无线信号采集软件编程

金向东

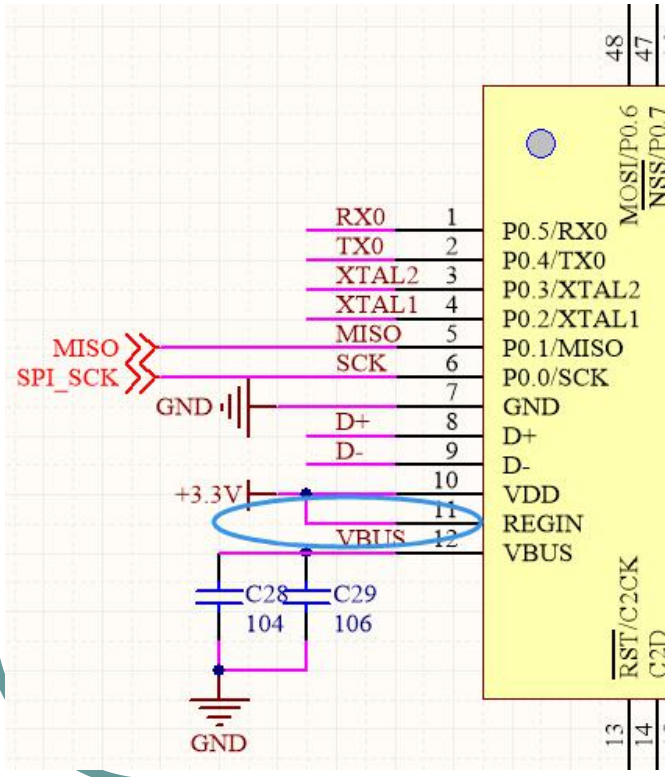
2020/2

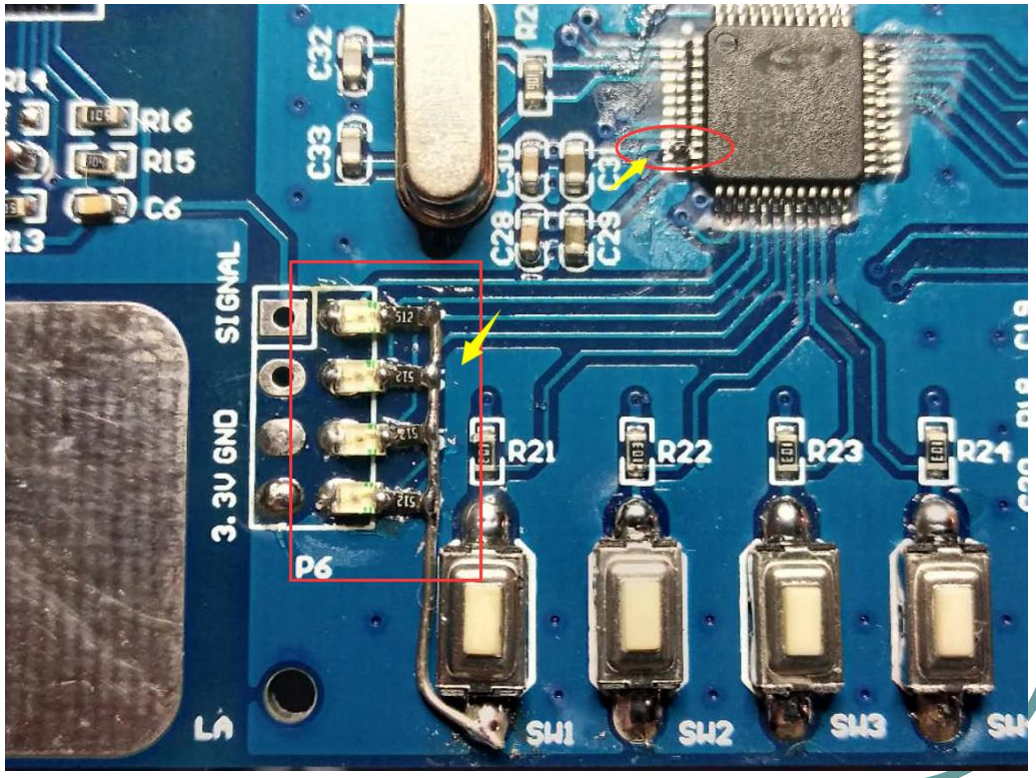
内容

- 硬件调整
- 编程环境搭建
- 单片机编程基础（DEMO）
  - 延时方式实现流水灯
  - 定时中断方式实现流水灯
  - 按键方式实现LED灯控制
  - 串行（Uart）通信实现
  - ADC及数据传输（Uart）
  - SPI接口及nRF905射频收发
- 移动ECG软件实现

硬件调整

- 1、增加4个LED，用于编程学习时状态指示
- 2、连接单片机的10，11脚，（设计疏漏）



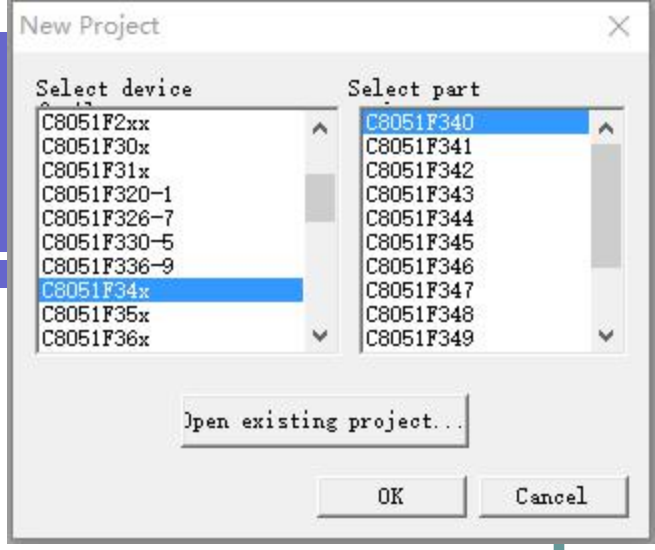


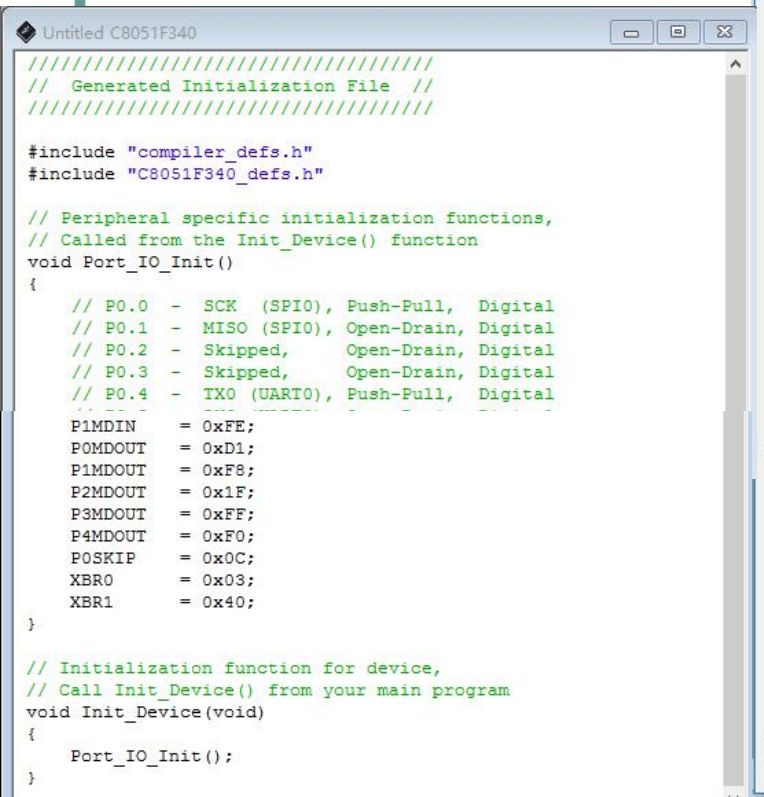
编程环境搭建

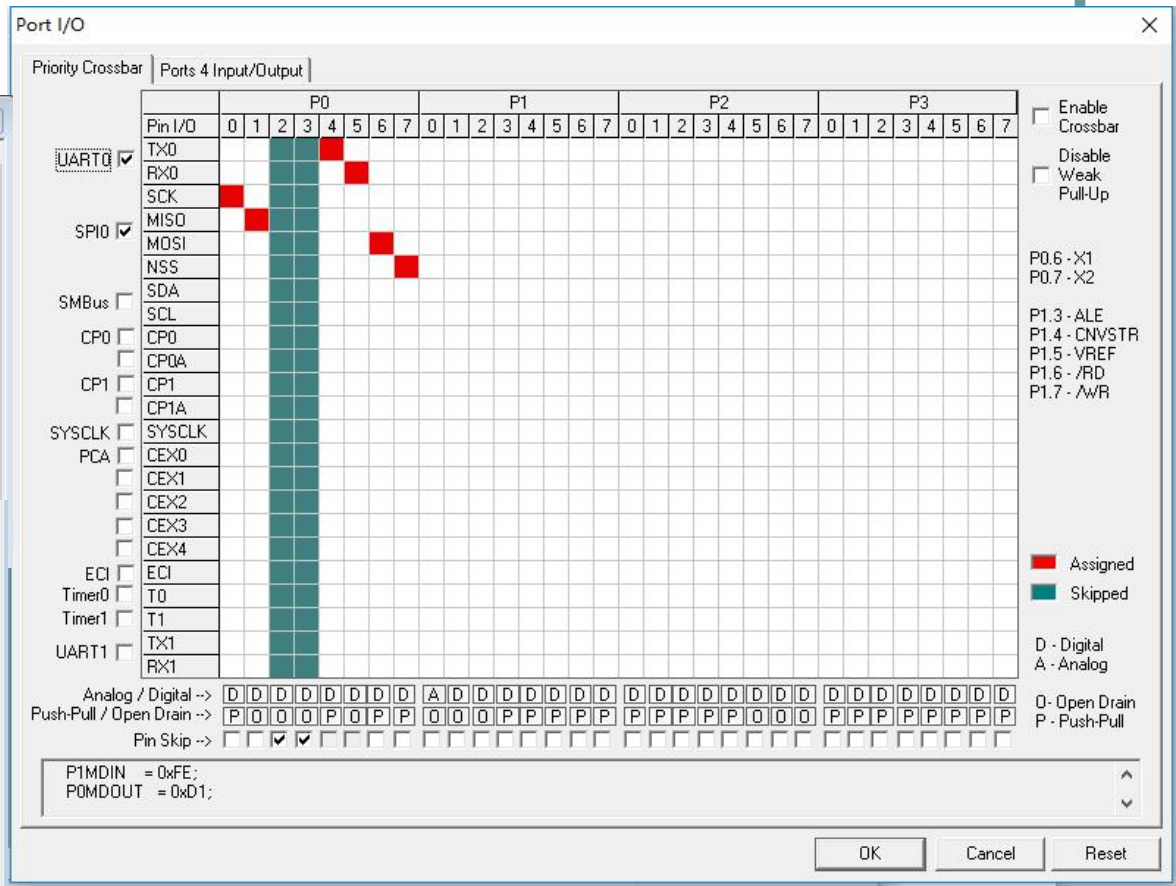
- 采用单片机芯片：C8051F340
- 软件相关网页：https://cn.silabs.com/products/development-tools/software/8-bit-8051-microcontroller-software
  - Silicon Labs 集成开发环境 (IDE) 是一个完整独立的软件程序，其包含项目管理器、源代码编辑器、源代码级调试器和其他实用工具。
  - 配置向导 2 实用工具通过自动生成初始化源代码以配置和启用大多数设计项目所需的片上资源，从而帮助加快开发。
  - 闪存编程工具支持将您的代码下载到设备并执行其他内存操作，而无需使用 Silicon Labs IDE。
  - Silicon Labs 和 Keil 软件相结合，为 Keil µVision 的最先进开发平台中的C8051F系列 MCU 提供支持。

采用配置向导 2进行接口规划

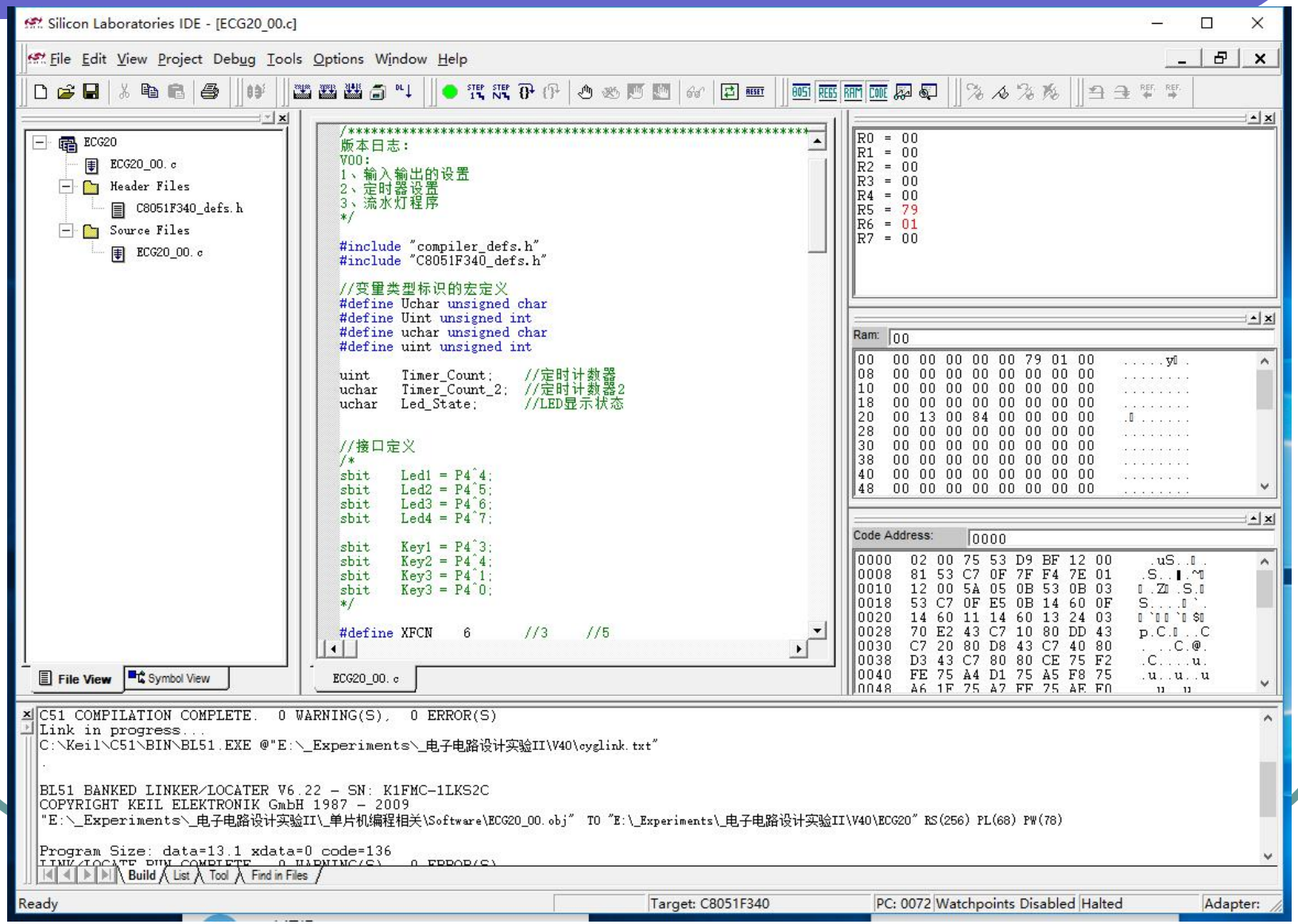
- 选择芯片：C8051F340
- 设置端口位置，及输出形式
- 生成端口配置代码



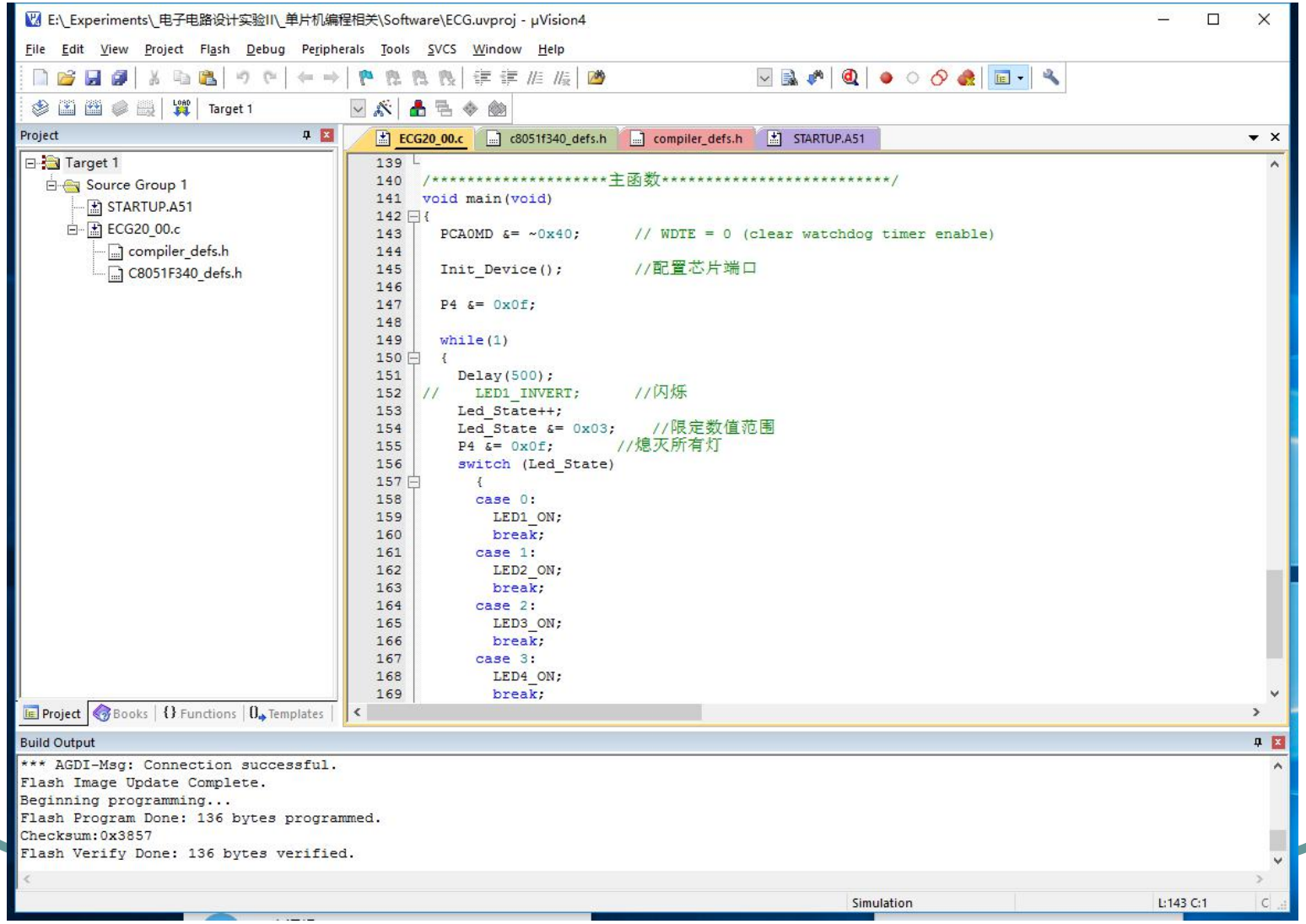




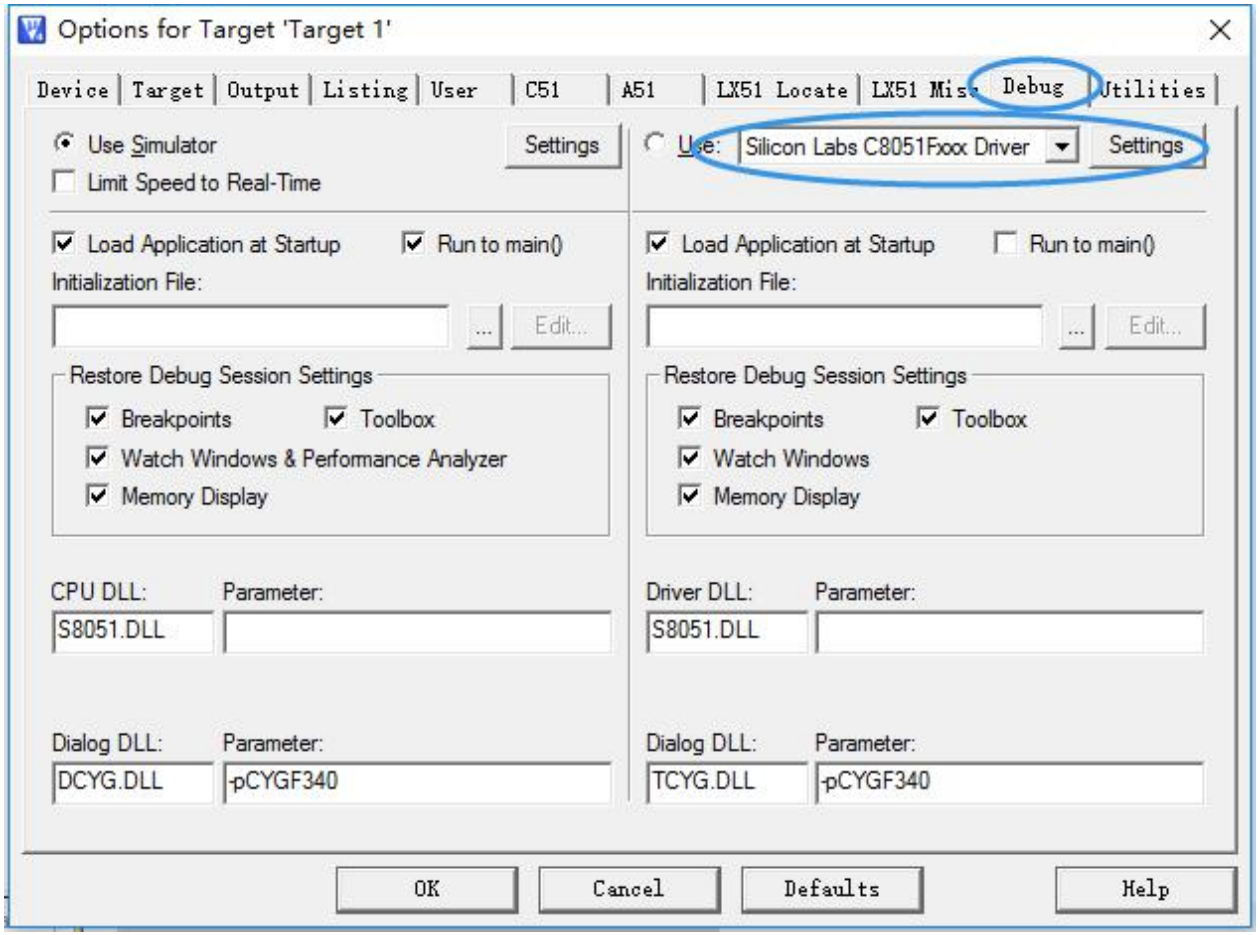
SiliconLab IDE



Keil


E:_Experiments_电子电路设计实验II_单片机编程相关\Software\ECG.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
Source Group 1
STARTUP.A51
ECG20_00.c
compiler_defs.h
C8051F340_defs.h
Project Books Functions Templates
ECG20_00.c
c8051f340_defs.h
compiler_defs.h
STARTUP.A51
139
140 /*********************主函数*************************/
141 void main(void)
142 {
143 PCA0MD &= ~0x40; // WDTE = 0 (clear watchdog timer enable)
144
145 Init_Device(); //配置芯片端口
146
147 P4 &= 0x0f;
148
149 while(1)
150 {
151 Delay(500);
152 // LED1_INVERT; //闪烁
153 Led_State++;
154 Led_State &= 0x03; //限定数值范围
155 P4 &= 0x0f; //熄灭所有灯
156 switch (Led_State)
157 {
158 case 0:
159 LED1_ON;
160 break;
161 case 1:
162 LED2_ON;
163 break;
164 case 2:
165 LED3_ON;
166 break;
167 case 3:
168 LED4_ON;
169 break;
Build Output
*** AGDI-Msg: Connection successful.
Flash Image Update Complete.
Beginning programming...
Flash Program Done: 136 bytes programmed.
Checksum:0x3857
Flash Verify Done: 136 bytes verified.
Simulation
L:143 C:1

Keil仿真器设置（需要先安装SiliconLab驱动）



程序结构

- **周期级（信息扫描与输出）程序：**

  主要用于收集系统来自各设备端口以及系统总体的信息，如：有线用户的摘/挂机、有线用户的拨号、中继端口的振铃、无线端口的FSK拨号、超时信息、RS232串口信息等等。除RS232串口信息是通过串口中断实现以外，其他的信息都是在定时中断程序内完成的。

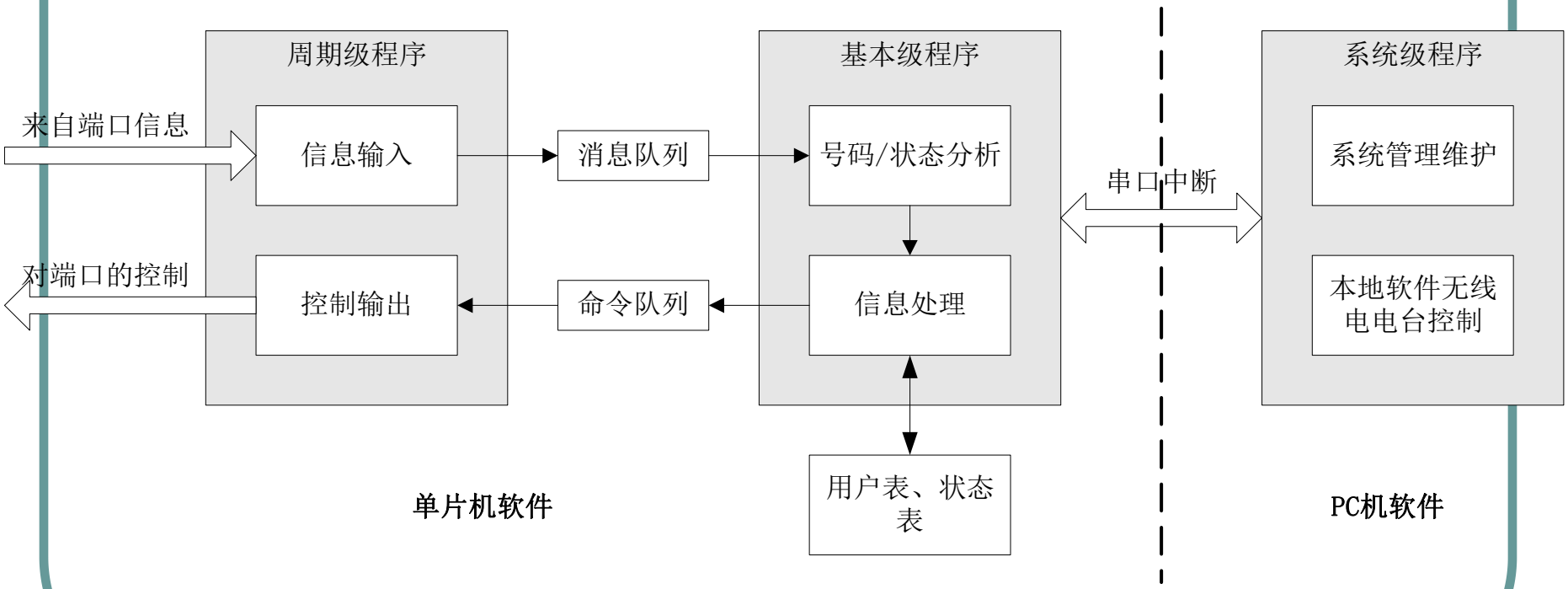
- **基本级（信息处理）程序：**

  是系统中最为复杂的部分，由主程序来完成。信息的处理要依据信息本身、信息来自的设备端口以及该端口的状态等综多因素来进行。由于设备端口的类型各不相同，同时，每一类端口的状态又多种多样，所以在信息处理程序设计时，必定采用状态迁移图的方式来进行。

- **系统级（系统管理维护）程序：**

  由单片机和PC机联合完成。主要实现用户的权限管理，对监听端口及其音量的控制、强行建链/拆链的控制、人工转接控制等等。PC机主要实现控制管理，单片机是具体执行者。

软件系统总体结构框图


周期级程序
基本级程序
系统级程序
来自端口信息
信息输入
消息队列
号码/状态分析
串口中断
系统管理维护
对端口的控制
控制输出
命令队列
信息处理
本地软件无线电电台控制
用户表、状态表
单片机软件
PC机软件

3-2-1 周期级软件设计

- 周期级软件用于周期地收集系统来自各设备端口以及系统本身的信息（如：有线用户的摘/挂机状态、有线用户的拨号、中继端口的振铃、无线端口的FSK拨号、定时信息等等），生成各种事件队列供基本级软件接收处理；
- 接收来自基本级软件的命令（如：中继占用、交换网络的控制、FSK拨号输出、信号音输出等等），去控制相应的端口。
- 位处理技术

定时中断程序

计数器1（1.11ms定时）

T1-0	1	
T1-1		1
模块	FSK接收模块	FSK发送模块

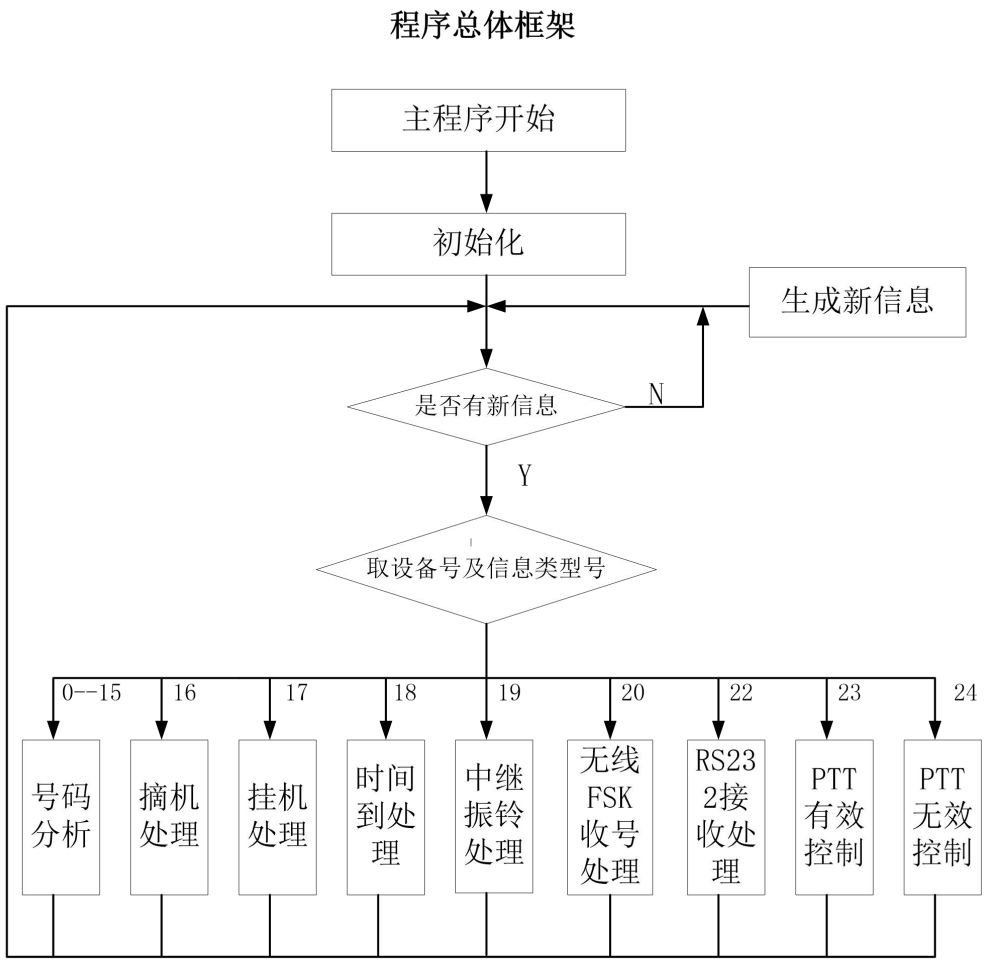
计数器2（10ms定时）

T2-0	1	1	1					
T2-1	1			1				
T2-2	1				1			
T2-3	1					1		
T2-4	1		1					
T2-5	1			1				
T2-6	1						1	
T1-7	1							1
模块	脉冲收号	定时器	摘挂机检测	DTMF收号	振铃检测	线路音产生	线路音控制	交换网控制

3-2-2 基本级软件设计

- 是系统中最为复杂的部分，由主程序来完成。
- 信息的处理要依据信息本身、信息来自的设备端口以及该端口的状态等众多因素来进行。
- 由于设备端口的类型各不相同，同时，每一类端口的状态又多种多样，所以在信息处理程序设计时，必定采用状态迁移图的方式来进行。

基本级软件的总体流程图



单片机编程基础-延时方式实现流水灯（DEMO-00）

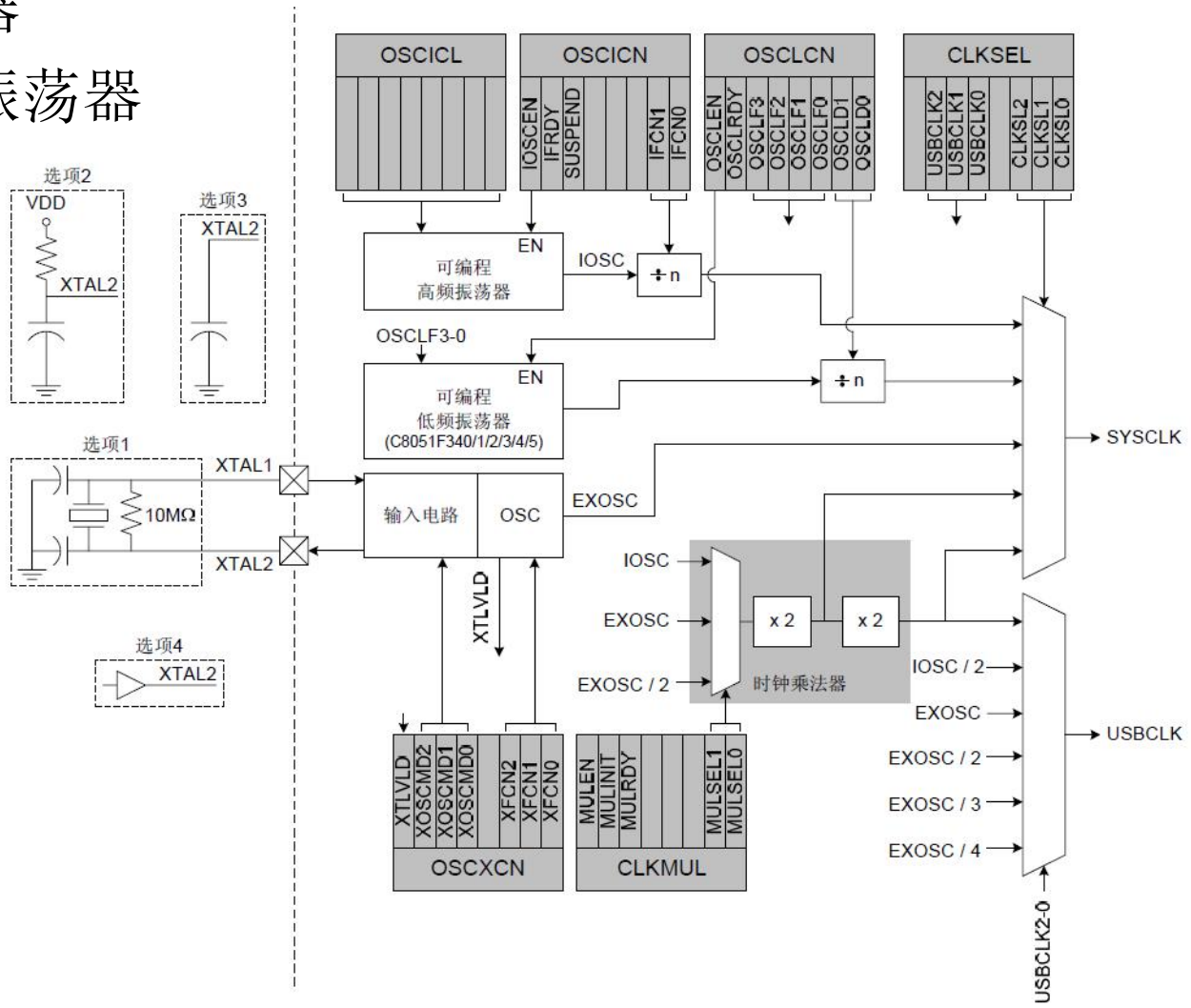
V00：
1、输入输出的设置
2、流水灯程序

- 系统端口配置
- 时钟源配置
- 延时函数实现
- 端口控制

```
/*******************主函数*************************/
void main(void)
{
    PCA0MD &= ~0x40;            // WDTE = 0 (clear watchdog timer enable)

    Init_Device();              //配置芯片端口

    P4 &= 0x0f;

    while(1)
    {
        Delay(500);
//      LED1_INVERT;            //闪烁
        Led_State++;
        Led_State &= 0x03;      //限定数值范围
        P4 &= 0x0f;             //熄灭所有灯
        switch (Led_State)
            {
            case 0:
                LED1_ON;
                break;
            case 1:
                LED2_ON;
                break;
            case 2:
                LED3_ON;
                break;
            case 3:
                LED4_ON;
                break;
            default:
                break;
            }

    } « end while 1 »
```

时钟模块

- 内部振荡器
- 外部晶体振荡器



定时中断方式实现流水灯（DEMO-01）

V01：
1、设置定时器中断
2、利用定时器实现流水灯

- 定时器设置
- 中断程序运行
  - 每隔0.5S，timer_flag=1
- 标志/信息传递

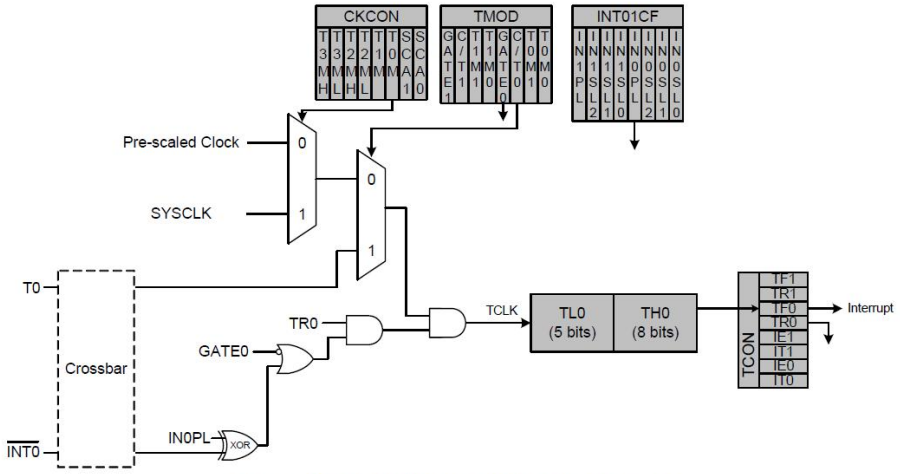


Figure 21.1. T0 Mode 0 Block Diagram

```
/*******************主函数*************************/
void main(void)
{
    PCA0MD &= ~0x40;                    // WDTE = 0 (clear watchdog timer
                                        // enable)
    Init_Device();

    P4 &= 0x0f;

    while(1)
    {
//      Delay(1000);
    if (timer_flag)                     //依据时间标志
        {
        timer_flag = 0;
        Led_State++;
        Led_State &= 0x03;              //限定数值范围
        P4 &= 0x0f;                     //熄灭所有灯
        switch (Led_State)              //依据状态控制LED
            {
            case 0:
                LED1_ON;
                break;
            case 1:
                LED2_ON;
                break;
            case 2:
                LED3_ON;
                break;
            case 3:
                LED4_ON;
                break;
            default:
                break;
            }
        } « end if timer_flag »
//      LED1_INVERT;

    } « end while 1 »

} « end main »
```

定时器初始化 & 定时中断程序

```
//-----------------------------------------------------------
// Timer0_Init
//-----------------------------------------------------------
// Return Value : None
// Parameters   : None
// T0设为16bit定时器，溢出时间1ms
//
//-----------------------------------------------------------
void Timer0_Init (void)
{
//   OSCICN = 0x83;      // Set the internal oscillator to
                         // 12 MHz
   //T0,模式设定
   TR0 = 0;              //停止计数
   ET0 = 1;              //允许中断
   PT0 = 1;              //高优先级中断
   TMOD = 0x01;          //#00000,0001,16位定时模式

   TH0 = 0;
   TL0 = 0;
   TR0 = 1;              //开始运行
}
```

```
155: /********************* Timer0中断函数************************/
156: void timer0_int (void) interrupt 1
157: {
158:     TH0 = 0xfc;
159:     TL0 = 0x18;          //1ms定时中断
160:
161: //  Key_Scan();
162: //  Run_treat();
163:
164:     ++Timer_Count;
165:     if (Timer_Count > 500)
166:     {
167:         Timer_Count = 0;
168:         timer_flag = 1;
169: //      LED1_INVERT;
170:     }
171:
172: }
173:
```

按键方式实现LED灯控制（DEMO-02）

V02:
1、取键值
2、Key1-4，分别控制Led1-4；

- 按键扫描方式
- 按键判断
- 按键处理

```
/****************** 按键处理 ************************/
void    Key_treat()
{
    if (Key1_press_flag)
    {
        LED1_INVERT;
        Key1_press_flag = 0;
    }

    if (Key2_press_flag)
    {
        LED2_INVERT;
        Key2_press_flag = 0;
    }

    if (Key3_press_flag)
    {
        LED3_INVERT;
        Key3_press_flag = 0;
    }

    if (Key4_press_flag)
    {
        LED4_INVERT;
        Key4_press_flag = 0;
    }

} « end Key_treat »

/******************主函数************************/
void main(void)
{
    PCA0MD &= ~0x40;                    // WDTE = 0 (clear watchdog timer
                                        // enable)
    Init_Device();

    P4 &= 0x0f;

    while(1)
    {
        Key_treat();
    }

}
```

```
/******************** 键盘扫描函数************************/
void    Key_Scan()
{
    P4_IN = P4;

    if (!Key1)              //键扫描1，检测下降沿
    {
        if(Key1_back)
        {
            Key1_press_flag = 1;
        }
    }

    if (!Key2)              //键扫描2
    {
        if(Key2_back)
        {
            Key2_press_flag = 1;
        }
    }
[]
    if (!Key3)              //键扫描3
    {
        if(Key3_back)
        {
            Key3_press_flag = 1;
        }
    }

    if (!Key4)              //键扫描4
    {
        if(Key4_back)
        {
            Key4_press_flag = 1;
        }
    }

    Key1_back = Key1;
    Key2_back = Key2;
    Key3_back = Key3;
    Key4_back = Key4;
} « end Key_Scan »
```

串行（Uart）通信实现（DEMO-03）

V03:
1、串行通信，速率：115200bps；8bit数据，1停止位，无奇偶校验；
2、每0.5秒钟发送一次数据；
3、编写发送字符串函数和发送数据函数；V031

- UART帧结构
- 发送
- 接收
- 发送接收处理

```
/*----------------------------
Initial Uart
----------------------------*/
void UartInit(void)   //115200bps@12MHz
{
    CKCON |=0X08;       //T1使用系统时钟
    SCON0 = 0x10;       //8位数据，可变波特率
    TMOD &= 0x0F;       //清定时器1模式位
    TMOD |= 0x20;       //设定定时器1为8位自动重装载方式
    TL1 = 0xCC;         //设定定时器初值
    TH1 = 0xCC;         //
    ET1 = 0;            //禁止定时器1中断
    TR1 = 1;            //启动定时器1
}
```

```
/******************** 串行发送 *************************/
void Uart_Send()
{   //发送缓冲区写入发送内容
        Uart_Send_Buff[0] = 'V';
        Uart_Send_Buff[1] = 'o';
        Uart_Send_Buff[2] = 'l';
        Uart_Send_Buff[3] = 't';
        Uart_Send_Buff[4] = 'a';
        Uart_Send_Buff[5] = 'g';
        Uart_Send_Buff[6] = ':';
        Uart_Send_Buff[7] = (Temp/1000)+0x30;        //16进制 to 10进制
        Uart_Send_Buff[8] = ((Temp%1000)/100)+0x30;
        Uart_Send_Buff[9] = ((Temp%100)/10)+0x30;
        Uart_Send_Buff[10] = '.';                    //小数点
        Uart_Send_Buff[11] = (Temp%10)+0x30;
        Uart_Send_Buff[12] = 0x0d;
        Uart_Send_Buff[13] = 0x0a;
        Uart_Send_Count =14;
    //逐个字节发送缓冲区那内容
        for (Uart_Send_Count=0;Uart_Send_Count<14;Uart_Send_Count++)
        {
        SBUF0 = Uart_Send_Buff[Uart_Send_Count];
        while(TI0 ==0){}
        TI0 = 0;
        }
 } « end Uart_Send »

/*******************主函数*************************/
void main(void)
{
    PCA0MD &= ~0x40;                    // WDTE = 0 (clear watchdog timer
                                        // enable)
    Init_Device();
    P4 &= 0x0f;
    while(1)
    {
        Key_treat();
        if (timer_flag)
            {
            Temp++;
            Uart_Send();
            timer_flag =0;
            }
    }
}
```

Uart结构&波特率

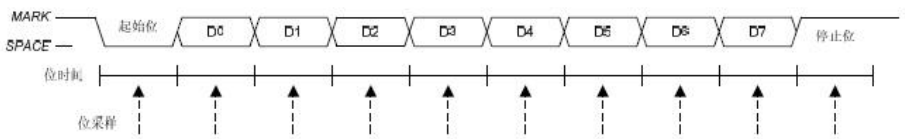


图 18.4　8 位 UART 时序图

$$UART波特率 = \frac{T1_{CLK}}{(256 - T1H)} \times \frac{1}{2}$$

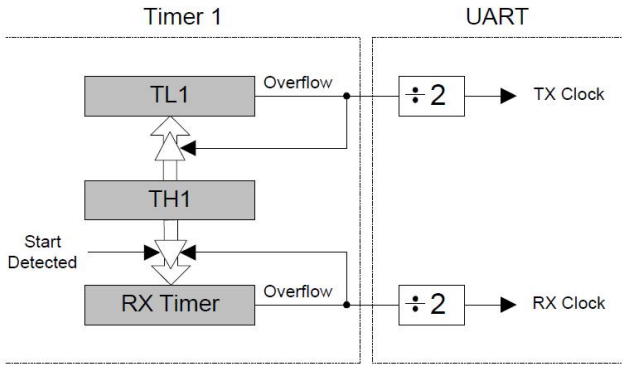


Figure 18.2. UART0 Baud Rate Logic

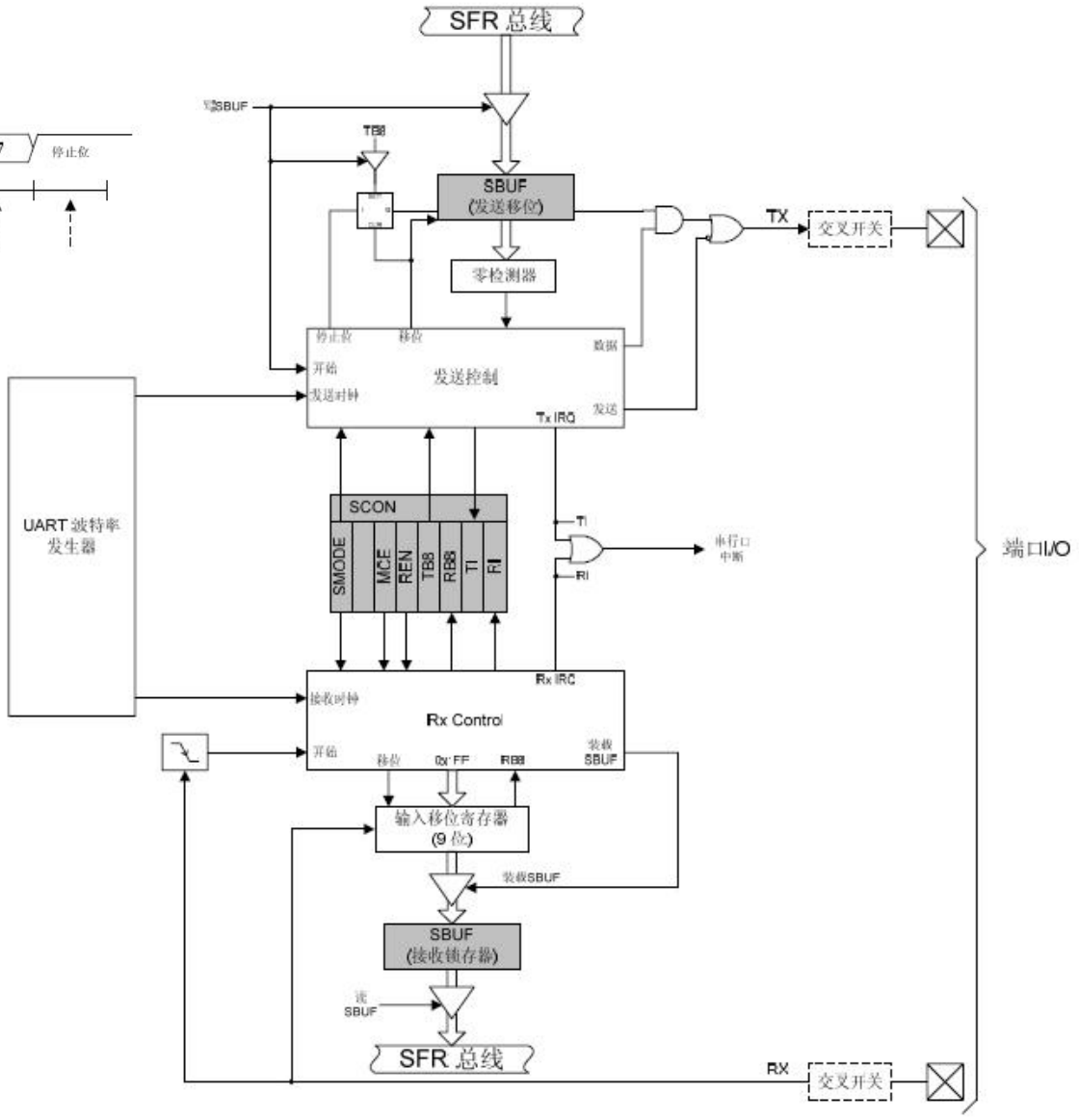


图 18.1　UART0 原理框图

ADC及数据传输（Uart）（DEMO-04）

- ADC结构
- ADC过程
- 信息传送

V04:
1、P1.1输入的电压经过ADC后，以115200bps速率通过Uart0发送；
2、ADC以Vcc为参考电压，通过AD0BUSY启动

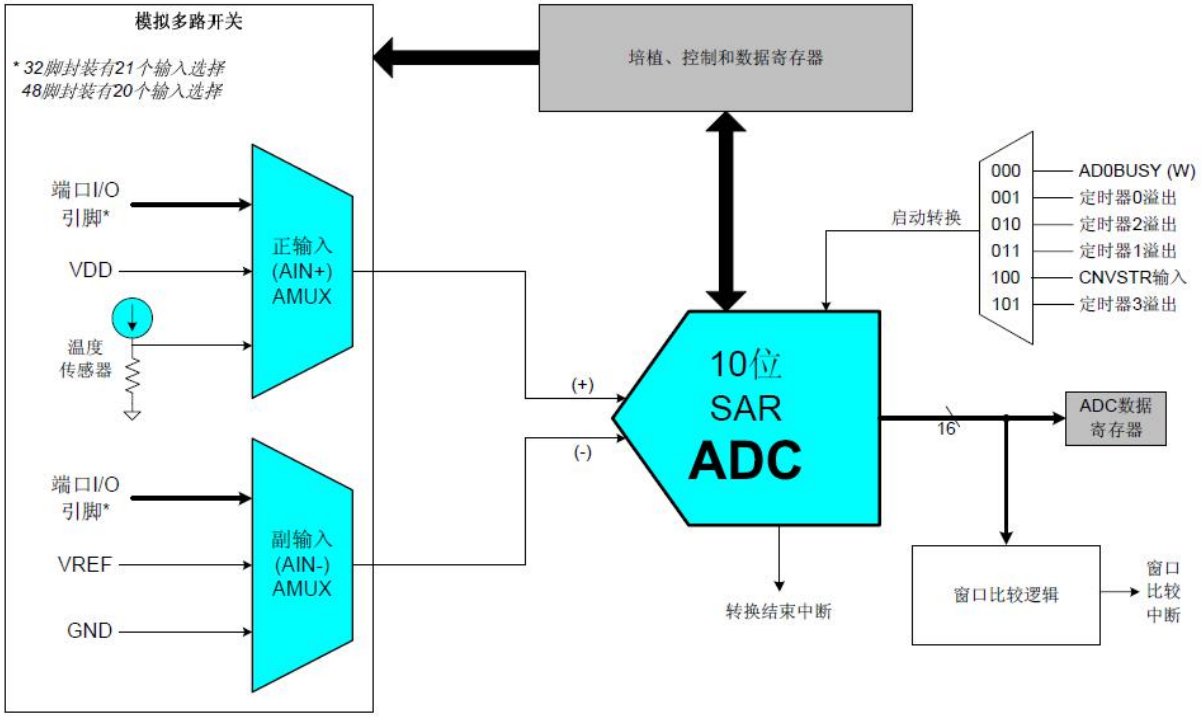
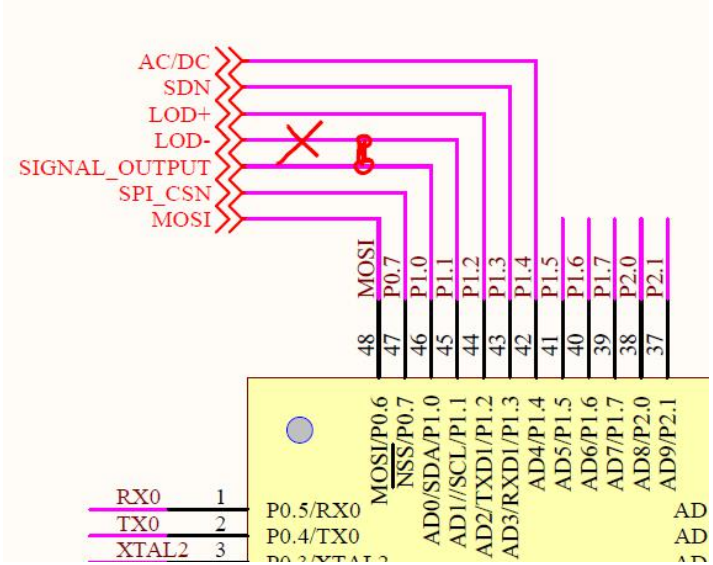


图1.9　10位ADC原理框图

ADC输入管脚调整

所选芯片不能采用P1.0作为ADC输入，改用P1.1作为ADC输入



SFR 定义 5.1　AMX0P: AMUX0 正输入通道选择寄存器

R	R	R	R/W	R/W	R/W	R/W	R/W	复位值
-	-	-	AMX0P4	AMX0P3	AMX0P2	AMX0P1	AMX0P0	00000000
位7	位6	位5	位4	位3	位2	位1	位0	SFR地址：0xBB

位 7-5：　未使用。读=000b，写=忽略。

位 4-0：　AMX0P4-0：AMUX0 正输入选择

AMX0P4-0	ADC0 正输入（32 脚封装）	ADC0 正输入（48 脚封装）
00000	P1.0	P2.0
00001	P1.1	P2.1
00010	P1.2	P2.2
00011	P1.3	P2.3
00100	P1.4	P2.5
00101	P1.5	P2.6
00110	P1.6	P3.0
00111	P1.7	P3.1
01000	P2.0	P3.4
01001	P2.1	P3.5
01010	P2.2	P3.7
01011	P2.3	P4.0
01100	P2.4	P4.3
01101	P2.5	P4.4
01110	P2.6	P4.5
01111	P2.7	P4.6
10000	P3.0	保留
10001	P0.0	P0.3
10010	P0.1	P0.4
10011	P0.4	P1.1
10100	P0.5	P1.2
10101~11101	保留	保留
11110	温度传感器	温度传感器
11111	VDD	VDD

ADC代码

```
172: void ADC0_Init (void)
173: {
174:    ADC0CN = 0x00;                     // ADC0 disabled, normal tracking,
175:                                       // conversion triggered on "ADC0BUSY" set
176:    REF0CN = 0x08;                     // disable on-chip VREF and buffer
177:    AMX0P = 0x13;                      // ADC0 positive input = P1.1
178:    AMX0N = 0x1F;                      // ADC0 negative input = GND
179:                                       // i.e., single ended mode
180:    ADC0CF = ((SYSCLK/3000000)-1)<<3;  // set SAR clock to 3MHz
181:    ADC0CF &= ~0x04;                   // right-justify results
182:    EIE1 |= 0x08;                      // enable ADC0 conversion complete int.
183:    AD0EN = 1;                         // enable ADC0
184: }
```

```
331: void ADC0_ISR (void) interrupt 10
332: {
333:    unsigned long result=0;
334:    unsigned long mV;           // Measured voltage in mV
335:
336:    AD0INT = 0;                 // Clear ADC0 conv. complete flag
337:     result = ADC0;
338:     // The 10-bit ADC value is averaged across 2048 measurements.
339:     // The measured voltage applied to P1.1 is then:
340:     //                              Vref (mV)
341:     //   measurement (mV) =   --------------- * result (bits)
342:     //                          (2^10)-1 (bits)
343:     Temp =  result * 3300 / 1023;
344:     //   printf("P1.1 voltage: %ld mV\n",mV);
345: }
```

```
431: void main(void)
432: {
433:     Init_Device();
434:
435:     while(1)
436:     {
437:         Key_treat();
438:         if (timer_flag)
439:             {
440:             AD0BUSY = 1;
441:             while(AD0BUSY) {}
442:             SendString("Voltage:");
443:             SendUint(Temp);
444:             SendString("mV\n");
445: //          printf("P1.1 voltage: %ld mV\n",Temp);
446:             timer_flag =0;
447:             }
448:     }
449: }
```

SPI接口及nRF905射频收发（DEMO-05）

V05:
1、实现SPI配置，实现SPI的连续输出（中断方式）；V051
2、改为查询方式，写入与读出共用一个函数；V052
3、对nRF905进行设置，并通过Uart输出其状态：V053
4、实现了射频信号的连续发送，可用于发射功率的测试，发射频率：433.0MHz；V054
5、实现了频率、功率的按键调整：K1：T/R；K2：Down；K3：Up；K4：Power；V055
6、用于通信测试，按K1切换收发模式，LED1点亮时为发送模式，串口能观察到数据收发情况；
发送流程为：发送端发送数据包，接收端接收数据并校验，数据正确时返回数据包，
发送端收到接收端返回的数据，并校验正确，才表示通信正常；V056

SPI结构及初始化

```
//-----------------------------------------------------------------------------
// SPI0_Init
//-----------------------------------------------------------------------------
// Return Value : None
// Parameters   : None
// Configures SPI0 to use 4-wire Single Master mode. The SPI timing is
// configured for Mode 0,0 (data centered on first edge of clock phase and
// SCK line low in idle state).
//-----------------------------------------------------------------------------
void SPI0_Init()
{
   SPI0CFG    = 0x40;                  // Enable the SPI as a Master
                                       // CKPHA = '0', CKPOL = '0'
   SPI0CN     = 0x0D;                  // 4-wire Single Master, SPI enabled
   // SPI clock frequency equation from the datasheet
   SPI0CKR    = (SYSCLK/(2*SPI_CLOCK))-1;
   ESPI0 = 1;                          // Enable SPI interrupts
}
```

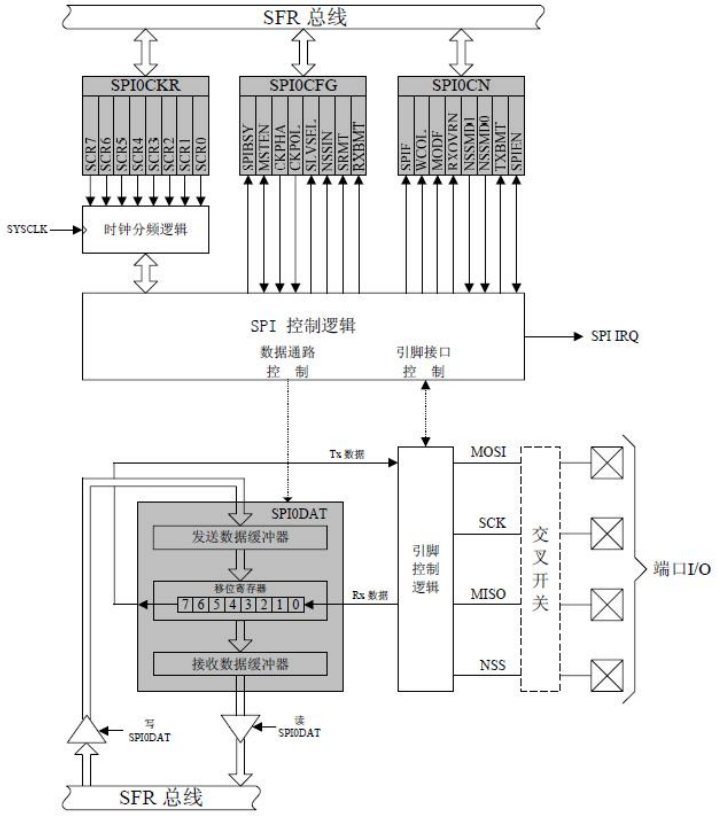


图 20.1 SPI 原理框图

SPI指令 for nRF905

Instruction set for the nRF905 SPI		
Instruction Name	**Instruction Format**	**Operation**
W_CONFIG (WC)	0000 AAAA	Write Configuration register. AAAA indicates which byte the write operation is to be started from. Number of bytes depends on start address AAAA.
R_CONFIG (RC)	0001 AAAA	Read Configuration register. AAAA indicates which byte the read operation is to be started from. Number of bytes depends on start address AAAA.
W_TX_PAYLOAD (WTP)	0010 0000	Write TX-payload: 1 – 32 bytes. A write operation always starts at byte 0.
R_TX_PAYLOAD (RTP)	0010 0001	Read TX-payload: 1 – 32 bytes. A read operation always starts at byte 0.
W_TX_ADDRESS (WTA)	0010 0010	Write TX-address: 1 – 4 bytes. A write operation always starts at byte 0.
R_TX_ADDRESS (RTA)	0010 0011	Read TX-address: 1 – 4 bytes. A read operation always starst at byte 0
R_RX_PAYLOAD (RRP)	0010 0100	Read RX-payload: 1 – 32 bytes. A read operation always starts at byte 0.
CHANNEL_CONFIG (CC)	1000 pphc cccc cccc	Special command for fast setting of CH_NO, HFREQ_PLL and PA_PWR in the CONFIGURATION REGISTER. CH_NO= ccccccccc, HFREQ_PLL = h PA_PWR = pp
STATUS REGISTER	N.A.	The content of the status register (S[7:0]) is always read to MISO after a high to low transition on CSN as shown in Figure 6. and Figure 7.

RF-CONFIG_REGISTER (R/W)		
Byte #	**Content bit[7:0], MSB = bit[7]**	**Init value**
0	CH_NO[7:0]	0110_1100
1	bit[7:6] not used, AUTO_RETRAN, RX_RED_PWR, PA_PWR[1:0], HFREQ_PLL, CH_NO[8]	0000_0000
2	bit[7] not used, TX_AFW[2:0] , bit[3] not used, RX_AFW[2:0]	0100_0100
3	bit[7:6] not used, RX_PW[5:0]	0010_0000
4	bit[7:6] not used, TX_PW[5:0]	0010_0000
5	RX_ADDRESS (device identity) byte 0	E7
6	RX_ADDRESS (device identity) byte 1	E7
7	RX_ADDRESS (device identity) byte 2	E7
8	RX_ADDRESS (device identity) byte 3	E7
9	CRC_MODE,CRC_EN, XOF[2:0], UP_CLK_EN, UP_CLK_FREQ[1:0]	1110_0111

```
// SPI命令字 SPI Commands
#define nCMD_W_CONFIG          0x00    // 写配置寄存器
#define nCMD_R_CONFIG          0x10    // 写配置寄存器
#define nCMD_W_TX_PAYLOAD      0x20    // 写TXFIFO
#define nCMD_R_TX_PAYLOAD      0x21    // 读TXFIFO
#define nCMD_W_TX_ADDRESS      0x22    // 写TX地址
#define nCMD_R_TX_ADDRESS      0x23    // 读TX地址
#define nCMD_R_RX_PAYLOAD      0x24    // 读RXFIFO
#define nCMD_CHANNEL_CONFIG    0x80    // 频率设置
```

SPI读写时序

```
//-------------------------------------------------------
// SPI_Byte_Write_Read
//-------------------------------------------------------
// Return Value : SPI0DAT
// Parameters   : Send_Byte
//-------------------------------------------------------
uchar SPI_Byte_Write_Read (uchar Send_Byte)
{
    SPI0DAT = Send_Byte;          //发送1字节
    while(!SPIF);                 //等待发送结束
    SPIF = 0;
    return (SPI0DAT);             //返回接收数据
}
```

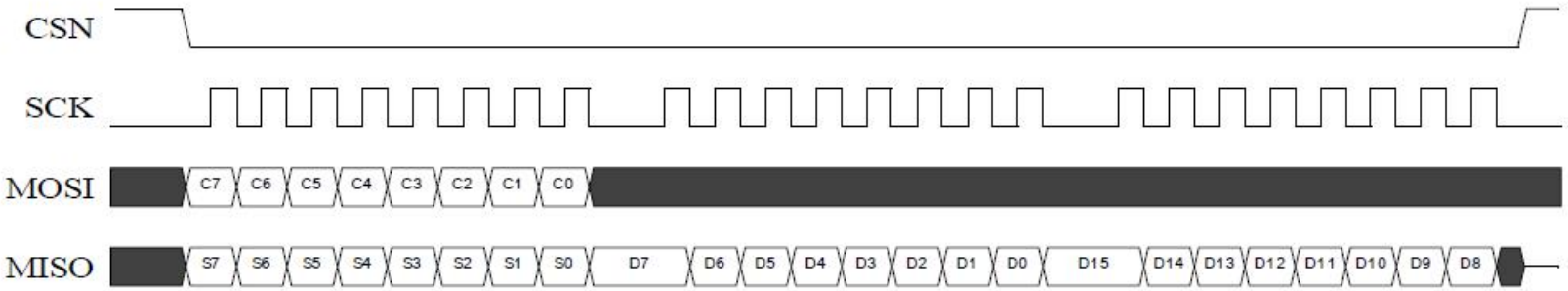


Figure 6. SPI read operation.

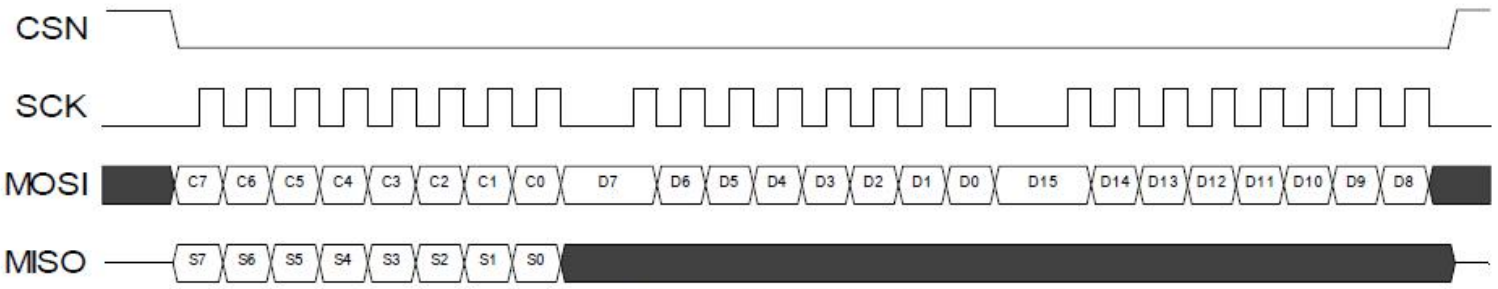


Figure 7. SPI write operation.

初始化nRF905

```
//-----------------------------------------------------------------------------
// 函数: n1P_Init_Dev()
//-----------------------------------------------------------------------------
// 描述: 初始化nRF905,并将其转换为接收状态
// 参数: 无
void n95_Init_Dev(uchar band,uint freq,uchar pwr)
{
    uchar i = 0;

    nPin_PWR_UP = 1;                                    // PWR_UP置高,nRF905进入上电模式
    nPin_TRX_CE = 0;                                    // TRX_CE置低,进入待机和SPI操作模式
    nPin_CSN = 0;                                       // CSN置低, 进入SPI操作模式
    SPI_Byte_Write_Read(nCMD_W_CONFIG);                 // 向nRF905发送"写配置寄存器命令"
    SPI_Byte_Write_Read(freq & 0xFF);                   // RF通道bit7:0
    SPI_Byte_Write_Read(nRCD_AUTO_RETRAN_disanble       // 禁用自动重发
                      | nRCD_RX_RED_PWR_disanble        // 禁用低功耗RX模式
                      | (pwr<<2)                        // 输出功率为10dBm
                      | (band<<1)                       // 工作频段设置
                      | (freq>>8) );                    // RF通道bit8
    SPI_Byte_Write_Read(nRCD_TX_AFW_4byte               // TX地址宽度为4byte
                      | nRCD_RX_AFW_4byte);             // RX地址宽度为4byte
    SPI_Byte_Write_Read(RF_DATA_WIDTH);                 // RX数据宽度
    SPI_Byte_Write_Read(RF_DATA_WIDTH);                 // TX数据宽度

    for(i=0; i<4; i++){
        SPI_Byte_Write_Read(n95_RF_Addr[i]);            // RX地址 byte0~3
    }

    SPI_Byte_Write_Read(nRCD_CRC_MODE_16crc             // 16bitCRC
                      | nRCD_CRC_EN_enable              // 启用CRC
                      | nRCD_XOF_16MHz                  // 外部晶振频率为16MHz
                      | nRCD_UP_CLK_EN_disanble         // 禁用外部时钟输出
                      | nRCD_UP_CLK_FREQ_4MHz);         // 时钟输出为4MHz

    nPin_CSN = 1;                                       // CSN置高, 退出SPI操作模式
    nPin_TX_EN = 0;                                     // TX_EN置低 ,进入接收模式
    nPin_TRX_CE = 1;                                    // TRX_CE置高,进入工作模式
} « end n95_Init_Dev »
```

发送数据包

```
//-----------------------------------------------------------------------------
// 函数: n1P_Turn_TX()
//-----------------------------------------------------------------------------
// 描述: 通过nRF905发送数据,数据发送结束后返回接收模式
// 参数: p 发送数据存放地址
void n95_Sendout(uchar *p)
{
    uchar i=0;
    nPin_PWR_UP = 1;                        // PWR_UP置高,nRF905进入上电模式
    nPin_TRX_CE = 0;                        // TRX_CE置低,进入待机和SPI操作模式
    nPin_TX_EN = 1;                         // TX_EN置高 ,进入发送模式

    nPin_CSN = 0;                           // CSN置低, 进入SPI操作模式
    SPI_Byte_Write_Read(nCMD_W_TX_ADDRESS);  // 向nRF905写入"写TX地址"指令
    for(i=0; i<4; i++){
        SPI_Byte_Write_Read(n95_RF_Addr[i]); // 写入TX地址 byte0~3,注意此处应与"nRCD_TX_AFW"的配置一致
    }
    nPin_CSN = 1;                           // CSN置高, 退出SPI操作模式
    Delay(1);
    nPin_CSN = 0;                           // CSN置低, 进入SPI操作模式
    SPI_Byte_Write_Read(nCMD_W_TX_PAYLOAD);  // 向nRF905写入"写TX数据"指令
    for(i=0; i<RF_DATA_WIDTH; i++){
        SPI_Byte_Write_Read(p[i]);           // 写入待发送数据
    }
    nPin_CSN = 1;                           // CSN置高, 退出SPI操作模式

    nPin_TRX_CE = 1;                        // TRX_CE置高,进入发送模式
    while(nPin_DR == 0);                    // 等待DR置高,发送完成
    nPin_TX_EN = 0;                         // TX_EN置低 ,进入接收模式
} « end n95_Sendout »
```

检查接收数据

```
//-------------------------------------------------------------------------------
// 函数: n95_Check_DR()
//-------------------------------------------------------------------------------
// 描述: 检查nRF905是否接收到数据,如接收到数据将数据存入接收缓冲区,并返回成功标志
// 参数: p      接收数据存放地址
//       return 接收成功标志,为1时表明数据接收成功
uchar n95_Check_DR(uchar *p)
{   uchar i=0;

    if(nPin_DR == 1){
        nPin_TRX_CE = 0;                          // TRX_CE置低,进入待机模式
        nPin_CSN = 0;                             // CSN置低, 进入SPI操作模式

        SPI_Byte_Write_Read(nCMD_R_RX_PAYLOAD); // 向nRF905写入"读取RXFIFO"指令

        for(i=0; i<RF_DATA_WIDTH; i++){
            p[i] =  SPI_Byte_Write_Read(0);       // 读取接收数据
        }

        nPin_CSN = 1;                             // CSN置高, 退出SPI操作模式
        nPin_TRX_CE = 1;                          // TRX_CE置高,进入工作模式
        return(1);                                // 返回接收成功标志
    }

    else    {
        return(0);                                // 返回未接收到数据标志
        }
} « end n95_Check_DR »
```

通信测试主函数

```
/*********************主函数*************************/
void main(void)
{
    Init_Device();
    n95_Init_IO();
    n95_Init_Dev(n95_band, n95_freq, n95_pwr);
    UartSend_RF_Set();

    while(1)    {
        Key_treat();

        if (timer_flag) {
            timer_flag =0;
            AD0BUSY = 1;                    //开始ADC转换
            while(AD0BUSY) {}

            //发送模式
            if (n95_rxtx ==1)   {
                //LED3_ON;
                n95_TX_Buff[28] = (Temp / 1000 % 10) +0x30;
                n95_TX_Buff[29] = (Temp / 100 % 10) +0x30;
                n95_TX_Buff[30] = (Temp / 10 % 10) +0x30;
                n95_TX_Buff[31] = (Temp %10) +0x30;

                if(n95_total>999){n95_total=0;n95_back =0;}else{n95_total++;}   // 计
                n95_Sendout(n95_TX_Buff);

                LED3_ON;
                Delay(100);
                LED3_OFF;
                SendString("Trans:");
                SendUint(n95_total);
                SendString(",");
                n95_rxtx = 2;
            }
        } « end if timer_flag »

        //接收模式
        if (n95_rxtx != 1)  {
            if(n95_Check_DR(n95_RX_Buff)){                          // 检查接收

                if(BUFF_Check(n95_TX_Buff,n95_RX_Buff,28)){         // 检查缓存
                    LED4_ON;
                    Delay(100);
                    LED4_OFF;

                    if(n95_rxtx==0){        // rx模式下的接收->回发数据
                        if(n95_total>999){n95_total=0;}else{n95_total++;}   // 计数
                        n95_Sendout(n95_TX_Buff);                           // 回传
                        LED3_ON;
                        Delay(100);
                        LED3_OFF;
                        SendString("Receiv:");
                        SendUint(n95_total);
                        SendString("\n");
                    }

                    if(n95_rxtx==2)         // tx模式下的接收->清除Error条件
                    {
                        n95_rxtx=1;
                        n95_back++;
                        LED4_ON;
                        Delay(100);
                        LED4_OFF;
                        SendString("Return:");
                        SendUint(n95_back);
                        SendString("\n");
                    }
                } « end if BUFF_Check(n95_TX_Buf... »
            } « end if n95_Check_DR(n95_RX_B... »
        } « end if n95_rxtx!=1 »
    } « end while 1 »
} « end main »
```

通信测试 & 拉距试验

发送端：

接收端：

移动ECG实现基本内容及评分

- 发送端（40分）
  - 调试ECG前端电路，生物测试能观察到明显的ECG信号（15）
  - 对外接ECG信号进行采样，采样频率100Hz（10）
  - 通过nRF905发送采样数据（10）
  - 通过Uart发送采样数据（5）
- 接收端（10分）
  - 定时检查nRF905有没收到数据，并获取数据（5分）
  - 通过Uart发送接收到的数据（5分）
- PC端（5分）
  - 利用窗口助手接收接收端传来的波形数据
  - 数据导入到Excel，绘制ECG波形（至少2个周期）

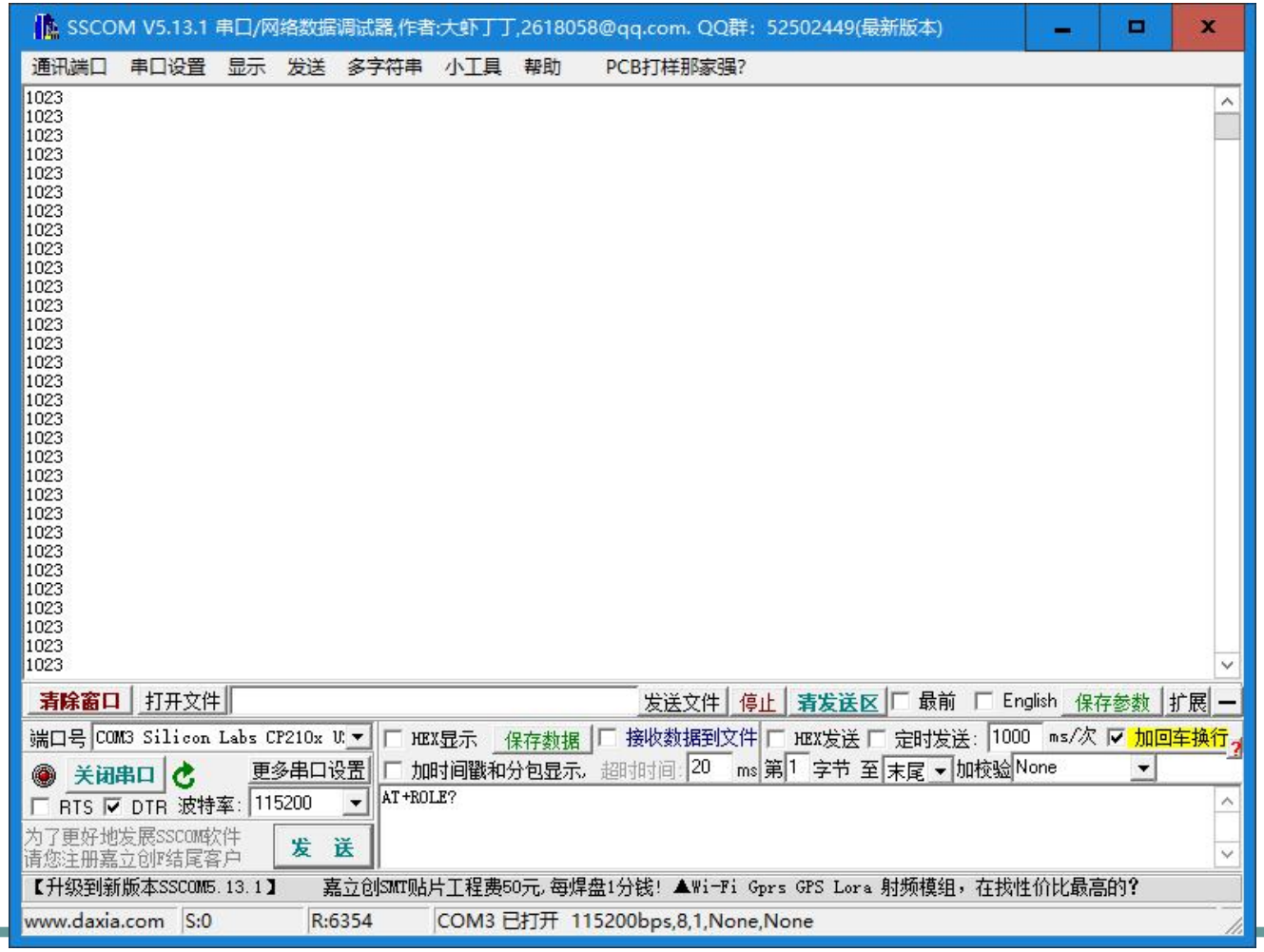
移动ECG实现提高内容及评分

- 设计制作螺旋天线，工作频率420-470MHz之间，回波损耗小于10dB，采用自制天线进行拉距试验，通信距离大于100M（10）
- 收、发端采用相同程序，收发模式通过按键K1切换，并有指示灯指示工作模式（5）
- 工作频率和发射功率可通过按键调整，频率范围423-472MHz，步距为1MHz，功率在-10、-2、6、10dBm之间循环切换，设置参数通过串口输出监视（10）
- 实现心率自动测量功能（10）
- 设计PC程序，实现ECG波形的自动绘制功能（5）
- 其他（5）

软件架构考虑

- 发送软件
  - 设计5ms的定时中断，在中断程序里边置位标志sample_flag,
  - 主函数判断sample_flag，启动一次AD转换
  - AD转换可以采用查询，或中断方式检测ADC是否结束
  - 若ADC结束，将ADC的结果取出
  - ADC结果，通过SPI接口发送给nRF905；
  - 再通过Uart发送
- 接收软件
  - 可以利用定时器产生一个读取nRF905状态的标志，如rev_flag
  - 主函数判断这个标志，取到ADC的结果
  - 将ADC结果通过

发送或接收模式下，PC通过Uart接收的数据



- 采集波形设置
- 拉距试验