综合

配置NRF52840

头文件

```
//延时头文件  #include "nrf_delay.h"
//GPIO头文件 #include "nrf_gpio.h"
///GPIOTE头文件 #include "nrf_gpiote.h"
//IIC头文件  #include "nrf_drv_twi.h"
```

1.GPIO API接口

```
//输入模式可以配置为没有pull，有上拉电阻，有下拉电阻，悬浮等4种状态
#include "nrf_gpio.h"//头文件

nrf_gpio_range_cfg_output(uint32_t pin_range_start, uint32_t pin_range_end);
//配置多个IO引脚为输出模式

nrf_gpio_range_cfg_input(uint32_t pin_range_start, uint32_t pin_range_end,
nrf_gpio_pin_pull_t pull_config)  //配置多个IO引脚为输入模式
```

```
nrf_gpio_cfg_output(uint32_t pin_number)  //设置一个引脚输出模式
//没有电平检测；驱动能力是最低等级；没有上下拉；没有启动input buffer；输出；

nrf_gpio_cfg_input(uint32_t pin_number, nrf_gpio_pin_pull_t pull_config)//设置一个引脚输入模式没有电平检测；驱动能力是最低等级；上下拉根据传入参数而定；启动input buffer；输入；

nrf_gpio_pin_set(uint32_t pin_number)  //设置某个引脚为高电平

nrf_gpio_pin_clear(uint32_t pin_number)//设置某个引脚为低电平

nrf_gpio_pin_toggle(uint32_t pin_number)//翻转某个引脚的电平

nrf_gpio_pin_write(uint32_t pin_number, uint32_t value) //写某个引脚的电平，可以写高，也可以写低

nrf_gpio_pin_read(uint32_t pin_number) //读取某个引脚的电平

nrf_gpio_cfg_default(uint32_t pin_number); //引脚释放调用接口，降低功耗
```

2.延时函数

```
#include "nrf_delay.h" //头文件
void nrf_delay_us(us_time);
void nrf_delay_ms(uint32_t ms_time);
```

3.中断

3.1中断引脚配置

```
/*使用GPIOTE中断需要先初始化GPIOTE*/
/*初始化函数nrf_drv_gpiote.h文件里，可以从引脚配置定义跳转*/
/*
NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(上升沿触发)
NRFX_GPIOTE_CONFIG_IN_SENSE_HITOLO(下降沿触发)
NRFX_GPIOTE_CONFIG_IN_SENSE_TOGGLE(上升沿下降沿均可触发)
*/
nrfx_gpiote_in_config_t button_config =NRFX_GPIOTE_CONFIG_IN_SENSE_HITOLO(true);
//创建引脚配置结构

button_config.pull = GPIO_PIN_CNF_PULL_Pullup;
//设置上拉

APP_ERROR_CHECK(nrfx_gpiote_in_init(PIN_1, &button_config, gpio_irq_handle));//绑定引脚1
APP_ERROR_CHECK(nrfx_gpiote_in_init(PIN_2, &button_config, gpio_irq_handle));//绑定引脚2
//按键绑定中断回调函数  gpio_irq_handle中断回调函数 如有多个引脚则需要绑定多次 回调函数可以同一个

nrfx_gpiote_in_event_enable(PIN_1, true);//使能引脚一
nrfx_gpiote_in_event_enable(PIN_2, true);//使能引脚二
//使能引脚 如有多个引脚则需要绑定多次
```

3.2中断函数配置

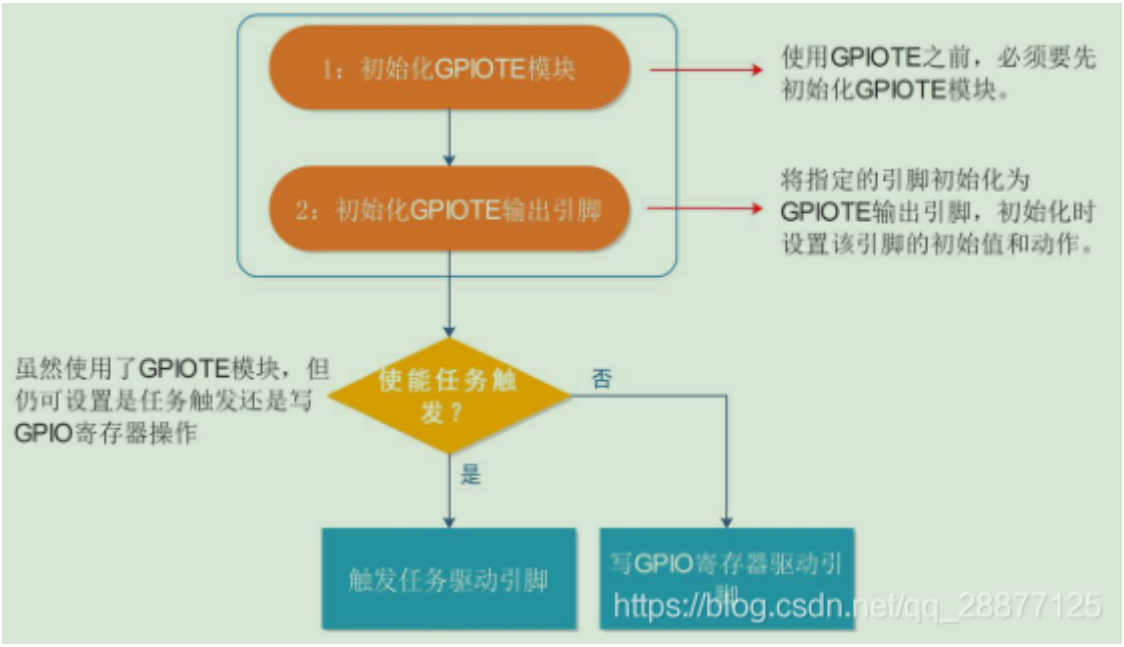
```
void BTN_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t
action)//BTN_pin_handler函数名可更改，参数不可更改
{
    /******
        code
    ********/

}
```

3.3注意事项

```
//初始化 nrfx_gpiote_init();
```

4.GPIOTE操作

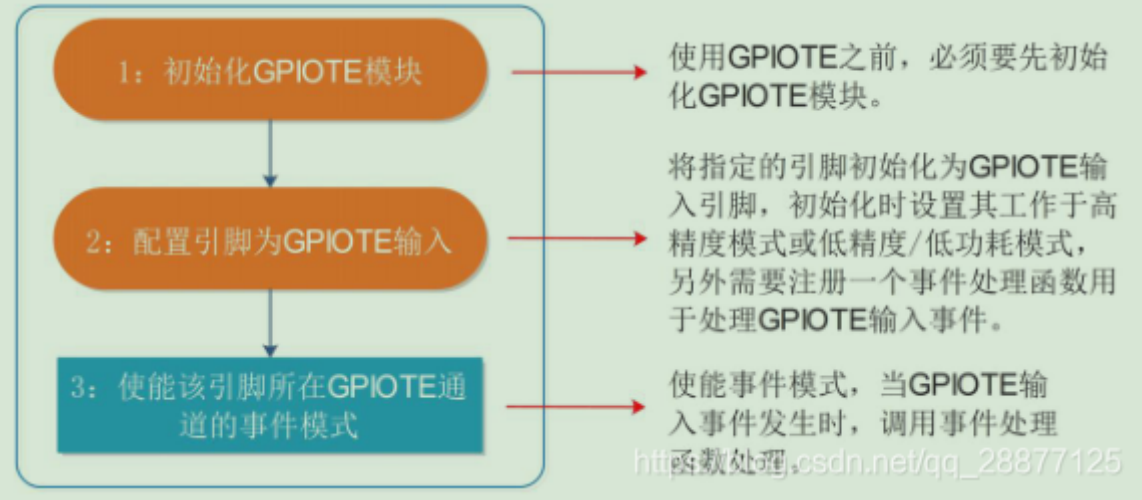
4.1GPIOTE输出




```
ret_code_t err_code;
//初始化 GPIOTE 程序模块
err_code = nrf_drv_gpiote_init();
//出错函数
APP_ERROR_CHECK(err_code);
//定义 GPIOTE 输出初始化结构体，并对其成员变量赋值
nrf_drv_gpiote_out_config_t config = GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
//初始化 GPIOTE 输出引脚  参数一：引脚 参数二：初始化的结构体
err_code = nrf_drv_gpiote_out_init(LED_1, &config);
//出错函数
APP_ERROR_CHECK(err_code);
//使能引脚 pin所在 GPIOTE 通道的任务触发
nrf_drv_gpiote_out_task_enable(pin);
```

```
while (true)
{
    //任务触发驱动引脚 pin 状态翻转，即指示灯 D1 翻转状态
    nrf_drv_gpiote_out_task_trigger(pin);
    //任务触发驱动引脚 pin 输出高电平，即指示灯 D1 熄灭
    //nrf_drv_gpiote_clr_task_trigger(pin);
    //任务触发驱动引脚 pin 输出低电平，即指示灯 D1 电亮
    //nrf_drv_gpiote_set_task_trigger(pin);
    nrf_delay_ms(150);
}
```

4.2GPIOTE输入




```
void in_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
nrf_drv_gpiote_out_toggle(LED_1);
}

ret_code_t err_code;
//初始化 GPIOTE 程序模块
err_code = nrf_drv_gpiote_init();
APP_ERROR_CHECK(err_code);
//定义 GPIOTE 输出初始化结构体，并对其成员变量赋值
nrf_drv_gpiote_out_config_t out_config =
    GPIOTE_CONFIG_OUT_SIMPLE(true);
//初始化 GPIOTE 输出引脚
err_code = nrf_drv_gpiote_out_init(LED_1, &out_config);
APP_ERROR_CHECK(err_code);

//高电平到低电平变化产生事件
nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO(true);

//低电平到高电平变化产生事件
//nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);

//任意电平变化产生事件
//nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_TOGGLE(true);

//开启 P0.13 引脚的上拉电阻
in_config.pull = NRF_GPIO_PIN_PULLUP;
//配置该引脚为 GPIOTE 输入
```

```
err_code = nrf_drv_gpiote_in_init(BUTTON_0, &in_config, in_pin_handler);
APP_ERROR_CHECK(err_code);
//使能该引脚所在 GPIOTE 通道的事件模式
nrf_drv_gpiote_in_event_enable(BUTTON_0, true);
while (true)
{
}
```

4.3注意事项

```
//初始化 nrfx_gpiote_init();
```

5.iic

5.1iic初始化

```
//配置twi结构体
const nrf_drv_twi_t m_twi_0 = NRF_DRV_TWI_INSTANCE(0);
void twi_init (void)
{
    ret_code_t err_code;
    // 初始化TWI配置结构体
    const nrf_drv_twi_config_t twi_config = {
       .scl                = ARDUINO_SCL_PIN,         // 配置TWI SCL引脚
       .sda                = ARDUINO_SDA_PIN,         // 配置TWI SDA引脚
       .frequency          = NRF_DRV_TWI_FREQ_400K,   // 配置TWI时钟频率
       .interrupt_priority = APP_IRQ_PRIORITY_HIGH,   // TWI中断优先级设置
       .clear_bus_init     = true                    //在初始化期间清除总线。
    };
    //初始化TWI
    //参数一：第几个TWI总线；初始化参数如下
    //const nrf_drv_twi_t m_twi_0 = NRF_DRV_TWI_INSTANCE(0);//第0个
    //const nrf_drv_twi_t m_twi_1 = NRF_DRV_TWI_INSTANCE(1);//第1个
    //参数二：TWI结构体
    //参数三：方法一：设置中断函数设置为非阻塞模式，方法二：设置为阻塞模式阻塞模式写NULL
    //参数四：设置NULL
    err_code = nrf_drv_twi_init(&m_twi, &twi_config, twi_handler, NULL);
    APP_ERROR_CHECK(err_code);

    //使能TWI
    nrf_drv_twi_enable(&m_twi);
}
```

5.2iic中断函数

```
//函数原型 void twi_handler(nrf_drv_twi_evt_t const *pEvent, void *pContext);

//例如：
void twi_handler(nrf_drv_twi_evt_t const *pEvent, void *pContext)
{
    switch (p_event->type)
      {

      case NRF_DRV_TWI_EVT_DONE://传送完成的事件
```

```
        if (global.freeRTOS_on)//判断RTOS是否打开
        {
            //释放互斥量
          xSemaphoreGiveFromISR(global.semaphone_handle_twi1_xfer_complete,
                                NULL);
        }
        break;
      case NRF_DRV_TWI_EVT_ADDRESS_NACK://发送地址后收到NACK
        if (global.freeRTOS_on)
        {
             //释放互斥量
          xSemaphoreGiveFromISR(global.semaphone_handle_twi1_xfer_complete,
                                NULL);
        }
        break;
      case NRF_DRV_TWI_EVT_DATA_NACK://发送一个数据字节后收到NACK
        if (global.freeRTOS_on)
        {
             //释放互斥量
          xSemaphoreGiveFromISR(global.semaphone_handle_twi1_xfer_complete,
                                NULL);
        }
        break;
      default:
        break;
      }
}
```

5.3iic发送数据

```
//官方函数原型
//参数一：实例  是指第几个TWI口；为初始化的m_twi_0
//参数二：特定从设备的地址
//参数三：指向发送缓冲区的指针 数据  需要写入的数据
//参数四：要发送的字节数 长度
//参数五：设置停止位 如果设置，则在总线上不生成停止条件 在传输成功完成之后(允许在下一次传输中重复启动)
ret_code_t  nrf_drv_twi_tx(nrf_drv_twi_t const * p_instance,
                           uint8_t               address,
                           uint8_t const *       p_data,
                           uint8_t               length,
                           bool                  no_stop);
```

5.3.1iic发送数据重新封装

```
//参数一：从设备地址
//参数二：寄存器地址
//参数三：数据
//参数四：数据长度
uint32_t twi0_write_buffer(uint8_t i2c_address, uint8_t reg, uint8_t *data,
                           uint16_t len)
{
  uint32_t ret = 0;
    //设置存储数据的buff
  uint8_t buffer[256] =
```

```
      {0};
    //存储信号量
  uint32_t ret_semaphore = 0;
  if (global.freeRTOS_on)//判断RTOS是否打开
  {
      //获取信号量避免设备使用
    ret_semaphore = xSemaphoreTake(global.semaphone_handle_twi0,
ms2ticks(1000));
  }
  if (ret_semaphore)
  {
    buffer[0] = reg;//存放寄存器地址
    memcpy(buffer + 1, data, len);//复制数据
       //通过m_twi_0从i2c_address发送buffer里面的数据，数据长度为len+1，且没有停止位
    ret = nrf_drv_twi_tx(&m_twi_0, i2c_address, buffer, len + 1, false);//调用官方
函数
    xSemaphoreTake(global.semaphone_handle_twi0_xfer_complete,
ms2ticks(1000));//获取信号量结束中断函数会释放信号量
    xSemaphoreGive(global.semaphone_handle_twi0);//释放信号量以便其他设备使用
  }
  return ret;
}
```

5.4iic读取数据

```
//官方函数原型
//参数一：实例   是指第几个TWI口；为初始化的m_twi_0
//参数二：特定从设备的地址
//参数三：指向接收缓冲区的指针
//参数四：要接收的字节数
ret_code_t nrf_drv_twi_rx(nrf_drv_twi_t const * p_instance,
                          uint8_t               address,
                          uint8_t *             p_data,
                          uint8_t               length);
```

5.4.1iic读取数据封装

```
//参数一：从设备地址
//参数二：寄存器地址
//参数三：数据
//参数四：数据长度
uint32_t twi0_read_buffer(uint8_t i2c_address, uint8_t reg, uint8_t *data,
                          uint16_t len)
{
  uint32_t ret = 0;
    //存储信号量的变量
  uint32_t ret_semaphore = 0;
  if (global.freeRTOS_on)//判断RTOS是否打开
  {
      //获取信号量，防止冲突
    ret_semaphore = xSemaphoreTake(global.semaphone_handle_twi0,
ms2ticks(1000));
  }
  if (ret_semaphore)
  {
      //先发送寄存器地址
```

```
    ret = nrf_drv_twi_tx(&m_twi_0, i2c_address, &reg, 1, true);
      //获取信号量判断是否发送成功
    ret_semaphore = xSemaphoreTake(global.semaphone_handle_twi0_xfer_complete,
                                   ms2ticks(1000));
    if (ret_semaphore)
    {
        //通过m_twi_0从i2c_address接收len个数据到data
      ret = nrf_drv_twi_rx(&m_twi_0, i2c_address, data, len);
      xSemaphoreTake(global.semaphone_handle_twi0_xfer_complete,
                     ms2ticks(1000));
    }
      //释放信号量以便其他设备使用
    xSemaphoreGive(global.semaphone_handle_twi0);
  }
  return ret;
}
```

6.SPI配置

6.1SPI添加官方配置文件

```
//nrf_drv_spi.c
//nrf_drv_spi.h
//nrf_drv_spis.c
//nrf_drv_spis.h
```

6.2SPI的引脚

```
//串行时钟线（SCLK）:           SPI1_SCK    NRF_GPIO_PIN_MAP(0, 28)
//主机输入/从机输出数据线 MISO: SPI1_MISO   NRF_GPIO_PIN_MAP(0, 3)
//主机输出/从机输入数据线 MOSI: SPI1_MOSI   NRF_GPIO_PIN_MAP(0, 2)
//低电平有效的从机选择线 SS
```

6.3SPI初始化

```
//配置SPI结构体
const nrf_drv_spi_t  m_spi_1 = NRF_DRV_SPI_INSTANCE(1);
void spi1_init(void)
{
    nrfx_err_t ret_code = 0;

    nrf_drv_spi_config_t spi1_config = NRF_DRV_SPI_DEFAULT_CONFIG;//使用SPI默认配
置，在nrf_drv_spi.h中有定义
    spi1_config.frequency = NRF_SPI_FREQ_8M;//设置SPI数据速率为8Mbps
    spi1_config.ss_pin = NRFX_SPI_PIN_NOT_USED;//不需要连接到引脚
    spi1_config.miso_pin = SPI1_MISO_PIN;//设置SPI的MISO引脚
    spi1_config.mosi_pin = SPI1_MOSI_PIN;//设置SPI的MOSI引脚
    spi1_config.sck_pin = SPI1_SCK_PIN;//设置SPI的SCK引脚
    spi1_config.mode = NRF_DRV_SPI_MODE_0;//设置为模式0，数据在第1个跳变沿（上升沿）采样
    //把结构体配置到SPI里
    ret_code = nrf_drv_spi_init(&m_spi_1, &spi1_config, spi1_event_handler,
                                NULL);
    (void)ret_code; //避免编译器警告
}
```

6.4SPI反初始化

```
//不使用时降低功耗
void spi1_uninit(void)
{
    //反初始化SPI官方函数调用
    nrf_drv_spi_uninit(&m_spi_1);
    //引脚释放调用接口降低功耗
    nrf_gpio_cfg_default(SPI1_MISO_PIN);
    nrf_gpio_cfg_default(SPI1_MOSI_PIN);
    nrf_gpio_cfg_default(SPI1_SCK_PIN);
}
```

6.5SPI中断函数

```
void spi1_event_handler(nrf_drv_spi_evt_t const *p_event, void *p_context)
{
    switch (p_event->type)
    {
    //判断是否传输完成
    case NRF_DRV_SPI_EVENT_DONE:
        // NRF_LOG_INFO("spi1_event_handler");
        if (global.freertos_on)//判断rtos是否开启
        {
            //传输完成处理的事情
            xSemaphoreGiveFromISR(global.semaphore_spi1_xfer_complete,
                                  NULL);
        }
        break;
    default:
        NRF_LOG_ERROR("[spi1_event_handler] error");
        break;
    }
}
```

6.6SPI的写函数

```
//参数一：写入的数据的地址
//参数二：要写入的数据长度
uint32_t spi1_write_buffer(const uint8_t *data, uint16_t len)
{
    //返回值
    uint8_t result = 0;
    if (global.freertos_on)//判断rtos是否打开
    {
        //获取信号量
        xSemaphoreTake(global.semaphore_spi1, portMAX_DELAY);
    }
    //用于设置TX发送传输的宏
    nrf_drv_spi_xfer_desc_t xfer_desc = NRF_DRV_SPI_XFER_TX(data, len);
    //调用执行宏的API接口可TX RX RTX
    result = nrf_drv_spi_xfer(&m_spi_1, &xfer_desc, 0);
    if (global.freertos_on)//判断rtos是否打开
    {
        //做需要的处理
```

```
        xSemaphoreTake(global.semaphore_spi1_xfer_complete, portMAX_DELAY);
        xSemaphoreGive(global.semaphore_spi1);
    }
    // NRF_LOG_INFO("spi1_write_buffer, result:%d",result);

    return result;//返回值判断是否写入成功
}
```

6.7SPI的读函数

```
//参数一：读取的数据的地址
//参数二：要读取的数据长度
uint32_t spi1_read_buffer(uint8_t *data, uint16_t len)
{
    uint8_t result = 0;
    if (global.freertos_on)
    {
        xSemaphoreTake(global.semaphore_spi1, portMAX_DELAY);
    }
    //用于设置RX发送传输的宏
    nrf_drv_spi_xfer_desc_t xfer_desc = NRF_DRV_SPI_XFER_RX(data, len);
    //调用执行宏的API接口可TX RX RTX
    result = nrf_drv_spi_xfer(&m_spi_1, &xfer_desc, 0);
    if (global.freertos_on)
    {
        xSemaphoreTake(global.semaphore_spi1_xfer_complete, portMAX_DELAY);
        xSemaphoreGive(global.semaphore_spi1);
    }
    // NRF_LOG_INFO("spi1_read_buffer, result:%d",result);
    return result;//返回值判断是否读取成功
}
```

6.8SPI的写入读取函数

```
//参数一：要发送数据的地址
//参数二：要存储数据的地址
//参数三：要发送(接收)数据的长度
uint32_t spi1_write_read_buffer(const uint8_t *tx_data, uint8_t *rx_data,
                                uint16_t len)
{
    uint8_t result = 0;
    if (global.freertos_on)//判断rtos是否打开
    {
        xSemaphoreTake(global.semaphore_spi1, portMAX_DELAY);
    }
    //用于设置TRX读写传输的宏
    nrf_drv_spi_xfer_desc_t xfer_desc =
        NRF_DRV_SPI_XFER_TRX(tx_data, len, rx_data, len);
    //调用执行宏的API接口可TX RX RTX
    result = nrf_drv_spi_xfer(&m_spi_1, &xfer_desc, 0);
    if (global.freertos_on)
    {
        xSemaphoreTake(global.semaphore_spi1_xfer_complete, portMAX_DELAY);
        xSemaphoreGive(global.semaphore_spi1);
    }
    // NRF_LOG_INFO("spi1_write_read_buffer, result:%d",result);
```

```
    return result;
```

7.看门狗

```
/*
看门狗定时器（WDT: Watchdog Timer）的作用是在发生软件故障时（如程序陷入死循环或者程序跑飞），强制复位单片机，让单片机重新运行程序。

看门狗定时器本质上是一个计数器，只不过这个计数器的作用是固定的，一旦计数值递增到设定的值（向上计数）或者计数值递减到0（向下计数），即“超时”时，看门狗定时器产生复位信号，复位系统。

程序正常运行时，会在看门狗定时器“超时”前清零计数值（向上计数）或重装计数值（向下计数），俗称“喂狗”，这样就保证了看门狗定时器永不会“超时”，而一旦程序运行出现故障，无法正常“喂狗”时，看门狗定时器最终会“超时”复位系统
*/
```

7.1独立看门狗

```

```

7.2窗口看门狗

7.2.1看门狗初始化

```
//设置看门狗通道
nrf_drv_wdt_channel_id m_channel_id;
```

```
void wdt_init(void)
{
    uint32_t err_code = NRF_SUCCESS;
    //配置看门狗
    nrf_drv_wdt_config_t config = NRF_DRV_WDT_DEAFULT_CONFIG;
    //重新加载值5000在芯片等同于5秒，五秒内不喂狗就复位
    /* 一个周期内 */
    config.reload_value = 5000;
    //初始化看门狗和中断 参数二：中断函数
    err_code = nrfx_wdt_init(&config, wdt_event_handler);
    APP_ERROR_CHECK(err_code);
    //分配看门狗通道
    err_code = nrf_drv_wdt_channel_alloc(&global.m_channel_id);
    APP_ERROR_CHECK(err_code);
    //看门狗使能
    nrf_drv_wdt_enable();
}
```

7.2.2看门狗中断

```
void wdt_event_handler(void)
{
    /*里面不用写东西*/
}
```

7.2.3喂狗

```
void wdt_feed(void)
{
    nrf_drv_wdt_channel_feed(global.m_channel_id);
}
```

7.2.4 使用方法

```
//创建一个任务函数优先级高点
//设置一定的时间喂狗 这里设置的是100ms绝对延时喂狗
void task_watch_dog(void *arg)
{
    uint32_t current_tick = xTaskGetTickCount();
    int32_t next_delay_tick = ms2ticks(100);
    for (;;)
    {
        vTaskDelayUntil(&current_tick, next_delay_tick);
        //喂狗
        wdt_feed();
    }
}
```

NRF52840外设配置

1.SC7a20

1.1TWI初始化

```
//TWI0初始化
twi0_init ();
```

1.2SC7a20初始化

1.2.1寄存器初始化

```
static uint8_t SC7A20_REG[10] = {0x2F, 0xbc, 0x00, 0x88, 0x15, 0x06, 0x06,
0x02};
static uint8_t INIT1_REG[3] = {0xFF,0x42, 5};

static void init(const sc7a20_dev *dev);
static uint32_t sc7a20_get_regs(const sc7a20_dev *dev, uint8_t reg_addr, uint8_t
*reg_data, uint8_t len);

static void init(const sc7a20_dev *dev)
{
    dev->write_buffer(dev->dev_addr, CTRL_REG1, &SC7A20_REG[0], 1); //ODR1 低功耗
模式(10 Hz)， 使能XYZ三轴， 正常模式
    dev->write_buffer(dev->dev_addr, CTRL_REG2, &SC7A20_REG[1], 1); // HPCLICK 开
启高通滤波器(滤掉地球G)(一定要开启，否则阈值要超过1G，而且动作也要超过1G)
    dev->write_buffer(dev->dev_addr,CTRL_REG3, &SC7A20_REG[2],1); //BDU块更新模式使
能，低字节数据在低地址,量程+/-2g，HR高精度模式
    dev->write_buffer(dev->dev_addr, CTRL_REG4, &SC7A20_REG[3], 1);
```

```
    // dev->write_buffer(dev->dev_addr, CLICK_CFG, &SC7A20_REG[4], 1);
    // dev->write_buffer(dev->dev_addr, CLICK_THS, &SC7A20_REG[5], 1);
    // dev->write_buffer(dev->dev_addr, TIME_LIMIT, &SC7A20_REG[6], 1);
    // dev->write_buffer(dev->dev_addr, TIME_LATENCY, &SC7A20_REG[7], 1);

    dev->write_buffer(dev->dev_addr, INT1_CFG, &INIT1_REG[0], 1); //中断 1 配置
    dev->write_buffer(dev->dev_addr, INT1_THS, &INIT1_REG[1], 1); //中断 1 阈值寄存
器
    dev->write_buffer(dev->dev_addr, INT1_DURATION, &INIT1_REG[2], 1);// 中断 1
持续时间
}
```

1.2.2检查设备中的空指针

```
static int8_t null_ptr_check(const sc7a20_dev *dev)
{
    int8_t rslt;
    if (dev == NULL)
    {
        /* 发现空指针 */
        rslt = SC7A20_E_NULL_PTR;
    }
    else
    {
        rslt = SC7A20_OK;
    }
    return rslt;
}
```

1.2.3给定寄存器地址读取数据

```
uint32_t sc7a20_get_regs(const sc7a20_dev *dev, uint8_t reg_addr, uint8_t
*reg_data, uint8_t len)
{
    int8_t rslt;

    rslt = null_ptr_check(dev);
    if ((rslt == SC7A20_OK) && (reg_data != NULL))
    {
        rslt = dev->read_buffer(dev->dev_addr, reg_addr, reg_data, len);
        NRF_LOG_ERROR("sc7a20_get_regs  rslt:%d", rslt);
        /*检查通信错误并使用内部错误码屏蔽  */
        if (rslt != SC7A20_OK)
        {
            rslt = SC7A20_E_COMM_FAIL;
        }
    }
    else
    {
        rslt = SC7A20_E_NULL_PTR;
    }
    return rslt;
}
```

1.2.4sc7a20初始化

```
int8_t sc7a20_init(sc7a20_dev *dev)
{
    int8_t rslt;
    uint8_t try_count = 5;
    //初始化判断五次初始化成功退出，以防没成功

    while (try_count)
    {
        rslt = sc7a20_get_regs(dev, SC7A20_WHOAMI_ADDR, &dev->chip_id, 1);
        if ((rslt == SC7A20_OK) &&
            (dev->chip_id == SC7A20_WHOAMI_VALUE))
        {
            init(dev);
            dev->init_flag = 1;
            break;
        }
        /* Wait for 10 ms */
        dev->delay_ms(10);
        --try_count;
    }
    return rslt;
}
```

1.3获取数据

```
            /*存放数据的共用体*/
/*
         typedef union
            {
                struct
                {
                    int16_t x;
                    int16_t y;
                    int16_t z;
                } val_xyz;
                uint8_t val[6];
            } acc_val_t;
            XYZ每个等于两个val里面的元素 好处不要转换
*/
            acc_val_t acc_val_now = {0};
            uint8_t src_click = 0;
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x28,
&acc_val_now.val[0], 1);
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x29,
&acc_val_now.val[1], 1);
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x2A,
&acc_val_now.val[2], 1);
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x2B,
&acc_val_now.val[3], 1);
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x2C,
&acc_val_now.val[4], 1);
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x2D,
&acc_val_now.val[5], 1);
```

```
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, 0x39, &src_click,
1);
            NRF_LOG_WARNING("SC7A20 CLICK_SRC:0x%02x, ACC VAL: %d %d %d",
src_click,
                            acc_val_now.val_xyz.x,
                            acc_val_now.val_xyz.y,
                            acc_val_now.val_xyz.z);
            switch (src_click)
            {
            case 0x5C:
                NRF_LOG_ERROR("Single click -> NZ");
                break;
            case 0x54:
                NRF_LOG_ERROR("Single click -> PZ");
                break;
            case 0x5A:
                NRF_LOG_ERROR("Single click -> NY");
                break;
            case 0x52:
                NRF_LOG_ERROR("Single click -> PY");
                break;
            case 0x59:
                NRF_LOG_ERROR("Single click -> NX");
                break;
            case 0x51:
                NRF_LOG_ERROR("Single click -> PX");
                break;
            }

            uint8_t int1_source = 0;
            global.sc7a20.read_buffer(global.sc7a20.dev_addr, INT1_SOURCE,
&int1_source, 1);
            NRF_LOG_DEBUG("INT1_SOURCE 0x%02x",int1_source);
            /*判断是哪个轴产生了中断*/
            if(int1_source & 0x40)
            {
                NRF_LOG_ERROR("INT1_SOURCE IA");
            }
            if(int1_source & 0x20)
            {
                NRF_LOG_ERROR("INT1_SOURCE ZH");
            }
            if(int1_source & 0x10)
            {
                NRF_LOG_ERROR("INT1_SOURCE ZL");
            }
            if(int1_source & 0x08)
            {
                NRF_LOG_ERROR("INT1_SOURCE YH");
            }
            if(int1_source & 0x04)
            {
                NRF_LOG_ERROR("INT1_SOURCE YL");
            }
            if(int1_source & 0x02)
            {
                NRF_LOG_ERROR("INT1_SOURCE XH");
            }
```

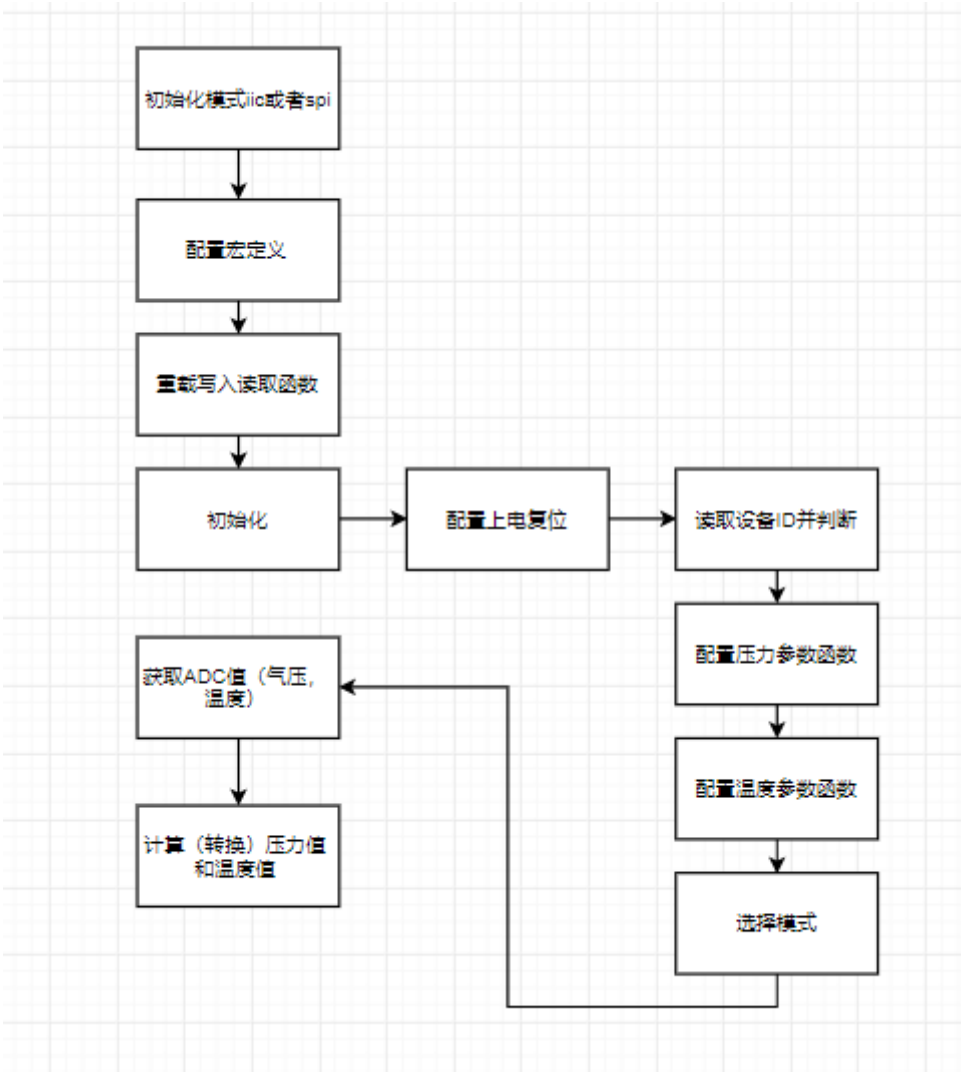
```
            if(int1_source & 0x01)
            {
                NRF_LOG_ERROR("INT1_SOURCE XL");
            }
```

1.4.注意事项

```
/*
1.配置了XYZ中断每次都会触发事件
2.AOI 位及 6D 位表示触发相应中断事件的规则为：若 AOI=0 为逻辑“或”，
    表示所配置好的所有中断事件只要有一个满足中断条件，就触发中断。若 AOI=1 为逻辑
    “与”，表示所配置好的所有中断事件必须所有都满足自身的中断条件，才能触发中断，相
    当于将所有中断进行了逻辑“与”操作。
3.可判断每次触发的事件是否有变化判断动静
4.中断用于判断是否按照自己需要的方向变化例如(单击双击自由落体)
*/
```

2.SPL06配置

流程图



2.1宏定义（寄存器和温度气压）

```
//气压测量速率(sample/sec),Background 模式使用
#define  PM_RATE_1          (0<<4)      //1 measurements pr. sec.
#define  PM_RATE_2          (1<<4)      //2 measurements pr. sec.
#define  PM_RATE_4          (2<<4)      //4 measurements pr. sec.
#define  PM_RATE_8          (3<<4)      //8 measurements pr. sec.
#define  PM_RATE_16         (4<<4)      //16 measurements pr. sec.
#define  PM_RATE_32         (5<<4)      //32 measurements pr. sec.
#define  PM_RATE_64         (6<<4)      //64 measurements pr. sec.
#define  PM_RATE_128        (7<<4)      //128 measurements pr. sec.

//气压重采样速率(times),Background 模式使用
#define PM_PRC_1            0       //Sigle         kP=524288   ,3.6ms
#define PM_PRC_2            1       //2 times       kP=1572864  ,5.2ms
#define PM_PRC_4            2       //4 times       kP=3670016  ,8.4ms
#define PM_PRC_8            3       //8 times       kP=7864320  ,14.8ms
#define PM_PRC_16           4       //16 times      kP=253952   ,27.6ms
#define PM_PRC_32           5       //32 times      kP=516096   ,53.2ms
#define PM_PRC_64           6       //64 times      kP=1040384  ,104.4ms
#define PM_PRC_128          7       //128 times     kP=2088960  ,206.8ms

//温度测量速率(sample/sec),Background 模式使用
#define  TMP_RATE_1         (0<<4)      //1 measurements pr. sec.
#define  TMP_RATE_2         (1<<4)      //2 measurements pr. sec.
#define  TMP_RATE_4         (2<<4)      //4 measurements pr. sec.
#define  TMP_RATE_8         (3<<4)      //8 measurements pr. sec.
#define  TMP_RATE_16        (4<<4)      //16 measurements pr. sec.
#define  TMP_RATE_32        (5<<4)      //32 measurements pr. sec.
#define  TMP_RATE_64        (6<<4)      //64 measurements pr. sec.
#define  TMP_RATE_128       (7<<4)      //128 measurements pr. sec.

//温度重采样速率(times),Background 模式使用
#define TMP_PRC_1           0       //Sigle
#define TMP_PRC_2           1       //2 times
#define TMP_PRC_4           2       //4 times
#define TMP_PRC_8           3       //8 times
#define TMP_PRC_16          4       //16 times
#define TMP_PRC_32          5       //32 times
#define TMP_PRC_64          6       //64 times
#define TMP_PRC_128         7       //128 times

//SPL06_MEAS_CFG
#define MEAS_COEF_RDY       0x80
#define MEAS_SENSOR_RDY     0x40        //传感器初始化完成
#define MEAS_TMP_RDY        0x20        //有新的温度数据
#define MEAS_PRS_RDY        0x10        //有新的气压数据

#define MEAS_CTRL_Standby               0x00    //空闲模式
#define MEAS_CTRL_PressMeasure          0x01    //单次气压测量
#define MEAS_CTRL_TempMeasure           0x02    //单次温度测量
#define MEAS_CTRL_ContinuousPress       0x05    //连续气压测量
#define MEAS_CTRL_ContinuousTemp        0x06    //连续温度测量
#define MEAS_CTRL_ContinuousPressTemp   0x07    //连续气压温度测量

//FIFO_STS
#define SPL06_FIFO_FULL     0x02
```

```
#define SPL06_FIFO_EMPTY    0x01

//INT_STS
#define SPL06_INT_FIFO_FULL     0x04
#define SPL06_INT_TMP           0x02
#define SPL06_INT_PRS           0x01

//CFG_REG
#define SPL06_CFG_T_SHIFT   0x08    //oversampling times>8时必须使用
#define SPL06_CFG_P_SHIFT   0x04

#define SP06_PSR_B2     0x00        //气压值
#define SP06_PSR_B1     0x01
#define SP06_PSR_B0     0x02
#define SP06_TMP_B2     0x03        //温度值
#define SP06_TMP_B1     0x04
#define SP06_TMP_B0     0x05

#define SP06_PSR_CFG    0x06        //气压测量配置
#define SP06_TMP_CFG    0x07        //温度测量配置
#define SP06_MEAS_CFG   0x08        //测量模式配置

#define SP06_CFG_REG    0x09
#define SP06_INT_STS    0x0A
#define SP06_FIFO_STS   0x0B

#define SP06_RESET      0x0C
#define SP06_ID         0x0D

#define SP06_COEF       0x10        //-0x21
#define SP06_COEF_SRCE  0x28

    /*spl06 校准参数初始化时需要使用*/
    typedef struct
    {

        int16_t C0;
        int16_t C1;
        int32_t C00;
        int32_t C10;
        int16_t C01;
        int16_t C11;
        int16_t C20;
        int16_t C21;
        int16_t C30;

        float kT;//温度补偿量表因素
        float kP;//气压补偿量表因素
    } T_SPL06_calibPara;

    /*! 传感器配置结构体*/
    typedef struct
    {
        uint8_t os_temp;
        uint8_t os_pres;
        uint8_t odr;
        uint8_t filter;
        uint8_t spi3w_en;
```

```
    }spl06_config;

    /*传感器状态结构*/
    struct spl06_status
    {
        uint8_t measuring;
        uint8_t im_update;
    };

    /*! @name Uncompensated data structure */
    struct spl06_uncomp_data
    {
        int32_t uncomp_temp;
        int32_t uncomp_press;
    };

    typedef uint32_t (*spl06_dev_write)(uint8_t, uint8_t, uint8_t *,
uint16_t);//重定义写函数
    typedef uint32_t (*spl06_dev_read)(uint8_t, uint8_t, uint8_t *, uint16_t);//
重定义读函数
    typedef void (*spl06_dev_delay)(uint32_t);//重定义延时函数

    /*! 设备结构 */
    typedef struct
    {
        uint8_t dev_addr;
        uint8_t init_flag;
        uint8_t chip_id;
        T_SPL06_calibPara calib_param;//校准参数结构体
        spl06_config conf;//传感器配置结构体
        spl06_dev_write write_buffer;//写重载函数
        spl06_dev_read read_buffer;//读重载函数
        spl06_dev_delay delay_ms;//延时重载函数
    } spl06_dev;
```

2.2重载函数

2.2.1重载写入函数

```
//参数一：存放设备数据结构体
//参数二：寄存器地址指针 因为可能不止写入一个寄存器
//参数三：要写入的数据指针
//参数四：要写入的寄存器个数
uint32_t spl06_set_regs(const spl06_dev *dev, uint8_t *reg_addr, const uint8_t
*reg_data, uint8_t len)
{
    int8_t rslt; //函数返回值
    uint8_t temp_buff[8]; /* 通常不要写超过4个寄存器*/
    uint16_t temp_len;//临时长度
    if (len > 4)//判断是否超过四个
    {
        len = 4;
    }
    //判断是否为空指针
    rslt = null_ptr_check(dev);
    //判断是否为空指针 && 判断寄存器地址是否为空 && 写入数据是否为空
    if ((rslt == SPL06_OK) && (reg_addr != NULL) && (reg_data != NULL))
```

```
    {
        if (len != 0)//判断长度是否为0
        {
            temp_buff[0] = reg_data[0];

            /* 写模式 */
            if (len > 1)
            {
                /* Interleave register address w.r.t data for burst write*/
                interleave_data(reg_addr, temp_buff, reg_data, len);
                temp_len = ((len * 2) - 1);
            }
            else
            {
                temp_len = len;
            }
            //写入数据
            rslt = dev->write_buffer(dev->dev_addr, reg_addr[0], temp_buff,
temp_len);

            /*检查通信错误并使用内部错误码屏蔽  */
            if (rslt != SPL06_OK)
            {
                rslt = SPL06_E_COMM_FAIL;
            }
        }
        else
        {
            rslt = SPL06_E_INVALID_LEN;
        }
    }
    else
    {
        rslt = SPL06_E_NULL_PTR;
    }

    return rslt;
}
```

2.2.2读取数据重载函数

```
//参数一：存放设备数据结构体
//参数二：寄存器地址
//参数三：存放读取到的数据变量的地址
//参数四：读取的长度
uint32_t spl06_get_regs(const spl06_dev *dev, uint8_t reg_addr, uint8_t
*reg_data, uint8_t len)
{
    int8_t rslt;//返回值
    //判断是否为空
    rslt = null_ptr_check(dev);
    //判断是否为空 && 存放数据的指针是否为空
    if ((rslt == SPL06_OK) && (reg_data != NULL))
    {
        //读取数据
        rslt = dev->read_buffer(dev->dev_addr, reg_addr, reg_data, len);
        /* 读取失败 检查通信错误并使用内部错误码屏蔽*/
```

```
        if (rslt != SPL06_OK)
        {
            rslt = SPL06_E_COMM_FAIL;
        }
    }
    else
    {
        /* 检查通信错误并使用内部错误码屏蔽*/
        rslt = SPL06_E_NULL_PTR;
    }
    return rslt;
}
```

2.3初始化

2.3.1上电复位

```
int8_t spl06_soft_reset(const spl06_dev *dev)
{
    int8_t rslt;//返回值
    uint8_t reset_reg = SPL06_RESET;//存放地址
    uint8_t reg_val = 0x89;//存放要写入的数据
    rslt = null_ptr_check(dev);//判断结构体是否为空指针
    if (rslt == SPL06_OK)//如果等于零则不是空指针
    {
        //将上电复位写入寄存器地址中
        rslt = spl06_set_regs(dev, &reset_reg, &reg_val, 1);//重定义后的往寄存器写函
数
        /*根据数据表，启动时间为2毫秒延时两毫秒 */
        dev->delay_ms(2);
    }
    return rslt;
}
```

2.3.2读取设备ID并判断

```
int8_t spl06_grt_id(const spl06_dev *dev)
{
    int8_t rslt;//返回值
    uint8_t coef[18];//存储初始化所用数据
    uint8_t try_count = 5;//尝试初始化五次以防出错
    rslt = null_ptr_check(dev);//判断是否是空指针
    if (rslt == SPL06_OK)//不是空指针
    {
        while (try_count)//循环尝试初始化五次机会
        {
            //上电复位
            rslt = spl06_soft_reset(dev);
            //复位后系数准备好需要至少40ms  给100ms
            dev->delay_ms(100);
            //判断是否写入成功
            if (rslt == SPL06_OK)
            {
                NRF_LOG_ERROR("spl06_init reset ok");
```

```
            }
            //读取id ID 正常情况是0x10
            rslt = spl06_get_regs(dev, SPL06_ID, &dev->chip_id, 1); //重载后写寄存
器函数
            /* 判断是否读取成功和id是否为0x10 */
            if ((rslt == SPL06_OK) &&
                        (dev->chip_id == 0x10))
            {
                //初始化
                spl06_get_regs(dev, SPL06_COEF, &coef, 18);
                dev->calib_param.c0 = ((int16_t)coef[0] << 4) + ((coef[1] &
0xF0) >> 4);
                dev->calib_param.c0 = (dev->calib_param.c0 & 0x0800) ? (0xF000 |
dev->calib_param.c0) : dev->calib_param.c0;
                dev->calib_param.c1 = ((int16_t)(coef[1] & 0x0F) << 8) +
coef[2];
                dev->calib_param.c1 = (dev->calib_param.c1 & 0x0800) ? (0xF000 |
dev->calib_param.c1) : dev->calib_param.c1;
                dev->calib_param.c00 = ((int32_t)coef[3] << 12) +
((int32_t)coef[4] << 4) + (coef[5] >> 4);
                dev->calib_param.c00 = (dev->calib_param.c00 & 0x080000) ?
(0xFFF00000 | dev->calib_param.c00)

: dev->calib_param.c00;
                dev->calib_param.c10 = ((int32_t)(coef[5] & 0x0F) << 16) +
((int32_t)coef[6] << 8) + coef[7];
                dev->calib_param.c10 = (dev->calib_param.c10 & 0x080000) ?
(0xFFF00000 | dev->calib_param.c10)

: dev->calib_param.c10;
                dev->calib_param.c01 = ((int16_t)coef[8] << 8) + coef[9];
                dev->calib_param.c11 = ((int16_t)coef[10] << 8) + coef[11];
                dev->calib_param.c20 = ((int16_t)coef[12] << 8) + coef[13];
                dev->calib_param.c21 = ((int16_t)coef[14] << 8) + coef[15];
                dev->calib_param.c30 = ((int16_t)coef[16] << 8) + coef[17];

                SPL06_Config_Pressure(dev, PM_RATE_4, PM_PRC_32);//配置压力参数函数
参照2.3.2.1
                SPL06_Config_Temperature(dev, PM_RATE_4, TMP_PRC_8);//配置温度参数
函数 参照2.3.2.2
                spl06_set_ctrl_mode(dev, MEAS_CTRL_ContinuousPressTemp); //启动连
续的气压温度测量函数

                dev->init_flag = 1;//初始化标志位置1
                break;
            }

            /*没初始化成功等待十毫秒*/
            dev->delay_ms(10);
            --try_count;
        }

        /*判断try_count是否为0；是则芯片id检查失败，超时 */
        if (!try_count)
        {
            rslt = SPL06_E_DEV_NOT_FOUND;
        }
    }
```

```
    return rslt;
}
```

2.3.2.1配置压力参数函数

```
//参数一：存放设备数据结构体
//参数二：气压测量速率
//参数三：气压重采样速率
void SPL06_Config_Pressure(spl06_dev *dev, uint8_t rate, uint8_t oversampling)
{
    uint8_t temp;//临时数据
    uint8_t psr_cfg = SPL06_PSR_CFG;//气压测量配置寄存器地址 0x06
    uint8_t cfg_reg = SPL06_CFG_REG;//FIFO配置(CFG_REG)寄存器地址 0x09
    //判断气压采样速率是多少  设置气压补偿量表因素
    //注：搭配好的不可轻易更改
    switch (oversampling)
    {
    case PM_PRC_1:
        dev->calib_param.kP = 524288;
        break;
    case PM_PRC_2:
        dev->calib_param.kP = 1572864;
        break;
    case PM_PRC_4:
        dev->calib_param.kP = 3670016;
        break;
    case PM_PRC_8:
        dev->calib_param.kP = 7864320;
        break;
    case PM_PRC_16:
        dev->calib_param.kP = 253952;
        break;
    case PM_PRC_32:
        dev->calib_param.kP = 516096;
        break;
    case PM_PRC_64:
        dev->calib_param.kP = 1040384;
        break;
    case PM_PRC_128:
        dev->calib_param.kP = 2088960;
        break;
    }

    rate = rate | oversampling; //气压速率 | 气压重采样速率
    spl06_set_regs(dev, &psr_cfg, &rate, 1);//写入压力配置寄存器 压力测量速率的配置
    //暂未知 但写
    if (oversampling > PM_PRC_8)
    {
        spl06_get_regs(dev, cfg_reg, &temp, 1);
        temp = temp | SPL06_CFG_P_SHIFT;
        spl06_set_regs(dev, &cfg_reg, &temp, 1);
    }
}
```

2.3.2.2配置温度参数函数

```
//参数一：存放设备数据结构体
//参数二：气压测量速率
//参数三：温度重采样速率
void SPL06_Config_Temperature(spl06_dev *dev, uint8_t rate, uint8_t
oversampling)
{
    uint8_t temp; //临时数据
    uint8_t tmp_cfg = SPL06_TMP_CFG; //温度测量配置寄存器 0x07
    uint8_t cfg_reg = SPL06_CFG_REG; //FIFO配置(CFG_REG)寄存器地址 0x09
    //判断温度重采样速率是多少  设置温度补偿量表因素
    //注：搭配好的不可轻易更改
    switch (oversampling)
    {
    case TMP_PRC_1:
        dev->calib_param.kT = 524288;
        break;
    case TMP_PRC_2:
        dev->calib_param.kT = 1572864;
        break;
    case TMP_PRC_4:
        dev->calib_param.kT = 3670016;
        break;
    case TMP_PRC_8:
        dev->calib_param.kT = 7864320;
        break;
    case TMP_PRC_16:
        dev->calib_param.kT = 253952;
        break;
    case TMP_PRC_32:
        dev->calib_param.kT = 516096;
        break;
    case TMP_PRC_64:
        dev->calib_param.kT = 1040384;
        break;
    case TMP_PRC_128:
        dev->calib_param.kT = 2088960;
        break;
    }
    //气压速率 | 气压重采样速率 | 0x80
    rate = rate | oversampling | 0x80;
    spl06_set_regs(dev, &tmp_cfg, &rate, 1); //写入温度配置寄存器 温度测量速率的配置 温
度每秒128次测量一次
    //暂时未知但写
    if (oversampling > TMP_PRC_8)
    {
        spl06_get_regs(dev, cfg_reg, &temp, 1);
        temp = temp | SPL06_CFG_T_SHIFT;
        spl06_set_regs(dev, &cfg_reg, &temp, 1);
    }
}
```

2.3.2.3选择模式函数

```
//参数一：存放设备数据结构体
//参数二：模式选择 0-空闲模式  1-单次气压测量  2-单次温度测量  5-连续气压测量  6-连续温度测量
7-连续气压温度测量
void spl06_set_ctrl_mode(spl06_dev *dev, uint8_t mode)
{
    uint8_t reg = SPL06_MEAS_CFG;
    spl06_set_regs(dev, &reg, &mode, 1);
}
```

2.3.2.4写入多个寄存器使用的函数

```
//参数一：多个寄存器地址的指针
//参数二：存放寄存器地址和数据的指针
//参数三：要写入的多个数据的指针
//参数四：要写入几个数据的长度
static void interleave_data(const uint8_t *reg_addr, uint8_t *temp_buff, const
uint8_t *reg_data, uint8_t len)
{
    uint8_t index;

    for (index = 1; index < len; index++)
    {
        temp_buff[(index * 2) - 1] = reg_addr[index];
        temp_buff[index * 2] = reg_data[index];
    }
}
```

2.3.2.5检测空指针函数

```
static int8_t null_ptr_check(const spl06_dev *dev)
{
    int8_t rslt;
    if (dev == NULL)
    {
        /* NULL指针错误宏定义*/
        rslt = SPL06_E_NULL_PTR;
    }
    else
    {
        //不是空
        rslt = SPL06_OK;
    }
    return rslt;
}
```

2.4获取ADC值

2.4.1获取气压ADC值

```
int32_t SPL06_Get_Pressure_Adc(spl06_dev *dev)
{
    //ADC的值由三个组成
    uint8_t buf[3];
    //ADC的实际值
    int32_t adc;
    //读取三个ADC的三值
    spl06_get_regs(dev, SPL06_PSR_B2, buf, 3);
    //组合成ADC
    adc = (int32_t)(buf[0] << 16) + (int32_t)(buf[1] << 8) + buf[2];
    adc = (adc & 0x800000) ? (0xFF000000 | adc) : adc;
    return adc;
}
```

2.4.2获取温度ADC值

```
int32_t SPL06_Get_Temperature_Adc(spl06_dev *dev)
{
    //ADC的值由三个组成
    uint8_t buf[3];
    //ADC的实际值
    int32_t adc;
    //读取三个ADC的三值
    spl06_get_regs(dev, SPL06_TMP_B2, buf, 3);
     //组合成ADC
    adc = (int32_t)(buf[0] << 16) + (int32_t)(buf[1] << 8) + buf[2];
    adc = (adc & 0x800000) ? (0xFF000000 | adc) : adc;
    return adc;
}
```

2.5计算压力值和温度值

```
float ReadSPL06_Pressure(spl06_dev *dev)
{
    //判断初始化标志位是否为真 判断是否初始化
    if (dev->init_flag)
    {
        float Traw_src, Praw_src;
        float Temp;
        float qua2, qua3;
        int32_t raw_temp, raw_press;//临时存放 ADC值

        raw_temp = SPL06_Get_Temperature_Adc(dev);
        raw_press = SPL06_Get_Pressure_Adc(dev);

        Traw_src = raw_temp / dev->calib_param.kT;
        Praw_src = raw_press / dev->calib_param.kP;

        //计算温度 公式直接套用
        Temp = 0.5f * dev->calib_param.C0 + Traw_src * dev->calib_param.C1;
        dev->current_temperature_val = (uint8_t)Temp;
```

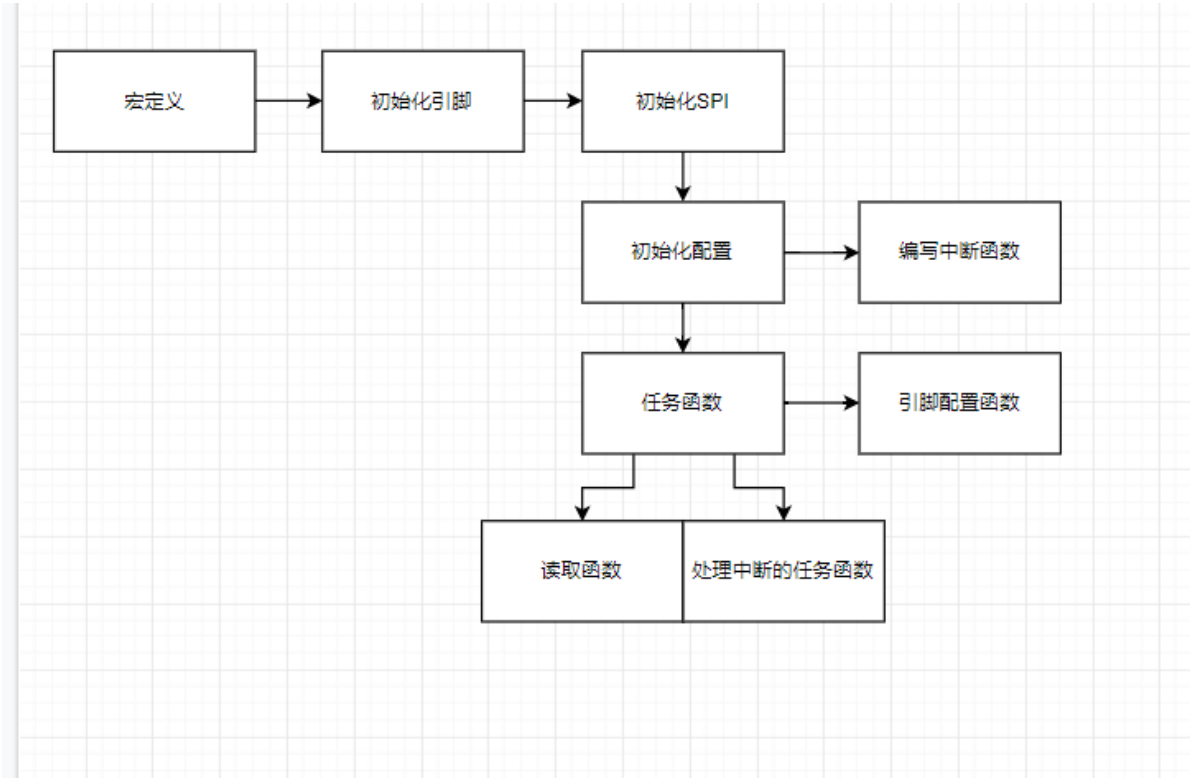
```
        //计算气压
        qua2 = dev->calib_param.C10 + Praw_src * (dev->calib_param.C20 +
Praw_src * dev->calib_param.C30);
        qua3 = Traw_src * Praw_src * (dev->calib_param.C11 + Praw_src * dev-
>calib_param.C21);
        //原dev->current_pressure_val = (uint16_t)((dev->calib_param.C00 +
Praw_src * qua2 + Traw_src * dev->calib_param.C01 + qua3) 1hpa=100Pa  从输出hPa 百
帕 除100 得 pa
        dev->current_pressure_val = (uint16_t)((dev->calib_param.C00 + Praw_src
* qua2 + Traw_src * dev->calib_param.C01 + qua3) / 100);
        return dev->current_pressure_val;
    }
    else
    {
        //返回错误
        return SPL06_E_DEV_NOT_FOUND;
    }
}
```

3.LORA(SX1268)

需要添加的文件and宏定义

```
sx126x.c
sx126x.h
```

流程图



3.1引脚

```
/*
//串行时钟线（SCLK）:           SPI1_SCK  NRF_GPIO_PIN_MAP(0, 28)
//主机输入/从机输出数据线 MISO: SPI1_MISO  NRF_GPIO_PIN_MAP(0, 3)
//主机输出/从机输入数据线 MOSI: SPI1_MOSI  NRF_GPIO_PIN_MAP(0, 2)
//低电平有效的从机选择线 SS
//SX1268_PWRCTRL_PIN NRF_GPIO_PIN_MAP(1, 9)
//SX1268的复位引脚:     SX1268_RESET_PIN NRF_GPIO_PIN_MAP(1, 15)  //低电平有效
//SX1268 占线指示器引脚:    SX1268_BUSY_PIN NRF_GPIO_PIN_MAP(1, 14)
//SX1268多用途数字IO引脚:   SX1268_DIO1_PIN NRF_GPIO_PIN_MAP(1, 13)
//SX1268多功能数字输入输出/射频开关控制: SX1268_DIO2_PIN NRF_GPIO_PIN_MAP(1, 12)
//SX1268的片选引脚:         SX1268_CS_PIN NRF_GPIO_PIN_MAP(0, 29)
//SX1268天线开关引脚:       SX1268_ANT_SW NRF_GPIO_PIN_MAP(0, 30) //为高电平打开天线，
四脚控制收发，低电平，天线关闭
*/
```

3.1.1网关物联网引脚配置

```
/*
SPI1 Lora sx1268 use
#define SPI1_SCK_PIN                    NRF_GPIO_PIN_MAP(0,04)
#define SPI1_MISO_PIN                   NRF_GPIO_PIN_MAP(0,27)
#define SPI1_MOSI_PIN                   NRF_GPIO_PIN_MAP(0,26)

// Lora sx1268
#define SX1268_1_PWRCTRL_PIN              NRF_GPIO_PIN_MAP(1,9)
#define SX1268_1_RESET_PIN                NRF_GPIO_PIN_MAP(1,8)
#define SX1268_1_BUSY_PIN                 NRF_GPIO_PIN_MAP(0,5)

#define SX1268_1_DIO1_PIN                 NRF_GPIO_PIN_MAP(0,7)
#define SX1268_1_DIO2_PIN                 NRF_GPIO_PIN_MAP(0,6)
#define SX1268_1_CS_PIN                   NRF_GPIO_PIN_MAP(0,31)
#define SX1268_1_TX_EN_PIN                NRF_GPIO_PIN_MAP(0,0)
#define SX1268_1_RX_EN_PIN                NRF_GPIO_PIN_MAP(0,0)
*/
```

3.1.2边缘物联网引脚配置

```
/*
#define SPI1_SCK_PIN                    NRF_GPIO_PIN_MAP(0,26)
#define SPI1_MISO_PIN                   NRF_GPIO_PIN_MAP(0,27)
#define SPI1_MOSI_PIN                   NRF_GPIO_PIN_MAP(0,4)

// Lora sx1268
#define SX1268_1_PWRCTRL_PIN              NRF_GPIO_PIN_MAP(0,12)
#define SX1268_1_RESET_PIN                NRF_GPIO_PIN_MAP(0,7)
#define SX1268_1_BUSY_PIN                 NRF_GPIO_PIN_MAP(0,31)

#define SX1268_1_DIO1_PIN                 NRF_GPIO_PIN_MAP(0,8)
#define SX1268_1_DIO2_PIN                 NRF_GPIO_PIN_MAP(1,8)
#define SX1268_1_CS_PIN                   NRF_GPIO_PIN_MAP(0,5)
*/
```

3.2初始化

3.2.1引脚初始化

```
void SX1268_gpio_init(void)
{
    //不一定有 功率控制
    nrf_gpio_cfg_output(SX1268_PWRCTRL_PIN);
    nrf_gpio_pin_set(SX1268_PWRCTRL_PIN);

    //配置复位引脚为输出，因为SX1268的复位是输入，这里配置的是nrf所以是输出到SX1268
    nrf_gpio_cfg_output(SX1268_RESET_PIN);
    //输出高电平不复位
    nrf_gpio_pin_set(SX1268_RESET_PIN);

    //设置片选引脚
    nrf_gpio_cfg_output(SX1268_CS_PIN);
    nrf_gpio_pin_set(SX1268_CS_PIN);

    //设置天线
    nrf_gpio_cfg_output(SX1268_ANT_SW);
    //低电平关闭天线
    nrf_gpio_pin_clear(SX1268_ANT_SW);

    //占线器配置输入使能上拉
    nrf_gpio_cfg_input(SX1268_BUSY_PIN, NRF_GPIO_PIN_PULLUP);

    //配置dio1的中断 发送和接受产生的中断
    nrfx_gpiote_in_config_t dio1_config =
NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
    dio1_config.pull = GPIO_PIN_CNF_PULL_Pulldown;
    APP_ERROR_CHECK(nrfx_gpiote_in_init(SX1268_DIO1_PIN, &dio1_config,
loar_gpio_irq_handle));
    nrfx_gpiote_in_event_enable(SX1268_DIO1_PIN, true);

    //配置dio2的中断
    nrfx_gpiote_in_config_t dio2_config =
NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
    dio2_config.pull = GPIO_PIN_CNF_PULL_Pulldown;
    APP_ERROR_CHECK(nrfx_gpiote_in_init(SX1268_DIO2_PIN, &dio2_config,
loar_gpio_irq_handle));
    nrfx_gpiote_in_event_enable(SX1268_DIO2_PIN, true);
}
```

3.2.2配置初始化

```
void sx1268_user_init(void)
{
    NRF_LOG_WARNING("Begin to init SX1268 .");
    SX1268.device_id = SX1268_DEVICE_ID;                 //ID
    SX1268.read_buffer = spi1_read_buffer;              // SPI读取函数，要求是同步操作
    SX1268.write_buffer = spi1_write_buffer;            // SPI写入函数，要求是同步操作
    SX1268.write_read_buffer = spi1_write_read_buffer; // SPI写入读取函数，要求是同步
操作
    SX1268.delay_ms = platform_delay_ms;                // 延迟函数
```

```
    SX1268.wait_on_busy = sx126x_wait_on_busy;          // 检查SX126x BUSY引脚是否已经为低电平，见下方
    SX1268.cs_high = SX1268_cs_high;                    // SX1268 SPI CS引脚设置为高电平函数
    SX1268.cs_low = SX1268_cs_low;                      // SX1268 SPI CS引脚设置为低电平函数
    SX1268.rst_high = SX1268_rst_high;                  // SX1268 RST引脚设置为高电平函数 复位
    SX1268.rst_low = SX1268_rst_low;                    // SX1268 RST引脚设置为低电平函数 复位
    SX1268.ant_sw_tx = SX1268_ant_sw_tx;                // SX1268 控制外部射频开关为发射  天线
    SX1268.ant_sw_rx = SX1268_ant_sw_rx;                // SX1268 控制外部射频开关为接收  天线

    SX1268.tx_power = global.SX1268_TX_POWER; // 发射功率 2~22
    SX1268.symble_timeout = 5;                  // 超时时间 如果是连续接收初始化时超时时间会设置为0 非必选
    SX1268.rx_continuous = true;                // 是否连续接收
    SX1268.dio2_as_rf_switch_ctrl_flag = false; // 是否将dio2作为射频开关切换控制

    SX1268.modulation_params.PacketType = SX126X_PACKET_TYPE_LORA;          // 传输类型为LORA
    SX1268.modulation_params.Params.LoRa.Bandwidth = SX126X_LORA_BW_250;  // 带宽设置为250KHz
    SX1268.modulation_params.Params.LoRa.CodingRate = SX126X_LORA_CR_4_5; // 表示LoRa包类型的编码速率值 CR=4/5
    SX1268.modulation_params.Params.LoRa.LowDatarateOptimize = 0;          // 低速率优化开关
    SX1268.modulation_params.Params.LoRa.SpreadingFactor = LORA_SF7;       // SF=7

    SX1268.packet_params.PacketType = SX126X_PACKET_TYPE_LORA;
  // 数据包类型lora
    //如果头是显式的，它将在GFSK包中传输。 如果标头是隐式的，它将不会被传输
    //SX126X_RADIO_PACKET_FIXED_LENGTH显示的    SX126X_RADIO_PACKET_VARIABLE_LENGTH隐示的
    SX1268.packet_params.Params.LoRa.HeaderType = SX126X_LORA_PACKET_FIXED_LENGTH; // 包含Header，Header中带有数据长度 数据包的两头已知包中没头
    //表示LoRa数据包类型的CRC模式
    //SX126X_LORA_CRC_ON CRC打开  SX126X_LORA_CRC_OFF  CRC不打开 校验打开可检测数据是否正确
    SX1268.packet_params.Params.LoRa.CrcMode = SX126X_LORA_CRC_ON;
  // CRC校验打开
    //数据有效负载的长度
    SX1268.packet_params.Params.LoRa.PayloadLength = 5;
  // PayLoad长度
    //表示LoRa包类型的IQ模式
    SX1268.packet_params.Params.LoRa.InvertIQ = SX126X_LORA_IQ_NORMAL;
  // IQ 配置
    //前导码的长度默认是12个符号长度，LoRaWAN中使用8个符号长度
    SX1268.packet_params.Params.LoRa.PreambleLength = 8;
  // 前导码长度

    vTaskDelay(ms2ticks(100));

    nrf_gpio_pin_clear(LED_BLUE);
```

```
    //调用官方初始化函数
    if (sx126x_init(&SX1268))
    {
        NRF_LOG_WARNING("SX1268 init success.");
        nrf_gpio_pin_clear(LED_GREEN);
        global.lora_init = 1;
    }
    else
    {
        NRF_LOG_WARNING("SX1268 init fail.");
        nrf_gpio_pin_set(LED_BLUE);
    }
}
```

3.3中断回调函数

```
//中断回调函数
void loar_gpio_irq_handle(nrfx_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
    msg_t msg;
    //存放接收或发送消息的时间
    msg.tick = xTaskGetTickCountFromISR();
    if (pin == SX1268_DIO1_PIN)
    {
        NRF_LOG_INFO("SX1268_DIO1_PIN");
        //往队列里发消息说明产生了这个dio1中断
        xQueueSendFromISR(global.queue_SX1268_irq_msg, &msg, NULL);
    }
    if (pin == SX1268_DIO2_PIN)
    {
        NRF_LOG_INFO("SX1268_DIO2_PIN");
        //往队列里发消息说明产生了这个dio2中断
        xQueueSendFromISR(global.queue_SX1268_irq_msg, &msg, NULL);
    }
}
```

3.4引脚配置函数

```
//判断是否为低电平从而判断是否空闲
void sx126x_wait_on_busy()
{
    while (nrf_gpio_pin_read(SX1268_BUSY_PIN) == 1)
    {
        // NRF_LOG_INFO("sx126x_wait_on_busy");
        NRF_LOG_FLUSH();
    }
}
//CS引脚拉高
void SX1268_cs_high()
{
    nrf_gpio_pin_set(SX1268_CS_PIN);
    // NRF_LOG_INFO("SX1268_cs_high");
}
//CS引脚拉低
void SX1268_cs_low()
{
```

```
    nrf_gpio_pin_clear(SX1268_CS_PIN);
    // NRF_LOG_INFO("SX1268_cs_low");
}
//复位引脚拉高
void SX1268_rst_high()
{
    nrf_gpio_pin_set(SX1268_RESET_PIN);
    // NRF_LOG_INFO("SX1268_rst_high");
}
//复位引脚拉低
void SX1268_rst_low()
{
    nrf_gpio_pin_clear(SX1268_RESET_PIN);
    // NRF_LOG_INFO("SX1268_rst_low");
}
//天线拉高发送数据
void SX1268_ant_sw_tx()
{
    nrf_gpio_pin_set(SX1268_ANT_SW);
    // NRF_LOG_INFO("SX1268_ant_sw_tx");
}
//天线拉低读取数据
void SX1268_ant_sw_rx()
{
    nrf_gpio_pin_clear(SX1268_ANT_SW);
    // NRF_LOG_INFO("SX1268_ant_sw_rx");
}
```

3.5任务函数

3.5.1初始化和读取任务函数

```
void task_sx1268()
{
    msg_t msg;//存储数据
    uint8_t ret = 0;

    //初始化GPIO口
    SX1268_gpio_init();
    //延时一会让配置好点
    platform_delay_ms(100);
    //初始化spi，必须先初始化spi因为读取sx1268寄存器需要通过spi
    spi1_init();
    //释放spi信号量
    xSemaphoreGive(global.semaphore_spi1);
    //初始化sx1268芯片配置
    sx1268_user_init();

     /*向task_handle_SX1268,发送通知，使task_sx1268_dio其解除阻塞状态 */
    xTaskNotifyGive(global.task_handle_SX1268);
    //初始化标志位置一
    global.lora_init = 1;

    for (;;)
    {
        //阻塞等待接收队列
```

```
        if (pdTRUE == xQueueReceive(global.queue_sx1268_send_msg, &msg,
portMAX_DELAY))
        {
            if (global.lora_init)//判断初始化完成没，防止误操作
            {
                nrf_gpio_pin_toggle(LED_GREEN);
                //打印发送数据
                NRF_LOG_INFO("SX1268 send data, buf:0x%02x", msg.buf[0]);
                //发送数据
                sx126x_tx(&SX1268, msg.buf, msg.len, 50);
                //等待发送中断产生的信号量
                ret = xSemaphoreTake(global.semphore_SX1268_tx_done,
ms2ticks(100));
                if (!ret)
                {
                    //超时未发送成功
                    NRF_LOG_INFO("SX1268_TX_TIMEOUT");
                }
            }
        }
        //释放申请的内存
        freepoint(msg.buf);
    }
}
```

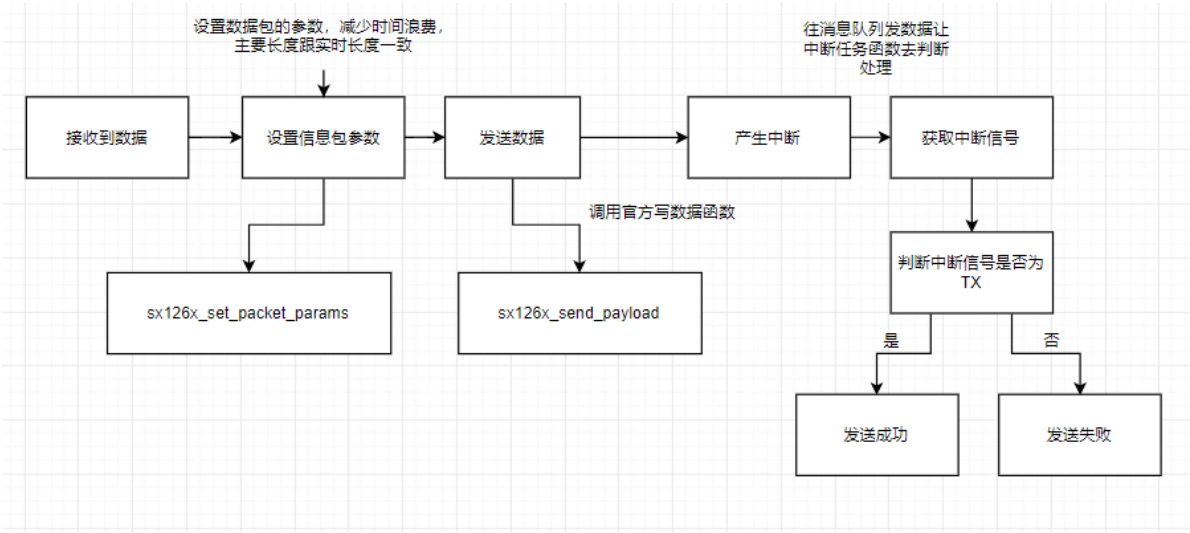
3.5.2处理的中断任何函数

```
void task_sx1268_dio()
{
     /* 等待task_sx1268的通知，进入阻塞 */
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    msg_t msg;
    NRF_LOG_INFO("1268_1 task start");

    uint8_t payload_len = 0;
    uint8_t payload_data[255] = {0};
    uint16_t sx1268_irq_status;
    //设置射频频率
    sx126x_set_rf_frequency(&SX1268, 570377000);
    //设置为接收模式
    sx126x_rx(&SX1268, 0);
    for (;;)
    {
        //等待中断产生信号
        if (pdTRUE == xQueueReceive(global.queue_SX1268_irq_msg, &msg,
portMAX_DELAY))
        {
            //提取中断产生的信号
            sx1268_irq_status = sx126x_get_irq_status(&SX1268);
            //打印中断信号
            NRF_LOG_ERROR("SX1268 IRQ status %x", sx1268_irq_status);
            //扫清终端请求
            sx126x_clear_irq_status(&SX1268, 0xFFFF);
            //判断产生的中断是不是发送
            if ((sx1268_irq_status & SX126X_IRQ_TX_DONE) == SX126X_IRQ_TX_DONE)
            {
```

```
                //释放发送信号
                xSemaphoreGive(global.semphore_SX1268_tx_done);
                NRF_LOG_INFO("SX1268_TX_OK");
                //设置为接收模式
                sx126x_rx(&SX1268, 0);
            }
            //判断中断产生的信号是否为接收
            if ((sx1268_irq_status & SX126X_IRQ_RX_DONE) == SX126X_IRQ_RX_DONE)
            {
                //获取每种报文类型的报文状态
                sx126x_get_packet_status(&SX1268, &SX1268.packet_status);
                //最后一个数据包的接收信号强度指示
                NRF_LOG_INFO("SX1268_RSSI:~%d$",
SX1268.packet_status.Params.LoRa.RssiPkt);
                //数据长度清零
                payload_len = 0;
                //数组清零
                memset(payload_data, 0, sizeof(payload_data));
                //读取到的数据payload_data，读取收到的有效载荷长度payload_len
                sx126x_get_payload(&SX1268, payload_data, &payload_len, 5);
                //打印读取到的数据长度和地址
                NRF_LOG_INFO("SX1268 got payload len=  %d, payload: 0x%02x",
payload_len, payload_data[0]);
                //判断是否是自己需要读取到的数据
                if (payload_len == 5 && (payload_data[0] == 0xA1 ||
payload_data[0] == 0xB1 || payload_data[0] == 0xC1))
                {
                    NRF_LOG_INFO("SX1268_RX_OK")
                }
            }
        }
    }
```

3.6 lora常用函数解读

3.6.1 sx126x_tx解读




```
//函数作用发送数据
//参数一：lora结构体
//参数二：数据的地址
```

```
//参数三：传输数据的长度
//参数四：超时时间  超时时间可通过软件计算
void sx126x_tx (sx126x_dev *dev, uint8_t *data, uint16_t len,
                uint32_t timeout_ms)
{
    //设置掩码
  sx126x_set_dio_irq_params (dev, SX126X_IRQ_TX_DONE | SX126X_IRQ_RX_TX_TIMEOUT,
                             SX126X_IRQ_TX_DONE | SX126X_IRQ_RX_TX_TIMEOUT,
SX126X_IRQ_RADIO_NONE,
                             SX126X_IRQ_RADIO_NONE);

  SX126X_PacketParams_t packet_params;
  memcpy (&packet_params, &dev->packet_params, sizeof(SX126X_PacketParams_t));
    //判断传输类型是不是lora
  if (dev->packet_type == SX126X_PACKET_TYPE_LORA)
  {
      //lora的PayLoad长度等于tx的长度
    packet_params.Params.LoRa.PayloadLength = len;
  }
  else
  {
    packet_params.Params.Gfsk.PayloadLength = len;
  }
    sx128x_set_packet_params(&sx1280_1, &sx1280_1.packet_params);
                sx128x_send_payload(&sx1280_1, msg.buf, msg.len,
sx1280TickTime);
  //设置报文参数
  sx126x_set_packet_params (dev, &packet_params);
    //发送数据
  sx126x_send_payload (dev, data, len, timeout_ms * 1000 * 1000 / 15625);
}
```

3.6.2 sx126x_init解读

```
bool sx126x_init (sx126x_dev *dev)
{
  uint8_t tmp_reg = 0;
  //芯片复位
  sx126x_reset (dev);

  //芯片叫醒
  sx126x_wakeup (dev);
  //设置待机模式允许降低能源消耗
  sx126x_set_standby (dev, SX126X_STDBY_RC);
  //保存lora调制解调器同步字值的寄存器的地址 地址应该是0x14
  tmp_reg = sx126x_read_register (dev, SX126X_REG_LR_SYNCWORD);
  if (tmp_reg != 0x14)
  {
    return false;
  }
  //指示是否使用DIO2控制射频开关  参数二在初始化配置的时候选择
  sx126x_set_dio2_as_rf_switch_ctrl ( dev, dev->dio2_as_rf_switch_ctrl_flag );
  //设置功率调节器的工作模式 未设置默认仅使用LDO意味着Rx或Tx电流加倍
  sx126x_set_regulator_mode (dev, dev->regulator_mode);
  // 初始化TCXO控制
  if (dev->tcxo_enable)
  {
```

```
    dev->tcxo_enable ();
  }
  // Force image calibration
  dev->image_calibrated = false;

  //为传输和接收设置数据缓冲基地地址
  sx126x_set_buffer_base_address (dev, 0x00, 0x00);
  //设置传输参数  参数一发射功率 参数二表示功率放大器的斜坡时间
  sx126x_set_tx_params (dev, dev->tx_power, SX126X_RADIO_RAMP_200_US);
  //设置中断寄存器
  sx126x_set_dio_irq_params (dev, SX126X_IRQ_RADIO_ALL, SX126X_IRQ_RADIO_ALL,
SX126X_IRQ_RADIO_NONE,
                             SX126X_IRQ_RADIO_NONE);
  //决定哪个中断将停止内部无线电rx定时器 fasle报头/同步字检测后定时器停止
  sx126x_set_stop_rx_timer_on_preamble_detect (dev, false);
  //判断模式是不是GFSK
  if (SX126X_PACKET_TYPE_GFSK == dev->modulation_params.PacketType)
  {
    sx126x_set_sync_word (dev, ( uint8_t[]
        )
        { 0xC1, 0x94, 0xC1, 0x00, 0x00, 0x00, 0x00, 0x00 });
    sx126x_set_whitening_seed (dev, 0x01FF);
  }
  //设置调制参数
  sx126x_set_modulation_params (dev, &dev->modulation_params);
  //设置信息包参数
  sx126x_set_packet_params (dev, &dev->packet_params);
  //判断是否是连续接收
  if (dev->rx_continuous)
  {
    //超时时间为0
    dev->symble_timeout = 0;
  }
  //判断模式是不是lora
  if (SX126X_PACKET_TYPE_LORA == dev->modulation_params.PacketType)
  {
    //设置无线电将等待的符号数以验证接收
    sx126x_set_lora_symb_num_timeout (dev, dev->symble_timeout);
    // WORKAROUND - Optimizing the Inverted IQ Operation, see DS_SX1261-2_V1.2
datasheet chapter 15.4 判断IQ设置是不是倒转
    if (SX126X_LORA_IQ_INVERTED == dev->packet_params.Params.LoRa.InvertIQ)
    {
      // RegIqPolaritySetup = @address 0x0736
      sx126x_write_register (dev, 0x0736,
                             sx126x_read_register (dev, 0x0736) & ~(1 << 2));
    }
    else
    {
      // RegIqPolaritySetup @address 0x0736
      sx126x_write_register (dev, 0x0736,
                             sx126x_read_register (dev, 0x0736) | (1 << 2));
    }
    // WORKAROUND END
  }
  return true;
}
```

3.6.3 sx126x_get_payload 解读获取包数据

```
//函数功能获取数据包
//函数原型
//参数二：存放数据包的地址 参数三：存放数据包的长度 参数四：数据长度不得超过的值
uint8_t sx126x_get_payload (sx126x_dev *dev, uint8_t *buffer, uint8_t *size,
                            uint8_t maxSize)
{
  uint8_t offset = 0;
    //size：数据的真实长度  offset：上次接收的数据包缓冲区地址指针 防止发的太快没被读取 可以接着读防止覆盖 不做这步只能读取最新数据
  sx126x_get_rx_buffer_status (dev, size, &offset);
  //判断真实长度是不是大于最大长度设定
  if (*size > maxSize)
  {
    return 1;
  }
  //从dev中的offset地址读取size个数据到buffer
  sx126x_read_buffer (dev, offset, buffer, *size);
  return 0;
}
```

3.6.5 sx126x_set_dio_irq_params 配置中断寄存器

```
/*
中断号对应的中断功能 该位置一表示开启 开启触发才会产生中断信号
根据所选的帧和芯片模式，总共有10个可能的中断源。每一个都可以启用或屏蔽。此外，每个节点都可以映射到DIO1、DIO2或DIO3。
0包传输完成 1包读取完成  2检测到的预处理程序 3检测到有效的同步word
4收到有效的LoRa头  5ora头的CRC校验错误  6收到错误的CRC  7通道活动检测已完成
8检测到通道活动 9发送或接受超时
*/
```

8.5 IRQ Handling

In total there are 10 possible interrupt sources depending on the selected frame and chip mode. Each one can be enabled or masked. In addition, each one can be mapped to DIO1, DIO2 or DIO3.

Table 8-4: IRQ Status Registers

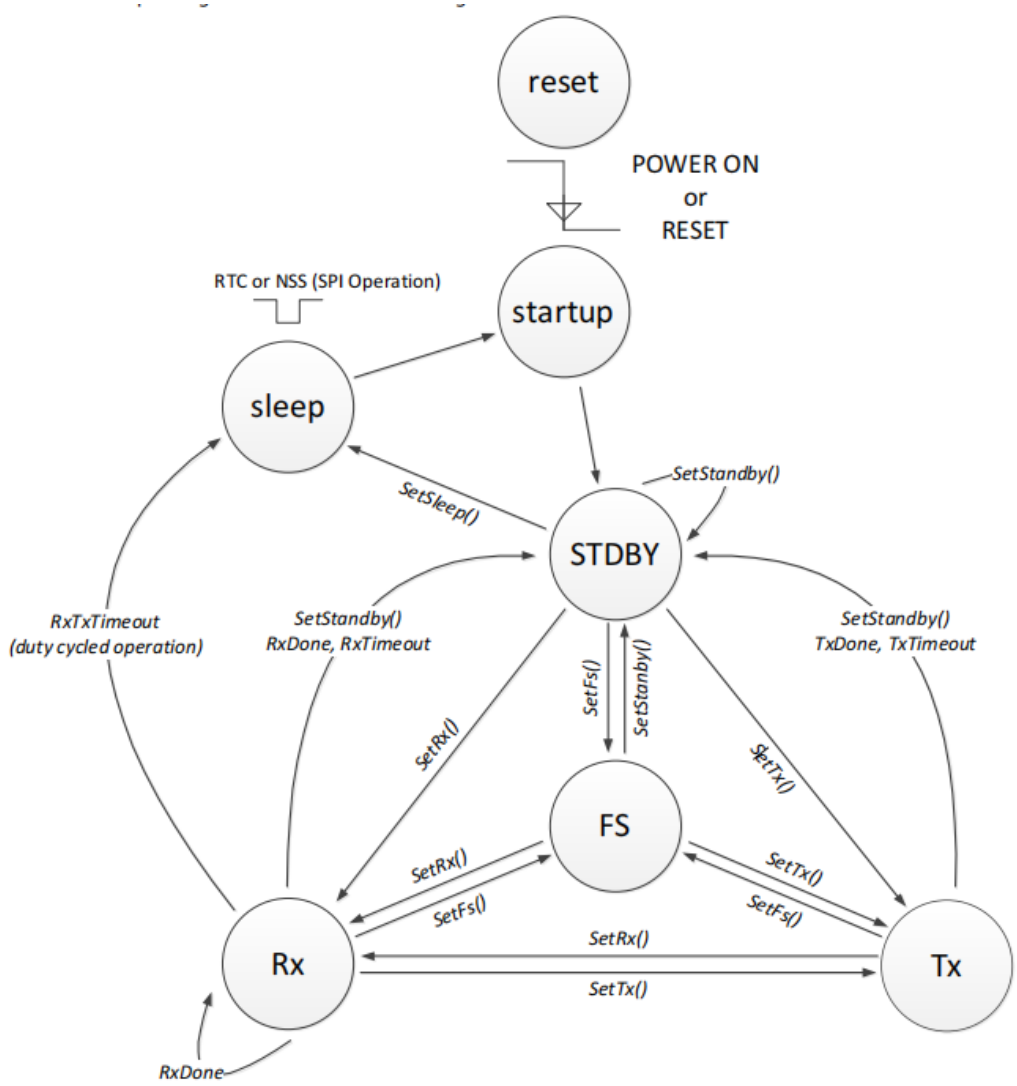
Bit	IRQ	Description	Protocol
0	TxDone	Packet transmission completed	All
1	RxDone	Packet received	All
2	PreambleDetected	Preamble detected	All
3	SyncWordValid	Valid Sync Word detected	FSK
4	HeaderValid	Valid LoRa Header received	LoRa®
5	HeaderErr	LoRa® header CRC error	LoRa®
6	CrcErr	Wrong CRC received	All
7	CadDone	Channel activity detection finished	LoRa®
8	CadDetected	Channel activity detected	LoRa®
9	Timeout	Rx or Tx Timeout	All

For more information on how to setup IRQ and DIOs, refer to the function *SetDioIrqParams()* in Section 13.3.1 "SetDioIrqParams" on page 77.

```
//函数功能配置中断寄存器
//参数二：控制中断的开启与否
//参数三-五：控制中断的映射在哪个dio口
void sx126x_set_dio_irq_params (sx126x_dev *dev, uint16_t irqMask,
                                uint16_t dio1Mask, uint16_t dio2Mask,
                                uint16_t dio3Mask)
{
  uint8_t buf[8];

  buf[0] = (uint8_t) ((irqMask >> 8) & 0x00FF);//高位在前大端
  buf[1] = (uint8_t) (irqMask & 0x00FF);//低八位
  buf[2] = (uint8_t) ((dio1Mask >> 8) & 0x00FF);
  buf[3] = (uint8_t) (dio1Mask & 0x00FF);
  buf[4] = (uint8_t) ((dio2Mask >> 8) & 0x00FF);
  buf[5] = (uint8_t) (dio2Mask & 0x00FF);
  buf[6] = (uint8_t) ((dio3Mask >> 8) & 0x00FF);
  buf[7] = (uint8_t) (dio3Mask & 0x00FF);
  sx126x_write_command (dev, RADIO_CFG_DIOIRQ, buf, 8);
}
```

4.lora函数

lora工作图



注意事项

```
/*
1.lora的发送和接受产生的中断都是dio1 配置更改需要更改寄存器
2.发送结束需要延时一会否则会错误
3.睡眠模式后需要片选引脚才能唤醒 或者定时器唤醒
4.lora的配置要根据标准配置不能乱配置
5.sx1268的传输功率最大22 官方手册如下图
6.影响到接收是否正确的配置有
**********
  global.sx1268.modulation_params.PacketType = SX126X_PACKET_TYPE_LORA; //传输类型
  global.sx1268.modulation_params.Params.LoRa.Bandwidth = SX126X_LORA_BW_250;/宽
带设置
  global.sx1268.modulation_params.Params.LoRa.CodingRate = SX126X_LORA_CR_4_5;//
编码速率
  global.sx1268.modulation_params.Params.LoRa.LowDatarateOptimize = 0;//速率优化开
关
  global.sx1268.modulation_params.Params.LoRa.SpreadingFactor = LORA_SF7;

  global.sx1268.packet_params.PacketType = SX126X_PACKET_TYPE_LORA;//数据包类型
  global.sx1268.packet_params.Params.LoRa.HeaderType =
SX126X_LORA_PACKET_IMPLICIT;//隐式头
  global.sx1268.packet_params.Params.LoRa.CrcMode = SX126X_LORA_CRC_ON;//CRC校验码
  global.sx1268.packet_params.Params.LoRa.PayloadLength = 49;//数据长度
  global.sx1268.packet_params.Params.LoRa.InvertIQ = SX126X_LORA_IQ_NORMAL;/iq配
置
```

```
  global.sx1268.packet_params.Params.LoRa.PreambleLength = 8;//前导码长度
**********
7.发送超时可能因为没有打开优化速率开关
global.sx1268.modulation_params.Params.LoRa.LowDatarateOptimize = 0;
8.sx1268的数据长度影响接收，但sx1280的数据长度不影响接收但是需要配置数据包
9.lora如果两个芯片共用一个spi两个芯片的发送同一时间最好用事件标志组，先发送的数据会接收不到
9.天线的开关和天线使能开关不一定有要看电路图 但CS和RESET一定有
10.若果两个芯片要同时发送给对方，最好使用事件标志组防止共用spi。消息队列发送数据发送第一个后必须
延时15ms以上数据越长延时越长，作用：为了释放cpu让另一个芯片去捕捉到这个芯片所发送的数据
*/
```

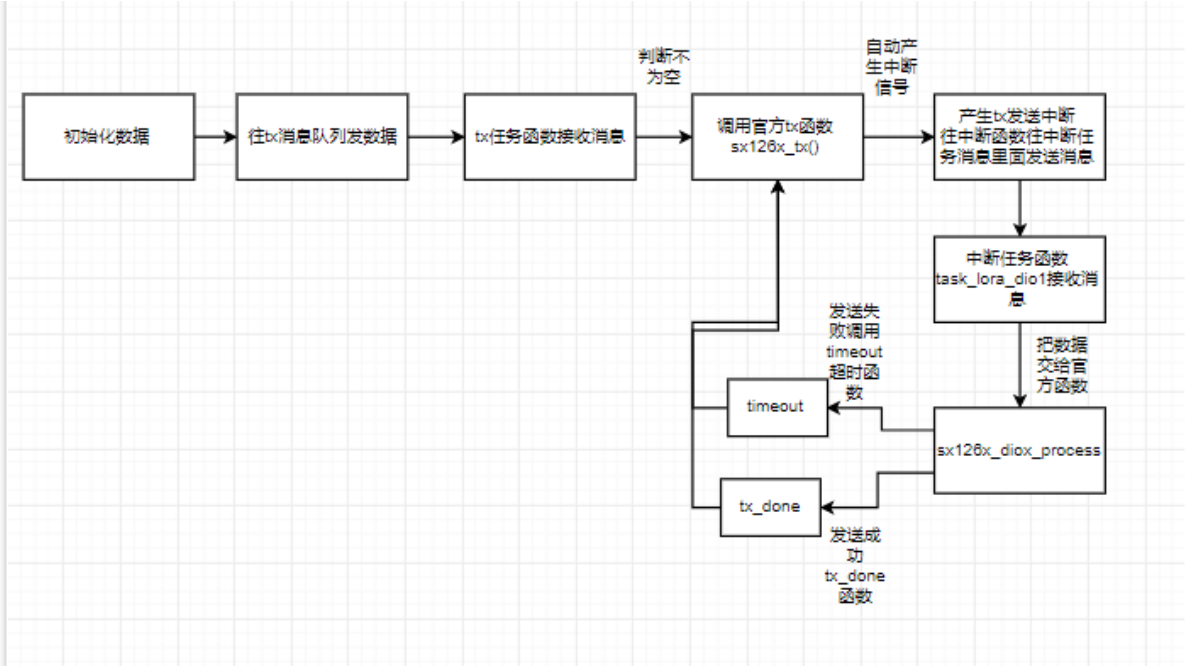
测试问题

```
/*
测试问题
**********接收发送不了 || 接收数据不正确***************
常见问题：
1.接收发送不了
2.发送正常接收不正常
3.CRC校验码错误
检查SPI是否初始化正常 >> 检查芯片引脚是否正确 >> 检查芯片配置是否正确 >> 检查频率是否正确
注：如果怀疑是硬件问题可将代码换芯片尝试看是否正确
*/
```

Symbol	Description	Conditions	Min	Typ	Max	Unit
TXOP	Maximum RF output power	Highest power step setting	-	+22	-	dBm
TXDRP	RF output power drop versus supply voltage	under DC-DC or LDO VDDop range from 1.8 to 3.7 V	-	0.5	-	dB
		at +22 dBm, VBAT = 2.7 V	-	2	-	dB
		at +22 dBm, VBAT = 2.4 V	-	3	-	dB
		at +22 dBm, VBAT = 1.8 V	-	6	-	dB
TXPRNG	RF output power range	Programmable in 31 steps, typical value	TXOP-31	-	TXOP	dBm
TXACC	RF output power step accuracy		-	± 2	-	dB
TXRMP	Power amplifier ramping time	Programmable	10	-	3400	µs
TS_TX	Tx wake-up time	Frequency Synthesizer enabled	-	36 + PA ramping	-	µs

4.1 lora（sx1268）发送

流程图



4.1.1lora发送函数调用过程

```
/*
1.初始化数据后发送消息队列让发送任务函数处理数据，发送的数据夹带着要发送的数据
2.task_lora_tx接收到数据后判断是否是发送，后调用sx126x_tx官方函数发送数据，然后等待是否发送成功
3.sx126x_tx发送产生发送中断判断是dio几的中断，然后发送消息队列给中断处理任务函数
4.中断处理任务函数接收到消息队列，再把数据发送给sx126x_dio1_process官方函数处理
5.sx126x_dio1_process判断是发送成功还是超时；发送成功调用tx_done函数发送失败调用timeout函数
6.tx_done成功则发送任务通知
7.task_lora_tx等待任务通知，等待时间自定，到时没有tx_done则不会收到任务通知则发送失败，发送失败调用sx126x_init()再初始化，
8.释放申请的空间调用vPortFree()函数，然后将指针指向NULL
9.发送成功必须延时2ms，否则数据会错乱
[00:00:04.155,273] <error> app: [SET FREQ] Freq: 470377000, channle: 0
[00:00:04.156,860] <error> app: [SET FREQ] Freq: 470994000, channle: 1
[00:00:04.158,447] <error> app: [SET FREQ] Freq: 471611000, channle: 2
[00:00:04.159,912] <error> app: [SET FREQ] Freq: 472228000, channle: 3
*/
```

4.1.2发送函数

4.1.2.1 初始化数据

```
msg_t msg;//定义数据结构体
msg.buf_len = 49;//定义buf的长度
msg.buf = pvPortMalloc(msg.buf_len);//申请内存
msg.buf[0] = 0x77;//给数据
xQueueSend(global.queue_handle_lora_tx, &msg, portMAX_DELAY);//往消息队列里发送数据
```

4.1.2.2 lora发送数据任务函数

```
void task_lora_tx_li(void *arg)
{
  BaseType_t ret_code;
  msg_t lora_tx_msg = {.buf = NULL};
```

```
  //等待任务通知，先初始化好lora再通知
  ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
  for(;;)
  {
      //等待消息来
    if(pdTRUE == xQueueReceive (global.queue_handle_lora_tx, &lora_tx_msg,
portMAX_DELAY))
    {
        //判断buf是否为空，判断是不是接收到数据
      if(lora_tx_msg.buf != NULL)
      {
          //打印数据长度
        NRF_LOG_INFO("[task_lora_tx_li] Send data length: %d",
lora_tx_msg.buf_len);
          //调用官方tx函数
        sx126x_tx(&global.sx1268, lora_tx_msg.buf, lora_tx_msg.buf_len, 55);
          //等待任务通知，等100ms，任务通知在tx_done函数中
        ret_code = ulTaskNotifyTake(pdTRUE, ms2ticks(100));
        /* 根据等待通知的结果，执行对应操作 */
        if (ret_code == 1)
        {
            //发送成功，延时一会释放cpu
           vTaskDelay(ms2ticks(2));
        }
        else
        {
            //超时
          NRF_LOG_INFO("SX1268 tx timeout.");
          //重新初始化一下
         sx126x_init(&global.sx1268);
        }
      }
        //设置为接收模式
       sx126x_rx(&global.sx1268, 0);
    }
      //判断buf为真嘛
    if (lora_tx_msg.buf)
    {
        //释放空间
      vPortFree(lora_tx_msg.buf);
      //buf指向null
      lora_tx_msg.buf = NULL;
    }
  }
}
```

4.1.2.3 lora接收中断任务函数

```
void task_lora_dio1_li(void *arg)
{
  msg_t msg;//申请结构体
  msg.buf = NULL;//buf指向空
  for(;;)
  {
      //等待中断函数发送的消息
    if(pdPASS == xQueueReceive (global.queue_lora_dio1_msg, &msg,
portMAX_DELAY))
```

```
    {
        //把数据交给官方函数处理
      sx126x_dio1_process (&global.sx1268);
    }
  }
}
```

4.1.2.4 sx126x_dio1_process官方处理函数

```
void sx126x_dio1_process (sx126x_dev *dev)
{
  volatile uint16_t irq_status = 0;
  //获取中断状态
  irq_status = sx126x_get_irq_status (dev);
  //清除中断状态
  sx126x_clear_irq_status (
      dev, (SX126X_IRQ_RX_DONE | SX126X_IRQ_CRC_ERROR | SX126X_IRQ_RX_TX_TIMEOUT
| SX126X_IRQ_TX_DONE)); // This handler only handle these IRQ
  //判断是不是rx模式
  if (SX126X_MODE_RX == dev->op_mode)
  {
    //判断是不是lora模式
    if (SX126X_PACKET_TYPE_LORA == dev->modulation_params.PacketType)
    {
      //判断rx持续打开没
      if (dev->rx_continuous == false)
      {
        //!< Update operating mode state to a value lower than \ref
MODE_STDBY_XOSC
        dev->op_mode = SX126X_MODE_STDBY_RC;//待机模式
      }
      //产生的中断是不是rx
      if ((irq_status & SX126X_IRQ_RX_DONE) == SX126X_IRQ_RX_DONE)
      {
        //判断校验码是否对
        if ((irq_status & SX126X_IRQ_CRC_ERROR) == SX126X_IRQ_CRC_ERROR)
        {
          if ((dev != NULL) && (dev->callbacks.rxError != NULL))
          {
            dev->callbacks.rxError (SX126X_IRQ_CRC_ERROR_CODE);
          }
        }
        else
        {
            //rx连续是否没开启
          if (dev->rx_continuous == false)
          {
            //待机模式
            dev->op_mode = SX126X_MODE_STDBY_RC;
            sx126x_write_register ( dev, 0x0902, 0x00);
            sx126x_write_register ( dev, 0x0944, sx126x_read_register (dev,
0x0944) | (1 << 1));

          }
          //清空数据
          memset (dev->radio_rx_payload, 0, sizeof(dev->radio_rx_payload));
          //获取数据
```

```
          sx126x_get_payload (dev, dev->radio_rx_payload,
                              &dev->radio_rx_payload_len, 255);
          //得到包状态
          sx126x_get_packet_status (dev, &dev->packet_status);
          //得到滴答时钟
          dev->rx_done_tick = dev->dio1_tick;
          //判断是否为空
          if ((dev != NULL) && (dev->callbacks.rxDone != NULL))
          {
            //调用rxdone函数
            dev->callbacks.rxDone (dev->radio_rx_payload,
                                   dev->radio_rx_payload_len,
                                   dev->packet_status.Params.LoRa.RssiPkt,
                                   dev->packet_status.Params.LoRa.SnrPkt);
          }
        }
      }
      //判断状态是否是超时
      if ((irq_status & SX126X_IRQ_RX_TX_TIMEOUT) == SX126X_IRQ_RX_TX_TIMEOUT)
      {
        //判断是否为空
        if ((dev != NULL) && (dev->callbacks.rxTimeout != NULL))
        {
          //调用超时函数
          dev->callbacks.rxTimeout ();
        }
      }
    }
    else if (SX126X_PACKET_TYPE_GFSK == dev->modulation_params.PacketType)
    {
      //TODO
    }
  }
  //判断模式是不是tx
  else if (SX126X_MODE_TX == dev->op_mode)
  {
    //待机模式
    dev->op_mode = SX126X_MODE_STDBY_RC;
    //判断产生的中断是不是tx
    if ((irq_status & SX126X_IRQ_TX_DONE) == SX126X_IRQ_TX_DONE)
    {
      //tx_done滴答时钟等于dio1滴答时钟
      dev->tx_done_tick = dev->dio1_tick;
      //判断是否为空
      if ((dev != NULL) && (dev->callbacks.txDone != NULL))
      {
        //调用txdone函数
        dev->callbacks.txDone ();
      }
    }
    //判断产生的中断是否是tx_timeout
    if ((irq_status & SX126X_IRQ_RX_TX_TIMEOUT) == SX126X_IRQ_RX_TX_TIMEOUT)
    {
      //判断是否为空
      if ((dev != NULL) && (dev->callbacks.txTimeout != NULL))
      {
        //调用tx_timeout函数
        dev->callbacks.txTimeout ();
```

```
        }
      }
    }
  }
```

4.1.2.5 tx_done官方处理函数

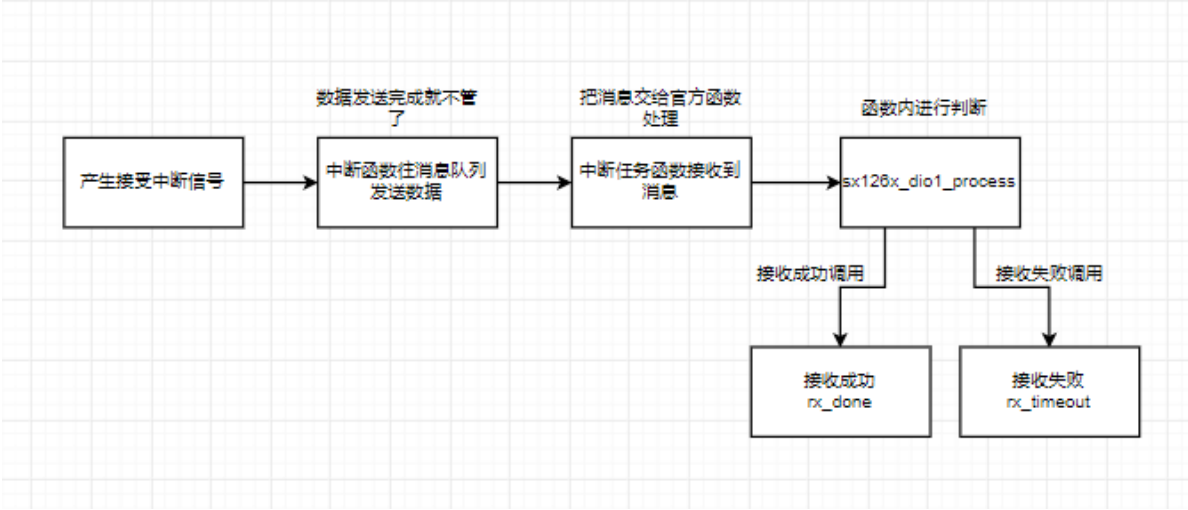
```
void lora_tx_done()
{
  //发送任务通知
  xTaskNotifyGive(global.task_handle_lora_tx_li);
  // 发送完成后进入sleep模式，降低功耗 启动时为热启动
  sx126x_set_sleep_warm_start(&global.sx1268);
  NRF_LOG_INFO("SX1268 tx done.");
}
```

4.1.2.6 tx_timeout官方处理函数

```
void lora_tx_timeout()
{
  // 发送完成后进入sleep模式，降低功耗，一般不会进入到这里
  sx126x_set_sleep_warm_start (&global.sx1268);
}
```

4.2lora（sx1268）接收

流程图



4.2.1lora发送函数调用过程

```
/*
1.配置好中断，写中断处理函数
2.中断处理函数往消息队列发消息
3.中断处理任务函数接收到消息
4.获取当前时间，把消息扔给sx126x_dio1_process处理
5.成功则调用txDone，超时则调用txTimeout
6.然后设置为接收模式
7.如果tx_timeout则初始化一次
8.无论接收成功与否都要释放申请的buf内存
*/
```

4.2.2 lora发送配置函数

4.2.2.1中断配置

```
nrfx_gpiote_in_config_t dio1_config = NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
dio1_config.pull = GPIO_PIN_CNF_PULL_Pulldown;
APP_ERROR_CHECK(
    nrfx_gpiote_in_init(SX1268_DIO1_PIN, &dio1_config, gpio_irq_handle));
nrfx_gpiote_in_event_enable(SX1268_DIO1_PIN, true);

nrfx_gpiote_in_config_t dio2_config = NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
dio2_config.pull = GPIO_PIN_CNF_PULL_Pulldown;
APP_ERROR_CHECK(
    nrfx_gpiote_in_init(SX1268_DIO2_PIN, &dio2_config, gpio_irq_handle));
nrfx_gpiote_in_event_enable(SX1268_DIO2_PIN, true);
```

4.2.2.2 中断函数

```
void gpio_irq_handle(nrfx_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
  msg_t msg;
  msg.tick = xTaskGetTickCountFromISR();
  switch (pin)
  {
  case SX1268_DIO1_PIN:
    xQueueSendFromISR(global.queue_lora_dio1_msg, &msg, NULL);
    break;
  case SX1268_DIO2_PIN:
    xQueueSendFromISR(global.queue_lora_dio2_msg, &msg, NULL);
    break;
  }
}
```

4.3lora(sx1268)常用函数

```
//初始化sx1268
bool sx126x_init(sx126x_dev *dev);
//sx1268 tx传输
void sx126x_tx(sx126x_dev *dev, uint8_t* data, uint16_t len, uint32_t
timeout_ms);
//sx1268读取
void sx126x_rx(sx126x_dev *dev, uint32_t timeout_ms);
//如果是睡眠模式则唤醒
void sx126x_check_device_ready( sx126x_dev *dev );
```

```
//保存要发送到无线电缓冲区的有效载荷
void sx126x_set_payload( sx126x_dev *dev, uint8_t *payload, uint8_t size );
//读取收到的有效载荷
uint8_t sx126x_get_payload( sx126x_dev *dev, uint8_t *payload, uint8_t *size,
uint8_t maxSize );
//发送有效载荷
void sx126x_send_payload( sx126x_dev *dev, uint8_t *payload, uint8_t size,
uint32_t timeout );
//设置睡眠热启动 配置保留配置保留
void sx126x_set_sleep_warm_start (sx126x_dev *dev);
//设置睡眠冷启动 重新创建
void sx126x_set_sleep_cold_start (sx126x_dev *dev);
//设置睡眠模式
void sx126x_set_sleep( sx126x_dev *dev, SX126X_SleepParams_t sleepConfig );
//设置待机模式 SX126X_STDBY_RC为待机模式
void sx126x_set_standby( sx126x_dev *dev, SX126X_RadioStandbyModes_t mode );
//读取寄存器中的值
void sx126x_read_registers( sx126x_dev *dev, uint16_t address, uint8_t *buffer,
uint16_t size );
//指示是否使用DIO2控制射频开关  参数二在初始化配置的时候选择
void sx126x_set_dio2_as_rf_switch_ctrl( sx126x_dev *dev, uint8_t enable );
//设置功率调节器的工作模式 不设置默认仅使用LDO意味着Rx或Tx电流加倍
void sx126x_set_regulator_mode( sx126x_dev *dev, SX126X_RadioRegulatorMode_t
mode );
//为传输和接收设置数据缓冲基地地址 默认发送读取地址都为0x00
void sx126x_set_buffer_base_address( sx126x_dev *dev, uint8_t txBaseAddress,
uint8_t rxBaseAddress );
//设置传输参数 参数二发射功率配置时设置  参数三功率放大器斜坡时间
void sx126x_set_tx_params( sx126x_dev *dev, int8_t power, RadioRampTimes_t
rampTime );
//设置中断寄存器
void sx126x_set_dio_irq_params( sx126x_dev *dev, uint16_t irqMask, uint16_t
dio1Mask, uint16_t dio2Mask, uint16_t dio3Mask );
//决定哪个中断将停止内部无线电rx定时器 fasle报头/同步字检测后定时器停止
void sx126x_set_stop_rx_timer_on_preamble_detect( sx126x_dev *dev, bool enable
);
```

4.2.1lora(sx1280)引脚

lora（sx1280）边缘物联网引脚

```
/*
// SPI1 Lora sx1280 use
#define SPI2_SCK_PIN                    NRF_GPIO_PIN_MAP(0,20)
#define SPI2_MISO_PIN                   NRF_GPIO_PIN_MAP(0,24)
#define SPI2_MOSI_PIN                   NRF_GPIO_PIN_MAP(0,23)

// Lora sx1280
 // output pin
#define SX1280_1_PWRKEY_PIN             NRF_GPIO_PIN_MAP(1,9)
#define SX1280_1_CS_PIN                 NRF_GPIO_PIN_MAP(0,6)
#define SX1280_1_RESET_PIN              NRF_GPIO_PIN_MAP(0,2)
#define SX1280_1_TX_EN_PIN              NRF_GPIO_PIN_MAP(0,0)
#define SX1280_1_RX_EN_PIN              NRF_GPIO_PIN_MAP(0,0)
 // input pin
#define SX1280_1_BUSY_PIN               NRF_GPIO_PIN_MAP(0,28)
#define SX1280_1_DIO1_PIN               NRF_GPIO_PIN_MAP(0,29)
```

```
#define SX1280_1_DIO2_PIN               NRF_GPIO_PIN_MAP(0,30)

 // output pin
#define SX1280_2_PWRKEY_PIN             NRF_GPIO_PIN_MAP(0,21)
#define SX1280_2_CS_PIN                 NRF_GPIO_PIN_MAP(0,19)
#define SX1280_2_RESET_PIN              NRF_GPIO_PIN_MAP(0,14)
#define SX1280_2_TX_EN_PIN              NRF_GPIO_PIN_MAP(0,0)
#define SX1280_2_RX_EN_PIN              NRF_GPIO_PIN_MAP(0,0)
 // input pin
#define SX1280_2_BUSY_PIN               NRF_GPIO_PIN_MAP(0,15)
#define SX1280_2_DIO1_PIN               NRF_GPIO_PIN_MAP(0,16)
#define SX1280_2_DIO2_PIN               NRF_GPIO_PIN_MAP(0,17)
*/
```

lora（sx1280）网关物联网引脚

```
/*
// SPI1 Lora sx1280 use
#define SPI2_SCK_PIN                    NRF_GPIO_PIN_MAP(0,22)
#define SPI2_MISO_PIN                   NRF_GPIO_PIN_MAP(0,24)
#define SPI2_MOSI_PIN                   NRF_GPIO_PIN_MAP(0,23)

// Lora sx1280
 // output pin
#define SX1280_1_PWRKEY_PIN             NRF_GPIO_PIN_MAP(0,13)
#define SX1280_1_CS_PIN                 NRF_GPIO_PIN_MAP(0,12)
#define SX1280_1_RESET_PIN              NRF_GPIO_PIN_MAP(0,17)
#define SX1280_1_TX_EN_PIN              NRF_GPIO_PIN_MAP(0,19)
#define SX1280_1_RX_EN_PIN              NRF_GPIO_PIN_MAP(0,11)
 // input pin
#define SX1280_1_BUSY_PIN               NRF_GPIO_PIN_MAP(0,16)
#define SX1280_1_DIO1_PIN               NRF_GPIO_PIN_MAP(0,15)
#define SX1280_1_DIO2_PIN               NRF_GPIO_PIN_MAP(0,14)
 // output pin
#define SX1280_2_PWRKEY_PIN             NRF_GPIO_PIN_MAP(0,25)
#define SX1280_2_CS_PIN                 NRF_GPIO_PIN_MAP(0,21)
#define SX1280_2_RESET_PIN              NRF_GPIO_PIN_MAP(1,1)
#define SX1280_2_TX_EN_PIN              NRF_GPIO_PIN_MAP(1,2)
#define SX1280_2_RX_EN_PIN              NRF_GPIO_PIN_MAP(0,20)
 // input pin
#define SX1280_2_BUSY_PIN               NRF_GPIO_PIN_MAP(0,10)
#define SX1280_2_DIO1_PIN               NRF_GPIO_PIN_MAP(0,9)
#define SX1280_2_DIO2_PIN               NRF_GPIO_PIN_MAP(1,0)
*/
```

4.2.2lora （sx1280）的发送

```
/*
注释:
数据可以通过信号量来传递给任务处理函数
*/
```

```
//任务处理函数
//设置1280的lora数据长度为需要发送的数据长度
sx1280_1.packet_params.Params.LoRa.PayloadLength = msg.len;
//重新配置lora的数据包，主要是配置长度避免浪费时间 如果每次的长度都是初始化配置好的这两步可忽略
sx128x_set_packet_params(&sx1280_1, &sx1280_1.packet_params);
//发送数据包 sx1280TickTime参数需定义
sx128x_send_payload(&sx1280_1, msg.buf, msg.len, sx1280TickTime);
//等待发送完成 释放信号量
xSemaphoreTake(global.semphore_sx1280_1_tx_done, ms2ticks(1000));
```

```
/*然后发送数据包会正常会触发TX中断*/
/*触发中断，发送消息给中断任务函数处理*/
```

```
//获取中断事件
sx1280irq_status = sx128x_get_irq_status(&sx1280_1);
//获取最后接收到的包有效载荷长度
sx128x_get_packet_status(&sx1280_1, &sx1280_1.packet_status);
//清除中断事件
sx128x_clear_irq_status(&sx1280_1, SX128X_IRQ_RADIO_ALL);
```

```
/*判断中断事件是否是写事件*/
((sx1280irq_status & SX128X_IRQ_TX_DONE) == SX128X_IRQ_TX_DONE)
```

```
/*如果是写事件则释放信号量给任务处理函数表示写完了，然后再设置为读模式*/
xSemaphoreGive(global.semphore_sx1280_1_tx_done);
sx128x_set_rx(&sx1280_1, sx1280TickTime);
```

4.2.2 lora（sx1280）的接收

```
//获取中断事件
sx1280irq_status = sx128x_get_irq_status(&sx1280_1);
//获取最后接收到的包有效载荷长度
sx128x_get_packet_status(&sx1280_1, &sx1280_1.packet_status);
//清除中断事件
sx128x_clear_irq_status(&sx1280_1, SX128X_IRQ_RADIO_ALL);
```

```
/*判断中断事件是否是读事件*/
((sx1280irq_status & SX128X_IRQ_RX_DONE) == SX128X_IRQ_RX_DONE)
```

```
/*因为不获取上一次接收数据包的缓存状态就会在现有的地址上覆盖，可能会导致数据丢失，
获取最后接收数据包缓存区状态可以保证不会出现数据丢失的情况*/
```

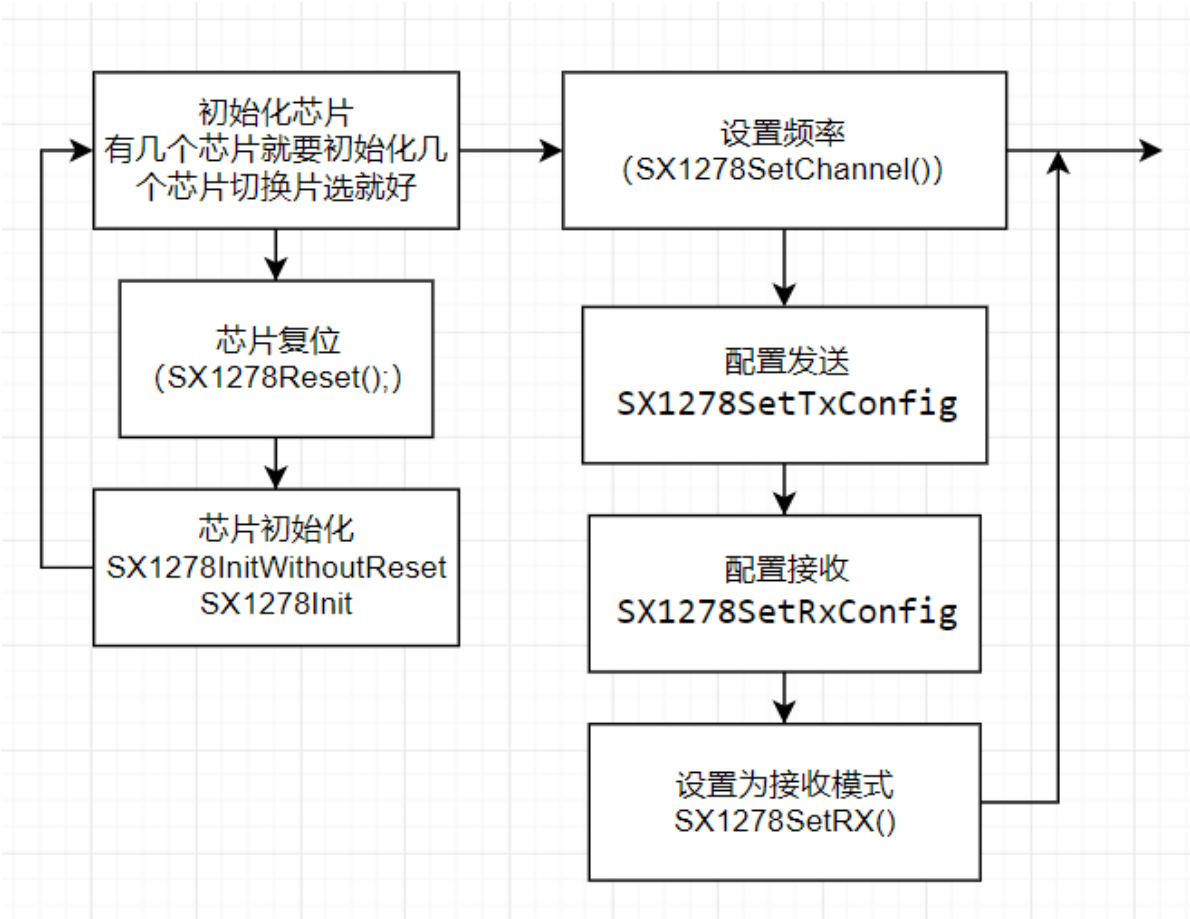
```
/*获取最后接收的数据包缓冲区状态*/
//函数原型 参数二：数据包的长度  参数三：上次接收的数据包缓冲区地址指针
void sx128x_get_rx_nuffer_status (sx128x_dev *dev, uint8_t *payloadLength,
                                  uint8_t *rxStartBufferPointer)
sx128x_get_rx_nuffer_status(&sx1280_1, &tempsize, &tempoffset);
```

```
/*获取数据：函数原型*/
//参数二：获取到的数据的存放地址 参数三：指向获取到的数据长度的地址  参数四：最大的长度
uint8_t sx128x_get_payload (sx128x_dev *dev, uint8_t *buffer, uint8_t *size,
                            uint8_t maxSize)
//方法一：不固定存放地址，这样可以避免出现数据丢失
//参数二：数组中的tempoffset是上一步获取的值，这样就不会出现参数覆盖的现象
sx128x_get_payload(&sx1280_1, &payload_buff[tempoffset], &payload_buffer_size,
10);
//方法二：固定存放地址，这样如果数据过多会出现丢失现象
//参数二：存放数据的地址  参数四：数据的最大长度，这里可以使用上面获取到的数据包的长度或者是初始化时定义的数据长度
sx128x_get_payload(&sx1280_1, payload_buff, &payload_buffer_size, 10);
```

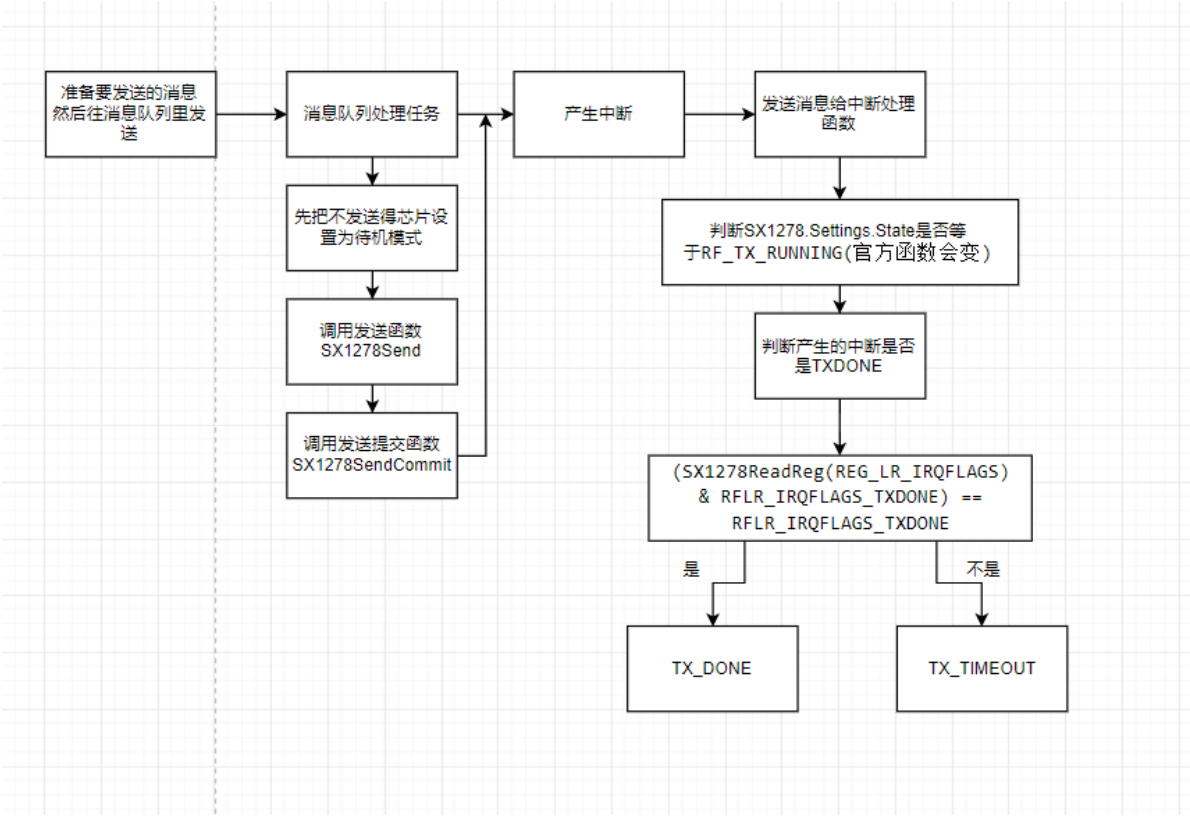
5.lora(sx1278)

5.0 流程图

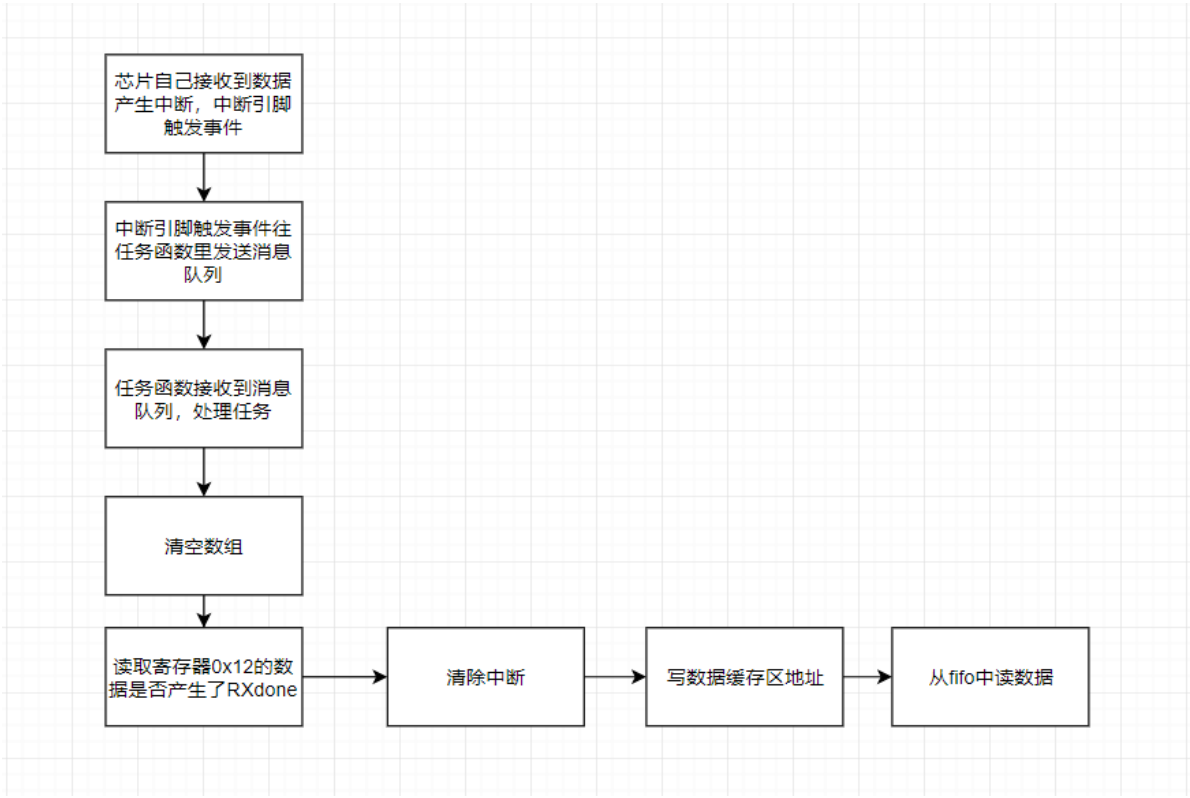
5.0.1初始化流程图



5.0.2发送流程图



5.0.3接收流程图



5.1 函数简介

注意事项

```
/*
由于中断没有TXTIMEOUT,所以我们选择配置一个寄存器，来判断是否发送成功，当然也可以跟sx1268一样使用信号量来判断是否发送成功，
一定的时间没有获取到信号量或者停止定时器则判断为未发送成功，最好重新初始化和设置频率一遍以便下一次不会出现问题
*/
```

5.1.0 复位

```
//芯片复位
SX1278Reset();
```

5.1.1 初始化

```
//初始化sx1278 参数多为false
bool SX1278Init(bool clkout_switch);
//初始化但不重新设定 参数多为false
bool SX1278InitWithoutReset(bool clkout_switch);
/*
两个都可以用作初始化 但后者不重新设定 建议用后者
*/
//示例
/*
复位芯片 可选
1. SX1278Reset();
初始化芯片  二选一
2.ret_code = SX1278InitWithoutReset(false);
初始化芯片  二选一
2.ret_code = SX1278Init(false);
*/
```

5.1.2 引脚配置

```
/* 射频自组网引脚配置 */
void radio_peripheral_init(void)
{
#if(sync_time_test)
  nrf_gpio_cfg_output(BLE_EN_PIN);
  nrf_gpio_pin_clear(BLE_EN_PIN);
#endif
  nrf_gpio_cfg_output(SX1278_CS_1_PIN);
  nrf_gpio_cfg_output(SX1278_CS_2_PIN);
  nrf_gpio_cfg_output(SX1278_CS_3_PIN);
  nrf_gpio_cfg_output(SX1278_CS_4_PIN);
  nrf_gpio_pin_set(SX1278_CS_1_PIN);
  nrf_gpio_pin_set(SX1278_CS_2_PIN);
  nrf_gpio_pin_set(SX1278_CS_3_PIN);
  nrf_gpio_pin_set(SX1278_CS_4_PIN);

  nrf_gpio_cfg_output(TWI0_SCL_PIN);
  nrf_gpio_pin_clear(TWI0_SCL_PIN);

  nrf_gpio_cfg_output(NOR_FLASH_CS_PIN);
  nrf_gpio_pin_set(NOR_FLASH_CS_PIN);
```

```
  nrf_gpio_cfg_output(SX1278_RESET_PIN);
  nrf_gpio_pin_set(SX1278_RESET_PIN);

  nrfx_gpiote_in_config_t dio0_config =
NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
  dio0_config.pull = GPIO_PIN_CNF_PULL_Pulldown;
  APP_ERROR_CHECK(nrfx_gpiote_in_init(SX1278_DIO0_PIN, &dio0_config,
gpio_irq_handler));
  nrfx_gpiote_in_event_enable(SX1278_DIO0_PIN, true);

  nrfx_gpiote_in_config_t dio3_config =
NRFX_GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
  dio3_config.pull = GPIO_PIN_CNF_PULL_Pulldown;
  APP_ERROR_CHECK(nrfx_gpiote_in_init(SX1278_DIO3_PIN, &dio3_config,
gpio_irq_handler));
  nrfx_gpiote_in_event_enable(SX1278_DIO3_PIN, true);
//#if (HELMET_KZ_V2_1) ||(GW_LPWAN_1_V1_1)
    spi1_init();
//#endif
  spi2_init();
}
```

5.1.3 中断事件

```
void gpio_irq_handler(nrfx_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
  if (pin == SX1278_DIO0_PIN)
  {
    //处理自定义处理并非固定
    radio_event_deliver(1, RADIO_DIO0, NULL);
      //自己写的
      radio.type = 1;
    xQueueSendFromISR(global.queue_handle_radio_tx, &radio, NULL);

  }
  else if (pin == SX1278_DIO3_PIN)
  {
      //处理自定义处理并非固定
    radio_event_deliver(1, RADIO_DIO3, NULL);
      //自己写的
      radio.type = 3;
    xQueueSendFromISR(global.queue_handle_radio_tx, &radio, NULL);
  }
}
```

5.1.2接收发送配置

5.1.2.0 设置频点和接受发送配置

```
void rf_channel_set_freq(void)
{
  uint8_t index;
    //设置频点
  uint32_t freq_my[4] ={470377000, 470994000, 471611000, 472228000};
  for (index = 0; index < 4; index++)
  {
```

```
    global.rxch_info[index].logical_chanlId = index;
    global.rxch_info[index].physical_chanlId = index;
    global.rxch_info[index].freq = freq_my[index];
      //主要函数 参数一：频点 参数二：第几个芯片片选
    rf_set_freq(freq_my[index], global.rxch_info[index].physical_chanlId);
    NRF_LOG_ERROR("[SET FREQ] Freq: %d, channle: %d",freq_my[index],
global.rxch_info[index].physical_chanlId);
  }
}
//参数一：频点 参数二：第几个
void rf_set_freq(uint32_t freq, uint8_t ch)
{
  int8_t power;
  uint8_t paFlag;
    //只有四个芯片，判断是不是大于四
  if (ch >= 4)
    {
      return;
    }
    //切换引脚
  global.loraCsPin = global.cs_pin_table[ch];
    //设置发射功率
  if (SX1278_CS_1_PIN == global.loraCsPin)
  {
      power = LORA2W_TX_OUTPUT_POWER;
      paFlag = LORA2W_SX1278_RF_OUT_PIN;
  }
  else
  {
      power = TX_OUTPUT_POWER;
      paFlag = SX1278_RF_OUT_PIN;
  }
    //设置频点
  SX1278SetChannel(freq);
    //配置发送
  SX1278SetTxConfig(MODEM_LORA, power, 0, LORA_BANDWIDTH,
                                LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                LORA_CRC_ON, 0, 0, LORA_IQ_INVERSION_ON,
TX_TIMEOUT_VALUE, paFlag);
    //配置接收
  SX1278SetRxConfig(MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                                LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                                LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                                LORA_FIX_PAYLOAD_LENGTH, LORA_CRC_ON, 0, 0,
LORA_IQ_INVERSION_ON, true);

  /* 设置接收模式，第一块设置为接收完成中断 */
  SX1278SetRx(0, 0);
}
```

5.1.2.1接收配置

```
//函数介绍：SX1278接收配置函数
//参数一modem：  传输类型：[0: FSK, 1: LoRa]
//参数二bandwidth：  宽带设置 配置参数：[0: 125 kHz, 1: 250 kHz,2: 500 kHz, 3:
Reserved]
//参数三datarate：  设置数据速率 [6: 64, 7: 128, 8: 256, 9: 512,10: 1024, 11: 2048,
12: 4096  chips][6-12]
//参数四coderate：  编码数率 [1: 4/5, 2: 4/6, 3: 4/7, 4: 4/8] 参数1-4
//参数五bandwidthAfc：  频带宽度 lora使用时设置0
//参数六preambleLen：  前导码长度
//参数七symbTimeout：  象征超时
//参数八fixLen：  固定长度的数据包 [0: variable 可变的, 1: fixed 固定的]
//参数九payloadLen：有效负载长度 ：设置使用固定长度时的有效负载长度
//参数十crcOn：是否开启CRC校验 [true 开启 , false 不开启]
//参数十一freqHopOn：跳频开关 表示禁用报文内跳频  lora模式[ 0：断开，1：接通] 默认：0
//参数十二hopPeriod：跳频周期 每一跳之间的符号数目 ：[0 就行]
//参数十三iqInverted：IQ配置 [false:不倒转，true：倒转] 一般为：false
//参数十四rxContinuous：是否联系接收 [false: 单个接收, true: 连续接收]
void SX1278SetRxConfig( RadioModems_t modem, uint32_t bandwidth,
                        uint32_t datarate, uint8_t coderate,
                        uint32_t bandwidthAfc, uint16_t preambleLen,
                        uint16_t symbTimeout, bool fixLen,
                        uint8_t payloadLen,
                        bool crcOn, bool freqHopOn, uint8_t hopPeriod,
                        bool iqInverted, bool rxContinuous );
```

5.1.2.2 发送配置

```
//函数介绍：SX1278发送配置函数
//参数一modem：  传输类型：[0: FSK, 1: LoRa]
//参数二power：  设置输出功率
//参数三fdev：  设置频率偏差 lora[0]
//参数四bandwidth：  宽带设置 配置参数：[0: 125 kHz, 1: 250 kHz,2: 500 kHz, 3:
Reserved]
//参数五datarate：  设置数据速率 [6: 64, 7: 128, 8: 256, 9: 512,10: 1024, 11: 2048,
12: 4096  chips][6-12]
//参数六coderate：  编码数率 [1: 4/5, 2: 4/6, 3: 4/7, 4: 4/8] 参数1-4
//参数七preambleLen：  前导码长度
//参数八fixLen：  固定长度的数据包 [0: variable 可变的, 1: fixed 固定的]
//参数九crcOn：是否开启CRC校验 [true 开启 , false 不开启]
//参数十freqHopOn：跳频开关 表示禁用报文内跳频  lora模式[ 0：断开，1：接通] 默认：0
//参数十一hopPeriod：跳频周期 每一跳之间的符号数目 ：[0 就行]
//参数十二iqInverted：IQ配置 [0:不倒转，1：倒转] 一般为0
//参数十三timeout：传输超时时间：ms
//参数十四paFlag：  根据该值配置使用RF0输出还是PA_BOOST 0为RF0,1位PA_BOOST
void SX1278SetTxConfig( RadioModems_t modem, int8_t power, uint32_t fdev,
                        uint32_t bandwidth, uint32_t datarate,
                        uint8_t coderate, uint16_t preambleLen,
                        bool fixLen, bool crcOn, bool freqHopOn,
                        uint8_t hopPeriod, bool iqInverted, uint32_t timeout,
uint8_t paFlag );
```

5.1.3发送和中断处理任务函数

5.1.3.1 发送任务函数

```
void task_radio_send(void *arg)
{
  uint32_t current_tick = xTaskGetTickCount();
    //100ms执行一次
  uint32_t next_delay_tick = us2ticks(100 * 1000);
  uint32_t count = 0;
  uint8_t radio_send[49];
    //等待初始化结束释放
  ulTaskNotifyTake( pdTRUE, portMAX_DELAY);
  for(;;)
  {
    vTaskDelayUntil(&current_tick, next_delay_tick);
    count++;
    if (((count % 50) == 0))
     {
         NRF_LOG_INFO("Dog_Count:~%d$", count);
     }
      //三秒发送一次
    if(((count % 30) == 0))
    {
        //设置发射芯片为第几个
      global.tx_channel_id = 3;
      for(int i = 0; i < 49; i++)
      {
        radio_send[i] = i+150;
      }
      for (count = 0; count < 4; count++)
       {
         if (count == global.tx_channel_id)
         {
           continue;
         }
          //切换片选
         global.loraCsPin = global.cs_pin_table[count];
         //设置为待机模式
          SX1278SetStby();
       }
        //发送数据
      rf_send(radio_tx_item.data, MSG_MAX_LEN);
        //提交发送请求
      rf_send_commit();
        //开启定时器
      xTimerStart(global.timer_handle_radio_li, 0);
    }
  }
}
```

5.1.3.2 中断处理任务函数

```
void task_radio_process_li(void *arg)
{
    BaseType_t ret_code;
    uint8_t count = 0;
```

```
    lora_location_t lora_location_nearby={0};
    bool lora_data_valid_flag=false;
    //初始化四个芯片全部复位下
    rf_init(0, 1); //reset
    rf_init(1, 1);
    rf_init(2, 1);
    rf_init(3, 1);
   //启动看门狗定时器
   vTaskDelay(us2ticks(4 * 1000 * 1000));
    //设置频点和配置接收发送
   rf_channel_set_freq();
    //释放信号量告诉发送任务函数可以发送了
   xTaskNotifyGive(global.task_handle_radio_send);
   while (1)
   {
       //等待信号量中断信号量
     ret_code = xQueueReceive(global.queue_handle_radio_tx, &radio_tx_item,
portMAX_DELAY);
     if (ret_code != pdTRUE)
     {
       continue;
     }
       //判断是什么信号
     switch (radio_tx_item.type)
     {
             //DIO0:产生的信号量
       case 1:
             //判断这个官方的是否 == RF_RX_RUNNING
         if (SX1278.Settings.State == RF_RX_RUNNING)
         {
             //产生的是读取中断但不确定是哪个芯片 所以四个都读一遍
           for (count = 0; count < 4; count++)
           {
               //判断频点是否为0
             if (0 == global.rxch_info[count].freq)
             {
               continue;
             }
               //如果为0就清除数组
             if(count == 0)
             {
               memset(&lora_location_nearby,0,sizeof(lora_location_nearby));
             }
               //切换片选引脚
             global.loraCsPin = global.cs_pin_table[count];
               //
             bool rxFlag = sx1278_multi_ch_rx_li(count,
&rx_data_payload[count]);
             if(!rxFlag) continue;
             if(rx_data_payload[count].payload[0]==0)continue;
             if(global.loraCsPin == 11 && rx_data_payload[count].len == 49 &&
rx_data_payload[count].payload[0] == 0x04 )
             {
               NRF_LOG_ERROR("lora_%d RX OK data[45]:0x%02x.............",
global.loraCsPin,
rx_data_payload[count].payload[45]);
             }
```

```
            if((global.loraCsPin == 12 || global.loraCsPin == 11) &&
rx_data_payload[count].len == 49 && rx_data_payload[count].payload[0] == 0x2C &&
rx_data_payload[count].payload[48] == 0x5C)
            {
              NRF_LOG_ERROR("lora_%d RX OK data[48]:0x%02x.............",
global.loraCsPin, rx_data_payload[count].payload[48]);
            }
            if(global.loraCsPin == 13 && rx_data_payload[count].len == 49 &&
rx_data_payload[count].payload[0] == 0x2C && rx_data_payload[count].payload[48]
== 0x5C)
            {
              NRF_LOG_ERROR("lora_%d RX OK data[48]:0x%02x.............",
global.loraCsPin, rx_data_payload[count].payload[48]);
            }
            if(global.loraCsPin == 14 && rx_data_payload[count].len == 49 &&
rx_data_payload[count].payload[0] == 0x96 && rx_data_payload[count].payload[48]
== 0xC6)
             {
               NRF_LOG_ERROR("lora_%d RX OK data[48]:0x%02x.............",
global.loraCsPin, rx_data_payload[count].payload[48]);
             }
          }
        }
        else if (SX1278.Settings.State == RF_TX_RUNNING)
        {
          xTimerStop(global.timer_handle_radio_li, 0);
          if((SX1278ReadReg(REG_LR_IRQFLAGS) & RFLR_IRQFLAGS_TXDONE) ==
RFLR_IRQFLAGS_TXDONE)
          {
            NRF_LOG_WARNING("LORA TX OK .........");
          }
          for (count = 0; count < 4; count++)
          {
            global.loraCsPin = global.cs_pin_table[count];
            SX1278WriteReg(REG_LR_IRQFLAGS, RFLR_IRQFLAGS_TXDONE);
            if (0 != global.rxch_info[count].freq)
            {
              SX1278SetRx(0, 0);
            }
          }
        }
        break;
      case 4:
        NRF_LOG_WARNING("LORA TX time out .........");
        radio_txto_switch();
        vTaskDelay(us2ticks(1000 * 1000));
        break;
    }
  }
}
```

5.1.4 其他函数

5.1.4.0 接收函数

```
bool sx1278_multi_ch_rx(uint8_t ch, msg_payload_t *p_data)
{
  volatile uint8_t irqFlags = 0;
  uint16_t sum_len = 0;
    //读取寄存器产生的中断是否是RXDONE
  if ((SX1278ReadReg(REG_LR_IRQFLAGS) & RFLR_IRQFLAGS_RXDONE) == 0)
  {
    return false;
  }
    //清除中断
  SX1278WriteReg(REG_LR_IRQFLAGS, RFLR_IRQFLAGS_RXDONE);
  /* 添加读取Fifo的代码，兼容1个或多个SX1278模块 */
   uint16_t len;
   /* 判断是可变还是固定长度 */
   if (SX1278.Settings.LoRa.FixLen == 0)
   {
     len = SX1278ReadReg(REG_LR_RXNBBYTES);
   }
   else
   {
     len = SX1278.Settings.LoRa.PayloadLen;
   }
   if (len != 0)
   {
       //写入缓存区地址
     SX1278WriteReg(REG_LR_FIFOADDRPTR,
SX1278ReadReg(REG_LR_FIFORXCURRENTADDR));
       //读取数据
     SX1278ReadFifo(p_data->payload, len);
   }
    //数据长度等于这个长度
   p_data->len = len;
   p_data->vaild = true;

   return true;
}
```

5.1.4.1 SX1278SetRx

```
//函数介绍：SX1278设置为接收模式
//参数一timeout：设置超时时间 [0: continuous, others timeout] 一般设置：0
//参数二flag：标志位：[0：打开接收中断，1：屏蔽接收中断，2：则打开接收中断，屏蔽VALIDHEADER
中断，3：]
void SX1278SetRx( uint32_t timeout, uint8_t flag)
```

5.1.4.2 SX1278SetTx

```
//函数介绍：在给定的时间内设置SX1278在传输模式
//参数一timeout：设置超时时间 [0: continuous, others timeout] 一般设置：0
//参数二flag：标志位：[0：常发送后，设置发送完成中断，1：不需要设置发送完成中断]
void SX1278SetRx( uint32_t timeout, uint8_t flag)
```

5.1.4.3 SX1278Send

```
//函数介绍：发送函数
//参数一buffer：要发送的数据指针
//参数二size：要发送的数据长度
//参数三flag：调用SX1278SetTx使用到的标志位 默认为:0
void SX1278Send( uint8_t *buffer, uint8_t size, uint8_t flag)
```

5.1.4.4 SendCommit

```
//函数介绍：发送提交
void SX1278SendCommit(void);
//函数
void SX1278SendCommit(void)
{
    //设置状态为RF_TX_RUNNING
    SX1278.Settings.State = RF_TX_RUNNING;
    //设置模式为发送
    SX1278SetOpMode( RF_OPMODE_TRANSMITTER );
}
```

5.1.4.5 SX1278SetChannel

```
//函数介绍：设置通道配置
//参数freq：通道射频频率 等同于sx1280的通道频率
//基站频点470377000, channle: 0
//       470994000, channle: 1
//       471611000, channle: 2
//       472228000, channle: 3
void SX1278SetChannel( uint32_t freq );
```

6.LTE_4G模块

6.0 模块文档参数

开关机						
管脚名	管脚	上电状态	I/O	管脚描述	电气特性	备注
PWRKEY	67,68	INPUT PULL_UP	I	模块开机/关机; 内部上拉到VBAT	VILmax=0.5V	1. VBAT电压域 2. 内部上拉 3. 关机状态下把管脚拉低1.5s以上模块开机 4. 开机状态下把管脚拉低1.5s以上模块关机（具体请参考开机时序图）

模块状态指示						
管脚名	管脚号	上电状态	I/O	管脚描述	电气特性	备注
NET_STATUS (GPIO_1)	58	INPUT PULL_DOWN	O	网络状态指示	VOHmin=1.35V VOLmax=0.45V Iomax=2mA	1.8V 电压域 不用则悬空

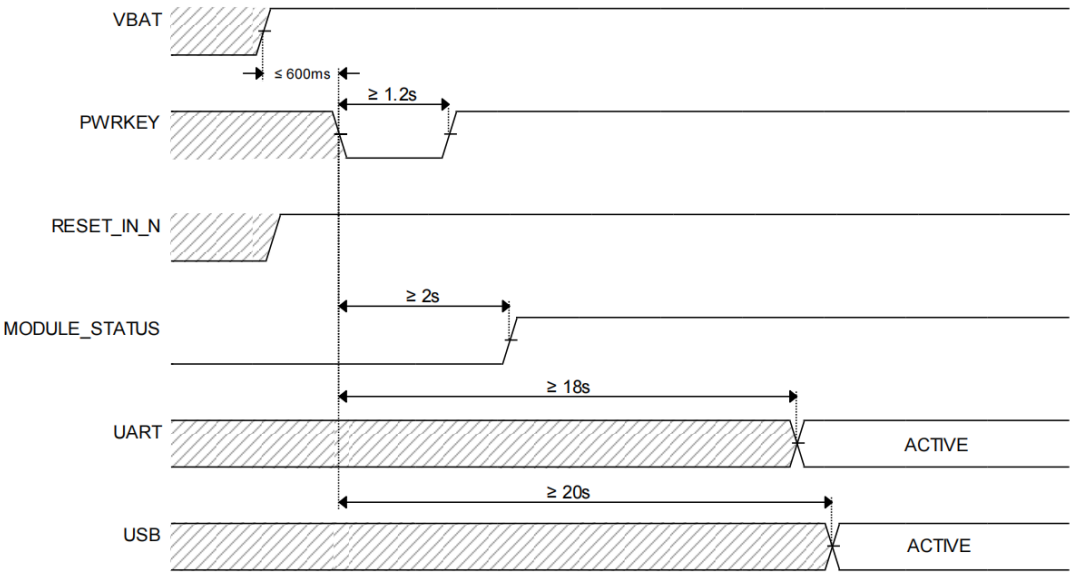
表格 3：工作模式

模式	功能	
正常工作	SLEEP	在模块没有任何任务处理则会自动进入睡眠模式。睡眠模式下，模块的功耗会降到非常低，但模块仍然能够收发数据、短消息和来电。
	IDLE	软件正常运行。模块注册上网络，没有数据，语音和短信交互。
	TALK/Data	连接正常工作。有数据或者语音或者短信交互。此模式下，模块功耗取决于环境信号的强弱，动态 DTX 控制以及射频工作频率。
关机模式	此模式下PMU停止给基带和射频供电，软件停止工作，串口不通，但VBAT管脚依然通电	
最少功能模式(保持供电电压)	此模式下，射频和SIM卡都不工作，但是串口仍然可以访问	
飞行模式	AT+CFUN=4可以将模块设置为飞行模式，此模式下模块射频不工作	

6.0.1开机

```
/*
3.4.1. 开机
Air724UG_Air723UGx系列模块可以通过PWRKEY管脚开机。关机状态下长按开机键一段时间以上，模块会进入开机流程，软件会检测VBAT管脚电压若VBAT管脚电压大于软件设置的开机电压（3.1V），会继续开机动作直至系统开机完成；否则，会停止执行开机动作，系统会关机。

3.4.1.1 PWRKEY 管脚开机
VBAT上电后，PWRKEY管脚可以启动模块，把PWRKEY管脚拉低持续一段时间后（请参考时序图）之后开机，开机成功后PWRKEY管脚可以释放。可以通过检测 V_GLOBAL_1V8 管脚的电平来判别模块是否开机。推荐
使用开集驱动电路来控制PWRKEY管脚

注意事项：
注意：模块在上电后 600ms 内会检测开机关机，600ms 时间内拉低 PWRKEY 且大于 1.2s 能稳定开机。上电 600ms 后如果未检测到开机事件软件会进
入关闭流程，此过程会持续 1 到 2s，如果这个时候拉低 PWRKEY 将不会被检测到。因此，如果不能保证在上电 600ms 内拉低 PWRKEY，为了确保能稳定开
机建议 PWRKEY 时间适当加长，建议 4S 以上。
*/
/*****************上电开机*******************/
/*
将模块的 PWRKEY 直接接地可以实现上电自动开机功能。需要注意，在上电开机模式下，将无法关机，
只要 VBAT 管脚的电压大于开机电压即使软件调用关机接口，模块仍然会再开机起来。另外，在此模式下，要想成功开机起来 VBAT 管脚电压仍然要大于软件设定的开机电压值（3.1V），如果不满足，模块会关闭，就会
出现反复开关机的情况。
*/
```

6.0.1.1 开机时序图



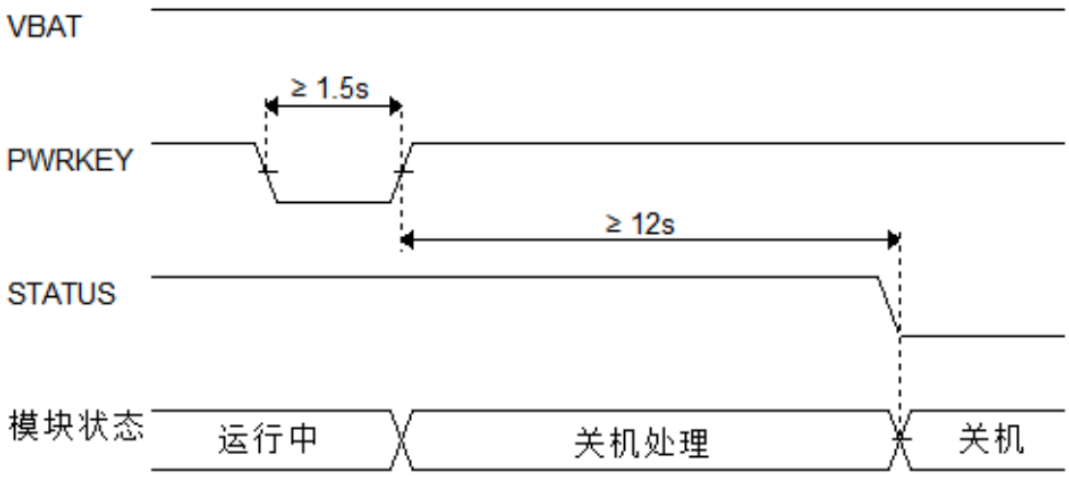
6.0.2关机

```
/*
以下的方式可以关闭模块：
1. 正常关机：使用PWRKEY管脚关机
2. 正常关机：通过AT指令AT+CPOWD关机
3. 低压自动关机：模块检测到低压（3.1V以下）时关机
*/
```

6.0.2.1 使用PWRKEY管脚关机

```
/*
PWRKEY 管脚拉低 1.5s 以上时间，模块会执行关机动作。
关机过程中，模块需要注销网络，注销时间与当前网络状态有关，经测定用时约2s~12s，因此建议延长
12s后再进行断电或重启，以确保在完全断电之前让软件保存好重要数据。
*/
//时序图如下：
```

时序图如下：



6.0.2.2 低电压自动关机

```
/*
模块在运行状态时当 VBAT 管脚电压低于软件设定的关机电压时（默认设置 3V），软件会执行关机动作
关闭模块，以防低电压状态下运行出现各种异常。
*/
```

RTOS

头文件

```
//RTOS头文件          #include "FreeRTOS.h"
//任务函数头文件       #include "task.h"
//队列头文件          #include "queue.h"
//信号量互斥量头文件   #include"semphr.h"
```

使用建议

```
/*
1.每个任务函数中最好不要耦合性太强
2.每个任务函数可以通过信号量、消息队列、事件标志组来通信，可以降低耦合度，减少出错率
*/
```

1.任务函数

```
/*
 FreeRTOS 的任务可认为是一系列独立任务的集合。每个任务在自己的环境中运行。在任何时刻，只有一个
任务得到运行，FreeRTOS 调度器决定运行哪个任务。调度器会不断的启动、停止每一个任务，宏观看上去所
有的任务都在同时在执行。作为任务，不需要对调度器的活动有所了解，在任务切入切出时保存上下文环境
（寄存器值、堆栈内容）是调度器主要的职责。为了实现这点，每个 FreeRTOS 任务都需要有自己的栈空
间。
 FreeRTOS 中的任务是抢占式调度机制，高优先级的任务可打断低优先级任务，低优先级任务必须在高优先
级任务阻塞或结束后才能得到调度。同时 FreeRTOS 也支持时间片轮转调度方式，只不过时间片的调度是不
允许抢占任务的 CPU 使用权。
任务通常会运行在一个死循环中，也不会退出，如果一个任务不再需要，可以调用FreeRTOS 中的任务删除
API 函数接口显式地将其删除。
*/
```

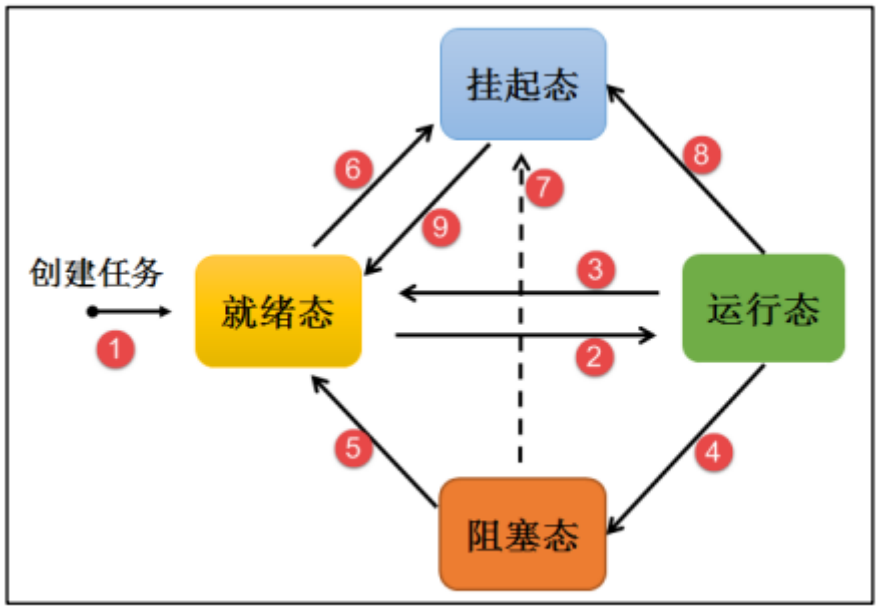


图 16-1 任务状态迁移图

```
/*
任务状态迁移
FreeRTOS 系统中的每一个任务都有多种运行状态

(1)：创建任务→就绪态（Ready）：任务创建完成后进入就绪态，表明任务已准备就绪，随时可以运行，只等待调度器进行调度。

(2)：就绪态→运行态（Running）：发生任务切换时，就绪列表中最高优先级的任务被执行，从而进入运行态。

(3)：运行态→就绪态：有更高优先级任务创建或者恢复后，会发生任务调度，此刻就绪列表中最高优先级任务变为运行态，那么原先运行的任务由运行态变为就绪态，依然在就绪列表中，等待最高优先级的任务运行完毕继续运行原来的任务（此处可以看做是 CPU 使用权被更高优先级的任务抢占了）。

(4)：运行态→阻塞态（Blocked）：正在运行的任务发生阻塞（挂起、延时、读信号量等待）时，该任务会从就绪列表中删除，任务状态由运行态变成阻塞态，然后发生任务切换，运行就绪列表中当前最高优先级任务。

(5)：阻塞态→就绪态：阻塞的任务被恢复后（任务恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的任务会被加入就绪列表，从而由阻塞态变成就绪态；如果此时被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务将再次转换任务状态，由就绪态变成运行态。

(6) (7) (8)：就绪态、阻塞态、运行态→挂起态（Suspended）：任务可以通过调用 vTaskSuspend() API 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到CPU 的使用权，也不会参与调度，除非它从挂起态中解除。

(9)：挂起态→就绪态：把一个挂起状态的任务恢复的唯一途径就是调用 vTaskResume() 或 vTaskResumeFromISR() API 函数，如果此时被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务将再次转换任务状态，由就绪态变成运行态。
*/
```

1.1声明任务句柄

```
/*头文件*/
#include "task.h"
TaskHandle_t app_task1_handle=NULL;//要声明任务句柄
```

1.2创建任务

```
//任务原型
//参数一：任务函数名
//参数二：任务的名字测试用 同任务函数名一样即可
//参数三：分配给任务函数的堆栈大小
//参数四：传递给任务函数的参数 没有可为NULL
//参数五：任务优先级0~configMAX_PRIORITIES-1 优先级越小优先级越高
//参数六：任务句柄
BaseType_t xTaskCreate(TaskFunction_t pxTaskCode,
                       const char * const pcName,
                       const uint16_t usStackDepth,
                       void * const pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t * const pxCreatedTask)
/* 创建任务1
app_task1，任务函数
"app_task1"，任务的名字
512，分配给任务1的内存（栈）大小，单位为字
7,优先级为7，数值不能超过（configMAX_PRIORITIES -1）
app_task1_handle任务句柄
*/
xReturn=xTaskCreate(app_task1,"app_task1",512,&app_task1_arg,7,&app_task1_handle
);
/*返回值
1. pdTRUE
表明任务创建成功。
2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY
由于内存堆空间不足，FreeRTOS 无法分配足够的空间来保存任务
*/
```

1.3创建任务函数

```
//任务函数
void app_task1(void *pvParameters)//void类型 参数为void *
{
    //必须死循环 while(1)或for(;;)
    while(1)
    {
        /* 延时1000ms */
        vTaskDelay(1000);//使用RTOS延时释放cpu权限让别的任务调用
        printf("[app_task1] running ...\r\n");
    }
}
```

1.4开启任务调度

```
/* 开启任务调度 */
vTaskStartScheduler();
```

1.5删除任务

```
//删除任务  例如初始化任务函数
 vTaskDelete(Start_Task_Handle);
```

2.挂起恢复

```
/*
vTaskSuspend()
任务可以通过调用 vTaskSuspend()函数都可以将处于任何状态的任务挂起，被挂起的任务得不到 CPU 的使用权，也不会参与调度，它相对于调度器而言是不可见的，除非它从挂起态中解除。

vTaskSuspendAll()
将所有的任务都挂起，就是挂起任务调度器。调度器被挂起后则不能进行上下文切换，但是中断还是使能的。当调度器被挂起的时候，如果有中断需要进行上下文切换， 那么这个中断将会被挂起，在调度器恢复之后才响应这个中断。

vTaskResume()
任务恢复就是让挂起的任务重新进入就绪状态，恢复的任务会保留挂起前的状态信息，在恢复的时候根据挂起时的状态继续运行。

xTaskResumeFromISR()
xTaskResumeFromISR()与 vTaskResume()一样都是用于恢复被挂起的任务，不一样的是 xTaskResumeFromISR() 专门用在中断服务程序中。无论通过调用一次或多次vTaskSuspend()函数而被挂起的任务，也只需调用一次xTaskResumeFromISR()函数即可解挂。要想使用该函数必须在 FreeRTOSConfig.h中把INCLUDE_vTaskSuspend和INCLUDE_vTaskResumeFromISR 都定义为 1 才有效。任务还没有处于挂起态的时候，调用xTaskResumeFromISR()函数是没有任何意义的
*/
```

2.1挂起任务

```
/*********** 挂起任务，任务2为挂起态**********/
vTaskSuspend(app_task2_handle);
```

2.2恢复任务

```
/* **********恢复任务 ，任务2为就绪态***********/
vTaskResume(app_task2_handle);
```

3.临界区

```
/*
一旦这部分代码开始执行，则不允许任何中断打扰。为确保临界段代码的执行，在进入临界段之前要关中断，而临界段代码执行完以后要立即开中断。
*/
```

3.1进入临界区

```
/*任务和抢占优先级较低的硬件中断无法打断其执行*/
taskENTER_CRITICAL();
```

3.2退出临界区

```
/* 退出临界区，告诉内核已经保护完毕 */
taskEXIT_CRITICAL();
```

4.消息队列

```
/*
消息队列传递的是实际数据，并不是数据地址，RTX，uCOS-II 和 uCOS-III 是传递的地址）放入到队列。
同样，一个或者多个任务可以通过 RTOS 内核服务从队列中得到消息。通常，先进入消息队列的消息先传
给任务，也就是说，任务先得到的是最先进入到消息队列的消息，即先进先出的原则（FIFO），FreeRTOS
的消息队列支持 FIFO 和 LIFO 两种数据存取方式。
使用消息队列可以让 RTOS 内核有效地管理任务，而全局数组是无法做到的，任务的超时等机制需要用户自己
去实现。
使用了全局数组就要防止多任务的访问冲突，而使用消息队列则处理好了这个问题，用户无需担心。
使用消息队列可以有效地解决中断服务程序与任务之间消息传递的问题。
*/
```

4.1申请初始化消息队列

```
/*头文件*/
#include "queue.h"
// 定义队列句柄变量
QueueHandle_t xQueue;
// 申请队列
// 参数 1 : 队列深度
// 参数 2 ： 队列项内容大小
xQueue = xQueueCreate( 10, sizeof( unsigned long ) );
```

4.2消息队列发送数据

```
/*****************普通任务函数发送消息***********************/
/*创建要发送的消息类型，类型需与申请的一致*/
unsigned long pxMessage;
/*普通发送消息*/
// 发送消息
// 参数 1 : 队列句柄
// 参数 2 ： 队列内容指针
// 参数 3 ： 允许阻塞时间
//返回值：如果发送到队列成功，则为 pdPASS，否则为 errQUEUE_FULL。
xQueueSend( xQueue,(void *)&pxMessage, ( TickType_t ) 0 );
/*****************中断发送消息***********************/
/*此函数是用于中断服务程序中调用
第 1 个参数是消息队列句柄。
第 2 个参数要传递数据地址，每次发送都是将消息队列创建函数 xQueueCreate 所指定的单个消息大小复
制到消息队列空间中。
```

```
第3个参数用于保存是否有高优先级任务准备就绪。如果函数执行完毕后，此参数的数值是pdTRUE，说明有高优先级任务要执行，否则没有。
pdTRUE：恢复运行的任务的优先级 >= 正在运行的任务（被中断打断的任务），这意味着在退出中断服务函数后，必须进行一次上下文切换。
pdFALSE：恢复运行的任务的优先级 < 当前正在运行的任务（被中断打断的任务），这意味着在退出中断服务函数后，不需要进行上下文切换。
返回值，如果发送到队列成功，则为 pdPASS，否则为 errQUEUE_FULL。
*/
BaseType_t xQueueSendFromISR
    (
    QueueHandle_t xQueue, /* 消息队列句柄 */
    const void *pvItemToQueue, /* 要传递数据地址 */
    BaseType_t *pxHigherPriorityTaskWoken /* 高优先级任务是否被唤醒的状态保存 */
    );
//例如：
/*发送数据
参数一：个参数是消息队列句柄
参数二：存储数据的地址
参数三：声明的变量  参数三可为NULL
返回值：pdPASS 发送成功 pdTRUE和pdPASS均是发送成功宏定义
*/
//例子一：
//声明变量 高优先级任务是否被唤醒的状态保存  如果不需要可不定义设置为NULL
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
if(pdPASS = xQueueSendFromISR(xQueue1,(void
*)&g_uiCount,&xHigherPriorityTaskWoken));
/* 如果xHigherPriorityTaskWoken = pdTRUE，那么退出中断后切到当前最高优先级任务执行 */
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
//例子二：
if(pdPASS = xQueueSendFromISR(xQueue1,(void *)&g_uiCount,NULL));
```

4.3消息队列接受消息

```
/*****************普通任务函数接收消息***********************/
/*创建要接受的消息类型，类型需与申请的一致*/
unsigned long pxMessage;
// 接收消息
// 参数 1 : 队列句柄
// 参数 2 :  队列内容返回保存指针
// 参数 3 :  允许阻塞时间  portMAX_DELAY一直等待
//pdTRUE表示接收到了消息
if( pdTRUE == xQueueReceive( xQueue, &( pxMessage ), ( TickType_t ) 10 ) )
/*****************中断接收消息***********************/
/*此函数是用于中断服务程序中调用
第 1 个参数是消息队列句柄。
第 2 个参数指向缓冲区的指针，接收的项目将复制到该缓冲区中
第 3 个参数用于保存是否有高优先级任务准备就绪。如果函数执行完毕后，此参数的数值是pdTRUE，说明有高优先级任务要执行，否则没有。
返回值，如果发送到队列成功，则为 pdPASS，否则为 errQUEUE_FULL。
*/
BaseType_t xQueueReceiveFromISR
    (
    QueueHandle_t xQueue,
    无效 *pvBuffer,
    BaseType_t *pxHigherPriorityTaskWoken
    );
//例子
```

```
//声明变量 高优先级任务是否被唤醒的状态保存  如果不需要可不定义设置为NULL
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
/*发送数据
参数一：个参数是消息队列句柄
参数二：存储数据的地址
参数三：声明的变量 参数三可为NULL
返回值：pdPASS  发送成功 pdTRUE和pdPASS均是发送成功宏定义
*/
//例子一：
if(pdPASS == xQueueReceiveFromISR(xQueue,&buf, &xHigherPriorityTaskWoken ));
//例子二：
if(pdPASS == xQueueReceiveFromISR(xQueue,&buf, NULL ));
```

4.4查询消息个数

```
/*
uxQueueMessagesWaiting
用于查询队列中当前有效数据单元个数。切记不要在中断服务例程中调用 uxQueueMessagesWaiting()。
应当在中断服务中
使用其中断安全版本 uxQueueMessagesWaitingFromISR()。
*/
//零表示队列为空
UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue )
UBaseType_t uxQueueMessagesWaitingFromISR( const QueueHandle_t xQueue )
```

4.5注意事项

```
/*
1.中断发送读取必须在中断函数中 优先级参数可设置为NULL
2.中断只需要设置一次尽量不要放在任务函数中
3.任务函数的接收无论是接受中断发送的还是普通发送都只能使用普通接收函数
*/
```

5.信号量

5.1创建信号量

5.1.1创建二值型信号量

```
//声明二值信号量句柄
SemaphoreHandle_t xSemaphore;
/* 创建二值型信号量 旧版创建*/
// 返回值NULL: 二值信号量创建失败。
vSemaphoreCreateBinary(xSemaphore);

/* 创建二值型信号量 新版创建*/
// 返回值NULL: 二值信号量创建失败。
xSemaphore = xSemaphoreCreateBinary();
```

5.1.2创建计数信号量

```
/*函数原型
返回值：NULL表示创建失败
参数一：信号量可以计数的最大值
参数二：计数初始值
*/
xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
                                                UBaseType_t uxInitialCount);
//例如：
//声明计数信号量句柄
xSemaphoreHandle xSemaphore;
// 信号量可以计数的最大值为10,计数初始值为0.
xSemaphore = xSemaphoreCreateCounting( 10, 0 );
if( xSemaphore != NULL )
{
    // 信号量创建成功
    // 现在可以使用信号量了。
}
```

5.1.3创建互斥量

```
//函数原型  返回值为NULL表示创建失败 互斥量不可在中断服务函数中使用
SemaphoreHandle_t xSemaphoreCreateMutex( void );
//例如：
//声明互斥信号量句柄
SemaphoreHandle_t xSemaphore;
/* 创建一个互斥类型的信号量。*/
xSemaphore = xSemaphoreCreateMutex();

if( xSemaphore != NULL )
{
    /* 信号量创建成功并且
       可以使用。*/
}
```

5.2释放信号量

```
/****普通任务释放信号量****/
/*
返回值：成功返回：pdPASS 失败返回：err_QUEUE_FULL
参数：要释放的信号量句柄
不可在中断服务中使用该函数
*/
xSemaphoreGive(xSemaphore );
/*******中断释放信号量*******/
/*
返回值：成功返回：pdPASS 失败返回：err_QUEUE_FULL
参数一：要释放的信号量句柄
参数二：如果*pxHigherPriorityTaskWoken为pdTRUE，则需要在中断退出前人为的经行一次上下文切换。从FreeRTOS V7.3.0开始，该参数为可选参数，并可以设置为NULL
*/
xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,BaseType_t
*pxHigherPriorityTaskWoken );
```

5.3获取信号量

```
/*****普通获取信号量******/
/*
返回值：失败返回：pdFALSE 成功返回：pdPASS
参数一：要获取的信号量句柄
参数二：阻塞时间，信号量无效时等待时间
为portMAX_DELAY则一直阻塞无限等待；设置0表示不设置等待时间
*/
 xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
/****中断获取信号量****/
/*
返回值：失败返回：pdFALSE 成功返回：pdPASS
参数一：要获取的信号量
参数二：用户只需要提供一 个变量来保存这个值就行了；7.3.0可设置为NULL
*/
xSemaphoreTakeFromISR( SemaphoreHandle_t xSemaphore,signed BaseType_t
*pxHigherPriorityTaskWoken );
```

5.4对其他信号量的操作

```
/* 查询信号量的数目 */
UBaseType_t uxSemaphoreGetCount( SemaphoreHandle_t xSemaphore );
/* 查询拥有互斥锁的任务句柄 */
TaskHandle_t xSemaphoreGetMutexHolder( SemaphoreHandle_t xMutex );
```

5.5注意事项

```
/*
1.互斥信号量必须由同一个任务申请，同一个任务释放，其他任务释放无效
2.二值型信号量，一个任务申请成功后可以由另一个任务释放
3.互斥信号量必须归还
*/
```

6.软件定时器

6.1初始化软件定时器

```
//创建定时器句柄
TimerHandle_t timer_handle;
//函数原型
//参数一：定时器名称方便识别不同的定时器
//参数二：定时多长时间
//参数三：选择周期模式还是单次模式，若参数为pdTRUE，则表示选择周期模式，若参数为pdFALSE，则表
示选择单次模式。
//参数四：定时器id，可选NULL
//参数五：定时器结束调用的回调函数
TimerHandle_t xTimerCreate(
    const char * const pcTimerName,  //定时器名称（调试用）
    const TickType_t xTimerPeriodInTicks,//周期（单位tick）
    const UBaseType_t uxAutoReload,//自动装载（pdTURE）
    void * const pvTimerID,//定时器ID，可以判断是哪一个定时器
```

```
    TimerCallbackFunction_t pxCallbackFunction //回调函数
)
//示例
timer_handle = xTimerCreate("timer_handle", ms2ticks(5000), pdFALSE, NULL,
timer_handle_timeout);
```

6.2回调函数

```
//定时器回调函数
//参数固定
void timer_handle_timeout(TimerHandle_t xTimer);
//示例
void timer_handle_timeout(TimerHandle_t xTimer)
{
    //要执行的代码
}
```

6.3软件定时器启动

```
//成功pdPASS，队列已满pdFAIL
//参数二：为portMAX_DELAY则一直阻塞无限等待；设置0表示不设置等待时间
BaseType_t xTimerStart(
    TimerHandle_t xTimer,//定时器句柄
    TickType_t xToclsToWait//阻塞等待时间（内部是消息队列）
)
//示例
xTimerStart(timer_handle,0);
//中断中启动软件定时器
BaseType_t xTimerStartFromISR( TimerHandle_t xTimer, BaseType_t
*pxHigherPriorityTaskWoken )
//示例
//成功pdPASS，队列已满pdFAIL
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken )
```

6.4软件定时器停止

```
//成功pdPASS，队列已满pdFAIL
//停止计时器让其进入休眠
BaseType_t xTimerStop( TimerHandle_t xTimer,
                       TickType_t xBlockTime )
//示例
xTimerStop(xTimer, 0);
//在中断中停止计时器
BaseType_t xTimerStopFromISR( TimerHandle_t xTimer, BaseType_t
*pxHigherPriorityTaskWoken )
//示例
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
xTimerStopFromISR(xTimer,&xHigherPriorityTaskWoken)
```

6.5重启软件定时器

```
//重启软件定时器
//如果定时器已经启动，重新计算时间
//如果没有启动，则启动
//参数二：为portMAX_DELAY则一直阻塞无限等待；设置0表示不设置等待时间
BaseType_t xTimerReset(
    TimerHandle_t xTimer,//定时器句柄
    TickType_t xToclsToWait//阻塞等待时间（内部是消息队列）
)
//示例：
xTimerReset(timer_handle,0);
```

6.6获取定时器ID

```
//获取定时器ID pvTimerGetTimerID()
void* pvTimerGetTimerID(TimerHandle_t xTimer);
//示例
pvTimerGetTimerID(timer_handle);
```

6.7更改定时器周期

```
//如果定时器没启动，会启动定时器
//参数二：定时多长时间
//参数三：为portMAX_DELAY则一直阻塞无限等待；设置0表示不设置等待时间
BaseType_t xTimerChangePeriod(
    TimerHandle_t xTimer,//定时器句柄
    TickType_t xMew{eropd,//新的定时周期
    TickType_t xToclsToWait//阻塞等待时间
)
//示例
xTimerChangePeriod)(timer_handle, 1000, 0);
```

7.任务通知

7.1发送通知

```
//参数一：任务句柄
//参数二：忽略
//参数三：忽略
BaseType_t xTaskNotify( TaskHandle_txTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction);
//宏替换函数：
xTaskNotifyGive(TaskHandle_txTaskToNotify);
//示例：
/*向prvTask2(),发送通知，使其解除阻塞状态 */
 xTaskNotifyGive( xTask2 );
```

7.2接收通知

```
/* 等待prvTask2()的通知，进入阻塞 */
ulTaskNotifyTake( pdTRUE, portMAX_DELAY);
```

8. 时间函数

8.1 延时函数

```
//函数原型 vTaskDelay()
//相对延时 延时时间可能跟时间设定时间不同 要考虑优先级
void vTaskDelay(const TickType_t xTicksToDelay ); /* 延迟时间长度 */
//示例：vTaskDelay(500);
//绝对延时 vTaskDelayUntil()
//参数一：存储任务上次处于非阻塞状态时刻的变量地址
//参数二：周期性延迟时间
//注意：使用此函数需要在 FreeRTOSConfig.h 配置文件中配置如下宏定义为 1
// #define INCLUDE_vTaskDelayUntil 1
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime,
                      const TickType_t xTimeIncrement );
//示例：
//设置存储系统时间的变量
TickType_t xLastWakeTime;
//获取当前系统时间
xLastWakeTime = xTaskGetTickCount();
/* vTaskDelayUntil是绝对延迟，vTaskDelay是相对延迟。*/
vTaskDelayUntil(&xLastWakeTime, 100);
```

8.2 获取时钟

```
//在任务中获取时间节拍数
volatile TickType_t xTaskGetTickCount( void );
//在中断中获取时间节拍数
volatile TickType_t xTaskGetTickCountFromISR( void );
```

9. 事件标志组

注意事项

```
/*
同时要在 FreeRTOSConfig.h
文件中使能如下三个宏定义：
#define INCLUDE_xEventGroupSetBitFromISR 1
#define configUSE_TIMERS 1
#define INCLUDE_xTimerPendFunctionCall 1
*/
```

9.1 创建事件标志组

```
//创建事件标志组 内存自动分配
//返回值为NULL创建失败
EventGroupHandle_t xEventGroupCreate( void )
//创建事件标志组 内存自己分配
//参数：指向一个StaticEventGroup_t类型的变量，用来保存时间组结构体
//返回值为NULL创建失败
EventGroupHandle_t xEventGroupCreateStatic( StaticEventGroup_t
*pxEventGroupBuffer )
//示例：
//创建事件标志组句柄
EventGroupHandle_t EventGroupHandle;    // 事件标志组句柄
//创建事件标志组
EventGroupHandle =  xEventGroupCreate();
if(NULL == EventGroupHandle)
{
    printf("创建任务事件组失败");
}
```

9.2 设置标志位

```
/*
标志位代表某个事件的状况
*/
//示例：
// 定义事件位
#define BIT_0    (1<<0)
#define BIT_1    (1<<1)
#define BIT_2    (1<<2)   //后面还有其他事件可以此类推加最多24
```

9.3 设置事件位

```
/*
所有设置事件位函数成功返回：pdPASS
                失败返回：pdFALSE：
*/
```

9.3.1任务函数清零

```
//函数原型：
//参数一：事件标志组句柄
//参数二：置零的事件位
EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup,
                                  const EventBits_t uxBitsToClear )
//示例：
//将事件标志组中的BIT_0 置零
xEventGroupSetBits(EventGroupHandle,BIT_0);
```

9.3.2 中断清零

```
//函数原型：
//参数一：事件标志组句柄
//参数二：置零的事件位
BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t xEventGroup,
                                        const EventBits_t uxBitsToSet )
//示例：
//将事件标志组中的BIT_0 置零
xEventGroupClearBitsFromISR(EventGroupHandle,BIT_1);
```

9.3.3任务函数置一

```
//函数原型：
//参数一：事件标志组句柄
//参数二：置一的事件位
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, /* 事件标志组句柄
*/
                                const EventBits_t uxBitsToSet ); /* 事件标志位设置
*/
//示例：
//将事件标志组中的BIT_0 置一
xEventGroupSetBits(EventGroupHandle,BIT_2);
```

9.3.4中断置一

```
//函数原型：
//参数一：事件标志组句柄
//参数二：置一的事件位
//参数三：标记退出此函数以后是否进行任务切换
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,
                                      const EventBits_t uxBitsToSet,
                                      BaseType_t *pxHigherPriorityTaskWoken )
//示例：
//将事件标志组中的BIT_0 置一
BaseType_t pxHigherPriorityTaskWoken = pdFALSE;
BaseType_t xResult;
xResult =
xEventGroupSetBitsFromISR(EventGroupHandle,BIT_2,&pxHigherPriorityTaskWoken);
if(xResult == pdPASS)
{
    //如果发送成功 判断是否有高优先级任务准备就绪，如果有则高优先级
    portYLELD_FROM_ISR(pxHigherPriorityTaskWoken);
}
```

9.4 获取事件标志组值

9.4.1任务函数获取

```
//函数原型
EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup )
//示例：
//创建存储变量
EventBits_t uxBits;
uxBits = xEventGroupGetBits(EventGroupHandle);
```

9.4.2 中断中获取

```
//任务原型
EventBits_t xEventGroupGetBitsFromISR( EventGroupHandle_t xEventGroup )
//示例：
//创建存储变量
EventBits_t uxBits;
uxBits = xEventGroupGetBitsFromISR(EventGroupHandle);
```

9.5 等待事件

```
//函数原型：  只能在任务函数中调用
//参数二：标志位可设置多个用或例如：BIT_0 | BIT_1
//参数三：等待结束是否将等待的标志位置零  置零则 pdTRUE 不置零则 pdFALSE
//参数四：是否等待所有的标志位被设置  等待则所有事件触发才等待成功  与或 pdFALSE为不等待有一个事件触发则返回  pdTRUE则等待所有事件触发
//参数五：等待时间portMAX_DELAY则一直等待
//返回值：为触发的事件
EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup, /* 事件标志组句柄 */
                                const EventBits_t uxBitsToWaitFor, /* 等待被设置的事件标志位 */
                                const BaseType_t xClearOnExit, /* 选择是否清零被置位的事件标志位 */
                                const BaseType_t xWaitForAllBits, /* 选择是否等待所有标志位都被设置 */
                                TickType_t xTicksToWait ); /* 设置等待时间 */
//示例：
//定义存储触发事件的变量
EventBits_t uxBits;
//等待事件函数
//等待EventGroupHandle中的BIT_0或者BIT_1触发  一直等待  触发以后则将该BIT位置零
uxBits = xEventGroupWaitBits(EventGroupHandle,
                             BIT_0 | BIT_1,//等待bit0和bit1（与同步）  等待bit0或bit1置1（或同步）
                             pdTRUE,
                             pdFALSE,//或同步，只要其中一个bit置1，就直接返回
                             portMAX_DELAY//一直等待
                             );
//示例二：
//等待EventGroupHandle中的BIT_0和BIT_1触发  两个触发则返回  一直等待  触发以后则将该BIT位置零
uxBits = xEventGroupWaitBits(EventGroupHandle,
                             BIT_0 | BIT_1,//等待bit0和bit1（与同步）  等待bit0或bit1置1（或同步）
                             pdTRUE,
                             pdTRUE,//与同步，所有置一，就直接返回
                             portMAX_DELAY//一直等待
                             );
```