

零死角玩转STM32

与野火同行 乐意惬意无边



原创教程，完全开源。



由浅入深，结合实操。



通俗易懂，详尽解读。



配套板子，全面玩转。



强强联合，不断更新。



野火团队 Wild Fire Team

0、友情提示

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

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



1、调试必备-串口（USART1）

当我们在学习一款 CPU 的时候，最经典的实验莫过于流水灯了，会了流水灯的话就基本等于学会操作 I/O 口了。那么在学会操作 I/O 之后，面对那么多的片上外设我们又应该先学什么呢？有些朋友会说用到什么就学什么，听起来这也不无道理呀。

但对于野火来说会把学习串口的操作放在第二位。在程序运行的时候我们可以通过点亮一个 LED 来显示代码的执行的状态，但有时候我们还想把某些中间量或者其他程序状态信息打印出来显示在电脑上，那么这时串口的作用就可想而知了。

1.1 异步串口通讯协议

阅读过《STM32 中文参考手册》的读者会发现，STM32 的串口非常强大，它不仅支持最基本的通用串口同步、异步通讯，还具有 LIN 总线功能(局域互联网)、IRDA 功能(红外通讯)、SmartCard 功能。

为实现最迫切的需求，利用串口来帮助我们调试程序，本章介绍的为串口最基本、最常用的方法，全双工、异步通讯方式。图 1-1 为串口异步通讯协议。

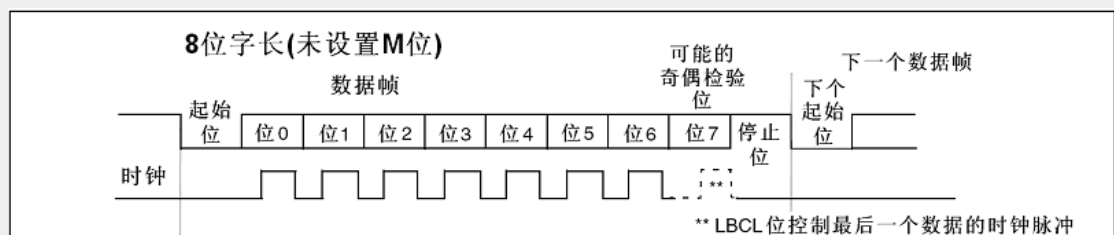


图 1-1 异步串口通讯协议

重温串口的通讯协议，我们知道要配置串口通讯，至少要设置以下几个参数：**字长(一次传送的数据长度)**、**波特率(每秒传输的数据位数)**、**奇偶校验位**、**还有停止位**。对 ST 库函数的使用已经上手的读者应该能猜到，在初始化串口的时候，必然有一个**串口初始化结构体**，这个结构体的几个成员肯定就是用来存储这些控制参数的。



所以野火建议大家设计板子时，尽量采用与 PC 相同的标准串口接法。

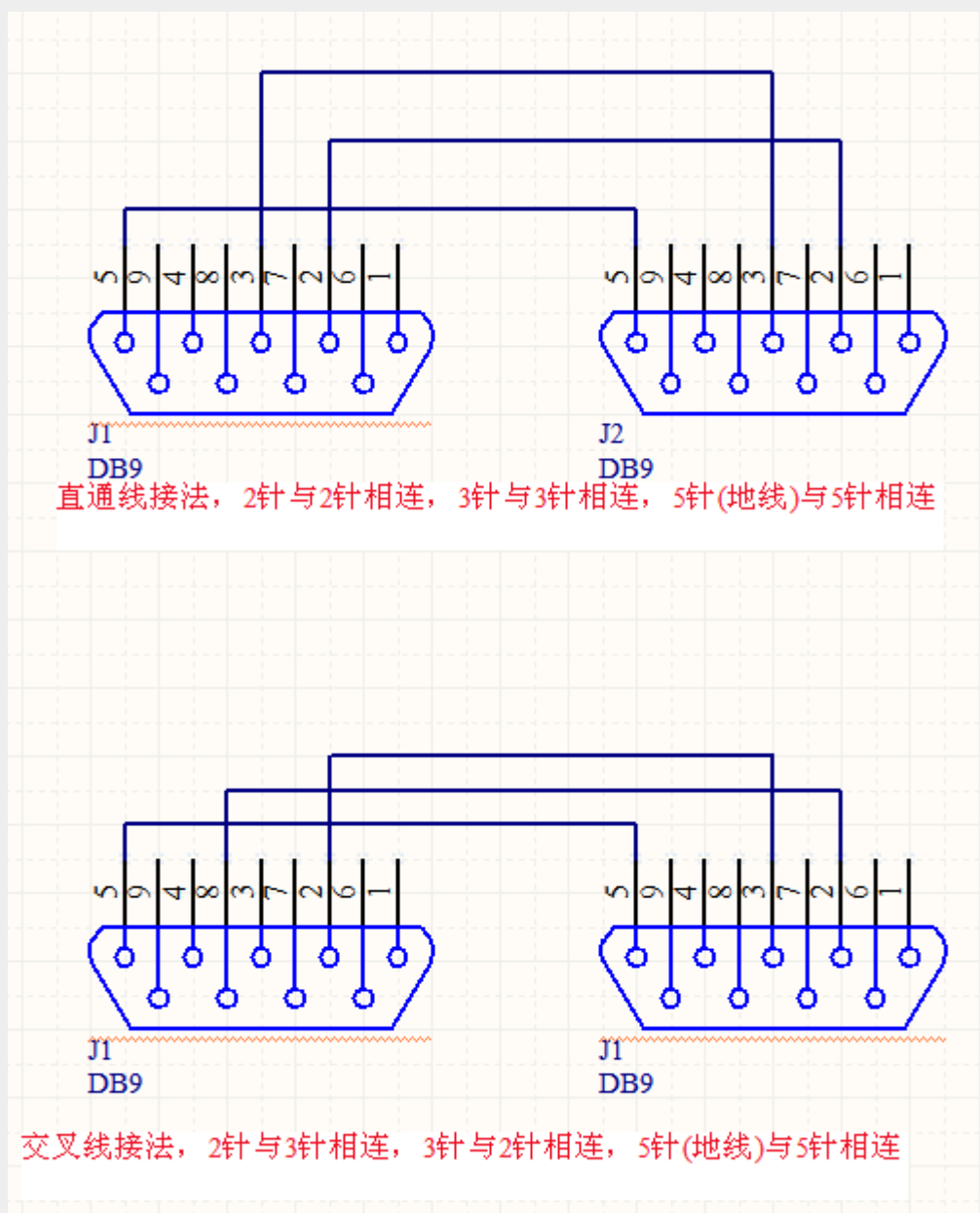


图 1-3 交叉线与直通线的区别

介绍直通线与交叉线的区别，一来是野火发现某些读者因为线的问题而花费了大量宝贵的时间。二来是介绍串口线的 **DIY** 方法。要实现基本的全双工异步通讯，只要 3 条线，分别为 **Rx**、**Tx**、和 **GND**。如果读者正在为直通线、交叉线、公头、母头不匹配而烦恼，可以根据自己的需要，参照图 1-3，用三根杜邦线连接即可。



1.3 串口工作过程分析

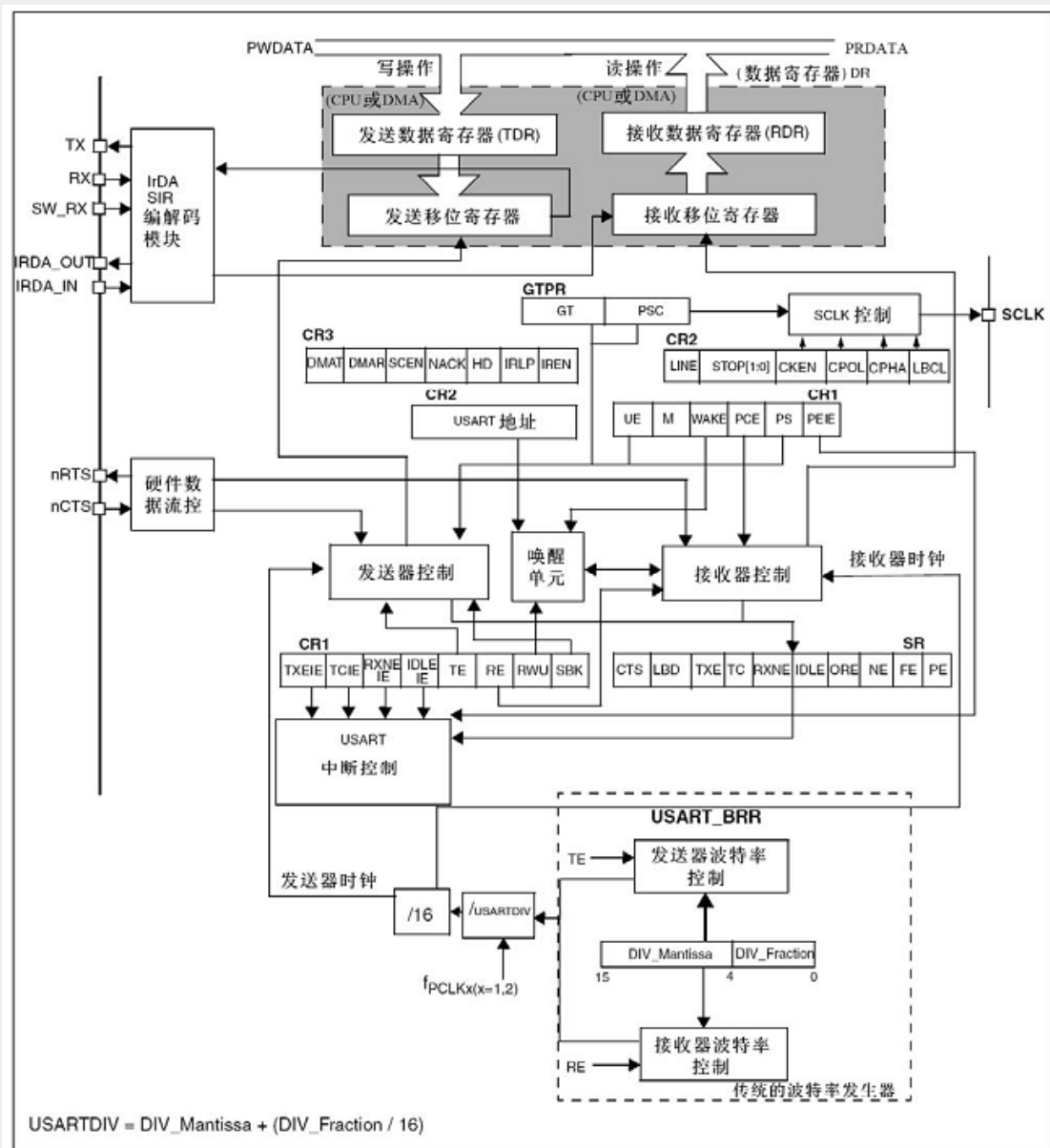


图 1-4 串口架构图

串口外设的架构图看起来十分复杂，实际上对于软件开发人员来说，我们只需要大概了解串口发送的过程即可。

从下至上，我们看到串口外设主要由三个部分组成，分别是波特率的控制部分、收发控制部分及数据存储转移部分。



1.3.1 波特率控制

波特率，即每秒传输的二进制位数，用 b/s (bps)表示，通过对时钟的控制可以改变波特率。在配置波特率时，我们向 *波特比率寄存器 USART_BRR* 写入参数，修改了串口时钟的 *分频值 USARTDIV*。*USART_BRR 寄存器* 包括两部分，分别是 *DIV_Mantissa* (USARTDIV 的整数部分) 和 *DIVFraction* (USARTDIV 的小数) 部分，最终，计算公式为

$$\text{USARTDIV} = \text{DIV_Mantissa} + (\text{DIVFraction} / 16)。$$

USARTDIV 是对串口外设的时钟源进行分频的，对于 *USART1*，由于它是挂载在 *APB2* 总线上的，所以它的时钟源为 f_{PCLK2} ；而 *USART2、3* 挂载在 *APB1* 上，时钟源则为 f_{PCLK1} ，串口的时钟源经过 *USARTDIV* 分频后分别输出作为 *发送器时钟* 及 *接收器时钟*，控制发送和接收的时序。

1.3.2 收发控制

围绕着发送器和接收器控制部分，有好多个寄存器：CR1、CR2、CR3、SR，即 USART 的三个 *控制寄存器* (Control Register) 及一个 *状态寄存器* (Status Register)。通过向寄存器写入各种 *控制参数*，来控制发送和接收，如奇偶校验位，停止位等，还包括对 USART 中断的控制；串口的 *状态* 在任何时候都可以从状态寄存器中查询得到。具体的控制和状态检查，我们都是使用库函数来实现的，在此就不具体分析这些寄存器位了。

1.3.3 数据存储转移部分

收发控制器根据我们的寄存器配置，对 *数据存储转移部分* 的 *移位寄存器* 进行控制。

当我们需要发送数据时，内核或 DMA 外设(一种数据传输方式，在下一章介绍)把数据从内存(变量)写入到 *发送数据寄存器 TDR* 后，*发送控制器* 将适时地自动把数据从 *TDR* 加载到 *发送移位寄存器*，然后通过 *串口线 Tx*，把数据一位一位地发送出去，在数据从 *TDR* 转移到 *移位寄存器* 时，会产生 *发送寄存器*



TDR 已空事件 TXE，当数据从 *移位寄存器* 全部发送出去时，会产生数据 *发送完成事件 TC*，这些事件可以在 *状态寄存器* 中查询到。

而 *接收数据* 则是一个 *逆过程*，数据从 *串口线 Rx* 一位一位地输入到 *接收移位寄存器*，然后自动地转移到 *接收数据寄存器 RDR*，最后用内核指令或 DMA 读取到内存(变量)中。

1.4 串口通讯实验分析

1.4.1 实验描述及工程文件清单

实验描述	重新实现 C 库中的 printf() 函数到串口 1，这样我们就可以像用 C 库中的 printf() 函数一样将信息通过串口打印到电脑，非常方便我们程序的调试。
硬件连接	PA9 - USART1(Tx) PA10 - USART1(Rx)
用到的库文件	<i>startup/start_stm32f10x_hd.c</i> <i>CMSIS/core_cm3.c</i> <i>CMSIS/system_stm32f10x.c</i> <i>FWlib/stm32f10x_gpio.c</i> <i>FWlib/stm32f10x_rcc.c</i> <i>FWlib/stm32f10x_usart.c</i>
用户编写的文件	<i>USER/main.c</i> <i>USER/stm32f10x_it.c</i> <i>USER/usart1.c</i>

1.4.2 配置工程环境

串口实验中我们用到了 *GPIO*、*RCC*、*USART* 这三个外设的库文件 *stm32f10x_gpio.c*、*stm32f10x_rcc.c*、*stm32f10x_usart.c*，所以我们要把这



个库文件添加进工程，新建用户文件 *usart1.c*。并在 *stm32f10x_conf.h* 中把相应的头文件的注释去掉。

```
1.  /**
2.  ****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  **** /
9.
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
12. #include "stm32f10x_usart.h"
```

1.4.3 main 文件

配置好要用的库的环境之后，我们就从 **main** 函数看起，层层剥离源代码。

```
1.  /**
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.      /* USART1 config 115200 8-N-1 */
10.     USART1_Config();
11.
12.     printf("\r\n this is a printf demo \r\n");
13.
14.     printf("\r\n 欢迎使用野火 M3 实验板:) \r\n");
15.
16.     USART1_printf(USART1, "\r\n This is a USART1_printf demo \r\n");
17.
18.     USART1_printf(USART1, "\r\n (\"__DATE__ \" -
19.     \" __TIME__ \") \r\n");
20.     for(;;)
21.     {
22.
23.     }
24. }
```

首先调用函数 *USART1_Config()*，函数 *USART1_Config()* 主要做了如下工作：

1. 使能了串口 1 的时钟
2. 配置好了 *usart1* 的 I/O



3. 配置好了 *usart1* 的工作模式，具体为波特率为 115200、8 个数据位、1 个停止位、无硬件流控制。即 115200 8-N-1。

1.4.4 USART 初始化配置

具体的 *USART1_Config()* 在 *usart1.c* 这个用户文件中实现：

```
1.  /*
2.  * 函数名: USART1_Config
3.  * 描述   : USART1 GPIO 配置,工作模式配置。115200 8-N-1
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void USART1_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     USART_InitTypeDef USART_InitStructure;
12.
13.     /* config USART1 clock */
14.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);
15.
16.     /* USART1 GPIO config */
17.     /* Configure USART1 Tx (PA.09) as alternate function push-pull */
18.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
19.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
20.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21.     GPIO_Init(GPIOA, &GPIO_InitStructure);
22.     /* Configure USART1 Rx (PA.10) as input floating */
23.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
24.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
25.     GPIO_Init(GPIOA, &GPIO_InitStructure);
26.
27.     /* USART1 mode config */
28.     USART_InitStructure.USART_BaudRate = 115200;
29.     USART_InitStructure.USART_WordLength = USART_WordLength_8b;
30.     USART_InitStructure.USART_StopBits = USART_StopBits_1;
31.     USART_InitStructure.USART_Parity = USART_Parity_No ;
32.     USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
33.     USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
34.     USART_Init(USART1, &USART_InitStructure);
35.     USART_Cmd(USART1, ENABLE);
36. }
```

在代码的第 14 行，调用了库函数 *RCC_APB2PeriphClockCmd()* 初始化了 *USART1* 和 *GPIOA* 的时钟，这是因为使用了 *GPIOA* 的 PA9 和 PA10 的默认复用 *USART1* 的功能，在使用复用功能的时候，要开启相应的功能时钟 *USART1*。

接下来，这段串口初始化代码分为两个部分，第一部分为 *GPIO* 的初始化，第二部分才是串口的模式、波特率的初始化。



1.4.4.1 GPIO 初始化

在错误！未找到引用源。中提到过，GPIO 具有默认的复用功能，在使用它的复用功能的时候，我们首先要把相应的 GPIO 进行初始化。此时我们使用的 GPIO 的复用功能为串口，但为什么是 PA9 和 PA10 用作串口的 Tx 和 Rx，而不是其它 GPIO 引脚呢？这是从《STM32F103CDE 增强型系列数据手册》的引脚功能定义中查询到的。

D12	C9	D2	42	68	101	PA9	I/O	FT	PA9	USART1_TX ⁽⁷⁾ TIM1_CH2 ⁽⁷⁾	
D11	D10	D3	43	69	102	PA10	I/O	FT	PA10	USART1_RX ⁽⁷⁾ / TIM1_CH3 ⁽⁷⁾	

图 1-5 GPIO 引脚功能说明图

选定了这两个引脚，并且 PA9 为 Tx，PA10 为 Rx，那么它们的 GPIO 模式要如何配置呢？Tx 为发送端，**输出引脚**，而且现在 GPIO 是使用**复用功能**，所以要把它配置为**复用推挽输出(GPIO_Mode_AF_PP)**；而 Rx 引脚为**接收端**，**输入引脚**，所以配置为**浮空输入模式 GPIO_Mode_IN_FLOATING**。如果在使用复用功能的时候，对 GPIO 的模式不太确定的话，我们可以从《STM32 参考手册》的 GPIO 章节中查询得到，见图 1-6。

USART引脚	配置	GPIO配置
USARTx_TX	全双工模式	推挽复用输出
	半双工同步模式	推挽复用输出
USARTx_RX	全双工模式	浮空输入或带上拉输入
	半双工同步模式	未用，可作为通用I/O
USARTx_CK	同步模式	推挽复用输出
USARTx_RTS	硬件流量控制	推挽复用输出
USARTx_CTS	硬件流量控制	浮空输入或带上拉输入

图 1-6 GPIO 复用功能模式设置

1.4.4.2 USART 初始化

从代码的第 28 行开始，进行 USART1 的初始化，也就是填充 USART 的初始化结构体。这部分内容，是根据串口通讯协议来设置的。

1. `.USART_BaudRate = 115200;`



波特率设置，利用库函数，我们可以直接这样配置波特率，而不需要自行计算 *USARTDIV* 的分频因子。在这里把串口的波特率设置为 115200，也可以设置为 9600 等常用的波特率，在《STM32 参考手册》中列举了一些常用的波特率设置及其误差，见图 1-7。如果配置成 9600，那么在和 PC 通讯的时候，也应把 PC 的串口传输波特率设置为相同的 9600。通讯协议要求两个通讯器件之间的波特率、字长、停止位奇偶校验位都相同。

波特率		$f_{PCLK} = 36MHz$			$f_{PCLK} = 72MHz$		
序号	Kbps	实际	置于波特率寄存器中的值	误差%	实际	置于波特率寄存器中的值	误差%
1	2.4	2.400	937.5	0%	2.4	1875	0%
2	9.6	9.600	234.375	0%	9.6	468.75	0%
3	19.2	19.2	117.1875	0%	19.2	234.375	0%
4	57.6	57.6	39.0625	0%	57.6	78.125	0%
5	115.2	115.384	19.5	0.15%	115.2	39.0625	0%
6	230.4	230.769	9.75	0.16%	230.769	19.5	0.16%
7	460.8	461.538	4.875	0.16%	461.538	9.75	0.16%
8	921.6	923.076	2.4375	0.16%	923.076	4.875	0.16%
9	2250	2250	1	0%	2250	2	0%
10	4500	不可能	不可能	不可能	4500	1	0%

图 1-7 STM32 常用波特率及其误差

2. *.USART_WordLength = USART_WordLength_8b;*

配置串口传输的字长。本例程把它设置为最常用的 8 位字长，也可以设置为 9 位。

3. *.USART_StopBits = USART_StopBits_1;*

配置停止位。把通讯协议中的停止位设置为 1 位。

4. *.USART_Parity = USART_Parity_No ;*

配置奇偶校验位。本例程不设置奇偶校验位。

5. *.USART_HardwareFlowControl= USART_HardwareFlowControl_None;*

配置硬件流控制。不采用硬件流。

硬件流，在 STM32 的很多外设都具有硬件流的功能，其功能表现为：当外设硬件处于准备好的状态时，硬件启动自动控制，而不需要软件再进行干预。



在串口这个外设的硬件流具体表现为：使用串口的 *RTS (Request to Send)* 和 *CTS (Clear to Send)* 针脚，当串口已经准备好接收新数据时，由硬件流自动把 **RTS** 针拉低(向外表示可接收数据)；在发送数据前，由硬件流自动检查 **CTS** 针是否为低(表示是否可以发送数据)，再进行发送。本串口例程没有使用到 **CTS** 和 **RTS**，所以不采用硬件流控制。

6. *USART_Mode = USART_Mode_Rx | USART_Mode_Tx;*

配置串口的模式。为了配置双线全双工通讯，需要把 **Rx** 和 **Tx** 模式都开启。

7. 填充完结构体，调用库函数 *USART_Init()*向寄存器写入配置参数。

8. 最后，调用 *USART_Cmd()* 使能 *USART1* 外设。在使用外设时，不仅要使能其时钟，还要调用此函数使能外设才可以正常使用。

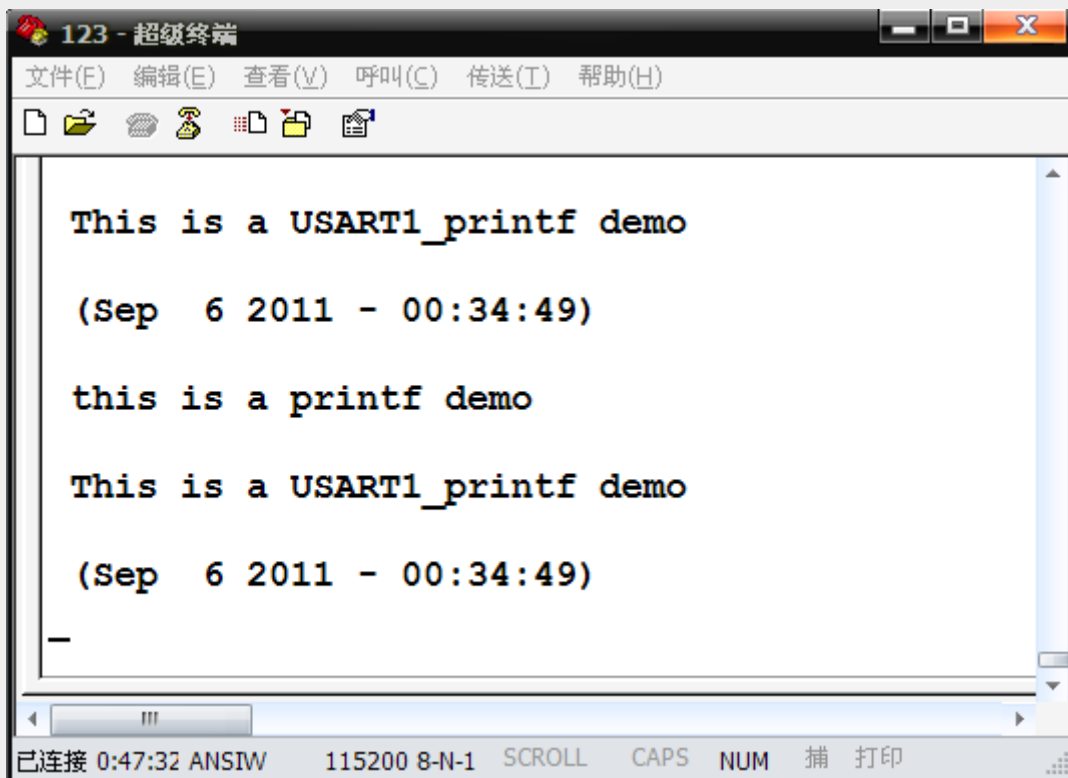
1.4.5 printf() 函数重定向

在 **main** 文件中，配置好串口之后，就通过下面的几行代码由串口往电脑里面的超级终端打印信息，打印的信息为一些字符串和当前的日期。

```
1. printf("\r\n this is a printf demo \r\n");
2.
3. printf("\r\n 欢迎使用野火 M3 实验板:) \r\n");
4.
5. USART1_printf(USART1, "\r\n This is a USART1_printf demo \r\n");
6.
7. USART1_printf(USART1, "\r\n (\"__DATE__ \" - \" __TIME__ \") \r\n");
```

下面是电脑超级终端的截图，从图可以看出程序是运行正确的。





调用这三个函数看似很简单，但在这三个函数的背后还得做些工作，我们先来看 `printf()` 这个函数。要想 `printf()` 函数工作的话，我们需要把 `printf()` 重新定向到串口中。**重定向**，是指用户可以自己重写 `c` 的库函数，当连接器检查到用户编写了与 `C` 库函数相同名字的函数时，优先采用用户编写的函数，这样用户就可以实现对库的修改了。

为了实现 **重定向** `printf()` 函数，我们需要重写 `fputc()` 这个 `c` 标准库函数，因为 `printf()` 在 `c` 标准库函数中实质是一个宏，最终是调用了 `fputc()` 这个函数的。

重定向的这部分工作，由 `usart.c` 文件中的 `fputc(int ch, FILE *f)` 这个函数来完成，这个函数具体实现如下：

```
1.  /*
2.  * 函数名: fputc
3.  * 描述   : 重定向 c 库函数 printf 到 USART1
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 由 printf 调用
7.  */
8.  int fputc(int ch, FILE *f)
9.  {
10.     /* 将 Printf 内容发往串口 */
```



```
11.     USART_SendData(USART1, (unsigned char) ch);
12. //   while (!(USART1->SR & USART_FLAG_TXE));
13.     while( USART_GetFlagStatus(USART1,USART_FLAG_TC) != SET);
14.     return (ch);
15. }
```

这个代码中调用了两个 ST 库函数。*USART_SendData()* 和 *USART_GetFlagStatus()* 其说明见图 1-8 及图 1-9。

重定向时，我们把 *fputc()* 的形参 *ch*，作为串口将要发送的数据，也就是说，当使用 *printf()*，它调用这个 *fputc()* 函数时，然后使用 ST 库的串口发送函数 *USART_SendData()*，把数据转移到发送数据寄存器 *TDR*，触发我们的串口向 PC 发送一个相应的数据。调用完 *USART_SendData()* 后，要使用 *while(USART_GetFlagStatus(USART1,USART_FLAG_TC) != SET)* 语句不停地检查串口发送是否完成的标志位 *TC*，一直检测到标志为完成，才进入下一步的操作，避免出错。在这段 *while* 的循环检测的延时中，串口外设已经由发送控制器根据我们的配置把数据从移位寄存器一位一位地通过串口线 *Tx* 发送出去了。

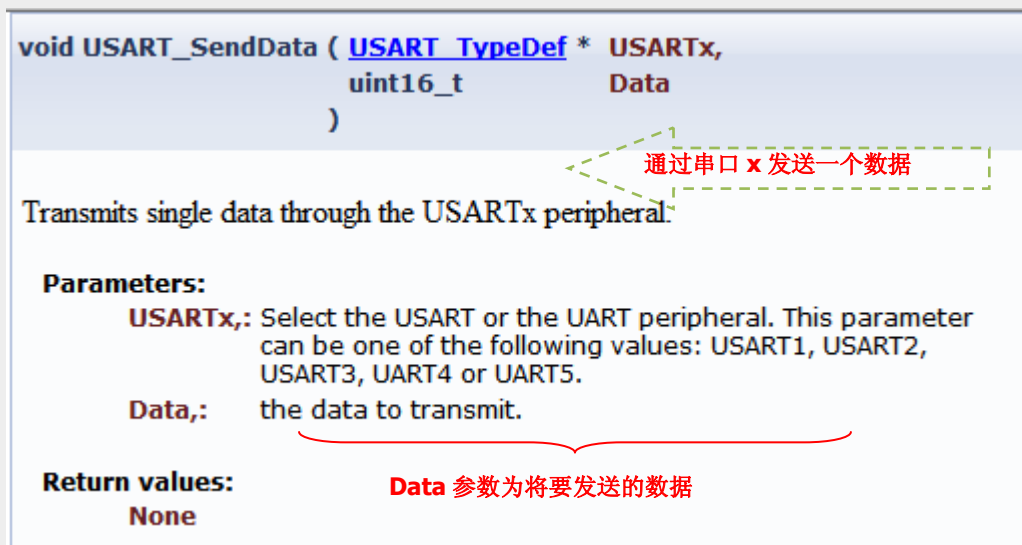


图 1-8 串口发送函数



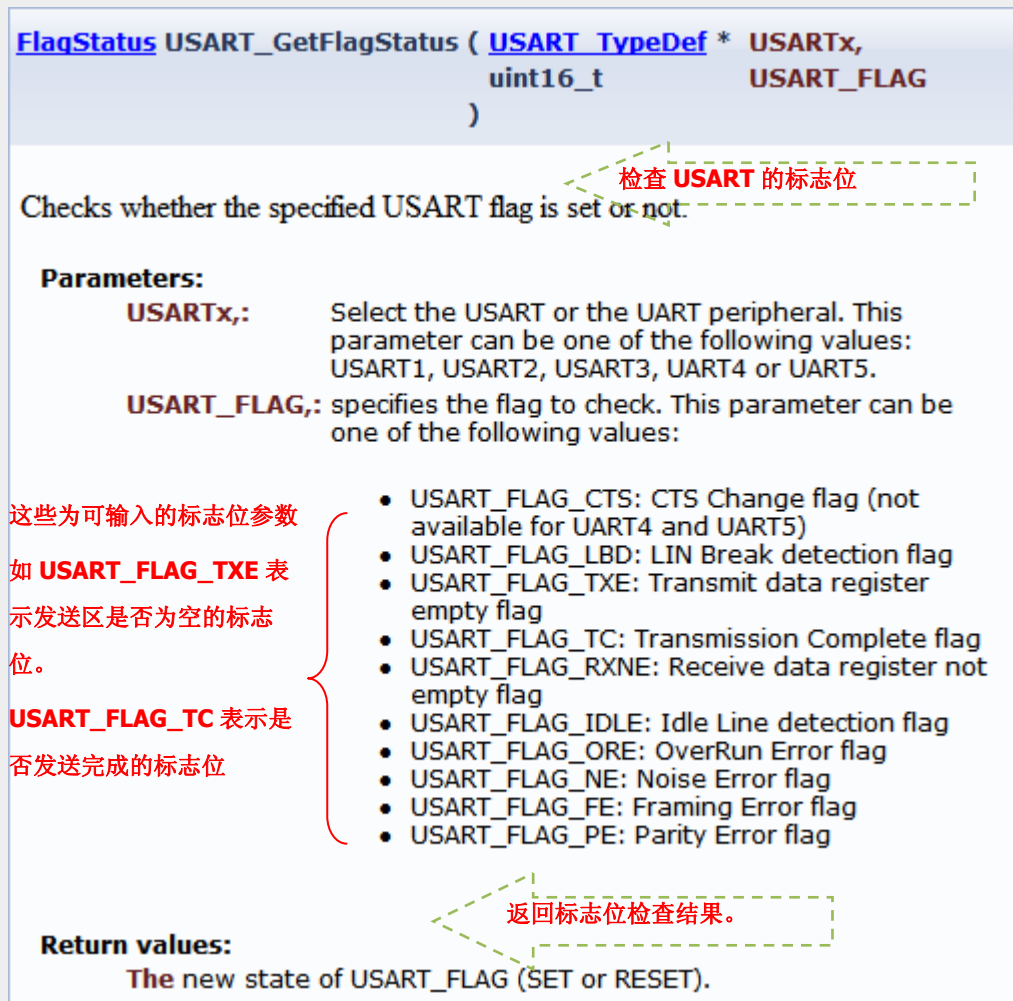


图 1-9 串口标志位检查函数

在使用 c 标准输出库函数，要添加什么头文件？搞嵌入式的可不要把这个忘记了哦。在我们的 *main.c* 文件中要把 *stdio.h* 这个头文件包含进来，还要在编译器中设置一个选项 Use MicroLIB (使用微库)，见图 1-10。这个微库是 keil MDK 为嵌入式应用量身定做的 C 库，我们要先具有库，才能重定向吧？勾选使用之后，我们就可以使用 *printf()* 这个函数了。



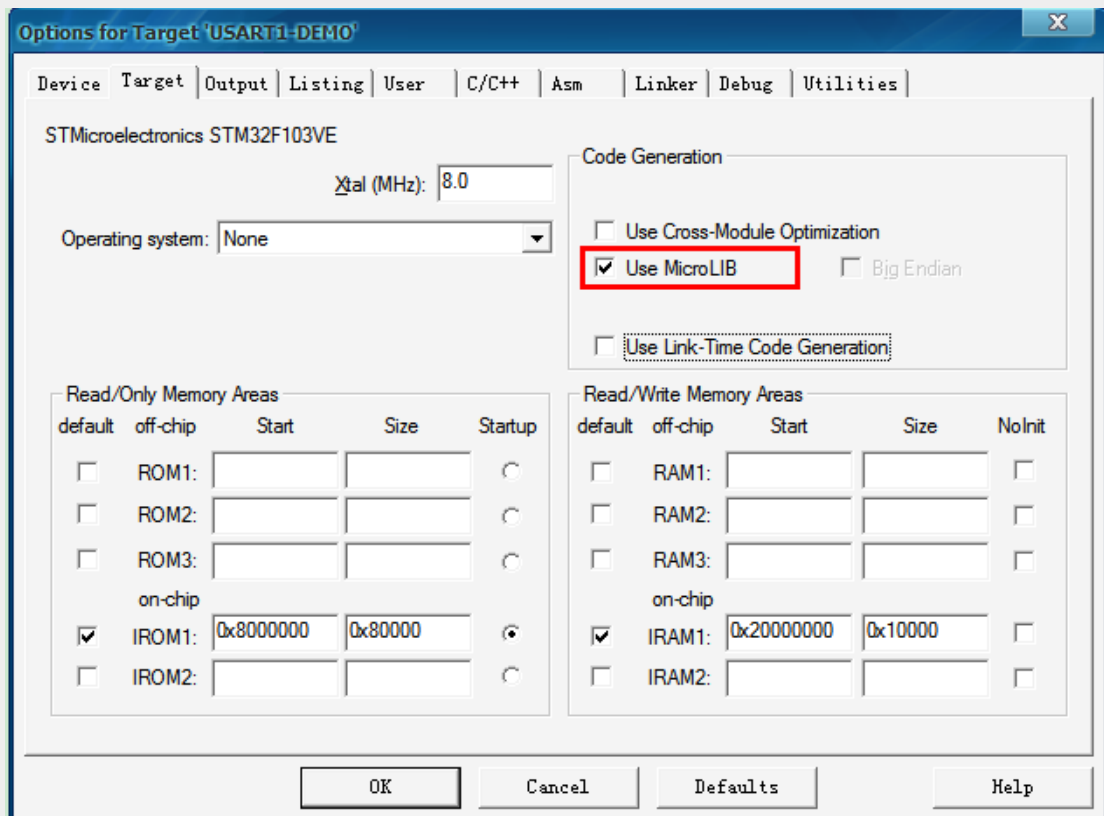


图 1-10 勾选使用微库

1.4.6 USART1_printf() 函数

除了重定向的方法，我们还可以自己编写格式输入输出函数。

*USART1_printf()*便是一个完全自定义的格式输出函数，它的功能与重定向之后的 *printf* 类似。

让我们再来看看

USART1_printf(USART_TypeDef USARTx, uint8_t *Data,...)* 这个函数的实现，它调用了 *itoa(int value, char *string, int radix)* 函数。关于这两个函数的具体实现请看 *usart.c* 中的源代码。这两个函数中有些变量是定义在 *stdarg.h* 这个头文件中的，所以在 *usart.c* 中我们需要把这个头文件包含进来，这个头文件位于 KDE 的根目录下。我们可以在这个路径下找到它：

C:\Keil\ARM\RV31\INC。

```
1. /*
2.  * 函数名: itoa
3.  * 描述   : 将整形数据转换成字符串
```



```
4.  * 输入   : -radix =10 表示 10 进制, 其他结果为 0
5.  *          -value 要转换的整形数
6.  *          -buf 转换后的字符串
7.  *          -radix = 10
8.  * 输出   : 无
9.  * 返回   : 无
10. * 调用   : 被 USART1_printf() 调用
11. */
12. static char *itoa(int value, char *string, int radix)
13. {
14.     int     i, d;
15.     int     flag = 0;
16.     char    *ptr = string;
17.
18.     /* This implementation only works for decimal numbers. */
19.     if (radix != 10)
20.     {
21.         *ptr = 0;
22.         return string;
23.     }
24.
25.     if (!value)
26.     {
27.         *ptr++ = 0x30;
28.         *ptr = 0;
29.         return string;
30.     }
31.
32.     /* if this is a negative value insert the minus sign. */
33.     if (value < 0)
34.     {
35.         *ptr++ = '-';
36.         /* Make the value positive. */
37.         value *= -1;
38.     }
39.     for (i = 10000; i > 0; i /= 10)
40.     {
41.         d = value / i;
42.         if (d || flag)
43.         {
44.             *ptr++ = (char)(d + 0x30);
45.             value -= (d * i);
46.             flag = 1;
47.         }
48.     }
49.
50.     /* Null terminate the string. */
51.     *ptr = 0;
52.     return string;
53. } /* NCL Itoa */
54.
55. /*
56. * 函数名: USART1_printf
57. * 描述   : 格式化输出, 类似于 C 库中的 printf, 但这里没有用到 C 库
58. * 输入   : -USARTx 串口通道, 这里只用到了串口 1, 即 USART1
59. *          -Data 要发送到串口的内容的指针
60. *          -... 其他参数
61. * 输出   : 无
62. * 返回   : 无
63. * 调用   : 外部调用
64. *          典型应用 USART1_printf( USART1, "\r\n this is a demo \r\n" );
```





```
65. *           USART1_printf( USART1, "\r\n %d \r\n", i );
66. *           USART1_printf( USART1, "\r\n %s \r\n", j );
67. */
68. void USART1_printf(USART_TypeDef* USARTx, uint8_t *Data,...)
69. {
70.     const char *s;
71.     int d;
72.     char buf[16];
73.     va_list ap;
74.     va_start(ap, Data);
75.     while ( *Data != 0)           // 判断是否到达字符串结束符
76.     {
77.         if ( *Data == 0x5c )  //'\'
78.         {
79.             switch ( **++Data )
80.             {
81.                 case 'r':           //回车符
82.                     USART_SendData(USARTx, 0x0d);
83.                     Data ++;
84.                     break;
85.
86.                 case 'n':           //换行符
87.                     USART_SendData(USARTx, 0x0a);
88.                     Data ++;
89.                     break;
90.
91.                 default:
92.                     Data ++;
93.                     break;
94.             }
95.         }
96.         else if ( *Data == '%' )
97.         {                               //
98.             switch ( **++Data )
99.             {
100.                case 's':           //字符串
101.                    s = va_arg(ap, const char *);
102.                    for ( ; *s; s++)
103.                    {
104.                        USART_SendData(USARTx,*s);
105.                        while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
106.                    }
107.                    Data++;
108.                    break;
109.
110.                case 'd':           //十进制
111.                    d = va_arg(ap, int);
112.                    itoa(d, buf, 10);
113.                    for (s = buf; *s; s++)
114.                    {
115.                        USART_SendData(USARTx,*s);
116.                        while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
117.                    }
118.                    Data++;
119.                    break;
120.                default:
121.                    Data++;
122.                    break;
123.            }
124.        } /* end of else if */
125.        else USART_SendData(USARTx, *Data++);
126.        while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
127.    }
128. }
```

这部分代码有点多，在格式上编排不是很好，野火推荐大家直接看源码好点。

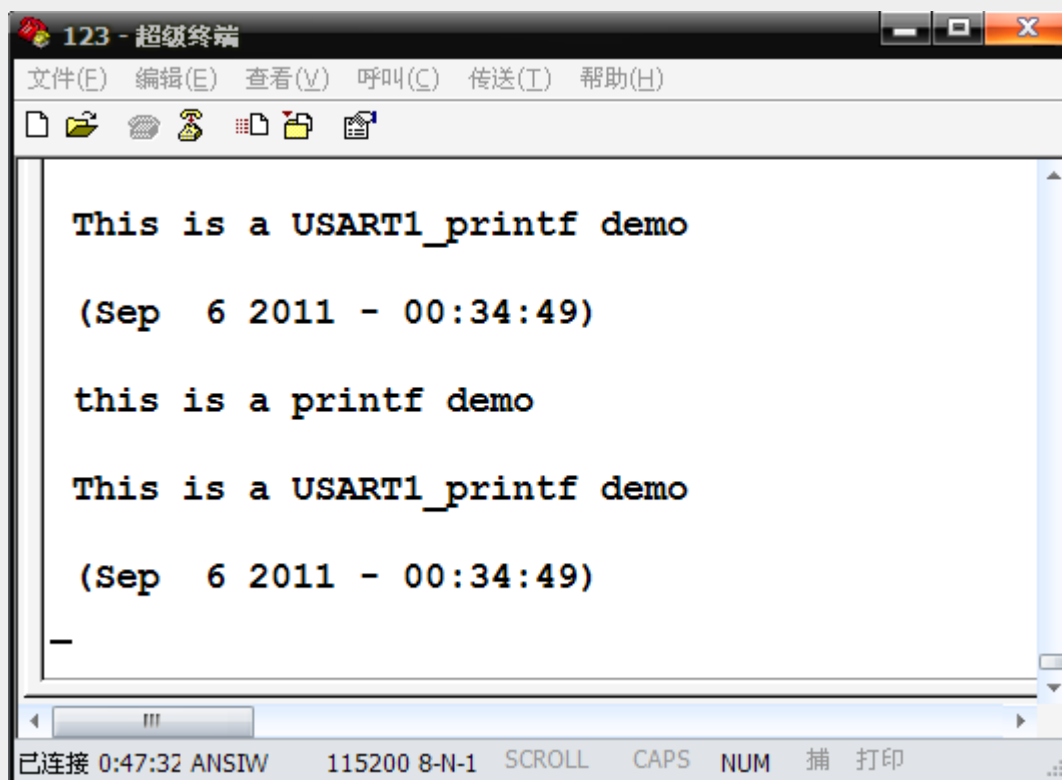
综上，我们已经可以用 `printf()` 和 `USART1_printf()` 这两个函数来打印信息了，但到底用哪个比较好呢？其实各有千秋，`printf()` 函数会受缓冲区大小



的影响，有时候在用它打印的时候程序会发生莫名奇妙的错误，而实际上就是由于使用 `printf()` 这个函数引起的，其优点就是这种情况很少见且支持的格式较多。而 `USART1_printf()` 则不会受缓冲区的影响产生莫名的错误，但其支持的格式较少。不过，相比之下，野火还是比较喜欢用 `USART1_printf()`。

1.4.7 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线，没有的话就 DIY 一个吧 ^_^)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：



2、ADC（DMA 模式）

2.1 ADC 简介

ADC (Analog to Digital Converter)，模/数转换器。在模拟信号需要以数字形式处理、存储或传输时，模/数转换器几乎必不可少。

STM32 在片上集成的 ADC 外设非常强大。在 STM32F103xC、STM32F103xD 和 STM32F103xE 增强型产品，内嵌 3 个 12 位的 ADC，每个 ADC 共用多达 21 个外部通道，可以实现单次或多次扫描转换。如野火 STM32 开发板用的是 STM32F103VET6，属于增强型的 CPU，它有 18 个通道，可测量 16 个外部和 2 个内部信号源。各通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阈值。

2.2 STM32 的 ADC 主要技术指标

对于 ADC 来说，我们最关注的就是它的分辨率、转换速度、ADC 类型、参考电压范围。

- 分辨率

12 位分辨率。不能直接测量负电压，所以没有符号位，即其最小量化单位

$$\text{LSB} = V_{\text{ref+}} / 2^{12}。$$

- 转换时间

转换时间是可编程的。采样一次至少要用 14 个 ADC 时钟周期，而 ADC 的时钟频率最高为 14MHz，也就是说，它的采样时间**最短为 1us**。足以胜任中、低频数字示波器的采样工作。

- ADC 类型

ADC 的类型决定了它性能的极限，STM32 的是**逐次比较型 ADC**。

- 参考电压范围



STM32 的 ADC 参考电压输入见图 2-1。

表62 ADC引脚

名称	信号类型	注解
V _{REF+}	输入，模拟参考正极	ADC使用的高端/正极参考电压， $2.4V \leq V_{REF+} \leq V_{DDA}$
V _{DDA} ⁽¹⁾	输入，模拟电源	等效于V _{DD} 的模拟电源且： $2.4V \leq V_{DDA} \leq V_{DD}(3.6V)$
V _{REF-}	输入，模拟参考负极	ADC使用的低端/负极参考电压， $V_{REF-} = V_{SSA}$
V _{SSA} ⁽¹⁾	输入，模拟电源地	等效于V _{SS} 的模拟电源地
ADCx_IN[15:0]	模拟输入信号	16个模拟输入通道

1. V_{DDA}和V_{SSA}应该分别连接到V_{DD}和V_{SS}。

图 2-1 参考电压

从图中可知，它的参考电压负极是要接地的，即 $V_{ref-} = 0V$ 。而参考电压正极的范围为 $2.4V \leq V_{ref+} \leq 3.6V$ ，所以 STM32 的 ADC 是不能直接测量负电压的，而且其输入的电压信号的范围为： $V_{REF-} \leq V_{IN} \leq V_{REF+}$ 。当需要测量负电压或测量的电压信号超出范围时，要先经过运算电路进行平移或利用电阻分压。

2.3 ADC 工作过程分析

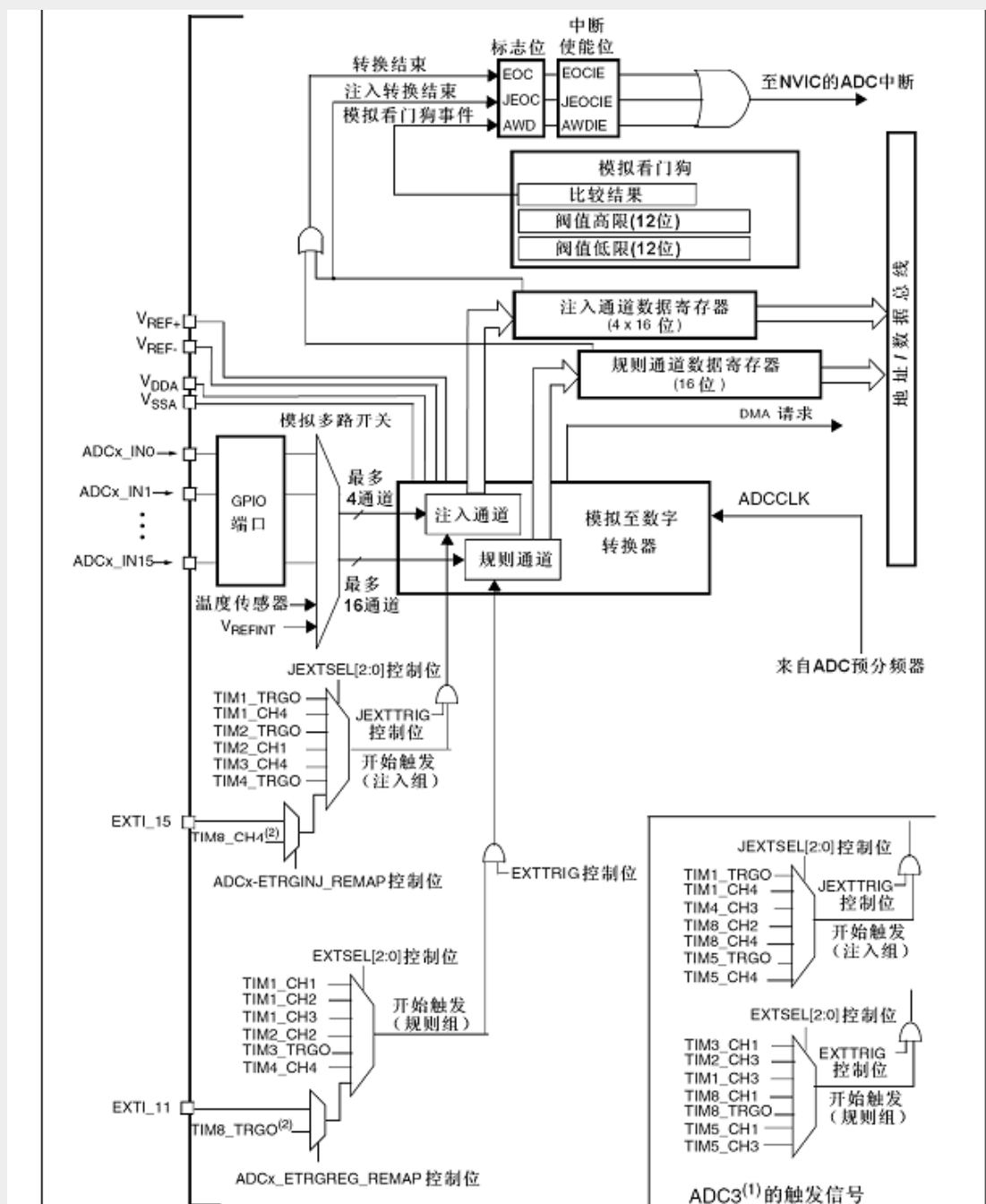


图 2-2 ADC 架构图

我们以ADC的规则通道转换来进行过程分析。所有的器件都是围绕中间的模拟至数字转换器部分（下面简称ADC部件）展开的。它的左端为 V_{REF+} 、 V_{REF-} 等ADC参考电压， $ADCx_IN0 \sim ADCx_IN15$ 为ADC的输入信号通道，即某些GPIO引脚。输入信号经过这些通道被送到ADC部件，ADC部件需要受到触

发信号才开始进行转换，如 *EXTI* 外部触发、定时器触发，也可以使用软件触发。ADC 部件接收到触发信号之后，在 *ADCCLK* 时钟的驱动下对输入通道的信号进行采样，并进行模数转换，其中 *ADCCLK* 是来自 *ADC* 预分频器的。

ADC 部件转换后的数值被保存到一个 16 位的规则通道数据寄存器(或注入通道数据寄存器)之中，我们可以通过 *CPU* 指令或 *DMA* 把它读取到内存(变量)。模数转换之后，可以触发 *DMA* 请求，或者触发 *ADC* 的转换结束事件。如果配置了模拟看门狗，并且采集得的电压大于阈值，会触发看门狗中断。

2.4 ADC 采集实例分析

使用 *ADC* 时常常需要不间断采集大量的数据，在一般的器件中会使用中断进行处理，但使用中断的效率还是不够高。在 *STM32* 中，使用 *ADC* 时往往采用 *DMA* 传输的方式，由 *DMA* 把 *ADC* 外设转换得的数据传输到 *SRAM*，再进行处理，甚至直接把 *ADC* 的数据转移到串口发送给上位机。本小节对 *ADC* 的 *DMA* 方式采集数据实例进行讲解，在讲解 *ADC* 的同时让读者进一步熟悉 *DMA* 的使用。

2.4.1 实验描述及工程文件清单

实验描述	串口 1(USART1)向电脑的超级终端以一定的时间间隔打印当前 ADC1 的转换电压值。
硬件连接	PC1 - ADC1 连接外部电压(通过一个滑动变阻器分压而来)。
用到的库文件	<i>startup/start_stm32f10x_hd.c</i> <i>CMSIS/core_cm3.c</i> <i>CMSIS/system_stm32f10x.c</i> <i>FWlib/stm32f10x_gpio.c</i> <i>FWlib/stm32f10x_rcc.c</i> <i>FWlib/stm32f10x_usart.c</i> <i>FWlib/stm32f10x_adc.c</i>



	<i>FWlib/stm32f10x_dma.c</i>
用户编写的文件	<i>USER/stm32f10x_it.c</i> <i>USER/usart1.c</i> <i>USER/adc.c</i>

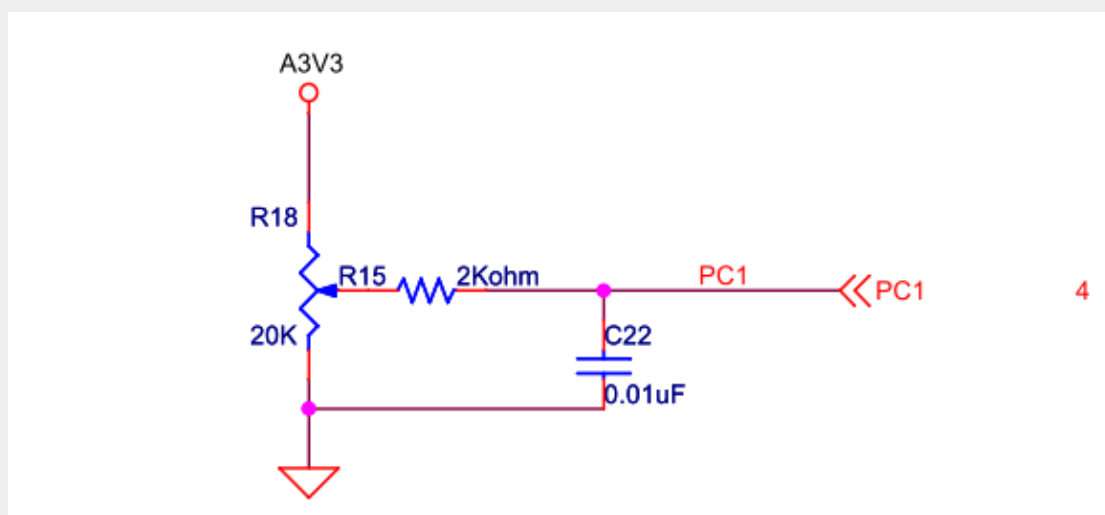


图 2-3 野火 STM32 开发板 ADC 硬件原理图

2.4.2 配置工程环境

本 ADC (DMA 方式) 实验中我们用到了 *GPIO*、*RCC*、*USART*、*DMA* 及 *ADC* 外设，所以我们要把以下库文件添加到工程 *stm32f10x_gpio.c*、*stm32f10x_rcc.c*、*stm32f10x_usart.c*、*stm32f10x_dma.c*、*stm32f10x_adc.c*，添加旧工程中的外设用户文件 *usart1.c*，新建 *adc.c* 及 *adc.h* 文件，并在 *stm32f10x_conf.h* 中把使用到的 ST 库的头文件注释去掉。

```
1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****
9.  #include "stm32f10x_adc.h"
10. #include "stm32f10x_dma.h"
11. #include "stm32f10x_gpio.h"
12. #include "stm32f10x_rcc.h"
13. #include "stm32f10x_usart.h"
```



2.4.3 main 文件

配置好工程环境之后，我们就从 *main* 文件开始分析：

```
1. #include "stm32f10x.h"
2. #include "usart1.h"
3. #include "adc.h"
4.
5. // ADC1 转换的电压值通过 MDA 方式传到 SRAM
6. extern __IO uint16_t ADC_ConvertedValue;
7.
8. // 局部变量，用于保存转换计算后的电压值
9.
10. float ADC_ConvertedValueLocal;
11.
12. // 软件延时
13. void Delay(__IO uint32_t nCount)
14. {
15.     for(; nCount != 0; nCount--);
16. }
17.
18. /**
19.  * @brief  Main program.
20.  * @param  None
21.  * @retval : None
22.  */
23.
24. int main(void)
25. {
26.     /* USART1 config */
27.     USART1_Config();
28.
29.     /* enable adc1 and config adc1 to dma mode */
30.     ADC1_Init();
31.
32.     printf("\r\n -----这是一个 ADC 实验-----\r\n");
33.
34.     while (1)
35.     {
36.         ADC_ConvertedValueLocal = (float) ADC_ConvertedValue/4096*3.3;
37.         // 读取转换的 AD 值
38.         printf("\r\n The current AD value = 0x%04X \r\n", ADC_ConvertedValue);
39.         printf("\r\n The current AD value = %f V \r\n", ADC_ConvertedValueLocal);
40.
41.         Delay(0xffffee); // 延时
42.
43.     }
44. }
45. }
```

浏览一遍 *main* 函数，在调用了用户函数 *USART1_Config()* 及 *ADC1_Init()* 配置好串口和 ADC 之后，就可以直接使用保存了 ADC 转换值的变量 *ADC_ConvertedValue* 了，在 *main* 函数中并没有对 *ADC_ConvertedValue* 重新赋值，这个变量是在什么时候改变的呢？除了可能在中断服务函数修改了变量



值，就只有 DMA 有这样的能耐了，而且大家知道，在使用 DMA 传输时，由于不是内核执行的指令，所以修改变量值是绝对不会出现赋值语句的。

2.4.4 ADC 初始化

本实验代码中完全没有使用中断，而 ADC 及 DMA 的配置工作都由用户函数 `ADC1_Init()` 完成了。配置完成 ADC 及 DMA 后，ADC 就不停地采集数据，而 DMA 自动地把 ADC 采集得的数据转移至内存中的变量 `ADC_ConvertedValue` 中，所以在 `main` 函数的 `while` 循环中使用的 `ADC_ConvertedValue` 都是实时值。接下来重点分析 `ADC1_Init()` 这个函数是如何配置 ADC 的。

`ADC1_Init()` 函数使能了 `ADC1`，并使 `ADC1` 工作于 `DMA` 方式。`ADC1_Init()` 这个函数是由在用户文件 `adc.c` 中实现的用户函数：

```
1.  /*
2.  * 函数名: ADC1_Init
3.  * 描述   : 无
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8.  void ADC1_Init(void)
9.  {
10.     ADC1_GPIO_Config();
11.     ADC1_Mode_Config();
12. }
```

`ADC1_Init()`调用了 `ADC1_GPIO_Config()`和 `ADC1_Mode_Config()`。这两个函数的作用分别是配置好 ADC1 所用的 I/O 端口；配置 ADC1 初始化及 DMA 模式。

2.4.4.1 配置 GPIO 端口

`ADC1_GPIO_Config()`代码：

```
1.  /*
2.  * 函数名: ADC1_GPIO_Config
3.  * 描述   : 使能 ADC1 和 DMA1 的时钟，初始化 PC.01
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8.  static void ADC1_GPIO_Config(void)
9.  {
```



```
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* Enable DMA clock */
13.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
14.
15.     /* Enable ADC1 and GPIOC clock */
16.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOC,
17.                             ENABLE);
18.     /* Configure PC.01 as analog input */
19.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
20.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
21.     GPIO_Init(GPIOC, &GPIO_InitStructure); // PC1, 输入时不用设置速率
22. }
```

ADC1_GPIO_Config() 代码非常简单，就是使能 DMA 时钟，GPIO 时钟及 ADC1 时钟。然后把 ADC1 的通道 11 使用的 GPIO 引脚 PC1 配置成模拟输入模式，在作为 ADC 的输入时，必须使用模拟输入。

这里涉及到 ADC 通道的知识，每个 ADC 通道都对应着一个 GPIO 引脚端口，GPIO 的引脚在设置为模拟输入模式后可用于模拟电压的输入端。STM32F103VET6 有三个 ADC，这三个 ADC 共用 16 个外部通道，从《STM32 数据手册》的引脚定义可找到 ADC 的通道与 GPIO 引脚的关系。见图 2-4

H1	F1	E8	8	15	26	PC0	I/O	PC0	ADC123_IN10	
H2	F2	F8	9	16	27	PC1	I/O	PC1	ADC123_IN11	
H3	E2	D6	10	17	28	PC2	I/O	PC2	ADC123_IN12	
H4	F3	-	11	18	29	PC3	I/O	PC3	ADC123_IN13	

图 2-4 部分 ADC 通道引脚图

表中的引脚名称标注中出现的 ADC12_INx(x 表示 4~9 或 14~15 之间的整数)，表示这个引脚可以是 ADC1_INx 或 ADC2_INx。例如：ADC12_IN9 表示这个引脚可以配置为 ADC1_IN9，也可以配置为 ADC2_IN9。

本实验中使用的 PC1 对应的默认复用功能为 *ADC123_IN11*，也就是说可以使用 *ADC1* 的通道 11、*ADC2* 的通道 11 或 *ADC3* 的通道 11 来采集 PC1 上的模拟电压数据，我们选择 ADC1 的通道 11 来采集。

2.4.4.2 配置 DMA

ADC 模式及其 DMA 传输方式都是在用户函数 *ADC1_Mode_Config()* 中实现的。

ADC1_Mode_Config() 函数代码如下：





```
1.  /* 函数名: ADC1_Mode_Config
2.   * 描述   : 配置 ADC1 的工作模式为 MDA 模式
3.   * 输入   : 无
4.   * 输出   : 无
5.   * 调用   : 内部调用
6.   */
7. static void ADC1_Mode_Config(void)
8. {
9.     DMA_InitTypeDef DMA_InitStructure;
10.    ADC_InitTypeDef ADC_InitStructure;
11.
12.    /* DMA channel1 configuration */
13.    DMA_DeInit(DMA1_Channel1);
14.    DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address; /*ADC
地址*/
15.    DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue; /*
内存地址*/
16.    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //外设为数据源
17.    DMA_InitStructure.DMA_BufferSize = 1;
18.    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; /*
外设地址固定*/
19.    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; /*内存地
址固定*/
20.    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_
HalfWord; //半字
21.    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord
;
22.    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; //循环传输
23.    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
24.    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
25.    DMA_Init(DMA1_Channel1, &DMA_InitStructure);
26.
27.    /* Enable DMA channel1 */
28.    DMA_Cmd(DMA1_Channel1, ENABLE);
29.
30.    /* ADC1 configuration */
31.
32.    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; /*独立 ADC 模式
*/
33.    ADC_InitStructure.ADC_ScanConvMode = DISABLE ; /*禁止扫描模式, 扫描
模式用于多通道采集*/
34.    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE; /*开启连续转换模
式, 即不停地进行 ADC 转换*/
35.    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None
; /*不使用外部触发转换*/
36.    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; /*采集数据
右对齐*/
37.    ADC_InitStructure.ADC_NbrOfChannel = 1; /*要转换的通道数目 1*/
38.    ADC_Init(ADC1, &ADC_InitStructure);
39.
40.    /*配置 ADC 时钟, 为 PCLK2 的 8 分频, 即 9Hz*/
41.    RCC_ADCCLKConfig(RCC_PCLK2_Div8);
42.    /*配置 ADC1 的通道 11 为 55.5 个采样周期 */
43.    ADC_RegularChannelConfig(ADC1, ADC_Channel_11, 1, ADC_SampleTime_5
5Cycles5);
44.
45.    /* Enable ADC1 DMA */
46.    ADC_DMACmd(ADC1, ENABLE);
47.
48.    /* Enable ADC1 */
49.    ADC_Cmd(ADC1, ENABLE);
50.
51.    /*复位校准寄存器 */
52.    ADC_ResetCalibration(ADC1);
```



```
53.     /*等待校准寄存器复位完成 */
54.     while(ADC_GetResetCalibrationStatus(ADC1));
55.
56.     /* ADC 校准 */
57.     ADC_StartCalibration(ADC1);
58.     /* 等待校准完成*/
59.     while(ADC_GetCalibrationStatus(ADC1));
60.
61.     /* 由于没有采用外部触发，所以使用软件触发 ADC 转换 */
62.     ADC_SoftwareStartConvCmd(ADC1, ENABLE);
63. }
```

ADC 的 DMA 配置部分跟串口 DMA 配置部分很类似，它的 DMA 整体上被配置为：

使用 DMA1 的通道 1，数据从 ADC 外设的数据寄存器(*ADC1_DR_Address*)转移到内存(*ADC_ConvertedValue* 变量)，内存、外设地址都固定，每次传输的数据大小为半字(16 位)，使用 DMA 循环传输模式。

其中 ADC1 外设的 DMA 请求通道为 DMA1 的通道 1，初始化时要注意。

DMA 传输的外设地址 *ADC1_DR_Address* 是一个自定义的宏：

```
1. #define ADC1_DR_Address    ((u32)0x40012400+0x4c)
```

ADC_DR 数据寄存器保存了 ADC 转换后的数值，以它作为 DMA 的传输源地址。它的地址是由 ADC1 外设的基地址(0x4001 2400) 加上 ADC 数据寄存器(ADC_DR)的地址偏移 (0x4c)计算得到的。见截自《SM32 参考手册》的说明图 2-6。

0x4001 2800 - 0x4001 2BFF	ADC2
0x4001 2400 - 0x4001 27FF	ADC1

图 2-5 ADC1 起始地址



11.12.14 ADC规则数据寄存器(ADC_DR)

地址偏移: 0x4C

复位值: 0x0000 0000

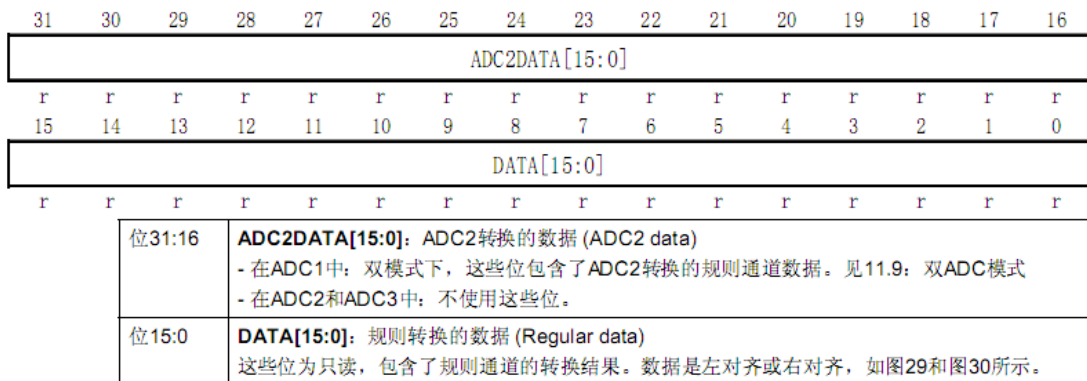


图 2-6 ADC_DR 寄存器地址偏移

2.4.4.3 配置 ADC 模式

从 *ADC1_Mode_Config()* 函数代码的第 32 行开始, 为 ADC 模式的配置, 主要为对 ADC 的初始化结构体进行赋值。下面对这些结构体成员进行介绍:

1) *.ADC_Mode*

STM32 具有多个 ADC, 而不同的 ADC 又是 *共用通道* 的, 当两个 ADC 采集同一个通道的先后顺序、时间间隔不同, 就演变出了各种各样的模式, 如同步注入模式、同步规则模式等 10 种, 根据应用要求选择适合的模式以适应采集数据的要求。

本实验用于测量电阻分压后的电压值, 要求不高, 只使用一个 ADC 就可以满足要求了, 所以本成员被赋值为 *ADC_Mode_Independent* (*独立模式*)。

2) *.ADC_ScanConvMode*

当有多个通道需要采集信号时, 可以把 ADC 配置为按一定的顺序来对各个通道进行扫描转换, 即 *轮流采集各通道的值*。若采集多个通道, 必须开启此模式。

本实验只采集一个通道的信号, 所以 *DISABLE* (*禁止*) 使用扫描转换模式。

3) *.ADC_ContinuousConvMode*



连续转换模式，此模式与单次转换模式相反，单次转换模式 ADC 只采集一次数据就停止转换。而连续转换模式则在上一次 ADC 转换完成后，立即开启下一次转换。

本实验需要循环采集电压值，所以 **ENABLE(使能)**连续转换模式。

4) .ADC_ExternalTrigConv

ADC 需要在接收到**触发信号**才开始进行模数转换，这些触发信号可以是**外部中断触发(EXTI 线)**、**定时器触发**。这两个为外部触发信号，如果不使用外部触发信号可以使用**软件控制触发**。

本实验中使用软件控制触发所以该成员被赋值为 **ADC_ExternalTrigConv_None** (不使用外部触发)。

5) .ADC_DataAlign

数据对齐方式。ADC 转换后的数值是被保存到数据寄存器(ADC_DR)的 0~15 位或 16~32 位，数据**宽度为 16 位**，而 ADC 转换精度为 **12 位**。把 12 位的数据保存到 16 位的区域，就涉及**左对齐**和**右对齐**的问题。这里的左、右对齐跟 word 文档中的文本左、右对齐是一样的意思。

左对齐即 ADC 转换的数值最高位 D12 与存储区域的最高位 Bit 15 对齐，存储区域的低 4 位无意义。右对齐则相反，ADC 转换的数值最低位 D0 保存在存储区域的最低位 Bit 0，高 4 位无意义。见图 0-8。

规则组															
0	0	0	0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

图 0-7 数据右对齐

规则组															
D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0

图 0-8 数据左对齐

本实验中 ADC 的转换值最后被保存在一个 16 位的变量之中，选择 **ADC_DataAlign_Right (右对齐)** 会比较方便。

6) .ADC_NbrOfChannel

这个成员保存了要进行 ADC 数据转换的通道数，可以为 1~16 个。

本实验中只需要采集 PC1 这个通道，所以把成员**赋值为 1**就可以了。



填充完结构体，就可以调用外设初始化函数进行初始化了，ADC 的初始化使用 `ADC_Init()` 函数，初始化完成后别忘记调用 `ADC_Cmd()` 函数来使能 ADC 外设，用 `ADC_DMACmd()` 函数来使能 ADC 的 DMA 接口。在本实验中初始化 ADC1。

2.4.4.4 ADC 转换时间配置

配置好了 ADC 的模式，还要设置 ADC 的时钟(ADCCLK)，ADC 时钟频率越高，转换速度也就越快，但 *ADC 时钟有上限值，不能超过 14MHz。*

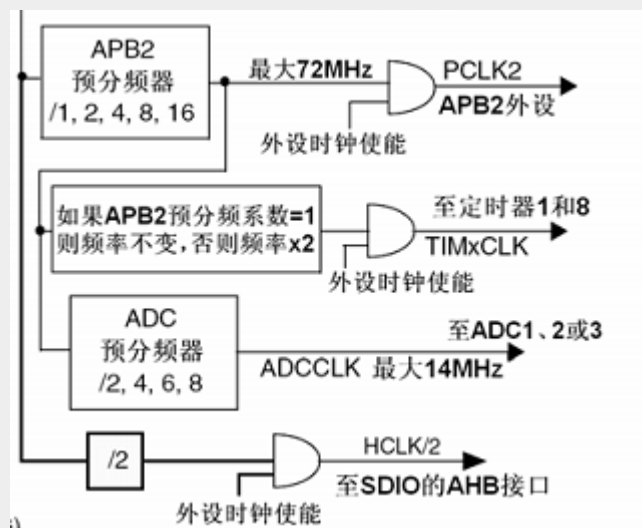


图 0-9 ADC 时钟

配置 ADC 时钟，可使用函数 `RCC_ADCCLKConfig()` 配置，见图 2-10



```
void RCC_ADCCLKConfig ( uint32_t RCC_PCLK2 )
```

Configures the ADC clock (ADCCLK).

Parameters:

RCC_PCLK2,: defines the ADC clock divider. This clock is derived from the APB2 clock (PCLK2). This parameter can be one of the following values:

可以对 **PCLK2** 进行
2、4、6、8 分频，
但 **ADCCLK** 时钟的
最大值为 **14MHz**

- RCC_PCLK2_Div2: ADC clock = PCLK2/2
- RCC_PCLK2_Div4: ADC clock = PCLK2/4
- RCC_PCLK2_Div6: ADC clock = PCLK2/6
- RCC_PCLK2_Div8: ADC clock = PCLK2/8

Return values:

None

图 2-10 ADCCLK 时钟配置函数

ADC 的时钟(ADCCLK)为 ADC 预分频器的输出，而 ADC 预分频器的输入则为高速外设时钟(PCLK2)。使用 `RCC_ADCCLKConfig()` 库函数实质就是设置 ADC 预分频器的分频值，可设置为 PCLK2 的 2、4、6、8 分频。

PCLK2 的常用时钟频率为 72MHz，而 ADCCLK 必须低于 14MHz，所以在这个情况下，ADCCLK 最高频率为 PCLK2 的 8 分频，即 ADCCLK=9MHz。若希望使 ADC 以最高的频率 14MHz 运行，可以把 PCLK2 配置为 56MHz，然后再 4 分频得到 ADCCLK。

ADC 的转换时间不仅与 ADC 的时钟有关，还与采样周期相关。



每个不同的 ADC 通道，都可以设置为不同的采样周期。配置时用库函数 *ADC_RegularChannelConfig()*，它的说明见图 2-11。

```
void ADC_RegularChannelConfig ( ADC_TypeDef * ADCx,
                                uint8_t      ADC_Channel,
                                uint8_t      Rank,
                                uint8_t      ADC_SampleTime
                                )
```

Configures for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.

Parameters:

ADCx,: where x can be 1, 2 or 3 to select the ADC peripheral.

ADC_Channel,: the ADC channel to configure. This parameter can be one of the following values:

- ADC_Channel_0: ADC Channel0 selected
- ADC_Channel_1: ADC Channel1 selected
- ADC_Channel_2: ADC Channel2 selected
- ADC_Channel_3: ADC Channel3 selected
- ADC_Channel_4: ADC Channel4 selected
- ADC_Channel_5: ADC Channel5 selected
- ADC_Channel_6: ADC Channel6 selected
- ADC_Channel_7: ADC Channel7 selected
- ADC_Channel_8: ADC Channel8 selected
- ADC_Channel_9: ADC Channel9 selected
- ADC_Channel_10: ADC Channel10 selected
- ADC_Channel_11: ADC Channel11 selected
- ADC_Channel_12: ADC Channel12 selected
- ADC_Channel_13: ADC Channel13 selected
- ADC_Channel_14: ADC Channel14 selected
- ADC_Channel_15: ADC Channel15 selected
- ADC_Channel_16: ADC Channel16 selected
- ADC_Channel_17: ADC Channel17 selected

Rank,: The rank in the regular group sequencer. This parameter must be between 1 to 16.

ADC_SampleTime,: The sample time value to be set for the selected channel. This parameter can be one of the following values:

- ADC_SampleTime_1Cycles5: Sample time equal to 1.5 cycles
- ADC_SampleTime_7Cycles5: Sample time equal to 7.5 cycles
- ADC_SampleTime_13Cycles5: Sample time equal to 13.5 cycles
- ADC_SampleTime_28Cycles5: Sample time equal to 28.5 cycles
- ADC_SampleTime_41Cycles5: Sample time equal to 41.5 cycles
- ADC_SampleTime_55Cycles5: Sample time equal to 55.5 cycles
- ADC_SampleTime_71Cycles5: Sample time equal to 71.5 cycles
- ADC_SampleTime_239Cycles5: Sample time equal to 239.5 cycles

Return values:
None

选择要配置的 ADC 通道

RANK 参数: 配置的数值为多通道扫描时, 此通道的采样顺序

可配置的采样周期: 1.5、7.5、13.5 等 ADC 时钟周期

图 2-11 规则通道配置函数

ADC_RegularChannelConfig() 函数中的 *RANK* 值是指在多通道扫描模式时，本通道的扫描顺序。例如通道 1、4、7 的 *RANK* 值分别为被配置为 3、2、1 的话，在 ADC 扫描时，扫描的顺序为通道 7、通道 4、最后扫描通道 1。

本实验中只采集一个 ADC 通道，把以把 ADC1 的通道 11 *RANK* 值配置为 1。



ADC_SampleTime 的参数值则用于配置本通道的 *采样周期*，最短可配置为 1.5 个采样周期，这里的周期指 ADCCLK 时钟周期。

本实验中把 ADC1 通道 11 配置为 55.5 个采样周期，而 ADCCLK 在前面已经配置为 9MHz，根据 STM32 的 *ADC 采样时间计算公式*：

$$T_{\text{CONV}} = \text{采样周期} + 12.5 \text{ 个周期}$$

公式中的采样周期就是本函数中配置的 *ADC_SampleTime*，而后边加上的 12.5 个周期为固定的数值。

所以，本实验中 ADC1 通道 11 的转换时间 $T_{\text{CONV}} = (55.5 + 12.5) \times 1/9 \approx 7.56\mu\text{s}$ 。

2.4.4.4.5 ADC 自校准

在开始 ADC 转换之前，需要启动 ADC 的自校准。ADC 有一个内置自校准模式，校准可大幅减小因内部电容器组的变化而造成的准精度误差。在校准期间，在每个电容器上都会计算出一个误差修正码(数字值)，这个码用于消除在随后的转换中每个电容器上产生的误差。

以下为在 *ADC1_Mode_Config()* 函数中的 ADC 自校准时调用的库函数和使用步骤。

```
1.  /*复位校准寄存器 */
2.  ADC_ResetCalibration(ADC1);
3.  /*等待校准寄存器复位完成 */
4.  while(ADC_GetResetCalibrationStatus(ADC1));
5.
6.  /* ADC 校准 */
7.  ADC_StartCalibration(ADC1);
8.  /* 等待校准完成*/
9.  while(ADC_GetCalibrationStatus(ADC1));
```

在调用了复位校准函数 *ADC_ResetCalibration()* 和开始校准函数 *ADC_StartCalibration()* 后，要检查标志位等待校准完成，确保完成后才开始 ADC 转换。建议在每次上电后都进行一次自校准。

在校准完成后，就可以开始进行 ADC 转换了。本实验代码中配置的 ADC 模式为软件触发方式，我们可以调用库函数 *ADC_SoftwareStartConvCmd()* 来开启软件触发。其说明见图 2-12。



```
void ADC_SoftwareStartConvCmd ( ADC_TypeDef * ADCx,  
                                FunctionalState NewState  
                                )
```

Enables or disables the selected ADC software start conversion .

Parameters:

ADCx,: where x can be 1, 2 or 3 to select the ADC peripheral.

NewState,: new state of the selected ADC software start conversion. This parameter can be: ENABLE or DISABLE.

Return values:

None

图 2-12 ADC 软件触发控制函数

调用这个函数使能了 ADC1 的软件触发后，ADC 就开始进行转换了，每次转换完成后，由 DMA 控制器把转换值从 ADC 数据寄存器(ADC_DR)中转移到变量 *ADC_ConvertedValue* 中，当 DMA 传输完成后，在 main 函数中使用的 *ADC_ConvertedValue* 的内容就是 ADC 转换值了。

2.4.4.4.6 volatile 变量

现在我们来认识 *ADC_ConvertedValue* 这个变量：

```
1. // ADC1 转换的电压值通过 MDA 方式传到 flash  
2. extern __IO u16 ADC_ConvertedValue;  
3.
```

ADC_ConvertedValue 在文件 *adc.c* 中定义，这个。这里要注意一点的是，这个变量要用 c 语言的 *volatile* 关键字来修饰，为的是让编译器不要去优化这个变量。这样每次用到这两个变量时都要回到相应变量的内存中去取值，而 *volatile* 字面意思就是“可变的，不确定的”。

例如：不使用 *volatile* 关键字修饰的变量 *a* 在被访问的时候可能会直接从 CPU 的寄存器中取出（因为之前变量 *a* 被访问过，也就是说之前就从内存中取出 *a* 的值保存到某个 CPU 寄存器中），之所以直接从寄存器中取值，而不去内存中取值，是因为编译器优化代码的结果（访问 CPU 寄存器比访问内存快的多）。这里的 CPU 寄存器指 R0、R1 等 CPU 通用寄存器，用于 CPU 运算是暂存数据的，不是指外设中的寄存器。



用 *volatile* 声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。

因为 *ADC_ConvertedValue* 这个变量值随时都是会被 *DMA 控制器* 改变的，所以我们用 *volatile* 来修饰它，确保每次读取到的都是实时的 ADC 转换值。

2.4.5 计算电压值

回到 main 函数中循环打印电压值部分：

```
1. while (1)
2. {
3.     ADC_ConvertedValueLocal = (float) ADC_ConvertedValue/4096*3.3; //
    读取转换的 AD 值
4.
5.     printf("\r\n The current AD value = 0x%04X \r\n", ADC_ConvertedValue);
6.     printf("\r\n The current AD value = %f V\r\n", ADC_ConvertedValueLocal);
7.
8.     Delay(0xffff); // 延时
9.
10.
11. }
```

float 型变量 *ADC_ConvertedValueLocal* 保存了由转换值计算出来的电压值，其计算公式是 ADC 通用的：

$$\text{实际电压值} = \text{ADC 转换值} * \text{LSB}$$

STM32 的 ADC 的精度为 12 位，而野火板子中 $V_{\text{ref+}}$ 接的参考电压值为 3.3v，所以 $\text{LSB} = 3.3/2^{12}$ 。

2.4.6 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：

当旋转开发板开发板上的滑动变阻器时，ADC1 转换的电压值则会改变。板载的是 20K 的精密电阻，旋转的圈数要多点才能看到 ADC 值的明显变化。



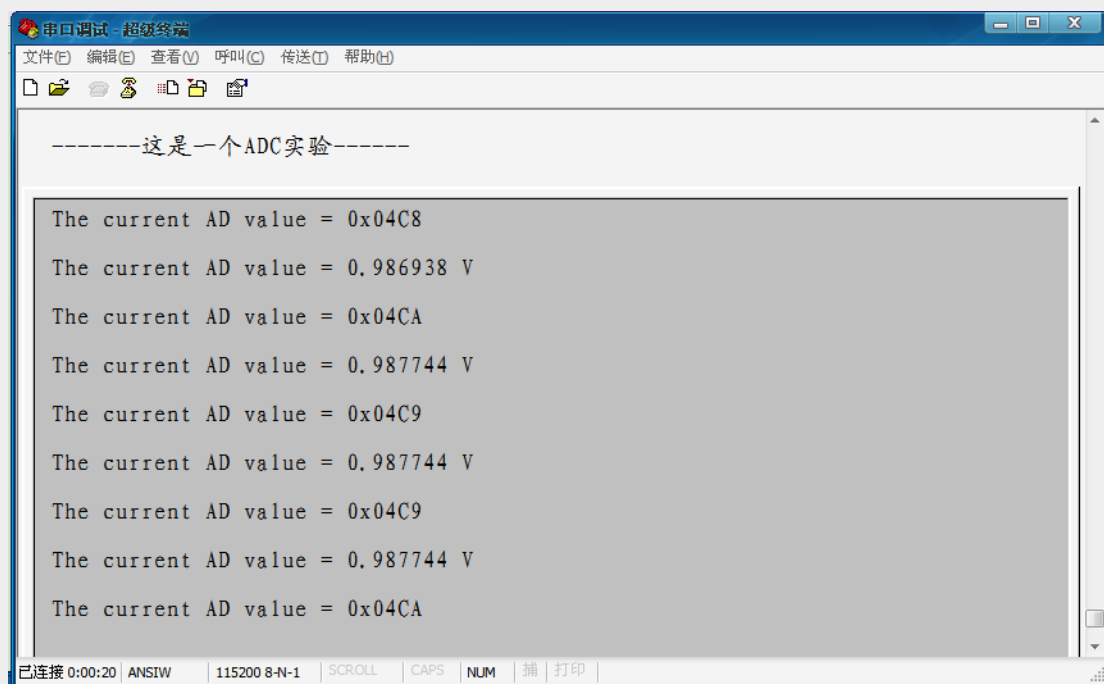


图 2-13 ADC 采集外部电压实验



3、Temperature（芯片温度）

3.1 实验描述及工程文件清单

实验描述	串口 1(USART1)向电脑的超级终端以 1s 为时间间隔打印当前 STM32F103VET6 芯片内部的温度值。
硬件连接	温度传感器在芯片内部和 ADCx_IN16 输入通道相连接
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_adc.c FWlib/stm32f10x_dma.c FWlib/stm32f10x_flash.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/adc.c

3.2 ADC 及内部温度传感器简介

STM32F103xC、STM32F103xD 和 STM32F103xE 增强型产品，内嵌 3 个 12 位的模拟/数字转换器(ADC)，每个 ADC 共用多达 21 个外部通道，可以实现单次或多次扫描转换。STM32 开发板用的是 STM32F103VET6，属于增强型的 CPU。它有 18 个通道，可测量 16 个外部和 2 个内部信号源，分别是 ADCx_IN16（温度传感器）和 ADCx_IN1740(VREFINT)。各通道的 A/D 转换可以

单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阈值。

STM32 内部的温度传感器和 ADCx_IN16 输入通道相连接，此通道把传感器输出的电压值转换成数字值。STM 内部的温度传感器支持的温度范围：-40 到 125 摄氏度。精度较差，误差为+（-）1.5 度左右，听起来有点蛋疼。

ADC 可以使用 DMA(direct memory access)方式操作。

本实验用的是 ADC1 的通道 16，采用 DMA 的方式操作。

内部温度传感器的基本操作步骤：（STM32 参考手册）

1. 选择 ADCx_IN16 输入通道
2. 选择采样时间大于 $2.2 \mu s$ （推荐值为 $17.1 \mu s$ ）
3. 设置 ADC 控制寄存器 2 (ADC_CR2) 的 TSVREFE 位，以唤醒关电模式下的温感器
4. 通过设置 ADON 位启动 ADC 转换(或用外部触发)
5. 读 ADC 数据寄存器上的 VSENSE 数据结果
6. 利用下列公式得出温度

$$\text{温度} (^{\circ} C) = \{(V_{25} - V_{\text{SENSE}}) / \text{Avg_Slope}\} + 25$$

式中 V_{25} 是 V_{SENSE} 在 25 摄氏度时的数值（典型值为 1.42V）

Avg_Slope 是温度与 V_{SENSE} 曲线的平均斜率（典型值为 $4.3 \text{mV}/^{\circ}C$ ）

PS： 对于 12 位的 AD，3.3V 的 AD 值为 0xfff；1.42V 对应的 AD 值为：0x6E2；4.3mV 对应的 AD 值为：0x05（用系统自带计算器可轻易算得）这些是计算温度值的时候用得到的，也可以用其他方法计算。详情请参考 STM32 手册。

3.3 代码分析

首先要添加用的库文件，在工程文件夹下 Fwlib 下我们需添加以下库文件：

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



```
4. stm32f10x_adc.c
5. stm32f10x_dma.c
6. stm32f10x_flash.c
```

还要在 [stm32f10x_conf.h](#) 中将相应头文件的注释去掉：

```
1. /* Uncomment the line below to enable peripheral header file inclusion */
2. #include "stm32f10x_adc.h"
3. /* #include "stm32f10x_bkp.h" */
4. /* #include "stm32f10x_can.h" */
5. /* #include "stm32f10x_crc.h" */
6. /* #include "stm32f10x_dac.h" */
7. /* #include "stm32f10x_dbgmcu.h" */
8. #include "stm32f10x_dma.h"
9. /* #include "stm32f10x_exti.h" */
10. #include "stm32f10x_flash.h"
11. /* #include "stm32f10x_fsmc.h" */
12. #include "stm32f10x_gpio.h"
13. /* #include "stm32f10x_i2c.h" */
14. /* #include "stm32f10x_iwdg.h" */
15. /* #include "stm32f10x_pwr.h" */
16. #include "stm32f10x_rcc.h"
17. /* #include "stm32f10x_rtc.h" */
18. /* #include "stm32f10x_sdio.h" */
19. /* #include "stm32f10x_spi.h" */
20. /* #include "stm32f10x_tim.h" */
21. #include "stm32f10x_usart.h"
22. /* #include "stm32f10x_wwdg.h" */
23. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
    on to CMSIS functions) */
```

配置好所需的库文件之后，我们就从 [main](#) 函数开始分析：

```
1. /**
2.  * @brief Main program.
3.  * @param None
4.  * @retval : None
5.  */
6.
7. int main(void)
8. {
9.     /* config the sysclock to 72M */
10.    SystemInit();
11.
12.    /* USART1 config */
```



```
13.     USART1_Config();
14.
15.     /* enable adc1 and config adc1 to dma mode */
16.     Temp_ADC1_Init();
17.
18.     printf("\r\n Print current Temperature \r\n");
19.
20.     while (1)
21.     {
22.         ADC_tempValueLocal= ADC_ConvertedValue;    // 读取转换的 AD 值
23.         Delay(0xffffee);                          // 延时
24.         Current_Temp=(V25-ADC_tempValueLocal)/Avg_Slope+25;    //计算方法 2
25.
26.         printf("\r\n The current temperature = %02d C\r\n", Current_Temp); //10 进制示
27.     }
```

系统库函数 `SystemInit()`；将系统时钟设置为 72M，`USART1_Config()`；配置串口，关于这两个函数的具体讲解可以参考前面的教程，这里不再详述。

`Temp_ADC1_Init()`；函数使能了 ADC1，并使 ADC1 工作于 DMA 方式。

`Temp_ADC1_Init()`；这个函数是由我们用户在 `tempad.c` 文件中实现的应用程序：

```
1.  /*
2.  * 函数名:Temp_ADC1_Init();
3.  * 描述   : 无
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8.  void Temp_ADC1_Init(void)
9.  {
10.     ADC1_GPIO_Config();
11.     ADC1_Mode_Config();
12. }
```

`Temp_ADC1_Init()`；调用了 `ADC1_GPIO_Config()`；和 `ADC1_Mode_Config()`；这两个函数的作用分别是配置好 ADC 和 DMA 的时钟、配置它的工作模式为 MDA 模式。

```
1.  /*
2.  * 函数名: ADC1 GPIO Config
3.  * 描述   : 使能 ADC1 和 DMA1 的时钟
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8.  static void ADC1_GPIO_Config(void)
9.  {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     /* Enable DMA clock */
12.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
13.
14.     /* Enable ADC1 and GPIOC clock */
15.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
16.
17. }
```



代码非常简单，大家就自己花点心思看看吧。这两个函数都用 `static` 关键字修饰了，都属于内部调用，我们只向其他用户提供了这个 `Temp_ADC1_Init()`；接口，使得更方便简洁。

```
1.  /* 函数名: ADC1_Mode_Config
2.   * 描述   : 配置 ADC1 的工作模式为 MDA 模式
3.   * 输入   : 无
4.   * 输出   : 无
5.   * 调用   : 内部调用
6.   */
7.  static void ADC1_Mode_Config(void)
8.  {
9.      DMA_InitTypeDef DMA_InitStructure;
10.     ADC_InitTypeDef ADC_InitStructure;
11.
12.     /* DMA channel1 configuration */
13.     DMA_DeInit(DMA1_Channel1);
14.     DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
15.     DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;
16.     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
17.     DMA_InitStructure.DMA_BufferSize = 1;
18.     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
19.     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
20.     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
21.
22.     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
23.     DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
24.     DMA_InitStructure.DMA_Priority = DMA_Priority_High;
25.     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
26.     DMA_Init(DMA1_Channel1, &DMA_InitStructure);
27.
28.     /* Enable DMA channel1 */
29.     DMA_Cmd(DMA1_Channel1, ENABLE);
30.
31.     /* ADC1 configuration */
32.     ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
33.     ADC_InitStructure.ADC_ScanConvMode = ENABLE;
34.     ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
35.     ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
36.     ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
37.     ADC_InitStructure.ADC_NbrOfChannel = 1;
38.     ADC_Init(ADC1, &ADC_InitStructure);
39.
40.     /* ADC1 regular channel16 configuration */
41.     ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_55Cycles5);
42.
43.     /*Enable TempSensorVrefintCmd*/
44.     ADC_TempSensorVrefintCmd(ENABLE);
45.
46.     /* Enable ADC1 DMA */
47.     ADC_DMACmd(ADC1, ENABLE);
48.
49.     /* Enable ADC1 */
50.     ADC_Cmd(ADC1, ENABLE);
51.
52.     /* Enable ADC1 reset calibration register */
53.     ADC_ResetCalibration(ADC1);
54.     /* Check the end of ADC1 reset calibration register */
55.     while(ADC_GetResetCalibrationStatus(ADC1));
56.
57.     /* Start ADC1 calibration */
58.     ADC_StartCalibration(ADC1);
59.     /* Check the end of ADC1 calibration */
60.     while(ADC_GetCalibrationStatus(ADC1));
61.
62.     /* Start ADC1 Software Conversion */
63.     ADC_SoftwareStartConvCmd(ADC1, ENABLE);
64. }
```

`Temp_ADC1_Init()` 里面重要的一步是使能内部的温度传感器，因为内部的温度传感器默认关闭的。



现在我们来认识几个变量：

```
4. // ADC1 转换的电压值通过 MDA 方式传到 flash
5. extern IO u16 ADC_ConvertedValue;
6.
7. // 局部变量，用于存从 flash 读到的电压值
8. __IO u16 ADC_tempValueLocal;
9. //存放计算后的温度值
10. __IO u16 Current_Temp;
11.
12. //温度为 25 摄氏度时的电压值
13. __IO u16 V25=0x6E2;
14.
15. //温度、电压对应的的斜率 每摄氏度 4.35mV 对应每摄氏度 0x05
16. __IO u16 Avg_Slope=0x05;
```

ADC_ConvertedValue 在 adc.c 中定义，ADC_tempValueLocal 在 main.c 中定义，关于这两个变量的作用可看代码的描述。这里要注意一点的是，这两个变量都要用 volatile 关键字来修饰，为的是不让编译器优化这个变量，这样每次用到这两个变量时都要回到相应变量的内存中去取值，因为这两个变量的值随时都是可变的，而 volatile 字面意思就是“可变的，不确定的”。有关 volatile 关键字的详细用法大家可去查与 C 语言有关的书，这里推荐一个 C 语言的小册子《C 语言深度解剖-陈正冲编著》，里面对 volatile 这个关键字的讲解就非常好，还有其他有关 C 语言的知识也讲得非常好，挺值得大家看看。

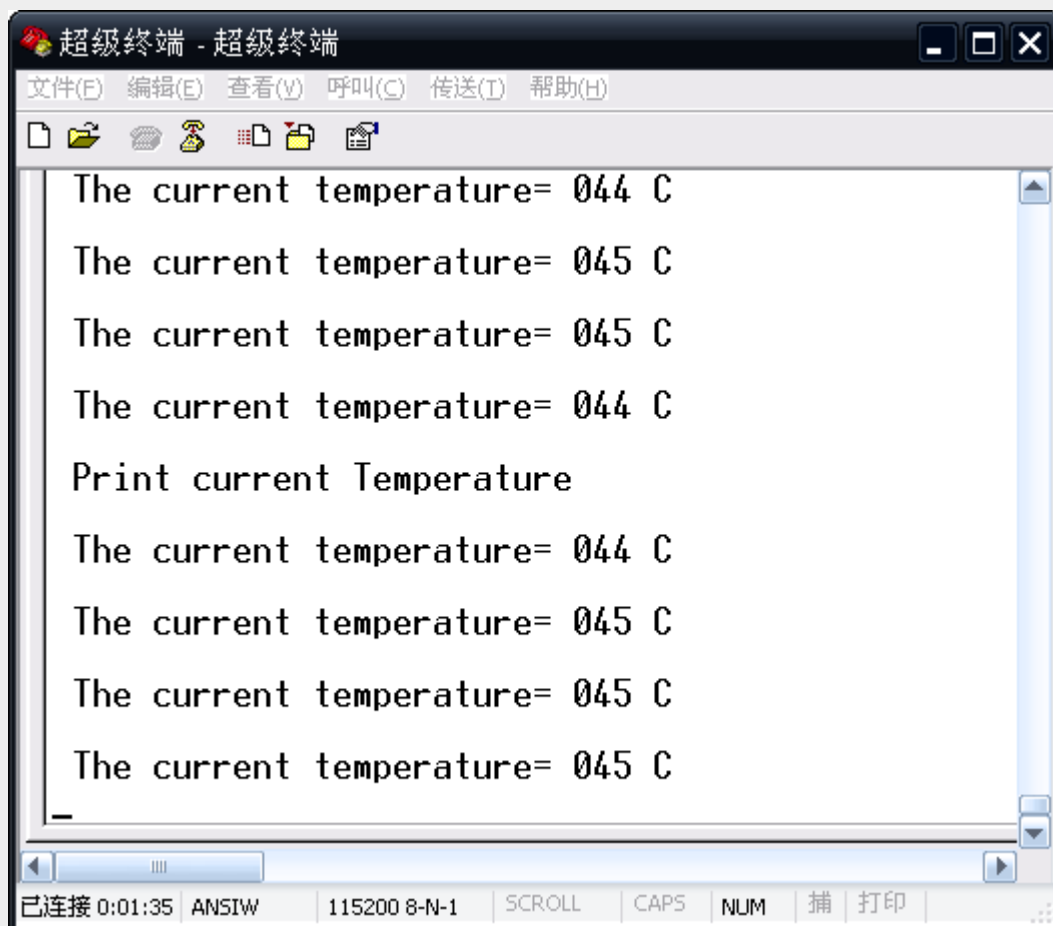
主函数的最后以一个无限循环不断地往串口打印检测到芯片的内部温度值：

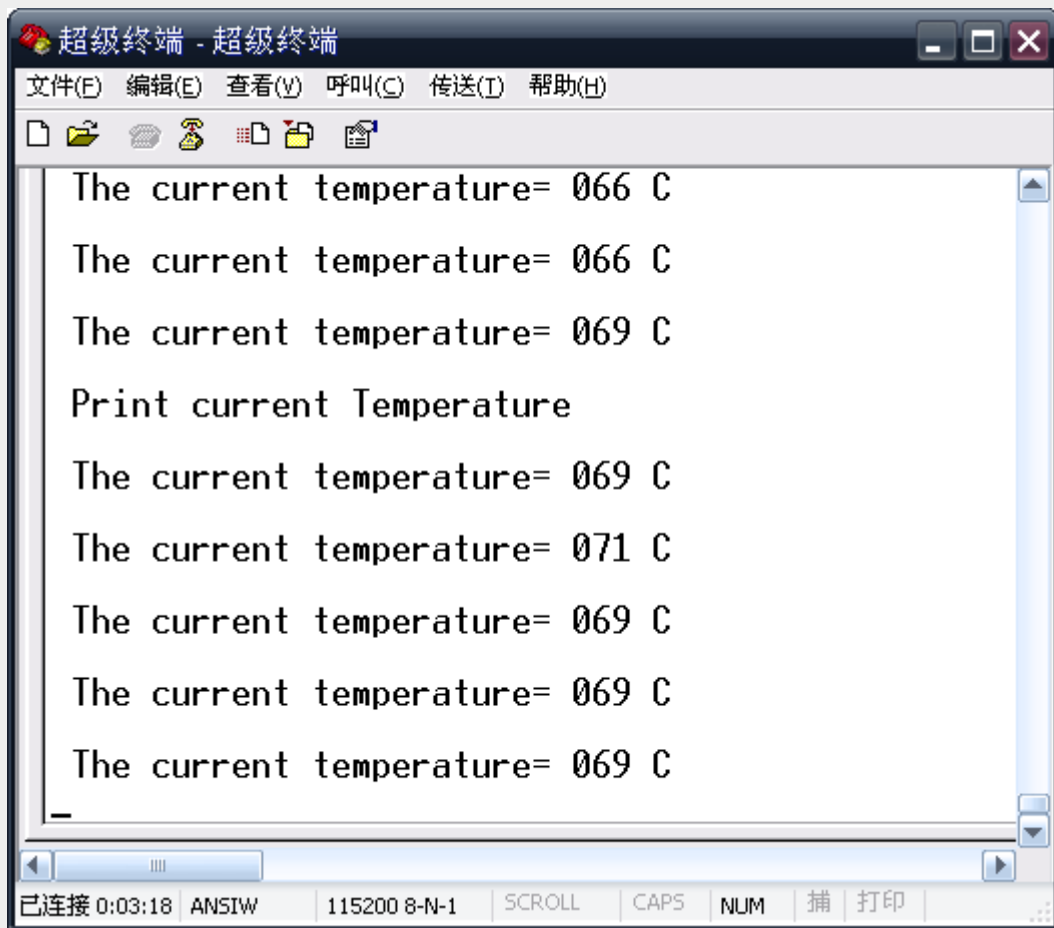
```
1. while (1)
2. {
3.     ADC_tempValueLocal = ADC_ConvertedValue; // 读取转换的 AD 值
4.     Delay(0xffffee); // 延时
5.     Current_Temp=(V25-ADC_tempValueLocal)/Avg_Slope+25; //温度转换计算
6.     printf("\r\n The current temperature = %03d C\r\n", Current_Temp); //10 进制
    显示
7. }
```



3.4 实验想象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：（检测到的是芯片的内部温度）





```
超级终端 - 超级终端
文件(E) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

The current temperature= 066 C
The current temperature= 066 C
The current temperature= 069 C
Print current Temperature
The current temperature= 069 C
The current temperature= 071 C
The current temperature= 069 C
The current temperature= 069 C
The current temperature= 069 C

已连接 0:03:18 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

当用加热的电烙铁轻轻地放在我们的芯片上（或者用热风筒吹芯片）时会看到温度有明显的变化。



4、RTC（万年历）

4.1 实验描述及工程文件清单

实验描述	<p>利用 STM32 的 RTC 实现一个简易的电子时钟。在超级终端中显示时间值。</p> <p>显示格式为 Time: XX:XX:XX(时：分：秒)，当时间计数为：23：59：59 时将刷新为：00：00：00。</p>
硬件连接	VBAT 引脚需外接电池。
用到的库文件	<p>startup/start_stm32f10x_hd.c</p> <p>CMSIS/core_cm3.c</p> <p>CMSIS/system_stm32f10x.c</p> <p>FWlib/stm32f10x_gpio.c</p> <p>FWlib/stm32f10x_rcc.c</p> <p>FWlib/stm32f10x_usart.c</p> <p>FWlib/stm32f10x_pwr.c</p> <p>FWlib/stm32f10x_bkp.c</p> <p>FWlib/stm32f10x_rtc.c</p> <p>FWlib/stm32f10x_misc.c</p>
用户编写的文件	<p>USER/main.c</p> <p>USER/stm32f10x_it.c</p> <p>USER/usart.c</p> <p>USER/rtc.c</p>



4.2 RTC（实时时钟）简介

实时时钟是一个独立的定时器。RTC 模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。

RTC 模块和时钟配置系统(RCC_BDCR 寄存器)是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变。

系统复位后，禁止访问后备寄存器和 RTC，防止对后备区域(BKP)的意外写操作。执行以下操作使能对后备寄存器和 RTC 的访问：

- 设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位来使能电源和后备接口时钟。
- 设置寄存器 PWR_CR 的 DBP 位使能对后备寄存器和 RTC 的访问。

当我们需要在掉电之后，又需要 RTC 时钟正常运行的话，单片机的 VBAT 脚需外接 3.3V 的锂电池。当我们重新上电的时候，主电源给 VBAT 供电，当系统掉电之后 VBAT 给 RTC 时钟工作，RTC 中的数据都会保持在后备寄存器当中。

野火 STM32 开发板的 VBAT 引脚接了 3.3V 的锂电。

4.3 代码分析

首先添加需要的库文件：

```
FWlib/stm32f10x_gpio.c
FWlib/stm32f10x_rcc.c
FWlib/stm32f10x_usart.c
FWlib/stm32f10x_pwr.c
FWlib/stm32f10x_bkp.c
FWlib/stm32f10x_rtc.c
FWlib/stm32f10x_misc.c
```



在 `stm32f10x_conf.g` 中将相应库文件的头文件的注释去掉，这样才能够真正使用这些库，否则将会编译错误。

```
1.  /* Uncomment the line below to enable peripheral header file inclusion */
2.  /* #include "stm32f10x_adc.h" */
3.  #include "stm32f10x_bkp.h"
4.  /* #include "stm32f10x_can.h" */
5.  /* #include "stm32f10x_crc.h" */
6.  /* #include "stm32f10x_dac.h" */
7.  /* #include "stm32f10x_dbgmcu.h" */
8.  /* #include "stm32f10x_dma.h" */
9.  /* #include "stm32f10x_exti.h" */
10. /*#include "stm32f10x_flash.h"*/
11. /* #include "stm32f10x_fsmc.h" */
12. #include "stm32f10x_gpio.h"
13. /* #include "stm32f10x_i2c.h" */
14. #include "stm32f10x_iwdg.h"
15. #include "stm32f10x_pwr.h"
16. #include "stm32f10x_rcc.h"
17. #include "stm32f10x_rtc.h"
18. /* #include "stm32f10x_sdio.h" */
19. /* #include "stm32f10x_spi.h" */
20. /* #include "stm32f10x_tim.h" */
21. #include "stm32f10x_usart.h"
22. /* #include "stm32f10x_wwdg.h" */
23. #include "misc.h" /* High level functions for NVIC and SysTick (add-
    on to CMSIS functions) */
```

好嘞，配置好库的环境之后，我们就从 `main` 函数开始分析。`main` 函数有点长，大家给点耐心，好好分析下，也不难。

```
1.  /**
2.   * @brief Main program.
3.   * @param None
4.   * @retval : None
5.   */
6.
7.  int main(void)
8.  {
9.      /* config the sysclock to 72M */
10.     SystemInit();
11.
12.     /* USART1 config */
13.     USART1_Config();
14. }
```





```
15.      /* 配置 RTC 秒中断优先级 */
16.      NVIC_Configuration();
17.
18.      printf( "\r\n This is a RTC demo..... \r\n" );
19.
20.      if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
21.      {
22.          /* Backup data register value is not correct or not yet programmed (when
23.             the first time the program is executed) */
24.          printf("\r\nThis is a RTC demo!\r\n");
25.          printf("\r\n\n RTC not yet configured....");
26.
27.          /* RTC Configuration */
28.          RTC_Configuration();
29.
30.          printf("\r\n RTC configured....");
31.
32.          /* Adjust time by values entered by the user on the hyperterminal */
33.          Time_Adjust();
34.
35.          BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
36.      }
37.      else
38.      {
39.          /* Check if the Power On Reset flag is set */
40.          if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
41.          {
42.              printf("\r\n\n Power On Reset occurred....");
43.          }
44.          /* Check if the Pin Reset flag is set */
45.          else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
46.          {
47.              printf("\r\n\n External Reset occurred....");
48.          }
49.
50.          printf("\r\n No need to configure RTC....");
51.          /* Wait for RTC registers synchronization */
52.          RTC_WaitForSynchro();
53.
54.          /* Enable the RTC Second */
55.          RTC_ITConfig(RTC_IT_SEC, ENABLE);
56.          /* Wait until last write operation on RTC registers has finished */
57.          RTC_WaitForLastTask();
58.      }
59.
60. #ifdef RTCClockOutput_Enable
61.      /* Enable PWR and BKP clocks */
62.      RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);
63.
64.      /* Allow access to BKP Domain */
65.      PWR_BackupAccessCmd(ENABLE);
66.
67.      /* Disable the Tamper Pin */
68.      BKP_TamperPinCmd(DISABLE); /* To output RTCCLK/64 on Tamper pin, the tamper
69.                                     functionality must be disabled */
70.
71.      /* Enable RTC Clock Output on Tamper Pin */
72.      BKP_RTCOutputConfig(BKP_RTCOutputSource_CalibClock);
73. #endif
74.
75.      /* Clear reset flags */
76.      RCC_ClearFlag();
77.
78.      /* Display time in infinite loop */
79.      Time_Show();
80.      while (1)
81.      {
82.      }
83.
84. }
85.
```



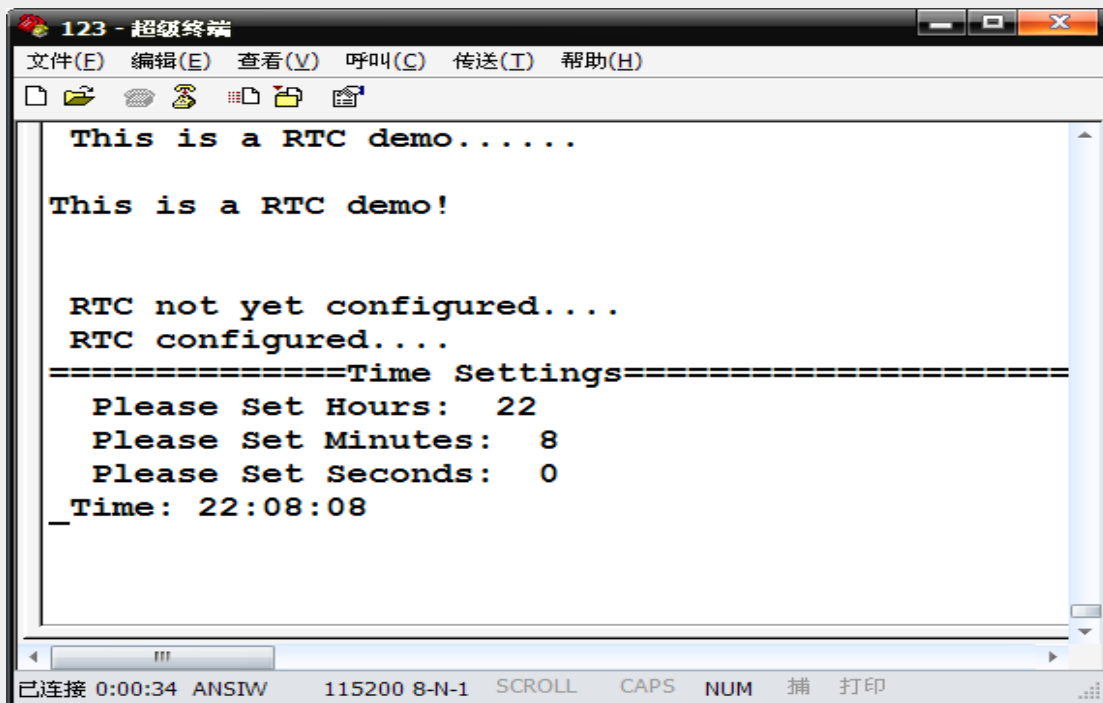
在 main 函数开始首先调用库函数 `SystemInit();` 将我的系统时钟初始化为 72M。

因为我们在实验中需要用到串口，所以我们调用 `USART1_Config();` 函数将串口配置好。`SystemInit();` 和 `USART1_Config();` 这两个函数已在前面相关的教程中讲解过，这里不再详述。

`NVIC_Configuration();` 函数用于配置 RTC（实时时钟）的中断优先级，我们将它的主优先级设置为 1，次优先级为 0。这里只用到了 RTC 一个中断，所以 RTC 的主和次优先级不必太关心。

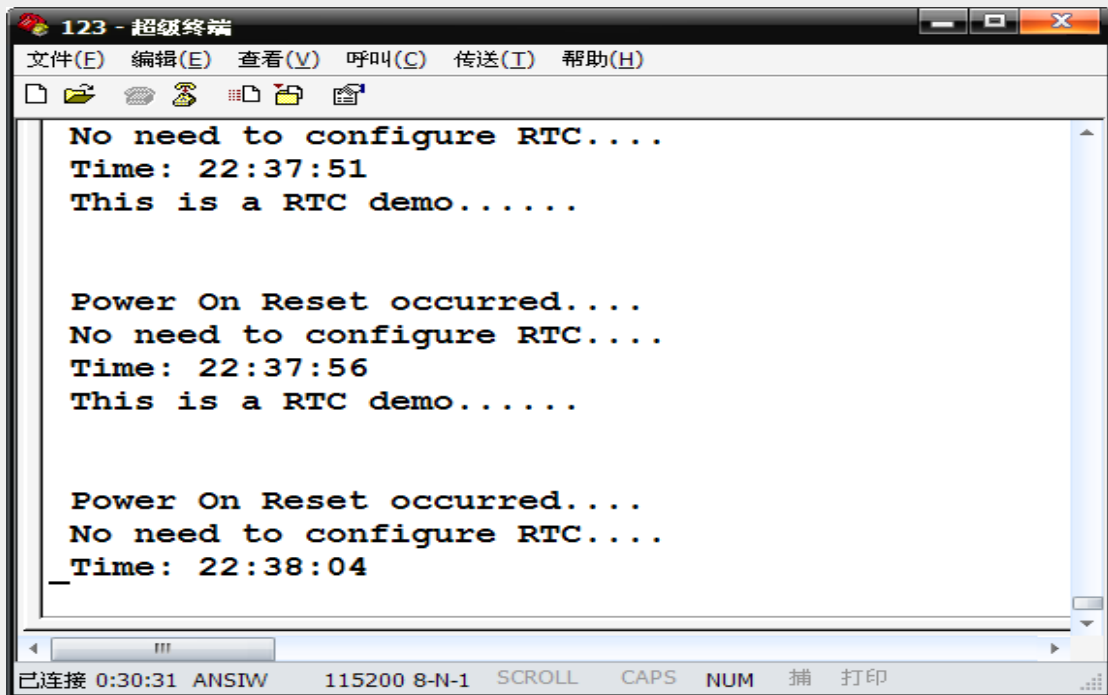
接下来的代码部分就是真正跟 RTC 有关的啦：

(1) `if()` 部分首先读取 RTC 备份寄存器里面的值，看看备份寄存器里面的值是否正确（如果 RTC 曾经被设置过的话，备份寄存器里面的值为 0XA5A5）或判断这是不是第一次对 RTC 编程。如果这两种情况有任何一种发生的话，则调用 `RTC_Configuration();`（在 `rtc.c` 中实现）函数来初始化 RTC，并往电脑的超级终端打印出相应的调试信息。初始化好 RTC 之后，调用函数 `Time_Adjust();`（在 `rtc.c` 中实现）让用户键入（通过超级终端输入）时间值，如下截图所示：



当我们输入时间值后，RTC 时钟就运行起来了。我这里显示的时间是与我电脑上的时间一样的。设置好时间后，我们把 0XA5A5 这个值写入 RTC 的备份寄存器，这样当我们下一次上电时就不用重新输入 RTC 里面的时间值了。

(2) 如果 RTC 值曾经被设置过，则进入 `else()` 部分。`else` 部分检测是上电复位 还是按键复位，根据不同的复位情况在超级终端中打印出不同的调试信息，但这两种复位都不需要重新设置 RTC 里面的时间值。当检测到系统上电复位时，打印出如下信息：



```
123 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
No need to configure RTC....
Time: 22:37:51
This is a RTC demo.....

Power On Reset occurred....
No need to configure RTC....
Time: 22:37:56
This is a RTC demo.....

Power On Reset occurred....
No need to configure RTC....
Time: 22:38:04
_

已连接 0:30:31 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

当检测到系统按键复位时，打印出如下信息：





```
123 - 超级终端
文件(E) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
No need to configure RTC....
Time: 22:38:47
This is a RTC demo.....

External Reset occurred....
No need to configure RTC....
Time: 22:38:48
This is a RTC demo.....

External Reset occurred....
No need to configure RTC....
Time: 22:38:53
_

已连接 0:31:20 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

```
123 - 超级终端
文件(E) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
RTC not yet configured....

RTC configured....
=====Time Settings=====
请输入年份(Please Set Years): 20
年份被设置为: 2011

请输入月份(Please Set Months):
月份被设置为: 9

请输入日期(Please Set Dates):
日期被设置为: 7

请输入时钟(Please Set Hours):
时钟被设置为: 20

请输入分钟(Please Set Minutes):
分钟被设置为: 59

请输入秒钟(Please Set Seconds):
秒钟被设置为: 30

今天农历: 2011,08,10 辛卯年八月初十 离白露还有01天
当前时间为: 2011年(兔年) 9月 7日 (星期三) 21:00:28_

已连接 4:04:13 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```



(3) 条件编译选项部分问我们是否需要 `output RTCCLK/64 on`

`Tamperpin`, (在 `rtc.c` 中实现) 因为 RTC 可以在 PC13 这个引脚输出时钟信号, 这个时钟信号可以作为其他外设的时钟。野火 STM32 开发板中没用到这个时钟信号, 所以我们没定义 `RTCClockOutput_Enable` 这个宏。假如用户需要用到这个时钟信号的话, 只需在头文件中定义 `RTCClockOutput_Enable` 这个宏即可。

(4) 一切就绪之后, 我们调用 `Time_Show()`; 函数将我们的时间显示在电脑的超级终端上。 `Time_Show()`; 在 `rtc.c` 中实现:

```
1.  /*
2.  * 函数名: Time_Show
3.  * 描述   : 在超级终端中显示当前时间值
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8.  void Time_Show(void)
9.  {
10.     printf("\n\r");
11.
12.     /* Infinite loop */
13.     while (1)
14.     {
15.         /* If 1s has passed */
16.         if (TimeDisplay == 1)
17.         {
18.             /* Display current time */
19.             Time_Display(RTC_GetCounter());
20.             TimeDisplay = 0;
21.         }
22.     }
23. }
```

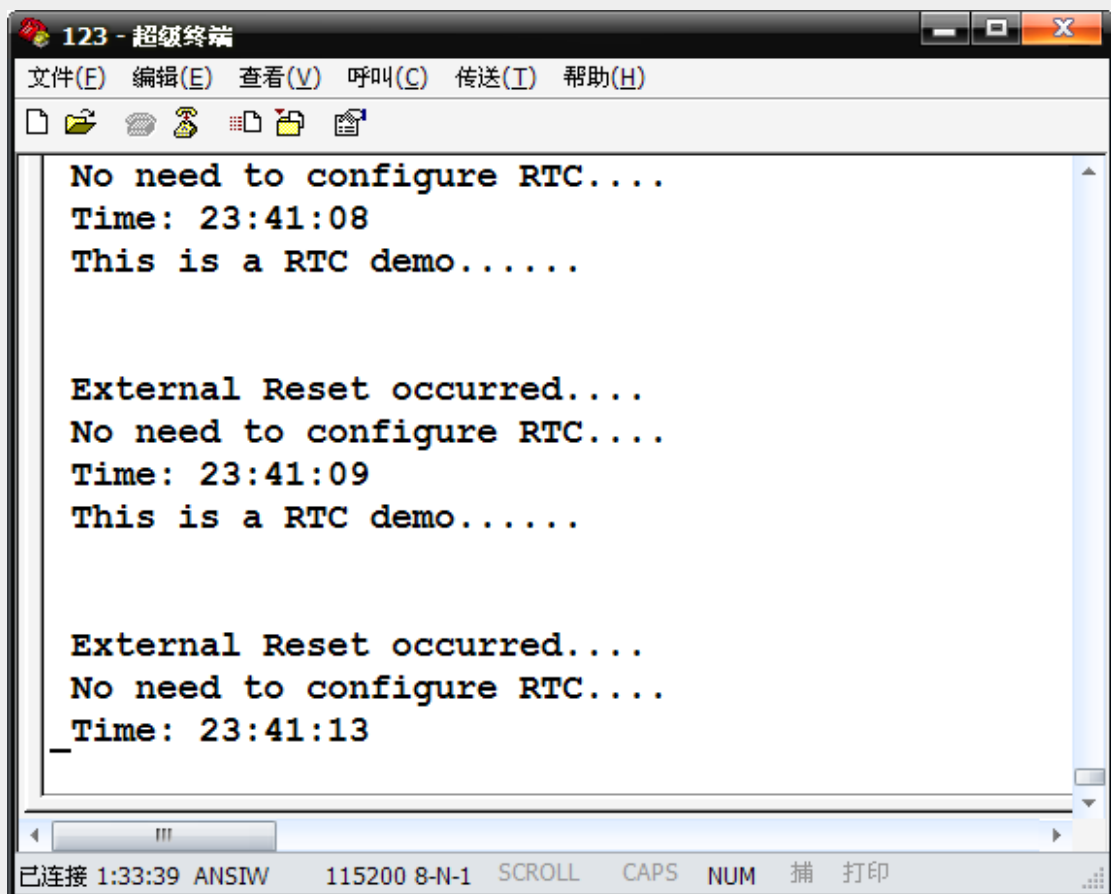
其中 `TimeDisplay` 是 RTC 秒中断标志, 当 RTC 秒中断一次的话, RTC 时间计数器就实现秒加 1, 函数 `Time_Display(RTC_GetCounter());` 就将这些时间值转换成 HH:MM:SS 的格式显示出来, 时间更新显示完之后 `TimeDisplay` 清 0。 `TimeDisplay` 在 RTC 秒中断服务程序中置位:

```
1.  /**
2.  * @brief This function handles RTC global interrupt request.
3.  * @param None
4.  * @retval : None
5.  */
6.  void RTC_IRQHandler(void)
7.  {
8.     if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
9.     {
10.        /* Clear the RTC Second interrupt */
11.        RTC_ClearITPendingBit(RTC_IT_SEC);
12.
13.        /* Toggle GPIO_LED pin 6 each 1s */
```



```
14. //GPIO_WriteBit(GPIO_LED, GPIO_Pin_6, (BitAction)(1 -
    GPIO_ReadOutputDataBit(GPIO_LED, GPIO_Pin_6)));
15.
16. /* Enable time update */
17. TimeDisplay = 1;
18.
19. /* Wait until last write operation on RTC registers has finished */
20. RTC_WaitForLastTask();
21. /* Reset RTC Counter when Time is 23:59:59 */
22. if (RTC_GetCounter() == 0x00015180)
23. {
24.     RTC_SetCounter(0x0);
25.     /* Wait until last write operation on RTC registers has finished */
26.     RTC_WaitForLastTask();
27. }
28. }
29. }
```

现在我电脑的时间是 23:41:13。我们把它写到 RTC 的寄存器中，然后在超级终端上显示出来，效果图如下：



过瘾哩，以前我们想实现个时钟的时候还需借助时钟芯片，如 DS1302 或 DS12C887，而现在我们用一个定时器就搞定。不过我们接下来来点更过瘾的，只用 RTC 这个定时器 实现我们的超级日历，日历包括如下功能：

时钟： 如 23: 40: 50

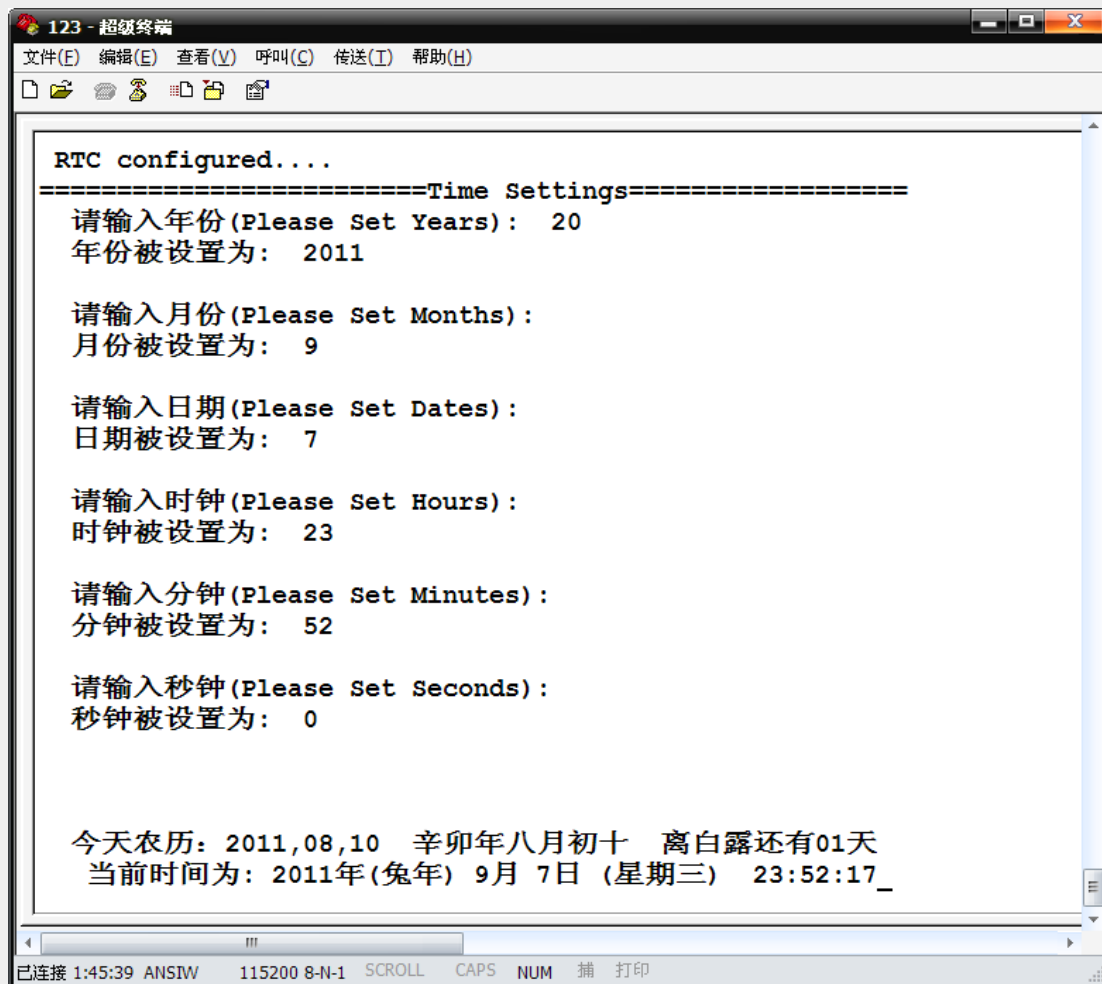
阴历： 如 辛卯年八月初十



阳历： 如 2011-9-7

年份： 如 兔年

24 节气：如 夏至



这个日历以 1970 年为计时元年，用 32bit 的时间寄存器可以运行到 2100 年左右。之所以以 1970 年为计时元年是因为这份代码是从 LINUX 里面移植过来的，而 LINUX 的诞辰就是 1970 年，我想这样做应该为为了纪念 LINUX (纯属个人观点，没有考证)。不过计时起始元年可调，可调到随便纪念谁:))。

这里就暂且给出个效果图先，分析源码的任务就给大家了。里面涉及到些算法和结构体的知识，大家就好好琢磨吧。源码在野火 STM32 开发板的光盘目录下《11-野火 M3-Calendar》。

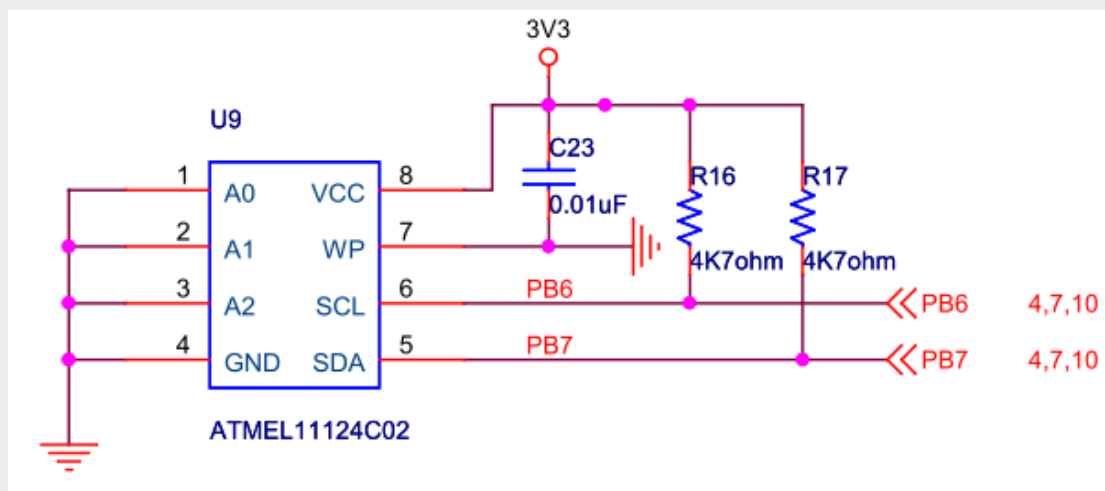


5、IIC (EEPROM)

5.1 实验描述及工程文件清单

实验描述	向 EEPROM 写入数据，再读取出来，进行校验，通过串口打印写入与读取出来的数据，并输出校验结果。
硬件连接	PB6-I2C1_SCL, PB7-I2C1_SDA
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_i2c.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/i2c_ee.c

野火 STM32 开发板 I2C-EEPROM 硬件原理图：



5.2 I2C 简介

I2C(芯片间)总线接口连接微控制器和串行 I2C 总线。它提供多主机功能，控制所有 I2C 总线特定的时序、协议、仲裁和定时。支持标准和快速两种模式，stm32 的 I2C 可以使用 DMA 方式操作。

野火 STM32 开发板用的是 [STM32F103VET6](#)。它有 2 个 I2C 接口。I/O 口定义为 [PB6-I2C1_SCL](#), [PB7-I2C1_SDA](#); [PB10-I2C2_SCL](#), [PB11-I2C2_SDA](#)。本实验使用 [I2C1](#)，对应地连接到 EERPOM（型号：[AT24C02](#)）的 [SCL](#) 和 [SDA](#) 线。实现 I2C 通讯，对 EERPOM 进行读写。

本实验采用主模式，分别用作主发送器和主接收器。通过查询事件的方式来确保正常通讯。

5.3 代码分析

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

```
1. stm32f10x_gpio.c
2. stm32f10x_rcc.c
3. stm32f10x_usart.c
4. stm32f10x_i2c.c
```

还要在 [stm32f10x_conf.h](#) 中把相应的头文件添加进来：

```
1. /* Uncomment the line below to enable peripheral header file inclusion
   */
2. /* #include "stm32f10x_adc.h" */
3. /* #include "stm32f10x_bkp.h" */
4. /* #include "stm32f10x_can.h" */
5. /* #include "stm32f10x_crc.h" */
6. /* #include "stm32f10x_dac.h" */
7. /* #include "stm32f10x_dbgmcu.h" */
8. /* #include "stm32f10x_dma.h" */
9. /* #include "stm32f10x_exti.h" */
10. /* #include "stm32f10x_flash.h" */
11. /* #include "stm32f10x_fsmc.h" */
12. #include "stm32f10x_gpio.h"
13. #include "stm32f10x_i2c.h"
14. /* #include "stm32f10x_iwdg.h" */
15. /* #include "stm32f10x_pwr.h" */
16. #include "stm32f10x_rcc.h"
17. /* #include "stm32f10x_rtc.h" */
18. /* #include "stm32f10x_sdio.h" */
```



```
19. /* #include "stm32f10x_spi.h" */
20. /* #include "stm32f10x_tim.h" */
21. #include "stm32f10x_usart.h"
22. /* #include "stm32f10x_wwdg.h" */
23. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (a
    dd-on to CMSIS functions) */
```

配置好所需的库文件之后，我们就从 `main` 函数开始分析：

```
1. /*
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 返回   : 无
7.  */
8. int main(void)
9. {
10.    /* 配置系统时钟为 72M */
11.    SystemInit();
12.
13.    /* 串口1 初始化 */
14.    USART1_Config();
15.
16.    /* I2C 外设初(AT24C02)始化 */
17.    I2C_EE_Init();
18.
19.    USART1_printf(USART1, "\r\n 这是一个 I2C 外设(AT24C02)读写测试例
    程 \r\n");
20.    USART1_printf(USART1, "\r\n (" __DATE__ " -
    " __TIME__ ") \r\n");
21.
22.    I2C_Test();
23.
24.    while (1)
25.    {
26.    }
27. }
```

系统库函数 `SystemInit()`；将系统时钟设置为 72M，`USART1_Config()`；配置串口，关于这两个函数的具体讲解可以参考前面的教程，这里不再详述。/*

```
1.  * 函数名: I2C_EE_Init
2.  * 描述   : I2C 外设(EEPROM)初始化
3.  * 输入   : 无
4.  * 输出   : 无
5.  * 调用   : 外部调用
6.  */
7. void I2C_EE_Init(void)
8. {
9.
10.    I2C_GPIO_Config();
11.
12.    I2C_Mode_Config();
13.
14.    /* 根据头文件 i2c_ee.h 中的定义来选择 EEPROM 要写入的地址 */
15.    #ifdef EEPROM_Block0_ADDRESS
16.    /* 选择 EEPROM Block0 来写入 */
17.    EEPROM_ADDRESS = EEPROM_Block0_ADDRESS;
18.    #endif
```



```
19.
20. #ifndef EEPROM_Block1_ADDRESS
21.     /* 选择 EEPROM Block1 来写入 */
22.     EEPROM_ADDRESS = EEPROM_Block1_ADDRESS;
23. #endif
24.
25. #ifndef EEPROM_Block2_ADDRESS
26.     /* 选择 EEPROM Block2 来写入 */
27.     EEPROM_ADDRESS = EEPROM_Block2_ADDRESS;
28. #endif
29.
30. #ifndef EEPROM_Block3_ADDRESS
31.     /* 选择 EEPROM Block3 来写入 */
32.     EEPROM_ADDRESS = EEPROM_Block3_ADDRESS;
33. #endif }
```

`I2C_EE_Init()`; 是用户编写的函数, 其中调用了 `I2C_GPIO_Config()`; 配置好 I2C 所用的 I/O 端口, 调用 `I2C_Mode_Configu()`; 设置 I2C 的工作模式。并使能相关外设的时钟。其中的条件编译确定了 EEPROM 的器件地址, 按我们的硬件设置方式, 地址为 `0xA0`;

```
1. /*
2.  * 函数名: I2C_EE_Test
3.  * 描述   : I2C(AT24C02) 读写测试。
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 返回   : 无
7.  */
8. void I2C_Test(void)
9. {
10.     u16 i;
11.
12.     printf("写入的数据\n\r");
13.
14.     for ( i=0; i<=255; i++ ) //填充缓冲
15.     {
16.         I2c_Buf_Write[i] = i;
17.
18.         printf("0x%02X ", I2c_Buf_Write[i]);
19.         if(i%16 == 15)
20.             printf("\n\r");
21.     }
22.
23.     //将 I2c_Buf_Write 中顺序递增的数据写入 EEPROM 中
24.     I2C_EE_BufferWrite( I2c_Buf_Write, EEP_Firstpage, 256);
25.
26.     printf("\n\r 读出的数据\n\r");
27.     //将 EEPROM 读出数据顺序保持到 I2c_Buf_Read 中
28.     I2C_EE_BufferRead(I2c_Buf_Read, EEP_Firstpage, 256);
29.
30.     //将 I2c_Buf_Read 中的数据通过串口打印
31.     for (i=0; i<256; i++)
32.     {
33.         if(I2c_Buf_Read[i] != I2c_Buf_Write[i])
34.         {
35.             printf("0x%02X ", I2c_Buf_Read[i]);
36.             printf("错误:I2C EEPROM 写入与读出的数据不一致\n\r");
37.             return;
38.         }
39.         printf("0x%02X ", I2c_Buf_Read[i]);
```



```
40.     if(i%16 == 15)
41.         printf("\n\r");
42.
43.     }
44.     printf("I2C(AT24C02) 读写测试成功\n\r");
45. }
```

I2C_Test(void) 是这个例程中最主要的部分，把 0~255 按顺序填入缓冲区并通过串口打印到端口，接着把缓冲区的数据通过调用 I2C_EE_BufferWrite() 函数写入 EEPROM。

```
1.  /*
2.  * 函数名: I2C_EE_BufferWrite
3.  * 描述   : 将缓冲区中的数据写到 I2C EEPROM 中
4.  * 输入   : -pBuffer 缓冲区指针
5.  *          -WriteAddr 接收数据的 EEPROM 的地址
6.  *          -NumByteToWrite 要写入 EEPROM 的字节数
7.  * 输出   : 无
8.  * 返回   : 无
9.  * 调用   : 外部调用
10. */
11. void I2C_EE_BufferWrite(u8* pBuffer, u8 WriteAddr, u16 NumByteToWrite)
12. {
13.     u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0;
14.
15.     Addr = WriteAddr % I2C_PageSize;
16.     count = I2C_PageSize - Addr;
17.     NumOfPage = NumByteToWrite / I2C_PageSize;
18.     NumOfSingle = NumByteToWrite % I2C_PageSize;
19.
20.     /* If WriteAddr is I2C_PageSize aligned */
21.     if(Addr == 0)
22.     {
23.         /* If NumByteToWrite < I2C_PageSize */
24.         if(NumOfPage == 0)
25.         {
26.             I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
27.             I2C_EE_WaitEepromStandbyState();
28.         }
29.         /* If NumByteToWrite > I2C_PageSize */
30.         else
31.         {
32.             while (NumOfPage--)
33.             {
34.                 I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
35.                 I2C_EE_WaitEepromStandbyState();
36.                 WriteAddr += I2C_PageSize;
37.                 pBuffer += I2C_PageSize;
38.             }
39.
40.             if(NumOfSingle!=0)
41.             {
42.                 I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
43.                 I2C_EE_WaitEepromStandbyState();
44.             }
45.         }
46.     }
47.     /* If WriteAddr is not I2C_PageSize aligned */
48.     else
49.     {
50.         /* If NumByteToWrite < I2C_PageSize */
```



```
51.     if(NumOfPage== 0)
52.     {
53.         I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
54.         I2C_EE_WaitEepromStandbyState();
55.     }
56.     /* If NumByteToWrite > I2C_PageSize */
57.     else
58.     {
59.         NumByteToWrite -= count;
60.         NumOfPage =  NumByteToWrite / I2C_PageSize;
61.         NumOfSingle = NumByteToWrite % I2C_PageSize;
62.
63.         if(count != 0)
64.         {
65.             I2C_EE_PageWrite(pBuffer, WriteAddr, count);
66.             I2C_EE_WaitEepromStandbyState();
67.             WriteAddr += count;
68.             pBuffer += count;
69.         }
70.
71.         while (NumOfPage--)
72.         {
73.             I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
74.             I2C_EE_WaitEepromStandbyState();
75.             WriteAddr += I2C_PageSize;
76.             pBuffer += I2C_PageSize;
77.         }
78.         if(NumOfSingle != 0)
79.         {
80.             I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
81.             I2C_EE_WaitEepromStandbyState();
82.         }
83.     }
84. }
85. }
```

因为 AT24C02 型号的 EEPROM 按页写入方式中每页最大字节数为 8 字节，若超过 8 字节则会在该页的起始地址覆盖数据，因此需要

I2C_EE_BufferWrite() 函数处理写入位置和缓冲区的地址。把处理好的地址交给 I2C_EE_PageWrite() 函数，这个函数是与 EEPROM 进行 I2C 通讯的最底层函数，以下我们通过分析 I2C_EE_PageWrite() 来了解 stm32 的 I2C 通讯方法。

```
1.  /*
2.  * 函数名: I2C_EE_PageWrite
3.  * 描述   : 在 EEPROM 的一个写循环中可以写多个字节，但一次写入的字节数
4.  *         不能超过 EEPROM 页的大小。AT24C02 每页有 8 个字节。
5.  * 输入    : -pBuffer 缓冲区指针
6.  *         -WriteAddr 接收数据的 EEPROM 的地址
7.  *         -NumByteToWrite 要写入 EEPROM 的字节数
8.  * 输出    : 无
9.  * 返回    : 无
10. * 调用    : 外部调用
11. */
12. void I2C_EE_PageWrite(u8* pBuffer, u8 WriteAddr, u8 NumByteToWrite)
13. {
14.     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)); // Added by Najoua
15.     27/08/2008
16.     /* Send START condition */
```

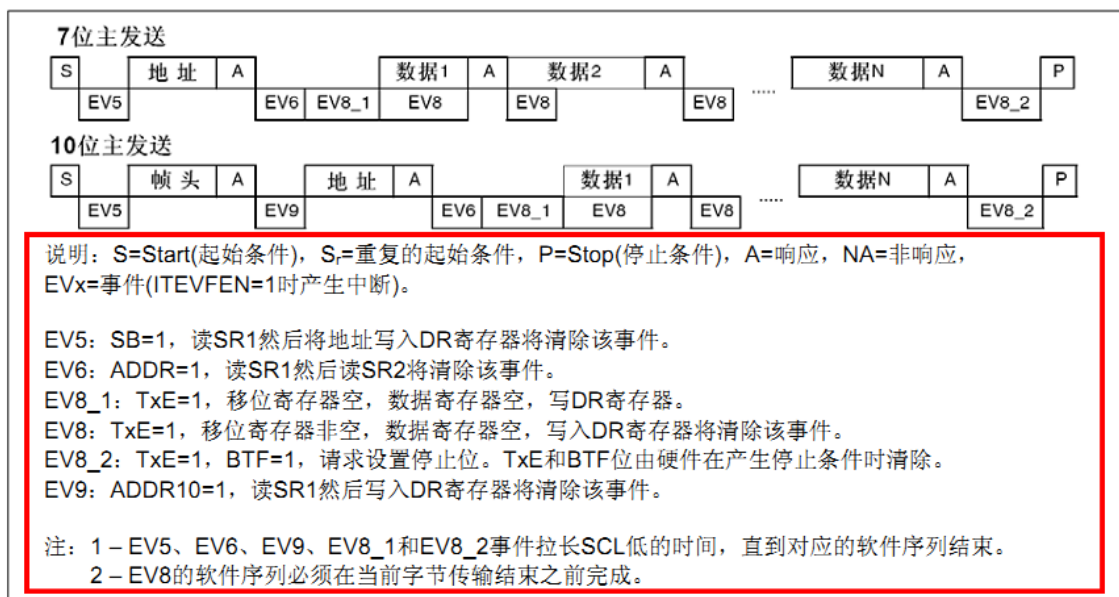


```
17. I2C_GenerateSTART(I2C1, ENABLE);
18.
19. /* Test on EV5 and clear it */
20. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
21.
22. /* Send EEPROM address for write */
23. I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter)
;
24.
25. /* Test on EV6 and clear it */
26. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECT
ED));
27.
28. /* Send the EEPROM's internal address to write to */
29. I2C_SendData(I2C1, WriteAddr);
30.
31. /* Test on EV8 and clear it */
32. while(! I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
33.
34. /* While there is data to be written */
35. while(NumByteToWrite--)
36. {
37.     /* Send the current byte */
38.     I2C_SendData(I2C1, *pBuffer);
39.
40.     /* Point to the next byte to be written */
41.     pBuffer++;
42.
43.     /* Test on EV8 and clear it */
44.     while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
45. }
46. /* Send STOP condition */
47. I2C_GenerateSTOP(I2C1, ENABLE);
48. }
```

从 stm32 参考手册的序列图可以看到，在 I2C 的通讯过程中，会产生一系列的事件，出现事件后在相应的寄存器中会产生标志位。

截图来自《STM32 参考手册中文》。

图245 主发送器传送序列图



我们在做出 I2C 通讯操作时，可以通过循环调用库函数 `I2C_CheckEvent()` 进行查询，以确保上一操作完成后才发出下一个

I2C 通讯信号。如：在确定 SDA 总线空闲的之后，作为主发送器的 stm32 发出起始信号，若成功，这时会产生“事件 5”（EV5），我们调用

`while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));` 来检测这个事件，确保检测到之后再执行下一操作。“`I2C_EVENT_MASTER_MODE_SELECT`”在固件函数库中可以查到这就是“EV5”的宏，后面的相应操作类似。

现在我们回到 `I2C_EE_BufferWrite()` 这个函数，在每次调用完 `I2C_EE_PageWrite()` 后，都调用了一个 `I2C_EE_WaitEepromStandbyState()` 函数。

```
1.  /*
2.  * 函数名: I2C_EE_WaitEepromStandbyState
3.  * 描述   : Wait for EEPROM Standby state
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 返回   : 无
7.  * 调用   :
8.  */
9. void I2C_EE_WaitEepromStandbyState(void)
10. {
11.     vu16 SR1_Tmp = 0;
12.
13.     do
14.     {
15.         /* Send START condition */
16.         I2C_GenerateSTART(I2C1, ENABLE);
17.         /* Read I2C1 SR1 register */
18.         SR1_Tmp = I2C_ReadRegister(I2C1, I2C_Register_SR1);
19.         /* Send EEPROM address for write */
20.         I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);
21.     } while (!(I2C_ReadRegister(I2C1, I2C_Register_SR1) & 0x0002));
22.
23.     /* Clear AF flag */
24.     I2C_ClearFlag(I2C1, I2C_FLAG_AF);
25.     /* STOP condition */
26.     I2C_GenerateSTOP(I2C1, ENABLE); // Added by Najoua 27/08/2008
27. }
```

这是利用了 EEPROM 在接收完数据后，启动内部周期写入数据的时间内不会对主机的请求作出应答的特性。所以这个函数循环发送起始信号，若检测到 EEPROM 的应答，则说明 EEPROM 已经完成上一步的数据写入，进入 Standby 状态，可以进行下一步的操作了。

回到 `I2C_Test()` 这个函数，再分析一下它调用的读 EEPROM 函数

`I2C_EE_BufferRead()`。



```
1.  /*
2.  * 函数名: I2C_EE_BufferRead
3.  * 描述   : 从EEPROM里面读取一块数据。
4.  * 输入   : -pBuffer 存放从EEPROM读取的数据的缓冲区指针。
5.  *          -WriteAddr 接收数据的EEPROM的地址。
6.  *          -NumByteToWrite 要从EEPROM读取的字节数。
7.  * 输出   : 无
8.  * 返回   : 无
9.  * 调用   : 外部调用
10. */
11. void I2C_EE_BufferRead(u8* pBuffer, u8 ReadAddr, u16 NumByteToRead)
12. {
13.     /*((u8 *)0x4001080c) |= 0x80;
14.     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)); // Added by Najoua
15.     27/08/2008
16.
17.     /* Send START condition */
18.     I2C_GenerateSTART(I2C1, ENABLE);
19.     /*((u8 *)0x4001080c) &= ~0x80;
20.
21.     /* Test on EV5 and clear it */
22.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
23.
24.     /* Send EEPROM address for write */
25.     I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter)
26.     ;
27.     /* Test on EV6 and clear it */
28.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECT
29.     ED));
30.     /* Clear EV6 by setting again the PE bit */
31.     I2C_Cmd(I2C1, ENABLE);
32.
33.     /* Send the EEPROM's internal address to write to */
34.     I2C_SendData(I2C1, ReadAddr);
35.
36.     /* Test on EV8 and clear it */
37.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
38.
39.     /* Send STRAT condition a second time */
40.     I2C_GenerateSTART(I2C1, ENABLE);
41.
42.     /* Test on EV5 and clear it */
43.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
44.
45.     /* Send EEPROM address for read */
46.     I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Receiver);
47.
48.     /* Test on EV6 and clear it */
49.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED)
50.     );
51.     /* While there is data to be read */
52.     while(NumByteToRead)
53.     {
54.         if(NumByteToRead == 1)
55.         {
56.             /* Disable Acknowledgement */
57.             I2C_AcknowledgeConfig(I2C1, DISABLE);
58.
59.             /* Send STOP Condition */
60.             I2C_GenerateSTOP(I2C1, ENABLE);
61.         }
62.     }
```



```
63.     /* Test on EV7 and clear it */
64.     if(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED))
65.     {
66.         /* Read a byte from the EEPROM */
67.         *pBuffer = I2C_ReceiveData(I2C1);
68.
69.         /* Point to the next location where the byte read will be saved
        */
70.         pBuffer++;
71.
72.         /* Decrement the read bytes counter */
73.         NumByteToRead--;
74.     }
75. }
76.
77. /* Enable Acknowledgement to be ready for another reception */
78. I2C_AcknowledgeConfig(I2C1, ENABLE);
79. }
```

这个读 EEPROM 函数与写的类似，也是利用 `I2C_CheckEvent()` 来确保通讯正常进行的，要注意一下的是读取数据时遵循 I2C 的标准，主发送器 stm32 要发出两次起始 I2C 讯号才能建立通讯。

最后，总结一下在 stm32 如何建立与 EEPROM 的通讯。

1、 配置 I/O 端口，确定并配置 I2C 的模式，使能 GPIO 和 I2C 时钟。

2、 写：

检测 SDA 是否空闲；

->按 I2C 协议发出起始讯号；

->发出 7 位器件地址和写模式；

->要写入的存储区首地址；

->用页写入方式或字节写入方式写入数据；

每个操作之后要检测“事件”确定是否成功。写完后检测 EEPROM 是否进入 standby 状态。

3、 读：

检测 SDA 是否空闲；

->按 I2C 协议发出起始讯号；

->发出 7 位器件地址和写模式（伪写）；

->发出要读取的存储区首地址；



->重发起始讯号;

->发出 7 位器件地址和读模式;

->接收数据;

类似写操作, 每个操作之后要检测“事件”确定是否成功。

5.4 实验现象

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 插上串口线(两头都是母的交叉线), 打开超级终端, 配置超级终端为 115200 8-N-1, 将编译好的程序下载到开发板, 即可看到超级终端打印出如下信息:

写入的数据:

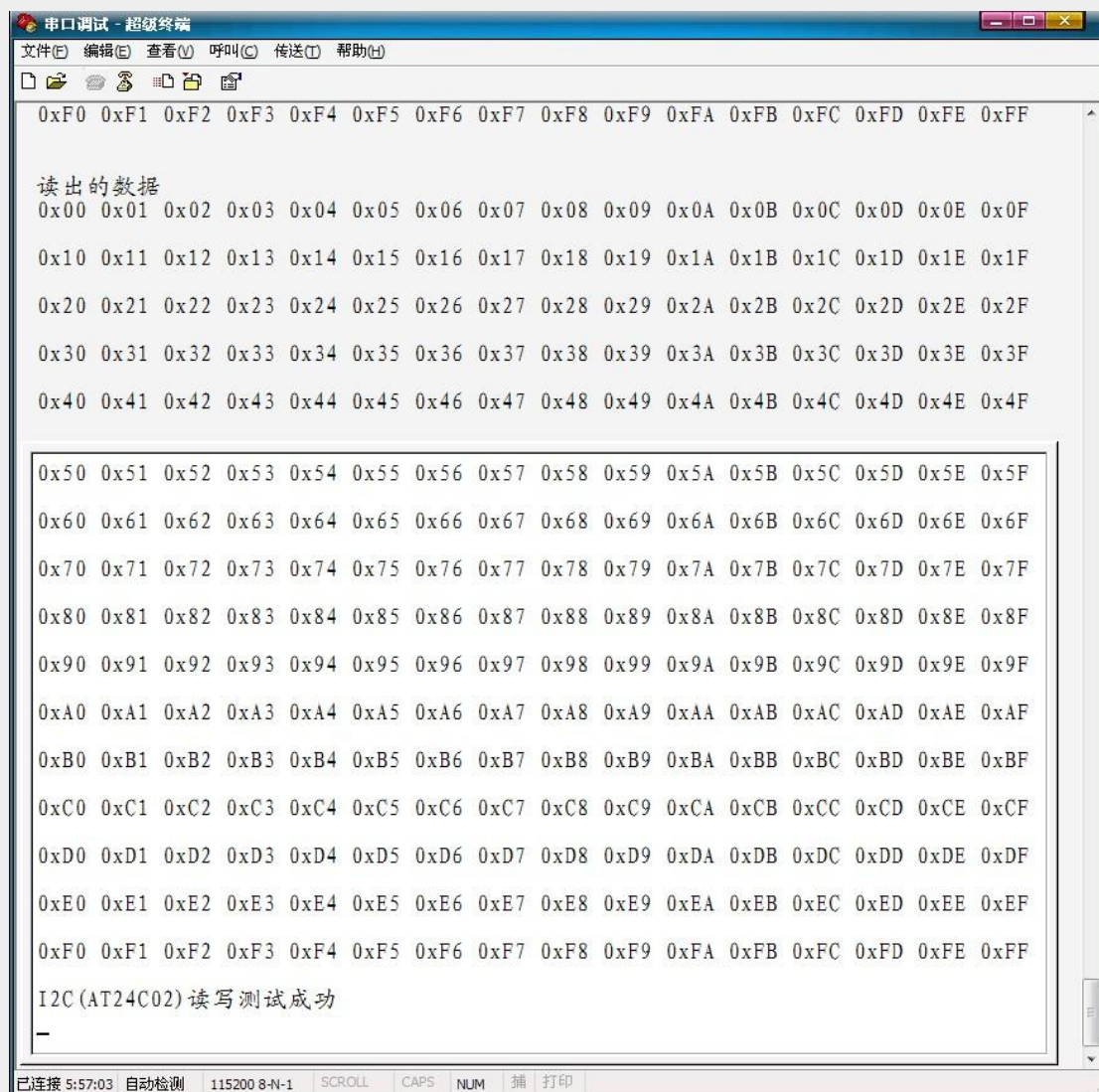
```
0xF0 0xF1 0xF2 0xF3 0xF4 0xF5 0xF6 0xF7 0xF8 0xF9 0xFA 0xFB 0xFC 0xFD 0xFE 0xFF

这是一个I2C外设(AT24C02)读写测试例程

(Oct 27 2011 - 20:00:11)
写入的数据
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2A 0x2B 0x2C 0x2D 0x2E 0x2F
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3A 0x3B 0x3C 0x3D 0x3E 0x3F
0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4A 0x4B 0x4C 0x4D 0x4E 0x4F
0x50 0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5A 0x5B 0x5C 0x5D 0x5E 0x5F
0x60 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68 0x69 0x6A 0x6B 0x6C 0x6D 0x6E 0x6F
0x70 0x71 0x72 0x73 0x74 0x75 0x76 0x77 0x78 0x79 0x7A 0x7B 0x7C 0x7D 0x7E 0x7F
0x80 0x81 0x82 0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8A 0x8B 0x8C 0x8D 0x8E 0x8F
0x90 0x91 0x92 0x93 0x94 0x95 0x96 0x97 0x98 0x99 0x9A 0x9B 0x9C 0x9D 0x9E 0x9F
0xA0 0xA1 0xA2 0xA3 0xA4 0xA5 0xA6 0xA7 0xA8 0xA9 0xAA 0xAB 0xAC 0xAD 0xAE 0xAF
0xB0 0xB1 0xB2 0xB3 0xB4 0xB5 0xB6 0xB7 0xB8 0xB9 0xBA 0xBB 0xBC 0xBD 0xBE 0xBF
0xC0 0xC1 0xC2 0xC3 0xC4 0xC5 0xC6 0xC7 0xC8 0xC9 0xCA 0xCB 0xCC 0xCD 0xCE 0xCF
0xD0 0xD1 0xD2 0xD3 0xD4 0xD5 0xD6 0xD7 0xD8 0xD9 0xDA 0xDB 0xDC 0xDD 0xDE 0xDF
0xE0 0xE1 0xE2 0xE3 0xE4 0xE5 0xE6 0xE7 0xE8 0xE9 0xEA 0xEB 0xEC 0xED 0xEE 0xEF
0xF0 0xF1 0xF2 0xF3 0xF4 0xF5 0xF6 0xF7 0xF8 0xF9 0xFA 0xFB 0xFC 0xFD 0xFE 0xFF
```



读出的数据：校验结果，读取出的数据与写入的一致，实验成功！



```
串口调试-超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
[Icons]

0xF0 0xF1 0xF2 0xF3 0xF4 0xF5 0xF6 0xF7 0xF8 0xF9 0xFA 0xFB 0xFC 0xFD 0xFE 0xFF

读出的数据
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2A 0x2B 0x2C 0x2D 0x2E 0x2F
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3A 0x3B 0x3C 0x3D 0x3E 0x3F
0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4A 0x4B 0x4C 0x4D 0x4E 0x4F

0x50 0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5A 0x5B 0x5C 0x5D 0x5E 0x5F
0x60 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68 0x69 0x6A 0x6B 0x6C 0x6D 0x6E 0x6F
0x70 0x71 0x72 0x73 0x74 0x75 0x76 0x77 0x78 0x79 0x7A 0x7B 0x7C 0x7D 0x7E 0x7F
0x80 0x81 0x82 0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8A 0x8B 0x8C 0x8D 0x8E 0x8F
0x90 0x91 0x92 0x93 0x94 0x95 0x96 0x97 0x98 0x99 0x9A 0x9B 0x9C 0x9D 0x9E 0x9F
0xA0 0xA1 0xA2 0xA3 0xA4 0xA5 0xA6 0xA7 0xA8 0xA9 0xAA 0xAB 0xAC 0xAD 0xAE 0xAF
0xB0 0xB1 0xB2 0xB3 0xB4 0xB5 0xB6 0xB7 0xB8 0xB9 0xBA 0xBB 0xBC 0xBD 0xBE 0xBF
0xC0 0xC1 0xC2 0xC3 0xC4 0xC5 0xC6 0xC7 0xC8 0xC9 0xCA 0xCB 0xCC 0xCD 0xCE 0xCF
0xD0 0xD1 0xD2 0xD3 0xD4 0xD5 0xD6 0xD7 0xD8 0xD9 0xDA 0xDB 0xDC 0xDD 0xDE 0xDF
0xE0 0xE1 0xE2 0xE3 0xE4 0xE5 0xE6 0xE7 0xE8 0xE9 0xEA 0xEB 0xEC 0xED 0xEE 0xEF
0xF0 0xF1 0xF2 0xF3 0xF4 0xF5 0xF6 0xF7 0xF8 0xF9 0xFA 0xFB 0xFC 0xFD 0xFE 0xFF

I2C (AT24C02) 读写测试成功
-

已连接 5:57:03 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```



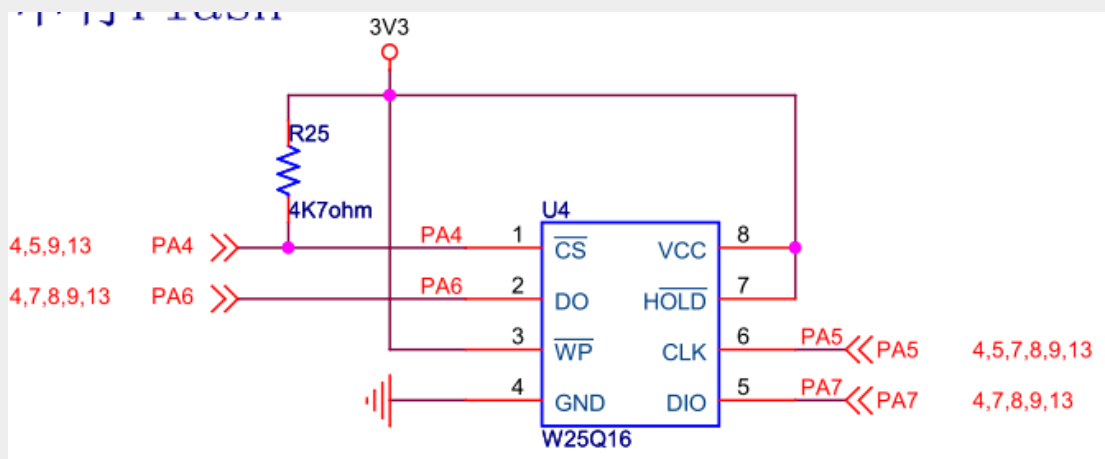
6、SPI（2M-Flash）

6.1 实验描述及工程文件清单

实验描述	读取 FLASH 的 ID 信息，写入数据，并读取出来进行校验，通过串口打印写入与读取出来的数据，输出测试结果。
硬件连接	PA4-SPI1-NSS : W25X16-CS PA5-SPI1-SCK : W25X16-CLK PA6-SPI1-MISO : W25X16-DO PA7-SPI1-MOSI : W25X16-DIO
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_spi.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/ spi_flash.c



野火 STM32 开发板 I2C-EEPROM 硬件原理图：



6.2 SPI 简介

SPI 是一种串行同步通讯协议，由一个主设备和一个或多个从设备组成，主设备启动一个与从设备的同步通讯，从而完成数据的交换。该总线大量用在与 **EEPROM**、**ADC**、**FRAM** 和显示驱动器之类的慢速外设器件通信。stm32 的 SPI 可以工作在全双工，单向发送，单向接收模式，可以使用 DMA 方式操作。

野火 STM32 开发板用的是 **STM32F103VET6**。它有 2 个 SPI 接口。本实验使用 **SPI1**，各信号线相应连接到 FLASH（型号：**W25X16**）的 **CS**、**CLK**、**DO** 和 **DIO** 线。实现 SPI 通讯，对 FLASH 进行读写。

本实验采用主模式，全双工通讯。

通过查询发送数据寄存器和接收数据寄存器状态确保通讯正常。

6.3 代码分析

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

```
5. stm32f10x_gpio.c
6. stm32f10x_rcc.c
7. stm32f10x_usart.c
8. stm32f10x_spi.c
```



还要在 `stm32f10x_conf.h` 中把相应的头文件添加进来:

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

配置好所需的库文件之后,我们就从 `main` 函数开始分析:

```
1. /*
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7. int main(void)
8. {
9.     /* 配置系统时钟为 72M */
10.    SystemInit();
11.
12.    /* 配置串口 1 为: 115200 8-N-1 */
13.    USART1_Config();
14.    printf("\r\n 这是一个 2M 串行 flash(W25X16)实验 \r\n");
15.
16.    /* 2M 串行 flash W25X16 初始化 */
17.    SPI_FLASH_Init();
18.
19.    /* Get SPI Flash Device ID */
20.    DeviceID = SPI_FLASH_ReadDeviceID();
21.
22.    Delay( 200 );
23.
24.    /* Get SPI Flash ID */
25.    FlashID = SPI_FLASH_ReadID();
26.
27.    printf("\r\n FlashID is 0x%X, Manufacturer Device ID is 0x%X\r\n",
28.           FlashID, DeviceID);
29.
30.    /* Check the SPI Flash ID */
31.    if (FlashID == sFLASH_ID) /* #define sFLASH_ID 0xEF3015 */
32.    {
33.        printf("\r\n 检测到华邦串行 flash W25X16 !\r\n");
34.
35.        /* Erase SPI FLASH Sector to write on */
36.        SPI_FLASH_SectorErase(FLASH_SectorToErase);
37.
38.        /* 将发送缓冲区的数据写到 flash 中 */
39.        SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, BufferSize);
40.
41.        printf("\r\n 写入的数据为: %s \r\t", Tx_Buffer);
42.
43.        /* 将刚刚写入的数据读出来放到接收缓冲区中 */
44.        SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);
45.        printf("\r\n 读出的数据为: %s \r\n", Rx_Buffer);
46.
47.        /* 检查写入的数据与读出的数据是否相等 */
48.        TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);
```



```
49.         if( PASSED == TransferStatus1 )
50.         {
51.             printf("\r\n 2M 串行 flash(W25X16) 测试成功!\n\r");
52.         }
53.         else
54.         {
55.             printf("\r\n 2M 串行 flash(W25X16) 测试失败!\n\r");
56.         }
57.     } // if (FlashID == sFLASH_ID)
58.     else
59.     {
60.         printf("\r\n 获取不到 W25X16 ID!\n\r");
61.     }
62.
63.     SPI_Flash_PowerDown();
64.     while(1);
65. }
```

系统库函数 `SystemInit()`；将系统时钟设置为 72M，`USART1_Config()`；配置串口，关于这两个函数的具体讲解可以参考前面的教程，这里不再详述。

```
1.  /*****
2.  * Function Name   : SPI_FLASH_Init
3.  * Description     : Initializes the peripherals used by the SPI FLASH d
   river.
4.  * Input           : None
5.  * Output          : None
6.  * Return          : None
7.  *****/
8.  void SPI_FLASH_Init(void)
9.  {
10.     SPI_InitTypeDef  SPI_InitStructure;
11.     GPIO_InitTypeDef GPIO_InitStructure;
12.
13.     /* Enable SPI1 and GPIO clocks */
14.     /*!< SPI_FLASH_SPI_CS_GPIO, SPI_FLASH_SPI_MOSI_GPIO,
15.        SPI_FLASH_SPI_MISO_GPIO, SPI_FLASH_SPI_DETECT_GPIO
16.        and SPI_FLASH_SPI_SCK_GPIO Periph clock enable */
17.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOD,
   ENABLE);
18.
19.     /*!< SPI_FLASH_SPI Periph clock enable */
20.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
21.     /*!< AFIO Periph clock enable */
22.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
23.
24.
25.     /*!< Configure SPI_FLASH_SPI pins: SCK */
26.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
27.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
28.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
29.     GPIO_Init(GPIOA, &GPIO_InitStructure);
30.
31.     /*!< Configure SPI_FLASH_SPI pins: MISO */
32.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
33.     GPIO_Init(GPIOA, &GPIO_InitStructure);
34.
35.     /*!< Configure SPI_FLASH_SPI pins: MOSI */
36.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
37.     GPIO_Init(GPIOA, &GPIO_InitStructure);
38.
39.     /*!< Configure SPI_FLASH_SPI_CS_PIN pin: SPI_FLASH Card CS pin */
40.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
```



```
41. GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
42. GPIO_Init(GPIOA, &GPIO_InitStructure);
43.
44. /* Deselect the FLASH: Chip Select high */
45. SPI_FLASH_CS_HIGH();
46.
47. /* SPI1 configuration */
48. // W25X16: data input on the DIO pin is sampled on the rising edge o
  f the CLK.
49. // Data on the DO and DIO pins are clocked out on the falling edge o
  f CLK.
50. SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;

51. SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
52. SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
53. SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
54. SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
55. SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
56. SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;

57. SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
58. SPI_InitStructure.SPI_CRCPolynomial = 7;
59. SPI_Init(SPI1, &SPI_InitStructure);
60.
61. /* Enable SPI1 */
62. SPI_Cmd(SPI1, ENABLE);
63. }
```

`SPI_FLASH_Init()` 是用户编写的函数，调用 `GPIO_Init()` 配置好 SPI 所用的 I/O 端口复用（CS 端口为普通 IO），调用 `SPI_Init()` 来设置 SPI 的工作模式。并使能相关外设的时钟。其中 `CPOL` 和 `CPHA` 用来设置 SPI 数据采样条件，根据 FLASH 的数据手册（光盘中附带资料）。

截图：

10.1 SPI OPERATIONS

10.1.1 SPI Modes

The W25X16/32/64 is accessed through an SPI compatible bus consisting of four signals: Serial Clock (CLK), Chip Select (/CS), Serial Data Input/Output (DIO) and Serial Data Output (DO). Both SPI bus operation Modes 0 (0,0) and 3 (1,1) are supported. The primary difference between Mode 0 and Mode 3 concerns the normal state of the CLK signal when the SPI bus master is in standby and data is not being transferred to the Serial Flash. For Mode 0 the CLK signal is normally low. For Mode 3 the CLK signal is normally high. In either case data input on the DIO pin is sampled on the rising edge of the CLK. Data on the DO and DIO pins are clocked out on the falling edge of CLK.

截图：

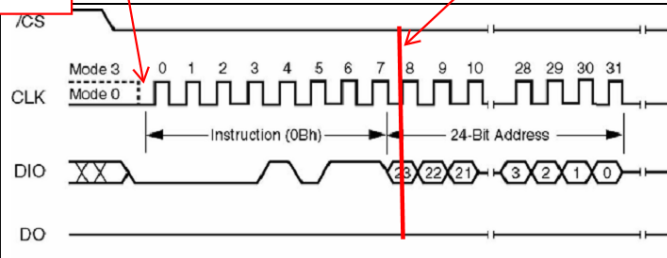


模式3的第一个时钟边沿为下降沿，FLASH的数据采样时刻为上升沿（第二个边沿），所以CPHA设置为第二个边沿有效

Fast Read (0Bh)

Fast Read instruction is similar to the Read Data instruction except that it can operate at the possible frequency of FR (see AC Electrical Characteristics). This is accomplished by adding "dummy" clocks after the 24-bit address as shown in the diagram. The dummy clocks allow the internal circuits additional time for setting up the initial data. The dummy clocks the DIO pin is a "don't care".

上升沿采集数据



这个 FLASH 支持以模式 0 和模式 3 通讯。在通讯前 CLK 既可以一直为低电平（模式 0），也可以一直为高电平（模式 3），而数据采样时刻为上升沿；也就是说野火的 `CPOL=SPI_CPOL_High` 指令，选择了模式 3，默认情况下时钟为高。因为 FLASH 只在上升沿采集数据，在时钟默认为高的情况下，第二个边沿为上升沿。所以 `CPHA = SPI_CPHA_2Edge`。CPHA 相应地设置为第二个时钟边沿（上升沿）为数据采样时刻。

```
1.  /*****
2.  * Function Name   : SPI_FLASH_ReadID
3.  * Description    : Reads FLASH identification.
4.  * Input          : None
5.  * Output         : None
6.  * Return         : FLASH identification
7.  *****/
8.  u32 SPI_FLASH_ReadDeviceID(void)
9.  {
10.     u32 Temp = 0;
11.     /* Select the FLASH: Chip Select low */
12.     SPI_FLASH_CS_LOW();
13.     /* Send "RDID " instruction */
14.     SPI_FLASH_SendByte(W25X_DeviceID);
15.     SPI_FLASH_SendByte(Dummy_Byte);
16.     SPI_FLASH_SendByte(Dummy_Byte);
17.     SPI_FLASH_SendByte(Dummy_Byte);
18.     /* Read a byte from the FLASH */
19.     Temp = SPI_FLASH_SendByte(Dummy_Byte);
20.     /* Deselect the FLASH: Chip Select high */
21.     SPI_FLASH_CS_HIGH();
22.     return Temp;
23. }
```

接下来的是这个读 FLASH 器件 ID 的函数。实质是通过 SPI 接口向 FLASH 输入命令。以下是 FLASH 的各种命令：



11.2.2 Instruction Set ⁽¹⁾

INSTRUCTION NAME	BYTE 1 CODE	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6	N-BYTES
Write Enable	06h						
Write Disable	04h						
Read Status Register	05h	(S7-S0) ⁽¹⁾					(2)
Write Status Register	01h	S7-S0					
Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0)	(Next byte)	continuous
Fast Read	0Bh	A23-A16	A15-A8	A7-A0	dummy	(D7-D0)	(Next Byte) continuous
Fast Read Dual Output	3Bh	A23-A16	A15-A8	A7-A0	dummy	I/O = (D6,D4,D2,D0) O = (D7,D5,D3,D1)	(one byte per 4 clocks, continuous)
Page Program	02h	A23-A16	A15-A8	A7-A0	(D7-D0)	(Next byte)	Up to 256 bytes
Block Erase (64KB)	D8h	A23-A16	A15-A8	A7-A0			
Sector Erase (4KB)	20h	A23-A16	A15-A8	A7-A0			
Chip Erase	C7h						
Power-down	B9h						
Release Power-down / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0) ⁽⁴⁾		
Manufacturer/ Device ID ⁽³⁾	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)	
JEDEC ID	9Fh	(M7-M0) Manufacturer	(ID15-ID8) Memory Type	(ID7-ID0) Capacity			

在 datasheet 的后面有这些命令的详细解释。

下面以 `SPI_FLASH_ReadDeviceID()` 为例讲解指令表：

1. `SPI_FLASH_CS_LOW()`，这是一个自定义的宏拉低 CS 端口，以使能 FLASH 器件；
2. 利用 `SPI_FLASH_SendByte()`（后面再详细分析这个函数的具体实现）来向 FLASH 发送“W25X_DeviceID”（0xAB）的命令；
3. 根据指令表，发送完这个指令后，后面紧跟着三个字节的“dummy byte”意思是任意数据，野火把 `Dummy_Byte` 宏定义为“0xff”，实际上改成其它数据也无影响。
4. 在第 5 字节 FLASH 在 DIO 端口输出它的器件的 ID7-ID0 位，stm32 调用 `SPI_FLASH_SendByte()` 返回数据。
5. CS 端口拉高，结束通讯。

可以看到实验例程通过串口打印出来的跟表中的一样喔。



11.2.1 Manufacturer and Device Identification

MANUFACTURER ID	(M7-M0)	
Winbond Serial Flash	EFH	
Device ID	(ID7-ID0)	(ID15-ID0)
Instruction	ABh, 90h	9Fh
W25X16	14h	3015h
W25X32	15h	3016h
W25X64	16h	3017h

了解这个读器件流程后我们再来具体分析 `SPI_FLASH_SendByte()`。

```
1.  /*****
2.  * Function Name   : SPI_FLASH_SendByte
3.  * Description    : Sends a byte through the SPI interface and return t
   he byte
4.  *               received from the SPI bus.
5.  * Input          : byte : byte to send.
6.  * Output         : None
7.  * Return         : The value of the received byte.
8.  *****/
9.  u8 SPI_FLASH_SendByte(u8 byte)
10. {
11.     /* Loop while DR register in not empty */
12.     while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
13.
14.     /* Send byte through the SPI1 peripheral */
15.     SPI_I2S_SendData(SPI1, byte);
16.
17.     /* Wait to receive a byte */
18.     while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
19.
20.     /* Return the byte read from the SPI bus */
21.     return SPI_I2S_ReceiveData(SPI1);
22. }
```

1. 调用库函数 `SPI_I2S_GetFlagStatus()` 等待发送数据寄存器清空。
2. 发送数据寄存器准备好后，调用库函数 `SPI_I2S_GetFlagStatus()` 发送数据。
3. 调用库函数 `SPI_I2S_GetFlagStatus()` 等待接收数据寄存器非空。
4. 接收寄存器非空，调用 `SPI_I2S_ReceiveData()` 返回 DIO 端口接收回来的数据。

这是最底层的发送数据和接收数据的函数，只有几句代码，很好理解。



回到 main 函数中，其它的一些命令函数，如：`SPI_FLASH_ReadID()`、`SPI_FLASH_SectorErase()`和在 `SPI_FLASH_BufferWrite()` 中调用的 `SPI_FLASH_PageWrite()`。

它们的具体实现都跟读器件 ID 的类似，参考指令表了解一下流程就可以理解。

其中指令表中的 `A0~A23` 指地址；`M0~M7` 为器件的 制造商 ID (MANUFACTURER ID)；`D0~D7` 为数据。

根据 FLASH 的存储原理，在写入数据前，要先对存储区域进行擦除，所以执行 `SPI_FLASH_SectorErase()` 函数对要写入的扇区进行擦除，预写。

最后是 `SPI_FLASH_BufferWrite()` 和 `SPI_FLASH_BufferRead()` 函数，我们可以分别调用它们来把缓冲区数据写入 FLASH 和从 FLASH 读出数据到缓冲区。具体实现跟 I2C-EEPROM 的很类似，处理好地址后就执行命令进行读或写就行了。提一下这跟 EEPROM 的区别，这个

截图：

11.2.10 Page Program (02h)

The Page Program instruction allows up to 256 bytes of data to be programmed at previously erased to all 1s (FFh) memory locations. A Write Enable instruction must be executed before the device will accept the Page Program Instruction (Status Register bit WEL must equal 1). The instruction is initiated by driving the /CS pin low then shifting the instruction code "02h" followed by a 24-bit address (A23-A0) and at least one data byte, into the DIO pin. The /CS pin must be held low for the entire length of the instruction while data is being sent to the device. The Page Program instruction sequence is shown in figure 11.

此 FLASH 的页最大字节数为 256 字节，同样地，超过页最大字节继续写入数据的话，数据会从该页的起始地址覆盖写入。

对于读数据，发出一个命令后，可以无限制地一直把整个 FLASH 的数据都读取完，不需要读取整个 FLASH 的则以 CS 拉高为命令的结束的标置。

截图：



11.2.7 Read Data (03h)

The Read Data instruction allows one more data bytes to be sequentially read from the memory. The instruction is initiated by driving the /CS pin low and then shifting the instruction code "03h" followed by a 24-bit address (A23-A0) into the DIO pin. The code and address bits are latched on the rising edge of the CLK pin. After the address is received, the data byte of the addressed memory location will be shifted out on the DO pin at the falling edge of CLK with most significant bit (MSB) first. The address is automatically incremented to the next higher address after each byte of data is shifted out allowing for a continuous stream of data. **This means that the entire memory can be accessed with a single instruction as long as the clock continues.** The instruction is completed by driving /CS high. The Read Data instruction sequence is shown in figure 8. If a Read Data instruction is issued while an Erase, Program or Write cycle is in process (BUSY=1) the instruction is ignored and will not have any effects on the current cycle. The Read Data instruction allows clock rates from D.C. to a maximum of fr (see AC Electrical Characteristics).

最后，总结一下在 stm32 如何建立与 SPI-FLASH 的通讯。

- 4、 配置 I/O 端口,使能 GPIO;
- 5、 根据将要进行通讯器件的 SPI 模式, 配置 stm32 的 SPI, 使能 SPI 时钟;
- 6、 配置好后, 可以发送各种 FLASH 命令。

注意：在写操作前要先进行存储扇区的擦除操作，擦除操作前要先发出“写使能”命令。

6.4 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：



7、PWM（软件仿真）

7.1 实验描述及工程文件清单

实验描述	<p>通用定时器 TIM3 产生 4 路不同占空比的 PWM 波。</p> <p>$TIM3\ Channel1\ duty\ cycle = (TIM3_CCR1 / TIM3_ARR) * 100 = 50\%$</p> <p>$TIM3\ Channel2\ duty\ cycle = (TIM3_CCR2 / TIM3_ARR) * 100 = 37.5\%$</p> <p>$TIM3\ Channel3\ duty\ cycle = (TIM3_CCR3 / TIM3_ARR) * 100 = 25\%$</p> <p>$TIM3\ Channel4\ duty\ cycle = (TIM3_CCR4 / TIM3_ARR) * 100 = 12.5\%$</p>
硬件连接	<p>PA.06: (TIM3_CH1)</p> <p>PA.07: (TIM3_CH2)</p> <p>PB.00: (TIM3_CH3)</p> <p>PB.01: (TIM3_CH4)</p>
用到的库文件	<p>startup/start_stm32f10x_hd.c</p> <p>CMSIS/core_cm3.c</p> <p>CMSIS/system_stm32f10x.c</p> <p>FWlib/stm32f10x_gpio.c</p> <p>FWlib/stm32f10x_rcc.c</p> <p>FWlib/stm32f10x_flash.c</p> <p>FWlib/stm32f10x_tim.c</p>
用户编写的文件	<p>USER/main.c</p> <p>USER/stm32f10x_it.c</p> <p>USER/pwm_output.c</p>



7.2 STM32 通用定时器简介

STM32 总共有 8 个定时器，TIM1 和 TIM8 是 16 位的高级定时器，TIM2、TIM3、TIM4、TIM5 是通用定时器。本实验中只是讲解通用定时器 TIM3，利用 TIM3 产生 4 路不同占空比的方波。

7.3 代码分析

首先我们需在工程中添加我们需要用到的库文件，有关库文件的配置参考前面的教程，这里不再详述。

接下来我们从 main 函数讲起：

```
1.  /*
2.   * 函数名: main
3.   * 描述   : 主函数
4.   * 输入   : 无
5.   * 输出   : 无
6.   */
7.  int main(void)
8.  {
9.      /* 配置系统时钟为 72M */
10.     SystemInit();
11.     /* TIM3 PWM 波输出初始化, 并使能 TIM3 PWM 输出 */
12.     TIM3_PWM_Init();
13.
14.     while (1)
15.     {}
16. }
```

进入 main 函数我们首先调用库函数 `SystemInit()`，将我们的系统时钟配置为 72MHZ。有关库函数 `SystemInit()` 的讲解请参考前面的教程。

函数用于初始化 TIM3 的 PWM 信号 I/O，配置 PWM 信号的模式，如周期、极性、占空比等。`TIM3_PWM_Init()` 由我们用户在 `pwm_output.c` 中实现：



```
1.  /*
2.  * 函数名: TIM3_Mode_Config
3.  * 描述   : TIM3 输出 PWM 信号初始化, 只要调用这个函数
4.  *          TIM3 的四个通道就会有 PWM 信号输出
5.  * 输入   : 无
6.  * 输出   : 无
7.  * 调用   : 外部调用
8.  */
9. void TIM3_PWM_Init(void)
10. {
11.     TIM3_GPIO_Config();
12.     TIM3_Mode_Config();
13. }
```

其中用来 `TIM3_GPIO_Config()`;配置 GPIO, 代码很简单, `TIM3_Mode_Config()`;用来配置 PWM 信号的模式, 详细代码如下, 主要做了如下工作:

1->设定 TIM 信号周期

2->设定 TIM 预分频值

3->设定 TIM 分频系数

4->设定 TIM 计数模式

5->根据 `TIM_TimeBaseInitStruct` 这个结构体里面的值初始化 TIM

6->设定 TIM 的 OC 模式

7->TIM 输出使能

8->设定电平跳变值

9->设定 PWM 信号的极性

10->使能 TIM 信号通道

11->使能 TIM 重载寄存器 CCRX

12->使能 TIM 重载寄存器 ARR

13->使能 TIM 计数器

```
1.  /*
2.  * 函数名: TIM3_Mode_Config
3.  * 描述   : 配置 TIM3 输出的 PWM 信号的模式, 如周期、极性、占空比
4.  * 输入   : 无
```





```
5.  * 输出 : 无
6.  * 调用 : 内部调用
7.  */
8.  static void TIM3_Mode_Config(void)
9.  {
10.     TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
11.     TIM_OCInitTypeDef TIM_OCInitStructure;
12.
13.     /* PWM 信号电平跳变值 */
14.     u16 CCR1_Val = 500;
15.     u16 CCR2_Val = 375;
16.     u16 CCR3_Val = 250;
17.     u16 CCR4_Val = 125;
18.
19.     /* -----
20.     TIM3 Configuration: generate 4 PWM signals with 4 different duty cycles:
21.     TIM3CLK = 36 MHz, Prescaler = 0x0, TIM3 counter clock = 36 MHz
22.     TIM3 ARR Register = 999 => TIM3 Frequency = TIM3 counter clock/(ARR + 1)
23.     TIM3 Frequency = 36 KHz.
24.     TIM3 Channel1 duty cycle = (TIM3_CCR1/ TIM3_ARR)* 100 = 50%
25.     TIM3 Channel2 duty cycle = (TIM3_CCR2/ TIM3_ARR)* 100 = 37.5%
26.     TIM3 Channel3 duty cycle = (TIM3_CCR3/ TIM3_ARR)* 100 = 25%
27.     TIM3 Channel4 duty cycle = (TIM3_CCR4/ TIM3_ARR)* 100 = 12.5%
28.     ----- */
29.
30.     /* Time base configuration */
31.     //当定时器从 0 计数到 999, 即为 1000 次, 为一个定时周期
32.     TIM_TimeBaseStructure.TIM_Period = 999;
33.
34.     //设置预分频: 不预分频, 即为 36MHz
35.     TIM_TimeBaseStructure.TIM_Prescaler = 0;
36.
37.     TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分频系数: 不分频
38.     TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式
39.
40.     TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
41.
42.     /* PWM1 Mode configuration: Channel1 */
43.     TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; //配置为 PWM 模式 1
44.     TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
45.
46.     //设置跳变值, 当计数器计数到这个值时, 电平发生跳变
47.     TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
48.
49.     //当定时器计数值小于 CCR1_Val 时为高电平
50.     TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
51.
52.     TIM_OC1Init(TIM3, &TIM_OCInitStructure); //使能通道 1
53.     TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);
54.
55.     /* PWM1 Mode configuration: Channel2 */
56.     TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
57.
58.     //设置通道 2 的电平跳变值, 输出另外一个占空比的 PWM
59.     TIM_OCInitStructure.TIM_Pulse = CCR2_Val;
60.
61.     TIM_OC2Init(TIM3, &TIM_OCInitStructure); //使能通道 2
62.     TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);
63.
64.     /* PWM1 Mode configuration: Channel3 */
65.     TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
66.
67.     //设置通道 3 的电平跳变值, 输出另外一个占空比的 PWM
68.     TIM_OCInitStructure.TIM_Pulse = CCR3_Val;
69.
70.     TIM_OC3Init(TIM3, &TIM_OCInitStructure); //使能通道 3
71.     TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);
72.
73.     /* PWM1 Mode configuration: Channel4 */
74.     TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
75.
76.     //设置通道 4 的电平跳变值, 输出另外一个占空比的 PWM
77.     TIM_OCInitStructure.TIM_Pulse = CCR4_Val;
78.     TIM_OC4Init(TIM3, &TIM_OCInitStructure); //使能通道 4
```



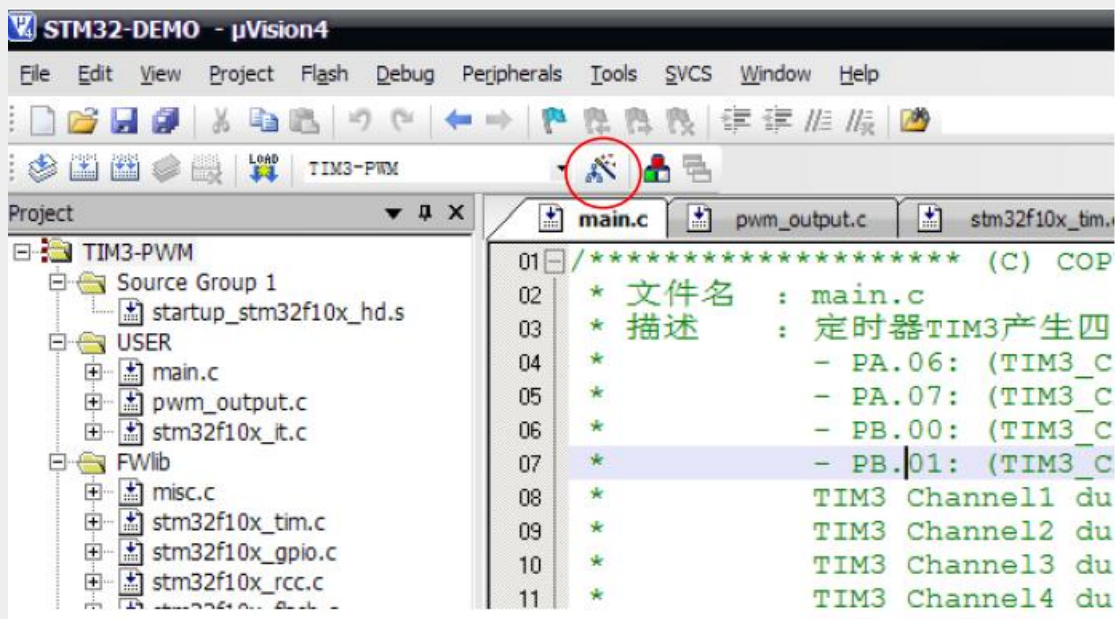
```
79. TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Enable);
80.
81. TIM_ARRPreloadConfig(TIM3, ENABLE);
82.
83. /* TIM3 enable counter */
84. TIM_Cmd(TIM3, ENABLE); //使能定时器 3
85. }
```

现在，TIM3 的通道 1(PA.06)、2(PA.07)、3(PB.00)、4(PB.01)就会输出不同占空比的 PWM 信号了。PWM 信号可以通过示波器看到。考虑到并不是每个用户手头上都有示波器，我们在这里采用软件仿真的方式来验证我们的程序。

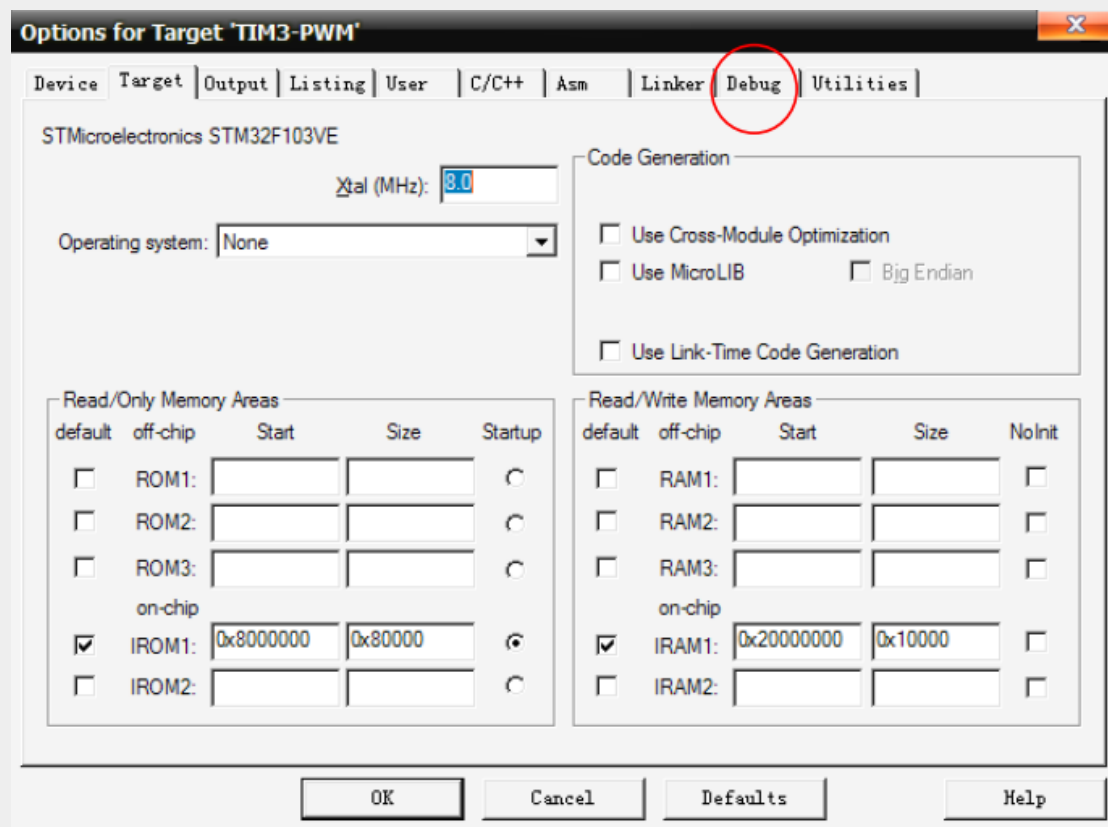
7.4 软件仿真

以前我们都是通过 JLINK 直接将我们的代码烧到开发板的 flash 中去调试，现在要换成软件仿真，得首先设置下我们的开发环境，按照如下步骤所示：

1、点击 Target Options...选项。

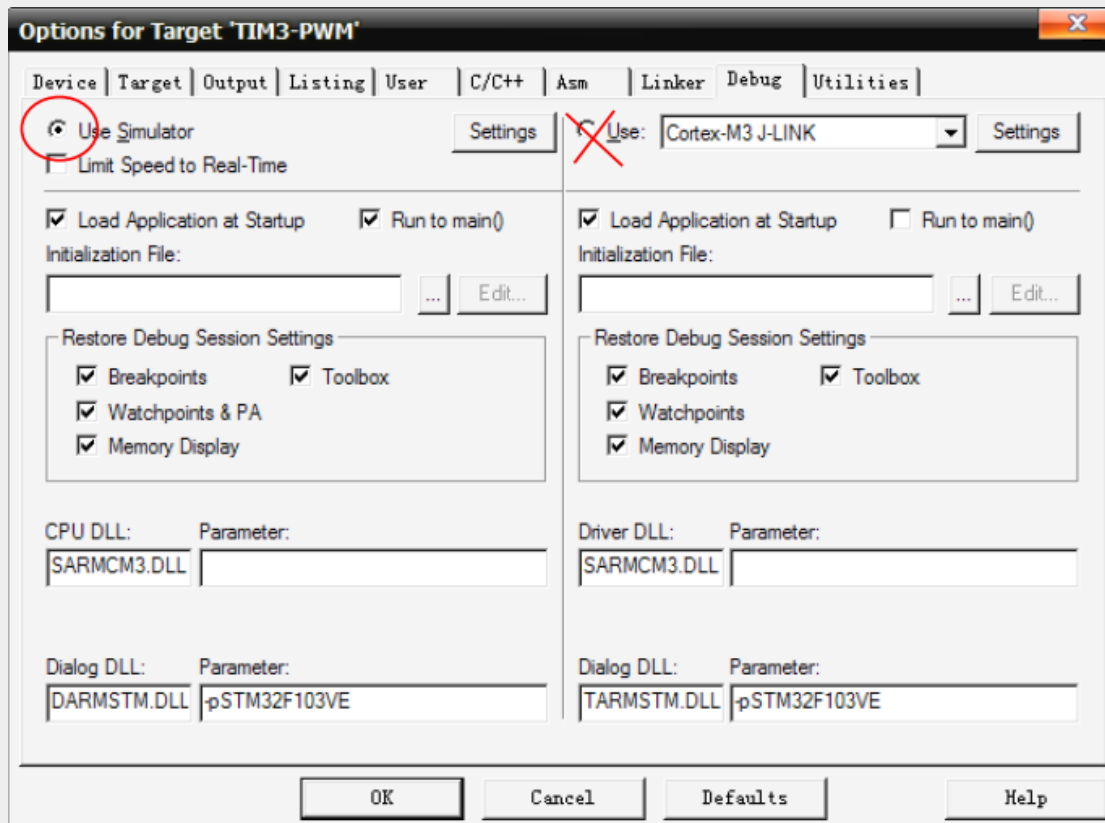


2、选中 Debug 选项卡。



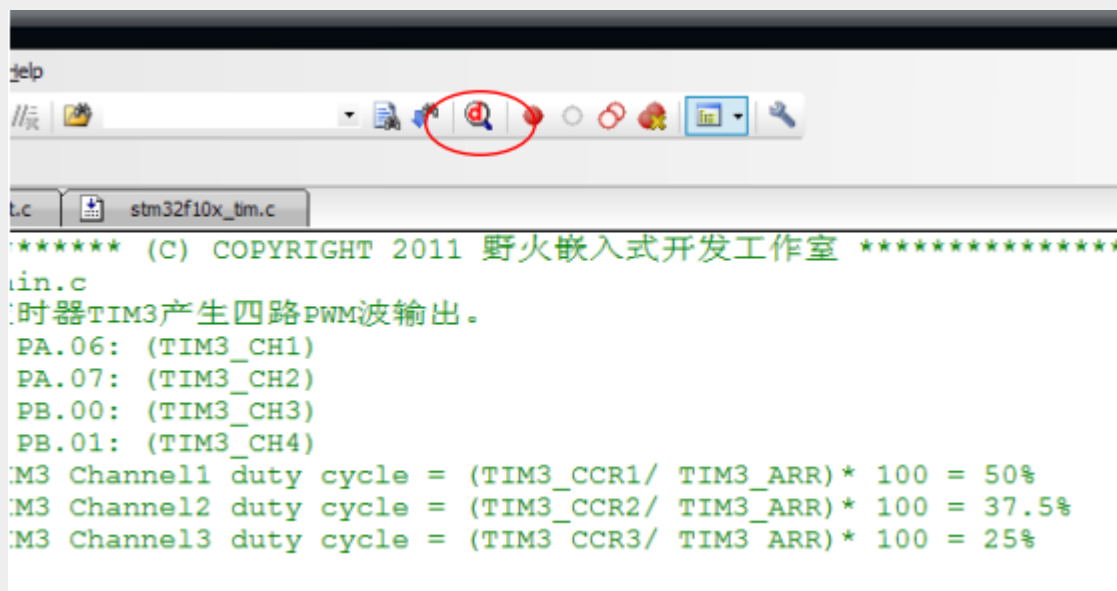
3、选中 Use Simulator 选项，然后点击 OK 即可。





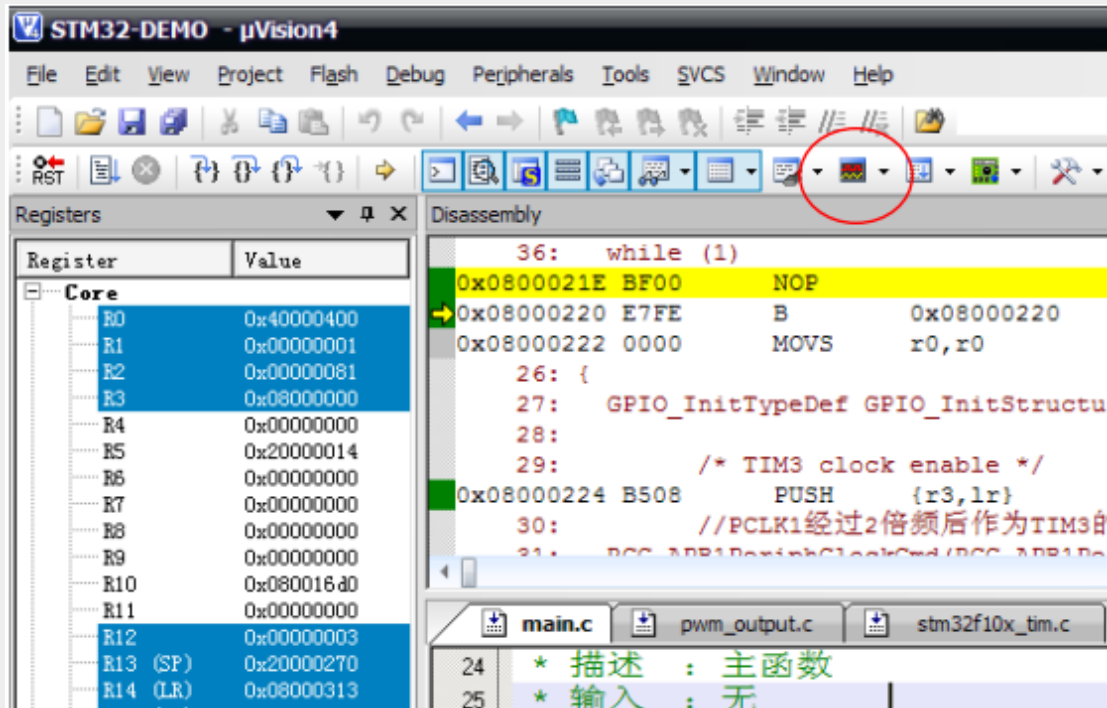
下面我们开始进行软件仿真，按照如下步骤进行：

1. 1-> 点击 Start/Stop Debug Session 选项。

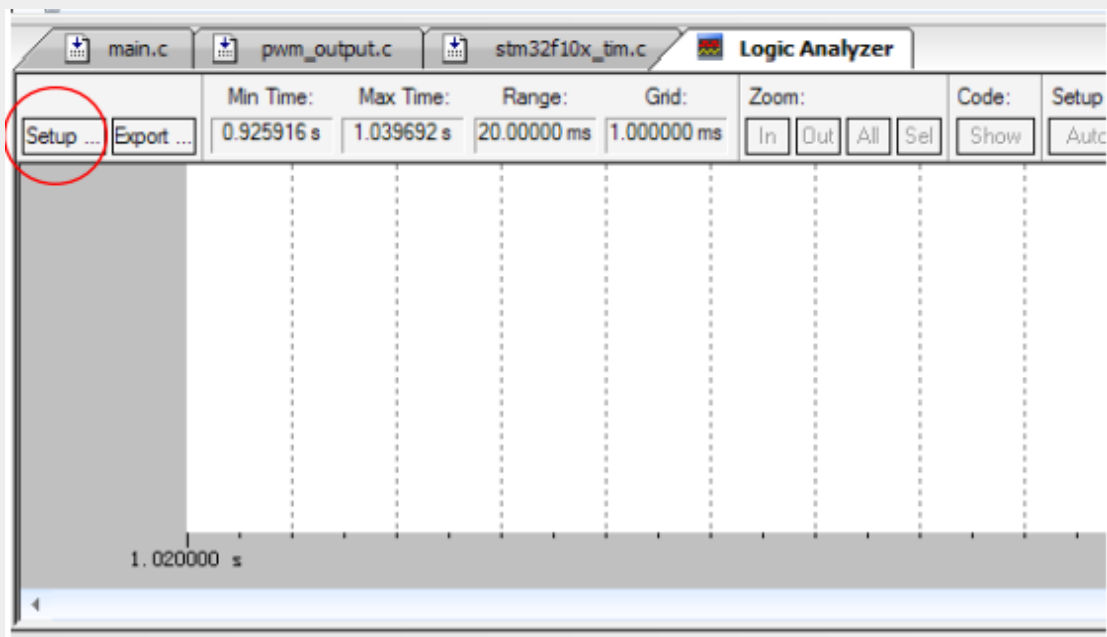


2. 2-> 点击 Analysis Windows 选项。



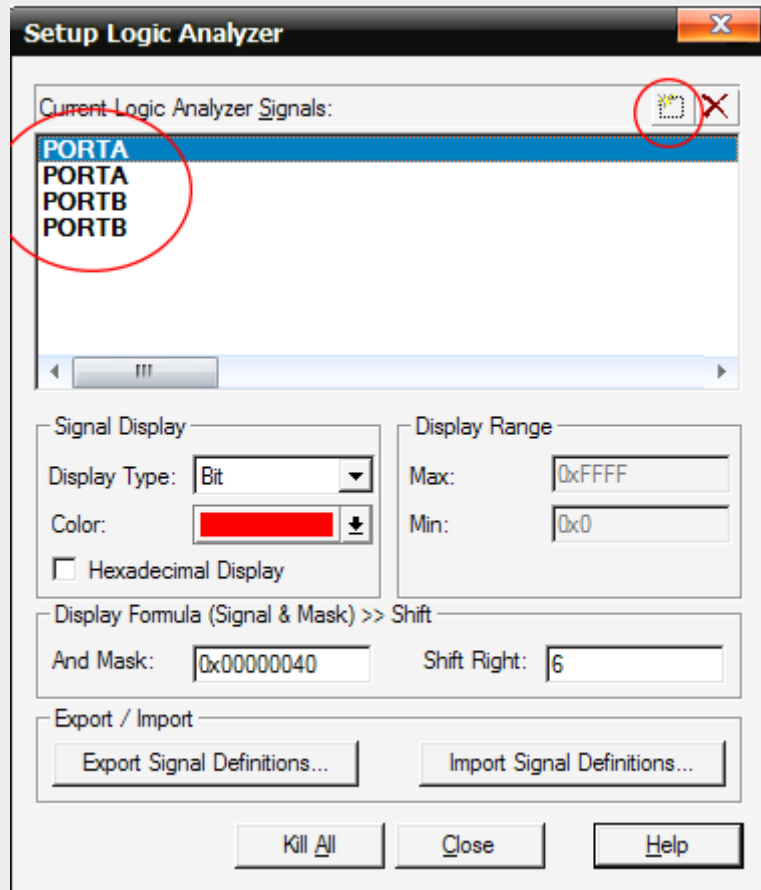


3. 3->点击 Setup... 选项卡。

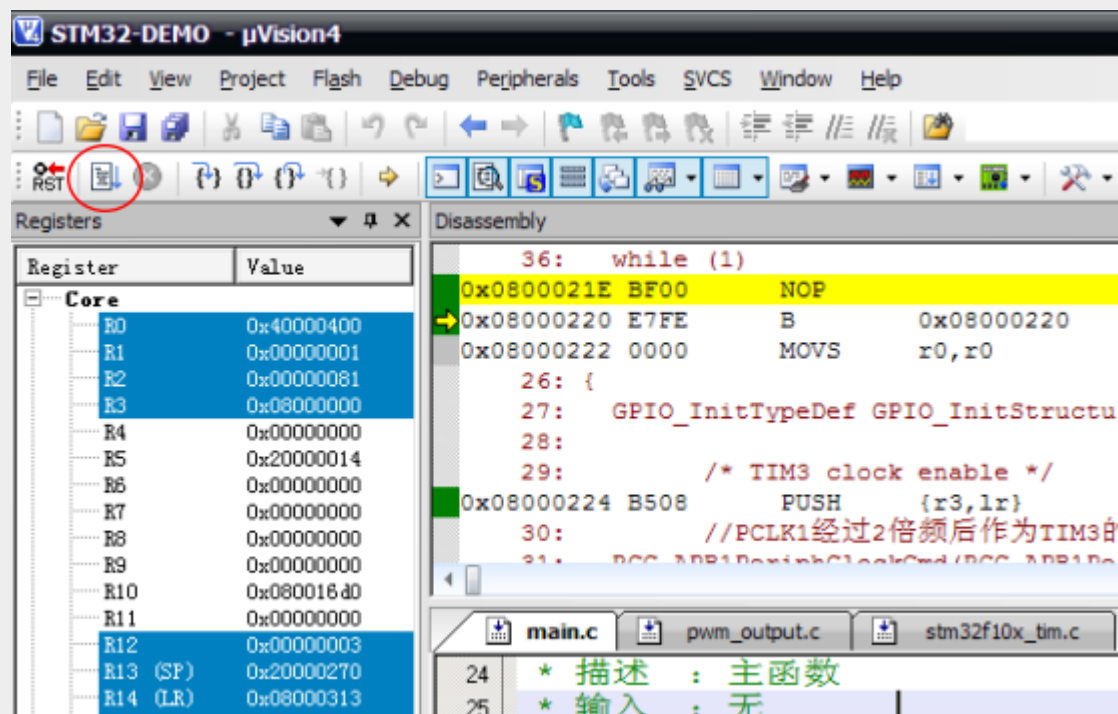


4. 4->点击 NEW(Insert), 在下面的文本框中输入 TMI3 的 PWM 通道, 这里分别是: PORTA.6、PORTA.7、PORTB.0、PORTB.1。然后点击 Close。



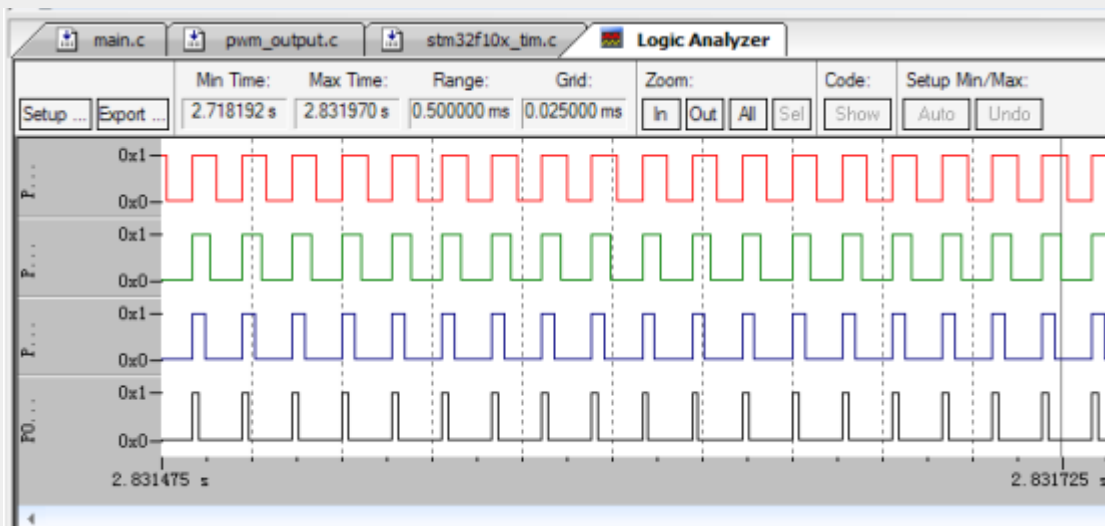


5. 5->点击运行。

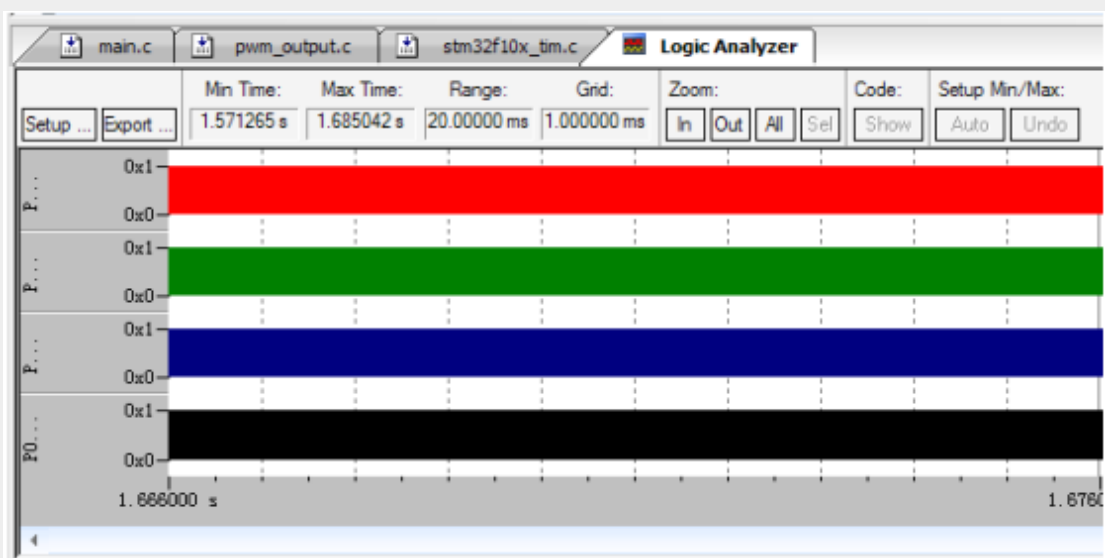


6. 6->这时候正常的话则是出现下面的 PWM 信号。



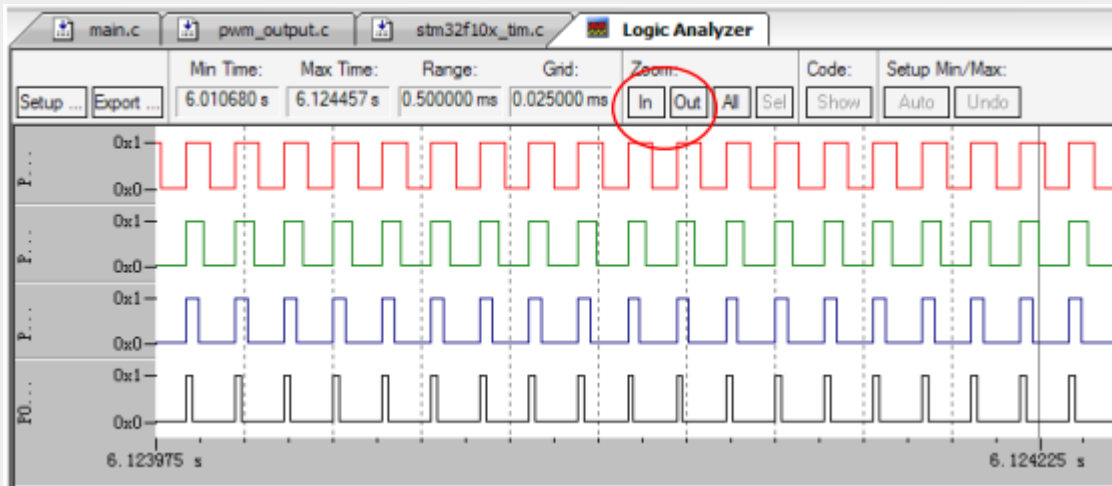


不正常的话则出现下面的情况，一团糟，根本看不到 PWM 信号。这时我们不免会有些抓狂呀，哈哈，但请大家放心，看看接下来我们是怎么解决的吧。

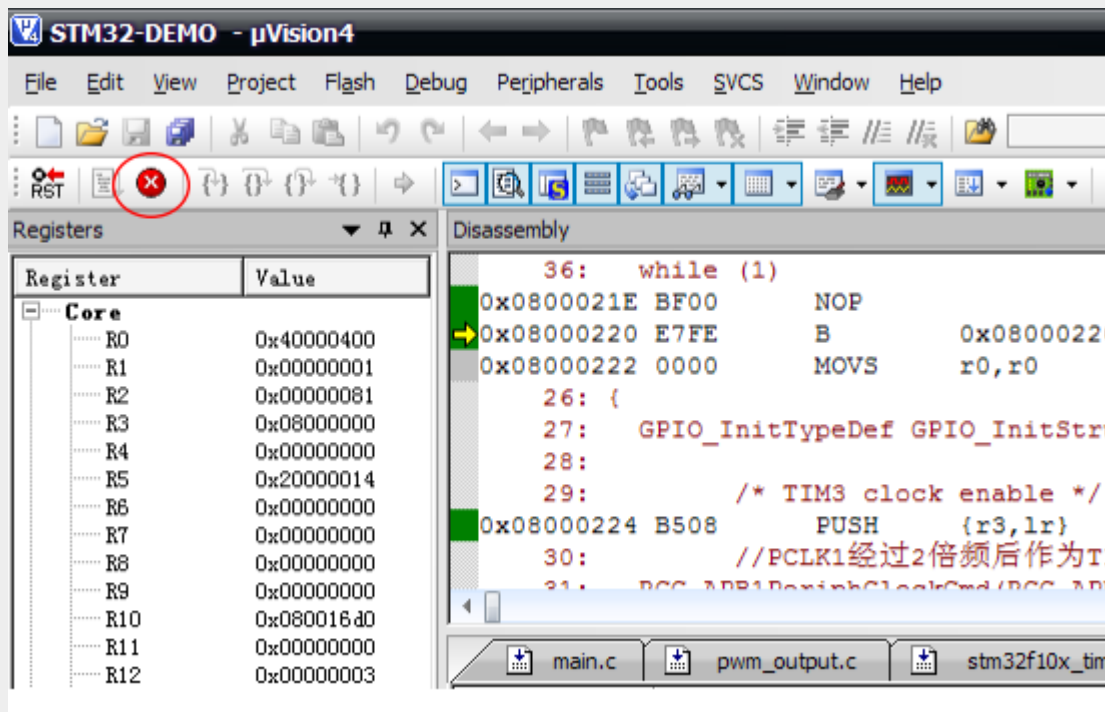


7. 7->其实出现上面的情况是因为我们显示 PWM 信号时没有放大的缘故，我们可以点击 **In** 这个按钮来将 PWM 信号显示的大点，如下所示，一切搞定。

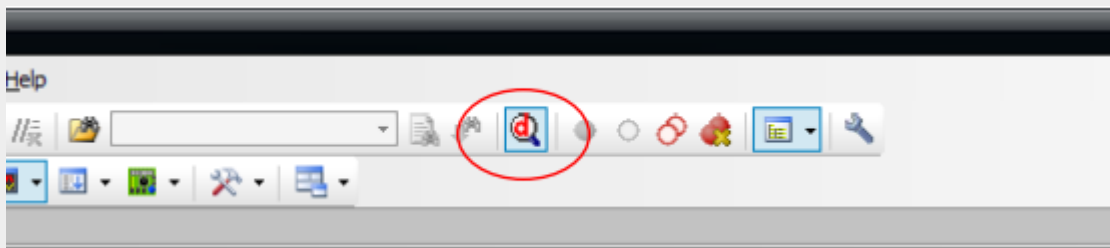




8. 8->我们可以点击停止按钮，让 PWM 信号静止显示。



9. 9->仿真完毕之后，点击 Start/Stop Debug Session 选项就可以回到正常的代码编辑模式。



10. 10->要是您有示波器的话，看到的效果则更真实，更给力。



8、CAN（Looback）

8.1 实验描述及工程文件清单

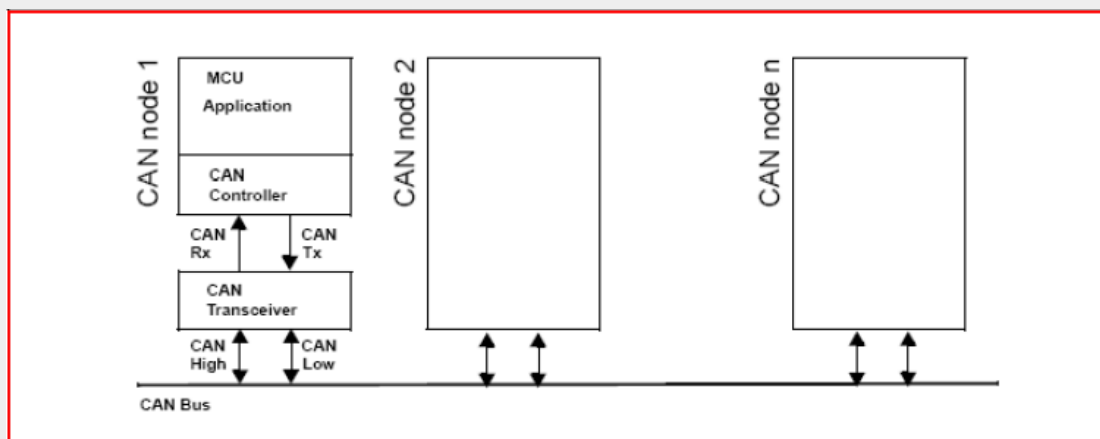
实验描述	can 测试实验(中断模式和回环)，并将测试信息通过 USART1 在超级终端中打印出来。
硬件连接	PB8-CAN-RX PB9-CAN-TX
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_can.c FWlib/misc.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/led.c USER/usart.c USER/can.c

8.2 CAN 简介

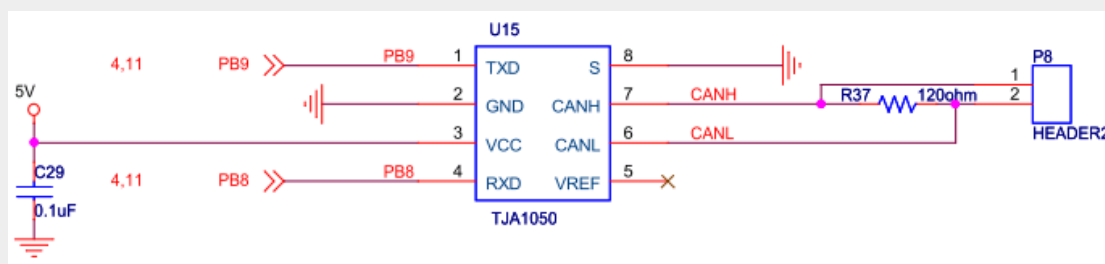
CAN 是控制器局域网络(Controller Area Network, CAN)的简称，是由研发和生产汽车电子产品著称的德国 BOSCH 公司开发的，并最终成为国际标准（ISO11878）。是国际上应用最广泛的现场总线之一。在北美和西欧，CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。

近年来，其所具有的高可靠性和良好的错误检测能力受到重视，被广泛应用于汽车计算机控制系统和环境温度恶劣、电磁辐射强和振动大的工业环境。

野火 STM32 开发板的 CPU(cpu 型号为: STM32F103VET6)自带了一个 CAN 控制器。具体 I/O 定义为 PB8-CAN-RX、PB9-CAN-TX。板载的 CAN 外接了一个 TJA1050 CAN 收发器，外部的 CAN 设备可以作为一个设备节点挂接到板载的 CAN 收发器中，实现 CAN 通信，多个 CAN 节点通信图如下：



野火 STM32 开发板中 CAN 硬件原理图如下：



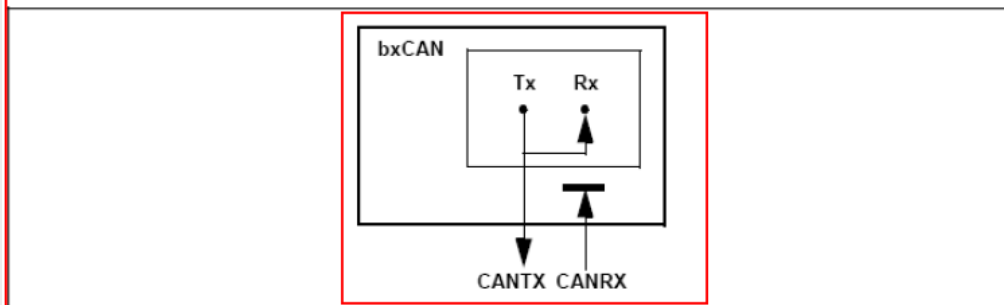
在本实验中并没有用得到双 CAN 通信，只是用了 CAN 的回环测试，这样我们就不需要挂接外部的 CAN 节点。有关双 CAN 通信的实验，大家可参考野火 STM32 光盘自带的例程《16-野火 M3-CAN (Mutual)》当我们用 CAN 的回环测试时，硬件会在内部将 TX 和 RX 连接起来，实现内部的收和发，从而达到测试的目的。



环回模式

通过对CAN_BTR寄存器的LBKM位置'1'，来选择环回模式。在环回模式下，bxCAN把发送的报文当作接收的报文并保存(如果可以通过接收过滤)在接收邮箱里。

图194 bxCAN工作在环回模式



环回模式可用于自测试。为了避免外部的影响，在环回模式下CAN内核忽略确认错误(在数据/远程帧的确认位时刻，不检测是否有显性位)。在环回模式下，bxCAN在内部把Tx输出回馈到Rx输入上，而完全忽略CANRX引脚的实际状态。发送的报文可以在CANTX引脚上检测到。

8.3 代码分析

首先在工程中添加需要用到的头文件：

```
FWlib/stm32f10x_gpio.c
FWlib/stm32f10x_rcc.c
FWlib/stm32f10x_usart.c
FWlib/stm32f10x_can.c
FWlib/misc.c
```

还要将c文件对应的头文件添加进来，在库头文件 `stm32f10x_conf.h` 中实现：

```
1.  /* Includes -----*/
2.  /* Uncomment the line below to enable peripheral header file inclusion */
3.  /* #include "stm32f10x_adc.h" */
4.  /* #include "stm32f10x_bkp.h" */
5.  #include "stm32f10x_can.h"
6.  /* #include "stm32f10x_crc.h" */
7.  /* #include "stm32f10x_dac.h" */
8.  /* #include "stm32f10x_dbgmcu.h" */
9.  /* #include "stm32f10x_dma.h" */
10. /* #include "stm32f10x_exti.h" */
11. /* #include "stm32f10x_flash.h" */
12. /* #include "stm32f10x_fsmc.h" */
13. #include "stm32f10x_gpio.h"
14. /* #include "stm32f10x_i2c.h" */
15. /* #include "stm32f10x_iwdg.h" */
16. /* #include "stm32f10x_pwr.h" */
17. #include "stm32f10x_rcc.h"
18. /* #include "stm32f10x_rtc.h" */
19. /* #include "stm32f10x_sdio.h" */
20. /* #include "stm32f10x_spi.h" */
21. /* #include "stm32f10x_tim.h" */
22. #include "stm32f10x_usart.h"
23. /* #include "stm32f10x_wdg.h" */
24. #include "misc.h" /* High level functions for NVIC and SysTick (add-on to CMSIS functions) */
```

OK，库环境已经配置好，接下来我们就开始分析 `main` 函数吧：



```

1.  /**
2.   * @brief Main program.
3.   * @param None
4.   * @retval : None
5.   */
6.  int main(void)
7.  {
8.      /* config the sysclock to 72M */
9.      SystemInit();
10.
11.     /* USART1 config */
12.     USART1_Config();
13.     /* LED config */
14.     LED_GPIO_Config();
15.     printf( "\r\n 这个一个 CAN（回环模式和中断模式）测试程序..... \r\n" );
16.     USER_CAN_Init();
17.     printf( "\r\n CAN 回环测试初始化成功..... \r\n" );
18.     USER_CAN_Test();
19.     printf( "\r\n CAN 回环测试成功..... \r\n" );
20.     while (1)
21.     {
22.     }
23. }

```

首先我们调用库 `SystemInit()` 函数将我们的系统时钟设置为 **72MHZ**，紧接着初始化串口 `USART1_Config()` 和 **LED** `LED_GPIO_Config()`，因为在本实验中我们需要用到串口来打印一些调试信息，用 **LED** 来显示程序的运行状态。有关这三个函数的详细讲解，请参考前面章节的教程，这里不再详述，其中 `USART1_Config()` 和 `LED_GPIO_Config()` 是由用户实现的应用函数，并非库函数。

`USER_CAN_Init()` 实现了 **CAN I/O** 端口的初始化和中断优先级的初始化（因为等下要用到 **CAN** 的中断模式）。在 `can.c` 中实现：

```

1.  /**
2.   * 函数名: CAN_Init
3.   * 描述   : CAN 初始化，包括端口初始化和中断优先级初始化
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8.  void USER_CAN_Init(void)
9.  {
10.     CAN_NVIC_Configuration();
11.     CAN_GPIO_Config();
12. }

```

`USER_CAN_Init()` 调用了两个内部函数 `CAN_NVIC_Configuration()` 和 `CAN_GPIO_Config()`，见名知义就可知道这两个函数的功能是什么啦，他们也均在 `can.c` 中实现：

```

1.  /**

```



```

2.  * 函数名: CAN_NVIC_Configuration
3.  * 描述   : CAN RX0 中断优先级配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8.  static void CAN_NVIC_Configuration(void)
9.  {
10.     NVIC_InitTypeDef NVIC_InitStructure;
11.
12.     /* Enable CAN1 RX0 interrupt IRQ channel */
13.     NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
14.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;      // 主优先级为 0
15.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;             // 次优先级为 0
16.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
17.     NVIC_Init(&NVIC_InitStructure);
18. }

```

在 `CAN_GPIO_Config(void)` 这个函数中我们要注意一点：**CAN** 的 **RX** 脚要设置为上拉输入，**TX** 脚要设置为复用推挽输出，其他就跟配置普通 **I/O** 口一样。

```

1.  /*
2.  * 函数名: CAN_GPIO_Config
3.  * 描述   : CAN GPIO 和时钟配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8.  static void CAN_GPIO_Config(void)
9.  {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOB, ENABLE);
12.
13.     /* CAN1 Periph clock enable */
14.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
15.
16.     /* Configure CAN pin: RX */                                     // PB8
17.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;

```



```
17.   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;           // 上拉输入
18.   GPIO_Init(GPIOB, &GPIO_InitStructure);
19.
20.   /* Configure CAN pin: TX */                                // PB9
21.   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
22.   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;           // 复用推挽输出
23.   GPIO_Init(GPIOB, &GPIO_InitStructure);
24.
25.   //#define GPIO_Remap_CAN      GPIO_Remap1_CAN1  本实验没有用到重映射 I/O
26.   GPIO_PinRemapConfig(GPIO_Remap1_CAN1, ENABLE);
27. }
```

当我们将 CAN 初始化好之后，接下来就是重头戏了。我们调用 `USER_CAN_Test()` 函数。这个函数里面包含了 CAN 的回环和中断两种测试，在 `can.c` 中实现：

```
1.  /*
2.  * 函数名: CAN_Test
3.  * 描述   : CAN 回环模式跟中断模式测试
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8.  void USER_CAN_Test(void)
9.  {
10.     /* CAN transmit at 100Kb/s and receive by polling in loopback mode */
11.     TestRx = CAN_Polling();
12.     if (TestRx == FAILED)
13.     {
14.         LED1( OFF );    // LED1 OFF
15.     }
16.     else
17.     {
18.         LED1( ON );     // LED1 ON;
19.     }
20.     /* CAN transmit at 500Kb/s and receive by interrupt in loopback mode */
21.     TestRx = CAN_Interrupt();
22.     if (TestRx == FAILED)
23.     {
24.         LED2( OFF );    // LED2 OFF;
25.     }
26.     else
27.     {
28.         LED2( ON );     // LED2 ON;
29.     }
30. }
```

在这个函数中通过板载的 LED 的状态来显示回环和中断模式测试是否成功。其中回环模式测试调用了 `CAN_Polling()` 这个函数，通信速率为 100Kb/s。中断模式测试调用了 `CAN_Interrupt()` 这个函数，通信速率为 500Kb/s。

下面我们来重点分析下 `CAN_Polling()` 和 `CAN_Interrupt()` 这两个函数，这两个函数也在 `can.c` 中实现：



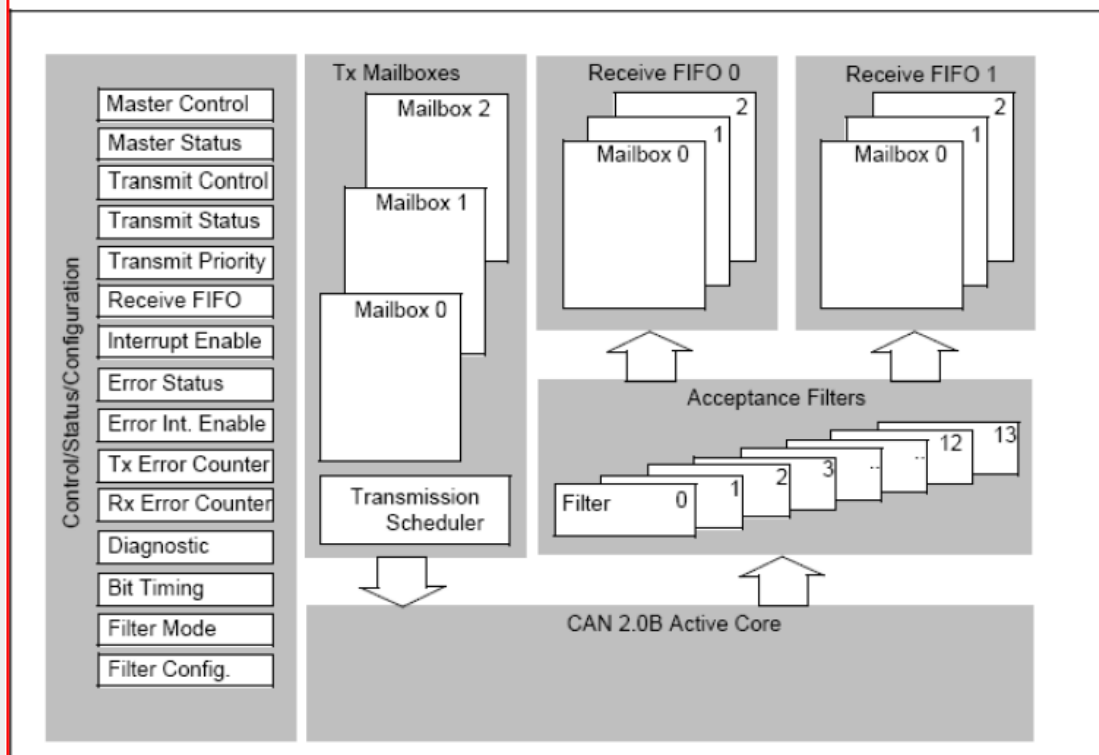
```
1.  /*
2.  * 函数名: CAN_Polling
3.  * 描述   : 配置 CAN 的工作模式为 回环模式
4.  * 输入   : 无
5.  * 输出   : -PASSED   成功
6.  *         -FAILED   失败
7.  * 调用   : 内部调用
8.  */
9. TestStatus CAN_Polling(void)
10. {
11.     CAN_InitTypeDef      CAN_InitStructure;
12.     CAN_FilterInitTypeDef CAN_FilterInitStructure;
13.     CanTxMsg TxMessage;
14.     CanRxMsg RxMessage;
15.     uint32_t i = 0;
16.     uint8_t TransmitMailbox = 0;
17.
18.     /* CAN register init */
19.     CAN_DeInit(CAN1);
20.     CAN_StructInit(&CAN_InitStructure);
21.
22.     /* CAN cell init */
23.     CAN_InitStructure.CAN_TTCM=DISABLE;
24.     CAN_InitStructure.CAN_ABOM=DISABLE;
25.     CAN_InitStructure.CAN_AWUM=DISABLE;
26.     CAN_InitStructure.CAN_NART=DISABLE;
27.     CAN_InitStructure.CAN_RFLM=DISABLE;
28.     CAN_InitStructure.CAN_TXFP=DISABLE;
29.     CAN_InitStructure.CAN_Mode=CAN_Mode_LoopBack; // 回环模式
30.     CAN_InitStructure.CAN_SJW=CAN_SJW_1tq;
31.     CAN_InitStructure.CAN_BS1=CAN_BS1_8tq;
32.     CAN_InitStructure.CAN_BS2=CAN_BS2_7tq;
33.     CAN_InitStructure.CAN_Prescaler=5;           // 分频系数为 5
34.     CAN_Init(CAN1, &CAN_InitStructure);         // 初始化 CAN
35.
36.     /* CAN filter init */
37.     CAN_FilterInitStructure.CAN_FilterNumber=0;
38.     CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
39.     CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;
40.     CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000;
41.     CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;
42.     CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000;
43.     CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;
44.     CAN_FilterInitStructure.CAN_FilterFIFOAssignment=0;
45.     CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;
46.     CAN_FilterInit(&CAN_FilterInitStructure);
47.
48.     /* transmit */
49.     TxMessage.StdId=0x11;           // 设定标准标识符 (11 位, 扩展的为 29 位)
50.     TxMessage.RTR=CAN_RTR_DATA;    // 传输消息的帧类型为数据帧 (还有远程帧)
51.     TxMessage.IDE=CAN_ID_STD;      // 消息标志符实验标准标识符
52.     TxMessage.DLC=2;               // 发送两帧, 一帧 8 位
53.     TxMessage.Data[0]=0xCA;        // 第一帧数据
54.     TxMessage.Data[1]=0xFE;        // 第二帧数据
55.
56.     TransmitMailbox=CAN_Transmit(CAN1, &TxMessage);
57.     i = 0;
58.     // 用于检查消息传输是否正常
59.     while((CAN_TransmitStatus(CAN1, TransmitMailbox) != CANTXOK) && (i != 0xFF))
60.     {
61.         i++;
62.     }
63.
64.     i = 0;
65.     // 检查返回的挂号的信息数目
66.     while((CAN_MessagePending(CAN1, CAN_FIFO0) < 1) && (i != 0xFF))
67.     {
68.         i++;
69.     }
70.     /* receive */
71.     RxMessage.StdId=0x00;
72.     RxMessage.IDE=CAN_ID_STD;
73.     RxMessage.DLC=0;
74.     RxMessage.Data[0]=0x00;
75.     RxMessage.Data[1]=0x00;
76.     CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
77.
78.     if (RxMessage.StdId!=0x11)
79.     {
80.         return FAILED;
81.     }
82.     if (RxMessage.IDE!=CAN_ID_STD)
83.     {
84.         return FAILED;
85.     }
```



```
86.     if (RxMessage.DLC!=2)
87.     {
88.         return FAILED;
89.     }
90.     if ((RxMessage.Data[0]<<8 | RxMessage.Data[1]) != 0xCAFE)
91.     {
92.         return FAILED;
93.     }
94.     //printf("receive data:0x%X,0x%X",RxMessage.Data[0], RxMessage.Data[1]);
95.     return PASSED; /* Test Passed */
96. }
```

在分析这两个函数之前，我们先来看下下面这幅图，截图来自《STM32 参考手册中文》。

图191 CAN功能框图



上图中左边的 Control / Sdatus / Configuration 的具体细节我们先不管它，先放着。这个图中有三个专业名词，我们先解释下：

1-> Tx Mailboxes(发送邮箱)

STM32 的 CAN 中共有 3 个发送邮箱供软件来发送报文。发送调度器根据优先级决定哪个邮箱的报文先被发送。

2->Acceptance Filters(接收过滤器)

STM32 的 CAN 中共有 14 个位宽可变/可配置的标识符过滤器组，软件通过对它们编程，从而在引脚收到的报文中选择它需要的报文，而把其它报文丢弃掉。



3-> Receive FIFO(接收 FIFO)

STM32 的 CAN 中共有 2 个接收 FIFO，每个 FIFO 都可以存放 3 个完整的报文。它们完全由硬件来管理。在 CAN 回环测试实验中我们把要发送的数据放在 **Tx Mailboxes** 中，数据集经过 **Acceptance Filters** 发送到 **Receive FIFO**，再将接收到的数据存到相应的缓冲区中，通过比较接收和发送的数据是否相同来验证我们的实验是否正确。实验过程看起来很简单：发送 -> 过滤 -> 接收 -> 比较。但只是宏观上把握罢了，这期间还需要做许多的工作。现在我们再回到 `CAN_Polling(void)` 这个函数中，看看这一过程我们到底做了些什么：

1-> CAN 寄存器初始化，全部初始化为默认值。

2->CAN cell 初始化。具体可参考源码，代码里面有详细注释。

3->CAN 过滤器初始化。

4->发送数据。这其中包括了设置设定标准标识符、传输消息的帧类型、要发送多少帧、邮箱中的帧数据是什么等。具体可参考源码，代码里面有详细注释。

5->接收数据。要初始化接收邮箱，用于接收数据，比较报文中设定的标准标识符是否相等、最后比较发送的数据和接收的数据是否相等。具体可参考源码，代码里面有详细注释。

6->最后返回 **TestStatus** 信息，**PASSED** 表示成功，**FAILED** 表示失败。

在阅读这部分代码时，需要参考《[STM32 参考手册中文](#)》第 21 章<控制器局域网 [bxcn](#)>，这样对理解代码有很大的帮助。刚开始的时候我们也没太大必要完全去搞懂每一句代码是什么意思，先在宏观上把握他，看下这段代码是否真的工作了，一步一步去修改它，测试它，看看实验会发生什么效果。这样学习的时间长了，理解就自然深刻了，以前那些你觉得很纳闷的问题，那些瓶颈问题都会瞬间解决，一下子豁然开朗。其实这也是我的一种学习方法。量变了，质变还会远吗？

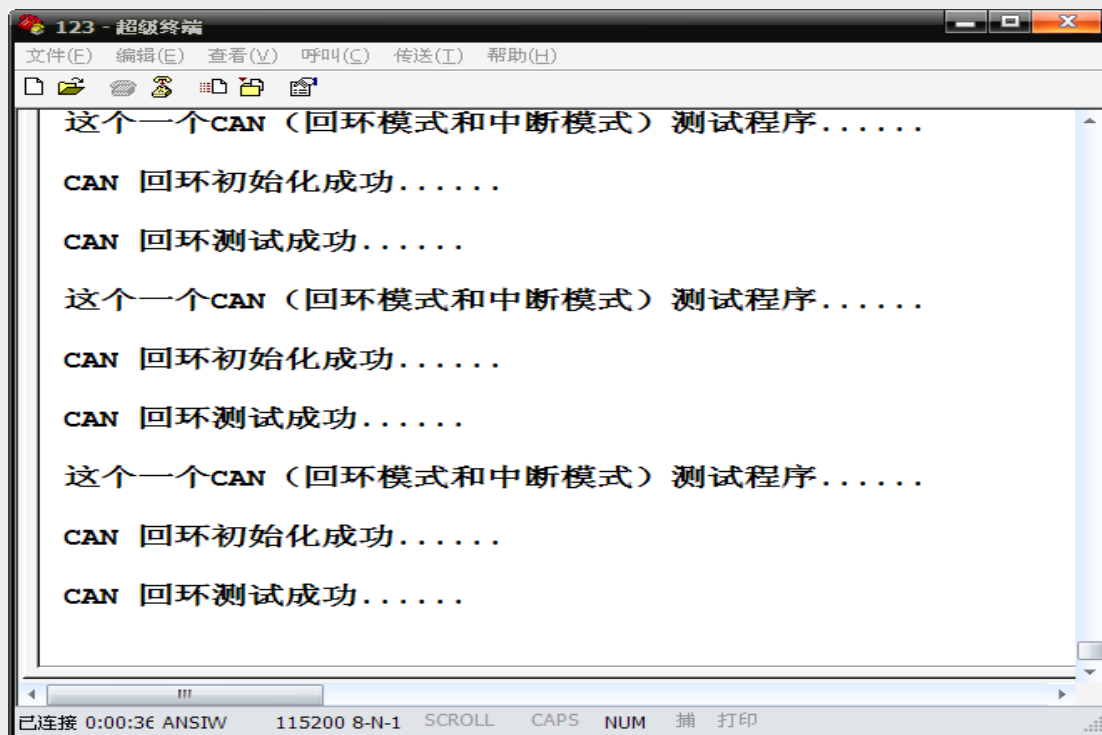
CAN 的中断模式跟回环模式是类似的，具体的大家自行阅读代码吧。唯一不同的是我们需要在 `stm32f10x_it.c` 中添加 CAN 的中断服务程序：



```
1.  /*
2.  * 函数名: USB_LP_CAN1_RX0_IRQHandler
3.  * 描述   : USB 中断和 CAN 接收中断服务程序, USB 跟 CAN 公用 I/O, 这里只用到 CAN 的中断。
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 无
7.  */
8.  void USB_LP_CAN1_RX0_IRQHandler(void)
9.  {
10.     CanRxMsg RxMessage;
11.
12.     RxMessage.StdId=0x00;
13.     RxMessage.ExtId=0x00;
14.     RxMessage.IDE=0;
15.     RxMessage.DLC=0;
16.     RxMessage.FMI=0;
17.     RxMessage.Data[0]=0x00;
18.     RxMessage.Data[1]=0x00;
19.
20.     CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
21.
22.     if((RxMessage.ExtId==0x1234) && (RxMessage.IDE==CAN_ID_EXT)
23.        && (RxMessage.DLC==2) && ((RxMessage.Data[1]|RxMessage.Data[0]<<8)==0xDECA))
24.     {
25.         ret = 1;
26.     }
27.     else
28.     {
29.         ret = 0;
30.     }
31. }
```

8.4 实验现象

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 插上串口线(两头都是母的交叉线), 打开超级终端, 配置超级终端为 115200 8-N-1, 将编译好的程序下载到开发板, 可看到 LED1 和 LED2 全亮, 超级终端打印出如下信息:

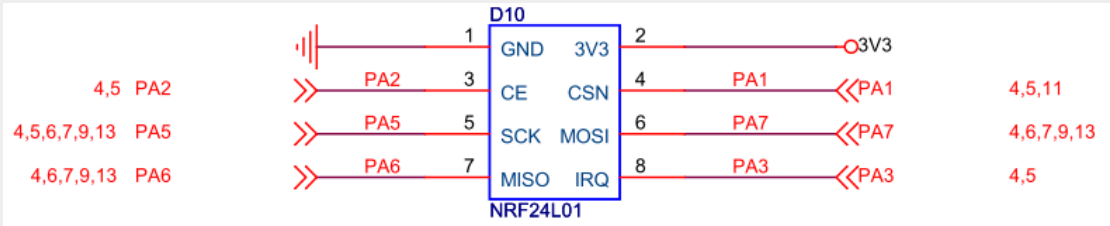


9、2.4G 无线（NRF24L01+）

9.1 实验描述及工程文件清单

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

野火 STM32 开发板 2.4G 无线模块接口图：

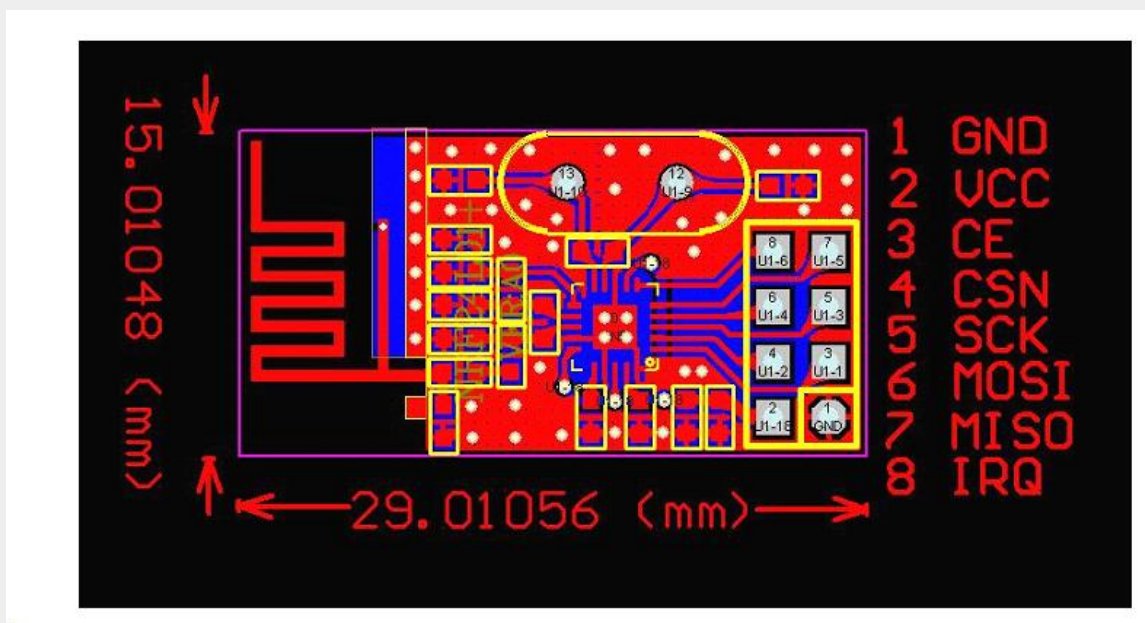


9.2 NRF24L01 模块简介

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

以下是该模块的硬件电路：

截图来自《NRF24L01 模块说明书.pdf》，page3



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

引脚说明及本实验中与开发板的连接：

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



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

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

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

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

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

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

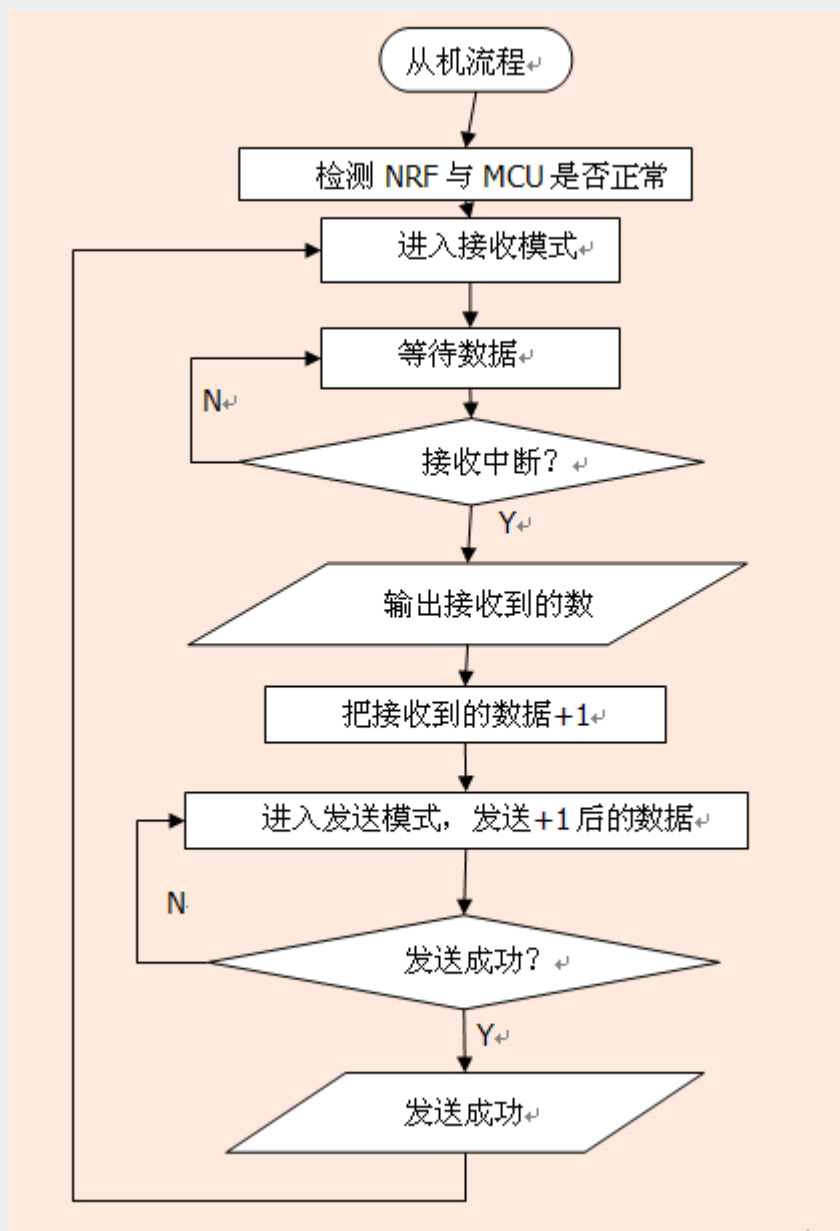
Mode	PWR_UP register	PRIM_RX register	CE input pin	FIFO state
RX mode	1	1	1	-
TX mode	1	0	1	Data in TX FIFOs. Will empty all levels in TX FIFOs ^a .
TX mode	1	0	Minimum 10μs high pulse	Data in TX FIFOs. Will empty one level in TX FIFOs ^b .
Standby-II	1	0	1	TX FIFO empty.
Standby-I	1	-	0	No ongoing packet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

9.3 代码分析

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

附从机流程图：





下面以的从机的代码为例进行分析。

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

```
9. stm32f10x_gpio.c
10. stm32f10x_rcc.c
11. stm32f10x_usart.c
12. stm32f10x_spi.c
```

还要在 **stm32f10x_conf.h** 中把相应的头文件添加进来：

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



进入 **main** 函数，边看代码边了解程序的流程：

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

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



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

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

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



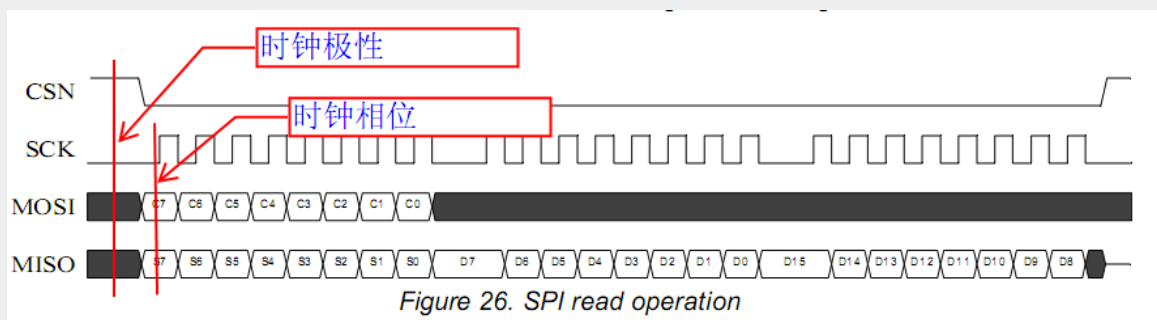
基本流程就是：

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

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

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

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



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

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



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

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

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

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

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

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

`SPI_NRF_WriteBuf()`流程：

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



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

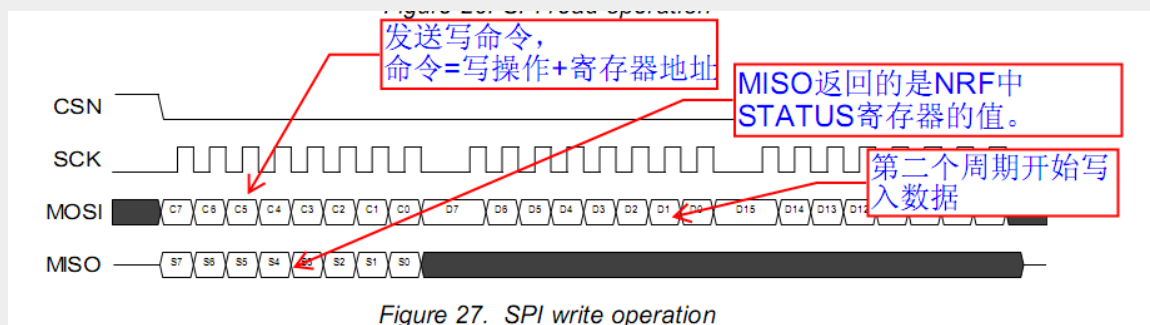
命令的组织形式: 完整的写命令 = 写命令+寄存器地址。

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

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

写时序:

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



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

式 `NRF_RX_Mode()` 函数:

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



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

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

《nRF24L01P(新版无线模块控制 IC).PDF》, page57

要注意的是从机和主机的参数要一致。

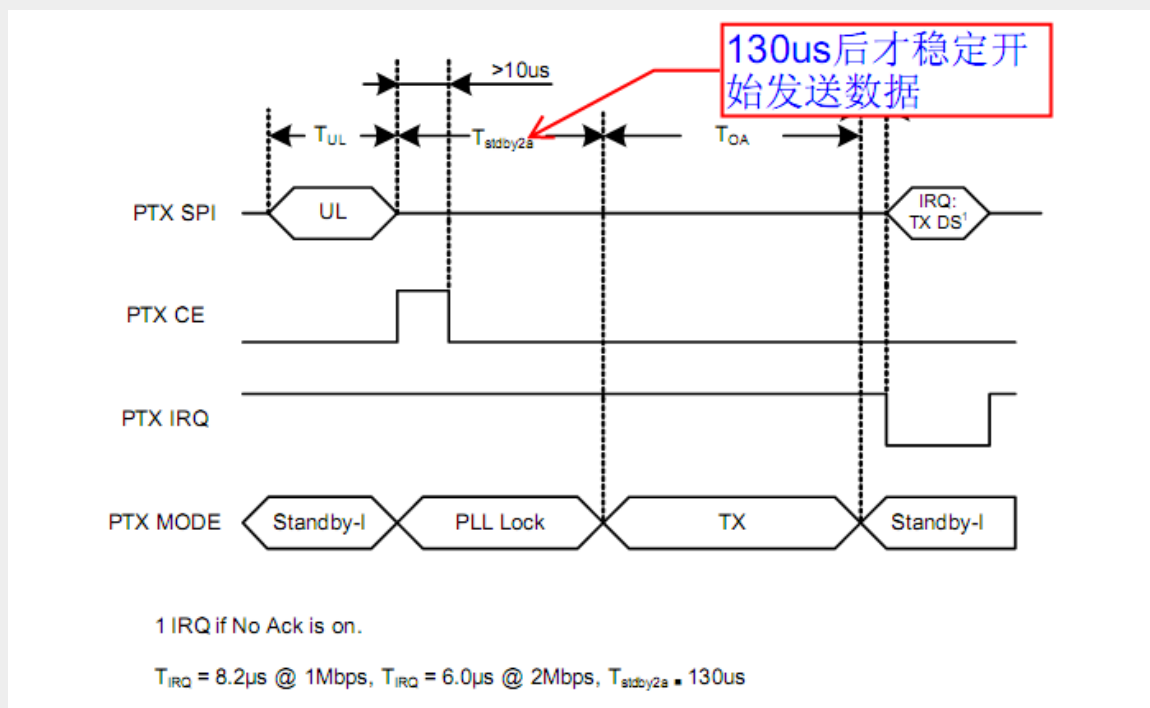
下面是发送模式的配置:

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

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



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



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

由 `NRF_Rx_Dat()` 函数来实现:

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



接收流程:

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

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

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

到这里就把所有的 NRF 驱动应用函数解说完啦! ^_^

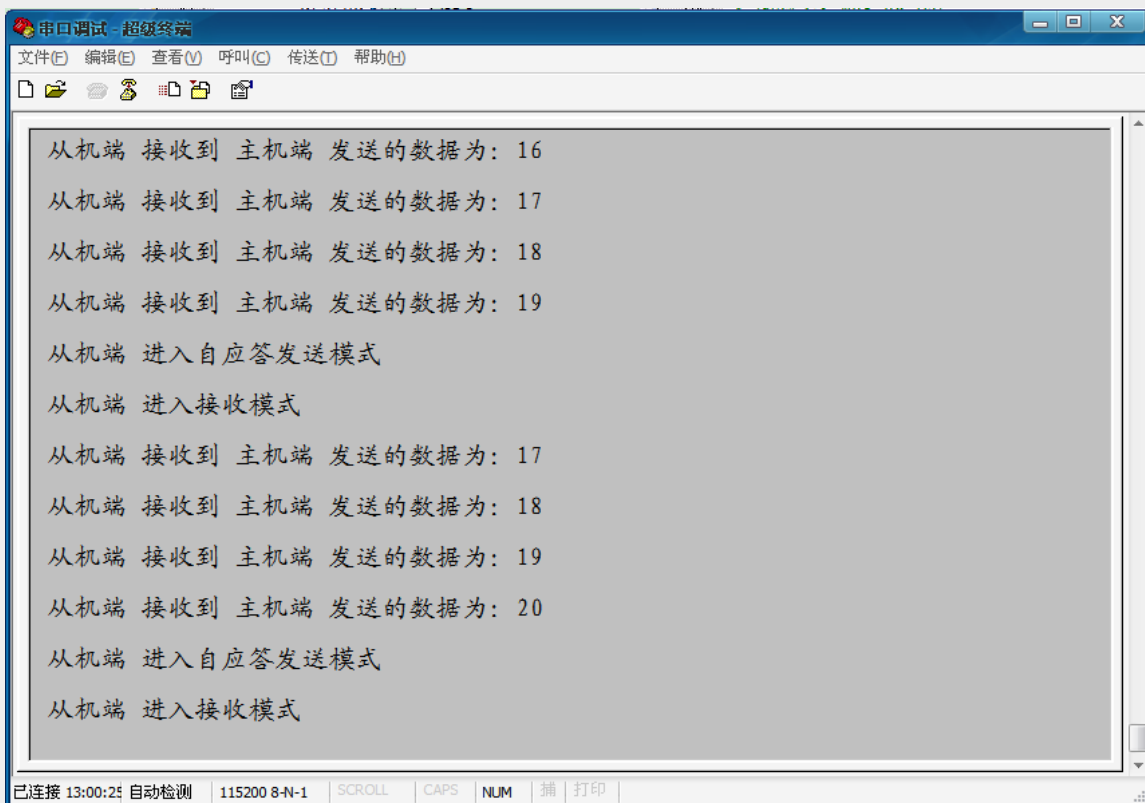
回到 main 函数, 从机把接收到的数据都加 1 之后再发送给主机, 主机又把数据发送回从机。。。如此循环, 就是整个实验的流程。

9.4 实验想象

实验时请先开启从机的电源, 再开启主机的电源, 这是代码的流程决定的。

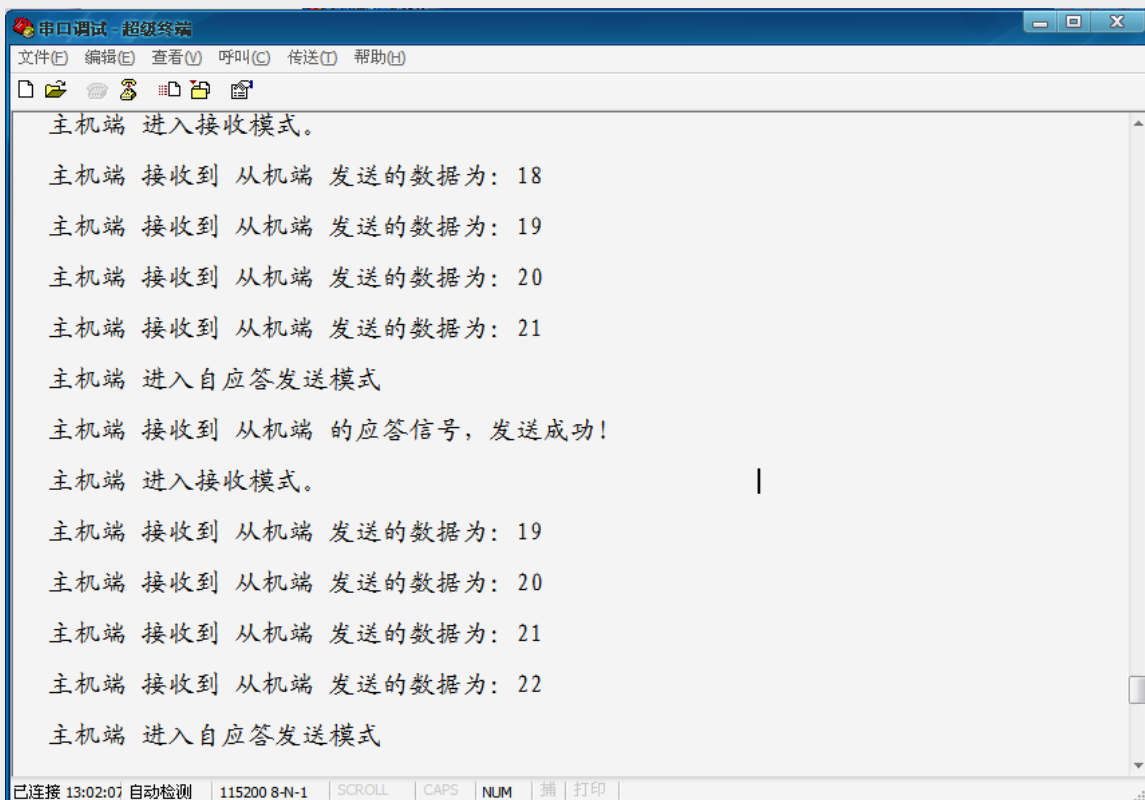


实验效果：从机的反馈：



```
从机端 接收到 主机端 发送的数据为：16
从机端 接收到 主机端 发送的数据为：17
从机端 接收到 主机端 发送的数据为：18
从机端 接收到 主机端 发送的数据为：19
从机端 进入自应答发送模式
从机端 进入接收模式
从机端 接收到 主机端 发送的数据为：17
从机端 接收到 主机端 发送的数据为：18
从机端 接收到 主机端 发送的数据为：19
从机端 接收到 主机端 发送的数据为：20
从机端 进入自应答发送模式
从机端 进入接收模式
```

主机的反馈：



```
主机端 进入接收模式。
主机端 接收到 从机端 发送的数据为：18
主机端 接收到 从机端 发送的数据为：19
主机端 接收到 从机端 发送的数据为：20
主机端 接收到 从机端 发送的数据为：21
主机端 进入自应答发送模式
主机端 接收到 从机端 的应答信号，发送成功！
主机端 进入接收模式。
主机端 接收到 从机端 发送的数据为：19
主机端 接收到 从机端 发送的数据为：20
主机端 接收到 从机端 发送的数据为：21
主机端 接收到 从机端 发送的数据为：22
主机端 进入自应答发送模式
```



10、重力感应/三轴加速（MMA7455）

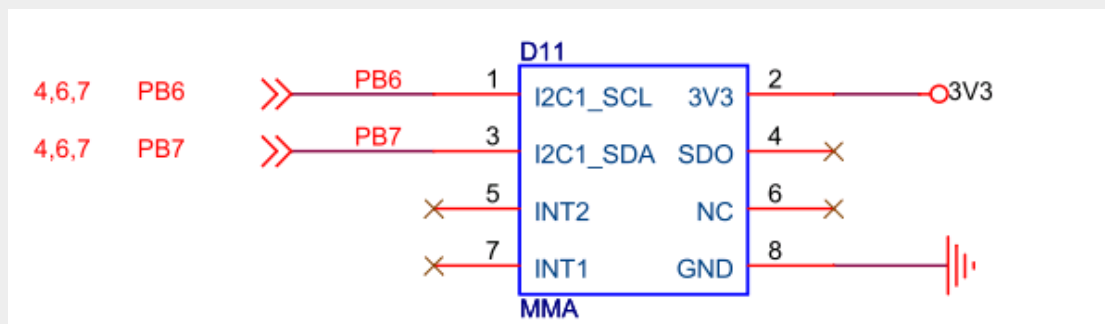
10.1 实验描述及工程文件清单

实验描述	控制倾角传感器（MMA7455）测量加速度和倾角，通过串口打印测量得的数据。
硬件连接	PB6-I2C1_SCL PB7-I2C1_SDA
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/stm32f10x_i2c.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/I2C_MMA.c

在这个加速度传感器例程中野火采用使用了集成模块，只需要把模块的电源线和 SDA，SCL 端口连接到开发板上的 I2C 总线即可。EEPROM 的也使用了 I2C 总线，所以可以参照 EEPROM 的件原理图。



野火 STM32 开发板 2.4G 无线模块接口图:



10.2 MMA7455 简介

MMA7455 是一款数字三轴加速度传感器，可以利用测量出来的重力加速度在某方向上的分量来计算器件与水平面间的夹角，测量倾角是很多有趣应用的基础。

本实验采用 I2C 方式与传感器通讯，控制传感器用 2g 的量程。

在测量前对传感器进行校准。

10.3 代码分析

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

```
13. stm32f10x_gpio.c
14. stm32f10x_rcc.c
15. stm32f10x_usart.c
16. stm32f10x_i2c.c
```

还要在 `stm32f10x_conf.h` 中把相应的头文件添加进来：

```
5. #include "stm32f10x_gpio.h"
6. #include "stm32f10x_i2c.h"
7. #include "stm32f10x_rcc.h"
8. #include "stm32f10x_usart.h"
```

配置好所需的库文件之后，我们就从 `main` 函数开始分析：

```
1. *
2. * 函数名: main
3. * 描述   : 主函数
4. * 输入   : 无
```



```
5.  * 输出 : 无
6.  * 返回 : 无
7.  */
8.  int main(void)
9.  {
10.     /* 配置系统时钟为 72M */
11.     SystemInit();
12.
13.     /* 串口1 初始化 */
14.     USART1_Config();
15.
16.     /*重力传感器初始化*/
17.     I2C_MMA_Init();
18.
19.     /*重力传感器校准*/
20.     I2C_MMA_Cal();
21.
22.     printf("\r\n-----这是一个重力传感器测试程序-----\r\n");
23.
24.
25.     /* 检测倾角*/
26.     I2C_MMA_Test(&X_Value);
27.     I2C_MMA_Test(&Y_Value);
28.     I2C_MMA_Test(&Z_Value);
29.
30.     printf("\r\n-----X 方向的数据-----\r\n");
31.     I2C_MMA_Printf(&X_Value);
32.
33.     printf("\r\n-----Y 方向的数据-----\r\n");
34.     I2C_MMA_Printf(&Y_Value);
35.
36.     printf("\r\n-----Z 方向的数据-----\r\n");
37.     I2C_MMA_Printf(&Z_Value);
38.
39.     /*进入省电模式*/
40.     if(I2C_MMA_Standby() == SUCCESS )
41.
42.         printf("\r\n Acceleration enter standby mode! \r\n");
43.     else
44.         printf("\r\n Standby mode ERROR! \r\n");
45.
46. }
```

系统库函数 `SystemInit()`；将系统时钟设置为 72M，`USART1_Config()`；配置串口，关于这两个函数的具体讲解可以参考前面的教程，这里不再详述。

```
1.  /*
2.  * 函数名: I2C_MMA_Init
3.  * 描述 : I2C 外设(MMA7455)初始化
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 外部调用
7.  */
8.  void I2C_MMA_Init(void)
9.  {
10.     I2C_GPIO_Config();
11.     I2C_Mode_Config();
12. }
```



`I2C_MMA_Init()`; 是用户编写的函数, 其中调用了 `I2C_GPIO_Config()`; 配置好 I2C 所用的 I/O 端口, 调用 `I2C_Mode_Config()`; 设置 I2C 的工作模式, 并使能相关外设的时钟。

`I2C_MMA_Cal()` 函数是用来校正传感器的, 校正需要先测量原始数据, 我们先看

`I2C_MMA_Test()` 函数

```
1.  /*
2.  * 函数名: I2C_MMA_Test
3.  * 描述   : 测量倾角 和 加速度 (量程 0-2g)
4.  * 输入   : 数据结构体的指针
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void I2C_MMA_Test(MMA_Dat* MMA_Value)
9. {
10.     u8 temp;
11.
12.     /*MMA 进入 2g 量程测试模式*/
13.     I2C_MMA_ByteWrite(0x05,MMA_MCTL_Addr);
14.
15.     /*DRDY 标置位,等待测试完毕*/
16.     while(!(I2C_MMA_ByteRead(MMA_STATUS_Addr)&0x01));
17.
18.     /*读取测得的数据*/
19.     MMA_Value->Out = I2C_MMA_ByteRead(MMA_Value->Addr);
20.
21.     if((MMA_Value->Out&0x80) ==0x00) /*读出的原始值为正数 */
22.     {
23.         temp = MMA_Value->Out;
24.
25.         /*将原始值转换为加速度, 乘以 -1 为方向处理*/
26.         MMA_Value->Acc = (float) (-1)*temp *ACC_Gravity/64;
27.
28.         /*将原始值转换为角度*/
29.         if(temp >=64)
30.             /*加速度值大于 1g */
31.             MMA_Value->Angle = 90.0;
32.
33.         else
34.             /*加速度小于 1 g, Angle = asin(Acc/9.8)*57.32; 弧度制转换
35.             57.32 = 180/3.14*/
36.             MMA_Value->Angle = asin((float) temp/64)*57.32;
37.
38.         /*读出的原始值为负数 */
39.     else
40.     {
41.         temp = MMA_Value->Out;
42.
43.         /*二补码转换*/
44.         temp -= 1;
45.         temp = ~temp;
46.
47.         /*将原始值转换为加速度*/
48.         MMA_Value->Acc = (float) temp *ACC_Gravity/64;
49.
50.         /*将原始值转换为角度, 乘以 -1 为方向处理*/
51.         if(temp>=64)
52.             MMA_Value->Angle = -90.0;
53.         else
54.             /* Angle = asin(Acc/9.8)*57.32 */
55.             MMA_Value->Angle = (-1)*asin((float) temp/64)*57.32;
```



```
56.     }  
57. }
```

这个函数调用了 `I2C_MMA_ByteWrite()`，以下是函数原型：

```
1.  /*  
2.   * 函数名: I2C_MMA_ByteWrite  
3.   * 描述  : 写一个字节到 I2C MMA 寄存器中  
4.   * 输入  : -pBuffer 缓冲区指针  
5.   *          -WriteAddr 接收数据的 MMA 寄存器的地址  
6.   * 输出  : 无  
7.   * 返回  : 无  
8.   * 调用  : 内部调用  
9.   */  
10. static void I2C_MMA_ByteWrite(u8 pBuffer, u8 WriteAddr)  
11. {  
12.     /*wait until I2C bus is not busy*/  
13.     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));  
14.  
15.     /* Send START condition */  
16.     I2C_GenerateSTART(I2C1, ENABLE);  
17.  
18.     /* Test on EV5 and clear it */  
19.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));  
20.  
21.     /* Send MMA address for write */  
22.     I2C_Send7bitAddress(I2C1, MMA_ADRESS, I2C_Direction_Transmitter);  
23.  
24.     /* Test on EV6 and clear it */  
25.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECT  
26.                             ED));  
27.  
28.     /* Send the MMA's Register address to write to */  
29.     I2C_SendData(I2C1, WriteAddr);  
30.  
31.     /* Test on EV8 and clear it */  
32.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));  
33.  
34.     /* Send the byte to be written */  
35.     I2C_SendData(I2C1, pBuffer);  
36.  
37.     /* Test on EV8 and clear it */  
38.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));  
39.  
40.     /* Send STOP condition */  
41.     I2C_GenerateSTOP(I2C1, ENABLE);  
42. }
```

这与 EEPROM 的 `I2C_EE_PageWrite()` 函数很类似，第一个参数为要写入的数值，第二个参数为待写入寄存器的地址。这些寄存器的地址在 `I2C_MMA.h` 文件中都有宏定义。

其中要强调的地方是 MMA7455 的器件地址（分清器件地址与寄存器地址的区别哦！），在 MMA7455 的 DataSheet 中查到的器件地址为 `0X1D`，但这个地址是不正确的，与野火使用的这块芯片有区别，市面上的传感器一般这个器件地址也为 `0x3A`。



在用 Jlink 调试程序的时候，程序循环运行在第 25 行就可以知道是器件地址出了问题，其它通讯的错误也可以用这样的方式查找出来，省了示波器 ^_^。

I2C_MMA_ByteRead() 类似，功能是读取寄存器的值，就不分析了。

回到 I2C_MMA_Test() 函数中，I2C_MMA_Test() 的第 13 行代码是向 MMA7455 写入命令，开启 2g 转换模式，开启了检测模式后，通过查询 MMA7455 的 DRDY 位，等待 MMA7455 的加速度转化完成。以下为 DataSheet 的说明，附上野火的注释：

截图来自《MMA7455L》

\$16: Mode Control Register (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0	Bit
--	DRPD	SPI3W	STON	GLVL[1]	GLVL[0]	MODE[1]	MODE[0]	Function
0	0	0	0	0	0	0	0	Default

Table 5. Configuring the g-Select for 8-bit output using Register \$16 with GLVL[1:0] bits

GLVL [1:0]	g-Range	Sensitivity
00	8g	16 LSB/g
01	2g	64 LSB/g
10	4g	32 LSB/g

2g模式最小量化单位

Standby Mode

This digital output 3-axis accelerometer provides a standby mode that is ideal for battery operated products. When standby mode is active, the device outputs are turned off, providing significant reduction of operating current. When the device is in standby mode the current will be reduced to 2.5 μ A typical. In standby mode the device can read and write to the registers with the I²C/SPI available, but no new measurements can be taken in this mode as all current consuming parts are off. The mode of the device is controlled through the mode control register by accessing the two mode bits as shown in Table 6.

Table 6. Configuring the Mode using Register \$16 with MODE[1:0] bits

MODE [1:0]	Function
00	Standby Mode
01	Measurement Mode
10	Level Detection Mode
11	Pulse Detection Mode

使用检测模式

Measurement Mode

The device can read XYZ measurements in this mode. The pulse and threshold interrupt mode, continuous measurements on all three axes enabled. The g-range for 2g, 4g, or 8g is selectable with 10-bit data. The sample rate during measurement mode is 250 Hz with the 125 Hz filter selected. Therefore, when a conversion is complete (signaled by the DRDY flag), the next measurement will be ready.

读取数据前检查 DRDY 寄存器

When measurements on all three axes are completed, a logic high level is output to the DRDY pin, indicating "measurement data is ready." The DRDY status can be monitored by the DRDY bit in Status Register (Address: \$09). The DRDY pin is kept high until one of the three Output Value Registers are read. If the next measurement data is written before the previous data is read, the DOVR bit in the Status Register will be set. Also note that in measurement mode, level detection mode and pulse detection mode are not available.

转化完成后根据传入的参数，把相应的 MMA7455 数据寄存器中的数据读取出来，各个方向的原始数据储存在相应的结构体的 .Out 变量中：

```
1. typedef struct
2. {
3.     uc8 Addr;           //寄存器的地址
4.     uc8 Name;           //数据的方向，X，Y 或 Z
5.     s8 Out;             //寄存器的值
6.     float Acc;          //加速度值
7.     float Angle;        //角度值
8. }MMA_Dat;
```

在结构体初始化的时候把寄存器的地址和名称定义好了：



```
1.  /*****/
2.   MMA_Dat X_Value={MMA_XOUT8_Addr, 'X'};
3.   MMA_Dat Y_Value={MMA_YOUT8_Addr, 'Y'};
4.   MMA_Dat Z_Value={MMA_ZOUT8_Addr, 'Z'};
```

要注意的是 MMA7455 中的数据以**二补数**的形式来储存:

截图来自《MMA7455L》

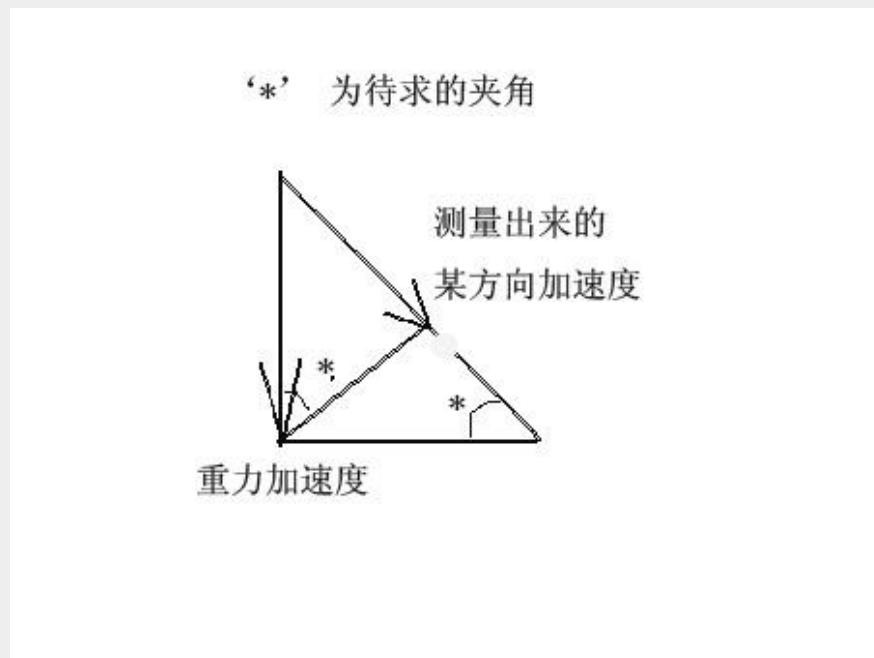
\$00: 10bits Output Value X LSB (Read only)

D7	D6	D5	D4	D3	D2	D1	D0	Bit
XOUT [7]	XOUT [6]	XOUT [5]	XOUT [4]	XOUT [3]	XOUT [2]	XOUT [1]	XOUT [0]	Function
0	0	0	0	0	0	0	0	Default

Signed byte data (2's complement): $0g = 10'h000$

把二补数转换为原码后就可以像处理 ADC 的数据一样了, 我们选择的是 **2g** 量程, 灵敏度为: **64LSB/g**。

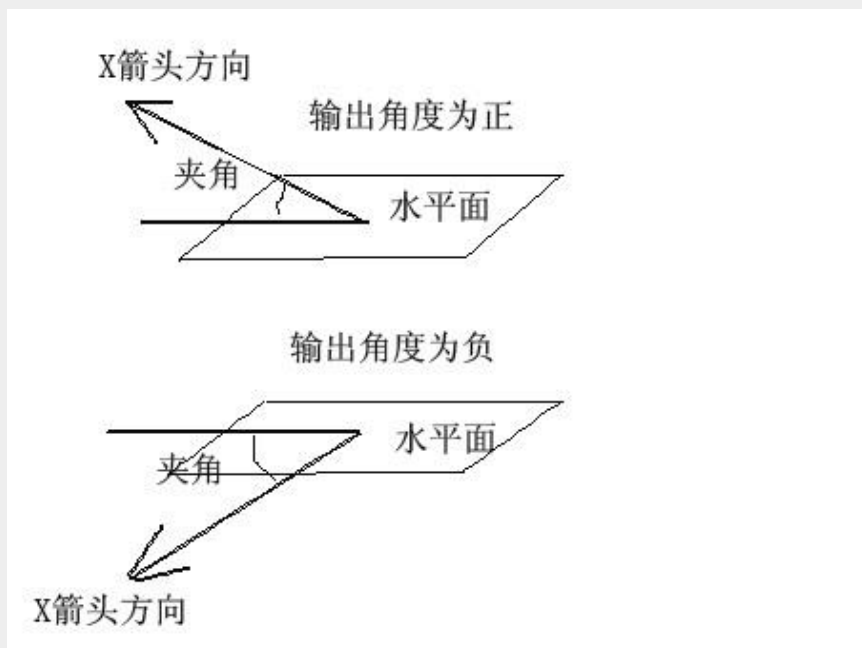
至于角度的计算公式, 把重力加速度按各方向的分解, 可以推导出来。



计算出来的 **加速度或角度** 乘以 **-1** 是野火根据传感器上标注的**箭头的方向**, 并以此为**正方向**进行调整的, 不同的厂家生产的传感器稍有区别。

下图是以 X 方向输出的角度为例:





输出的角度为各箭头方向的直线与水平面间的夹角，在水平面上方为正，下方为负。

接下来分析一下 校准函数：

```
1. /*
2.  * 函数名: I2C_MMA_Cal
3.  * 描述 : MMA7455 校准
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 外部调用, 在初始化后调用
7.  */
8. void I2C_MMA_Cal(void)
9. {
10.     I2C_MMA_Write(0x00, MMA_YOFFL_Addr); /*校正 x 值 00 */
11.     I2C_MMA_Write(0x30, MMA_YOFFL_Addr); /*校正 y 值 48*/
12.     I2C_MMA_Write(0xE2, MMA_ZOFFL_Addr); /*校正 z 值 -30 的补码 */
13.     I2C_MMA_Write(0xFF, MMA_ZOFFH_Addr); /*校正 z 值, 校正值为负数, 要把
        高位写 1;*/
14. }
```

`I2C_MMA_Cal()` 函数是用来校正传感器的，称为 **0g 校准**。这个函数在第一次测量前必须调用，而且每个传感器的校正值都有不同，其中的校正参数就要大家亲手去调试出来啦。

参照 DataSheet《AN3745》按以下步骤校准：

1. 把传感器按水平方式放置，读取各方向寄存器输出值。

这个情况下，Z 轴方向标准输出应为 **1g**，X 轴和 Y 轴均为 **0**。对应到各个寄存器的原始数据就应是 **ZOUT8 = 64**，**XOUT8 = 0**，**YOUT8 = 0**。但是未校准前，各个寄存器的输出会有一定的**偏差**。



以下为传感器水平放置，野火的例程中未校准得出的数据

```

-----这是一个重力传感器测试程序-----
-----X方向的数据-----
寄存器的原始数据是: XOUT8 = 0
以 箭头标注 方向为 正 方向, X方向加速度是: 0.00 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: 0.00 度
-----Y方向的数据-----
寄存器的原始数据是: YOUT8 = -19
以 箭头标注 方向为 正 方向, Y方向加速度是: 2.91 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -17.28 度
-----Z方向的数据-----
寄存器的原始数据是: ZOUT8 = 75
以 箭头标注 方向为 正 方向, Z方向加速度是: -11.48 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: 90.00 度
Acceleration enter standby mode!

```

2.向相应的 OffSet 寄存器写入校准值:

要注意两个问题。

一是校准寄存器中的值为 **1/2 LSB**，所以我们写入的值要相应地**乘以 2**倍:

XOUT8 很标准，不用写入校准值，或向 **XOFFL** 写入 0;

YOUT8 输出为-19，所以应 **YOFFL** 写入 $38 = 2 \times 19$;

ZOUT8 输出为 75，所以应写入 $-22 = 2 \times (64 - 75)$;



截图来自《MMA7455L》

\$10: Offset Drift X LSB (Read/Write)

The following Offset Drift Registers are used for setting and storing the offset calibrations to eliminate the 0g offset. Please refer to Freescale application note AN3745 for detailed instructions on the process to set and store the calibration values.

D7	D6	D5	D4	D3	D2	D1	D0	Bit
XOFF[7]	XOFF[6]	XOFF[5]	XOFF[4]	XOFF[3]	XOFF[2]	XOFF[1]	XOFF[0]	Function
0	0	0	0	0	0	0	0	Default

Signed byte data (2's complement): User level offset trim value for X-axis

Bit	XOFF[7]	XOFF[6]	XOFF[5]	XOFF[4]	XOFF[3]	XOFF[2]	XOFF[1]	XOFF[0]
Weight*	64 LSB	32 LSB	16 LSB	8 LSB	4 LSB	2 LSB	1 LSB	0.5 LSB

*Bit weight is for 8g 10-bit data output. Typical value for reference only. Variation is specified in "Electrical Characteristics" section.

\$11: Offset Drift X MSB (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0	Bit
--	--	--	--	--	XOFF[10]	XOFF[9]	XOFF[8]	Function
0	0	0	0	0	0	0	0	Default

Signed byte data (2's complement): User level offset trim value for X-axis

当校准值要写入负数时，要注意第二个问题：

就是在 MMA7455 寄存器中数值是以补码的形式储存的，所以实际写入的数据要经过转化：-22 的补码为 0xEA。但是，向 ZOFFL 写入 0xEA 还是未能校准，因为校准寄存器的还有高 8 位，高 8 位必须全写入 1 才是 -22 的补码，所以还要向 ZOFFH 写入 0xff。

选取恰当的校准值往往要经过多次的检测，为了提高准确度，就辛苦一点吧。

最后讲解一下 Standby 模式，只要向 MCTL 寄存器写入 0x04 命令就可进入 Standby 模式，这时传感器不工作，功耗大大降低：

```
1. *
2. * 函数名: I2C_MMA_Standby
3. * 描述   : 重力传感器进入 Standby 模式，省电
4. * 输入   : 无
5. * 输出   : 无
6. * 返回   : 是否成功进入 Standby 模式
7. * 调用   : 外部调用
8. */
9. u8 I2C_MMA_Standby(void)
10. {
11.     u8 MMA_Test;
12.
13.     /*MMA Standby Mode*/
14.     I2C_MMA_ByteWrite(0x04, MMA_MCTL_Addr);
15.
16.     /*MMA_Test*/
17.     MMA_Test = I2C_MMA_ByteRead(MMA_MCTL_Addr);
18.
19.     if(MMA_Test == 0x04)
20.         return SUCCESS;
21.     else
22.         return ERROR;
23. }
```

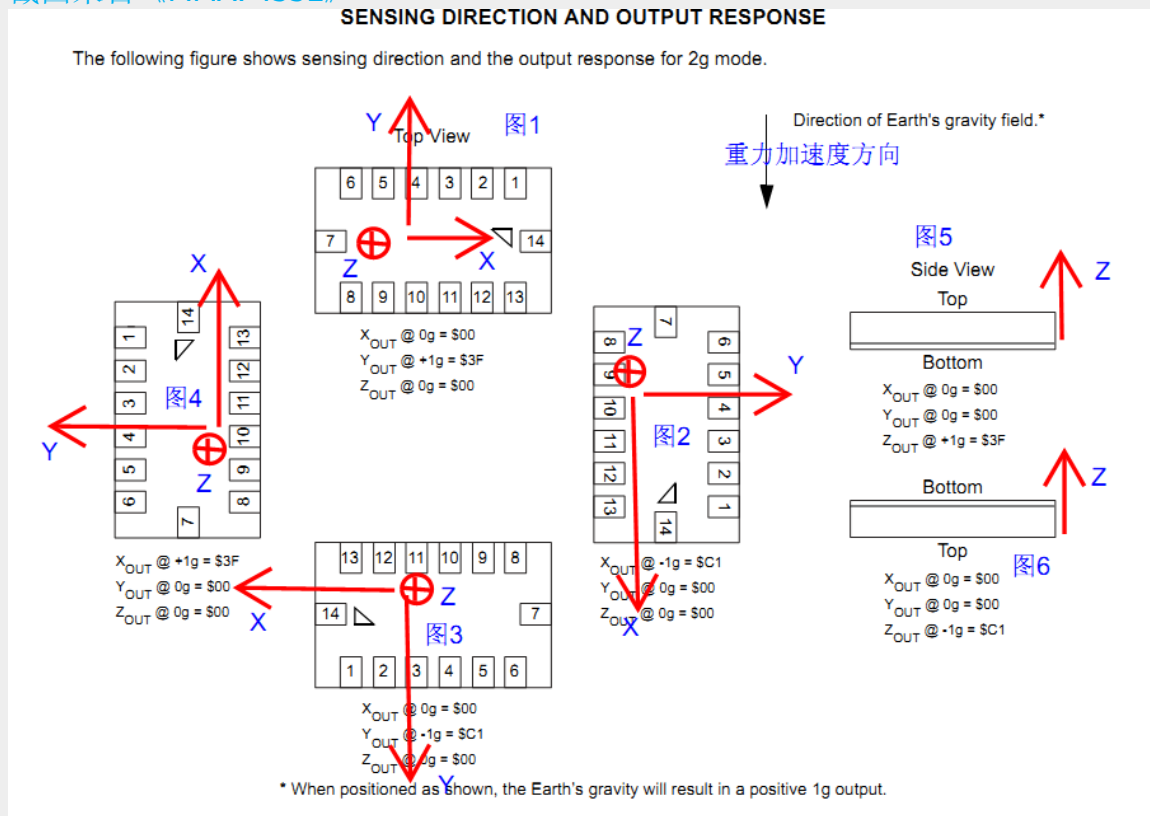


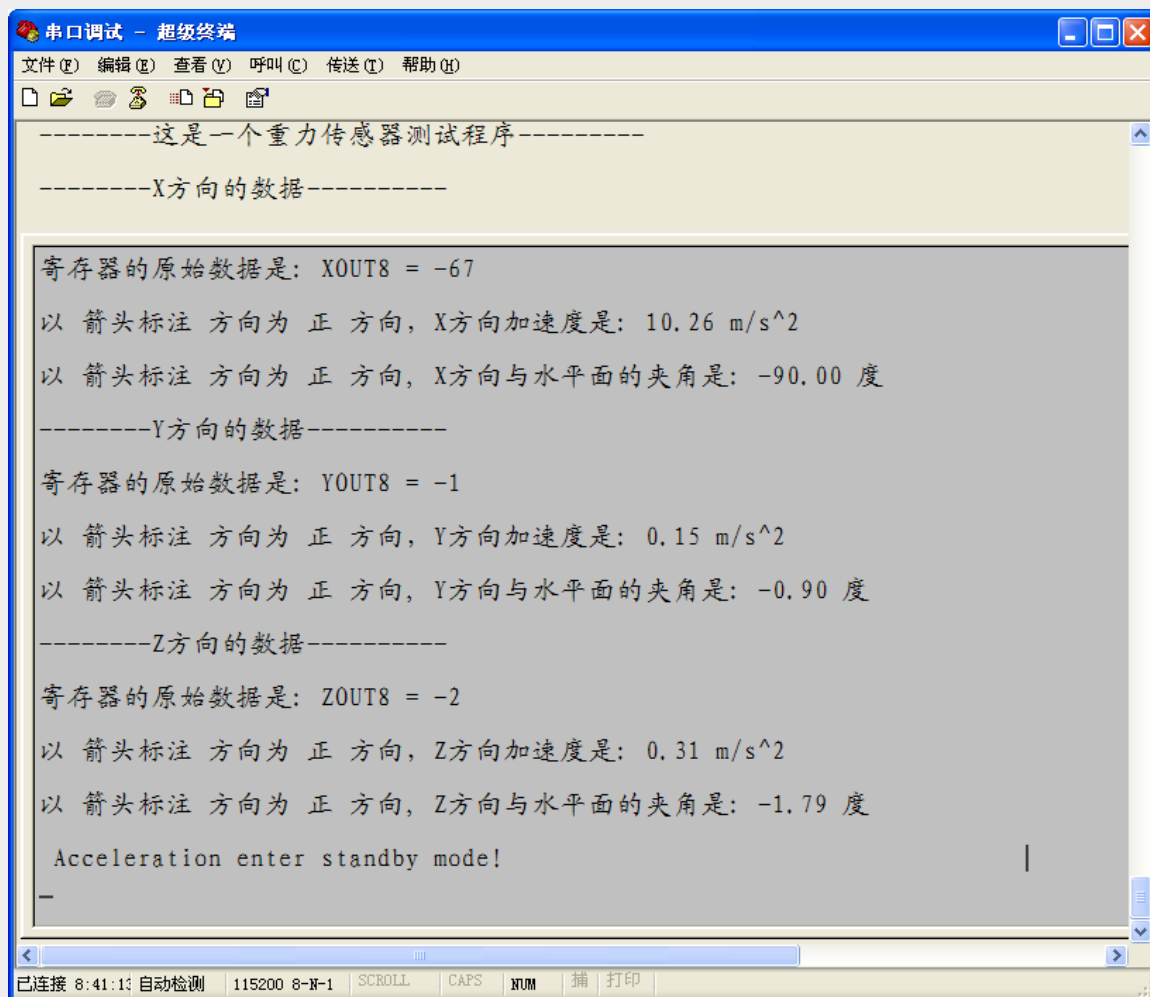
10.4 实验现象

很多第一次使用加速度传感器的人都会被它的方向弄糊涂，以下为传感器分别按 DataSheet 的这图 3，图 4，图 5，三个个方向来放置时没得的数据。

注释中的各个 XYZ 轴的方向为野火使用的传感器上标注的箭头方向，设为正方向。将野火 STM32 开发板供电(DC5V)，插上 JLINK。

截图来自《MMA7455L》

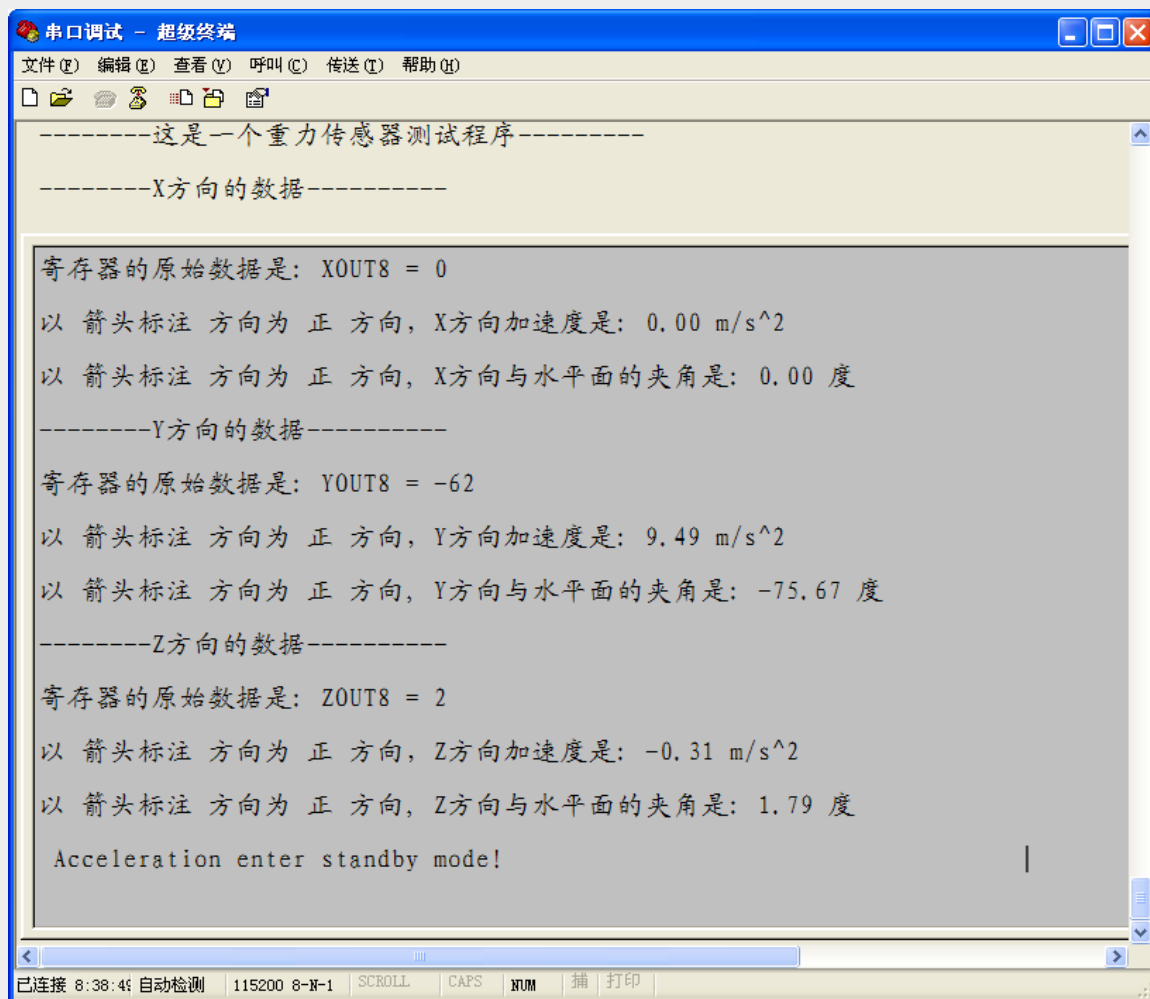




```
串口调试 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
-----这是一个重力传感器测试程序-----
-----X方向的数据-----
寄存器的原始数据是: XOUT8 = -67
以 箭头标注 方向为 正 方向, X方向加速度是: 10.26 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: -90.00 度
-----Y方向的数据-----
寄存器的原始数据是: YOUT8 = -1
以 箭头标注 方向为 正 方向, Y方向加速度是: 0.15 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -0.90 度
-----Z方向的数据-----
寄存器的原始数据是: ZOUT8 = -2
以 箭头标注 方向为 正 方向, Z方向加速度是: 0.31 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: -1.79 度
Acceleration enter standby mode!
-
已连接 8:41:13 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

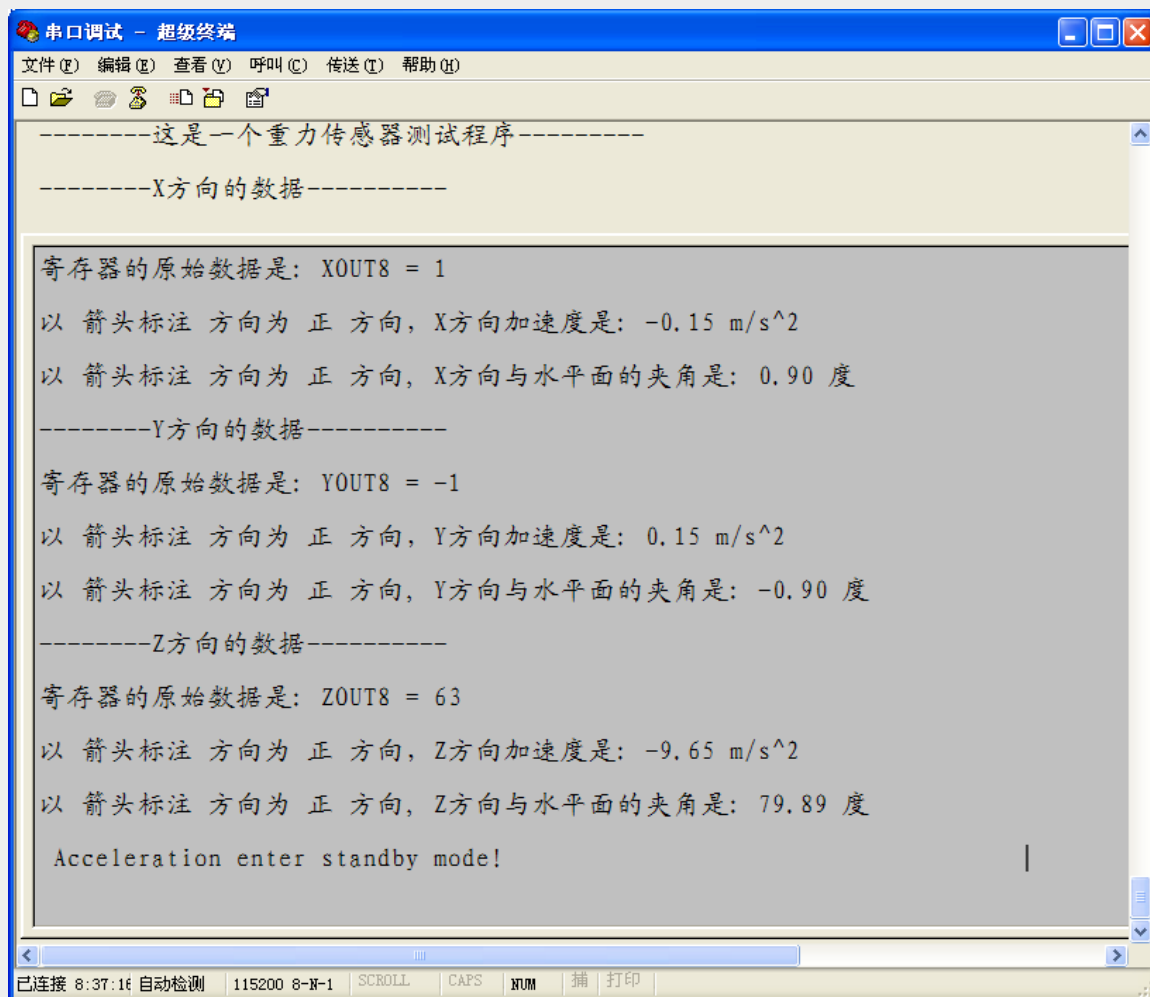
按图 2 方向放置





按图 3 方向放置





```
串口调试 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
-----这是一个重力传感器测试程序-----
-----X方向的数据-----
寄存器的原始数据是: XOUT8 = 1
以 箭头标注 方向为 正 方向, X方向加速度是: -0.15 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: 0.90 度
-----Y方向的数据-----
寄存器的原始数据是: YOUT8 = -1
以 箭头标注 方向为 正 方向, Y方向加速度是: 0.15 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -0.90 度
-----Z方向的数据-----
寄存器的原始数据是: ZOUT8 = 63
以 箭头标注 方向为 正 方向, Z方向加速度是: -9.65 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: 79.89 度
Acceleration enter standby mode!
已连接 8:37:16 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

