nRF24L01模块

官网链接: https://www.nordicsemi.com/Products/nRF24-series

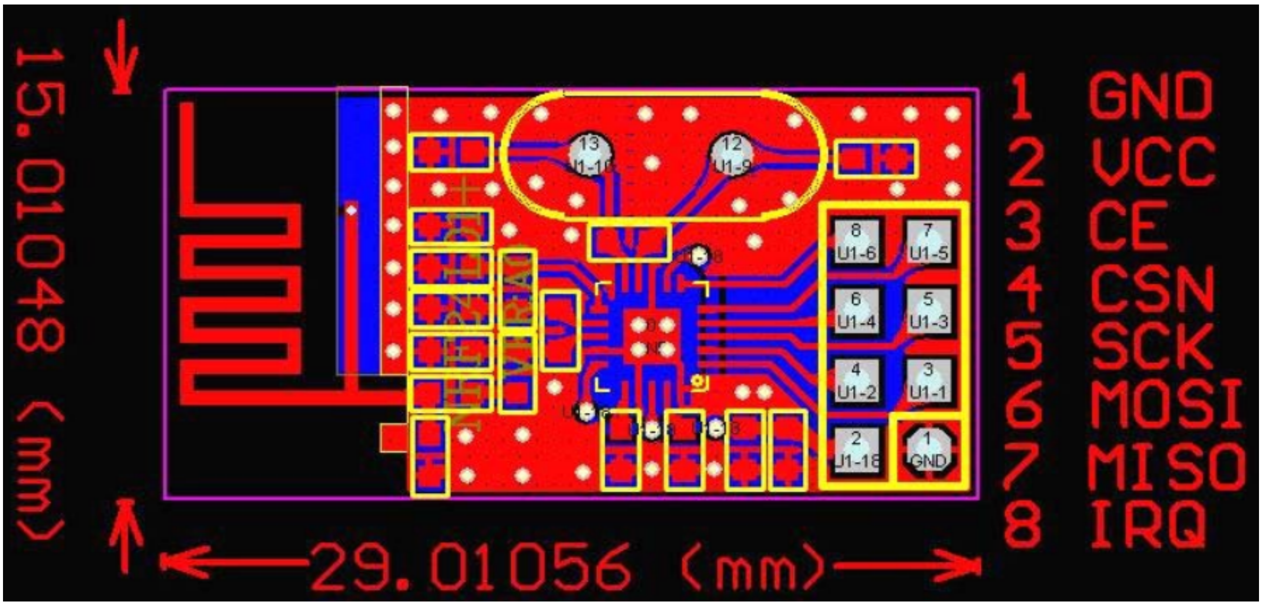
常见的无线收发模块, 工作在2.4GHz频段, 适合近距离遥控和数据传输.

nRF24L01是一个能兼顾距离和数据速率的无线模块, 在空旷环境下， 2M速率15米, 1M速率30米, 250K速率能达到50米. 和蓝牙相比距离更远, 和ESP8266这类以太网WiFi相比环境适应力更强.

参数

- 2.4GHz ISM频段
- 250Kbps, 1Mbps, 2Mbps三种空中传输速率
- 输出功率为 0dBm时发射功耗为11.3mA
- 空中传输速率为2Mbps时接收功耗为13.5mA
- Power down模式功耗低至900nA, Standby-I模式功耗低至26uA
- 1.9-3.6V的电压工作范围
- 支持6个接收通道(地址)
- IO口能承受5V电压
- ±60ppm 16MHz晶体振荡器
- 4×4mm QFN封装

nRF24L01 模块PIN布局



元件面朝自己,天线朝左(晶振在上), 右侧PIN脚的定义为:

IRQ	MISO
MOSI	CSK
CSN	CE
VCC	GND

PIN脚定义

CE	数字信号输入	Chip Enable, RX和TX模式选择

CSN	数字信号输入	SPI chip select, 低电平使能
SCK	数字信号输入	SPI serial clock
MOSI	数字信号输入	从机数据输入
MISO	数字信号输出	从机数据输出, 有三种状态选项
IRQ	数字信号输出	可屏蔽中断脚, 低电平使能
VDD	电源	1.9 - 3.6V
VSS	电源	接地 (0V)

PIN: IRQ

正常状态为高电位, 只有当STATUS寄存器的以下三个位被置位(拉高)时会拉低电压输出, 要清除中断, 需要相应地往这三个位写入 `1` .

1. RX_DR(Received Ready)
2. TX_DS (Data Sent)
3. MAX_RT(Transmit Failed After Max Retransmits)

可以通过寄存器地址 `0x00` 分别对这三种中断进行屏蔽

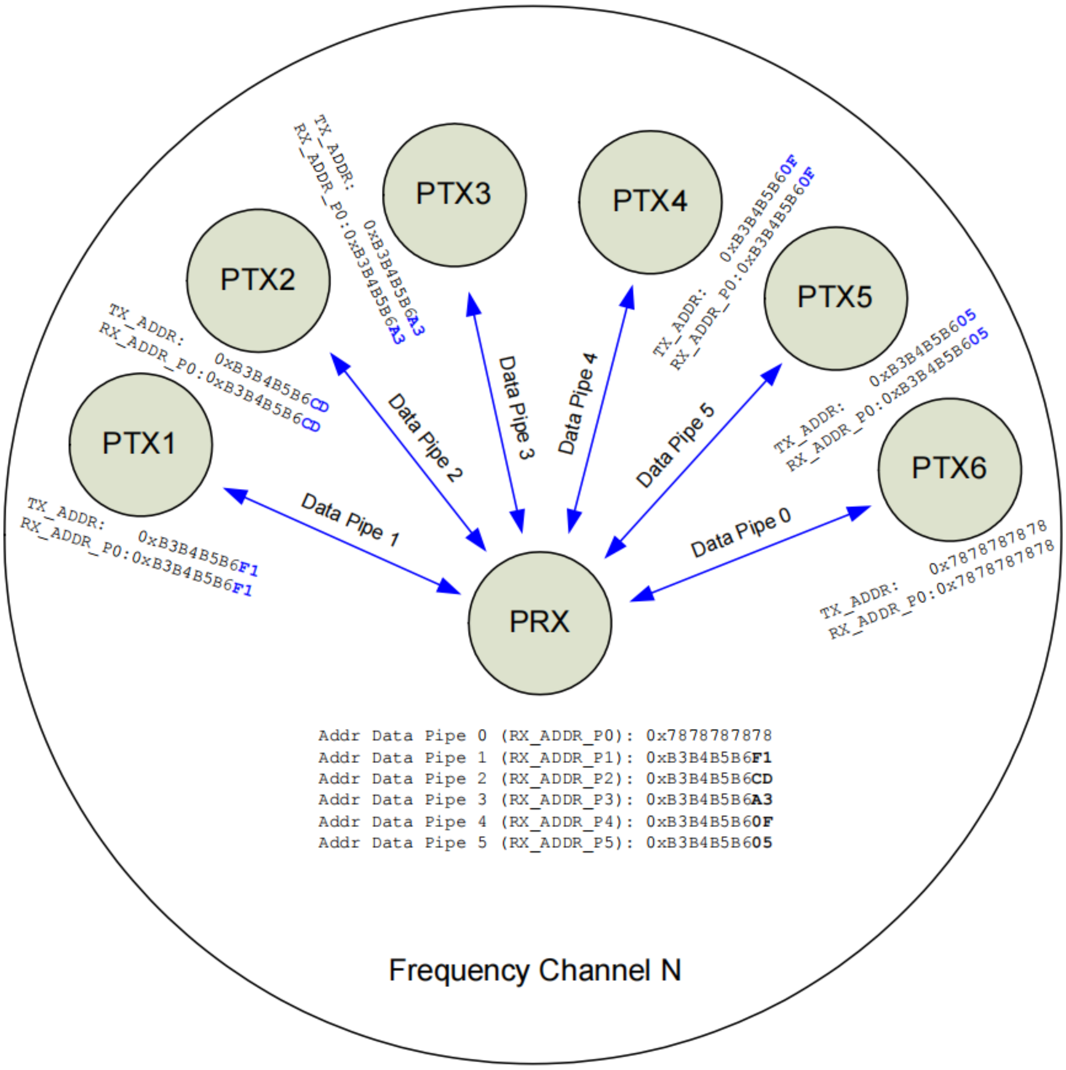
```
    The nRF24L01 has an active low IRQ pin. The IRQ pin has three sources: it is activated
when TX_DS (Transmit Data Sent), RX_DR (Receive Data Ready) or MAX_RT (Max Retransmit) bit
are set high by the Enhanced ShockBurst in the STATUS register. The IRQ pin resets when MCU
writes '1' in the STATUS register on the bit associated to the IRQ active source. As example,
we can suppose that a MAX_RT events happens. So, we will detect an IRQ transition from high
to low and checking STATUS register we will found that MAX_RT bit is high. So we should take
necessary actions and than set MAX_RT to high in the STATUS register to clear IRQ.
    To detect IRQ we can use EXT driver from ChibiOS/HAL. This driver launches a callback
when detects a transition (on rising, falling or both edge according to its configuration),
anyway, we will discuss this callback in detail later.
```

状态之间的转移

- Power Down 模式
  在该模式下, nRF24L01+的功耗最小, 不能进行发送或者接收. 但是所有寄存器的值保持不变, SPI处于有效状态, 允许对寄存器, TX/RX FIFO进行操作, PWR_UP(此位在CONFIG寄存器中)清0即进入该状态.
- Standby-I 模式
  将PWR_UP置1, 即进入Standby-I模式, 该模式既降低了nRF24L01+的平均功耗, 同时又保持尽可能短的启动时间, 将CE置1然后后清0, 就可以进入TX/RX模式, 然后又返回到Standby-I模式.
- Standby-II 模式
  当nRF24L01+设置为接收机(PTX), 并且CE=1, TX FIFO为空时即进入该模式. 相比Standby-I模式, 这种模式相对耗电, 一旦发送FIFO有新数据, 就会立即将数据打包发送出去.
- TX 模式, 进入该模式需要 PWR_UP=1, PRIM_RX=0, TX FIFO不为空, CE=1脉冲宽度超过10us
- RX 模式, 进入该模式需要 PWR_UP=1, PRIM_RX=1, CE=1

工作流程

发送流程



1. MCU通过SPI对NRF24L01进行基本配置, 配置自动应答通道使能, 设置自动重发次数不为0(在此设置可以重发数据包)设置为发送模式, 还有其他配置等等
2. MCU把要发送的数据和接收数据设备的地址通过SPI写入NRF24L01
3. CE引脚置高, 启动发送
4. 此时有两种情况
   - 在有限时间内收到应答信号, 则TX_DS置高(发送数据成功标志位), 并引发IRQ中断(引脚IRQ置低), 并清除TX buff(发送缓冲寄存器, 自行写代码清除), IRQ中断需要写状态寄存器进行复位(因为此处IRQ由TX_DS引发, 将TX_DS复位即可使IRQ复位)
   - 重发数据次数超过设定值, 则 MAX_RT 置高(达到最多重发次数标志位), 并引发IRQ中断(引脚IRQ置低), 不清除TX buff, IRQ中断需要写状态寄存器进行复位(因为此处IRQ由MAX_RT引发，将MAX_RT复位即可使IRQ复位)
5. 接收到应答信号产生中断或者达到最大重发次数产生中断后，NRF24L01继续发下一包数据
6. 当TX buff为空时, 进入待机模式二(当CE为高, TX buff为空时进入待机模式二), NRF24L01的工作模式图表在后面. 只要在适当时候拉高CE进行发送即可, 配置NRF24L01时CE置低)

接收流程

1. 与发送模式一样, 一开始MCU通过SPI对NRF24L01进行基本配置, 设置数据通道自动应答使能(在EN_AA寄存器进行设置, 即收到数据后自动向主机发送应答信号), 还有进行接收数据通道使能(在EN_RXADDR寄存器配置), 即选择六个接收通道的某一通道来接收数据, 设置为接收模式, 以及其他配置.
2. 拉高CE引脚(CE置高), 启动接收状态
3. 接收到一个有效数据包后, 数据存储在RX buff, 并产生RX_DR中断(RX_DR为接收数据成功标志位, 接收成功置1), 中断和发送模式一样, 需要复位
4. 接收设备自动向发送设备发送确认信号(这步是自动的)
5. 设置CE引脚为低, NRF24L01进入待机模式一
6. MCU通过SPI读取NRF24L01收到的数据

发送状态出现 1E (MAX_RT) 的原因汇总

经过各种环境的测试, 总结一下出现 1E 错误的原因

1. 发送状态下, 发送的目标地址 **TX_ADDR** 必须与**RX_ADDR_P0**相同!! 这个地址将用于接受对方返回的ACK, 如果RX_ADDR_P0填的不对, 对方依然能收到数据, 但是本地会多次重试后产生MAX_RT中断
2. 如果对方能收到数据, 上面的P0地址也与 TX_ADDR 一致, 仍然出现 MAX_RT, 可以检查一下发射功率, 经测试, 在1Mbps速率时, 如果功率设置为-12dbm, 就会频繁出现 MAX_RT 中断, 如果设为-6dbm和0dbm就不会.
3. 还有一种会出现MAX_RT中断的情况, 就是在处理发送的方法内使用了 `printf("%d, ...", u8_variable)` 这样的语句, 使用 `%x` 没问题, 但是如果使用 `%d` 就会出现, 具体原因未知.
4. 对于STC89/STC12系列的MCU, 如果以上都没问题, 可以换一个USB2TTL试试, 最近遇到的一个问题就跟我使用的PL2303的USB2TTL有关, 换成CH340就能正常收到2E状态.

总结

1. 发送过程
   - MCU通过SPI对NRF24L01进行基本配置，配置好NRF24L01
   - MCU将要发送的数据与接收数据设备的地址写入NRF24L01
   - CE引脚置高，启动发送
2. 接收过程
   - MCU通过SPI对NRF24L01进行基本配置，配置好NRF24L01
   - CE引脚置高，启动接收
   - MCU对 NRF24L01进行数据读取

NRF24L01的USB串口调试设备

淘宝上有配套出售的一种USB转接卡, 用于将NRF24L01通过USB到电脑, 此时NRF24L01相对于电脑成为一个串口设备, 通过 `AT` 命令进行通信.

- 所有命令均为大写
- 标点符号必须英文状态下的半角标点
- 无空格
- 不可更改的参数
  - 地址长度必须为5位
  - 数据长度必须是32个字节
  - 发射功率为0dbm
- 如果用上位机和这个USB转接卡进行调试的话, 要注意这里有个坑: 用户可用的字节为1-31个, **第0位不可用**, 这个字节系统保留, 用于记录传输的数据包长度. 例如: 串口发送 `abc` (ASCII码, 3 bytes), 实际传输 `3abc` (第0个字节就为3), 接收端根据第0字节中的数来判断收到的数据包长度, 再通过串口TX输出给电脑的就是 `abc` .

相关的命令有

- **AT?** 系统信息查询
- **AT+BAUD=n** n为1,2,3,4,5,6,7分别对应4800,9600,14400,19200,38400,115200的波特率
- **AT+RATE=n** n为1,2,3分别对应250Kbps ,1Mbps,2Mbps的传输速率
- **AT+RXA=0x??,0x??,0x??,0x??,0x??** 0x??为十六进制本机地址, 英文逗号分隔
- **AT+TXA=0x??,0x??,0x??,0x??,0x??** 目标地址, 其他同上
- **AT+FREQ=2.xxxG** 2. xxx为要设定的频率, 范围2.400GHz-2.525GHz, 超过范围无效, 小数点后面为三位数字, 不足三位需补零, 大写字母G不可缺少
- **AT+CRC=n** n等于8或者16, 设置8位或16位CRC校验
- **发送消息** 直接往串口输出, 且不符合上面命令格式的, 都会发送到目标地址

STM32F103C8T6

接线方式

STM32	nRF24L01
PA4 SPI1_NSS	N/A
PA5 SPI1_SCK	SCK
PA6 SPI1_MISO	MISO
PA7 SPI1_MOSI	MOSI
PB13	IRQ
PB14	CE
PB15	CSN

代码示例

Github项目: https://github.com/IOsetting/stm32f103-nrf24l01

常量定义

```
// SPI(nRF24L01) commands
#define NRF24L01_CMD_REGISTER_R     0x00 // Register read
#define NRF24L01_CMD_REGISTER_W     0x20 // Register write
#define NRF24L01_CMD_ACTIVATE       0x50 // (De)Activates R_RX_PL_WID, W_ACK_PAYLOAD,
W_TX_PAYLOAD_NOACK features
#define NRF24L01_CMD_RX_PLOAD_WID_R 0x60 // Read RX-payload width for the top R_RX_PAYLOAD in
the RX FIFO.
#define NRF24L01_CMD_RX_PLOAD_R     0x61 // Read RX payload
#define NRF24L01_CMD_TX_PLOAD_W     0xA0 // Write TX payload
#define NRF24L01_CMD_ACK_PAYLOAD_W  0xA8 // Write ACK payload
#define NRF24L01_CMD_TX_PAYLOAD_NOACK_W 0xB0 //Write TX payload and disable AUTOACK
#define NRF24L01_CMD_FLUSH_TX       0xE1 // Flush TX FIFO
#define NRF24L01_CMD_FLUSH_RX       0xE2 // Flush RX FIFO
#define NRF24L01_CMD_REUSE_TX_PL    0xE3 // Reuse TX payload
#define NRF24L01_CMD_LOCK_UNLOCK    0x50 // Lock/unlock exclusive features
#define NRF24L01_CMD_NOP            0xFF // No operation (used for reading status register)

// SPI(nRF24L01) register address definitions
#define NRF24L01_REG_CONFIG         0x00 // Configuration register
#define NRF24L01_REG_EN_AA          0x01 // Enable "Auto acknowledgment"
#define NRF24L01_REG_EN_RXADDR      0x02 // Enable RX addresses
#define NRF24L01_REG_SETUP_AW       0x03 // Setup of address widths
#define NRF24L01_REG_SETUP_RETR     0x04 // Setup of automatic re-transmit
#define NRF24L01_REG_RF_CH          0x05 // RF channel
#define NRF24L01_REG_RF_SETUP       0x06 // RF setup
#define NRF24L01_REG_STATUS         0x07 // Status register
#define NRF24L01_REG_OBSERVE_TX     0x08 // Transmit observe register
#define NRF24L01_REG_RPD            0x09 // Received power detector
#define NRF24L01_REG_RX_ADDR_P0     0x0A // Receive address data pipe 0
#define NRF24L01_REG_RX_ADDR_P1     0x0B // Receive address data pipe 1
#define NRF24L01_REG_RX_ADDR_P2     0x0C // Receive address data pipe 2
#define NRF24L01_REG_RX_ADDR_P3     0x0D // Receive address data pipe 3
#define NRF24L01_REG_RX_ADDR_P4     0x0E // Receive address data pipe 4
#define NRF24L01_REG_RX_ADDR_P5     0x0F // Receive address data pipe 5
#define NRF24L01_REG_TX_ADDR        0x10 // Transmit address
#define NRF24L01_REG_RX_PW_P0       0x11 // Number of bytes in RX payload in data pipe 0
#define NRF24L01_REG_RX_PW_P1       0x12 // Number of bytes in RX payload in data pipe 1
#define NRF24L01_REG_RX_PW_P2       0x13 // Number of bytes in RX payload in data pipe 2
#define NRF24L01_REG_RX_PW_P3       0x14 // Number of bytes in RX payload in data pipe 3
#define NRF24L01_REG_RX_PW_P4       0x15 // Number of bytes in RX payload in data pipe 4
```

```
#define NRF24L01_REG_RX_PW_P5       0x16 // Number of bytes in RX payload in data pipe 5
#define NRF24L01_REG_FIFO_STATUS    0x17 // FIFO status register
#define NRF24L01_REG_DYNPD          0x1C // Enable dynamic payload length
#define NRF24L01_REG_FEATURE        0x1D // Feature register

// Register bits definitions
#define NRF24L01_CONFIG_PRIM_RX     0x01 // PRIM_RX bit in CONFIG register
#define NRF24L01_CONFIG_PWR_UP      0x02 // PWR_UP bit in CONFIG register
#define NRF24L01_FEATURE_EN_DYN_ACK 0x01 // EN_DYN_ACK bit in FEATURE register
#define NRF24L01_FEATURE_EN_ACK_PAY 0x02 // EN_ACK_PAY bit in FEATURE register
#define NRF24L01_FEATURE_EN_DPL     0x04 // EN_DPL bit in FEATURE register
#define NRF24L01_FLAG_RX_DREADY     0x40 // RX_DR bit (data ready RX FIFO interrupt)
#define NRF24L01_FLAG_TX_DSENT      0x20 // TX_DS bit (data sent TX FIFO interrupt)
#define NRF24L01_FLAG_MAX_RT        0x10 // MAX_RT bit (maximum number of TX re-transmits
interrupt)

// Register masks definitions
#define NRF24L01_MASK_REG_MAP       0x1F // Mask bits[4:0] for CMD_RREG and CMD_WREG commands
#define NRF24L01_MASK_CRC           0x0C // Mask for CRC bits [3:2] in CONFIG register
#define NRF24L01_MASK_STATUS_IRQ    0x70 // Mask for all IRQ bits in STATUS register
#define NRF24L01_MASK_RF_PWR        0x06 // Mask RF_PWR[2:1] bits in RF_SETUP register
#define NRF24L01_MASK_RX_P_NO       0x0E // Mask RX_P_NO[3:1] bits in STATUS register
#define NRF24L01_MASK_DATARATE      0x28 // Mask RD_DR_[5,3] bits in RF_SETUP register
#define NRF24L01_MASK_EN_RX         0x3F // Mask ERX_P[5:0] bits in EN_RXADDR register
#define NRF24L01_MASK_RX_PW         0x3F // Mask [5:0] bits in RX_PW_Px register
#define NRF24L01_MASK_RETR_ARD      0xF0 // Mask for ARD[7:4] bits in SETUP_RETR register
#define NRF24L01_MASK_RETR_ARC      0x0F // Mask for ARC[3:0] bits in SETUP_RETR register
#define NRF24L01_MASK_RXFIFO        0x03 // Mask for RX FIFO status bits [1:0] in FIFO_STATUS
register
#define NRF24L01_MASK_TXFIFO        0x30 // Mask for TX FIFO status bits [5:4] in FIFO_STATUS
register
#define NRF24L01_MASK_PLOS_CNT      0xF0 // Mask for PLOS_CNT[7:4] bits in OBSERVE_TX
register
#define NRF24L01_MASK_ARC_CNT       0x0F // Mask for ARC_CNT[3:0] bits in OBSERVE_TX register

// Register masks definitions
#define NRF24L01_MASK_REG_MAP       0x1F // Mask bits[4:0] for CMD_RREG and CMD_WREG commands

#define NRF24L01_ADDR_WIDTH         5    // RX/TX address width
#define NRF24L01_PLOAD_WIDTH        32   // Payload width
```

基础方法

初始化

```
static void NRF24L01_SPI_Init()
{
  GPIO_InitTypeDef GPIO_InitStruct;

  if(NRF24L01_SPIx == SPI1) {
    // A5, A6, A7
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO,ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1,ENABLE);

    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStruct);

  } else if(NRF24L01_SPIx == SPI2) {
    // B13, B14, B15
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO,ENABLE);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2,ENABLE);
```

```
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStruct);
  }

  SPI_InitTypeDef SPI_InitStruct;
  SPI_InitStruct.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
  SPI_InitStruct.SPI_CPHA = SPI_CPHA_1Edge;
  SPI_InitStruct.SPI_CPOL = SPI_CPOL_Low;
  SPI_InitStruct.SPI_CRCPolynomial = 7;
  SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b;
  SPI_InitStruct.SPI_Direction= SPI_Direction_2Lines_FullDuplex;
  SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB;
  SPI_InitStruct.SPI_Mode = SPI_Mode_Master;
  SPI_InitStruct.SPI_NSS = SPI_NSS_Soft;

  SPI_Init(NRF24L01_SPIx, &SPI_InitStruct);

  SPI_Cmd(NRF24L01_SPIx, ENABLE);
}

void NRF24L01_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct;
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
  // CE CSN Initialize
  GPIO_InitStruct.GPIO_Pin = NRF24L01_GPIO_CE | NRF24L01_GPIO_CSN;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(NRF24L01_GPIOx, &GPIO_InitStruct);
  // IRQ Initialize
  GPIO_InitStruct.GPIO_Pin = NRF24L01_GPIO_IRQ;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_Init(NRF24L01_GPIOx, &GPIO_InitStruct);

  NRF24L01_SPI_Init();
  CSN(1);
}
```

单次读写SPI(所有交互的基础操作)

```
/**
* Basic SPI operation: Write to SPIx and read
*/
static u8 SPI_Write_Then_Read(u8 data)
{
  while(SPI_I2S_GetFlagStatus(NRF24L01_SPIx, SPI_I2S_FLAG_TXE) == RESET);
  SPI_I2S_SendData(NRF24L01_SPIx, data);

  while(SPI_I2S_GetFlagStatus(NRF24L01_SPIx, SPI_I2S_FLAG_RXNE)==RESET);
  return SPI_I2S_ReceiveData(NRF24L01_SPIx);
}
```

单个字节的读写

```
/**
* Read a 1-bit register
*/
u8 NRF24L01_Read_Reg(u8 reg)
{
  u8 value;
```

```
  CSN(0);
  SPI_Write_Then_Read(reg);
  value = SPI_Write_Then_Read(NRF24L01_CMD_NOP);
  CSN(1);
  return value;
}

/**
* Write a 1-byte register
*/
u8 NRF24L01_Write_Reg(u8 reg, u8 value)
{
  u8 status;
  CSN(0);
  if (reg < NRF24L01_CMD_REGISTER_W) {
    // This is a register access
    status = SPI_Write_Then_Read(NRF24L01_CMD_REGISTER_W | (reg & NRF24L01_MASK_REG_MAP));
    SPI_Write_Then_Read(value);

  } else {
    // This is a single byte command or future command/register
    status = SPI_Write_Then_Read(reg);
    if ((reg != NRF24L01_CMD_FLUSH_TX)
        && (reg != NRF24L01_CMD_FLUSH_RX)
        && (reg != NRF24L01_CMD_REUSE_TX_PL)
        && (reg != NRF24L01_CMD_NOP)) {
      // Send register value
      SPI_Write_Then_Read(value);
    }
  }
  CSN(1);
  return status;
}
```

多个字节的读写

```
/**
* Read a multi-byte register
*  reg  - register to read
*  buf  - pointer to the buffer to write
*  len  - number of bytes to read
*/
u8 NRF24L01_Read_To_Buf(u8 reg, u8 *buf, u8 len)
{
  CSN(0);
  u8 status = SPI_Write_Then_Read(reg);
  while (len--) {
    *buf++ = SPI_Write_Then_Read(NRF24L01_CMD_NOP);
  }
  CSN(1);
  return status;
}

/**
* Write a multi-byte register
*  reg - register to write
*  buf - pointer to the buffer with data
*  len - number of bytes to write
*/
u8 NRF24L01_Write_From_Buf(u8 reg, u8 *buf, u8 len)
{
  CSN(0);
  u8 status = SPI_Write_Then_Read(reg);
```

```
  while (len--) {
    SPI_Write_Then_Read(*buf++);
  }
  CSN(1);
  return status;
}
```

RX和TX模式配置

```
/**
* Common configurations of RX and TX, internal function
*/
void _NRF24L01_Config(u8 *tx_addr)
{
  // TX Address
  NRF24L01_Write_From_Buf(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_TX_ADDR, tx_addr,
NRF24L01_ADDR_WIDTH);
  // RX P0 Payload Width
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_RX_PW_P0, NRF24L01_PLOAD_WIDTH);
  // Enable Auto ACK
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_EN_AA, 0x3f);
  // Enable RX channels
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_EN_RXADDR, 0x3f);
  // RF channel: 2.400G  + 0.001 * x
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_RF_CH, 40);
  // 000+0+[0:1Mbps,1:2Mbps]+[00:-18dbm,01:-12dbm,10:-6dbm,11:0dbm]+[0:LNA_OFF,1:LNA_ON]
  // 01:1Mbps,-18dbm; 03:1Mbps,-12dbm; 05:1Mbps,-6dbm; 07:1Mbps,0dBm
  // 09:2Mbps,-18dbm; 0b:2Mbps,-12dbm; 0d:2Mbps,-6dbm; 0f:2Mbps,0dBm,
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_RF_SETUP, 0x03);
  // 0A:delay=250us,count=10, 1A:delay=500us,count=10
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_SETUP_RETR, 0x0a);
}

/**
* Switch NRF24L01 to RX mode
*/
void NRF24L01_RX_Mode(u8 *rx_addr, u8 *tx_addr)
{
  CE(0);
  _NRF24L01_Config(tx_addr);
  // RX Address of P0
  NRF24L01_Write_From_Buf(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_RX_ADDR_P0, rx_addr,
NRF24L01_ADDR_WIDTH);
  /**
  REG 0x00:
  0)PRIM_RX     0:TX            1:RX
  1)PWR_UP      0:OFF           1:ON
  2)CRCO        0:8bit CRC      1:16bit CRC
  3)EN_CRC      Enabled if any of EN_AA is high
  4)MASK_MAX_RT 0:IRQ low       1:NO IRQ
  5)MASK_TX_DS  0:IRQ low       1:NO IRQ
  6)MASK_RX_DR  0:IRQ low       1:NO IRQ
  7)Reserved    0
  */
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_CONFIG, 0x0f);
//RX,PWR_UP,CRC16,EN_CRC
  CE(1);
}

/**
* Switch NRF24L01 to TX mode
*/
void NRF24L01_TX_Mode(u8 *rx_addr, u8 *tx_addr)
```

```
{
  CE(0);
  _NRF24L01_Config(tx_addr);
  // On the PTX the **TX_ADDR** must be the same as the **RX_ADDR_P0** and as the pipe
address for the designated pipe
  // RX_ADDR_P0 will be used for receiving ACK
  NRF24L01_Write_From_Buf(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_RX_ADDR_P0, tx_addr,
NRF24L01_ADDR_WIDTH);
  NRF24L01_Write_Reg(NRF24L01_CMD_REGISTER_W + NRF24L01_REG_CONFIG, 0x0e);
//TX,PWR_UP,CRC16,EN_CRC
  CE(1);
}
```

接收和发送操作

```
/**
* Hold till data received and written to rx_buf
*/
u8 NRF24L01_RxPacket(u8 *rx_buf)
{
  u8 status, result = 0;
  while(IRQ);
  CE(0);
  status = NRF24L01_Read_Reg(NRF24L01_REG_STATUS);
  printf("Interrupted, status: %02X\r\n", status);

  if(status & NRF24L01_FLAG_RX_DREADY) {
    NRF24L01_Read_To_Buf(NRF24L01_CMD_RX_PLOAD_R, rx_buf, NRF24L01_PLOAD_WIDTH);
    for (int i = 0; i < 32; i++) {
      printf("%02X ", RX_BUF[i]);
    }
    result = 1;
    NRF24L01_ClearIRQFlag(NRF24L01_FLAG_RX_DREADY);
  }
  CE(1);
  return result;
}

/**
* Send data in tx_buf and wait till data is sent or max re-tr reached
*/
u8 NRF24L01_TxPacket(u8 *tx_buf, u8 len)
{
  u8 status = 0x00;
  CE(0);
  len = len > NRF24L01_PLOAD_WIDTH? NRF24L01_PLOAD_WIDTH : len;
  NRF24L01_Write_From_Buf(NRF24L01_CMD_TX_PLOAD_W, tx_buf, len);
  CE(1);
  while(IRQ != 0); // Waiting send finish

  CE(0);
  status = NRF24L01_Read_Reg(NRF24L01_REG_STATUS);
  printf("Interrupted, status: %02X\r\n", status);
  if(status & NRF24L01_FLAG_TX_DSENT) {
    printf("Data sent: ");
    for (u8 i = 0; i < len; i++) {
      printf("%02X ", tx_buf[i]);
    }
    printf("\r\n");
    NRF24L01_ClearIRQFlag(NRF24L01_FLAG_TX_DSENT);

  } else if(status & NRF24L01_FLAG_MAX_RT) {
    printf("Sending exceeds max retries\r\n");
```

```
    NRF24L01_FlushTX();
    NRF24L01_ClearIRQFlag(NRF24L01_FLAG_MAX_RT);
  }
  CE(1);
  return status;
}
```

如果需要使用中断读取, 读方法要改成非阻塞的方式, 就是上面的RX方法去掉了while IRQ的等待.

```
/**
* Read received data and written to rx_buf, No blocking.
*/
u8 NRF24L01_IRQ_Handler(u8 *rx_buf)
{
  u8 status, result = 0;
  CE(0);
  status = NRF24L01_Read_Reg(NRF24L01_REG_STATUS);
  printf("Reg status: %02X\r\n", status);
  if(status & NRF24L01_FLAG_RX_DREADY) {
    NRF24L01_Read_To_Buf(NRF24L01_CMD_RX_PLOAD_R, rx_buf, NRF24L01_PLOAD_WIDTH);
    for (int i = 0; i < 32; i++) {
      printf("%02X ", RX_BUF[i]);
    }
    result = 1;
    NRF24L01_FlushRX();
    NRF24L01_ClearIRQFlag(NRF24L01_FLAG_RX_DREADY);

  } else if(status & NRF24L01_FLAG_TX_DSENT) {
    printf("Data sent\r\n");
    NRF24L01_FlushTX();
    NRF24L01_ClearIRQFlag(NRF24L01_FLAG_TX_DSENT);
  } else if(status & NRF24L01_FLAG_MAX_RT) {
    printf("Sending exceeds max retries\r\n");
    NRF24L01_FlushTX();
    NRF24L01_ClearIRQFlag(NRF24L01_FLAG_MAX_RT);
  }
  CE(1);
  return result;
}
```

一个非常好用的配置打印函数

```
/**
* Dump nRF24L01 configuration
*/
void NRF24L01_DumpConfig(void) {
  uint8_t i,j;
  uint8_t aw;
  uint8_t buf[5];

  // CONFIG
  i = NRF24L01_Read_Reg(NRF24L01_REG_CONFIG);
  printf("[0x%02X] 0x%02X MASK:%02X CRC:%02X PWR:%s MODE:P%s\r\n",
      NRF24L01_REG_CONFIG,
      i,
      i >> 4,
      (i & 0x0c) >> 2,
      (i & 0x02) ? "ON" : "OFF",
      (i & 0x01) ? "RX" : "TX"
    );
  // EN_AA
  i = NRF24L01_Read_Reg(NRF24L01_REG_EN_AA);
  printf("[0x%02X] 0x%02X ENAA: ",NRF24L01_REG_EN_AA,i);
```

```
    for (j = 0; j < 6; j++) {
      printf("[P%1u%s]%s",j,
          (i & (1 << j)) ? "+" : "-",
          (j == 5) ? "\r\n" : " "
        );
    }
    // EN_RXADDR
    i = NRF24L01_Read_Reg(NRF24L01_REG_EN_RXADDR);
    printf("[0x%02X] 0x%02X EN_RXADDR: ",NRF24L01_REG_EN_RXADDR,i);
    for (j = 0; j < 6; j++) {
      printf("[P%1u%s]%s",j,
          (i & (1 << j)) ? "+" : "-",
          (j == 5) ? "\r\n" : " "
        );
    }
    // SETUP_AW
    i = NRF24L01_Read_Reg(NRF24L01_REG_SETUP_AW);
    aw = (i & 0x03) + 2;
    printf("[0x%02X] 0x%02X EN_RXADDR=%03X (address width = %u)\r\n",NRF24L01_REG_SETUP_AW,i,i
  & 0x03,aw);
    // SETUP_RETR
    i = NRF24L01_Read_Reg(NRF24L01_REG_SETUP_RETR);
    printf("[0x%02X] 0x%02X ARD=%04X ARC=%04X (retr.delay=%uus, count=%u)\r\n",
        NRF24L01_REG_SETUP_RETR,
        i,
        i >> 4,
        i & 0x0F,
        ((i >> 4) * 250) + 250,
        i & 0x0F
      );
    // RF_CH
    i = NRF24L01_Read_Reg(NRF24L01_REG_RF_CH);
    printf("[0x%02X] 0x%02X (%.3uGHz)\r\n",NRF24L01_REG_RF_CH,i,2400 + i);
    // RF_SETUP
    i = NRF24L01_Read_Reg(NRF24L01_REG_RF_SETUP);
    printf("[0x%02X] 0x%02X CONT_WAVE:%s PLL_LOCK:%s DataRate=",
        NRF24L01_REG_RF_SETUP,
        i,
        (i & 0x80) ? "ON" : "OFF",
        (i & 0x80) ? "ON" : "OFF"
      );
    switch ((i & 0x28) >> 3) {
      case 0x00:
        printf("1M");
        break;
      case 0x01:
        printf("2M");
        break;
      case 0x04:
        printf("250k");
        break;
      default:
        printf("???");
        break;
    }
    printf("pbs RF_PWR=");
    switch ((i & 0x06) >> 1) {
      case 0x00:
        printf("-18");
        break;
      case 0x01:
        printf("-12");
        break;
      case 0x02:
```

```
      printf("-6");
      break;
    case 0x03:
      printf("0");
      break;
    default:
      printf("???");
      break;
  }
  printf("dBm\r\n");
  // STATUS
  i = NRF24L01_Read_Reg(NRF24L01_REG_STATUS);
  printf("[0x%02X] 0x%02X IRQ:%03X RX_PIPE:%u TX_FULL:%s\r\n",
      NRF24L01_REG_STATUS,
      i,
      (i & 0x70) >> 4,
      (i & 0x0E) >> 1,
      (i & 0x01) ? "YES" : "NO"
    );

  // OBSERVE_TX
  i = NRF24L01_Read_Reg(NRF24L01_REG_OBSERVE_TX);
  printf("[0x%02X] 0x%02X PLOS_CNT=%u ARC_CNT=%u\r\n",NRF24L01_REG_OBSERVE_TX,i,i >> 4,i &
0x0F);

  // RPD
  i = NRF24L01_Read_Reg(NRF24L01_REG_RPD);
  printf("[0x%02X] 0x%02X RPD=%s\r\n",NRF24L01_REG_RPD,i,(i & 0x01) ? "YES" : "NO");

  // RX_ADDR_P0
  NRF24L01_Read_To_Buf(NRF24L01_REG_RX_ADDR_P0,buf,aw);
  printf("[0x%02X] RX_ADDR_P0 \"",NRF24L01_REG_RX_ADDR_P0);
  for (i = 0; i < aw; i++) printf("%X ",buf[i]);
  printf("\"\r\n");

  // RX_ADDR_P1
  NRF24L01_Read_To_Buf(NRF24L01_REG_RX_ADDR_P1,buf,aw);
  printf("[0x%02X] RX_ADDR_P1 \"",NRF24L01_REG_RX_ADDR_P1);
  for (i = 0; i < aw; i++) printf("%X ",buf[i]);
  printf("\"\r\n");

  // RX_ADDR_P2
  printf("[0x%02X] RX_ADDR_P2 \"",NRF24L01_REG_RX_ADDR_P2);
  for (i = 0; i < aw - 1; i++) printf("%X ",buf[i]);
  i = NRF24L01_Read_Reg(NRF24L01_REG_RX_ADDR_P2);
  printf("%X\"\r\n",i);

  // RX_ADDR_P3
  printf("[0x%02X] RX_ADDR_P3 \"",NRF24L01_REG_RX_ADDR_P3);
  for (i = 0; i < aw - 1; i++) printf("%X ",buf[i]);
  i = NRF24L01_Read_Reg(NRF24L01_REG_RX_ADDR_P3);
  printf("%X\"\r\n",i);

  // RX_ADDR_P4
  printf("[0x%02X] RX_ADDR_P4 \"",NRF24L01_REG_RX_ADDR_P4);
  for (i = 0; i < aw - 1; i++) printf("%X ",buf[i]);
  i = NRF24L01_Read_Reg(NRF24L01_REG_RX_ADDR_P4);
  printf("%X\"\r\n",i);

  // RX_ADDR_P5
  printf("[0x%02X] RX_ADDR_P5 \"",NRF24L01_REG_RX_ADDR_P5);
  for (i = 0; i < aw - 1; i++) printf("%X ",buf[i]);
  i = NRF24L01_Read_Reg(NRF24L01_REG_RX_ADDR_P5);
  printf("%X\"\r\n",i);
```

```
    // TX_ADDR
    NRF24L01_Read_To_Buf(NRF24L01_REG_TX_ADDR,buf,aw);
    printf("[0x%02X] TX_ADDR \"",NRF24L01_REG_TX_ADDR);
    for (i = 0; i < aw; i++) printf("%X ",buf[i]);
    printf("\"\r\n");

    // RX_PW_P0
    i = NRF24L01_Read_Reg(NRF24L01_REG_RX_PW_P0);
    printf("[0x%02X] RX_PW_P0=%u\r\n",NRF24L01_REG_RX_PW_P0,i);

    // RX_PW_P1
    i = NRF24L01_Read_Reg(NRF24L01_REG_RX_PW_P1);
    printf("[0x%02X] RX_PW_P1=%u\r\n",NRF24L01_REG_RX_PW_P1,i);

    // RX_PW_P2
    i = NRF24L01_Read_Reg(NRF24L01_REG_RX_PW_P2);
    printf("[0x%02X] RX_PW_P2=%u\r\n",NRF24L01_REG_RX_PW_P2,i);

    // RX_PW_P3
    i = NRF24L01_Read_Reg(NRF24L01_REG_RX_PW_P3);
    printf("[0x%02X] RX_PW_P3=%u\r\n",NRF24L01_REG_RX_PW_P3,i);

    // RX_PW_P4
    i = NRF24L01_Read_Reg(NRF24L01_REG_RX_PW_P4);
    printf("[0x%02X] RX_PW_P4=%u\r\n",NRF24L01_REG_RX_PW_P4,i);

    // RX_PW_P5
    i = NRF24L01_Read_Reg(NRF24L01_REG_RX_PW_P5);
    printf("[0x%02X] RX_PW_P5=%u\r\n",NRF24L01_REG_RX_PW_P5,i);
}
```

STM32F401CCU6

STM32F4可用的SPI口

	SPI1		SPI2		SPI3		SPI4	
NSS	PA4	PA15	PB12	PB9	PA15	PA4	PE4	PE11
SCK	PA5	PB3	PB13	PB10	PC10	PB3	PE2	PE12
MISO	PA6	PB4	PB14	PC2	PC11	PB4	PE5	PE13
MOSI	PA7	PB5	PB15	PC3	PC12	PB5	PE6	PE14

因此连接方式与STM32F103完全相同

STM32	nRF24L01
PA4 SPI1_NSS	N/A
PA5 SPI1_SCK	SCK
PA6 SPI1_MISO	MISO
PA7 SPI1_MOSI	MOSI
PB13	IRQ
PB14	CE

STM32	nRF24L01
PB15	CSN

唯一区别是初始化方式

```
static void NRF24L01_SPI_Init()
{
  GPIO_InitTypeDef GPIO_InitStructure;

  if(NRF24L01_SPIx == SPI1) {
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

    // SCK:PA5, MISO:PA6, MOSI:PA7 or PB3, PB4, PB5
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_PinAFConfig(GPIOA, GPIO_PinSource5, GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_SPI1);

    RCC_APB2PeriphResetCmd(RCC_APB2Periph_SPI1, ENABLE); // reset SPI1
    RCC_APB2PeriphResetCmd(RCC_APB2Periph_SPI1, DISABLE);// stop reset SPI1

  } else if(NRF24L01_SPIx == SPI2) {
    // B13, B14, B15
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);

  } else { // SPI3,4,5,6
    RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI3, ENABLE);

  }

  SPI_InitTypeDef SPI_InitStructure;
  SPI_StructInit(&SPI_InitStructure); // set default settings
  SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
  SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge; // data sampled at first edge
  SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;    // clock is low when idle
  SPI_InitStructure.SPI_CRCPolynomial = 7;
  SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; // one packet of data is 8 bits wide
  SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; // set to full duplex
mode, seperate MOSI and MISO lines
  SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; // data is transmitted MSB first
        SPI_InitStructure.SPI_Mode = SPI_Mode_Master; // transmit in master mode, NSS pin has
to be always high
        SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; // set the NSS management to internal and
pull internal NSS high
        SPI_Init(NRF24L01_SPIx, &SPI_InitStructure);
        SPI_Cmd(NRF24L01_SPIx, ENABLE);

}

void NRF24L01_Init(void)
{
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

  GPIO_InitTypeDef GPIO_InitStructure;
```

```
    GPIO_InitStructure.GPIO_Pin = NRF24L01_GPIO_CE|NRF24L01_GPIO_CSN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(NRF24L01_GPIOx, &GPIO_InitStructure);
    // IRQ Initialize
    GPIO_InitStructure.GPIO_Pin = NRF24L01_GPIO_IRQ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
    GPIO_Init(NRF24L01_GPIOx, &GPIO_InitStructure);

    NRF24L01_SPI_Init();
    /*
      CSN is high initially (active low).
      CE is low initially (active high).
    */
    CSN(1);
    printf("## nRF24L01 Initialized ##\r\n");
}
```

在STM32F401CCU6上, 使用中断进行接收的例子

```
void EXTILine13_Config(void)
{
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
    /* Enable SYSCFG clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

    GPIO_InitTypeDef   GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_13;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* Connect EXTI Line13 to PG13 pin */
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource13);

    EXTI_InitTypeDef   EXTI_InitStructure;
    EXTI_InitStructure.EXTI_Line    = EXTI_Line13;
    EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    NVIC_InitTypeDef   NVIC_InitStructure;
    NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x01;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x01;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

void EXTI15_10_IRQHandler(void) {
    printf("EXTI15_10_IRQHandler\r\n");
    /* Make sure that interrupt flag is set */
    if (EXTI_GetITStatus(EXTI_Line13) != RESET) {
      NRF24L01_IRQ_Handler(RX_BUF);
      /* Clear interrupt flag */
      EXTI_ClearITPendingBit(EXTI_Line13);
    }
}

int main(void)
```

```
{
  Systick_Init();
  USART1_Init();
  NRF24L01_Init();
  NRF24L01_DumpConfig();
  while(NRF24L01_Check() != 0) {
    printf("nRF24L01 check failed\r\n");
    Systick_Delay_ms(2000);
  }
  printf("nRF24L01 check succeeded\r\n");

  printf("nRF24L01 in RECEIVE mode\r\n");
  NRF24L01_RX_Mode(RX_ADDRESS, TX_ADDRESS);

  LED_Init();
  EXTILine13_Config();

  while(1) {}
}
```

STC89C52

51单片机的连接方式

```
sbit CE =  P1^5;
sbit CSN=  P1^4;
sbit SCK=  P1^3;
sbit MOSI= P1^2;
sbit MISO= P1^1;
sbit IRQ = P1^0;
```

常量定义

```
// SPI(nRF24L01) commands
#define READ_REG    0x00  // Define read command to register
#define WRITE_REG   0x20  // Define write command to register
#define RD_RX_PLOAD 0x61  // Define RX payload register address
#define WR_TX_PLOAD 0xA0  // Define TX payload register address
#define FLUSH_TX    0xE1  // Define flush TX register command
#define FLUSH_RX    0xE2  // Define flush RX register command
#define REUSE_TX_PL 0xE3  // Define reuse TX payload register command
#define NOP         0xFF  // Define No Operation, might be used to read status register

// SPI(nRF24L01) registers(addresses)
#define CONFIG      0x00  // 'Config' register address
#define EN_AA       0x01  // 'Enable Auto Acknowledgment' register address
#define EN_RXADDR   0x02  // 'Enabled RX addresses' register address
#define SETUP_AW    0x03  // 'Setup address width' register address
#define SETUP_RETR  0x04  // 'Setup Auto. Retrans' register address
#define RF_CH       0x05  // 'RF channel' register address
#define RF_SETUP    0x06  // 'RF setup' register address
#define STATUS      0x07  // 'Status' register address
#define OBSERVE_TX  0x08  // 'Observe TX' register address
#define CD          0x09  // 'Carrier Detect' register address
#define RX_ADDR_P0  0x0A  // 'RX address pipe0' register address
#define RX_ADDR_P1  0x0B  // 'RX address pipe1' register address
#define RX_ADDR_P2  0x0C  // 'RX address pipe2' register address
#define RX_ADDR_P3  0x0D  // 'RX address pipe3' register address
#define RX_ADDR_P4  0x0E  // 'RX address pipe4' register address
#define RX_ADDR_P5  0x0F  // 'RX address pipe5' register address
#define TX_ADDR     0x10  // 'TX address' register address
#define RX_PW_P0    0x11  // 'RX payload width, pipe0' register address
```

```
#define RX_PW_P1    0x12  // 'RX payload width, pipe1' register address
#define RX_PW_P2    0x13  // 'RX payload width, pipe2' register address
#define RX_PW_P3    0x14  // 'RX payload width, pipe3' register address
#define RX_PW_P4    0x15  // 'RX payload width, pipe4' register address
#define RX_PW_P5    0x16  // 'RX payload width, pipe5' register address
#define FIFO_STATUS 0x17  // 'FIFO Status Register' register address
```

基础方法

```
void init_io(void)
{
  CE  = 0;        // 待机
  CSN = 1;        // SPI禁止
  SCK = 0;        // SPI时钟置低
  IRQ = 1;        // 中断复位
  LED = 1;        // 关闭指示灯
}

void delay_ms(uchar x)
{
  uchar i, j;
  i = 0;
  for(i=0; i<x; i++) {
    j = 250;
    while(--j);
    j = 250;
    while(--j);
  }
}

uchar SPI_RW(uchar byte)
{
  uchar i;
  for(i=0; i<8; i++) {
    MOSI = (byte & 0x80);   // byte最高位输出到MOSI
    byte <<= 1;             // 左移一位
    SCK = 1;                // 拉高SCK，nRF24L01从MOSI读入1位数据，同时从MISO输出1位数据
    byte |= MISO;           // 读MISO到byte最低位
    SCK = 0;                // SCK置低
  }
  return byte;
}

uchar SPI_RW_Reg(uchar reg, uchar value)
{
  uchar status;
  CSN = 0;                   // CSN置低，开始传输数据
  status = SPI_RW(reg);      // 选择寄存器，同时返回状态字
  SPI_RW(value);             // 然后写数据到该寄存器
  CSN = 1;                   // CSN拉高，结束数据传输
  return(status);            // 返回状态寄存器
}

uchar SPI_Read(uchar reg)
{
  uchar reg_val;
  CSN = 0;                    // CSN置低，开始传输数据
  SPI_RW(reg);                // 选择寄存器
  reg_val = SPI_RW(0);        // 然后从该寄存器读数据
  CSN = 1;                    // CSN拉高，结束数据传输
  return(reg_val);            // 返回寄存器数据
}
```

```
uchar SPI_Read_Buf(uchar reg, uchar * pBuf, uchar bytes)
{
  uchar status, i;
  CSN = 0;                    // CSN置低，开始传输数据
  status = SPI_RW(reg);       // 选择寄存器，同时返回状态字
  for(i=0; i<bytes; i++)
    pBuf[i] = SPI_RW(0);      // 逐个字节从nRF24L01读出
  CSN = 1;                    // CSN拉高，结束数据传输
  return(status);             // 返回状态寄存器
}

uchar SPI_Write_Buf(uchar reg, uchar * pBuf, uchar bytes)
{
  uchar status, i;
  CSN = 0;                    // CSN置低，开始传输数据
  status = SPI_RW(reg);       // 选择寄存器，同时返回状态字
  for(i=0; i<bytes; i++)
    SPI_RW(pBuf[i]);          // 逐个字节写入nRF24L01
  CSN = 1;                    // CSN拉高，结束数据传输
  return(status);             // 返回状态寄存器
}
```

STC12C5A60S2

STC12系列带了硬件SPI, 在代码上与89C52有区别.

连线

```
sbit NRF_CE   = P3^7;
sbit NRF_CSN  = P1^4;
sbit NRF_MISO = P1^6;
sbit NRF_MOSI = P1^5;
sbit NRF_SCK  = P1^7;
sbit NRF_IRQ  = P3^2;
```

代码

```
#include "STC12C5A60S2.H"

#define uchar unsigned char
#define uint  unsigned int

/**********  NRF24L01寄存器操作命令  ***********/
#define READ_REG        0x00  //读配置寄存器,低5位为寄存器地址
#define WRITE_REG       0x20  //写配置寄存器,低5位为寄存器地址
#define RD_RX_PLOAD     0x61  //读RX有效数据,1~32字节
#define WR_TX_PLOAD     0xA0  //写TX有效数据,1~32字节
#define FLUSH_TX        0xE1  //清除TX FIFO寄存器.发射模式下用
#define FLUSH_RX        0xE2  //清除RX FIFO寄存器.接收模式下用
#define REUSE_TX_PL     0xE3  //重新使用上一包数据,CE为高,数据包被不断发送.
#define NOP             0xFF  //空操作,可以用来读状态寄存器
/**********  NRF24L01寄存器地址   *************/
#define CONFIG          0x00  //配置寄存器地址
#define EN_AA           0x01  //使能自动应答功能
#define EN_RXADDR       0x02  //接收地址允许
#define SETUP_AW        0x03  //设置地址宽度(所有数据通道)
#define SETUP_RETR      0x04  //建立自动重发
#define RF_CH           0x05  //RF通道
#define RF_SETUP        0x06  //RF寄存器
#define STATUS          0x07  //状态寄存器
```

```
#define OBSERVE_TX      0x08  // 发送检测寄存器
#define CD              0x09  // 载波检测寄存器
#define RX_ADDR_P0      0x0A  // 数据通道0接收地址
#define RX_ADDR_P1      0x0B  // 数据通道1接收地址
#define RX_ADDR_P2      0x0C  // 数据通道2接收地址
#define RX_ADDR_P3      0x0D  // 数据通道3接收地址
#define RX_ADDR_P4      0x0E  // 数据通道4接收地址
#define RX_ADDR_P5      0x0F  // 数据通道5接收地址
#define TX_ADDR         0x10  // 发送地址寄存器
#define RX_PW_P0        0x11  // 接收数据通道0有效数据宽度(1~32字节)
#define RX_PW_P1        0x12  // 接收数据通道1有效数据宽度(1~32字节)
#define RX_PW_P2        0x13  // 接收数据通道2有效数据宽度(1~32字节)
#define RX_PW_P3        0x14  // 接收数据通道3有效数据宽度(1~32字节)
#define RX_PW_P4        0x15  // 接收数据通道4有效数据宽度(1~32字节)
#define RX_PW_P5        0x16  // 接收数据通道5有效数据宽度(1~32字节)
#define FIFO_STATUS     0x17  // FIFO状态寄存器

/*********  STATUS寄存器bit位定义      *******/
#define MAX_TX    0x10  //达到最大发送次数中断
#define TX_OK     0x20  //TX发送完成中断
#define RX_OK     0x40  //接收到数据中断

/*********  24L01发送接收数据宽度定义    ***********/
#define TX_ADR_WIDTH    5   //5字节地址宽度
#define RX_ADR_WIDTH    5   //5字节地址宽度
#define TX_PLOAD_WIDTH  32  //32字节有效数据宽度
#define RX_PLOAD_WIDTH  32  //32字节有效数据宽度


const uchar TX_ADDRESS[TX_ADR_WIDTH]={0x68,0x86,0x66,0x88,0x28};
const uchar RX_ADDRESS[RX_ADR_WIDTH]={0x68,0x86,0x66,0x88,0x28};

sbit NRF_CE = P3^7;
sbit NRF_CSN = P1^4;
sbit NRF_MISO = P1^6;
sbit NRF_MOSI = P1^5;
sbit NRF_SCK = P1^7;
sbit NRF_IRQ = P3^2;

unsigned char rece_buf[32];

void SPI_Init(void)
{
  SPSTAT |= 0XC0;
  SPCTL = 0XD0;
}

uchar SPI_RW(uchar tr_data)
{
  uchar i=0;

  SPSTAT |= 0Xc0; // 清高两位,
  SPDAT=tr_data;
  while(((SPSTAT&0X80)!=0X80)&&(i<20))
  {
    i++;
    delay_ms(1);
  }
  return SPDAT;
}

uchar NRF24L01_Write_Reg(uchar reg,uchar value)
{
  uchar status;
```

```
  NRF_CSN=0;                //CSN=0;
    status = SPI_RW(reg);   //发送寄存器地址,并读取状态值
  SPI_RW(value);
  NRF_CSN=1;                //CSN=1;

  return status;
}

uchar NRF24L01_Read_Reg(uchar reg)
{
  uchar value;

  NRF_CSN=0;               //CSN=0;
    SPI_RW(reg);           //发送寄存器值(位置),并读取状态值
  value = SPI_RW(NOP);
  NRF_CSN=1;               //CSN=1;

  return value;
}

uchar NRF24L01_Read_Buf(uchar reg,uchar *pBuf,uchar len)
{
  uchar status,u8_ctr;
  NRF_CSN=0;                    //CSN=0
    status=SPI_RW(reg);//发送寄存器地址,并读取状态值
  for(u8_ctr=0;u8_ctr<len;u8_ctr++)
  pBuf[u8_ctr]=SPI_RW(0XFF);//读出数据
  NRF_CSN=1;                  //CSN=1
    return status;         //返回读到的状态值
}

uchar NRF24L01_Write_Buf(uchar reg, uchar *pBuf, uchar len)
{
  uchar status,u8_ctr;
  NRF_CSN=0;
    status = SPI_RW(reg);//发送寄存器值(位置),并读取状态值
    for(u8_ctr=0; u8_ctr<len; u8_ctr++)
  SPI_RW(*pBuf++); //写入数据
  NRF_CSN=1;
    return status;           //返回读到的状态值
}

uchar NRF24L01_RxPacket(uchar *rxbuf)
{
  uchar state;

  state=NRF24L01_Read_Reg(STATUS);  //读取状态寄存器的值
  NRF24L01_Write_Reg(WRITE_REG+STATUS,state); //清除TX_DS或MAX_RT中断标志
  if(state&RX_OK)//接收到数据
  {
    NRF24L01_Read_Buf(RD_RX_PLOAD,rxbuf,RX_PLOAD_WIDTH);//读取数据
    NRF24L01_Write_Reg(FLUSH_RX,0xff);//清除RX FIFO寄存器
    return 0;
  }
  return 1;//没收到任何数据
}

uchar NRF24L01_TxPacket(uchar *txbuf)
{
  uchar state;
  NRF_CE=0;//CE拉低, 使能24L01配置
  NRF24L01_Write_Buf(WR_TX_PLOAD,txbuf,TX_PLOAD_WIDTH);//写数据到TX BUF  32个字节
  NRF_CE=1;//CE置高, 使能发送
```

```
  while(NRF_IRQ==1);//等待发送完成
  state=NRF24L01_Read_Reg(STATUS);  //读取状态寄存器的值
  NRF24L01_Write_Reg(WRITE_REG+STATUS,state); //清除TX_DS或MAX_RT中断标志
  if(state&MAX_TX)//达到最大重发次数
  {
    NRF24L01_Write_Reg(FLUSH_TX,0xff);//清除TX FIFO寄存器
    return MAX_TX;
  }
  if(state&TX_OK)//发送完成
  {
    return TX_OK;
  }
  return 0xff;//发送失败
}

uchar NRF24L01_Check(void)
{
  uchar check_in_buf[5]={0x11,0x22,0x33,0x44,0x55};
  uchar check_out_buf[5]={0x00};

  NRF_CE=0;

  NRF24L01_Write_Buf(WRITE_REG+TX_ADDR, check_in_buf, 5);

  NRF24L01_Read_Buf(READ_REG+TX_ADDR, check_out_buf, 5);

  if((check_out_buf[0] == 0x11)&&\
     (check_out_buf[1] == 0x22)&&\
     (check_out_buf[2] == 0x33)&&\
     (check_out_buf[3] == 0x44)&&\
     (check_out_buf[4] == 0x55))return 0;
  else return 1;
}

void NRF24L01_RT_Init(void)
{
  NRF_CE=0;
  NRF24L01_Write_Reg(WRITE_REG+RX_PW_P0,RX_PLOAD_WIDTH);//选择通道0的有效数据宽度
  NRF24L01_Write_Reg(FLUSH_RX,0xff);//清除RX FIFO寄存器
  NRF24L01_Write_Buf(WRITE_REG+TX_ADDR,(uchar*)TX_ADDRESS,TX_ADR_WIDTH);//写TX节点地址
  NRF24L01_Write_Buf(WRITE_REG+RX_ADDR_P0,(uchar*)RX_ADDRESS,RX_ADR_WIDTH); //设置TX节点地址,主
要为了使能ACK
  NRF24L01_Write_Reg(WRITE_REG+EN_AA,0x01);     //使能通道0的自动应答
  NRF24L01_Write_Reg(WRITE_REG+EN_RXADDR,0x01); //使能通道0的接收地址
  NRF24L01_Write_Reg(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
  NRF24L01_Write_Reg(WRITE_REG+RF_CH,40);       //设置RF通道为125
  NRF24L01_Write_Reg(WRITE_REG+RF_SETUP,0x27);  //7db增益,250kbps
  NRF24L01_Write_Reg(WRITE_REG+CONFIG,0x0E);    //配置基本工作模式的参
数;PWR_UP,EN_CRC,16BIT_CRC,发送模式,开启所有中断
  NRF_CE=1; //CE置高
}

void main(void)
{
  delay_ms(100); // 延时待系统稳定
  SPI_Init(); // 初始化SPI口
  while(NRF24L01_Check()); // 等待检测到NRF24L01，程序才会向下执行
  NRF24L01_RT_Init(); // 配置NRF24L01为接收模式
  rece_buf[0]=1;
  rece_buf[1]=0x88;

  while (1)
  {
```

```
    delay_ms(500);
    NRF24L01_TxPacket(rece_buf);         // 无线发送数据
  }
}
```

参考资料

- 百度文库上一个比较全的资料(51 MCU的)
https://wenku.baidu.com/view/45d4ba90dd88d0d233d46a41.html
- https://blog.csdn.net/qq_38405680/article/details/80456164
1，电源必须在它的电压符合范围之内，不能接5V，会烧掉，某宝有1117稳压模块比较好。
2，供电电源波纹必须在80mv以内，就是波动不能超过0.08V，某宝有1117稳压模块，并联一个100UF的点解电容和105的瓷片电容。
3，NRF的IRQ脚会坏，表现为发送端正常发送，接受端无法接受到信号，接收端IRQ电平恒高。
4，使用洞洞板时，切记杜邦线会影响NRF之间的通信，如果想要最佳的通信，用铜柱将模块放到高的位置，并且铜柱接地，用金属网包裹整个电路（除NRF外）并接地。
5，旁边不能有强干扰，例如手钻，电钻，切割机之类的。
- 这是一个综合的介绍 https://www.playembedded.org/blog/a-radio-frequency-transceiver-library-nrf24l01-and-chibiosrt/
- SPI初始化和方法, 与目录中的 `NRF24L01_STM32_code` 和 `STM32F103-Example` 结合着看, 模块部分都是正点原子的代码
https://blog.csdn.net/weixin_45555616/article/details/110501179
- 另一篇带源码的说明
https://www.cnblogs.com/whlook/p/5967156.html
- 这篇对24L01的工作机制有描述
https://www.cnblogs.com/mr-bike/p/3520141.html
- https://github.com/elmot/nrf24l01-lib/blob/master/nrf24l01/nrf24.h
- 用外部中断处理IRQ
https://github.com/r2aiv/NRF24L01-1/blob/master/inc/nrf24l01.h
https://stackoverflow.com/questions/25932299/stm32-rising-and-falling-button-interrupt-detection
- 8051操作nRF24L01 https://blog.csdn.net/fzf1996/article/details/90601375
- 8051 https://blog.csdn.net/wzk456/article/details/78948122
- 8051 http://www.51hei.com/bbs/dpj-92127-1.html
- 8051 http://www.51hei.com/bbs/dpj-135749-1.html
- STM32F4的库文件 https://github.com/MaJerle/stm32f429/blob/master/00-STM32F429_LIBRARIES/tm_stm32f4_nrf24l01.c
- https://stm32f4-discovery.net/2014/06/library-17-nrf24l01-stm32f4xx/
- https://github.com/knielsen/stm32f4_wireless_bootloader/blob/master/wireless_bootloader.c
- https://github.com/AmberHan/STM32F4_Send/blob/main/终端设备A1/HARDWARE/NRF24L01/24l01.c