



## 野火 stm32 开发板简介

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	<b>313303034</b>
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	<b>ST3.0.0</b>

大家好，我是野火，下面是我们野火嵌入式开发工作室的成员和 QQ 联系方式：

野火： 313303034      山外メ雲シ： 860317732

昨夜溪风： 859857177      木易木木木： 772406599

如果您在学习 stm32 时遇到有什么问题或者对我们的野火 stm32 开发板有什么建议，

我们很欢迎您跟我们讨论，这是我们的 QQ 交流群：**野火 stm32 高级群 4:**

**178385907**。野火非常乐意与大家分享学习 stm32 的乐趣，新手交流，能者指教……

^\_^。

---

在开始介绍野火 stm32 开发板之前，野火首先尝试回答几个初学者的问题，这些问题来源于网上各大论坛和 QQ 交流群。

问：为什么要学习 stm32

答： ST 在 2007 年拿 ARM 公司的 cortex-M3 的内核开发出了 stm32，在 2008 年开始登陆中国内陆，到目前为止仅仅几年的时间就受到了那么多工程师的青睐，这是有原因的。

首先，stm32 功耗低，性能强劲，价格便宜(**这点老板最喜欢了^\_^**)，它是基于 ARM 公司的最新内核(ARMV7)的一款单片机，而 ARM9、ARM11 是老一代的 ARMV6



---

内核，目前 ARMV7 内核有三个系列，A、R、M 系列，其中 A 系列的用于高端的智能手机和平板电脑，像摩托罗拉的 **Android** 里程碑 2 就是 cortex-a8 内核，里程碑 3 则是 cortex-a9 双内核，还有 HTC 用的高通处理器也是基于这个内核的，R 系列则用于军工产品，几乎不在消费市场出现，M 系列主打中低端的控制领域，就是目前我们学习的 51、AVR、PIC 等占领的市场。学会了 M3，向 A 系列进军的日子还会远吗？

第二是 stm32 的价格优势，目前 64pin 的才卖到 8 快钱，芯片到了这个价可以说是白菜价了，但它的功能远比比它价格贵的 AVR 强 N 倍，且 ST 为我们开发者提供了最底层的函数库，封装了全部寄存器的操作，可以让我们不管底层寄存器的操作，我们只有调用库函数就可以轻松地完成我们的应用程序开发，这在一个时间非常紧迫的产品开发中起到的作用是非常非常大的。

所以，在未来的中低端的领域，M3 一定会大放异彩。控制芯片它可以不是 stm32，因为用 M3 内核生产 SOC 的不止 ST 一家公司，但我想说它一定是 M3 内核的。在这里野火跟大家分享句话(在未来中低端控制领域): **未来之芯，卓我 M3。**

问：我想学习的是嵌入式、是 ARM、是 LINUX，而不是像 51 那样的单片机。

答：在这里野火要告诉大家，其实 stm32 就是 ARM 的一种，内核还是 ARM 公司的最新内核 ARMV7，它的兄弟像 cortexA8 是应用在高端的智能手机的，而 M3 的定位是中低端的市场，而 ARM7、ARM9、ARM11 这些都还是老一代的内核，M3 的功能虽没有 ARM9 和 ARM11 强劲，但是绝对可以超过 ARM7。因为 M3 定位的是中低端的控制领域，所以就不能像 ARM9 和 ARM11 那样跑 LINUX 这样庞大的操作系统，但是在工业控制中还有一个操作系统 **u/COS-II**，M3 的出现，简直就是为 u/COS-II 而诞生的，M3 凭借出色的性能，内部含实时机制和保护机制，这些正是 u/COS-II 应用于工业控制、航天航空和汽车控制领域中所强烈要求的，这两者结合起来简直就是无敌。

**u/COS-II** 还配有华丽的界面 **UCGUI**，可用于做完美的人机交互界面，这在一些企业是非常受欢迎的。

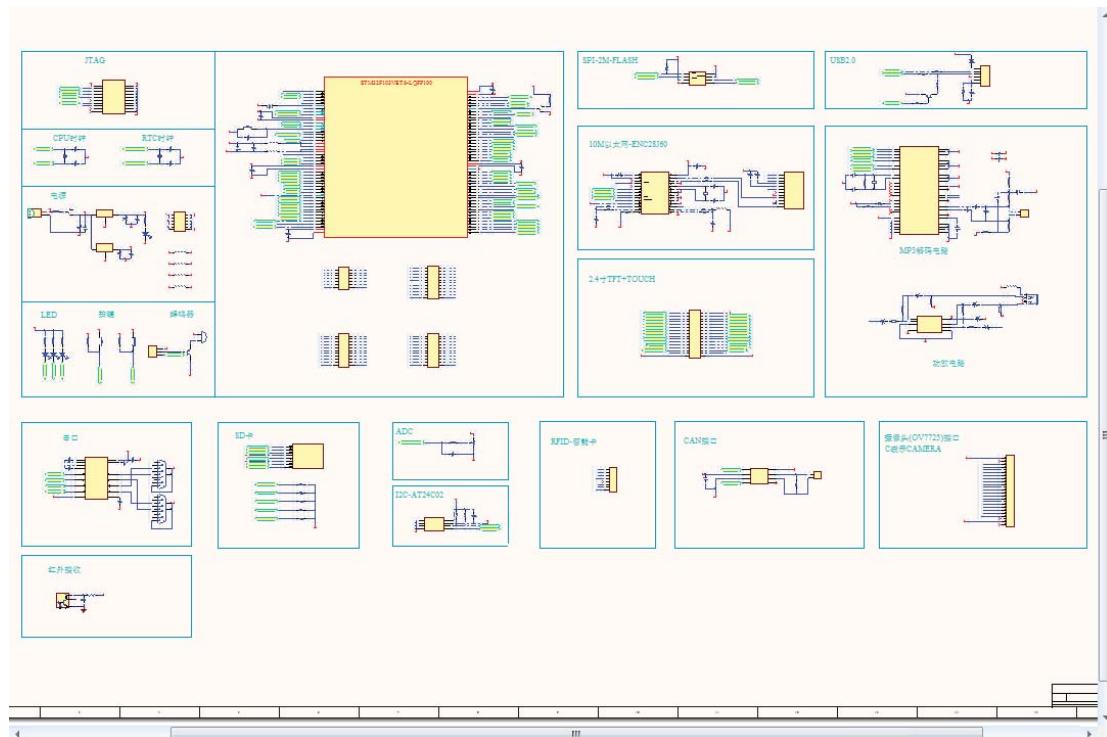
假如你只是学过 51、AVR 这些简单的单片机，又没丰富的项目经历，又没写过大型的代码，但是你又想学习嵌入式，学习 LINUX 的话，那么 stm32 是一个不错的选择，它可以帮助您跨越学习嵌入式的障碍，再配合 **u/COS-II** 来学习的话那效果定会更不一样，将来您入手 LINUX 的话定然不会迷茫不堪，不知从哪里下手^\_^。



下面野火为大家相信介绍下这块开发板。

### 硬件篇->

野火 stm32 开发板原理图(模块化的设计，繁而不乱):



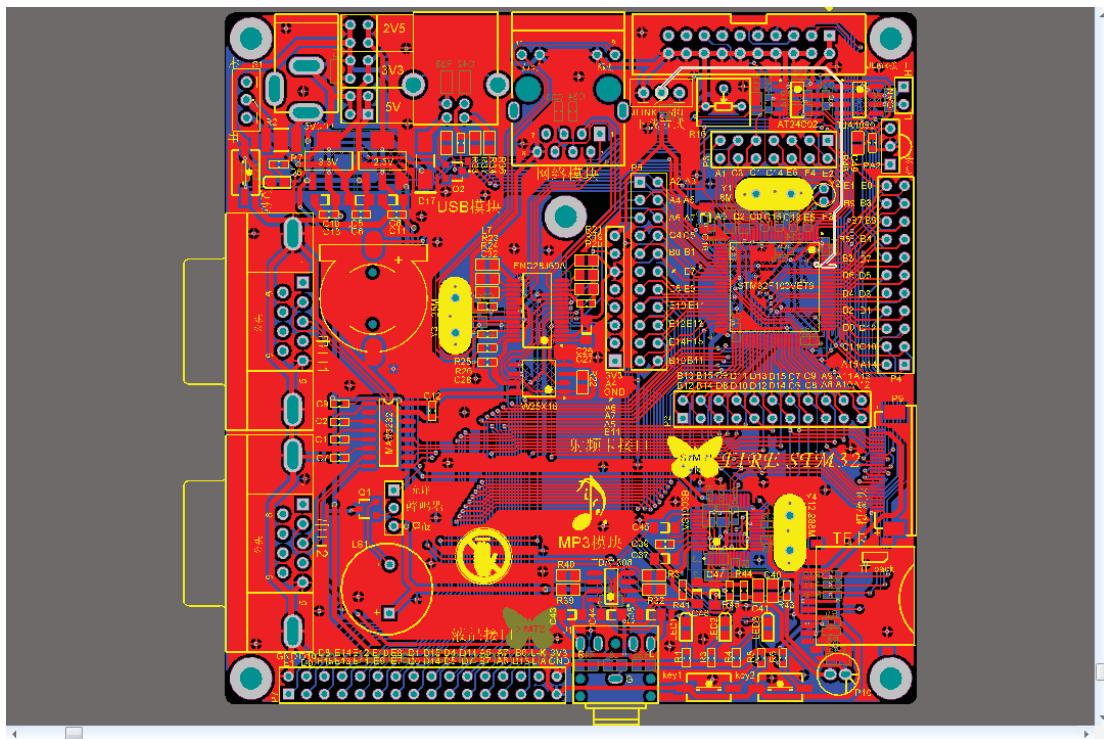
整板 **PCB** 全手工布线，力排 **EMS**(电磁兼容)的干扰，例如：不同的模块之间进行包地 / 割地处理、整板敷铜，**USB** 信号线差分走线，晶振下不走线还要做包地处理、摄像头的时钟线要等长、退耦电容要紧靠芯片等，这些都要考虑到，这样板子运行起来才能更稳定，更流畅。

麻雀虽小，但五脏俱全，整版尺寸为 10\*10，绝对 **mini**。在淘宝上的 **stm32** 开发板像野火这样集成了这么多功能模块又做的这么 **mini** 的绝无二加，野火之所以能做到是因为电阻电容都换成了 **0805** 的封装，有些还是 **0603** 的封装，电解电容淘汰了老掉牙的铝电解电容，全都换成了胆电容，不仅体积小，性能还更高，但缺点是价格贵了好几倍。对野火的 **PCB** 布线能力有了更高的要求，要考虑的地方越多，尽管如此，野火一直在努力，希望做得更好，为的是给大家呈现一个更完美的学习平台。

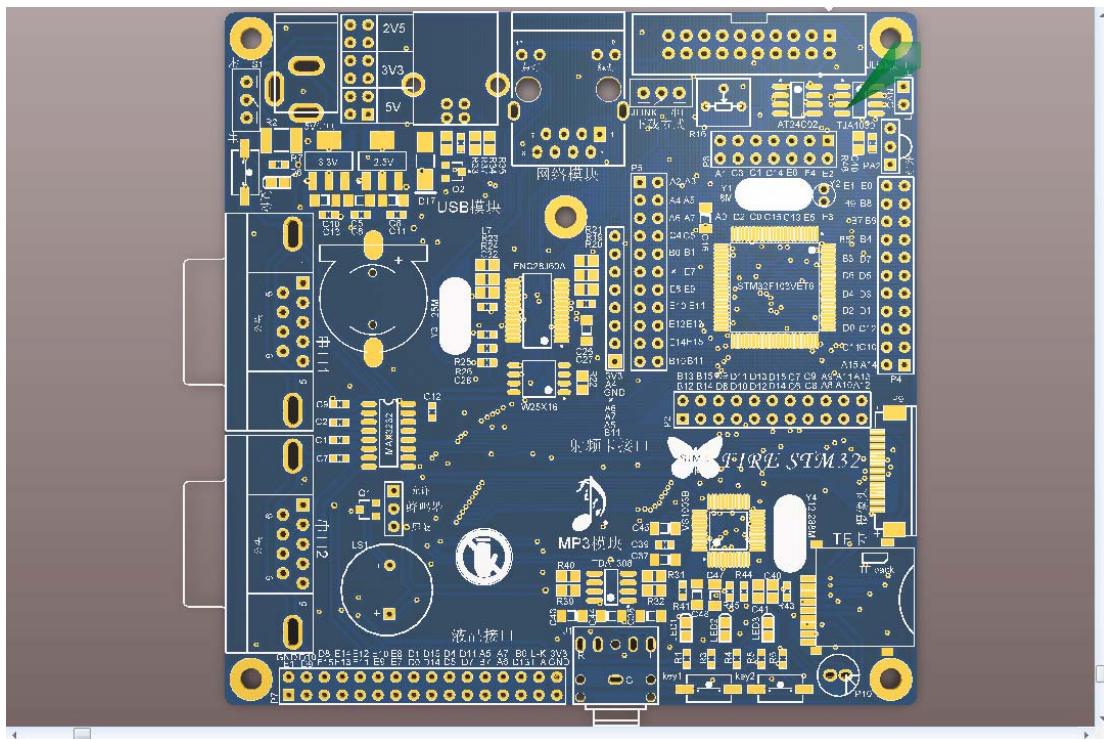


野火 stm32 开发板淘宝官方专卖: <http://firestm32.taobao.com>

野火 stm32 开发板 2D 正面图(10cm\*10cm):

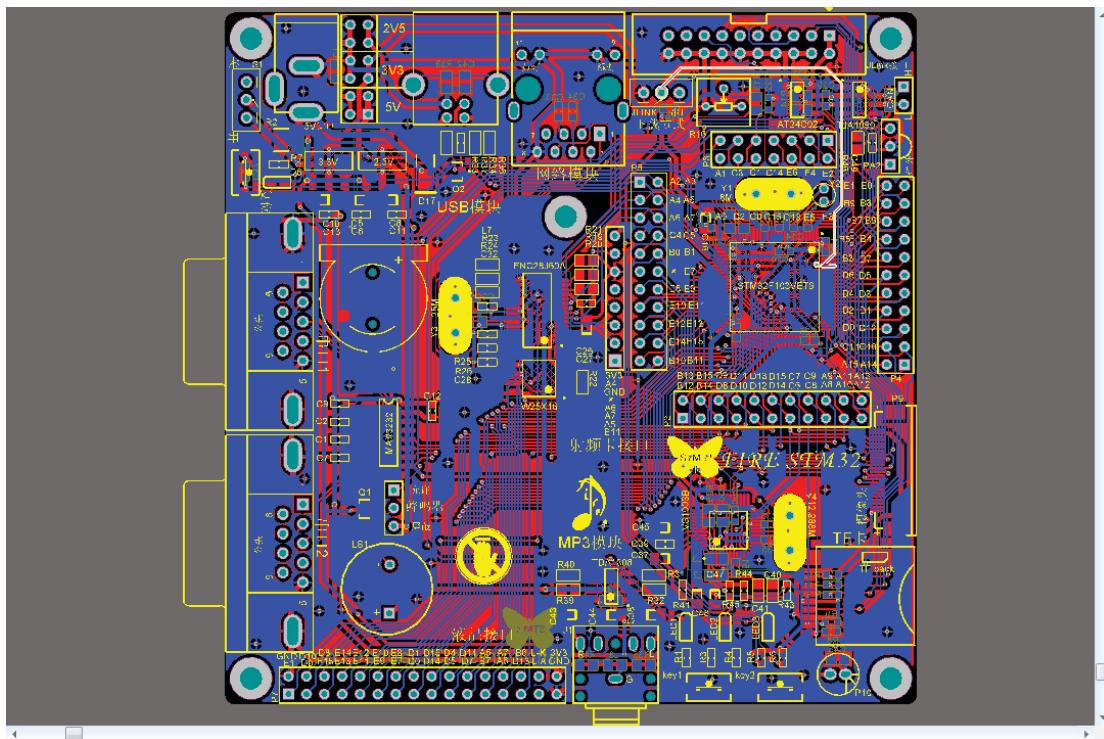


野火 stm32 开发板 3D 正面图(10cm\*10cm):

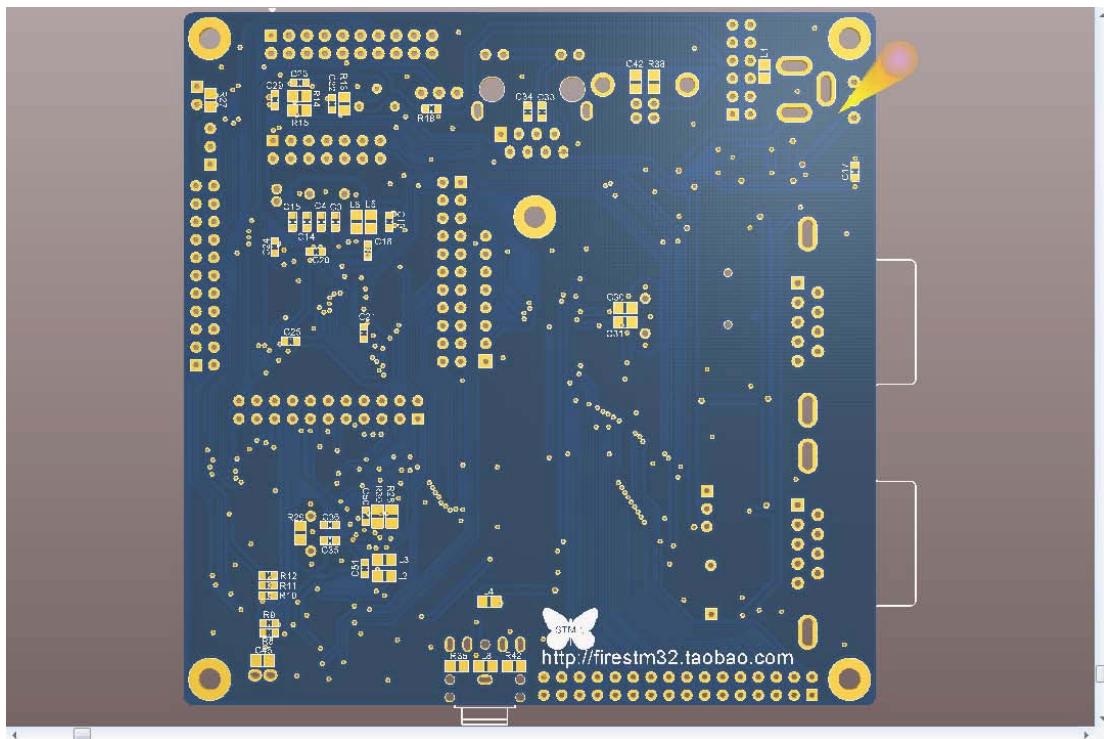




野火 stm32 开发板 2D 背面图(10cm\*10cm):



野火 stm32 开发板 3D 背面图(10cm\*10cm):





---

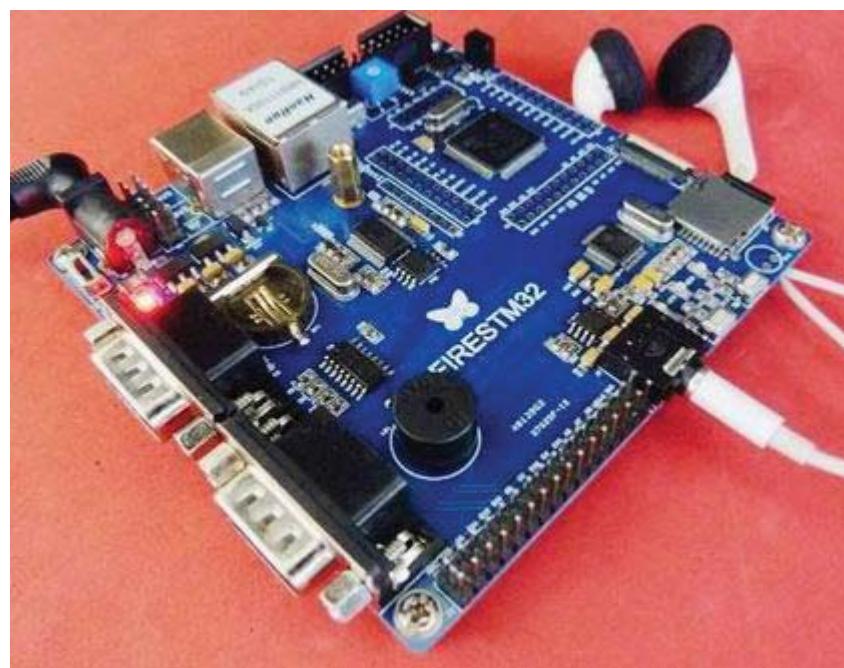
### 野火 stm32 开发板硬件一览:

- 1、主 CPUstm32f103vet6 一个, 64KRAM, 512FLASH
- 2、一个 DC5V 供电接口
- 3、一个电源开关
- 4、一个 300ma 过流自恢复保险丝
- 5、两个 5V、两个 3V3、两个 2V5、六个 GND 电源接口
- 6、一个系统复位开关
- 7、一个 RTC 电池座+3V 锂电池
- 8、两个串口, 串口 1 和串口 2
- 9、一个蜂鸣器
- 10、一个 TFT 液晶接口, 带触摸
- 11、一个 USB2.0
- 12、一个 10M 以太网( ENC28J60 )
- 13、一个 JTAG 下载接口
- 14、一个 JTAG 和串口下载选择开关
- 15、一个滑动电位器, 用于 ADC 实验
- 16、一个 CAN
- 17、一个红外接收头 HS0038B
- 18、一个 EEPROM ( AT24C02 )
- 19、一个 2M flash ( W25X16 )
- 20、一个 RFID 智能卡接口
- 21、一个 MP3
- 22、一个摄像头接口
- 23、一个 MicroSD 卡( 带自弹式卡套 )
- 24、4 个定位铜柱
- 25、所有的 I/O 均引出(2.54 pin)

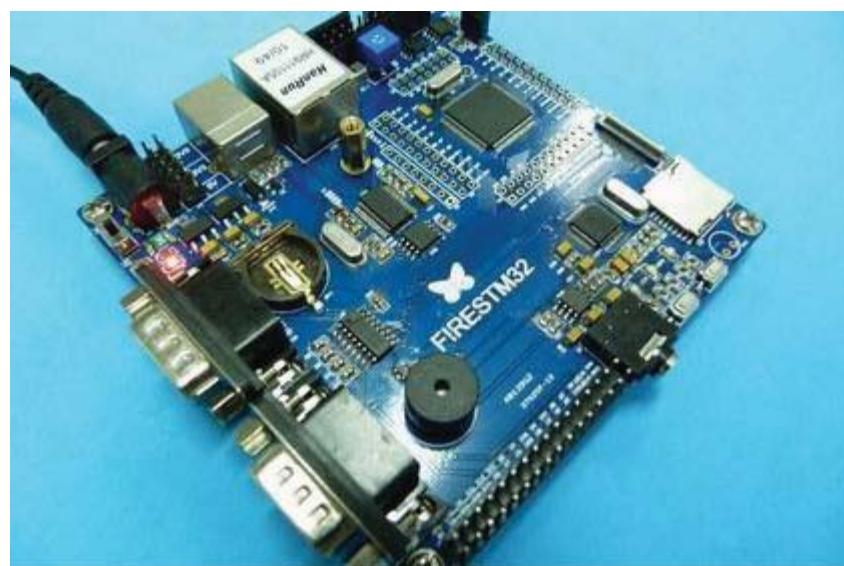


## 野火 stm32 开发板图片欣赏

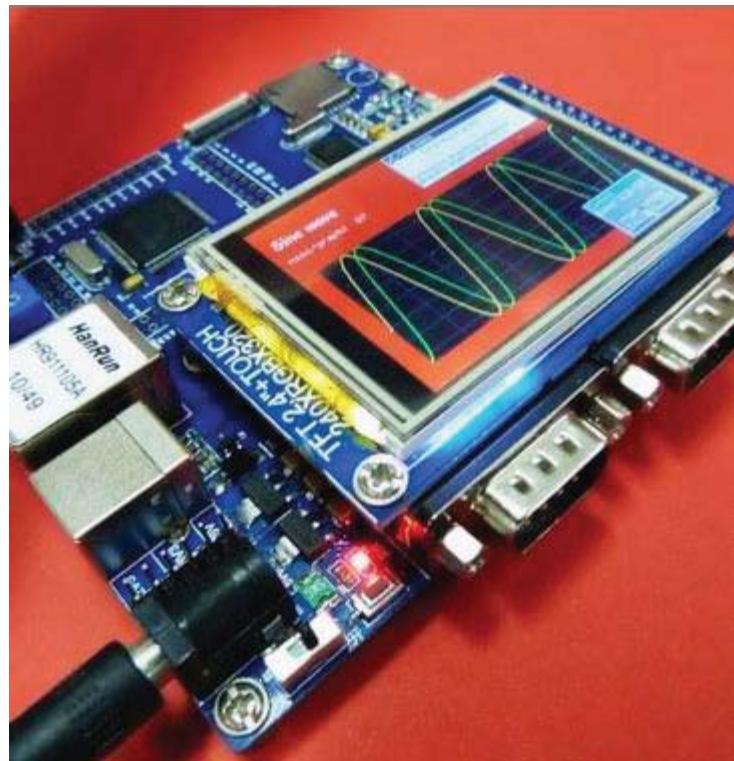
野火 stm32 单板( 板载了上面的全部资源 )



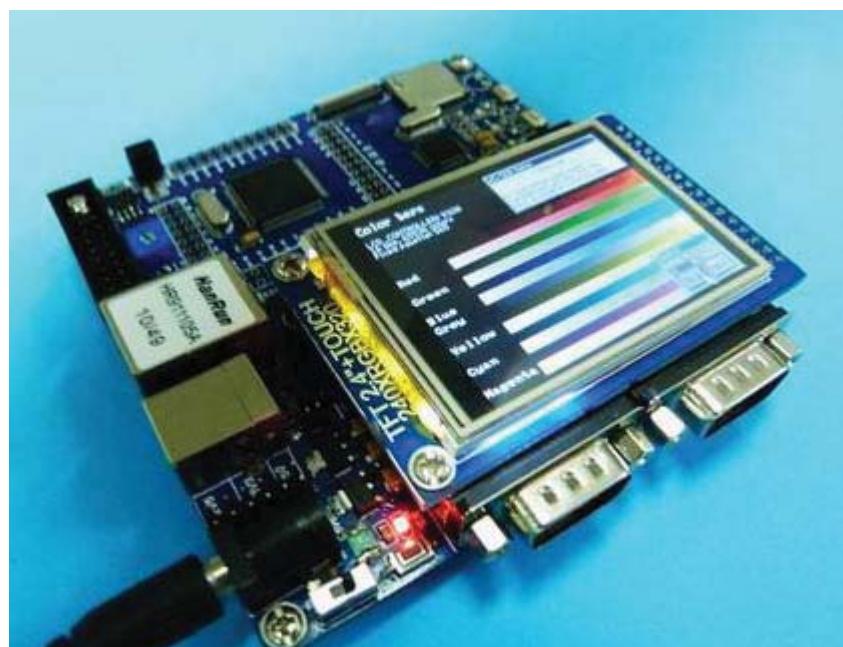
板载了 MP3, VS1003 硬解码+立体耳机功放 TDA1308, 音质堪比电脑。



整板尺寸为 10cm\*10cm, 绝对 mini.....



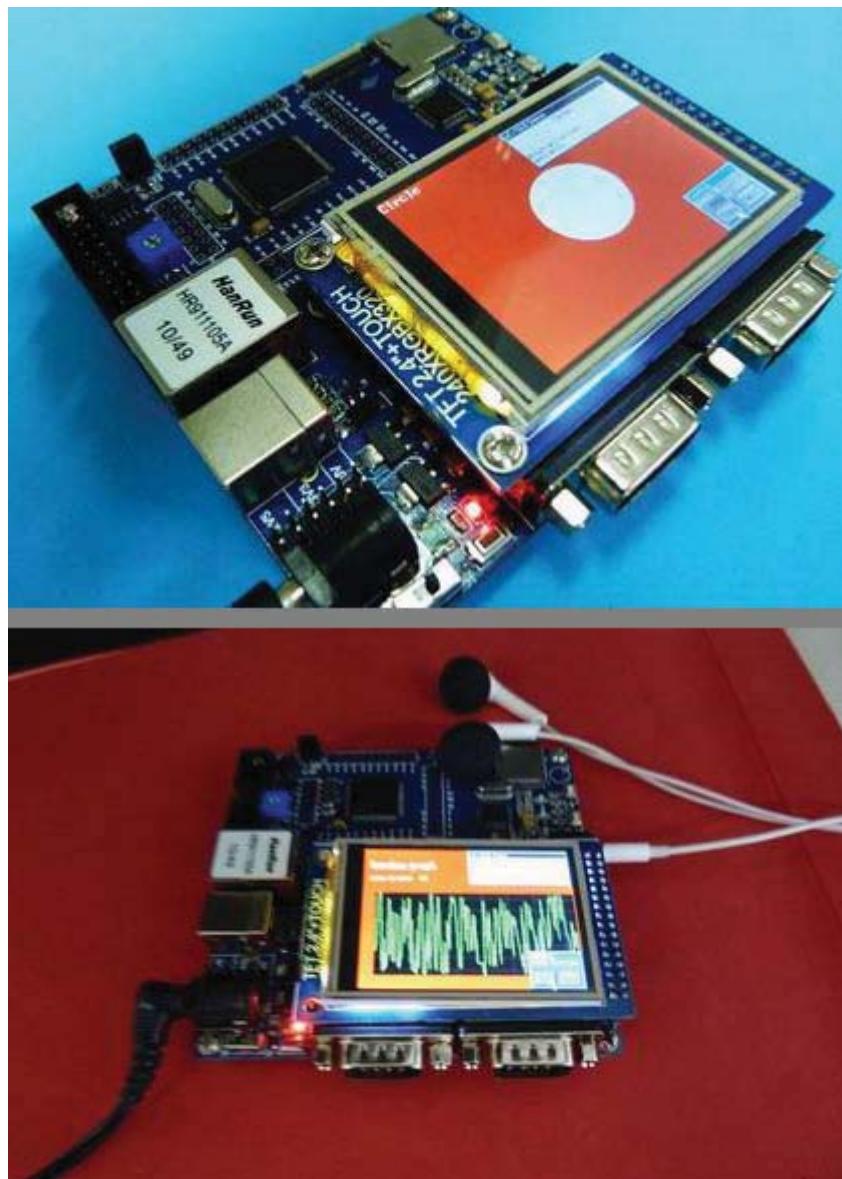
野火 stm32 在运行 u/CGUI2.9( 正弦波 )



野火 stm32 在运行 u/CGUI2.9( 彩条 )



野火 stm32 开发板淘宝官方专卖: <http://firestm32.taobao.com>





野火 **stm32** 板载了彩色摄像头 **OV7725** (可用于车载视频, 用于倒车),  
还有截图功能, 图像可保存在 **MicroSD** 卡中, 可以说是一个简易的照相机  
啦^\_^……, 配 **stm32** 驱动代码和详细 **PDF** 教程。



野火 **stm32** 除了板载了摄像头接口外, 还板载了 **RFID** 智能卡接口,  
该智能卡可应用于校园卡、公交卡、小区热水充值、个人名片等。  
配 **stm32** 驱动代码和详细 **PDF** 教程。



**ENC28J60 10M 以太网模块。**野火 stm32 开发板中板载了这个模块，这里是独立模块的图片。可应用于智能家居，远程控制，工业车间的数据交互，可局域网和跨网使用。配 stm32 驱动代码(目前 demo 只限于局域网，跨网应用的 demo 正在研究)和详细 PDF 教程。



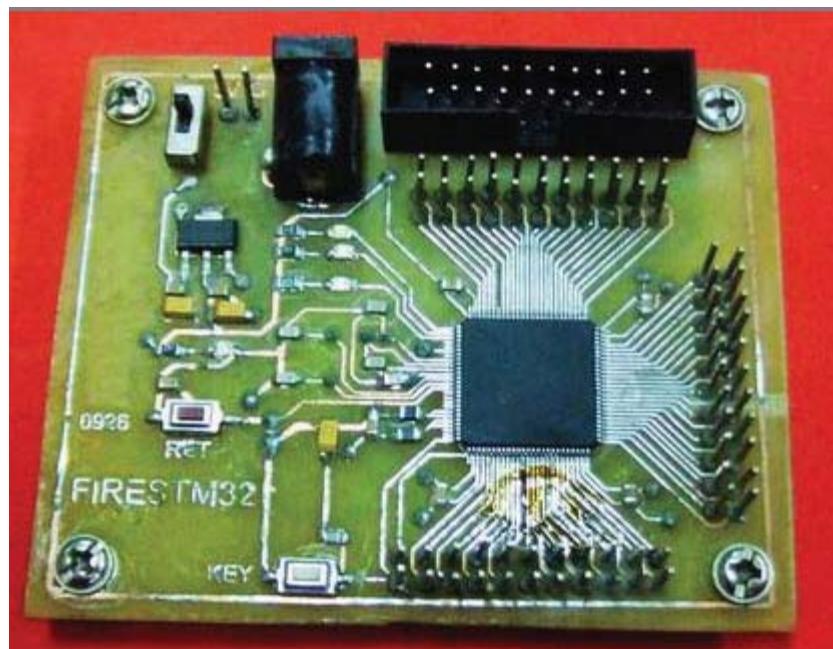
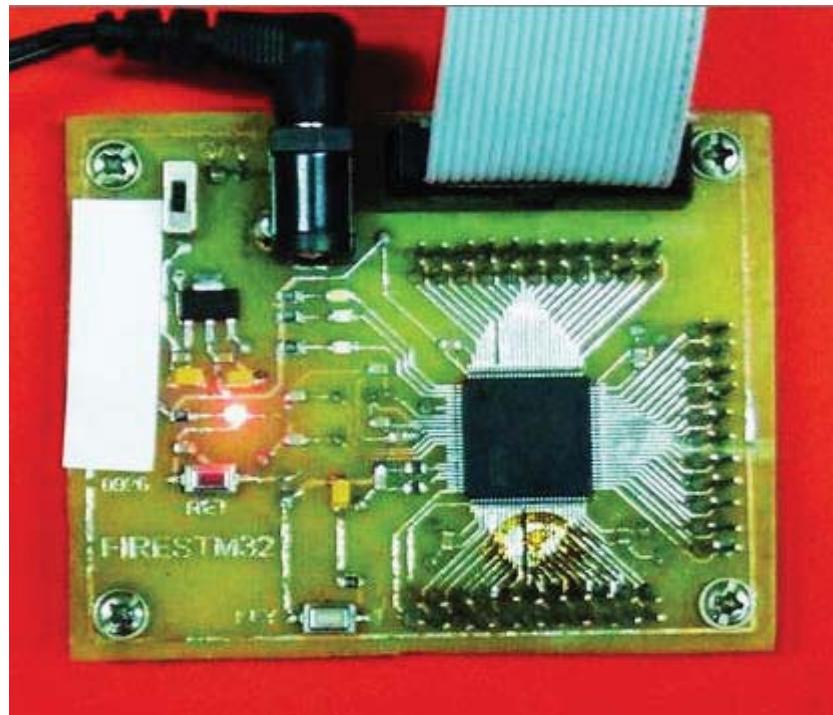
下面是野火调试的时候做的手工作品，望大家别见笑^\_^

野火 **stm32** 最小系统，纯手工：信号线 **10mil**（工厂标准），插件焊盘

大小为 **59.095mil**（工厂标准），电源为 **20mil**，过孔（当然不能是工厂  
标准^\_^）。



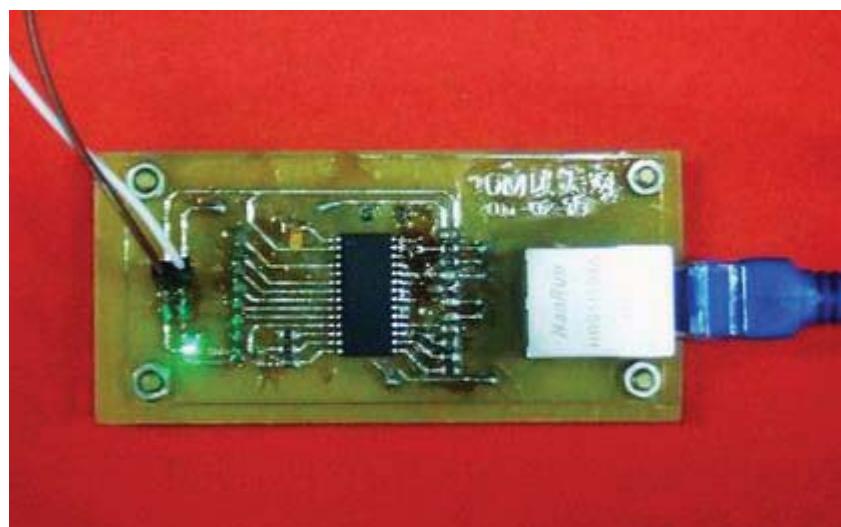
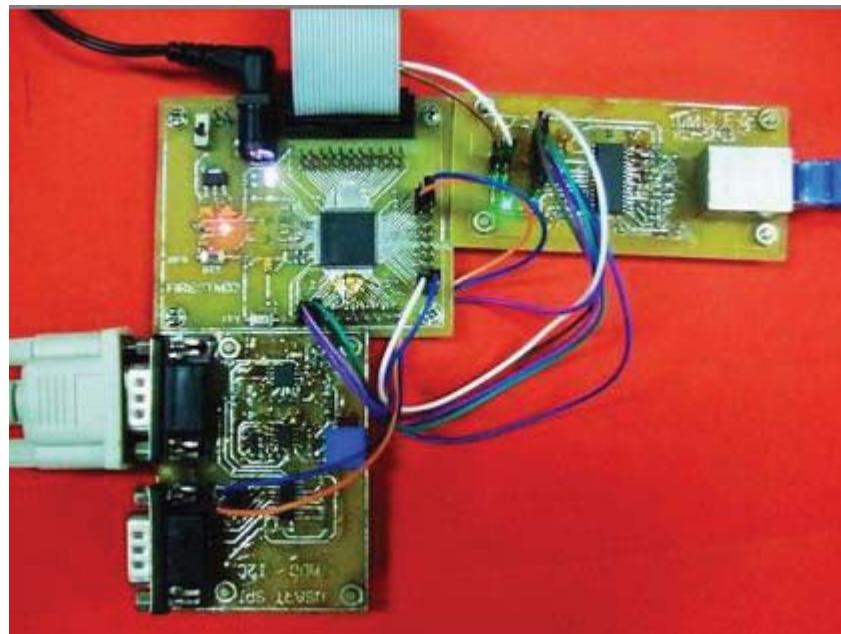
野火 stm32 开发板淘宝官方专卖: <http://firestm32.taobao.com>



纯手工: 野火 stm32 最小系统+ENC28G60 10M 以太网+MAX3232

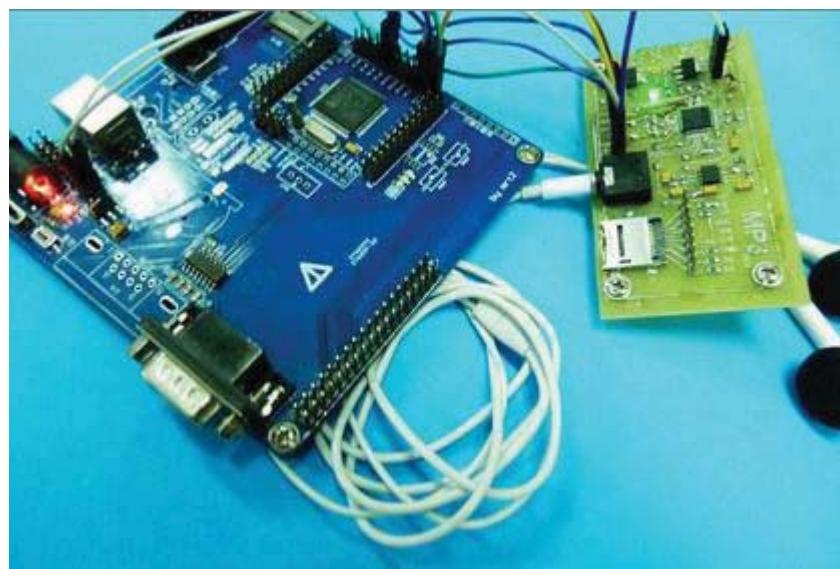
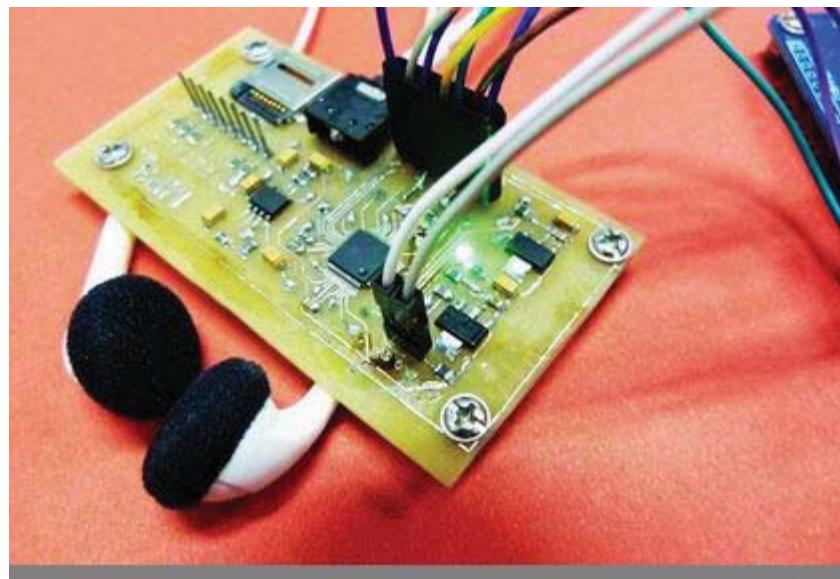


野火 stm32 开发板淘宝官方专卖: <http://firestm32.taobao.com>





野火纯手工 MP3，VS1003 硬解码+立体耳机功放 TDA1308，音质堪比电脑。





## 软件篇->

野火 stm32 为大家精选了 31 个实验例程，每个实验例程均配有详细的 pdf 教程，让您一板在手，学习无忧。当然也会不断地升级软硬件，跟大家一起分享学习 stm32 的乐趣。野火秉承的原则是：新手交流，能者指教^\_^……

下面是整板的软件清单：（难度系数随着序号递增）

- 01、**工程模板**（野火所有的 demo 都是基于此模板的）
- 02、**LED**（流水灯）
- 03、**串口 1**（实现了 C 语言中的 printf() 函数，可方便打印调试信息）
- 04、**串口 2**（实现了 C 语言中的 printf() 函数，可方便打印调试信息）
- 05、**SysTick**（系统滴答定时器，可实现 us/ms 级延时，也可测软件运行时间）
- 06、**key**（中断 / 查询模式）
- 07、**蜂鸣器**
- 08、**芯片 ID**
- 09、**CRC 循环冗余校验**
- 10、**ADC**（采集外部电压，串口打印转换值）
- 11、**I2C**（EEPROM-AT24C02）
- 12、**SPI**（2M U 盘 W25X16）
- 13、**RTC**（电子时钟，实现了 时:分:秒）
- 14、**日历**（实现了 闰/平年、24 节气、阴历/阳历、日期，时:分:秒）
- 15、**TIM3 定时**（us/ms 级定时）
- 16、**TIM3 输出 4 路不同占空比的 PWM 信号**
- 17、**CAN**（LoopBack 模式）
- 18、**红外接收**
- 19、**MicroSD 卡**（单纯读 block，没有跑文件系统）
- 20、**MicroSD 卡+文件系统**（文件系统版本为 R0.07C，通信速率 1M/S）
- 21、**MP3**（SPI1 接口，歌曲存放在 MicroSD 卡中）



- 22、**MP3** (SPI2 接口, 歌曲存放在 MicroSD 卡中)
- 23、**USB 读取 MicroSD 卡**( 模拟 U 盘 )
- 24、**LCD 显示** (LCD 显示中英文、数字、图片(数组形式)、BMP 图片 )
- 25、**LCD 触摸** ( 调色板实验, 根据笔的取色写不同颜色的字 )
- 26、**10M 以太网 ENC28J60** ( 实现了 WEB 服务器来控制开发板中的 LED )
- 27、**RFID 智能卡实验** ( 可用于公交, 校园卡, 热水充值, 上班名片等 )
- 28、**计算器** ( LCD 触摸操作, 涉及指针、数组、结构体、链表、动态内存分配)
- 29、**摄像头**(LCD 触摸操作, 图像在 LCD 中实时显示, 图像可保持 MicroSD 卡)
- 30、**ucOS II V2.86 /uCGUI 3.9**
- 31、**自动翻译笔** ( 类似于文曲星, 可将英文翻译成中文, 要摄像头的支持 )
- 

正在开发的例程:

- 1、2.4G ( 24L01, 传输速率 2M/S ) 无线模块
- 2、蓝牙模块
- 3、GSM 短信模块
- 4、GPS 模块
- 5、重力感应模块( MA7455 )
- 6、FPGA + 摄像头 + STM32 的超级套餐
- 7、BRF24L01+ 摄像头 + STM32 的无线图像传输系统。
- 8、u/COS-II 操作系统例程。
- .....

野火 stm32 开发板全部例程均基于 ST 的库编写, 编程语言为 C, 代码结构清晰, 注释明了, 代码模块化程度高, 可移植性好, 代码风格均参考了 LINUX 的代码风格, 具有非常好的可读性。

下面是一些代码的截图:

野火在这里以 MP3 这个例子来说明。



最底层的操作芯片外设的函数我们将它封装中一个 c 文件中，完全分离了与应用层程序的混叠。在模块 c 文件的开头均有详细的描述，并列出了 I/O 硬件连接，这可以让您在不看原理图的情况下就知道 I/O 是如何连接的^\_^，如下截图。

```
001 //***** (C) COPYRIGHT 2011 野火嵌入式开发工作室*****
002 * 文件名 : vs1003.c
003 * 描述 : vs1003B(音频解码芯片)应用函数库
004 *
005 * 实验平台: 野火STM32开发板
006 * 硬件连接: -----
007 * | PB13-SPI2_SCK : VS1003B-SCLK |
008 * | PB14-SPI2_MISO: VS1003B-SO |
009 * | PB15-SPI2_MOSI: VS1003B-SI |
010 * | PB12-SPI2_NSS : VS1003B-XCS |
011 * | PB11 : VS1003B-XRET |
012 * | PC6 : VS1003B-XDCS |
013 * | PC7 : VS1003B-DREQ |
014 *
015 * 库版本 : ST3.0.0
016 *
017 * 作者 : fire QQ: 313303034
018 * 博客 : firestm32.blog.chinaunix.net
019 *****/
020
021 #include "vs1003.h"
022 #include "SysTick.h" /* 延时函数头文件 */
023
```

代码注释清晰完整，格式使用 tab 缩进，跨越行数多的循环均有注释，如下截图：

```
082 PutChinese_strings22(80, 200, id3v1.artist, 0, 1);
083 }
084 else
085 { //有些MP3文件没有ID3V1 tag,只有ID3V2 tag
086     res = f_lseek(&fsrc, 0);
087     Read_ID3V2(&fsrc, &id3v2);
088
089     printf( "\r\n 曲目 : %s \r\n", id3v2.title );
090     printf( "\r\n 艺术家 : %s \r\n", id3v2.artist );
091
092     PutChinese_strings22(10, 200, "曲目", 0, 1);
093     PutChinese_strings22(80, 200, id3v2.title, 0, 1);
094
095     PutChinese_strings22(10, 180, "艺术家", 0, 1);
096     PutChinese_strings22(80, 180, id3v2.artist, 0, 1);
097 }
098
099 /* 使文件指针 fsrc 重新指向文件头, 因为在调用Read_ID3V1/Read_I
100 res = f_open( &fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ )
101 //res = f_lseek(&fsrc, 0);
102
103     br = 1; /* br 为全局变量 */
104     TXDCS_SET( 0 ); /* 选择vs1003的数据接口 */
105 /* ----- 一曲开始 -----*/
106     USART1_printf( USART1, "\r\n 开始播放 \r\n" );
107     for ();}
108     {
109         res = f_read( &fsrc, buffer, sizeof(buffer), &br );
110         if ( res == 0 )
```



所有模块的测试均在主函数中完成，这不仅使得 main 函数简洁易懂，还保持了模块函数的完整性，更易于移植，main 函数的头部均有详细的描述信息来说明这个例程实现了什么功能，并且在野火的 pdf 教程里面还有实验现象的截图。代码截图如下：

```
001 // **** (C) COPYRIGHT 2011 野火嵌入式开发工作室 ****
002 * 文件名 : main.c
003 * 描述 : 将MicroSD卡(以文件系统FATFS访问)里面的mp3文件
004 *          通过vs1003B解码, 然后将解
005 *          码后的数据送到功放TDA1308后通过耳机播放出来。
006 * 实验平台: 野火STM32开发板
007 * 库版本 : ST3.0.0
008 *
009 * 作者 : fire QQ: 313303034
010 * 博客 : firestm32.blog.chinaunix.net
011 ****
012 #include "stm32f10x.h"
013 #include "mp3play.h"
014
015 SD_CardInfo SDCardInfo; // 存放SD卡的信息
016
017
018 /* 函数原型声明 */
019 void NVIC_Configuration(void);
020 SD_Error SD_USER_Init(void);
021
022
023 /*
024 * 函数名: main
025 * 描述 : 主函数
026 * 输入 : 无
027 * 输出 : 无
028 */
029 int main(void)
030 {
031     /* 配置系统时钟为 72M */
032     SystemInit();
033
034     /* 配置SysTick 为10us中断一次 */
035     SysTick_Init();
036
037     /* 配置串口1 115200 8-N-1 */
038     USART1_Config();
039
040     /* LED初始化, 用于调试 */
041     LED_GPIO_Config();
042
043     /* 2.4TFT 初始化 */
044     LCD_Init();
045
046     /* 文件系统初始化-----汉字字库保存在sd卡中, 并将盘符设置为0 */
047     sd_fs_init();
```



```
048 Set_direction(0);           /* 设置LCD的扫描方向, 这里为垂直扫描 */
049 LCD_CLEAR(0,0,240,320);
050
051 USART1_printf( USART1, "\r\n 这是一个MP3测试例程 !\r\n" );
052
053 /* MP3硬件I/O初始化 */
054 VS1003_SPI_Init();
055
056 /* MP3就绪, 准备播放, 在vs1003.c实现 */
057 MP3_Start();
058
059 LED1(ON);
060
061 /* 播放SD卡(FATFS)里面的音频文件 */
062 MP3_Play();
063
064 while (1)
065 {
066 }
067 } /* end of main */
```

关于野火 stm32 开发板野火就给大家介绍到这，有什么不明白的地方或者建议可以给野火发邮件 [firestm32@foxmail.com](mailto:firestm32@foxmail.com)，也可以加野火的 QQ: 313303034，野火非常乐意与大家分享学习 stm32 的乐趣^\_^.....

野火在这里祝大家学习愉快^\_^.....



## 野火 stm32 开发板实验代码简介

作者	fire
E-Mail	firestm32@foxmail.com
QQ	313303034
博客	firestm32.blog.chinaunix.net
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

野火 stm32 开发板实验代码一览 ( 难度系数随序号递增 )				
实验序号	实验名称	所用片上资源	实验描述	PDF 实验手册
01	工程模板	无 ( 需要库源码 ST3.0.0 )	利用 ST 官方库新建一个工程文件 , 库的版本为 ST3.0.0 , 后面的实验例程均基于这个工程模板。	有
02	LED	rcc、gpio	操作 GPIO 点亮 LED。	有
03	串口 1	rcc、gpio、usart	利用串口 1 实现 printf 函数 , 方便于打印调试信息。	有
04	串口 2	rcc、gpio、usart	利用串口 2 实现 printf 函数 , 方便于打印调试信息。	无 ( 可参考串口 1 )
05	SysTick	rcc、gpio	利用系统定时器实现精确的延时 (us+ms)。	有
06	Key1	rcc、gpio	利用传统的查询 I/O 电平的状态来判断按键的状态 , 从而来实现相应的服务程序。	无 ( 很简单 )
07	Key2	rcc、gpio、exti、misc	采用 I/O 中断工作方式(即 EXTI)来检测按键模式 ( 按下或者没按下 ) , 在中断函数中处理用户程序。	有
08	蜂鸣器	rcc、gpio、usart	利用 GPIO 使蜂鸣器发出不同频率的声音 , 同时串口打印出相应的调试信息。	有
09	芯片 ID	rcc、gpio、usart、flash	读取 flash 里面的 96 位芯片 ID , 并通过串口 1 在电脑的超级终端打印出来。	有
10	9-CRC ( 循环冗余校验 )	rcc、gpio、usart、crc	利用 STM32 内部的 CRC 校验电路产生 CRC 校验码 , 并通过串口在电脑的超级终端打印出来。	有
11	ADC	rcc、gpio、usart、adc、dma、flash	利用 STM32 内部的 AD 将外部的电压 ( 外部电压可调 ) 转换为数字量 , 并通过串口在电脑的超级终端打印出来。	有



12	EEPROM-AT24C02	rcc、gpio、uart、i2c	以 i2c 总线方式操作 EEPROM，并用串口将调试信息打印在电脑的超级终端上。	有
13	2MU 盘-W25X16	rcc、gpio、uart、spi	以 spi 总线方式操作外部 flash，并用串口将调试信息打印在电脑的超级终端上。	有
14	RTC ( 日历 )	rcc、gpio、uart、pwr、 bckp、rtc	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。 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。	有
15	超强日历 ( 含 24 节气和闰平年 )	rcc、gpio、uart、pwr、 bckp、rtc	超强的日历，支持农历，24 节气几乎所有日历的功能 日历时间以 1970 年为元年，用 32bit 的时间寄存器可以运行到 2100 年左右。这个例程是在 RTC 的基础上实现的。	无 ( 可参考 RTC )
16	TIM3 定时 ( ms )	rcc、gpio、uart、misc、 tim	利用内部定时器 TIM3 产生 ms 级延时，从而来控制 LED 的闪烁频率。	无 ( 很简单 )
17	TIM3 产生 4 路 PWM 信号	rcc、gpio、tim、flash	利用内部定时器 TIM3 产生 4 路不同占空比的方波。	有
18	CAN ( LoopBack )	rcc、gpio、uart、can	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	有
19	红外接收	正在调试		
20	MicroSD 卡	rcc、gpio、uart、sdio、 misc、dma	MicroSD 卡(SDIO 模式)测试实验，并将测试信息通过串口 1 在电脑的超级终端上打印出来。只是单纯的操作 block，没有跑文件系统。	有
21	MicroSD 卡+文件系统	rcc、gpio、uart、sdio、 misc、dma (+文件系统源码，文件系统为 FATFS R0.07C )	MicroSD 卡文件系统 FATFS R0.07C 测试，包括在 sd 卡里面创建文件，读取 sd 卡里面的文件。	有
22	<b>MP3+SD 卡 +FATFS+ ( SPI1 接口 )</b>	rcc、gpio、uart、sdio、 misc、dma、misc、spi	将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码，然后将解码后的数据送到功放 TDA1308 后通过耳机播放出来。	有
23	<b>MP3+SD 卡 +FATFS+ ( SPI2 接口 )</b>	rcc、gpio、uart、sdio、 misc、dma、misc、spi	将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码，然后将解码后的数据送到功放 TDA1308 后通过耳机播放出来。	有
24	<b>MP3+SD 卡 +FATFS+LCD 显示 ( SPI2 接口 )</b>	rcc、gpio、uart、sdio、 misc、dma、misc、spi <b>( LCD )</b>	将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码，然后将解码后的数据送到功放 DA1308 后通过耳机播放出来。同时在 LCD 上显示曲目和艺术家。	有 ( 超详细 )
25	USB 读取 MicroSD 卡 ( 模拟 U 盘 )	rcc、gpio、uart、sdio、 misc、dma、flash	这是一个 USB Mass Storage 实验，用 USB 线连接 PC 机与开发板，在电脑上	有



			就可以像操作普通 U 盘那样来操作开发板中的 MicroSD 卡。	
26	LCD 显示 ( 中英文、BMP 图片 , 需 SD 卡中的字库支持 )	rcc、gpio、uart、fsmc、misc、dma、sdio	LCD 显示 MicroSD 卡里面的 BMP 图片 , 字库可通过宏定义保存在 MicroSD 卡中或者 内部 flash 中。	有
27	LCD 触摸彩色画板	rcc、gpio、uart、fsmc、misc、dma、sdio、spi	LCD 触摸实验 , 开机先校验屏幕 , 画笔通过触摸板取色。	有
28	以太网 ENC28J60	rcc、gpio、uart、spi	在浏览器上创建一个 web 服务器 , 通过 web 里面的命令来控制开发板上的 LED 的亮灭。	有
30	RFID 智能卡	rcc、gpio、uart、tim、misc、flash	通过串口 1 对智能卡 RC522 进行发送卡号、读卡、写卡、修改密码操作等。具体可应用于 : 校园卡、公交卡、热水充值等。	有
31	计算器	gpio、rcc、uart、fsmc、misc、systick、exti、dma、sdio	计算器测试程序 , 实现功能有 : 加减乘除的混合运算 , 支持括号的符号运算 , 支持浮点运算 , 计算结果为科学计数法。涉及到 C 知识点为 : 指针、数组、结构体、动态内存分配。要是把这个例程完全搞懂的话 , 你的 C 语言也就大概入门了。	有
32	摄像头	gpio、rcc、uart、fsmc、misc、systick、exti、tim、sdio、dma	将摄像头采集的图像信息在 LCD 中显示出来 , 具备截图功能 , 图片保存在 MicroSD 卡中 , 图片格式为 BMP 。	有
33	自动翻译笔		类似于文曲星 , 将英文翻译为中文 , 需摄像头的支持。	涉及到专利问题 , 暂时不开源 , 见谅

### 野火 stm32 开发板系统例程 ( 全部基于 μC/OS-II+μCGUI )

系统例程将长期更新

34	μC/OS-II V2.86 +μCGUI 3.9	( TFT+TPUCH )	μC/OS-II V2.86 uCGUI 3.9 测试程序 , 在液晶上不断地显示图片 , 可暂停显示 , 可切换到下一幅图显示。	正在写
35	μC/OS-II + LED		μC/OS-II 开启一个 LED 任务 , 让 LED 以一定的频率闪烁	正在写
36	μCGUI 3.9 + LED		通过触摸屏来控制 LED 的亮灭	正在写
37	μCGUI 3.9 + 万年历		在 LCD 上显示时间 , 时间可调。	正在写
38	.....			



下面是野火嵌入式开发工作室正在开发的项目

- 1、2.4G ( 24L01 , 传输速率 2M/S ) 无线模块
- 2、重力感应模块( MMA7455 )
- 3、FPGA + 摄像头 + STM32 的超级套餐
- 4、BRF24L01+ 摄像头 + STM32 的无线图像传输系统。
- 5、μC/OS-II + UCGUI 操作系统例程 ( 例程 + 详细 pdf 教程 ) 。
- .....



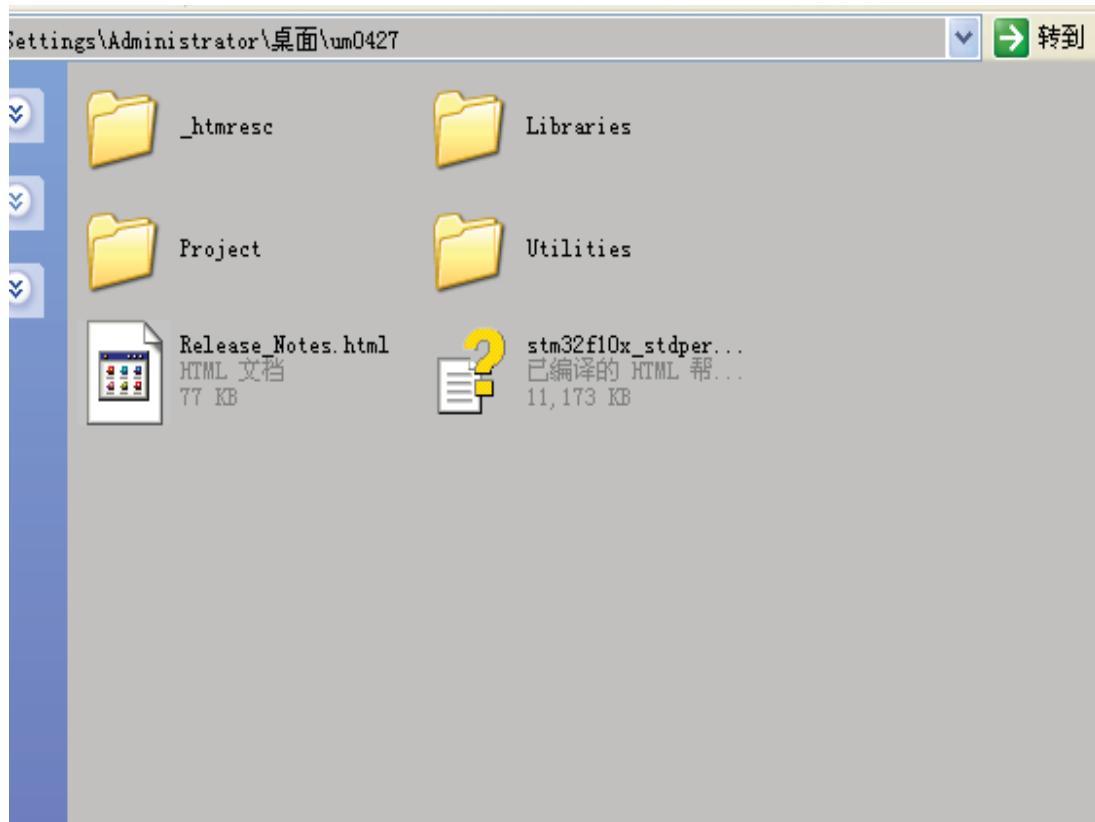
## 详解用 STM32 官方库来开发自己的程序

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

这里用的库是 ST3.0.0 版本，解压缩后的文件名为 um0427，库的源代码可从 ST 的官网下载到。

<http://www.st.com/mcu/familiesdocs-110.html>(网址可能会有变动)

首先，让我们来分析下这个库的目录结构，如下图所示：

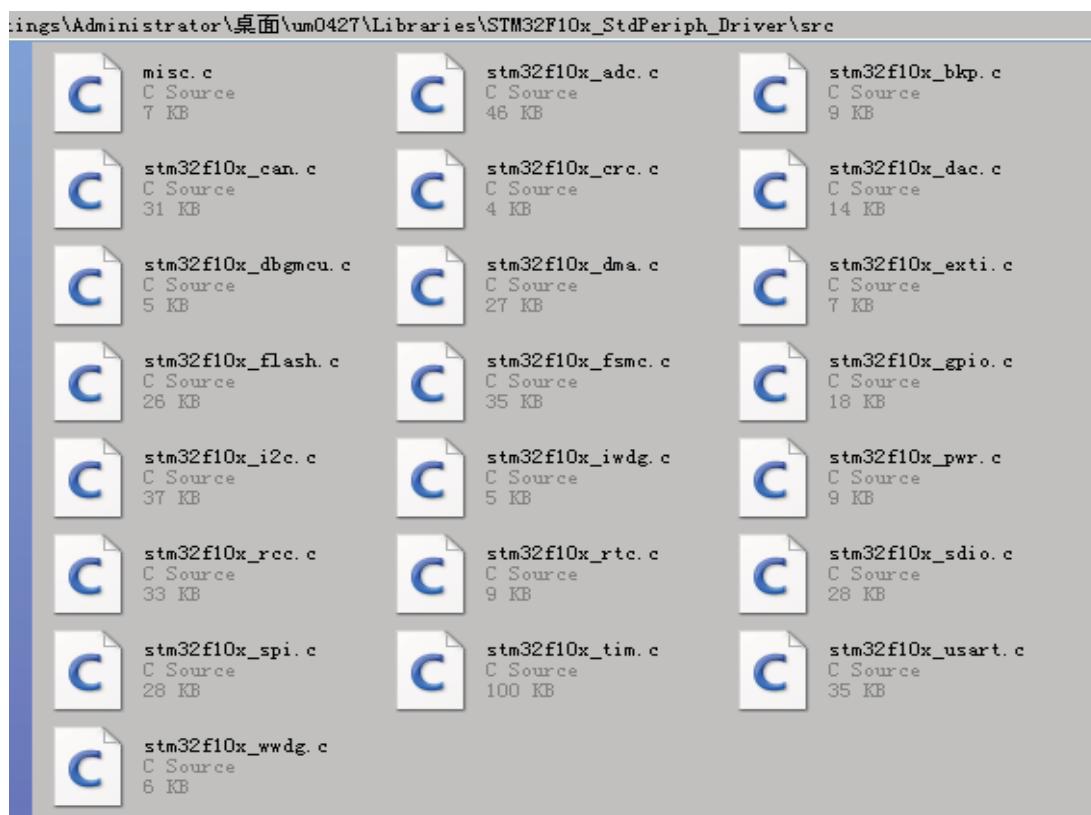




**Libraries** 文件夹下是驱动库的源代码跟启动文件。**Project** 文件夹下是用驱动库写的例子跟一个工程模板。还有一个已经编译好的 **HTML** 文件，主要讲的是如何使用驱动库来编写自己的应用程序，说得形象一点，这个 **HTML** 就是告诉我们：ST 公司已经为你写好了每个外设的驱动了，想知道如何运用这些例子就来向我求救吧。既然 ST 给我们提供的美味大餐（驱动源码）就在眼前，我又何必去找品尝大餐的方法呢，还不如直接一头直接扎进大餐中，大吃一顿再说（直接阅读库的源码）。但当我们吃的有点呛口的时候回去找下方法还是很好的。其他三个文件作用不大，我们可以不用管它。

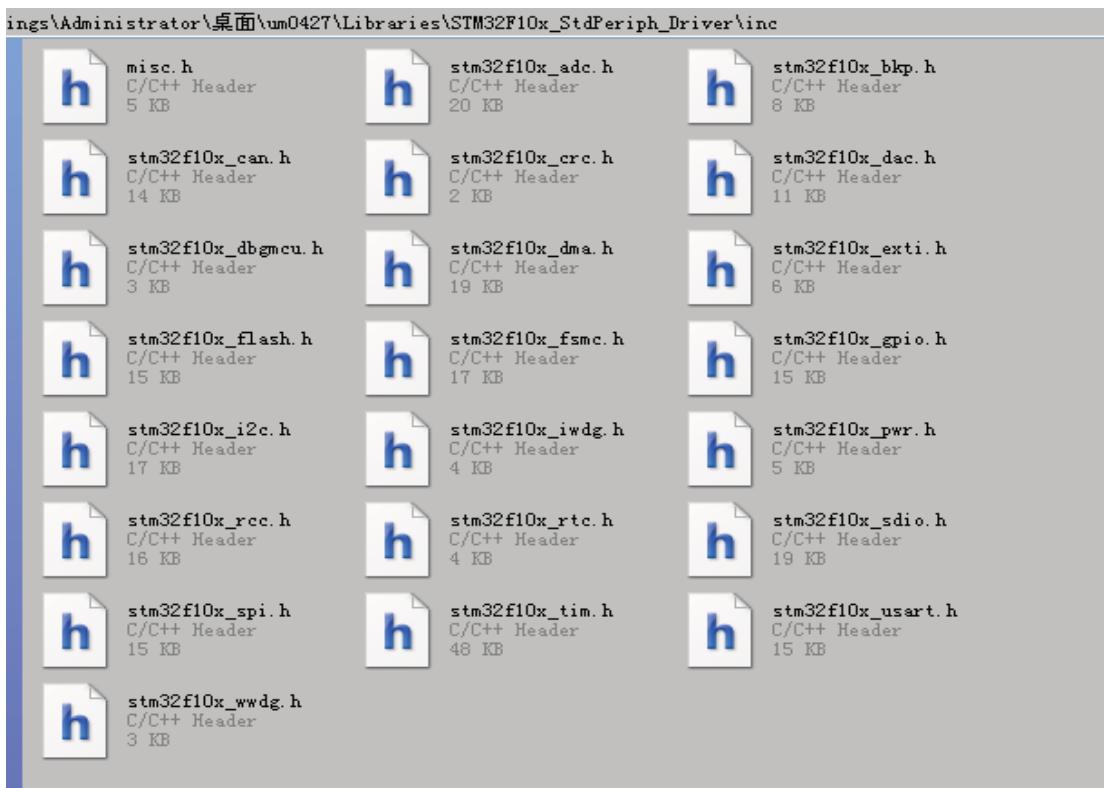
接下来我们重点来分析下 **Libraries** 文件夹下的内容。

**Libraries\STM32F10x\_StdPeriph\_Driver** 文件夹下有 **inc** (**include** 的缩写) 跟 **src** (**source** 的简写) 这两个文件，**src** 里面是每个片上外设的驱动程序，这些外设当中很多是芯片制造商在 **Cortex-M3** 核上加进去的，**Cortex-M3** 核自带的外设是通用的，放在 **CMSIS** 文件夹下。如下图所示：

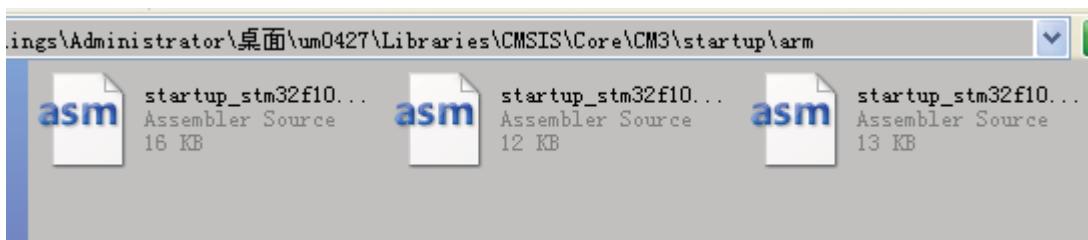




**Libraries\STM32F10x\_StdPeriph\_Driver\inc** 文件夹下是每个驱动文件对应的头文件。当我们的应用程序需要用到某个外设的驱动程序的话只需将它的头文件包含进我们的应用程序即可。

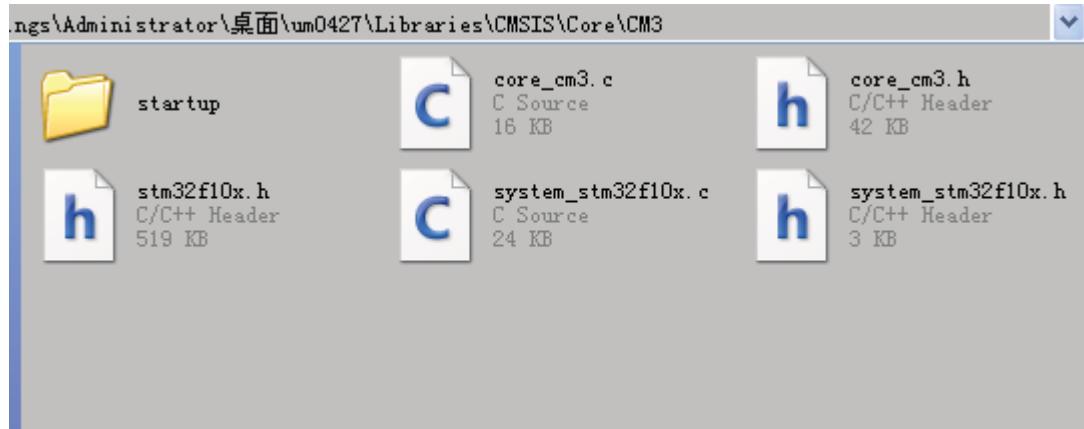


**Libraries\CMSIS\Core\CM3\startup\arm** 文件夹下是三个汇编编写的系统启动文件，分别对应于小（LD）中（MD）大（HD）容量 Flash 的单片机，在我们新建工程的时候需要将它包含到我们的工程中去。启动文件是任何处理器在上点复位之后最先运行的一段汇编程序。启动文件的作用是：1、初始化堆栈指针 SP，2、初始化程序计数器指针 PC，3、设置异常向量表的入口地址，4、配置外部 SRAM 作为数据存储器（这个由用户配置，一般的开发板可没有外部 SRAM），5、设置 C 库的分支入口 \_\_main（最终用来调用 main 函数）。如若要详细了解启动文件的详细过程可参考如下网友的文章：<http://blog.ednchina.com/likee/138130/message.aspx>。搞不明白也没太大关系，我们新建工程的时候只要将它包含进来就可以了。





Libraries\CMSIS\Core\CM3 文件夹下除了放有 startup 启动文件外，还有这几个文件 core\_cm3.c 、 core\_cm3.h ， stm32f10x.h 、 system\_stm32f10x.c , system\_stm32f10x.h 。



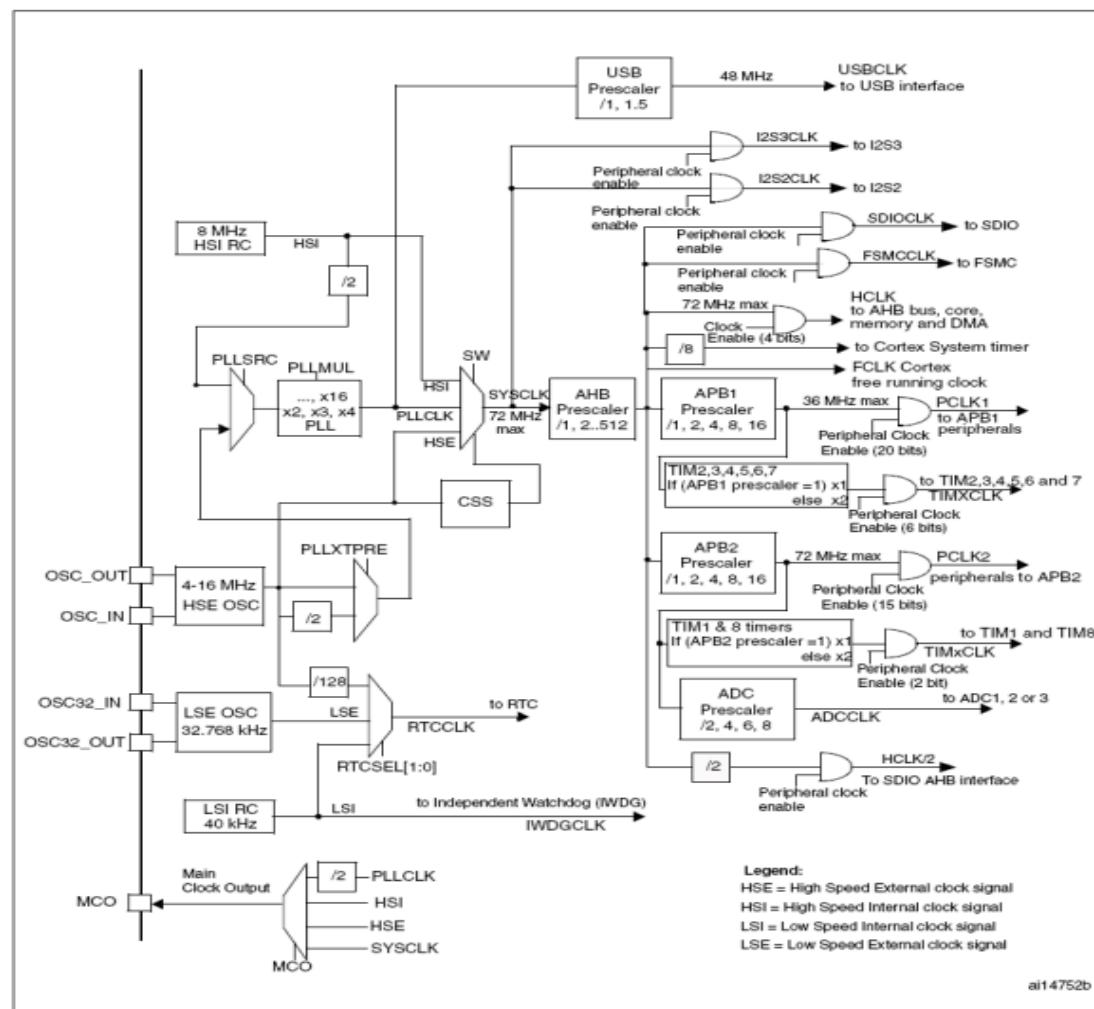
**core\_cm3.c** 是 CMSIS Cortex-M3 核外设接入层的源文件，在所有符合 CMSIS 标准的 Cortex-M3 核系列单片机都适用，独立于芯片制造商，由 ARM 公司提供。它的作用是为那些采用 Cortex-M3 核设计 SOC 的芯片商设计的芯片外设提供一个进入 M3 内核的接口。至于这些功能是怎样用源码实现的，我们可以不用管它，我们只需把这个文件加进我们的工程文件即可。该文件还定义了一些与编译器相关的符号。在文件中包含了 **stdin.h** 这个头文件，这是一个 ANSIC C 文件，是独立于处理器之外的，就像我们熟知的 C 语言头文件 **stdio.h** 文件一样。位于 RVMDK 这个软件的安装目录下，主要作用是提供一些类型定义，如：

```
036     /* exact-width signed integer types */
037     typedef signed char int8_t;
038     typedef signed short int int16_t;
039     typedef signed int int32_t;
040     typedef signed __int64 int64_t;
041
042     /* exact-width unsigned integer types */
043     typedef unsigned char uint8_t;
044     typedef unsigned short int uint16_t;
045     typedef unsigned int uint32_t;
046     typedef unsigned __int64 uint64_t;
047
```



**core\_cm3.c** 跟启动文件一样都是底层文件，都由 ARM 公司提供，遵守 CMSIS 标准，即所有 CM3 芯片的库都带有这个文件，这样软件在不同的 CM3 器件的移植工作就得以化简。**core\_cm3.c** 里面包含了一些跟编译器相关的信息，如：RealView Compiler, ICC Compiler, GNU Compiler。**core\_cm3.h** 这个文件实现了 CM3 内核里面的 NVIC 和 SysTick 这两个资源的所有功能，NVIC 是嵌套向量中断控制器，SysTick 是 CM3 内核里的一个简单的定时器，其时钟由外部时钟源（STCLK）或内核时钟（FCLK）来提供，一般我们在编程的时候选择 FCLK 作为它的运行时钟，FCLK 由 SYSCLK 八分频得到。NVIC 的寄存器是以存储器映射的方式来访问的，所以 **core\_cm3.h** 头文件中也包含了寄存器的存储映射和一些宏声明。

**system\_stm32f10x.c** 的性质跟 **core\_cm3.c** 是一样的，也是由 ARM 公司提供，遵守 CMSIS 标准。该文件的功能是根据 HSE 或者 HSI 设置系统时钟和总线时钟（AHB、APB1、APB2 总线）。系统时钟可以由 HSI 单独提供，也可以让 HSI 二分频之后经过 PLL（锁相环）提供，也可以由 HSE 经过 PLL 之后获得。具体可参考 STM32 的时钟树：（该图截自 STM32 参考手册中文版 47 页）。





注意: **system\_stm32f10x.c** 文件只是设置了系统时钟和总线时钟, 至于那些外设的时钟是在 **rcc.c** 这个文件中实现的。因为各个 SOC 厂商在 CM3 内核的基础上添加的外设工作的速率是不一样的, 有的是高速外设(时钟经过 APB2 高速总线获得), 有的为低速外设(时钟经过 APB1 低速总线获得), 所以这一功能的实现放在芯片驱动文件夹 **src/rcc.c** 下。这篇文档分析的是 ST (意法半导体) 公司的 STM32, 对于其他公司的芯片可能不太一样, 但是, 不论是哪个厂商, 系统时钟都是由 ARM 公司实现, 为的是软件移植的方便。然后, 再从系统时钟里面分频来得到各个外设的时钟。  
**system\_stm32f10x.c** 在实现系统时钟的时候要用到 PLL (锁相环), 这就需要操作寄存器, 寄存器都是以存储器映射的方式来访问的, 所以该文件中包含了 **stm32f10x.h** 这个头文件。

```
24
25 #include "stm32f10x.h"
26
```

**stm32f10x.h** 这个文件非常重要, 是一个非常底层的文件。以前我在学习其他单片机的时候只是纯粹地操作寄存器, 但不知道寄存器到底是个神马东东。但在这里我们可以学习到寄存器, 也就是内存, 我们访问寄存器也就是在访问内存。它先把内存地址先强制类型转换为指针, 再把该指针实现为一个宏。下面我们通过一个例子来说明这一过程 (并不非常准确, 寄存器也是随便取的, 只是为了帮助理解)。

```
#define PORTA *((volatile unsigned long *) (0x40000000))
PORTA = 0xFFFF; // 往端口A赋值
```

以前我们在写应用程序时, 只是这样写

```
PORTA = 0xFFFF; // 往端口A赋值
```

但为什么这样写就可以操作内存呢, 其实系统为我们提供的头文件已经帮我们做了很多工作了。假如换种方式来操作寄存器, 如下。这就再简单不过了。

```
*((volatile unsigned long *) (0x40000000)) = 0xFFFF;
```



学过 C 语言的朋友都明白：这是将一个十六进制值通过强制类型转换为一个指针，再对这个指针进行解引用操作。

假如我们在操作寄存器的时候都是用这种方法的话会有什么缺点没。当然有：1、地址容易写错 2、我们需要查大量的手册来确定哪个地址对应哪个寄存器 3、看起来还不好看，且容易造成编程的错误，效率低，影响开发进度。所以处理器厂商都会将对内存的操作封装成一个宏，即我们通常说的寄存器，并且把这些实现封装成一个系统文件，包含在相应的开发环境中。这样，我们在开发自己的应用程序的时候只要将这个文件包含进来就可以了。

**stm32f10x.h** 就是实现这么个功能的文件，并且它把这种功能发挥得更好。下面我们通过分析其源码来看看 STM32 是怎么样来实现其存储器映射的。要是有 STM32 存储系统的知识的话，对这些源码理解的会更快。关于 STM32 存储器系统大家可以参考《ARM Cortex-M3 权威指南》中文版 宋岩译 第五章：存储器系统，重点看 83 页。要是把这章的内容都理解了，那么 **stm32f10x.h** 这个文件的源码也就理解的差不多了。

下面我们以 RCC 来说明这一功能的实现过程。RCC 是英文 Reset and Clock Control 的简写，是复位和时钟控制，是 STM32 片上的一个外设，这个外设里面包含了很多寄存器。ST 库把这些寄存器都定义在一个结构体里面，寄存器的长度都是无符号整型 32 位的，如下所示：

```
0686 //**
0687 * @brief Reset and Clock Control
0688 */
0689
0690 typedef struct
0691 {
0692     __IO uint32_t CR;
0693     __IO uint32_t CFGR;
0694     __IO uint32_t CIR;
0695     __IO uint32_t APB2RSTR;
0696     __IO uint32_t APB1RSTR;
0697     __IO uint32_t AHBENR;
0698     __IO uint32_t APB2ENR;
0699     __IO uint32_t APB1ENR;
0700     __IO uint32_t BDCR;
0701     __IO uint32_t CSR;
0702 } RCC_TypeDef;
```



```
0105 #define __I volatile const      /*!< defines 'read only' permissions */
0106 #define __O volatile          /*!< defines 'write only' permissions */
0107 #define __IO volatile         /*!< defines 'read / write' permissions */
```

`__IO` 在 `core_cm3.h` 这个头文件中定义, `uint32_t` 在 `stdin.h` 这个头文件定义。

`stm32f10x.h` 这个文件中包含了 `core_cm3.h` 和 `stdin.h` 这两个头文件。如下所示:

```
0196 #include "core_cm3.h"
0197 #include "system_stm32f10x.h"
0198 #include <stdint.h>
```

从上可知 `RCC_TypeDef` 这个结构体中声明了很多的变量, 这些变量的名字跟 `RCC` 里面的寄存器的名字是对应的。关于 `RCC` 寄存器的具体内容可以参考《STM32 参考手册中文》。但是, 对变量的操作怎么转化为对寄存器的操作呢? 接着往下看.....

STM32 是 32 位的 CPU, 其寻址空间可以达到 4GB (2 的 32 次方等于 4GB), 即从 0X00000000 到 0XFFFFFF。这 4GB 的空间被分成了许多功能模块, 其中地址 0X40000000 到 0X5FFFFFF 这 512M 空间用于片上外设, 即片上外设的寄存器区。详细内容可参考《ARM Cortex-M3 权威指南》中文版 宋岩译, 第 83 页。以下是代码实现 `RCC` 寄存器是如何映射的。

声明片上外设的所有寄存器的起始地址, 当要定义某些寄存器的地址的时候只需在这个地址上加上一定的偏移量即可。

```
0879 #define PERIPH_BASE ((uint32_t)0x40000000)
```

声明高速寄存器的起始地址, 因为外设要高速和低速之分, 这从它们的时钟频率上可以体现出来, `RCC` 就属于高速的外设。

```
0886 #define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
```

声明 `RCC` 寄存器的开始地址, 从这个地址开始的一段内存都是 `RCC` 的寄存器, 具体这段内存有多大, 由上面的 `RCC_TypeDef` 这个结构体的长度确定。



```
0943 #define RCC_BASE (AHBPERIPH_BASE + 0x1000)
```

将 RCC 声明为一个 RCC\_TypeDef \* 型的指针，用于指向 RCC 寄存器的起始地址。这样，我们就可以通过 RCC 这个指针来访问寄存器了。如：RCC->CR = 0xFFFFFFFF;

```
1017 #define RCC ((RCC_TypeDef *) RCC_BASE)
```

现在我们把上面这个例子的代码整理在一起，整体来看下。

```
043 typedef struct
044 {
045     __IO uint32_t CR;
046     __IO uint32_t CFGR;
047     __IO uint32_t CIR;
048     __IO uint32_t APB2RSTR;
049     __IO uint32_t APB1RSTR;
050     __IO uint32_t AHBENR;
051     __IO uint32_t APB2ENR;
052     __IO uint32_t APB1ENR;
053     __IO uint32_t BDCR;
054     __IO uint32_t CSR;
055 } RCC_TypeDef;
056
057 #define PERIPH_BASE ((uint32_t)0x40000000)
058 #define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
059 #define RCC_BASE (AHBPERIPH_BASE + 0x1000)
060 #define RCC ((RCC_TypeDef *) RCC_BASE)
061
062 RCC->CR = 0xFFFFFFFF; // 用户代码
```

STM32 有非常多的寄存器，这里只是以外设 RCC 来作为例子。注意，这些源码在 **stm32f10x.h** 头文件中并不是紧靠在一起的，而是每个功能模块都分开的，如全部寄存器的结构声明放在一起，外设存储器映射地址声明放在一起，外设的指针声明放在一起。我们这里把他们放在一起是为了好理解。具体情况可查看源码。**stm32f10x.h** 头文件还包含了寄存器的位声明，要知道 STM32 是具有非常强大的位处理功能的，具体这里不详述，可参考《ARM Cortex-M3 权威指南》第五章的 5.5 位带操作部分。



上面的 **RCC->CR = 0xFFFFFFFF;** 是我们以前学习单片机时直接操作寄存器的方法，但 ST 官方库为我们想到了一个更绝妙的方法，就是将写到寄存器里面的值都实现为一个宏，宏名即是实现什么功能的英文描述，达到了让人一看宏名就知道往这个寄存器里面写这个宏是实现什么功能，非常的方便。当我们往寄存器里面写一个值时，实际上是操作了寄存器的所有位的值，但更绝的是在这个库里面将实现每个位功能的值也实现为一个宏。这些功能的实现都包含在相应外设的头文件中，如 RCC 则为 `stm32f10x_rcc.h`, ADC 则为 `stm32f10x_adc.h`。有很多网友觉得这样很多，眼花缭乱，看不过来，干脆就不用这些宏，而是自个去查看数据手册，逐位查看寄存器每位的值，再将这些值写到寄存器里面去，就像 **RCC->CR = 0xFFFFFFFF;** 这也是网上流行的直接操作寄存器，而不使用库的方法。但请大家认真想想，是真的没有使用库吗？且这样做还要花费大量的精力，实在是不讨好，聪明的人都应该把好钢用到刀刃上。在每个外设里面的驱动文件里面则实现了该外设的所有功能，这些功能的实现大量运用了这些宏。不过也有很多朋友还是自个重新实现这些功能。但这样做也有缺点：1、加长了产品的开发时间，耗费了大量不必要的精力；2、重新实现这些功能就得自个去查阅数据手册，对数据手册的理解也就不一定能达到百分百的正确，除非你觉得你比开发这些芯片的工程师理解的好。所以，我们还是应该使用官方给的库，要站在巨人的肩膀上开发更好的应用程序。假如还是劈头盖脸的从最基础的部分做起，这在真正的产品开发中是要不得的。聪明的工程师都会善于利用别人搭好的舞台来唱自个的戏。大家不愿意用库，究其原因：一个可能是摆脱不了以前学习单片机的方法；另一个可能是没见过这么大的代码量（注：ST 的库文件还是比较庞大的），因为学单片机的朋友很少是像计算机那样搞纯软件的，动辄就上万行的代码，所以初次见到这么庞大的代码还是有点抗拒的，但要是你学过 LINUX 的话且看过其内核的源码的话，这库简直就是小菜一碟啦。当初我用 LINUX+QT+FFMPEG 在 ARM9 平台上做 MP4 项目的时候，要研究的代码比这个都不知要复杂多少倍。

下面我们简单举一个 GPIO 的例子来帮助我们理解上面的这段话。我们知道对 I/O 口的操作分为两种：一种是读数据，一种是写数据。但是 I/O 口数据的操作有总线操作和端口操作，总线操作即是在 16 个端口里面操作数据，而端口操作是在这 16 个端口里面的某一个端口操作数据（注：GPIOA 有 16 个端口，为 0~15）。以下源码来自 `stm32f10x_gpio.h` ,只是库里头的 I/O 所有功能函数的声明，它们的具体实现在 `stm32f10x_gpio.c` 这个文件中，这些功能函数的实现都是通过操作寄存器来实现的。当我们要用到时，只需在相应的数据手册下查看下要用到的寄存器即可，没必要重新

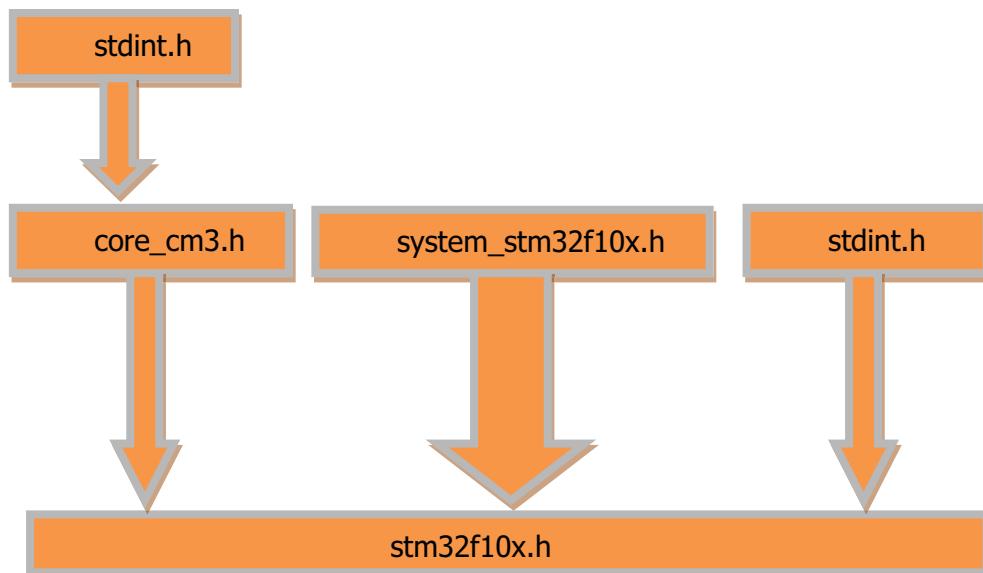


去自个实现。我们在应用程序中需要用到 I/O 时只需将 `stm32f10x_gpio.h` 这个头文件包含进来就可以了。

```
292 void GPIO_DeInit(GPIO_TypeDef* GPIOx);
293 void GPIO_AFIODeInit(void);
294 void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
295 void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
296 uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
297 uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
298 uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
299 uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
300 void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
301 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
302 void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal);
303 void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
304 void GPIO_PinLockConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
305 void GPIO_EventOutputConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
306 void GPIO_EventOutputCmd(FunctionalState NewState);
307 void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
308 void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
```

看到这里，假如你还是坚持要直接操作寄存器且要自个去重新实现功能函数而不使用库的话，那我这个文档没能说服你，我有罪，阿门！！！

最后，我们再来梳理下这个库里面的关键的头文件之间的关系。





## stdint.h

是 ANSIC C 头文件，位于 RVMDK 这个开发环境的安装目录，是开发环境自带的，其功能是提供一些数据类型的定义。

## core\_cm3.h 、 system\_stm32f10x.h

是由 ARM 公司为 SOC 厂商提供的基于 Cortex-M3 核的外设接口层，独立于芯片厂商符合 CMSIS 标准。CMSIS 标准是 ARM 公司联合其他 SOC 厂商制定的，所以基于 Cortex-M3 核的 SOC 都必须遵守，这样不管是哪个公司生产的芯片都可以用 ATM 公司提供的这几个文件，方便了软件上的开发。**这两个头文件分别对应一个 C 文件，我们在新建工程的时候需要把这两个 C 文件添加进来。还要将启动文件也加进来。**

## stm32f10x.h

实现了存储器映射和寄存器的声明，是一个非常重要的头文件。这个头文件包含了 core\_cm3.h、system\_stm32f10x.h、stdint.h 这三个头文件。

综上：在我们的应用程序中只需将 **stm32f10x.h** 这个头文件包含进来即可，这件就可以通过 **stm32f10x\_conf.h** 这个头文件方便的选择某些外设的驱动程序。这个配置头文件里面包含了每个外设驱动的头文件，如下图所示：

```
25 // * Includes -----  
26 // * Uncomment the line below to enable periph  
27 // * #include "stm32f10x_adc.h" */  
28 // * #include "stm32f10x_bkp.h" */  
29 // * #include "stm32f10x_can.h" */  
30 // * #include "stm32f10x_crc.h" */  
31 // * #include "stm32f10x_dac.h" */  
32 // * #include "stm32f10x_dbgmcu.h" */  
33 // * #include "stm32f10x_dma.h" */  
34 // * #include "stm32f10x_exti.h" */  
35 // * #include "stm32f10x_flash.h" */  
36 // * #include "stm32f10x_fsmc.h" */  
37 // *#include "stm32f10x_gpio.h"*/  
38 // * #include "stm32f10x_i2c.h" */  
39 // * #include "stm32f10x_iwdg.h" */  
40 // * #include "stm32f10x_pwr.h" */  
41 // *#include "stm32f10x_rcc.h"*/  
42 // * #include "stm32f10x_rtc.h" */  
43 // * #include "stm32f10x_sdio.h" */  
44 // * #include "stm32f10x_spi.h" */  
45 // * #include "stm32f10x_tim.h" */  
46 // * #include "stm32f10x_usart.h" */  
47 // * #include "stm32f10x_wwdg.h" */  
48 // * #include "misc.h" */ /* High level funct:
```



我们需要哪个驱动，只需将它的注释去掉即可。

**stm32f10x\_conf.h** 包含在 **stm32f10x.h** 中，如下图所示：

```
6970 #ifdef USE_STDPERIPH_DRIVER  
6971     #include "stm32f10x_conf.h"  
6972 #endif
```

注意：宏 `USE_STDPERIPH_DRIVER` 我们在新建工程的时候在  这个工具的 `C/C++` 选项卡中声明，如下所示：

```
Define: USE_STDPERIPH_DRIVER, STM32F10X_HD
```

M3 的库就简单介绍到这，至于具体到每个外设的应用还需大家多花时功力呀，我这可不能一一详述，但在接下来的模块教程中会详细讲到芯片中每个外设资源的使用。

实验讲解完毕，野火祝大家学习愉快^\_^。

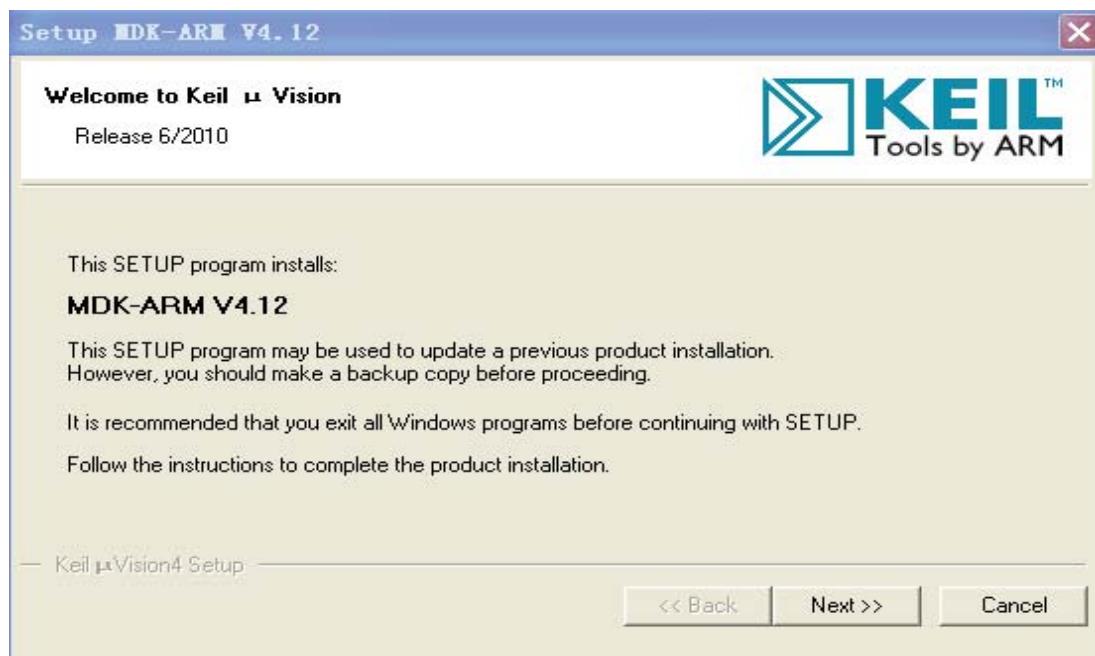


## 利用 STM32 的官方库在 RVMDK 中新建一个工程文件

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

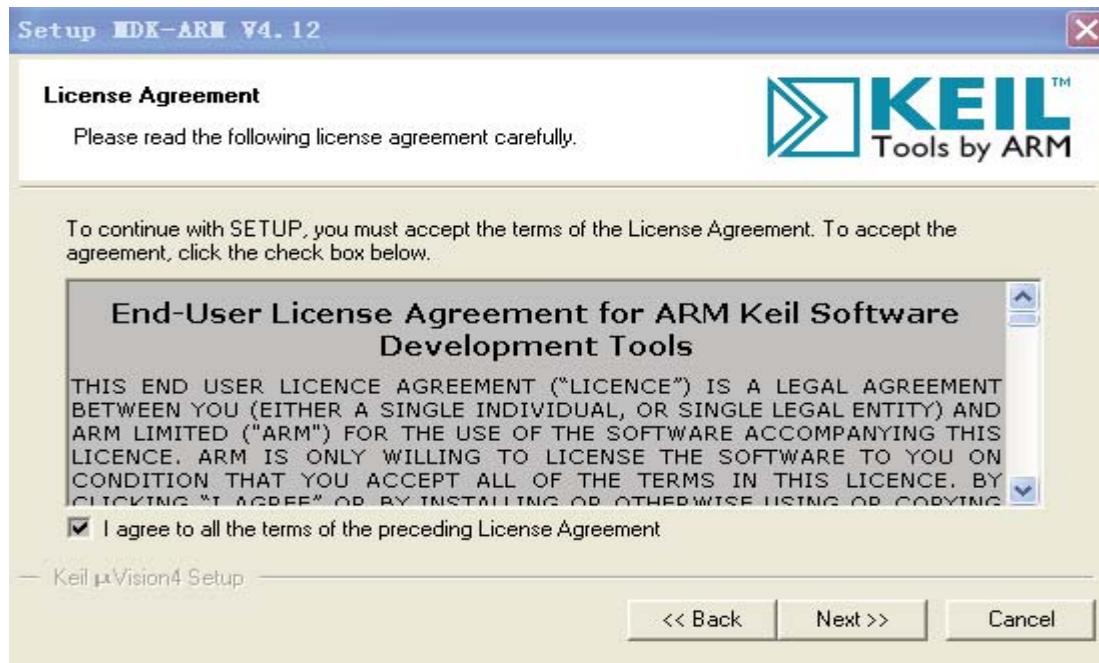
在新建工程之前我们先要把 RVMDK 这个软件安装好，这里用的版本是 V4.10，在安装完成之后可以在工具栏 help->about μVision 选项卡中查看到版本信息。μVision 是一个集代码编辑，编译，链接和下载于一体的集成开发环境（KDE），其支持我们常见的 arm7、arm9 和 arm 最新内核的 M3 系列，其前身就是 51 中的大名鼎鼎的 keil。安装过程如下所示：

- 1、点击 Next。

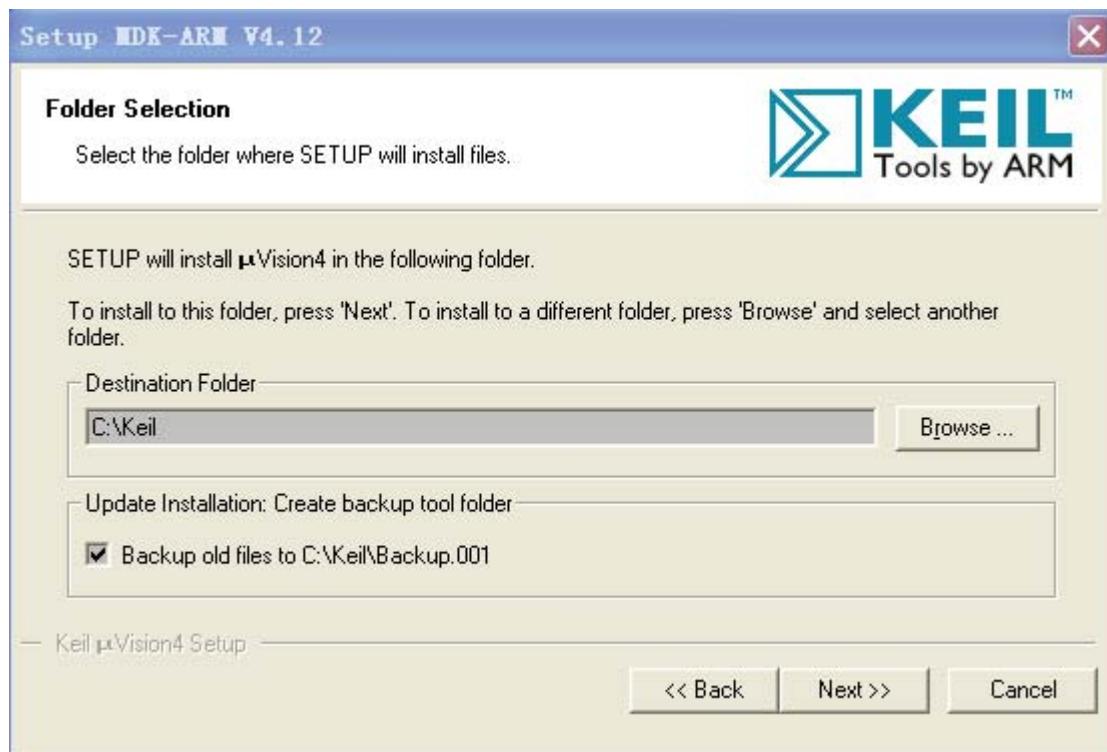




2、把勾勾上，点击 Next。

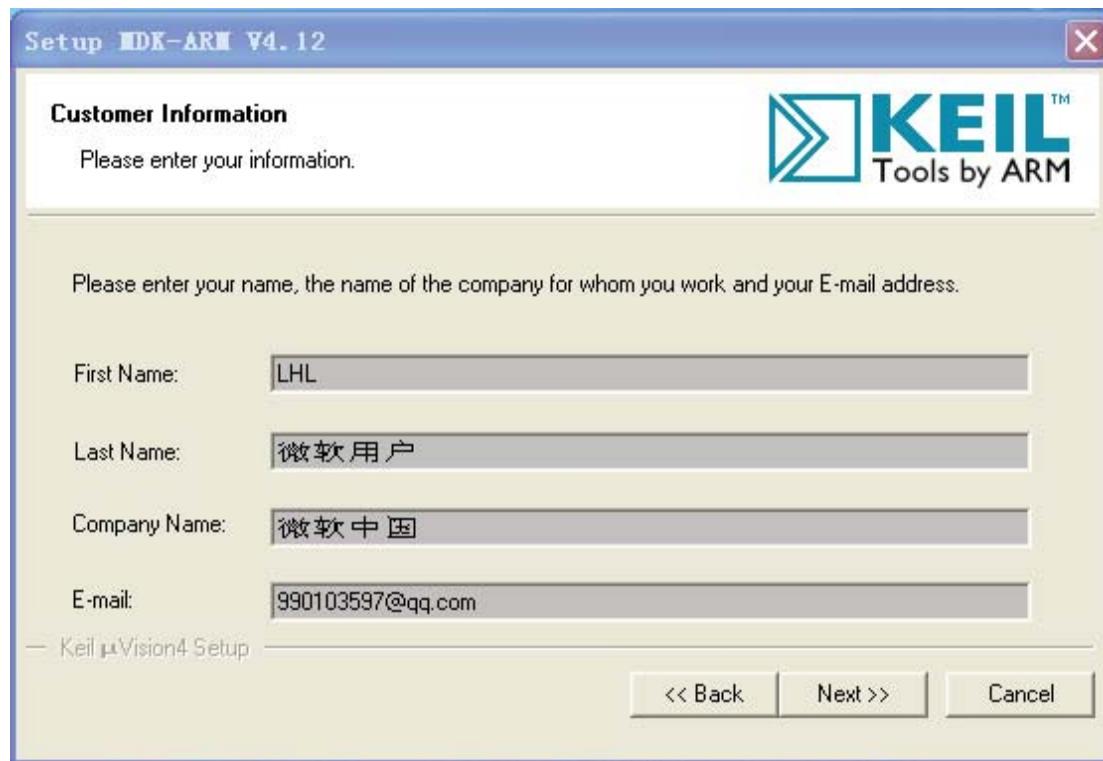


3、点击 Next， 默认安装在 C:\keil 目录下。

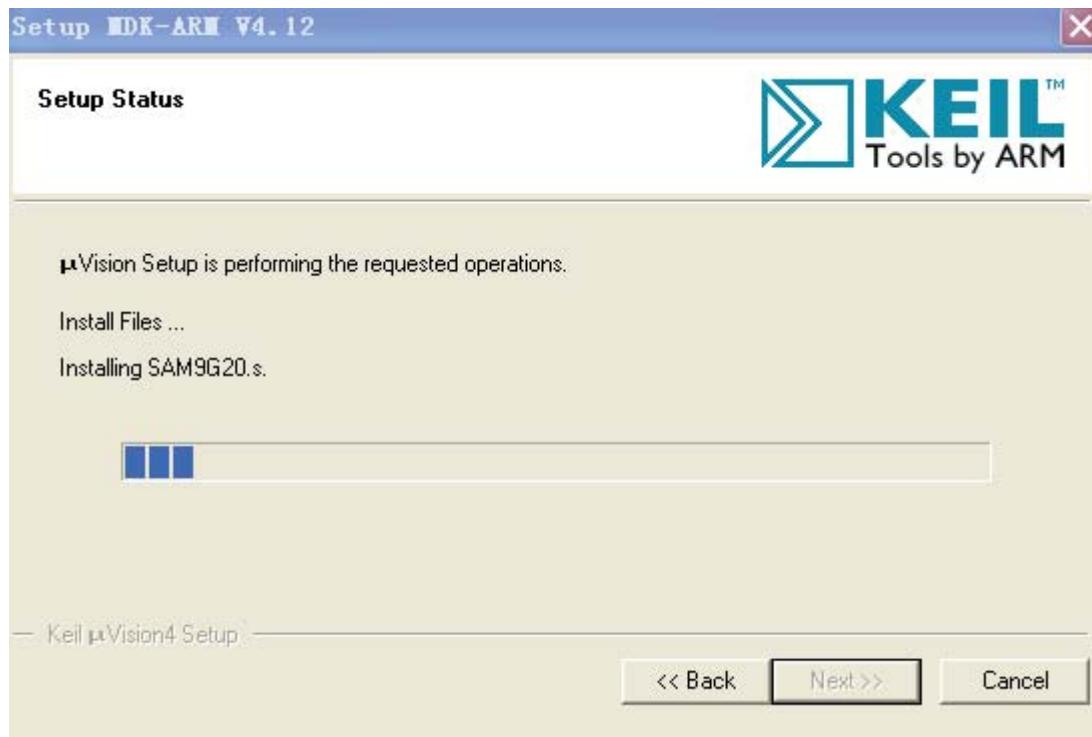




- 4、在用户名中填入名字（可随便写，可空格），在邮件地址那里填入邮件地址（可随便写，可空格），点击 Next。

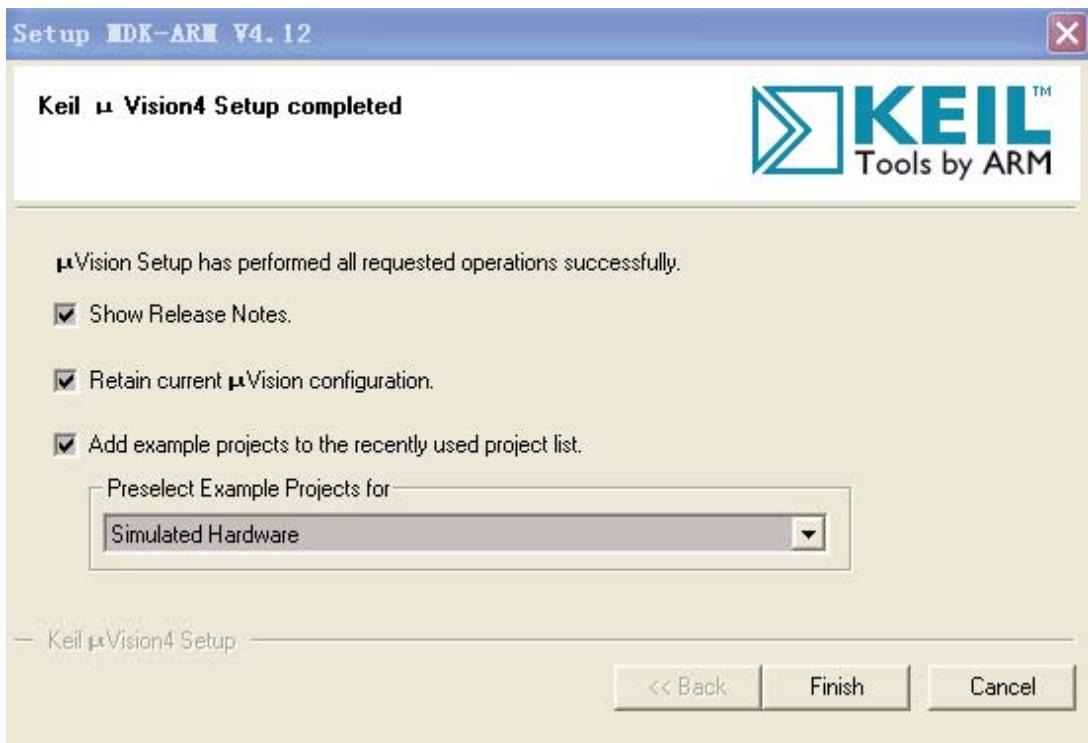


- 5、正在安装，请耐心等待。





6、点击 Finish，安装完成。



7、此时就可在桌面看到 μVision 的快捷图标，如下所示：

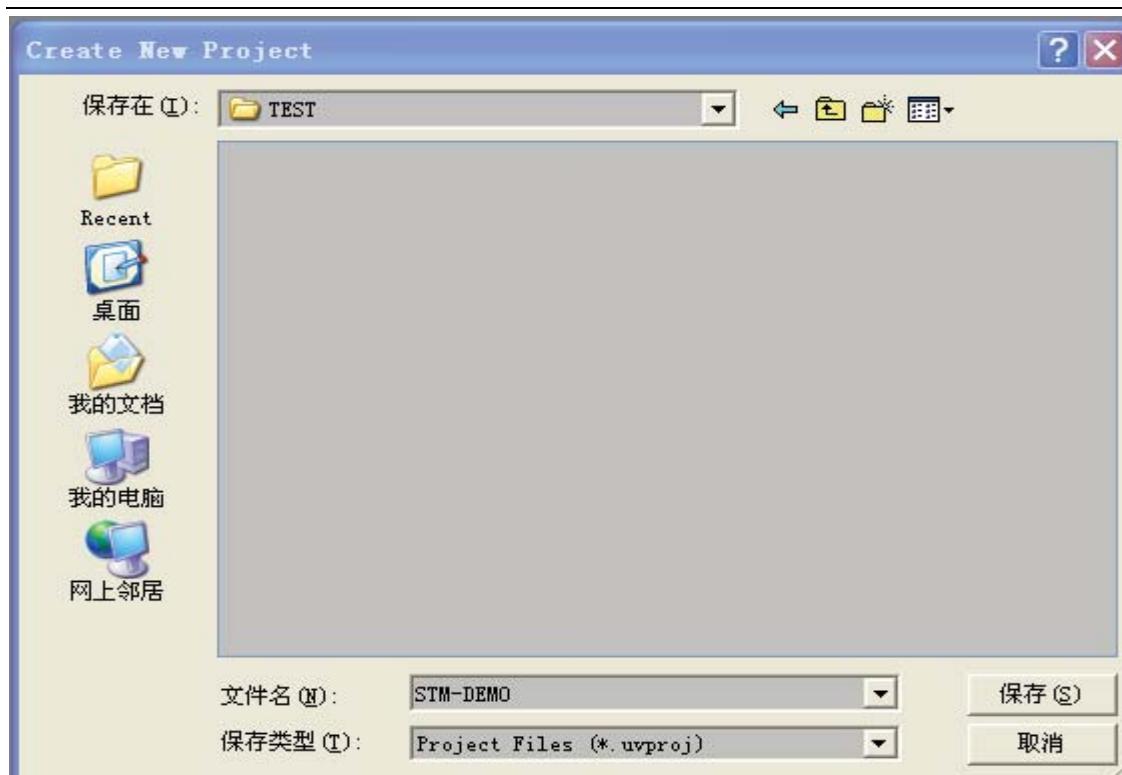


安装完成之后，先到 ST 的官方网站 <http://www.st.com/mcu/familiesdocs-110.html>（网址可能会随着时间有变动）下载 ST 的官方库，这里用的是 ST3.0.0 版本。

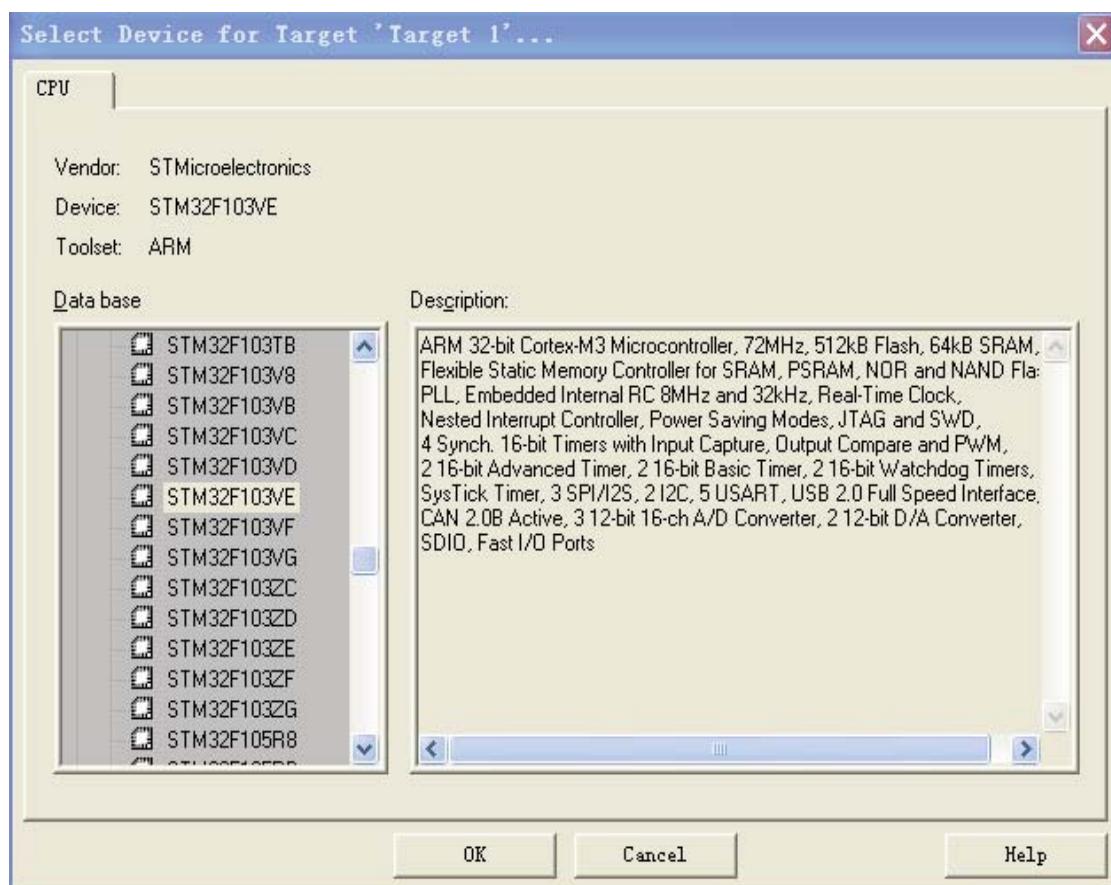
紧接着我们开始利用 STM32 的官方库来构建自己的工程模板。

1、点击桌面 μVision4 图标，启动软件。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏 Project->Close Project 选项把它关掉。

2、在工具栏 Project->New μVision Project... 新建我们的工程文件，我们将新建的工程文件保存在桌面的 TEST 文件夹下，文件名取为 STM-DEMO（英文 DEMO 的意思是例子），名字可以随便取，点击保存。



3、接下来的窗口是让我们选择公司跟芯片的型号，我们用的芯片是 ST 公司的 STM32F103VE，有 64K SRAM,512K Flash，属于高集成度的芯片。按如下选择即可。





4、接下来的窗口问我们是否需要拷贝 STM32 的启动代码到工程文件中，这份启动代码在 M3 系列中都是适用的，一般情况下我们都点击是，但我们这里用的是 ST 的库，库文件里面也自带了这一份启动代码，所以为了保持库的完整性，我们就不需要开发环境为我们自带的启动代码了，稍后我们自己手动添加，这里我们点击否。



5、此时我们的工程新建成功，如下图所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。



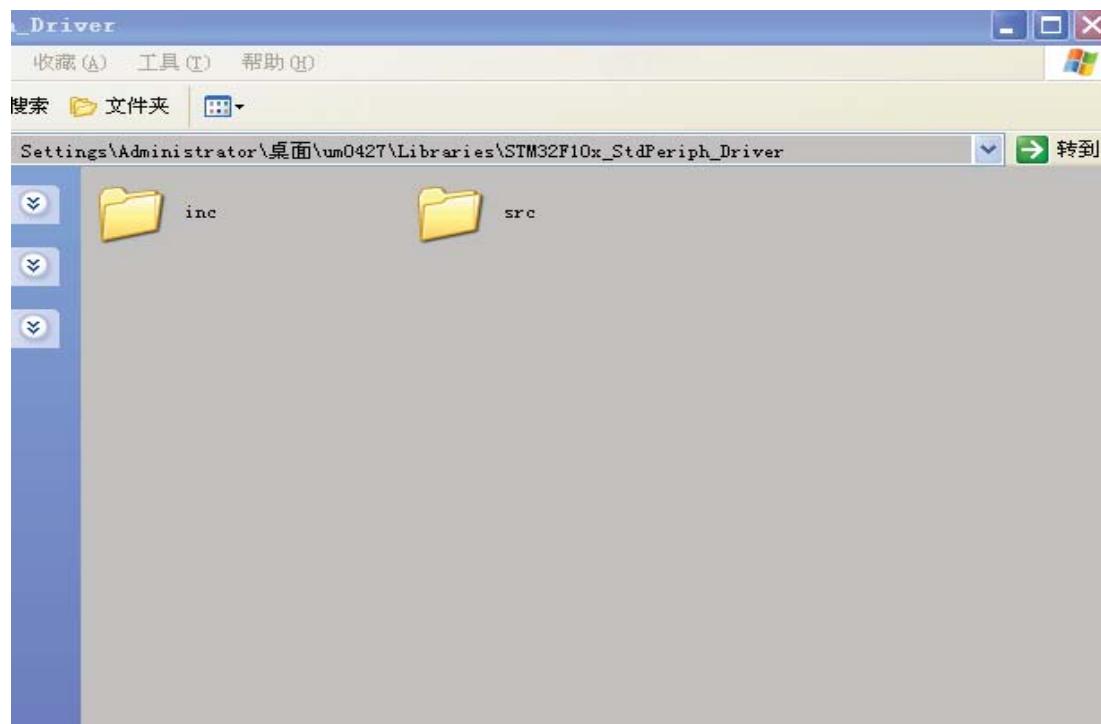
6、在 TEST 文件夹中，新建三个文件夹，分别为 USER、FWlib、CMSIS。USER 用来存放工程文件和用户代码，包括主函数 main.c，FWlib 用来存放 STM32 库里面的 inc 和 src 这两个文件，这两个文件包含了芯片上的所有驱动。CMSIS 用来存放库为我们自带的启动文件和一些 M3 系列通用的文件。CMSIS 里面存放的文件适合任何 M3 内核的单片机。CMSIS 的缩写为：Cortex Microcontroller Software Interface Standard，



是 ARM Cortex 微控制器软件接口标准，是 ARM 公司为芯片厂商提供的一套通用的且独立于芯片厂商的处理器软件接口。



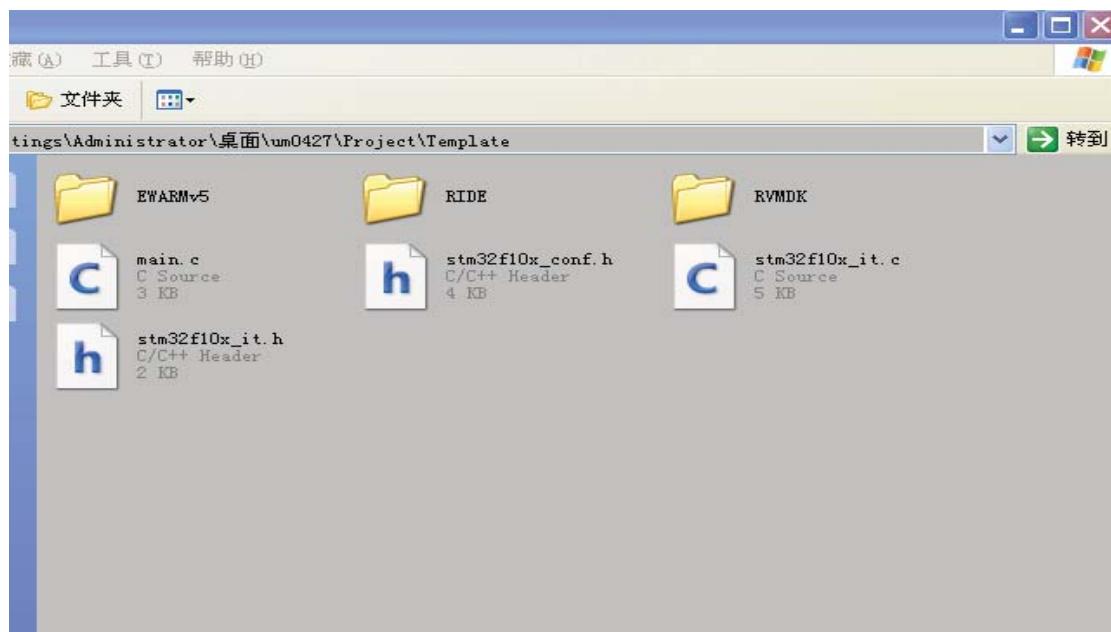
7、把库 um0427\Libraries\STM32F10x\_StdPeriph\_Driver 文件夹下的 inc 跟 src 这两个文件夹拷贝的 FWlib 文件夹中。





---

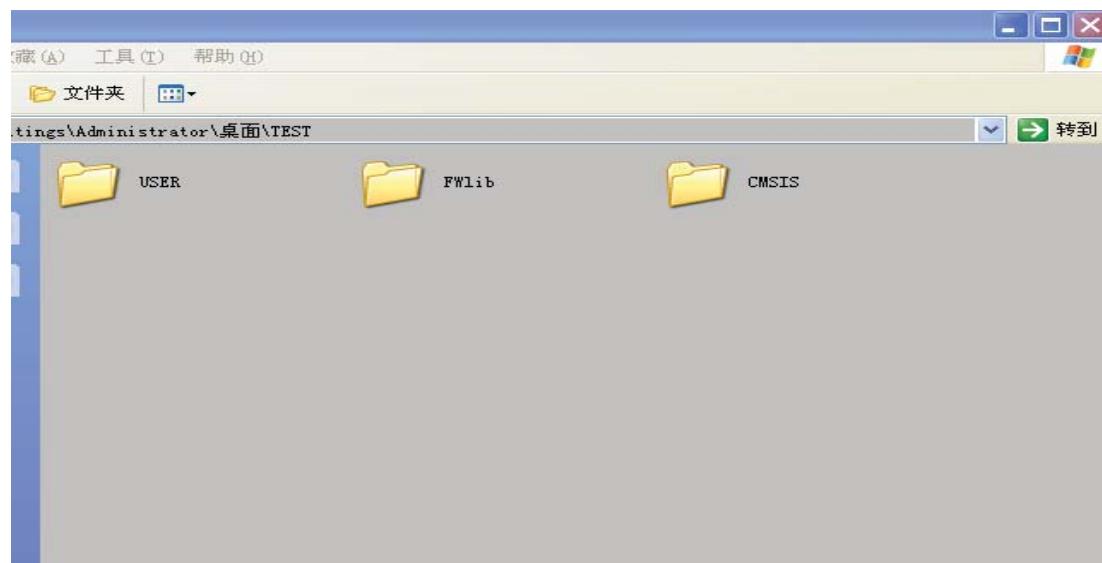
8、把库文件夹 um0427\Project\Template 下的 main.c、stm32f10x\_conf.h、stm32f10x\_it.h、stm32f10x\_it.c 拷贝到 USER 目录下。stm32f10x\_it.h 和 stm32f10x\_it.c 这两个文件里面是中断函数，里面为空，并没有写任何的中断服务程序。stm32f10x\_conf.h 是用户需要配置的头文件，当我们需要用到芯片中的某部分外设的驱动时，我们只需要在该文件下将该驱动的头文件包含进来即可，片上外设的驱动在 src 文件夹中，inc 文件夹里面是它们的头文件。这三个文件是用户在编程时需要修改的文件，其他库文件一般不需要修改。



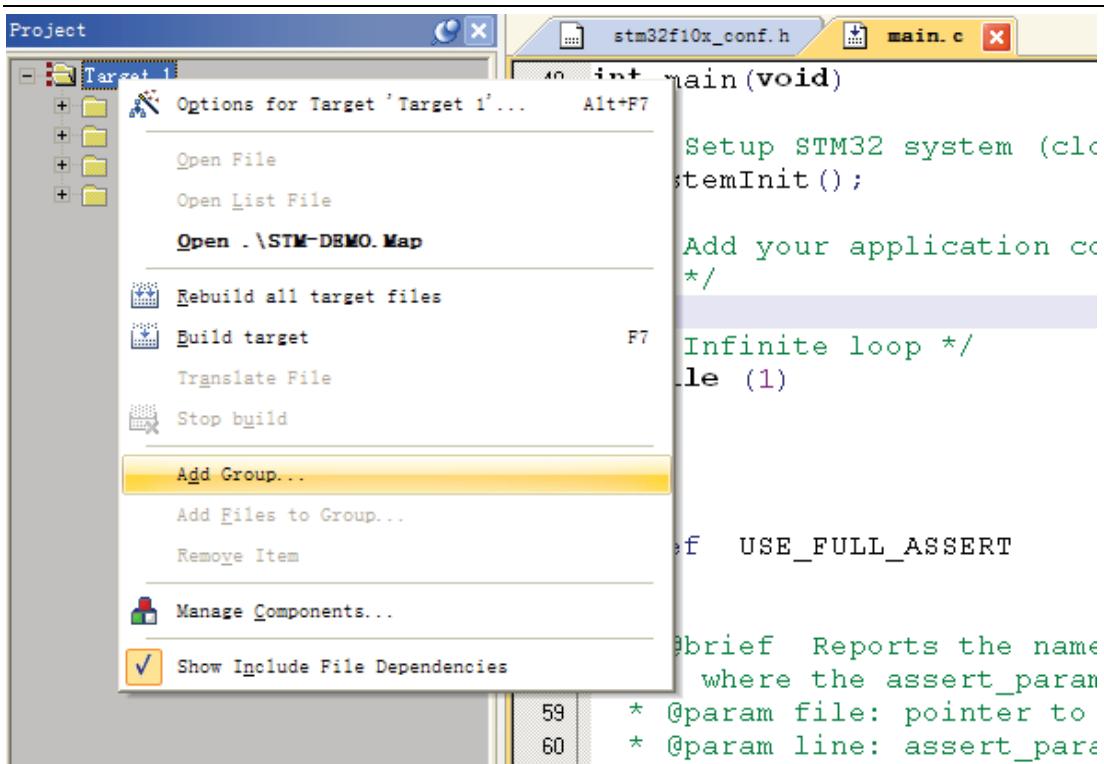
9、将库文件 um0427\Libraries\CMSIS\Core\CM3 文件夹下的全部文件拷贝到 CMSIS 文件夹中。Startup/arm 里面有三个启动文件，分别为 startup\_stm32f10x\_ld.s、startup\_stm32f10x\_md.s、startup\_stm32f10x\_hd.s，按顺序是小容量、中容量、大容量 Flash 单片机的启动文件。我们这里用的是 STM32F103VE 有 512K Flash，属于大容量的，所以等下我们把 startup\_stm32f10x\_hd.s 添加到我们的工程中。具 ST 的官方资料：Flash 在 16 ~32 Kbytes 为小容量，64 ~128 Kbytes 为中容量，256 ~512 Kbytes 为大容量，不同大小的 Flash 对应的启动文件不一样，这点要注意。其他几个文件也是 M3 内核单片机通用的，是独立于芯片厂商的，其功能由 ARM 公司决定，具体作用这里先不详述。



10、最后我们将我们的工程跟其他的一些编译出来的文件也放在 USER 目录下，这样看起来显得没那么乱。



11、回到我们的工程中，选中 Target 右键选中 Add Group...选项新建四个组，分别命名为 STARTCODE、USER、FWlib、CMSIS。STARTCODE 从名字就可以看得出我们是用它来放我们的启动代码的，USER 用来存放用户自定义的应用程序，FWlib 用来存放库文件，CMSIS 用来存放 M3 系列单片机通用的文件。



The screenshot shows the Keil MDK-ARM IDE interface. On the left is the Project Manager with a tree view of 'Target 1' containing various project files. The main window shows a portion of the 'main.c' source code. A context menu is open over the Project Manager, with the option 'Add Group...' highlighted in yellow. Other options in the menu include 'Add Files to Group...', 'Remove Item', 'Manage Components...', and 'Show Include File Dependencies'. The code in the main window is as follows:

```
int main(void)
{
    /* Setup STM32 system (clock, memory, peripherals, etc)
     * stemInit();
     *
     * Add your application code here
     */
    Infinite loop
}

/* (1) */

#define USE_FULL_ASSERT

/* brief Reports the name
 * where the assert_param
 * @param file: pointer to
 * @param line: assert_param
 */
```

12、接下来我们往我们这些新建的组中添加文件，**双击**哪个组就可以往哪个组里面添加文件。我们在 **STARTCOKE** 里面添加 **startup\_stm32f10x\_hd.s**，在 **USER** 组里面添加 **main.c** 和 **stm32f10x\_it.c** 这两个文件，在 **FWlib** 组里面添加 **src** 里面的全部驱动文件，当然，**src** 里面的驱动文件也可以需要哪个就添加哪个。这里将它们全部添加进去是为了后续开发的方便，况且我们可以通过配置 **stm32f10x\_conf.h** 这个头文件来选择性添加，只有在 **stm32f10x\_conf.h** 文件中配置的文件才会被编译。注意，这些组里面添加的都是汇编文件跟 C 文件，头文件是不需要添加的。最终效果如下图：



The screenshot shows the Keil MDK-ARM IDE interface. On the left is the Project Explorer window titled 'Project' with 'Target 1' selected. It lists several source files under different folders: STARTCODE, USER, FWlib, and CMSIS. On the right is the Code Editor window with the tab 'main.c' selected. The code in 'main.c' is as follows:

```
40 int main(void)
41 {
42     /* Setup STM32 system
43     SystemInit();
44
45     /* Add your application
46     */
47
48     /* Infinite loop */
49     while (1)
50     {
51     }
52 }
53
54 #ifdef USE_FULL_ASSERT
55
56 /**
57 * @brief Reports the file
58 * where the assert_
59 * @param file: pointer
60 * @param line: assert_
61 * @retval : None
62 */
63 void assert_failed(uint8_t
64 {
65     /* User can add his own
66     ex: printf("Wrong p
67
68     /* Infinite loop */
```

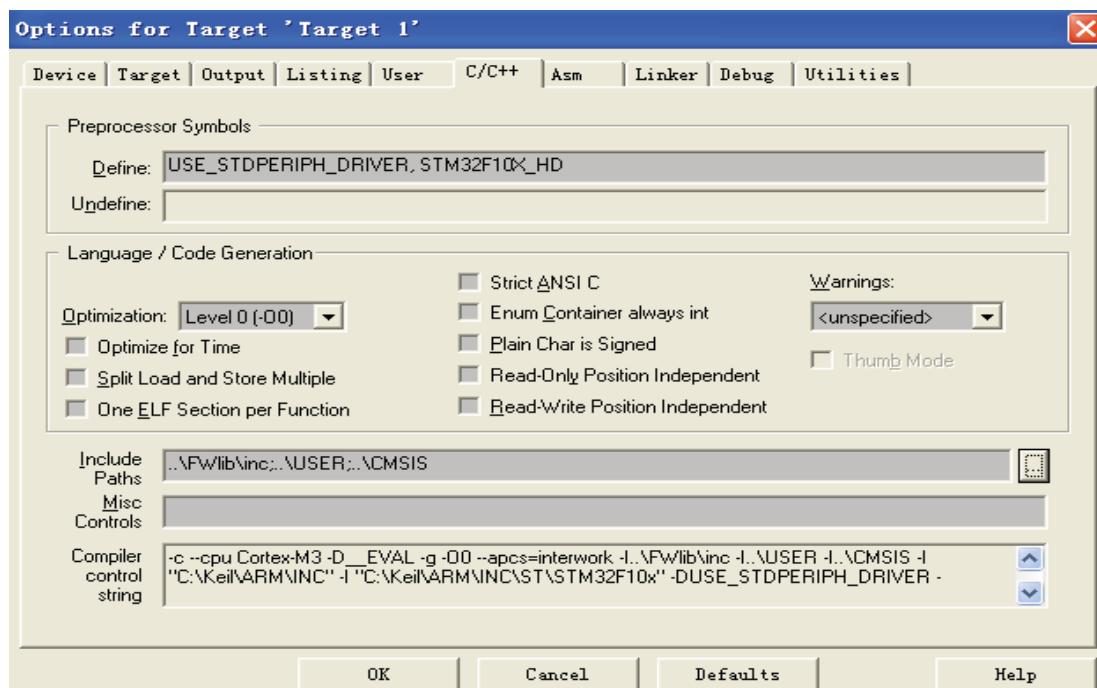
13、至此，我们的工程一基本建好，现在点击工具栏图标 来编译下，结果发现了 N 多的错误，如下图所示。究其原因是编译器在编译时搜索的库路径是：  
C:\Keil\ARM\INC\ST\STM32F10x，这里面也有 ST 官方的驱动库的头文件，里面的文件跟刚刚我们的 inc 文件夹下的内容差不多，只是版本旧了点，在编译我们新版本的库时存在不兼容性。为了解决这个问题，我们需要屏蔽掉编译器默认库的搜索路径。



```
Build Output

C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h(62): error: #101: "ERROR" has already been declared
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:     typedef enum {ERROR = 0, SUCCESS = !ERROR} Error;
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h(62): error: #101: "SUCCESS" has already been declared
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:     typedef enum {ERROR = 0, SUCCESS = !ERROR} Error;
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h(62): error: #256: invalid redeclaration of type name
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:     typedef enum {ERROR = 0, SUCCESS = !ERROR} Error;
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_type.h:
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_conf.h(147): warning: #47-D: incompatible redefinition of
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_conf.h:     #define HSE_Value ((u32)8000000) /* Value of t
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_conf.h:
C:\Keil\ARM\INC\ST\STM32F10x\stm32f10x_conf.h: ..\CMSIS\system_stm32f10x.c: 1 warning, 21 errors
Target not created
```

14、点击工具栏中的魔术棒按钮 , 在弹出来的窗口中选中 **C/C++** 选项卡，在这里添加库文件的搜索路径，这样就可以屏蔽掉默认的搜索路径。但当编译器在我们指定的路径下 搜索不到的话还是会回到标准目录去搜索，就像有些 **ANSIC C** 的库文件，如 **stdin.h**、**stdio.h**。

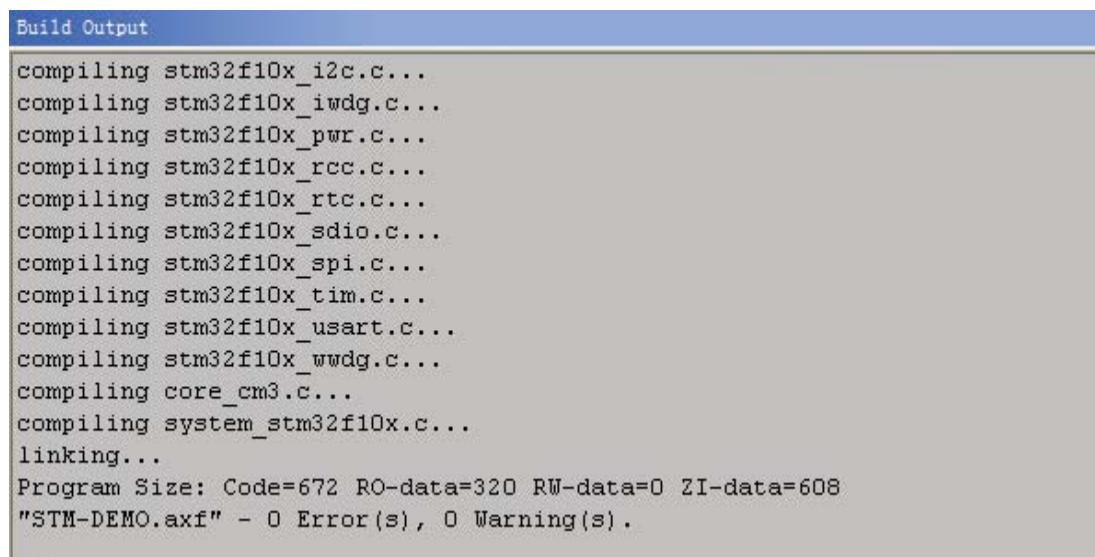




15、库文件路径修改成功之后如下所示:

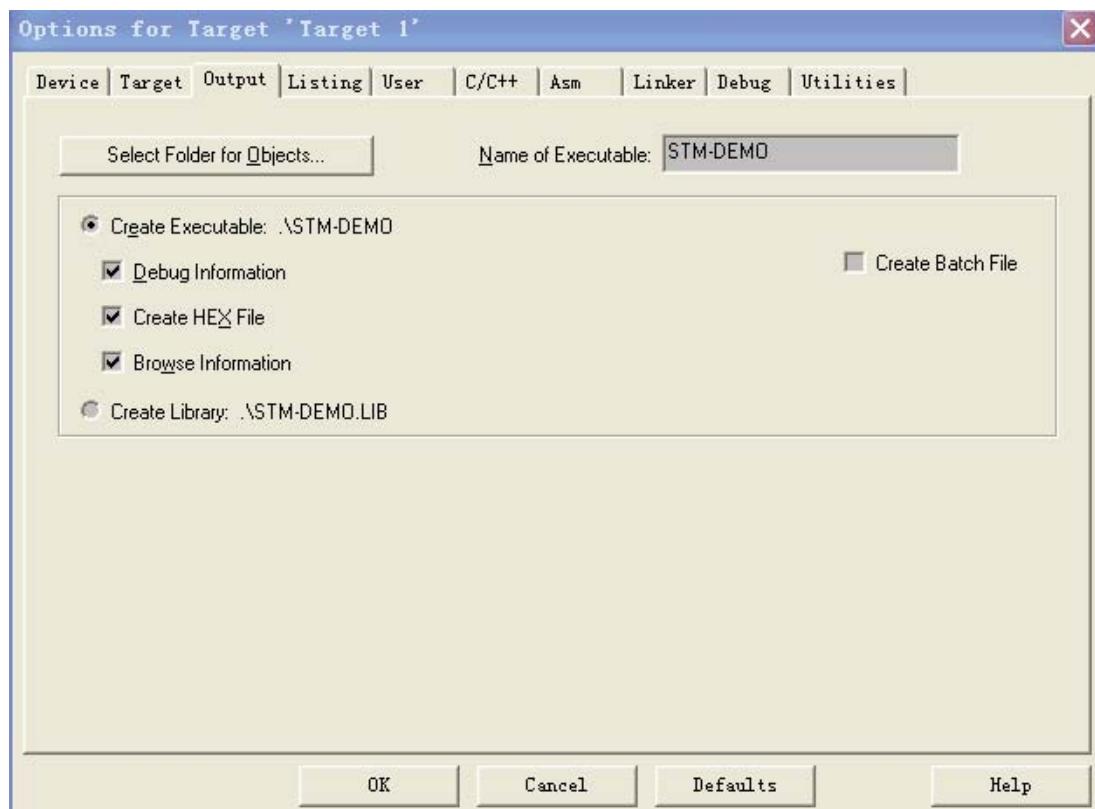


16、现在我们再编译下，就会发现刚刚出现的那些错误都没了，如下图所示:

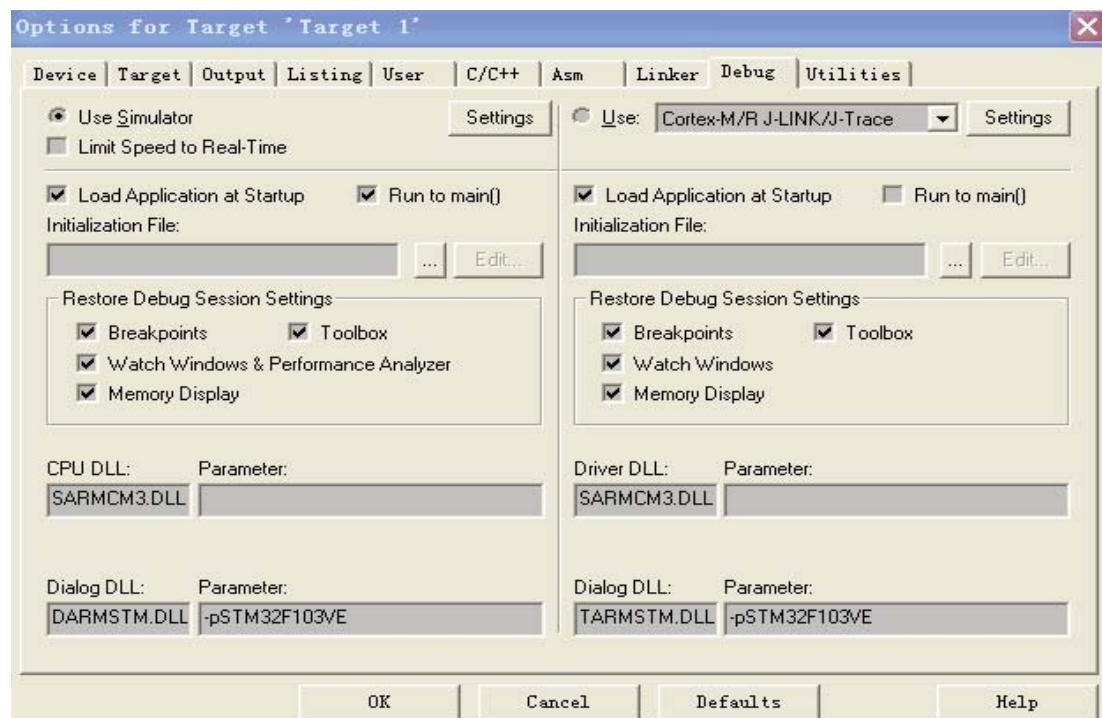
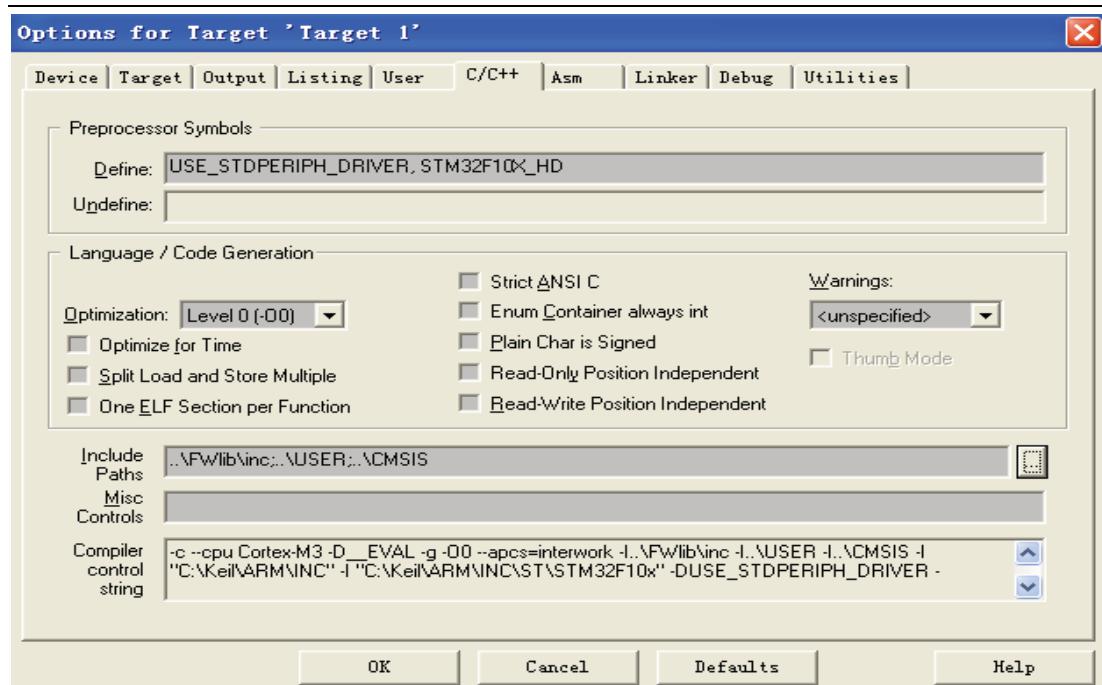




17、现在我们还需要再做一些工作我们的工程才算完成，主要修改如下几个选项卡，其他保持默认即可，如下图所示：



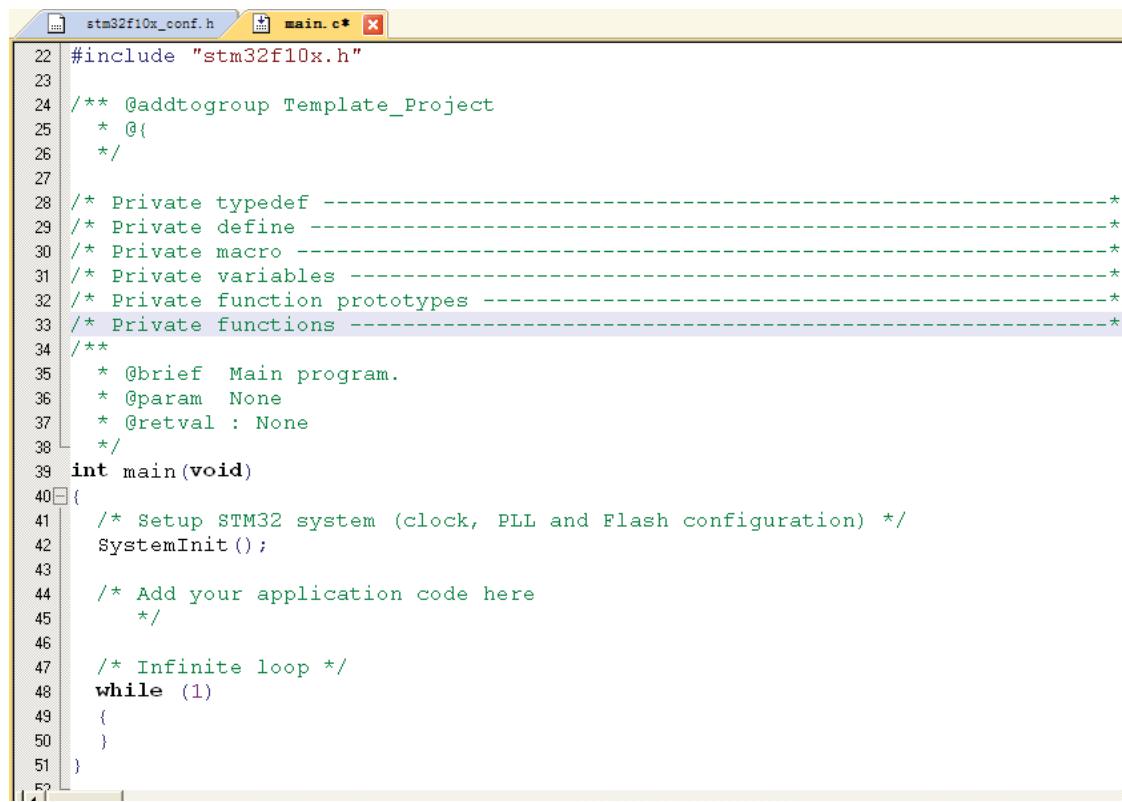
在 Define 里面添加 USE\_STDPERIPH\_DRIVER, STM32F10X\_HD。添加 USE\_STDPERIPH\_DRIVER 是为了使用 ST 的库，添加 STM32F10X\_HD 是因为我们用的芯片是大容量的，添加了 STM32F10X\_HD 这个宏之后，库文件里面为大容量定义的寄存器我们就可以用了。芯片是小或中容量的时候宏要换成 STM32F10X\_LD 或者 STM32F10X\_MD。其实不管是什么容量的，我们只要添加上 STM32F10X\_HD 这个宏即可，当你用小或者中容量的芯片时，那些为大容量定义的寄存器我不去访问就是了，反正也访问不了。





---

17、至此，大功告成，我们就可以在 main.c 函数中写自己的程序了。

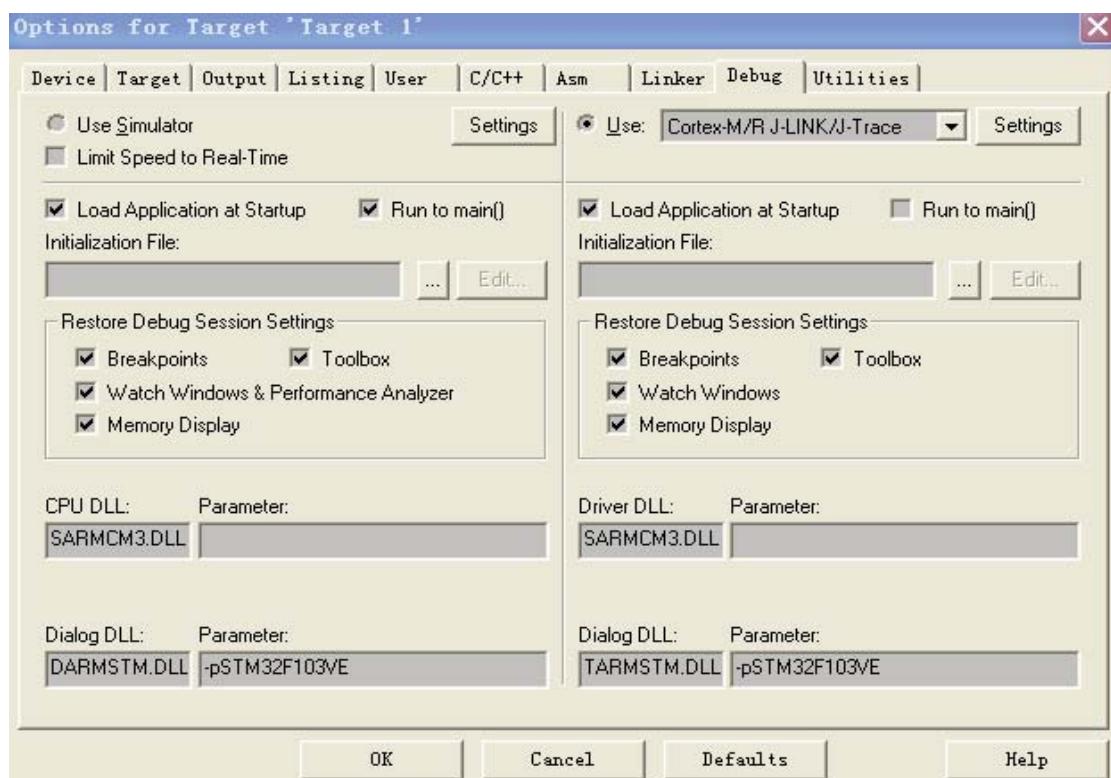


```
22 #include "stm32f10x.h"
23
24 /** @addtogroup Template_Project
25  * @{
26  */
27
28 /* Private typedef -----*/
29 /* Private define -----*/
30 /* Private macro -----*/
31 /* Private variables -----*/
32 /* Private function prototypes -----*/
33 /* Private functions -----*/
34 /**
35  * @brief Main program.
36  * @param None
37  * @retval : None
38  */
39 int main(void)
40{
41    /* Setup STM32 system (clock, PLL and Flash configuration) */
42    SystemInit();
43
44    /* Add your application code here
45     */
46
47    /* Infinite loop */
48    while (1)
49    {
50    }
51}
```



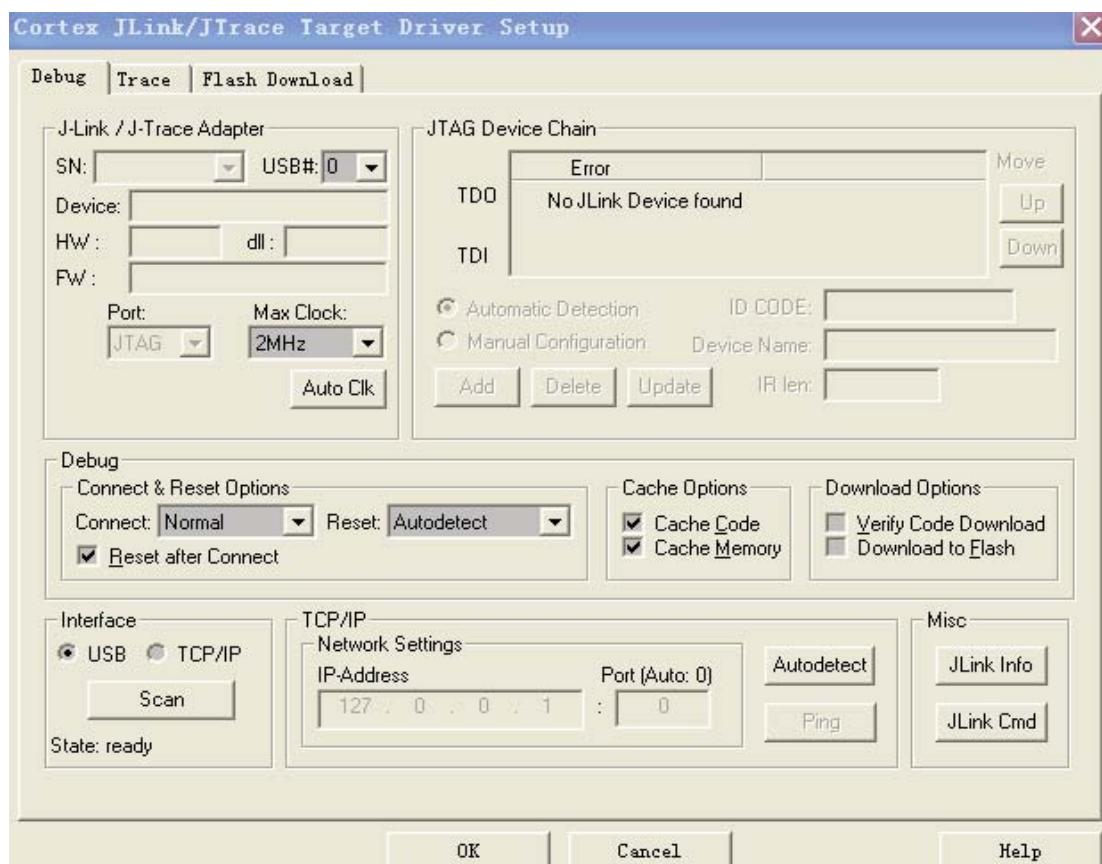
**小结:** 学会新建工程是后续程序开发的一个非常重要的工作, 如果工程没法建成, 那何来的开发呢? 在建立工程时需要注意的是: 1、因为我们用的是 ST 官方的新版本的库, 跟编译器自带的库会存在不兼容性, 所以我们需要修改库的搜索路径。2、这个工程我们是设置成软件仿真, 如果是用开发板加 J-LINK 调试的话, 还需要在开发环境中做如下修改。实际上, 我们开发程序的时候 80%都是在硬件上调试的。

具体配置如下图所示:

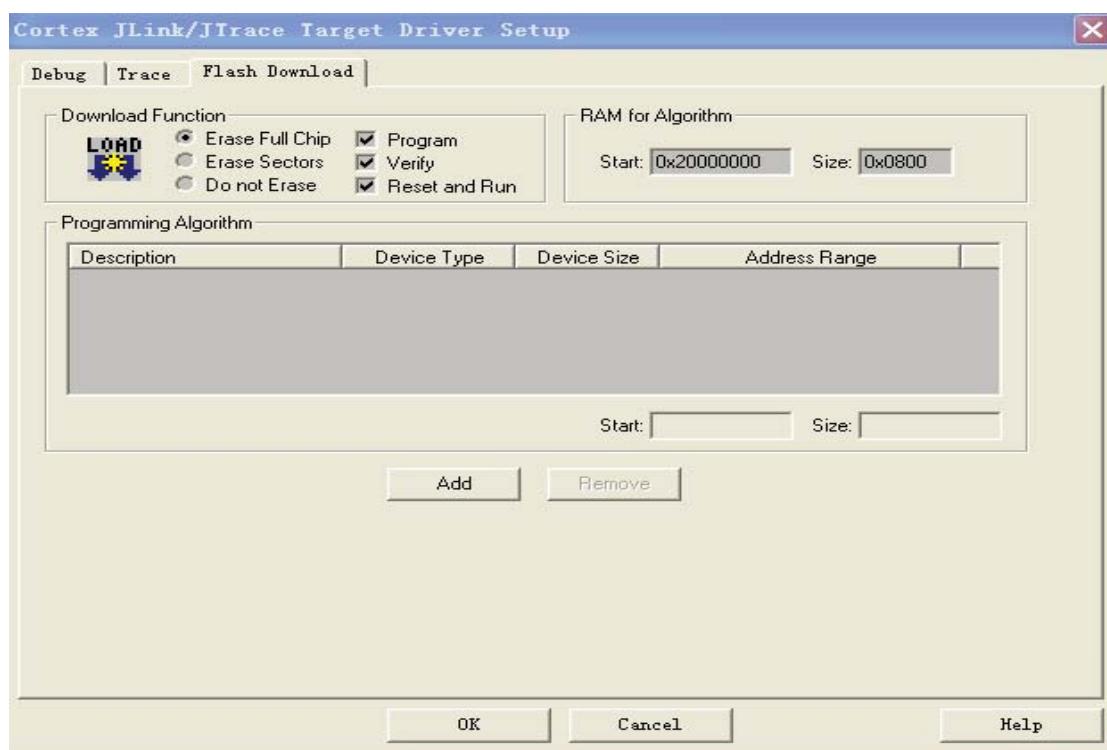




注意: 以下三个图是 J-LINK 没接到开发板上且没上电时的截图  
没检测到 CPU 的 ID。



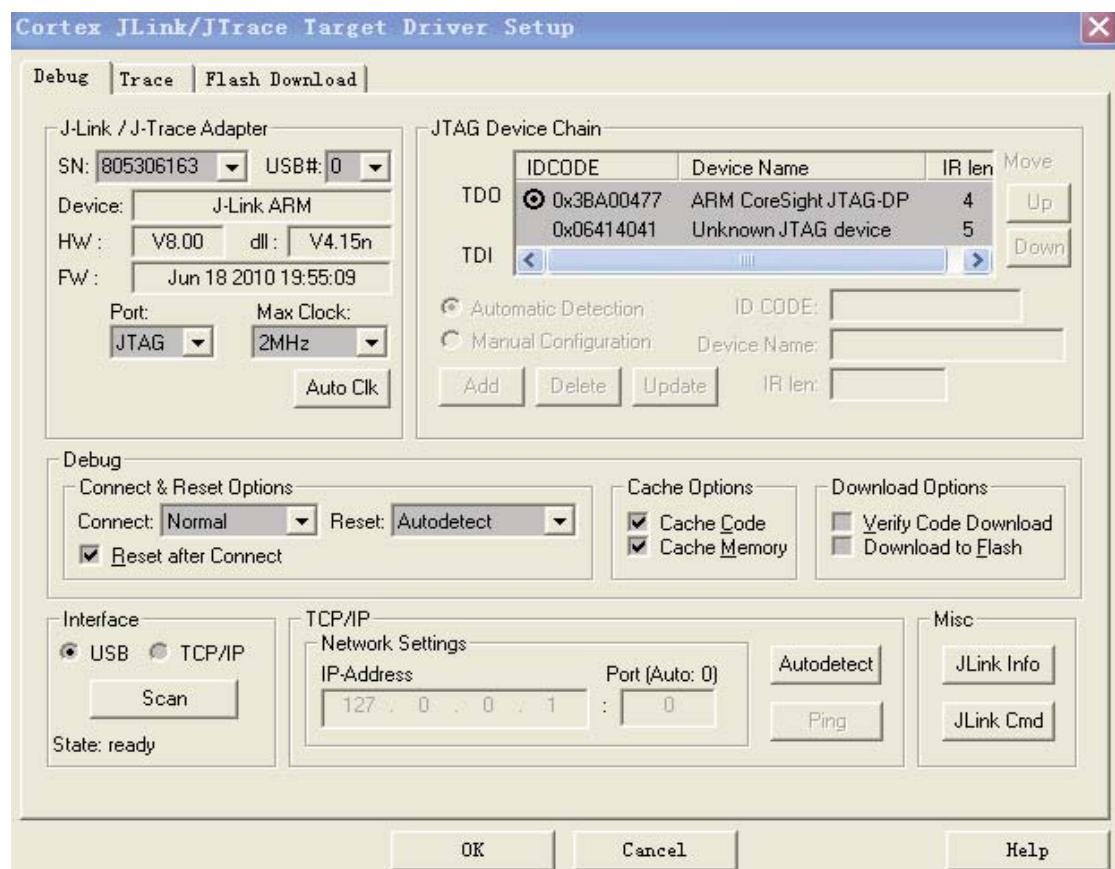
没有添加 CPU 支持的 flash。





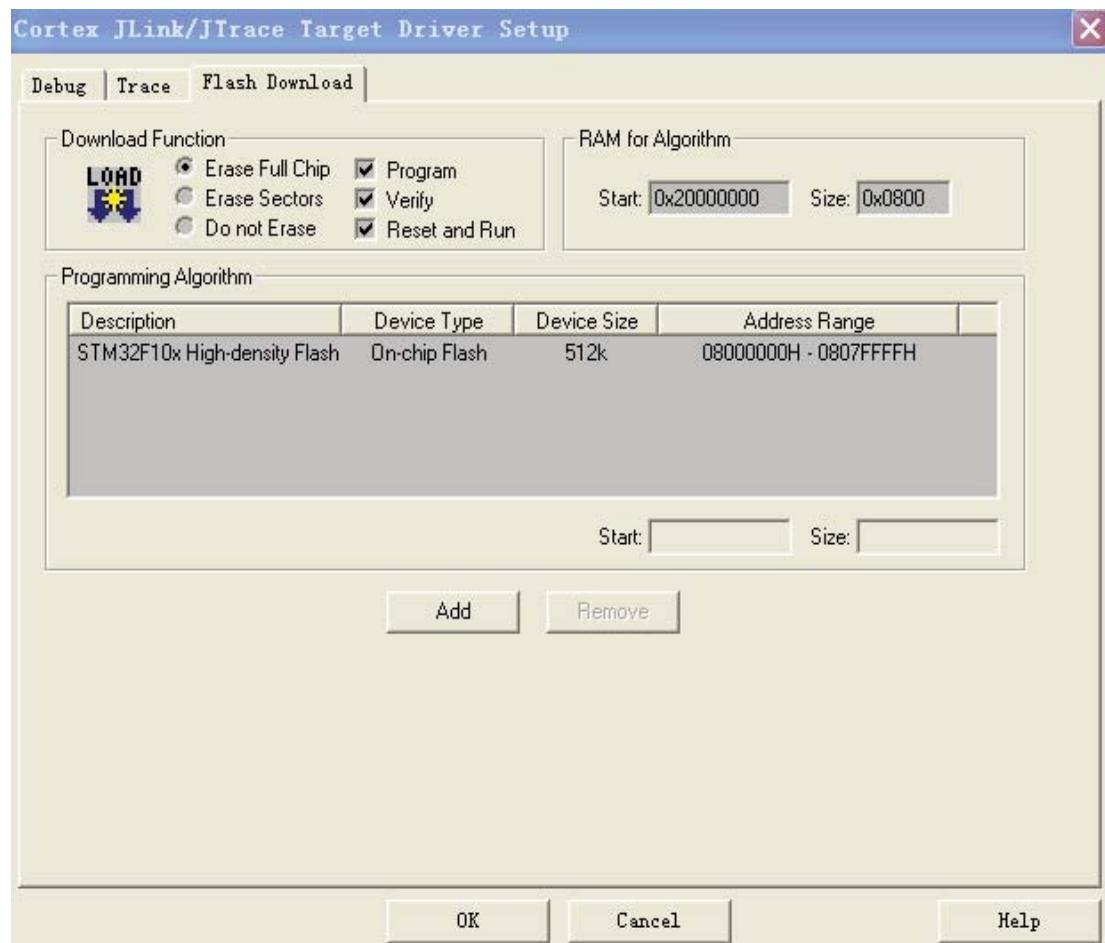
以下两幅是 J-LINK 接到开发板且上电时的截图:

检测到 CPU 的 ID





添加 CPU 支持的 flash，这一步很重要，否则程序将无法下载。



至此，一个真正完整的工程就算建立了。

实验讲解完毕，野火祝大家学习愉快^\_^。



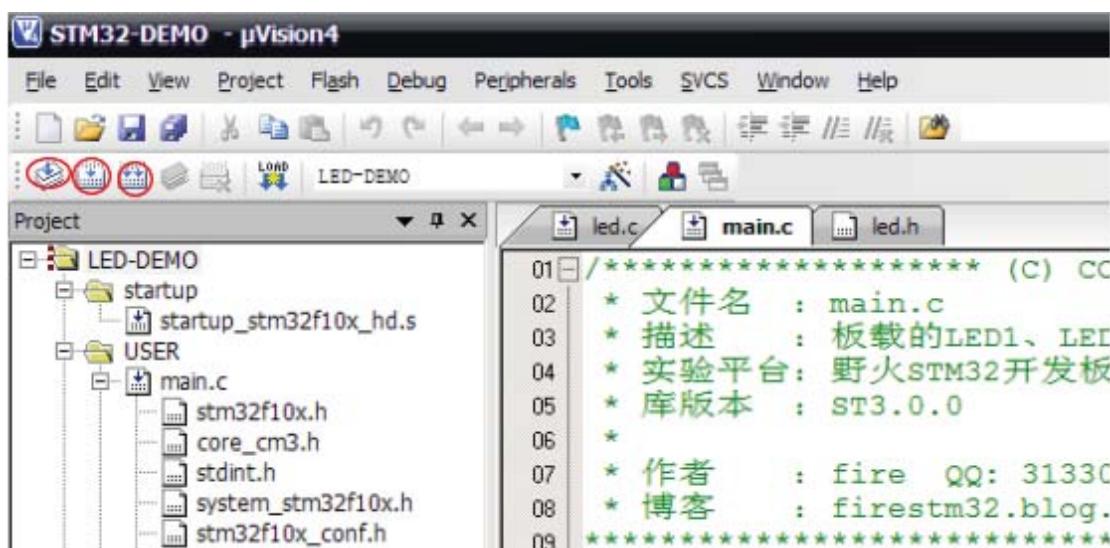
## 如何编译和下载程序

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

**注:** 在看这个教程之前, 我们首先得学会新建工程, 会配置 MDK 的一些选项, 或者手头上有一个可以用的工程文件, 如野火 STM32 开发板自带的例程就可以。有关如何新建工程请参考第一个教程《新建工程》

### 一、如何编译程序

首先打开一个 MDK 工程, 在界面的左边的工具栏中有三个按钮, 我们从左往右来介绍下这三个按钮的功能。





---

第一个按钮: **Translate** 就是翻译当下修改过的文件, 说明白点就是检查下有没有语法错误, 并不会去链接库文件, 也不会生成可执行文件。

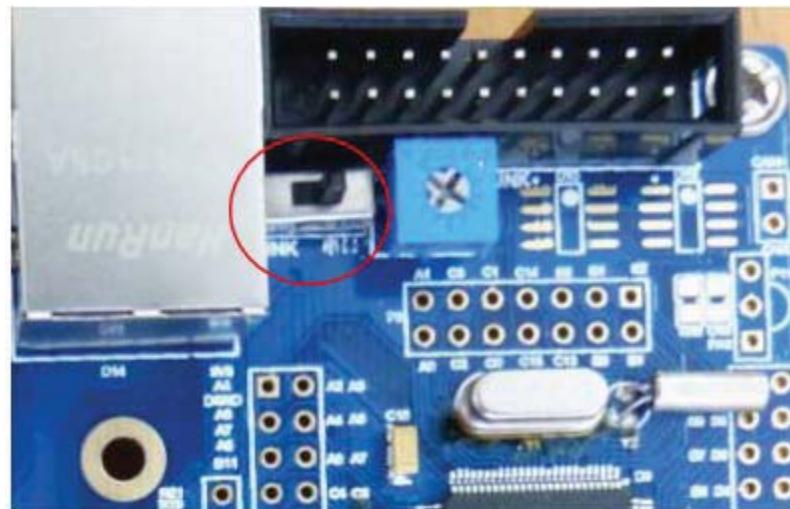
第二个按钮: **Build** 就是编译当下修改过的文件, 它包含了语法检查, 链接动态库文件, 生成可执行文件。

第三个按钮: **Rebuild** 重新编译整个工程, 跟 **Build** 这个按钮实现的功能是一样的, 但有所不同的是它编译的是整个工程的所有文件, 耗时巨大。

综上: 当我们编辑好我们的程序之后, 只需要用第二个 **Build** 按钮就可以, 即方便又省时。第一个跟第三个按钮用的非常少。

### 一、如何下载程序

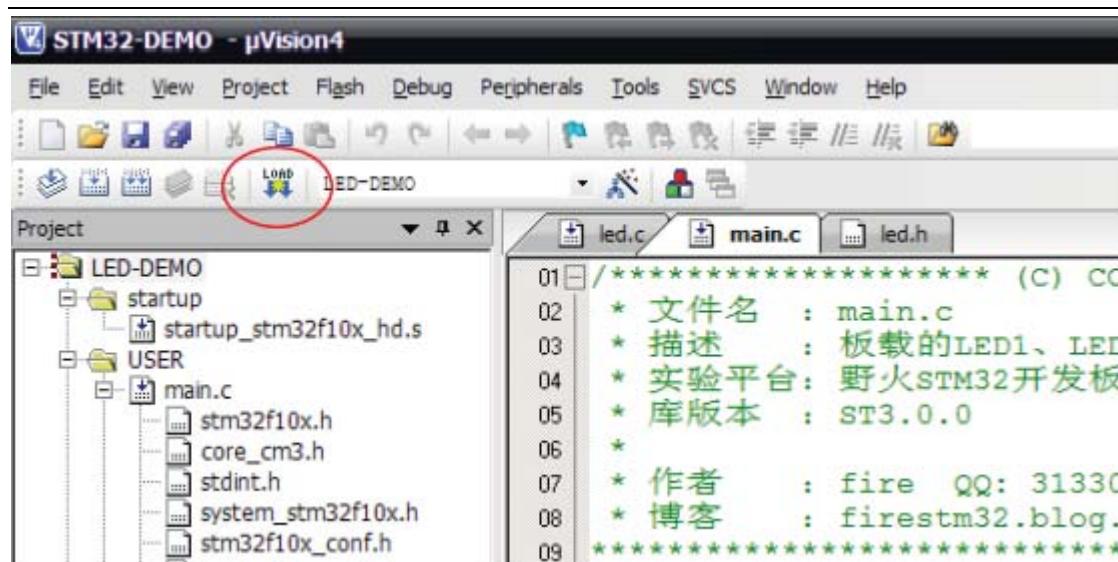
STM32 有两种下载方式, **JLINK** 下载和串口下载, 我们的开发板中用一个开关来控制这两种下载模式, 默认情况下我们是将开关打到 **JLINK** 下载那一端。这个开关的在开发板中的位置如下截图:



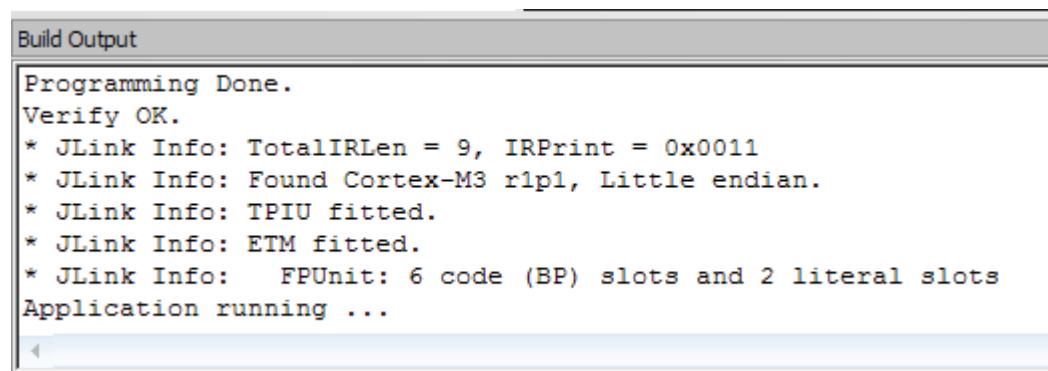
#### JLINK 下载->

1->给开发板供电(DC5V), 插上 **JLINK**, 将下载模式开关打到 **JLINK** 模式。

2->点击 **MDK** 工具栏中的 **Load** 按钮就可将编译好的程序下载到开发板的 **flash**。

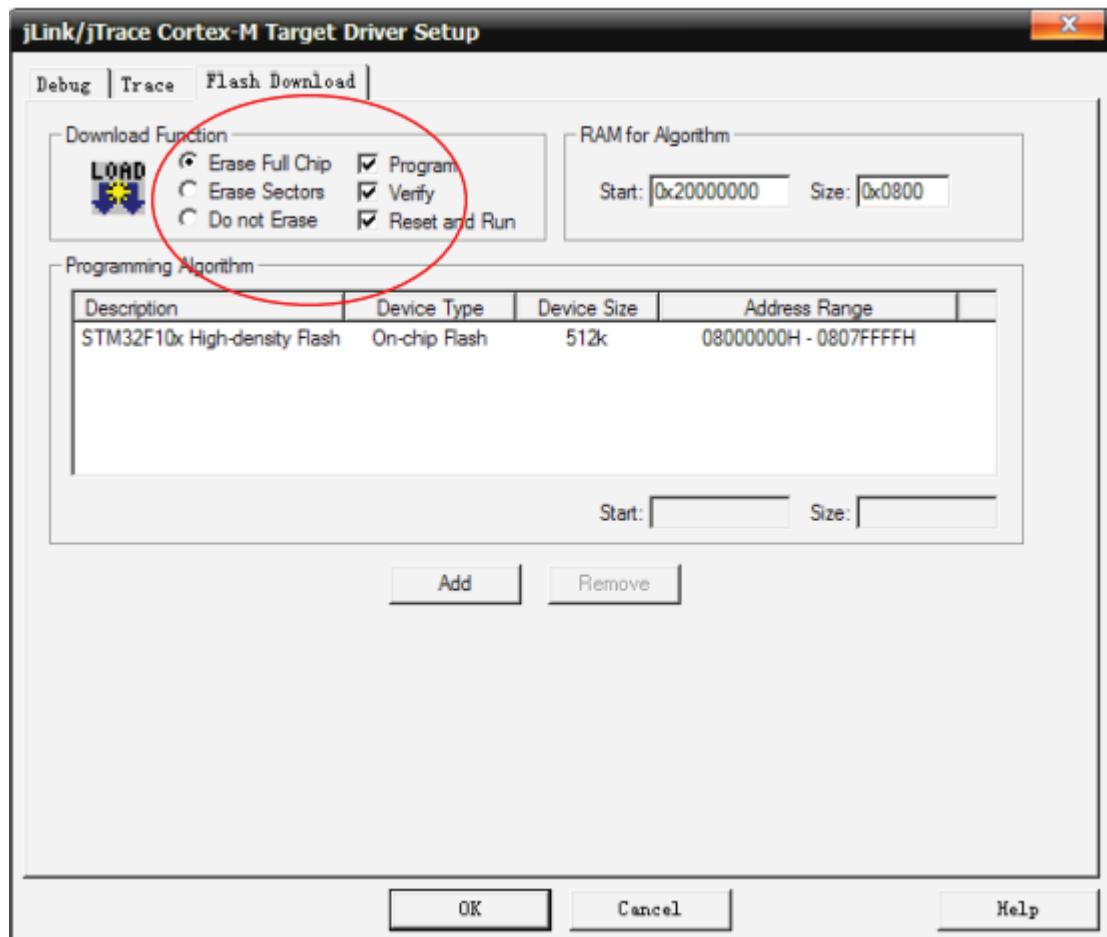


3->下载成功之后，程序就会自动运行。





程序是否自动运行，是由我们自己设定的，这个在  
Target Options...->Debug->Setting->Flash DownLoad 中设置：



如果没有设置为自动运行的话，我们需要在程序下载完毕之后进行手动复位，手动复位可以是按键复位和上电复位。

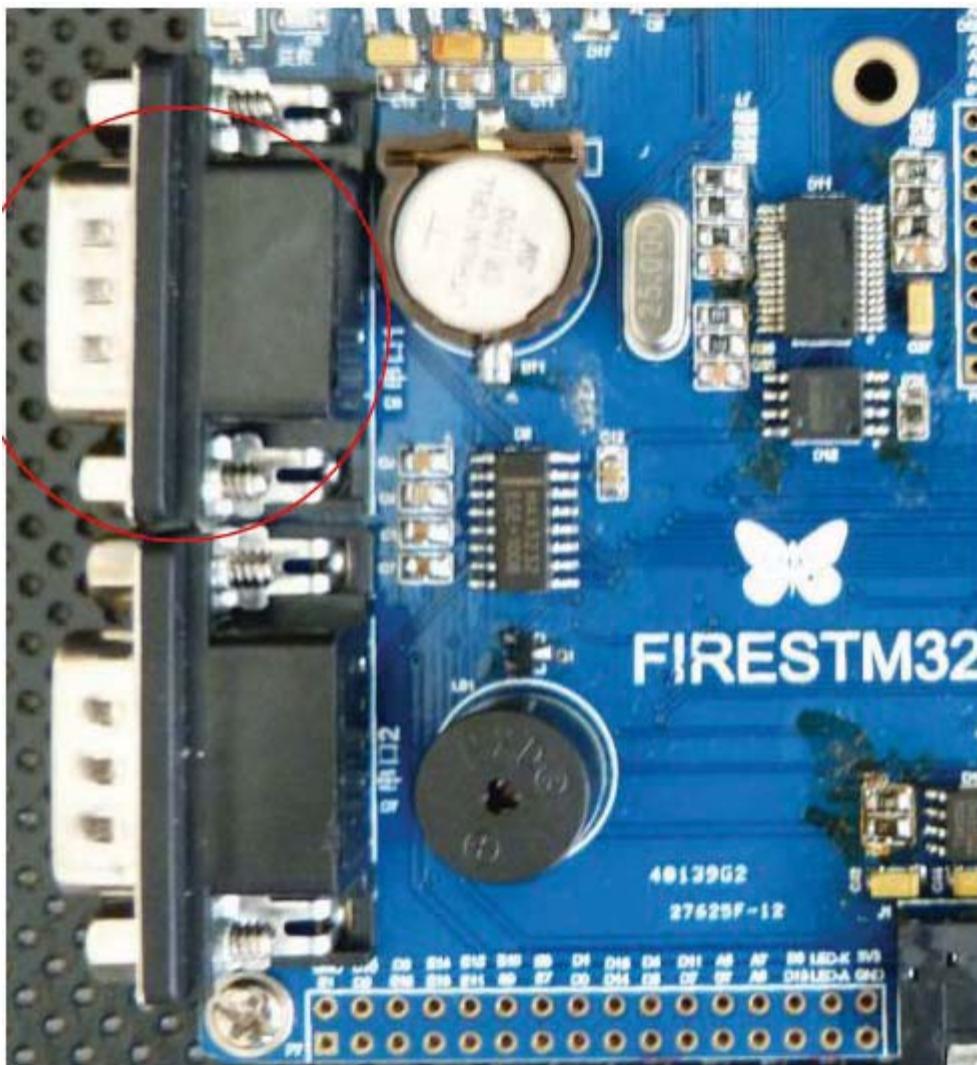
**这里要注意一点：**在程序下载到开发板之后，开发板要供电，JLINK 一端连开发板，另一端连 PC，这样程序才能运行。有些用户在下载程序之后，第二次用的时候只是给开发板供电，JLINK 的一端只连了开发板而没有连 PC，这样程序是不能工作的。要想只在供电的情况下要程序运行，只需把 JLINK 从开发板中拔掉即可，即只连电源，不接 JLINK，明白？^\_^。



---

串口下载->

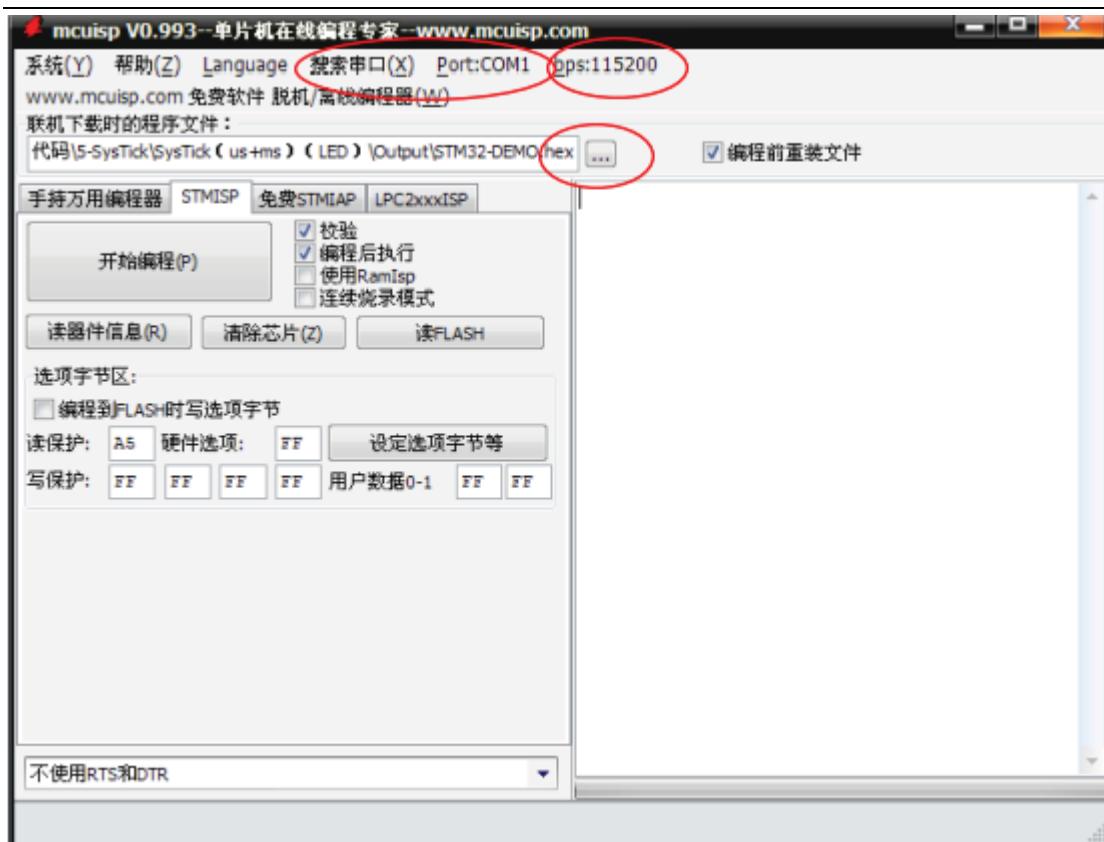
1->给开发板供电(DC5V), 拔掉 JLINK, 插上串口线(注意是两头都是母的交叉线), 接的是串口 1, 串口 2 是下载不了程序的, 再将下载模式开关打到串口下载模式。



在这里我们用的串口下载软件是 mcuisp, 这个一个绿色的软件, 可从网上自由下载, 我们也在光盘资料里面提供了这个软件。

打开 mcuisp, mcuisp 是很智能的, 只要开发板上电且接好了串口, 它就会自动搜索串口, 我这里用的是电脑主板后面的串口, 这个串口都会被默认认为是串口 1。假如你是笔记本用户, 用的是 USB 转串, 那么端口号可能就不是 COM1, 需要自己查看。

设置波特率为 115200。选择要下载的程序。在开发板自带的例程中, 可执行文件(hex 文件)都在工程目录下的 Output 这个文件下。



然后点击 **开始编程** 按钮，如果程序下载成功后则会打印出下面红色框中的信息。





程序下载成功之后，可是在开发板上看不到实验现象呀。怎么办，是不是出什么问题了呀？这是因为我们是通过串口将我们的程序烧写到 flash 里面去了，而我们想要从 flash 里面执行我们的程序的话，就需要将我们的 JLINK-串口 下载模式选择开关打到 JLINK 这个位置，然后按下我们的复位按键就可以看到实验想象了。

在我们点击 **开始编程** 按钮时，还会出现 mcuisp 一直处于连接的状态，导致程序下载不了，如下截图：



解决的方法是只需我们按一下开发板中的复位按键即可。

在没有 JLINK 的情况下，我们可以选择用串口下载。但是用串口下载有一些缺点：下载速度慢、需要多次选择 JLINK-串口开关、不能够进行硬件在线仿真。鉴于这三个缺点我们还是建议用户买一个 JLINK，特别是在我们在调试程序的时候，JLINK 的在线调试功能给予我们的帮助可是非常大的。

实验讲解完毕，野火祝大家学习愉快^\_^。



## LED ( 流水灯 )实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: 该实验讲解了如何运用 ST 的库来操作 I/O 口, 使 I/O 口产生置位(1)和复位(0)信号, 从而来控制 LED 的亮灭。

硬件连接: PC3 – LED1

PC4 – LED2

PC5 – LED3

库文件 : [startup/start\\_stm32f10x\\_hd.c](#)

[CMSIS/core\\_cm3.c](#)

[CMSIS/system\\_stm32f10x.c](#)

[FWlib/stm32f10x\\_gpio.c](#)

[FWlib/stm32f10x\\_rcc.c](#)

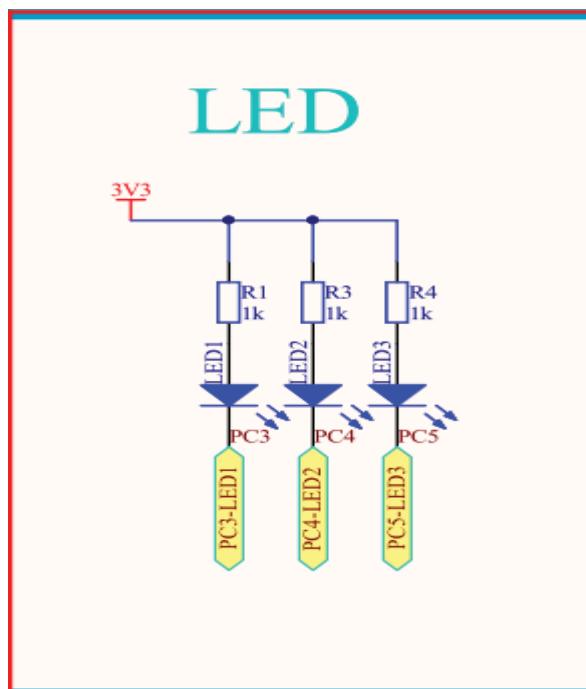
用户文件: [USER/main.c](#)

[USER/stm32f10x\\_it.c](#)

[USER/led.c](#)



野火 STM32 开发板 LED 硬件原理图:



### 实验讲解开始->

LED 实验用到了 GPIO 和 RCC 这两个片上外设，所以在操作 I/O 之前我们需要把关于这两个外设的库文件包含进来，`stm32f10x_rcc.c` 用于配置系统时钟，是每个外设都需要用到的库文件，`stm32f10x_gpio.c` 用于操作 I/O。在添加完这两个库文件之后就编译的话会出错，因为每个外设库对应于一个 `stm32f10x_xxx.c` 文件的同时还对应着一个 `stm32f10x_xxx.h` 头文件，只有我们把这个头文件包含进来了才能够使用这些外设库。在我们的库文件中有个专门的文件 `stm32f10x_conf.h` 来管理这些头文件，`stm32f10x_conf.h` 源码如下：

```
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_wwdg.h" */
24. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
on to CMSIS functions) */
```

这是没有修改过的代码， 默认情况下所有外设的库文件都被注释掉了。当我们需要用到哪个外设驱动时,直接把对应的注释去掉即可， 非常方便。如本 LED 实验中我们用到了 RCC 跟 GPIO 这两个外设，所有我们应修改成如下所示：

```
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_wwdg.h" */
24. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
on to CMSIS functions) */
```

到这里我们就可以用库自带的函数来操作 I/O 口了，这时我们可以编译一下，会发现既没有 Warning 也没有 Error。

```
Build Output
compiling main.c...
compiling stm32f10x_it.c...
compiling led.c...
compiling stm32f10x_gpio.c...
compiling stm32f10x_rcc.c...
compiling system_stm32f10x.c...
linking...
Program Size: Code=2324 RO-data=336 RW-data=20 ZI-data=516
FromELF: creating hex file...
"..\Output\STM32-DEMO.axf" - 0 Error(s), 0 Warning(s).
```

前期工作搞定，接下来我们就可以专心编写自己的应用程序了。我们把应用程序放在 **USER** 这个文件夹下，这个文件夹下至少包含了 **main.c**、**stm32f10x\_it.c**、**xxx.c**



这三个文件。其中我们的 `main` 函数就位于 `main.c` 这个 c 文件中，`main` 函数只是用来测试我们的应用程序，至于应用程序的具体实现，我们放在了 `xxx.c` 这个文件中，`xxx` 是应用程序的名字，用户可自由命名。程序的实现和应用分开在不同的文件中，这样就实现了很好的封装性。`stm32f10x_it.c` 为我们提供了 M3 所有中断函数的入口，默认情况下这些中断服务程序都为空，等到用到的时候需要用户自己编写。所以我们把 `stm32f10x_it.c` 包含到 `USER` 这个目录就是。

现在我们看看在 `main` 函数中我们具体做了什么吧，瞧瞧我们是如何用库函数来点亮 LED 的。

```
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.    /* LED 端口初始化 */
12.    LED_GPIO_Config();
13.    while (1)
14.    {
15.        LED1( ON );           // 亮
16.        Delay(0xFFFFFEF);
17.        LED1( OFF );         // 灭
18.
19.        LED2( ON );
20.        Delay(0xFFFFFEF);
21.        LED2( OFF );
22.
23.        LED3( ON );
24.        Delay(0xFFFFFEF);
25.        LED3( OFF );
26.    }
27. }
```

在 `main` 函数中我们首先调用库函数 `SystemInit()`；将系统时钟配置为 72M，这一步非常重要，倘若我们没配置好系统时钟就去操作片上外设的话，那一切都将是徒劳的。在数字电路中，时钟之于 CPU 的作用就如心脏之于人的作用一样，此消彼亡。

`SystemInit()` 在 `system_stm32f10x.c` 中实现：

```
1. void SystemInit (void)
2. {
3.     /*!< RCC system reset(for debug purpose) */
4.     /*!< Set HSION bit */
5.     RCC->CR |= (uint32_t)0x00000001;
6.     /*!< Reset SW[1:0], HPRE[3:0], PPRE1[2:0], PPRE2[2:0], ADCPRE[1:0] and MCO[2:0] bits */
7.     RCC->CFGR &= (uint32_t)0xF8FF0000;
8.     /*!< Reset HSEON, CSSON and PLLON bits */
9.     RCC->CR &= (uint32_t)0xFEF6FFFF;
```



```
10. /*!< Reset HSEBYP bit */
11. RCC->CR &= (uint32_t)0xFFFFBFFF;
12. /*!< Reset PLLSRC, PLLXTPRE, PLLMUL[3:0] and USBPRE bits */
13. RCC->CFGR &= (uint32_t)0xFF80FFFF;
14. /*!< Disable all interrupts */
15. RCC->CIR = 0x00000000;
16.
17. /*!< Configure the System clock frequency, HCLK, PCLK2 and PCLK1 prescalers */
18. /*!< Configure the Flash Latency cycles and enable prefetch buffer */
19. SetSysClock();
20. }
```

SystemInit() 调用了 SetSysClock(), SetSysClock() 源码如下:

```
1. static void SetSysClock(void)
2. {
3. #ifdef SYSCLK_FREQ_HSE
4.     SetSysClockToHSE();
5. #elif defined SYSCLK_FREQ_20MHz
6.     SetSysClockTo20();
7. #elif defined SYSCLK_FREQ_36MHz
8.     SetSysClockTo36();
9. #elif defined SYSCLK_FREQ_48MHz
10.    SetSysClockTo48();
11. #elif defined SYSCLK_FREQ_56MHz
12.    SetSysClockTo56();
13. #elif defined SYSCLK_FREQ_72MHz
14.    SetSysClockTo72();
15. #endif
16.
17. /*!< If none of the define above is enabled, the HSI is used as System clock
18. source (default after reset) */
19. }
```

SetSysClock() 函数根据不同的宏来实现不同的系统时钟配置。在 [system\\_stm32f10x.c](#) 文件的开头 我们定义了宏 `SYSCLK_FREQ_72MHz` 来调用 `SetSysClockTo72()`; 函数将我们的系统时钟配置为 **72M**。即我们可以通过不同的宏编译选项来设置我们的系统时钟，在本实验我们设置为 **72M**.

```
1. /*!< Uncomment the line corresponding to the desired System clock (SYSCLK)
2. frequency (after reset the HSI is used as SYSCLK source) */
```



```
3. // #define SYSCLK_FREQ_HSE      HSE_Value
4. // #define SYSCLK_FREQ_20MHz   20000000
5. // #define SYSCLK_FREQ_36MHz   36000000
6. // #define SYSCLK_FREQ_48MHz   48000000
7. // #define SYSCLK_FREQ_56MHz   56000000
8. #define SYSCLK_FREQ_72MHz   72000000
```

其中 [SetSysClockTo72\(\)](#) 函数就是最底层的函数了，那些跟寄存器打交道的活都是由这厮来完成的，如果大家想知道我们的系统时钟是如何配置成 72M 的话，可以研究这个函数的源码。但我想说的是大可不必这样，我们应该抛开传统的，直接跟寄存器打交道来学单片机的方法，直接用 ST 的库给我们提供的上层接口，这样会简化我们很多的工作，还能提高我们开发产品的效率，何乐而不为呢？[SetSysClockTo72\(\)](#) 将外部 8M 的晶振通过 PLL 配置系统时钟为 72M，具体代码实现如下：

```
1. #elif defined SYSCLK_FREQ_72MHz
2. /**
3.  * @brief Sets System clock frequency to 72MHz and configure HCLK, PCLK2
4.  *        and PCLK1 prescalers.
5.  * @param None.
6.  * @arg None.
7.  * @note : This function should be used only after reset.
8.  * @retval value: None.
9. */
10. static void SetSysClockTo72(void)
11. {
12.     __IO uint32_t StartUpCounter = 0, HSEStatus = 0;
13.
14.     /*!< SYSCLK, HCLK, PCLK2 and PCLK1 configuration -----
15.      */
16.     /*!< Enable HSE */
17.     RCC->CR |= ((uint32_t)RCC_CR_HSEON);
18.
19.     /*!< Wait till HSE is ready and if Time out is reached exit */
20.     do
21.     {
22.         HSEStatus = RCC->CR & RCC_CR_HSERDY;
23.         StartUpCounter++;
24.     } while((HSEStatus == 0) && (StartUpCounter != HSEStartUp_TimeOut));
25.
26.     if ((RCC->CR & RCC_CR_HSERDY) != RESET)
27.     {
28.         HSEStatus = (uint32_t)0x01;
29.     }
30.     else
31.     {
32.         HSEStatus = (uint32_t)0x00;
33.     }
34.     if (HSEStatus == (uint32_t)0x01)
35.     {
36.         /*!< Enable Prefetch Buffer */
37.         FLASH->ACR |= FLASH_ACR_PRFTBE;
38.
39.         /*!< Flash 2 wait state */
40.         FLASH->ACR &= ((uint32_t)((uint32_t)~FLASH_ACR_LATENCY));
41.         FLASH->ACR |= (uint32_t)FLASH_ACR_LATENCY_2;
42.
43.         /*!< HCLK = SYSCLK */
44.         RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;
45.
```



```
46.     /*!< PCLK2 = HCLK */
47.     RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;
48.
49.     /*!< PCLK1 = HCLK */
50.     RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV2;
51.
52.     /*!< PLLCLK = 8MHz * 9 = 72 MHz */
53.     RCC-
>CFGREG &= (uint32_t)((RCC_CFGR_PLLSRC | RCC_CFGR_PLLXTPRE | RCC_CFGR_PLLMULL));
54.     RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC | RCC_CFGR_PLLMULL9);
55.     /*!< Enable PLL */
56.     RCC->CR |= RCC_CR_PLLON;
57.
58.     /*!< Wait till PLL is ready */
59.     while((RCC->CR & RCC_CR_PLLRDY) == 0)
60.     {
61.     }
62.     /*!< Select PLL as system clock source */
63.     RCC->CFGR &= (uint32_t)((RCC_CFGR_SW));
64.     RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;
65.     /*!< Wait till PLL is used as system clock source */
66.     while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) != (uint32_t)0x08)
67.     {
68.     }
69. }
70. else
71. { /*!< If HSE fails to start-up, the application will have wrong clock
72.      configuration. User can add here some code to deal with this error */
73.     /*!< Go to infinite loop */
74.     while (1)
75.     {
76.     }
77. }
78. }
79. #endif
```

总之，我们只需在主函数中调用 [SystemInit\(\)](#) 即可，其他底层的寄存器操作全部交给库去搞定，我们只管一心开发我们的应用程序即可。

在配置好系统时钟之后，紧接着就配置 LED 的控制 I/O，由 main 函数中的 [LED\\_GPIO\\_Config\(\)](#) 完成，[LED\\_GPIO\\_Config\(\)](#) 的具体实现是在 led.c 这个文件中，是由用户编写的，具体代码如下：/\*

```
1.  * 函数名: LED_GPIO_Config
2.  * 描述 : 配置 LED 用到的 I/O 口
3.  * 输入 : 无
4.  * 输出 : 无
5.  */
6. void LED_GPIO_Config(void)
7. {
8.     GPIO_InitTypeDef GPIO_InitStructure;
9.     RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE );
10.
11.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5;
12.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
```



```
13.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
14.     GPIO_Init(GPIOC, &GPIO_InitStructure);
15.
16.     GPIO_SetBits(GPIOC, GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
17. }
```

在 `LED_GPIO_Config()` 这个函数中首先使能 `GPIOC` 的端口时钟，在 STM32 中，每个外设都有独立的时钟，要想外设工作除了要使能系统时钟(由 `SystemInit()` 这个函数实现)之外，还有使能响应的 I/O 时钟，假如用到了 I/O 的第二功能的话还要使能第二功能的时钟。关于外设时钟的配置的函数主要两个，它们均在 `stm32f10x_rcc.c` 这个文件中实现，分别是：

```
1. /* 使能高速外设 */
2. void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState)

3. /* 使能低速外设 */
4. void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState)
```

一个用于使能高速外设的时钟，另一个用于使能低速外设的时钟，至于外设是高速还是低速，可以找到这两个函数的源码，去阅读函数的用法即可知道。在这里我们先记住 STM32 的所有 I/O 口都是高速的 I/O 口。

```
1. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5;
2. GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
3. GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
4. GPIO_Init(GPIOC, &GPIO_InitStructure);
```

前三句的功能分别是选定了具体的 I/O 口为 3、4、5，I/O 的工作模式为推挽输出，I/O 的工作速率为 50MHZ。最后通过调用库函数

`GPIO_Init(GPIOC, &GPIO_InitStructure);` 将刚刚配置好的信息写到 `GPIOC` 中。如果是其他 I/O 口的话只需将 `GPIOC` 改成相应的 I/O 即可，如 `GPIOD`、`GPIOA` 等。STM32 的 I/O 的工作模式共有 8 中，这 8 中模式在 `GPIOMode_TypeDef` 这个数据结构中定义，`GPIOMode_TypeDef` 这个数据结构在 `stm32f10x_gpio.h` 中定义。

```
1. typedef enum
2. { GPIO_Mode_AIN = 0x0,           // 模拟输入
```



```
3.     GPIO_Mode_IN_FLOATING = 0x04,      // 浮空输入
4.     GPIO_Mode_IPD = 0x28,                // 下拉输入
5.     GPIO_Mode_IPU = 0x48,                // 上拉输入
6.     GPIO_Mode_Out_OD = 0x14,              // 开漏输出
7.     GPIO_Mode_Out_PP = 0x10,              // 推挽输出
8.     GPIO_Mode_AF_OD = 0x1C,              // 复用开漏输出
9.     GPIO_Mode_AF_PP = 0x18              // 复用推挽输出

}GPIOMode_TypeDef;
```

其中用的最多的就是推挽输出，至于每种工作模式的区别可以去查 ST 官网的资料。

关于 I/O 的工作速率也有几种选择，一般情况下只有在 I/O 工作在输出模式下才会设置速率，在输入模式时一般不考虑。

```
1.  typedef enum
2.  {
3.     GPIO_Speed_10MHz = 1,
4.     GPIO_Speed_2MHz,
5.     GPIO_Speed_50MHz
6. }GPIOSpeed_TypeDef;
```

一般情况下都设置成 `GPIO_Speed_50MHz`。

至此，初始化一个普通 I/O 口所要做的工作已经完成了。现在我们再来整理下初始化一个 I/O 的步骤：

```
1-> GPIO_InitTypeDef GPIO_InitStruct 定义一个结构体，用于初始化 I/O。
2-> 使能 I/O 时钟
3-> 选定具体 I/O 端口
4-> 设定 I/O 的工作模式
5-> 设定 I/O 的工作速率，如果是输入的话则可不用设置，I/O 的工作速率一般针对的是 I/O 的输出模式
6-> 调用函数 void GPIO_Init(GPIO_TypeDef* GPIOx,
                           GPIO_InitTypeDef* GPIO_InitStruct)
```

将上面配置好的信息写到相应的 I/O 中。



现在我们就可以利用库函数来操作 I/O 口产生置位(1)和复位(0)信号了，这主要通过以下两个函数来实现，这两个函数在 [stm32f10x\\_gpio.c](#) 中实现。实际上关于 I/O 操作的函数都在这个文件中实现。

```
1. /* 置位 */
2. void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
3. /* 复位 */
4. void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

在 `LED_GPIO_Config()` 函数的最后，我们调用

```
GPIO_SetBits(GPIOC, GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
```

将所有 LED 都关掉。以为 LED 的正极接的是 VCC，负极接的是 I/O 口，是属于低电平控制，当 I/O 口都输出高电平(置位)时，LED 就全灭了。

在 `stm32f10x_gpio.h` 这个头文件中我们可以看到有关 I/O 操作的所有函数的原型。

```
1. void GPIO_DeInit(GPIO_TypeDef* GPIOx);
2. void GPIO_AFIODeInit(void);
3. void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
4. void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
5. uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
6. uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
7. uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
8. uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
9. void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
10. void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
11. void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal);

12. void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
13. void GPIO_PinLockConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
14. void GPIO_EventOutputConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);

15. void GPIO_EventOutputCmd(FunctionalState NewState);
16. void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
17. void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
```



有关每个函数的具体用法可以到 `stm32f10x_gpio.c` 中每个函数的实现中查看。在接下来的教程中用到这些函数时会一一讲解，这里先不详述。

主函数的最后是一个无限循环，实现了 LED 以固定的频率闪烁，延时是通过一个简单的延时函数来实现，暂时没用到定时器，有关定时器的应用后续的教程会讲到。

```
1. while (1)
2. {
3.     LED1( ON );           // 亮
4.     Delay(0xFFFFEF);
5.     LED1( OFF );         // 灭
6.
7.     LED2( ON );
8.     Delay(0xFFFFEF);
9.     LED2( OFF );
10.
11.    LED3( ON );
12.    Delay(0xFFFFEF);
13.    LED3( OFF );
14. }
```

其中控制 LED 亮灭的函数(即 I/O 的置位和复位)已在 `led.h` 中实现为一个宏，这样使得代码比较简洁，也提高了代码的可移植性。

```
1. /* the macro definition to trigger the led on or off
2. * 1 - off
3. - 0 - on
4. */
5. #define ON 0    // 亮
6. #define OFF 1   // 灭
7.
8. #define LED1(a) if (a) \
9.                 GPIO_SetBits(GPIOC,GPIO_Pin_3); \
10.                else \
11.                  GPIO_ResetBits(GPIOC,GPIO_Pin_3)
12.
13. #define LED2(a) if (a) \
14.                 GPIO_SetBits(GPIOC,GPIO_Pin_4); \
15.                else \
16.                  GPIO_ResetBits(GPIOC,GPIO_Pin_4)
17.
18. #define LED3(a) if (a) \
19.                 GPIO_SetBits(GPIOC,GPIO_Pin_5); \
20.                else \
21.                  GPIO_ResetBits(GPIOC,GPIO_Pin_5)
```

### 实验现象->

将野火 STM32 开发板供电(DC5V)，插上 JLINK，将编译好的程序下载到开发板，即可看到板子上的 3 个 LED 以固定的频率轮流闪烁。

实验讲解完毕，野火祝大家学习愉快^\_^。



## SysTick (系统滴答定时器) 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: 3 个 LED 在 SysTick 的控制下, 以 500ms 的频率闪烁。

硬件连接: 无

库文件 : startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/led.c

USER/ SysTick.c

### SysTick 定时器简介->

SysTick 定时器被捆绑在 NVIC 中, 用于产生 SysTick 异常 (异常号: 15)。在以前, 操作系统还有所有使用了时基的系统, 都必须要一个硬件定时器来产生需要的“滴答”中断, 作为整个系统的时基。滴答中断对操作系统尤其重要。例如, 操作系统可以为多个任务许以不同数目的时间片, 确保没有一个任务能霸占系统; 或者把每个定时器周期的某个时间范围赐予特定的任务等, 还有操作系统提供的各种定时功



---

能，都与这个滴答定时器有关。因此，需要一个定时器来产生周期性的中断，而且最好还让用户程序不能随意访问它的寄存器，以维持操作系统“心跳”的节律。

Cortex-M3 处理器内部包含了一个简单的定时器。因为所有的 CM3 芯片都带有这个定时器，软件在不同 CM3 器件间的移植工作就得以化简。该定时器的时钟源可以是内部时钟（FCLK，CM3 上的自由运行时钟），或者是外部时钟（CM3 处理器上的 STCLK 信号）。不过，STCLK 的具体来源则由芯片设计者决定，因此不同产品之间的时钟频率可能会大不相同。因此，需要检视芯片的器件手册来决定选择什么作为时钟源。[在 STM32 中 SysTick 以 FCLK 作为运行时钟](#)。

SysTick 定时器能产生中断，CM3 为它专门开出一个异常类型，并且在向量表中有它的一席之地。它使操作系统和其它系统软件在 CM3 器件间的移植变得简单多了，因为在所有 CM3 产品间，SysTick 的处理方式都是相同的。

SysTick 定时器除了能服务于操作系统之外，还能用于其它目的：如作为一个闹铃，用于测量时间等。[在本实验中会讲到 SysTick 用作闹铃和测量时间的功能](#)。

### 实验讲解->

我们从看 main 函数看起：

```
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.     /* config the sysclock to 72mhz */
10.    SystemInit();
11.    /* config led gpio */
12.    LED GPIO Config();
13.    /* 配置 SysTick 为 10us 中断一次 */
14.    SysTick_Init();
15.
16.    while (1)
17.    {
18.        //SysTick->CTRL = 1 << SYSTICK_ENABLE;           // 使能滴答定时器
19.        LED1( 0 );
20.        Delay_us(50000);      // 50000 * 10us = 500ms
21.        LED1( 1 );
22.
23.        LED2( 0 );
24.        Delay_us(50000);      // 50000 * 10us = 500ms
25.        LED2( 1 );
26.
27.        LED3( 0 );
28.        Delay_us(50000);      // 50000 * 10us = 500ms
29.        LED3( 1 );
30.        //SysTick->CTRL = 0 << SYSTICK_ENABLE;           // 失能滴答定时器
31.    }
32. }
```



SystemInit(); 将我们的系统时钟配置为 72M。LED\_GPIO\_Config(); 配置与 LED 相关的 I/O，这在 LED 实验中已讲解过，这里不再详述。接下来我们重点看下 SysTick\_Init(); 这个函数，它是由用户在 SysTick.c 这个文件中实现的，其功能是启动系统滴答定时器 SysTick，并将 SysTick 配置为 10 us 中断一次：

```
1. /*
2.  * 函数名: SysTick_Init
3.  * 描述 : 启动系统滴答定时器 SysTick
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 外部调用
7. */
8. void SysTick_Init(void)
9. {
10.    /* SystemFrequency / 1000      1ms 中断一次
11.     * SystemFrequency / 100000    10us 中断一次
12.     * SystemFrequency / 1000000   1us 中断一次
13.     */
14.    if (SysTick_Config(SystemFrequency / 100000))
15.    {
16.        /* Capture error */
17.        while (1);
18.    }
19. }
```

SysTick\_Init() 函数又调用了库函数 SysTick\_Config(uint32\_t ticks)：

```
1. /**
2.  * @brief Initialize and start the SysTick counter and its interrupt.
3.  *
4.  * @param uint32_t ticks is the number of ticks between two interrupts
5.  * @return none
6.  *
7.  * Initialise the system tick timer and its interrupt and start the
8.  * system tick timer / counter in free running mode to generate
9.  * periodical interrupts.
10. */
11. static __INLINE uint32_t SysTick_Config(uint32_t ticks)
12. {
13.    /* Reload value impossible */
14.    if (ticks > SYSTICK_MAXCOUNT)  return (1);
15.    /* set reload register */
16.    SysTick->LOAD  = (ticks & SYSTICK_MAXCOUNT) - 1;
17.    /* set Priority for Cortex-M0 System Interrupts */
18.    NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
19.
20.    /* Load the SysTick Counter Value */
21.    SysTick->VAL   = (0x00);
22.
23.    /* Enable SysTick IRQ and SysTick Timer */
24.    SysTick->CTRL = (1 << SYSTICK_CLKSOURCE) |
25.                     (1 << SYSTICK_ENABLE) |
26.                     (1 << SYSTICK_TICKINT);
27.
28.    /* Function successful */
29.    return (0);
30. }
```

其中库函数 SysTick\_Config(uint32\_t ticks) 在库文件 core\_cm3.h 中实现。实际上真正开启 SysTick 定时器的就是这个函数，前面的 SysTick\_Init(void) 只是将这个函数封装起来了而已。



一旦我们调用这个函数，**SysTick** 定时器就被开启，按照设定好的定时周期递减计数，**SysTick** 定时器有一个 24 位的计数寄存器，当计数寄存器里面的值减为 0 时，就进入中断函数，当中断函数执行完毕之后由重新计时，如此循环，除非它被关闭，**SysTick** 定时器的开启和关闭由其控制及状态寄存器的第 0 位来控制，下面是具体的代码：

```
1. // 使能滴答定时器
2. SysTick->CTRL = 1 << SYSTICK_ENABLE;
3. // 失能滴答定时器
4. SysTick->CTRL = 0 << SYSTICK_ENABLE;
```

当我们调用 `SysTick_Config(uint32_t ticks)` 时，默认情况下已经开启了 **SysTick** 定时器。

**SysTick** 定时器的中断周期由 `uint32_t ticks` 这个参数决定，以下是几种常用的中断周期配置：

```
1. /* ticks 常取以下值 */
2. SystemFrequency / 1000           // 1ms 中断一次
3. SystemFrequency / 100000         // 10us 中断一次
4. SystemFrequency / 1000000        // 1us 中断一次
```

在本实验室中我们配置为 **10us** 中断一次。当中断到来时，我们需要编写相应的中断服务程序，中断程序在 `stm32f10x_it.c` 中实现：

```
1. /**
2.  * @brief This function handles SysTick Handler.
3.  * @param None
4.  * @retval : None
5. */
6. void SysTick_Handler(void)
7. {
8.     TimingDelay_Decrement();
9. }
```

在中断服务程序中我们调用了 `TimingDelay_Decrement()`；这个函数。

`TimingDelay_Decrement()` 是由用户编写的一个应用程序，在 `SysTick.c` 中实现：

```
1. /*
2.  * 函数名: TimingDelay_Decrement
3.  * 描述 : 获取节拍程序
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 在 SysTick 中断函数 SysTick_Handler() 调用
7. */
8. void TimingDelay_Decrement(void)
9. {
10.    if (TimingDelay != 0x00)
11.    {
```



```
12.     TimingDelay--;
13. }
14. }
```

最终我们就可以编写自己的延时函数了, `Delay_us()` 在 `sysTick.c` 中实现, 具体用法可看函数说明:

```
1.  /*
2.   * 函数名: Delay_us
3.   * 描述 : us 延时程序, 10us 为一个单位
4.   * 输入 : - nTime
5.   * 输出 : 无
6.   * 调用 : Delay_us( 1 ) 则实现的延时为 1 * 10us = 10us
7.   *       : 外部调用
8.   */
9. void Delay_us(__IO u32 nTime)
10. {
11.     TimingDelay = nTime;
12.     while(TimingDelay != 0);
13. }
```

在 `main` 函数的最后我们使 LED 工作在一个无限循环中:

```
1. while (1)
2. {
3.     //SysTick->CTRL = 1 << SYSTICK_ENABLE;           // 使能滴答定时器
4.     LED1( 0 );
5.     Delay_us(50000);        // 50000 * 10us = 500ms
6.     LED1( 1 );
7.
8.     LED2( 0 );
9.     Delay_us(50000);        // 50000 * 10us = 500ms
10.    LED2( 1 );
11.
12.    LED3( 0 );
13.    Delay_us(50000);        // 50000 * 10us = 500ms
14.    LED3( 1 );
15.    //SysTick->CTRL = 0 << SYSTICK_ENABLE;           // 失能滴答定时器
16. }
```

以上所讲就是 `SysTick` 定时器的定时功能, 几个函数直接调来调去, 看上去有点乱, 大家慢慢体会下吧, 不难。

下面我们再开看看 `SysTick` 的测量时间的功能。

当我们开启 `SysTick` 定时器后, 定时器开始工作, 我们可以定义一个变量, 在定时器进入中断时, 这个变量就`++`, 当我们关闭定时器后, 将变量的数值乘于定时器的中断周期就等于了测量时间。这个功能我一般用于测量程序的运行时间, 特别是涉及到算法的程序, 这对于我优化算法是有非常大的帮助。假如你的算法的是 `us` 级别的, 那么 `SysTick` 就应该设定为 `us` 级中断, 如果是 `ms` 级别的, 就将 `SysTick` 设定为 `ms` 级中断。



---

实验现象->

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 将编译好的程序下载到开发板, 即可看到板载的 3 个 LED 以 500ms 的频率闪烁。

实验讲解完毕, 野火祝大家学习愉快^\_^。



## USART1（串口 1）实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

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)

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](#)

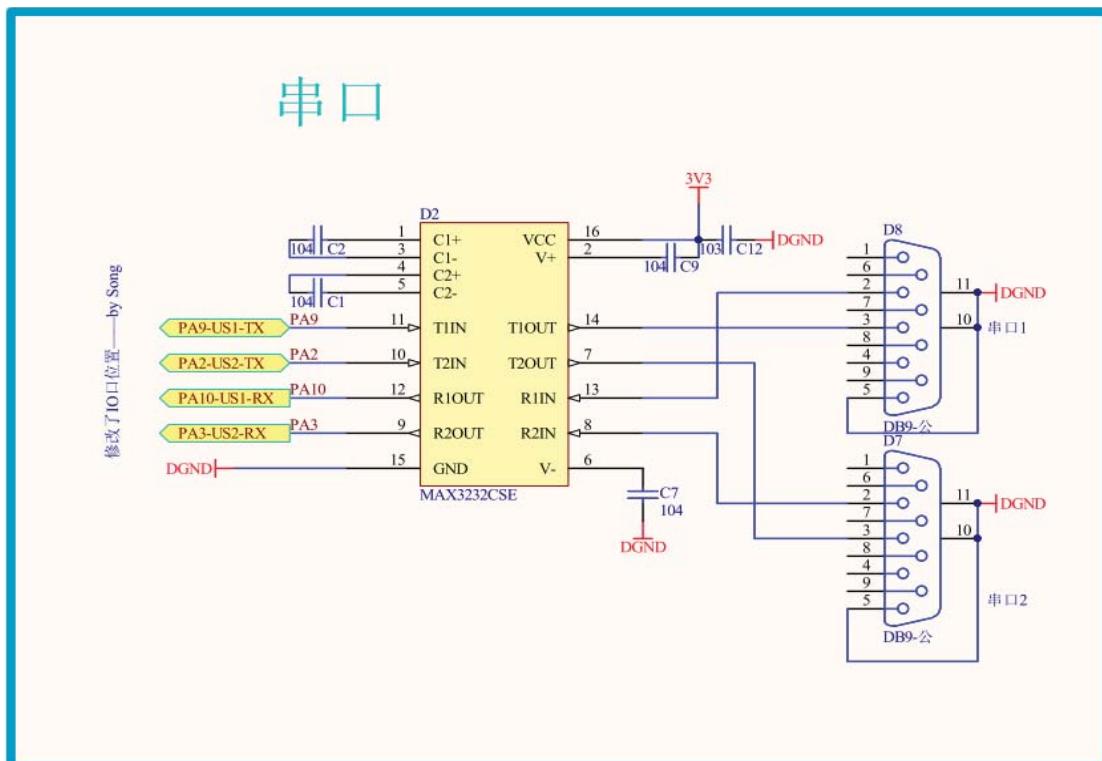
用户文件: [USER/main.c](#)

[USER/stm32f10x\\_it.c](#)

[USER/usart1.c](#)



野火 STM32 开发板串口硬件原理图:



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。但对于我个人来说我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 来显示我们的状态，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。

### 实验讲解->

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`，所以我们先要把这个库文件添加进来，并在 `stm32f10x_conf.h` 中把相应的头文件的注释去掉，如下所示：

```
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_wwdg.h" */
24. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
   on to CMSIS functions) */
```

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.     /* 配置系统时钟为 72M */
10.    SystemInit();
11.    /* USART1 config 115200 8-N-1 */
```



```
12.     USART1_Config();
13.     printf("\r\n this is a printf demo \r\n");
14.
15.     USART1_printf(USART1, "\r\n This is a USART1_printf demo \r\n");
16.     USART1_printf(USART1, "\r\n ("__DATE__" - " __TIME__") \r\n");
17.     while (1)
18.     { } // 无限循环
19. }
20.
```

首先调用库函数 `SystemInit()` 启动 CPU 的心脏，配置系统时钟为 72M。

紧接着 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 `uart1` 的 I/O
- 3- 配置好了 `uart1` 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

`USART1_Config()` 在 `uart1.c` 这个文件中实现：

```
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 */
```

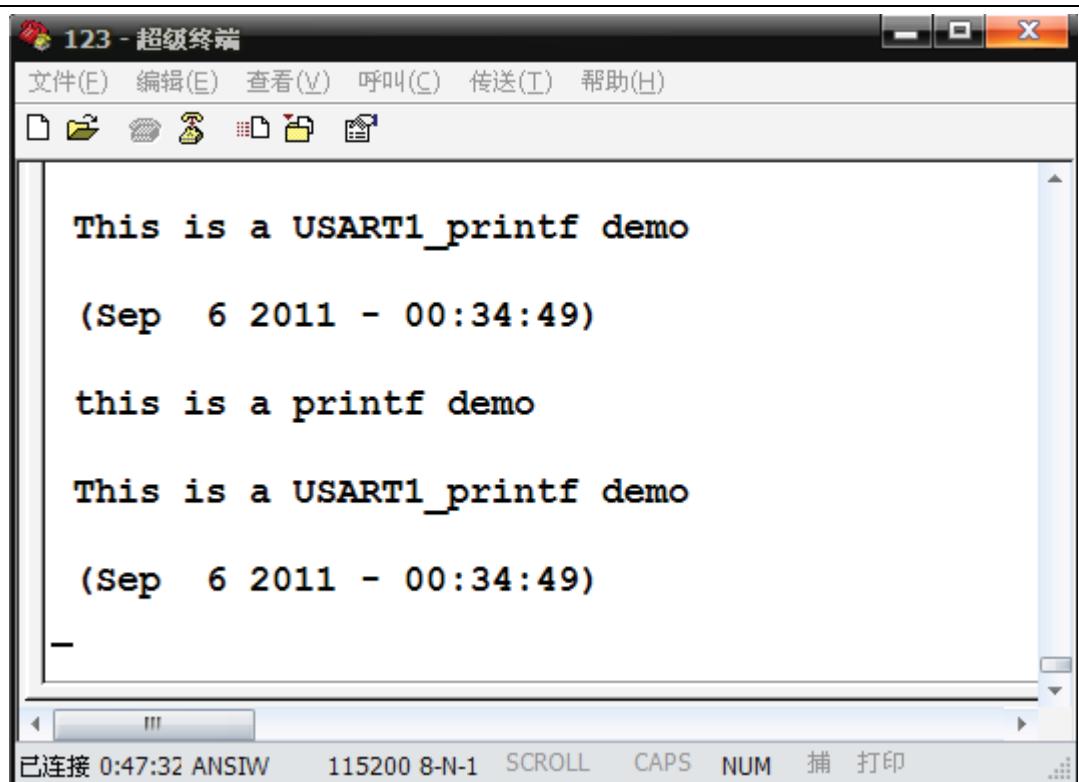


```
18.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
19.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
20.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21.     GPIO_Init(GPIOA, &GPIO_InitStructure);
22.     /* Configure USART1 Rx (PA.10) as input floating */
23.     GPIO_InitStructure.GPIO_Pin = 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_InitStructureUSART_BaudRate = 115200;
29.     USART_InitStructureUSART_WordLength = USART_WordLength_8b;
30.     USART_InitStructureUSART_StopBits = USART_StopBits_1;
31.     USART_InitStructureUSART_Parity = USART_Parity_No ;
32. USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
33.
34.     USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx;
35.     USART_Init(USART1, &USART_InitStructure);
36.     USART_Cmd(USART1, ENABLE);
```

然后我们就通过下面的三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. USART1_printf(USART1, "\r\n This is a USART1_printf demo \r\n");
4.
5. 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，这部分的工作是由 `fputc(int ch, FILE *f)` 这个函数来完成的，这个函数在 `uart.c` 中实现：

```
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.
14. return (ch);
15. }
```



为了能够读懂 `fputc(int ch, FILE *f)` 里面的代码，我们需要了解下 `uart` 的几个关键的寄存器位，这样我们就会明白串口是如何将数据发送出去的，做到知其然又知其所以然。

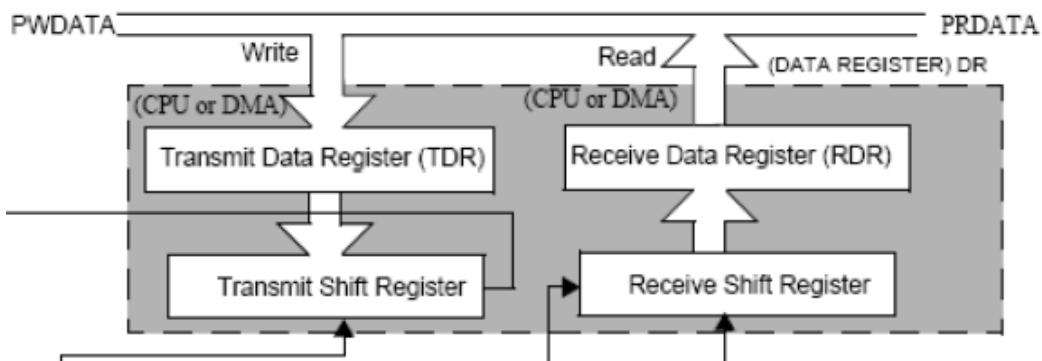
下面是 `USART_SR` 寄存器（状态寄存器）中的位 7、位 6、位 5 的截图。理解好这几位的功能的话就可以大概理解串口是如何发送数据的了。其实我们在用库写自己的应用程序的时候要关注的最多的也就是这几个状态位，至于其他更底层的功能实现我们交给库就行了，倘我们要深究的话，只要阅读库中相应的源码即可。[截图来自《stm32 中文参考手册》uart 这一章。](#)

位7	<b>TXE:</b> 发送数据寄存器空 当TDR寄存器中的数据被硬件转移到移位寄存器的时候，该位被硬件置位。如果USART_CR1寄存器中的TXEIE为1，则产生中断。 <a href="#">对USART_DR的写操作，将该位清零。</a> 0: 数据还没有被转移到移位寄存器； 1: 数据已经被转移到移位寄存器。 注意：单缓冲器传输中使用该位。
位6	<b>TC:</b> 发送完成 <a href="#">当包含有数据的一帧发送完成后，由硬件将该位置位。</a> 如果USART_CR1中的TCIE为1，则产生中断。 <a href="#">由软件序列清除该位(先读USART_SR，然后写入USART_DR)。</a> TC位也可以通过写入0来清除，只有在多缓存通讯中才推荐这种清除程序。 0: 发送还未完成； 1: 发送完成成。
位5	<b>RXNE:</b> 读数据寄存器非空 当RDR移位寄存器中的数据被转移到USART_DR寄存器中，该位被硬件置位。如果USART_CR1寄存器中的RXNEIE为1，则产生中断。 <a href="#">对USART_DR的读操作可以将该位清零。</a> RXNE位也可以通过写入0来清除，只有在多缓存通讯中才推荐这种清除程序。 0: 数据没有收到； 1: 收到数据，可以读出。

当 `TXE` 置位时，就表示发送数据寄存器 (`TDR`) 中的数据已经移到了发送移位寄存中，如果使能中断的话，就产生中断，在中断服务程序中我们就可以调用 `USART_SendData()` 函数将发送移位寄存中的数据通过串口发送出去，同时将该位清 0([硬件自动清 0](#))。当 `RXNE` 置位时，表示接收移位寄存器中的数据已经移到接收数据寄存器 (`RDR`) 中了，如果中断使能的话，则产生中断，在中断服务程序中我们可以调用 `USART_ReceiveData()` 将接收数据寄存器中的内容送到我们自定义的缓冲区中，比如数组，同时该位清 0([硬件自动清 0](#))。这只是描述了 `uart` 工作在中断模式的情况。



但 `fputc(int ch, FILE *f)` 函数里面用到的是查询的模式，其实道理还是一样的。我们先调用 `USART_SendData(USART1, (unsigned char) ch);` 将我们要发送的数据送到 `TDR` 中，之后我们就等待 `TXE` 置位，当 `TXE` 置位时就表示 `TDR` 中的数据转移到了发送移位寄存器中了，这时我们就不用管后面的工作了，发送移位寄存器中的数据会由串口硬件自动发送，如此循环，直到将我们要发送的数据全部发送完为止。



现在我们再使点劲，在我们的 `main.c` 文件中把 `stdio.h` 这个头文件包含进来，这样我们就可以使用 `printf()` 这个函数了。

趁热打铁，让我们再来看看

`USART1_Printf(USART_TypeDef* USARTx, uint8_t *Data,...)` 这个函数吧，它又调用了 `itoa(int value, char *string, int radix)` 函数。关于这两个函数的具体实现请看 `uart.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 `uart.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 */
```

```
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':
82.              USART_SendData(USARTx, 0x0d);
83.              Data++;
84.              break;
85.
86.            case 'n': //换行符
87.              USART_SendData(USARTx, 0xa);
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其妙的错误，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，我还是比较喜欢用 `USART1_printf()`。

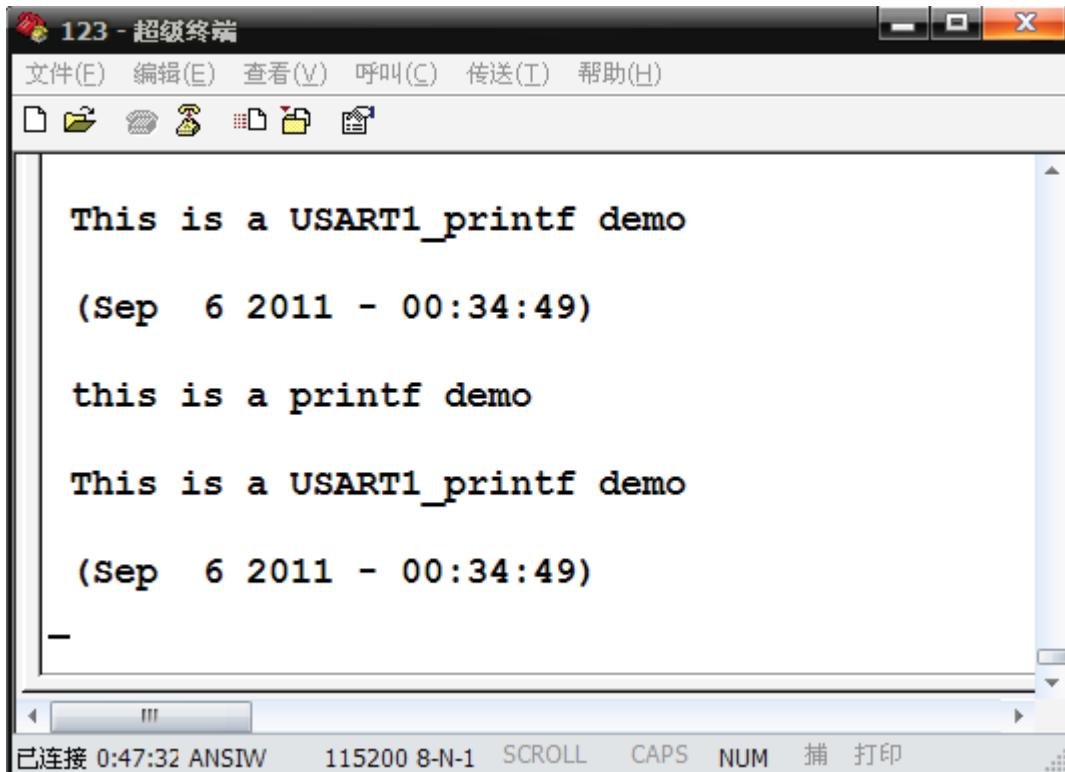
至此，关于串口的普通应用就差不多了，至于它的中断操作将会在接下来的教程中讲解。



---

### 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:



The screenshot shows a Windows-style terminal window titled "123 - 超级终端". The menu bar includes "文件(E)", "编辑(E)", "查看(V)", "呼叫(C)", "传送(T)", and "帮助(H)". Below the menu is a toolbar with icons for copy, paste, cut, find, and others. The main text area displays the following output:

```
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)
-
```

At the bottom of the window, status information is displayed: "已连接 0:47:32 ANSIW" and "115200 8-N-1 SCROLL CAPS NUM 捕 打印".

实验讲解完毕，野火祝大家学习愉快^\_^。



## Key (查询模式) 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述：PB0 连接到 key1，用扫描的方式查询是否有按键按下，key1 按下时，  
LED1 状态取反。

硬件连接： PB0 - key1

PB1 – key2

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/misc.c](#)

用户文件：[USER/main.c](#)

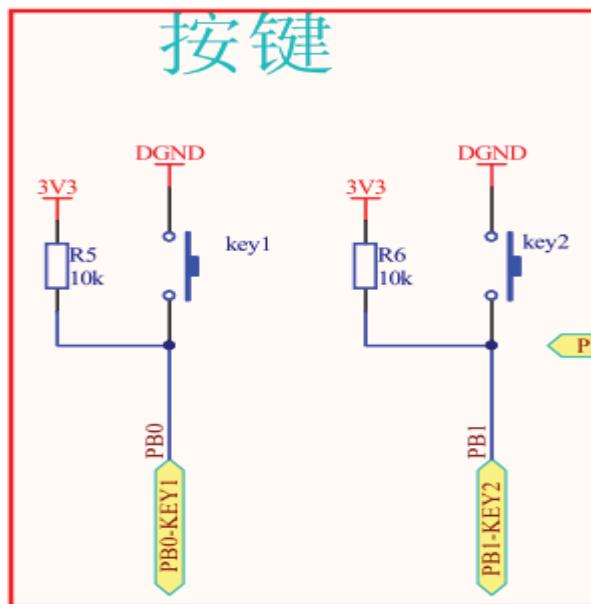
[USER/stm32f10x\\_it.c](#)

[USER/led.c](#)

[USER/key.c](#)



野火 STM32 开发板 按键 硬件原理图:



## 实验讲解-&gt;

我们从 main 函数开始分析:

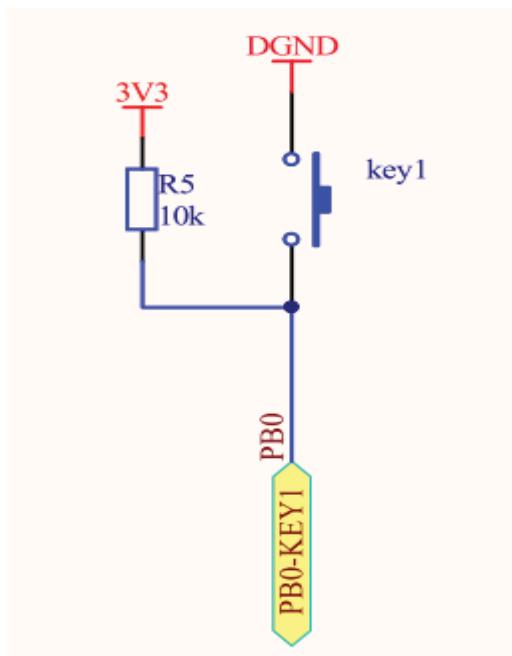
```
1. /**
2.  * @brief Main program.
3.  * @param None
4.  * @retval : None
5. */
6. int main(void)
7. {
8.     /* config the sysclock to 72m */
9.     SystemInit();
10.
11.    /* config the led */
12.    LED_GPIO_Config();
13.    LED1( ON );
14.    /*config key*/
15.    Key_GPIO_Config();
16.
17.
18.    while(1)
19.    {
20.        if( Key_Scan(GPIOB,GPIO_Pin_0) == KEY_ON )
21.        {
22.            /*LED1 反转*/
23.            GPIO_WriteBit(GPIOC, GPIO_Pin_3,
24.                           (BitAction)((1-
25.                           GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));}
26.    }
27. }
```



首先调用库函数 `SystemInit()` 将我们的系统时钟配置为 72MHz, `LED_GPIO_Config()` 配置 LED 用到的 I/O。这两个函数的具体讲解可参考前面的教程。现在我们分析一下 `Key_GPIO_Config()` 这个函数。

```
1.  /*
2.   * 函数名: Key_GPIO_Config
3.   * 描述  : 配置按键用到的 I/O 口
4.   * 输入  : 无
5.   * 输出  : 无
6.   */
7. void Key_GPIO_Config(void)
8. {
9.     GPIO_InitTypeDef GPIO_InitStructure;
10.
11.    /*开启按键端口（PB0）的时钟*/
12.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
13.
14.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
15.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
16.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
17.
18.    GPIO_Init(GPIOB, &GPIO_InitStructure);
19. }
```

跟 LED 的 I/O 端口设置类似，只是 PB0 的模式设置为适合按键处理的上拉输入方式。在外围电路上我们将 PB0 接到了 key1 上：



当按键没有按下时，PB0 始终为高，当按键按下时 PB0 变为低，从而 PB0 上产生一个低电平跳变，在按键扫描函数中可以检测到电平的变化，通过消抖处理，对按键消息进行确认。



```
1.  /*
2.   * 函数名: Key_Scan(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
3.   * 描述 : 检测是否有按键按下
4.   * 输入  : GPIOx: x 可以是 A, B, C, D 或者 E
5.           GPIO_Pin: 待读取的端口位
6.   * 输出  : KEY_OFF(没按下按键)、KEY_ON(按下按键)
7.   */
8. u8 Key_Scan(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
9. {
10.     /*检测是否有按键按下 */
11.     if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON )
12.     {
13.         /*延时消抖*/
14.         Delay(10000);
15.         if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON )
16.         {
17.             /*等待按键释放 */
18.             while(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON );
19.             return KEY_ON;
20.         }
21.         else
22.             return KEY_OFF;
23.     }
24.     else
25.         return KEY_OFF;
26. }
```

相信消抖的原理大家已经在学习其它单片机的时候已经了解，这里主要介绍一下 `Key_Scan()` 的形参。这个函数看起来是不是很像 ST 官方的库函数？其实这是野火自己写的一个用户函数。<sup>^\_^</sup>

这里利用了在 `stm32f10x.h` 文件中的数据 `GPIO` 类型定义。形参 `GPIO_Pin` 也是样实现的，所以如果在调用 `Key_Scan()` 函数时，把实参数改成 `GPIOB, GPIO_Pin_1`，就可以用 `Key-2` 来控制 `LED1` 啦，是不是很方便呢，利用官方的库，我们可以很方便地开发出这一类用户函数，这就是库的魅力呀！

```
1. typedef struct
2. {
3.     __IO uint32_t CRL;
4.     __IO uint32_t CRH;
5.     __IO uint32_t IDR;
6.     __IO uint32_t ODR;
7.     __IO uint32_t BSRR;
8.     __IO uint32_t BRR;
9.     __IO uint32_t LCKR;
10. } GPIO_TypeDef;
```

还有实现 LED 反转的代码

```
1. GPIO_WriteBit(GPIOC, GPIO_Pin_3,
2.                 (BitAction)((1-
3.                 GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
```



这是利用库函数来读出 LED1 端口状态再反转。其实还可以使用位带操作的方式，实现反转起来和使用 51 的端口一样简单直接比如：PA0=~PA0;

#### 实验现象->

将野火 STM32 开发板供电(DC5V)，插上 JLINK，将编译好的程序下载到开发板，LED1 亮，按下按键时 LED1 灭，再按下按键时 LED1 亮，如此循环。

实验讲解完毕，野火祝大家学习愉快^\_^。



## EXTI (key) 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: PB0 连接到 key1, PB0 配置为线中断模式, key1 按下时, 进入线中断处理函数, LED1 状态取反。

硬件连接: PB0 - key1

PB1 – key2

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\\_exti.c](#)

[FWlib/misc.c](#)

用户文件: [USER/main.c](#)

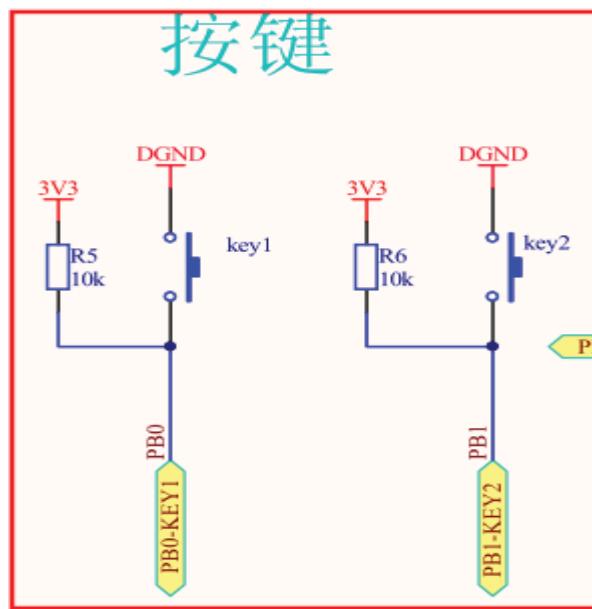
[USER/stm32f10x\\_it.c](#)

[USER/led.c](#)

[USER/exti.c](#)

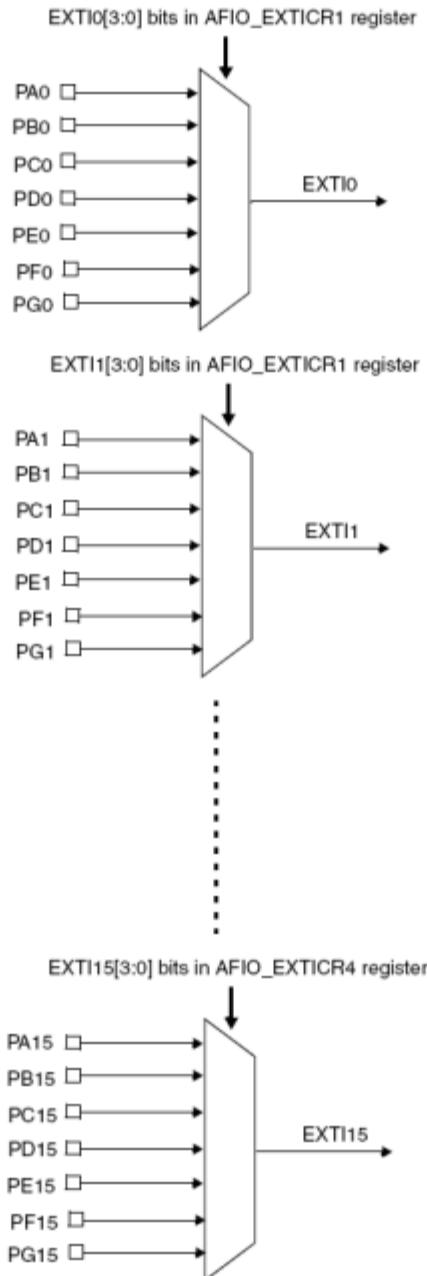


野火 STM32 开发板 按键 硬件原理图:



### EXTI 简介->

Stm32 单片机的所有 I/O 端口都可以配置为 EXTI 中断模式，用来捕捉外部信号，可以配置为下降沿中断，上升沿中断和上升下降沿中断这三种模式。他们以下图的方式连接到 16 个外部中断/事件线上：截图来自《stm32 中文参考手册》



PA0~PG0 连接到 EXTI0、PA1~PG1 连接到 EXTI1、.....、PA15~PG15 连接到 EXTI15。这里大家要注意的是：PAx~PGx 端口的中断事件都连接到了 EXTIx，即同一时刻 EXTIx 只能相应一个端口的事件触发，不能够同一时间响应所有端口的事件，但可以分时复用。EXTI 普通的应用就是接一个按键，用来检测按键。我们也可以用它来检测摄像头的行信号和场信号，在摄像头中的应用我们会在摄像头 OV7725 这个教程中讲到。



---

## 实验讲解->

我们从 main 函数开始分析:

```
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.    /* config the led */
12.    LED_GPIO_Config();
13.    LED1( ON );
14.
15.    /* exti line config */
16.    EXTI_PB0_Config();
17.
18.    /* wait interrupt */
19.    while(1)
20.    {
21.    }
22. }
```

首先调用库函数 `SystemInit()`, 将我们的系统时钟配置为 72MHz.

`LED_GPIO_Config()`; 配置 LED 用到的 I/O。这两个函数的具体讲解可参考前面的教程。

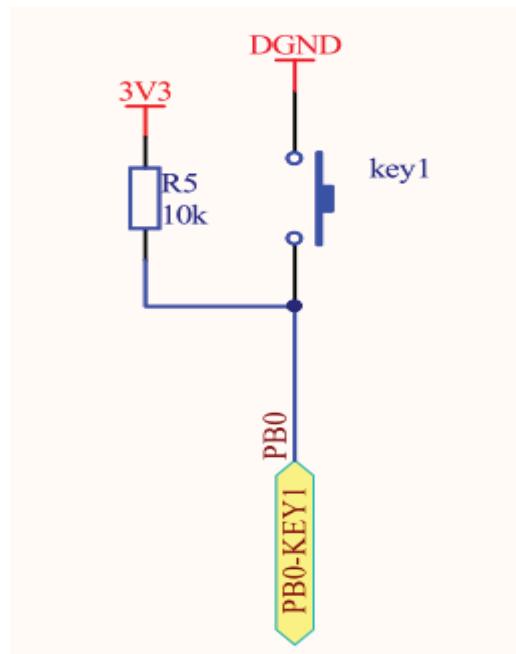
现在我们重点分析下 `EXTI_PB0_Config()`; 这个函数, 这是一个由用户在 `exti.c` 文件中实现的函数, 主要完成了如下功能, 这也是一般配置一个 I/O 为 EXTI 中断的步骤:

- 1-> 使能 `extiline( which I/O )` 时钟和第二功能时钟
- 2-> 配置 `extiline` 的中断优先级
- 2-> 配置 `EXTI line I/O`
- 3-> 选定要配置为 EXTI 的 I/O 口线和 I/O 口的工作模式
- 4-> EXTI line 工作模式配置



```
1.  /*
2.   * 函数名: EXTI_PB0_Config
3.   * 描述 : 配置 PB0 为线中断口, 并设置中断优先级
4.   * 输入 : 无
5.   * 输出 : 无
6.   * 调用 : 外部调用
7.   */
8. void EXTI_PB0_Config(void)
9. {
10.    GPIO_InitTypeDef GPIO_InitStructure;
11.    EXTI_InitTypeDef EXTI_InitStructure;
12.
13.    /* config the extiline(PB0) clock and AFIO clock */
14.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO, ENABLE);
15.
16.    /* config the NVIC(PB0) */
17.    NVIC_Configuration();
18.
19.    /* EXTI line gpio config(PB0) */
20.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
21.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD; // 上拉输入
22.    GPIO_Init(GPIOB, &GPIO_InitStructure);
23.
24.    /* EXTI line(PB0) mode config */
25.    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource0);
26.    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
27.    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
28.    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿中断
29.    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
30.    EXTI_Init(&EXTI_InitStructure);
31. }
```

在这里我们把 **PB0** 连接到内部的 **EXTI0**, 配置为上拉输入, 工作在下降沿中断。  
在外围电路上我们将 **PB0** 接到了 **key1** 上:



当按键没有按下时，**PB0** 始终为高，当按键按下时 **PB0** 变为低，从而 **PB0** 上产生一个低电平跳变，**EXTI0** 会捕捉到这一跳变，并产生相应的中断，中断服务程序在 [stm32f10x\\_it.c](#) 中实现：

```
1. /* I/O 线中断，中断线为 PB0 */
2. void EXTI0_IRQHandler(void)
3. {
4.     if(EXTI_GetITStatus(EXTI_Line0) != RESET) //确保是否产生了 EXTI Line 中断
5.     {
6.         // LED1 取反
7.         GPIO_WriteBit(GPIOC, GPIO_Pin_3,
8.             (BitAction)((!GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
9.
10.        //清除中断标志位
11.        EXTI_ClearITPendingBit(EXTI_Line0);
12.    }
13. }
```

中断服务程序比较简单，很容易读懂，但我们在写中断函数入口的时候要注意函数名的写法，函数名只有两种命名方法：

```
1-> EXTI0_IRQHandler      ; EXTI Line 0
    EXTI1_IRQHandler      ; EXTI Line 1
```



```
EXTI2_IRQHandler ; EXTI Line 2
EXTI3_IRQHandler ; EXTI Line 3
EXTI4_IRQHandler ; EXTI Line 4
2-> EXTI9_5_IRQHandler ; EXTI Line 9..5
EXTI15_10_IRQHandler ; EXTI Line 15..10
```

只要是中断线在 5 之后的就不能像 0~4 那样单独一个函数名，都必须写成 `EXTI9_5_IRQHandler` 和 `EXTI15_10_IRQHandler`。假如写成 `EXTI5_IRQHandler`、`EXTI6_IRQHandler`.....`EXTI15_IRQHandler` 这样子的话编译器是不会报错的，只是中断服务程序不能工作罢了。如果你不知道的话，会让你搞半天也不知问题出现在哪。关于中断函数的函数名的写法在 `startup_stm32f10x_hd.s` 这个文件中都可以找到，下面只是列举了部分：

```
1. DCD EXTI0_IRQHandler ; EXTI Line 0
2. DCD EXTI1_IRQHandler ; EXTI Line 1
3. DCD EXTI2_IRQHandler ; EXTI Line 2
4. DCD EXTI3_IRQHandler ; EXTI Line 3
5. DCD EXTI4_IRQHandler ; EXTI Line 4
6. DCD DMA1_Channel1_IRQHandler ; DMA1 Channel 1
7. DCD DMA1_Channel2_IRQHandler ; DMA1 Channel 2
8. DCD DMA1_Channel3_IRQHandler ; DMA1 Channel 3
9. DCD DMA1_Channel4_IRQHandler ; DMA1 Channel 4
10. DCD DMA1_Channel5_IRQHandler ; DMA1 Channel 5
11. DCD DMA1_Channel6_IRQHandler ; DMA1 Channel 6
12. DCD DMA1_Channel7_IRQHandler ; DMA1 Channel 7
13. DCD ADC1_2_IRQHandler ; ADC1 & ADC2
14. DCD USB_HP_CAN1_TX_IRQHandler ; USB High Priority or CAN1 TX
15. DCD USB_LP_CAN1_RX0_IRQHandler ; USB Low Priority or CAN1 RX0
16. DCD CAN1_RX1_IRQHandler ; CAN1 RX1
17. DCD CAN1_SCE_IRQHandler ; CAN1 SCE
18. DCD EXTI9_5_IRQHandler ; EXTI Line 9..5
19. DCD TIM1_BRK_IRQHandler ; TIM1 Break
20. DCD TIM1_UP_IRQHandler ; TIM1 Update
21. DCD TIM1_TRG_COM_IRQHandler ; TIM1 Trigger and Commutation
22. DCD TIM1_CC_IRQHandler ; TIM1 Capture Compare
23. DCD TIM2_IRQHandler ; TIM2
24. DCD TIM3_IRQHandler ; TIM3
25. DCD TIM4_IRQHandler ; TIM4
```



```
26. DCD    I2C1_EV_IRQHandler      ; I2C1 Event
27. DCD    I2C1_ER_IRQHandler      ; I2C1 Error
28. DCD    I2C2_EV_IRQHandler      ; I2C2 Event
29. DCD    I2C2_ER_IRQHandler      ; I2C2 Error
30. DCD    SPI1_IRQHandler        ; SPI1
31. DCD    SPI2_IRQHandler        ; SPI2
32. DCD    USART1_IRQHandler      ; USART1
33. DCD    USART2_IRQHandler      ; USART2
34. DCD    USART3_IRQHandler      ; USART3
35. DCD    EXTI15_10_IRQHandler    ; EXTI Line 15..10
```

### 实验现象->

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 将编译好的程序下载到开发板, LED1 亮, 按下按键时 LED1 灭, 再按下按键时 LED1 灭, 如此循环。

实验讲解完毕, 野火祝大家学习愉快^\_^。



## ADC 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

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 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)。

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](#)

用户文件: [USER/main.c](#)

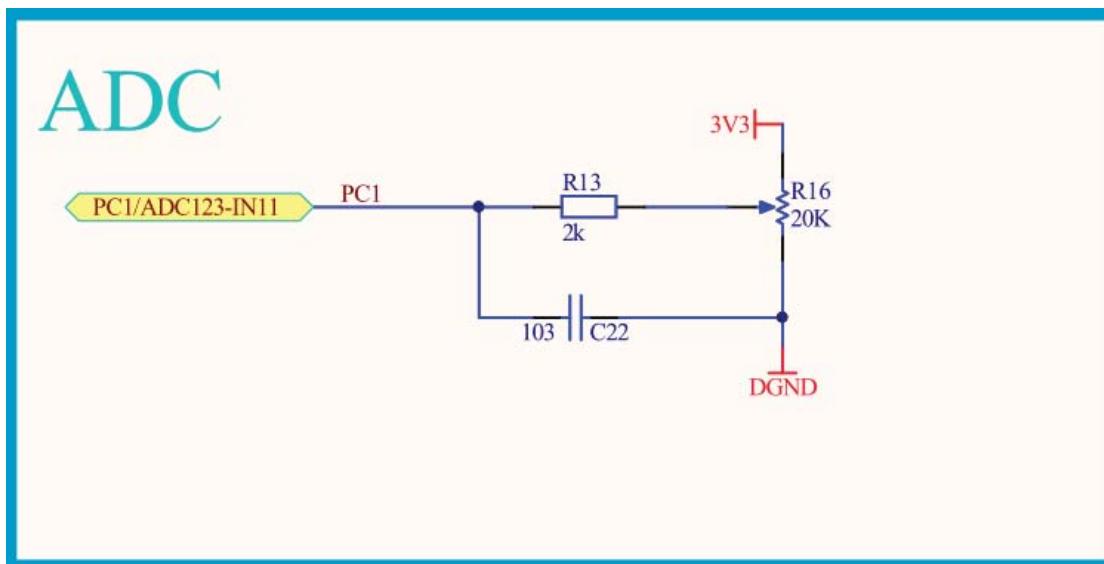
[USER/stm32f10x\\_it.c](#)

[USER/usart1.c](#)

[USER/adc.c](#)



野火 STM32 开发板 ADC 硬件原理图:



### ADC 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。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。

ADC 可以使用 DMA(data memory access)方式操作。

本实验用的是 ADC1 的通道 11，采用 DMA 的方式操作。

### 实验讲解->

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.    ADC1_Init();
17.
18.    printf("\r\n -----这是一个ADC实验-----\r\n");
```



```
19.     while (1)
20.     {
21.         ADC_ConvertedValueLocal = ADC_ConvertedValue; // 读取转换的 AD 值
22.         Delay(0xfffffee); // 延时
23.         printf("\r\n The current AD value = 0x%04X \r\n", ADC_ConvertedValueLocal
24.     );
25. }
26. }
```

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。

`ADC1_Init()`; 函数使能了 `ADC1`, 并使 `ADC1` 工作于 `DMA` 方式。`ADC1_Init()`; 这个函数是由我们用户在 `adc.c` 文件中实现的应用程序:

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

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 `ADC1_Init()`; 接口, 先得更方便简洁。



```
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 channell 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 channell */
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 channel11 configuration */
41.    ADC-RegularChannelConfig(ADC1, ADC_Channel_11, 1, ADC_SampleTime_55Cycles5);
42.
43.    /* Enable ADC1 DMA */
44.    ADC_DMACmd(ADC1, ENABLE);
45.
46.    /* Enable ADC1 */
47.    ADC_Cmd(ADC1, ENABLE);
48.
49.    /* Enable ADC1 reset calibration register */
50.    ADC_ResetCalibration(ADC1);
51.    /* Check the end of ADC1 reset calibration register */
52.    while(ADC_GetResetCalibrationStatus(ADC1));
53.
54.    /* Start ADC1 calibration */
55.    ADC_StartCalibration(ADC1);
56.    /* Check the end of ADC1 calibration */
57.    while(ADC_GetCalibrationStatus(ADC1));
58.
59.    /* Start ADC1 Software Conversion */
60.    ADC_SoftwareStartConvCmd(ADC1, ENABLE);
61. }
```

现在我们再来认识两个变量：

```
1. // ADC1 转换的电压值通过 MDA 方式传到 flash
2. extern __IO u16 ADC_ConvertedValue;
3.
4. // 局部变量, 用于存从 flash 读到的电压值
5. __IO u16 ADC_ConvertedValueLocal;
```



---

ADC\_ConvertedValue 在 adc.c 中定义, ADC\_ConvertedValueLocal 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 ADC1 转换的电压值:

```
1. while (1)
2. {
3.     ADC_ConvertedValueLocal = ADC_ConvertedValue; // 读取转换的 AD 值
4.     Delay(0xfffffe); // 延时
5.     printf("\r\n The current AD value = 0x%04X \r\n", ADC_ConvertedValueLocal);
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:



The screenshot shows a terminal window titled "123 - 超级终端". The window contains the following text:

```
-----这是一个ADC实验-----
The current AD value = 0x0FFE
The current AD value = 0x0FFC
The current AD value = 0x0FFE
The current AD value = 0x0FFC
-
```

At the bottom of the terminal window, there is a status bar displaying connection information: 已连接 0:01:24 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印.



当旋转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 AD 值的明显变化。

The current AD value = 0x0CA4  
The current AD value = 0x0C78  
The current AD value = 0x0C78  
The current AD value = 0x0C74  
The current AD value = 0x0C74

已连接 0:02:45 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印

实验讲解完毕，野火祝大家学习愉快^\_^。



## TIM3 产生 4 路 PWM 信号实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: 通用定时器 TIM3 产生 4 路不同占空比的 PWM 波。

TIM3 Channel1 duty cycle = (TIM3\_CCR1/ TIM3\_ARR)\* 100 = 50%

TIM3 Channel2 duty cycle = (TIM3\_CCR2/ TIM3\_ARR)\* 100 = 37.5%

TIM3 Channel3 duty cycle = (TIM3\_CCR3/ TIM3\_ARR)\* 100 = 25%

TIM3 Channel4 duty cycle = (TIM3\_CCR4/ TIM3\_ARR)\* 100 = 12.5%

硬件连接: PA.06: (TIM3\_CH1)

PA.07: (TIM3\_CH2)

PB.00: (TIM3\_CH3)

PB.01: (TIM3\_CH4)

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\_flash.c

FWlib/stm32f10x\_tim.c



---

用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/pwm\_output.c

## 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，在 103 这个系列中没有 TIM5 定时器，其中 TIM6、TIM7 是基本定时器。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。

## 实验讲解->

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.
```

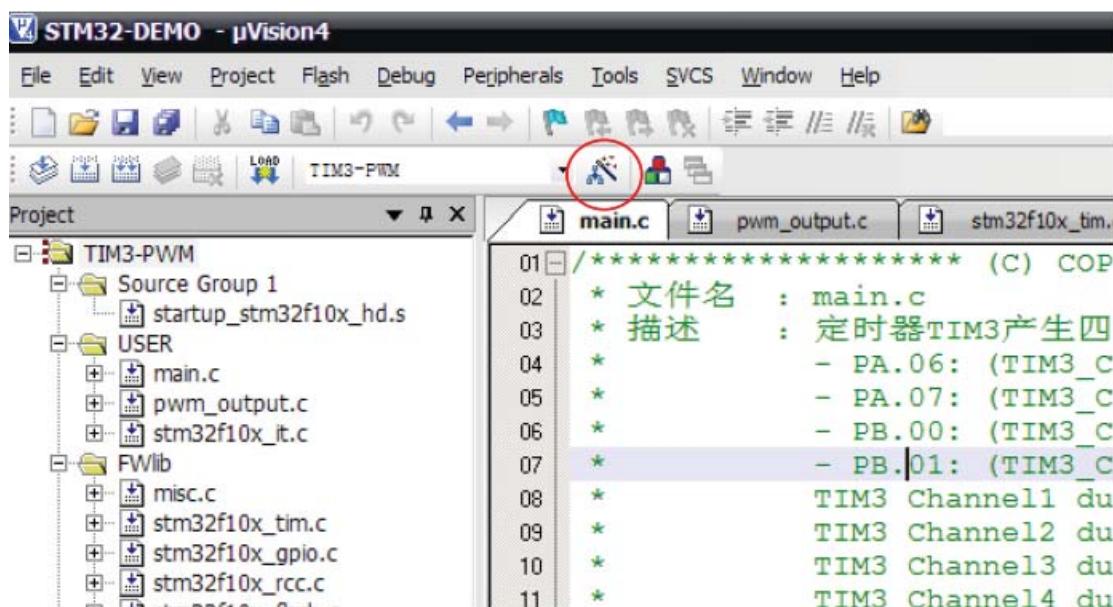


```
67. //设置通道 3 的电平跳变值, 输出另外一个占空比的 PWM
68. TIM_OCInitStructure.TIM_Pulse = CCR3_Val;
69.
70. TIM_OC3Init(TIM3, &TIM_OCInitStructure); //使能通道 3
71. TIM_OC3PreloadConfig(TIM3, TIM_OCPRELOAD_ENABLE);
72.
73. /* PWM1 Mode configuration: Channel4 */
74. TIM_OCInitStructure.TIM_OutputState = TIM_OUTPUTSTATE_ENABLE;
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。

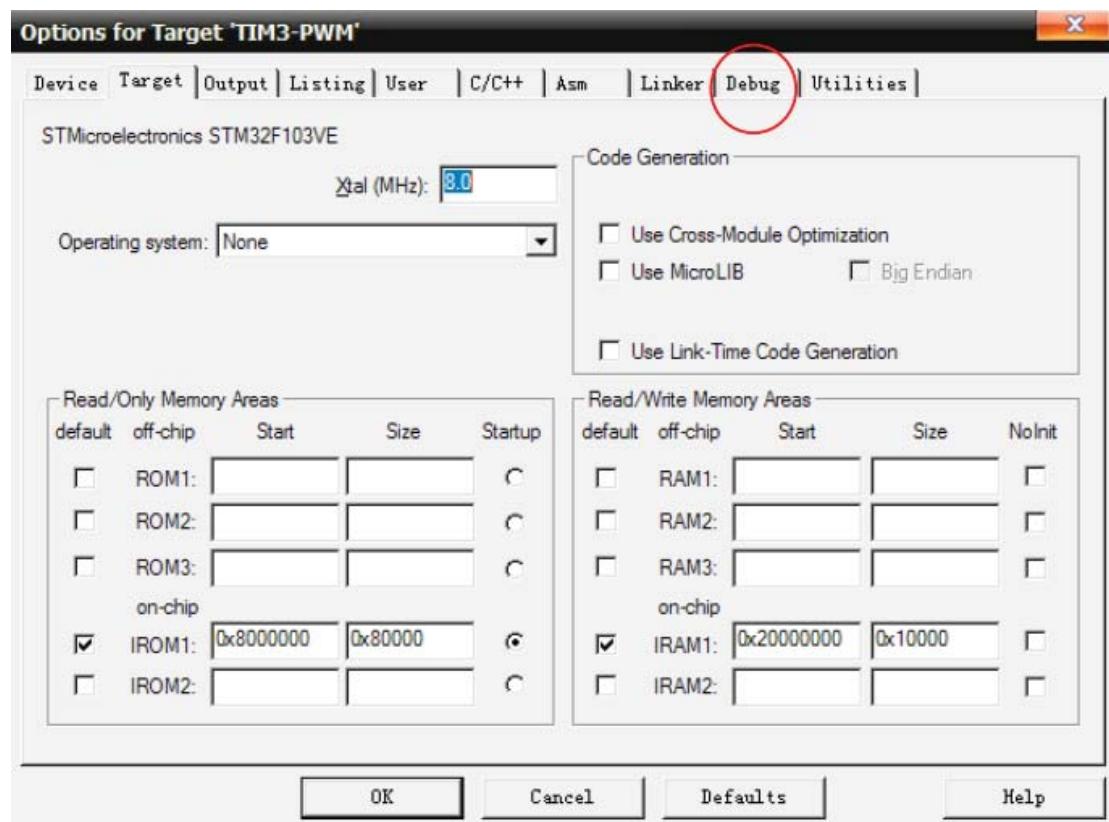
以前我们都是通过 JLINK 直接将我们的代码烧到开发板的 flash 中去调试, 现在要换成软件仿真, 得首先设置下我们的开发环境, 按照如下步骤所示:

1、点击 Target Options...选项。

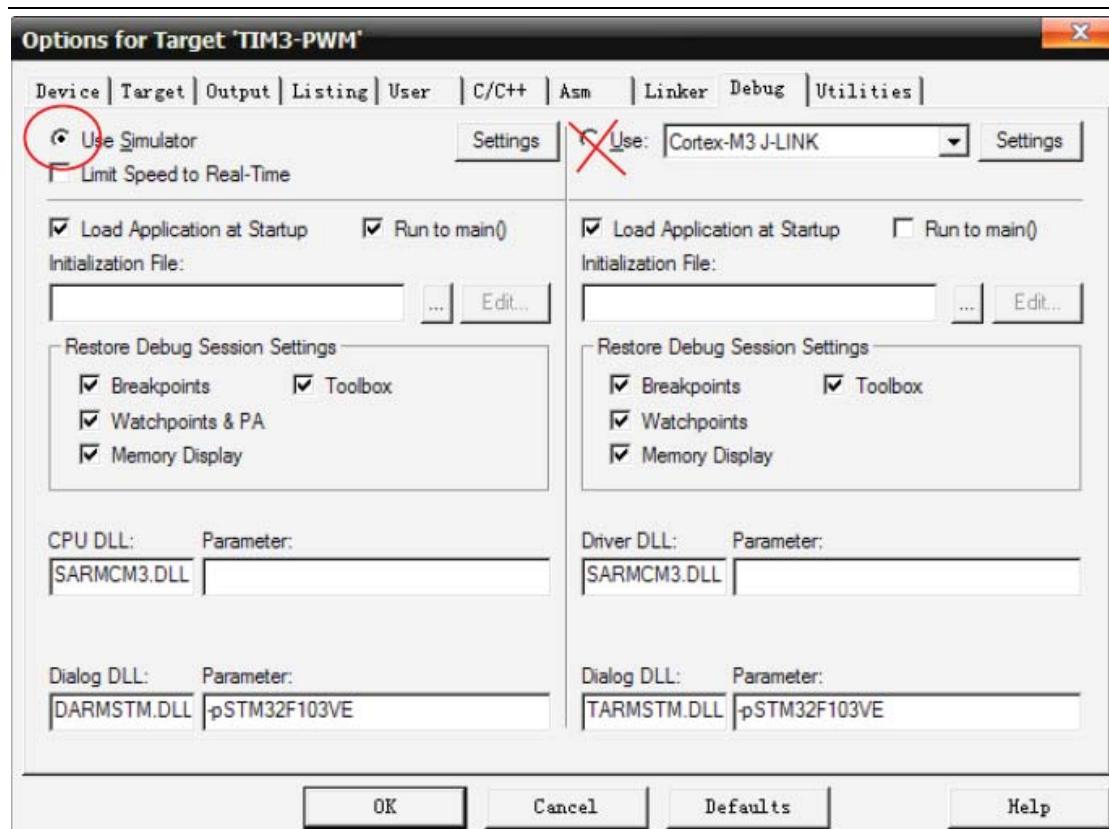




2、选中 Debug 选项卡。

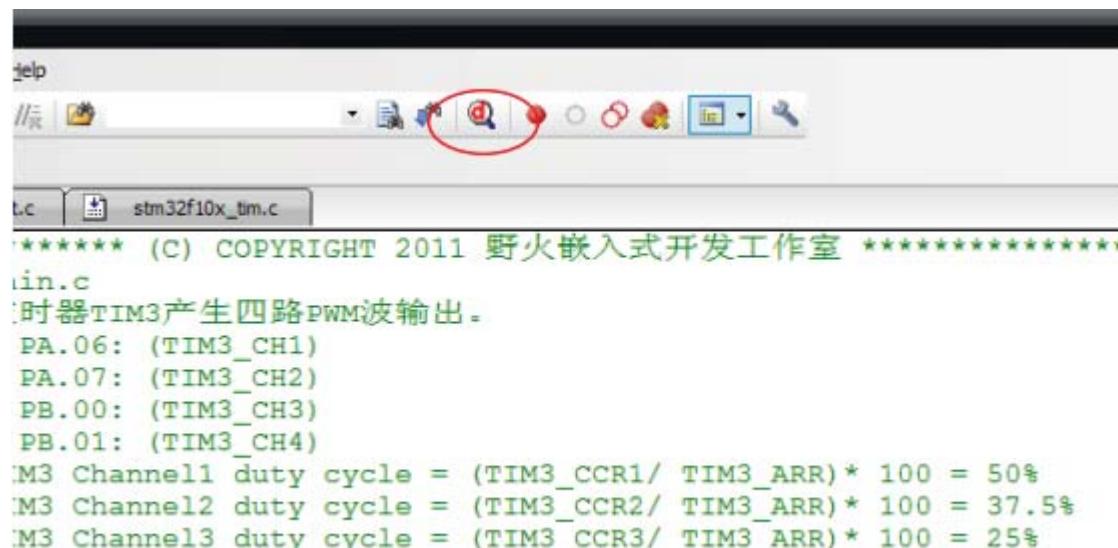


3、选中 Use Simulator 选项，然后点击 OK 即可。

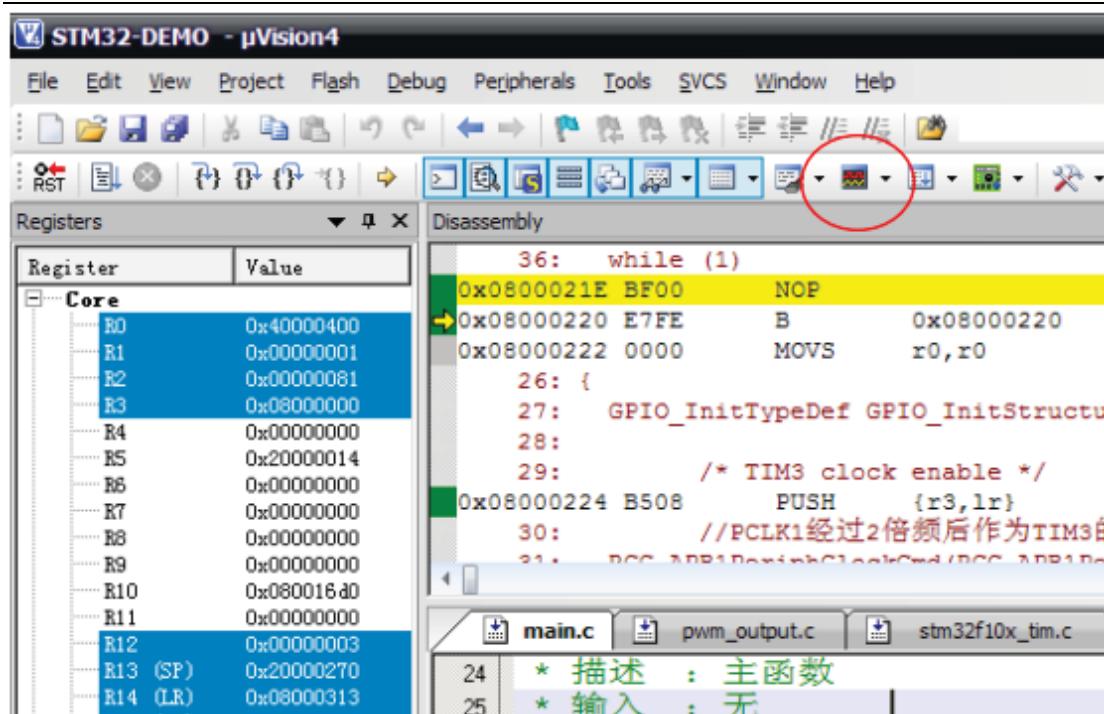


下面我们开始进行软件仿真，按照如下步骤进行：

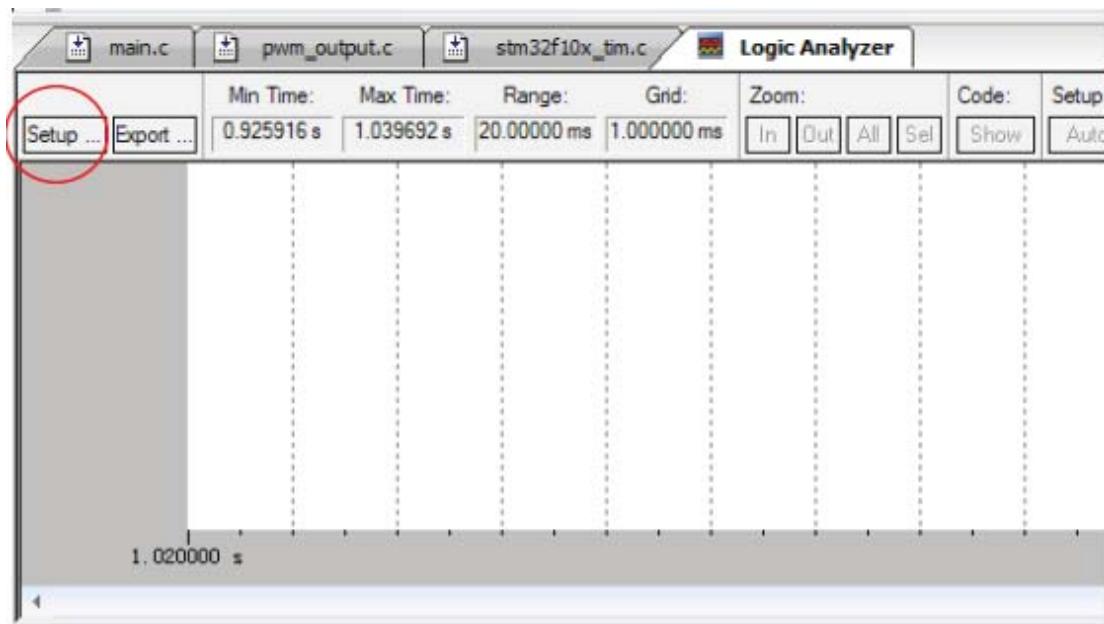
- 1-> 点击 Start/Stop Debug Session 选项。



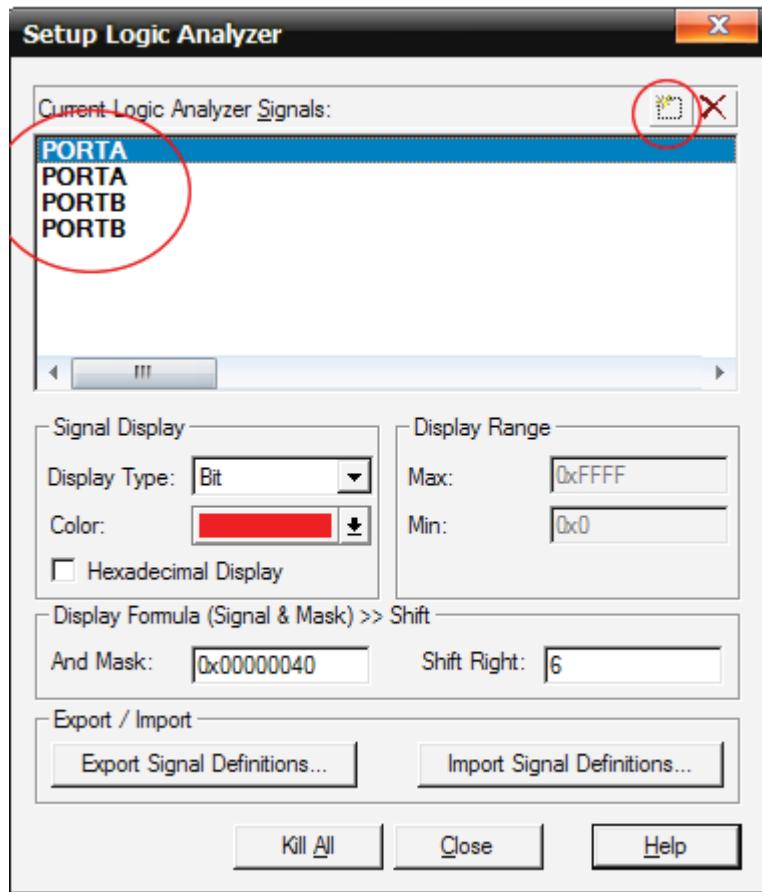
- 2->点击 Analysis Windows 选项。



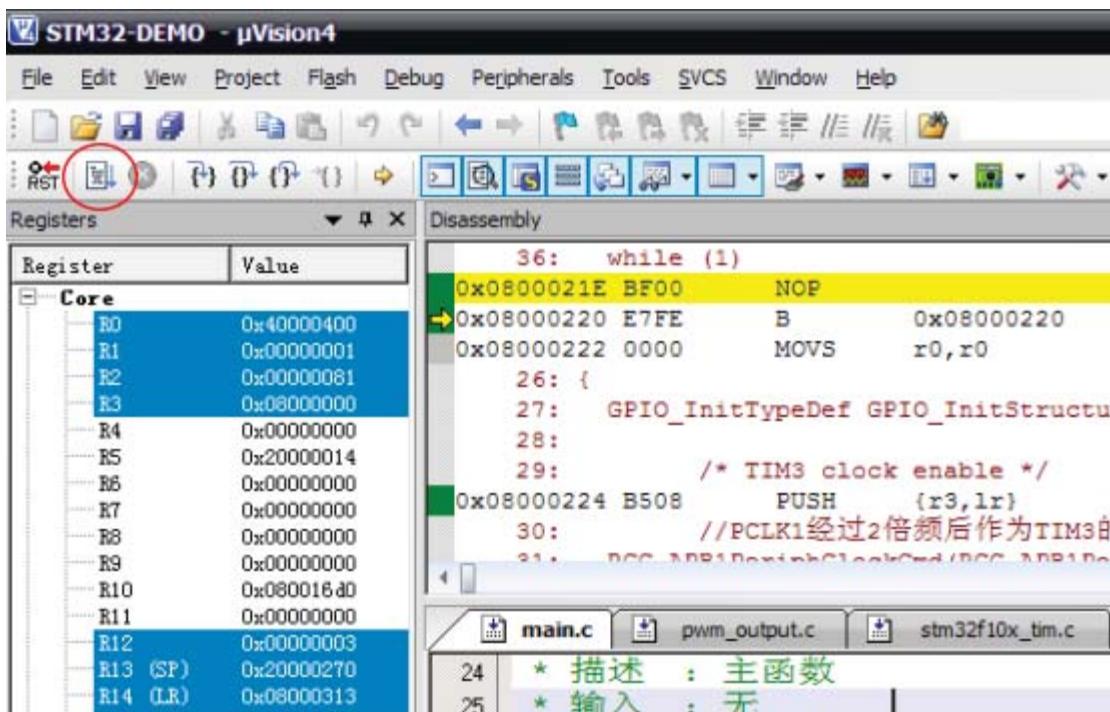
- 3->点击 Setup... 选项卡。



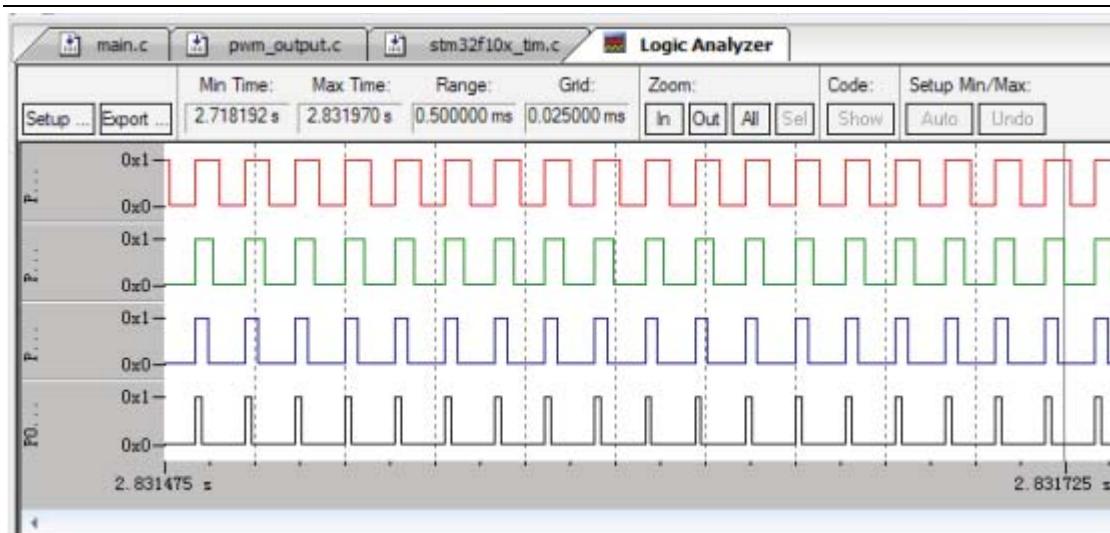
- 4->点击 NEW(Insert), 在下面的文本框中输入 TIM3 的 PWM 通道, 这里分别是: PORTA.6、PORTA.7、PORTB.0、PORTB.1。然后点击 Close。



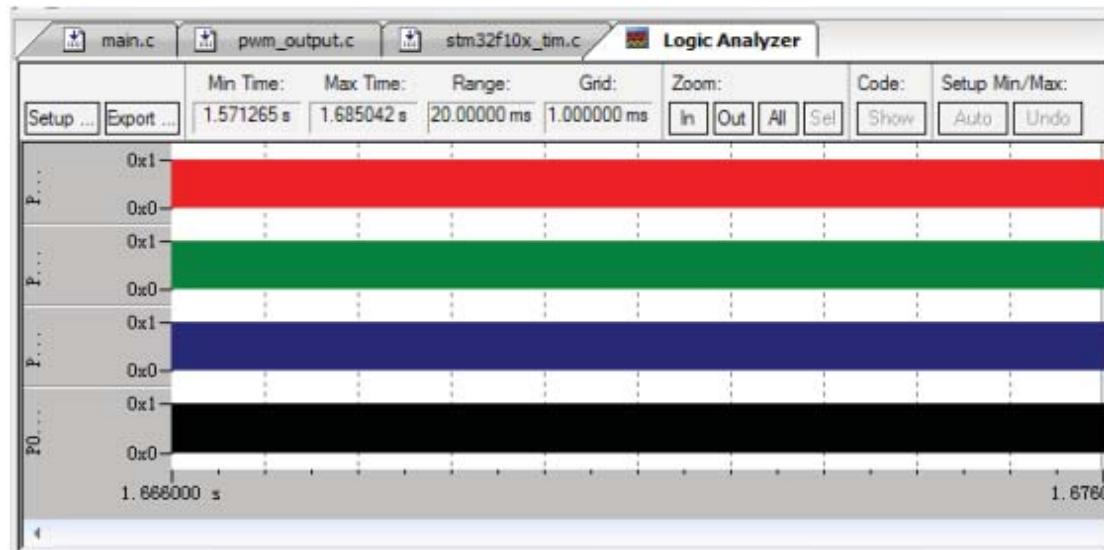
- 5->点击运行。



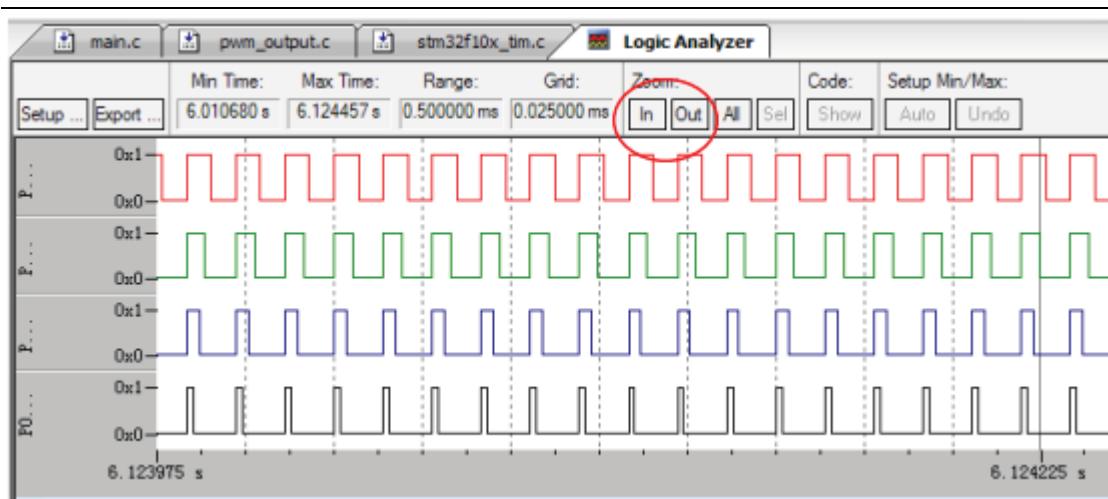
- 6->这时候正常的话则是出现下面的 PWM 信号。



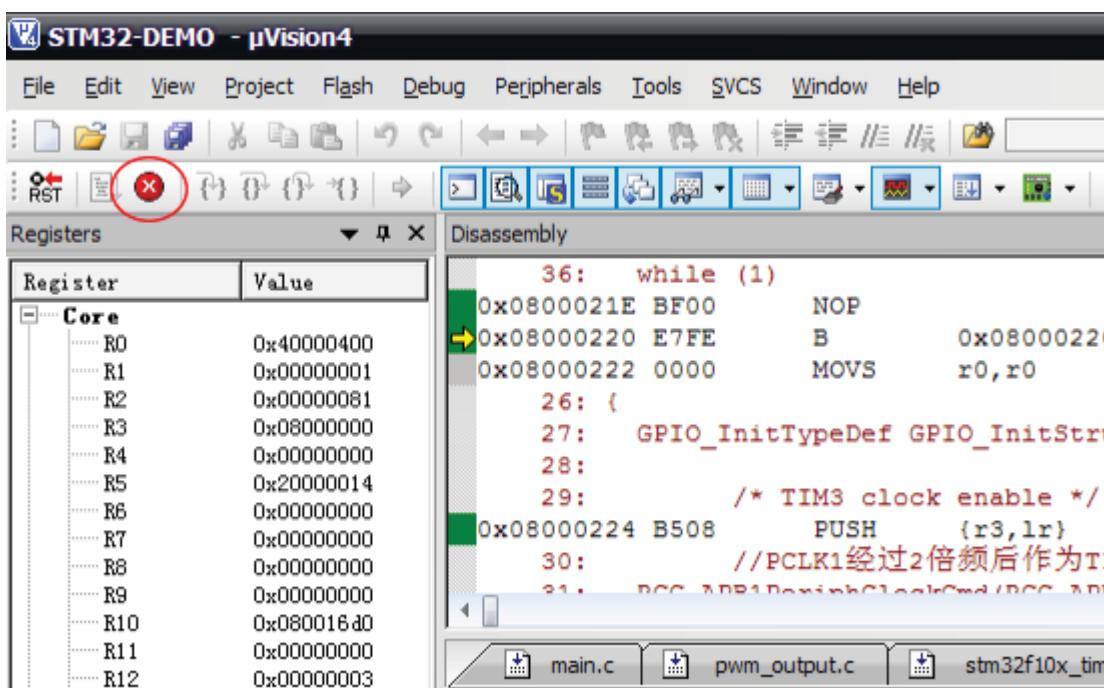
不正常的话则出现下面的情况，一团糟，根本看不到 PWM 信号。这时我们不免会有些抓狂呀，哈哈，但请大家放心，看看接下来我们是怎么解决的吧。



- 7->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->我们可以点击停止按钮，让 PWM 信号静止显示。





- 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->要是您有示波器的话，看到的效果则更真实，更给力。野火用的是泰克的 180M 的数字示波器^\_.....

实验讲解完毕，野火祝大家学习愉快。



## I2C ( EEPROM-AT24C02 )实验

作者	fire
E-Mail	firestm32@foxmail.com
QQ	313303034
博客	firestm32.blog.chinaunix.net
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

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

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

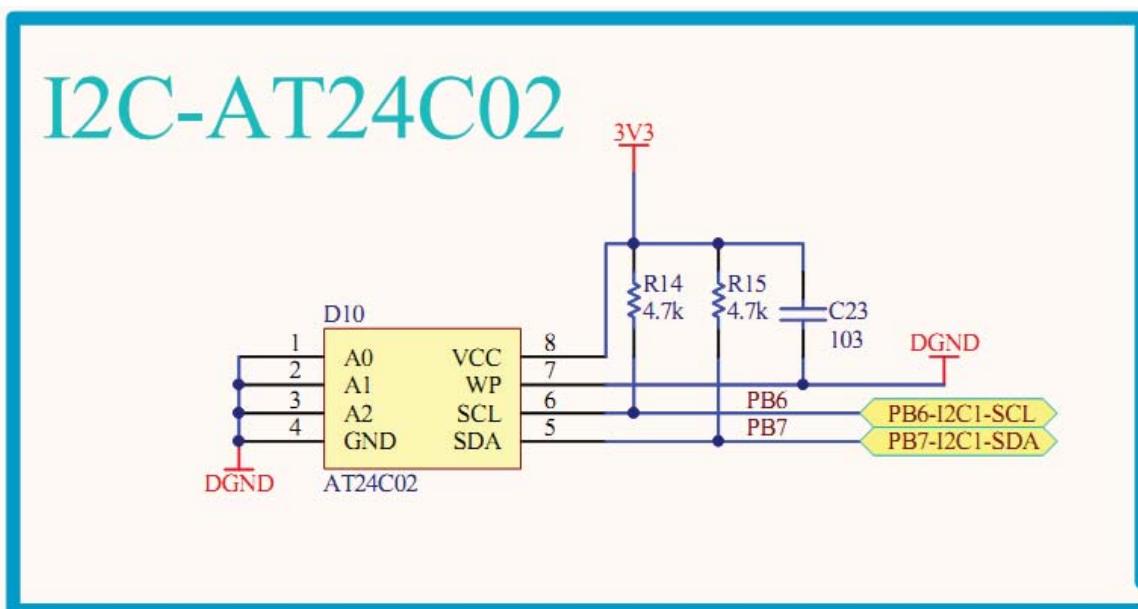
USER/stm32f10x\_it.c

USER/usart1.c

USER/i2c\_ee.c



## 野火 STM32 开发板 I2C-EEPROM 硬件原理图:



## I2C 简介-&gt;

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 [EEPROM](#) (型号: [AT24C02](#)) 的 [SCL](#) 和 [SDA](#) 线。实现 I2C 通讯，对 [EEPROM](#) 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。

## 实验讲解-&gt;

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. #ifdef EEPROM_Block1_ADDRESS
21.   /* 选择 EEPROM Block1 来写入 */
22.   EEPROM_ADDRESS = EEPROM_Block1_ADDRESS;
23. #endif
24.
25. #ifdef EEPROM_Block2_ADDRESS
26.   /* 选择 EEPROM Block2 来写入 */
27.   EEPROM_ADDRESS = EEPROM_Block2_ADDRESS;
28. #endif
29.
30. #ifdef 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_Config()`; 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
27/08/2008
15.
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 */
```

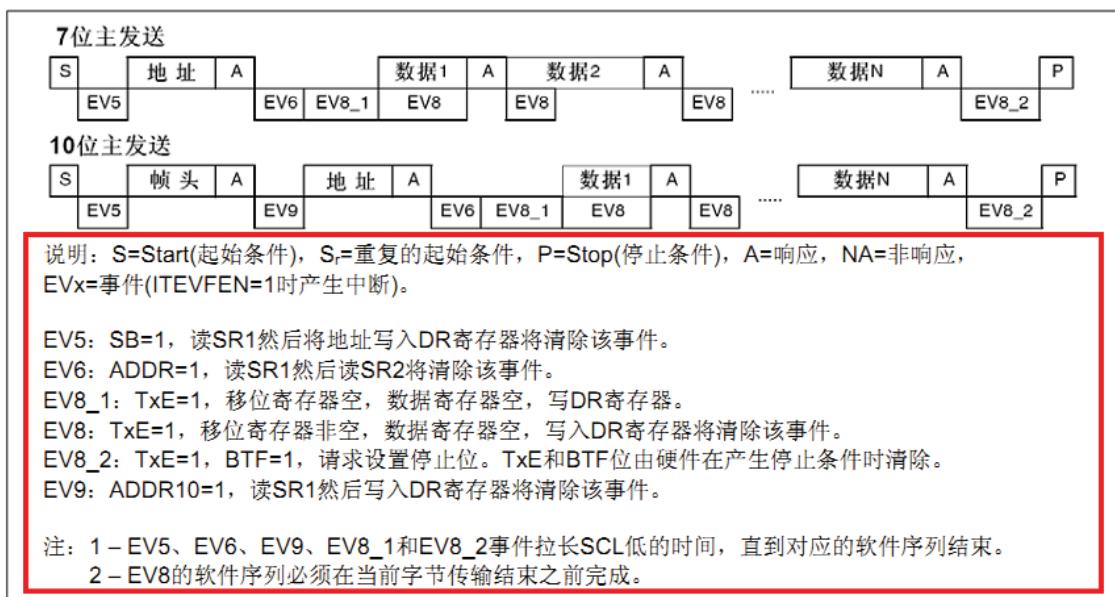


```
44.     while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));  
45. }  
46. /* Send STOP condition */  
47. I2C_GenerateSTOP(I2C1, ENABLE);  
48. }
```

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置位。

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 通讯讯号。如：在确定 SDA 总线空闲的之后，作为主发送器的 stm32 发出起始讯号，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讯号，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
ED));
29.
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。

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:

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

已连接 5:58:07 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印



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！

The screenshot shows a Windows-style terminal window titled "串口调试 - 超级终端". The window contains a large amount of hex data representing the read data from the AT24C02 I2C device. The data starts with 0xF0 and continues through various memory addresses up to 0xEF. Below this, a message reads "I2C(AT24C02) 读写测试成功". The terminal window has standard Windows controls at the top and a status bar at the bottom indicating connection details.

```
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
```

实验讲解完毕，野火祝大家学习愉快^\_^。



野火 stm32 开发板淘宝官方专卖 : <http://firestm32.taobao.com>

---



## SPI( 2M-Flash-W25X16 )实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

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

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

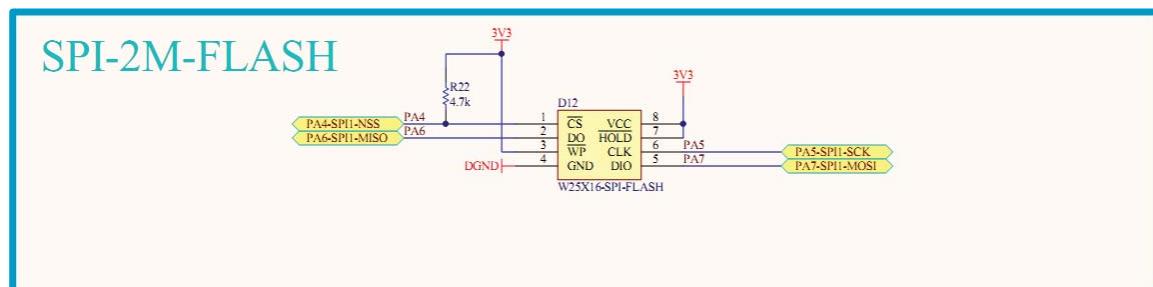
USER/stm32f10x\_it.c

USER/usart1.c

USER/ spi\_flash.c



### 野火 STM32 开发板 I2C-EEPROM 硬件原理图：



### 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。

### 实验讲解->

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\_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.
34.        /* Erase SPI FLASH Sector to write on */
35.        SPI_FLASH_SectorErase(FLASH_SectorToErase);
36.
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.
50.        if( PASSED == TransferStatus1 )
51.        {
52.            printf("\r\n 2M 串行 flash(W25X16) 测试成功!\r\n");
53.        }
54.
```



```
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,
18.                           ENABLE);
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 of
   // the CLK.
49. // Data on the DO and DIO pins are clocked out on the falling edge of
   // 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：

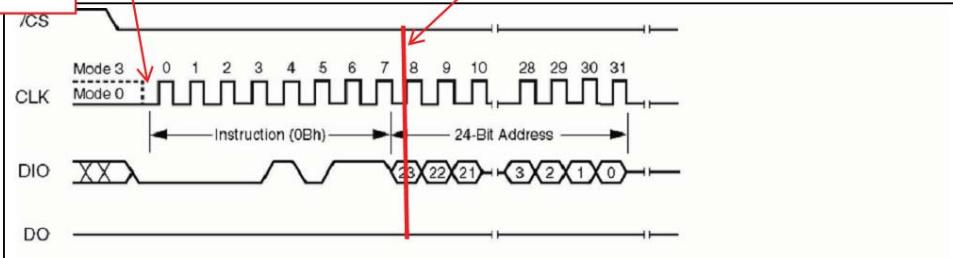


模式3的第一个时钟边沿为下降沿, FLASH的数据采样时刻为上升沿(第二个边沿), 所以CPHA设置为第二个边沿有效

### 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 figure. These dummy clocks allow the internal circuits additional time for setting up the initial value on 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<sup>(1)</sup>

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) <sup>(1)</sup>					(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) <sup>(4)</sup>		
Manufacturer/Device ID <sup>(3)</sup>	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, 以便能 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_Byt`e 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 the 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_SetData()` 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 or 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。

- 1、 配置 I/O 端口,使能 GPIO;
- 2、 根据将要进行通讯器件的 SPI 模式, 配置 stm32 的 SPI, 使能 SPI 时钟;
- 3、 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。

#### 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:



串口调试 - 超级终端

文件(E) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

这是2M串行flash(W25X16)实验  
FlashID is 0xEF3015, Manufacturer Device ID is 0x14  
检测到华邦串行flash W25X16 !  
写入的数据为: 感谢您选用野火stm32开发板  
http://firestm32.taobao.com  
读出的数据为: 感谢您选用野火stm32开发板  
http://firestm32.taobao.com

已连接 12:10:07 自动检测 | 115200 8-N-1 | SCROLL | CAPS | NUM | 插 | 打印 |

实验讲解完毕，野火祝大家学习愉快^\_^。



## CAN (回环和中断模式) 通信实验

作者	fire
E-Mail	firestm32@foxmail.com
QQ	313303034
博客	firestm32.blog.chinaunix.net
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

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

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

用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/led.c

USER/usart.c

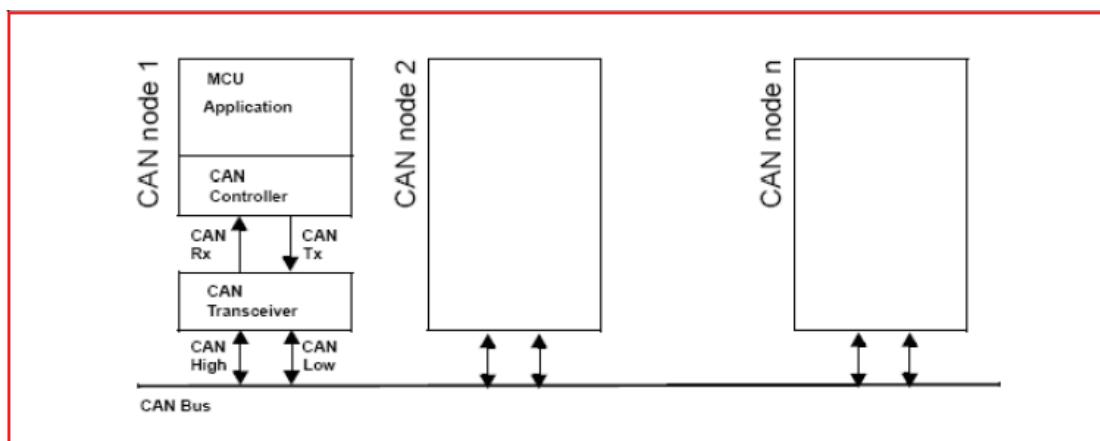
USER/can.c



## CAN 简介->

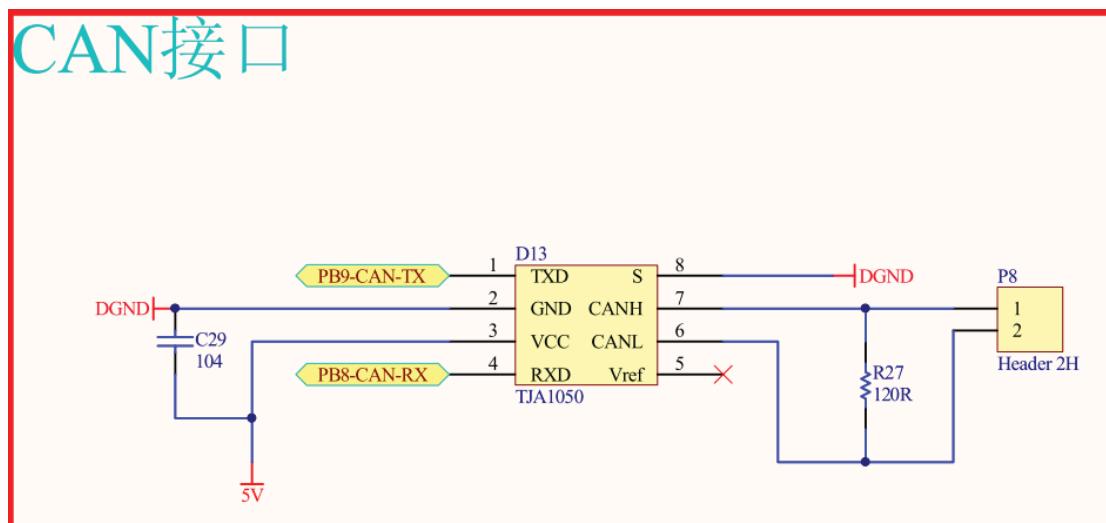
CAN 是控制器局域网络(Controller Area Network, CAN)的简称，是由研发和生产汽车电子产品著称的德国 BOSCH 公司开发的，并最终成为国际标准（ISO11898）。是国际上应用最广泛的现场总线之一。在北美和西欧，CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。近年来，其所具有的高可靠性和良好的错误检测能力受到重视，被广泛应用于汽车计算机控制系统和环境温度恶劣、电磁辐射强和振动大的工业环境。

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：

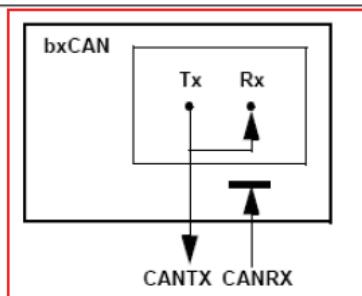


在本实验中并没有用得到双 CAN 通信，只是用了 CAN 的回环测试，这样我们就不需要挂接外部的 CAN 节点。当我们用 CAN 的回环测试时，硬件会在内部将 TX 和 RX 连接起来，实现内部的收和发，从而达到测试的目的。

### 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。



---

## 实验讲解->

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_wwdg.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.    /* CAN1 Periph clock enable */
13.    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
14.
15.    /* Configure CAN pin: RX */                      // PB8
16.    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;
```

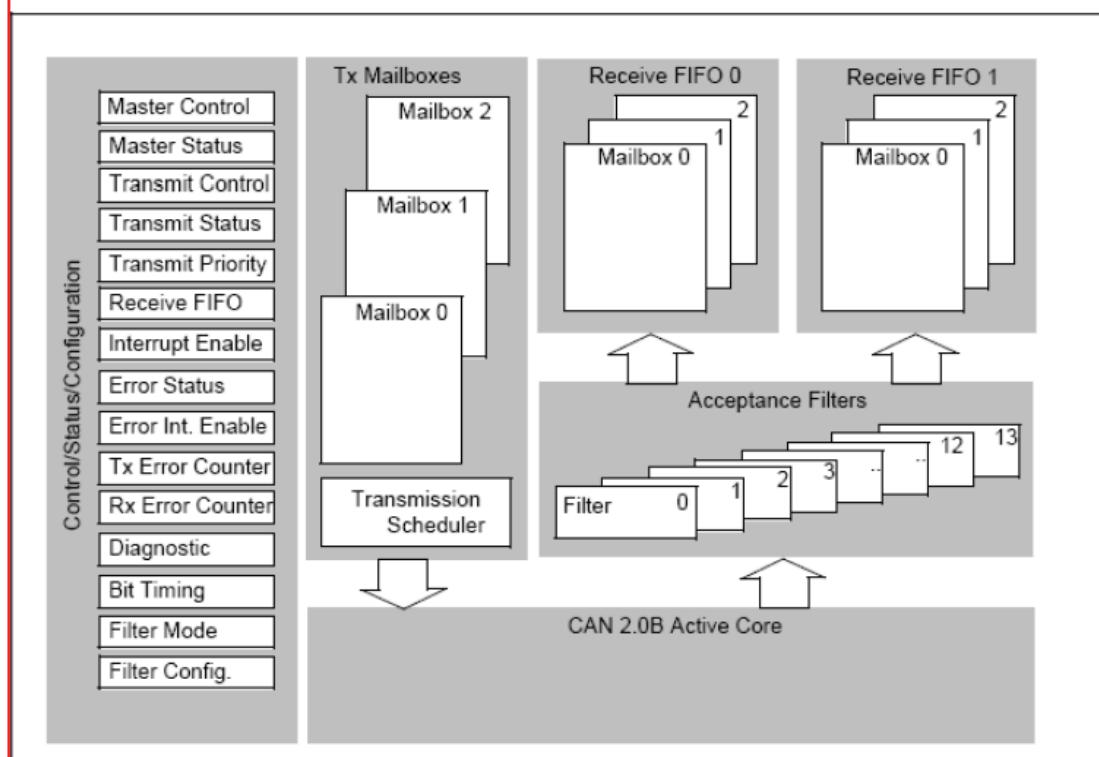


```
28.     CAN_InitStructure.CAN_TXFP=DISABLE;
29.     CAN_InitStructure.CAN_Mode=CAN_Mode_LoopBack; // 回环模式
30.     CAN_InitStructure.CAN_SJW=CAN_SJW_1tq;
31.     CAN_InitStructure.CAN_BS1=CAN_BS1_8tq;
32.     CAN_InitStructure.CAN_BS2=CAN_BS2_7tq;
33.     CAN_InitStructure.CAN_Prescaler=5;           // 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 / Sstatus / 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 `bxcan`>，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。唯一一个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. }
```



### 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:

The screenshot shows a terminal window titled "123 - 超级终端". The window contains the following text output:

```
这个一个CAN (回环模式和中断模式) 测试程序.....  
CAN 回环初始化成功.....  
CAN 回环测试成功.....  
这个一个CAN (回环模式和中断模式) 测试程序.....  
CAN 回环初始化成功.....  
CAN 回环测试成功.....  
这个一个CAN (回环模式和中断模式) 测试程序.....  
CAN 回环初始化成功.....  
CAN 回环测试成功.....
```

At the bottom of the terminal window, there is a status bar displaying: 已连接 0:00:36 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印.

实验讲解完毕, 野火祝大家学习愉快^\_^。



## MicroSD 卡 (SDIO 模式) 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: MicroSD 卡(SDIO 模式)测试实验, 没有跑文件系统, 只是单纯地读 block  
并将测试信息通过串口 1 在电脑的超级终端上 打印出来。

硬件连接: PC8-SDIO-D0 : DATA0

PC9-SDIO-D1 : DATA1

PC10-SDIO-D2 : DATA2

PC11-SDIO-D3 : CD/DATA3

PC12-SDIO-CLK: CLK

PD2-SDIO-CMD : CMD

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\_sdio.c

FWlib/stm32f10x\_dma.c

FWlib/misc.c



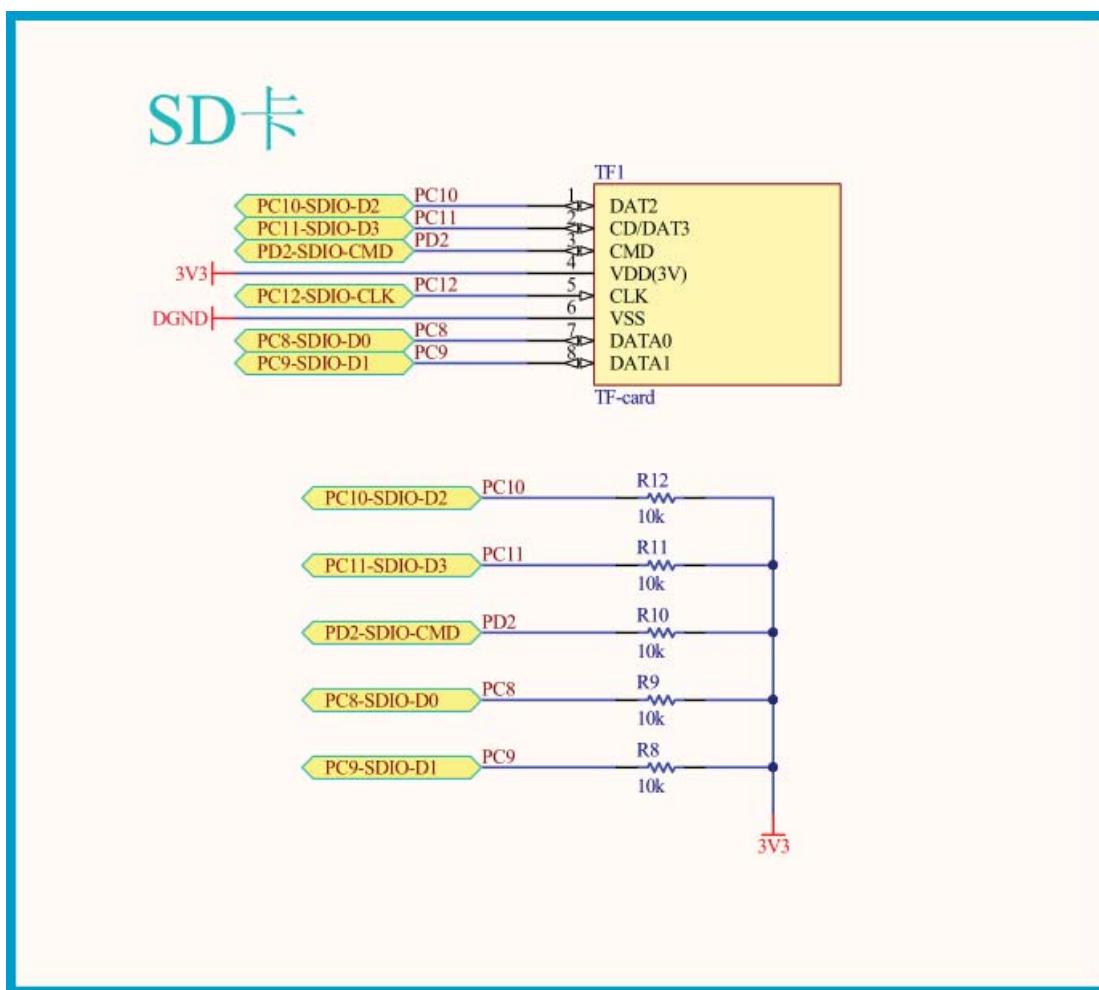
用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/usart.c

USER/sdcard.c

野火 STM32 开发板 MicroSD 卡硬件原理图:



### SDIO 简介->

野火 STM32 开发板的 CPU ( STM32F103VET6 )具有一个 SDIO 接口。

SD/SDIO/MMC 主机接口可以支持 MMC 卡系统规范 4.2 版中的 3 个不同的数据总线模式：1 位(默认)、4 位和 8 位。在 8 位模式下，该接口可以使数据传输速率达到



48MHz，该接口兼容 SD 存储卡规范 2.0 版。SDIO 存储卡规范 2.0 版支持两种数据总线模式：1 位(默认)和 4 位。

目前的芯片版本只能一次支持一个 SD/SDIO/MMC 4.2 版的卡，但可以同时支持多个 MMC 4.1 版或之前版本的卡。除了 SD/SDIO/MMC，这个接口完全与 CE-ATA 数字协议版本 1.1 兼容。

本实验用的是 MicroSD 卡 1 位( 默认 )模式，速率为 1M/S，MicroSD 的容量为 1G。

## 实验讲解->

首先在工程中添加我们所需要的库文件：

[FWlib/stm32f10x\\_gpio.c](#)  
[FWlib/stm32f10x\\_rcc.c](#)  
[FWlib/stm32f10x\\_usart.c](#)  
[FWlib/stm32f10x\\_sdio.c](#)  
[FWlib/stm32f10x\\_dma.c](#)  
[FWlib/misc.c](#)

然后在 `stm32f10x_conf.h` 这个头文件中将用到的库的头文件的注释去掉：

```
1. /* Includes -----  
2. */  
3. /* Uncomment the line below to enable peripheral header file inclusion */  
4. /* #include "stm32f10x_adc.h" */  
5. /* #include "stm32f10x_bkp.h" */  
6. /* #include "stm32f10x_can.h" */  
7. /* #include "stm32f10x_crc.h" */  
8. /* #include "stm32f10x_dac.h" */  
9. /* #include "stm32f10x_dbgmcu.h" */  
10. /* #include "stm32f10x_dma.h" */  
11. /*#include "stm32f10x_exti.h" */  
12. /*#include "stm32f10x_flash.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_wwdg.h" */  
24. #include "misc.h" /* High level functions for NVIC and SysTick (add-  
25. on to CMSIS functions) */
```

配置完需要用到的头文件之后，我们将从 `main` 函数开始分析：



```
1.  /*
2.   * 函数名: main
3.   * 描述 : 无
4.   * 输入 : 无
5.   * 输出 : 无
6.   */
7. int main(void)
8. {
9.     /* config the sysclock to 72M */
10.    SystemInit();
11.
12.    /* USART1 config */
13.    USART1_Config();
14.
15.    /* Interrupt Config */
16.    NVIC_Configuration();
17.
18.    printf( "\r\n 这是一个 MicroSD 卡实验(没有跑文件系统)..... " );
19.
20.    /*----- SD Init -----*/
21.    Status = SD_Init();
22.
23.    if (Status == SD_OK)
24.    {
25.        /*----- Read CSD/CID MSD registers -----*/
26.        printf( "\r\n SD_Init is ok " );
27.        Status = SD_GetCardInfo(&SDCardInfo);
28.    }
29.
30. //printf( "\r\n SD_GetCardInfo Status is: %d ", Status );
31. printf( "\r\n CardType is : %d ", SDCardInfo.CardType );
32. printf( "\r\n CardCapacity is : %d ", SDCardInfo.CardCapacity );
33. printf( "\r\n CardBlockSize is : %d ", SDCardInfo.CardBlockSize );
34. printf( "\r\n RCA is : %d ", SDCardInfo.RCA );
35. printf( "\r\n ManufacturerID is : %d ", SDCardInfo.SD_cid.ManufacturerID )
;
36.
37.    if (Status == SD_OK)
38.    {
39.        printf("\r\n SD_GetCardInfo is ok ");
40.        /*----- Select Card -----*/
41.        Status = SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));
42.    }
43.
44.    if (Status == SD_OK)
45.    {
46.        printf(" \r\n SD_SelectDeselect is ok ");
47.        // SDIO_BusWide_4b 会进入死循环,至今还未解决
48.        Status = SD_EnableWideBusOperation(SDIO_BusWide_1b);
49.    }
50.
51.    /*----- Block Erase -----*/
52.    if (Status == SD_OK)
53.    {
54.        /* Erase NumberOfBlocks Blocks of WRITE_BL_LEN(512 Bytes) */
55.        Status = SD_Erase(0x00, (BlockSize * NumberOfBlocks));
56.    }
57.
58.    if (Status == SD_OK)
59.    {
60.        printf(" \r\n SD_Erase is ok ");
61.        Status = SD_SetDeviceMode(SD DMA MODE);
62.    }
63.
64.    if (Status == SD_OK)
65.    {
66.        printf(" \r\n SD_SetDeviceMode is ok ");
67.        // 4bit 模式时会停在这里了, 进入死循环, 1bit 模式时则可以读取成功
68.        Status = SD_ReadMultiBlocks(0x00, Buffer_MultiBlock_Rx, BlockSize, NumberOfBlocks);
69.    }
70.
71.    if (Status == SD_OK)
72.    {
```



```
73.     printf(" \r\n SD ReadMultiBlocks is ok ");
74.     EraseStatus = eBuffercmp(Buffer_MultiBlock_Rx, MultiBufferWordsSize);
75.     printf( " \r\n eBuffercmp EraseStatus is %d ", EraseStatus );
76. }
77.
78. /*----- Block Read/Write -----*/
79. /* Fill the buffer to send */
80. Fill_Buffer(Buffer_Block_Tx, BufferWordsSize, 0x12345678);
81.
82.
83. if (Status == SD_OK)
84. {
85.     /* Write block of 512 bytes on address 0 */
86.     Status = SD_WriteBlock(0x00, Buffer_Block_Tx, BlockSize);
87. }
88.
89. if (Status == SD_OK)
90. {
91.     printf( " \r\n SD WriteBlock is ok " );
92.     /* Read block of 512 bytes from address 0 */
93.     Status = SD_ReadBlock(0x00, Buffer_Block_Rx, BlockSize);
94. }
95.
96. if (Status == SD_OK)
97. {
98.     printf(" \r\n SD ReadBlock is ok ");
99.     /* Check the correctness of written dada */
100.    TransferStatus1 = Buffercmp(Buffer_Block_Tx, Buffer_Block_Rx, BufferWordsSize);
101. }
102.
103. if (TransferStatus1 == PASSED)
104. {
105.     /* 写进去的值和读出来的值相同 */
106.     printf( "\r\n Single Block Read/Write is ok " );
107. }
108.
109. /*----- Multiple Block Read/Write -----*/
110. /* Fill the buffer to send */
111. Fill Buffer(Buffer MultiBlock Tx, MultiBufferWordsSize, 0x0);
112.
113. if (Status == SD_OK)
114. {
115.     /* Write multiple block of many bytes on address 0 */
116.     Status = SD_WriteMultiBlocks(0x00, Buffer_MultiBlock_Tx, BlockSize,
117.                               NumberOfBlocks);
118. }
119.
120. if (Status == SD_OK)
121. {
122.     /* Read block of many bytes from address 0 */
123.     Status = SD ReadMultiBlocks(0x00, Buffer MultiBlock Rx, BlockSize,
124.                               NumberOfBlocks);
125.
126. if (Status == SD_OK)
127. {
128.     /* Check the correctness of written dada */
129.     /* 写进去的值和读出来的值相同 */
130.     TransferStatus2 = Buffercmp(Buffer MultiBlock Tx, Buffer MultiBlock
131.                               _Rx, MultiBufferWordsSize);
132. }
133.
134. if ( TransferStatus2 == PASSED )
135. {
136.     printf( " \r\n Multiple Block Read/Write is ok " );
137.
138.     printf(" \r\n 好消息: MicroSD 卡读写测试成功 \r\n ");
139.
140. while (1)
141. {
142. }
143. } /* end of main */
```



`main` 函数有点长，但认真分析下还是很好理解的，大伙给点耐心，用到的很多都是库操作，都是 ST 的工程师为用户写好的。

首先，我们还是一如既往地先调用我们的库函数 `SystemInit()`; 将我们的系统时钟设置为 72MHZ。函数 `USART1_Config()`; 用来初始化我们要用到的串口。关于这两个函数的详细讲解可参考前面的教程，这里不在详述。

`NVIC_Configuration()`; 用来设置 SDIO 的中断优先级，因为 SDIO 有三种工作模式：  
`SD_DMA_MODE`、`SD_POLLING_MODE`、`SD_INTERRUPT_MODE`，其中工作在中断模式时就需要设置中断优先级，本实验用的是 SDIO 的 DMA 模式，所以本实验不用调用这个函数也可以，调用了也无所谓。

接下来的代码是与 MicroSD 卡的操作有关的，与 MicroSD 卡有关的函数都在 `socard.c` 这个文件中实现，`socard.c` 不是库文件，而是需要用户编写的。

1-> 首先调用 `SD_Init()`; 用于初始化 MicroSD 卡，使卡处于就绪状态(准备传输数据)。

2-> 初始化 MicroSD 卡成功之后，调用函数 `SD_GetCardInfo(&SDCardInfo);` 来获取与卡的属性有关的信息。紧接着通过串口将这些信息在电脑的超级终端中打印出来。

3-> 获取到卡的正确信息后，调用函数

`SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));` 来选中这张卡。

4-> 确定好目标卡之后，调用函数 `SD_EnableWideBusOperation(SDIO_BusWide_1b);` 设定卡的数据位的宽度，这里设置为 1 位，之所以设置为 1 位而不是 4 位，是因为 4 位模式我还没有调试成功，仍需努力呀！

5-> 紧接着就调用函数 `SD_SetDeviceMode(SD_DMA_MODE);` 设置 SDIO 的工作模式为 DMA 方式。



6->到了这里与卡有关的初始化都已完成，假如程序能运行到这里的话，卡的操作基本上已经成功了一半。

7->Block Read/Write 和 Multiple Block Read/Write 测试，将数据写到卡里面去，再将数据读出来，如果写进去的和读出来的数据相同，即为成功。

8->最后打印调试信息。

**注意：**这个例程是没有跑文件系统的，而是直接就去读卡的 **block**，这样的话就会破坏卡的分区，在实验完成之后，你再把卡插到电脑上时，电脑会提示你要重新初始化卡，这是正常想象，并不是本实验把你的卡弄坏了。但跑文件系统时就不会出现这种问题，有关文件系统的操作将在下一讲的教程中讲解。



### 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), 插上 MicroSD 卡( 我用的是 1G ), 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:

```
这是一个MicroSD卡实验(没有跑文件系统).....  
SD_Init is ok  
CardType is : 1  
CardCapacity is : 966787072  
CardBlockSize is : 512  
RCA is : 60268  
ManufacturerID is : 2  
SD_GetCardInfo is ok  
SD_SelectDeselect is ok  
SD_Erase is ok  
SD_SetDeviceMode is ok  
SD_ReadMultiBlocks is ok  
eBuffercmp EraseStatus is 0  
SD_WriteBlock is ok  
SD_ReadBlock is ok  
Single Block Read/Write is ok  
Multiple Block Read/Write is ok  
好消息: MicroSD卡读写测试成功
```

已连接 1:30:24 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印

实验讲解完毕, 野火祝大家学习愉快^\_^。



## MicroSD 卡+文件系统(R0.07C) 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: MicroSD 卡文件系统 FATFS R0.07C 测试实验。在 MicroSD 卡里面创建一个 DEMO.TXT 文本文件, 在文件里面写入字符串“感谢您选用 野火 STM32 开发板 ! ^\_^”, 然后通过串口将这些内容打印在电脑的超级终端上。并图解了文件系统移植的全部过程。

硬件连接: PC8-SDIO-D0 : DATA0

PC9-SDIO-D1 : DATA1

PC10-SDIO-D2 : DATA2

PC11-SDIO-D3 : CD/DATA3

PC12-SDIO-CLK : CLK

PD2-SDIO-CMD : CMD

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\_sdio.c

FWlib/stm32f10x\_dma.c



## FWlib/misc.c

用户文件: USER/main.c

USER/stm32f10x\_it.c

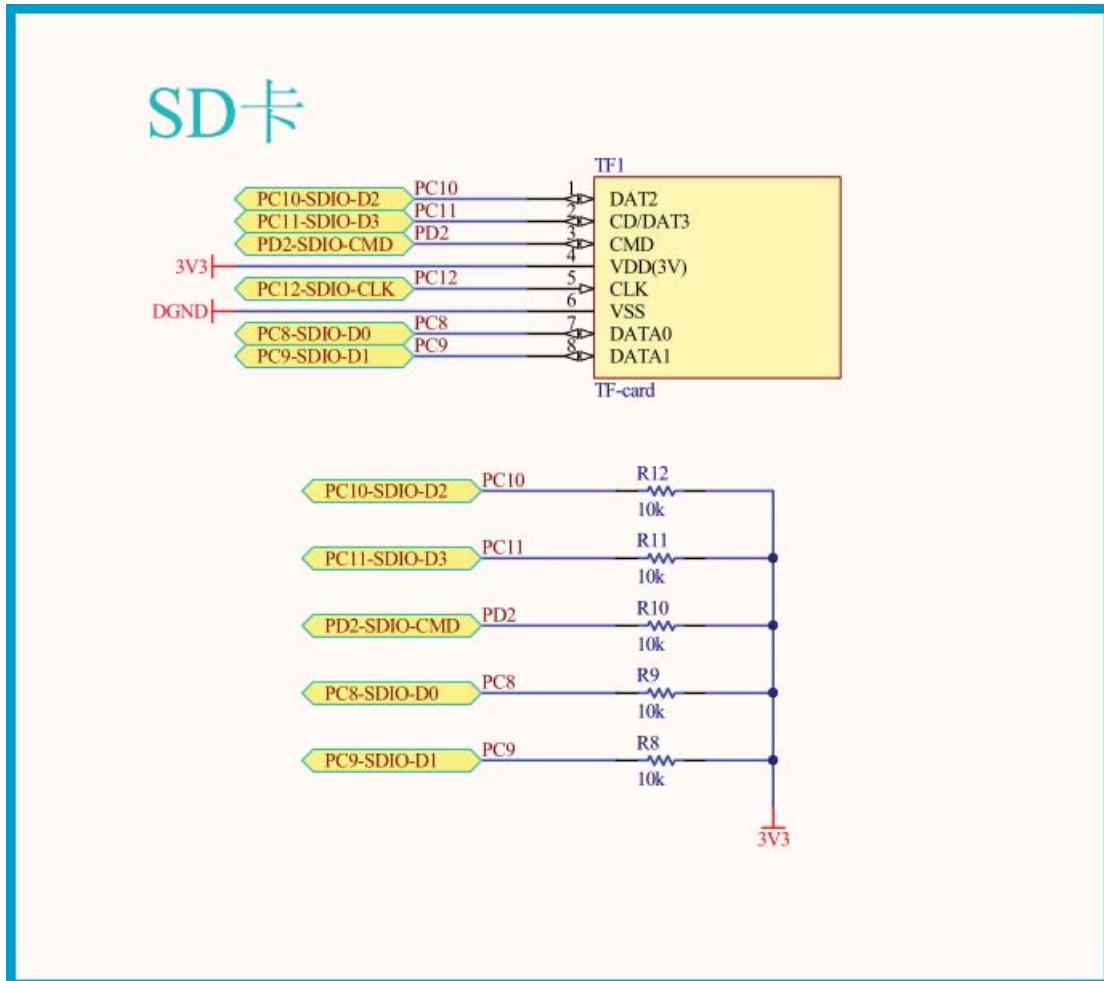
USER/usart.c

USER/sdcard.c

USER/diskio.c

USER/ff.c

野火 STM32 开发板 MicroSD 卡硬件原理图:





本实验是在上一讲《MicroSD 卡(SDIO 模式)实验》的基础上讲解的，只有上一讲的实验成功了，文件系统才能跑起来。有关卡的底层的初始化，这里不再详述，[这里](#)着重讲解文件系统的移植和文件系统的应用。文件系统的版本是 R0.07C，现在最新的版本是 R0.08C，野火会在后续的例程中将文件系统升级到 R0.08C。

### FATFS 文件系统简介->

FAFFS 是面向小型嵌入式系统的一种通用的 FAT 文件系统。FATFS 完全是由 AISI C 语言编写并且完全独立于底层的 I/O 介质。因此它可以很容易地不加修改地移植到其他的处理器当中，如 8051、PIC、AVR、SH、Z80、H8、ARM 等。FATFS 支持 FAT12、FAT16、FAT32 等格式。

本实验是将 FATFS 移植到野火 STM32 开发板中，CPU 为 [STM32F103VET6](#)，是采用 [ARM](#) 公司最新内核 [ARMV7](#) 的一款单片机。

### 移植前的工作->

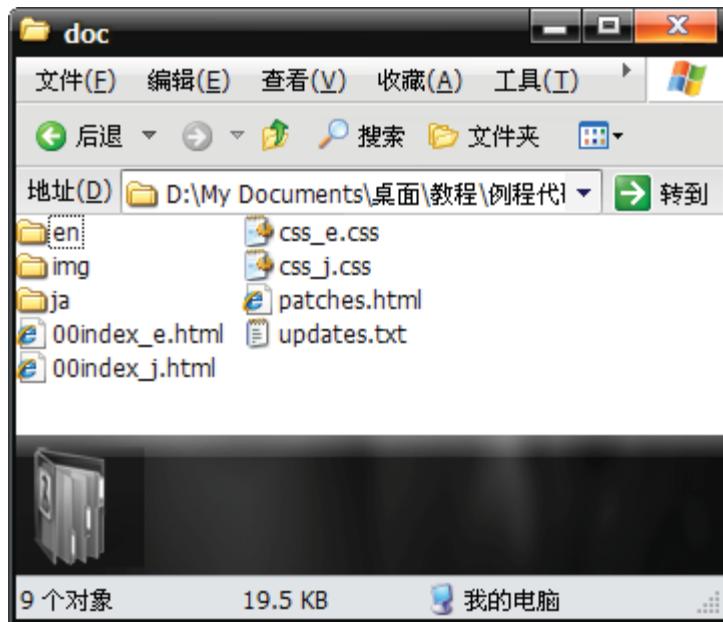
#### 分析 **FATFS** 的目录结构:

在移植 FATFS 文件系统之前，我们先要到 [FAT](#) 的官网获取源码，版本为 R0.07C。解压之后可看到里面有 doc 和 src 这两个文件夹。doc 文件夹里面是一些使用文档，src 里面是文件系统的源码。



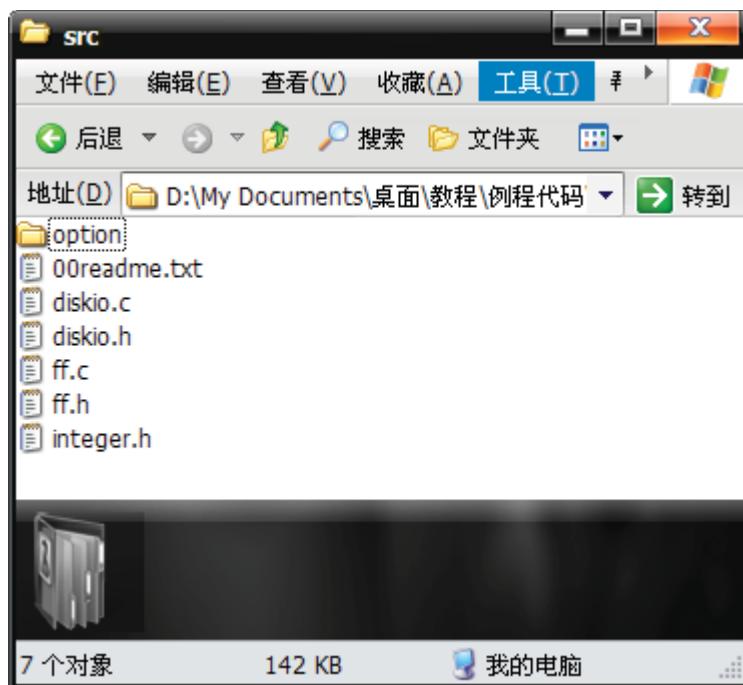


打开 doc 文件夹，可看到如下文件目录：



其中 en 和 ja 这两个文件夹里面是编译好的 html 文档，讲的是 FATFS 里面各个函数的使用方法，这些函数就如 LINUX 下的系统调用，是封装的非常好的函数，利用这些函数我们就可以操作我们的 MicroSD 卡了。有关具体的函数我们在用到的时候再讲解。这两个文件夹的唯一区别就是 en 文件夹下的文档是英文的，ja 文件夹下的是日文的。偏偏就是没中文的，真狗血呀。00index\_e.html 是一些关于 FATFS 的英文简介，updates.txt 是 FATFS 的更新信息，至于其他几个文件可以不看。

打开 src 文件夹，可看到如下目录：





---

option 文件夹下是一些可选的外部 c 文件，关于这些文件的具体应用本实验没用到。

00readme.txt 说明了当前目录下 `diskio.c`、`diskio.h`、`ff.c`、`ff.h`、`integer.h` 的功用、涉及了 FATFS 的版权问题( 是自由软件 )，还讲到了 FATFS 的版本更新信息。

`integer.h`: 是一些数值类型定义

`diskio.c` : 底层磁盘的操作函数，这些函数需要用户自己实现

`ff.c` : 独立于底层介质操作文件的函数，完全由 ANSI C 编写

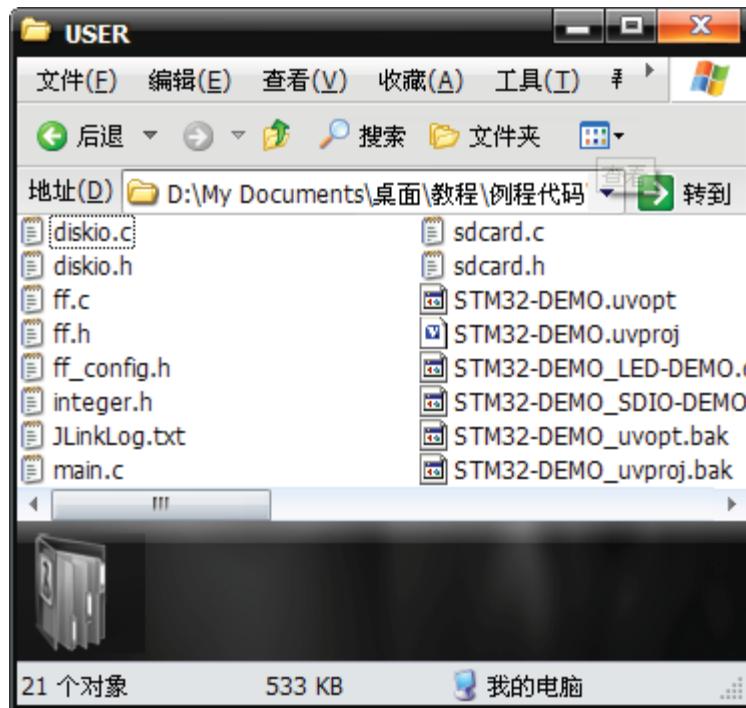
建议阅读这些源码的顺序为：`integer.h -> diskio.c -> ff.c`。

关于具体源码的分析不是我能力所及呀，大家就自己研究吧。我的主要工作是带领大家把这个文件系统移植到我们的开发板上，让这个文件系统先跑起来，这样才是硬道理呀。文件系统工作起来了的话，源码的分析那自然是大家的活啦。

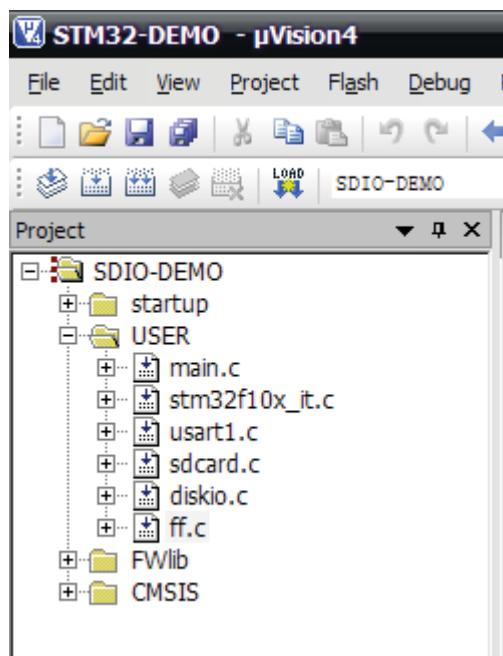
### 现在开始移植->

首先我们要获取一个完全没有修改过的文件系统源码，然后在 [10-MicroSD 卡](#) 这个文件夹下的实验代码下移植，这个实验代码实现的是卡的底层的块操作。注意，我们在移植这个文件系统的过程中会尽量保持文件系统源码的纯净，尽量做到在修改最少量的源码的情况下移植成功。

首先将 `integer.h`、`diskio.h`、`diskio.c`、`ff.h`、`ff.c` 添加到工程目录下的 `USER` 文件夹下，如下截图：



然后并回到 MDK 界面下将 diskio.c 、 ff.c 这两个文件添加到 USER 目录下，如下截图：



因为我们要用到这两个 c 文件，所以我们在 main.c 中将这两个 c 文件对应的头文件 diskio.h 、 ff.h 包含进来，如下截图：



```
/* ***** (C) COPYRIGHT 2011 *****  
* 文件名 : main.c  
* 描述 : MicroSD卡文件系统 FATFS R0.07  
* 实验平台: 野火STM32开发板  
* 库版本 : ST3.0.0  
*  
* 作者 : fire QQ: 313303034  
* 博客 : firestm32.blog.chinaunix.net  
*****  
#include "stm32f10x.h"  
#include "usart1.h"  
#include "sdcard.h"  
#include "ff.h"  
#include "diskio.h"  
  
#include <string.h>
```

好嘞，下面我们开始编译，这时会出现如下错误：

```
Build Output  
main.c(70): warning: #870-D: invalid multibyte character sequence  
compiling diskio.c...  
..\CMSIS\stm32f10x.h(237): error: #101: "FALSE" has already been declared in the current scope  
..\CMSIS\stm32f10x.h(237): error: #101: "TRUE" has already been declared in the current scope  
compiling ff.c...  
Target not created
```

意思是说 FALSE 跟 TRUE 这两个变量已经定义过了，为什么会出现这个错误呢？

因为在 integer.h 和我们的 M3 库头文件 stm32f10x.h 中都定义了这两个变量，所以就产生了重复定义：

integer.h

```
31 | /* Boolean type */  
32 | typedef enum { FALSE = 0, TRUE } BOOL;  
33 |
```

stm32f10x.h

```
0236 |  
0237 | 0237 | typedef enum { FALSE = 0, TRUE = !FALSE } bool;  
0238 |
```



怎么解决呢，很简单，只要搞掉一个即可，但要去掉哪个呢？我们考虑到外面的 M3 库很多文件都包含了 `stm32f10x.h` 这个头文件，假如是修改 `stm32f10x.h` 的话工作量会非常大，鉴于此就只能委屈 `integer.h` 了，修改如下，将它注释掉：

```
31 /* Boolean type */
32 //|typedef enum { FALSE = 0, TRUE } BOOL; /* add by fire */
33
```

好嘞，修改之后，我们再编译下，接着又出现如下错误：

```
diskio.h(29): error: #20: identifier "BOOL" is undefined
compiling ff.c...
diskio.h(29): error: #20: identifier "BOOL" is undefined
ff.c(584): error: #20: identifier "BOOL" is undefined
ff.c(861): error: #20: identifier "FALSE" is undefined
ff.c(915): error: #20: identifier "FALSE" is undefined
ff.c(1008): error: #20: identifier "TRUE" is undefined
ff.c(2254): error: #20: identifier "FALSE" is undefined
Target not created
```

意思是说在 `diskio.h`、`ff.c` 中 `BOOL`、`FALSE`、`TRUE` 没定义。刚刚才把人家注释掉了，不报错才怪。

解决方法如下：

1、将 `integer.h` 中有关 `BOOL` 的那句注释掉，注释掉也没太大关系，因为注释掉的不会怎么用到：

```
28
29 //|BOOL assign_drives (int argc, char *argv[]); /* add by fire */
30
```

2、在 `ff.c` 文件的开头重新定义一个布尔变量，取名为 `bool`，与 `stm32f10x.h` 中的名字一样：

```
0076 //|#include "stm32f10x.h" /* add by fire */
0077 |typedef enum {FALSE = 0, TRUE = !FALSE} bool; /* add by fire */
0078
```



同时在 ff.c 的第 585 行做如下修改:

```
0581 static
0582     FRESULT dir_next ( /* FR_OK:Succeeded, FR_NO
0583         DIR *dj, /* Pointer to directory object
0584         //BOOL streach /* FALSE: Do not streach tal
0585         bool streach /* add by fire */
0586     )
```

现在我们再编译下, 发现既没警告也没错误:

```
Build Output
Build target 'SDIO-DEMO'
compiling main.c...
compiling diskio.c...
compiling ff.c...
linking...
Program Size: Code=22822 RO-data=342 RW-data=636 ZI-data=3428
FromELF: creating hex file...
"..\Output\STM32-DEMO.axf" - 0 Error(s), 0 Warning(s).
```

到这里我们算是把文件系统移植成功了, 接下来的任务就是调用文件系统的函数来操作我们的卡了。其实这里的移植是非常非常简单的, 要是你学过 LINUX 的话, 那里面的 UBOOT 移植, 系统移植, 那才叫人头疼, 就光是目录里面的文件夹都几千个, 更别说是找到要修改的源代码了, 刚接触的话绝对叫你吐血, 就连下载个交叉编译器都涉及到移植。

### 接下来是实验代码讲解->

FATFS 是独立于底层介质的应用函数库, 对底层介质的操作都要交给用户去实现, 其仅仅是提供了一个函数接口而已, 函数为空, 要用户添加代码。

这几个函数的原型如下, 在 diskio.c 中定义:

```
1. /* Inidialize a Drive */
2. DSTATUS disk_initialize (
3.     BYTE drv /* Physical drive nmuber (0..) */
4. )
```



```
1. /* Return Disk Status */
2. DSTATUS disk_status (
3.     BYTE drv      /* Physical drive nmuber (0..) */
4. )
```

```
1. /* Read Sector(s) */
2. DRESULT disk_read (
3.     BYTE drv,      /* Physical drive nmuber (0..) */
4.     BYTE *buff,   /* Data buffer to store read data */
5.     DWORD sector,  /* Sector address (LBA) */
6.     BYTE count    /* Number of sectors to read (1..255) */
7. )
```

```
1. /* Write Sector(s) */
2. #if _READONLY == 0
3. DRESULT disk_write (
4.     BYTE drv,          /* Physical drive nmuber (0..) */
5.     const BYTE *buff,  /* Data to be written */
6.     DWORD sector,     /* Sector address (LBA) */
7.     BYTE count        /* Number of sectors to write (1..255) */
8. )
```

```
1. /* Miscellaneous Functions */ 
2. DRESULT disk_ioctl (
3.     BYTE drv,      /* Physical drive nmuber (0..) */
4.     BYTE ctrl,     /* Control code */
5.     void *buff    /* Buffer to send/receive control data */
6. )
```



---

这些函数都是操作底层介质的函数，都需要用户自己实现，然后 **FATFS** 的应用函数就可以调用这些函数来操作我们的卡了。关于这些底层介质函数是如何实现的，请参考源码，这里就不贴出来了。

在 **diskio.c** 的最后我们还得提供了获取时间的函数，因为 **ff.c** 中调用了这个函数，而 **FATFS** 库又没有给出这个函数的原型，所以需要用户实现，不然会编译出错，函数体为空即可：

```
281 /* 得到文件Calendar格式的建立日期,是DWORD get_fattime (void) 逆变换 */
282 /*-----*/
283 /* User defined function to give a current time to fatfs module */
284 /* 31-25: Year(0-127 org.1980), 24-21: Month(1-12), 20-16: Day(1-31) */
285 /* 15-11: Hour(0-23), 10-5: Minute(0-59), 4-0: Second(0-29 *2) */
286 DWORD get_fattime (void)
287 {
288     return 0;
289 }
```

实现好底层介质的操作函数之后，我们就可以回到应用层了，下面我们从 **main** 函数开始看起。有关系统初始化和串口初始化这部分请参考前面的教程，这里不再详述。

首先我们调用函数 `disk_initialize( 0 );` 将我们的底层硬件初始化好，这一步非常重要，如果不成功的话，接下来什么都干不了。

`f_mount(0, &fs[0]);` 将在文件系统中注册一个工作区，工作区的设备号为 0，这个设备号就相当于盘符，范围为 0~9。`fs` 为指向工作区的指针。

`f_open( &fsrc , "0:/Demo.TXT" , FA_CREATE_NEW | FA_WRITE);` 将在刚刚开辟的工作区的盘符 0 下打开一个名为 **Demo.TXT** 的文件，以只写的方式打开，如果文件不存在的话则创建这个文件。并将 **Demo.TXT** 这个文件关联到 `fsrc` 这个结构指针，以后我们操作文件就是通过这个结构指针来完成的。

`f_write(&fsrc, textFileBuffer, sizeof(textFileBuffer), &br);` 将缓冲区的数据写到刚刚打开的 **Demo.TXT** 文件中。写完之后调用  
`f_close(&fsrc);` 关闭文件，



`f_open(&fsrc, "0:/Demo.TXT", FA_OPEN_EXISTING | FA_READ);` 以只读的方式打开刚刚的文件。

`f_read( &fsrc, buffer, sizeof(buffer), &br );` 将文件的内容读到缓冲区，然后调用 `printf("\r\n %s ", buffer);` 将数据打印到电脑的超级终端。

最后调用 `f_close(&fsrc);` 关闭文件。当被打开的文件操作完成之后都要调用 `f_close();` 将它关闭，就像一块动态分配的内存用完之后都要调用 `free()` 来将它释放。

这里涉及到了 FATFS 文件系统库函数的操作，如果你学过 LINUX 系统调用的话，操作这些函数将是非常简单，没有学过的话也没太大的关系，因为 FATFS 源码目录 doc 这个文件夹中提供了每个应用函数的用法，如 `f_mount()`：

## f\_mount

The `f_mount` function registers/unregisters a work area to the FatFs module.

```
FRESULT f_mount (
    BYTE   Drive,           /* Logical drive number */
    FATFS* FileSystemObject /* Pointer to the work area */
);
```

### Parameters

#### Drive

Logical drive number (0-9) to register/unregister the work area.

#### FileSystemObject

Pointer to the work area (file system object) to be registered.

### Return Values

#### FR\_OK (0)

The function succeeded.

#### FR\_INVALID\_DRIVE

The drive number is invalid.



## Description

The `f_mount` function registers/unregisters a work area to the FatFs module. To unregister a work area, specify a NULL to the `FileSystemO`. This function only initializes the given work area and registers its address. The process is performed on first file access after `f_mount` or media change.

## References

[FATFS](#)

[Return](#)

### 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)，插上 MicroSD 卡( 野火用的是 1G )，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：

```
这是一个MicroSD卡文件系统实验(FATFS R0.07c)

disk_initialize starting.....
disk_initialize is ok

Demo.TXT 文件创建成功

感谢您选用 野火STM32开发板 ! ^_^

0 : /DEMO.TXT |

2097151 MB total drive space.
920 MB available.

-
```

实验讲解完毕，野火祝大家学习愉快^\_^。



## LCD 显示中英文 + BMP 图片

作者	fire
E-Mail	firestm32@foxmail.com
QQ	313303034
博客	firestm32.blog.chinaunix.net
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: 使用软件制作自定义类型的字库, 然后将字库放入 **SD** 卡中, 并且在 **SD** 卡中放入一张 **bmp** 图片作为 **LCD** 背景。并且调用截屏函数截取 **LCD** 背景并保存为 **bmp** 图片。

硬件连接:

MicroSD 卡

PC8-SDIO-D0	----DATA0
PC9-SDIO-D1	----DATA1
PC10-SDIO-D2	----DATA2
PC11-SDIO-D3	----CD/DATA3
PC12-SDIO-CLK	----CLK
PD2-SDIO-CMD	----CMD

TFT 数据线

PD14-FSMC-D0	----LCD-DB0
PD15-FSMC-D1	----LCD-DB1
PD0-FSMC-D2	----LCD-DB2
PD1-FSMC-D3	----LCD-DB3
PE7-FSMC-D4	----LCD-DB4
PE8-FSMC-D5	----LCD-DB5



---

PE9-FSMC-D6	----LCD-DB6
PE10-FSMC-D7	----LCD-DB7
PE11-FSMC-D8	----LCD-DB8
PE12-FSMC-D9	----LCD-DB9
PE13-FSMC-D10	----LCD-DB10
PE14-FSMC-D11	----LCD-DB11
PE15-FSMC-D12	----LCD-DB12
PD8-FSMC-D13	----LCD-DB13
PD9-FSMC-D14	----LCD-DB14
PD10-FSMC-D15	----LCD-DB15

TFT 控制信号线

PD4-FSMC-NOE	----LCD-RD
PD5-FSMC-NEW	----LCD-WR
PD7-FSMC-NE1	----LCD-CS
PD11-FSMC-A16	----LCD-DC
PE1-FSMC-NBL1	----LCD-RESET
PD13-FSMC-A18	----LCD-BLACK-LIGHT

库文件 : startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_rcc.c

FWlib/misc.c

Fwlib/stm32f10x\_systick.c

FWlib/stm32f10x\_exti.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_sdio.c

FWlib/stm32f10x\_dma.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_fsmc.c



用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/systick.c

USER/usart1.c

USER/lcd.c

USER/ff.c

USER/sdcard.c

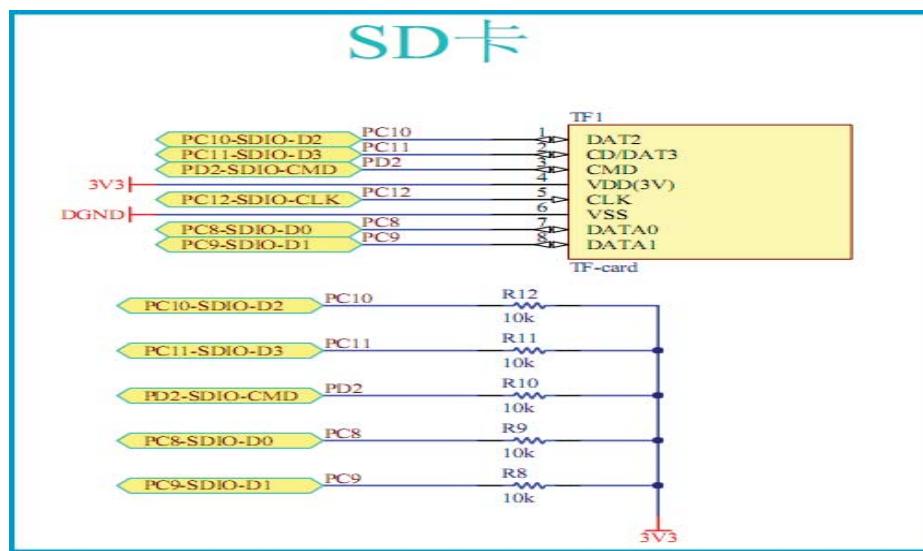
USER/diskio.c

USER/sd\_fs\_app.c

USER/Sd\_bmp.c

野火 STM32 开发板 LCD 和 SD 卡 硬件连接图:

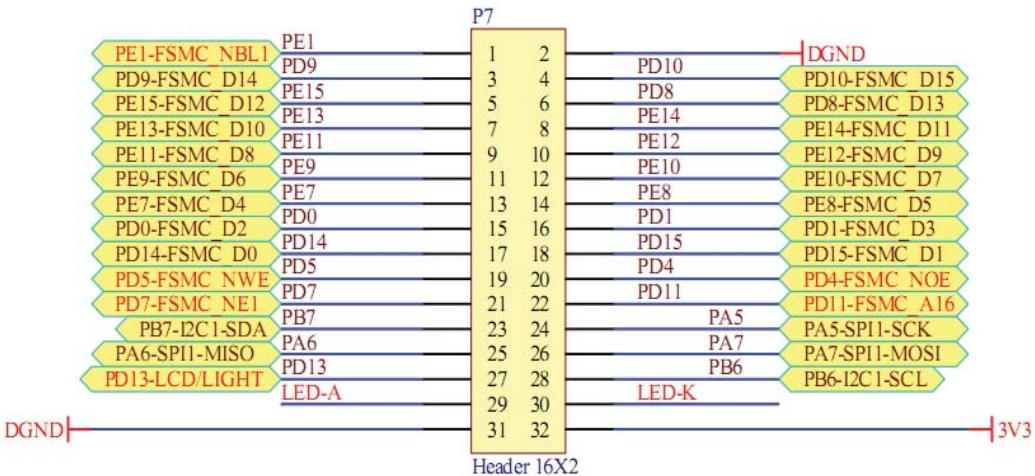
SD 卡接口连接如下



LCD 接口连接如下



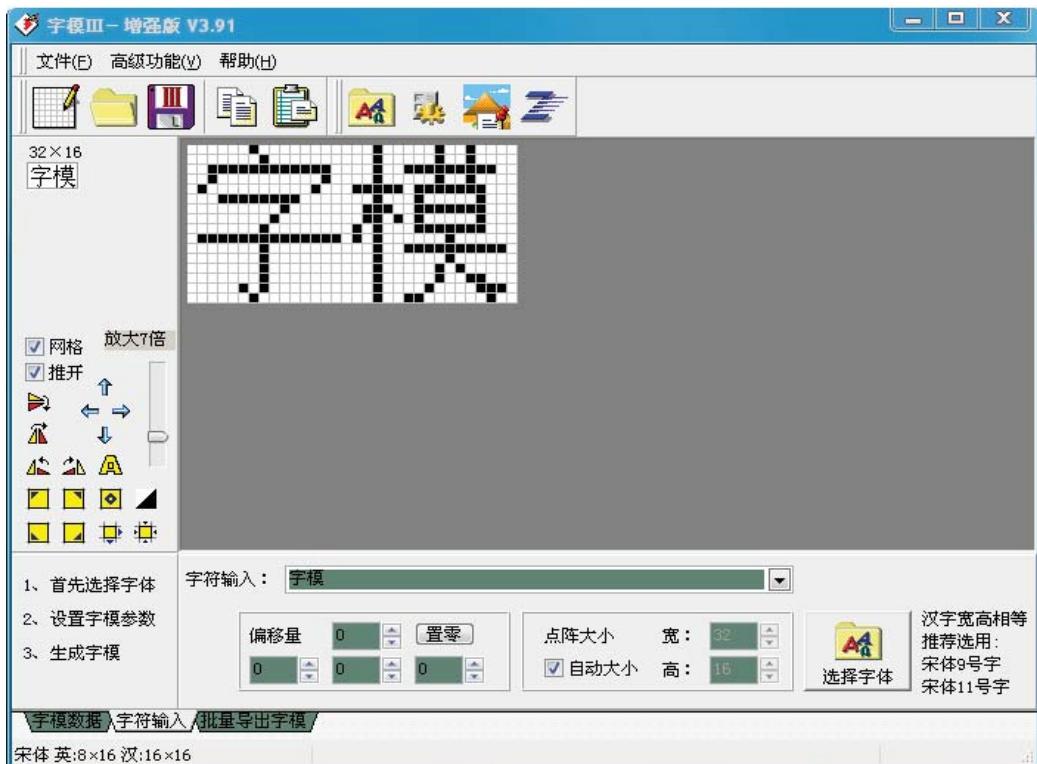
## 2.4寸TFT+TOUCH



### 字库制作详细流程

我们采用“字模 III-增强版 v3.91”  软件来制作中文字库。

#### 1 打开字模软件



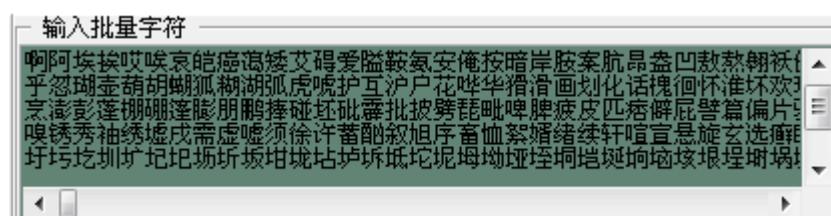
2 点击“自动批量生成字库”按钮选项 .

软件界面左下角将出现一下几个按钮选项:



3 点击选择“二级汉字库”按钮。

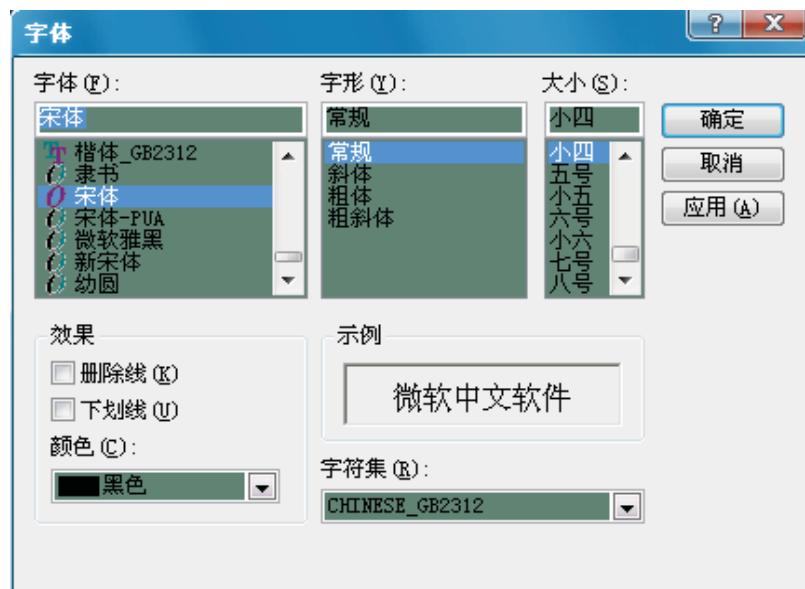
在“输入批量字符”框里面将会列出二级汉字的所有汉字，其中共收录了 6768 个汉字字符，非特殊情况下都能够满足大家的要求啦，如图：



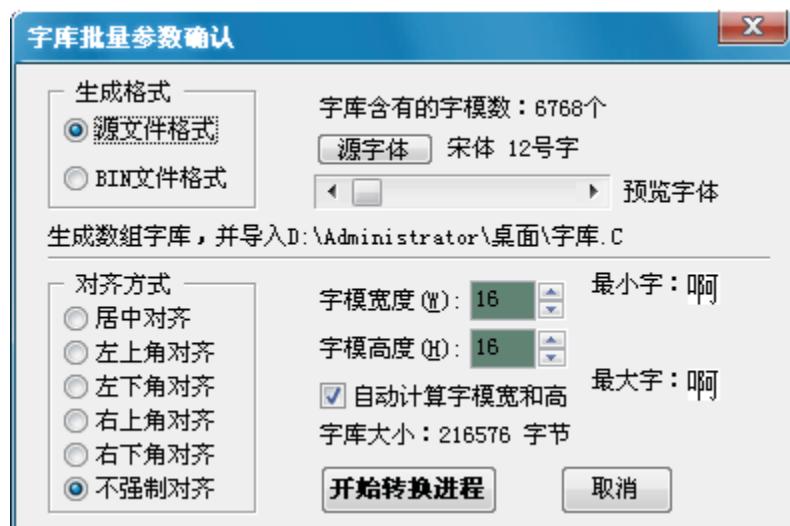
4 点击“字库智能生成”按钮 ，弹出“字库批量参数确认”对话框。



我们在“源字体”选项里面做如下设置，需要注意的是大小问题，因为我们本次的设计目标是实现 16\*16 的汉字，所以在此选择‘小四’字体。



设置好之后如下：



5 点击“开始转换进程”按钮 **开始转换进程**，就会在安装目录下或者你设置好的目录下生成.c 后缀的字库文件。

6 对于 LCD 显示来说，只要能够在指定的位置描写制定颜色的点，那么就能够很好地根据汉字字模信息来描写汉字。在此，为了能够更好的清楚字模的取向和高低位的排列顺序，我们可以现先在 pc 测试我们刚才制作好的库文件。

在这里我们取“当”字符的数据来测试。



```
1459 /* 当 */
1460 0x00,0x80,0x10,0x90,0x08,0x98,0x0C,0x90,0x08,0xA0,0x00,0x80,0x3F,0xFC,0x00,0x04,
1461 0x00,0x04,0x1F,0xFC,0x00,0x04,0x00,0x04,0x04,0x3F,0xFC,0x00,0x04,0x00,0x00,
```

VC6.0 测试源码如下:

```
1. #include <stdio.h>
2.
3. unsigned char cc[] =
4. {/*"当"字符串*/
5. 0x00,0x80,0x10,0x90,0x08,0x98,0x0C,0x90,0x08,0xA0,0x00,0x80,0x3F,0xFC,
0x00,0x04,
6. 0x00,0x04,0x1F,0xFC,0x00,0x04,0x00,0x04,0x04,0x3F,0xFC,0x00,0x04,
0x00,0x00
7. };
8.
9. void main()
10. {
11.     int i,j;
12.     unsigned char kk;
13.     for ( i=0; i<16; i++)
14.     {
15.         for(j=0; j<8; j++)
16.         {
17.             kk = cc[2*i] << j ; //左移 j 位
18.
19.             if( kk & 0x80) //如果最高位为 1
20.             {
21.                 printf("8");
22.             }
23.             else
24.             {
25.                 printf(" ");
26.             }
27.         }
28.
29.         for(j=0; j<8; j++)
30.         {
31.
32.             kk = cc[2*i+1] << j ; //左移 j 位
33.
34.             if( kk & 0x80) //如果最高位为 1
35.             {
36.                 printf("8");
37.             }
38.             else
39.             {
40.                 printf(" ");
41.             }
42.
43.         }
44.
45.         printf("\n");
46.
47.     }
48.     printf("\n\n");
49.
50. }
```



测试结果如下:

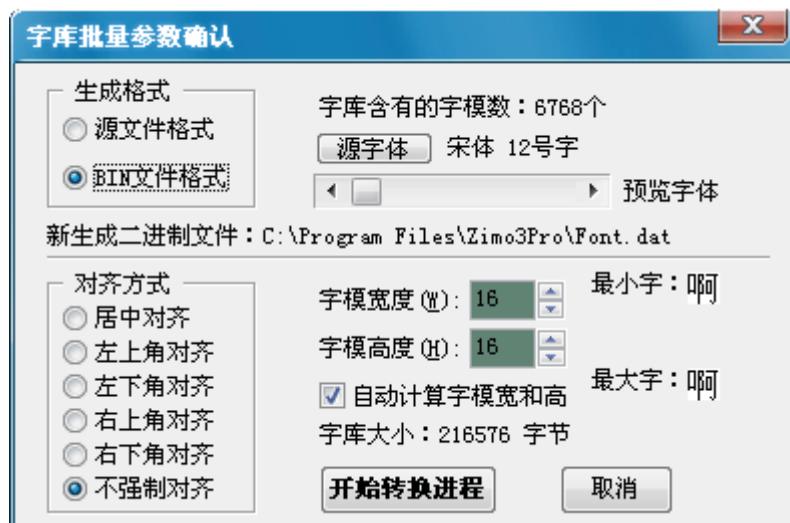
```
8 8 8  
8 8 88  
88 8 8  
8 8 8  
8  
888888888888  
8  
8  
888888888888  
8  
8  
8  
888888888888  
8
```

Press any key to continue

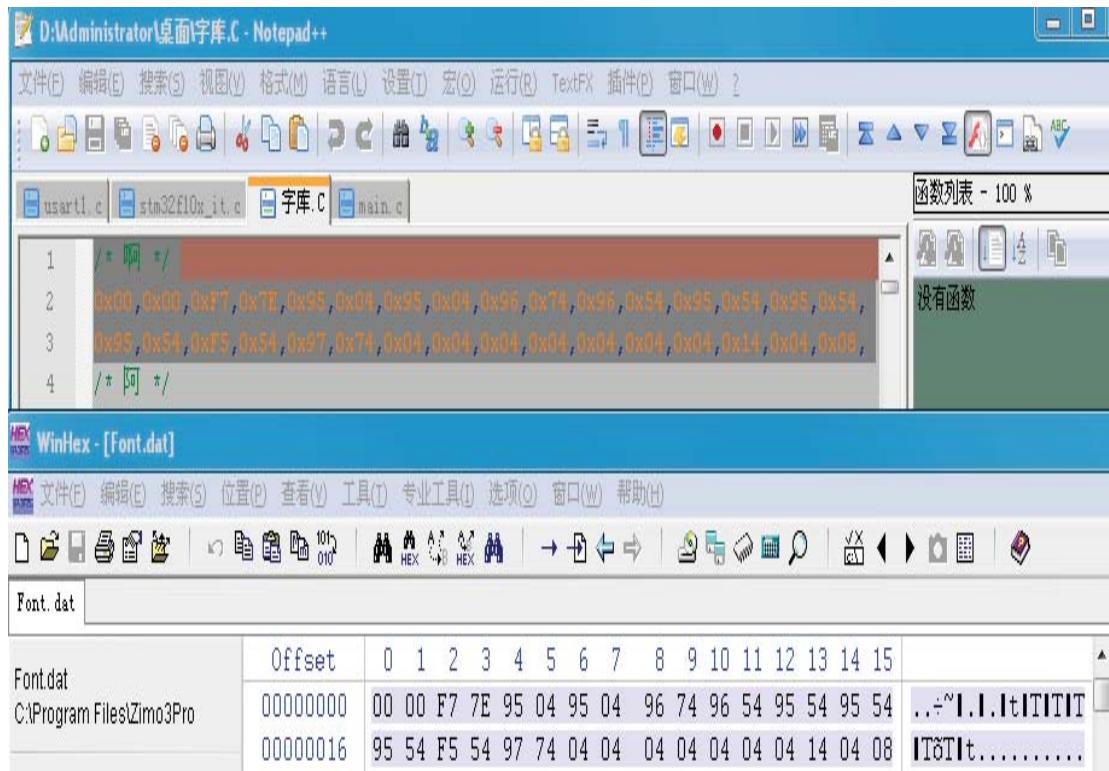
搜狗拼音 半:

看到以上的测试结果，相信大家对汉字的取模方向和高低位的排列顺序有了比较直观的了解。

7 回到“字模 III-增强版 v3.91”软件，采用与之前同样的方式生成 bin 格式的字库文件(即“生成格式”选项设置为“bin 文件格式” )。



在软件安装目录下会生成 Font.dat 文件,我们用“WinHex”软件查看他的具体内容，与刚才制作的.c 字库的文件内容是一致的，对比如下：



将生成的汉字字库拷贝到 SD 卡根目录下并重命名为“HZLIB.bin”。

## 软件实现过程

先看看 **main** 函数先吧，源码如下：

```
1. int main(void)
2. {
3.     RCC_Configuration(); //时钟配置
4.
5.     LCD_Init(); //LCD 初始化
6.     USART1_Config(); //串口 1 初始化
7.     sd_fs_init(); //文件系统化
8.     Set_direction(0); //LCD AP 移动方向设置
9.     LCD_CLEAR(0, 0, 240, 320); //清屏
10.    Lcd_show_bmp(0, 0, "/test.bmp"); //在 LCD 上显示 SD 卡上存放的
11.      test.bmp 图片
12.      //横屏显示
13.      LCD_ShowString(50, 10, 1, "6X12 ASCII");
14.      LCD_Show_8x16_String(50, 30, 1, "8X16 ASCII");
15.      PutChinese_strings21(50, 50, "内存卡字库例程", 0, 1);
16.      PutChinese_strings11(50, 70, "内存卡字库例程", 0, 0xffff);
17.      //竖屏显示
18.      LCD_ShowString2(80, 240, 1, "6X12 ASCII");
19.      LCD_Show_8x16_String2(80, 220, 1, "8X16 ASCII");
20.      PutChinese_strings22(80, 200, "内存卡字库例程", 0, 1);
21.      PutChinese_strings12(80, 180, "内存卡字库例程", 0, 0xffff);
```



```
22.  
23. PutChinese_strings11(50,120,"正在截图",0,0xffff);  
24. Screen_shot(0, 0, 320, 240, "/myScreen"); //截图操作  
25. PutChinese_strings11(50,120,"截图完成",0,0xffff);  
26. }
```

Main 函数的工作就是在 LCD 上显示一些指定类型的字符，并完成显示 bmp 图像和截图功能。

这里先说汉字字符的显示，第 14 行：PutChinese\_strings21(50,50,"内存卡字库例程",0,1);

该函数功能是显示汉子字符串，源码如下：

```
1. /*****  
2. * 函数名: PutChinese_strings21  
3. * 描述 : 显示汉字字符串  
4. * 输入 : pos: 0~(319-16)  
5. *          Ypos: 0~(239-16)  
6. *          str: 中文字符串首址  
7. *          Color: 字符颜色  
8. *          mode: 0--文字背景色为白色  
9. *                  1--文字悬浮  
10. * 输出 : 无  
11. * 举例 : PutChinese_strings2(200,100,"好人",0,0);  
12. * 注意 : 无  
13. *****/  
14. void PutChinese_strings21(uint16_t Xpos,uint16_t Ypos,uint8_t *str,uint16_t Color,u8 mode) //横屏  
15. {  
16.  
17.     uint16_t Tmp_x, Tmp_y;  
18.     uint8_t *tmp_str=str;  
19.     Tmp_x = Xpos;  
20.     Tmp_y = Ypos;  
21.  
22.     while(*tmp_str != '\0')  
23.     {  
24.         PutChinese21(Tmp_x,Tmp_y,tmp_str,Color,mode); //显示该汉字  
25.  
26.         tmp_str += 2 ;  
27.         Tmp_x += 16 ;  
28.     }  
29. }
```

该函数其实没做到什么工作，只是把字符串中的汉字一个一个提取出来并调用单字符显示函数 PutChinese21 显示出来，PutChinese21 函数的源码如下：

```
1. /*****  
2. * 函数名: PutChinese21  
3. * 描述 : 显示单个汉字字符串  
4. * 输入 : pos: 0~(319-16)  
5. *          Ypos: 0~(239-16)  
6. *          str: 中文字符串首址  
7. *          Color: 字符颜色  
8. *          mode: 0--文字背景色为白色
```



```
9.  *          1--文字悬浮
10. * 输出   : 无
11. * 举例   : PutChinese21(200,100,"好",0,0);
12. * 注意   : 如果输入大于1的汉字字符串, 显示将会截断, 只显示最前面一个汉字
13. ****
14. void PutChinese21(uint16_t Xpos,uint16_t Ypos,uint8_t *str,uint16_t Color,u8 mode)
15. {
16.     uint8_t i,j;
17.     uint8_t buffer[32];           //32字节用于保存字模数据
18.     uint16_t tmp_char=0;
19.     Set_direction(0);
20.     GetGBKCode_from_sd(buffer,str);    // 从 sd 卡中取出字模数据
21.
22.     for (i=0;i<16;i++)
23.     {
24.         tmp_char=buffer[i*2];
25.         tmp_char=(tmp_char<<8);
26.         tmp_char|=buffer[2*i+1];
27.         for (j=0;j<16;j++)
28.         {
29.             if ((tmp_char >> 15-j) & 0x01 == 0x01)
30.             {
31.                 LCD_Draw_ColorPoint(Ypos+i,Xpos+j,Color);
32.             }
33.             else
34.             {
35.                 if ( mode == 0 )
36.                     LCD_Draw_ColorPoint(Ypos+i,Xpos+j,0xffff); //背景
37.                 else if ( mode == 1 )
38.                 {
39.                     //不写入
40.                 }
41.             }
42.         }
43.     }
44. }
```

在 `PutChinese21` 这个函数中，首先从 `sd` 卡中读出我们需要显示在 `LCD` 上的指定汉字的字模数据，之后 22~43 行的代码就根据字模数据来描写，描写这一部分在这里就不再说啦，和前面 `VC` 测试部分思路是一样的。读者现在可能在想，字库里面保存着大量的汉字字幕信息，我现在输入 `GetGBKCode_from_sd(buffer,str)` 就能够拷贝出这个字符的字模数据，是怎样定位字模信息所在的位置的呢，换句话说，假如我现在要显示“吾”字，是怎样根据这个字来确定“吾”字符在字库中的保存位置的呢？其实这里面有一定的映射关系，那就是接下来要说的汉字“区码”和“位码”。

在汉字区位码表中，我们是如何定位我们指定汉字的编号的呢？接下来看看 `vc6.0` 测试源码：

```
1. #include <stdio.h>
2. void main ()
3. {
4.     unsigned char * s , * e = "A" , * c = "古" ;
```



```
5.     unsigned char high_byte,lower_byte; //内码高字节, 内码低字节
6.     printf ( "字母'%s'的ASCII 码'=%",e ) ;
7.     s = e ;
8.
9.     while ( * s != 0 ) //C 的字符串以 0 为结束符*
10.    {
11.        printf ( "%3d," , *s ) ;
12.        s ++ ;
13.    }
14.    printf ( "\n 汉字内码(10 进制) '%s'=%",c ) ;
15.
16.    s = c ;
17.    while ( *s != 0 )
18.    {
19.        printf ( "%3d," , * s );
20.        s ++ ;
21.    }
22.
23.    printf ( "\n 汉字内码(16 进制) '%s'=%",c ) ;
24.
25.    s = c ;
26.    while ( *s != 0 )
27.    {
28.        printf ( "%0X," , * s );
29.        s ++ ;
30.    }
31.
32.
33.    s = c ;
34.    high_byte = *s;
35.
36.    s ++ ;
37.    lower_byte = *s;
38.
39.    printf("\n\n 汉字'%s'对应的\n 内码高字节:%d\n 内码低字
节:%d\n",c,high_byte,lower_byte);
40.    printf("\n\n 汉字'%s'对应的\n 区码为:%d-160 = %d\n 位码为:%d-
160 = %d\n",c,high_byte,high_byte-160,lower_byte,lower_byte-160);
41.
42.    printf("\n\n 汉字'%s'在区位码表中的位置为%d%d\n",c,high_byte-
160,lower_byte-160);
43.    printf("汉字区位码表可参考网
站:http://cs.scu.edu.cn/~wangbo/others/quweima.htm\n");
44.    printf("通过在线查阅, 编号为%d%d 对应的汉字刚好就是'%s'\n\n",high_byte-
160,lower_byte-160,c);
45.
46. }
```

测试结果如下:



D:\Administrator\桌面>main.exe  
字母'A的ASCII码'= 65,  
汉字内码<10进制>'古'=185,197,  
汉字内码<16进制>'古'=B9,C5,  
  
汉字'古'对应的  
内码高字节:185  
内码低字节:197  
  
汉字'古'对应的  
区码为:185-160 = 25  
位码为:197-160 = 37  
  
汉字'古'在区位码表中的位置为2537  
汉字区位码表可参考网站:<http://cs.scu.edu.cn/~wangbo/others/quweima.htm>  
通过在线查阅,编号为2537对应的汉字刚好就是'古'  
  
D:\Administrator\桌面>pause  
请按任意键继续...  
搜狗拼音 半:

打开汉字区位码表在线查询网站: <http://cs.scu.edu.cn/~wangbo/others/quweima.htm>

查询“古”汉字的区位码刚好如计算所得, 为

上面的测试结果说明了每一个汉字的内码具体作用。下面就来看看 PutChinese21 函数中调用到的函数 GetGBKCode\_from\_sd 的源码:

```
1. /*****
2. * 函数名: GetGBKCode_from_sd
3. * 描述 : 从 sd 卡上的字库文件中拷贝指定汉字的字模数据
4. * 输入 : pBuffer---数据保存地址
5. *          c--汉字字符低字节码
6. * 输出 :      0          (成功)
7. *          -1         (失败)
8. *****/
9.
10.int GetGBKCode_from_sd(unsigned char* pBuffer,unsigned char * c)
11.{ 
12.    unsigned char High8bit,Low8bit;
13.    unsigned int pos;
14.    High8bit=*c;
15.    Low8bit=*(c+1);
16.
17.    pos = ((High8bit-0xb0)*94+Low8bit-0xa1)*2*16 ;
18.    f_mount(0, &myfs[0]);
19.    myres = f_open(&myfs[0], "0:/HZLIB.bin", FA_OPEN_EXISTING | FA_READ);
20.
21.    if ( myres == FR_OK )
22.    {
23.        f_lseek (&myfs[0], pos);           //制定读取位置
```



```
24.         myres = f_read( &myfsrc, pBuffer, 32, &mybr ); //16*16 大小的汉
    字其字模占用 16*2 个字节
25.         f_close(&myfsrc);
26.
27.         return 0;
28.     }
29.
30.     else
31.         return -1;
32.
33. }
```

`pos = ((High8bit-0xb0)*94+Low8bit-0xa1)*2*16` 这条语句就是根据约定的映射关系由汉字内码求得该汉字字模在字库中的存放起始位置。之后就到指定的位置去拷贝字模数据就可以了。

以上就是 SD 卡字库制作和实现的具体流程。

## 实现 SD 卡 BMP 图像的读取与保存

BMP 文件格式，又称为 Bitmap（位图）或是 DIB(Device-Independent Device，设备无关位图），是 Windows 系统中广泛使用的图像文件格式。BMP 文件保存了一幅图象中所有的像素。这种文件格式可以保存单色位图、16 色或 256 色索引模式像素图、24 位真彩色图象，每种模式种单一像素的大小分别为 1/8 字节，1/2 字节，1 字节和 3 字节。目前最常见的是 256 位色 BMP 和 24 位色 BMP。这种文件格式还定义了像素保存的几种方法，包括不压缩、RLE 压缩等。常见的 BMP 文件大多是不压缩的。

Windows 所使用的 BMP 文件，在开始处有一个文件头，大小为 54 字节。保存了包括文件格式标识、颜色数、图像大小、压缩方式等信息，因为我们仅讨论 24 位色不压缩的 BMP，所以文件头中的信息基本不需要注意，只有“大小”这一项对我们比较有用。图像的宽度和高度都是一个 32 位整数，在文件中的地址分别为 0x0012 和 0x0016。54 个字节以后，如果是 16 色或 256 色 BMP，则还有一个颜色表，但 24 位色 BMP 没有这个，我们这里不考虑。接下来就是实际的像素数据了。因此总的来说 BMP 图片的优点是简单。

图片分析：

下面来用 WinHex 软件来分析一下 bmp 图像的文件内容：

测试图片 123.bmp，用 ACDSee 软件打开，其图像内容为



可知，这幅图像的大小是 15\*8，是 24 位真彩色 bmp 图像。

图片数据为：

1234.bmp		Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1234.bmp	D:\Administrator\桌面	00000000	42	4D	B6	01	00	00	00	00	00	00	36	00	00	28	00	
		00000016	00	00	0F	00	00	00	08	00	00	00	01	00	18	00	00	
文件大小:	438 B	00000032	00	00	00	00	00	00	C4	0E	00	00	C4	0E	00	00	00	
	438 字节	00000048	00	00	00	00	00	00	31	31	31	31	31	31	31	31	31	
缺省编辑模式		00000064	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
状态:	原始的	00000080	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
撤销级数:	0	00000096	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
反向撤销:	n/a	00000112	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
创建时间:	2011-09-10 20:52:08	00000128	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
		00000144	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
最后写入时间:	2011-09-10 20:52:08	00000160	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
属性:	A	00000176	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
图标:	1	00000192	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
模式:	十六进制	00000208	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
字符集:	ANSI ASCII	00000224	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
偏移地址:	decimal	00000240	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
每页字节数:	37x16=592	00000256	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
当前窗口:	1	00000272	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
窗口总数:	1	00000288	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
剪贴板:	可用	00000304	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
暂存文件夹:	S:\TEMP	00000320	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
		00000336	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
		00000352	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
		00000368	30	30	30	30	31	31	31	31	31	31	31	31	31	31	31	
		00000384	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	
		00000400	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	
		00000416	30	30	30	30	31	31	31	31	31	31	31	31	31	31	31	
		00000432	31	31	31	00	00	00	00	00	00	00	00	00	00	00	00	

## 1 文件头部信息部分(前面 54 字节):

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00000000	42	4D	B6	01	00	00	00	00	00	00	36	00	00	28	00	
00000016	00	00	0F	00	00	00	08	00	00	00	01	00	18	00	00	00
00000032	00	00	00	00	00	00	C4	0E	00	00	C4	0E	00	00	00	00
00000048	00	00	00	00	00	00	31	31	31	31	31	31	31	31	31	31

如上图，阴影部分就是文件头部信息。



## 2 图像像素数据部分(如果是 24 位真彩色, 则 54 字节之后就是像素部分):

00000048	00 00 00 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000064	31 31
00000080	31 31
00000096	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000112	31 31
00000128	31 31
00000144	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000160	31 31
00000176	31 31
00000192	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000208	31 31
00000224	31 31
00000240	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000256	31 31
00000272	31 31
00000288	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000304	31 31
00000320	31 31
00000336	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000352	31 30 30
00000368	30 30 30 30 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000384	31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000400	31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 30 30
00000416	30 30 30 30 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
00000432	31 31 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

## 3 对文件头部信息部分的解析:

这里可以分为两块: **BMP** 文件头和位图信息头

从 0 开始:

0 和 1 字节 是文件的类型, 如果是位图文件类型, 必须分别为 0x42 和 0x4D。

接下来的部分可以用一个结构体来描述。

## 3 到 14 字节的意义如下:

```
1. typedef struct tagBITMAPFILEHEADER
2. {
3.     //attention: sizeof(DWORD)=4   sizeof(WORD)=2
4.     DWORD bfSize;           //文件大小
5.     WORD bfReserved1;      //保留字, 不考虑
6.     WORD bfReserved2;      //保留字, 同上
```



```
7.     DWORD bfOffBits;    //实际位图数据的偏移字节数, 即前三个部分长度之和  
8. } BITMAPFILEHEADER, tagBITMAPFILEHEADER;
```

头部信息剩下的部分就是位图信息头信息:

14 到 53 字节内容的意义如下:

```
1. typedef struct tagBITMAPINFOHEADER  
2. {  
3.     //attention: sizeof(DWORD)=4  sizeof(WORD)=2  
4.     DWORD biSize;           //指定此结构体的长度, 为 40  
5.     LONG biWidth;          //位图宽  
6.     LONG biHeight;         //位图高  
7.     WORD biPlanes;         //平面数, 为 1  
8.     WORD biBitCount;       //采用颜色位数, 可以是 1, 2, 4, 8, 16, 24 新的可以是 32  
9.     DWORD biCompression;   //压缩方式, 可以是 0, 1, 2, 其中 0 表示不压缩  
10.    DWORD biSizeImage;     //实际位图数据占用的字节数  
11.    LONG biXPelsPerMeter; //X 方向分辨率  
12.    LONG biYPelsPerMeter; //Y 方向分辨率  
13.    DWORD biClrUsed;      //使用的颜色数, 如果为 0, 则表示默认值(2^颜色位数)  
14.    DWORD biClrImportant; //重要颜色数, 如果为 0, 则表示所有颜色都是重要的  
15.  
16. } BITMAPINFOHEADER, tagBITMAPINFOHEADER;
```

由上述分析与 WinHex 软件的分析内容结合得到该图片的信息如下:

文件大小:438

保留字:0

保留字:0

实际位图数据的偏移字节数:54

位图信息头:

结构体的长度:40

位图宽:15

位图高:8



biPlanes 平面数:1

biBitCount 采用颜色位数:24

压缩方式:0

biSizeImage 实际位图数据占用的字节数:0

X 方向分辨率:3780

Y 方向分辨率:3780

使用的颜色数:0

重要颜色数:0

#### 4 文件像素部分的分析

有些读者可能已经发现，在像素部分夹杂着一些值为 0 的数据信息，如下图灰色部分所示：

00000096 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 31  
整张图片都是灰色的，为什么会有 0 像素的出现呢？

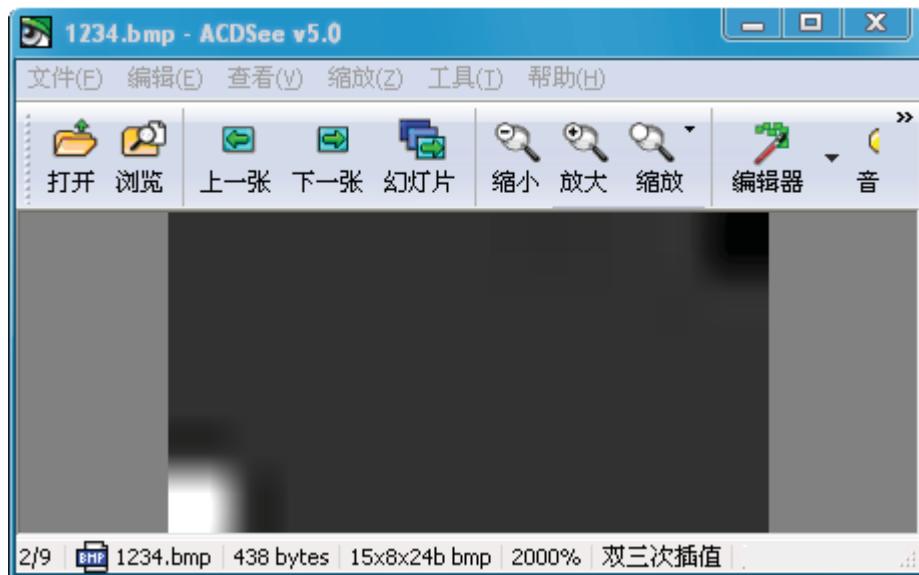
由于对齐的数据，操作系统或者硬件更容易把它调入 cache，使得 cache 的命中率提高，从而提高访问效率。也就是基于性能上得考虑，所以 Windows 规定一个扫描行所占的字节数必须是 4 的倍数(即以 long 为单位)，不足的以 0 填充。上面分析可知，位图宽:15，因此，未补充字节前字节数为  $15*3$  等于 45 字节，要到 4 的倍数，必须向上取 4 的倍数即 48，所以就有了不上 3 个字节 0 的结果。

至此，整个图像文件的大小为： $54 + (15*3 + 3) * 8$  等于 438 字节，和前面所得到的信息文件大小:438 是一致的。

另外一点需要注意的是显示图像的顺序是由下到上，由左到右，即像素数据部分的第一像素数据是我们见到的图像的左下角的像素的数据；而像素数据的最后一个有效数据是我们见到的图像的右上角的像素的数据。

下面我们修改一下图片来验证一下：

我们将左下角画上白色，右上角画上黑色，图片放大之后如下



图片数据分析如下:

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00000000	42	4D	B6	01	00	00	00	00	00	00	36	00	00	00	28	00
00000016	00	00	0F	00	00	00	08	00	00	00	01	00	18	00	00	00
00000032	00	00	80	01	00	00	C4	0E	00	00	C4	0E	00	00	00	00
00000048	00	00	00	00	00	00	FF	FF	FF	31	31	31	31	31	31	31
00000064	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000080	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000096	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000112	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000128	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000144	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000160	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000176	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000192	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000208	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000224	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000240	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000256	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000272	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000288	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000304	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000320	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
00000336	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31	31
00000352	31	31	31	31	31	31	31	31	31	31	31	31	31	31	30	30
00000368	30	30	30	30	31	31	31	31	31	31	31	31	31	31	00	00
00000384	00	00	00	00	00	00	31	31	31	31	31	31	31	31	31	31
00000400	31	31	31	31	31	31	31	31	31	31	31	31	31	31	30	30
00000416	30	30	30	30	31	31	31	31	31	31	31	31	31	31	31	30
00000432	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00



我们可以看到像素部分开始部分是白色像素(灰色部分):

00000048 | 00 00 00 00 00 00 FF FF FF 31 31 31 31 31 31 31 31

对应我们图像的左下角。

后尾部分是黑色像素 (灰色部分) :

00000416 | 30 30 30 30 31 31 31 31 31 31 31 31 31 31 00 00 00  
00000432 | 00 00 00 00 00 00

对应我们图像的右上角。

对 **BMP** 图像文件内容有了具体的了解之后我们开始编写我们的应用程序，根据图片的头部信息，合理地读出其中的像素部分（由于 **pc** 机上，**24** 位真彩色使用比较广，在此只分析 **24** 位为真彩色）。

再回到前面的 **main** 函数，其中调用了一个 **bmp** 图片显示函数 **Lcd\_show\_bmp**，其源码如下：

```
1.  ****
2.  * 函数名: Lcd_show_bmp
3.  * 描述  : LCD 显示 RGB888 位图图片
4.  * 输入  : x           --显示纵坐标 (0-239)
5.  *         y           --显示横坐标 (0-319)
6.  *         pic_name     --图片名称
7.  * 输出  : 无
8.  * 举例  : Lcd_show_bmp(0, 0, "/test.bmp");
9.  * 注意  : 图片为 24 位真彩色位图图片
10. *          图片宽度不能超过 320
11. *          图片在 LCD 上的粘贴范围为: 纵向: [x, x+图像高度] 横
    向 [Y, Y+图像宽度]
12. *          当图片为 320*240 时--建议 x 输入 0 y 输入 0
13. * ****
14. BYTE pColorData[960];
15. FATFS bmpfs[2];
16. FIL bmpfsrc, bmpfdst;
17. FRESULT bmpres;
18. void Lcd_show_bmp(unsigned short int x, unsigned short int y, unsigned
    char *pic_name)
19. {
20.     int k;
21.     int i;
22.     int j;
23.     int width;
24.     int height;
25.     int l_width;
26.     BYTE red, green, blue;
```



```
27.     BITMAPFILEHEADER bitHead;
28.     BITMAPINFOHEADER bitInfoHead;
29.     WORD fileType;
30.     unsigned int read_num;
31.     unsigned char tmp_name[20];
32.     sprintf((char*)tmp_name,"0:%s",pic_name);
33.     f_mount(0, &bmpfs[0]);
34.
35.     bmpres = f_open(&bmpfs, (char *)tmp_name, FA_OPEN_EXISTING | F
A_READ); //打开 bmp 文件
36.
37.     if(bmpres == FR_OK)
38.     {
39.         USART1_printf(USART1,"Open file success\r\n");
40.
41.         //读取位图文件类型信息
42.         f_read(&bmpfs, &fileType, sizeof(WORD), &read_num); //读取 1、2 字
节(文件类型)
43.
44.         if(fileType != 0x4d42) //0x4d42 是 bmp 图像类型标志
45.         {
46.             USART1_printf(USART1,"file is not .bmp file!\r\n");
47.             return;
48.         }
49.     else
50.     {
51.         USART1_printf(USART1,"Ok this is .bmp file\r\n");
52.     }
53.
54.         //读取位图文件头信息结构(该结构已经在前面描述过啦)
55.         f_read(&bmpfs, &bitHead, sizeof(tagBITMAPFILEHEADER), &read_num
);
56.
57.         showBmpHead(&bitHead);
58.         USART1_printf(USART1, "\r\n");
59.
60.         //读取位图信息头信息(该结构已经在前面描述过啦)
61.         f_read(&bmpfs, &bitInfoHead, sizeof(BITMAPINFOHEADER), &read_nu
m);
62.
63.         showBmpInforHead(&bitInfoHead);
64.         USART1_printf(USART1, "\r\n");
65.     }
66.
67.     else
68.     {
69.         USART1_printf(USART1,"file open fail!\r\n");
70.         return;
71.     }
72.
73.
74.     width = bitInfoHead.biWidth;
75.     height = bitInfoHead.biHeight;
76.
77.     l_width = WIDTHBYTES(width* bitInfoHead.biBitCount);
//计算位图的实际宽度并确保它为 32 的倍数(4 字节倍数)
78.
79.     if(l_width>960)
80.     {
81.         USART1_printf(USART1, "\nSORRY, PIC IS TOO BIG (<=320)\n");
82.         return;
83.     }
84.
85.     if(bitInfoHead.biBitCount>=24)
86.     {
87. }
```



```
88.         bmp(x,y,width, height);           //LCD 参数相关设置
89.
90.         for(i=0;i<height; i++)
91.         {
92.
93.             for(j=0; j<l_width; j++)          //将一行数据全部读入
94.             {
95.
96.                 f_read(&bmpfsr, pColorData+j, 1, &read_num);
97.             }
98.
99.
100.            for(j=0; j<width; j++)          //一行有效信息
101.            {
102.                k = j*3;                  //一行中第 K 个像素的起
点
103.                red = pColorData[k+2];
104.                green = pColorData[k+1];
105.                blue = pColorData[k];
106.                MY_LCD_WR_Data(RGB24TORGB16(red,green,blue));
107.            }                                //写入 LCD-GRAM
108.
109.
110.        }
111.
112.        bmp3();                         //lcd 扫描方向复原
113.
114.    }
115.
116.    else
117.    {
118.
119.        USART1_printf(USART1, "SORRY, THIS PIC IS NOT A 24BITS R
EAL COLOR");
120.        return ;
121.    }
122.
123.    f_close(&bmpfsr);
124.
125. }
```

该函数的主要工作流程是：读取头部信息确定宽度和高度并确定每一行后面具体需要读出的字节数(保证是 4 字节的倍数)----->读取一行并显示-->读取下一行并显示-> .....直至读完所有行。

另外一点就是：RGB24TORGB16 是个宏定义，因为图像数据是 RGB888 即 24 为真彩色，而我们的 LCD 是 RGB565 即 16 位色度的，所以我们需要按比例将 24 为真彩色压缩为 16 位。宏定义如下：

```
1. #define RGB24TORGB16(R,G,B) ((unsigned short int) (((R)>>3)<<11) | (((G)
>>2)<<5) | ((B)>>3))
```



---

完成 LCD 截图功能。

我们可以根据用户的长和高来构造我们前面的信息头，并且根据长与四字节对齐的关系来补充需要的 0 大小字节。在 main 函数中，我们调用了 Screen\_shot 这个函数来截图。保存的图片是 24 位的真彩色。Screen\_shot 的源码如下：

```
1.  ****
2.  * 函数名: Screen_shot
3.  * 描述  : 截取 LCD 指定位置 指定宽高的像素 保存为 24 位真彩色 bmp 格式图片
4.  * 输入  :      x          ---水平位置
5.  *           y          ---竖直位置
6.  *           Width       ---水平宽度
7.  *           Height      ---竖直高度
8.  *           filename    ---文件名
9.  * 输出  :      0          ---成功
10. *           -1         ---失败
11. *           8          ---文件已存在
12. * 举例  : Screen_shot(0, 0, 320, 240, "/myScreen");全屏截图
13. * 注意  : x 范围[0,319]  y 范围[0,239]  Width 范围[0,320-x]  Height 范围
14. *           [0,240-y]
15. *           如果文件已存在,将直接返回
16. ****
17. int Screen_shot(unsigned short int x, unsigned short int y, unsigned short int Width, unsigned short int Height, unsigned char *filename)
18. {
19.     //直接构造前面 54 字节的头部信息
20.     unsigned char header[54] =
21.     {
22.         0x42, 0x4d, 0, 0, 0, 0,
23.         0, 0, 0, 0, 54, 0,
24.         0, 0, 40, 0, 0, 0,
25.         0, 0, 0, 0, 0, 0,
26.         0, 0, 1, 0, 24, 0,
27.         0, 0, 0, 0, 0, 0,
28.         0, 0, 0, 0, 0, 0,
29.         0, 0, 0
30.     };
31.     int i;
32.     int j;
33.     long file_size;
34.     long width;
35.     long height;
36.     unsigned short int tmp_rgb;
37.     unsigned char r,g,b;
38.     unsigned char tmp_name[30];
39.     unsigned int mybw;
40.     char kk[4]={0,0,0,0};
41.
42.
43.     file_size = (long)Width * (long)Height * 3 + Height*(Width%4)
44.     + 54;      //宽*高 +补充的字节 + 头部信息
45.     header[2] = (unsigned char)(file_size & 0x000000ff);
46.     header[3] = (file_size >> 8) & 0x000000ff;
47.     header[4] = (file_size >> 16) & 0x000000ff;
48.     header[5] = (file_size >> 24) & 0x000000ff;
49.
50.
51.     width=Width;
```



```
52.     header[18] = width & 0x000000ff;
53.     header[19] = (width >> 8) &0x000000ff;
54.     header[20] = (width >> 16) &0x000000ff;
55.     header[21] = (width >> 24) &0x000000ff;
56.
57.     height = Height;
58.     header[22] = height &0x000000ff;
59.     header[23] = (height >> 8) &0x000000ff;
60.     header[24] = (height >> 16) &0x000000ff;
61.     header[25] = (height >> 24) &0x000000ff;
62.
63.     sprintf((char*)tmp_name,"0:%s.bmp",filename);
64.     f_mount(0, &bmpfs[0]);
65.     bmpres = f_open(&bmpfsrc, (char*)tmp_name, FA_CREATE_NEW | FA_WRI-
   TE);
66.
67.     if ( bmpres == FR_OK )
68.     {
69.         bmpres = f_write(&bmpfsrc, header,sizeof(unsigned char)*54, &m-
   ybw);
70.         for(i=0;i<Height;i++) //高
71.         {
72.             if (!(Width%4))
73.             {
74.                 for(j=0;j<Width;j++) //宽
75.                 {
76.
77.                     tmp_rgb = bmp4(Height-i+x,j+y); //获取该位置 LCD 的像
   素值
78.
79.                     r = GETR_FROM_RGB16(tmp_rgb);
80.                     g = GETG_FROM_RGB16(tmp_rgb);
81.                     b = GETB_FROM_RGB16(tmp_rgb);
82.
83.                     bmpres = f_write(&bmpfsrc, &b,sizeof(unsigned char),
   &mybw);
84.                     bmpres = f_write(&bmpfsrc, &g,sizeof(unsigned char),
   &mybw);
85.                     bmpres = f_write(&bmpfsrc, &r,sizeof(unsigned char),
   &mybw);
86.
87.                 }
88.
89.             }
90.             else
91.             {
92.                 for(j=0;j<Width;j++)
93.                 {
94.
95.                     tmp_rgb = bmp4(Height-i+x,j+y); //获取该位置 LCD 的像
   素值
96.
97.                     r = GETR_FROM_RGB16(tmp_rgb);
98.                     g = GETG_FROM_RGB16(tmp_rgb);
99.                     b = GETB_FROM_RGB16(tmp_rgb);
100.
101.                    bmpres = f_write(&bmpfsrc, &b,sizeof(unsign
   ed char), &mybw);
102.                    bmpres = f_write(&bmpfsrc, &g,sizeof(unsign
   ed char), &mybw);
103.                    bmpres = f_write(&bmpfsrc, &r,sizeof(unsign
   ed char), &mybw);
104.
105.
106.                }
107.
```



```
108.                 bmpres = f_write(&bmpfsrc, kk, sizeof(unsigned c
  har) * (Width%4), &mybw);
109.
110.
111.             }
112.         }
113.
114.         f_close(&bmpfsrc);
115.         return 0;
116.     }
117.     else if ( bmpres == FR_EXIST ) //如果文件已经存在
118.     {
119.         return FR_EXIST;           //8
120.     }
121.
122.     else
123.     {
124.         return -1;
125.     }
126.
127. }
```

该函数中，调用了 `bmp4` 这个函数，该函数返回 LCD 上指定位置的像素信息。

`GETG_FROM_RGB16`、`GETB_FROM_RGB16` 和 `GETR_FROM_RGB16` 都是宏定义，将 `RGB565` 即 16 位色度抽取出其中的 `RGB` 数据并分别将其线性映射为 8 位数据即映射为 `RGB888` 真彩色。宏定义的内容如下：

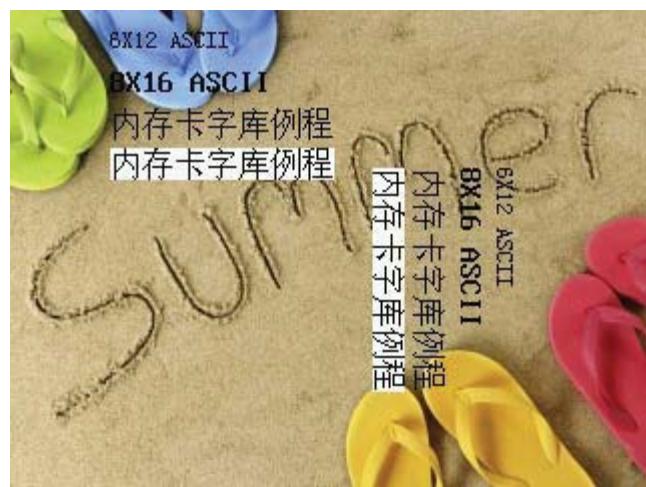
```
1. #define GETR_FROM_RGB16(RGB565) (((unsigned char)((((unsigned short i
  nt )RGB565) >>11)<<3)))           //返回 8 位 R
2. #define GETG_FROM_RGB16(RGB565) (((unsigned char)((((unsigned short i
  nt )(RGB565 & 0x7ff) >>5)<<2)))   //返回 8 位 G
3. #define GETB_FROM_RGB16(RGB565) (((unsigned char)((((unsigned short i
  nt )(RGB565 & 0x1f))<<3))))        //返回 8 位 B
```

## 实验现象如下：

放入 sd 卡的背景图：“`test.bmp`”如下：



程序运行之后截图保存为“MYSCREEN.bmp”如下:



( ^ ^ 截图保存图片功能+摄像头模块 就构成了一个完整的照相机啦！！)

实验讲解完毕，野火祝大家学习愉快^\_\_^.....



## MP3+MicroSD+FATFS (SPI2) 实验

作者	fire
E-Mail	firestm32@foxmail.com
QQ	313303034
博客	firestm32.blog.chinaunix.net
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

**实验描述:** 将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码, 然后将解码后的数据送到功放 TDA1308 后通过耳机播放出来。

硬件连接: PB13-SPI2\_SCK : VS1003B-SCLK

PB14-SPI2\_MISO : VS1003B-SO

PB15-SPI2\_MOSI : VS1003B-SI

PB12-SPI2\_NSS : VS1003B-XCS

PB11 : VS1003B-XRET

PC6 : VS1003B-XDCS

PC7 : VS1003B-DREQ

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\_sdio.c

FWlib/stm32f10x\_dma.c



FWlib/stm32f10x\_spi.c

FWlib/misc.c

用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/sdcard.c

USER/diakio.c

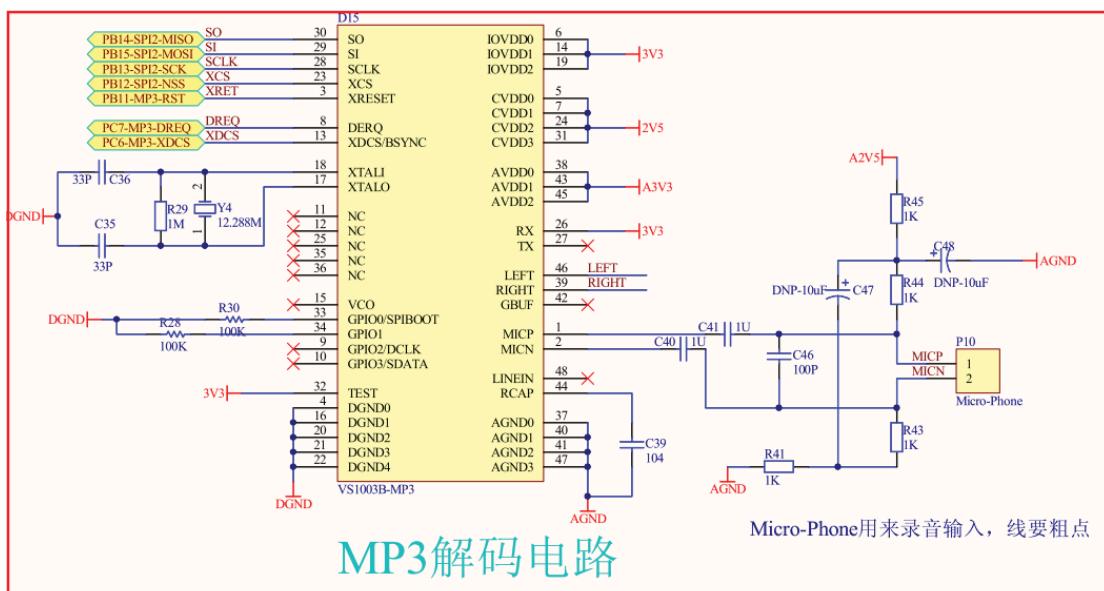
USER/ff.c

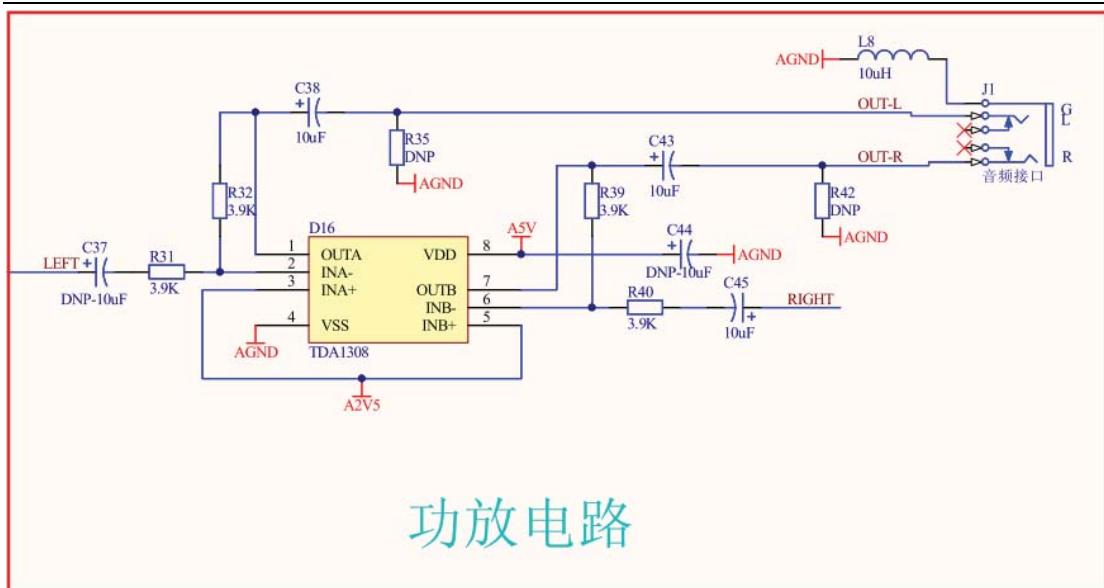
USER/usart1.c

USER/vs1003.c

USER/SysTick.c

### 野火 STM32 开发板中 MP3 的硬件原理图:





## 功放电路

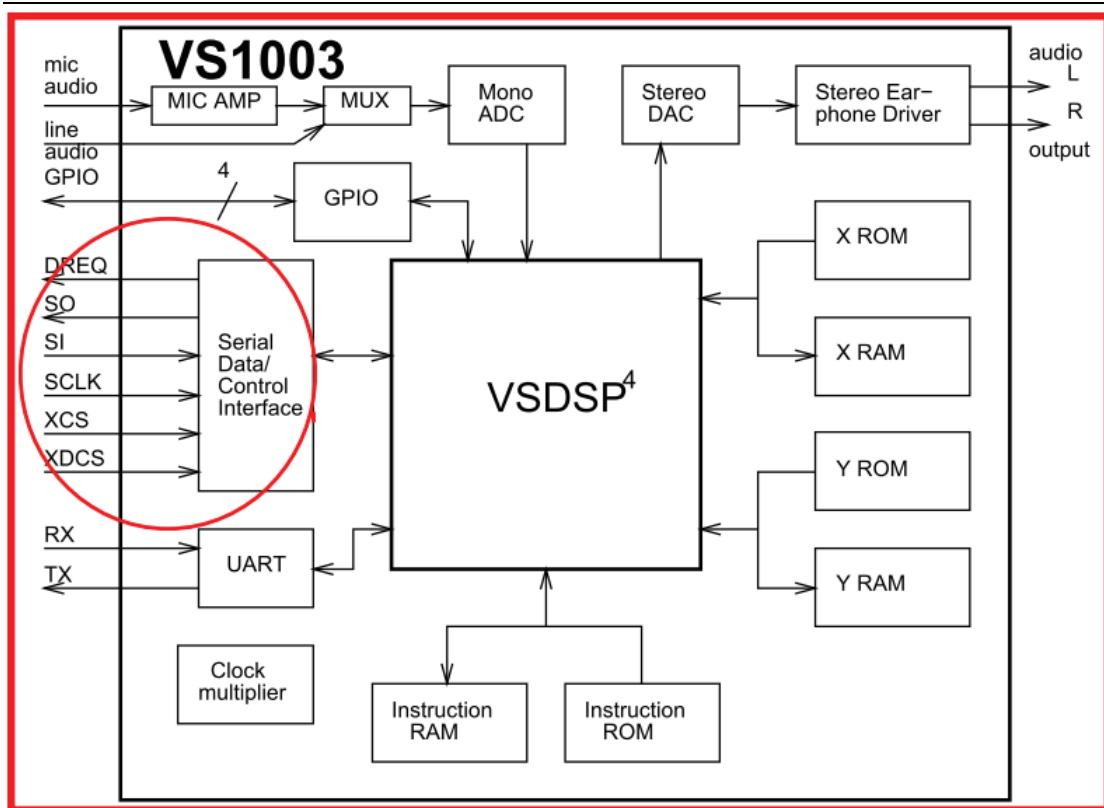
解码部分采用 **VS1003-MP3/WMA** 音频解码器，然后将解码后的数据送 **TDA1308** 放大后由音频接口外播出来。

### **VS1003+TDA1308 简介->**

**VS1003** 是一个单片 MP3/WMA/MIDI 音频解码器和 ADPCM 编码器。它包含一个高性能，自主产权的低功耗 DSP 处理器核 **VS\_DSP 4**，工作数据存储器，为用户应用提供 5KB 的指令 RAM 和 0.5KB 的数据 RAM。串行的控制和数据接口，4 个常规用途的 I/O 口，一个 UART，也有一个高品质可变采样率的 ADC 和立体声 DAC，还有一个耳机放大器和地线缓冲器。

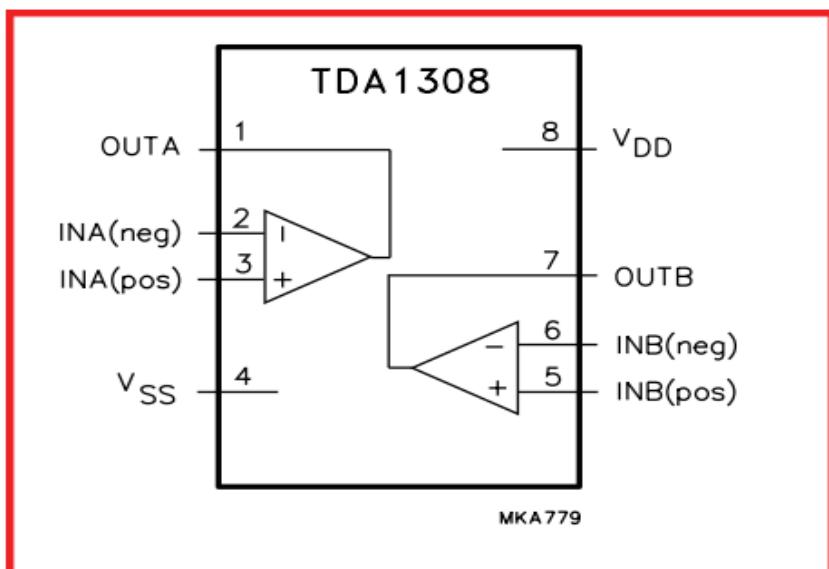
**VS1003** 通过一个串行接口来接收输入的比特流，它可以作为一个系统的从机。输入的比特流被解码，然后通过一个数字音量控制器到达一个 18 位过采样多位 ε - Δ DAC。通过串行总线控制解码器。除了基本的解码，在用户 RAM 中它还可以做其他特殊应用，例如 DSP 音效处理。

**VS1003** 原理框图：



本实验中我们只用了红色圆圈中的那几个数据口，这些数据口是串行模式的，我们用到了开发板中的 SPI2 来控制。其中数据经 SI 接口进去，经解码后由 L、R 这两个左右声道引脚出来，因为 VS1003 内部集成了一个 DA，所以出来的数据是模拟的，可直接驱动耳机，但由于功率太小，音效不佳，所以我们将信号送往 TDA1308 放大后再通过耳机外放出来，**经过这样出来之后音质跟电脑上的有的比**。现在市面上的 MP3 模块基于成本考虑都没加音频功放，而是直接驱动音频耳机，效果可想而知。

TDA1308 是一款双通道的立体耳机驱动器，是一款专门用于声音驱动的功放。其



原理框图如右：



---

有关 VS1003B 和 TDA1308 的详细应用，大家可参考官方的 datasheet，野火就不在这里罗嗦啦^\_^。

本实验是在《MicroSD 卡+FATFS》这个实验基础上进行的。没做过这个实验的话可参考前面的教程，否则有些代码会让您犯糊涂。

### 实验讲解->

首先需要将需要用到的库文件添加进来，有关库的配置可参考前面的教程，这里不再详述。在配置好库的环境之后我们从 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 SysTick 为 10us 中断一次 */
13.    SysTick_Init();
14.
15.    /* 配置串口 1 115200 8-N-1 */
16.    USART1_Config();
17.
18.    /* SD 卡中断配置 */
19.    NVIC_Configuration();
20.
21.    USART1_printf( USART1, " \r\n 这是一个 MP3 测试例程 !\r\n " );
22.
23.    /* MP3 硬件初始化 */
24.    VS1003_SPI_Init();
25.
26.    /* SD 卡硬件初始化，并初始化盘符为 0 */
27.    disk_initialize( 0 );
28.
29.    /* MP3 就绪，准备播放 */
30.    MP3_Start();
31.
32.    /* 播放 SD 卡(FATFS)里面的音频文件 */
33.    MP3_Play();
34.
35.    while (1)
36.    {
37.    }
38.
39. } /* end of main */
```

库函数 `SystemInit();` 将我们的系统时钟设置为 **72MHZ**，在所有工作之前首先要做就是先设置系统时钟，这可千万别忘了。在 **ST3.0.0** 版本之后的库中，这部分工作都放在了启动文件中了，由汇编实现，只要用户代码一进入 `main` 函数就表示已经初始化好系统时钟了，完全不用用户考虑，用户不知道这点的话还以为不需要初始化系统



时钟呢。但我们这里用的库版本是 **ST3.0.0**, 所以还需要调用库函数 `SystemInit()`; 来初始化我们的系统时钟。至于 **ST3.0.0** 和之后高版本的库有什么区别, 我想说的是没什么大的区别, 代码的目录结构基本没有改变, 只是在代码的功能增多了, 支持更完善的外设。

`SysTick` 为 10us 中断一次 用于 `SysTick` 为 10us 中断一次, 用于后面的延时函数。

`USART1_Config()`; 配置串口 1 波特率为 115200 , 8 个数据位, 1 个停止位, 无硬件流控制。

`NVIC_Configuration()`; 用于配置 MicroSD 卡的中断优先级。

`VS1003_SPI_Init()`; 用于初始化 MP3 解码芯片 VS1003B 需要用到的 I/O 口, 包括数据口(SPI2)和控制 I/O。`vs1003_SPI_Init()`; 由用户在 `vs1003.c` 中实现:

```
1.  /*
2.   * 函数名: VS1003 SPI Init
3.   * 描述 : VS1003 所用 I/O 初始化
4.   * 输入 : 无
5.   * 输出 : 无
6.   * 调用 : 外部调用
7.   */
8. void VS1003_SPI_Init(void)
9. {
10.    SPI_InitTypeDef SPI_InitStructure;
11.    GPIO_InitTypeDef GPIO_InitStructure;
12.
13.    /* 使能 VS1003B 所用 I/O 的时钟 */
14.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC, ENABLE);
15.    /* 使能 SPI2 时钟 */
16.    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
17.
18.    /* 配置 SPI2 引脚: PB13-SCK, PB14-MISO 和 PB15-MOSI */
19.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
20.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
22.    GPIO_Init(GPIOB, &GPIO_InitStructure);
23.
24.    /* PB12-XCS(片选) */
25.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
26.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
27.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
28.    GPIO_Init(GPIOB, &GPIO_InitStructure);
29.
30.    /* PB11-XRST(复位) */
31.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
32.    GPIO_Init(GPIOB, &GPIO_InitStructure);
33.
34.
35.    /* PC6-XDCS(数据命令选择) */
36.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
37.    GPIO_Init(GPIOC, &GPIO_InitStructure);
```



```
38.
39.     /* PC7-DREQ(数据中断) */
40.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
41.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
42.     GPIO_Init(GPIOC, &GPIO_InitStructure);
43.
44.     /* SPI2 configuration */
45.     SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
46.     SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
47.     SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
48.     SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
49.     SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
50.     SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
51.     SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_32;
52.     SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
53.     SPI_InitStructure.SPI_CRCPolynomial = 7;
54.     SPI_Init(SPI2, &SPI_InitStructure);
55.
56.     /* Enable SPI2 */
57.     SPI_Cmd(SPI2, ENABLE);
58. }
```

假如我们要将数据口换成 SPI1 或者改变其他控制 I/O，只需改变这个函数即可，移植性非常强。

`disk_initialize( 0 );` 用于初始化 MicroSD 卡的底层硬件，并将卡的盘符初始化为 0，盘符的范围为 0~9，用户可选，在这里我们选为 0。

`MP3_Start();` 使 MP3 进入就绪模式(standby)，随时播放音乐。`MP3_Start();` 在 `vs1003.c` 中实现：

```
1.  * 函数名: MP3_Start
2.  * 描述  : 使 MP3 进入就绪模式，随时准备播放音乐。
3.  * 输入  : 无
4.  * 输出  : 无
5.  * 调用  : 外部调用
6.  */
7. void MP3_Start(void)
8. {
9.     u8 BassEnhanceValue = 0x00;          // 低音值先初始化为 0
10.    u8 TrebleEnhanceValue = 0x00;         // 高音值先初始化为 0
11.    TRST_SET(0);
12.    Delay_us( 1000 );                  // 1000*10us = 10ms
13.
14.    VS1003_WriteByte(0xff);             // 发送一个字节的无效数据，启动 SPI 传输
15.    TXDCS_SET(1);
16.    TCS_SET(1);
17.    TRST_SET(1);
18.    Delay_us( 1000 );
19.
20.    Mp3WriteRegister( SPI_MODE, 0x08, 0x00 );      // 进入 vs1003 的播放模式
21.    Mp3WriteRegister(3, 0x98, 0x00);            // 设置 vs1003 的时钟,3 倍频
22.    Mp3WriteRegister(5, 0xBB, 0x81);            // 采样率 48k, 立体声
23.    // 设置重低音
24.    Mp3WriteRegister(SPI_BASS, TrebleEnhanceValue, BassEnhanceValue);
25.    Mp3WriteRegister(0x0b, 0x00, 0x00);        // VS1003 音量
26.    Delay_us( 1000 );
27.
28.    while( DREQ == 0 );                      // 等待 DREQ 为高 表示能够接受音乐数据输入
29. }
```



函数中涉及到的宏定义都在 `vs1003.h` 这个头文件中实现。关于函数中为什么要这样操作寄存器，或者为什么要按照这个顺序来操作寄存器，请大家查阅 `vs1003` 的 pdf，里面讲得很详细，有英文跟中文资料。

`MP3_Play()`：这个函数逐个扫描我们卡里面的音频文件，并将这些音频文件都通过耳机播放出来，最后停止，暂时还没实现循环播放的功能。`MP3_Play()` 在 `vs1003.c` 中实现：

```
1.  /*
2.   * 函数名: MP3_Play
3.   * 描述 : 读取 SD 卡里面的音频文件，并通过耳机播放出来
4.   *          支持的格式: mp3,mid,wav,wma
5.   * 输入 : 无
6.   * 输出 : 无
7.   * 说明 : 暂不支持长文件名，不支持中文。
8.   */
9. void MP3_Play(void)
10. {
11.     FRESULT res;
12.     FILINFO finfo;
13.     DIR dirs;
14.     u16 count = 0;
15.     char j = 0;
16.     char path[50] = {"\0"};
17.     char *result1, *result2, *result3, *result4;
18.
19.     f_mount(0, &fs);                                /* 挂载文件系统到 0 区 */
20.
21.     if (f_opendir(&dirs, path) == FR_OK)           /* 打开根目录 */
22.     {
23.         while (f_readdir(&dirs, &finfo) == FR_OK)    /* 依次读取文件名 */
24.         {
25.             if (finfo.fattrib & AM_ARC)               /* 判断是否为存档型文档 */
26.             {
27.                 if (!finfo.fname[0])                  /* 文件名为空即到达了目录的末尾，退出 */
28.                     break;
29.                 USART1_printf(USART1, "\r\n the music file name is: %s \r\n", finfo.fname);
30.
31.                 result1 = strstr(finfo.fname, ".mp3"); /* 判断是否为音频文件 */
32.                 result2 = strstr(finfo.fname, ".mid");
33.                 result3 = strstr(finfo.fname, ".wav");
34.                 result4 = strstr(finfo.fname, ".wma");
35.
36.                 if (result1!=NULL || result2!=NULL || result3!=NULL || result4
37.                     !=NULL)
38.                     {
39.                         res = f_open(&fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ); /* 以只读方式打开 */
40.                         TXDCS_SET(0);           /* 选择 VS1003 的数据接口 */
41.                         /* ----- 一曲开始 -----*/
42.                         USART1_printf(USART1, "\r\n 开始播放 \r\n");
43.                         for (;;) {
44.                             {
45.                                 res = f_read(&fsrc, buffer, sizeof(buffer), &br);
46.                                 if (res == 0)
47.                                     {
48.                                         count = 0;                      /* 512
   * 字节完重新计数 */

```



```
49.                     Delay_us( 1000 );           /* 10ms 延时 */
50.                     while ( count < 512 )           /* SD 卡读取一个
   sector, 一个 sector 为 512 字节 */
51.                     {
52.                         if ( DREQ != 0 )           /* 等待 DREQ 为高, 请求数据输入 */
53.                         {
54.                             for (j=0; j<32; j++) /* VS1003 的 FIFO
   只有 32 个字节的缓冲 */
55.                             {
56.                                 VS1003_WriteByte( buffer[count] );
57.                                 count++;
58.                             }
59.                         }
60.                     }
61.                 }
62.             if (res || br == 0) break; /* 出错或者到了文件尾 */
63.         }
64.         USART1_printf( USART1, "\r\n 播放结束 \r\n" );
65. /* ----- 一曲结束 -----*/
66.         count = 0;
67.         /* 根据 VS1003 的要求, 在一曲结束后需发送 2048 个 0 来确保下一首的正常播放 */
68.         while ( count < 2048 )
69.         {
70.             if ( DREQ != 0 )
71.             {
72.                 for ( j=0; j<32; j++ )
73.                 {
74.                     VS1003_WriteByte( 0 );
75.                     count++;
76.                 }
77.             }
78.         }
79.         count = 0;
80.         TXDCS_SET( 1 ); /* 关闭 VS1003 数据端口 */
81.         f_close(&fsrc); /* 关闭打开的文件 */
82.     }
83. }
84. } /* end of while */
85. }
86. } /* end of MP3_Play */
```

由于代码比较长，在格式编排上不是很好，野火建议大家还是配合源代码一起阅读^\_^。

现在我们来大概分析下 `MP3_Play()` 这个函数，这里涉及到一些文件系统操作的函数，关于这部分函数的操作大家可参考前面的教程或者阅读 `FATFS` 的官方文档，其实我的教程也不完全正确，阅读官方的文档才是最可靠的。

函数 `f_mount(0, &fs);` 为我们在文件系统中注册一个工作区，并初始化盘符的名为 0。

函数 `f_opendir(&dirs, path)` 用于打开卡的根目录，并将这个根目录关联到 `dirs` 这个结构指针，然后我们就可以通过这个结构指针来操作这个目录了，其实这个结构指



---

针就类似 **LINUX** 下系统编程中的文件描述符，不论是操作还是目录都得通过文件描述符才能操作。

`f_readdir(&dirs, &finfo)` 函数通过刚刚的 `dirs` 结构指针来读取目录里面的信息，并将目录的信息储存在 `finfo` 这个结构体变量中。

紧接着判断文件的属性，如果是存档型文件的话就将文件名打印出来，然后比较文件的后缀名，查看是否为音频文件，支持的音频格式有 `mp3`、`mid`、`wav`、`wma`。

如果是音频文件的话则调用 `f_open( &fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ );` 打开这个音频文件。

`TXDGS_SET( 0 );` 用于选择 `vs1003` 的数据端口，准备往 `vs1003` 中输入数据。其中 `TXDGS_SET( 0 );` 是在 `vs1003.h` 中实现的一个宏：

```
1. #define XDGS    (1<<6) // PC6-XDGS
2.
3. #define TXDGS_SET(x) GPIOC->ODR= (GPIOC->ODR&~XDGS) | (x ? XDGS:0)
```

紧接着进入一个大循环中播放我们的 `mp3` 文件。

函数 `f_read( &fsrc, buffer, sizeof(buffer), &br );` 从文件中读取 **512** 个字节的数据到缓冲区中，至于为什么是 **512** 个字节，而不是 **1024** 或者更多，这是因为卡的一个 `sector` 是 **512** 个字节，一次只能读取一个 `sector`。

函数 `vs1003_WriteByte( buffer[count] );` 将缓冲区中的数据写入 `vs1003` 的数据缓冲区。注意，这里一次只能写入 **32** 个字节，这是因为 `vs1003` 的 **FIFO** 的大小为 **32** 个字节，写多了无效。

当文件出错或者一曲播放完毕时就跳出 `for` 循环，并打印出“**播放结束**”的调试信息。



根据 VS1003 的要求, 在一曲结束后需发送 2048 个 0 来确保下一首的正常播放。

一曲播放完毕我们关闭 vs1003 的数据端, 关闭打开的文件, 等待下一曲的播放, 直到目录下的音频文件播放完为止。还暂不支持循环播放的功能。

这里面涉及到了 vs1003 操作的一些特性, 需大家参考 vs1003 的 datasheet 来帮助理解。

### 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), 插上 MicroSD 卡(我用的是 1G), 在卡的根目录下要有 mp3 文件, 文件名要是英文, 暂不支持中文和长文件名, 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:

```
the music file name is: yydt.mp3
开始播放
播放结束
the music file name is: ysbg.mp3
开始播放
播放结束
the music file name is: yg.mp3
开始播放
```

已连接 4:53:56 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印



我的卡的根目录下放了 3 个 mp3 文件，都是 Eason 的歌(因为个人非常喜欢陈奕迅的歌^\_^)，分别是遥远的她(yydt.mp3)、一丝不挂(ysbg.mp3)、预感(yg.mp3)。插上耳机，音质堪比电脑，音量可通过耳机来调，前提是你的耳机要能调节音量才行。

实验讲解完毕，野火祝大家学习愉快^\_^。



## 基于 STM32(ARMV7)的 MP3 播放器

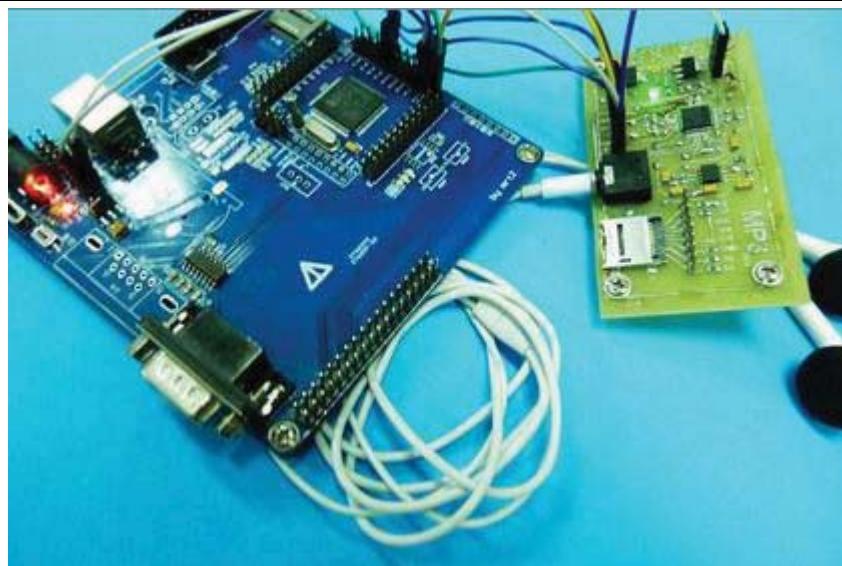
作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

**实验描述:** 将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码，然后将解码后的数据送到功放 TDA1308 后通过耳机播放出来，并在 LCD 中显示曲目和艺术家，支持中英文显示。这部分需要 LCD 的支持，有关 LCD 的操作请查看 LCD 的那部分 pdf 教程。

注：野火 stm32 开发板中板载的 mp3 跟这里的接口是一样的。

纯手工打造 MP3，VS1003 硬解码，外加立体耳机功放 TDA1308，音质堪比电脑。





### MP3 与开发板的接线

硬件连接: PB13-SPI2\_SCK : VS1003B-SCLK

PB14-SPI2\_MISO : VS1003B-SO

PB15-SPI2\_MOSI : VS1003B-SI

PB12-SPI2\_NSS : VS1003B-XCS

PB11 : VS1003B-XRET

PC6 : VS1003B-XDCS

PC7 : VS1003B-DREQ

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\_sdio.c

FWlib/stm32f10x\_dma.c

FWlib/stm32f10x\_spi.c

FWlib/misc.c

FWlib/fsmc.c



用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/sdcard.c

USER/diakio.c

USER/ff.c

USER/usart1.c

USER/vs1003.c

USER/SysTick.c

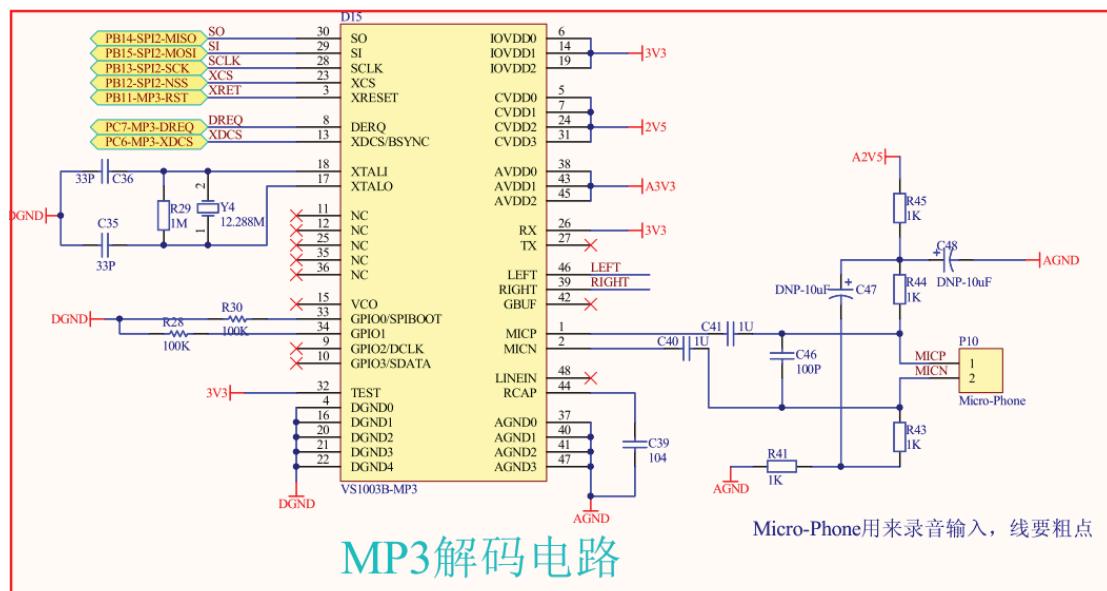
USER/lcd.c

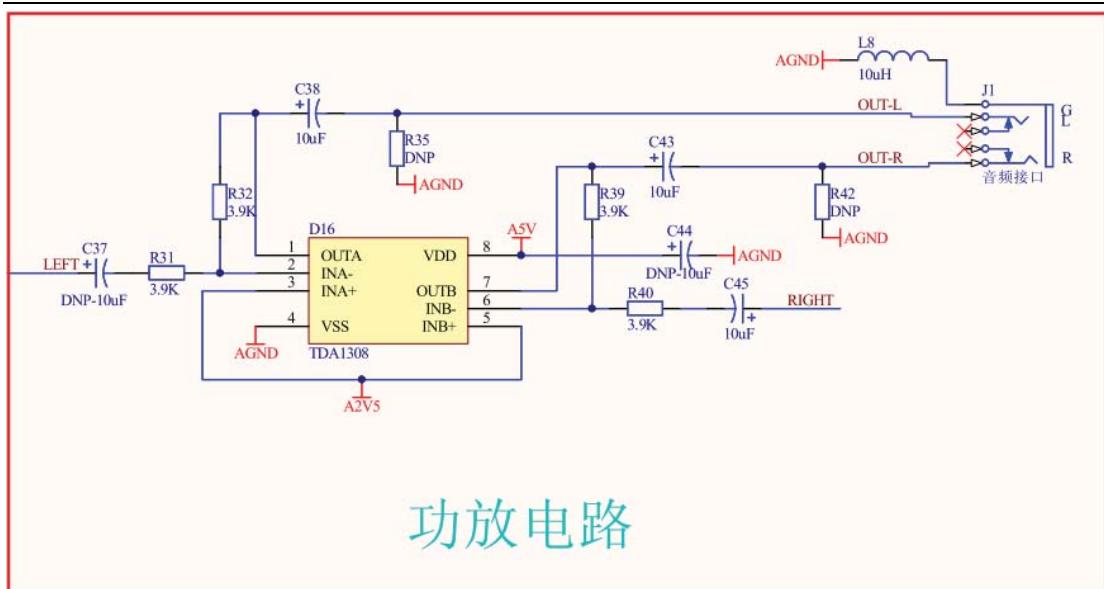
USER/sd\_fs\_app.c

USER/sd\_bmp.c

USER/mp3play.c

### 野火 STM32 开发板中 MP3 的硬件原理图:





## 功放电路

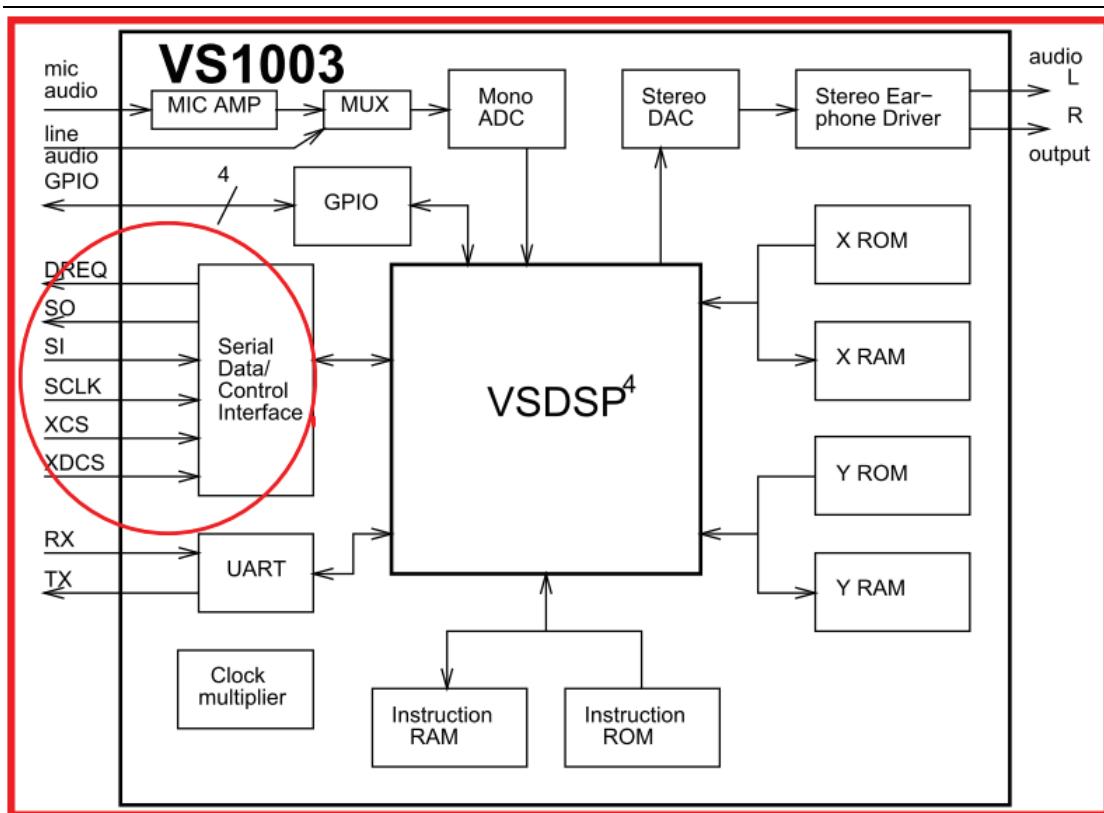
解码部分采用 VS1003-MP3/WMA 音频解码器，然后将解码后的数据送 TDA1308 放大后由音频接口外播出来。

### VS1003+TDA1308 简介->

VS1003 是一个单片 MP3/WMA/MIDI 音频解码器和 ADPCM 编码器。它包含一个高性能，自主产权的低功耗 DSP 处理器核 VS\_DSP 4，工作数据存储器，为用户应用提供 5KB 的指令 RAM 和 0.5KB 的数据 RAM。串行的控制和数据接口，4 个常规用途的 I/O 口，一个 UART，也有一个高品质可变采样率的 ADC 和立体声 DAC，还有一个耳机放大器和地线缓冲器。

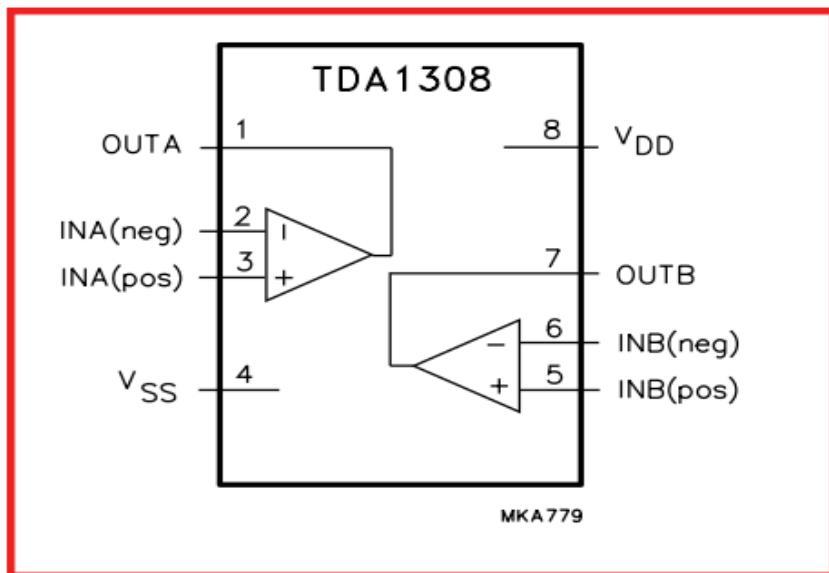
VS1003 通过一个串行接口来接收输入的比特流，它可以作为一个系统的从机。输入的比特流被解码，然后通过一个数字音量控制器到达一个 18 位过采样多位 ε - Δ DAC。通过串行总线控制解码器。除了基本的解码，在用户 RAM 中它还可以做其他特殊应用，例如 DSP 音效处理。

VS1003 原理框图：



本实验中我们只用了红色圆圈中的那几个数据口，这些数据口是串行模式的，我们用到了开发板中的 SPI2 来控制。其中数据经 SI 接口进去，经解码后由 L、R 这两个左右声道引脚出来，因为 VS1003 内部集成了一个 DA，所以出来的数据是模拟的，可直接驱动耳机，但由于功率太小，音效不佳，所以我们将信号送往 TDA1308 放大后再通过耳机外放出来，经过这样出来之后音质跟电脑上的有的比。现在市面上的 MP3 模块基于成本考虑都没加音频功放，而是直接驱动音频耳机，效果可想而知。

TDA1308 是一款双通道的立体耳机驱动器，是一款专门用于声音驱动的功放。其原理框图如下：



有关 VS1003B 和 TDA1308 的详细应用，大家可参考官方的 datasheet。

本实验是在《MicroSD 卡+FATFS》这个实验基础上进行的。没做过这个实验的话可参考前面的教程。

#### -----MP3 知识扫盲-----

在讲解实验之前，先大概了解下 MP3 的知识。

- MP3 文件概述->

MP3 文件是由帧(frame)构成的，帧是 MP3 文件最小的组成单位。MP3 的全称为 MPEG1Layer-3 音频文件，MPEG(Moving Pictures Experts Group)在汉语中翻译为活动图像专家组，特指活动影音压缩标准，MPEG 音频文件是 MPEG1 标准中的声音部分，也叫 MPEG 音频层，它根据压缩质量和编码复杂程度划分为三层，即 Layer1、Layer2、Layer3，且分别对应 MP1、MP2、MP3 这三种音频文件，并根据不同的用途，使用不同层次的编码。MPEG 音频编码的层次越高，编码器越复杂，压缩率也越高，MP1 和 MP2 的压缩率分别为 4: 1 和 6: 1-8: 1，而 MP3 的压缩率则高达 10: 1-12: 1。不过 MP3 对音频信号采用的是有损压缩方式，未来降低声音失真度，MP3 采用了“感官编码技术”，即编码时先对音频文件进行频谱分析，然后用过滤器滤掉噪音电平，接着通过量化的方式将剩下的每一位打散排列，也就是 Huffman 无损压缩编



---

码，最后形成具有较高压缩比的 MP3 文件，并使压缩后的文件在回放时能够达到比较接近原音的声效。

### ● MP3 文件组成->

MP3 文件大体分为 3 部分:TAG\_V2(ID3V2),Frame,TAG\_V1(ID3V1)。要注意有些 MP3 文件尾是没有 ID3V1 tag 的，有的是 MP3 文件头部的 ID3V2 tag。其中 ID3V1 存放在 MP3 文件的末尾，用 16 进制的编辑器打开一个 MP3 文件，查看其末尾的 128 个顺序存放字节，数据结构定义如下：

```
1.  typedef struct tagID3V1
2.  {
3.      char Header[3];           // 标签头必须是"TAG"，否则被认为没有标签头
4.      char Title[30];          // 标题
5.      char Artist[30];         // 作者
6.      char Album[30];          // 专辑
7.      char Year[4];           // 出品年代
8.      char Comment[28];        // 备注
9.      char reserve;           // 保留
10.     char track;             // 音轨
11.     char Genre;              // 风格
12. }ID3V1, *pID3V1;
```

现在很多 MP3 文件有的只是 ID3V2，基本上都没有了尾部的 ID3V1 了，有 ID3V1 的是比较老的歌曲，野火发现像 beyond 的歌都含有 ID3V1 tag，但信息也不如上面的结构体中描述得多，一般有的是 title 和 artist。这在接下来的代码中可看到 ID3V1 的应用。

ID3V1 的各项都是顺序存放，没有任何标志将其分开，比如标题信息不足 30 个字节，则使用'\0'补足，否则将造成信息错误。

1、 每帧的播放时间：无论帧长是多少，每帧的播放时间是 26ms。

2、 数据帧大小的计算：

```
3、 *1、每帧的播放时间：无论帧长是多少，每帧的播放时间是 26ms。
4、 *2、
    FrameSize = ( (( MPEGVersion == MPEG1 ? 144 : 72 ) * Bitrate) / SampleRate
                  ) + PaddingBit
5、 */
```



◆ **ID3V2:** 包含了作者, 作曲, 专辑等信息, 长度不固定, 扩展了 ID3V1 的信息量。

◆ **Frame:** 一系列的帧, 个数由文件大小和帧长决定。每个 **frame** 的帧长可能固定, 也可能不固定, 由位率 **bitrate** 决定。每个帧又分为帧头和数据实体两部分。帧头记录了 mp3 的位率, 采样率, 版本等信息, 每个帧之间相互独立。

◆ **ID3V1:** 包含了作者, 作曲。专辑等信息, 长度为 **128 byte**。

每个 MP3 数据帧有一个帧头 **FRAMHEADER**, 长度是 4 个 byte(32bit), 帧头后面可能有两个字节的 **CRC** 校验码, 这两个字节的是否存在决定于 **FRAMHEADER** 的第 16bit, 为 0 则帧头后面没校验, 为 1 则有校验, 校验值长度为 2 个字节, 紧跟在 **FRAMHEADER** 后面, 接着就是帧的实体数据了, 格式如下:

FRAMHEADER	CRC(free)	MAIN_DATA
4byte	0 or 2 byte	长度由帧头计算得出

关于 MP3 的知识野火先讲解到这, 其他的稍后再讲解, 下面我们来具体看看源码的实现。

### 实验讲解->

首先需要将需要用到的库文件添加进来, 有关库的配置可参考前面的教程, 这里不再详述。在配置好库的环境之后我们从 **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 SysTick 为 10us 中断一次 */
13.    SysTick_Init();
```



```
14.
15. /* 配置串口 1 115200 8-N-1 */
16. USART1_Config();
17.
18. /* LED 初始化, 用于调试 */
19. LED_GPIO_Config();
20.
21. /* 2.4TFT 初始化 */
22. LCD_Init();
23.
24. /* 文件系统初始化----汉字字库保存在 sd 卡中 */
25. sd_fs_init();
26.

27. Set_direction(0);           /* 设置 LCD 的扫描方向, 这里为垂直扫描*/
28.
29. LCD_CLEAR(0,0,240,320);
30. //Lcd_show_bmp(0, 0,"/PIC.bmp");
31.
32. USART1_printf( USART1, " \r\n 这是一个 MP3 测试例程 !\r\n " );
33.
34. /* MP3 硬件 I/O 初始化 */
35. VS1003_SPI_Init();
36.
37. /* SD 卡硬件初始化, 并初始化盘符为 0 */
38. //disk_initialize( 0 );
39.
40. /* MP3 就绪, 准备播放, 在 vs1003.c 实现 */
41. MP3_Start();
42.
43. LED1(ON);
44.
45. /* 播放 SD 卡 (FATFS) 里面的音频文件 */
46. MP3_Play();
47.
48. while (1)
49. {
50. }
51. } /* end of main */
```

库函数 `SystemInit()`; 将我们的系统时钟设置为 **72MHZ**, 在所有工作之前首先要做的就是先设置系统时钟, 这可千万别忘了。在 **ST3.0.0** 版本之后的库中, 这部分工作都放在了启动文件中了, 由汇编实现, 只要用户代码一进入 `main` 函数就表示已经初始化好系统时钟了, 完全不用用户考虑, 用户不知道这点的话还以为不需要初始化系统时钟呢。但我们这里用的库版本是 **ST3.0.0**, 所以还是需要调用库函数 `SystemInit()`; 来初始化我们的系统时钟。至于 **ST3.0.0** 和之后高版本的库有什么区别, 我想说的是没什么大的区别, 代码的目录结构基本没有改变, 只是在代码的功能增多了, 支持更完善的外设。

`SysTick` 为 10us 中断一次用于 `SysTick` 为 10us 中断一次, 用于后面的延时函数。

`USART1_Config()`; 配置串口 1 波特率为 **115200** , 8 个数据位, 1 个停止位, 无硬件流控制。

`LCD_Init()`; 用于初始化我们的 **2.4TFT**, 我们将用它来显示我们的 **MP3** 歌曲的信息。



`sd_fs_init()`; 用于初始化 MicroSD 卡底层硬件，并将盘符初始化为 0，还设置了 MicroSD 卡的中断优先级，这个函数里面调用了 `SDIO_NVIC_Configuration()` 和 `disk_initialize(0)` 这两个函数。调用 `sd_fs_init()` 函数是非常重要的，不然 MicroSD 卡是工作不了的，文件系统根本跑不起来，MP3 文件根本就读不出来。

`disk_initialize(0)`; 用于初始化 MicroSD 卡的底层硬件，并将卡的盘符初始化为 0，盘符的范围为 0~9，用户可选，在这里我们选为 0。

接下来的几个函数大伙看注释吧，比较简单，野火就不再啰嗦浪费大家的时间啦 ^\_^。

`VS1003_SPI_Init()`; 用于初始化 MP3 解码芯片 VS1003B 需要用到的 I/O 口，包括数据口(SPI2)和控制 I/O。`VS1003_SPI_Init()`由用户在 `VS1003.c` 中实现：

```
1.  * 函数名: VS1003_SPI_Init
2.  * 描述 : VS1003 所用 I/O 初始化
3.  * 输入 : 无
4.  * 输出 : 无
5.  * 调用 : 外部调用
6.  */
7. void VS1003_SPI_Init(void)
8. {
9.     SPI_InitTypeDef SPI_InitStructure;
10.    GPIO_InitTypeDef GPIO_InitStructure;
11.
12.    /* 使能 VS1003B 所用 I/O 的时钟 */
13.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB
2Periph_GPIOC, ENABLE);
14.    /* 使能 SPI2 时钟 */
15.    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
16.
17.    /* 配置 SPI2 引脚: PB13-SCK, PB14-MISO 和 PB15-MOSI */
18.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
19.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
20.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
21.    GPIO_Init(GPIOB, &GPIO_InitStructure);
22.
23.    /* PB12-XCS(片选) */
24.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
25.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
26.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
27.    GPIO_Init(GPIOB, &GPIO_InitStructure);
28.
29.    /* PB11-XRST(复位) */
30.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
31.    GPIO_Init(GPIOB, &GPIO_InitStructure);
32.
33.
34.    /* PC6-XDCS(数据命令选择) */
35.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
36.    GPIO_Init(GPIOC, &GPIO_InitStructure);
37.
38.    /* PC7-DREQ(数据中断) */
39.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
40.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
41.    GPIO_Init(GPIOC, &GPIO_InitStructure);
42.
43.    /* SPI2 configuration */
44.    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
45.    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
```



```
46.     SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
47.     SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
48.     SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
49.     SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
50.     SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_32;
51.     SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
52.     SPI_InitStructure.SPI_CRCPolynomial = 7;
53.     SPI_Init(SPI2, &SPI_InitStructure);
54.
55.     /* Enable SPI2 */
56.     SPI_Cmd(SPI2, ENABLE);
57. }
```

假如我们要将数据口换成 SPI1 或者改变其他控制 I/O，只需改变这个函数即可，移植性非常强。

`disk_initialize( 0 );` 用于初始化 MicroSD 卡的底层硬件，并将卡的盘符初始化为 0，盘符的范围为 0~9，用户可选，在这里我们选为 0。

`MP3_Start();` 使 MP3 进入就绪模式(standby)，随时播放音乐。`MP3_Start();` 在 `vs1003.c` 中实现：

```
1.  /*
2.   * 函数名: MP3_Start
3.   * 描述 : 使 MP3 进入就绪模式, 随时准备播放音乐。
4.   * 输入 : 无
5.   * 输出 : 无
6.   * 调用 : 外部调用
7.   */
8. void MP3_Start(void)
9. {
10.    u8 BassEnhanceValue = 0x00;           // 低音值先初始化为 0
11.    u8 TrebleEnhanceValue = 0x00;          // 高音值先初始化为 0
12.    TRST_SET(0);
13.    Delay_us( 1000 );                   // 1000*10us = 10ms
14.
15.    VS1003_WriteByte(0xff);             // 发送一个字节的无效数据, 启动 SPI 传输
16.    TXDCS_SET(1);
17.    TCS_SET(1);
18.    TRST_SET(1);
19.    Delay_us( 1000 );
20.
21.    Mp3WriteRegister( SPI_MODE, 0x08, 0x00 ); // 进入 vs1003 的播放模式
22.    Mp3WriteRegister(3, 0x98, 0x00);        // 设置 vs1003 的时钟, 3 倍频
23.    Mp3WriteRegister(5, 0xBB, 0x81);        // 采样率 48k, 立体声
24.    // 设置重低音
25.    Mp3WriteRegister(SPI_BASS, TrebleEnhanceValue, BassEnhanceValue);
26.    Mp3WriteRegister(0x0b, 0x00, 0x00);      // VS1003 音量
27.    Delay_us( 1000 );
28.
29.    while( DREQ == 0 );                  // 等待 DREQ 为高 表示能够接受音乐数据输入
30. }
```

函数中涉及到的宏定义都在 `vs1003.h` 这个头文件中实现。关于函数中为什么要这样操作寄存器，或者为什么要按照这个顺序来操作寄存器，请大家查阅 `vs1003` 的 pdf，里面讲得很详细，有英文跟中文资料。



`MP3_Play();` 这个函数逐个扫描我们 SD 卡里面的音频文件，并将这些音频文件都通过耳机播放出来，最后停止，暂时还没实现循环播放的功能。`MP3_Play();` 在 `vs1003.c` 中实现：

```
1.  /*
2.   * 函数名: MP3_Play
3.   * 描述 : 读取 SD 卡里面的音频文件，并通过耳机播放出来
4.   *          支持的格式: mp3,mid,wav,wma
5.   * 输入 : 无
6.   * 输出 : 无
7.   * 说明 : 暂不支持长文件名，不支持中文。
8.   */
9. void MP3_Play(void)
10. {
11.     FRESULT res;
12.     FILINFO finfo;
13.     DIR dirs;
14.     u16 count = 0;
15.     char j = 0;
16.     char path[50] = {"\"}; /* MicroSD 卡根目录 */
17.     char *result1, *result2, *result3, *result4;
18.
19.     f_mount(0, &fs); /* 挂载文件系统到 0 区 */
20.
21.     if (f_opendir(&dirs, path) == FR_OK) /* 打开根目录 */
22.     {
23.         while (f_readdir(&dirs, &finfo) == FR_OK) /* 依次读取文件名 */
24.         {
25.             if (finfo.fattrib & AM_ARC) /* 判断是否为存档型文档 */
26.             {
27.                 if( !finfo.fname[0] ) /* 文件名为空即到达了目录的末尾，退出 */
28.                     break;
29.                 USART1_printf( USART1, "\r\n 文件名为: %s \r\n", finfo.fname );
30.
31.                 result1 = strstr( finfo.fname, ".mp3" ); /* 判断是否为音频文件 */
32.                 result2 = strstr( finfo.fname, ".mid" );
33.                 result3 = strstr( finfo.fname, ".wav" );
34.                 result4 = strstr( finfo.fname, ".wma" );
35.
36.                 if ( result1!=NULL || result2!=NULL || result3!=NULL || result4
37.                     !=NULL )
38.                 {
39.                     PutChinese_strings22(10, 220, "开始播放", 0, 1);
40.                     res = f_open( &fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ ); /* 以只读方式打开 */
41.
42.                     /* 获取歌曲信息(ID3V1 tag / ID3V2 tag) */
43.                     if ( Read_ID3V1(&fsrc, &id3v1) == TRUE )
44.                         { // ID3V1 tag
45.                             printf( "\r\n 曲目      : %s \r\n", id3v1.title );
46.                             printf( "\r\n 艺术
家 : %s \r\n", id3v1.artist );
47.
48.                             PutChinese_strings22(10, 200, "曲目", 0, 1);
49.                             PutChinese_strings22(80, 200, id3v1.title, 0, 1);
50.
51.                             PutChinese_strings22(10, 180, "艺术家", 0, 1);
52.                             PutChinese_strings22(80, 200, id3v1.artist, 0, 1);
53.
54.                         }
55.                     else
56.                 }
57.                 f_close( &fsrc );
58.             }
59.         }
60.     }
61. }
```



```
55.          // 有些 MP3 文件没有 ID3V1 tag, 只有
56.          ID3V2 tag
57.          res = f_lseek(&fsrc, 0);
58.          Read_ID3V2(&fsrc, &id3v2);
59.          printf( "\r\n 曲目 : %s \r\n", id3v2.title );
60.          printf( "\r\n 艺术家 : %s \r\n", id3v2.artist );
61.
62.          PutChinese_strings22(10, 200, "曲目", 0, 1);
63.          PutChinese_strings22(80, 200, id3v2.title, 0, 1);
64.
65.          PutChinese_strings22(10, 180, "艺术家", 0, 1);
66.          PutChinese_strings22(80, 180, id3v2.artist, 0, 1);
67.
68.
69.          /* 使文件指针 fsrc 重新指向文件头, 因为在调用
70.             Read_ID3V1/Read_ID3V2 时, fsrc 的位置改变了 */
71.          res = f_open( &fsrc, finfo.fname, FA OPEN EXISTING | FA REA
D );
72.          //res = f_lseek(&fsrc, 0);
73.
74.          br = 1;                                /* br 为全局变量 */
75.          TXDCS_SET( 0 );                      /* 选择 VS1003 的数据接口 */
76.          /* ----- 一曲开始 -----*/
77.          USART1 printf( USART1, "\r\n 开始播放 \r\n" );
78.          for (;;)
79.          {
80.              res = f_read( &fsrc, buffer, sizeof(buffer), &br );
81.              if ( res == 0 )
82.              {
83.                  count = 0;                            /* 512
字节完重新计数 */
84.                  Delay_us( 1000 );                  /* 10ms 延时 */
85.                  while ( count < 512 )                /* SD 卡读取一个
sector, 一个 sector 为 512 字节 */
86.                  {
87.                      if ( DREQ != 0 )                  /* 等待 DREQ 为高, 请求数据输
入 */
88.                      {
89.                          for (j=0; j<32; j++) /* VS1003 的 FIFO
只有 32 个字节的缓冲 */
90.                          VS1003_WriteByte( buffer[count] );
91.                          count++;
92.                      }
93.                  }
94.              }
95.          }
96.          if (res || br == 0) break;    /* 出错或者到了 MP3 文件
尾 */
97.      }
98.      USART1 printf( USART1, "\r\n 播放结束 \r\n" );
99.      /* ----- 一曲结束 -----*/
100.     count = 0;
101.     /* 根据 VS1003 的要求, 在一曲结束后需发送 2048 个 0 来确保下
一首的正常播放 */
102.     while ( count < 2048 )
103.     {
104.         if ( DREQ != 0 )
105.         {
106.             for ( j=0; j<32; j++)
107.             {
108.                 VS1003_WriteByte( 0 );
109.             }
110.             count++;
111.         }
112.     }
```



```
110.          }
111.          }
112.      }
113.      count = 0;
114.      TXDCS SET( 1 ); /* 关闭 VS1003 数据端口 */
115.      f_close(&fsrc); /* 关闭打开的文件 */
116.      PutChinese_strings22(10, 220, "播放结束", 0, 1);
117.  }
118.  }
119. } /* while (f_readdir(&dirs, &finfo) == FR_OK) */
120. } /* if (f_opendir(&dirs, path) == FR_OK) */
121. /* end of MP3_Play */
```

1.

这个函数的代码有点长，格式代码有点乱，野火推荐大家还是配合源码一起看，源码对齐风格非常好^\_^。

现在我们来大概分析下 `MP3_Play()` 这个函数，这里边涉及到一些文件系统操作的函数，关于这部分函数的操作大家可参考前面的教程或者阅读 `FATFS` 的官方文档，其实我的教程也不完全正确，阅读官方的文档才是最可靠的。

函数 `f_mount(0, &fs)` 为我们在文件系统中注册一个工作区，并初始化盘符的名为 0，盘符可以初始化为 0~9，我们这里初始化为 0。

函数 `f_opendir(&dirs, path)` 用于打开卡的根目录，并将这个根目录关联到 `dirs` 这个结构指针，然后我们就可以通过这个结构指针来操作这个目录了，其实这个结构指针就类似 `LINUX` 下系统编程中的文件描述符，不论是操作还是目录都得通过文件描述符才能操作，学过 `LINUX` 系统编程的朋友对这点肯定非常了解。

`f_readdir(&dirs, &finfo)` 函数通过刚刚的 `dirs` 结构指针来读取目录里面的信息，并将目录的信息储存在 `finfo` 这个结构体变量中。这个结构体在 `ff.h` 这个头文件中声明。有关具体结构成员的作用大家可直接阅读注释。

```
1.  /* File object structure */
2.
3.  typedef struct _FIL_ {
4.      FATFS* fs;           /* Pointer to the owner file system object */
5.      WORD id;            /* Owner file system mount ID */
6.      BYTE flag;          /* File status flags */
7.      BYTE csect;          /* Sector address in the cluster */
8.      DWORD fptr;          /* File R/W pointer */
9.      DWORD fsize;          /* File size */
10.     DWORD org_clust; /* File start cluster */
11.     DWORD curr_clust; /* Current cluster */
12.     DWORD dsect;          /* Current data sector */
13. #if !_FS_READONLY
14.     DWORD dir_sect; /* Sector containing the directory entry */
15.     BYTE* dir_ptr; /* Pointer to the directory entry in the window */
16. #endif
17. #if !_FS_TINY
18.     BYTE buf[_MAX_SS]; /* File R/W buffer */
19. #endif
20. } FIL;
```



紧接着判断文件的属性，如果是存档型文件的话就将文件名打印出来，然后比较文件的后缀名，查看是否为音频文件，支持的音频格式有 **mp3**、**mid**、**wav**、**wma**。

如果是音频文件的话则调用 `f_open( &fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ );` 打开这个音频文件。

在打开音频文件之后，我们调用函数 `Read_ID3V1()` 来获取 MP3 文件尾部的 ID3V1 tag，如果存在 TAG 标签的话则在电脑的超级终端和 LCD 上显示出曲目和艺术家。

`Read_ID3V1()` 在 `mp3play.c` 中实现：

```
1.  /*
2.   * 函数名: Read_ID3V1
3.   * 描述 : 从 MP3 文件尾读取 ID3V1 的信息，并将这些信息保存在一个全局数组中
4.   * 输入 : - FileObject: file system
5.   *        - info: struct tag_info
6.   * 输出 : 无
7.   * 说明 : 现在的 mp3 基本上都没有 ID3V1 tag(在文件尾部)，有的是 ID3V2 tag(在文件头部)。
8.   */
9. //void Read_ID3V1(FIL *FileObject, struct tag_info *info)
10. int Read_ID3V1(FIL *FileObject, struct tag_info *info)
11. {
12.     /* ID3V1 的信息包含在文件末尾的 128 个 byte 中 */
13.     res = f_lseek(FileObject, FileObject->filesize - 128 );
14.
15.     /* 将 ID3V1 中的 128 个 byte 的信息读到缓冲区 */
16.     res = f_read(FileObject, &readBuf, 128 , &n_Read);
17.
18.     /* ID3V1 的标签头必须是“TAG”，否则认为没有标签 */
19.     if (strncmp("TAG", (char *) readBuf, 3) == 0)
20.     {
21.         strncpy(info->title, (char *) readBuf + 3, MIN(30, sizeof(info->title) -
1));
22.         /* 标题 */
23.         strncpy(info->artist, (char *) readBuf + 3 + 30, MIN(30, sizeof(info->artist) - 1));
24.
25.         /* 作者 */
26.         strncpy(info->album, (char *) readBuf + 3 + 30 + 30, MIN(30, sizeof(info->album) -
1));
27.
28.         /* 专辑 */
29.         strncpy(info->year, (char *) readBuf + 3 + 30 + 30 + 30, MIN(4, sizeof(info->year) -
1)); /* 时间 */
30.         return TRUE;
31.     }
32.     else
33.     {
34.         return FALSE;
35.     }
36. }
```

要想完全读懂这个函数，必须具备一定的 MP3 文件格式的知识，不然你看了也白看，完全不知所以然^\_^。

我们将读取到的关于 MP3 的信息存储在 `id3v1` 这个结构体中，结构体原型在 `mp3play.h` 这个头文件中声明：

```
1. typedef struct tagID3V1
2. {
```



```
3.     char Header[3];      // 标签头必须是"TAG", 否则被认为没有标签头
4.     char Title[30];       // 标题
5.     char Artist[30];      // 作者
6.     char Album[30];       // 专辑
7.     char Year[4];        // 出品年代
8.     char Comment[28];     // 备注
9.     char reserve;        // 保留
10.    char track;          // 音轨
11.    char Genre;          // 风格
12. }ID3V1, *pID3V1;
```

当我们看了这个结构体的原型之后，或许代码又多看懂了一些^\_^。

如果 MP3 尾部没有 TAG 标签的话则从 MP3 的头部获取 MP3 的信息，即 ID3V2 tag，这里我们要知道只要它是 MP3 文件就一定有 ID3V2 tag。所以我们在 else 中调用函数 `Read_ID3V2()`；函数 `Read_ID3V2()` 同样在 `mp3play.c` 中实现：

```
1.  /*
2.   * 函数名: Read_ID3V2
3.   * 描述 : 从 MP3 文件头部读取 ID3V2 的信息，并将这些信息保存在一个全局数组中
4.   * 输入 : - FileObject: file system
5.   *         - info: struct tag_info
6.   * 输出 : 无
7.   * 说明 : MP3 文件都有 ID3V2，包含了作者，曲目，专辑等信息，长度不固定，扩展了 ID3V1 的信息量。
8.   */
9. void Read_ID3V2(FIL *FileObject, struct tag_info *info)
10. {
11.     uint32_t p = 0;
12.
13.     res = f_lseek(FileObject, 0); /* 定位到文件的头部 */
14.     res = f_read(FileObject, &readBuf, READBUF_SIZE, &n_Read); /* 从文件头部开始读取
READBUF_SIZE 个字节 */
15.
16.     if (strncmp("ID3", (char *) readBuf, 3) == 0) /* ID3V3 的头部标签必须为"ID3" */
17.     {
18.         uint32_t tag_size, frame_size, i;
19.         uint8_t version_major;
20.         int frame_header_size;
21.
22.         /* 标签大小，包括标签头的 10 个字节和所有标签帧的大小 */
23.         tag_size = ((uint32_t) readBuf[6] << 21) | ((uint32_t) readBuf[7] << 14) | ((uint1
6_t) readBuf[8] << 7) | readBuf[9];
24.
25.         info->data_start = tag_size;
26.         version_major = readBuf[3]; /* 版本号为 ID3V2.3 时就记录为 3 */
27.
28.         if (version_major >= 3)
29.         {
30.             frame_header_size = 10; /* ID3V2.3 的标签头为 10 个字节 */
31.         }
32.         else
33.         {
34.             frame_header_size = 6;
35.         }
36.         i = p = 10;
37.
38.         /* iterate through frames */
39.         while (p < tag_size) /* 从标签头后开始处理 */
40.         {
41.             if (version_major >= 3)
42.             {
```



```
43.         frame_size = ((uint32_t) readBuf[i + 4] << 24) | ((uint32_t) readBuf[i + 5]
44.             ] << 16) | ((uint16_t) readBuf[i + 6] << 8) | readBuf[i + 7];
45.     }
46.     else
47.     {
48.         frame_size = ((uint32_t) readBuf[i + 3] << 14) | ((uint16_t) readBuf[i + 4]
49.             ] << 7) | readBuf[i + 5];
50.     }
51.     if (i + frame_size + frame_header_size + frame_header_size >= sizeof(readBuf))
52.     {
53.         if (frame_size + frame_header_size > sizeof(readBuf))
54.         {
55.             res = f_lseek(FileObject, FileObject-
>fptra + p + frame_size + frame_header_size);
56.             res = f_read(FileObject, &readBuf, READBUF_SIZE , &n_Read);
57.             p += frame_size + frame_header_size;
58.             i = 0;
59.             continue;
60.         }
61.     else
62.     {
63.         int r = sizeof(readBuf) - i;
64.         memmove(readBuf, readBuf + i, r);
65.         res = f_read(FileObject, (char *) readBuf + r, i , &n_Read);
66.         i = 0;
67.     }
68.     /* 帧标识"TT2"/"TIT2"表示内容为这首歌的 标题 */
69.     if (strncmp("TT2", (char *) readBuf + i, 3) == 0 || strncmp("TIT2", (char *) r
eadBuf + i, 4) == 0)
70.     {
71.         strncpy(info->title, (char *) readBuf + i + frame_header_size + 1, MIN(frame_size - 1, sizeof(info-
>title) - 1));
72.         if( ( info->title[0] == 0xFE && info->title[1] == 0xFF ) || ( info-
>title[0] == 0xFF && info->title[1] == 0xFE ) )
73.         {
74.             /* unicode 格式*/
75.             memset(info->title, 0,sizeof(info->title));
76.             printf( "-- MP3 title no support unicode \r\n");
77.         }
78.     }
79.     /* 帧标识"TP1"/"TPE1"表示内容为这首歌的 作者 */
80.     else if (strncmp("TP1", (char *) readBuf + i, 3) == 0 || strncmp("TPE1", (char
*) readBuf + i, 4) == 0)
81.     {
82.         strncpy(info->artist, (char *) readBuf + i + frame_header_size + 1, MIN(frame_size - 1, sizeof(info-
>artist) - 1));
83.         if( ( info->artist[0] == 0xFE && info->artist[1] == 0xFF ) || ( info-
>artist[0] == 0xFF && info->artist[1] == 0xFE ) )
84.         {
85.             /* unicode 格式*/
86.             memset(info->artist, 0,sizeof(info->artist));
87.             printf( "-- MP3 artist no support unicode \r\n");
88.         }
89.     }
90.     }
91.     }
92.     p += frame_size + frame_header_size;
93.     i += frame_size + frame_header_size;
94. }
95. }
96. }
97. }
```

要想读懂这个函数，我们又要来一次 MP3 知识的扫盲了^\_^.....

ID3V2 到现在一共有 4 个版本，但流行的播放软件一般只支持第三版本，即 ID3V2.3。由于 ID3V1 记录在文件尾部了，故 ID3V2 就只能记录在头部了，也正是这个原因，对 ID3V2 的操作要比 ID3V1 慢，这也是我们在程序中先检测 ID3V1 的原因，



---

只要我们检测到 ID3V1 就不用去检测 ID3V2 了，况且检测 ID3V2 的代码实现要复杂得多。而且 ID3V2 结构比 ID3V1 结构要复杂很多，但比 ID3V1 全面且可以伸缩和扩展。

每个 ID3V2.3 的标签都有一个标签头和若干个标签帧或一个扩展标签头组成。关于曲目的信息如标题、作者等信息都存放在不同的标签帧中，扩展标签头和标签帧并不是必要的，但每个标签至少有一个标签帧。标签头和标签帧一起存放在 MP3 文件的首部。

### ● 标签头

在文件的首部顺序记录 10 个字节的 ID3V2.3 的头部，数据结构如下：

```
1. char Header[3];      // 必须为“ID3”，否则认为标签不存在
2. char Ver             // 版本号为 ID3V2.3 就记录为 3
3. char Revision        // 副版本号此版本记录为 0
4. char Flag            // 存放标志的字节，这个版本只定义了三位
5. char Size[4]          // 标签大小，包括标签头的 1 个字节和所有的标签帧的大小
```

标签大小的计算：

一共四个字节，但每个字节只用 7 位，最高位恒为 0，格式如下：

0xxxxxxxx, 0xxxxxxxx, 0xxxxxxxx, 0xxxxxxxx

计算的时候要将 0 去掉，得到一个 28 位的二进制数，就是标签大小，

计算公式如下：

```
1. int total_size;
2. total_size = (Size[0]&0x7f) << 21
3.           +(Size[1]&0x7f) << 14
4.           +(Size[3]&0x7f) << 7
5.           +(Size[4]&0x7f);
```

看到这对代码的实现是不是有点感觉了呀，别急，下面我们继续来拨开那层层迷雾^\_^.....

### ● 帧标识

用四个字符标识一个帧，说明一个帧的内容含义，常用的对照如下：

**TIT2**: 标题，表示内容为这首歌的标题

**TPE1**: 作者

**TALB**: 专辑



---

**TRCK:** 音轨(格式: N/M, 其中 N 为专辑中的第 N 首, M 为专辑中共 M 首, N 和 M 为 ASCII 码表示的数字)

**TYPE:** 年代 是 ASCII 码表示的数字

**TCON:** 类型 直接用字符串表示

**COMM:** 备注 格式: “eng\0 备注内容”, 其中 eng 表示备注所使用的自然语言

但要注意, 这里面有些帧标识 ID3V2 里面是没有的, 但标题和作者一般都会有。

要想知道 ID3V2 是否包含有这些信息, 我们可以用 16 进制的编辑器来查看 MP3 文件, ID3V2 在头部, ID3V1 在尾部。考虑到大家可能没有用过 16 进制的编辑器, 所以野火在光盘资料《常用代码编辑软件》这个文件夹下提供了 Uedit32 这个 16 进制编辑器, 这个编辑器还可以阅读 c/c++ 代码, 功能非常强大。

### ● 帧大小的计算

这个可没有标签头的算法那么麻烦, 每个字节的 8 位全用, 格式如下:

XXXXXXXX, XXXXXXXX, XXXXXXXX, XXXXXXXX

代码如下:

```
1. int FSize;
2. FSize = Size[0] * 0x100000000
3.      +Size[1] * 0x10000
4.      +Size[2] * 0x100
5.      +Size[3];
```

---

讲到这里, `Read_ID3V2()` 函数里面的代码大伙也应该理解的差不多了^\_^。下面我们再接再厉, 趁着势头把后面的代码也一口气读完吧。

`TXDCS_SET( 0 );` 用于选择 `vs1003` 的数据端口, 准备往 `vs1003` 中输入数据。其中 `TXDCS_SET( 0 )`, 是在 `vs1003.h` 中实现的一个宏:

```
1. #define XDCS    (1<<6)    // PC6-XDCS
2.
3. #define TXDCS_SET(x)  GPIOC->ODR= (GPIOC->ODR&~XDCS) | (x ? XDCS:0)
```



紧接着进入一个大循环中播放我们的 mp3 文件。

函数 `f_read( &fsrc, buffer, sizeof(buffer), &br );` 从文件中读取 512 个字节的数据到缓冲区中，至于为什么是 512 个字节，而不是 1024 或者更多，这是因为卡的一个 sector 是 512 个字节，一次只能读取一个 sector。

函数 `vs1003_WriteByte( buffer[count] );` 将缓冲区中的数据写入 vs1003 的数据缓冲区。注意，这里一次只能写入 32 个字节，这是因为 vs1003 的 FIFO 的大小为 32 个字节，写多了无效。

当文件出错或者一曲播放完毕时就跳出 for 循环，并打印出 “播放结束”的调试信息。

根据 VS1003 的要求，在一曲结束后需发送 2048 个 0 来确保下一首的正常播放。

一曲播放完毕我们关闭 vs1003 的数据端，关闭打开的文件，等待下一曲的播放，直到目录下的音频文件播放完为止。还暂不支持循环播放的功能。

这里面涉及到了 vs1003 操作的一些特性，需大家参考 vs1003 的 datasheet 来帮助理解，野火建议大家仔仔细细地阅读，不要因为难度或者是繁琐而退缩，要知道作为一个嵌入式软件开发工程师，阅读 ic datasheet（中文/英文）是一种必备的技能。

### 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](#))，插上 MicroSD 卡( 我用的是 1G )，在卡的根目录下要有 mp3 文件，文件名要是英文，暂不支持中文和长文件名，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（在 LCD 里面的效果图这里就没有拍照了，大家见谅^\_^）：

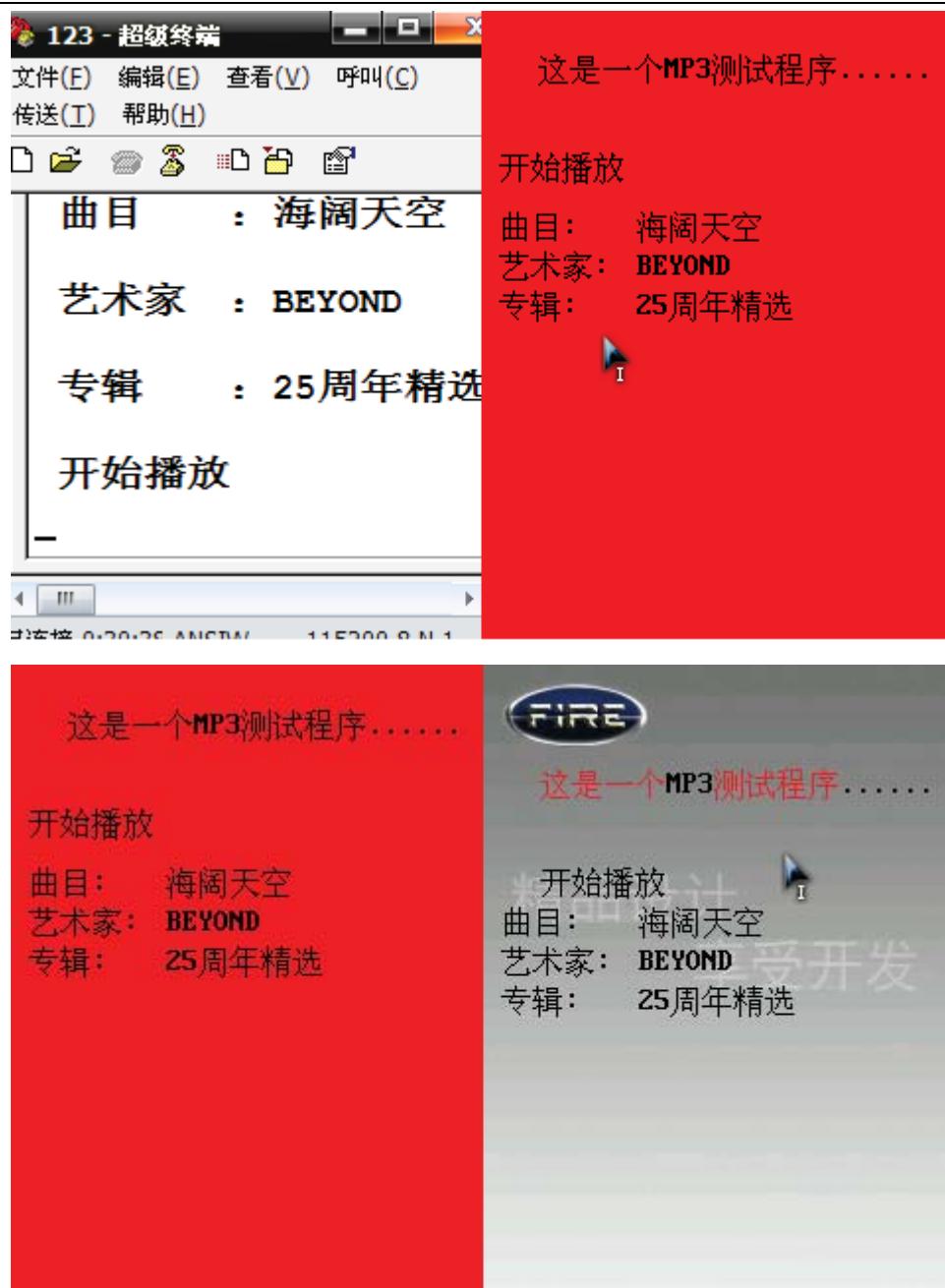


```
123 - 超级终端
文件(E) 编辑(E) 查看(V) 呼叫(C) 传送(I) 帮助(H)
□ 文件夹 图像 文本档 目录 网页

the music file name is: yydt.mp3
开始播放
播放结束
the music file name is: ysbg.mp3
开始播放
播放结束
the music file name is: yg.mp3
开始播放
-
已连接 4:53:58 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(I) 帮助(H)
□ 文件夹 图像 文本档 目录 网页

开始播放
这是一个MP3测试例程 !
文件名为: HZLIB.bin
文件名为: PIC.bmp
文件名为: yugan.mp3
曲目      : 预感
艺术家    : 陈奕迅
开始播放
-
已连接 3:12:30 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```



这两张是 LCD 里面显示的信息，不过在测试时野火换了首歌曲^\_^。

我的卡的根目录下放了 3 个 mp3 文件，都是 Eason 的歌(因为个人非常喜欢陈奕迅的歌^\_^)，分别是遥远的她(yydt.mp3)、一丝不挂(ysbg.mp3)、预感(yugan.mp3)。插上耳机，音质堪比电脑，音量可通过耳机来调，前提是你的耳机要能调节音量才行。

实验讲解完毕，野火祝大家学习愉快^\_^。



## RTC (电子时钟/日历) 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: 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。

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。

硬件连接: VBAT 引脚需外接电池。

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\_pwr.c

FWlib/stm32f10x\_bkp.c

FWlib/stm32f10x\_rtc.c

FWlib/stm32f10x\_misc.c

用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/usart.c

USER/rtc.c



---

## 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 BKREN 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。

## 实验讲解->

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\r\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\r\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\r\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：

```
This is a RTC demo.....  
This is a RTC demo!  
  
RTC not yet configured....  
RTC configured....  
=====Time Settings=====  
Please Set Hours: 22  
Please Set Minutes: 8  
Please Set Seconds: 0  
_Time: 22:08:08
```

已连接 0:00:34 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印



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：

The screenshot shows a Windows-style terminal window titled "123 - 超级终端". The menu bar includes "文件(F)", "编辑(E)", "查看(V)", "呼叫(C)", "传送(T)", and "帮助(H)". Below the menu is a toolbar with icons for file operations. The main text area displays three sets of RTC configuration logs:

```
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
```

At the bottom of the window, status information is shown: 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：

```
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



```
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 插 打印
```

3-> 条件编译选项部分问我们是否需要 `output RTCCLK/64 on Tamper pin,`  
(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：



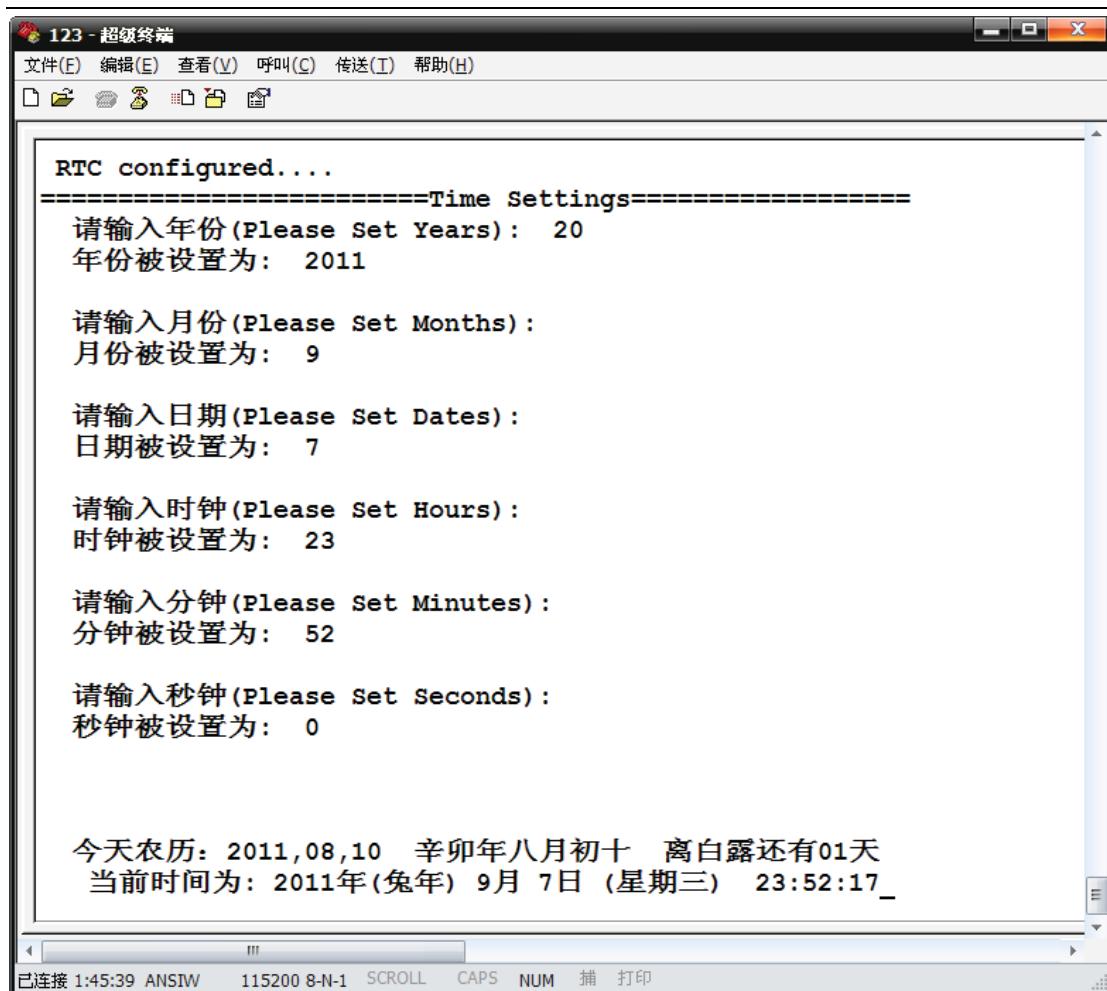
The screenshot shows a terminal window titled "123 - 超级终端". The window contains three sets of text output from an RTC demo. Each set includes a timestamp, a message indicating an external reset, and the RTC configuration message. The terminal window has a standard Windows-style interface with a menu bar, toolbar, and status bar at the bottom.

```
No need to configure RTC....  
Time: 23:41:08  
This is a RTC demo.....  
  
External Reset occurred....  
No need to configure RTC....  
Time: 23:41:09  
This is a RTC demo.....  
  
External Reset occurred....  
No need to configure RTC....  
Time: 23:41:13
```

已连接 1:33:39 ANSIW 115200 8-N-1 SCROLL CAPS NUM 插 打印

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)。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，分析源码的任务就交给大家了。里面涉及到些算法和结构体的知识，大家就好好琢磨吧。源码目录放在 [RTC+CALENDAR](#) 这个文件夹中。

实验讲解完毕，野火祝大家学习愉快^\_^。



## RFID 智能卡 **RC522+USART+TIM3** 实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 <b>STM32</b> 开发板
库版本	<b>ST3.0.0</b>

实验描述: 定时器 **3** 每 **50ms** 扫描一次, 判断串口 **1** 输入的控制信息; 通过控制信息可对 **RC522** 进行发送卡号、读卡、写卡、修改密码操作。**RFID** 智能卡可应用于校园卡, 公交卡, 小区热水充值、上班个人名片等。

硬件连接:      PC7 : TDIN

                  PC5 : TCLK

                  PB2 : TCS

                  PB1 : RST

                  PB6 : DOUT

库文件 : CMSIS/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\_flash.c

FWlib/stm32f10x\_tim.c

FWlib/misc.c



用户文件: USER/main.c

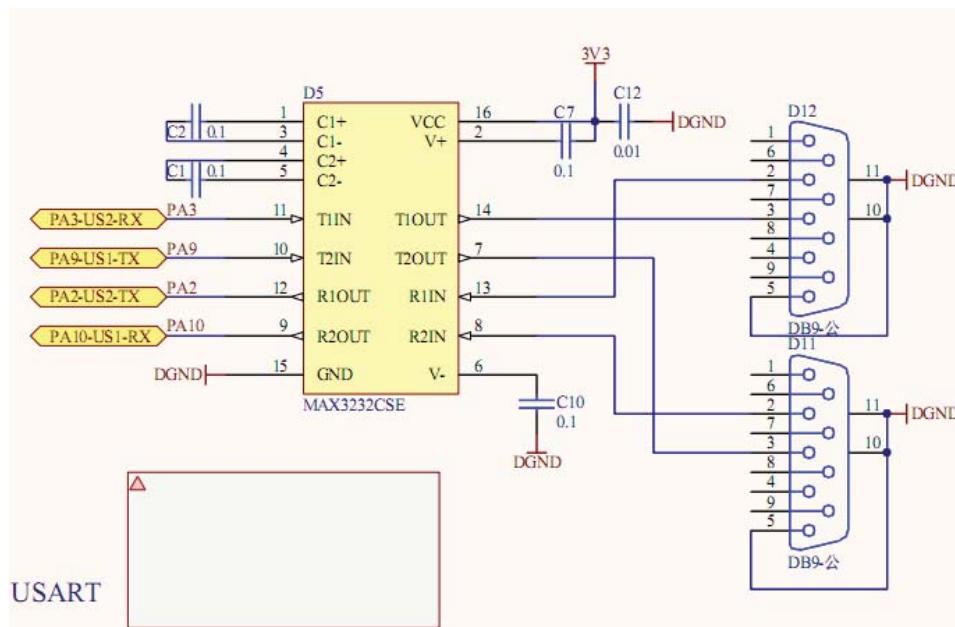
USER/stm32f10x\_it.c

USER/rc522.c

USER/Com.c

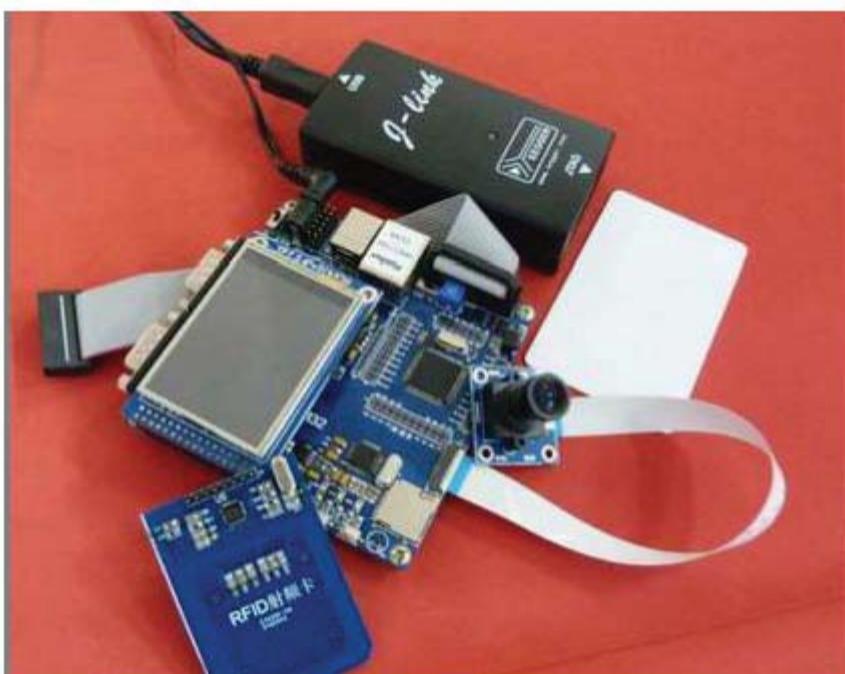
USER/RFID.c

野火 STM32 开发板中 USART 的硬件原理图:





野火 stm32 除了板载了摄像头接口外，还板载了 **RFID** 智能卡接口，  
该智能卡可应用于校园卡、公交卡、小区热水充值、个人名片等。  
配 **stm32** 驱动代码和详细 **PDF** 教程。



野火 stm32 开发板为 **RFID** 引出了 7 个接口，其中 2 个为电源接口（VCC-GND），  
因为此例程用的是模拟 **SPI**，所以剩下的 5 个只需用普通的 IO 口（硬件连接提到，可  
根据自己需要修改）；如果不是用模拟的 **SPI**，就必须要用固定的 **SPI** 接口，这个读者  
可以另外去尝试。本实验要用串口控制 **RFID** 的收、读、写，所以必须了解串口的使用。



## RFID 射频卡简介->

Mifare 1 芯片内部结构较为复杂，可分为射频接口、数字处理单元、E2PROM（1K 字节）三部分，逻辑框图如下：



## 存储器组织结构

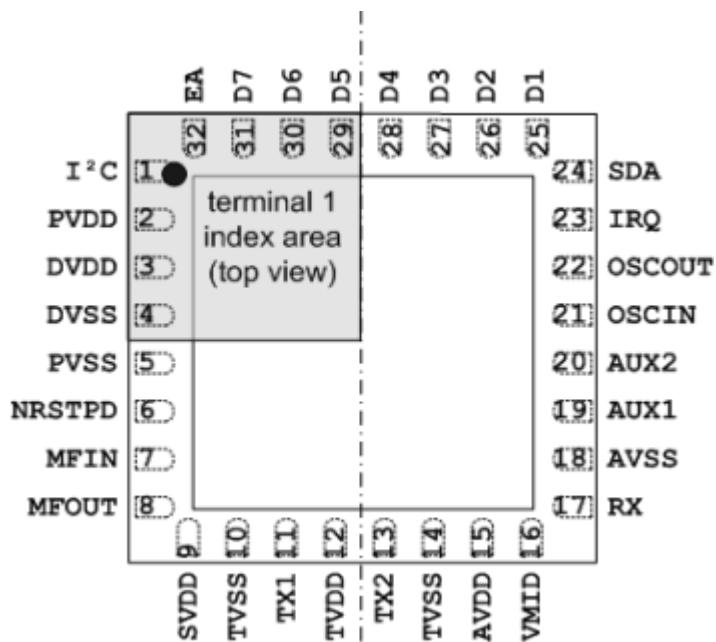
扇区	块	描述
15	63	第 15 扇区尾块
	62	数据块
	61	数据块
	60	数据块
14	59	第 14 扇区尾块
	58	数据块
	57	数据块
	56	数据块
1	7	第 1 扇区尾块
	6	数据块
	5	数据块
	4	数据块
0	3	第 0 扇区尾块
	2	数据块
	1	数据块
	0	厂商标志块

Mifare 1 卡片的存储容量为  $8192 \times 1$  位字长（即  $1K \times 8$  位字长），采用 E2PROM 作为存储介质。整个结构划分为 16 个扇区，编为扇区 0~15。每个扇区有 4 个块 (Block)，分别为块 0, 块 1, 块 2 和块 3。每个块有 16 个字节。一个扇区共有  $16 \text{ Byte} \times 4 = 64 \text{ Byte}$ 。上如所示。每个扇区的块 3 (即第四块) 也称作尾块，包含了该扇区的密码 A(6 个字节)、存取控制(4 个字节)、密码 B(6 个字节)。其余三个块是一般的数据块。扇区 0 的块 0 是特殊的块，包含了厂商代码信息，在生产卡片时写入，不可改写。其中：第 0~4 个字节为卡片的序列号，第 5 个字节为序列号的校验码；第 6



个字节为卡片的容量“SIZE”字节；第 7，8 个字节为卡片的类型号字节，即 Tagtype 字节；VS1003 通过一个串行接口来接收输入的比特流，它可以作为一个系统的从机。输入的比特流被解码，然后通过一个数字音量控制器到达一个 18 位过采样多位  $\epsilon - \Delta$  DAC。通过串行总线控制解码器。除了基本的解码，在用户 RAM 中它还可以做其他特殊应用，例如 DSP 音效处理。

### RC522 的引脚图：



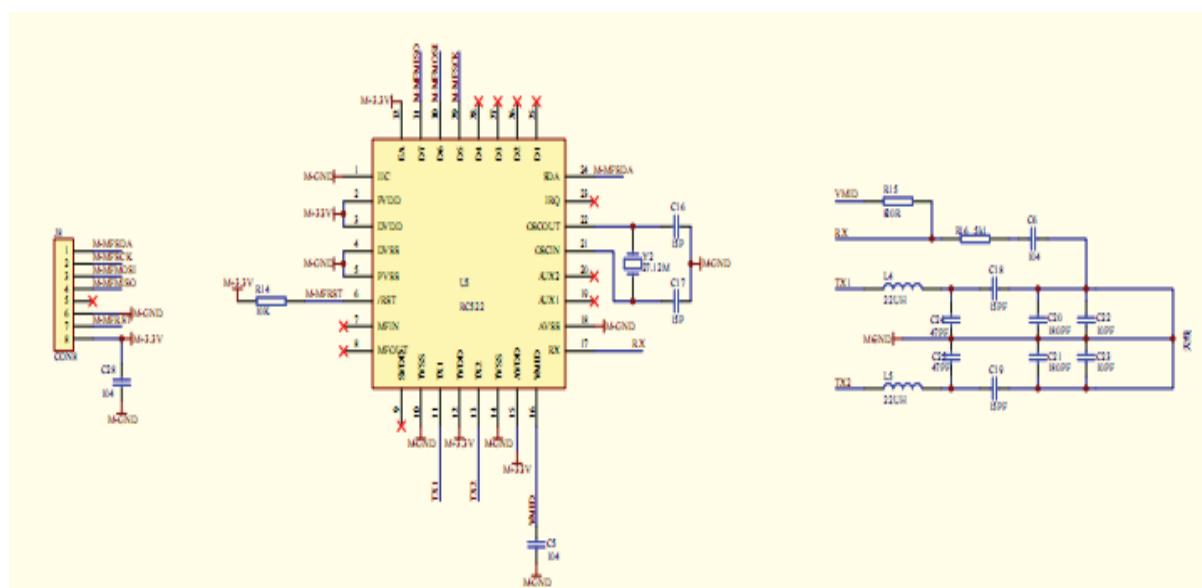
引脚描述：



Symbol	Pin	Type	Description
I <sup>2</sup> C	1	I	I <sup>2</sup> C enable <sup>[2]</sup>
PVDD	2	PWR	Pad power supply
DVDD	3	PWR	Digital Power Supply
DVSS	4	PWR	Digital Ground <sup>[1]</sup>
PVSS	5	PWR	Pad power supply ground
NRSTPD	6	I	Not Reset and Power-down: When LOW, internal current sinks are switched off, the oscillator is inhibited, and the input pads are disconnected from the outside world. With a positive edge on this pin the internal reset phase starts.
MFIN	7	I	Mifare Signal Input
MFOUT	8	O	Mifare Signal Output
SVDD	9	PWR	MFIN / MFOUT Pad Power Supply: provides power to for the MFIN / MFOUT pads
TVSS	10, 14	PWR	Transmitter Ground: supplies the output stage of TX1 and TX2
TX1	11	O	Transmitter 1: delivers the modulated 13.56 MHz energy carrier
TVDD	12	PWR	Transmitter Power Supply: supplies the output stage of TX1 and TX2
TX2	13	O	Transmitter 2: delivers the modulated 13.56 MHz energy carrier
TVSS	10, 14	PWR	Transmitter Ground: supplies the output stage of TX1 and TX2
AVDD	15	PWR	Analog Power Supply
VMID	16	PWR	Internal Reference Voltage: This pin delivers the internal reference voltage.
RX	17	I	Receiver Input. Pin for the received RF signal.
AVSS	18	PWR	Analog Ground
AUX1	19	O	Auxiliary Outputs: These pins are used for testing.
AUX2	20	O	
OSCIN	21	I	Crystal Oscillator Input: input to the inverting amplifier of the oscillator. This pin is also the input for an externally generated clock ( $f_{osc} = 27.12$ MHz).
OSCOUT	22	O	Crystal Oscillator Output: Output of the inverting amplifier of the oscillator.
IRQ	23	O	Interrupt Request: output to signal an interrupt event
SDA	24	I	Serial Data Line <sup>[2]</sup>
D1	25	I/O	Data Pins for different interfaces (test port, I <sup>2</sup> C, SPI, UART) <sup>[2]</sup>
D2	26	I/O	
D3	27	I/O	
D4	28	I/O	
D5	29	I/O	
D6	30	I/O	
D7	31	I/O	
EA	32	I	External Address: This Pin is used for coding I <sup>2</sup> C Address <sup>[2]</sup>



根据 RC522 的天线匹配要求及引脚功能，设计出 RC522 的外接图如下：



天线部分包括发送和接收部分（设计时主要参考 MFRC500 的匹配电路和天线的应用指南.pdf）；本实验中我们用了 STM32 的 7 个接口，采用模拟 SPI 的方法与 RC522 通信。RC522 的晶振为 27.120MHZ（引脚描述有写到）。此方法设计射频距离约为 6 厘米（这种类型的卡最多是 10 厘米，若想距离更加长的话，自己可以另行设计）。

有关 rc522 的详细应用，大家可参考官方的 datasheet，例如 RC500 书籍.pdf, RC522.pdf, 设计 MFRC500 的匹配电路和天线的应用指南.pdf。



本实验是在《RFID+COM+模拟 SPI》这个实验基础上进行的。没做过这个实验的话可参考前面的教程。

### 实验讲解->

首先需要将需要用到的库文件添加进来，有关库的配置可参考前面的教程，这里不再详述。在配置好库的环境之后我们从 `main` 函数开始分析：

```
1. /***** (C) COPYRIGHT 2011 野火嵌入式开发工作
室 *****/
2. * 文件名 : main.c
3. * 描述   : 通过串口 1 可对 RC522 进行发送卡号、读卡、写卡、修改密码操作
4.
5. * 实验平台: 野火 STM32 开发板
6. * 库版本  : ST3.0.0
7.
8. * 作者    : fire QQ: 313303034
9. * 博客    : firestm32.blog.chinaunix.net
10. ****
*****/
11.
12. #include "stm32f10x.h"
13. /*函数原型声明*/
14. extern void InitAll(void);
15. extern void RC522(void);
16. ****
*****/
17. * 描述   : 主函数
18. * 输入   : 无
19. * 输出   : 无
20. * 返回   : 无
21. ****
*****/
22. int main(void)
23. {
24.
25.     InitAll(); //初始化所有设置，串口波特率 115200，系统时钟 72MHZ
26.     while (1)
27.     {
28.         RC522();
29.     }
30. }
31.
32.
33.
34. /***** (C) COPYRIGHT 2011 野火嵌入式开发工作
室 *****END OF FILE****/
```

我们先来看一下 `InitAll()`；这个是初始化函数（已带有详解、），其内容如下：



```
1.  /*****
2.  * 描述 : 初始化所有设置, 串口波特率 115200
3.  * 输入 : 无
4.  * 输出 : 无
5.  * 返回 : 无
6. *****/
7. void InitAll(void)
8. {
9.     /* System Clocks Configuration */
10.    delay_init(72);
11.    SystemInit();           //初始化系统时钟 72MHZ
12.    GPIO_Configuration(); //IO 口初始化
13.    NVIC_Configuration(); //配置 USART1、TIM3 中断优先级
14.    TIM_Configuration(); //配置定时器 3, 定时时长为 20ms
15.    USART1_InitConfig(115200); //串口波特率 115200
16.    InitRc522();          //初始化 RC522
17.    SysTime=0;             //定时器定时次数初始化为 0
18.    KuaiN=0;               //rc522 块区初始化为 0
19.    oprationcard=0;        //初始化 rc522 的操作类型
20.    uart_count=0;          //串口收到的字节数初始化为 0
21.    uart_comp=0;           //初始化串口标志
22. }
```

再者，就是 RC522();这个函数了，这个函数一直循环执行。功能主要是每进入两次定时器中断（即 50ms）则对操作类型重新选择（判断）。

```
1.  /*****
2.  * 描述 : 每进入两次定时器中断（即 50ms）则对操作类型重新选择（判断）
3.  * 输入 : 无
4.  * 输出 : 无
5.  * 返回 : 无
6. *****/
7. void RC522(void)
8. {
9.     if(uart_comp)      //配置读卡, 写卡, 修改密码
10.    {
11.        ctrl_uart();
12.    }
```



```
13.     if(SysTime>=2)      // 每进入两次定时器中断则检查一次
14.     {
15.         SysTime=0;
16.         ctrlprocess();    //每进入两次定时器中断（即 50ms）则对操作类型重新选择
17.     }
18. }
```

uart\_comp 为 **发送完已设的字节数** 设的标志。当此标志为真时，重新预设 RC522 的操作类型。其中当串口收到的都二个字节数为 0XA0 时表示发送卡号，为 0XA1 时表示读卡，为 0XA2 时表示写卡，为 0XA3 时表示修改密码。串口要以怎样的格式发送控制信息，《用前必读.txt》以及下面的**实验现象** 有讲到。。

SysTime 为定时器 3 中断的次数，当次数大于等于 2 时（即 50ms 时）则对卡重新操作。

这个函数里面所涉及的两个函数都是非常重要的，下面一一细看。

**void ctrl\_uart(void)** 这个函数的功能是预设操作类型（发送卡号、读卡、写卡、修改密码）。

```
1.  ****
2.  * 描述 : 预设操作类型（发送卡号、读卡、写卡、修改密码）
3.  * 输入 : 无
4.  * 输出 : 无
5.  * 返回 : 无
6.  ****
7. void ctrl_uart(void)
8. {
9.     u8 ii;
10.    switch(RevBuffer[1])
11.    {
12.        case 0xa0: //发送卡号
13.            oprationcard=SENDID;
14.            break;
```



```
15.     case 0xa1://读数据
16.         oprationcard=READCARD;
17.         for(ii=0;ii<6;ii++)
18.         {
19.             PassWd[ii]=RevBuffer[ii+2];
20.         }
21.         KuaiN=RevBuffer[8];
22.         break;
23.     case 0xa2://写数据
24.         oprationcard=WRITECARD;
25.         for(ii=0;ii<6;ii++)
26.         {
27.             PassWd[ii]=RevBuffer[ii+2];
28.         }
29.         KuaiN=RevBuffer[8];
30.         for(ii=0;ii<16;ii++)
31.         {
32.             WriteData[ii]=RevBuffer[ii+9];
33.         }
34.         break;
35.     case 0xa3: //修改密码
36.         oprationcard=KEYCARD;
37.         for(ii=0;ii<6;ii++)
38.         {
39.             PassWd[ii]=RevBuffer[ii+2];
40.         }
41.         KuaiN=RevBuffer[8];
42.         for(ii=0;ii<6;ii++)
43.         {
44.             NewKey[ii]=RevBuffer[ii+9];
45.             NewKey[ii+10]=RevBuffer[ii+9];
46.         }
47.         break;
48.     default: break;
49. }
50. }
```

从这个函数可以看出除了发送卡号之后，其它类型的操作都要带密码操作的，这也是射频卡的一个特征。这里面对其中几个变量和数组进行讲解

Oprationcard 表示的是操作类型；卡里面有 16 个扇区，每个扇区有 4 个块，KuaiN 表示要访问的块；数组 PassWd[] 存放访问的块的密码；NewKey[] 则存



放要修改的新密码，只有操作类型为 oprationcard=KEYCARD (即修改密码才用到； WriteData[] 则放要写如卡里的数据，相应的地只有操作类型为 oprationcard=WRITECARD 才用到。

下面这个函数是非常重要的，要完全读懂这个函数，请去细看 RC522 的 Datasheep.

```
1.  ****
2.  * 描述 : 对寻卡、防冲撞、选卡、发送卡号、读卡、写卡、修改密码进行操作
3.          成功则 LED1 灯亮
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 返回 : 无
7.  ****
8.  ****/
9. void ctrlprocess(void)
10. {
11.     unsigned char ii;
12.     char status;
13.     PcdReset();
14.     status=PcdRequest(PICC_REQIDL,&RevBuffer[0]); //寻天线区内未进入休眠状态
15.     的卡，返回卡片类型 2 字节
16.     if(status!=MI_OK) return;
17.     status=PcdAnticoll(&RevBuffer[2]); //防冲撞，返回卡的序列号 4 字节
18.     if(status!=MI_OK) return;
19.     memcpy(MLastSelectedSnr,&RevBuffer[2],4);
20.     status=PcdSelect(MLastSelectedSnr); //选卡
21.     if(status!=MI_OK) return;
22.     if(oprationcard==KEYCARD) //修改密码
23.     {
24.         oprationcard=0;
25.         status=PcdAuthState(PICC_AUTHENT1A,Kuain,PassWd,MLastSelectedSnr);
26.         if(status!=MI_OK) return;
```



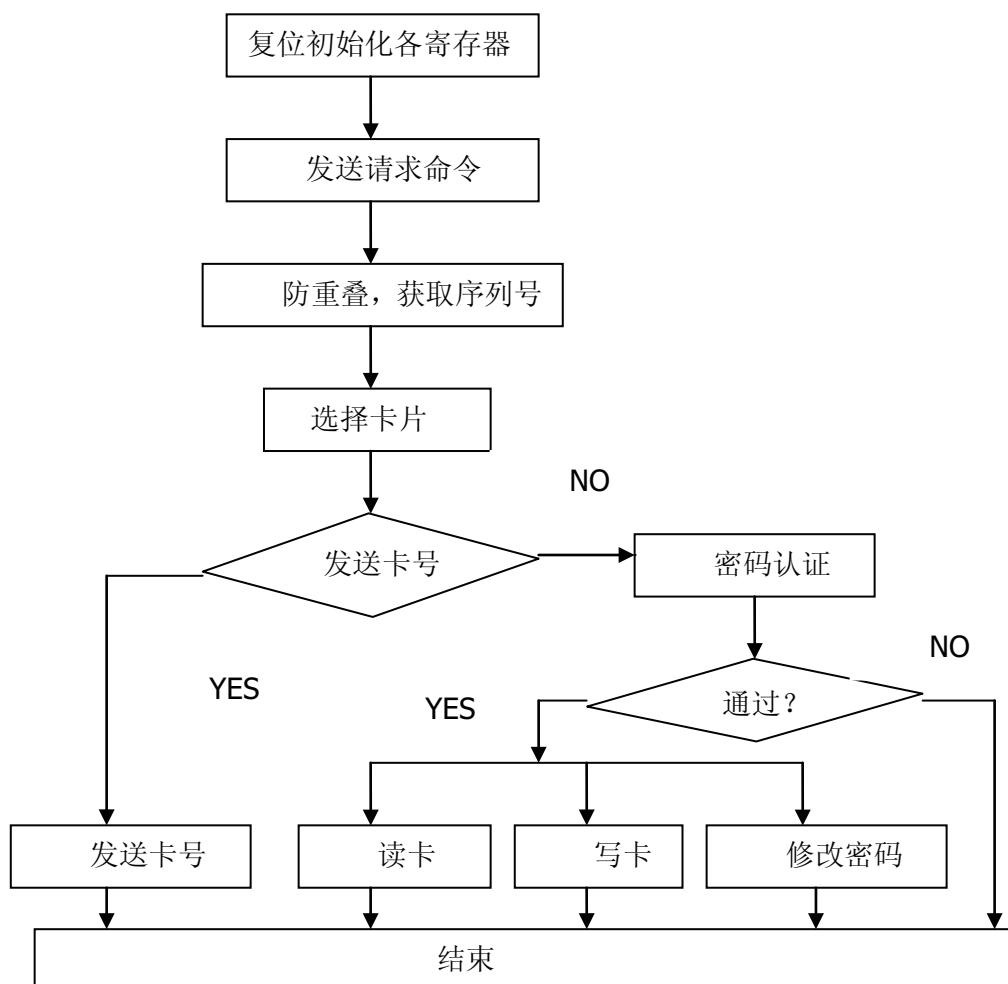
```
26.     status=PcdWrite(KuaiN,&NewKey[0]);
27.     if(status!=MI_OK)
28.         return;
29.     PcdHalt();
30.     uart_count=0;
31.     CLR_BEEP;
32. }
33. else if(oprationcard==READCARD) //读卡
34. {
35.     oprationcard=0;
36.     status=PcdAuthState(PICC_AUTHENT1A,KuaiN,PassWd,MLastSelectedSnr);
//验证 A 密匙
37.     if(status!=MI_OK)    return;
38.
39.     status=PcdRead(KuaiN,Read_Data);
40.     if(status!=MI_OK)    return;
41.
42.     for(ii=0;ii<16;ii++)UART1_SendByte(Read_Data[ii]);
43.     PcdHalt();
44.     uart_count=0;
45.     CLR_BEEP;
46. }
47. else if(oprationcard==WRITECARD) //写卡
48. {
49.     oprationcard=0;
50.     status=PcdAuthState(PICC_AUTHENT1A,KuaiN,PassWd,MLastSelectedSnr);
51.     if(status!=MI_OK)          return;
52.     status=PcdWrite(KuaiN,&WriteData[0]);
53.     if(status!=MI_OK)
54.     {
55.         return;
56.     }
57.     PcdHalt();
58.     uart_count=0;
59.     CLR_BEEP;
60. }
61. else if(oprationcard==SENDID) //发送卡号
62. {
63.     oprationcard=0;
64.     for(ii=0;ii<4;ii++)
65.     {
66.         cardID[ii]=MLastSelectedSnr[ii];
```



```
67.     UART1_SendByte (MLastSelectedSnr[ii]);  
68. }  
69. uart_count=0;  
70. CLR_BEEP;  
71. }  
72.  
73. }
```

整个函数都是调用 RC522 的库函数，库函数在 rc522.c 文件里，读起来非常清晰。 CLR\_BEEP 是当每个操作成功时 LED1 亮。下面则是这个函数的流程图：

流程图：



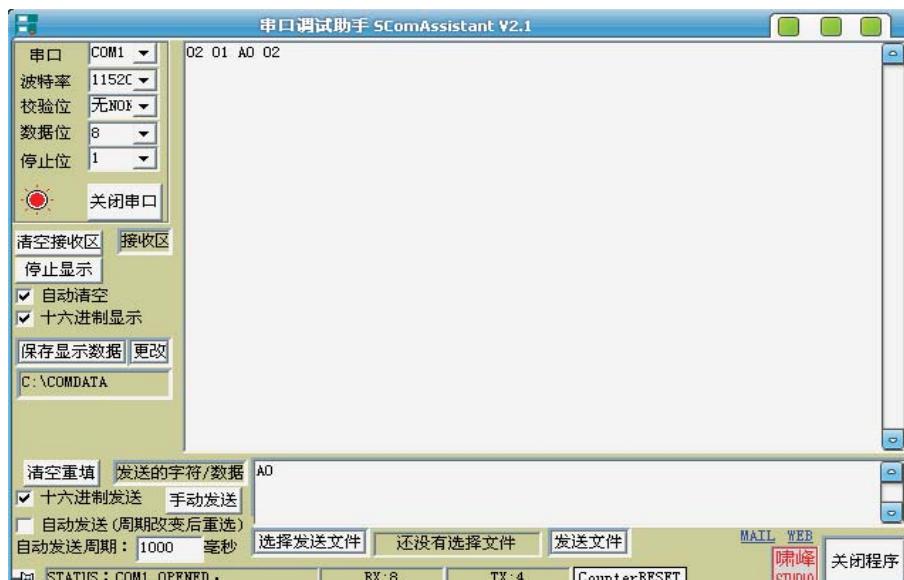


### 实验现象->

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), 将 RC522 连接上 STM32, 打开串口调试助手, 配置为 115200 8-N-1, 将编译好的程序下载到开发板。

#### 1、读卡号 02 A0

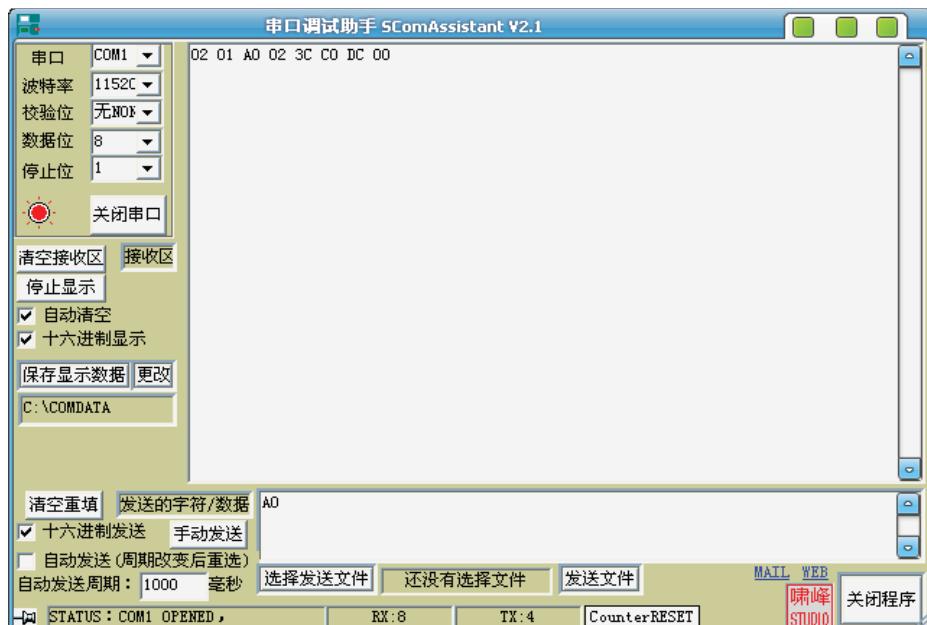
复位后在串口调试助手中以十六进制输入 02 A0 , 现象如下:



第一字节为要发送控制信息的总字节数, 第二个字节和第四个字节分别为 01 和 02, 是为了方便看已输入多少个字节所用的【注: 下面所看到的现象有 01, 02, 03, 04, 05 ... 也是此作用】。在 `stm32f10x_it.c` 文件里, `void USART1_IRQHandler(void)` 中断函数中可将【 `uart_count++; USART1_SendByte(uart_count); //为了方便得知输入数字的个数, 可省略` 】修改或删除。



然后平行地将卡靠近 RFID 射频器，距离为 6 厘米之内，将看到下面现象：



其中 3C C0 DC 00 为卡号；

## 2、读数据

09 A1 Key0 Key1 Key2 Key3 Key4 Key5 Kn.

例:0xA1 为读数据标志。

09 为发送的总字节数；

该卡密码 A 为 16 进制:ff ff ff ff ff ff 对应 Key0 Key1 Key2 Key3 Key4 Key5;

要读的块数为第 8 块 即 Kn=8；

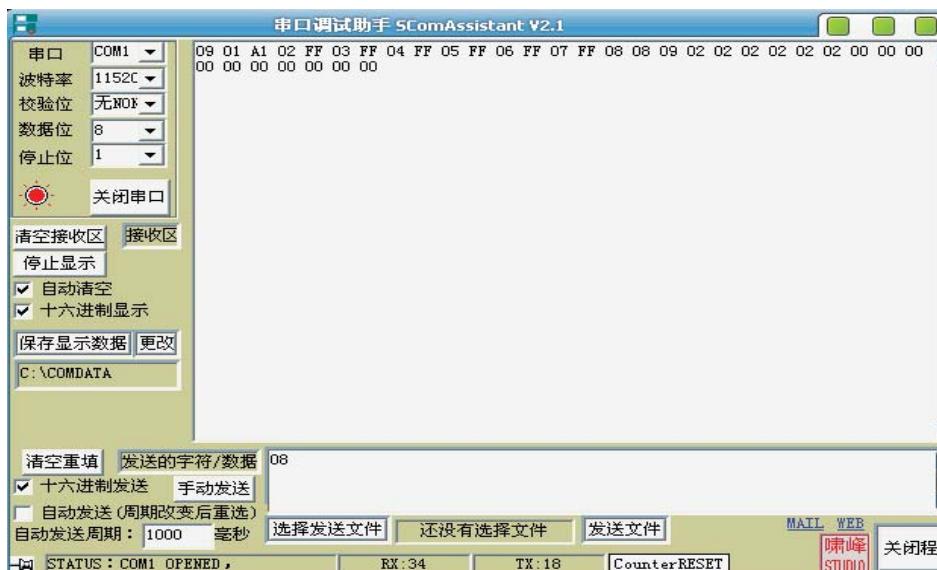
则发送:09 A1 ff ff ff ff ff ff 08 .

返回第 8 块的 16 字节数据.

复位后在串口调试助手中以十六进制输入 09 A1 ff ff ff ff ff ff 08，现象如下：



然后平行地将卡靠近 RFID 射频器，距离为 6 厘米之内，将看到下面现象：



其中 02 02 02 02 02 02 00 00 00 00 00 00 00 00 00 00 是在卡里读出的数据

### 3、写数据

19 A2 Key0 Key1 Key2 Key3 Key4 Key5 Kn Num0 Num1 Num2 Num3 Num4  
Num5 Num6 Num7 Num8 Num9 Num10 Num11 Num12 Num13 Num14 Num15.

例:0xA2 为写数据标志。

该卡密码 A 为 16 进制:FF FF FF FF FF FF 对应 Key0 Key1 Key2 Key3 Key4 Key5;

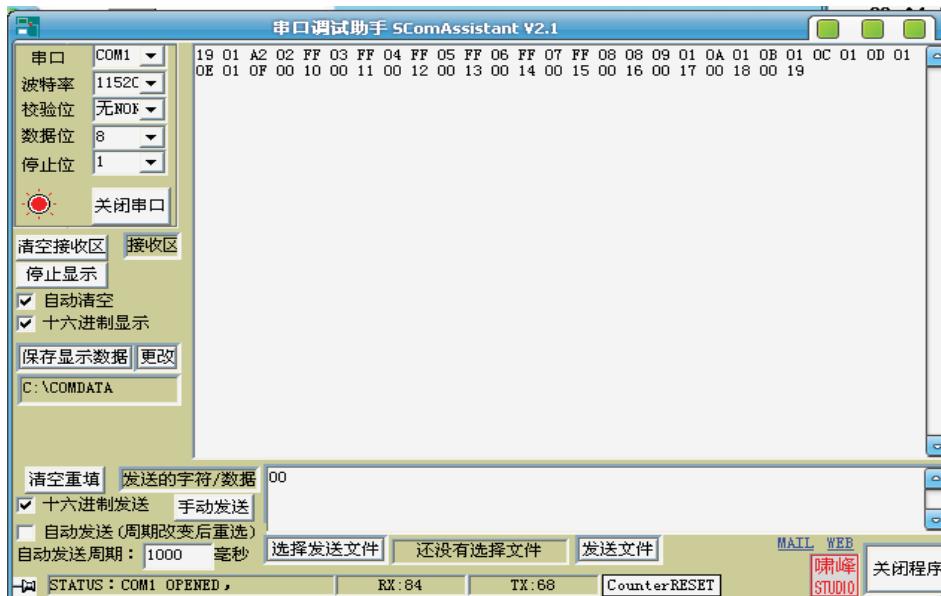
要写的块数为第 8 块 即 Kn=8;

要写的数据位 01 01 01 01 01 01 00 00 00 00 00 00 00 00 00 00

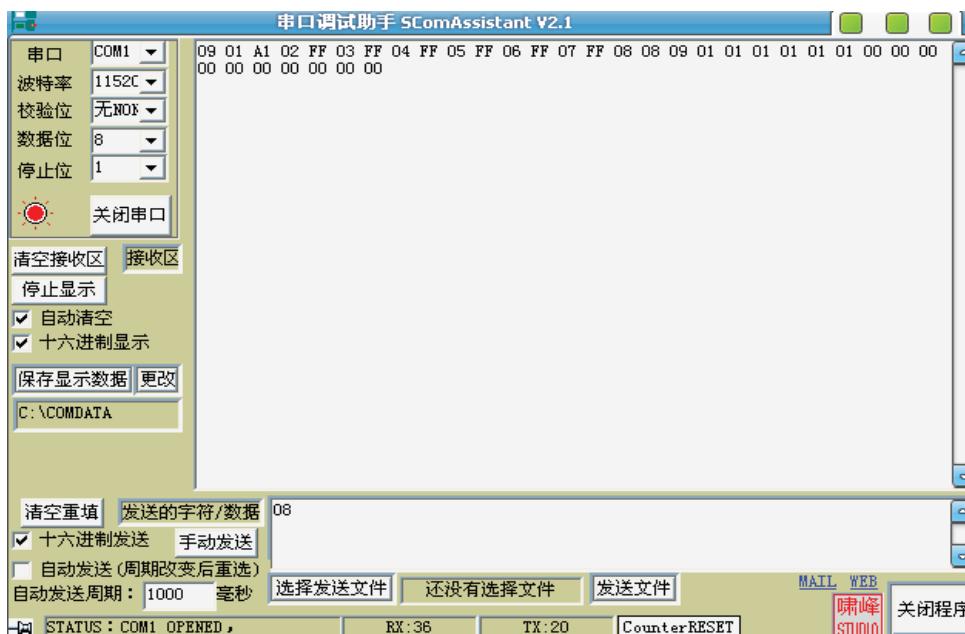


则发送:19 A2 FF FF FF FF FF FF 08 01 01 01 01 01 01 01 00 00 00 00 00 00 00 00 00 00 00  
00 00

复位后在串口调试助手中以十六进制输入 19 A2 FF FF FF FF FF FF FF 08 01 01 01 01  
01 01 00 00 00 00 00 00 00 00 00 00 00 00，现象如下：



再访问块 8，读出来的数据如下：



可以看出我们读出的数据与写入的数据是一样的，所以写入成功。

## 4、修改密码



OF A3 Key0 Key1 Key2 Key3 Key4 Key5 Kn New0 New1 New2 New3 New4  
New5 .

例:0xA3 为修改密码标志。

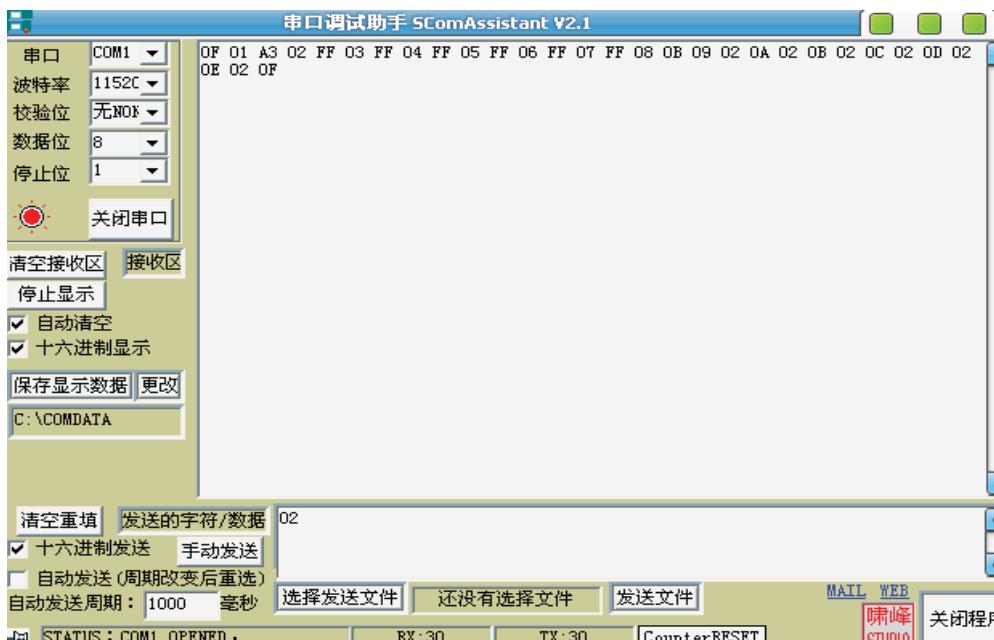
该卡原密码 A 为 16 进制:FF FF FF FF FF FF 对应 Key0 Key1 Key2 Key3 Key4  
Key5;

要修改的密码块数为第 11 块 即 Kn=11 (0X0B); (密码保存在扇区尾块, 分别  
为 7,11,15,19.....)

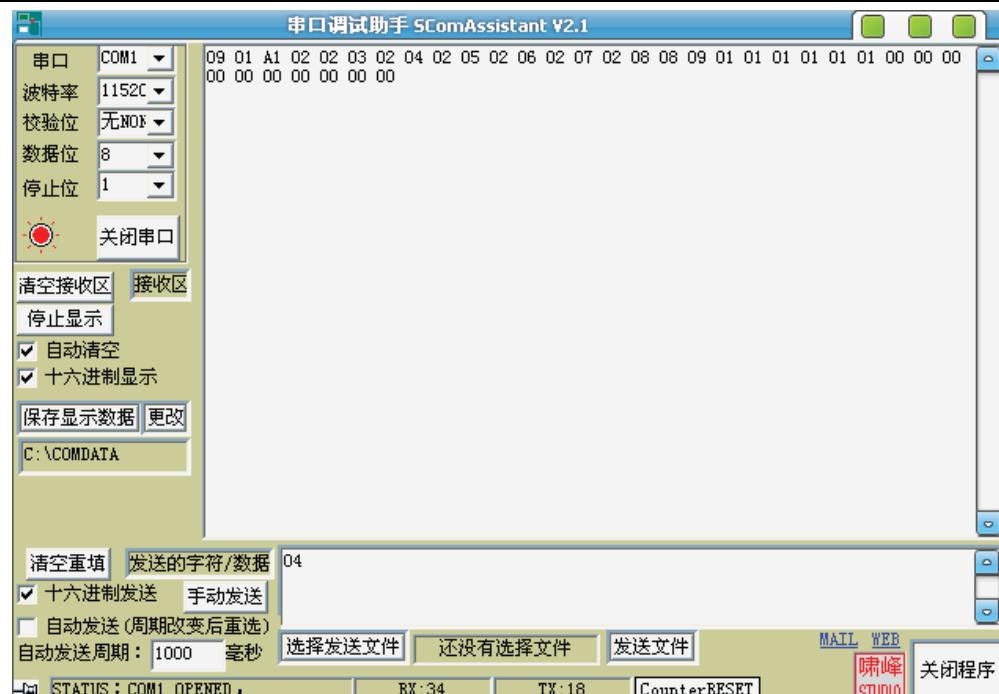
要修改成的密码为 02 02 02 02 02 02 对应 New0 New1 New2 New3 New4  
New5

则发送:OF A3 FF FF FF FF FF FF 0B 02 02 02 02 02.

复位后在串口调试助手中以十六进制输入 OF A3 FF FF FF FF FF FF 0B 02 02 02 02  
02 02, 现象如下:



以新密码 02 02 02 02 02 02 访问块 8, 现象如下:



可见密码修改成功。

实验讲解完毕，野火祝大家学习愉快。



## USB 读取 MicroSD 卡（模拟 U 盘）实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

实验描述: 这是一个 **USB Mass Storage** 实验, 用 **USB** 线连接 **PC** 机与开发板, 在电脑上就可以像操作普通 **U 盘** 那样来操作开发板中的 **MicroSD** 卡, 并在超级终端中打印出相应的调试信息。在这里野火用的 **MicroSD** 卡的容量是 **1G**。

硬件连接: PE3-USB-MODE (PE3 为普通 I/O)

PA11-USBDM(D-)

PA12-USBDP(D+)

ST 库 : [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/misc.c](#)

[FWlib/stm32f10x\\_dma.c](#)

[FWlib/stm32f10x\\_sdio.c](#)

[FWlib/stm32f10x\\_flash.c](#)



USB 库 : `usb_core.c`

`usb_init.c`

`usb_mem.c`

`usb_regs.c`

`usb_bot.c`

`usb_scsi.c`

`usb_data.c`

`usb_desc.c`

`usb_endp.c`

用户文件: `USER/main.c`

`USER/stm32f10x_it.c`

`USER/usart1.c`

`USER/sdcard.c`

`USER/usb_istr.c`

`USER/usb_prop.c`

`USER/usb_pwr.c`

`USER/hw_config.c`

`USER/memory.c`

其中 `USER` 文件夹下红色标注的 5 个 `c` 文件也是 `USB` 库文件，只因为我们需要修改它们才没有把它们放在 `USBLIB` 目录下。

`stm32f10x_it.c`: 该文件中包含 `USB` 中断服务程序，由于 `USB` 中断有很多情况，这里的中断服务程序只是调用 `usb_Istr.c` 文件中的 `USB_Istr` 函数，由 `USB_Istr` 函数再做轮询处理。

`usb_istr.c`: 该文件中只有一个函数，即 `USB` 中断的 `USB_Istr` 函数，该函数对各类引起 `USB` 中断的事件作轮询处理。



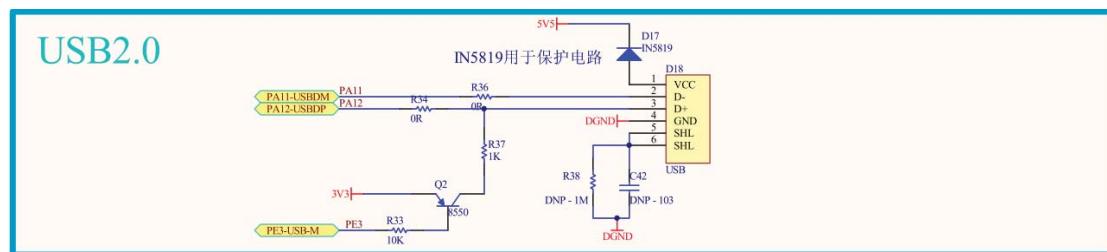
**usb\_prop.c:** 该文件用于实现相关设备的 USB 协议，例如初始化、SETUP 包、IN 包、OUT 包等等。

**usb\_pwr.c:** 该文件中包含处理上电、调电、挂起和恢复事件的函数。

**memory.c:** 该文件中包含 USB 读写 SD 卡的函数。

**hw\_config.c:** 该文件中包含系统配置的函数。

### 野火 STM32 中 USB 硬件原理图：



### USB 简介->

USB 模块为 PC 主机和微控制器所实现的功能之间提供了符合 USB 规范的通信连接。PC 主机和微控制器之间的数据传输是通过共享一专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512 字节缓冲区，最多可用于 16 个单向或 8 个双向端点。

USB 模块同 PC 主机通信，根据 USB 规范实现实令牌分组的检测，数据发送/接收的处理，和握手分组的处理。整个传输的格式由硬件完成，其中包括 CRC 的生成和校验。每个端点都有一个缓冲区描述块，描述该端点使用的缓冲区地址、大小和需要传输的字节数。当 USB 模块识别出一个有效的功能/端点的令牌分组时，(如果需要传输数据并且端点已配置)随之发生相关的数据传输。

USB 模块通过一个内部的 16 位寄存器实现端口与专用缓冲区的数据交换。在所有的数据传输完成后，如果需要，则根据传输的方向，发送或接收适当的握手分组。在



数据传输结束时，USB 模块将触发与端点相关的中断，通过读状态寄存器和/或者利用不同的中断处理程序，微控制器可以确定：

- 哪个端点需要得到服务
- 产生如位填充、格式、CRC、协议、缺失 ACK、缓冲区溢出/缓冲区未满等错误时，正在进行的是哪种类型的传输。

有关更多 USB 的介绍请参考《STM32 参考手册中文》。USB 是一个很复杂的设备，要想全面的学习 USB，光靠做完这个实验和看《STM32 参考手册中文》是远远不够的，还要大量阅读 ST 的官方 USB 文档。还有网上有本关于 USB 方面的书《圈圈教你学 USB》很不错，大家可以去买来研究研究。总之，一句话，USB 耐学，^\_^。

### 实验讲解->

- ✧ 友情提示：USB 是一个比较复杂的知识点，单单 USB 就可以出一本书，在这里
- ✧ 野火不可能为大家讲解地非常详细，但这个文档可以为大家扫清代码阅读的障碍。
- ✧ 要想更深入的学习 USB，仍需大家的努力，网上有本叫《圈圈教你学 USB》的书就将得非常好，有兴趣的朋友可以买来看看，虽然有电子版，但大伙还是买本书支持下圈圈，毕竟圈圈写书也不容易呀^\_^.....

在配置好我们需要用到的库文件之后，我们就从 main 函数开始分析，关于如何添加库文件请参考前面的教程，这里不再详述。

```
1.  /*
2.   * 函数名: main
3.   * 描述 : 无
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.    /* 串口 1 配置 15200-8-N-1 */
13.    USART1_Config();
14.
15.    /* SD 卡中断配置，优先级最高 */
16.    NVIC_Configuration();
17.
18.    /* 等待 SD 卡底层初始化成功 */
19.    while ( SD_USER_Init() != SD_OK );
20.
21.
22.    /* 获取 SD 卡容量信息，并在串口打印出来 */
23.    Get_Medium_Characteristics();
24.
```



```
25.     /* 配置 USB 时钟为 48M */
26.     Set_USBClock();
27.
28.     /* 配置 USB 中断 */
29.     USB_Interrupts_Config();
30.
31.     /* 配置 USB D+ 线为全速模式 */
32.     USB_Cable_Config (ENABLE);
33.
34.     /* USB 系统初始化 */
35.     USB_Init();
36.
37.     /* 等待 USB 中断到来, 在中断函数中进行数据的传输 */
38.     while (1)
39.     {
40.     }
41. }
```

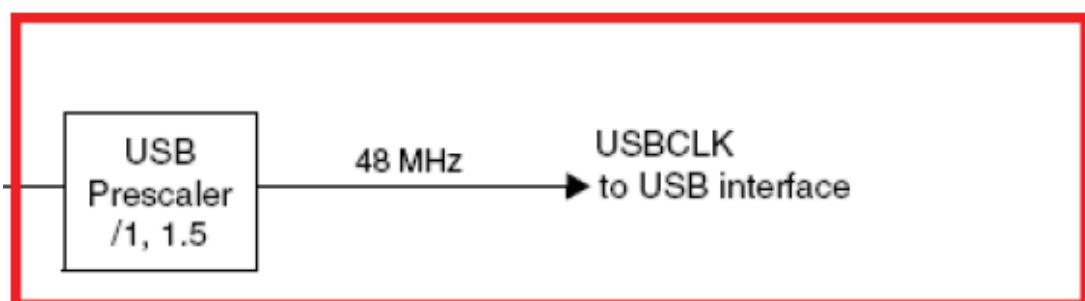
系统库函数 `SystemInit()`; 将系统时钟配置为 **72MHZ**, `USART1_Config()`; 初始化等下要用到的串口 **1**, 用于打印 MicroSD 卡的容量信息。有关这两个函数的详细介绍请参考前面的教程, 这里不再详述。

`NVIC_Configuration()`; 用于配置 MicroSD 卡的中断优先级, 本实验中配置为最高优先。

`while ( SD_USER_Init() != SD_OK );` 用于等待 MicroSD 卡底层硬件初始化成功, `SD_USER_Init()` 在 `sdcard.c` 中实现。只有这个初始化成功了, 接下来才能通过 **USB** 的方式来访问, 所以才采用 `while ( );` 的写法, 如果初始化不成功的话则一直等待, 直到初始化成功为止。`SD_USER_Init()` 在 `main.c` 中实现。

`Get_Medium_Characteristics()`; 用来获取 MicroSD 卡的容量信息, 并通过串口 **1** 在超级终端中打印出来。`Get_Medium_Characteristics()`; 也在 `main.c` 中实现。

`Set_USBClock()`; 将 **USB** 的时钟设置为 **48MHZ**, 其实 **USB** 的时钟必须设置为 **48MHZ**, 如下图所示, 截图来自《STM32 参考手册中文》第 47 页。`Set_USBClock()`; 在 `hw_config.c` 中实现。





```
1.  /*
2.   * 函数名: Set_USBClock
3.   * 描述 : 配置 USB 时钟(48M)
4.   * 输入 : 无
5.   * 输出 : 无
6.  */
7. void Set_USBClock(void)
8. {
9.     /* USBCLK = PLLCLK */
10.    RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);
11.
12.    /* Enable USB clock */
13.    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
14. }
```

`USB Interrupts Config()`; 用于配置 USB 的中断优先级, 本实验中 USB 的中断优先级次于 MicroSD 卡的中断优先级。`USB Interrupts Config()`; 在 `hw_config.c` 中实现。

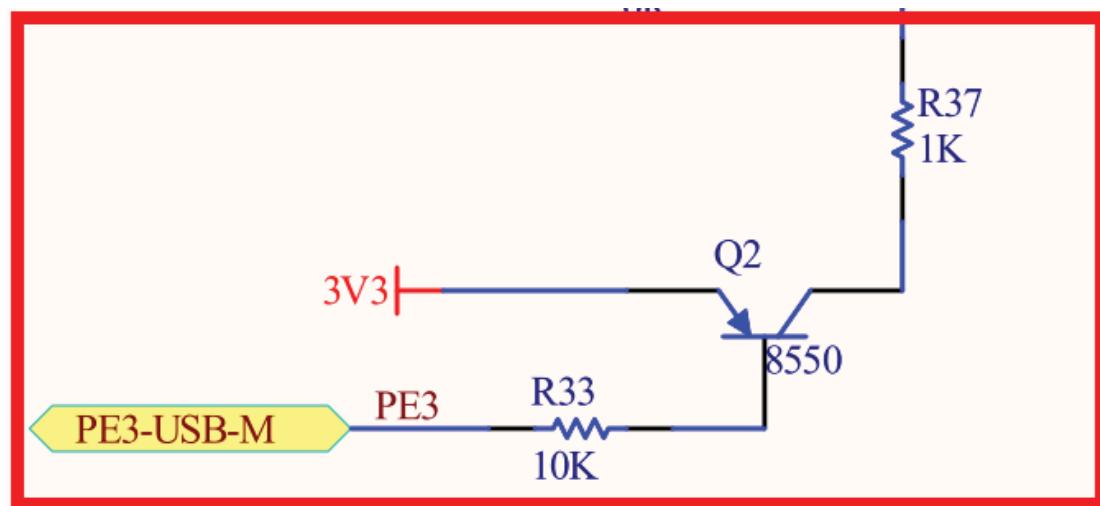
`USB Cable Config (ENABLE)`; 用于配置 USB D+ 这跟数据线为全速模式, 即 D+这根数据线接一个上拉电阻。当 D-数据线接上拉电阻时则工作于低速模式。

`USB Cable Config ()`; 在 `hw_config.c` 中实现:

```
1. /*
2.  * 函数名: USB_Cable_Config
3.  * 描述 : Software Connection/Disconnection of USB Cable
4.  * 输入 : -NewState: new state
5.  * 输出 : 无
6.  */
7. void USB_Cable_Config (FunctionalState NewState)
8. {
9.     GPIO_InitTypeDef GPIO_InitStructure;
10.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
11.
12.    /* PE3 输出 0 时 D+ 接上拉电阻工作于全速模式 */
13.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
14.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;      /* 开漏输出 */
16.    GPIO_Init(GPIOE, &GPIO_InitStructure);
17.
18.    if (NewState != DISABLE)
19.    {
20.        GPIO_ResetBits(GPIOE, GPIO_Pin_3);                /* USB 全速模式 */
21.    }
22.    else
23.    {
24.        GPIO_SetBits(GPIOE, GPIO_Pin_3);                  /* 普通模式 */
25.    }
26. }
```



在硬件设计上我们通过 I/O 控制一个三极管的通断来决定 D+这跟数据线是否上拉:



`USB_Init()`; 用于系统初始化, `USB_Init()`; 在 `usb_init.c` 中实现:

```
1.  ****
2.  * Function Name   : USB_Init
3.  * Description      : USB system initialization
4.  * Input            : None.
5.  * Output           : None.
6.  * Return           : None.
7.  ****
8.  void USB_Init(void)
9.  {
10.    pInformation = &Device_Info;
11.    pInformation->ControlState = 2;
12.    pProperty = &Device_Property;
13.    pUser_Standard_Requests = &User_Standard_Requests;
14.    /* Initialize devices one by one */
15.    pProperty->Init();
16. }
```

如果我们在调试程序时, 不成功的话, 一般都是这个函数出了问题, 而一般的问题又会是硬件的问题, 我当初调试的时候就郁闷了很久。



最后让程序在一个 `while (1) { }` 无限循环总等待 USB 中断的到来，然后进行中断服务程序的处理。USB 中断函数 `void USB_Istr(void)` 在 `usb_istr.c` 中实现：



```
70. #ifdef SUSP_CALLBACK
71.     SUSP_Callback();
72. #endif
73. }
74. #endif
75. /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
76. #if (IMR_MSK & ISTR_SOF)
77.     if (wIstr & ISTR_SOF & wInterrupt_Mask)
78.     {
79.         SetISTR((u16)CLR_SOF);
80.         bIntPackSOF++;
81.
82. #ifdef SOF_CALLBACK
83.     SOF_Callback();
84. #endif
85. }
86. #endif
87. /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
88. #if (IMR_MSK & ISTR_ESOF)
89.     if (wIstr & ISTR_ESOF & wInterrupt_Mask)
90.     {
91.         SetISTR((u16)CLR_ESOF);
92.         /* resume handling timing is made with ESOFs */
93.         Resume(RESUME_ESOF); /* request without change of the machine state */
94.
95. #ifdef ESOF_CALLBACK
96.     ESOF_Callback();
97. #endif
98. }
99. #endif
100.    /*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
101. #if (IMR_MSK & ISTR_CTR)
102.     if (wIstr & ISTR_CTR & wInterrupt_Mask)
103.     {
104.         /* servicing of the endpoint correct transfer interrupt */
105.         /* clear of the CTR flag into the sub */
106.         CTR_LP();
107.         #ifdef CTR_CALLBACK
108.             CTR_Callback();
109.         #endif
110.     }
111. #endif
112. } /* USB_Istr */
```

实验到了这里，我们已经可以通过 **USB** 来操作我们开发板上的 **MicroSD** 卡了。有关更多实验的细节请大家阅读源码，^\_^。



### 实验现象->

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 插上 MicroSD 卡, 插上方口的 USB 线, 将编译好的程序下载到开发板, 如果程序运行成功, 则可在电脑上看到开发板上的 U 盘(我用的是 1G 的卡), 如下所示:



打开 U 盘, 可看到里面的文件。还可以进行新建、复制、粘贴、格式化等操作。与操作我们的普通 U 盘没什么区别。



实验讲解完毕, 野火祝大家学习愉快。



# 计算器 (Calculator) 实验

作者	山外メ雲ジ
E-Mail	<a href="mailto:minisong@foxmail.com">minisong@foxmail.com</a>
QQ	860317732
博客	<a href="http://sosong.blog.chinaunix.net">sosong.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

## 目录

计算器 (Calculator) 实验 .....	1
版权声明 .....	2
实验描述 .....	2
用户文件 .....	2
功能简介 .....	3
操作界面 .....	3
设计思路、流程图及代码 .....	3
main 函数 .....	4
Calculate 函数 .....	10
移植 .....	14



## 版权声明

本程序 **COUNT** 模块部分（包括 **main** 函数）由本人独立完成，如有雷同，必属抄袭。

请尊重他人劳动成品，采用本程序源代码时请注明作者信息。

## 实验描述

将 MicroSD 卡(以文件系统 FATFS 访问)里面的 BMP 格式文件显示到液晶屏幕上作为主界面，通过触摸 触摸屏上的按键来控制计算器，程序后台运行计算结果并显示到屏幕上。

注意：本程序运行，还需要把 “.\资源备份\CAL.bmp” 放入 SD 卡根目录。

## 用户文件

### 核心模块文件

<b>USER\ COUNT\COUNT.H</b>	数据类型的定义、函数的声明
<b>USER\ COUNT\ COUNT_CFG.H</b>	模块的配置文件（独立出来是为了增强移植性）
<b>USER\ COUNT\COUNT.C</b>	函数的定义

### 其他需要用到的文件

<b>USER/main.c</b>	主函数
<b>USER/stm32f10x_it.c</b>	自带库
<b>USER/lcd.c</b>	LCD 液晶显示
<b>USER/Touch.c</b>	屏幕触摸
<b>USER/ Sd_bmp</b>	读取 SD 卡里 BMP 图片函数，用于显示主界面
<b>USER/usart1.c</b>	串口发送



USER/SysTick.c	延时
.....	

## 功能简介

野火 STM32 开发板计算器模块，支持负数运算、小数运算、括号运算、复合运算、加减乘除连续运算……

由于模块的运算是不涉及硬件的，模块从一开始设计就考虑到移植性的问题，具有较好的移植性。可以通过修改仅仅修改 COUNT\_CFG.H 就移植到其他单片机上（单片机需要支持递归，部分单片机需要特殊设置才能支持递归的，如 51。移植时就要特殊修改）。

## 操作界面



“D” , Delete, 表示退格，按一次删除式子最后一个输入符号。

“C” , Clean, 表示清空，按一次清空式子。

LCD 的第一行显示运算式，第二行显示运算结果。

## 设计思路、流程图及代码

设计思路很简单，主要是输入式子，把输入的式子以数组的形式存放，当按下等于号就调用 calculate 计算结果，再用 float2str，把计算结果转换为字符串再显示出来。



为了方便操作输入式子及计算式子，模块定义了两个结构体：

```
1. //*****定义式子计算结构体，调用 calculate 计算结果用到*****
2. struct num_point{
3.     float      result;      //计算结果
4.     song_u8    *point;      //指向式子中计算到的位置
5. };
6.
7. //*****定义式子数组，便于对式子添加删除操作*****
8. struct str_point{
9.     song_u8    str[SN];    //输入的计算式子
10.    song_u8   strn;       //式子长度
11. };
```

对式子的添加元素、删除元素、清空操作函数：

```
1. //对计算式子操作
2. #define LastSP(sp) (sp.strn>0 ? sp.str[sp.strn-1]:0)           //查询式子最后一位
3. #define LastSPp(sp) (sp->strn>0 ? sp->str[sp->strn-1]:0)
4. extern void    DelSP    (struct str_point *sp);                  //删除式子最后一位
5. extern void    ClearSP  (struct str_point *sp);                  //清空式子
6. extern song_u8 AddCheckSP(struct str_point *sp,song_u8 key);
7.             //带简单检测功能的添加字符到式子尾部，添加字符有效返回 1，否则返回 0
```

计算式子及将结果转换为字符串：

```
1. extern struct  num_point  calculate  (song_u8 *str,song_u8 N);
2.                           //计算 式子（检查最后一位是否为空）
3. extern void        float2str   (float num,song_u8 *str); //实型转字符串
4. extern void        float2stre  (float num,song_u8 *str); //实型转字符串，科学记数法表示
5.
```

## main 函数

这个程序有 3 个 main 函数，通过宏定义 MODE 来选择编译哪个 main 函数的。

```
1. #define MODE    1 //这里有 3 个 main 函数，即 3 个模式
2.                     //模式 1：调用 CountMain，是计算器模块自带的一个测试程序
3.                     //模式 2，是 CountMain 的展开，可以方便嵌入自己的程序
4.                     //模式 3，是测量触摸屏按键用的，测试用而已
```

使用模式 1 需要对 COUNT\_CFG.H 进行配置输入输出函数：

```
1. //-----以下是测试程序 CountMain 用到-----//
```



```
2.  /*****定义显示、清屏函数*****  
3.  * 这里定义的是液晶的函数，当然也可以改成是串口之类的。  
4.  * 例如串口的话：不需要清屏函数，可以把 CLEAN_* 定义为空。  
5.  *           其他的函数参考 COUT 编写串口输出  
6.  *****/  
7.  #include "lcd.h"  
8.  #define CLEAN_S          LCD_Str_R(26,304,"  
   ",36,0x0000,0xffff)           //清屏（输入式子的位置）  
9.  #define CLEAN_R          LCD_Str_R(59,304,(u8 *)"           ",14,0x0000,0xf  
fff)  
10. #define ERROR           LCD_Str_R(59,304,(u8 *)"           error",14,0x0000,0xf  
fff)  
11. #define PRINT_S(x)       LCD_Str_R(26,304,(u8 *)x,36,0x0000,0xffff)  
                           //显示输出结果  
12. #define PRINT_R(x)       LCD_Str_R(59,304,(u8 *)x,14,0x0000,0xffff)  
                           //显示输入式子  
13.  
14. /*****定义按键函数*****  
15. * 需要用户编写的扫描按键函数，传递进去的是指针变量  
16. * 有按键按下就返回非 0  
17. * x 是按键变量，取值： 0 1 2 3 4 5 6 7 8 9 . = + - * / ( ) D C  
18. * D 表示退格 del, C 表示清空 clean  
19. * 本来打算用枚举表示的，但反而显示符号没那么直观，就放弃了。  
20. * 下面给出的是例子而已，可以是改成是按键、触摸、串口接收之类  
21. *****/  
22. #include "includes.h"  
23. #define SCAN_KEY(x)      ScanTouch(&x)  
24.  
25. /*****定义延时函数*****  
26. #include "systick.h"  
27. #define DELAY           delay_ms(100) //延时函数，避免触摸屏按的速度太快。  
28. //#define DELAY(x)      delay_ms(x) //不用这个，方便用户可以自己在这个 CFG 文件中修改延时  
29.  
30. //-----以上的是测试程序 CountMain 用到-----//
```



移植时如果不用到 **CountMain 函数** 就不需要修改上面的配置文件。然后 **main** 函数就可以非常简洁了。**CountMain** 函数里面的变量 定义为静态变量，即下次调用时仍然保留原来数据。

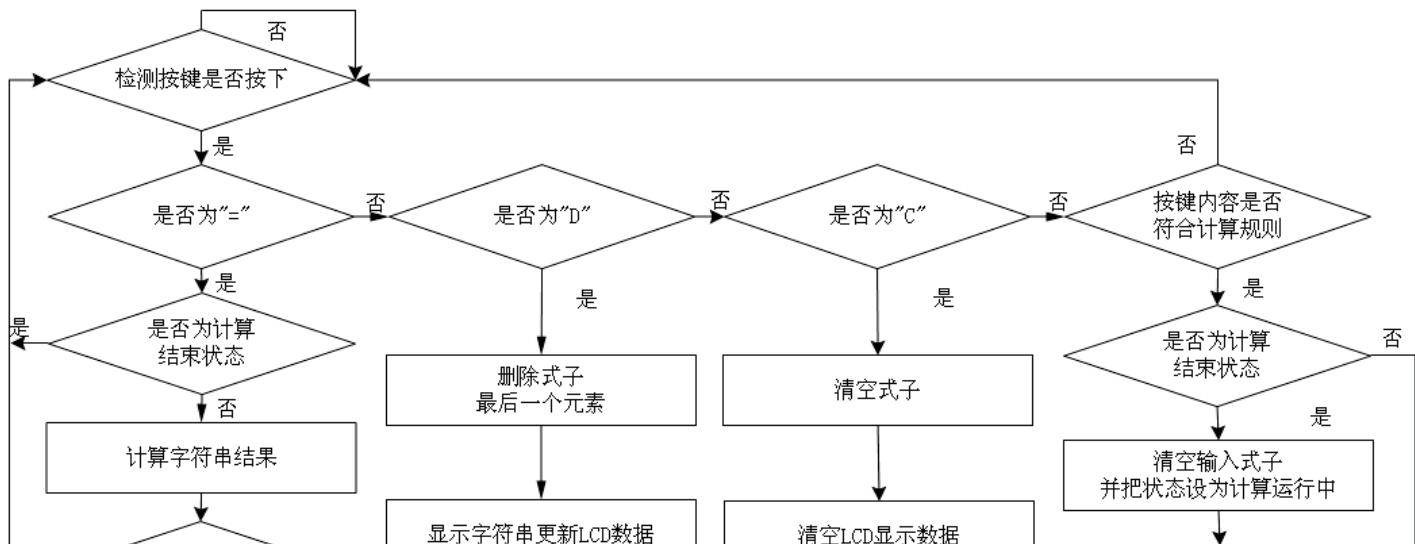
```
1. int main(void)
2. {
3.     Init();                                // 初始化
4.     LCD_Show_8x16_String(2, 29, 0, "10");   // 显示 左上角 10 进制
5.
6.     while (1)
7.     {
8.         CountMain();                      // 计算器任务
9.         // 其他用户需要的任务……
10.    } // while
11. } // main
```

提示：加入其他用户需要的任务时，可以修改 **COUNT\_CFG.H** 里的延时宏定义，适当减少延时提高灵敏度。

**CountMain 函数** 的执行流程跟模式 2 是一样的，独立 **CountMain 函数** 出来是为了用户编写程序时简单一些，代码不用太长。上面的代码是不是很短啊？呵呵，我把计算的执行流程隐藏掉了，方便用户添加自己的其他任务。用户想在计算器的基础上实现其他功能，建议直接在模式 1 中添加，而不像模式 2 那样长长的代码，让人看到都头晕 😊

具体的执行流程如何？这里就直接讲模式 2，模式 2 其实就是 **CountMain 函数** 的展开：

先来流程图，流程图比较容易了解思路：





```
1. #define N 10          //N 是 N 进制, 可以改成其他进制, 不过输出还是科学计数法, 即 10 进制的
2. int main(void)
3. {
4.     char key=0;           //按下的键
5.     u8 Result[RN];        //计算结果字符串, 把计算结果转换为字符串
6.     struct num_point tmp; //计算结果
7.     struct str_point sp;  //计算式子
8.     char CountStatus=calnot; //计算状态
9.
10.    Init();              //初始化
11.    LCD_Show_8x16_String(2,29,0,"10");
12.
13.    ClearSP(&sp);
14.
15.    while (1)
16.    {
17.        if(SCAN_KEY(key))      /*如果触笔有按下*/
18.        {
19.            printf("key=%c ",key );
20.            switch(key)
21.            {
22.                case '=': //-----计算式子-----
23.                    if(CountStatus==caled) continue;
24.                    CountStatus=caled;           //计算完毕
25.            }
```



```
26.                         tmp=calculate(sp.str,N);           //计算字符串结果
27.                         if(tmp.point==0)
28.                         {
29.                             ERROR;                      //输入式子有误
30.                             break;
31.                         }
32.                         CLEAN_R;                   //清屏
33.                         float2stre(tmp.result,Result); //把计算结果转为字符串
34.                         PRINT_R(Result);           //显示计算结果
35.                         break;
36.             case 'C':    //-----清空式子-----
37.                         CountStatus=caling;
38.                         ClearSP(&sp);
39.                         CLEAN_R;                   //清屏
40.                         CLEAN_S;
41.                         break;
42.             case 'D':    //-----退格-----
43.                         CountStatus=caling;
44.                         DelSP(&sp);
45.                         CLEAN_R;                   //清屏
46.                         CLEAN_S;
47.                         PRINT_S(sp.str);          //更新屏幕字符串
48.                         break;
49.             default:    //-----输入式子-----
50.                                     // (剩下的就是运算表达式)
51.                         if(CountStatus==caled)        //如果计算完成后，就清空
52.                         {
53.                             CountStatus=caling;
54.                             ClearSP(&sp);
55.                             CLEAN_R;                   //清屏
56.                             CLEAN_S;
57.                         }
58.                         if(AddCheckSP(&sp,key))      //输入 key 有效
```



```
59.          {
60.              PRINT_S(sp.str);           //更新屏幕字符串
61.          }
62.      } //switch
63.      delay_ms(100);
64.  } //if
65. } //while
66. } //main
```

为什么有了模式 1，还给个模式 2 呢？主要是很多人都习惯把程序代码写在 `main` 函数上，容易看到程序的执行流程。而且很多人连静态变量的声明也不会（用 `static` 声明），直接讲 模式 1，他们不知道程序退出后为什么数据不会被销毁。

看到上面的程序，用法很简单，觉得挺简单吧？ 😊

还看不懂？ OMG，上帝与你同在 😊 那就来个精简版的代码吧，这样肯定能看明白……删掉液晶显示之类，留下对式子控制和计算：

```
1. while (1)
2. {
3.     if (SCAN_KEY(key))           /*如果触笔有按下*/
4.     {
5.         switch (key)
6.         {
7.             case '=':    //-----计算式子-----
8.                 tmp=calculate(sp.str,N);      //计算字符串结果
9.                 if (tmp.point==0)  break;      //计算有误时返回 0
10.                float2stre(tmp.result,Result); //把计算结果转为字符串
11.                PRINT_R(Result);           //显示计算结果
12.                break;
13.             case 'C':    //-----清空式子-----
14.                 ClearSP(&sp);
15.                 break;
16.             case 'D':    //-----退格-----
17.                 DelSP(&sp);
18.                 break;
19.             default:   //-----输入式子-----
```



```
20.                                     // (剩下的就是运算表达式)
21.         if (AddCheckSP(&sp, key))           //输入 key 有效
22.             PRINT_S(sp.str);            //更新屏幕字符串
23.         break;
24.     } //switch
25. } //if
26. } //while
```

代码简单了好多了吧？其实就是学如何调用我写好的库函数而已。熟悉了如何调用后，就开始讲核心的计算式子，这是这个计算器的关键代码，当然，看懂也是有难度的，做好心理准备哦。

## Calculate 函数

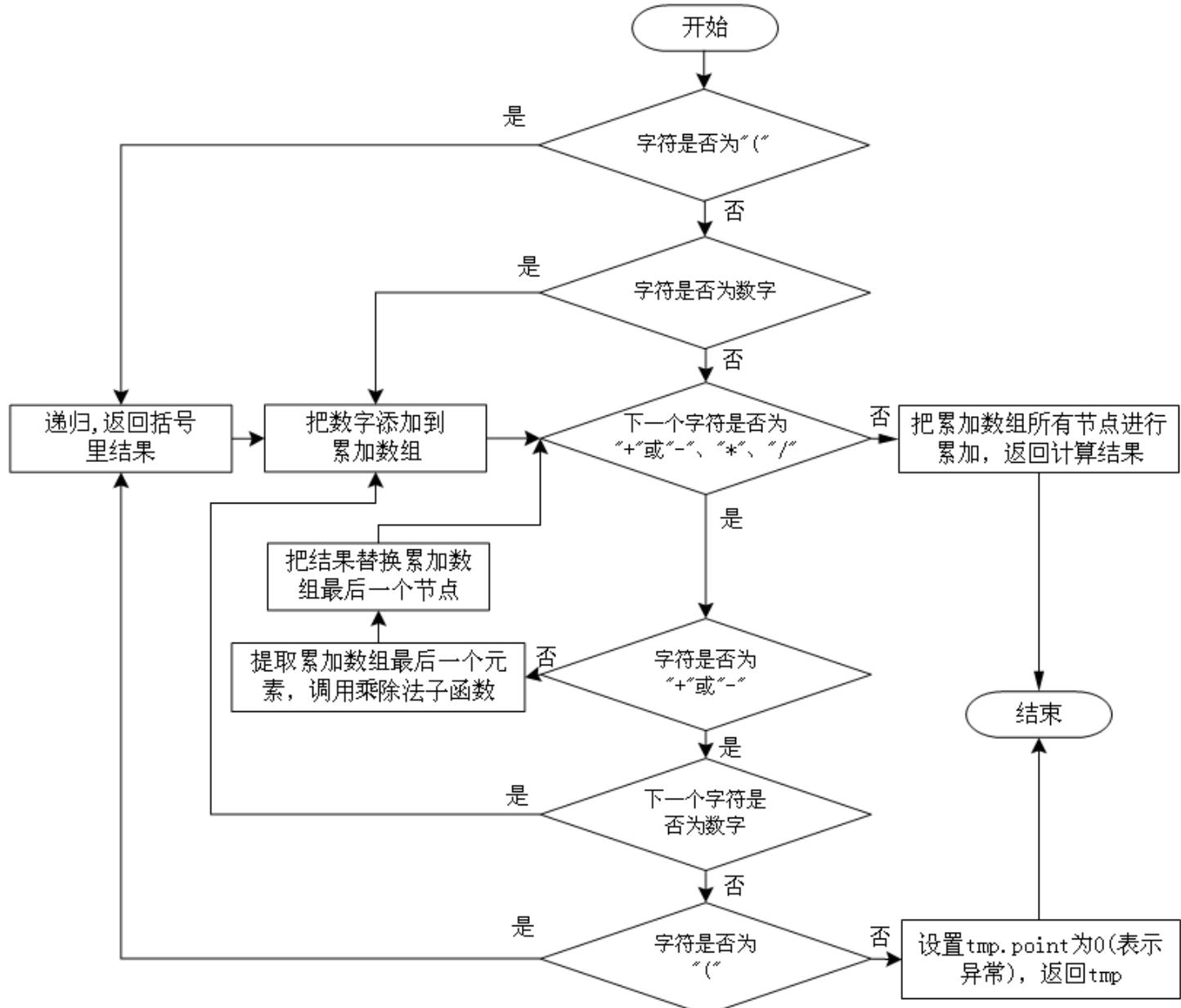
跳进 calculate 函数的函数体看下，就会发现其实 calculate 函数是包了外壳的 con\_add 函数，带验证结果是否最终运算结果或者运算错误。

con\_add 函数，累加的意思，提取式子的加减数存进累加数组；遇到乘除号就把最后添加到累加数组的元素提取出来，进行乘除运算后放回进去；遇到括号就把括号里面的式子当成一条独立的式子来调用 con\_add 函数计算，即递归，把返回值添加累加数组。遇到式子结尾或者其他错误计算表达时返回，结束运算。

里面用到几个内部函数：

```
1. static struct num_point str2num (song_u8 *str,song_u8 N);      //N 为 N 进制
2. static struct num_point str_division (float cal,song_u8 *str,song_u8 N); //乘除法
3. static struct num_point cal_bk (song_u8 *str,song_u8 N);
4.                                         // calculate bracket, 递归 con_add 函数来计算括号内式子
```

先看流程图吧（ 高手就直接跳到下面的源代码吧）：



举例说一下计算流程:

3 . 1 4 + 5 / 3 + 6 \* ( 8 + 1 / 3 )

调用 str2num 把

3.14 加入累加数组

调用 str2num 把 5

加入累加数组  
遇到除号, 提取 5 出来, 调用

str\_division 计算 5/3 替换累加数组里的 5  
调用 str2num 把 6 加入累加数组

遇到乘号, 提取 6 出来, 乘号后面是前括号, 调用 cal\_bk 计算括号里面的式子, 返回的结果与 6 相乘后, 替换累加数组里的 6

3.14	0	0	0	0
------	---	---	---	---

3.14	5	0	0	0
------	---	---	---	---

3.14	1.67	0	0	0
------	------	---	---	---

3.14	1.67	6	0	0
------	------	---	---	---

3.14	1.67	50	0	0
------	------	----	---	---



## 累加数组 ADDS

最后一步就是把累加数组的全部数据进行累加:  $3.14+1.66666+50$



原理不难吧? 呵呵, 那就看下源代码了。如果你仅仅想移植, 而不是研究其内部实现代码, 可以直接跳过源代码, 直接看移植部分。

con\_add 函数的源代码:

```
1. struct num_point con_add(song_u8 *str,song_u8 N)
2. {
3.     struct num_point tmp;           //返回结果
4.     float ADDS[ADDN];             //累加数组
5.     float *adds=ADDS;             //指示式子计算的到的位置
6.     song_u8 i;
7.
8.     for(i=0;i<ADDN;i++) *(ADDS+i)=0; //清空 ADDS
9.
10.    if(*str=='(')   //开头括号, 则调用递归函数求出括号内部的值
11.    {
12.        tmp = cal_bk(str,N);if(tmp.point==0) return tmp //递归
13.        *adds+=tmp.result;          //插入数据
14.        str=tmp.point;            //保存剩余字符串
15.    }
16.
17.    if(*str>='0' && *str<='9')           //数字或者加减号 (加减号就是正负数)
18.    {
19.        tmp = str2num(str,N);      //此处仅识别接着的数字, 不会产生空指针, 故不检错
20.        *adds+=tmp.result;        //插入数据
21.        str = tmp.point;
22.    } else *adds+=0;              //顶点为括号就前面添加 加 0 , 保证 p 指向一个已经累加的数
23.
24.    while( *str=='+' || *str=='-' || *str=='*' || *str=='/' )
25.                      //加减乘除, 前面已经检查过'('了
26.    {
27.        if( *str=='+' || *str=='-' ) //加减号之后只能是数字或者括号
```



```
28.         {
29.             if(* (str+1) == ' ') //括号则调用递归函数计算括号内的数据
30.             {
31.                 tmp=cal_bk(str+1,N);
32.                 if(tmp.point==0) return tmp; //递归
33.             }
34.             else if(* (str+1)>='0'&&* (str+1)<='9') tmp = str2num(str+1,N);
35.             //数字则直接保存
36.             else{CERROR("加减号后有误");tmp.point=0;return tmp;}
37.             //+-号后面不是数字或者括号，就报错： * / )号
38.             if(*str=='-') tmp.result = -tmp.result; //保存 +-数据
39.             *adds++=tmp.result;
40.             str=tmp.point;
41.         }
42.
43.         else if(*str=='*' || *str=='/') //乘除号之后只能是数字或者括号
44.         {
45.             if(* (str+1) == ' ') //括号则调用递归函数计算括号内的数据
46.             {
47.                 tmp = cal_bk(str+1,N); //递归
48.                 if(tmp.point==0)
49.                 {
50.                     CERROR("乘除号后调用括号函数有误");
51.                     tmp.point=NULL;return tmp;
52.                 }
53.                 if(*str=='*') *(adds-1)= *(adds-1) * tmp.result;
54.                 else *(adds-1)= *(adds-1) / tmp.result;
55.                 str=tmp.point;
56.             }else if(* (str+1)>='0'&&* (str+1)<='9')
57.             //数字则与原来数据进行乘除再保存，即乘除优先级比加减高
58.             {
59.                 tmp=str_division(*(adds-1),str,N);
60.                 if(tmp.point==0) return tmp;
61.                 str = tmp.point;
```



```
62.           * (adds-1)      =      tmp.result; //乘除法不需要移动，直接先计算乘除法
63.       }
64.   else
65.   {
66.       tmp.point=0;
67.       return tmp;
68.   }
69. }

70. //对于其他非法字符(例如: ) ), 就不用计算, 结束计算, 交给调用此函数的上一层函数处理
71. }
72. tmp.result=0;
73. for(i=0;i<ADDN;i++)tmp.result+=*(ADDS+i); //累加 ADDS
74. tmp.point = str;
75. return tmp;
76. }
```

有没有头晕。不管你晕不晕，反正我晕了……考虑的条件太多，调试了好久才写出来觉得稳定的。

## 移植



写在前面的话：请尊重他人劳动成品，采用本程序源代码时请注明作者信息。

本程序在设计时已经考虑到移植性的问题，已经测试过仅仅修改 COUNT\_CFG.H 就移植飞思卡尔的 MC9S12XS128 单片机上，对于 51 单片机，需要仅仅修改一下声明为递归函数也能快速移植。

本来程序设计时最初是采用链表方式实现的（可以动态分配内存，也可以作为教程来学链表），后来发现占用内存太大，不适合移植到其他小内存的单片机上，最终换成数组实现。配置文件为：COUNT\_CFG.H

COUNT\_CFG.H 源代码：

```
1.
2. #ifndef _COUNT_CFG_H_
3. #define _COUNT_CFG_H_
4.
5. #include "stm32f10x.h"
6.
7. define DUBUG //Debug 模式
```



```
8.  
9.  
10. /*****设置数据类型*****  
11. typedef unsigned char song_u8; //无符号型  
12. typedef unsigned short int song_u16;  
13. typedef unsigned int song_u32;  
14.  
15. typedef char song_s8; //有符号型  
16. typedef short int song_s16;  
17. typedef int song_s32;  
18. /*****设置数据类型*****  
19.  
20.  
21. /*****设置计算式子*****  
22. #define ADDN 20 //连加数组个数  
23. #define SN 80 //字符串长度（最多可以接受的计算式子字符）  
24. #define RN 30 //计算结果字符串长度  
25.  
26.  
27. /*****定义输出*****  
28. * 调用时可以不添加双引号: COUT ("xx"); COUT(xx);  
29. * 这两种表达都行，前者编辑器不识别中文字符，后者打印时多了双引号  
30. * 下面给的仅仅是例子，可以改成在液晶上显示、串口发送之类的  
31. *****  
32. #include "uart1.h"  
33. #define COUT(str) printf("\nout:#str\n") //定义正常输出  
34. #define CERROR(str) printf("\nerror:#str\n") //定义异常输出  
35.  
36. #ifdef DBUG  
37. #define CDBUG(str) printf("\ndebug:#str\n") //定义调试输出  
38. #else  
39. #define CDBUG(str) //空，即非 DBUG 模式不产生输出  
40. #endif
```



```
41.  
42. //-----以下的是测试程序 CountMain 用到-----//  
43.  
44. /*****定义显示、清屏函数*****  
45. * 这里定义的是液晶的函数，当然也可以改成是串口之类的。  
46. * 例如串口的话：不需要清屏函数，可以把 CLEAN_* 定义为空。  
47. * 其他的函数参考 COUT 编写串口输出  
48. *****  
49. #include "lcd.h"  
50. #define CLEAN_S LCD_Str_R(26,304,"  
      ",36,0x0000,0xffff) //清屏（输入式子的位置）  
51. #define CLEAN_R LCD_Str_R(59,304,(u8 *)"           ",14,0x0000,0xf  
      fff) //清屏（计算结果的位置）  
52. #define ERROR LCD_Str_R(59,304,(u8 *)"       error",14,0x0000,0xf  
      fff) //输出有误（前面加空格是为了清屏）  
53. #define PRINT_S(x) LCD_Str_R(26,304,(u8 *)x,36,0x0000,0xffff)  
      //显示输出结果  
54. #define PRINT_R(x) LCD_Str_R(59,304,(u8 *)x,14,0x0000,0xffff)  
      //显示输入式子  
55.  
56.  
57. /*****定义按键函数*****  
58. * 需要用户编写的扫描按键函数，传递进去的是指针变量  
59. * 有按键按下就返回非 0  
60. * x 是按键变量，取值： 0 1 2 3 4 5 6 7 8 9 . = + - * / ( ) D C  
61. * D 表示退格 del, C 表示清空 clean  
62. * 本来打算用枚举表示的，但反而显示符号没那么直观，就放弃了。  
63. * 下面给出的是例子而已，可以是改成是按键、触摸、串口接收之类  
64. *****  
65. #include "includes.h"  
66. #define SCAN_KEY(x) ScanTouch (&x)  
67.  
68. /*****定义延时函数*****  
69. #include "systick.h"  
70. #define DELAY delay_ms(100) //延时函数，避免触摸屏按的速度太快。  
71. /*#define DELAY(x) delay_ms(x); //不用这个，方便用户修改延时
```



```
72.  
73. //-----以上的是测试程序 CountMain 用到-----//  
74.  
75. #endif // _COUNT_CFG_H_
```

上面的源代码已经注解得很详细了，**CountMain** 函数用到的宏定义前面也已经说过  
了，这里就不多说了，主要是下面这 3 个宏定义：

```
1. #define ADDN          20 //连加数组个数  
2. #define SN           80 //字符串长度（最多可以接受的计算式子字符）  
3. #define SCAN_KEY(x) ScanTouch(&x)
```

前面两个是设置数组的，决定你输入式子的长度和最终能输入多少个加减号。

后面一个是 **CountMain** 函数的配置，注意有个取地址符号：“&”。即调用宏定义函数是  
传进去看上去不是指针，定义 **ScanTouch** 时传递进去的是指针哦。即

```
1. //声明函数  
2. void ScanTouch(char *x); //是指针哦  
3.  
4. //函数中调用  
5. char key;             //不是指针哦  
6. SCAN_KEY(key)
```

教程就讲得这里，是不是有点乱 😱 技术不过关，语言表达能力欠缺 😞 \_ 😞 .....

实验讲解完毕，野火嵌入式开发工作室祝大家学习愉快^\_^.....



## 以太网（ENC28J60）实验

作者	fire
E-Mail	<a href="mailto:firestm32@foxmail.com">firestm32@foxmail.com</a>
QQ	313303034
博客	<a href="http://firestm32.blog.chinaunix.net">firestm32.blog.chinaunix.net</a>
硬件平台	野火 STM32 开发板
库版本	ST3.0.0

**实验描述:** 在浏览器上创建一个 web 服务器，通过 web 里面的命令来控制开发板上的 LED 的亮灭。

应用->

1:在 PC 机的 DOS 界面输入: ping 192.168.1.15 , 看能否 ping 通。

2:在 IE 浏览器中输入: <http://192.168.1.15/123456> 则会出现一个网页，通过网页中的命令可以控制开发板中的 LED 的亮灭。

硬件连接: PB13 : ENC28J60-INT  
PA6-SPI1-MISO : ENC28J60-SO  
PA7-SPI1-MOSI : ENC28J60-SI  
PA5-SPI1-SCK : ENC28J60-SCK  
PA4-SPI1-NSS : ENC28J60-CS  
PE1 : ENC28J60-RST

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

用户文件: USER/main.c

USER/stm32f10x\_it.c

USER/led.c

USER/usart.c

USER/spi\_enc28j60.c

USER/enc28j60.c

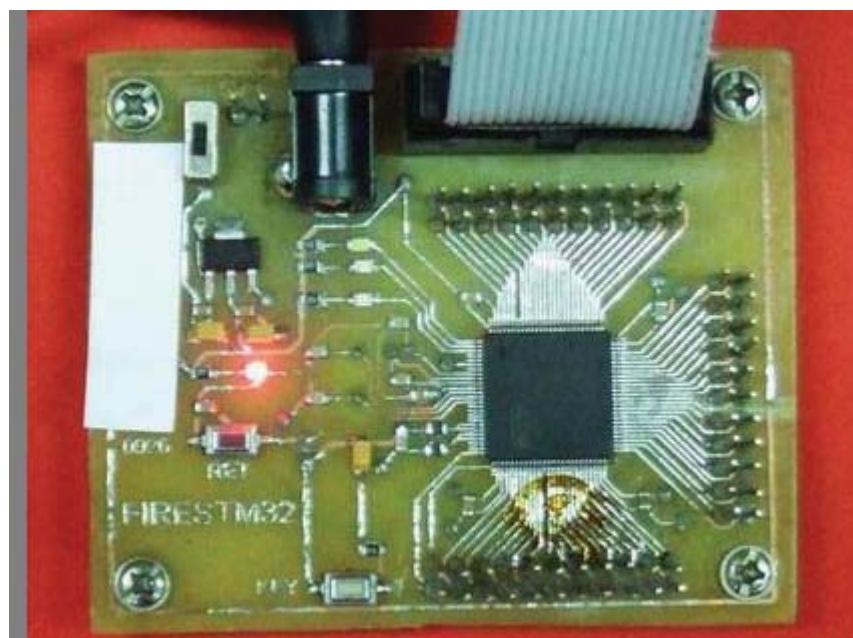
USER/ip\_arp\_udp\_tcp.c

USER/web\_server.c

注: 野火 **stm32** 开发板中板载的 **10M** 以太网 跟这里的接口是一样的。

纯手工野火 **stm32** 最小系统+**10M** 以太网+**MAX3232** 串口调试模块。

野火 **stm32** 最小系统

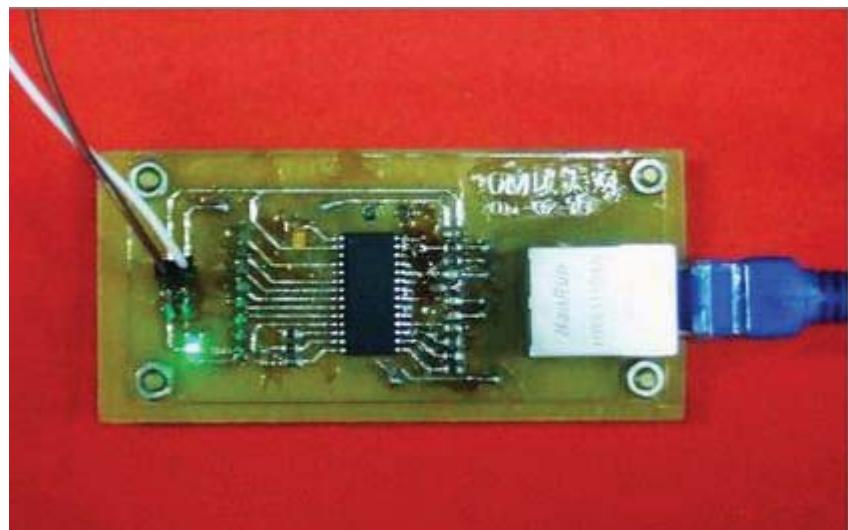




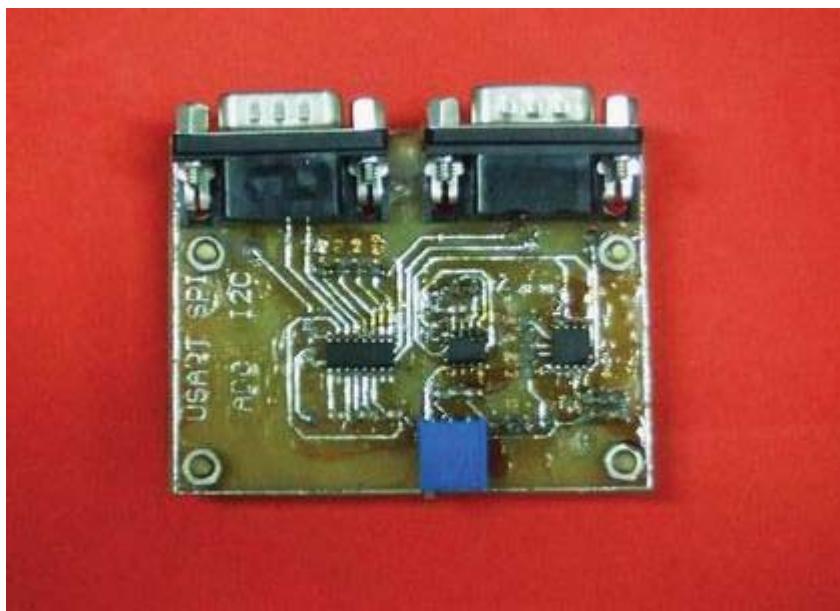
野火 stm32 开发板淘宝官方专卖: <http://firestm32.taobao.com>

---

### 10M 以太网 ENC28J60

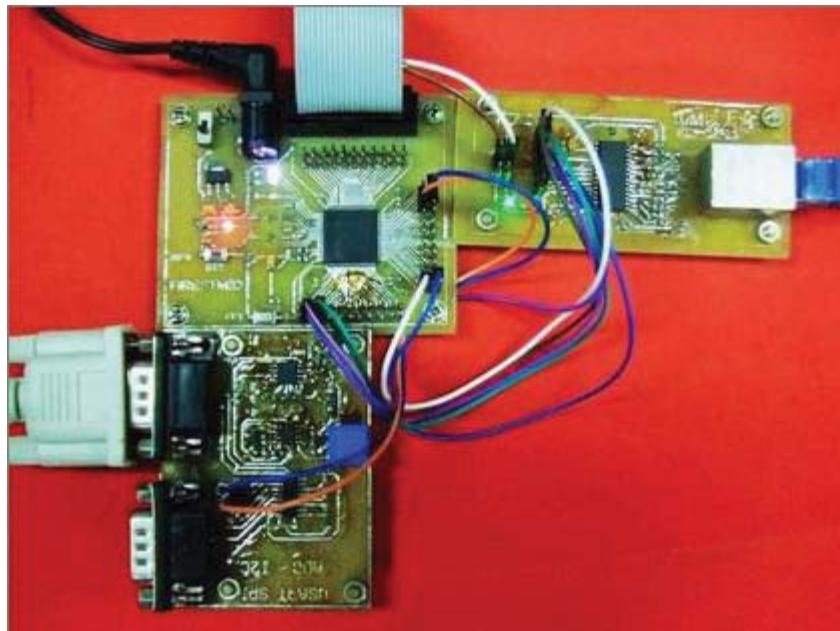


### MAX3232 (3.3V) 串口调试模块

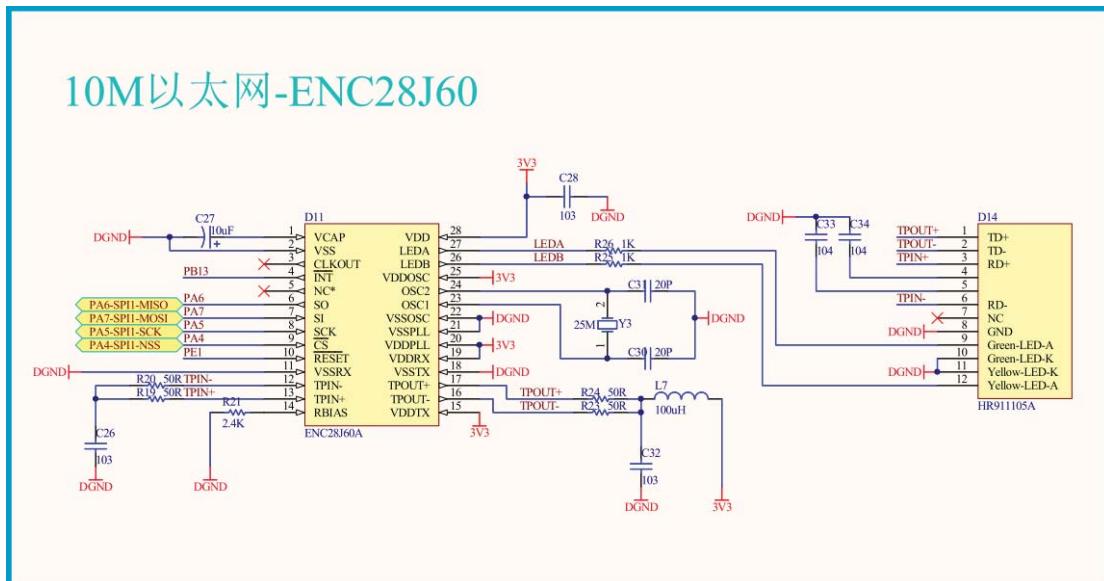




全家福



## 野火 STM32 开发板中 10M 以太网 ENC28J60 的硬件原理图：



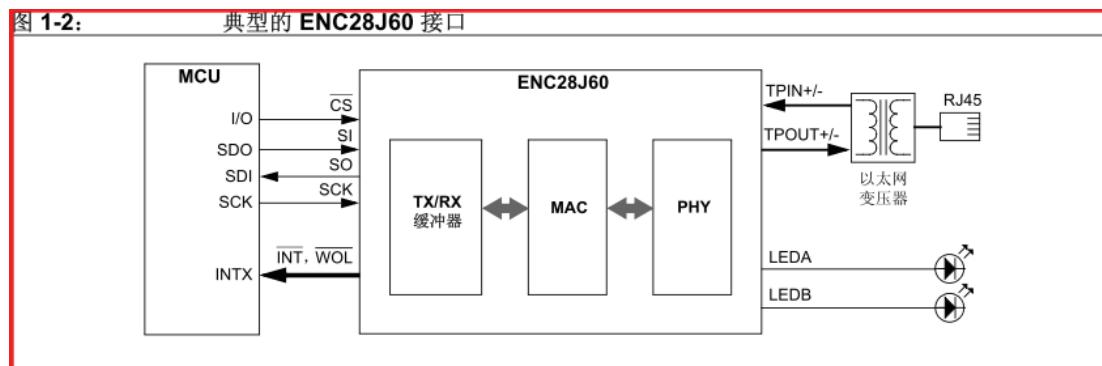
ENC28J60（以太网芯片）简介->

**ENC28J60** 是带有行业标准串行外设接口（Serial Peripheral Interface, SPI）的独立以太网控制器。它可作为任何配备有 SPI 的控制器的以太网接口。

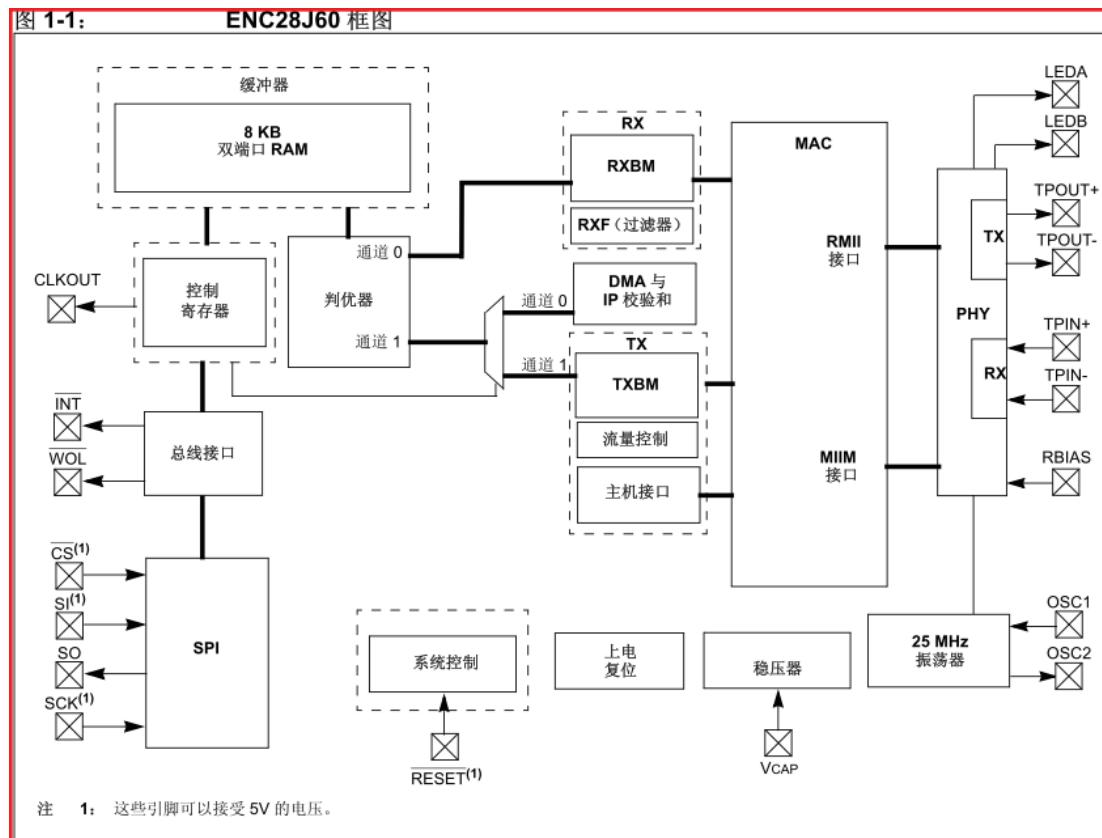


**ENC28J60** 符合 IEEE 802.3 的全部规范，采用了一系列包过滤机制以对传入数据包进行限制。它还提供了一个内部 DMA 模块，以实现快速数据吞吐和硬件支持的 IP 校验和计算。与主控制器的通信通过两个中断引脚和 SPI 实现，数据传输速率高达 10 Mb/s。两个专用的引脚用于连接 LED，进行网络活动状态指示。

图 1-1 所示为 ENC28J60 的简化框图。图 1-2 所示为使用该器件的典型应用电路。要将单片机连接到速率为 10 Mbps 的以太网，只需 ENC28J60、两个脉冲变压器和一些无源元件即可。



本开发板中用的网络变压器的型号为 **HR911105A**。





---

**ENC28J60** 由七个主要功能模块组成:

1. **SPI 接口**——充当主控制器和 ENC28J60 之间通信通道。
2. **控制寄存器**——用于控制和监视 ENC28J60。
3. **双端口 RAM 缓冲器**——用于接收和发送数据包。
4. **判优器**——当 DMA、发送和接收模块发出请求时对 RAM 缓冲器的访问进行控制。
5. **总线接口**——对通过 SPI 接收的数据和命令进行解析。
6. **MAC (Medium Access Control) 模块**——实现符合 IEEE 802.3 标准的 MAC 逻辑。
7. **PHY (物理层) 模块**——对双绞线上的模拟数据进行编码和译码。该器件还包括其他支持模块，诸如振荡器、片内稳压器、电平变换器（提供可以接受 5V 电压的 I/O 引脚）和系统控制逻辑。

实验讲解开始->

建议阅读程序的顺序为: `spi_enc28j60.c -> enc28j60.c -> ip_arp_udp_tcp.c -> web_server.c`。

`spi_enc28j60.c` : ENC28J60(以太网芯片) SPI 接口应用函数库。

`enc28j60.c` : Microchip ENC28J60 Ethernet Interface Driver。

`ip_arp_udp_tcp.c` : IP, Arp, UDP and TCP functions (这部分野火仍在学习)。

`web_server.c` : web 服务程序应用函数库。

其中 `enc28j60.c`、`ip_arp_udp_tcp.c` `web_server.c` 是从 **ATMEGA88 with ENC28J60** 移植过来的，这是 AVR 的一块 ATMEGA88 评估板，源文件基本上没有做修改。`spi_enc28j60.c` 是由我们用户实现的底层函数接口，还有我们修改了 `web_server.c` 这个文件中网页命令控制部分的服务程序。



在配置好需要用的库文件之后，下面我们从 **main** 函数开始讲解，有关库函数是如何添加的情参考前面的教程，这里不再详述。

```
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 LED */
13.    LED_GPIO_Config();
14.
15.    /* ENC28J60 SPI 接口初始化 */
16.    SPI_Enc28j60_Init();
17.
18.    /* ENC28J60 WEB 服务程序 */
19.    Web_Server();
20.
21.    //return 0;
22. }
```

在进入 **main** 函数代码段后，我们首先调用系统库函数 `SystemInit()`，将我们的系统时钟配置为 **72MHz**。

`LED_GPIO_Config()`；用于初始化 LED，因为我们我们在我们的 web 服务器中要控制的就是 LED，所以在这里要先把 LED 配置好，好让它接下来能工作。

`SPI_Enc28j60_Init()`；用于配置以太网芯片 ENC28J60 所用到的数据通信口 SPI2 和其他控制 I/O。这是我们用户在 `spi_enc28j60.c` 中实现的底层程序。

```
1.  /*
2.   * 函数名: SPI1_Init
3.   * 描述 : ENC28J60 SPI 接口初始化
4.   * 输入 : 无
5.   * 输出 : 无
6.   * 返回 : 无
7.   */
8. void SPI_Enc28j60_Init(void)
9. {
10.    GPIO_InitTypeDef GPIO_InitStructure;
11.    SPI_InitTypeDef SPI_InitStructure;
12.
13.    /* 使能 SPI1 时钟 */
14.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_SPI1,
15.                           ENABLE);
16.    /*
17.     * PA5-SPI1-SCK :ENC28J60_SCK
18.     * PA6-SPI1-MISO:ENC28J60_SO
```



```
19.     * PA7-SPI1-MOSI:ENC28J60_SI
20.     */
21.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7
22.     ;
23.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
24.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;           // 复用输出
25.     GPIO_Init(GPIOA, &GPIO_InitStructure);
26. 
27.     /* PA4-SPI1-NSS:ENC28J60_CS */ // 片选
28.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
29.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
30.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;           // 推免输出
31.     GPIO_Init(GPIOA, &GPIO_InitStructure);
32.     GPIO_SetBits(GPIOA, GPIO_Pin_4);
33. 
34.     /* PB13:ENC28J60_INT */ // 中断引脚没用到
35.     /* PE1:ENC28J60_RST*/ // 复位似乎不用也可以
36. 
37. 
38.     /* SPI1 配置 */
39.     SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
40. 
41.     SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
42.     SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
43.     SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
44.     SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
45.     SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
46.     SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
47. 
48.     SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
49.     SPI_InitStructure.SPI_CRCPolynomial = 7;
50.     SPI_Init(SPI1, &SPI_InitStructure);
51. 
52. }
```

在这个函数中不知大家有没注意到没有这两条注释:

```
1.     /* PB13:ENC28J60_INT */ // 中断引脚没用到
2. 
3.     /* PE1:ENC28J60_RST*/ // 复位似乎不用也可以
```

enc28j60 的中断引脚没用到很正常，但是复位引脚也没用到，这我就很纳闷了。

我想原因可能是 enc28j60 有个上电自动复位的功能，这里它的复位引脚只能暂时没有用到而已，也或许是我们的开发板中引脚 PE1（接 enc28j60 的复位脚）收到什么信号的干扰，产生了类似复位的信号。这里我们先把这问题搁一边先，毕竟程序还是工作了。至于具体的原因我以后有时间再深究下。

`Web_Server();` 函数实现的功能是创建一个网页服务器，在这个网页服务器上我可以点击我们设定好的命令按钮来控制我们开发板上 LED 的亮灭。其实，从这里面我们就可以看到有点智能家居的味道了，所谓智能家居就是通过网络来控制我们家电的状态。



---

态，如开和断。举个例子：我们可以在遥远的地方可以通过网络来控制家里的电视、电冰箱等，是不是很神奇哩？只要你学会了这个实验，再经过自己的再深入学习，这些都是小菜一碟。^\_^，本实验旨在引导大家入门。

`Web_Server()`; 在 `web_server.c` 中实现。在 `web_server.c` 的开头包含了头文件：

```
1. #include "enc28j60.h"  
2. #include "ip_arp_udp_tcp.h"  
3. #include "net.h"
```

`enc28j60.h` : Microchip ENC28J60 Ethernet Interface Driver Header file

`ip_arp_udp_tcp.h` : IP, Arp, UDP and TCP functions Header file . For more  
Information Please See  
<http://www.gnu.org/licenses/gpl.html>

`net.h` : Based on the `net.h` file from the AVRlib library by Pascal Stang.  
For AVRlib See <http://www.procyonengineering.com/>  
Used with explicit permission of Pascal Stang.

有关这三个头文件对应的 C 文件的功能，请大家阅读源码。

接下来我们具体看看 `Web_Server()`; 这个函数具体做了什么，由于这个函数的源码较多，在这里我就不贴出来了。

`enc28j60Init(mymac);` 这个函数用来初始化 `enc28j60` 的 MAC 地址(物理地址),这个函数必须要调用一次。 `mymac` 在 `web_server.c` 的开头定义：

`static unsigned char mymac[6] = {0x54,0x55,0x58,0x10,0x00,0x24};` 这里要注意的是：`mac` 地址在局域网内必须唯一，否则将与其他主机冲突，导致连接不成功。  
`mymac` 数组里面的数值可随便初始化，但万万不可与局域网内的 `mac` 地址冲突，否则当另外一部主机在上网时，你是上不了网的。



enc28j60PhyWrite(PHLCON,0x476);这个函数用于设定网络变压器中 LED 的颜色，不同的颜色指示不同的状态。网络变压器在没有工作的情况下，这两个 LED 是不会被点亮的，当网络变压器工作正常时，绿色 LED 表示 link 状态，黄色 LED 表示通信状态。本实验中，我们的程序工作正常时，绿色 LED 常亮，黄色 LED 是一闪一闪的。

init\_ip\_arp\_udp\_tcp(mymac,myip,mywwwport); 这个函数用于初始化以太网的 IP 层。这里面涉及到三个参数：mymac、myip、mywwwport。其中 mymac 在前面已经讲解过。ip 指的是我们开发板的 ip 地址，要想通过网页来访问我们的服务器(即开发板)，则必须要有 ip。ip 地址跟 mac 地址一样，在局域网能要保持唯一，不能够与其他主机的 ip 地址产生冲突。还要注意的一点就是：开发板的 ip 与我们电脑的 ip 必须保持在同一个网段，即 ip 地址的前三段要保持一致，后面一段不同。在本地链接中可查看到我们电脑的 ip 地址。我的电脑的 ip 地址是：**192.168.1.106**，如下截图所示：





所以，在此设置我们开发板的 ip 为 `static unsigned char myip[4] = {192,168,1,15};` ip 的最后一段的值 15 可改为其他值，但要注意不要产生冲突。

接下来的是一个无限循环 `while (1)`，在这里面我们创建了一个网页，在网页中注入了我们自己的信息，并随时监控我们命令状态的改变，好实时地控制我们的 LED，有关这些功能的实现请大家自行阅读源码，这里不再详述。我们仅来看看 LED 控制的部分：

```
1. if (cmd==1)      // 用户程序
2. {
3.     LED1(ON);
4.     i=1;           // 命令 = 1
5. }
6. if (cmd==0)      // 用户程序
7. {
8.     LED1(OFF);
9.     i=0;           // 命令 = 0
10. }
```

当我们在网页上点击 点亮 LED 这个按钮时，网页发送命令 1 给开发板，这是开发板中的 LED 被点亮，反之，LED 则被关闭。

这里我们仅提供简单的 web 服务程序，如果要实现更加复杂的功能，仍需大家的努力。当然，我们工作室也会跟大家一起奋斗，开发出更多的应用程序跟大家分享。

这里面的操作涉及到很多 enc28j60 的知识，特别是寄存器的操作，具体的请大家参考 enc28j60 的 datasheet，大家一定要看看，而且是要认认真真、仔仔细细地看，直到弄懂为止。

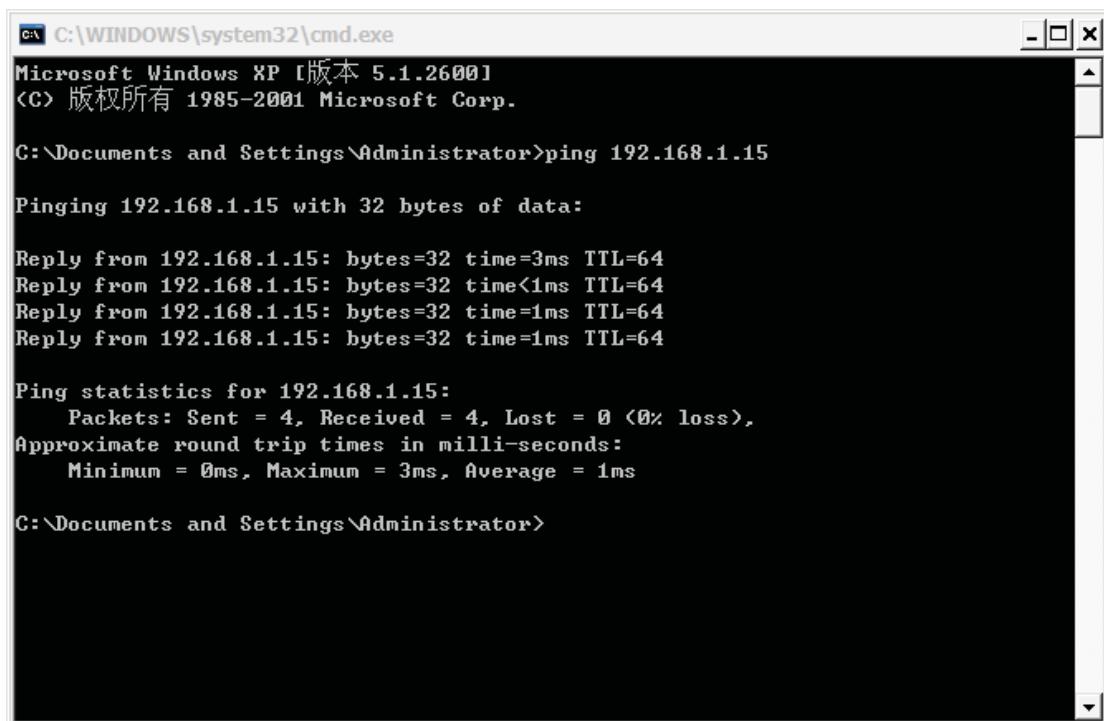


---

### 实验现象->

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 插上网口线, 网口线一端连接路由器, 一端连开发板。注意: 电脑跟开发板的网线连接的路由器要同在一个局域网中。将编译好的程序下载到开发板。

1: 打开电脑的 DOS 界面, 输入: **ping 192.168.1.15**, 看看能否 ping 通。打印出如下信息则表示 ping 通, 其中 **192.168.1.15** 是我们开发板的 ip。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>ping 192.168.1.15

Pinging 192.168.1.15 with 32 bytes of data:
Reply from 192.168.1.15: bytes=32 time=3ms TTL=64
Reply from 192.168.1.15: bytes=32 time<1ms TTL=64
Reply from 192.168.1.15: bytes=32 time=1ms TTL=64
Reply from 192.168.1.15: bytes=32 time=1ms TTL=64

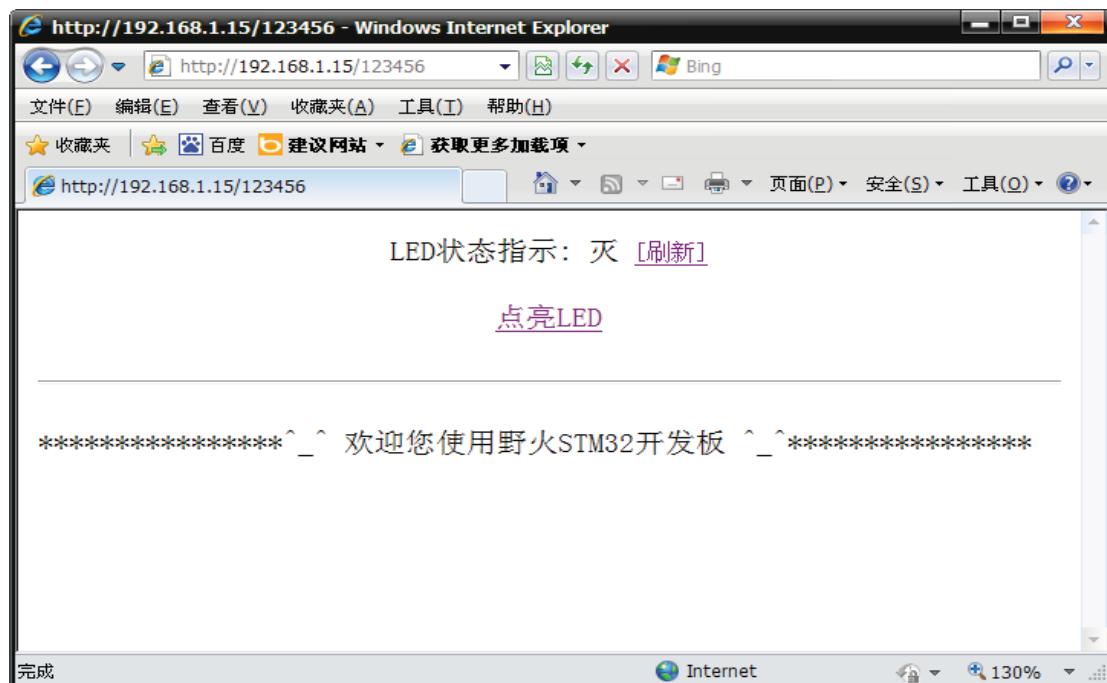
Ping statistics for 192.168.1.15:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 3ms, Average = 1ms

C:\Documents and Settings\Administrator>
```

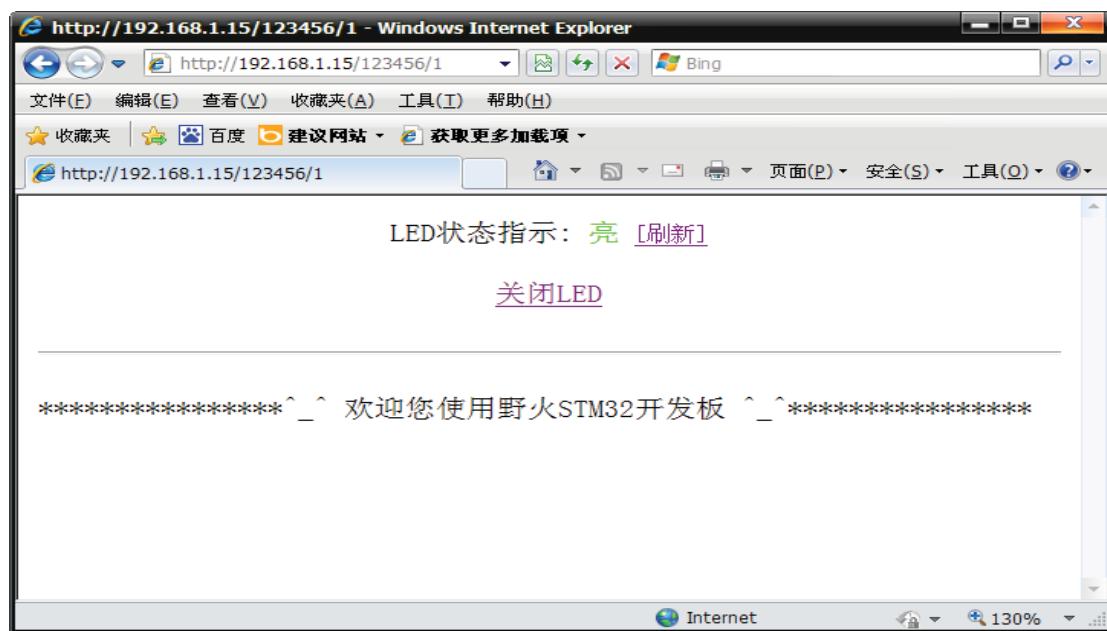


2: 打开 IE 浏览器, 在地址栏输入: <http://192.168.1.15/123456> , 其中 123456 是我们自己设置的密码。然后按 Enter 键进入, 如果成功则会出现一个如下网页, 通过网页中的命令按钮 点亮 LED / 关闭 LED 就可以控制我们 开发板中 LED 的亮灭, 这里控制的是开发板中的 LED1。

### LED1 灭



### LED1 亮



实验讲解完毕, 野火祝大家学习愉快。