1　串行外设接口（SPI）

1.1　SPI简介

SPI，是英语Serial Peripheral interface的缩写，串行外设接口(SPI)允许芯片与外部设备以半/全双工、同步、串行方式通信。此接口可以被配置成主模式，并为外部从设备提供通信时钟(SCK)。

接口还能以多主配置方式工作。它可用于多种用途，包括使用一条双向数据线的双线单工同步传输，还可使用CRC校验的可靠通信。

1.2　SPI主要特征

- 3线全双工同步传输
- 带或不带第三根双向数据线的双线单工同步传输
- 8或16位传输帧格式选择
- 主或从操作
- 支持多主模式
- 8个主模式波特率预分频系数(最大为fpclk/2，40Mhz)
- 从模式频率 (最大为fpclk/2)
- 主模式和从模式的快速通信
- 主模式和从模式下均可以由软件或硬件进行NSS管理：主/从操作模式的动态改变
- 可编程的时钟极性和相位
- 可编程的数据顺序， MSB在前或LSB在前
- 可触发中断的专用发送和接收标志
- SPI总线忙状态标志
- 支持可靠通信的硬件CRC
  – 在发送模式下， CRC值可以被作为最后一个字节发送
  – 在全双工模式中对接收到的最后一个字节自动进行CRC校验
- 触发中断的主模式故障、过载以及CRC错误标志
- 持DMA功能的1字节发送和接收缓冲器：产生发送和接受请求

1.3　SPI功能描述

1.3.1　SPI 概述

通常SPI通过4个引脚与外部器件相连：

- MISO：主设备输入/从设备输出引脚。该引脚在从模式下发送数据，在主模式下接收数据。
- MOSI：主设备输出/从设备输入引脚。该引脚在主模式下发送数据，在从模式下接收数据。
- SCK：串口时钟，作为主设备的输出，从设备的输入
- NSS：从设备选择。这是一个可选的引脚，用来选择主/从设备。 数据发送完成后，要判断busy位，来让硬件把数据发送出去再拉高CS!!!

> NSS(CS)引脚由SPI_CR1中的NSS和SSM控制，对应控制函数为：
>
> ```
> SSOE  ->  hal_spi_ssoe_en();
> SSM   ->  hal_spi_nss_cfg();
> ```
>
> 通过这两个函数可以设置CS引脚状态：
>
> 1. SSM = 1：软件管理CS引脚，从器件的选择信息由SPI_CR1寄存器中得SSI位的值驱动。外部CS引脚可供他用，这个模式只适合从模式下使用。

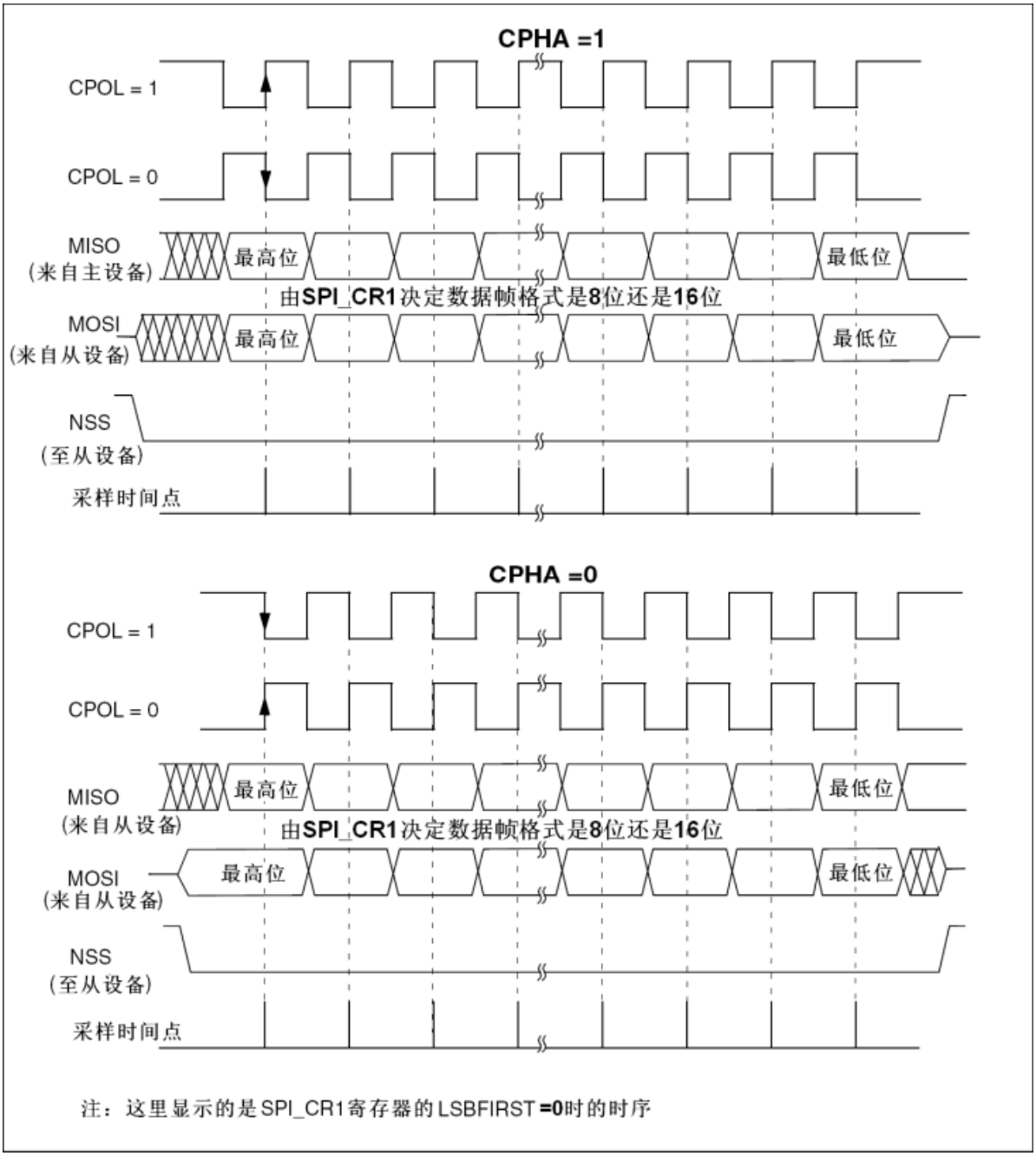
2. SSM = 0: 硬件管理CS引脚，根据SSOE的值，可以分为两种情况：（不推荐使用a模式，自行使用标准GPIO驱动SPI会使软件更灵活）
    - CS输出使能（SSOE = 1），主模式下才能使用此功能，当主机开始工作时，会把CS拉低，停止工作后恢复CS。当一个SPI设备需要发送广播数据，它必须拉低NSS信号，以通知所有其它的设备它是主设备；如果它不能拉低NSS，这意味着总线上有另外一个主设备在通信，则这个SPI设备进入主模式失败状态：即MSTR位被自动清除，此设备进入从模式.
    - CS输出失能（SSOE = 0），从模式下使用该模式，会在CS为低电平时选中该从器件，CS为高电平时，取消从器件的片选

时钟信号的相位和极性

SPI_CR寄存器的CPOL和CPHA位，能够组合成四种可能的时序关系。CPOL(时钟极性)位控制在没有数据传输时时钟的空闲状态电平，此位对主模式和从模式下的设备都有效。如果CPOL被清’0’，SCK引脚在空闲状态保持低电平；如果CPOL被置’1’， SCK引脚在空闲状态保持高电平。如果CPHA(时钟相位)位被置’1’， SCK时钟的第二个边沿(CPOL位为0时就是下降沿， CPOL位为’1’时就是上升沿)进行数据位的采样，数据在第二个时钟边沿被锁存。如果CPHA位被清’0’，SCK时钟的第一边沿(CPOL位为’0’时就是下降沿， CPOL位为’1’时就是上升沿)进行数据位采样，数据在第一个时钟边沿被锁存。

注意：

1. 在改变CPOL/CPHA位之前，必须清除SPE位将SPI禁止。
2. 主和从必须配置成相同的时序模式。
3. SCK的空闲状态必须和SPI_CR1寄存器指定的极性一致(CPOL为’1’时，空闲时应上拉SCK为高电平； CPOL为’0’时，空闲时应下拉SCK为低电平)。
4. 数据帧格式(8位或16位)由SPI_CR1寄存器的DFF位选择，并且决定发送/接收的数据长度。



1.3.2 SPI 主模式

配置步骤

1. 通过SPI_CR1寄存器的BR[2:0]位定义串行时钟波特率。
2. 选择CPOL和CPHA位，定义数据传输和串行时钟间的相位关系(见图212)。
3. 设置DFF位来定义8位或16位数据帧格式。
4. 配置SPI_CR1寄存器的LSBFIRST位定义帧格式。
5. 如果需要NSS引脚工作在输入模式，硬件模式下，在整个数据帧传输期间应把NSS脚连接到高电平；在软件模式下，需设置SPI_CR1寄存器的SSM位和SSI位。如果NSS引脚工作在输出模式，则只需设置SSOE位。**数据发送完成后，要判断busy位，来让硬件把数据发送出去再拉高CS!!!**
6. 必须设置MSTR位和SPE位(只当NSS脚被连到高电平，这些位才能保持置位)。在这个配置中，MOSI引脚是数据输出，而MISO引脚是数据输入。

数据发送过程

当写入数据至发送缓冲器时，发送过程开始。

在发送第一个数据位时，数据字被并行地(通过内部总线)传入移位寄存器，而后串行地移出到MOSI脚上； MSB在先还是LSB在先，取决于SPI_CR1寄存器中的LSBFIRST位的设置。数据从发送缓冲器传输到移位寄存器时TXE标志将被置位，如果设置了SPI_CR1寄存器中的TXEIE位，将产生中断。

数据接收过程

对于接收器来说，当数据传输完成时：

- 送移位寄存器里的数据到接收缓冲器，并且RXNE标志被置位。
- 果设置了SPI_CR2寄存器中的RXNEIE位，则产生中断。
  在最后采样时钟沿， RXNE位被设置，在移位寄存器中接收到的数据字被传送到接收缓冲器。读SPI_DR寄存器时， SPI设备返回接收缓冲器中的数据。读SPI_DR寄存器将清除RXNE位。一旦传输开始，如果下一个将发送的数据被放进了发送缓冲器，就可以维持一个连续的传输流。在试图写发送缓冲器之前，需确认TXE标志应该为’ 1’ 。

> 注： 在NSS硬件模式下，从设备的NSS输入由NSS引脚控制或另一个由软件驱动的GPIO引脚控制。 **数据发送完成后，要判断busy位，来让硬件把数据发送出去再拉高CS!!!**

1.3.3 SPI 从模式

在从模式下， SCK引脚用于接收从主设备来的串行时钟。 SPI_CR1寄存器中BR[2:0]的设置不影响数据传输速率。

> 建议在主设备发送时钟之前使能SPI从设备，否则可能会发生意外的数据传输。在通信时钟的第一个边沿到来之前或正在进行的通信结束之前，从设备的数据寄存器必须就绪。在使能从设备和主设备之前，通信时钟的极性必须处于稳定的数值。

配置步骤

1. 设置DFF位以定义数据帧格式为8位或16位。
2. 选择CPOL和CPHA位来定义数据传输和串行时钟之间的相位关系。为保证正确的数据传输，从设备和主设备的CPOL和CPHA位必须配置成相同的方式。
3. 帧格式(SPI_CR1寄存器中的LSBFIRST位定义的”MSB在前”还是”LSB在前”)必须与主设备相同。
4. 硬件模式下，在完整的数据帧(8位或16位)传输过程中，NSS引脚必须为低电平。在NSS软件模式下，设置SPI_CR1寄存器中的SSM位并清除SSI
   位。
5. 清除MSTR位、设置SPE位(SPI_CR1寄存器)，使相应引脚工作于SPI模式下。在这个配置中，MOSI引脚是数据输入， MISO引脚是数据输出。

数据发送过程

在写操作中，数据字被并行地写入发送缓冲器。
当从设备收到时钟信号，并且在MOSI引脚上出现第一个数据位时，发送过程开始(译注：此时第一个位被发送出去)。余下的位(对于8位数据帧格式，还有7位；对于16位数据帧格式，还有15位)被装进移位寄存器。当发送缓冲器中的数据传输到移位寄存器时， SPI_SP寄存器的TXE标志被设置，如果设置了SPI_CR2寄存器的TXEIE位，将会产生中断。

数据接收过程

对于接收器，当数据接收完成时：

- 移位寄存器中的数据传送到接收缓冲器， SPI_SR 寄存器中的RXNE标志被设置。
- 如果设置了SPI_CR2寄存器中的RXNEIE位，则产生中断。
  在最后一个采样时钟边沿后， RXNE位被置’ 1’ ，移位寄存器中接收到的数据字节被传送到接收

缓冲器。当读SPI_DR寄存器时， SPI设备返回这个接收缓冲器的数值。读SPI_DR寄存器时，RXNE位被清除。

1.4 SPI测试代码

1.4.1 SPI主机测试代码

测试SPI的时候请注意SPI的接线最好是用焊接，使用杜邦线会出现接触不良的情况。

```
/**
 * @file     ln_spi_master_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-18
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/**
        SPI主机调试说明:
                    1. 接线说明:
                                LN8620          25WQ16

                                PB5     ->      SCLK(6)
                                PB6     ->      CS(1)
                                PB7     ->      MOSI(5)
                                PB8     ->      MISO(2)
                                VCC      ->     HOLD(7)
                                VCC      ->     WP(3)
                                GND      ->     GND(4)
                                VCC      ->     VCC(8)

                    2. SPI测试分为两部分，一部分是SPI直接读写25WQ16 Flash芯片，另一部分是通过SPI + DMA读写Flash芯片。

                    3. 测试中SPI CS引脚使用GPIO引脚控制，这样更灵活，更好用。数据发送完成后，要判断busy位，来让硬件把数据发送出去再拉高CS!!!

                    4. CS引脚也可以由SPI_CR1中的NSS和SSM控制，对应控制函数为:

                        SSOE -> hal_spi_ssoe_en();
                        SSM  -> hal_spi_nss_cfg();

                        通过这两个函数可以设置CS引脚状态:

                        (1) SSM = 1: 软件管理CS引脚，从器件的选择信息由SPI_CR1寄存器中得SSI位的值驱动。外部CS引脚可供他用，这个模式只适合从模式下使用。

                        (2) SSM = 0: 硬件管理CS引脚，根据SSOE的值，可以分为两种情况:（不推荐使用a模式，自行使用标准GPIO驱动SPI会使软件更灵活）
```

a. CS输出使能（SSOE = 1），主模式下才能使用此功能，当主机开始工作时，会把CS拉低，停止工作后恢复CS。

当一个SPI设备需要发送广播数据，它必须拉低NSS信号，以通知所有其它的设备它是主设备；如果它不能拉低NSS，这意味着总线上有另外一个主设备在通信，则这个SPI设备进入主模式失败状态：即MSTR位被自动清除，此设备进入从模式.

b. CS输出失能（SSOE = 0），从模式下使用该模式，会在CS为低电平时选中该从器件，CS为高电平时，取消从器件的片选

5. 测试成功会有LOG打印，请注意观察LOG.

*/

```
#include "ln_spi_master_test.h"
#include "hal/hal_spi.h"
#include "hal/hal_gpio.h"
#include "hal/hal_dma.h"
#include "hal/hal_misc.h"
#include "log.h"

/* 配置SPI CS引脚 */
#define LN_SPI_TEST_CS_PORT_BASE    GPIOB_BASE
#define LN_SPI_TEST_CS_PIN          GPIO_PIN_6

#define LN_SPI_CS_LOW
hal_gpio_pin_reset(LN_SPI_TEST_CS_PORT_BASE,LN_SPI_TEST_CS_PIN)
#define LN_SPI_CS_HIGH
hal_gpio_pin_set(LN_SPI_TEST_CS_PORT_BASE,LN_SPI_TEST_CS_PIN)


/* 初始化变量 */
unsigned char LN_DMA_EN_STATUS  = 0;

static unsigned char tx_data[100];
static unsigned char rx_data[100];

static unsigned char status[2];
static unsigned int err_cnt = 0;


/* 函数声明 */
void hal_25wq16_read_flash(uint32_t spi_x_base , unsigned int addr ,unsigned
char *data, unsigned int length);
void hal_25wq16_write_flash(uint32_t spi_x_base , unsigned int addr ,unsigned
char *data, unsigned int length);
void hal_25wq16_erase_flash(uint32_t spi_x_base ,unsigned int add);
void hal_25wq16_read_status(unsigned int spi_x_base,unsigned char *status);
void hal_25wq16_read_id(unsigned int spi_x_base,unsigned char *id);
void hal_spi_read_data_with_addr(unsigned int spi_x_base,unsigned char
*addr,unsigned int addr_len,unsigned char *rec_data,unsigned int data_len);
void hal_spi_write_data_with_addr(unsigned int spi_x_base,unsigned char
*addr,unsigned int addr_len,unsigned char *send_data,unsigned int data_len);
void hal_spi_write_and_read_data(unsigned int spi_x_base,unsigned char
*send_data,unsigned char *rec_data,unsigned int data_len);
```

```c
void ln_spi_master_init()
{
    /* 1. 配置引脚 */
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_5,SPI0_CLK);
    //hal_gpio_afio_select(REG_GPIOB_BASE,GPIO_PIN_6,SPI0_CSN);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_7,SPI0_MOSI);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_8,SPI0_MISO);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_5,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_6,HAL_DISABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_7,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_8,HAL_ENABLE);

    //配置CS引脚
    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));
    gpio_init.dir = GPIO_OUTPUT;
    gpio_init.pin = LN_SPI_TEST_CS_PIN;
    gpio_init.speed = GPIO_HIGH_SPEED;
    hal_gpio_init(LN_SPI_TEST_CS_PORT_BASE,&gpio_init);
    hal_gpio_pin_set(LN_SPI_TEST_CS_PORT_BASE,LN_SPI_TEST_CS_PIN);

    /* 2. 配置SPI */
    spi_init_type_def spi_init;
    memset(&spi_init,0,sizeof(spi_init));

    spi_init.spi_baud_rate_prescaler = SPI_BAUDRATEPRESCALER_256;        //设置波特
率
    spi_init.spi_mode      = SPI_MODE_MASTER;                            //设置主从
模式
    spi_init.spi_data_size = SPI_DATASIZE_8B;                            //设置数据
大小
    spi_init.spi_first_bit = SPI_FIRST_BIT_MSB;                          //设置帧格
式
    spi_init.spi_cpol      = SPI_CPOL_LOW;                               //设置时钟
极性
    spi_init.spi_cpha      = SPI_CPHA_1EDGE;                             //设置时钟
相位
    hal_spi_init(SPI0_BASE,&spi_init);                                   //初始化
SPI

    hal_spi_en(SPI0_BASE,HAL_ENABLE);                                    //SPI使能
    hal_spi_ssoe_en(SPI0_BASE,HAL_DISABLE);                              //关闭CS
OUTPUT
}


void ln_spi_dma_init(void)
{
    /* init the spi rx dma */
    dma_init_t_def dma_init;
    memset(&dma_init,0,sizeof(dma_init));

    dma_init.dma_mem_addr = (uint32_t)rx_data;       //设置DMA内存地址
    dma_init.dma_data_num = 0;                       //设置DMA传输次数
    dma_init.dma_dir = DMA_READ_FORM_P;              //设置DMA方向
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;        //设置DMA内存是否增长
    dma_init.dma_p_addr = SPI0_DATA_REG;             //设置DMA外设地址
    hal_dma_init(DMA_CH_3,&dma_init);                //初始化DMA
```

```
    //hal_dma_it_cfg(DMA_CH_3,DMA_IT_FLAG_TRAN_COMP,HAL_ENABLE);          //配置DMA
中断

    hal_dma_en(DMA_CH_3,HAL_DISABLE);                                     //DMA使能
    hal_spi_dma_en(SPI0_BASE,SPI_DMA_RX_EN,HAL_ENABLE);                   //SPI
DMA RX 使能

    /* init the spi tx dma */
    memset(&dma_init,0,sizeof(dma_init));

    dma_init.dma_mem_addr = (uint32_t)tx_data;        //设置DMA内存地址
    dma_init.dma_data_num = 0;                        //设置DMA传输次数
    dma_init.dma_dir = DMA_READ_FORM_MEM;             //设置DMA方向
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;         //设置DMA内存是否增长
    dma_init.dma_p_addr = SPI0_DATA_REG;              //设置DMA外设地址
    hal_dma_init(DMA_CH_4,&dma_init);                 //初始化DMA

    //hal_dma_it_cfg(DMA_CH_4,DMA_IT_FLAG_TRAN_COMP,HAL_ENABLE);          //配置DMA
中断

    hal_dma_en(DMA_CH_4,HAL_DISABLE);                                     //DMA使能
    hal_spi_dma_en(SPI0_BASE,SPI_DMA_TX_EN,HAL_ENABLE);                   //SPI
DMA TX 使能
}

void hal_spi_write_and_read_data_with_dma(unsigned int spi_x_base,unsigned char
*send_data,unsigned char *rec_data,unsigned int data_len)
{
    //开始传输，先拉低CS。
    LN_SPI_CS_LOW;

    //配置DMA传输参数。
    hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)rec_data);
    hal_dma_set_mem_addr(DMA_CH_4,(uint32_t)send_data);
    hal_dma_set_data_num(DMA_CH_3,data_len);
    hal_dma_set_data_num(DMA_CH_4,data_len);

    //开始传输。
    hal_dma_en(DMA_CH_3,HAL_ENABLE);
    hal_dma_en(DMA_CH_4,HAL_ENABLE);

    //等待传输完成。
    while(hal_dma_get_data_num(DMA_CH_3) != 0 || hal_dma_get_data_num(DMA_CH_4)
!= 0);
    hal_spi_wait_bus_idle(SPI0_BASE,10000);

    //传输结束，拉高CS
    LN_SPI_CS_HIGH;

    //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
    hal_dma_en(DMA_CH_3,HAL_DISABLE);
    hal_dma_en(DMA_CH_4,HAL_DISABLE);
}


/* 25WQ16ES FLASH 芯片调试 */
```

```
void hal_spi_write_and_read_data(unsigned int spi_x_base,unsigned char
*send_data,unsigned char *rec_data,unsigned int data_len)
{
    if(!LN_DMA_EN_STATUS)
    {
        LN_SPI_CS_LOW;
        for(int i = 0; i < data_len; i++)
        {
            hal_spi_wait_bus_idle(SPI0_BASE,1000000);

            if(hal_spi_wait_txe(SPI0_BASE,1000000) == 1)
                hal_spi_send_data(SPI0_BASE,send_data[i]);
            if(hal_spi_wait_rxne(SPI0_BASE,1000000) == 1)
                rec_data[i] = hal_spi_recv_data(SPI0_BASE);

            hal_spi_wait_bus_idle(SPI0_BASE,1000000);
        }
        LN_SPI_CS_HIGH;
    }
    else
    {
        //开始传输，先拉低CS。
        LN_SPI_CS_LOW;

        //配置DMA传输参数。
        hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)rec_data);
        hal_dma_set_mem_addr(DMA_CH_4,(uint32_t)send_data);
        hal_dma_set_data_num(DMA_CH_3,data_len);
        hal_dma_set_data_num(DMA_CH_4,data_len);
        hal_dma_set_mem_inc_en(DMA_CH_3,DMA_MEM_INC_EN);
        hal_dma_set_mem_inc_en(DMA_CH_4,DMA_MEM_INC_EN);

        //开始传输。
        hal_dma_en(DMA_CH_3,HAL_ENABLE);
        hal_dma_en(DMA_CH_4,HAL_ENABLE);

        //等待传输完成。
        while(hal_dma_get_data_num(DMA_CH_3) != 0 ||
hal_dma_get_data_num(DMA_CH_4) != 0);
        hal_spi_wait_bus_idle(SPI0_BASE,10000);

        //传输结束，拉高CS
        LN_SPI_CS_HIGH;

        //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
        hal_dma_en(DMA_CH_3,HAL_DISABLE);
        hal_dma_en(DMA_CH_4,HAL_DISABLE);
    }
}

void hal_spi_write_data_with_addr(unsigned int spi_x_base,unsigned char
*addr,unsigned int addr_len,unsigned char *send_data,unsigned int data_len)
{
    unsigned char rec_data_buffer[255];

    if(!LN_DMA_EN_STATUS)
    {
        LN_SPI_CS_LOW;
```

```
        for(int i = 0; i < addr_len; i++)
        {
            hal_spi_wait_bus_idle(SPI0_BASE,1000000);

            if(hal_spi_wait_txe(SPI0_BASE,1000000) == 1)
                hal_spi_send_data(SPI0_BASE,addr[i]);
            if(hal_spi_wait_rxne(SPI0_BASE,1000000) == 1)
                rec_data_buffer[0] = hal_spi_recv_data(SPI0_BASE);

            hal_spi_wait_bus_idle(SPI0_BASE,1000000);
        }
        for(int i = 0; i < data_len; i++)
        {
            hal_spi_wait_bus_idle(SPI0_BASE,1000000);

            if(hal_spi_wait_txe(SPI0_BASE,1000000) == 1)
                hal_spi_send_data(SPI0_BASE,send_data[i]);
            if(hal_spi_wait_rxne(SPI0_BASE,1000000) == 1)
                rec_data_buffer[0] = hal_spi_recv_data(SPI0_BASE);

            hal_spi_wait_bus_idle(SPI0_BASE,1000000);
        }
        LN_SPI_CS_HIGH;
    }
    else
    {
        //开始传输，先拉低CS。
        LN_SPI_CS_LOW;

        //配置DMA传输参数。
        hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)rec_data_buffer);
        hal_dma_set_mem_addr(DMA_CH_4,(uint32_t)addr);
        hal_dma_set_data_num(DMA_CH_3,addr_len);
        hal_dma_set_data_num(DMA_CH_4,addr_len);
        hal_dma_set_mem_inc_en(DMA_CH_3,DMA_MEM_INC_EN);
        hal_dma_set_mem_inc_en(DMA_CH_4,DMA_MEM_INC_EN);

        //开始传输。
        hal_dma_en(DMA_CH_3,HAL_ENABLE);
        hal_dma_en(DMA_CH_4,HAL_ENABLE);

        //等待传输完成。
        while(hal_dma_get_data_num(DMA_CH_3) != 0 ||
hal_dma_get_data_num(DMA_CH_4) != 0);
        hal_spi_wait_bus_idle(SPI0_BASE,10000);


        //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
        hal_dma_en(DMA_CH_3,HAL_DISABLE);
        hal_dma_en(DMA_CH_4,HAL_DISABLE);

        //配置DMA传输参数。
        hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)rec_data_buffer);
        hal_dma_set_mem_addr(DMA_CH_4,(uint32_t)send_data);
        hal_dma_set_data_num(DMA_CH_3,data_len);
        hal_dma_set_data_num(DMA_CH_4,data_len);
        hal_dma_set_mem_inc_en(DMA_CH_3,DMA_MEM_INC_EN);
```

```
        hal_dma_set_mem_inc_en(DMA_CH_4,DMA_MEM_INC_EN);

        //开始传输。
        hal_dma_en(DMA_CH_3,HAL_ENABLE);
        hal_dma_en(DMA_CH_4,HAL_ENABLE);

        //等待传输完成。
        while(hal_dma_get_data_num(DMA_CH_3) != 0 ||
hal_dma_get_data_num(DMA_CH_4) != 0);
        hal_spi_wait_bus_idle(SPI0_BASE,10000);

        //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
        hal_dma_en(DMA_CH_3,HAL_DISABLE);
        hal_dma_en(DMA_CH_4,HAL_DISABLE);

        //传输结束，拉高CS
        LN_SPI_CS_HIGH;

    }
}

void hal_spi_read_data_with_addr(unsigned int spi_x_base,unsigned char
*addr,unsigned int addr_len,unsigned char *rec_data,unsigned int data_len)
{
    unsigned char buffer[255];
    LN_SPI_CS_LOW;
    if(!LN_DMA_EN_STATUS)
    {
        for(int i = 0; i < addr_len; i++)
        {
            hal_spi_wait_bus_idle(SPI0_BASE,1000000);

            if(hal_spi_wait_txe(SPI0_BASE,1000000) == 1)
                hal_spi_send_data(SPI0_BASE,addr[i]);
            if(hal_spi_wait_rxne(SPI0_BASE,1000000) == 1)
                buffer[0] = hal_spi_recv_data(SPI0_BASE);

            hal_spi_wait_bus_idle(SPI0_BASE,1000000);
        }
        for(int i = 0; i < data_len; i++)
        {
            hal_spi_wait_bus_idle(SPI0_BASE,1000000);

            if(hal_spi_wait_txe(SPI0_BASE,1000000) == 1)
                hal_spi_send_data(SPI0_BASE,0xFF);
            if(hal_spi_wait_rxne(SPI0_BASE,1000000) == 1)
                rec_data[i] = hal_spi_recv_data(SPI0_BASE);

            hal_spi_wait_bus_idle(SPI0_BASE,1000000);
        }
        LN_SPI_CS_HIGH;
    }
    else
    {
        //开始传输，先拉低CS。
        LN_SPI_CS_LOW;

        //配置DMA传输参数。
```

```
        hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)rec_data);
        hal_dma_set_mem_addr(DMA_CH_4,(uint32_t)addr);
        hal_dma_set_data_num(DMA_CH_3,addr_len);
        hal_dma_set_data_num(DMA_CH_4,addr_len);
        hal_dma_set_mem_inc_en(DMA_CH_3,DMA_MEM_INC_EN);
        hal_dma_set_mem_inc_en(DMA_CH_4,DMA_MEM_INC_EN);

        //开始传输。
        hal_dma_en(DMA_CH_3,HAL_ENABLE);
        hal_dma_en(DMA_CH_4,HAL_ENABLE);

        //等待传输完成。
        while(hal_dma_get_data_num(DMA_CH_3) != 0 ||
hal_dma_get_data_num(DMA_CH_4) != 0);
        hal_spi_wait_bus_idle(SPI0_BASE,10000);


        //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
        hal_dma_en(DMA_CH_3,HAL_DISABLE);
        hal_dma_en(DMA_CH_4,HAL_DISABLE);

        //配置DMA传输参数。
        hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)rec_data);
        hal_dma_set_mem_addr(DMA_CH_4,(uint32_t)buffer);
        hal_dma_set_data_num(DMA_CH_3,data_len);
        hal_dma_set_data_num(DMA_CH_4,data_len);
        hal_dma_set_mem_inc_en(DMA_CH_4,DMA_MEM_INC_EN);
        hal_dma_set_mem_inc_en(DMA_CH_3,DMA_MEM_INC_EN);
        //开始传输。
        hal_dma_en(DMA_CH_3,HAL_ENABLE);
        hal_dma_en(DMA_CH_4,HAL_ENABLE);

        //等待传输完成。
        while(hal_dma_get_data_num(DMA_CH_3) != 0 ||
hal_dma_get_data_num(DMA_CH_4) != 0);
        hal_spi_wait_bus_idle(SPI0_BASE,10000);

        //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
        hal_dma_en(DMA_CH_3,HAL_DISABLE);
        hal_dma_en(DMA_CH_4,HAL_DISABLE);

        //传输结束，拉高CS
        LN_SPI_CS_HIGH;
    }
}


void hal_25wq16_read_id(unsigned int spi_x_base,unsigned char *id)
{
    unsigned char CMD_READ_ID[4] = {0x9F,0xFF,0xFF,0xFF};
    unsigned char buffer[4];
    hal_spi_write_and_read_data(spi_x_base,CMD_READ_ID,buffer,4);
    memcpy(id,buffer+1,3);
}

void hal_25wq16_read_status(unsigned int spi_x_base,unsigned char *status)
{
    unsigned char CMD_READ_STATUS_1[2] = {0x05,0xFF};
```

```
    unsigned char CMD_READ_STATUS_2[2] = {0x35,0xFF};
    unsigned char buffer[2] = {0,0};
    hal_spi_write_and_read_data(spi_x_base,CMD_READ_STATUS_1,buffer,2);
    memcpy(status,buffer+1,1);
    hal_spi_write_and_read_data(spi_x_base,CMD_READ_STATUS_2,buffer,2);
    memcpy(status+1,buffer+1,1);
}

void hal_25wq16_write_enable(unsigned int spi_x_base)
{
    unsigned char CMD_WRITE_ENABLE[1] = {0x06};
    unsigned char buffer[1];
    hal_spi_write_and_read_data(spi_x_base,CMD_WRITE_ENABLE,buffer,1);
}
void hal_25wq16_erase_flash(uint32_t spi_x_base ,unsigned int add)
{
    hal_25wq16_write_enable(spi_x_base);
    unsigned char CMD_ERASE_FLASH[4] = {0x20,0x00,0x00,0x00};
    unsigned char buffer[4];
    CMD_ERASE_FLASH[1] = (add >> 16) & 0xFF;
    CMD_ERASE_FLASH[2] = (add >> 8)  & 0xFF;
    CMD_ERASE_FLASH[3] = (add >> 0)  & 0xFF;
    hal_spi_write_and_read_data(spi_x_base,CMD_ERASE_FLASH,buffer,4);
}

void hal_25wq16_write_flash(uint32_t spi_x_base , unsigned int addr ,unsigned
char *data, unsigned int length)
{
    unsigned char buffer[4] = {0x02,0x00,0x00,0x00};
    hal_25wq16_write_enable(spi_x_base);
    buffer[1] = (addr >> 16) & 0xFF;
    buffer[2] = (addr >> 8)  & 0xFF;
    buffer[3] = (addr >> 0)  & 0xFF;
    hal_spi_write_data_with_addr(spi_x_base,buffer,4,data,length);
}

void hal_25wq16_read_flash(uint32_t spi_x_base , unsigned int addr ,unsigned
char *data, unsigned int length)
{
    unsigned char buffer[4] = {0x03,0x00,0x00,0x00};
    buffer[1] = (addr >> 16) & 0xFF;
    buffer[2] = (addr >> 8)  & 0xFF;
    buffer[3] = (addr >> 0)  & 0xFF;

    hal_spi_read_data_with_addr(spi_x_base,buffer,4,data,length);
}

void ln_spi_master_test(void)
{
    //SPI主机模式初始化
    ln_spi_master_init();

    //初始化数据buffer
    for(int i = 0; i < 100 ; i++)
    {
        tx_data[i] = i + 2;
    }
```

```
    LOG(LOG_LVL_INFO,"ln8620 SPI + 25WQ16 test start! \n");

    /*********************************SPI直接读写Flash芯片
*****************************************************************/

    //读取ID
    hal_25wq16_read_id(SPI0_BASE,rx_data);
    LOG(LOG_LVL_INFO,"ln8620 25WQ16 ID: %x %x %x
\n",rx_data[0],rx_data[1],rx_data[2]);

    //读取状态
    hal_25wq16_read_status(SPI0_BASE,rx_data);

    //写使能
    hal_25wq16_write_enable(SPI0_BASE);

    //读取状态
    hal_25wq16_read_status(SPI0_BASE,rx_data);

    //擦除芯片
    hal_25wq16_erase_flash(SPI0_BASE,0x200);

    //等待擦除完成
    while(1)
    {
        hal_25wq16_read_status(SPI0_BASE,status);
        if((status[0] & 0x01) == 0x00)
            break;
    }

    //向Flash芯片中写入数据
    hal_25wq16_write_flash(SPI0_BASE,0x200,tx_data,100);

    //等待写入完成
    while(1)
    {
        hal_25wq16_read_status(SPI0_BASE,status);
        if((status[0] & 0x01) == 0x00)
            break;
    }

    //从Flash芯片中读出数据
    hal_25wq16_read_flash(SPI0_BASE,0x200,rx_data,100);

    //判断写入的数据是否正确
    for(int i = 0 ; i < 100; i++)
    {
        if(rx_data[i] != tx_data[i])
        {
            err_cnt++;
        }
    }

    //打印LOG
    if(err_cnt != 0)
    {
        LOG(LOG_LVL_INFO,"ln8620 SPI + 25WQ16 test fail! \n");
    }
```

```
    else
    {
        LOG(LOG_LVL_INFO,"ln8620 SPI + 25WQ16 test success! \n");
    }


    /*********************************SPI + DMA 读写Flash芯片
*********************************************************************/
    LOG(LOG_LVL_INFO,"ln8620 SPI + DMA + 25WQ16 test start! \n");

    LN_DMA_EN_STATUS = 1;
    ln_spi_dma_init();

    //读取ID
    hal_25wq16_read_id(SPI0_BASE,rx_data);
    LOG(LOG_LVL_INFO,"ln8620 25WQ16 ID: %x %x %x
\n",rx_data[0],rx_data[1],rx_data[2]);

    //读取状态
    hal_25wq16_read_status(SPI0_BASE,rx_data);

    //写使能
    hal_25wq16_write_enable(SPI0_BASE);

    //读取状态
    hal_25wq16_read_status(SPI0_BASE,rx_data);

    //擦除芯片
    hal_25wq16_erase_flash(SPI0_BASE,0x200);

    //等待擦除完成
    while(1)
    {
        hal_25wq16_read_status(SPI0_BASE,status);
        if((status[0] & 0x01) == 0x00)
            break;
    }

    //向Flash芯片中写入数据
    hal_25wq16_write_flash(SPI0_BASE,0x200,tx_data,100);

    //等待写入完成
    while(1)
    {
        hal_25wq16_read_status(SPI0_BASE,status);
        if((status[0] & 0x01) == 0x00)
            break;
    }

    memset(rx_data,0,100);

    //从Flash芯片中读出数据
    hal_25wq16_read_flash(SPI0_BASE,0x200,rx_data,100);

    //判断写入的数据是否正确
    for(int i = 0 ; i < 100; i++)
    {
        if(rx_data[i] != tx_data[i])
```

```
        {
            err_cnt++;
        }
    }

    //打印LOG
    if(err_cnt != 0)
    {
        LOG(LOG_LVL_INFO,"ln8620 SPI + DMA + 25WQ16 test fail! \n");
    }
    else
    {
        LOG(LOG_LVL_INFO,"ln8620 SPI + DMA + 25WQ16 test success! \n");
    }

    while(1)
    {
        hal_25wq16_read_flash(SPI0_BASE,0x200,rx_data,100);
    }

}
```

1.4.2 SPI从机测试代码

FPGA 18M主机CLK 可以稳定接收数据

```
/**
 * @file     ln_spi_slave_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-19
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co.
Ltd
 *
 */

/**
        SPI从机调试说明：

                1. 接线说明：
                            LN8620(从机)    STM32(主机)
                            PA10     ->      PA4     -> SPI_CS
                            PA11     ->      PA5     -> SPI_SCK
                            PA12     ->      PA6     -> SPI_MISO
                            PA13     ->      PA7     -> SPI_MOSI
                            GND      ->      GND

                2. CS引脚由SPI_CR1中的NSS和SSM控制，对应控制函数为：

                            SSOE -> hal_spi_ssoe_en();
                            SSM  -> hal_spi_nss_cfg();

                    通过这两个函数可以设置CS引脚状态：
```

（1） SSM = 1：软件管理CS引脚，从器件的选择信息由SPI_CR1寄存器中得SSI位的值驱动。外部CS引脚可供他用，这个模式只适合从模式下使用。

（2） SSM = 0：硬件管理CS引脚，根据SSOE的值，可以分为两种情况：（不推荐使用a模式，自行使用标准GPIO驱动SPI会使软件更灵活）

a. CS输出使能（SSOE = 1），主模式下才能使用此功能，当主机开始工作时，会把CS拉低，停止工作后恢复CS。

b. CS输出失能（SSOE = 0），从模式下使用该模式，会在CS为低电平时选中该从器件，CS为高电平时，取消从器件的片选。

3. 8620 SPI SLAVE + DMA 可能会出现中断进不了的问题，在中断中，使能DMA之前加个读SPI_DR的指令，可能会解决这个问题。（非必现）

```
*/
#include "ln_spi_slave_test.h"
#include "hal/hal_spi.h"
#include "hal/hal_gpio.h"
#include "hal/hal_dma.h"
#include "log.h"


#define LN_SPI_TEST_CS_PORT_BASE GPIOB_BASE
#define LN_SPI_TEST_CS_PIN       GPIO_PIN_5

#define LN_SPI_CS_LOW
hal_gpio_pin_reset(LN_SPI_TEST_CS_PORT_BASE,LN_SPI_TEST_CS_PIN)
#define LN_SPI_CS_HIGH
hal_gpio_pin_set(LN_SPI_TEST_CS_PORT_BASE,LN_SPI_TEST_CS_PIN)

static unsigned char tx_data[100];
static unsigned char rx_data_1[100];
static unsigned char rx_data_2[100];

static unsigned int         data_num = 100;     //DMA传输次数
static unsigned char        data_sel = 0;       //双缓冲BUFFER选择
static unsigned char        data_comp= 0;       //数据接收完成标志位

void ln_spi_slave_init()
{
    /* 配置GPIO引脚 */
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_5,SPI0_CSN);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_6,SPI0_CLK);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_7,SPI0_MISO);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_8,SPI0_MOSI);

    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_5,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_6,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_7,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_8,HAL_ENABLE);

    /* 配置SPI */
    spi_init_type_def spi_init;
    memset(&spi_init,0,sizeof(spi_init));
```

```
    spi_init.spi_baud_rate_prescaler = SPI_BAUDRATEPRESCALER_2; //配置分频，SLAVE模
式无效果
    spi_init.spi_mode = SPI_MODE_SLAVE;                         //配置主从模式
    spi_init.spi_data_size = SPI_DATASIZE_8B;                   //配置数据位宽
    spi_init.spi_first_bit = SPI_FIRST_BIT_LSB;                 //配置最先发送的bit
位置
    spi_init.spi_cpha = SPI_CPHA_1EDGE;                         //配置时钟相位
    spi_init.spi_cpol = SPI_CPOL_LOW;                           //配置时钟相位
    hal_spi_init(SPI0_BASE,&spi_init);                          //初始化SPI

    hal_spi_ssoe_en(SPI0_BASE,HAL_DISABLE);                     //关闭CS引脚输出
    hal_spi_nss_cfg(SPI0_BASE,SPI_NSS_HARD);                    //配置CS引脚为硬件
模式

    //hal_spi_it_cfg(SPI0_BASE,SPI_IT_FLAG_OVR,HAL_ENABLE);     //配置SPI中断
}

void ln_spi_slave_dma_init(void)
{
    /* init the spi rx dma */
    dma_init_t_def dma_init;
    memset(&dma_init,0,sizeof(dma_init));

    hal_dma_en(DMA_CH_3,HAL_DISABLE);                           //失能DMA（为
了配置参数）
    dma_init.dma_mem_addr = (uint32_t)rx_data_1;                //配置DMA内存
地址
    dma_init.dma_data_num = 100;                                 //配置DMA传输
数量
    dma_init.dma_dir = DMA_READ_FORM_P;                         //配置DMA传输
方向
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;                   //配置DMA内存
是否自增
    dma_init.dma_p_addr = SPI0_DATA_REG;                         //配置DMA外设
地址
    hal_dma_init(DMA_CH_3,&dma_init);                           //初始化DMA
    hal_dma_it_cfg(DMA_CH_3,DMA_IT_FLAG_TRAN_COMP,HAL_ENABLE);  //配置DMA传输
完成中断
    hal_dma_en(DMA_CH_3,HAL_DISABLE);


    /* init the spi tx dma */
    memset(&dma_init,0,sizeof(dma_init));

    hal_dma_en(DMA_CH_4,HAL_DISABLE);                           //失能DMA（为
了配置参数）
    dma_init.dma_mem_addr = (uint32_t)tx_data;                  //配置DMA内存
地址
    dma_init.dma_data_num = 100;                                 //配置DMA传输
数量
    dma_init.dma_dir = DMA_READ_FORM_MEM;                       //配置DMA传输
方向
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;                   //配置DMA内存
是否自增
    dma_init.dma_p_addr = SPI0_DATA_REG;                        //配置DMA外设
地址
    dma_init.dma_circ_mode_en = DMA_CIRC_MODE_EN;               //使能DMA循环
发送
```

```
    hal_dma_init(DMA_CH_4,&dma_init);                                   //初始化DMA
    //hal_dma_it_cfg(DMA_CH_4,DMA_IT_FLAG_TRAN_COMP,HAL_ENABLE);        //配置DMA传输完成中断


    //hal_spi_dma_en(SPI0_BASE,SPI_DMA_TX_EN,HAL_ENABLE);               //使能SPI TX DMA传输
    hal_spi_dma_en(SPI0_BASE,SPI_DMA_RX_EN,HAL_ENABLE);                 //使能SPI RX DMA传输

    //hal_dma_en(DMA_CH_4,HAL_ENABLE);                                  //使能DMA
    hal_dma_en(DMA_CH_3,HAL_ENABLE);                                    //使能DMA

    hal_spi_en(SPI0_BASE,HAL_ENABLE);                                   //使能SPI
}

volatile unsigned int err_cnt_1 = 0;

void ln_spi_slave_test()
{
    ln_spi_slave_init();
    ln_spi_slave_dma_init();
    NVIC_SetPriority(DMA_IRQn,     4);                                  //配置DMA中断优先级
    NVIC_EnableIRQ(DMA_IRQn);                                           //使能DMA中断
    //NVIC_SetPriority(SPI0_IRQn,     4);                               //配置SPI中断优先级
    //NVIC_EnableIRQ(SPI0_IRQn);                                        //使能SPI中断

    data_sel = 0;

    while(1)
    {
        if(data_comp == 1)
        {
            if(data_sel == 1)
            {
                data_comp = 0;
                for(int i = 0; i < 99; i ++)
                {
                    if((unsigned char)(rx_data_1[i] + 2) != rx_data_1[i+1])
                    {
                        err_cnt_1++;
                    }
                }
            }
            else if(data_sel == 2)
            {
                data_comp = 0;
                for(int i = 0; i < 99; i ++)
                {
                    if((unsigned char)(rx_data_2[i] + 2) != rx_data_2[i+1])
                    {
                        err_cnt_1++;
                    }
                }
            }
        }
    }
```

```
}
void SPI0_IRQHandler()
{
    if(hal_spi_get_it_flag(SPI0_BASE,SPI_IT_FLAG_OVR) == HAL_SET)
    {
        hal_spi_recv_data(SPI0_BASE);
    }
}
static unsigned int int_cnt_1 = 0;      //中断计数器，调试使用
static unsigned int int_cnt_2 = 0;      //中断计数器，调试使用


void DMA_IRQHandler()
{
    if(hal_dma_get_it_flag(DMA_CH_3,DMA_IT_FLAG_TRAN_COMP) == HAL_SET)
      /* 数据接收 */
    {
        //ln_delay_ms(10);
        if(data_sel == 1)
        {
            int_cnt_1++;
            data_sel = 2;
            hal_dma_en(DMA_CH_3,HAL_DISABLE);                                   //
失能DMA（为了配置参数）
            hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)(rx_data_1));               //
重新设置DMA地址
            hal_dma_set_data_num(DMA_CH_3,100);                                 //
设置DMA数据长度
            hal_dma_en(DMA_CH_3,HAL_ENABLE);                                    //
设置DMA使能
        }
        else
        {
            int_cnt_2++;
            data_sel = 1;
            hal_dma_en(DMA_CH_3,HAL_DISABLE);                                   //
失能DMA（为了配置参数）
            hal_dma_set_mem_addr(DMA_CH_3,(uint32_t)(rx_data_2));               //
重新设置DMA地址
            hal_dma_set_data_num(DMA_CH_3,100);                                 //
设置DMA数据长度
            hal_dma_en(DMA_CH_3,HAL_ENABLE);                                    //
设置DMA使能
        }
        //hal_dma_clear_it_flag(DMA_CH_3,DMA_IT_FLAG_TRAN_COMP);
        data_comp = 1;
    }
}
```

2 高级定时器（ADV_Timer）

2.1 高级定时器简介

定时器是一个通过可编程预分频器驱动的16位自动装载计数器构成。

2.2 高级定时器主要特征

- 16位向上、向下、向上/下自动装载计数器
- 6位可编程预分频器，计数器时钟频率的分频系数为0～63之间的任意数值，0为不分频
- 12位可编程死区时间的互补输出
- 六个独立通道
- 支持PWM输出比较、输入捕获以及PWM触发ADC采样功能。

2.3 高级定时器概述

2.3.1 捕获功能

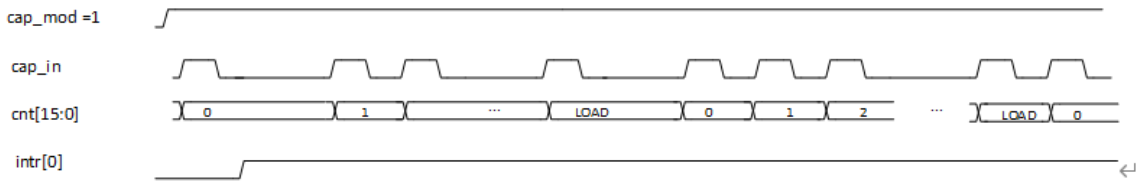
1. PWM输入捕获模式只在PWM0,PWM2,PWM4,PWM6,PWM8上有效。
2. PWM输入捕获模式只有脉冲计数。
3. 脉冲计数有两个模式，模式一是一旦发现脉冲就会产生中断，模式二是当脉冲数 >= load值的时候才产生中断。

- Capture

对 GPIO 的输入信号进行捕获， GPIO 信号的上升沿/下降沿/双沿都可以用来做捕获的触发信号。对应寄存器：cap_edg

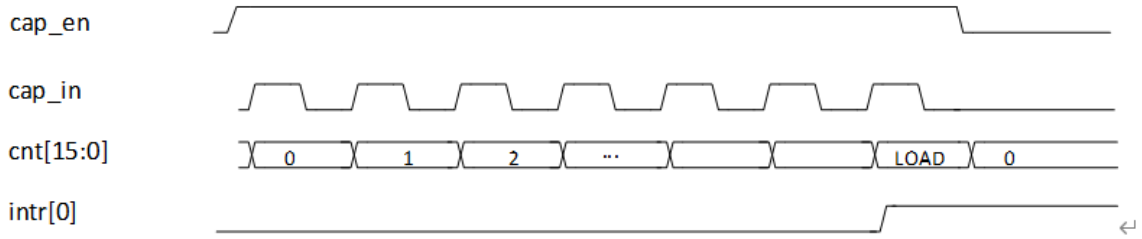
配置 LOAD --- 用来指明 cnt 的周期
配置 cap_en
配置 cap_mod/cap_edg

cap_mod=1
将捕获到的信号的数量采样输出，通过 cnt 寄存器观测。



注：cap_in 只要被捕获到，就会产生中断。

cap_mod =0



注：计数器会被清零，cap_en 也会被拉低，只产生一次中断。

2.3.2 脉冲输出功能（PWM）

1. PWM支持是三种对齐方式
    - ADV_TIMER_CNT_MODE_INC 是计数器向上计数，对应的PWM就是左对齐。
    - ADV_TIMER_CNT_MODE_DEC 是计数器向下计数，对应的PWM就是右对齐。

    - ADV_TIMER_CNT_MODE_BOTH　是计数器先向上计数然后向下计数，对应的PWM就是中间对齐，中间对齐时要注意PWM的频率左右对齐的二分之一（相同LOAD值的情况下）。
2. 配置PWM死区的时候，会自动让pwma和pwmb形成互补，此时，如果想改变波形的占空比只能通过改变pwma的通道值，改变pwmb无效。
3. LOAD值设置为65535的时候会无法产生LOAD中断。
4. 由于PWM中断标志位是通过W1C ISRR寄存器，因此无法通过写位域来清除相应标志位（因为写位域会先读再逻辑或再写，写的时候会把别的标志位也置1）。
5. PWM占空比设置到100%的时候，波形会出现毛刺。
6. PWM周期 p = (load) * 1 / (APB / DIV) s，占空比为 cmp_a / load,其中 load 为加载值，cmp_a 为比较值，APB为外设时钟，DIV为高级定时器的分频。

2.3.3　触发功能

由于触发功能需要配合ADC使用，所以触发功能代码请参考ADC部分

2.4　示例代码

```
/**
 * @file     ln_adv_timer_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-20
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */


/**
    PWM使用说明：

    1. 引脚说明：
            B5  ->  PWM引脚1
            B6  ->  PWM引脚2
            B7  ->  脉冲计数输入引脚

    2. 配置PWM死区的时候，会自动让pwma和pwmb形成互补，此时，如果想改变波形的占空比只能通过改变pwma的通道值，改变pwmb无效,死区时间 = dead_gap_value * 1 / (APB / DIV) s

    3. 如果想要改变PWM对齐方式，只需要改变cnt_mode这个寄存器，就可以了。

        * ADV_TIMER_CNT_MODE_INC        是计数器向上计数，对应的PWM就是左对齐。
        * ADV_TIMER_CNT_MODE_DEC        是计数器向下计数，对应的PWM就是右对齐。
        * ADV_TIMER_CNT_MODE_BOTH       是计数器先向上计数然后向下计数，对应的PWM就是中间对齐，中间对齐时要注意PWM的频率左右对齐的二分之一（相同LOAD值的情况下）。

    4. 请注意，load寄存器只能在PWM失能的情况下修改，否则无效。

    5. 设置PWM占空比的时候，请不要设置成100%，否则会出现毛刺！

    6. PWM的ISRR寄存器，是W1C，因此不支持使用位域的方式清除中断标志位。直接SET ISRR寄存器相应的标志位为1来清除中断标志位。
```

```
    7. PWM周期 p = (load) * 1 / (APB / DIV) s，占空比为 d = (load - cmp_a) / load,
其中 load 为加载值，cmp_a为比较值，APB为外设时钟，DIV为高级定时器的分频。
*/


#include "ln_adv_timer_test.h"
#include "ln_test_common.h"
#include "hal/hal_adv_timer.h"
#include "hal/hal_gpio.h"
#include "log.h"
#include "reg_adv_timer.h"

void ln_adv_timer_test(void)
{
        __NVIC_SetPriorityGrouping(4);
    __enable_irq();

    /* pwm 引脚初始化 */
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_5,ADV_TIMER_PWM0);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_6,ADV_TIMER_PWM1);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_7,ADV_TIMER_PWM2);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_5,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_6,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_7,HAL_ENABLE);

    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));         //清零结构体
    gpio_init.dir = GPIO_OUTPUT;                    //配置GPIO方向，输入或者输出
    gpio_init.pin = GPIO_PIN_8;                     //配置GPIO引脚号
    gpio_init.speed = GPIO_HIGH_SPEED;              //设置GPIO速度
    hal_gpio_init(GPIOB_BASE,&gpio_init);           //初始化GPIO

    /* PWM参数初始化 */
    adv_tim_init_t_def adv_tim_init;
    memset(&adv_tim_init,0,sizeof(adv_tim_init));
    adv_tim_init.adv_tim_clk_div = 0;                                   //设置时钟分
频，0为不分频
    adv_tim_init.adv_tim_load_value = 40000 - 1;                        //设置PWM频
率，1 / (40 / (1 / 40000)) = 1k
    adv_tim_init.adv_tim_cmp_a_value = 20000 ;                          //设置通道a比较
值，占空比为 50%
    adv_tim_init.adv_tim_cmp_b_value = 20000 ;                          //设置通道b比较
值，占空比为 50%
    adv_tim_init.adv_tim_dead_gap_value = 1000;                         //设置死区时间
    adv_tim_init.adv_tim_dead_en = ADV_TIMER_DEAD_DIS;                  //不开启死区
    adv_tim_init.adv_tim_cnt_mode = ADV_TIMER_CNT_MODE_INC;             //向上计数模式
    adv_tim_init.adv_tim_cha_en = ADV_TIMER_CHA_EN;                     //使能通道a
    adv_tim_init.adv_tim_chb_en = ADV_TIMER_CHB_EN;                     //使能通道b
    adv_tim_init.adv_tim_cha_it_mode = ADV_TIMER_CHA_IT_MODE_INC;       //使能通道a向上
计数中断
    adv_tim_init.adv_tim_chb_it_mode = ADV_TIMER_CHB_IT_MODE_INC;       //使能通道b向上
计数中断
    hal_adv_tim_init(ADV_TIMER_0_BASE,&adv_tim_init);                   //初始化
ADV_TIMER0


    /* 输入捕获-脉冲计数配置初始化 */
```

```
    memset(&adv_tim_init,0,sizeof(adv_tim_init));
    adv_tim_init.adv_tim_clk_div = 0;                                   //设置时钟分
频，0为不分频
    adv_tim_init.adv_tim_cap_edg = ADV_TIMER_EDG_RISE;                  //设置捕获上升
沿
    adv_tim_init.adv_tim_cap_mode = ADV_TIMER_CAP_MODE_2;               //设置捕获模式2
    adv_tim_init.adv_tim_cap_en = ADV_TIMER_CAP_EN;                     //捕获模式使能
    adv_tim_init.adv_tim_load_value = 0xFFFF;
    hal_adv_tim_init(ADV_TIMER_1_BASE,&adv_tim_init);                   //初始化
ADV_TIMER1

    /* enable the pwm interrupt */
    hal_adv_tim_it_cfg(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_CMPA,HAL_ENABLE); //使
能通道a中断
    hal_adv_tim_it_cfg(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_CMPB,HAL_ENABLE); //使
能通道b中断
    hal_adv_tim_it_cfg(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_LOAD,HAL_ENABLE); //使
能加载值中断

//    hal_adv_tim_it_cfg(ADV_TIMER_1_BASE,ADV_TIMER_IT_FLAG_CMPA,HAL_ENABLE); //
使能通道a中断
//    hal_adv_tim_it_cfg(ADV_TIMER_1_BASE,ADV_TIMER_IT_FLAG_CMPB,HAL_ENABLE); //
使能通道b中断
//    hal_adv_tim_it_cfg(ADV_TIMER_1_BASE,ADV_TIMER_IT_FLAG_LOAD,HAL_ENABLE); //
使能加载值中断

    NVIC_SetPriority(ADV_TIMER_IRQn,     4);
    NVIC_EnableIRQ(ADV_TIMER_IRQn);

    uint32_t cmp_a_value = 0;       //通道a的比较值
    float    duty_value  = 0;       //占空比
    uint8_t  inc_dir     = 0;       //占空比递增/减方向
    uint32_t pulse_cnt   = 0;       //脉冲计数

    while(1)
    {
        if(inc_dir == 0)
            duty_value += 0.01;
        else
            duty_value -= 0.01;

        if(duty_value >= 0.99)
            inc_dir = 1;
        if(duty_value <= 0.01)
            inc_dir = 0;

        hal_adv_tim_set_comp_a(ADV_TIMER_0_BASE,duty_value * 40000);

        ln_delay_ms(500);

        pulse_cnt = hal_adv_tim_get_trig_value(ADV_TIMER_1_BASE);
    }
}

void ADV_TIMER_IRQHandler()
{
    if(hal_adv_tim_get_it_flag(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_CMPB) == 1)
    {
```

```
        hal_adv_tim_clear_it_flag(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_CMPB);
        //hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_7);
    }
    if(hal_adv_tim_get_it_flag(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_CMPA) == 1)
    {
        hal_adv_tim_clear_it_flag(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_CMPA);
        //hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_8);
    }
    if(hal_adv_tim_get_it_flag(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_LOAD) == 1)
    {
        hal_adv_tim_clear_it_flag(ADV_TIMER_0_BASE,ADV_TIMER_IT_FLAG_LOAD);
        //hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_7);
    }

    if(hal_adv_tim_get_it_flag(ADV_TIMER_1_BASE,ADV_TIMER_IT_FLAG_LOAD) == 1)
    {
        hal_adv_tim_clear_it_flag(ADV_TIMER_1_BASE,ADV_TIMER_IT_FLAG_LOAD);
        hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_8);
    }
}
```

3 基本定时器（Timer）

3.1 定时器简介

定时器是一个通过可编程预分频器驱动的24位自动装载计数器构成。

3.2 定时器主要特征

- 8位可编程(可以实时修改)预分频器，计数器时钟频率的分频系数为0～255之间的任意数值，0为不分频。
- 四个完全独立的Timer
- 可以实时修改PWM占空比（但是只有CNT值为0的时候才能加载最新的LOAD值）
- 24位的向下计数器

3.3 定时器概述

Timer的分频配置是在REG_CMP中，配置完成之后，需要手动触发。详细请参考测试代码。

3.3.1 计数器功能

计数器功能有两种模式：

- 自由运行模式，计数器会自动加载最大值，Timer定时时间为 ((1 / APB_CLOCK / TIM_DIV) * tim_cnt) s
- 用户定义模式时，计数器会自动加载我们设定的计数值，Timer定时时间为 ((1 / APB_CLOCK / TIM_DIV) * 16777215) s

3.3.2 脉冲输出功能（PWM）

- 使用PWM时，配置PWM使能位，然后配置tim_cnt2,则PWM周期为 ((1 / APB_CLOCK / TIM_DIV) * (tim_cnt + 1 + tim_cnt2 + 1))s，其中 tim_cnt + 1 的时间为低电平时间，tim_cnt2 + 1 的时间为高电平时间。使用PWM时，定时器的中断会有两次。
- PWM占空比不可设置为0%或者100%,否则会出现毛刺，原因是因为设置的定时都是 tim_cnt + 1 的，所以最终还是有一点点跳变的。
- 可以实时修改PWM占空比（但是只有CNT值为0的时候才能加载最新的LOAD值）

3.4 示例代码

```
/**
 * @file     ln_timer_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-20
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/*
    Timer使用说明：

    1. Timer Mode寄存器使用方式：
                                0 ：  使用自由运行模式时，计数器会自动加载最大值。
                                1 ：  使用用户定义模式时，计数器会自动加载我们设定的计数值。


    2. 不使用PWM的情况下，定时时间为 ((1 / SystemCoreClock) * tim_cnt) s

    3. 使用PWM时，配置PWM使能位，然后配置tim_cnt2,则PWM周期为  ((1 / SystemCoreClock) * (tim_cnt + 1 + tim_cnt2 + 1))s，

       其中 tim_cnt + 1 的时间为低电平时间，tim_cnt2 + 1 的时间为高电平时间。使用PWM时，定时器的中断会有两次。

    4. PWM占空比不可设置为0%或者100%,否则会出现毛刺，原因是因为设置的定时都是 tim_cnt + 1 的，所以最终还是有一点点跳变的。

    5. 测试代码中，Timer0，Timer2，Timer3作为定时器使用，Timer1作为PWM使用

    6. 测试引脚：
                    PB5     Timer0 中断翻转IO
                    PB6     Timer2 中断翻转IO
                    PB7     Timer3 中断翻转IO
                    PB8     Timer1 PWM IO

    7. PWM的占空比在while(1)循环中，每隔500ms改变一次。
```

```
*/


#include "ln_timer_test.h"
#include "hal/hal_timer.h"
#include "hal/hal_gpio.h"
#include "ln_test_common.h"
#include "log.h"

void ln_timer_test(void)
{
    //GPIO10是测试使用,GPIO11是PWM输出引脚
    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));   
;     //清零结构体
    gpio_init.dir =
GPIO_OUTPUT;            &n
bsp;    //配置GPIO方向，输入或者输出
    gpio_init.pin =
GPIO_PIN_5;           &nb
sp;         //配置GPIO引脚号
    gpio_init.speed =
GPIO_HIGH_SPEED;          &nbs
p; //设置GPIO速度
    hal_gpio_init(GPIOB_BASE,&gpio_init);        //初始化GPIO

    gpio_init.pin =
GPIO_PIN_6;           &nb
sp;         //配置GPIO引脚号
    hal_gpio_init(GPIOB_BASE,&gpio_init);    
    gpio_init.pin =
GPIO_PIN_7;           &nb
sp;         //配置GPIO引脚号
    hal_gpio_init(GPIOB_BASE,&gpio_init);    

    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_8,TIMER1_PWM);     //配置TIMER1 PWM引
脚复用
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_8,HAL_ENABLE);         //使能引脚复用



    //Timer初始化
    tim_init_t_def tim_init;
    memset(&tim_init,0,sizeof(tim_init));

    tim_init.tim_cnt = 40000 - 1;                        //设置加载值，定时时间 = 1 /
40000000 * 40000 = 1ms
    tim_init.tim_mode = TIM_USER_DEF_CNT_MODE;           //设置定时器模式，设置为用户定义模
式

    tim_init.tim_div = 2;

    //Timer0初始化
    hal_tim_init(TIMER0_BASE,&tim_init);                 //初始化定时器
    hal_tim_en(TIMER0_BASE,HAL_ENABLE);                  //使能定时器模块
    NVIC_SetPriority(TIMER0_IRQn,     4);                //配置定时器优先级
    NVIC_EnableIRQ(TIMER0_IRQn);                         //使能定时器中断
    hal_tim_it_cfg(TIMER0_BASE,TIM_IT_FLAG_ACTIVE,HAL_ENABLE);      //配置中断
```

```
    //Timer2初始化
    hal_tim_init(TIMER2_BASE,&tim_init);              //初始化定时器
    hal_tim_en(TIMER2_BASE,HAL_ENABLE);               //使能定时器模块
    NVIC_SetPriority(TIMER2_IRQn,     4);             //配置定时器优先级
    NVIC_EnableIRQ(TIMER2_IRQn);                      //使能定时器中断
    hal_tim_it_cfg(TIMER2_BASE,TIM_IT_FLAG_ACTIVE,HAL_ENABLE);       //配置中断

    //Timer3初始化
    hal_tim_init(TIMER3_BASE,&tim_init);              //初始化定时器
    hal_tim_en(TIMER3_BASE,HAL_ENABLE);               //使能定时器模块
    NVIC_SetPriority(TIMER3_IRQn,     4);             //配置定时器优先级
    NVIC_EnableIRQ(TIMER3_IRQn);                      //使能定时器中断
    hal_tim_it_cfg(TIMER3_BASE,TIM_IT_FLAG_ACTIVE,HAL_ENABLE);       //配置中断

    //Timer PWM初始化
    tim_init.tim_cnt2 = 40000 - 1;                    //高电平时间 = 1 / 40000000 *
40000 = 1ms，所以PWM高电平时间=1ms,低电平时间 = 1ms.
    hal_tim_init(TIMER1_BASE,&tim_init);              //初始化定时器
    hal_tim_en(TIMER1_BASE,HAL_ENABLE);               //使能定时器模块
    hal_tim_pwm_en(TIMER1_BASE,HAL_ENABLE);           //使能PWM
    NVIC_SetPriority(TIMER1_IRQn,     1);             //配置定时器优先级
    NVIC_EnableIRQ(TIMER1_IRQn);                      //使能定时器中断
    hal_tim_it_cfg(TIMER1_BASE,TIM_IT_FLAG_ACTIVE,HAL_ENABLE);       //配置中断

    float duty = 0.01;
    uint8_t cnt_dir = 0;
    while(1)
    {

        /*占空比实时改变，CNT值为0时生效*/
        if(cnt_dir == 0)
            duty += 0.01;
        else
            duty -= 0.01;

        if(duty >= 0.98)
            cnt_dir = 1;
        if(duty <= 0.02)
            cnt_dir = 0;

        hal_tim_set_cnt_value(TIMER1_BASE,duty * 80000);
        hal_tim_set_cnt2_value(TIMER1_BASE,(1-duty) * 80000);

        hal_tim_set_cnt_value(TIMER0_BASE,duty * 80000);
        hal_tim_set_cnt2_value(TIMER0_BASE,(1-duty) * 80000);
        ln_delay_ms(500);
    }
}

void TIMER0_IRQHandler()
{
    if(hal_tim_get_it_flag(TIMER0_BASE,TIM_IT_FLAG_ACTIVE) == 1)
    {
        hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_5);                    //翻转GPIO，便于
测试
        hal_tim_clr_it_flag(TIMER0_BASE,TIM_IT_FLAG_ACTIVE);       //清除标志位
    }
```

```
}

void TIMER1_IRQHandler()
{
    if(hal_tim_get_it_flag(TIMER1_BASE,TIM_IT_FLAG_ACTIVE) == 1)
    {
        //hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_6);                //翻转GPIO，便于测
试
        hal_tim_clr_it_flag(TIMER1_BASE,TIM_IT_FLAG_ACTIVE);       //清除标志位
    }
}

void TIMER2_IRQHandler()
{
    if(hal_tim_get_it_flag(TIMER2_BASE,TIM_IT_FLAG_ACTIVE) == 1)
    {
        hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_6);                    //翻转GPIO，便于
测试
        hal_tim_clr_it_flag(TIMER2_BASE,TIM_IT_FLAG_ACTIVE);       //清除标志位
    }
}

void TIMER3_IRQHandler()
{
    if(hal_tim_get_it_flag(TIMER3_BASE,TIM_IT_FLAG_ACTIVE) == 1)
    {
        hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_7);                    //翻转GPIO，便于
测试
        hal_tim_clr_it_flag(TIMER3_BASE,TIM_IT_FLAG_ACTIVE);       //清除标志位
    }
}
```

4 看门狗（WDT）

4.1 看门狗简介

看门狗通常被用来监测，由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在设定值变成0前被刷新，看门狗电路在达到预置的时间周期时，会产生一个MCU复位。

4.2 看门狗主要特征

- 独立的32K时钟
- 可配置的看门狗中断

4.3 看门狗概述

- 通过设置WDT运行模式来设置WDT是否产生中断。模式0：计数器溢出直接复位； 模式1：计数器溢出先产生中断，如果再次溢出则复位。
- 喂狗时间 = 2^(8 + TOP) * (1/32k);

4.4 示例代码

```c
/**
 * @file     ln_wdt_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-24
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/*
        WDT使用说明：
                    1. WDT使用的是一个单独的32k时钟。
                    2. 通过设置WDT运行模式来设置WDT是否产生中断。0：计数器溢出直接复位；1：计数器溢出先产生中断，如果再次溢出则复位。
                    3. 喂狗时间 = 2^(8 + TOP) * (1/32k);

*/


#include "ln_wdt_test.h"
#include "hal/hal_wdt.h"
#include "hal/hal_gpio.h"
#include "log.h"

void ln_wdt_test(void)
{
    /* 引脚初始化 */
    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));
    gpio_init.dir = GPIO_OUTPUT;
    gpio_init.pin = GPIO_PIN_5;
    gpio_init.speed = GPIO_HIGH_SPEED;
    hal_gpio_init(GPIOB_BASE,&gpio_init);
    hal_gpio_pin_reset(GPIOB_BASE,GPIO_PIN_5);

    /* 看门狗初始化 */
    wdt_init_t_def wdt_init;
    memset(&wdt_init,0,sizeof(wdt_init));
    wdt_init.wdt_rmod = WDT_RMOD_1;          //等于0的时候，计数器溢出时直接复位，等于1的时候，先产生中断，如果再次溢出，则产生复位。
    wdt_init.wdt_rpl = WDT_RPL_32_PCLK;      //设置复位延时的时间
    wdt_init.top = WDT_TOP_VALUE_1;          //设置看门狗计数器的值,当TOP=1时，对应计数器的值为0x1FF，而看门狗是用的时钟是一个单独的32k时钟，
                                             //所以此时的喂狗时间必须在 (1/32k) * 0x1FF 内。
    hal_wdt_init(WDT_BASE,&wdt_init);

    /* 配置看门狗中断 */
    NVIC_SetPriority(WDT_IRQn,     4);
    NVIC_EnableIRQ(WDT_IRQn);

    /* 使能看门狗 */
    hal_wdt_en(WDT_BASE,HAL_ENABLE);

    /* 先喂下狗 */
```

```
    hal_wdt_cnt_restart(WDT_BASE);

    /* 测试引脚 */
    hal_gpio_pin_set(GPIOB_BASE,GPIO_PIN_5);
    while(1)
    {

    }
}

void WDT_IRQHandler()
{
    hal_wdt_cnt_restart(WDT_BASE);                //喂狗操作
    hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_5);       //测试引脚翻转
    LOG(LOG_LVL_INFO,"feed dog~! \r\n");          //LOG打印
}
```

5　时钟控制（CLOCK）

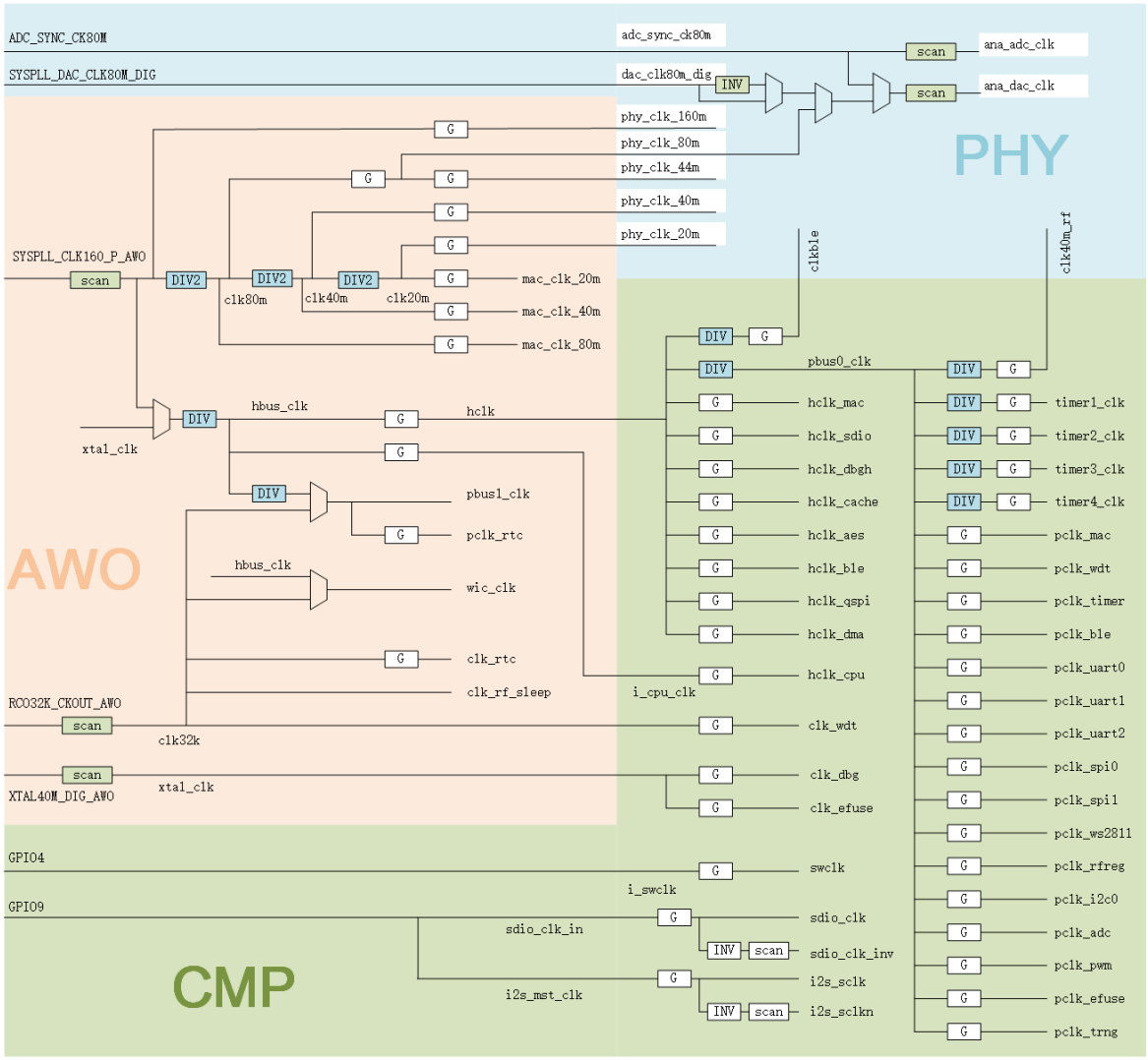
5.1　时钟简介

支持两种不同的时钟源驱动CPU时钟：

- XTAL 40M时钟（直接使用外部40M的晶振时钟）
- PLL 160M时钟（使用外部晶振倍频到的160M时钟，虽然可以通过修改倍频器改变这个值，但是由于WIFI RF参数问题，PLL必须为160M）

同时LN8620支持32K的内部低速RC振荡器，用于驱动独立看门狗、RTC，用于从睡眠模式下自动唤醒系统。

具体请参考下图的时钟树：



5.2 分频器简介

用户可自定义配置的分频器有：

- PLL倍频器，只有使用了PLL时钟，该倍频器才有效果，可配置倍频系数 1 - 6倍频
- AHB分频器，可配置分频系数 1 - 16分频
- APB分频器，可配置分频系数 1- 16分频

```
        if use pll clock:
                        APB  Clock = XTAL_CLOCK * PLL_MUL / AHB_DIV /
APB_DIV
                        AHB  Clock = XTAL_CLOCK * PLL_MUL / AHB_DIV
                        Core Clock = XTAL_CLOCK * PLL_MUL
        if not use pll clock:

                        APB  Clock = XTAL_CLOCK / AHB_DIV / APB_DIV
                        AHB  Clock = XTAL_CLOCK / AHB_DIV
                        Core Clock = XTAL_CLOCK
```

一般的，没有特殊情况的时候，我们将固定 XTAL_CLOCK = 40Mhz，PLL_CLOCK = 160Mhz。

有一些外设里面也有内部的分频器，如Timer、ADC_Timer等等，这些分频器的功能请参考相关外设描述。

用户不可配置的分频器：

- QSPI分频器，可以配置为0 - 65535 中任一偶数分频，设置为0会导致时钟失能，该分频器不推荐用户自行去配置，因为不正确配置这个分频器会打乱QSPI的时钟时序，导致程序异常，所以我们一般默认QSPI分频器为AHB时钟的二分频

5.3 示例代码

```
#include "hal/hal_common.h"
#include "hal/hal_misc.h"
#include "hal/hal_clock.h"
void SetSysClock(void)
{
/*
    if use pll clock:
                        APB0 Clock = XTAL_CLOCK * PLL_MUL / AHB_DIV / APB0_DIV
                        AHB  Clock = XTAL_CLOCK * PLL_MUL / AHB_DIV
                        Core Clock = XTAL_CLOCK * PLL_MUL
    if not use pll clock:

                        APB0 Clock = XTAL_CLOCK / AHB_DIV / APB0_DIV
                        AHB  Clock = XTAL_CLOCK / AHB_DIV
                        Core Clock = XTAL_CLOCK
*/
    clock_init_t clock_init;
    memset((void *)(&clock_init),0,sizeof(clock_init));

    /*
        APB0 Clock = 80M
        AHB  Clock = 160M
        Core Clock = 160M
    */
    clock_init.clk_src          = CLK_SRC_PLL;
    clock_init.clk_pclk0_div    = CLK_PCLK0_2_DIV;
    clock_init.clk_hclk_div     = CLK_HCLK_NO_DIV;
    clock_init.clk_pllclk_mul   = CLK_PLL_CLK_4_MUL;

    hal_clock_init(&clock_init);
    SystemCoreClock = hal_clock_get_src_clk();
}
```

6 实时时钟（RTC）

6.1 RTC简介

RTC是一个独立的计数器，使用MCU内部单独的32K时钟，可以MCU深度休眠时使用。

6.2 RTC主要特征

- 32位的计数器，使用MCU内部32K的时钟源
- 支持在MCU深度休眠时使用
- 支持一个可屏蔽中断

6.3 RTC概述

6.3.1 计数器功能

配置说明

1. rtc_warp_en = 1时，当计数器达到匹配值时会从0开始重新计数，否则计数器一直会计数下去，直到溢出。
2. RTC会从load计数到match。
3. 定时时间 T = (match - load) * (1 / 32k) s
4. 使用RTC定时比较长的时间时，可以使用SleepTimer进行校准。
    - 先进行校准
    - 使用校准值对32K进行校准，例如设置定时 1s时，rtc_match = (32000 - 0) * (1 / 32K) * (32k_cal / 32k),32k_cal为校准值。

6.4 示例代码

```
/**
        RTC使用说明：
                    1. RTC会从load计数到match，所以LOAD寄存器里面的值可以理解为初值。

                    2. rtc_warp_en = 1时，当计数器达到匹配值时会从0开始计数，否则计数器一直会计数下去，直到溢出。

                    3. RTC可以使用SleepTimer的校准值来校准。

                    4. GPIOB5为RTC中断翻转引脚

                    5. 定时时间 T = ( match - load ) * (1 / 32k) s

                    6. 使用RTC定时比较长的时间时，可以使用SleepTimer进行校准。

                        a. 先进行校准

                        b. 使用校准值对32K进行校准

                            例如设置定时 1s时，rtc_match = (32000 - 0) * (1 / 32K) * (32k_cal / 32k),32k_cal为校准值。
*/


#include "ln_rtc_test.h"
#include "hal_rtc.h"
#include "hal/hal_gpio.h"
#include "hal/hal_misc.h"
#include "log.h"

void ln_rtc_test(void)
{
    //使用SleepTimer对32K的时钟进行校准
    hal_misc_awo_set_o_cpu_sleep_counter_bp(0);

    //hal_misc_awo_get_i_rco32k_period_ns();    //32K的校准值
```

```
    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));         //清零结构体
    gpio_init.dir = GPIO_OUTPUT;                    //配置GPIO方向，输入或者输出
    gpio_init.pin = GPIO_PIN_5;                     //配置GPIO引脚号
    gpio_init.speed = GPIO_HIGH_SPEED;              //设置GPIO速度
    hal_gpio_init(GPIOB_BASE,&gpio_init);           //初始化GPIO

    //RTC初始化
    rtc_init_t_def rtc_init;
    rtc_init.rtc_warp_en = RTC_WRAP_EN;                         //达到匹配值之后从0
计数
    hal_rtc_init(RTC_BASE,&rtc_init);

    //RTC会从load计数到match，((32-0)*(1/32000)) = 1ms
    hal_rtc_set_cnt_match(RTC_BASE,32);                         //设置匹配计数器值
    hal_rtc_set_cnt_load(RTC_BASE,0);                           //设置初始计数器值
    hal_rtc_it_cfg(RTC_BASE,RTC_IT_FLAG_ACTIVE,HAL_ENABLE);     //配置中断

    NVIC_SetPriority(RTC_IRQn,4);
    NVIC_EnableIRQ(RTC_IRQn);

    hal_rtc_en(RTC_BASE,HAL_ENABLE);                            //使能RTC

    while(1);
}


void ln_sleeptimer_test(void)
{

    //使用之前先做校准
    hal_misc_awo_set_o_cpu_sleep_counter_bp(0);

    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));         //清零结构体
    gpio_init.dir = GPIO_OUTPUT;                    //配置GPIO方向，输入或者输出
    gpio_init.pin = GPIO_PIN_5;                     //配置GPIO引脚号
    gpio_init.speed = GPIO_HIGH_SPEED;              //设置GPIO速度
    hal_gpio_init(GPIOB_BASE,&gpio_init);           //初始化GPIO

    hal_misc_awo_set_o_cpu_sleep_time_l(320);       //((320-0)*(1/32000)) = 10ms
    hal_misc_awo_set_o_cpu_sleep_time_l(0);
    hal_misc_awo_set_o_cpu_sleep_inten(1);

    NVIC_SetPriority(RFSLP_IRQn,1);  //配置中断优先级
    NVIC_EnableIRQ(RFSLP_IRQn);
}

void RFSLP_IRQHandler()
{
    hal_misc_awo_set_o_cpu_sleep_inten(0);
    hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_5);
}

void RTC_IRQHandler()
{
   if(hal_rtc_get_it_flag(RTC_BASE,RTC_IT_FLAG_ACTIVE) == 1)
   {
```

```
        hal_gpio_toggle(GPIOB_BASE,GPIO_PIN_5);                    //翻转引脚
        hal_rtc_clear_it_flag(RTC_BASE,RTC_IT_FLAG_ACTIVE);        //清除标志位
    }
}
```

7 通用异步收发器（UART）

7.1 UART介绍

UART是一种通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输和接收。

7.2 UART主要特征

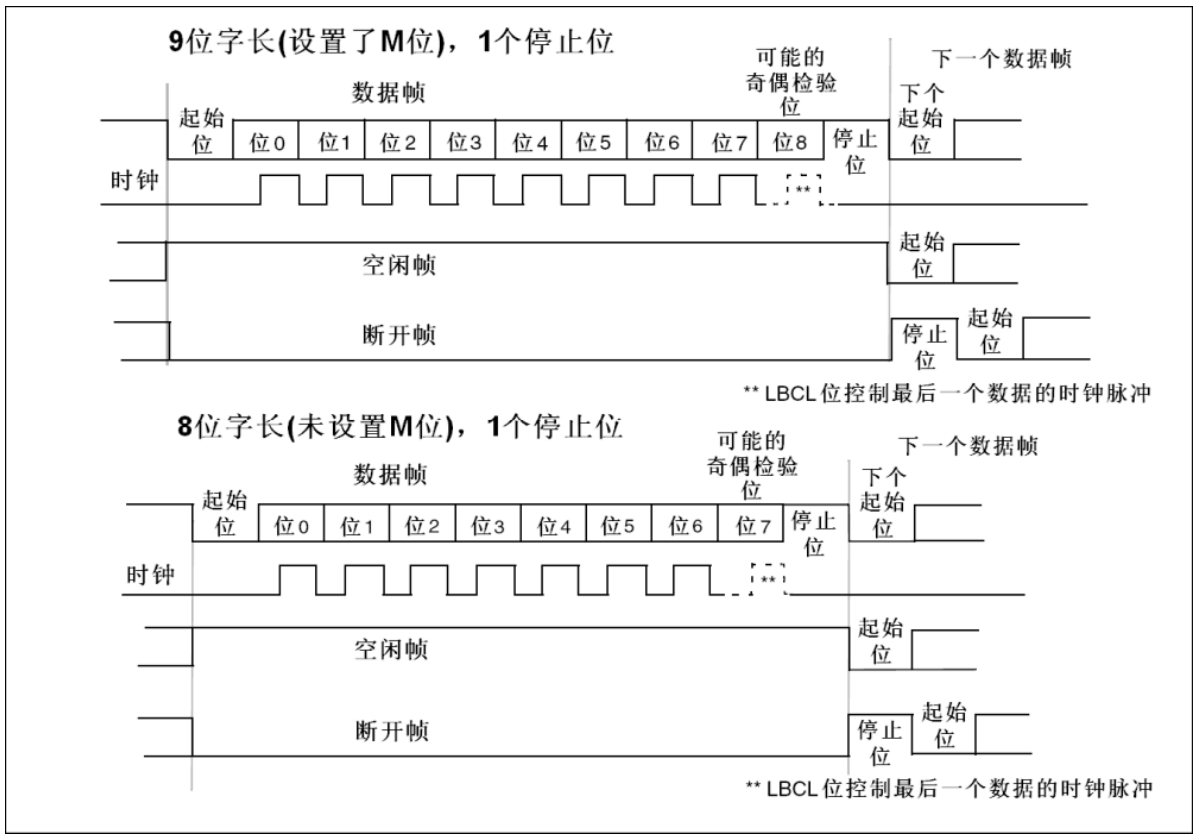
- 全双工的，异步通信
- 可编程数据字长度(8位或9位)
- 可配置的停止位 - 支持0.5、1、1.5、2这四种停止位配置
- 单独的发送器和接收器使能位
- 16字节的发送缓冲器和16字节的接收缓冲器
- 检测标志
  - 接收缓冲器满
  - 发送缓冲器空
  - 传输结束标志
- 校验控制
  - 发送校验位
  - 对接收数据进行校验
- 四个错误检测标志
  - 溢出错误
  - 噪音错误
  - 帧错误
  - 校验错误
- 9个带标志的中断源
  - CTS改变
  - 发送数据寄存器空
  - 发送完成
  - 接收数据寄存器满
  - 检测到总线为空闲
  - 溢出错误
  - 帧错误
  - 噪音错误
  - 校验错误
- 从静默模式中唤醒(通过空闲总线检测或地址标志检测)
- 两种唤醒接收器的方式：地址位(MSB，第9位)，总线空闲

7.3 UART功能概述

接口通过三个引脚与其他设备连接在一起。任何UART双向通信至少需要两个脚：接收数据输入(RX)和发送数据输出(TX)。

RX： 接收数据串行输。通过过采样技术来区别数据和噪音，从而恢复数据。
TX： 发送数据输出。当发送器被禁止时，输出引脚恢复到它的I/O端口配置。当发送器被激活，并且不发送数据 时， TX引脚处于高电平。

字长可以通过编程**UART_CR1** 寄存器中的**M** 位，选择成8或9位。在起始位期间， TX脚处于低电平，在停止位期间处于高电平。空闲符号被视为完全由’1’组成的一个完整的数据帧，后面跟着包含了数据的下一帧的开始位(‘1’的位数也包括了停止位的位数)。断开符号 被视为在一个帧周期内全部收到’0’(包括停止位期间，也是’0’)。在断开帧结束时，发送器再插入1或2个停止位(‘1’)来应答起始位。发送和接收由一共用的波特率发生器驱动，当发送器和接收器的使能位分别置位时，分别为其产生时钟。



7.3.1 UART发送器

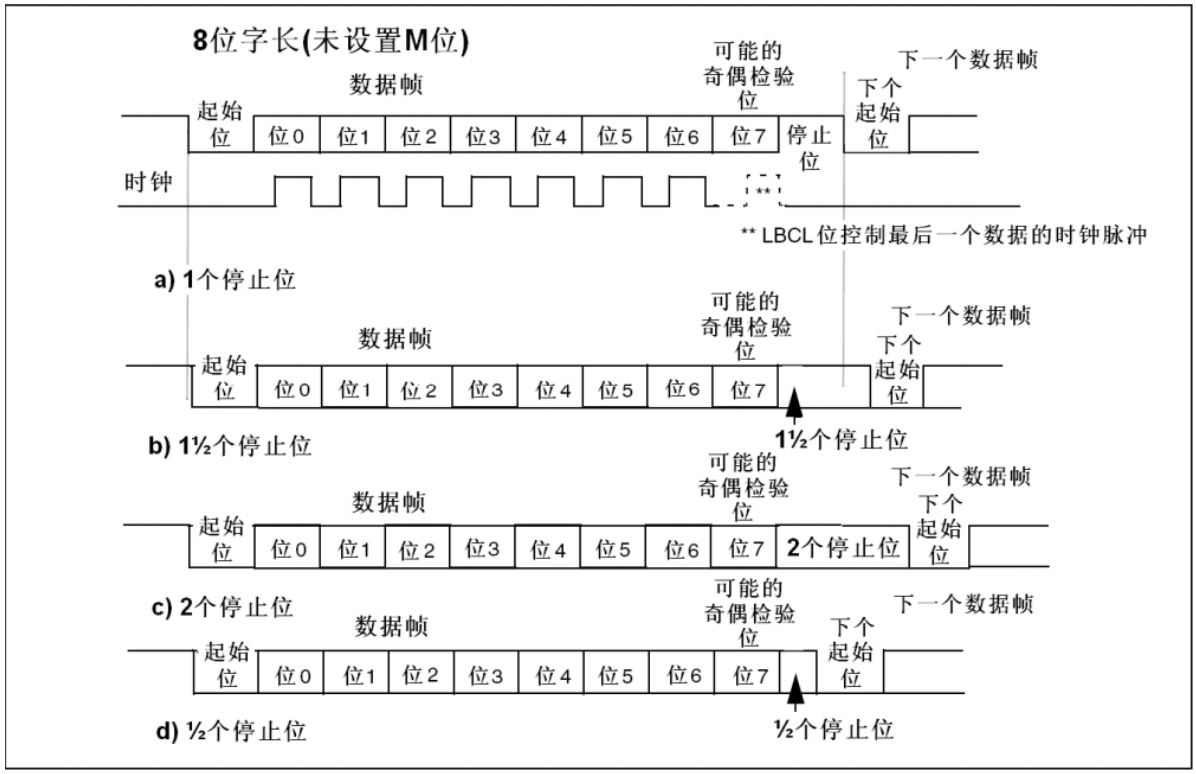
发送器根据M位的状态发送8位或9位的数据字。当发送使能位(TE)被设置时，发送移位寄存器中的数据在TX脚上输出 。在UART发送期间，在TX引脚上首先移出数据的最低有效位。在此模式里， UART_DR寄存器包含了一个内部总线和发送移位寄存器之间的缓冲器 。

每个字符之前都有一个低电平的起始位；之后跟着的停止位，其数目可配置。

UART支持多种停止位的配置： 0.5、 1、 1.5和2个停止位。

> 注：
>
> 1. 在数据传输期间不能复位TE位，否则将破坏TX脚上的数据，因为波特率计数器停止计数。 正在传输的当前数据将丢失。
> 2. TE位被激活后将发送一个空闲帧。

空闲帧包括了停止位。
断开帧是10位低电平，后跟停止位(当m=0时)；或者11位低电平，后跟停止位(m=1时)。不可能传输更长的断开帧(长度大于10或者11位)。



配置步骤：

1. 通过在UART_CR1寄存器上置位UE位来激活UART
2. 编程UART_CR1的M位来定义字长。
3. 在UART_CR2中编程停止位的位数。
4. 如果采用多缓冲器通信，配置UART_CR3中的DMA使能位(DMAT)。按多缓冲器通信中的描述配置DMA寄存器。
5. 利用UART_BRR寄存器选择要求的波特率。
6. 设置UART_CR1中的TE位，发送一个空闲帧作为第一次数据发送。
7. 把要发送的数据写进UART_DR寄存器(此动作清除TXE位)。在只有一个缓冲器的情况下，对每个待发送的数据重复步骤7。
8. 在UART_DR寄存器中写入最后一个数据字后，要等待TC=1，它表示最后一个数据帧的传输结束。当需要关闭UART或需要进入停机模式之前，需要确认传输结束，避免破坏最后一次传 输。

清零TXE位总是通过对数据寄存器的写操作来完成的。 TXE位由硬件来设置，它表明：

- 数据已经从TDR移送到移位寄存器，数据发送已经开始
- TDR寄存器被清空
- 下一个数据可以被写进UART_DR寄存器而不会覆盖先前的数据

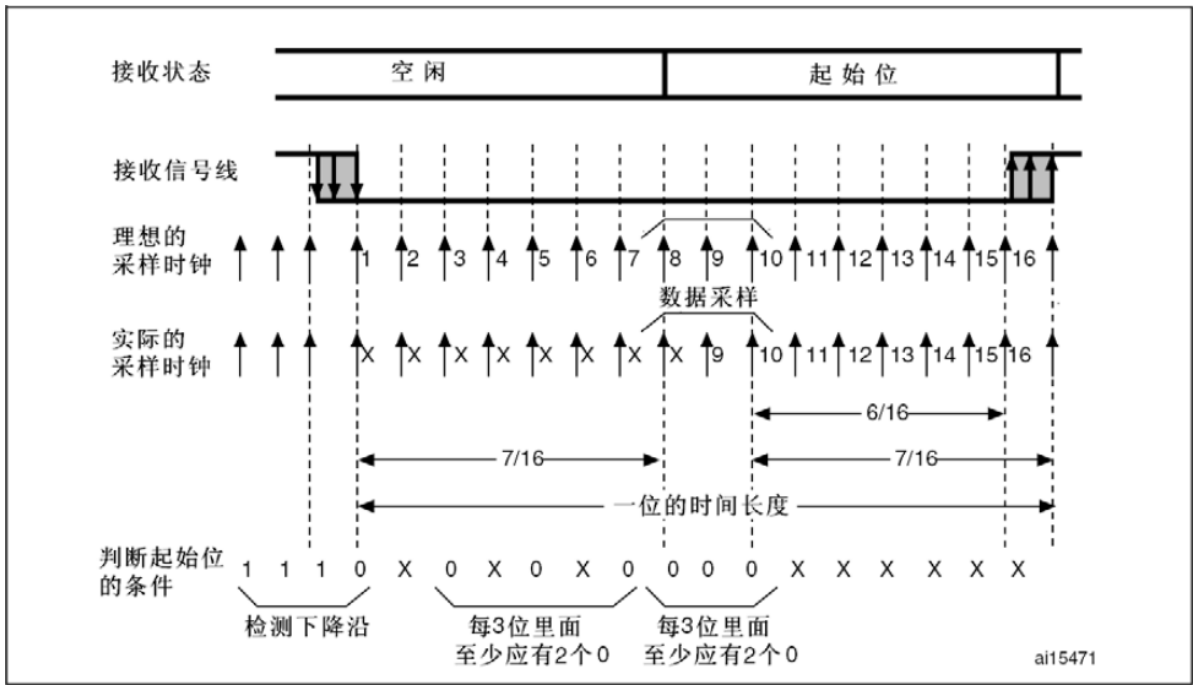
如果TXEIE位被设置，此标志将产生一个中断。

如果此时UART正在发送数据，对UART_DR寄存器的写操作把数据存进TDR寄存器，并在当前传输结束时把该数据复制进移位寄存器。如果此时UART没有在发送数据，处于空闲状态，对UART_DR寄存器的写操作直接把数据放进移位寄存器，数据传输开始， TXE位立即被置起。当一帧发送完成时(停止位发送后)并且设置了TXE位， TC位被置起，如果UART_CR1寄存器中的TCIE位被置起时，则会产生中断。

7.3.2 UART接收器

UART可以根据UART_CR1的M位接收8位或9位的数据字。

起始位侦测
在UART中，如果辨认出一个特殊的采样序列，那么就认为侦测到一个起始位。
该序列为： 1 1 1 0 X 0 X 0 X 0 0 0 0



> 注意： 如果该序列不完整，那么接收端将退出起始位侦测并回到空闲状态(不设置标志位)等待下降沿。如果3个采样点都为’0’(在第3、 5、 7位的第一次采样，和在第8、 9、 10的第二次采样都为’0’)，则确认收到起始位，这时设置RXNE标志位，如果RXNEIE=1，则产生中断。如果两次3个采样点上仅有2个是’0’(第3、 5、 7位的采样点和第8、 9、 10位的采样点)，那么起始位仍然是有效的，但是会设置NE噪声标志位。如果不能满足这个条件，则中止起始位的侦测过程，接收器会回到空闲状态(不设置标志位)。如果有一次3个采样点上仅有2个是’0’(第3、 5、 7位的采样点或第8、 9、 10位的采样点)，那么起始位仍然是有效的，但是会设置NE噪声标志位。

字符接收

在UART接收期间，数据的最低有效位首先从RX脚移进。在此模式里， UART_DR寄存器包含的缓冲器位于内部总线和接收移位寄存器之间。

配置步骤：

1. 将UART_CR1寄存器的UE置1来激活UART。
2. 编程UART_CR1的M位定义字长
3. 在UART_CR2中编写停止位的个数
4. 如果需多缓冲器通信，选择UART_CR3中的DMA使能位(DMAR)。按多缓冲器通信所要求的配置DMA寄存器。
5. 利用波特率寄存器UART_BRR选择希望的波特率。
6. 设置UART_CR1的RE位。激活接收器，使它开始寻找起始位。

当一字符被接收到时

- RXNE位被置位。它表明移位寄存器的内容被转移到RDR。换句话说，数据已经被接收并且可以被读出(包括与之有关的错误标志)。
- 如果RXNEIE位被设置，产生中断。
- 在接收期间如果检测到帧错误，噪音或溢出错误，错误标志将被置起，
- 在多缓冲器通信时， RXNE在每个字节接收后被置起，并由DMA对数据寄存器的读操作而清零。

- 在单缓冲器模式里，由软件读UART_DR寄存器完成对RXNE位清除。 RXNE标志也可以通过对它写0来清除。 RXNE位必须在下一字符接收结束前被清零，以避免溢出错误。

> 在接收数据时， RE位不应该被复位。如果RE位在接收时被清零，当前字节的接收被丢失。

断开符号

当接收到一个断开帧时， USART像处理帧错误一样处理它。

空闲符号

当一空闲帧被检测到时，其处理步骤和接收到普通数据帧一样，但如果IDLEIE位被设置将产生一个中断。

溢出错误

如果RXNE还没有被复位，又接收到一个字符，则发生溢出错误。数据只有当RXNE位被清零后才从移位寄存器转移到RDR寄存器。 RXNE标记是接收到每个字节后被置位的。如果下一个数据已被收或先前DMA请求还没被服务时， RXNE标志仍是置起的，溢出错误产生。

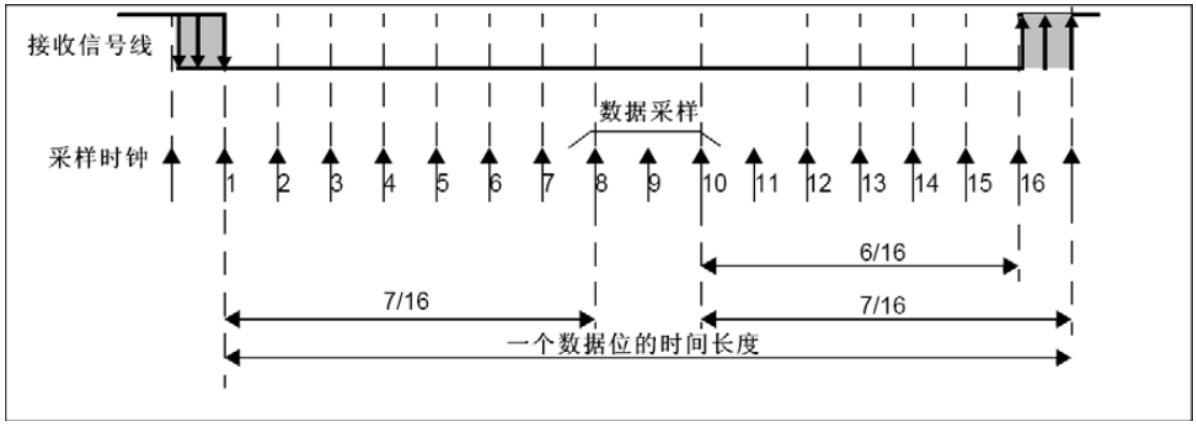
当溢出错误产生时：

- ORE位被置位。
- RDR内容将不会丢失。读UART_DR寄存器仍能得到先前的数据。
- 移位寄存器中以前的内容将被覆盖。随后接收到的数据都将丢失。
- 如果RXNEIE位被设置或EIE和DMAR位都被设置，中断产生。
- 顺序执行对UART_SR和UART_DR寄存器的读操作，可复位ORE位

> 注意： 当ORE位置位时，表明至少有1个数据已经丢失。有两种可能性：
>
> - 如果RXNE=1，上一个有效数据还在接收寄存器RDR上，可以被读出。
> - 如果RXNE=0，这意味着上一个有效数据已经被读走， RDR已经没有东西可读。当上一个有效数据在RDR中被读取的同时又接收到新的(也就是丢失的)数据时，此种情况可能发生。在读序列期间(在USART_SR寄存器读访问和USART_DR读访问之间)接收到新的数据，此种情况也可能发生。

噪音错误

使用过采样技术，通过区别有效输入数据和噪音来进行数据恢复。



采样值	NE状态	接收的位	数据有效性
000	0	0	有效
001	1	0	无效
010	1	0	无效
011	1	1	无效
100	1	0	无效
101	1	1	无效
110	1	1	无效
111	0	1	有效

当在接收帧中检测到噪音时：

- 在RXNE位的上升沿设置NE标志。
- 无效数据从移位寄存器传送到USART_DR寄存器。
- 在单个字节通信情况下，没有中断产生。然而，因为NE标志位和RXNE标志位是同时被设置，RXNE将产生中断。在多缓冲器通信情况下，如果已经设置了USART_CR3寄存器中EIE位，将产生一个中断。先读出USART_SR，再读出USART_DR寄存器，将清除NE标志位

帧错误

当以下情况发生时检测到帧错误：

由于没有同步上或大量噪音的原因，停止位没有在预期的时间上接和收识别出来。

当帧错误被检测到时：

- FE位被硬件置起
- 无效数据从移位寄存器传送到USART_DR寄存器。
- 在单字节通信时，没有中断产生。然而，这个位和RXNE位同时置起，后者将产生中断。在多缓冲器通信情况下，如果USART_CR3寄存器中EIE位被置位的话，将产生中断。顺序执行对USART_SR和USART_DR寄存器的读操作，可复位FE位。

接收期间的可配置的停止位：

被接收的停止位的个数可以通过控制寄存器2的控制位来配置，在正常模式时，可以是1或2个。

1. 0.5个停止位(智能卡模式中的接收)：不对0.5个停止位进行采样。因此，如果选择0.5个停止位则不能检测帧错误和断开帧。
2. 1个停止位：对1个停止位的采样在第8，第9和第10采样点上进行。
3. 1.5个停止位(智能卡模式)：当以智能卡模式发送时，器件必须检查数据是否被正确的发送出去。所以接收器功能块必须被激活(USART_CR1寄存器中的RE =1)，并且在停止位的发送期间采样数据线上的信号。如果出现校验错误，智能卡会在发送方采样NACK信号时，即总线上停止位对应的时间内时，拉低数据线，以此表示出现了帧错误。 FE在1.5个停止位结束时和RXNE一起被置起。对1.5个停止位的采样是在第16，第17和第18采样点进行的。 1.5个的停止位可以被分成2部分：一个是0.5个时钟周期，期间不做任何事情 。随后是1个时钟周期的停止位，在这段时间的中点处采样。
4. 2个停止位：对2个停止位的采样是在第一停止位的第8，第9和第10个采样点完成的。如果第一个停止位期间检测到一个帧错误，帧错误标志将被设置。第二个停止位不再检查帧错误。在第一个停止位结束时RXNE标志将被设置。

7.3.3 UART波特率

UART波特率和UART本身的过采样有关。通过设置UART寄存器中的BRR来改变UART波特率。

- 使用8倍过采样时：BRR = fpclk / (baudrate / 16)
- 使用16倍过采样时：BRR = fpclk / (baudrate / 8)

> - BRR 为UART_BRR寄存器。
> - fpclk 为MCU APB时钟。
> - baudrate 为UART目标波特率值。

7.4 示例代码

7.4.1 简单的UART + DMA发送接收代码

```
/**
 * @file     ln_uart_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-18
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/*
   UART测试说明：

    1. UART使用的是UART0，波特率115200，8位数据长度，一位停止位，无校验位，引脚如下：

                                    PA10      UART0_TX
                                    PA11      UART0_RX
                                    PA12      UART0_RTS
                                    PA13      UART0_CTS

    2. 由于UART0的中断和原本系统的LOG打印使用的中断相冲突，所以使用本测试文件的时候需要先在serial_hw.c 232行注释掉 UART0_IRQHandler函数。
       同时在本文件的最下方取消掉UART0_IRQHandler的注释。(DMA中断也可能会有类似的问题，请根据实际情况调整)

    3. 本测试有两种情况，一种是普通的UART发送接收，一种是UART+DMA,可以通过使能或者使能 "LN_UART_DMA_SEND_EN" 和 "LN_UART_DMA_RECV_EN"，来打开或者关系UART接收和发送DMA。

    4. DMA接收，使用了双缓冲BUFFER，便于提升效率

*/

#include "hal/hal_uart.h"
#include "hal/hal_dma.h"
#include "hal/hal_gpio.h"
#include "ln_test_common.h"
#include "ln_uart_test.h"
#include "hal/hal_common.h"
```

```
#define LN_UART_DMA_SEND_EN     1          //uart发送DMA使能
#define LN_UART_DMA_RECV_EN     1          //uart接收DMA使能


static volatile unsigned char tx_data[100];          //uart发送buffer
static volatile unsigned char rx_data_1[100];        //uart接收buffer1
static volatile unsigned char rx_data_2[100];        //uart接收buffer2

static volatile unsigned int tx_data_cnt = 0;        //发送计数
static volatile unsigned int rx_data_cnt = 0;        //接收计数
static unsigned int int_cnt_1 = 0;                   //中断计数器，调试使用
static unsigned int int_cnt_2 = 0;                   //中断计数器，调试使用

static unsigned int         data_num = 100;          //DMA传输次数
static unsigned char        data_sel = 0;            //双缓冲BUFFER选择
static unsigned char        data_comp= 0;            //数据接收完成标志位


void ln_uart_init(void)
{
    /* 1. 初始化UART0引脚 */
    hal_gpio_afio_select(GPIOA_BASE,GPIO_PIN_11,UART0_TX);
    hal_gpio_afio_select(GPIOA_BASE,GPIO_PIN_12,UART0_RX);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_3,UART0_RTS);
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_4,UART0_CTS);

    hal_gpio_afio_en(GPIOA_BASE,GPIO_PIN_11,HAL_ENABLE);
    hal_gpio_afio_en(GPIOA_BASE,GPIO_PIN_12,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_3,HAL_ENABLE);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_4,HAL_ENABLE);


    /* 2. 初始化UART0配置 */
    uart_init_t uart_init_struct;
    memset(&uart_init_struct,0,sizeof(uart_init_struct));

    uart_init_struct.baudrate = 115200;                  //设置波特率
    uart_init_struct.parity = UART_PARITY_NONE;          //设置校验位
    uart_init_struct.stop_bits = UART_STOP_BITS_1;       //设置停止位
    uart_init_struct.word_len = UART_WORD_LEN_8;         //设置数据长度

    hal_uart_init(UART0_BASE,&uart_init_struct);             //初始化UART寄存器
    hal_uart_rx_mode_enable(UART0_BASE,HAL_ENABLE);          //使能发送模式
    hal_uart_tx_mode_enable(UART0_BASE,HAL_ENABLE);          //使能接收模式
    hal_uart_enable(UART0_BASE,HAL_ENABLE);                  //使能UART模块

    //uart_hardware_flow_rts_enable(UART0_BASE,HAL_ENABLE);      //UART0 RTS配置
    //uart_hardware_flow_cts_enable(UART0_BASE,HAL_ENABLE);      //UART0 CTS配置

    /* 3. 初始化UART0中断 */
    NVIC_SetPriority(UART0_IRQn,     4);
    NVIC_EnableIRQ(UART0_IRQn);
    hal_uart_it_enable(UART0_BASE,USART_IT_RXNE);            //使能接收中断
}

void ln_uart_dma_init()
{
```

```
    dma_init_t_def dma_init;
    memset(&dma_init,0,sizeof(dma_init));

#if LN_UART_DMA_SEND_EN == 1

    /* 1.初始化UART发送DMA */
    dma_init.dma_data_num = 0;                      //设置DMA数据长度
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;       //设置DMA内存地址是否增长
    dma_init.dma_mem_addr = (uint32_t)tx_data;      //设置DMA内存地址
    dma_init.dma_dir = DMA_READ_FORM_MEM;           //设置DMA传输方向
    dma_init.dma_p_addr = UART0_TX_DATA_REG;        //设置DMA物理地址
    hal_dma_init(DMA_CH_6,&dma_init);               //初始化DMA

    hal_uart_dma_enable(UART0_BASE,USART_DMA_REQ_TX,HAL_ENABLE);         //设置
UART 发送 DMA使能
#endif

#if LN_UART_DMA_RECV_EN == 1
    /* 2.初始化UART接收DMA */
    memset(&dma_init,0,sizeof(dma_init));
    dma_init.dma_data_num = data_num;               //设置DMA数据长度
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;       //设置DMA内存地址是否增长
    dma_init.dma_mem_addr = (uint32_t)rx_data_1;//设置DMA内存地址
    dma_init.dma_dir = DMA_READ_FORM_P;             //设置DMA传输方向
    dma_init.dma_p_addr = UART0_RX_DATA_REG;        //设置DMA物理地址
    hal_dma_init(DMA_CH_5,&dma_init);               //初始化DMA

    hal_uart_dma_enable(UART0_BASE,USART_DMA_REQ_RX,HAL_ENABLE);    //设置UART 接收
DMA使能
    hal_dma_it_cfg(DMA_CH_5,DMA_IT_FLAG_TRAN_COMP,HAL_ENABLE);      //使能DMA传输完
成中断
    NVIC_EnableIRQ(DMA_IRQn);                                       //使能DMA中断

    hal_dma_en(DMA_CH_5,HAL_ENABLE);
#endif

    /* 3.初始化双缓冲buffer设置 */
    data_sel = 2;                                   //设置双缓冲下一个BUFFER
    data_comp = 0;                                  //设置传输完成标志位
}

void ln_uart_test()
{
    ln_uart_init();         //初始化UART
    ln_uart_dma_init();     //初始化UART DMA

    /* 给要发送的数据buff赋值 */
    for(int i = 0;i < 100; i++)
    {
        tx_data[i] = i ;
    }

    while(1)
    {

#if (LN_UART_DMA_SEND_EN == 1)
        //DMA方式发送数据
        {
```

```
            //先失能DMA，然后才能配置DMA相关参数。
            hal_dma_en(DMA_CH_6,HAL_DISABLE);                                   //失
能DMA（为了配置参数）
            hal_dma_set_mem_addr(DMA_CH_6,(uint32_t)(tx_data));                 //重
新设置DMA地址
            hal_dma_set_data_num(DMA_CH_6,data_num);                            //设
置DMA数据长度
            hal_dma_en(DMA_CH_6,HAL_ENABLE);                                    //设
置DMA使能
            while(hal_dma_get_data_num(DMA_CH_6));                              //等
待数据发送完成
            while(hal_dma_get_it_flag(DMA_CH_6,DMA_IT_FLAG_TRAN_COMP));         //等
待数据发送完成
        }
#else
        //普通方式发送数据
        {
            for(int i = 0; i < 100; i++)
            {
                while(!hal_uart_flag_get(UART0_BASE,USART_FLAG_TXE));
//等待发送完成
                hal_uart_send_data(UART0_BASE,tx_data[i]);
 //发送数据
            }
        }
#endif
        ln_delay_ms(2000);

    }

}

#if LN_UART_DMA_RECV_EN == 1

void DMA_IRQHandler()
{
    if(hal_dma_get_it_flag(DMA_CH_5,DMA_IT_FLAG_TRAN_COMP) == HAL_SET)
    {
        if(data_sel == 1)
        {
            int_cnt_1++;
            data_sel = 2;
            hal_dma_en(DMA_CH_5,HAL_DISABLE);                                   //
失能DMA（为了配置参数）
            hal_dma_set_mem_addr(DMA_CH_5,(uint32_t)(rx_data_1));               //
重新设置DMA地址
            hal_dma_set_data_num(DMA_CH_5,data_num);                            //
设置DMA数据长度
            hal_dma_en(DMA_CH_5,HAL_ENABLE);                                    //
设置DMA使能
        }
        else
        {
            int_cnt_2++;
            data_sel = 1;
            hal_dma_en(DMA_CH_5,HAL_DISABLE);                                   //
失能DMA（为了配置参数）
```

```
            hal_dma_set_mem_addr(DMA_CH_5,(uint32_t)(rx_data_2));                //
重新设置DMA地址
            hal_dma_set_data_num(DMA_CH_5,data_num);                             //
设置DMA数据长度
            hal_dma_en(DMA_CH_5,HAL_ENABLE);                                     //
设置DMA使能
        }
        data_comp = 1;
    }
}

#else

void UART0_IRQHandler()
{
   if(hal_uart_flag_get(UART0_BASE,USART_FLAG_RXNE) == 1 &&
hal_uart_it_en_status_get(UART0_BASE,USART_IT_RXNE))
   {
       rx_data_1[rx_data_cnt++] = hal_uart_recv_data(UART0_BASE);
       if(rx_data_cnt >= 99)rx_data_cnt = 0;
   }
}

#endif
```

9 AES模块

9.1 AES简介

密码学中的高级加密标准（Advanced Encryption Standard，AES），又称Rijndael加密法，是美国联邦政府采用的一种区块加密标准。

9.2 AES主要特征

- 支持ECB和CBC加密
- 支持128bit、192bit、256bit长度的秘钥
- 支持一次加密128bit长度的明文
- 支持加密完成中断
- 支持busy标志

9.3 AES概述

9.4 示例代码

注：如非必要代码中的大小端设置请不要自行更改。

```
/**
 * @file     ln_aes_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-24
 *
```

```
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */


/*
        AES使用说明：

                1. 需要注意的是，加解密操作一次只能操作16字节，如果超过16字节要重复加密的过程。

                2. 解密的时候，要重新设置秘钥和初始化向量。


                3. 如下为网上验证过的AES加密结果，可以试验对比

                    使用CBC方式加密,秘钥长度256bit
                    明文：1234567890ABCDEF
                    秘钥：FEDCBA0987654321FEDCBA0987654321
                    初始化向量：1234567890ABCDEF
                    加密之后的密文：A5 CB 75 C3 26 A5 2F 1C  51 FB E2 5F 78 08 0F FE

                    使用ECB方式加密,秘钥长度256bit
                    明文：1234567890ABCDEF
                    秘钥：FEDCBA0987654321FEDCBA0987654321
                    初始化向量：1234567890ABCDEF
                    加密之后的密文：E4 BE A6 DD A0 F4 4C 59  7B 9D 51 8C B9 38 52 80


*/




#include "hal_aes.h"
#include "ln_test_common.h"
#include "log.h"


// 输入明文
uint8_t input[33]  = "1234567890ABCDEF";
uint8_t output[33];
const uint8_t input_backup[33] = "11112222333344445555666677778888";

// 加密用到的key 和初始化向量
const uint8_t key[33] = "FEDCBA0987654321FEDCBA0987654321";
const uint8_t iv[16] = "1234567890ABCDEF";

uint8_t out[33];
uint32_t out_len = 0;

uint8_t decrpt_out[32];
```

```
void ln_aes_test()
{

    /* 1. 初始化AES模块 */
    aes_init_t_def aes_init;
    memset(&aes_init,0,sizeof(aes_init));

    aes_init.aes_cbc_mod = AES_CBC_MOD_ECB_MOD;      //设置AES为CBC加密模式
    aes_init.aes_key_len = AES_KEY_LEN_256_BIT;      //设置密码长度为256bit
    aes_init.aes_opcode  = AES_OPCODE_ENNCRYPT;      //设置当前模式为加密模式
    aes_init.aes_big_endian = AES_BIG_ENDIAN;        //默认配置（不要更改）

    hal_aes_init(AES_BASE,&aes_init);                //初始化AES


    /* 2. 设置要加密的字符串，加密秘钥，和初始化向量 */
    LOG(LOG_LVL_INFO,"set key: %s \r\n",key);
    LOG(LOG_LVL_INFO,"set iv: %s \r\n",iv);
    LOG(LOG_LVL_INFO,"set plain text: %s \r\n",input);

    hal_aes_set_key(AES_BASE,(uint8_t *)key,strlen((const char *)key));
    //设置加密秘钥
    hal_aes_set_vector(AES_BASE,(uint8_t *)iv,strlen((const char *)iv));
   //设置初始化向量
    hal_aes_set_plain_text(AES_BASE,(uint8_t *)input,strlen((const char
*)input));  //设置要加密的字符串，明文需要小于128bit,如果大于128bit，则需要循环这个流程，重
新开始。

    /* 3. 配置中断（加密完成后产生标志位） */
    hal_aes_it_cfg(AES_BASE,AES_DATA_INT_FLAG,HAL_ENABLE);

    /* 4. 启动加密 */
    hal_aes_start(AES_BASE);

    /* 5. 等待加密完成 */
    while(hal_aes_get_status_flag(AES_BASE,AES_DATA_STATUS_FLAG) == HAL_RESET);

    /* 6. 得到加密之后的密文 */
    hal_aes_get_cipher_text(AES_BASE,(uint8_t *)output,&out_len);

    LOG(LOG_LVL_INFO,"get cipher text:");
    for(int i = 0; i < 16; i++)
        LOG(LOG_LVL_INFO,"%X ",output[i]);

    hal_aes_clear_it_flag(AES_BASE,AES_DATA_INT_FLAG);

    /* 7. 开始解密密文，首先初始化AES模块，配置为解密模式 */
    memset(&aes_init,0,sizeof(aes_init));

    aes_init.aes_cbc_mod = AES_CBC_MOD_ECB_MOD;      //设置AES为CBC加密模式
    aes_init.aes_key_len = AES_KEY_LEN_256_BIT;      //设置密码长度为256bit
    aes_init.aes_opcode  = AES_OPCODE_DECRYPT;       //设置当前模式为解密模式
    aes_init.aes_big_endian = AES_BIG_ENDIAN;        //默认配置（不要更改）

    hal_aes_init(AES_BASE,&aes_init);                //初始化AES

    /* 8. 设置要解密的密文，其余的秘钥和初始化向量不变 */
```

```
    hal_aes_set_plain_text(AES_BASE,(uint8_t *)output,strlen((const char
*)out_len));
    hal_aes_set_key(AES_BASE,(uint8_t *)key,strlen((const char *)key));
    //设置加密秘钥
    hal_aes_set_vector(AES_BASE,(uint8_t *)iv,strlen((const char *)iv));
   //设置初始化向量

    /* 9. 启动解密 */
    hal_aes_start(AES_BASE);

    /* 10. 等待加密完成 */
    while(hal_aes_get_status_flag(AES_BASE,AES_DATA_STATUS_FLAG) == HAL_RESET);

    /* 11. 得到加密之后的密文 */
    hal_aes_get_cipher_text(AES_BASE,(uint8_t *)output,&out_len);

    LOG(LOG_LVL_INFO,"\r\nget decipher text: %s",output);


    while(1)
    {
        ln_delay_ms(500);
    }

}
```

10 DMA控制器（DMA）

10.1 DMA控制器简介

直接存储器存取(DMA)用来提供在外设和存储器之间或者存储器和存储器之间的高速数据传输。无须CPU干预，数据可以通过DMA快速地移动，这就节省了CPU的资源来做其他操作。

10.2 DMA控制器主要特征

- 7个独立的可配置的通道
- 每个通道都直接连接专用的硬件DMA请求。这些功能通过软件来配置。
- 多个请求间的优先权可以通过软件编程设置(共有三级：高、中等和低)，优先权设置相等时由硬件决定(请求0优先于请求1，依此类推) 。
- 存储器和存储器间的传输
- 外设和存储器、存储器和外设之间的传输
- 可编程的数据传输数目：最大为65535
- 每个通道都有3个事件标志(DMA半传输、 DMA传输完成和DMA传输出错)以及一个通道事件标志

10.3 DMA控制器概述

10.3.1 DMA处理

在发生一个事件后，外设向DMA控制器发送一个请求信号。 DMA控制器根据通道的优先权处理请求。当DMA控制器开始访问发出请求的外设时，DMA控制器立即发送给它一个应答信号。当从DMA控制器得到应答信号时，外设立即释放它的请求。一旦外设释放了这个请求， DMA控制器同时撤销应答信号。如果有更多的请求时，外设可以启动下一个周期。

总之，每次DMA传送由3个操作组成：

- 从外设数据寄存器或者从当前外设/存储器地址寄存器指示的存储器地址取数据，第一次传输时的开始地址是DMA_CPA或DMA_CMA寄存器指定的外设基地址或存储器单元。
- 存数据到外设数据寄存器或者当前外设/存储器地址寄存器指示的存储器地址，第一次传输时的开始地址是DMA_CPA或DMA_CMA寄存器指定的外设基地址或存储器单元。
- 执行一次DMA_NDT寄存器的递减操作，该寄存器包含未完成的操作数目。

设置DMA参数时，需要关闭DMA使能，否则参数不能设置成功。

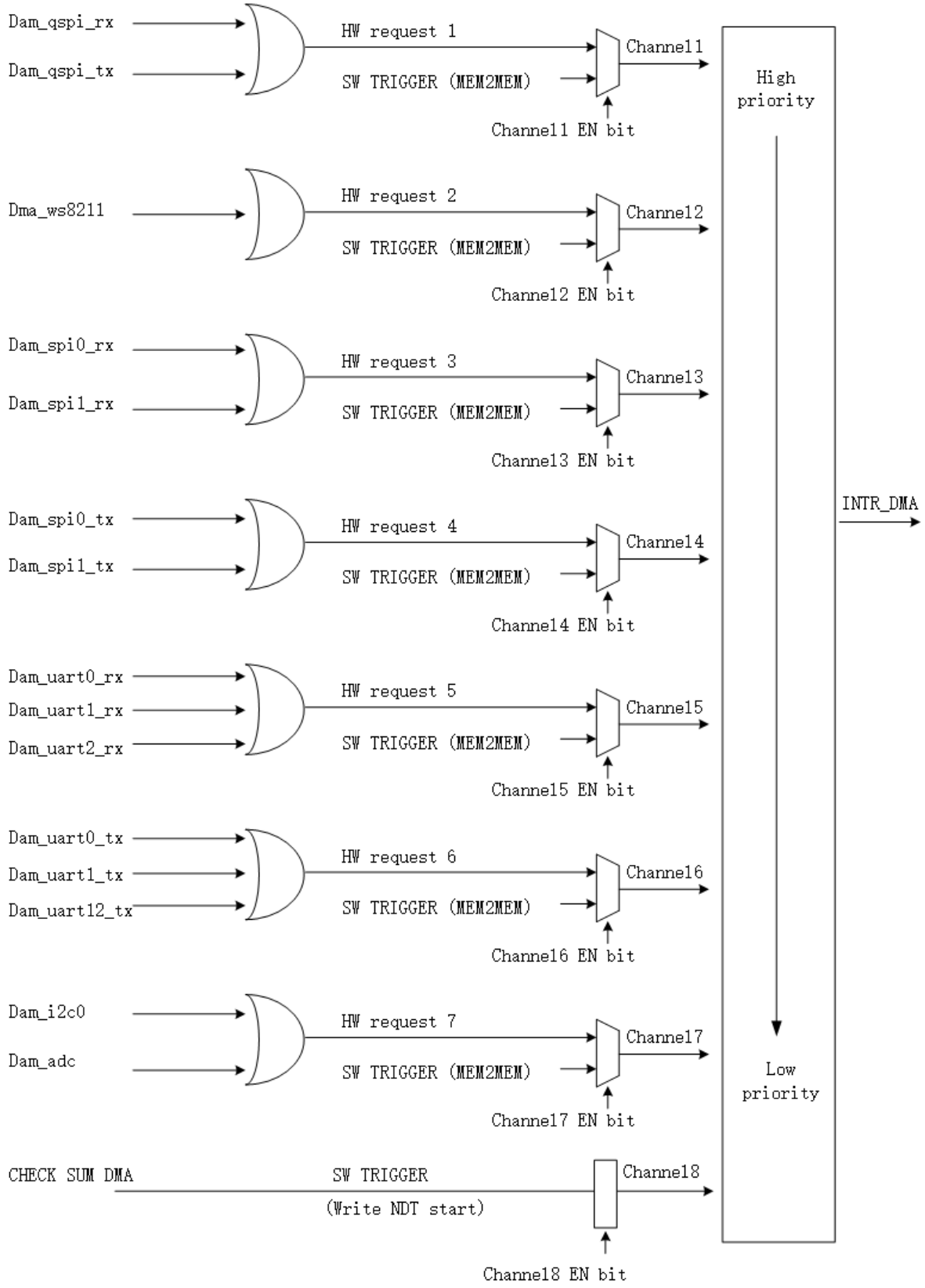
10.3.2 中断

每个DMA通道都可以在DMA传输过半、传输完成和传输错误时产生中断。为应用的灵活性考虑，通过设置寄存器的不同位来打开这些中断。

中断事件	事件标志位	使能控制位
传输过半	HTIF	HTIE
传输完成	TCIF	TCIE
传输错误	TEIF	TEIE

10.3.3 DMA请求映像

同一时间，同一DMA通道，只能有一个外设使用，DMA通道对应外设如下图。



10.4 示例代码

DMA的示例代码只有内存到内存的传输，外设与内存之间的DMA传输请参考对应外设的代码。

```
/**
 * @file     ln_dma_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-26
```

```
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/**
    DMA使用说明：
                1. DMA的测试函数只有内存拷贝模式，需要通过DMA操作外设的，请参考相关外设代码。

                2. DMA的IFCR是W1C不能使用位域的方式操作。

*/



#include "hal/hal_uart.h"
#include "hal/hal_dma.h"
#include "hal/hal_gpio.h"
#include "ln_dma_test.h"

unsigned char src_data[100];
unsigned char dst_data[100];

void ln_dma_test(void)
{

    //1. 内存拷贝模式
    dma_init_t_def dma_init;
    memset(&dma_init,0,sizeof(dma_init));

    dma_init.dma_mem_to_mem_en = DMA_MEM_TO_MEM_EN;      //使能DMA内存拷贝模式

    dma_init.dma_data_num = 100;                         //设置DMA内存拷贝的数量
    dma_init.dma_dir = DMA_READ_FORM_MEM;                //设置DMA方向
    dma_init.dma_mem_addr = (uint32_t)src_data;          //设置DMA源地址
    dma_init.dma_mem_size = DMA_MEM_SIZE_8_BIT;          //设置源地址数据长度
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;            //设置源地址是否增长

    dma_init.dma_p_addr = (uint32_t)dst_data;            //设置DMA目的地址
    dma_init.dma_p_inc_en = DMA_P_INC_EN;                //设置DMA目的地址是否增长
    dma_init.dma_p_size = DMA_P_SIZE_8_BIT;              //设置DMA目的地址数据长度

    dma_init.dma_pri_lev = DMA_PRI_LEV_MEDIUM;           //设置传输优先级

    hal_dma_init(DMA_CH_7,&dma_init);                    //初始化DMA

    hal_dma_it_cfg(DMA_CH_7,DMA_IT_FLAG_TRAN_COMP,HAL_ENABLE);       //配置DMA传输完成中断
    NVIC_EnableIRQ(DMA_IRQn);                                        //使能DMA中断

    for(int i = 0; i < 100; i++)
    {
        src_data[i] = i;
    }

    hal_dma_en(DMA_CH_7,HAL_ENABLE);                     //使能DMA传输
```

```
    while(hal_dma_get_data_num(DMA_CH_7) != 0 ||
hal_dma_get_status_flag(DMA_CH_7,DMA_STATUS_FLAG_TRAN_COMP));  //等待DMA传输完成

    while(1);

}


void DMA_IRQHandler()
{
    if(hal_dma_get_it_flag(DMA_CH_7,DMA_IT_FLAG_TRAN_COMP) == HAL_SET)
    {
        hal_dma_clear_it_flag(DMA_CH_7,DMA_IT_FLAG_TRAN_COMP);
    }
}
```

11 EXT中断

11.1 EXT中断简介

外部中断可以用来唤醒深度睡眠下的MCU

11.2 EXT中断主要特征

在LN8620中，只有如下图中的引脚才支持外部中断引脚。

		0 GPIO	1 SWI	2 ANA_MOD	3 DBG	4 SPIF	5 SDIO	6 I2S	7 FUNC	8 debug	9 wakeup	10 MISC	PU/P	TEST0	TEST1	DSEL
0	PAD_00	GPIO0[0]		ADC0					FULLMUX[0]	debug_port[0]	EXT INT[0]		PU	bist_rstn	scanrstn	dsel[0]
1	PAD_01	GPIO0[1]	SWD	ADC1					FULLMUX[1]	debug_port[1]	EXT INT[1]		PU			dsel[1]
2	PAD_02	GPIO0[2]							FULLMUX[2]		EXT INT[2]		PU			
3	PAD_03	GPIO0[3]							FULLMUX[3]		EXT INT[3]		PU			
5	PAD_05	GPIO0[5]							FULLMUX[5]		EXT INT[4]		PU			
6	PAD_06	GPIO0[6]			DBG_TX		sdio_io2	i2s_sdi0	FULLMUX[6]	debug_port[3]	EXT INT[5]		PD		scanzip	
7	PAD_07	GPIO0[7]			DBG_RX		sdio_io3		FULLMUX[7]	debug_port[4]	EXT INT[6]		PD	bist_clk	testclk	
25	PAD_25	GPIO1[9]		ANAOUT_IN					FULLMUX[19]	debug_port[15]	EXT INT[7]		PU	bist_fail_ble	scanout4	dsel[9]

11.3 EXT中断概述

暂无

11.4 示例代码

```
/**
 * @file     ln_ext_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-30
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co.
Ltd
```

```
 *
 */

#include "hal_common.h"
#include "hal_ext.h"
#include "hal_gpio.h"
#include "ln_test_common.h"
#include "ln_ext_test.h"

void ln_ext_test()
{
    hal_ext_init(EXT_INT_SENSE_2,EXT_INT_POSEDEG,HAL_ENABLE);

    NVIC_SetPriority(EXT_IRQn,1);                //配置中断优先级
    NVIC_EnableIRQ(EXT_IRQn);

    while(1)
    {
        ln_delay_ms(500);
    }
}



void EXT_IRQHandler()
{
    if(hal_ext_get_it_flag(EXT_INT_SENSE_2_IT_FLAG) == HAL_SET)
    {
        hal_ext_clear_it_flag(EXT_INT_SENSE_2_IT_FLAG);
    }
}
```

12 FLASH

12.1 FLASH简介

12.2 FLASH主要特征

12.3 FLASH概述

注意，如果要在Flash代码中操作Flash，那么用户需要把`hal_flash.c` `hal_cahce.c` `hal_qspi.c`这三个文件放到RAM里面运行，否则会出错。

另外写入Flash之前，必须先擦除Flash。

12.4 示例代码

```
/**
 * @file     ln_flash_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
```

```
 * @date     2021-08-30
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

#include "ln_test_common.h"
#include "ln_flash_test.h"
#include "hal_flash.h"


static uint32_t flash_id = 0;
static uint32_t flash_status = 0;

static uint8_t test_data[4096];
static uint8_t read_data[4096];

static uint32_t err_cnt = 0;


void ln_flash_test()
{
    /* 1. 读取flash ID */
    flash_id = FLASH_ReadID();

    /* 2. 擦除一部分Flash，要4K对齐的擦除 */
    FLASH_Erase(0x00100000,0x1000);

    /* 3. 生成测试用数据 */
    for(int i = 0; i < 4096; i++)
        test_data[i] = i * 7;

    /* 4. 生成测试用数据*/
    FLASH_Program(0x00100000,0x1000,test_data);

    /* 5. 读取并比对数据,根据 err_cnt 计算出是否写入出错 */
    FLASH_Read(0x00100000,0x1000,read_data);

    for(int i = 0; i < 4096; i++)
    {
        if(read_data[i] != test_data[i])
            err_cnt ++;
    }

    memset(read_data,0,4096);

    FLASH_ReadByCache(0x00100000,0x1000,read_data);

    for(int i = 0; i < 4096; i++)
    {
        if(read_data[i] != test_data[i])
            err_cnt ++;
    }


    while(1)
    {
```

```
        //do nothing.
        ln_delay_ms(500);
    }
}
```

13 GPIO和EXT中断

13.1 GPIO简介

GPIO（英语：General-purpose input/output），通用型之输入输出的简称，功能类似8051的P0—P3，其接脚可以供使用者由程控自由使用，PIN脚依现实考量可作为通用输入（GPI）或通用输出（GPO）或通用输入与输出（GPIO）

13.2 GPIO主要特征

根据数据手册中列出的每个I/O端口的特定硬件特征， GPIO端口的每个位可以由软件分别配置成多种模式。

- 输入上拉
- 输入下拉
- 模拟输入
- 上拉输出
- 下拉输出

每个GPIO引脚复位后都会配置为上拉或者下拉，详细请参考LN8620引脚相关的文档。

13.3 GPIO概述

需要注意的是GPIO翻转最大速率40M。

13.4 示例代码

请参考外设测试工程中的ln_gpio_test.c

14 I2C接口

14.1 I2C介绍

I2C总线是由Philips公司开发的一种简单、双向二线制同步串行总线。它只需要两根线即可在连接于总线上的器件之间传送信息。

14.2 I2C主要特征

- 产生和检测7位/10位地址
- 状态标志：
  – 发送器/接收器模式标志
  – 字节发送结束标志
  – I2C总线忙标志
- 错误标志
  – 主模式时的仲裁丢失

- 地址/数据传输后的应答(ACK)错误
- 检测到错位的起始或停止条件
- 禁止拉长时钟功能时的上溢或下溢

- 支持不同的通讯速度
  - 标准速度(高达100 kHz)
  - 快速(高达400 kHz)

14.3 I2C功能概述

14.3.1 I2C初始化

1. 首先配置IIC的引脚，通过引脚功能复用I2C SCL和I2C SDA。
2. 设置 **IIC_CR2** 中的 **freq** ，设置外设时钟，范围：0 - 63，单位Mhz,推荐使用默认值4。
3. 设置 **IIC_CR1** 中的 **pe** 置零，先失能IIC总线，为了设置IIC的时钟。
4. 设置 **IIC_CCR** 中的 **fs** 设置IIC为1，设置IIC为快速模式（0为标准模式）。
5. 设置 **IIC_CCR** 中的 **duty** ，这个寄存器决定着快速模式下，T_high和T_low的比例关系。0: T_low/T_high = 2； 1: T_low/T_high = 16/9（使用说明请看 **IIC_CCR** 中的 **ccr** 说明）。
6. 设置 **IIC_CCR** 中的 **ccr** ，设置总线频率。T = T_high + T_low。

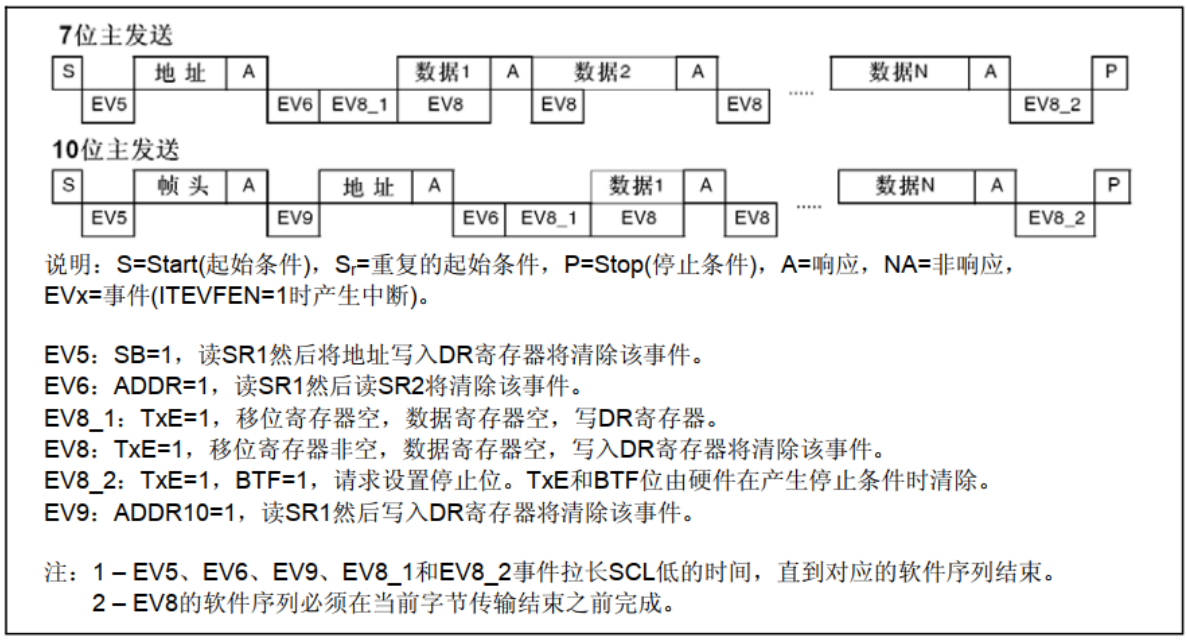
> - 标准模式下:
>   T_high = CCR * Tpclk,
>   T_low = 2 * CCR * Tpclk ,
>   T = CCR * Tpclk + 2 * CCR * Tpclk
> - 快速模式下:
>   - duty = 0时:
>   T_high = CCR * Tpclk
>   T_low = 2 × CCR * Tpclk
>   - duty = 1时:
>   T_high = 9 * CCR * Tpclk
>   T_low = 16 * CCR * Tpclk

> 时钟配置举例:
>
> 快速模式下：400k SCL周期为2.5us，假设外设工作时钟为40MHZ 所以Tpckl = 1/40M = 0.025us
>
> - 如果duty = 0;
>   fs = 1;
>   Thigh = CCR × TPCLK1
>   Tlow = 2 × CCR × TPCLK1
>   Thigh + Tlow = 2.5us
>   所以 3 * CCR × TPCLK1 = 2.5us CCR = 33
> - 如果duty = 1;
>   fs = 1;
>   Thigh = 9 * CCR × TPCLK1
>   Tlow = 16 × CCR × TPCLK1
>   Thigh + Tlow = 2.5us
>   所以 3 * CCR × TPCLK1 = 2.5us CCR = 4

```
duty    ->    I2C_CCR->duty
fs      ->    I2C_CCR->fs
Thigh   ->    I2C_CLK高电平时间
Tlow    ->    I2C_CLK低电平时间
TPCLK1  ->    外设时钟
CCR     ->    I2C_CCR->ccr
```

7. 设置 **IIC_TRISE** 中的 **trise** ，这个寄存器决定了IIC的SCL和SDA的电平上升时间。上升时间 = trise * Tpclk，推荐设置为0xF。
8. 设置 **IIC_CCR1** 中的 **smbus** 寄存器为0，设成IIC模式（1为SMBus模式）。
9. 设置 **IIC_CCR1** 中的 **pe** 为1，使能IIC外设。
10. 开始配置 **I2C_CR1** 中的 **swrst** ,复位IIC总线。先置位1，然后再置为0.

14.3.2 I2C写数据



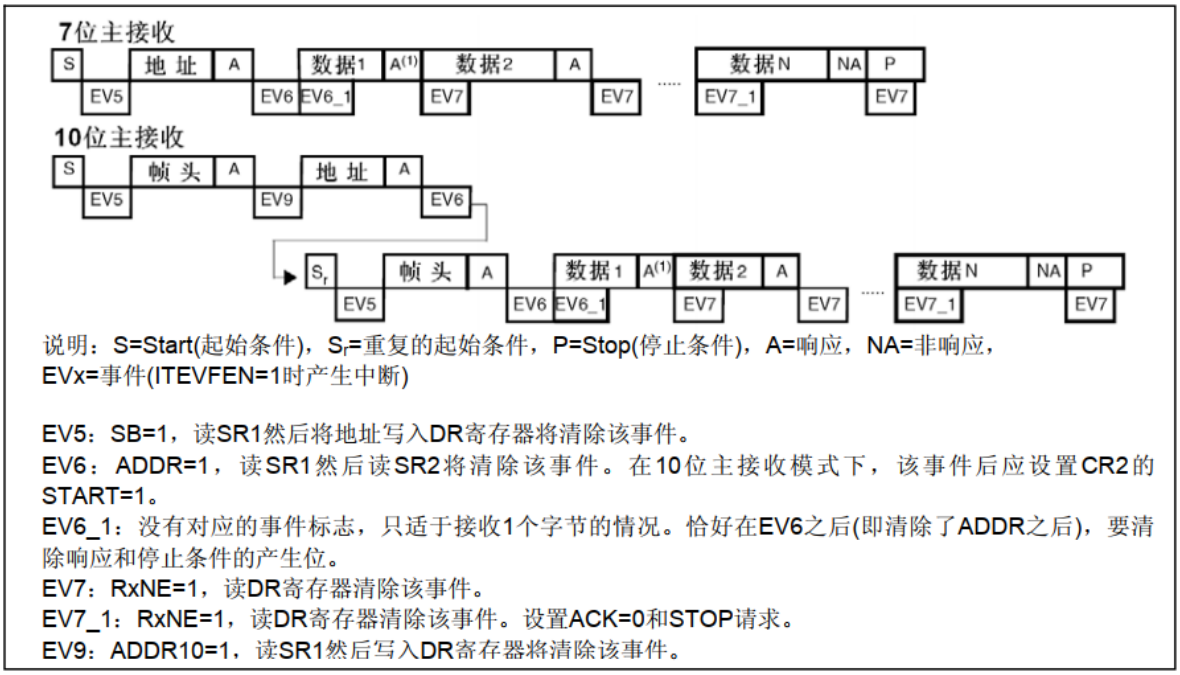
说明：S=Start(起始条件)，S_r=重复的起始条件，P=Stop(停止条件)，A=响应，NA=非响应，EVx=事件(ITEVFEN=1时产生中断)。

EV5：SB=1，读SR1然后将地址写入DR寄存器将清除该事件。
EV6：ADDR=1，读SR1然后读SR2将清除该事件。
EV8_1：TxE=1，移位寄存器空，数据寄存器空，写DR寄存器。
EV8：TxE=1，移位寄存器非空，数据寄存器空，写入DR寄存器将清除该事件。
EV8_2：TxE=1，BTF=1，请求设置停止位。TxE和BTF位由硬件在产生停止条件时清除。
EV9：ADDR10=1，读SR1然后写入DR寄存器将清除该事件。

注：1 – EV5、EV6、EV9、EV8_1和EV8_2事件拉长SCL低的时间，直到对应的软件序列结束。
2 – EV8的软件序列必须在当前字节传输结束之前完成。

1. 首先检查IIC总线，是否为busy状态，如果是的话则复位总线。
2. 然后首先发送start起始条件，并等待起始条件发送完成。
3. 发送设备地址，并等待地址发送完成。
4. 清除设备地址发送完成标志位。（如果ADDR = 1 ,通过读取SR1，然后读SR2清除该标志位）
5. 等待DR寄存器为空，然后开始写入数据。（此过程可以连续操作）
6. 等待数据发送完成。
7. 发送停止位，整个写入流程结束。

14.3.3 I2C读数据



说明：S=Start(起始条件)，S_r=重复的起始条件，P=Stop(停止条件)，A=响应，NA=非响应，
EVx=事件(ITEVFEN=1时产生中断)

EV5：SB=1，读SR1然后将地址写入DR寄存器将清除该事件。
EV6：ADDR=1，读SR1然后读SR2将清除该事件。在10位主接收模式下，该事件后应设置CR2的START=1。
EV6_1：没有对应的事件标志，只适于接收1个字节的情况。恰好在EV6之后(即清除了ADDR之后)，要清除响应和停止条件的产生位。
EV7：RxNE=1，读DR寄存器清除该事件。
EV7_1：RxNE=1，读DR寄存器清除该事件。设置ACK=0和STOP请求。
EV9：ADDR10=1，读SR1然后写入DR寄存器将清除该事件。

1. 首先检查IIC总线，是否为busy状态，如果是的话则复位总线。
2. 然后首先发送start起始条件，并等待起始条件发送完成。
3. 发送设备地址，并等待地址发送完成。（读设备数据的时候，设备地址要+1）
4. 清除设备地址发送完成标志位。（如果ADDR = 1 ,通过读取SR1，然后读SR2清除该标志位）
5. 当只接受一字节数据的时候，失能ack，等待rxne置为1，然后开始接收数据，当接收大于一个数据的时候，使能ack应答，等待rxne置为1，重复操作，直到接收最后一个数据时，失能ack，等待rxne置为1，然后开始接收最后一个数据。
6. 发送停止位，整个读取流程结束。

14.4 示例代码

14.4.1 I2C读写示例

```
/**
 * @file     ln_i2c_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-20
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/**
    I2C使用说明：
                1. 本测试分为五部分。第一部分是扫描设备地址；第二部分是读写AT24C04；第三部分是写OLED屏；第四部分是读取SHT30温湿度传感器；第五部分是读取MPU9250数据；

                2. AT24C04、OLED、SHT30、MPU9250是通过I2C挂载在一起的,I2C的SCL和SDA要接上拉电阻。接线如下：

                                                                    ln8620
   OLED                 AT24C04              SHT30                MPU9250
```

```
                                                                PB5         ->
  D0          ->      SCL         ->      SCL         ->      SCL
                                                                PB6         ->
  D1          ->      SDA         ->      SDA         ->      SDA
                                                                3.3v        ->
  3.3V        ->      3.3V        ->      3.3V        ->      3.3V
                                                                GND         ->
  GND         ->      GND         ->      GND         ->      GND
                                                                PB7         ->
  RES         ->
                                                                GND         ->
  CS          ->      WP
                                                                GND         ->
  GND         ->      A0
                                                                GND         ->
  GND         ->      A1
                                                                GND         ->
  GND         ->      A2
```

3. I2C波特率设置：　　快速模式下：400k SCL周期为2.5us，假设外设工作时钟为40MHZ 所以Tpckl = 1/40M = 0.025us

```
                                        如果duty = 0;
                                        fs = 1;
                                                Thigh = CCR × TPCLK1
                                                Tlow = 2 × CCR × TPCLK1
                                                Thigh + Tlow = 2.5us
                                                所以 3 * CCR × TPCLK1 = 2.5us
CCR = 33

                                        如果duty = 1;
                                        fs = 1;
                                                Thigh = 9 * CCR × TPCLK1
                                                Tlow = 16 × CCR × TPCLK1
                                                Thigh + Tlow = 2.5us
                                                所以 3 * CCR × TPCLK1 = 2.5us
CCR = 4

                                        duty    ->      I2C_CCR->duty
                                        fs      ->      I2C_CCR->fs
                                        Thigh   ->      I2C_CLK高电平时间
                                        Tlow    ->      I2C_CLK低电平时间
                                        TPCLK1  ->      外设时钟
*/

#include "ln_i2c_test.h"
#include "hal/hal_i2c.h"
#include "hal/hal_gpio.h"
#include "reg_i2c.h"
#include "log.h"
#include "ln_i2c_test.h"
#include "ln_oled_test.h"
#include "ln_mpu9250_test.h"
#include "ln_sht30_test.h"

#define TIMEOUT_CYCLE 2000
```

```
/************************************I2C INIT
START*******************************************************/
void ln_i2c_init()
{
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_5,I2C0_SCL);       //初始化I2C引脚
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_6,I2C0_SDA);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_5,HAL_ENABLE);         //使能功能复用
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_6,HAL_ENABLE);

    /* disable the i2c */
    hal_i2c_en(I2C_BASE,HAL_DISABLE);                           //配置参数前要先失能
I2C

    /* init the i2c */
    i2c_init_t_def i2c_init;
&amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;memset(&amp;i2c_init,0,sizeof(i2c_init));

    i2c_init.i2c_peripheral_clock_freq = 4;                      //时钟设置，默认4就
可以（滤波使用）
    i2c_init.i2c_master_mode_sel = I2C_FM_MODE;                  //fs    = 1
    i2c_init.i2c_fm_mode_duty_cycle = I2C_FM_MODE_DUTY_CYCLE_2; //duty = 1
    /*
        快速模式下：400k SCL周期为2.5us
        外设工作时钟为40MHZ 所以Tpckl = 1/40M = 0.025us

        duty = 1;fs = 1
        Thigh = 9  x CCR x TPCLK1
        Tlow  = 16 x CCR x TPCLK1
        Thigh + Tlow = 2.5us
        ccr =  4
    */
    i2c_init.i2c_ccr = 4;
    i2c_init.i2c_trise = 0xF;           //推荐默认值为0xF;

    hal_i2c_init(I2C_BASE,&i2c_init);

    hal_i2c_en(I2C_BASE,HAL_ENABLE);
}

//master tx 7bit_mode
int hal_i2c_master_7bit_write(uint32_t base, uint8_t dev_addr, const uint8_t *
buf, uint16_t buf_len)
{
    volatile uint32_t timeout = 0;

    //1. check busy
    while (1 == i2c_busy_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
i2c_swrst_setf(base, 1); i2c_swrst_setf(base, 0); return -1; } }

    //2. send start
    i2c_start_setf(base, 1);

    //3. wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }

    //4. send addr
```

```
    i2c_dr_set(base, dev_addr);

    //5. wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    //6. clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    for (uint16_t i = 0; i < buf_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //7. send data
        i2c_dr_set(base, buf[i]);
    }

    //8. wait send complete.
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    //9. stop the i2c.
    i2c_stop_setf(base, 1);
    return 0;
}
int hal_i2c_master_7bit_write_target_address(uint32_t base, uint8_t dev_addr,
uint8_t *target_addr,uint8_t target_addr_len,const uint8_t * buf, uint16_t
buf_len)
{
    volatile uint32_t timeout = 0;

    //check busy
    while (1 == i2c_busy_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
i2c_swrst_setf(base, 1); i2c_swrst_setf(base, 0); return -1; } }

    //send start
    i2c_start_setf(base, 1);

    //wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }

    //send addr
    i2c_dr_set(base, dev_addr);

    //wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
```

```
    for (uint16_t i = 0; i < target_addr_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //send addr
        i2c_dr_set(base, target_addr[i]);
    }
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    for (uint16_t i = 0; i < buf_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //send addr
        i2c_dr_set(base, buf[i]);
    }
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }


    i2c_stop_setf(base, 1);
    return 0;
}
//master rx 7bit_mode with target address
int hal_i2c_master_7bit_read_target_address(uint32_t base, uint8_t
dev_addr,uint8_t *target_addr,uint8_t target_addr_len,uint8_t * buf, uint16_t
buf_len)
{
    i2c_swrst_setf(base, 1);
    i2c_swrst_setf(base, 0);
    //hal_i2c_reset(base);
    // return ret;
    volatile uint32_t timeout = 0;

    //check busy
    while (1 == i2c_busy_getf(base)) {
        if (timeout++ > TIMEOUT_CYCLE) {
            i2c_swrst_setf(base, 1);
            i2c_swrst_setf(base, 0);
            return -1;
        }
    }
    //send start & ack
    i2c_start_setf(base, 1);
    i2c_ack_setf(base, 1);

    //wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }
```

```
    //send addr
    i2c_dr_set(base, dev_addr);

    //wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    for (uint16_t i = 0; i < target_addr_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //send addr
        i2c_dr_set(base, *target_addr);
        target_addr++;
    }
    //wait for the transfer finish
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);

    //delay some times ,wait the write
    timeout = 5000;
    while(timeout--);
    //reset the bus,if not do this,the bus can not run start operation
    i2c_swrst_setf(base, 1);
    i2c_swrst_setf(base, 0);

    //send start & ack
    i2c_start_setf(base, 1);

    //wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }

    //send addr,reading opeartion ,address need +1
    i2c_dr_set(base, dev_addr + 1);

    //wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) {if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);

    //clear the DR
    i2c_dr_get(base);

    for(int i = buf_len;i > 0; i --)
```

```
    {
        //when reading the last byte,do not send the ack
        if (buf_len == 1)
        {
            //do not send the ack
            i2c_ack_setf(base, 0);

            //wait tx empty flag
            timeout = 0;
            while (0 == i2c_rxne_getf(base)) {if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }
            //read data
            *buf = i2c_dr_get(base);
        }
        else
        {
            //send ack
            i2c_ack_setf(base, 1);
            //wait tx empty flag
            timeout = 0;
            while (0 == i2c_rxne_getf(base)) {if (timeout++ > TIMEOUT_CYCLE)
{return -1; } }
            //read data
            *buf = i2c_dr_get(base);
        }
        buf_len--;
        buf++;
    }

    //send stop
    i2c_stop_setf(base, 1);

    return 0;
}

/*************************************I2C INIT
END*******************************************************/
```

14.4.2 I2C扫描设备地址

```
//第一部分，I2C扫描设备地址
/*************************************I2C SCAN
START*******************************************************/


static unsigned int address[10];
static unsigned int address_num = 0;

void hal_i2c_address_scan(uint32_t base)
{
    volatile uint32_t timeout = 0;
    unsigned int dev_addr;
```

```
    i2c_ack_setf(base,00);
    //scan the iic address
    for(dev_addr = 0;dev_addr <= 254;dev_addr = dev_addr + 2 )
    {
        timeout = 0;
        //check busy
        while (1 == i2c_busy_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
i2c_swrst_setf(base, 1); i2c_swrst_setf(base, 0); break; } }
        if (timeout++ > TIMEOUT_CYCLE)  continue;

        //send start
        i2c_start_setf(base, 1);

        //wait start send ok
        timeout = 0;
        while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { break;
} }
        if (timeout++ > TIMEOUT_CYCLE)  continue;
        //send addr
        i2c_dr_set(base, dev_addr);

        //wait addr send ok
        timeout = 0;
        while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
break; } }
        if (timeout++ > TIMEOUT_CYCLE)  continue;

        //can not get the ack
        if(i2c_af_getf(base) == 1)
        {
            i2c_af_setf(base,0);
            i2c_swrst_setf(base, 1);
            i2c_swrst_setf(base, 0);
            continue;
        }
        else
        {
            //clear addr flag
            i2c_sr1_get(base);
            i2c_sr2_get(base);
            i2c_stop_setf(base, 1);

            //add the  address
            address[address_num] = dev_addr;
            address_num++;

            //reset the bus
            i2c_swrst_setf(base, 1);
            ln_delay_ms(200);
            i2c_swrst_setf(base, 0);
        }


    }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    i2c_stop_setf(base, 1);
```

```
}

/*************************************I2C SCAN
END*******************************************************/
```

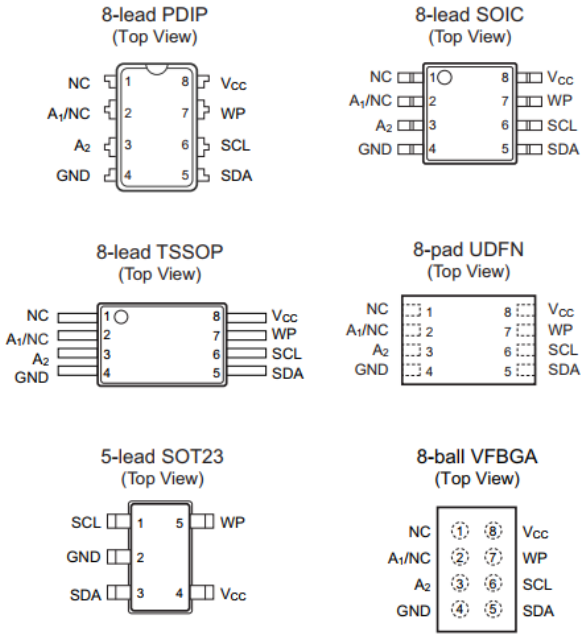
14.4.3 读写24C04示例

AT24C04接线说明：

1. Pin Configurations and Pinouts

Pin Name	Function
NC	No Connect
A_1	Address Input (4K Only)
A_2	Address Input
SDA	Serial Data
SCL	Serial Clock Input
WP	Write Protect
GND	Ground
V_{CC}	Power Supply

Notes: 1. For use of 5-lead SOT23, the software A2 and A1 bits in the device address word must be set to zero to properly communicate.
2. Drawings are note to scale.




```
//第二部分，读写24C04
/*************************************I2C R/W AT24C04
START*******************************************************/
#define AT24C04_ADDR_W 0xA0
#define AT24C04_WRITE_DELAY 50000
//word_addr : 字节地址；buf，读出的数据存放地址；buf_len,要读出数据的长度
int hal_24c04_read(uint16_t word_addr, uint8_t * buf, uint16_t buf_len)
{
   int ret = 0;
   uint8_t i = 0;
   //因为24C04一共有 32页 * 16 字节  数据，当想要读取大于 255地址的数据时，就必须通过增加设备地址来翻页，
   //地址0XA0的范围  0 - 255，地址0xA2的范围 256 - 512.
   if(word_addr + buf_len >= 256)
   {
       if(word_addr >= 256)
       {
            i = word_addr - 256;
            if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE,
AT24C04_ADDR_W | 0x02,&i, 1,buf , buf_len)))
```

```
            {
                return ret;
            }
        }
        else
        {
            i = word_addr;
            if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE,
AT24C04_ADDR_W,&i, 1,buf,256 - word_addr)))
            {
                return ret;
            }
            i = 0;
            if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE,
AT24C04_ADDR_W | 0x02,&i,1, buf + 256 - word_addr,buf_len - (256 - word_addr))))
            {
                return ret;
            }
        }


    }
    else
    {
        i = word_addr;
        if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE,
AT24C04_ADDR_W,&i, 1,buf,buf_len)))
        {
            return ret;
        }
    }
     return ret;
}

static int hal_24c04_write_byte(uint16_t word_addr, uint8_t data)
{
    int ret = 0;
    uint8_t buf[2];
    buf[0] = word_addr & 0Xff;
    buf[1] = data;
    //因为24C04一共有 32页 * 16 字节  数据，当想要写入大于 255地址的数据时，就必须通过增加设
备地址来翻页，
    //地址0XA0的范围  0 - 255，地址0xA2的范围 256 - 512.
    uint8_t drv_addr = AT24C04_ADDR_W;
    if(word_addr > 255)drv_addr = (AT24C04_ADDR_W | 0x02);
    if (0 != (ret = hal_i2c_master_7bit_write_target_address(I2C_BASE,
drv_addr,buf, 1 ,buf + 1, 1))) {
        return ret;
    }
    for (volatile uint32_t i = 0; i < AT24C04_WRITE_DELAY; i++);        //max
5ms

    return ret;
}
////word_addr : 字节地址；buf，要写入数据的地址；buf_len,要写入数据的长度
int hal_24c04_write(uint16_t word_addr, const uint8_t * buf, uint16_t buf_len)
{
    int ret = 0;
```

```
    uint16_t i = 0;
    for (i = 0; i < buf_len; i++) {
        if (0 != (ret = hal_24c04_write_byte(word_addr + i, buf[i]))) {
            return ret;
        }
    }
    return ret;
}

/*************************************I2C R/W AT24C04
END*********************************************************/
```

14.4.4 写OLED示例

由于OLED包含了字符库和独立的驱动文件，所以详细内容请参考I2C的测试代码。

接线说明：

	OLED引脚	芯片引脚
1	GND	GND
2	VCC	3-5V
3	D0	IIC SCL
4	D1	IIC SDA
5	RES	高电平，上电之后必须复位一次
6	DC	地址选择引脚，不接
7	CS	SPI片选，接地

因为对于OLED只有写操作，且写操作分为写命令和写数据，这两者通过写入的第一个数据区分

代码如下：

```
//第三部分，写OLED屏
/*************************************I2C W OLED
START*********************************************************/
unsigned char cmd_arr[2];
unsigned char data_arr[2];
#define OLED_ADDR 0x78
void hal_write_command_byte(uint32_t base, uint8_t cmd)
{
    cmd_arr[0] = 0x00;
    cmd_arr[1] =  cmd;
    hal_i2c_master_7bit_write(base, OLED_ADDR, cmd_arr, 2);
};
void hal_write_data_byte(uint32_t base, uint8_t data)
{
```

```
    data_arr[0] = 0x40;
    data_arr[1] = data;
    hal_i2c_master_7bit_write(base, OLED_ADDR, data_arr, 2);
};

/*************************************I2C W OLED
END*********************************************************/
```

14.4.5 读取SHT30示例

```
//第四部分，读取SHT30温湿度传感器
/*************************************I2C SHT30
INIT********************************************************/
unsigned char crc8_checksum(unsigned char *ptr,unsigned char len)
{
    unsigned char bit;        // bit mask
    unsigned char crc = 0xFF; // calculated checksum
    unsigned char byteCtr;    // byte counter

    // calculates 8-Bit checksum with given polynomial
    for(byteCtr = 0; byteCtr < len; byteCtr++)
    {
        crc ^= (ptr[byteCtr]);
        for(bit = 8; bit > 0; --bit)
        {
            if(crc & 0x80) crc = (crc << 1) ^ 0x131;
            else           crc = (crc << 1);
        }
    }

    return crc;
}

unsigned char ln_tem_hum_init()
{
    unsigned char data[2];
    unsigned char res = 0;
    data[0] = ((CMD_MEAS_PERI_2_M >> 8) & 0x00FF);
    data[1] = ((CMD_MEAS_PERI_2_M >> 0) & 0x00FF);
    res = hal_i2c_master_7bit_write(I2C_BASE, TEM_HUM_ADDR, data, 2);
    return res;
}

unsigned char tem_hum_read_sensor_data(unsigned char *data)
{
    unsigned char addr_data[10];
    addr_data[0] = ((CMD_FETCH_DATA >> 8) & 0x00FF);
    addr_data[1] = ((CMD_FETCH_DATA >> 0) & 0x00FF);

 hal_i2c_master_7bit_read_target_address(I2C_BASE,TEM_HUM_ADDR,addr_data,2,data,
6);

    if(data[2] != crc8_checksum(data,2))
```

```
    {
        return 1;
    }

    if(data[5] != crc8_checksum(data + 3,2))
    {
        return 1;
    }

    return 0;
}

/*************************************I2C SHT30
END*********************************************************/
```

14.4.6 读取MPU9250示例代码

这部分代码请参考I2C测试代码。

14.4.7 I2C测试代码

因为I2C测试代码包含了很多传感器器，所以有一部分传感器单独创建了传感器文件，然后调用了本测试代码的接口，所以如果需要全部的测试代码，请参考SDK中的外设测试部分。

```
/**
 * @file     ln_i2c_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-20
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co.
Ltd
 *
 */

/**
    I2C使用说明：
                1. 本测试分为五部分。第一部分是扫描设备地址；第二部分是读写AT24C04；第三部分
是写OLED屏；第四部分是读取SHT30温湿度传感器；第五部分是读取MPU9250数据；

                2. AT24C04、OLED、SHT30、MPU9250是通过I2C挂载在一起的,I2C的SCL和SDA要
接上拉电阻。接线如下：

                                                                    ln8620
   OLED                AT24C04             SHT30               MPU9250
```

```
                                                                PB5          ->
  D0          ->      SCL         ->      SCL         ->      SCL
                                                                PB6          ->
  D1          ->      SDA         ->      SDA         ->      SDA
                                                                3.3v         ->
  3.3V        ->      3.3V        ->      3.3V        ->      3.3V
                                                                GND          ->
  GND         ->      GND         ->      GND         ->      GND
                                                                PB7          ->
  RES         ->
                                                                GND          ->
  CS          ->      WP
                                                                GND          ->
  GND         ->      A0
                                                                GND          ->
  GND         ->      A1
                                                                GND          ->
  GND         ->      A2
```

3. I2C波特率设置：　　快速模式下：400k SCL周期为2.5us，假设外设工作时钟为40MHZ 所以Tpckl = 1/40M = 0.025us

如果duty = 0;
fs = 1;
Thigh = CCR × TPCLK1
Tlow = 2 × CCR × TPCLK1
Thigh + Tlow = 2.5us
所以 3 * CCR × TPCLK1 = 2.5us

CCR = 33

如果duty = 1;
fs = 1;
Thigh = 9 * CCR × TPCLK1
Tlow = 16 × CCR × TPCLK1
Thigh + Tlow = 2.5us
所以 3 * CCR × TPCLK1 = 2.5us

CCR = 4

duty -> I2C_CCR->duty
fs -> I2C_CCR->fs
Thigh -> I2C_CLK高电平时间
Tlow -> I2C_CLK低电平时间
TPCLK1 -> 外设时钟

```
*/

#include "ln_i2c_test.h"
#include "hal/hal_i2c.h"
#include "hal/hal_gpio.h"
#include "reg_i2c.h"
#include "log.h"
#include "ln_i2c_test.h"
#include "ln_oled_test.h"
#include "ln_mpu9250_test.h"
#include "ln_sht30_test.h"
#include "ln_test_common.h"

#define TIMEOUT_CYCLE 2000
```

```
/************************************I2C INIT
START*********************************************************/
void ln_i2c_init()
{
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_5,I2C0_SCL);       //初始化I2C引脚
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_6,I2C0_SDA);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_5,HAL_ENABLE);         //使能功能复用
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_6,HAL_ENABLE);

    /* disable the i2c */
    hal_i2c_en(I2C_BASE,HAL_DISABLE);                           //配置参数前要先失能
I2C

    /* init the i2c */
    i2c_init_t_def i2c_init;
    memset(&i2c_init,0,sizeof(i2c_init));

    i2c_init.i2c_peripheral_clock_freq = 4;                      //时钟设置，默认4就
可以（滤波使用）
    i2c_init.i2c_master_mode_sel = I2C_FM_MODE;                  //fs    = 1
    i2c_init.i2c_fm_mode_duty_cycle = I2C_FM_MODE_DUTY_CYCLE_2; //duty = 1
    /*
        快速模式下：400k SCL周期为2.5us
        外设工作时钟为40MHZ 所以Tpckl = 1/40M = 0.025us

        duty = 1;fs = 1
        Thigh = 9  x CCR x TPCLK1
        Tlow  = 16 x CCR x TPCLK1
        Thigh + Tlow = 2.5us
        ccr =  4
    */
    i2c_init.i2c_ccr = 4;
    i2c_init.i2c_trise = 0xF;           //推荐默认值为0xF;

    hal_i2c_init(I2C_BASE,&i2c_init);

    hal_i2c_en(I2C_BASE,HAL_ENABLE);
}

//master tx 7bit_mode
int hal_i2c_master_7bit_write(uint32_t base, uint8_t dev_addr, const uint8_t *
buf, uint16_t buf_len)
{
    volatile uint32_t timeout = 0;

    //1. check busy
    while (1 == i2c_busy_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
i2c_swrst_setf(base, 1); i2c_swrst_setf(base, 0); return -1; } }

    //2. send start
    i2c_start_setf(base, 1);

    //3. wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }
```

```
    //4. send addr
    i2c_dr_set(base, dev_addr);

    //5. wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    //6. clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    for (uint16_t i = 0; i < buf_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //7. send data
        i2c_dr_set(base, buf[i]);
    }

    //8. wait send complete.
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    //9. stop the i2c.
    i2c_stop_setf(base, 1);
    return 0;
}
int hal_i2c_master_7bit_write_target_address(uint32_t base, uint8_t dev_addr,
uint8_t *target_addr,uint8_t target_addr_len,const uint8_t * buf, uint16_t
buf_len)
{
    volatile uint32_t timeout = 0;

    //check busy
    while (1 == i2c_busy_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
i2c_swrst_setf(base, 1); i2c_swrst_setf(base, 0); return -1; } }

    //send start
    i2c_start_setf(base, 1);

    //wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }

    //send addr
    i2c_dr_set(base, dev_addr);

    //wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    //clear addr flag
```

```
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    for (uint16_t i = 0; i < target_addr_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //send addr
        i2c_dr_set(base, target_addr[i]);
    }
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }

    for (uint16_t i = 0; i < buf_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //send addr
        i2c_dr_set(base, buf[i]);
    }
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }


    i2c_stop_setf(base, 1);
    return 0;
}
//master rx 7bit_mode with target address
int hal_i2c_master_7bit_read_target_address(uint32_t base, uint8_t
dev_addr,uint8_t *target_addr,uint8_t target_addr_len,uint8_t * buf, uint16_t
buf_len)
{
    i2c_swrst_setf(base, 1);
    i2c_swrst_setf(base, 0);
    //hal_i2c_reset(base);
    // return ret;
    volatile uint32_t timeout = 0;

    //check busy
    while (1 == i2c_busy_getf(base)) {
        if (timeout++ > TIMEOUT_CYCLE) {
            i2c_swrst_setf(base, 1);
            i2c_swrst_setf(base, 0);
            return -1;
        }
    }
    //send start & ack
    i2c_start_setf(base, 1);
    i2c_ack_setf(base, 1);

    //wait start send ok
    timeout = 0;
```

```
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }

    //send addr
    i2c_dr_set(base, dev_addr);

    //wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    for (uint16_t i = 0; i < target_addr_len; i++)
    {
        //wait tx empty flag
        timeout = 0;
        while (0 == i2c_txe_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }

        //send addr
        i2c_dr_set(base, *target_addr);
        target_addr++;
    }
    //wait for the transfer finish
    while (0 == i2c_btf_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);

    //delay some times ,wait the write
    timeout = 5000;
    while(timeout--);
    //reset the bus,if not do this,the bus can not run start operation
    i2c_swrst_setf(base, 1);
    i2c_swrst_setf(base, 0);

    //send start & ack
    i2c_start_setf(base, 1);

    //wait start send ok
    timeout = 0;
    while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { return -1;
} }

    //send addr,reading opeartion ,address need +1
    i2c_dr_set(base, dev_addr + 1);

    //wait addr send ok
    timeout = 0;
    while (0 == i2c_addr_getf(base)) {if (timeout++ > TIMEOUT_CYCLE) { return
-1; } }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);

    //clear the DR
```

```
    i2c_dr_get(base);

    for(int i = buf_len;i > 0; i --)
    {
        //when reading the last byte,do not send the ack
        if (buf_len == 1)
        {
            //do not send the ack
            i2c_ack_setf(base, 0);

            //wait tx empty flag
            timeout = 0;
            while (0 == i2c_rxne_getf(base)) {if (timeout++ > TIMEOUT_CYCLE) {
return -1; } }
            //read data
            *buf = i2c_dr_get(base);
        }
        else
        {
            //send ack
            i2c_ack_setf(base, 1);
            //wait tx empty flag
            timeout = 0;
            while (0 == i2c_rxne_getf(base)) {if (timeout++ > TIMEOUT_CYCLE)
{return -1; } }
            //read data
            *buf = i2c_dr_get(base);
        }
        buf_len--;
        buf++;
    }

    //send stop
    i2c_stop_setf(base, 1);

    return 0;
}

/*************************************I2C INIT
END**********************************************************/



//第一部分，I2C扫描设备地址
/*************************************I2C SCAN
START*********************************************************/


static unsigned int address[10];
static unsigned int address_num = 0;

void hal_i2c_address_scan(uint32_t base)
{
    volatile uint32_t timeout = 0;
    unsigned int dev_addr;
    i2c_ack_setf(base,00);
    //scan the iic address
```

```
    for(dev_addr = 0;dev_addr <= 254;dev_addr = dev_addr + 2 )
    {
        timeout = 0;
        //check busy
        while (1 == i2c_busy_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
i2c_swrst_setf(base, 1); i2c_swrst_setf(base, 0); break; } }
        if (timeout++ > TIMEOUT_CYCLE)  continue;

        //send start
        i2c_start_setf(base, 1);

        //wait start send ok
        timeout = 0;
        while (0 == i2c_sb_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) { break;
} }
        if (timeout++ > TIMEOUT_CYCLE)  continue;
        //send addr
        i2c_dr_set(base, dev_addr);

        //wait addr send ok
        timeout = 0;
        while (0 == i2c_addr_getf(base)) { if (timeout++ > TIMEOUT_CYCLE) {
break; } }
        if (timeout++ > TIMEOUT_CYCLE)  continue;

        //can not get the ack
        if(i2c_af_getf(base) == 1)
        {
            i2c_af_setf(base,0);
            i2c_swrst_setf(base, 1);
            i2c_swrst_setf(base, 0);
            continue;
        }
        else
        {
            //clear addr flag
            i2c_sr1_get(base);
            i2c_sr2_get(base);
            i2c_stop_setf(base, 1);

            //add the  address
            address[address_num] = dev_addr;
            address_num++;

            //reset the bus
            i2c_swrst_setf(base, 1);
            ln_delay_ms(200);
            i2c_swrst_setf(base, 0);
        }


    }
    //clear addr flag
    i2c_sr1_get(base);
    i2c_sr2_get(base);
    i2c_stop_setf(base, 1);
}
```

```
/**************************************I2C SCAN
END*******************************************************/


//第二部分，读写24C04
/**************************************I2C R/W AT24C04
START*****************************************************/
#define AT24C04_ADDR_W 0xA0
#define AT24C04_WRITE_DELAY 50000
//word_addr : 字节地址；buf，读出的数据存放地址；buf_len,要读出数据的长度
int hal_24c04_read(uint16_t word_addr, uint8_t * buf, uint16_t buf_len)
{
   int ret = 0;
   uint8_t i = 0;
   //因为24C04一共有 32页 * 16 字节  数据，当想要读取大于 255地址的数据时，就必须通过增加设备地址来翻页，
   //地址0XA0的范围  0 - 255，地址0xA2的范围 256 - 512.
   if(word_addr + buf_len >= 256)
   {
       if(word_addr >= 256)
       {
            i = word_addr - 256;
            if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE, AT24C04_ADDR_W | 0x02,&i, 1,buf , buf_len)))
            {
                return ret;
            }
       }
       else
       {
            i = word_addr;
            if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE, AT24C04_ADDR_W,&i, 1,buf,256 - word_addr)))
            {
                return ret;
            }
            i = 0;
            if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE, AT24C04_ADDR_W | 0x02,&i,1, buf + 256 - word_addr,buf_len - (256 - word_addr))))
            {
                return ret;
            }
       }


   }
   else
   {
       i = word_addr;
       if (0 != (ret = hal_i2c_master_7bit_read_target_address(I2C_BASE, AT24C04_ADDR_W,&i, 1,buf,buf_len)))
       {
           return ret;
       }
   }
    return ret;
}
```

```c
static int hal_24c04_write_byte(uint16_t word_addr, uint8_t data)
{
    int ret = 0;
    uint8_t buf[2];
    buf[0] = word_addr & 0xff;
    buf[1] = data;
    //因为24C04一共有 32页 * 16 字节  数据，当想要写入大于 255地址的数据时，就必须通过增加设备地址来翻页，
    //地址0xA0的范围  0 - 255，地址0xA2的范围 256 - 512.
    uint8_t drv_addr = AT24C04_ADDR_W;
    if(word_addr > 255)drv_addr = (AT24C04_ADDR_W | 0x02);
    if (0 != (ret = hal_i2c_master_7bit_write_target_address(I2C_BASE, drv_addr,buf, 1 ,buf + 1, 1))) {
        return ret;
    }
    for (volatile uint32_t i = 0; i < AT24C04_WRITE_DELAY; i++);        //max 5ms

    return ret;
}
////word_addr : 字节地址；buf，要写入数据的地址；buf_len,要写入数据的长度
int hal_24c04_write(uint16_t word_addr, const uint8_t * buf, uint16_t buf_len)
{
    int ret = 0;
    uint16_t i = 0;
    for (i = 0; i < buf_len; i++) {
        if (0 != (ret = hal_24c04_write_byte(word_addr + i, buf[i]))) {
            return ret;
        }
    }
    return ret;
}

/*************************************I2C R/W AT24C04 END*************************************************/



//第三部分，写OLED屏
/*************************************I2C W OLED START*************************************************/
unsigned char cmd_arr[2];
unsigned char data_arr[2];
#define OLED_ADDR 0x78
void hal_write_command_byte(uint32_t base, uint8_t cmd)
{
    cmd_arr[0] = 0x00;
    cmd_arr[1] =  cmd;
    hal_i2c_master_7bit_write(base, OLED_ADDR, cmd_arr, 2);
};
void hal_write_data_byte(uint32_t base, uint8_t data)
{
    data_arr[0] = 0x40;
    data_arr[1] = data;
    hal_i2c_master_7bit_write(base, OLED_ADDR, data_arr, 2);
};
```

```
/*************************************I2C W OLED
END*********************************************************/



//第四部分，读取SHT30温湿度传感器
/*************************************I2C SHT30
INIT********************************************************/
unsigned char crc8_checksum(unsigned char *ptr,unsigned char len)
{
    unsigned char bit;        // bit mask
  unsigned char crc = 0xFF; // calculated checksum
  unsigned char byteCtr;    // byte counter

  // calculates 8-Bit checksum with given polynomial
  for(byteCtr = 0; byteCtr < len; byteCtr++)
  {
    crc ^= (ptr[byteCtr]);
    for(bit = 8; bit > 0; --bit)
    {
      if(crc & 0x80) crc = (crc << 1) ^ 0x131;
      else           crc = (crc << 1);
    }
  }
    
    return crc;
}

unsigned char ln_tem_hum_init()
{
    unsigned char data[2];
    unsigned char res = 0;
    data[0] = ((CMD_MEAS_PERI_2_M >> 8) & 0x00FF);
    data[1] = ((CMD_MEAS_PERI_2_M >> 0) & 0x00FF);
    res = hal_i2c_master_7bit_write(I2C_BASE, TEM_HUM_ADDR, data, 2);
    return res;
}

unsigned char tem_hum_read_sensor_data(unsigned char *data)
{
    unsigned char addr_data[10];
    addr_data[0] = ((CMD_FETCH_DATA >> 8) & 0x00FF);
    addr_data[1] = ((CMD_FETCH_DATA >> 0) & 0x00FF);

 hal_i2c_master_7bit_read_target_address(I2C_BASE,TEM_HUM_ADDR,addr_data,2,data,
6);

    if(data[2] != crc8_checksum(data,2))
    {
        return 1;
    }

    if(data[5] != crc8_checksum(data + 3,2))
    {
        return 1;
    }

    return 0;
```

```
}

/*************************************I2C SHT30
END*******************************************************/


//第五部分，读写九轴陀螺仪
//该部分代码在ln_mpu_9250中



/*****************************************************************************
**********************/

static unsigned char tx_data[100];              //测试BUFFER
static unsigned char rx_data[100];
static unsigned char sensor_data[6];
static unsigned int  err_cnt = 0;

static volatile double        tem_data = 0;     //温度数据
static volatile double        hum_data = 0;     //湿度数据

static volatile double        mpu_tem_data = 0;

static short         mpu_gx_data = 0;           //角速度信息
static short         mpu_gy_data = 0;
static short         mpu_gz_data = 0;

static short         mpu_ax_data = 0;           //加速度信息
static short         mpu_ay_data = 0;
static short         mpu_az_data = 0;

static short         mpu_mx_data = 0;           //磁力计信息
static short         mpu_my_data = 0;
static short         mpu_mz_data = 0;

void ln_i2c_test()
{
    // I2C初始化
    ln_i2c_init();

    /* 1.开始扫描地址 */
    LOG(LOG_LVL_INFO,"ln8620 i2c scan start! \n");

    hal_i2c_address_scan(I2C_BASE);

    if(address_num == 0)
    {
        LOG(LOG_LVL_INFO,"There is no device on I2C bus! \n");
    }
    else
    {
        for(int i = address_num; i > 0; i--)
        {
            LOG(LOG_LVL_INFO,"Found device %d,address : 0x%x! \n",address_num -
i,address[address_num - i]);
        }
        LOG(LOG_LVL_INFO,"Scan end. \n");
```

```
    }

    for(int i = 0; i < 100; i++)
    {
        tx_data[i] = i;
    }

    /* 2.开始读写AT24C04 */
    LOG(LOG_LVL_INFO,"ln8620 i2c + 24c04 test start! \n");

    hal_24c04_write(256,tx_data,100);
    hal_24c04_read(256,rx_data,100);

    for(int i = 0; i < 100; i++)
    {
        if(tx_data[i] != rx_data[i])
        {
            err_cnt++;
        }
    }
    if(err_cnt != 0)
        LOG(LOG_LVL_INFO,"ln8620 i2c + 24c04 test fail! \n");
    else
        LOG(LOG_LVL_INFO,"ln8620 i2c + 24c04 test success! \n");

    /* oled test */
    LOG(LOG_LVL_INFO,"ln8620 i2c + oled test start! \n");

    /* 3. 开始控制OLED显示屏 */
    gpio_init_t gpio_init;
    memset(&gpio_init,0,sizeof(gpio_init));   
;     //清零结构体
    gpio_init.dir =
GPIO_OUTPUT;           &n
bsp;    //配置GPIO方向，输入或者输出
    gpio_init.pin =
GPIO_PIN_7;           &nb
sp;        //配置GPIO引脚号
    gpio_init.speed =
GPIO_HIGH_SPEED;          &nbs
p; //设置GPIO速度
    hal_gpio_init(GPIOB_BASE,&gpio_init);        //初始化GPIO
    hal_gpio_pin_reset(GPIOB_BASE,GPIO_PIN_7);  //RESET OLED屏幕
    ln_delay_ms(100);
    hal_gpio_pin_set(GPIOB_BASE,GPIO_PIN_7);

    ln_oled_init(I2C_BASE);
    ln_oled_display_on();

    ln_oled_show_string(0,40,(unsigned char *)"Test success!",12);      //显示测试
成功的英文
    LOG(LOG_LVL_INFO,"ln8620 i2c + oled test success! \n");

    ln_oled_refresh();                                                  //刷新显示
    ln_delay_ms(1000);
    ln_oled_clear();

    /* 4.初始化温湿度传感器 */
```

```
    LOG(LOG_LVL_INFO,"ln8620 i2c + sht30 test start! \n");
    ln_tem_hum_init();


    /* 5.初始化九轴传感器 */
    LOG(LOG_LVL_INFO,"ln8620 i2c + MPU9250 test start! \n");
    ln_mpu_init();

    while(1)
    {
        unsigned char display_buffer[16];

        if(tem_hum_read_sensor_data(sensor_data) == 0)                    //获得温
湿度信息
        {

            memset(display_buffer,' ',16);
            tem_data = -45 + (175 * (sensor_data[0] * 256 + sensor_data[1]) *
1.0 / (65535 - 1)) ;       //转换为摄氏度
            hum_data = 100 * ((sensor_data[3] * 256 + sensor_data[4]) * 1.0 /
(65535 - 1)) ;             //转换为%

            sprintf((char *)display_buffer,"TEM : %0.2f",tem_data);       //oled
显示和串口打印数据
            ln_oled_show_string_with_len(0,32,display_buffer,16,15);
            memset(display_buffer,' ',16);
            sprintf((char *)display_buffer,"HUM : %0.2f",hum_data);
            ln_oled_show_string_with_len(0,48,display_buffer,16,15);
        }


        mpu_tem_data = ln_mpu_get_temperature();
        ln_mpu_get_gyroscope(&mpu_gx_data,&mpu_gy_data,&mpu_gz_data);      //获取
角速度信息
        ln_mpu_get_accelerometer(&mpu_ax_data,&mpu_ay_data,&mpu_az_data);  //或者
加速度信息
        ln_mpu_get_magnetometer(&mpu_mx_data,&mpu_my_data,&mpu_mz_data);   //获取
磁力计信息


        //显示和打印LOG。
        memset(display_buffer,' ',16);
        sprintf((char *)display_buffer,"x:%0.1f",mpu_ax_data * 1.0 * 0.005625);
        ln_oled_show_string_with_len(0,0,display_buffer,16,8);

        memset(display_buffer,' ',16);
        sprintf((char *)display_buffer,"y:%0.1f",mpu_ay_data * 1.0 * 0.005625);
        ln_oled_show_string_with_len(64,0,display_buffer,16,7);

        memset(display_buffer,' ',16);
        sprintf((char *)display_buffer,"z:%0.1f",mpu_az_data * 1.0 * 0.005625);
        ln_oled_show_string_with_len(0,16,display_buffer,16,8);

        LOG(LOG_LVL_INFO,"Tem = %0.2f  ",tem_data);
        LOG(LOG_LVL_INFO,"Hum = %0.2f%\n",hum_data);
        LOG(LOG_LVL_INFO,"ACC:  X: %0.3d    Y: %0.3d    Z: %0.3d
\n",mpu_ax_data,mpu_ay_data,mpu_az_data);
```

```
        LOG(LOG_LVL_INFO,"GYR:  X: %0.3d    Y: %0.3d    Z: %0.3d
\n",mpu_gx_data,mpu_gy_data,mpu_gz_data);
        LOG(LOG_LVL_INFO,"MAG:  X: %0.3d    Y: %0.3d    Z: %0.3d
\n\n",mpu_mx_data,mpu_my_data,mpu_mz_data);


        ln_oled_refresh();                                          //刷
新OLED显示

        ln_delay_ms(30);
    }
}
```

15 I2S接口

15.1 I2S简介

I2S(Inter—IC Sound)总线, 又称集成电路内置音频总线，是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准。

15.2 I2S主要特征

- 只支持从机模式
- 全双工模式双声道立体声
- 可配置的数据位宽，支持12、16、20、24和32位宽的数据
- 支持飞利浦I2S标准

15.3 I2S概述

1. I2S不支持DMA
2. I2S只有一路
3. 因为I2S只支持从机模式，所以I2S时钟由主机提供
4. 测试I2S的时候一定要注意先开主机再开从机，否则初始化从机的时候，主机可能会误收数据。
5. I2S作为从机接收的时候，可能会丢掉帧头帧尾的数据。

15.4 示例代码

```
/**
 * @file     ln_i2s_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-25
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co.
Ltd
 *
 */
```

```
/*
    I2S调试说明：

        1. 引脚
                    LN8620          STM32
                    A8(WS)          PB12
                    A9(CLK)         PB13
                    A10(SDO)        PB14(SD)
                    A6(SDI)

            注：STM32的I2S的DR是接收发送一体的，所以只有一个SD寄存器,同时8620的I2S引脚是固定的，通过CMP里面的I2S_O_EN0来选择是否打开

        2. 从机接收数据的时候可能会缺少帧头和帧尾的数据，发送数据的时候不会有这个问题。

        3. I2S不支持DMA。

        4. 本测试中，使用中断来进行I2S的接收和发送。



*/

#include "hal_i2s.h"
#include "ln_i2s_test.h"
#include "reg_sysc_cmp.h"
#include "log.h"
#include "ln_test_common.h"

static unsigned short tx_data_left[256];
static unsigned short tx_data_right[256];
static unsigned short rx_data_left[256];
static unsigned short rx_data_right[256];
static unsigned int data_index = 0;
static unsigned int tx_data_index = 0;
static unsigned int rx_data_index = 0;


void ln_i2s_test(void)
{
    //使能I2S IO
    sysc_cmp_i2s_io_en0_setf(1);

    for(int i = 0;i < 256; i++)
    {
        tx_data_left[i] = i+1 ;
        tx_data_right[i] = i+10;
    }

    __NVIC_SetPriorityGrouping(4);
    __enable_irq();
    NVIC_SetPriority(I2S_IRQn,     4);
    NVIC_EnableIRQ(I2S_IRQn);

    i2s_init_t_def i2s_init;
    memset(&i2s_init,0,sizeof(i2s_init));
```

```
    i2s_init.tra_fifo_trig_lev = 4;                         //为了让FIFO中同时有左声道和右声道数据，此处设为4（只有一个声道时可能会不发数据 ）。
    i2s_init.rec_fifo_trig_lev = 0;                         //设置接收FIFO深度门限（就是能触发门限中断的值）
    i2s_init.i2s_tra_config = I2S_TRA_CONFIG_16_BIT;    //设置发送数据宽度
    i2s_init.i2s_rec_config = I2S_REC_CONFIG_16_BIT;    //设置接收数据宽度
    hal_i2s_init(I2S_BASE,&i2s_init);                       //初始化I2S

    //tx
    hal_i2s_tx_fifo_flush(I2S_BASE);                            //清除发送FIFO
    hal_i2s_tx_en(I2S_BASE,HAL_ENABLE);                         //使能I2S发送
    hal_i2s_it_cfg(I2S_BASE,I2S_IT_FLAG_TXFE,HAL_ENABLE);   //使能发送中断（fifo为空产生中断）
    hal_i2s_en(I2S_BASE,HAL_ENABLE);

    //rx
    hal_i2s_rx_fifo_flush(I2S_BASE);                            //清除接收FIFO
    hal_i2s_rx_en(I2S_BASE,HAL_ENABLE);                         //使能I2S接收
    hal_i2s_it_cfg(I2S_BASE,I2S_IT_FLAG_RXDA,HAL_ENABLE);   //使能接收中断（fifo达到门限值产生中断）
    hal_i2s_en(I2S_BASE,HAL_ENABLE);                            //I2S模块使能

    while(1)
    {
        LOG(LOG_LVL_INFO,"running...\r\n");
        ln_delay_ms(1000);
    }

}

void I2S_IRQHandler(void)
{
    //tx
    if(hal_i2s_get_it_flag(I2S_BASE,I2S_IT_FLAG_TXFE) == 1)
    {

 hal_i2s_send_data(I2S_BASE,tx_data_left[tx_data_index],tx_data_right[tx_data_index]);

        tx_data_index++;
        if(tx_data_index >= 255)
            hal_i2s_en(I2S_BASE,HAL_DISABLE);               //发送一定数量的数据之后，关闭I2S，否则可能会卡死（因为一直处于中断中）
    }

    //rx
    if(hal_i2s_get_it_flag(I2S_BASE,I2S_IT_FLAG_RXDA) == 1)
    {
        //接收数据，接收左声道和右声道数据
        hal_i2s_rece_data(I2S_BASE,rx_data_left + rx_data_index,rx_data_right + rx_data_index);

        rx_data_index++;
        if(rx_data_index >= 250)
            rx_data_index = 0;
        data_index++;
    }
```

```
}
```

16 SDIO接口（SDIO）

16.1 SDIO简介

SDIO是一个通过可编程预分频器驱动的24位自动装载计数器构成。 SDIO调试时，CMD、D0、D1、D2、D3必须接上拉电阻，对于LN8XXX芯片，我们必须选择高速模式，否则会出现数据异常的问题。

16.2 SDIO主要特征

- 最大速度40Mhz，推荐速度25Mhz(实测最大到22Mhz,注意线材连接)
- 只能做从机使用

16.3 SDIO概述

该部分内容请参考SDIO协议。

16.4 示例代码

```
/**
 * @file     ln_sdio_test.c
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-30
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */

/*
    SDIO使用说明：

                1. 接线说明：

                        LN8620(从)        STM32F4阿波罗(主)
                    CMD -> PA8       ->       PD2
                    CLK -> PA9       ->       PC12
                    SD0 -> PA10      ->       PC8
                    SD1 -> PA11      ->       PC9
                    SD2 -> PA6       ->       PC10
                    SD3 -> PA7       ->       PC11

                2. 跳线测试速度推荐4Mhz，主机配置从机一些寄存器参数的时候，属于低速通信，当配置完成后，才会切到高速通信。

                3. SDIO 的 CMD、SD0、SD1、SD2、SD3需要接上拉电阻。

                4. SDIO数据CRC校验需要手动打开CRC的校验中断，打开CRC校验之后需要手动在数据块的末尾加上CRC校验，例如数据块大小256，那就在第255和第256处放两字节CRC.
```

```
                5.



*/


#include "hal_sdio_device.h"
#include "hal_misc.h"
#include "hal_gpio.h"
#include "ln_test_common.h"
#include "ln_sdio_test.h"

static uint8_t sdio_cis_fn0[128];
static uint8_t sdio_cis_fn1[128];

static uint8_t tx_data[512];
static uint8_t rx_data[512];

void ln_sdio_init()
{
    hal_gpio_pin_pull_set(GPIOA_BASE,GPIO_PIN_6,GPIO_PULL_UP);              //配
置引脚上拉
    hal_gpio_pin_pull_set(GPIOA_BASE,GPIO_PIN_7,GPIO_PULL_UP);              //配
置引脚上拉
    hal_gpio_pin_pull_set(GPIOA_BASE,GPIO_PIN_8,GPIO_PULL_UP);              //配
置引脚上拉
    hal_gpio_pin_pull_set(GPIOA_BASE,GPIO_PIN_9,GPIO_PULL_UP);              //配
置引脚上拉

    hal_gpio_pin_pull_set(GPIOA_BASE,GPIO_PIN_10,GPIO_PULL_UP);             //配
置引脚上拉
    hal_gpio_pin_pull_set(GPIOA_BASE,GPIO_PIN_11,GPIO_PULL_UP);             //配
置引脚上拉

    hal_misc_cmp_set_sdio_io_en(1);                                         //使
能SDIO引脚

    sdio_init_t sdio_init;
    memset(&sdio_init,0,sizeof(sdio_init));

    sdio_init.recv_buff = rx_data;                                          //设
置接收数据的BUFFER
    sdio_init.cis_fn0_base = sdio_cis_fn0;
    sdio_init.cis_fn1_base = sdio_cis_fn1;



    sdio_init.sdio_dev_csa_support = FN1_CSA_SUPPORT | FN2_CSA_SUPPORT | \
                                     FN3_CSA_SUPPORT | FN4_CSA_SUPPORT | \
                                     FN5_CSA_SUPPORT | FN6_CSA_SUPPORT | \
                                     FN7_CSA_SUPPORT;

    sdio_init.sdio_dev_scsi = SDIO_CCCR_CAP_SCSI_1;
    sdio_init.sdio_dev_smb  = SDIO_CCCR_CAP_SMB_1;
    sdio_init.sdio_dev_sdc  = SDIO_CCCR_CAP_SDC_1;

    sdio_init.sdio_dev_srw  = SDIO_CCCR_CAP_SRW_1;
```

```
    sdio_init.sdio_dev_sbs  = SDIO_CCCR_CAP_SBS_1;
    sdio_init.sdio_dev_4bls = SDIO_CCCR_CAP_4BLS_1;

    sdio_init.sdio_dev_func_num = SDIO_FUNC1;

    hal_sdio_device_init(&sdio_init);

    hal_sdio_device_it_cfg(FN1_ENABLE_INTERRPT     | RESET_FN1_INTERRPT       |
 \
                           FN1_WRITE_OVER_INTERRPT | FN1_READ_OVER_INTERRPT   |
 \
                           READ_ERROR_FN1_INTERRPT | WRITE_ERROR_FN1_INTERRPT ,
HAL_ENABLE); //     //crc check

    NVIC_EnableIRQ(SDIO_1_IRQn);

    NVIC_SetPriority(SDIO_1_IRQn,    1);
}

void sdio_boot_sdio_device_send_data(void)
{
    hal_sdio_device_set_send_buf_addr(tx_data);
    hal_sdio_device_set_send_buf_size(256);
    hal_sdio_device_trig_host_data1_int();
}




void ln_sdio_test()
{
    ln_sdio_init();

    for (int i = 0; i < 256; i++)
    {
        tx_data[i] = i;
    }

    while(1)
    {
        ln_delay_ms(1000);
    }
}




void SDIO_F1_IRQHandler(void)
{
    if(hal_sdio_device_it_get_flag(FN1_WRITE_OVER_INTERRPT) == HAL_SET)
    {
        unsigned int len = hal_sdio_device_get_recv_buf_size();
        unsigned char *src = (uint8_t *)hal_sdio_device_get_recv_buf_addr();

        if(len == 256)
        {
            // set the next receive buffer address.
```

```
            hal_sdio_device_set_recv_buf_addr(rx_data);

            // send data
            tx_data[0] ++;
            sdio_boot_sdio_device_send_data();
        }


        //clear the receive interrupt flag and clear the sdio device busy flag.
        hal_sdio_device_it_clr_flag(FN1_WRITE_OVER_INTERRPT);
        hal_sdio_device_clear_busy();
    }

    if(hal_sdio_device_it_get_flag(FN1_READ_OVER_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(FN1_READ_OVER_INTERRPT);
    }


    if(hal_sdio_device_it_get_flag(READ_ERROR_FN1_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(READ_ERROR_FN1_INTERRPT);
    }

    //CRC ERROR
    if(hal_sdio_device_it_get_flag(WRITE_ERROR_FN1_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(WRITE_ERROR_FN1_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(WRITE_ABORT_FN1_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(WRITE_ABORT_FN1_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(RESET_FN1_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(RESET_FN1_INTERRPT);
    }

    //when host enable the FN1 interrupt,will receive the interrupt.
    if(hal_sdio_device_it_get_flag(FN1_ENABLE_INTERRPT) == HAL_SET)
    {
       hal_sdio_device_it_clr_flag(FN1_ENABLE_INTERRPT);
    }

    if(hal_sdio_device_it_get_flag(FN1_STATUS_PCRRT_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(FN1_STATUS_PCRRT_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(FN1_STATUS_PCWRT_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(FN1_STATUS_PCWRT_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(FN1_RTC_SET_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(FN1_RTC_SET_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(FN1_CLINTRD_INTERRPT) == HAL_SET)
    {
```

```
        hal_sdio_device_it_clr_flag(FN1_CLINTRD_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(FN1_INT_EN_UP_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(FN1_INT_EN_UP_INTERRPT);
    }
    if(hal_sdio_device_it_get_flag(FN1_M2S_INT_INTERRPT) == HAL_SET)
    {
        hal_sdio_device_it_clr_flag(FN1_M2S_INT_INTERRPT);
    }
}
```

17 WS2811控制接口

17.1 WS2811控制接口简介

WS2811控制接口专门用于控制WS2811系列LED控制芯片（或支持WS2811芯片协议的其它芯片）而专门研发的外设。

17.2 WS2811控制接口主要特征

- 支持DMA传输
- 支持自定义波特率（7位宽度）
- 支持一路中断(DR为空时触发)

17.3 WS2811控制接口概述

1. WS2811支持通过DMA传输RGB数据。
2. 支持DR位空时，触发中断。
3. 支持7位宽的自定义波特率。

```
    对于WS2811来说，最重要的就是T0L,T1L,T0H,T1H的时间，下面简单叙述下这几个值如何设置。

    首先我们需要知道的是，我们已经人为的固定了T0H和T0L的比例为 1 : 4 ,T1H和T1L的比例为  4 : 1,(根据WS2811手册这个比例是成立的)

    而这里所说的波特率就是单个0(T0)或单个1(T1)的时间，所以T0 = T0H + T0L, T1 = T1H + T1L,

    根据WS2811手册和T0H,T0L,T1H,T1L之间的比例，一般我们认为(以下数据为实测值，可以直接使用):

        T0 = 1125 ns,T0L = 900ns,T0H = 225ns

        T1 = 1125 ns,T1L = 225ns,T1H = 900ns

    而 T0 = T1 = (br + 1) * 5 * (1 / pclk)

    其中br为WS2811 BR寄存器，pclk为外设时钟，FPGA上pclk = 40M,所以 br = 8;
```

17.4 示例代码

```
/**
 * @file     ln_ws2811_test.c
```

```
 * @author   BSP Team
 * @brief
 * @version  0.0.0.1
 * @date     2021-08-26
 *
 * @copyright Copyright (c) 2021 Shanghai Lightning Semiconductor Technology Co. Ltd
 *
 */


/*
        WS2811外设测试说明：


        1. 对于WS2811来说，最重要的就是T0L,T1L,T0H,T1H的时间，下面简单叙述下这几个值如何设置。

            首先我们需要知道的是，我们已经人为的固定了T0H和T0L的比例为 1 ： 4 ,T1H和T1L的比例为  4 : 1,(根据WS2811手册这个比例是成立的)

            而这里所说的波特率就是单个0(T0)或单个1(T1)的时间，所以T0 = T0H + T0L, T1 = T1H + T1L,

            根据WS2811手册和T0H,T0L,T1H,T1L之间的比例，一般我们认为(以下数据为实测值，可以直接使用):

                T0 = 1125 ns,T0L = 900ns,T0H = 225ns

                T1 = 1125 ns,T1L = 225ns,T1H = 900ns

            而 T0 = T1 = (br + 1) * 5 * (1 / pclk)

            其中br为WS2811 BR寄存器，pclk为外设时钟，FPGA上pclk = 40M,所以 br = 8;


        2. 一般WS2811都是配合DMA使用，DMA使用的是通道二。

        3. WS2811不发送数据的时候，会默认引脚为低电平，根据WS2811的协议，总线低电平持续280us 以上，会RESET 总线，完成传输。

        4. 使用WS2811 + DMA的时候，WS2811 Empty Interrupt会不响应。（中断的功能和DMA冲突，所以要把WS2811 DMA EN 关掉才行）
*/

#include "ln_ws2811_test.h"

#include "hal/hal_gpio.h"
#include "hal/hal_dma.h"
#include "hal/hal_ws2811.h"
#include "ln_test_common.h"
#include "log.h"

#define LED_AMOUNT  50

////////////          LED1                          LED2
LED3                        ...
```

```
////////////数据格式  R7 - R0   G7 - G0   B7 - B0  ;R7 - R0   G7 - G0   B7 - B0
;R7 - R0   G7 - G0   B7 - B0  ...

//          LED1                          LED2                          LED3
                 ...
//数据格式  B7 - B0   R7 - R0   G7 - G0  ;B7 - B0   R7 - R0   G7 - G0 ;B7 - B0
R7 - R0   G7 - G0  ...
static unsigned char led_data_arr[LED_AMOUNT * 3 ] = {0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00,
                                                      0x00,0x00,0xFF,
                                                      0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00,
                                                      0x00,0x00,0xFF,
                                                      0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00,
                                                      0x00,0x00,0xFF,
                                                      0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00,
                                                      0x00,0x00,0xFF,
                                                      0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00,
                                                      0x00,0x00,0xFF,
                                                      0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00,
                                                      0x00,0x00,0xFF,
                                                      0xFF,0x00,0x00,
                                                      0x00,0xFF,0x00};

static unsigned char led_data_arr1[LED_AMOUNT * 3] = {0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00,
                                                      0x00,0x00,0x00};


void ln_ws2811_send_data(unsigned char *send_data,unsigned int data_len)
{
    //配置DMA传输参数。
    hal_dma_set_mem_addr(DMA_CH_2,(uint32_t)send_data);
    hal_dma_set_data_num(DMA_CH_2,data_len);

    //开始传输。
    hal_dma_en(DMA_CH_2,HAL_ENABLE);
```

```
    //等待传输完成。
    while( hal_dma_get_data_num(DMA_CH_2) != 0);

    //发送完成后及时关闭DMA，为下次配置DMA参数做准备。
    hal_dma_en(DMA_CH_2,HAL_DISABLE);
}

void ln_ws2811_init()
{
    // 1. 配置WS2811引脚复用
    hal_gpio_afio_select(GPIOB_BASE,GPIO_PIN_5,WS2811_OUT);
    hal_gpio_afio_en(GPIOB_BASE,GPIO_PIN_5,HAL_ENABLE);

    // 2. 初始化WS2811配置
    ws2811_init_t_def ws2811_init;

    ws2811_init.br = 8;                                    //baud rate = (br+1)*5 *
(1 / pclk)
    hal_ws2811_init(WS2811_BASE,&ws2811_init);             //初始化WS2811

    hal_ws2811_en(WS2811_BASE,HAL_ENABLE);                 //使能WS2811
    hal_ws2811_dma_en(WS2811_BASE,HAL_ENABLE);             //使能WS2811 DMA

    hal_ws2811_it_cfg(WS2811_BASE,WS2811_IT_EMPTY_FLAG,HAL_ENABLE); //配置WS2811中
断

    NVIC_EnableIRQ(WS2811_IRQn);                           //使能WS2811中断
    NVIC_SetPriority(WS2811_IRQn,     1);                  //设置WS2811中断优先级

    // 3. 配置DMA
    dma_init_t_def dma_init;
    memset(&dma_init,0,sizeof(dma_init));

    dma_init.dma_mem_addr = (uint32_t)led_data_arr;        //配置内存地址
    dma_init.dma_data_num = 0;                             //设置传输数量
    dma_init.dma_dir = DMA_READ_FORM_MEM;                  //设置传输方向
    dma_init.dma_mem_inc_en = DMA_MEM_INC_EN;              //使之内存是否自增
    dma_init.dma_p_addr = WS2811_DATA_REG;                 //设置外设地址

    hal_dma_init(DMA_CH_2,&dma_init);                      //DMA初始化
    hal_dma_en(DMA_CH_2,HAL_DISABLE);                      //使能DMA

    while(1)
    {

        ln_ws2811_send_data(led_data_arr,3 * 3);                  //点亮三盏灯

        ln_delay_ms(500);

        ln_ws2811_send_data(led_data_arr1,LED_AMOUNT * 3);        //熄灭所有灯

        ln_delay_ms(500);
    }

}
```

```c
void ln_ws2811_test(void)
{
    ln_ws2811_init();
}

void WS2811_IRQHandler(void)
{
    hal_ws2811_set_data(WS2811_BASE,11);
}
```

18 模数转换器（ADC）

18.1 ADC简介

ADC，指模/数转换器或者模数转换器。是指将连续变化的模拟信号转换为离散的数字信号的器件。

18.2 ADC主要特征

- 支持6通道轮询采样
- 12bit分辨率
- 单次和连续转换模式
- 支持DMA
- 支持转换完成中断

18.3 ADC控制接口概述

粗测电压值对应ADC原始值关系

电压值（V）	ADC值
0	32
0.01	32
0.05	96
0.11	160
0.18	224
0.23	288
0.28	352
0.33	416
0.4	480
0.45	544
0.51	608
0.6	672
0.8	928
1	1120
1.1	1248
1.2	1376
1.3	1504
1.4	1568
1.5	1696
2.0	2272
2.5	2848
3.0	3360
3.1	3488
3.2	3616
3.3	3744

18.4　示例代码

注：如果要使用ADC的DMA，除了要使能ADC_DMA_EN，还要配置ADC的转换中断使能（不用配置NVIC），这样才能触发DMA采集数据。