_LEFT_DOWN:%d \n", (uint16_t)key_value);
                break;
            case EN_KEY_LEFT_UP:
                printf("KEY_LEFT_UP:%d \n", (uint16_t)key_value);
                break;
            case EN_KEY_LEFT_LONG:
                printf("KEY_LEFT_LONG:%d \n", (uint16_t)key_value);
                break;
            case EN_KEY_LEFT_REPEAT:
                printf("KEY_LEFT_REPEAT:%d \n", (uint16_t)key_value);
                break;
```

```
            case EN_KEY_MIDDLE_DOWN:
                printf("KEY_MIDDLE_DOWN:%d \n", (uint16_t)key_value);
                break;
            case EN_KEY_MIDDLE_UP:
                printf("KEY_MIDDLE_UP:%d \n", (uint16_t)key_value);
                break;
            case EN_KEY_MIDDLE_LONG:
                printf("KEY_MIDDLE_LONG:%d \n", (uint16_t)key_value);
                break;
            case EN_KEY_MIDDLE_REPEAT:
                printf("KEY_MIDDLE_REPEAT:%d \n", (uint16_t)key_value);
                break;
```

```

        case EN_KEY_RIGHT_DOWN:
            printf("KEY_RIGHT_DOWN:%d \n", (uint16_t)key_value);
            break;
        case EN_KEY_RIGHT_UP:
            printf("KEY_RIGHT_UP:%d \n", (uint16_t)key_value);
            break;
        case EN_KEY_RIGHT_LONG:
            printf("KEY_RIGHT_LONG:%d \n", (uint16_t)key_value);
            break;
        case EN_KEY_RIGHT_REPEAT:
            printf("KEY_RIGHT_REPEAT:%d \n", (uint16_t)key_value);
            break;

        default:
            break;
    }
    msleep(1);
}

```

13. 编译调试，烧录运行

把本课程资料中的 keypad_event 复制到 SDK 中的 src 目录下，然后进入 build 目录，运行以下命令编译。

```
cmake .. -DPROJ=keypad_event -G "MinGW Makefiles"
```

```
make
```

```

[ 95%] Linking C executable keypad_event
Generating .bin file ...
[100%] Built target keypad_event
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build> 

```

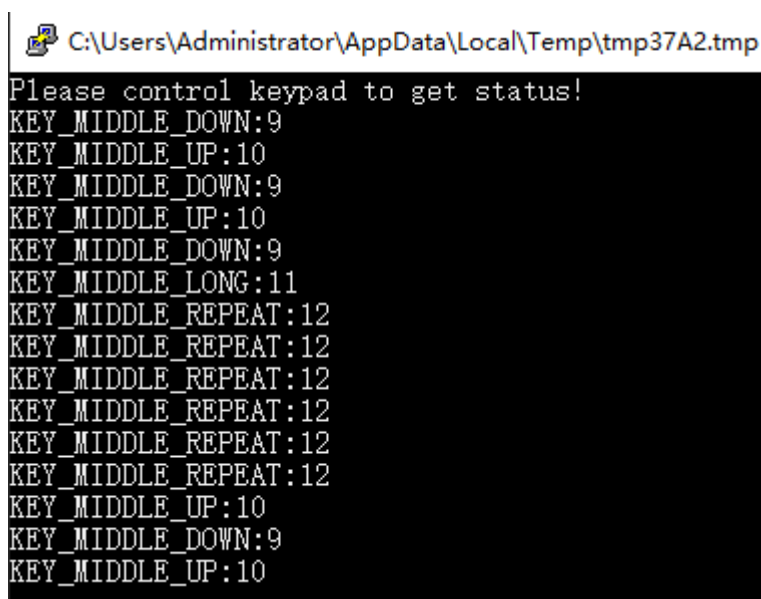
编译完成后，在 build 文件夹下会生成 keypad_event.bin 文件。

使用 type-C 数据线连接电脑与 K210 开发板，打开 kflash，选择对应的设备，再将程序固件烧录到 K210 开发板上。

五、实验现象

烧录完成固件后，系统会弹出一个终端界面，如果没有弹出终端界面的可以打开串口助手显示调试内容。

终端会打印 “Please control keypad to get status!” 提示操作 keypad 来读取 keypad 的状态；这里设置的 keypad 三个通道的功能都是一样的，所以以中间的那个按键来举例，当按下时，红灯亮，并且打印按下提示，如果立即松开则打印松开的提示，但是红灯不会熄灭，如果长按约 1 秒钟，红灯熄灭，并且打印长按提示，如果继续按住不放，就会进入重复打印的事件，功能是每 0.2 秒打印一次重复提示，蓝灯每 0.4 秒闪烁一次，此时松开蓝灯熄灭。



```
C:\Users\Administrator\AppData\Local\Temp\tmp37A2.tmp
Please control keypad to get status!
KEY_MIDDLE_DOWN:9
KEY_MIDDLE_UP:10
KEY_MIDDLE_DOWN:9
KEY_MIDDLE_UP:10
KEY_MIDDLE_DOWN:9
KEY_MIDDLE_LONG:11
KEY_MIDDLE_REPEAT:12
KEY_MIDDLE_REPEAT:12
KEY_MIDDLE_REPEAT:12
KEY_MIDDLE_REPEAT:12
KEY_MIDDLE_REPEAT:12
KEY_MIDDLE_REPEAT:12
KEY_MIDDLE_UP:10
KEY_MIDDLE_DOWN:9
KEY_MIDDLE_UP:10
```

六、实验总结

1. keypad 的内部原理其实是三个按键，只不过同一时间只能触发一个按键按下。
2. 通过定时器扫描 keypad 的方式，可以检测出 keypad 的事件，并且设置回调函数。

3. keypad 事件可以通过两种方式获取，第一种是设置回调函数，第二种是读取 keypad 的状态值。