

零死角玩转STM32

与野火同行 乐意惬意无边



原创教程，完全开源。



由浅入深，结合实操。



通俗易懂，详尽解读。



配套板子，全面玩转。



强强联合，不断更新。



野火团队 Wild Fire Team



0、友情提示

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

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



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。

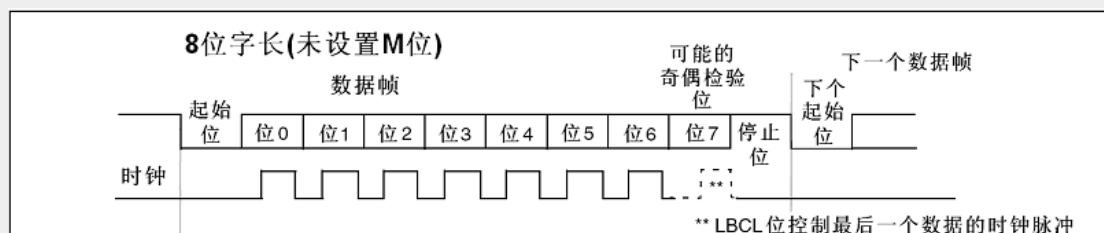


图 1-1 异步串口通讯协议

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。



1.2 直通线和交叉线

野火 STM32 开发板串口硬件原理图

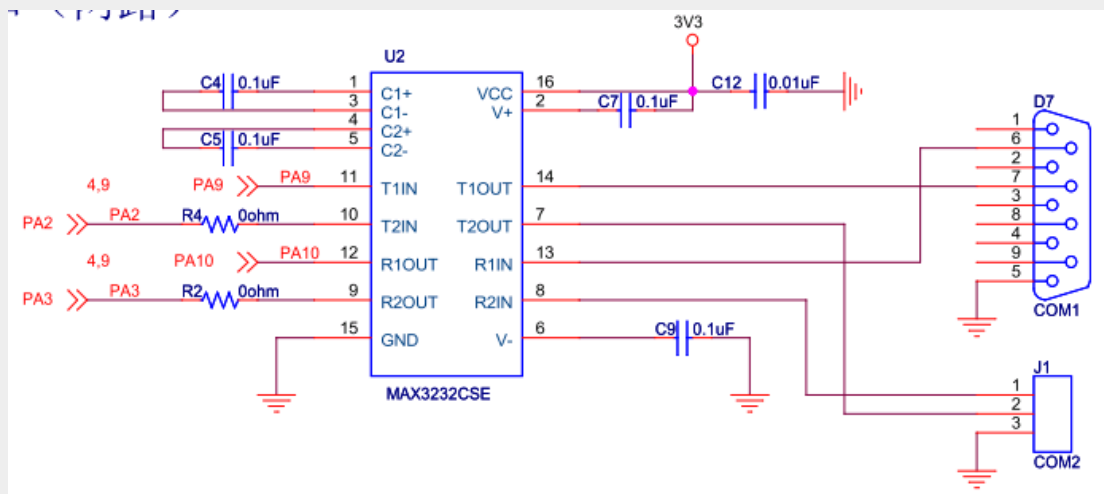


图 1-2 野火开发板串口硬件图

见图 1-2，这是野火 STM32 开发板的接线图，使用的为 MAX3232 芯片，把 STM32 的 **PA10** 引脚(复用功能为 **USART1** 的 **Rx**)接到了 DB9 接口的第 2 针脚，把 **PA9** 引脚(复用功能为 **USART** 的 **Tx**)连接到了 DB9 接口的第 3 针脚。

Tx (发送端) 接第 3 针脚, **Rx** (接收端) 接第 2 针脚。这种接法是跟 PC 的串口接法一样的，如果要实现 PC 跟野火板子通讯，就要使用两头都是母的交叉线。

串口线主要分两种，**直通线** (平行线) 和 **交叉线**。它们的区别见图 1-3。假如 PC 与板子之间要实现全双工串口通讯，必然是 PC 的 **Tx** 针脚要连接到板子的 **Rx** 针脚，而 PC 的 **Rx** 针脚则要连接至板子的 **Tx** 针脚了。由于板子和 pc 的串口接法是**相同**的，就要使用**交叉线**来连接了。如果有的开发板是 **Tx** 连接至 DB9 的第 2 针脚，而 **Rx** 连接至第 3 针脚，这与 PC 接法是**相反**的，这样的板子与 PC 通讯就需要使用**直通线**了。

为什么野火板子要使用 PC 的接法？

假如使用非 PC 接法，由于板子与 PC 的**接法相反**，通讯就要**使用直通线**；但两个板子之间想要进行串口通讯时，由于**接法相同**，就要使用**交叉线**。如果使用 PC 接法，板子与 PC 之间**接法相同**，通讯使用**交叉线**；两个相同板子之间接法也相同，通讯也是使用交叉线。

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。

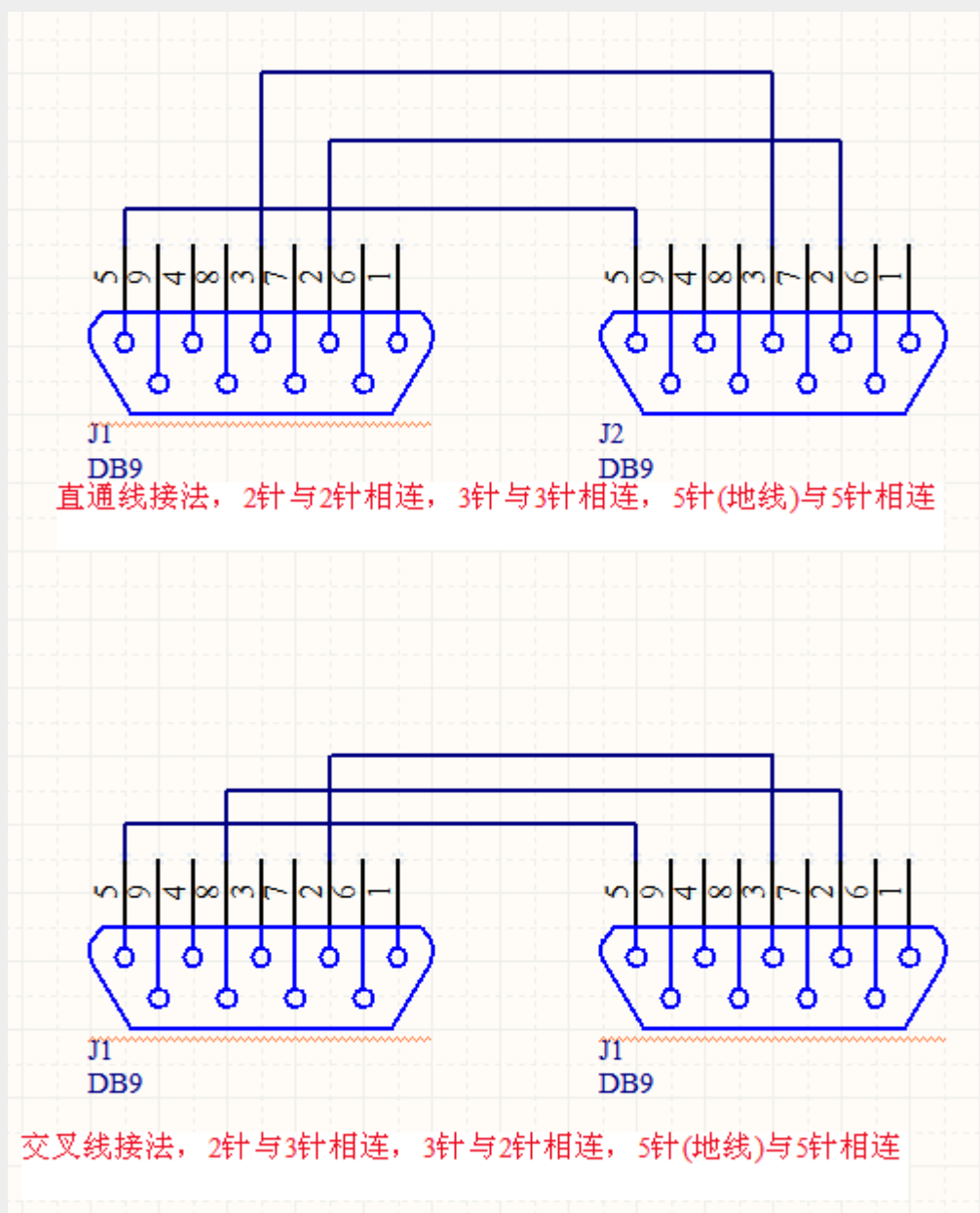


图 1-3 交叉线与直通线的区别

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

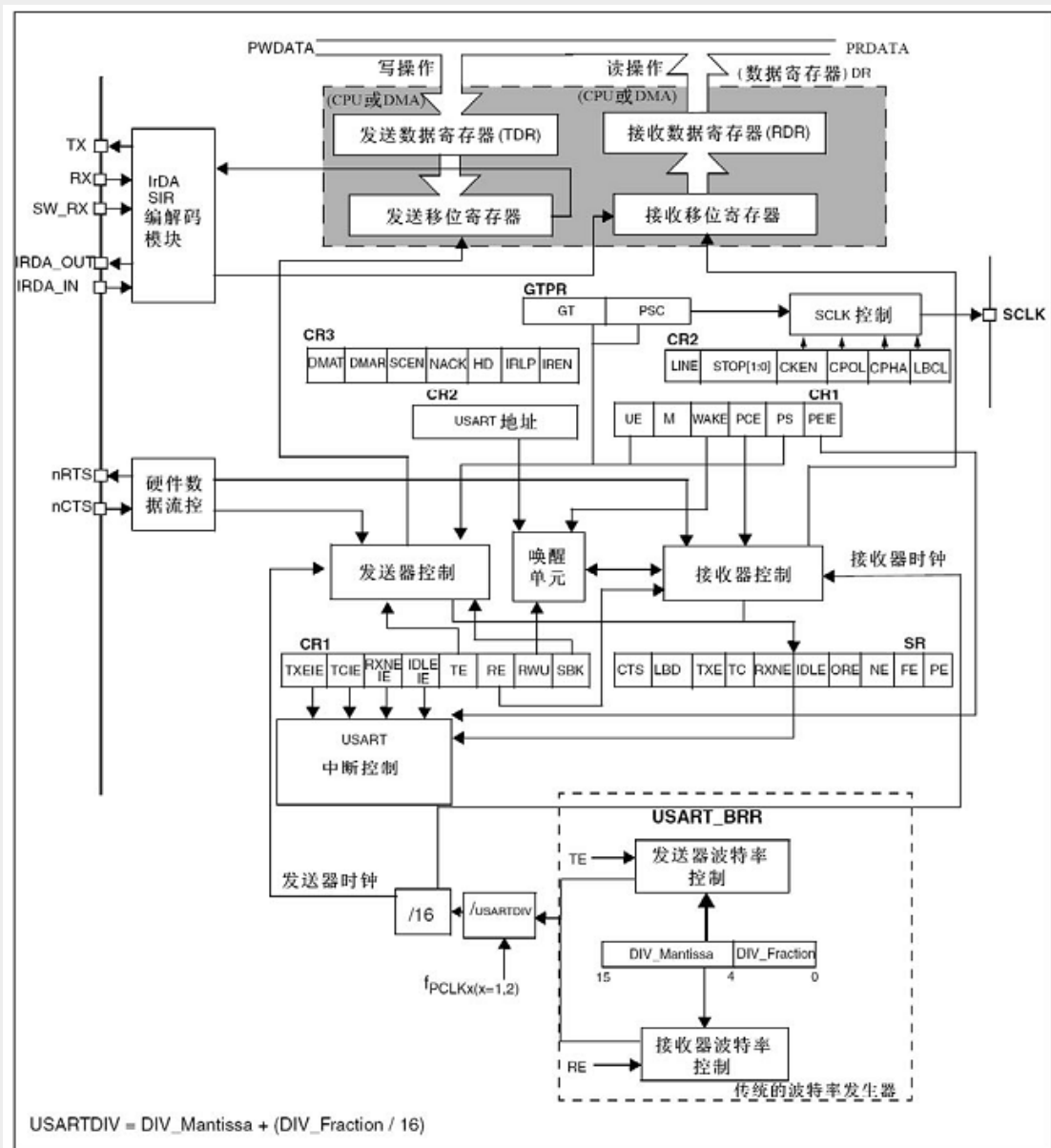


图 1-4 串口架构图

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。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。

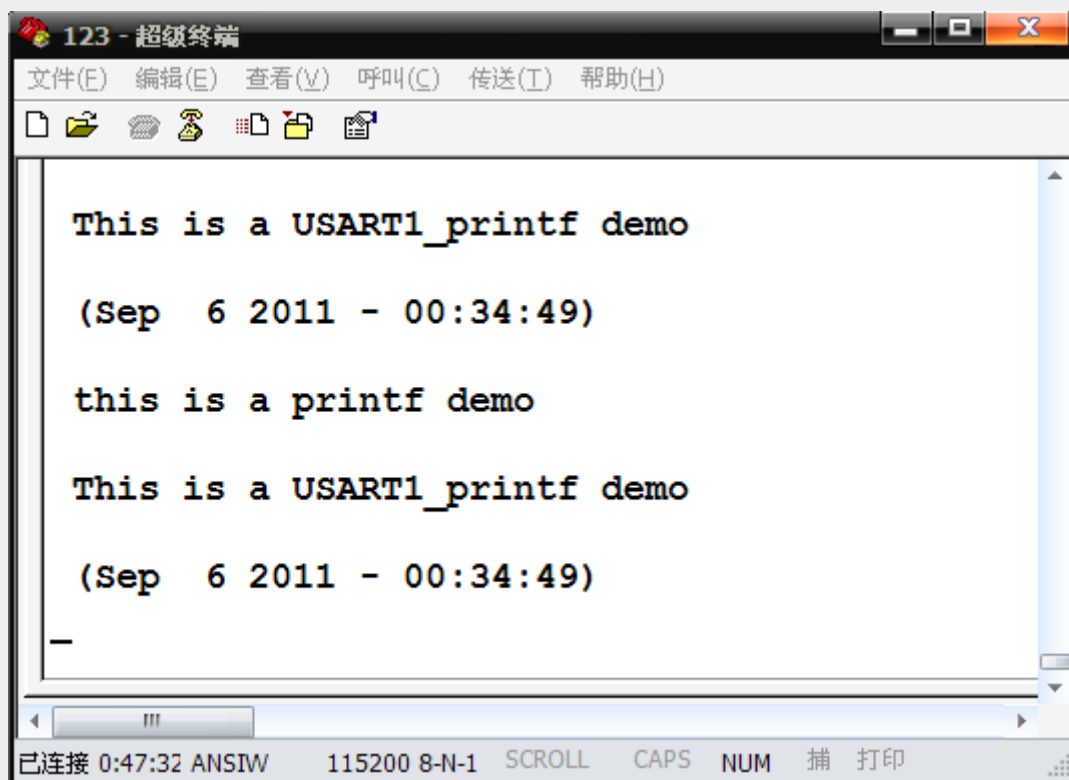
8. 最后，调用 *USART_Cmd()* 使能 *USART1* 外设。在使用外设时，不仅要使能其时钟，还要调用此函数使能外设才可以正常使用。

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。



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

This is a USART1_printf demo

(Sep 6 2011 - 00:34:49)

this is a printf demo

This is a USART1_printf demo

(Sep 6 2011 - 00:34:49)

已连接 0:47:32 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

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。

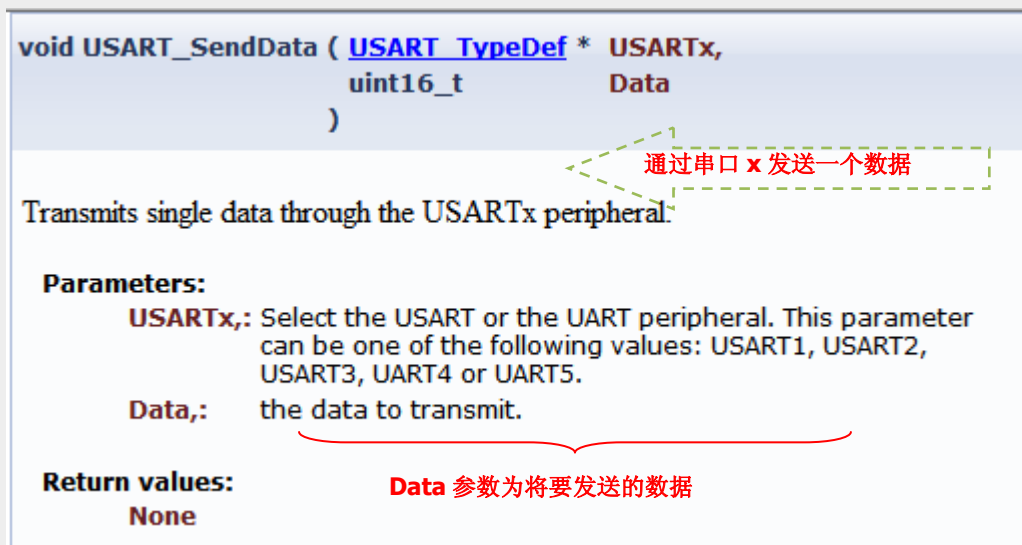


图 1-8 串口发送函数



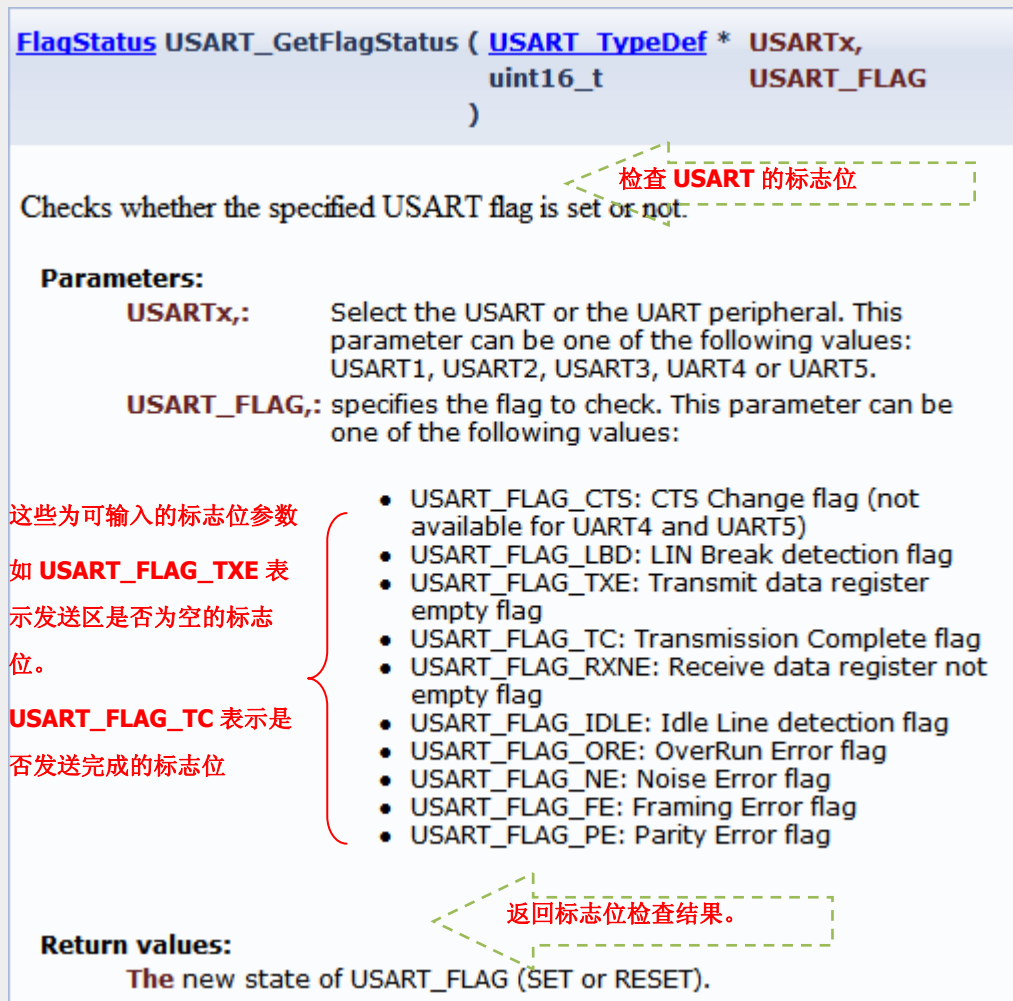


图 1-9 串口标志位检查函数

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。

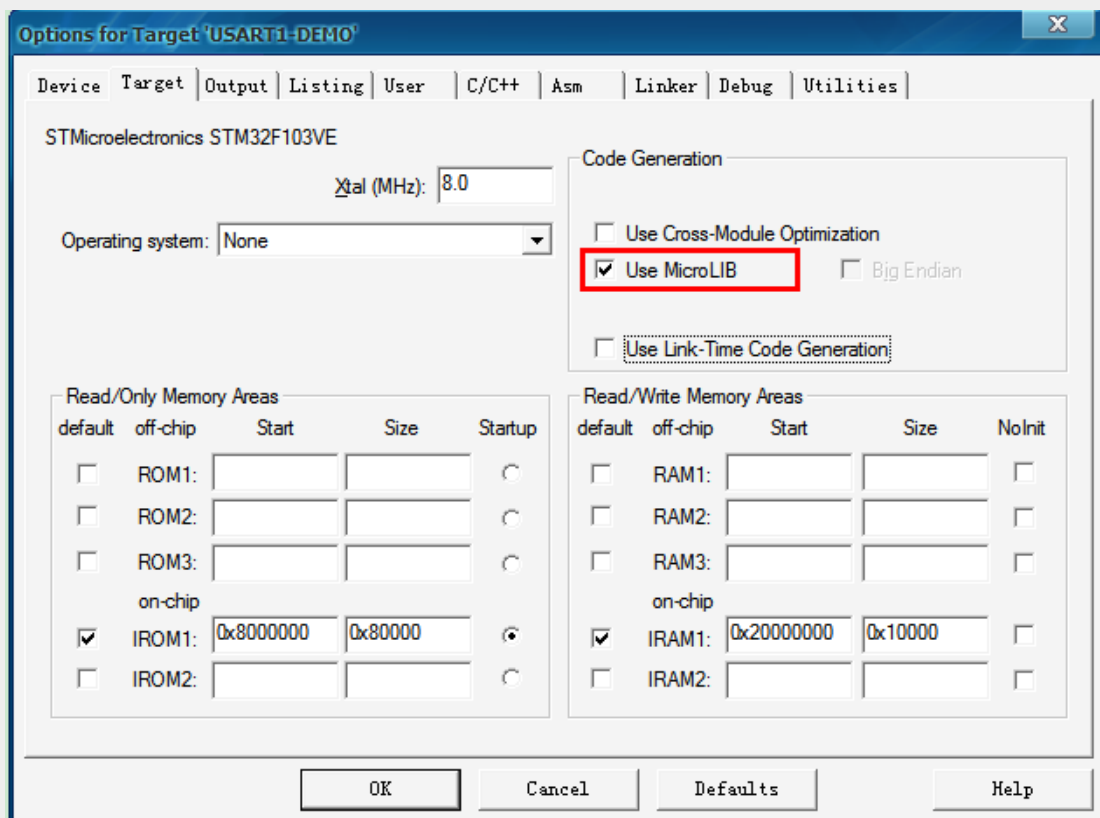


图 1-10 勾选使用微库

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. }
```

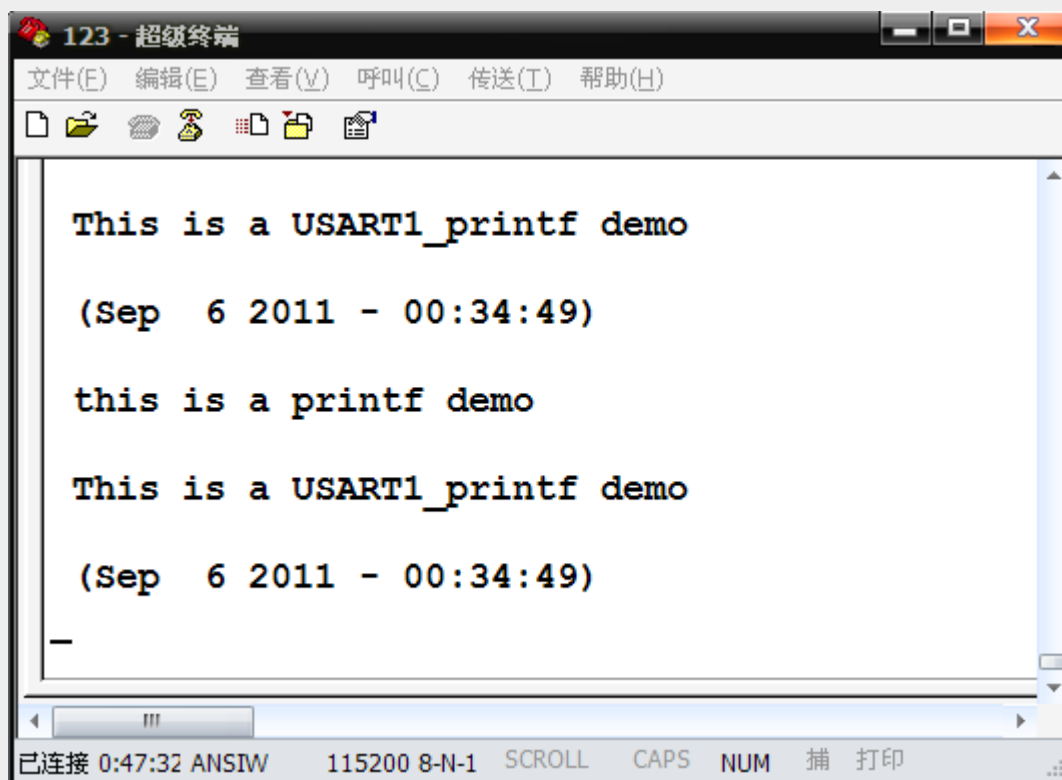
这部分代码有点多，在格式上编排不是很好，野火推荐大家直接看源码好点。

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：



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

This is a USART1_printf demo

(Sep 6 2011 - 00:34:49)

this is a printf demo

This is a USART1_printf demo

(Sep 6 2011 - 00:34:49)

已连接 0:47:32 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

