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)
            {
```

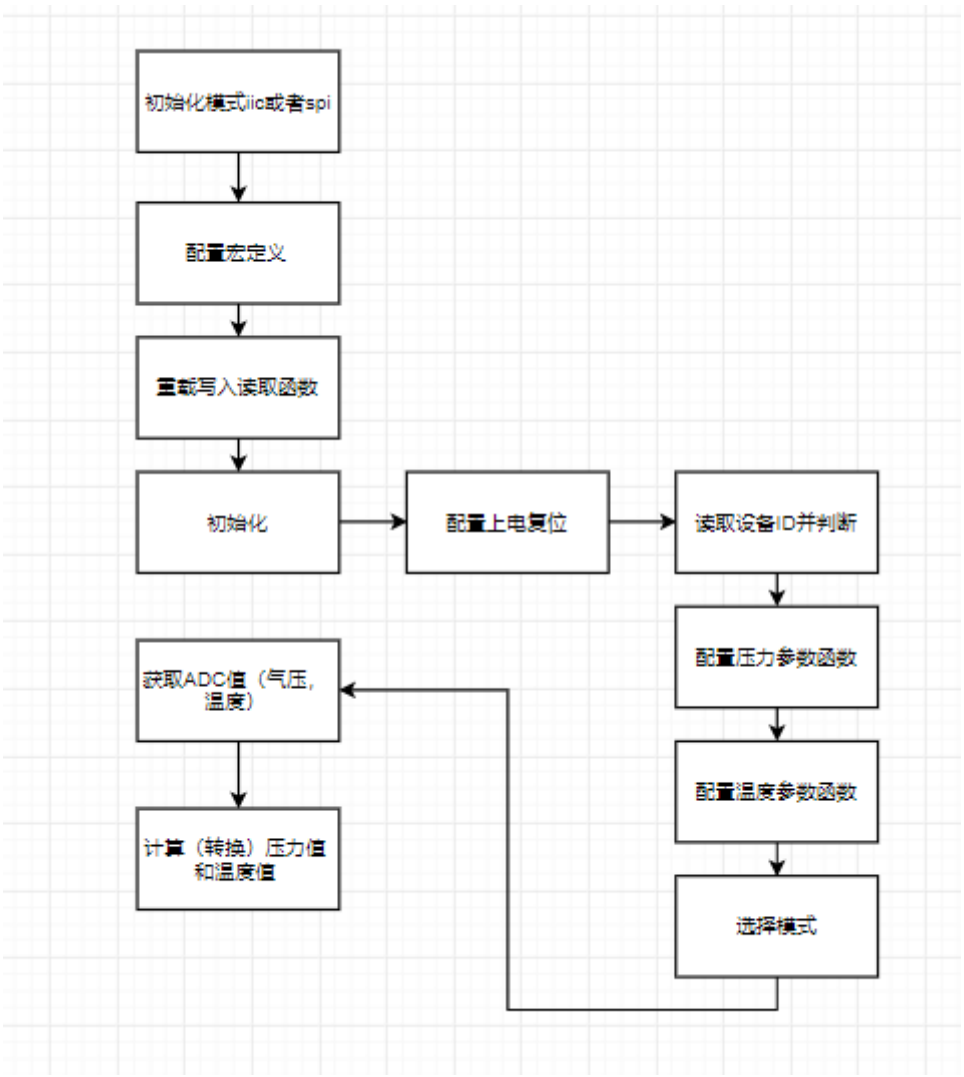
```
            NRF_LOG_ERROR("INT1_SOURCE XH");
        }
        if(int1_source & 0x01)
        {
            NRF_LOG_ERROR("INT1_SOURCE XL");
        }
```

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 SPO6_PSR_B2     0x00        //气压值
#define SPO6_PSR_B1     0x01
#define SPO6_PSR_B0     0x02
#define SPO6_TMP_B2     0x03        //温度值
#define SPO6_TMP_B1     0x04
#define SPO6_TMP_B0     0x05

#define SPO6_PSR_CFG    0x06        //气压测量配置
#define SPO6_TMP_CFG    0x07        //温度测量配置
#define SPO6_MEAS_CFG   0x08        //测量模式配置

#define SPO6_CFG_REG    0x09
#define SPO6_INT_STS    0x0A
#define SPO6_FIFO_STS   0x0B

#define SPO6_RESET      0x0C
#define SPO6_ID         0x0D

#define SPO6_COEF       0x10        //-0x21
#define SPO6_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;
```

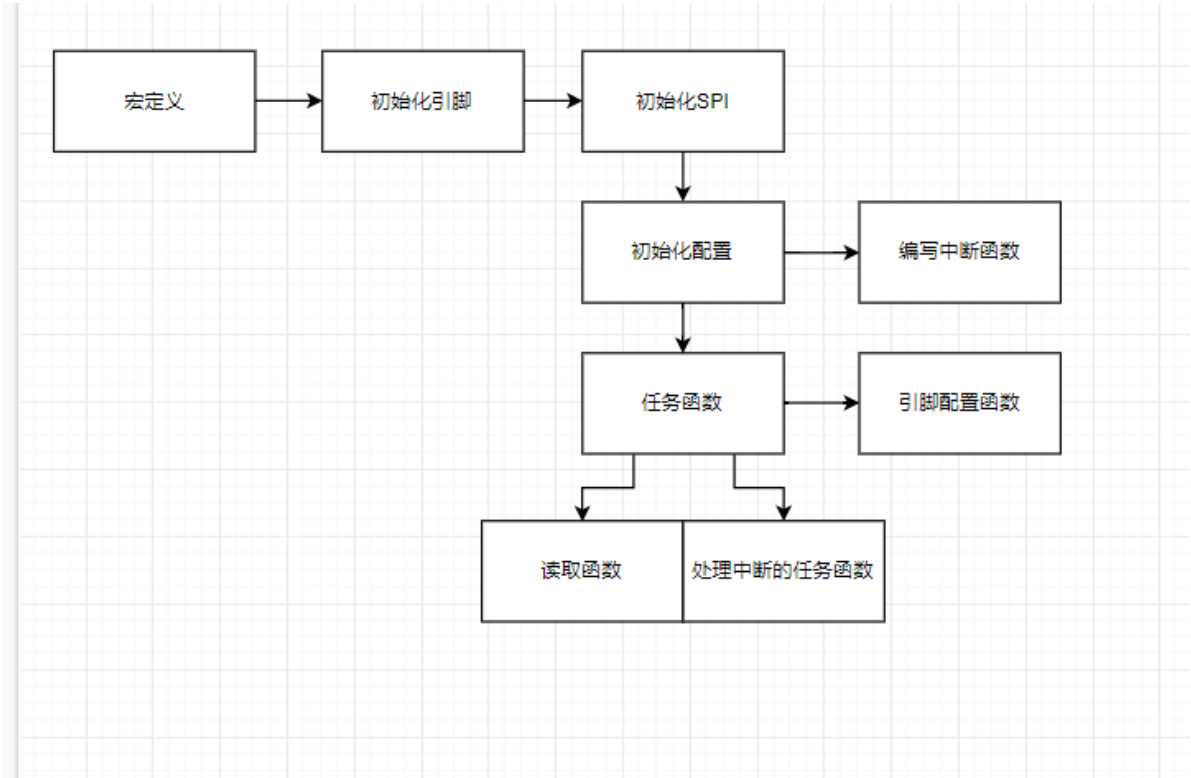
```
        //计算气压
        qua2 = dev->calib_param.C10 + Praw_src * (dev->calib_param.C20 +
Praw_src * dev->calib_param.C30);
        qua3 = Traw_src * Praw_src * (dev->calib_param.C11 + Praw_src * dev-
>calib_param.C21);
        //原dev->current_pressure_val = (uint16_t)((dev->calib_param.C00 +
Praw_src * 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

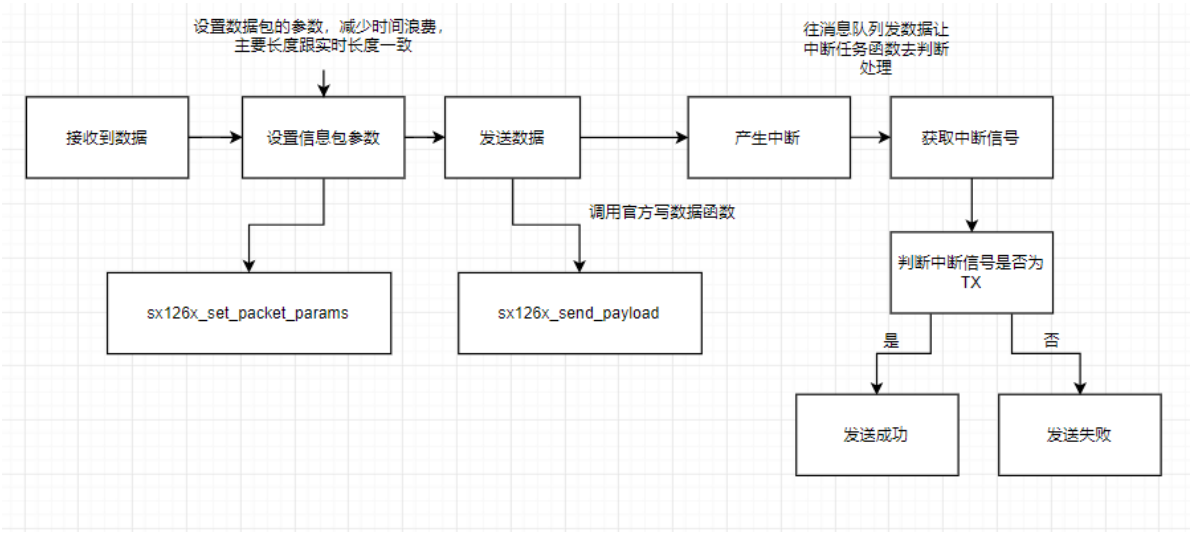
```
void task_sx1268_dio()
{
     /* 等待task_sx1268的通知，进入阻塞 */
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    msg_t msg;
    NRF_LOG_INFO("1268_1 task start");

    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

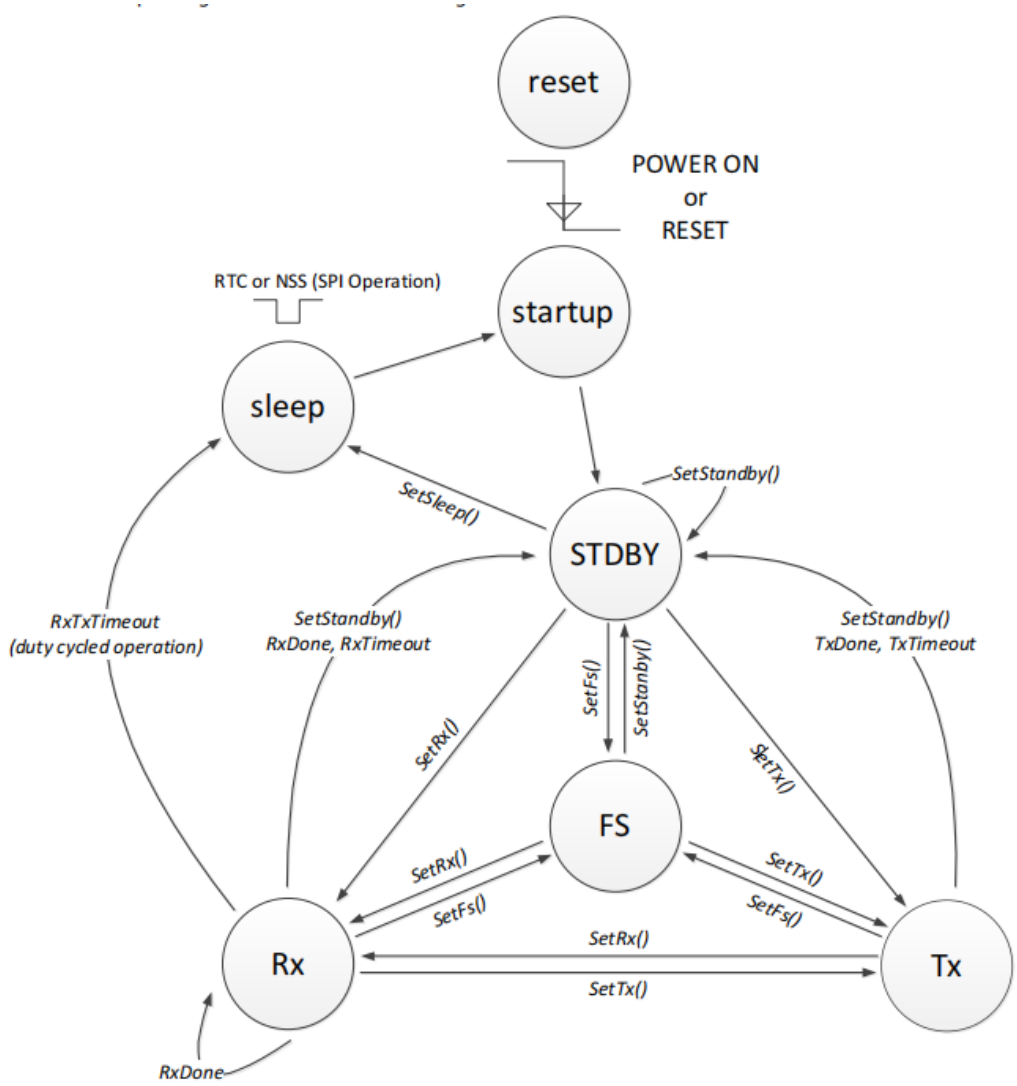
For more information on how to setup IRQ and DIOs, refer to the function *SetDioIrqParams()* in Section 13.3.1 "SetDioIrqParams" on page 77.

```
//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
*/
```

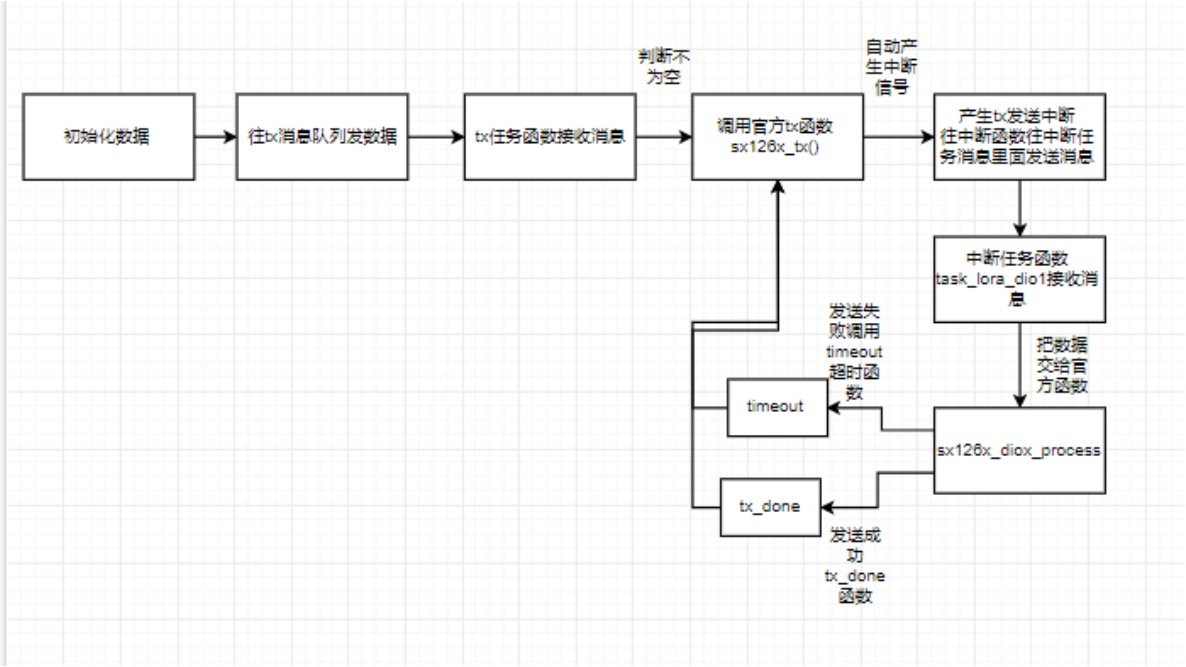
测试问题

```
/*
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
*/
```

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

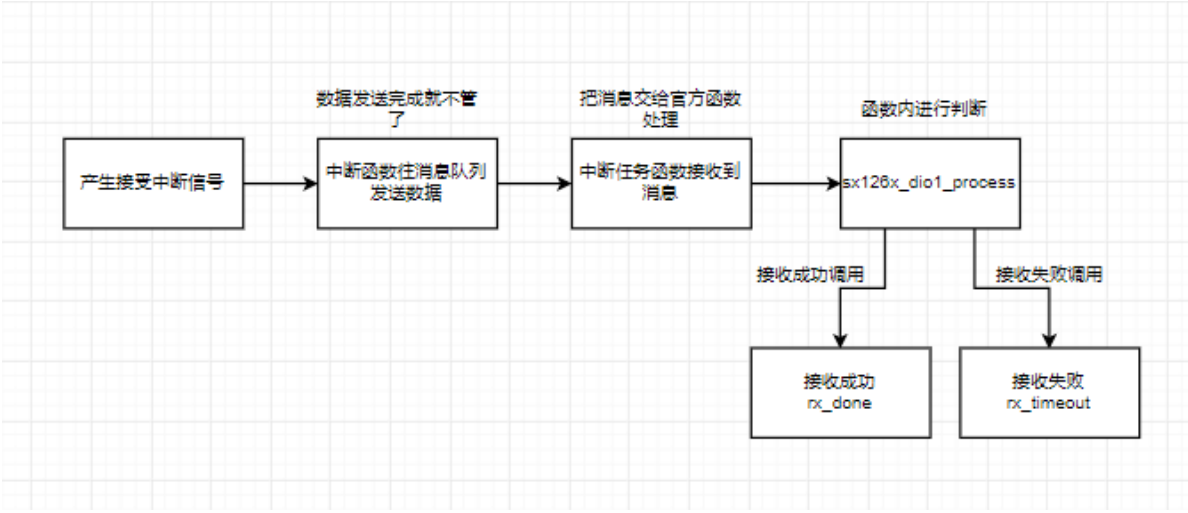
```
void lora_tx_done()
{
  //发送任务通知
  xTaskNotifyGive(global.task_handle_lora_tx_li);
  // 发送完成后进入sleep模式，降低功耗 启动时为热启动
  sx126x_set_sleep_warm_start(&global.sx1268);
  NRF_LOG_INFO("SX1268 tx done.");
}
```

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)

5.1 函数简介

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);
```

5.1.2接收发送配置

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：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：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：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：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 其他函数

5.1.3.1 SX1278SetRx

```
//函数介绍：
void SX1278SetRx( uint32_t timeout, uint8_t flag)
```