

# 零死角玩转STM32

与野火同行 乐意惬意无边



原创教程，完全开源。



由浅入深，结合实操。



通俗易懂，详尽解读。



配套板子，全面玩转。



强强联合，不断更新。



野火团队 Wild Fire Team

## 0、友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，与野火同行，乐意惬意无边。

另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

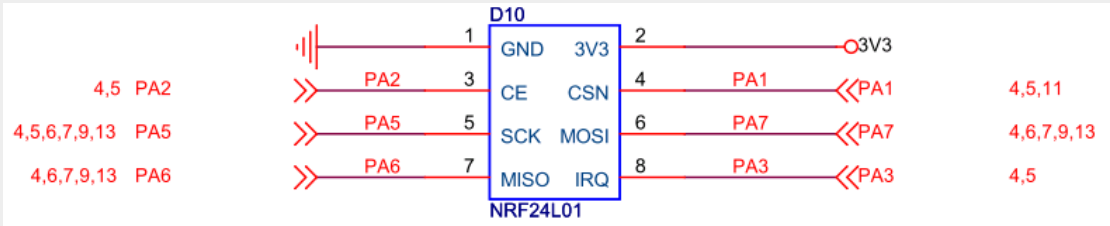


## 9、2.4G 无线（NRF24L01+）

### 9.1 实验描述及工程文件清单

实验描述	利用 NRF24L01+无线模块，使两块 STM32 开发板实现无线传输数据。用串口输出实验结果到 pc。
硬件连接	PA4-SPI1-NSS : W25X16-CS PA5-SPI1-SCK : W25X16-CLK PA6-SPI1-MISO : W25X16-DO PA7-SPI1-MOSI : W25X16-DIO
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_spi.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c SPI_NRF.c

野火 STM32 开发板 2.4G 无线模块接口图：

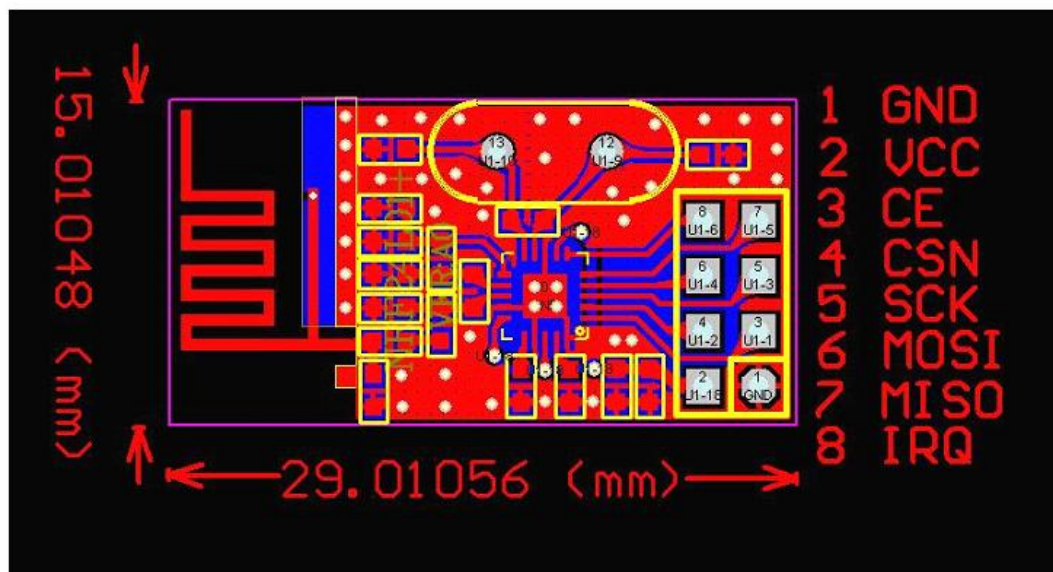


## 9.2 NRF24L01 模块简介

本实验采用的无线模块芯片型号为 **NRF24L01+**，是工作在 2.4~2.5GHz 频段的，具备自动重发功能，6 个数据传输通道，最大无线传输速率为 **2Mbps**。**MCU** 可与该芯片通过 **SPI** 接口访问芯片的寄存器进行配置。

以下是该模块的硬件电路：

截图来自《NRF24L01 模块说明书.pdf》，page3



注意：这个模块的工作电压为 3.3V，实验时请把 vcc 接到板上的 3v3 接口，超过 3.6v 该模块会烧坏！

引脚说明及本实验中与开发板的连接：

Pin	Name	Description	与开发板相连
1	CE	Chip Enable Activates RX or TX mode	排针 P5 的 PA2
2	CSN	SPI Chip Select	排针 P3 的 PA1
3	SCK	SPI Clock	排针 P5 的 PA5
4	MOSI	SPI Slave Data Input	排针 P5 的 PA7
5	MISO	SPI Slave Data Output, with tri-state option	排针 P5 的 PA6
6	IRQ	Maskable interrupt pin. Active low	排针 P5 的 PA3
7	VDD	Power Supply (+1.9V - +3.6V DC)	电源 3v3

8	VSS	Ground (0V)	电源 GND 接口
---	-----	-------------	-----------

这个例程采用的是 **STM32** 的 **SPI1** 接口，但其中的硬件 **SPI1-CSN** 端口（用于片选）已经在 **2M-FLASH** 上采用，所以本实验用一个空闲端口 **PA1** 用作无线模块的片选，由软件产生片选信号。

请注意区分这个模块的 **CSN** 片选信号与 **CE** 使能信号的功能。

**CSN** 端口是 **SPI** 通讯协议中的片选端。多个 **SPI** 设备可以共用 **STM32** 的 **SCK**, **MISO**, **MOSI** 端口，不同的设备间就是用 **CSN** 来区分。

**CE** 实际是 **NRF24L01** 的芯片使能端，通过配置 **CE** 可以使 **NRF24L01** 进入不同的状态。如下图示：

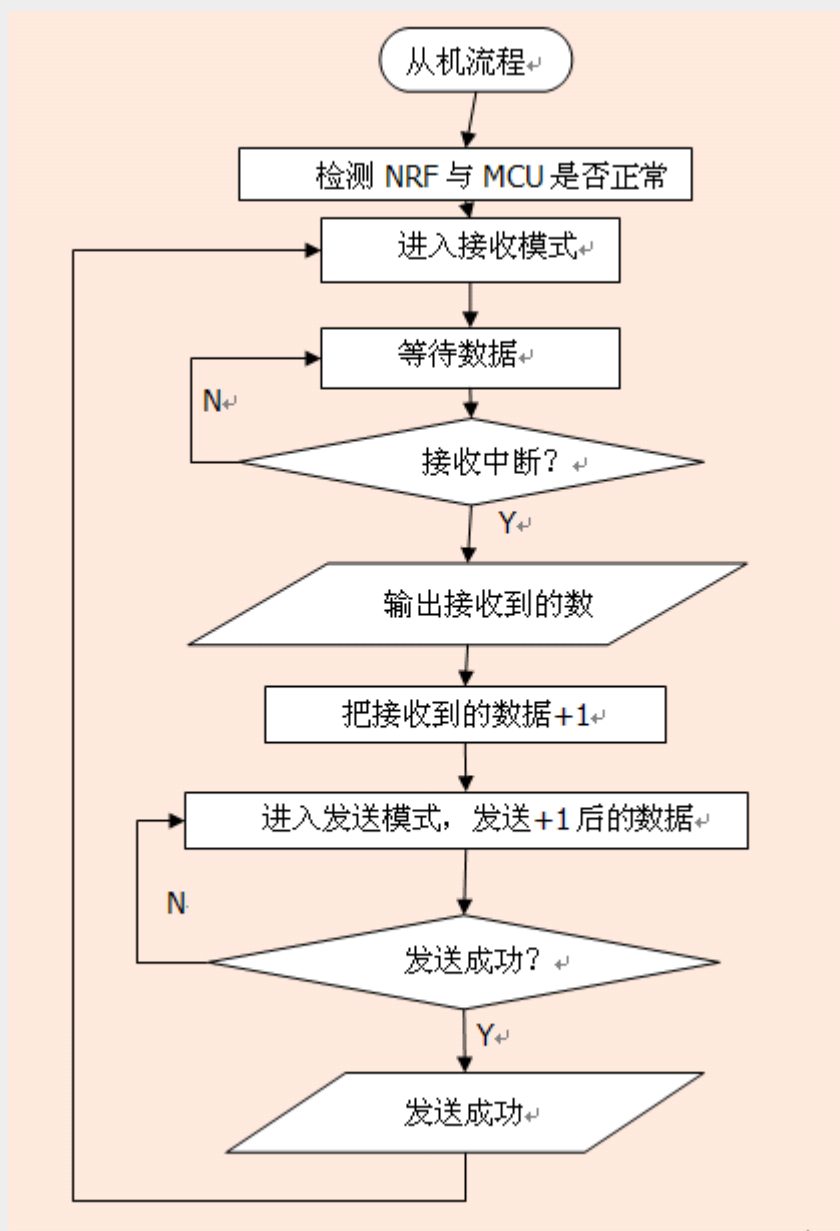
截图来自：《nRF24L01P(新版无线模块控制 IC).PDF》，page24.

Mode	PWR_UP register	PRIM_RX register	CE input pin	FIFO state
RX mode	1	1	1	-
TX mode	1	0	1	Data in TX FIFOs. Will empty all levels in TX FIFOs <sup>a</sup> .
TX mode	1	0	Minimum 10μs high pulse	Data in TX FIFOs. Will empty one level in TX FIFOs <sup>b</sup> .
Standby-II	1	0	1	TX FIFO empty.
Standby-I	1	-	0	No ongoing packet transmission.
Power Down	0	-	-	-

### 9.3 代码分析

这个实验用到两个代码，主机和从机的代码驱动是一样的，区别只是 **main** 中函数调用的流程不一样，从机接收模式的时候，相应的主机在发送模式。

附从机流程图：



下面以的从机的代码为例进行分析。

首先要添加用的库文件，在工程文件夹下 **Fwlib** 下我们需添加以下库文件：

```
1. stm32f10x_gpio.c
2. stm32f10x_rcc.c
3. stm32f10x_usart.c
4. stm32f10x_spi.c
```

还要在 **stm32f10x\_conf.h** 中把相应的头文件添加进来：

```
1. #include "stm32f10x_gpio.h"
2. #include "stm32f10x_spi.h"
3. #include "stm32f10x_rcc.h"
4. #include "stm32f10x_usart.h"
```



进入 `main` 函数，边看代码边了解程序的流程：

```
1. int main(void)
2. {
3.     /* 串口1 初始化 */
4.     USART1_Config();
5.
6.     /*SPI 接口初始化*/
7.     SPI_NRF_Init();
8.
9.     printf("\r\n 这是一个 NRF24L01 无线传输实验 \r\n");
10.    printf("\r\n 这是无线传输 从机端 的反馈信息\r\n");
11.    printf("\r\n    正在检测 NRF 与 MCU 是否正常连接。。。 \r\n");
12.
13.    /*检测 NRF 模块与 MCU 的连接*/
14.    status = NRF_Check();
15.    if(status == SUCCESS)
16.        printf("\r\n        NRF 与 MCU 连接成功\r\n");
17.    else
18.        printf("\r\n    正在检测 NRF 与 MCU 是否正常连接。。。 \r\n");
19.
20.    while(1)
21.    {
22.        printf("\r\n 从机端 进入接收模式\r\n");
23.        NRF_RX_Mode();
24.
25.        /*等待接收数据*/
26.        status = NRF_Rx_Dat(rxbuf);
27.
28.        /*判断接收状态*/
29.        if(status == RX_DR)
30.        {
31.            for(i=0;i<4;i++)
32.            {
33.                printf("\r\n 从机端 接收到 主机端 发送的数据
为: %d \r\n",rxbuf[i]);
34.                /*把接收的数据+1 后发送给主机*/
35.                rxbuf[i]+=1;
36.                txbuf[i] = rxbuf[i];
37.            }
38.
39.            printf("\r\n 从机端 进入自应答发送模式\r\n");
40.            NRF_TX_Mode();
41.
42.            /*不断重发，直至发送成功*/
43.            do
44.            {
45.                status = NRF_Tx_Dat(txbuf);
46.            }while(status == MAX_RT);
47.        }
48.    }
```

报告野火，这个代码错了，没有调用 `SystemInit()` 函数来设置时钟！是的，大家熟悉的 `SystemInit()` 函数不见了，但这样并没有出错，原因是这个例程的库是 **3.5 版本**的！在 **3.5 版本**的库中 `SystemInit()` 函数在启动文件 `startup_stm32f10x_hd.d` 中已用汇编语句调用了，设置的时钟为默认的 **72M**。所以在 `main` 函数就不需要再调用啦，当然，再调用一次也是没问题的。





关于 `USART1_Config()` 函数, 是用来配置串口的, 关于这两个函数的具体讲解可以参考前面的教程, 这里不再详述。

接着进入 `SPI_NRF_Init()` 函数是怎样配置 STM32 的 SPI 接口的:

```
1. void SPI_NRF_Init(void)
2. {
3.     SPI_InitTypeDef SPI_InitStructure;
4.     GPIO_InitTypeDef GPIO_InitStructure;
5.
6.     /*使能 GPIOB, GPIOD, 复用功能时钟*/
7.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOE|RCC_APB2Periph_AFIO, ENABLE);
8.
9.     /*使能 SPI1 时钟*/
10.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
11.
12.    /*配置 SPI_NRF_SPI 的 SCK, MISO, MOSI 引脚, GPIOA^5, GPIOA^6, GPIOA^7 */
13.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
14.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
15.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用功能
16.    GPIO_Init(GPIOA, &GPIO_InitStructure);
17.
18.    /*配置 SPI_NRF_SPI 的 CE 引脚, GPIOA^2 和 SPI_NRF_SPI 的 CSN 引脚: NSS GPIOA^1*/
19.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_1;
20.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
21.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
22.    GPIO_Init(GPIOA, &GPIO_InitStructure);
23.
24.    /*配置 SPI_NRF_SPI 的 IRQ 引脚, GPIOA^3*/
25.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
26.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
27.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
28.    GPIO_Init(GPIOA, &GPIO_InitStructure);
29.
30.    /* 这是自定义的宏, 用于拉高 csn 引脚, NRF 进入空闲状态 */
31.    NRF_CSN_HIGH();
32.
33.    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //
    // 双线全双工
34.    SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //
    // 主模式
35.    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; //
    // 数据大小 8 位
36.    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; //
    // 时钟极性, 空闲时为低
37.    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge; //
    // 第 1 个边沿有效, 上升沿为采样时刻
38.    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //
    // NSS 信号由软件产生
39.    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
    // 8 分频, 9MHz
40.    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //
    // 高位在前
41.    SPI_InitStructure.SPI_CRCPolynomial = 7;
42.    SPI_Init(SPI1, &SPI_InitStructure);
43.
44.    /* Enable SPI1 */
45.    SPI_Cmd(SPI1, ENABLE);
46. }
```





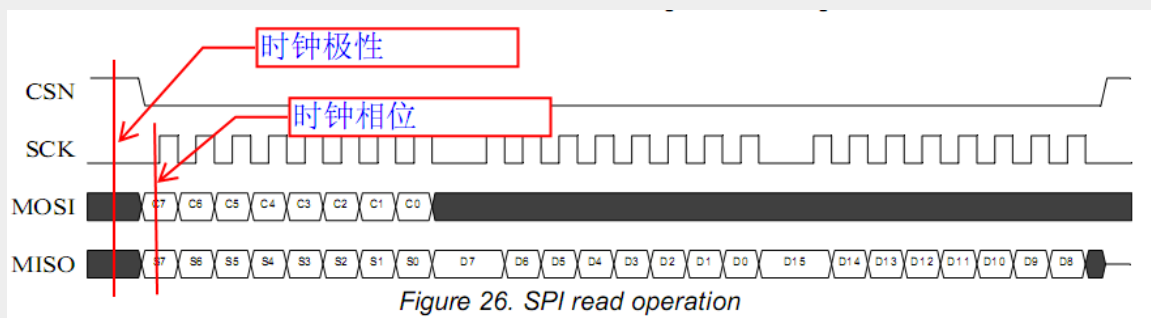
基本流程就是：

1. 开启所用到的端口的时钟 **GPIOA**，复用时钟 **AFIO**，**SPI1** 时钟。  
其中 **AFIO** 时钟是因为 **GPIOA** 是用作 **SPI** 方式的，所以别忘记开启它。
2. 配置 **SPI** 的模式，配置 **SPI** 模式的关键是时钟极性（**CPOL**）第 36 行，和时钟相位（**CPHA**）第 37 行。这是根据 **NRF24L01** 的时序图来设置的。

所谓**时钟极性**，就是 **SPI** 端口在空闲的时候，**SCK** 的电平，例程中为低电平。

而**时钟相位**，则是 **SPI** 采集数据的时钟边沿，例程中为第一个时钟边沿(上升沿)。

截图来自：《nRF24L01P(新版无线模块控制 IC).PDF》，page52



继续分析 **main** 函数，在第 14 行调用了 **NRF\_Check()** 函数，分析一下它：

```
1. u8 NRF_Check(void)
2. {
3.     u8 buf[5]={0xC2,0xC2,0xC2,0xC2,0xC2};
4.     u8 buf1[5];
5.     u8 i;
6.
7.     /*写入 5 个字节的地址. */
8.     SPI_NRF_WriteBuf(NRF_WRITE_REG+TX_ADDR,buf,5);
9.
10.    /*读出写入的地址 */
11.    SPI_NRF_ReadBuf(TX_ADDR,buf1,5);
12.
13.    /*比较*/
14.    for(i=0;i<5;i++)
15.    {
16.        if(buf1[i]!=0xC2)
17.            break;
18.    }
19.
20.    if(i==5)
21.        return SUCCESS ;           //MCU 与 NRF 成功连接
22.    else
```

```
23.         return ERROR ;           //MCU 与 NRF 不正常连接
24. }
```

这个函数是通过调用 `SPI_NRF_WriteBuf()` 和 `SPI_NRF_ReadBuf()` 函数，对 NRF 的地址寄存器进行读写，先向寄存器写入数据，再读取出来进行比较，以此来检验 **NRF24L01** 是否与 **MCU** 正常连接的。

`SPI_NRF_WriteBuf()` 和 `SPI_NRF_ReadBuf()` 函数很类似，前者实现向寄存器写入一串数据，后者实现读取数据功能。下面以 `SPI_NRF_WriteBuf()` 为例：

```
1. u8 SPI_NRF_WriteBuf(u8 reg ,u8 *pBuf,u8 bytes)
2. {
3.     u8 status,byte_cnt;
4.     NRF_CE_LOW();
5.     /*置低 CSN, 使能 SPI 传输*/
6.     NRF_CSN_LOW();
7.
8.     /*发送寄存器号*/
9.     status = SPI_NRF_RW(reg);
10.
11.    /*向缓冲区写入数据*/
12.    for(byte_cnt=0;byte_cnt<bytes;byte_cnt++)
13.        SPI_NRF_RW(*pBuf++);    //写数据到缓冲区
14.
15.    /*CSN 拉高, 完成*/
16.    NRF_CSN_HIGH();
17.
18.    return (status);    //返回 NRF24L01 的状态
19. }
```

`SPI_NRF_WriteBuf()` 调用了 `SPI_NRF_RW()` 函数：

```
1. u8 SPI_NRF_RW(u8 dat)
2. {
3.     /* 当 SPI 发送缓冲器非空时等待 */
4.     while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
5.
6.     /* 通过 SPI2 发送一字节数据 */
7.     SPI_I2S_SendData(SPI1, dat);
8.
9.     /* 当 SPI 接收缓冲器为空时等待 */
10.    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
11.
12.    /* Return the byte read from the SPI bus */
13.    return SPI_I2S_ReceiveData(SPI1);
14. }
```

`SPI_NRF_WriteBuf()`流程：

1. 使 **CE** 端口置低，**SPI** 通讯时先进入**待机模式**。使 **CSN** 端口置低，**开启 SPI 通讯**。



- 调用 `SPI_NRF_RW()` 向 NRF 发送将要写入的寄存器地址, `SPI_NRF_RW()` 返回的是 **STATUS** 寄存器的数据 (不是将要操作的寄存器的值哦!)
- 连续写入数据, 最后拉高 **CSN** 端口, 结束 SPI 传输。

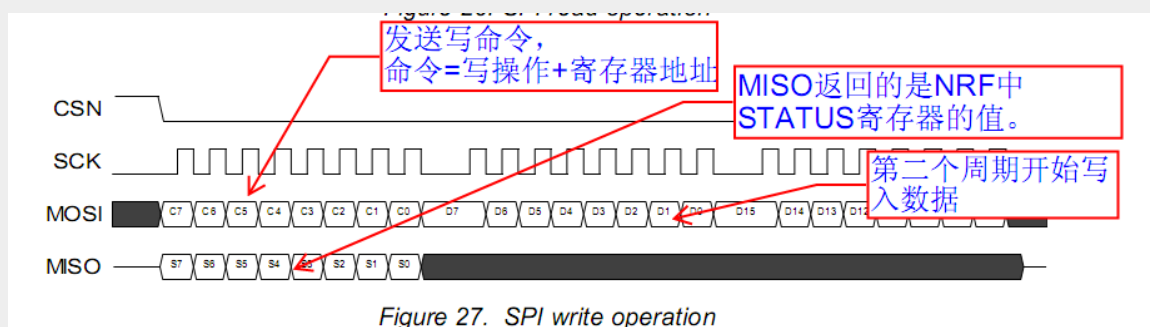
命令的组织形式: 完整的写命令 = 写命令+寄存器地址。

截图来自: 《nRF24L01P(新版无线模块控制 IC).PDF》, page51

Command name	Command word (binary)	# Data bytes	Operation
R_REGISTER	000A AAAA	1 to 5 LSByte first	Read command and status registers. AAAA = 5 bit Register Map Address
W_REGISTER	001A AAAA	1 to 5 LSByte first	Write command and status registers. AAAA = 5 bit Register Map Address Executable in power down or standby modes only.

写时序:

截图来自: 《nRF24L01P(新版无线模块控制 IC).PDF》, page52



回到 `main` 函数, 检查完 NRF 与 MCU 的连接, 从机开始进入接收模式

式 `NRF_RX_Mode()` 函数:

```
1. void NRF_RX_Mode(void)
2.
3. {
4.     NRF_CE_LOW();
5.
6.     SPI_NRF_WriteBuf(NRF_WRITE_REG+RX_ADDR_P0,RX_ADDRESS,RX_ADR_WIDTH);
7.     //写 RX 节点地址
8.     SPI_NRF_WriteReg(NRF_WRITE_REG+EN_AA,0x01); //使能通道 0 的自动应
9.     答
10.    SPI_NRF_WriteReg(NRF_WRITE_REG+EN_RXADDR,0x01); //使能通道 0 的接收地
11.    址
12.    SPI_NRF_WriteReg(NRF_WRITE_REG+RF_CH,CHANAL); //设置 RF 通信频
13.    率
```

```
14. SPI_NRF_WriteReg(NRF_WRITE_REG+RX_PW_P0,RX_PLOAD_WIDTH); //选择通道 0
    的有效数据宽度
15.
16. SPI_NRF_WriteReg(NRF_WRITE_REG+RF_SETUP,0x0f); //设置 TX 发射参数,0db
    增益,2Mbps,低噪声增益开启
17.
18. SPI_NRF_WriteReg(NRF_WRITE_REG+CONFIG, 0x0f); //配置基本工作模式的参
    数;PWR_UP,EN_CRC,16BIT_CRC,接收模式
19.
20. /*CE 拉高, 进入接收模式*/
21. NRF_CE_HIGH();
22. }
```

配置接收模式和发送模式都是向 NRF 寄存器写入配置参数, 这些参数具体意义在

《nRF24L01P(新版无线模块控制 IC).PDF》, page57

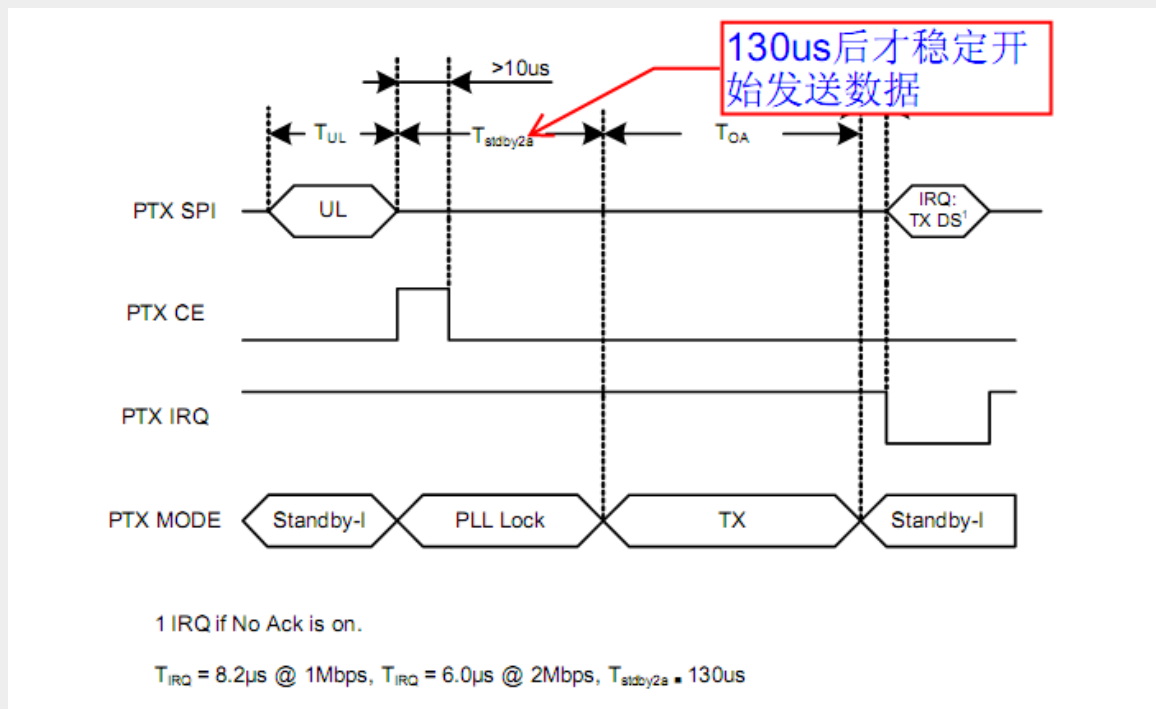
要注意的是从机和主机的参数要一致。

下面是发送模式的配置:

```
1. void NRF_TX_Mode(void)
2. {
3.     NRF_CE_LOW();
4.
5.     SPI_NRF_WriteBuf(NRF_WRITE_REG+TX_ADDR,TX_ADDRESS,TX_ADR_WIDTH);
    //写 TX 节点地址
6.
7.     SPI_NRF_WriteBuf(NRF_WRITE_REG+RX_ADDR_P0,RX_ADDRESS,RX_ADR_WIDTH);
    //设置 TX 节点地址,主要为了使能 ACK
8.
9.     SPI_NRF_WriteReg(NRF_WRITE_REG+EN_AA,0x01); //使能通道 0 的自动应
    答
10.
11.    SPI_NRF_WriteReg(NRF_WRITE_REG+EN_RXADDR,0x01); //使能通道 0 的接收地
    址
12.
13.    SPI_NRF_WriteReg(NRF_WRITE_REG+SETUP_RETR,0x1a); //设置自动重发间隔时
    间:500us + 86us;最大自动重发次数:10 次
14.
15.    SPI_NRF_WriteReg(NRF_WRITE_REG+RF_CH,CHANAL); //设置 RF 通道为
    CHANAL
16.
17.    SPI_NRF_WriteReg(NRF_WRITE_REG+RF_SETUP,0x0f); //设置 TX 发射参数,0db
    增益,2Mbps,低噪声增益开启
18.
19.    SPI_NRF_WriteReg(NRF_WRITE_REG+CONFIG,0x0e); //配置基本工作模式的参
    数;PWR_UP,EN_CRC,16BIT_CRC,发射模式,开启所有中断
20.
21.    /*CE 拉高, 进入发送模式*/
22.    NRF_CE_HIGH();
23.    Delay(0xffff); //CE 要拉高一段时间才进入发送模式
24. }
```

在发送模式配置要特别注意一点, 第 23 行, 延时, STM32 运行频率比 NRF 模块快得多, 不加延时直接发送数据的话很容易导致发送出错。

截图来自：《nRF24L01P(新版无线模块控制 IC).PDF》，page42



回到 main 函数，配置完接收模式后就开始等待 NRF 模块传来中断接收数据。

由 NRF\_Rx\_Dat() 函数来实现：

```
1. u8 NRF_Rx_Dat(u8 *rxbuf)
2. {
3.     u8 state;
4.     NRF_CE_HIGH(); //进入接收状态
5.     /*等待接收中断*/
6.     while(NRF_Read_IRQ() != 0);
7.
8.     NRF_CE_LOW(); //进入待机状态
9.     /*读取 status 寄存器的值 */
10.    state = SPI_NRF_ReadReg(STATUS);
11.
12.    /* 清除中断标志 */
13.    SPI_NRF_WriteReg(NRF_WRITE_REG + STATUS, state);
14.
15.    /*判断是否接收到数据*/
16.    if(state & RX_DR) //接收到数据
17.    {
18.        SPI_NRF_ReadBuf(RD_RX_PLOAD, rxbuf, RX_PLOAD_WIDTH); //读取数据
19.        SPI_NRF_WriteReg(FLUSH_RX, NOP); //清除 RX FIFO 寄存器
20.        return RX_DR;
21.    }
22.    else
23.        return ERROR; //没收到任何数据
24. }
```

接收流程:

1. 等待 IRQ 引脚的信号, NRF 模块在接收到数据, 发送完成数据, 或重发超过次数都会在 IRQ 引脚进行标志。(这个实验中采用的是 STM32 循环读取 IRQ 引脚信号, 实际上可以把这个引脚配置成外部中断来减轻 MCU 的负担, 读者可以一试。)

关于 NRF 整个无线传输过程可以参照《nRF24L01P(新版无线模块控制 IC).PDF》的 page36~37 页的流程图。

2. 读取状态寄存器的值来判断是否接收正常, 利用 `SPI_NRF_ReadBuf()` 来从接收缓冲区读取数据到 STM32。
3. 清中断 (通过向 STATUS 寄存器写 1 可以清除相应的中断位), 清空接收缓冲区。

到这里就把所有的 NRF 驱动应用函数解说完啦! ^\_^

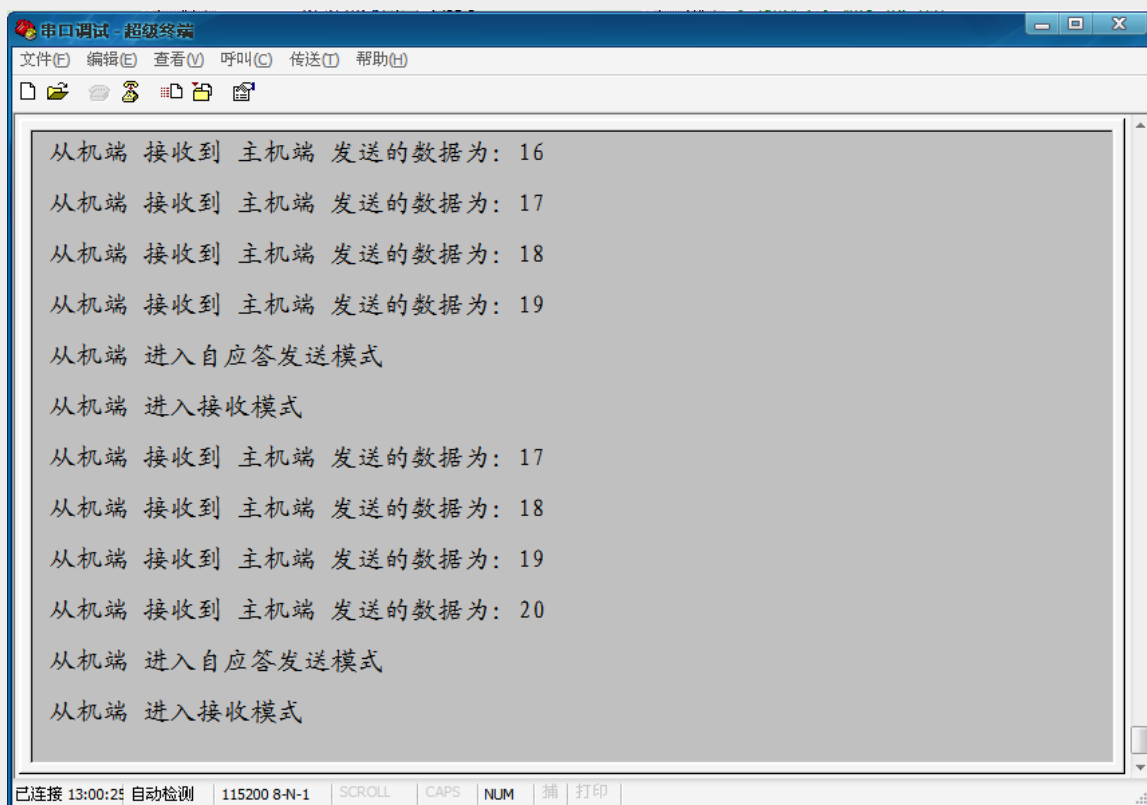
回到 main 函数, 从机把接收到的数据都加 1 之后再发送给主机, 主机又把数据发送回从机。。。如此循环, 就是整个实验的流程。

## 9.4 实验想象

实验时请先开启从机的电源, 再开启主机的电源, 这是代码的流程决定的。

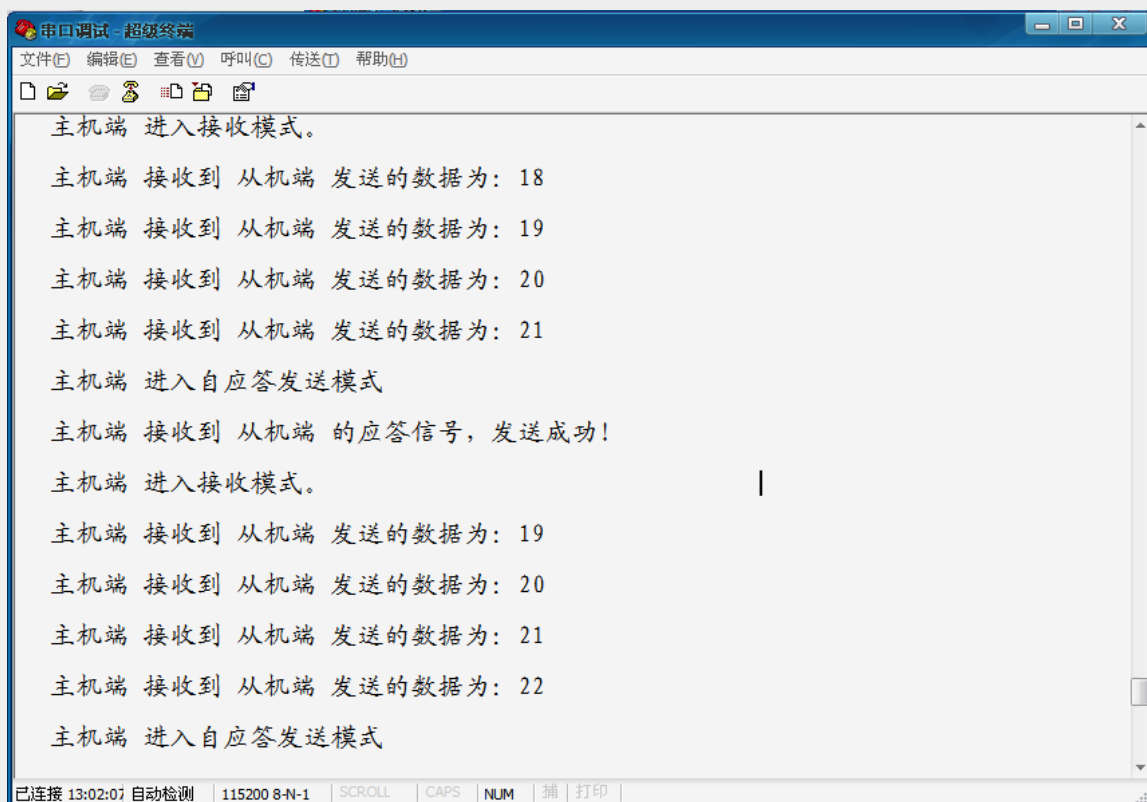


## 实验效果：从机的反馈：



```
串口调试-超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
从机端 接收到 主机端 发送的数据为：16
从机端 接收到 主机端 发送的数据为：17
从机端 接收到 主机端 发送的数据为：18
从机端 接收到 主机端 发送的数据为：19
从机端 进入自应答发送模式
从机端 进入接收模式
从机端 接收到 主机端 发送的数据为：17
从机端 接收到 主机端 发送的数据为：18
从机端 接收到 主机端 发送的数据为：19
从机端 接收到 主机端 发送的数据为：20
从机端 进入自应答发送模式
从机端 进入接收模式
已连接 13:00:25 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

## 主机的反馈：



```
串口调试-超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
主机端 进入接收模式。
主机端 接收到 从机端 发送的数据为：18
主机端 接收到 从机端 发送的数据为：19
主机端 接收到 从机端 发送的数据为：20
主机端 接收到 从机端 发送的数据为：21
主机端 进入自应答发送模式
主机端 接收到 从机端 的应答信号，发送成功！
主机端 进入接收模式。
主机端 接收到 从机端 发送的数据为：19
主机端 接收到 从机端 发送的数据为：20
主机端 接收到 从机端 发送的数据为：21
主机端 接收到 从机端 发送的数据为：22
主机端 进入自应答发送模式
已连接 13:02:07 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```



