NRF24L01+ 模块系列

说

明

书

一、模块介绍

(1) 2.4Ghz 全球开放 ISM 频段免许可证使用

(2) 最高工作速率 2Mbps，高效 GFSK 调制，抗干扰能力强，特别适合工业控制场合

(3) 126 频道，满足多点通信和跳频通信需要

(4) 内置硬件 CRC 检错和点对多点通信地址控制

(5) 低功耗 1.9 - 3.6V 工作，待机模式下状态为 22uA；掉电模式下为 900nA

(6) 内置 2.4Ghz 天线，体积种类多样

(7) 模块可软件设地址，只有收到本机地址时才会输出数据（提供中断指示)，可直接接各种单片机使用，软件编程非常方便

(8) 内置专门稳压电路，使用各种电源包括 DC/DC 开关电源均有很好的通信效果

(9) 2.54MM 间距接口，DIP 封装

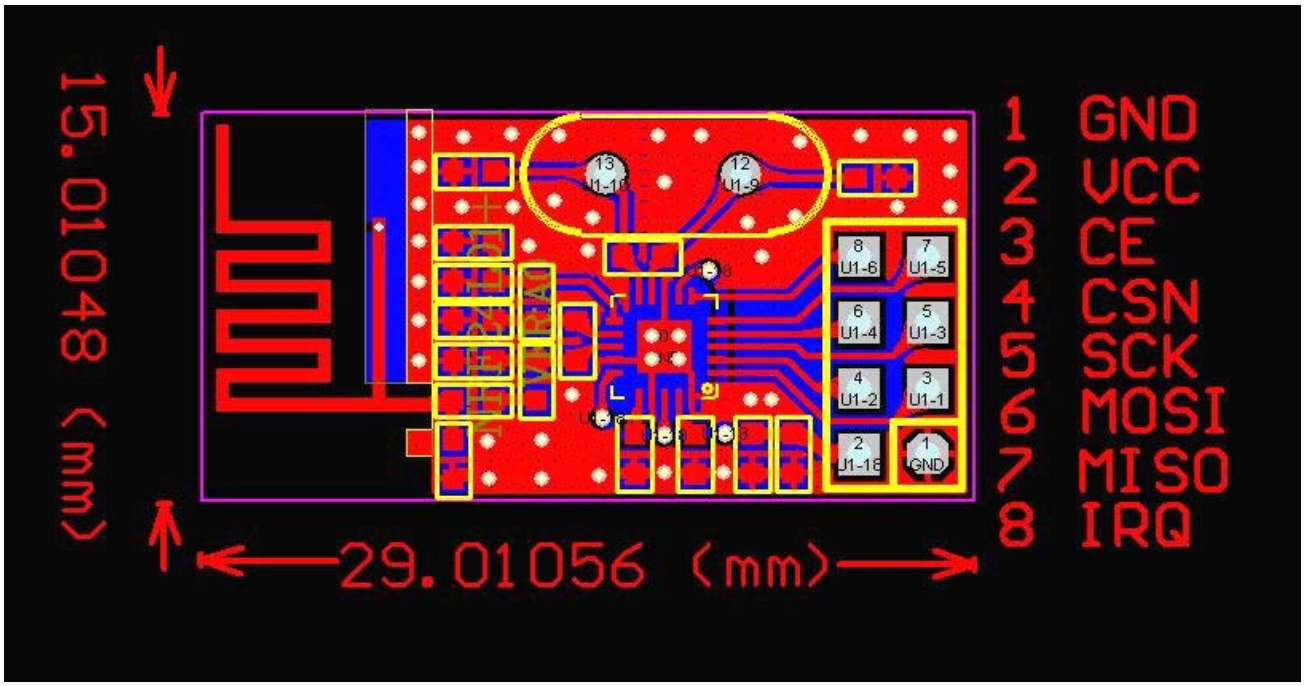
(10)工作于 Enhanced ShockBurst 具有 Automatic packet handling, Auto packet transaction handling,具有可选的内置包应答机制，极大的降低丢包率。

(11)与 51 系列单片机 P0 口连接时候，需要加 10K 的上拉电阻,与其余口连接不需要。

(12)其他系列的单片机，如果是 5V 的，请参考该系列单片机 IO 口输出电流大小，如果超过 10mA，需要串联电阻分压，否则容易烧毁模块！如果是3.3V的，可以直接和RF24l01模块的IO口线连接。比如AVR系列单片机

如果是5V 的，一般串接2K 的电阻

二、接口电路



说明：

1) VCC 脚接电压范围为 1.9V~3.6V 之间，不能在这个区间之外，超过 3.6V 将会烧毁模块。推荐电压 3.3V 左右。

(2) 除电源 VCC 和接地端，其余脚都可以直接和普通的 5V 单片机 IO 口

直接相连，无需电平转换。当然对 3V 左右的单片机更加适用了。

(3) 硬件上面没有 SPI 的单片机也可以控制本模块，用普通单片机 IO 口模拟 SPI 不需要单片机真正的串口介入，只需要普通的单片机 IO 口就可以了，当然用串口也可以了。

(4) 如果需要其他封装接口，比如密脚插针，或者其他形式的接口，可以联系我们定做。

三、模块结构和引脚说明

NRF24L01 模块使用 Nordic 公司的 nRF24L01 芯片开发而成。

1.2 Block diagram

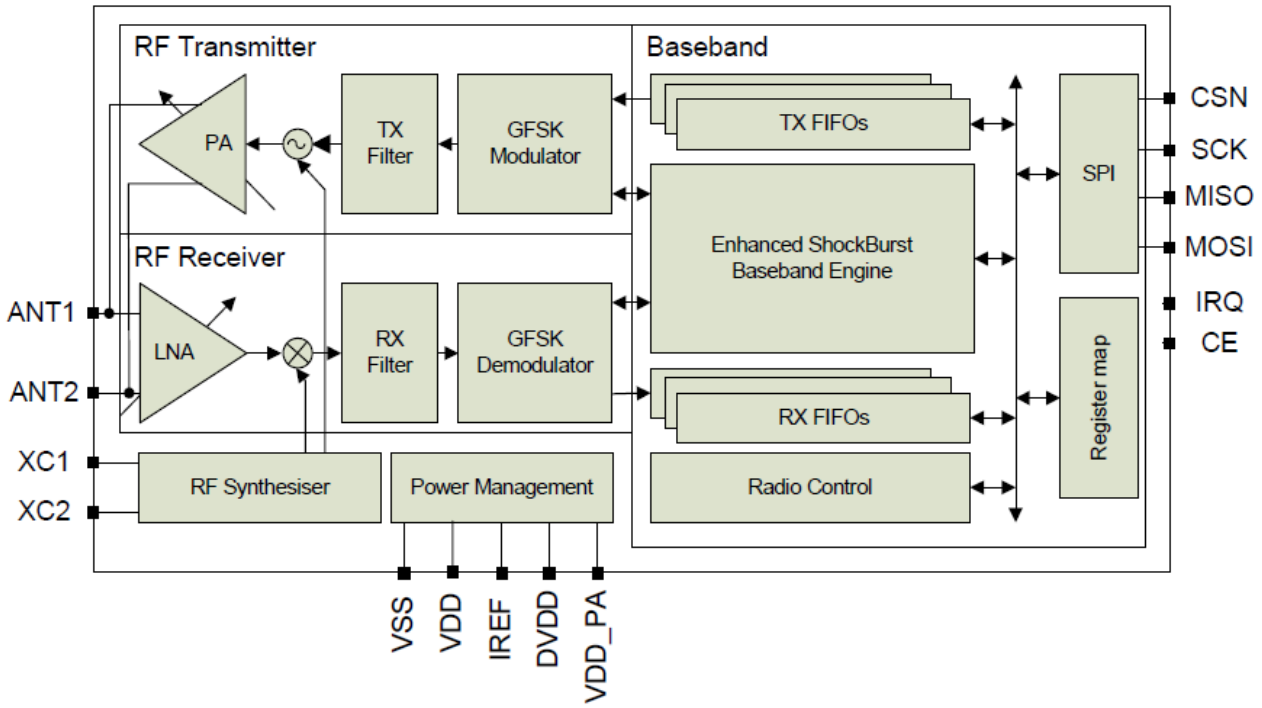


Figure 1. nRF24L01+ block diagram

内部方框图

2.1 Pin assignment

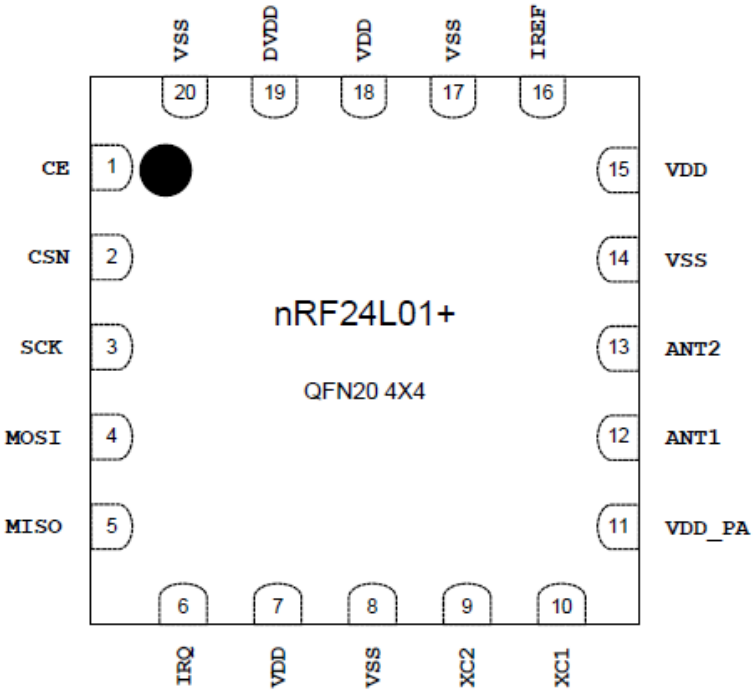
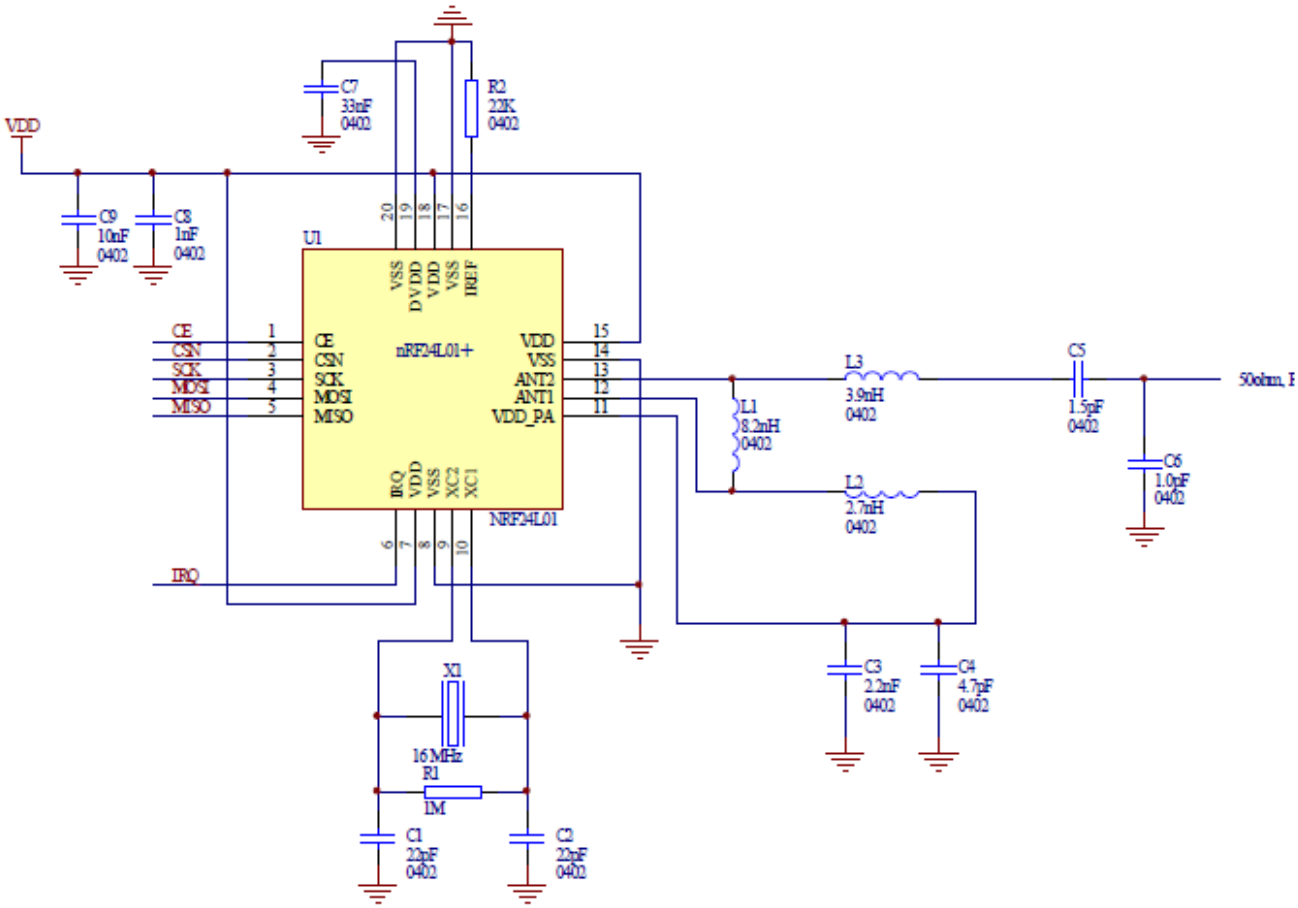


Figure 2. nRF24L01+ pin assignment (top view) for the QFN20 4x4 package

2.2 Pin functions

Pin	Name	Pin function	Description
1	CE	Digital Input	Chip Enable Activates RX or TX mode
2	CSN	Digital Input	SPI Chip Select
3	SCK	Digital Input	SPI Clock
4	MOSI	Digital Input	SPI Slave Data Input
5	MISO	Digital Output	SPI Slave Data Output, with tri-state option
6	IRQ	Digital Output	Maskable interrupt pin. Active low
7	VDD	Power	Power Supply (+1.9V - +3.6V DC)
8	VSS	Power	Ground (0V)
9	XC2	Analog Output	Crystal Pin 2
10	XC1	Analog Input	Crystal Pin 1
11	VDD_PA	Power Output	Power Supply Output (+1.8V) for the internal nRF24L01+ Power Amplifier. Must be connected to ANT1 and ANT2 as shown in Figure 32.
12	ANT1	RF	Antenna interface 1
13	ANT2	RF	Antenna interface 2
14	VSS	Power	Ground (0V)
15	VDD	Power	Power Supply (+1.9V - +3.6V DC)
16	IREF	Analog Input	Reference current. Connect a 22kΩ resistor to ground. See Figure 32.
17	VSS	Power	Ground (0V)
18	VDD	Power	Power Supply (+1.9V - +3.6V DC)
19	DVDD	Power Output	Internal digital supply output for de-coupling purposes. See Figure 32.
20	VSS	Power	Ground (0V)

Table 1. nRF24L01+ pin function



典型应用原理图

四、工作方式

NRF2401 有工作模式有五种：

1 收发模式

2 配置模式

3 空闲模式

4 关机模式

工作模式由 PWR_UP register 、PRIM_RX register 和 CE 决定，下表

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[a].
TX mode	1	0	Minimum 10µs high pulse	Data in TX FIFOs.Will empty one level in TX FIFOs[b].
Standby-II	1	0	1	TX FIFO empty.
Standby-I	1	-	0	No ongoing packet transmission.
Power Down	0	-	-	-

4.1 收发模式

收发模式有 Enhanced ShockBurstTM 收发模式、ShockBurstTM 收发模式和直接收发模式三种，收发模式由器件配置字决定，具体配置将在器件配置部分详细介绍。

4.1.1 Enhanced ShockBurstTM 收发模式

Enhanced ShockBurstTM 收发模式下，使用片内的先入先出堆栈区，数据低速从微控制器送入，但高速(1Mbps)发射，这样可以尽量节能，因此，使用低速的微控制器也能得到很高的射频数据发射速率。与射频协议相关的所有高速信号处理都在片内进行，这种做法有三大好处：尽量节能；低的系统费用(低速微处理器也能进行高速射频发射)；数据在空中停留时间短，抗干扰性高。Enhanced ShockBurstTM 技术同

时也减小了整个系统的平均工作电流。

在 Enhanced ShockBurstTM 收发模式下， NRF24L01 自动处理字头和 CRC 校验码。在接收数据时，自动把字头和 CRC 校验码移去。在送数据时，自动加上字头和 CRC 校验码，在发送模式下，置 CE 为高，至少 10us，将时发送过程完成后。

4.1.1.1 Enhanced ShockBurstTM 发射流程

A. 把接收机的地址和要发送的数据按时序送入 NRF24L01；

B. 配置 CONFIG 寄存器，使之进入发送模式。C. 微控制器把 CE 置高（至少 10us),激 发 NRF24L01 进行 Enhanced ShockBurstTM 发射； D. N24L01 的 Enhanced ShockBurstTM 发射(1) 给射频前端供电； (2) 射频数据打包(加字头、CRC 校验码)； (3) 高速发射数据包； (4) 发射完成，NRF24L01 进入空闲状态。4.1.1.2 Enhanced ShockBurstTM 接收流程 A. 配置本机地址和要接收的数据包大小；B. 配置 CONFIG 寄存器，使之进入接收模式，把 CE 置高。

C. 130us 后，NRF24L01 进入监视状态，等待数据包的到来； D.收到正确的数据包(正确的地址和 CRC 校验码)，NRF2401 自动把字头、地址和 CRC 校验位移去；

E. NRF24L01 通过把 STATUS 寄存器的 RX_DR 置位(STATUS 一般引起微控制器中断)通知微控制器；F. 微控制器把数据从 NewMsg_RF2401 读出；

G. 所有数据读取完毕后，可以清除 STATUS 寄存器。NRF2401 可以进入

4.1.2 ShockBurstTM 收发模式

ShockBurstTM 收发模式可以与 Nrf2401a,02,E1 及 E2 兼容，具体表述前

看本公司的 N-RF2401 文档。

4.2 空闲模式

NRF24L01 的空闲模式是为了减小平均工作电流而设计，其最大的优点是，实现节能的同时，缩短芯片的起动时间。在空闲模式下，部分片内晶振仍在工作，此时的工作电流跟外部晶振的频率有关。

4.4 关机模式

在关机模式下，为了得到最小的工作电流，一般此时的工作电流为 900nA 左右。关机模式下，配置字的内容也会被保持在 NRF2401 片内，这是该模式与断电状态最大的区别。

五、配置 NRF24L01 模块

NRF2401 的所有配置工作都是通过 SPI 完成，共有 30 字节的配置字。

我们推荐 NRF24L01 工作于 Enhanced ShockBurstTM 收发模式，这种工作模式下，系统的程序编制会更加简单，并且稳定性也会更高，因此，下文着重介绍把 NRF24L01 配置为 Enhanced ShockBurstTM 收发模式的器件配置方法。

ShockBurstTM 的配置字使 NRF24L01 能够处理射频协议，在配置完成后，在 NRF24L01 工作的过程中，只需改变其最低一个字节中的内容，以实现接收模式和发送模式之间切换。

ShockBurstTM 的配置字可以分为以下四个部分：

数据宽度：声明射频数据包中数据占用的位数。这使得 NRF24L01 能够区分接收数据包中的数据和 CRC 校验码；

地址宽度：声明射频数据包中地址占用的位数。这使得 NRF24L01 能够区分地址和数据；

地址：接收数据的地址，有通道 0 到通道 5 的地址；

CRC：使 NRF24L01 能够生成 CRC 校验码和解码。

当使用 NRF24L01 片内的 CRC 技术时，要确保在配置字(CONFIG 的 EN_CRC)

中 CRC 校验被使能，并且发送和接收使用相同的协议。

NRF24L01 配置字的 CONFIG 寄存器的位描述如下表所示。

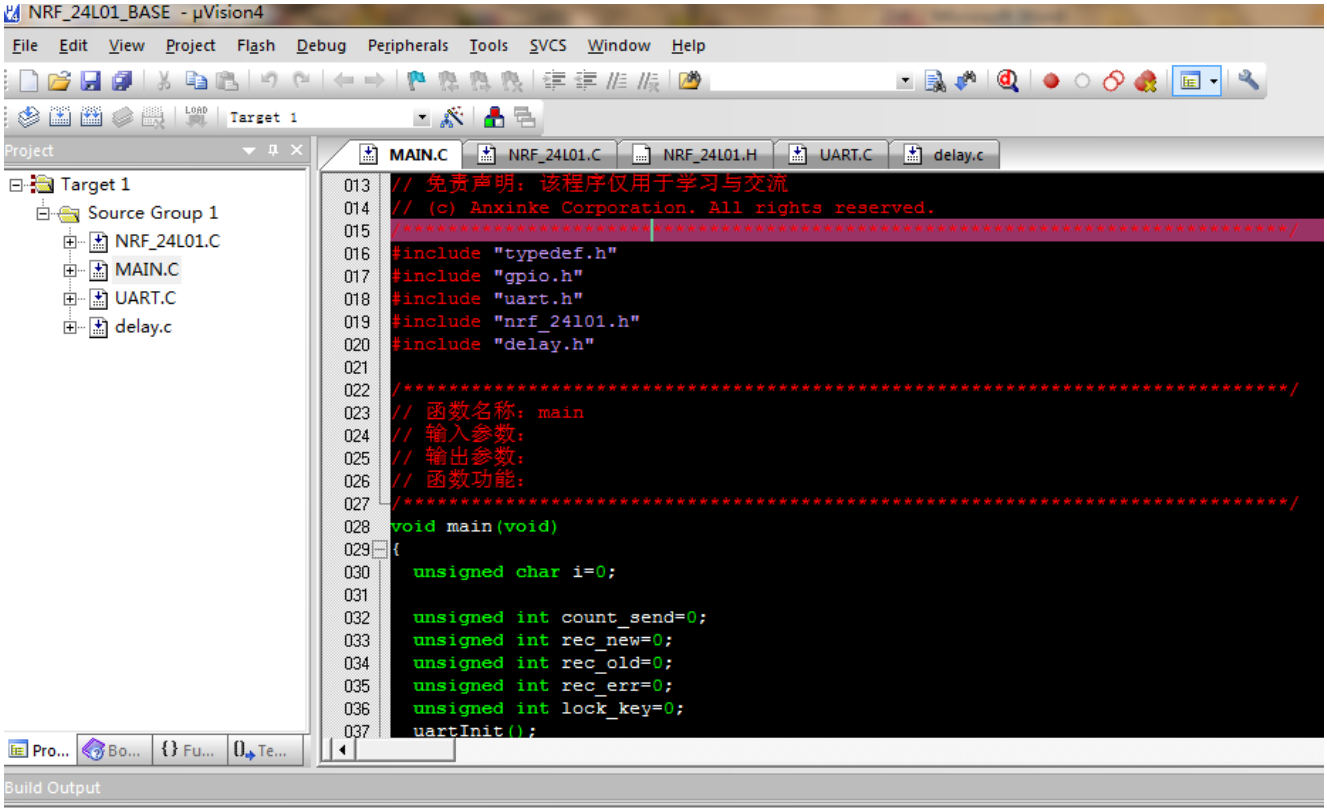
NRF24L01 CONFIG 配置字描述

NRF24L01 CONFIG 配置字描述

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
00	CONFIG				Configuration Register
	Reserved	7	0	R/W	Only '0' allowed
	MASK_RX_DR	6	0	R/W	Mask interrupt caused by RX_DR 1: Interrupt not reflected on the IRQ pin 0: Reflect RX_DR as active low interrupt on the IRQ pin
	MASK_TX_DS	5	0	R/W	Mask interrupt caused by TX_DS 1: Interrupt not reflected on the IRQ pin 0: Reflect TX_DS as active low interrupt on the IRQ pin
	MASK_MAX_RT	4	0	R/W	Mask interrupt caused by MAX_RT 1: Interrupt not reflected on the IRQ pin 0: Reflect MAX_RT as active low interrupt on the IRQ pin
	EN_CRC	3	1	R/W	Enable CRC. Forced high if one of the bits in the EN_AA is high
	CRCO	2	0	R/W	CRC encoding scheme '0' - 1 byte '1' – 2 bytes
	PWR_UP	1	0	R/W	1: POWER UP, 0:POWER DOWN
	PRIM_RX	0	0	R/W	1: PRX, 0: PTX

我们提供比较完善的测试程序，配套我们测试板可以让你迅速适应。

下面的程序是用 STC 单周期系列 15L204EA ，若用其它型号，请移植




```
/*********************************************************************************/
/*                          -- 深圳安信可科技有限公司 --                          */
/*
//  文件名：            NRF24L01.C
//  说明：
//  编写人员：
//  编写日期：
//  程序维护：
//  维护记录：
//
//
*/
// 免责声明：该程序仅用于学习与交流
// (c) Anxinke Corporation. All rights reserved.
/*********************************************************************************/
#include "typedef.h"
#include <intrins.h>
#include <string.h>
#include "NRF_24L01.h"
#include "UART.H"
#include "delay.h"
#define uchar unsigned char
uchar const TX_ADDRESS[TX_ADR_WIDTH]  = {0x34,0x43,0x10,0x10,0x01}; // Define a static TX address
idata uchar rx_buf[TX_PLOAD_WIDTH];
idata uchar tx_buf[TX_PLOAD_WIDTH]={0X02,0X02,0X03,0X05,0X09};
uchar flag;
/*************************************************/
//sbit LED1= P3^7;
/*************************************************/
unsigned char          bdata sta;
```

```
sbit                    RX_DR    =sta^6;
sbit                    TX_DS    =sta^5;
sbit                    MAX_RT   =sta^4;
/************************************************/
/************************************************
Function: init_io();
Description:
  flash led one time,chip enable(ready to TX or RX Mode),
  Spi disable,Spi clock line init high
/************************************************/
#define KEY 0xaa
void init_nrf24l01_io(void)
{
                        CE=0;                   // chip enable
                        CSN=1;                  // Spi disable
                        SCK=0;                  // Spi clock line init high
}
/************************************************
Function: SPI_RW();

Description:
  Writes one byte to nRF24L01, and return the byte read
  from nRF24L01 during write, according to SPI protocol
/************************************************/
bdata unsigned char st=0;
sbit st_1=st^0;
sbit st_2=st^1;
sbit st_3=st^2;
sbit st_4=st^3;
sbit st_5=st^4;
sbit st_6=st^5;
sbit st_7=st^6;
sbit st_8=st^7;
bdata unsigned char st1=0;
sbit st_11=st1^0;
sbit st_12=st1^1;
sbit st_13=st1^2;
sbit st_14=st1^3;
sbit st_15=st1^4;
sbit st_16=st1^5;
sbit st_17=st1^6;
sbit st_18=st1^7;
/*
uchar SPI_RW(uchar byte)
{
                        uchar bit_ctr;
                        for(bit_ctr=0;bit_ctr<8;bit_ctr++)   // output 8-bit
                        {
                         MOSI = (byte & 0x80);         // output 'byte', MSB to MOSI
                         byte = (byte << 1);           // shift next bit into MSB..
                         SCK = 1;                      // Set SCK high..
                         MISO=1;
                         byte |= MISO;                      // capture current MISO bit
                         SCK = 0;                           // ..then set SCK low again
                        }
    return(byte);                           // return read byte
}
*/

uchar SPI_RW(uchar byte)
{
//uchar bit_ctr;
    st=byte;
```

```
    MOSI=st_8;
    SCK = 1;
    st_18=MISO;
    SCK = 0;
    MOSI=st_7;
    SCK = 1;
    st_17=MISO;
    SCK = 0;、
    MOSI=st_6;
    SCK = 1;
    st_16=MISO;
    SCK = 0;

    MOSI=st_5;
    SCK = 1;
    st_15=MISO;
    SCK = 0;
    MOSI=st_4;
    SCK = 1;
    st_14=MISO;
    SCK = 0;
    MOSI=st_3;
    SCK = 1;
    st_13=MISO;
    SCK = 0;
    MOSI=st_2;
    SCK = 1;
    st_12=MISO;
    SCK = 0;
    MOSI=st_1;
    SCK = 1;
    st_11=MISO;
    SCK = 0;
    return(st1);                           // return read byte
}
/****************************************************/

/****************************************************
Function: SPI_RW_Reg();

Description:
  Writes value 'value' to register 'reg'
/****************************************************/
uchar SPI_RW_Reg(BYTE reg, BYTE value)
{
                        uchar status;
                        CSN = 0;                   // CSN low, init SPI transaction
                        status = SPI_RW(reg);      // select register
                        SPI_RW(value);             // ..and write value to it..
                        CSN = 1;                   // CSN high again
                        return(status);            // return nRF24L01 status byte
}
/****************************************************/

/****************************************************
Function: SPI_Read();

Description:
  Read one byte from nRF24L01 register, 'reg'
/****************************************************/
BYTE SPI_Read(BYTE reg)
{
                        BYTE reg_val;
```

```
                        CSN = 0;                // CSN low, initialize SPI communication...
                        SPI_RW(reg);            // Select register to read from..
                        reg_val = SPI_RW(0);    // ..then read registervalue
                        CSN = 1;                // CSN high, terminate SPI communication
                        return(reg_val);        // return register value
}
/**************************************************/

/**************************************************
Function: SPI_Read_Buf();

Description:
  Reads 'bytes' #of bytes from register 'reg'
  Typically used to read RX payload, Rx/Tx address
/**************************************************/
uchar SPI_Read_Buf(BYTE reg, BYTE *pBuf, BYTE bytes)
{
                        uchar status,byte_ctr;
                        CSN = 0;                                    // Set CSN low, init SPI tranaction
                        status = SPI_RW(reg);                       // Select register to write to and read
                        status byte
                        for(byte_ctr=0;byte_ctr<bytes;byte_ctr++)
                        pBuf[byte_ctr] = SPI_RW(0);    // Perform SPI_RW to read byte from nRF24L01
                        CSN = 1;                           // Set CSN high again
                        return(status);                    // return nRF24L01 status byte
}
/**************************************************/

/**************************************************
Function: SPI_Write_Buf();
Description:
  Writes contents of buffer '*pBuf' to nRF24L01
  Typically used to write TX payload, Rx/Tx address
/**************************************************/
uchar SPI_Write_Buf(BYTE reg, BYTE *pBuf, BYTE bytes)
{
                        uchar status,byte_ctr;
                        CSN = 0;                    // Set CSN low, init SPI tranaction
                        status = SPI_RW(reg);    // Select register to write to and read status byte
                        for(byte_ctr=0; byte_ctr<bytes; byte_ctr++) // then write all byte in buffer(*pBuf)
                        SPI_RW(*pBuf++);
                        CSN = 1;                 // Set CSN high again
                        return(status);          // return nRF24L01 status byte
}
/**************************************************/

/**************************************************
Function: RX_Mode();

Description:
  This function initializes one nRF24L01 device to
  RX Mode, set RX address, writes RX payload width,
  select RF channel, datarate & LNA HCURR.
  After init, CE is toggled high, which means that
  this device is now ready to receive a datapacket.
/**************************************************/
void power_off()
{
    CE=0;
    SPI_RW_Reg(WRITE_REG + CONFIG, 0x0D);
    CE=1;
    _delay_us(20);
}
```

```
void ifnnrf_rx_mode(void)
{
                        power_off();
                        CE=0;
                        SPI_Write_Buf(WRITE_REG + RX_ADDR_P0, TX_ADDRESS, TX_ADR_WIDTH); // Use the same
                        address on the RX device as the TX device

                        SPI_RW_Reg(WRITE_REG + EN_AA, 0x01);      // Enable Auto.Ack:Pipe0
                        SPI_RW_Reg(WRITE_REG + EN_RXADDR, 0x01);  // Enable Pipe0
                        SPI_RW_Reg(WRITE_REG + RF_CH, 70);        // Select RF channel 40
                        SPI_RW_Reg(WRITE_REG + RX_PW_P0, TX_PLOAD_WIDTH); // Select same RX payload width as
                        TX Payload width
                        SPI_RW_Reg(WRITE_REG + RF_SETUP, 0x26);   // TX_PWR:0dBm, Datarate:2Mbps, LNA:HCURR
                        SPI_RW_Reg(WRITE_REG + CONFIG, 0x0f);     // Set PWR_UP bit, enable CRC(2 bytes) &
                        Prim:RX. RX_DR enabled..
                        CE = 1; // Set CE pin high to enable RX device
  //  This device is now ready to receive one packet of 16 bytes payload from a TX device sending to address
  //  '3443101001', with auto acknowledgment, retransmit count of 10, RF channel 40 and datarate = 2Mbps.

}
/**************************************************/

/**************************************************
Function: TX_Mode();
Description:
  This function initializes one nRF24L01 device to
  TX mode, set TX address, set RX address for auto.ack,
  fill TX payload, select RF channel, datarate & TX pwr.
  PWR_UP is set, CRC(2 bytes) is enabled, & PRIM:TX.
  ToDo: One high pulse(>10us) on CE will now send this
  packet and expext an acknowledgment from the RX device.
**************************************************/
void ifnnrf_tx_mode(void)
{
                        power_off();
                        CE=0;
                        SPI_Write_Buf(WRITE_REG + TX_ADDR, TX_ADDRESS, TX_ADR_WIDTH);    // Writes TX_Address
                        to nRF24L01
                        SPI_Write_Buf(WRITE_REG + RX_ADDR_P0, TX_ADDRESS, TX_ADR_WIDTH); // RX_Addr0 same as
                        TX_Adr for Auto.Ack
                        SPI_Write_Buf(WR_TX_PLOAD, tx_buf, TX_PLOAD_WIDTH); // Writes data to TX payload
                        SPI_RW_Reg(WRITE_REG + EN_AA, 0x01);      // Enable Auto.Ack:Pipe0
                        SPI_RW_Reg(WRITE_REG + EN_RXADDR, 0x01);  // Enable Pipe0
                        SPI_RW_Reg(WRITE_REG + SETUP_RETR, 0x1a); // 500us + 86us, 10 retrans...
                        SPI_RW_Reg(WRITE_REG + RF_CH, 70);        // Select RF channel 40
                        SPI_RW_Reg(WRITE_REG + RF_SETUP, 0x26);   // TX_PWR:0dBm, Datarate:2Mbps, LNA:HCURR
                        SPI_RW_Reg(WRITE_REG + CONFIG, 0x0e);     // Set PWR_UP bit, enable CRC(2 bytes) &
                        Prim:TX. MAX_RT & TX_DS enabled..
                        CE=1;
}
void SPI_CLR_Reg(BYTE R_T)
{
                        CSN = 0;
                        if(R_T==1)                  // CSN low, init SPI transaction
                        SPI_RW(FLUSH_TX);             // ..and write value to it..
                        else
                        SPI_RW(FLUSH_RX);             // ..and write value to it..
                        CSN = 1;                   // CSN high again
}

void ifnnrf_CLERN_ALL()
{
  SPI_CLR_Reg(0);
```

```
  SPI_CLR_Reg(1);
  SPI_RW_Reg(WRITE_REG+STATUS,0xff);
  IRQ=1;
}
/////////////////////END OF 24L01.C//////////////////////
/*****************************************************************************/
/*                          -- 深圳安信可科技有限公司 --                        */
/*
//  文件名：main.c
//  说明：供客户测试模块通信使用程序
//  编写人员：
//  编写日期：
//  程序维护：
//  维护记录：
//
//
*/
// 免责声明：该程序仅用于学习与交流
// (c) Anxinke Corporation. All rights reserved.
/*****************************************************************************/
#include "typedef.h"
#include "gpio.h"
#include "uart.h"
#include "nrf_24l01.h"
#include "delay.h"

/*****************************************************************************/
// 函数名称：main
// 输入参数：
// 输出参数：
// 函数功能：
/*****************************************************************************/
void main(void)
{
  unsigned char i=0;
  unsigned int count_send=0;
  unsigned int rec_new=0;
  unsigned int rec_old=0;
  unsigned int rec_err=0;
  unsigned int lock_key=0;
  uartInit();
  Delay100ms();
  init_nrf24l01_io();
  led=0;
  Delay100ms();
  led=1;
  Delay100ms();
  led=0;
  Delay100ms();
  led=1;
  Delay100ms();
  TI=1;
  Delay100ms();
  uartSendString("READY!\r\n");
  ifnnrf_rx_mode();
  while(1)
  {
    KEY=1;
    KEY2 = 1;
    while(KEY == 0 || KEY2 == 0){
        IRQ=1;
        SPI_RW_Reg(WRITE_REG+STATUS,0xff);
        if (KEY == 0) {
```

```
            tx_buf[4]+=1;
        } else if (KEY2 == 0) {
            tx_buf[4]-=1;
        }
        ifnnrf_tx_mode();
        while(IRQ);
        sta=SPI_Read(STATUS);
        SPI_RW_Reg(WRITE_REG+STATUS,0xff);
        if(sta&STA_MARK_TX){
            if (KEY == 0){
                        led=0;
                        Delay100ms();
                        led=1;
                Delay100ms();
            } else {
                        led2=0;
                        Delay100ms();
                        led2=1;
                Delay100ms();
            }
                          } else {
            ifnnrf_CLERN_ALL();
                        }
                        lock_key=1;
    }                  //发送
    if(lock_key) {
        lock_key=0;
        ifnnrf_rx_mode();
        IRQ=1;
        while(IRQ==0);
        _delay_us(400);
    }
    IRQ=1;
    if(IRQ==0) {
        sta=SPI_Read(STATUS);
        SPI_RW_Reg(WRITE_REG+STATUS,0xff);
        if(sta&STA_MARK_RX) {
            SPI_Read_Buf(RD_RX_PLOAD,rx_buf,TX_PLOAD_WIDTH);
            if (rx_buf[0] == 0xAA) {
                uartSendString("你按下了左边按键\r\n");
                led=0;
                Delay100ms();
                led=1;
                Delay100ms();
            } else if (rx_buf[0] == 0x55)  {
                uartSendString("你按下了右边按键\r\n");
                        led2=0;
                        Delay100ms();
                        led2=1;
                Delay100ms();
            }
            rx_buf[0] = 0;
        } else {
            ifnnrf_CLERN_ALL();
            ifnnrf_rx_mode();
            IRQ=1;
            while(IRQ==0);
        }
    } //接收
  }
}
///////////////END OF MAIN.C//////////////////////////
/*****************************************************************************/
```

```
/*                          -- 深圳安信可科技有限公司 --                          */
/*
//  文件名：   uart.c
//  说明：     串口底层驱动函数，使用 16 位的定时器作为波特率发生器，自动重载模式
//  编写人员：Wu
//  编写日期：2012 年 11 月 12 日
//  程序维护：
//                      维护记录：
//
//
*/
// 免责声明：该程序仅用于学习与交流
// (c) Anxinke Corporation. All rights reserved.
/*****************************************************************************/
#include "typedef.h"
#include "gpio.h"
#include "uart.h"
static bit bUartFlag;
/*****************************************************************************/
// 函数名称：uartInit
// 输入参数：无
// 输出参数：无
// 函数功能：设置好定时器 0 的工作模式
/*****************************************************************************/
void uartInit(void)
{
    /*
     * 设置定时器 0 为 16 位自动重载定时器
    */
    AUXR |= 0x80; //定时器 0 为 1T 模式
    TMOD &= 0xF0; //设置定时器为模式 0(16 位自动重装载)
                        TL0 = (0xFFFF - MCU_FREQ / UART_BUAD) & 0xFF; //设置定时初值
TH0 = ((0xFFFF - MCU_FREQ / UART_BUAD) >> 8) & 0xFF; //设置定时初值
    TR0 = 0; //定时器 0 开始计时
    ET0 = 0; //使能定时器 0 中断
    EA = 1;
}

/*****************************************************************************/
// 函数名称：uartSendData
// 输入参数：ucData: 发送字节
// 输出参数：无
// 函数功能：使用串口发送一个字节数据
/*****************************************************************************/
void uartSendData(u8 ucData)
{
    u8 ucCnt;
    UART_TX_LOW(); //串口起始位开始
    TR0 = 1; //定时器 0 开始计时
    ET0 = 1; //使能定时器 0 中断
    bUartFlag = ON;
    while(bUartFlag == ON);
                        /*
                         * 由低位开始，将数据通过串口输出
                        */
    for (ucCnt = 0; ucCnt < 8; ucCnt++) {
        UART_TX_SET(ucData & 0x01);
        ucData >>= 1;
        bUartFlag = ON;
        while(bUartFlag == ON);
    }
    UART_TX_HIGH(); // 发送串口停止位
    bUartFlag = ON;
```

```
    while(bUartFlag == ON);
    TR0 = 0; //定时器 0 结束计时
    ET0 = 0; //禁能定时器 0 中断
}
/*****************************************************************************/
// 函数名称：uartSendString
// 输入参数：pS: 字符串首地址
// 输出参数：无
// 函数功能：发送字符串通过串口输出
/*****************************************************************************/
void uartSendString(char *pS)
{
    while (*pS)                     //检测字符串结束标志
    {
        uartSendData(*pS++);        //发送当前字符
    }
}

/*****************************************************************************/
// 函数名称：time0ISR
// 输入参数：无
// 输出参数：无
// 函数功能：串口 0 服务函数
/*****************************************************************************/
void time0ISR(void) interrupt 1 using 1
{
    EA = 0;
    bUartFlag = OFF;
    EA = 1;
}


///////////////END OF UART.C//////////////////////////////


/*****************************************************************************/
/*                        -- 深圳安信可科技有限公司 --                        */
/*
//  文件名：
//  说明：
//  编写人员：
//  编写日期：
//  程序维护：
//  维护记录：
//
//
*/
// 免责声明：该程序仅用于学习与交流
// (c) Anxinke Corporation. All rights reserved.
/*****************************************************************************/
#include "delay.h"
/*****************************************************************************/
// 函数名称：
// 输入参数：
// 输出参数：
// 函数功能：
/*****************************************************************************/
void Delay100ms(void)       //@22.1184MHz
{
                        unsigned char i, j, k;
                        _nop_();
                        _nop_();
                        i = 9;
```

```
                        j = 104;
                        k = 139;
                        do
                        {
                         do
                         {
                                 while (--k);
                         } while (--j);
                        } while (--i);
}

/*****************************************************************************/
// 函数名称：
// 输入参数：
// 输出参数：
// 函数功能：
/*****************************************************************************/
void _delay_us(unsigned int _us)
{
    char a=0;
    for(_us;_us;_us--)
        for(a=0;a<1;a++);
}
//////////////////////END OF DELAY.C//////////////////////
```

程序详解

芯片简介

NRF24L01 是NORDIC 公司最近生产的一款无线通信通信芯片，采用FSK 调制，内部集成NORDIC 自己的Enhanced Short Burst 协议。可以实现点对点或是1 对6 的无线通信。无线通信速度可以达到2M（bps）。NORDIC 公司提供通信模块的GERBER 文件，可以直接加工生产。嵌入式工程师或是单片机爱好者只需要为单片机系统预留5 个GPIO，1 个中断输入引脚，就可以很容易实现无线通信的功能，非常适合用来为MCU 系统构建无线通信功能。

1 NRF24L01 功能框图

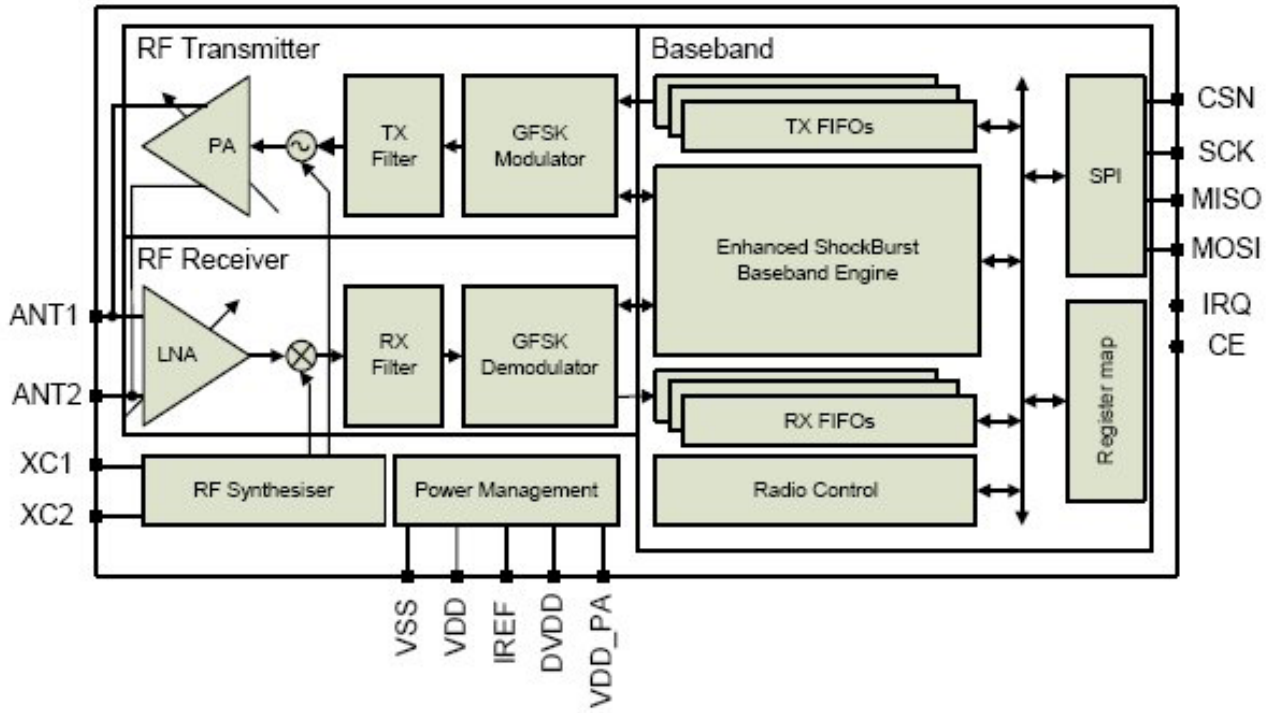


Fig.1 NRF24L01 BLOCK DIAGRAM

NRF24L01 的框图如Fig.1 所示，从单片机控制的角度来看，我们只需要关注Fig.1 右面
的六个控制和数据信号，分别为CSN、SCK、MISO、MOSI、IRQ、CE。
CSN：芯片的片选线，CSN 为低电平芯片工作。
SCK：芯片控制的时钟线（SPI 时钟）
MISO：芯片控制数据线（Master input slave output）
MOSI：芯片控制数据线（Master output slave input）
IRQ：中断信号。无线通信过程中MCU 主要是通过IRQ 与NRF24L01 进行通信。
CE： 芯片的模式控制线。在 CSN 为低的情况下，CE 协同NRF24L01 的CONFIG 寄存器共同决定NRF24L01 的状态（参照NRF24L01 的状态机）
。

2 NRF24L01 状态机

NRF24L01 的状态机见Fig.2 所示，对于NRF24L01 的固件编程工作主要是参照

NRF24L01 的状态机。主要有以下几个状态

Power Down Mode：掉电模式

Tx Mode：发射模式

Rx Mode：接收模式

Standby-1Mode：待机1 模式

Standby-2 Mode：待机2 模式

上面五种模式之间的相互切换方法以及切换所需要的时间参照

Fig.2

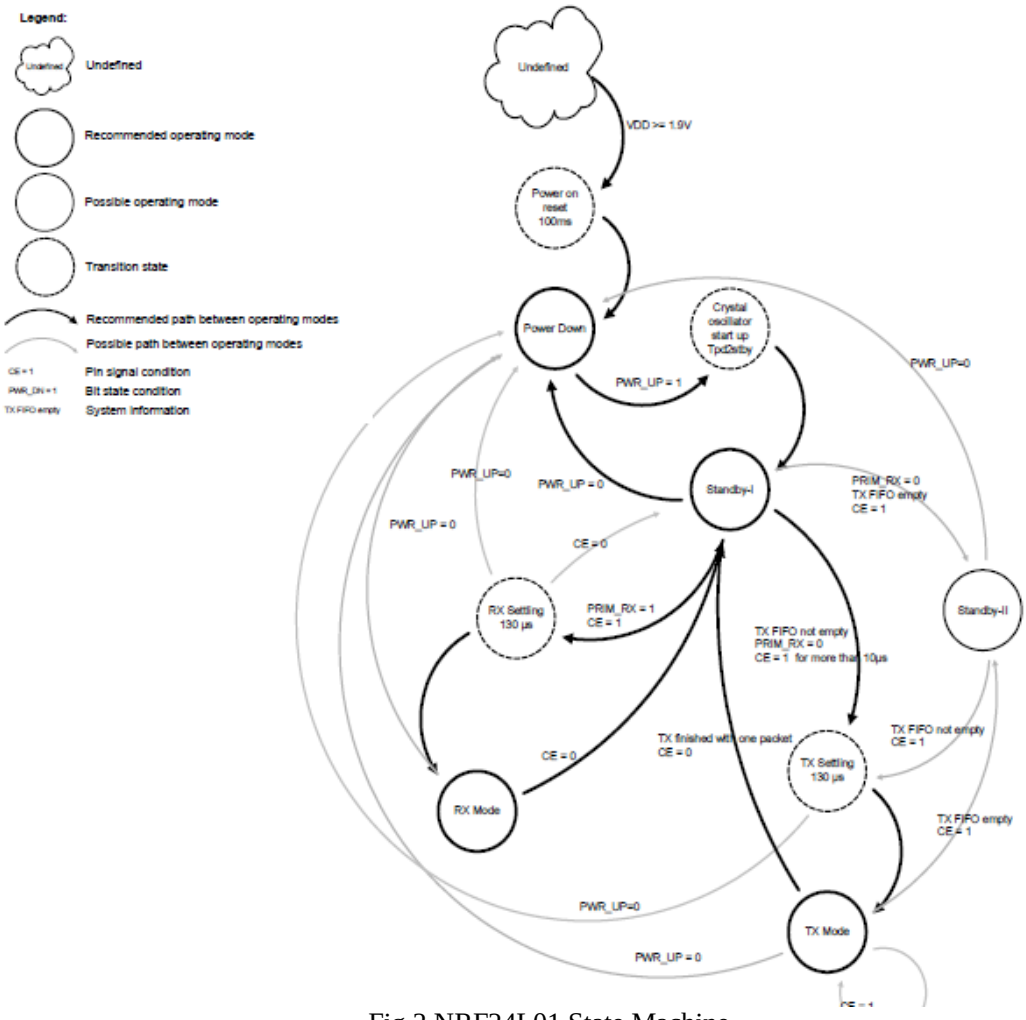


Fig.2 NRF24L01 State Machine

对24L01 的固件编程的基本思路如下：
1） 置 CSN 为低，使能芯片，配置芯片各个参数。（过程见3.Tx 与Rx 的配置过程）
配置参数在 Power Down 状态中完成。
2） 如果是 Tx 模式，填充Tx FIFO。
3） 配置完成以后，通过 CE 与CONFIG 中的PWR_UP 与PRIM_RX 参数确定24L01
要切换到的状态。
Tx Mode： PWR_UP=1; PRIM_RX=0; CE=1 (保持超过10us 就可以);
Rx Mode: PWR_UP=1; PRIM_RX=1; CE=1;
4) IRQ 引脚会在以下三种情况变低：
Tx FIFO 发完并且收到ACK（使能ACK 情况下）
Rx FIFO 收到数据
达到最大重发次数
将 IRQ 接到外部中断输入引脚，通过中断程序进行处理。

3 Tx 与Rx 的配置过程

本节只是叙述了采用ENHANCED SHORT BURST 通信方式的Tx 与Rx 的配置及通信
过程，熟悉了24L01 以后可以采用别的通信方式。

3.1 Tx 模式初始化过程

3.1 Tx 模式初始化过程

初始化步骤	24L01 相关寄存器
1）写 Tx 节点的地址	TX_ADDR
2）写 Rx 节点的地址（主要是为了使能 Auto Ack）	RX_ADDR_P0
3）使能 AUTO ACK	EN_AA
4）使能 PIPE 0	EN_RXADDR
5）配置自动重发次数	SETUP_RETR
6）选择通信频率	RF_CH
7）配置发射参数（低噪放大器增益、发射功率、无线速率）	RF_SETUP
8）选择通道 0 有效数据宽度	Rx_Pw_P0
9）配置 24L01 的基本参数以及切换工作模式	CONFIG

Tx 模式初始化过程

1）写Tx 节点的地址 TX_ADDR
2）写Rx 节点的地址（主要是为了使能Auto Ack） RX_ADDR_P0
3）使能AUTO ACK EN_AA
4）使能PIPE 0 EN_RXADDR
5）配置自动重发次数 SETUP_RETR
6）选择通信频率 RF_CH
7）配置发射参数（低噪放大器增益、发射功率、无线速率） RF_SETUP
8) 选择通道0 有效数据宽度 Rx_Pw_P0
9）配置24L01 的基本参数以及切换工作模式 CONFIG。

3.2 Rx 模式初始化过程

初始化步骤	24L01 相关寄存器
1）写 Rx 节点的地址	RX_ADDR_P0
2）使能 AUTO ACK	EN_AA
3）使能 PIPE 0	EN_RXADDR
4）选择通信频率	RF_CH
5) 选择通道 0 有效数据宽度	Rx_Pw_P0
6）配置发射参数（低噪放大器增益、发射功率、无线速率）	RF_SETUP
7）配置 24L01 的基本参数以及切换工作模式	CONFIG

Rx 模式初始化过程：

初始化步骤 24L01 相关寄存器
1）写Rx 节点的地址 RX_ADDR_P0
2）使能AUTO ACK EN_AA
3）使能PIPE 0 EN_RXADDR
4）选择通信频率 RF_CH
5) 选择通道0 有效数据宽度 Rx_Pw_P0
6）配置发射参数（低噪放大器增益、发射功率、无线速率） RF_SETUP
7）配置24L01 的基本参数以及切换工作模式 CONFIG。

4 控制程序详解

4.1 函数介绍

NRF24L01 的控制程序主要包括以下几个函数

```
uchar SPI_RW(uchar byte);
uchar SPI_RW_Reg(uchar reg, uchar value);
uchar SPI_Read(uchar reg);
uchar SPI_Read_Buf(uchar reg, uchar *pBuf, uchar bytes);
uchar SPI_Write_Buf(uchar reg, uchar *pBuf, uchar bytes);
void RX_Mode(void);
void TX_Mode(void);
```

4.1.1 uchar SPI_RW(uchar byte)

```
uchar SPI_RW(uchar byte)
{
uchar bit_ctr;
for(bit_ctr=0;bit_ctr<8;bit_ctr++) // output 8-bit
{
MOSI = (byte & 0x80); // output 'byte', MSB to MOSI
byte = (byte << 1); // shift next bit into MSB..
SCK = 1; // Set SCK high..
byte |= MISO; // capture current MISO bit
SCK = 0; // ..then set SCK low again
}
return(byte); // return read byte
}
```

最基本的函数，完成GPIO 模拟SPI 的功能。将输出字节（MOSI）从MSB 循环输出，
同时将输入字节（MISO）从LSB 循环移入。上升沿读入，下降沿输出。（从SCK 被初始化
为低电平可以判断出）。

4.1.2 uchar SPI_RW_Reg (uchar reg, uchar value)

```
uchar SPI_RW_Reg(uchar reg, uchar value)
{
uchar status;
CSN = 0; // CSN low, init SPI transaction
status = SPI_RW(reg); // select register
SPI_RW(value); // ..and write value to it..
CSN = 1; // CSN high again
return(status); // return nRF24L01 status byte
}
```

寄存器访问函数：用来设置24L01 的寄存器的值。基本思路就是通过WRITE_REG 命令（也
就是0x20+寄存器地址）把要设定的值写到相应的寄存器地址里面去，并读

取返回值。对于
函数来说也就是把value 值写到reg 寄存器中。
需要注意的是，访问NRF24L01 之前首先要enable 芯片（CSN=0；），访问完了以后再disable
芯片（CSN=1；）。

4.1.3 uchar SPI_Read (uchar reg);

```
uchar SPI_Read(uchar reg)
{
uchar reg_val;
CSN = 0; // CSN low, initialize SPI communication
SPI_RW(reg); // Select register to read from..
reg_val = SPI_RW(0); // ..then read registervalue
CSN = 1; // CSN high, terminate SPI communication
return(reg_val); // return register value
}
```

读取寄存器值的函数：基本思路就是通过READ_REG 命令（也就是0x00+寄存器地址），把
寄存器中的值读出来。对于函数来说也就是把reg 寄存器的值读到reg_val 中去。

4.1.4 uchar SPI_Read_Buf (uchar reg, uchar *pBuf, uchar bytes);

```
uchar SPI_Read_Buf(uchar reg, uchar *pBuf, uchar bytes)
{
uchar status,byte_ctr;
CSN = 0; // Set CSN low, init SPI tranaction
status = SPI_RW(reg); // Select register to write to and read status byte
for(byte_ctr=0;byte_ctr<bytes;byte_ctr++)
pBuf[byte_ctr] = SPI_RW(0); // Perform SPI_RW to read byte from nRF24L01
CSN = 1; // Set CSN high again
return(status); // return nRF24L01 status byte
}
```

接收缓冲区访问函数：主要用来在接收时读取FIFO 缓冲区中的值。基本思路就是通过
READ_REG 命令把数据从接收FIFO（RD_RX_PLOAD）中读出并存到数组里面去。

4.1.5 uchar SPI_Write_Buf (uchar reg, uchar *pBuf, uchar bytes);

```
uchar SPI_Write_Buf(uchar reg, uchar *pBuf, uchar bytes)
{
uchar status,byte_ctr;
CSN = 0; // Set CSN low, init SPI tranaction
status = SPI_RW(reg); // Select register to write to and read
```

```
status byte
Uart_Delay(10);
for(byte_ctr=0; byte_ctr<bytes; byte_ctr++) // then write all
byte in buffer(*pBuf)
SPI_RW(*pBuf++);
CSN = 1; // Set CSN high again
return(status); // return nRF24L01 status byte
}
```

发射缓冲区访问函数：主要用来把数组里的数放到发射FIFO 缓冲区中。基本思路就是通过
WRITE_REG 命令把数据存到发射FIFO（WR_TX_PLOAD）中去。

4.1.6 void RX_Mode(void)

设定24L01 为接收方式，配置过程详见3.2 Rx 模式初始化过程。

```
void RX_Mode(void)
{
CE=0;
SPI_Write_Buf(WRITE_REG + RX_ADDR_P0, TX_ADDRESS, TX_ADR_WIDTH);
SPI_RW_Reg(WRITE_REG + EN_AA, 0x01); // Enable Auto.Ack:Pipe0
SPI_RW_Reg(WRITE_REG + EN_RXADDR, 0x01); // Enable Pipe0
SPI_RW_Reg(WRITE_REG + RF_CH, 40); // Select RF channel 40
SPI_RW_Reg(WRITE_REG + RX_PW_P0, TX_PLOAD_WIDTH);
SPI_RW_Reg(WRITE_REG + RF_SETUP, 0x07);
SPI_RW_Reg(WRITE_REG + CONFIG, 0x0f); // Set PWR_UP bit, enable
CRC(2 bytes)
& Prim:RX. RX_DR enabled..
CE = 1; // Set CE pin high to enable RX device
// This device is now ready to receive one packet of 16 bytes
payload from a TX device
sending to address
// '3443101001', with auto acknowledgment, retransmit count of 10,
RF channel 40 and
datarate = 2Mbps.
}
```

4.1.7 void TX_Mode(void)

设定24L01 为发送方式，配置过程详见3.1 Tx 模式初始化过程。

```
void TX_Mode(void)
{
CE=0;
```

```
SPI_Write_Buf(WRITE_REG + TX_ADDR, TX_ADDRESS, TX_ADR_WIDTH);
SPI_Write_Buf(WRITE_REG + RX_ADDR_P0, TX_ADDRESS, TX_ADR_WIDTH);
SPI_Write_Buf(WR_TX_PLOAD, tx_buf, TX_PLOAD_WIDTH); // Writes
data to TX payload
SPI_RW_Reg(WRITE_REG + EN_AA, 0x01); // Enable Auto.Ack:Pipe0
SPI_RW_Reg(WRITE_REG + EN_RXADDR, 0x01); // Enable Pipe0
SPI_RW_Reg(WRITE_REG + SETUP_RETR, 0x1a); // 500us + 86us, 10
retrans...
SPI_RW_Reg(WRITE_REG + RF_CH, 40); // Select RF channel 40
SPI_RW_Reg(WRITE_REG + RF_SETUP, 0x07); // TX_PWR:0dBm,
Datarate:2Mbps,
LNA:HCURR
SPI_RW_Reg(WRITE_REG + CONFIG, 0x0e); // Set PWR_UP bit, enable
CRC(2 bytes)
& Prim:TX. MAX_RT & TX_DS enabled..
CE=1;
}
```

4.2 NRF24L01 相关命令的宏定义

nRF24L01 的基本思路就是通过固定的时序与命令，控制芯片进行发射与接收。控制命令如FIG..4.2.1 所示。

SPI 接口指令		
指令名称	指令格式	操作
R_REGISTER	000A AAAA	读配置寄存器。AAAAA 指出读操作的寄存器地址
W_REGISTER	001A AAAA	写配置寄存器。AAAAA 指出写操作的寄存器地址 只能在掉电模式或待机模式下操作。
R_RX_PAYLOAD	0110 0001	读 RX 有效数据：1-32 字节。读操作全部从字节 0 开始。 当读 RX 有效数据完成后，FIFO 寄存器中有效数据被清除。 应用于接收模式下。
W_RX_PAYLOAD	1010 0000	写 TX 有效数据：1-32 字节。写操作从字节 0 开始。 应用于发射模式下
FLUSH_TX	1110 0001	清除 TX FIFO 寄存器，应用于发射模式下。
FLUSH_RX	1110 0010	清除 RX FIFO 寄存器，应用于接收模式下。 在传输应答信号过程中不应执行此指令。也就是说，若传输应答信号过程中执行此指令的话将使得应答信号不能被完整的传输。
REUSE_TX_PL	1110 0011	应用于发射端 重新使用上一包发射的有效数据。当 CE=1 时，数据被不断重新发射。 在发射数据包过程中必须禁止数据包重利用功能。
NOP	1111 1111	空操作。可用来读状态寄存器。

前面提到的函数也要与这些命令配合使用,比如

```
SPI_RW_Reg(WRITE_REG + EN_RXADDR, 0x01);
SPI_Write_Buf(WRITE_REG + TX_ADDR, TX_ADDRESS, TX_ADR_WIDTH);
```
相关命令的宏定义如下：
```
#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
```
4.3 NRF24L01 相关寄存器地址的宏定义
```
#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
```
5 实际通信过程示波器图

对于 NRF24L01 的编程主要是通过命令（WRITE_REG，READ_REG 等等），
控制线CE、CSN）以及中断信号IRQ 共同完成的。
对于发射节点，如果使能 ACK 与IRQ 功能，则当通信成功以后（也就是发射节点收到了接收节点送回的ACK 信号）IRQ 线会置低。
对于接收节点，如果使能ACK与IRQ功能，则当通信成功以后（主要是根据Enhanced ShockBurst协议认为成功收到了有效数据宽度的数据）IRQ线会置低。
根据以上两种情况，用示波器抓了以下几个图形，分别介绍如下：

1） 发射节点CE 与IRQ 信号

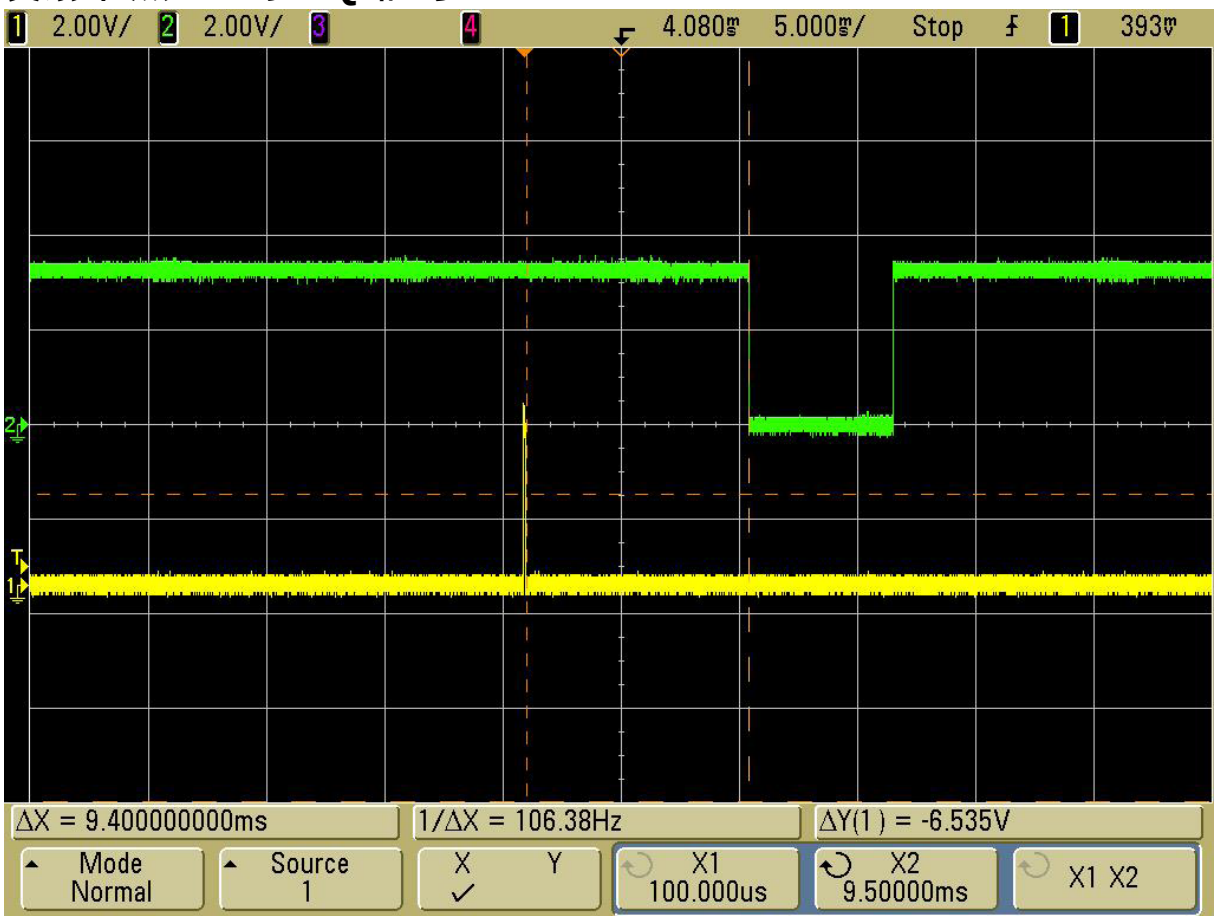


FIG5.1

黄色信号是CE，绿色信号是IRQ，当把节点配置为发射节点以后，将要传送的数据通过 SPI_Write_Buf(WRITE_REG + RX_ADDR_P0, TX_ADDRESS, TX_ADR_WIDTH)函数送到发送FIFO 缓冲区。CE 为高超过10us，缓冲区中的数据通过无线向外发出。如果使能IQR 的全部功能（TX_DS,RX_DS,MAX_RT）当发送节点收到接收节点发来的ACK（表示接收节点成功收到信号）或是达到最大发射次数，IRQ 会变为低电平，同时CONFIG 的相关标志位（）会置1。清除标志位（向CONFIG 的标志位写
1）以后，IRQ 又变为高电平。
从FIG5.1 可以看出，CE 置高后将近10msIRQ 才置低。IRQ 置低是由于达

到最大发射次数（MAX_RT=1）,出现该情况可能是由于接收节点的配置与发射节点不符（例如发射接收频率不同，或者发射接收字节不等），或者根本就没有接收节点（例如接收节点就根本没上电）。

2） SCK 与IRQ 信号（发送成功）

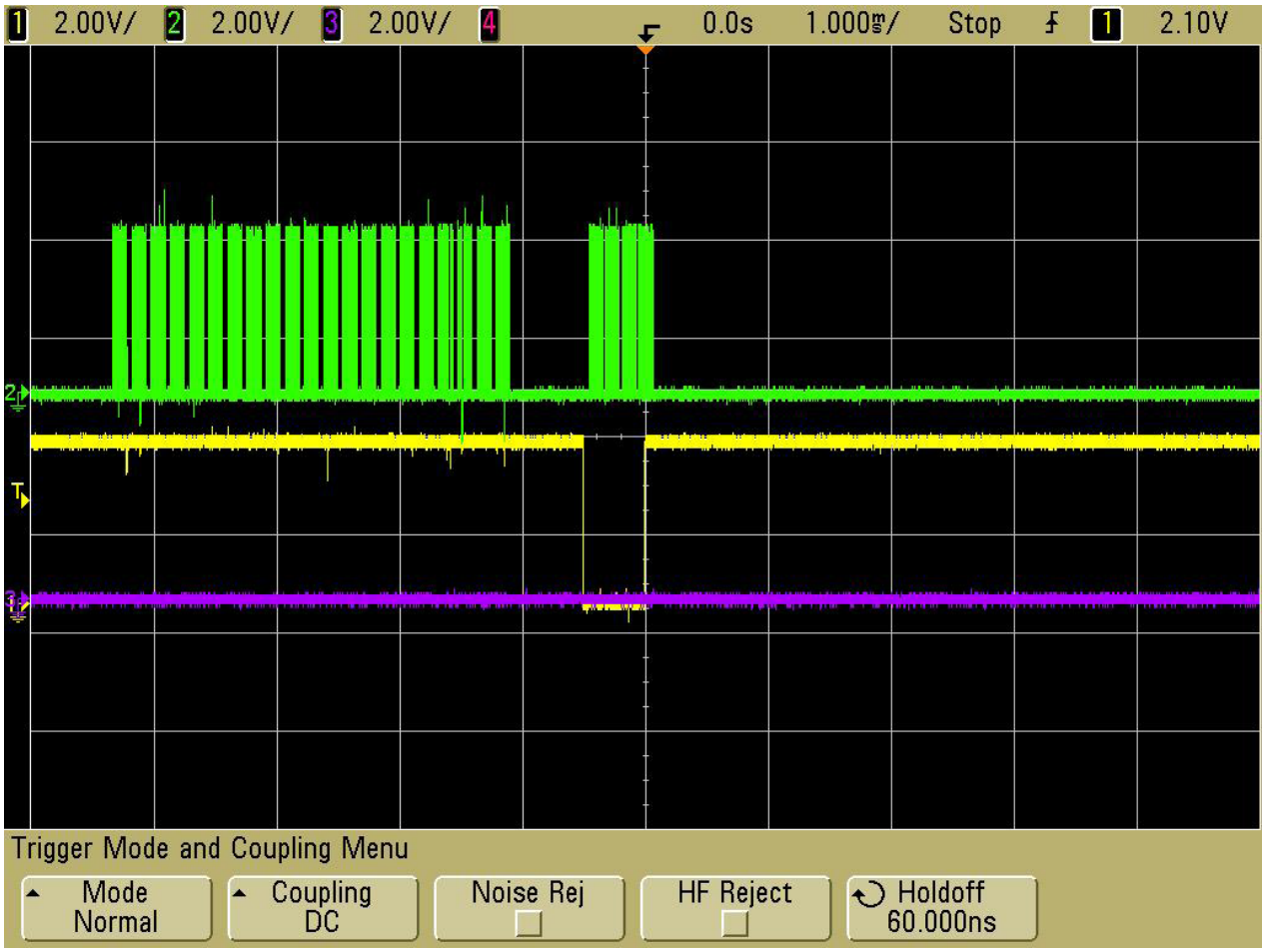


Fig5.2

中绿色信号是SCK，黄色信号是IRQ。第一批绿色信号表示节点的配置过程。MOSI 信号（Fig5.2 中未显示出）在SCK 的下降延送入24L01 节点。（配置一个寄存器需要两组SCK 信号，填充N 字节的BUFFER 需要N+1 组SCK 信号）。配置完信号以后，将 CE（Fig5.2 中未显出）置高，则24L01 开始发送（或接收）数据，当发送（或接收）完成以后（或是达到最大发射次数），IRQ 置低。单片机根据当时的状态进行相应的处理。第二批绿色信号表示单片机在 IRQ 为低时对24L01 的处理过程。可以是读FIFO（作为接收节点时），写FIFO（作为发射节点时），或是Reset 24L01（达到最大发射次数时）。

从 Fig5.2 可以看到，从第一批SCK 的最后一个信号到IRQ 置低大概需要1ms

（对比于Fig5.1 的12ms），说明通信成功（说明IRQ 不是MAX_RT 引起

的）。

3） SCK 与IRQ 信号（发送不成功）

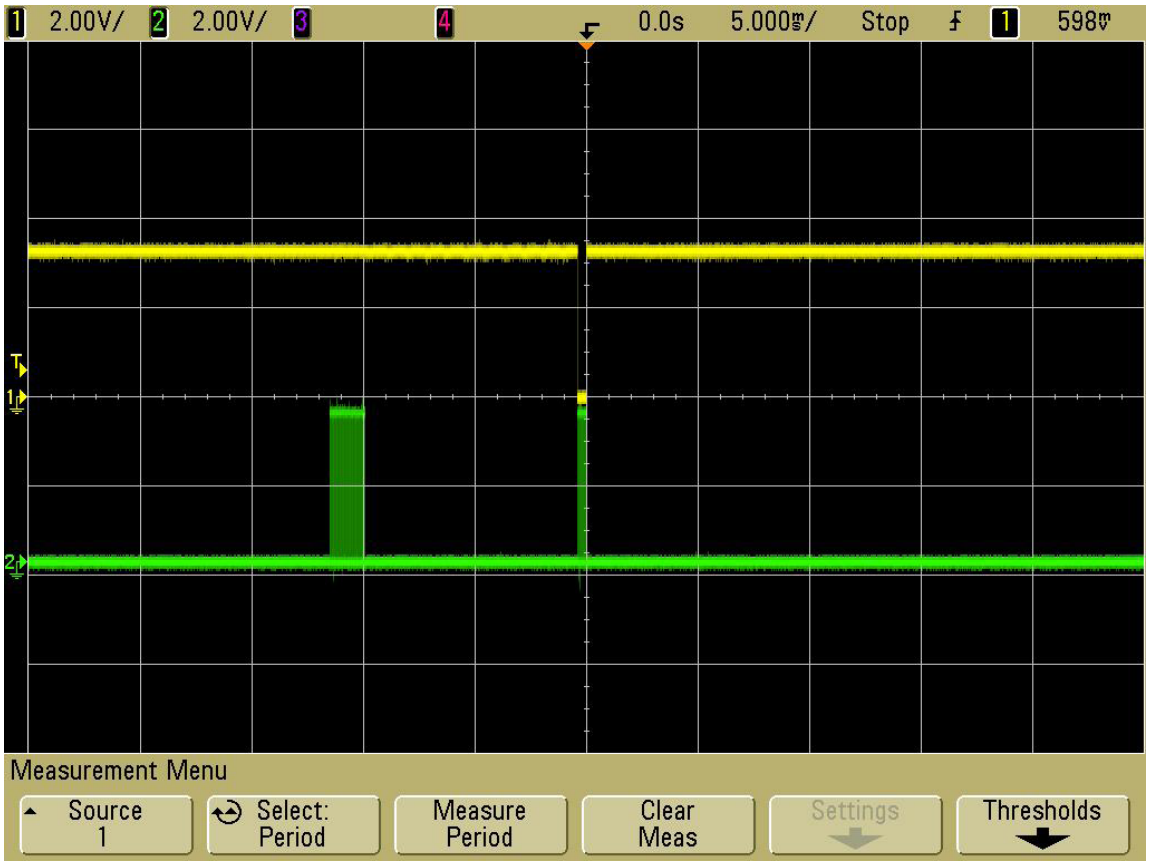


Fig5.3

Fig5.3 与Fig5.2 类似，只不过从第一批最后一个SCK 信号到IRQ 置低的时间间隔变为将近10ms，表明通信部不成功，IRQ 是由于达到最大发射次数引起的。

4） SCK、IRQ、CE 信号

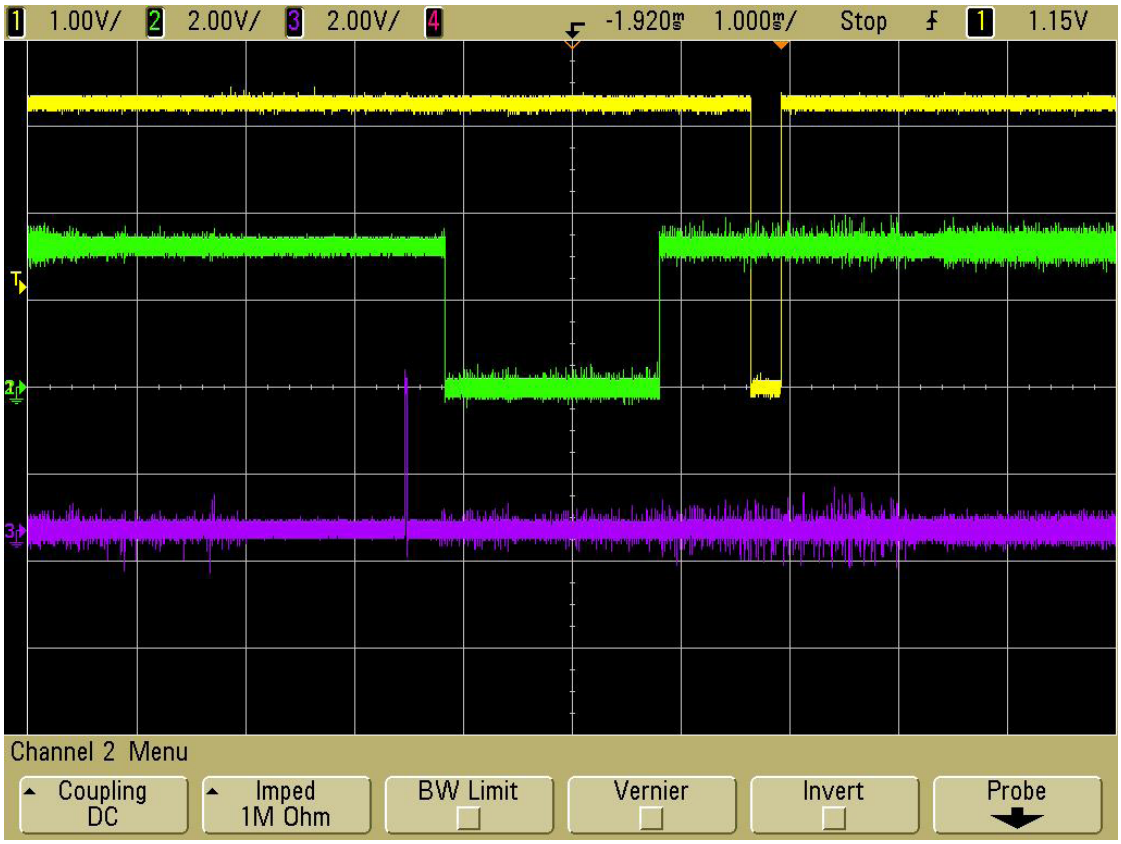


Fig5.4

Fig5.4 中紫色信号是发射端CE，绿色信号是接收端IRQ，黄色信号是发射端IRQ。Fig5.4 表示如下逻辑：

发射节点在配置完成以后（配制过程 Fig5.2 未显示），CE 置高，发射节点FIFO中的数据发出；接收节点成功接收到数据，IRQ 置低（从紫色信号与绿色信号之间的时间间隔可以判断出通信成功）；接收节点自动发射ACK（在发射和接收节点都使能ACK），发射节点收到ACK 后IRQ 置低，表示发送成功。不同通信环境可能造成发射节点的 IRQ 与接收节点的IRQ 产生将对的相位变化（表现在示波器上面就是黄色信号靠近绿色信号或者远离绿色信号）。出现这种情况主要是由于不同的通信环境造成接收端发送的ACK 信号要重发几次才能被发送端收到。

THE END!