

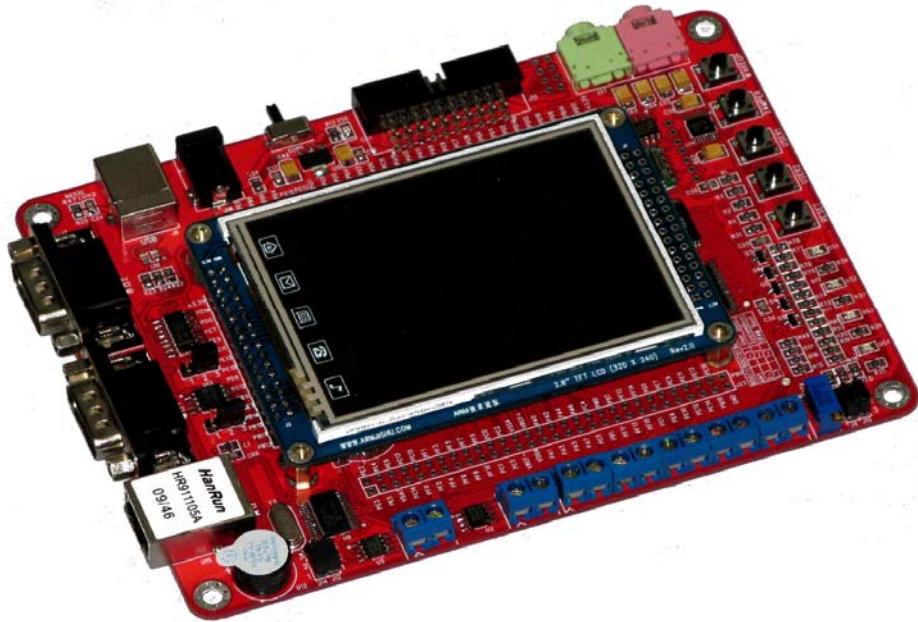


STM32 神舟 III 号开发板用户手册

2014 年 8 月 10 日 版本 V4.2 作者: WWW.ARMJISHU.COM

STM32神舟ARM系列技术开发板产品目录:

- 【神舟51单片机开发板（51+ARM）开发板】
- 神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏
- 神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- 【神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏】
- 神舟IV号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- STM32核心板: 四层核心板 (STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列 (STM32F103ZET核心板)
- 神舟王207系列 (STM32F207ZGT核心板)
- 神舟王407系列 (STM32F407ZGT/407IGT核心板)
- 神舟王全系列 (STM32F103ZET/207ZGT/407ZGT核心板): 全功能底板 (支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
- 神舟 51 开发板 (STC 51 单片机+STM32F103C8T6 核心板): 全功能底板 (支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)



2014 年 7 月 15 日

预告 A: 准备开始视频教程的发布

预告 B: 2014 年 7 月 26 日将发布 STM32 神舟 III 号 V4.1, 发布补充和支持以下这些模块:

- 1) 1602 屏例程与文档手册
- 2) 4.3 寸彩屏例程与实验说明
- 3) 7 寸彩屏例程与实验说明
- 4) VS1003 MP3 模块例程与文档手册

2014 年 7 月 20 日

2014 年 7 月 20 日恭喜, STM32 神舟 III 号文档即可发布 V4.1 版本, 总共增加内容:

- 1) 新增 1. DAC 数模转换例程与文档
- 2) 新增 1. 315M 模块例程
- 3) 新增 2. 1602 屏例程与文档手册
- 4) 新增 3. 4.3 寸彩屏例程与实验说明
- 5) 新增 4. 7 寸彩屏例程与实验说
- 6) 新增 5. VS1003 MP3 模块例程与文档手册

2014 年 7 月 24 日

2014 年 7 月 20 日恭喜，STM32 神舟 III 号文档即可发布 V4.1 版本，总共更新或增加内容：

- 1) 更新 1. 如何搭建开发环境文档
- 2) 新增 2. 如何搭建开发环境视频教程
- 3) 更新 3. 如何新建模板工程文档
- 4) 新增 4. 如何新建模板工程视频教程
- 5) 更新 5. 初识 STM32 库体系结构文档
- 6) 新增 6. 初识 STM32 库体系结构视频教程
- 7) 更新 7. 流水灯实验文档
- 8) 新增 8. 流水灯实验视频教程
- 9) 更新 9. 按键输入实验文档
- 10) 新增 10. 按键输入实验视频教程
- 11) 更新 11. 外部中断实验文档
- 12) 新增 12. 外部中断实验视频教程
- 13) 更新 13. Systick 精确延时实验文档
- 14) 新增 14. Systick 精确延时实验视频教程
- 15) 更新 15. 串口实验文档
- 16) 新增 16. 串口实验视频教程

2014 年 8 月 09 日

2014 年 8 月 10 日恭喜，STM32 神舟 III 号文档即可发布 V4.2 版本，总共更新或增加内容：

- 1) 新增 1. EXTI 315M 无线模块外部中断实验例程及文档
- 2) 新增 2. 串口高级例程之 Printf 中断接收实验例程及文档
- 3) 新增 3. 串口 IAP 在线升级实验例程及文档
- 4) 新增 4. 两块神舟 III 号开发板 CAN 收发实验例程及文档
- 5) 新增 5. SD 卡识别(SDIO 方式)例程及文档
- 6) 新增 6. SD 卡+Fatfs 文件系统例程及文档
- 7) 新增 7. 内部 SRAM+外部 SRAM 内存管理例程及文档

声 明

本手册版权归属 ARMJISHU.COM 所有，并保留一切权利。非经 ARMJISHU.COM 书面同意，任何单位或个人不得擅自摘录本手册部分或全部内容，违者我们将追究其法律责任。

本文档为 ARMJISHU.COM 网站推出的神舟 III 号 STM32 开发板配套用户手册，详细介绍 STM32 的开发过程和神舟 III 号的使用指导。

目 录

| | |
|--|----|
| 第一篇 硬件篇..... | 21 |
| 1.1. 神舟III号STM32 开发板简介 | 21 |
| 1.2. 神舟III号开发板硬件详解 | 29 |
| 1.2.1. 处理器最小系统..... | 29 |
| 1.2.2. LED指示灯 | 36 |
| 1.2.3. 按键..... | 36 |
| 1.2.4. USB接口与电源..... | 36 |
| 1.2.5. 液晶显示模块..... | 37 |
| 1.2.6. Nor Flash | 38 |
| 1.2.7. SRAM..... | 39 |
| 1.2.8. Nand Flash..... | 40 |
| 1.2.9. 串口与RS485 接口..... | 41 |
| 1.2.10. 收音机模块..... | 42 |
| 1.2.11. 音频解码电路..... | 42 |
| 1.2.12. SPI Flash..... | 43 |
| 1.2.13. EEPROM..... | 43 |
| 1.2.14. CAN总线接口..... | 44 |
| 1.2.15. SD卡..... | 44 |
| 1.2.16. 无线模块..... | 44 |
| 1.2.17. 以太网..... | 45 |
| 1.3. STM32 神舟III号开发板目前可以支持的外接模块 | 46 |
| 1.3.1 DS18B20 温度传感器..... | 46 |
| 1.3.2 DH11 温度传感器..... | 46 |
| 1.3.3 2.4G 无线模块 (WIFI也属于2.4G 范畴之内) | 46 |
| 1.3.4 315M 无线模块接收..... | 46 |
| 1.3.5 三轴加速度感应模块..... | 46 |
| 1.3.6 MP3 音乐播放模块..... | 46 |
| 1.3.7 2.8 寸液晶屏模块..... | 46 |
| 1.3.8 3.2 寸液晶屏模块..... | 46 |
| 1.3.9 4.3 寸液晶屏模块..... | 46 |
| 1.3.10 7 寸液晶屏模块..... | 46 |
| 第二章 软件篇..... | 48 |
| 2.1 RVMDK简介 | 48 |
| 2.2 如何安装RVMDK软件 | 48 |
| 2.3 注册MDK软件 | 52 |
| 2.4 MDK 4.12 集成开发环境的组成 | 55 |
| 2.5 MDK工程的编辑 | 56 |
| 2.5.1 新建RVMDK 工程..... | 56 |
| 2.5.2 建立文件..... | 60 |
| 2.5.3 添加文件到工程..... | 61 |
| 2.5.4 管理工程目录以及源文件..... | 62 |
| 2.5.5 编译和连接工程..... | 65 |
| 2.5.6 打开旧工程..... | 68 |
| 2.6 RVMDK使用技巧 | 69 |
| 2.6.1 快速定位函数/变量被定义的地方..... | 69 |
| 2.6.2 快速注释与快速消注释..... | 71 |
| 2.6.3 快速打开头文件..... | 71 |
| 2.7 JLINK V8 仿真器的安装与应用 | 73 |
| 2.7.1 JLINK V8 仿真器简介..... | 73 |

| | | |
|--|--|------------|
| 2.7.2 | JLINK ARM 主要特点 | 73 |
| 2.7.3 | 如何安装JLINK 软件 | 74 |
| 2.7.4 | JLINK V8 仿真器配置 (MDK KEIL环境) | 76 |
| 2.7.5 | 使用KEIL的DOWNLOAD功能..... | 79 |
| 2.8 | 在MDK开发环境中JLINK V8 的调试技巧 | 79 |
| 2.9 | 在MDK开发环境中调试 | 81 |
| 2.9.1 | KEIL仿真的应用 | 81 |
| 2.9.2 | KEIL软件仿真 | 81 |
| 2.9.3 | 硬件仿真 | 86 |
| 2.9.4 | DEBUG模式下不能watch的解决办法 | 90 |
| 2.10 | 如何设置程序空间在CPU内部FLASH， 变量空间在CPU内部RAM运行 | 90 |
| 2.11 | 如何设置程序空间和变量空间都在CPU内部RAM运行 | 91 |
| 2.12 | 如何设置程序空间和变量空间都在外部SRAM运行 | 94 |
| 第三章 STM32 神舟III号开发板硬件使用基础篇..... | | 103 |
| 3.1 | 如何给神舟III号板供电 | 103 |
| 3.1.1 | 使用USB供电 | 103 |
| 3.1.2 | 使用外接电源供电 | 103 |
| 3.1.3 | 使用JLINK V8 供电 | 103 |
| 3.2 | 如何使用JLINK软件下载固件到神舟开发板 | 104 |
| 3.2.1 | 如何使用J-FLASH ARM 烧写固件到芯片FLASH | 104 |
| 3.2.2 | 使用J-Link command 设置和查看相关调试信息 | 109 |
| 3.3 | 如何通过串口下载一个固件到神舟III号开发板 | 110 |
| 3.4 | 如何在IAR开发环境中使用JLINK在线调试 | 115 |
| 3.5 | 如何在MDK开发环境中使用JLINK在线调试 | 116 |
| 3.6 | 神舟III号跳线含义 | 119 |
| 3.6.1 | 启动模式选择跳线 | 120 |
| 3.6.2 | RS-232/RS-485 选择跳线 | 120 |
| 3.6.3 | RTC实时时钟跳线 | 120 |
| 3.6.4 | Nand Flash 访问选择跳线 | 120 |
| 第四章 STM32 神舟III号 零基础入门篇..... | | 120 |
| 4.1 | 如何从零开始新建STM32 工程模板 | 120 |
| 4.1.1 | 如何去官网下载最新的STM32 资料的方法 | 120 |
| 4.1.2 | 获取ST库源码 | 126 |
| 4.1.3 | 开始新建工程 | 126 |
| 4.1.4 | MDK环境设置 | 133 |
| 4.1.5 | 使用JLINK V8 仿真器硬件调试配置 | 138 |
| 4.2 | 理解芯片控制的原理 | 142 |
| 4.3 | 芯片管脚控制LED灯原理图解释 | 143 |
| 4.4 | STM32 相关的芯片手册有哪些？我们如何阅读这些资料 | 144 |
| 4.5 | STM32 芯片各个管脚是怎么控制以及被管理的？（如何阅读芯片手册） | 145 |
| 4.6 | STM32 芯片单个管脚是怎么被控制以及被管理的？（如何阅读芯片寄存器） | 148 |
| 4.7 | 分析一个最简单的例程 | 151 |
| 4.7.1 | 硬件原理图分析 | 151 |
| 4.7.2 | 例程环境搭建 | 154 |
| 4.7.3 | 实验现象 | 160 |
| 4.7.4 | 代码详细分析： | 160 |
| 4.7.5 | 程序代码详细说明 | 166 |
| 4.7.6 | 代码如何映射到芯片内部的寄存器 | 169 |
| 4.7.7 | Main 函数寄存器级分析（重点） | 170 |
| 4.7.8 | 函数与我们这个例程之间的关系 | 175 |
| 4.8 | STM32 重映射功能 | 175 |
| 4.8.1 | 什么是STM32 的重映射 | 175 |
| 4.8.2 | 所有的管脚都可以重映射吗 | 175 |

| | | |
|---|-----------------------------------|------------|
| 4.8.3 | 为什么要有STM32 重映射这个功能..... | 175 |
| 4.8.4 | 举例说明..... | 175 |
| 4.8.5 | 深入分析STM32 重映射内部架构原理..... | 176 |
| 4.8.6 | STM32 重映射关键指点..... | 178 |
| 4.9 | STM32 的内存管理研究 (KEIL编程环境下) | 178 |
| 4.9.1 | 研究意义..... | 178 |
| 4.9.2 | 举例说明并详细分析..... | 178 |
| 4.9.3 | 举例分析..... | 179 |
| 4.9.4 | 观察堆栈..... | 181 |
| 4.10 | STM32 芯片加密解密 | 183 |
| 4.10.1 | 关于芯片加密的定义..... | 183 |
| 4.10.2 | 关于芯片解密方法的理论总结..... | 183 |
| 4.10.3 | 常规芯片解密过程..... | 184 |
| 4.10.4 | 增加芯片解密难度的一些建议总结..... | 184 |
| 4.10.5 | STM32 加密思路-01 串口ISP 设置加密..... | 186 |
| 4.10.6 | STM32 加密思路-02 软件加密..... | 186 |
| 4.10.7 | STM32 加密思路-03 外置ID芯片..... | 186 |
| 4.10.8 | STM32 加密思路-04 程序自毁..... | 186 |
| 4.10.9 | STM32 加密思路-05 磨IC型号..... | 186 |
| 4.10.10 | STM32 加密思路-06 高端硬件加密..... | 186 |
| 4.10.11 | STM32 加密思路-07 AES加密..... | 186 |
| 4.11 | STM32 低功耗经验总结 | 187 |
| 4.11.1 | STM32 低功耗实战项目案例故事 (转载) | 187 |
| 4.11.2 | STM32 低功耗三种模式 | 191 |
| 4.11.3 | STM32 低功耗需注意的地方 | 191 |
| 4.12 | STM32 的中断与事件关系的区别 | 191 |
| 第五章 STM32 神舟III号 实战篇 (寄存器版本) | | 193 |
| 5.1 | 通用输入/输出 (GPIO) | 193 |
| 5.1.1 | 脚特性..... | 193 |
| 5.1.2 | GPIO应用领域..... | 193 |
| 5.1.3 | 管脚分配..... | 194 |
| 5.1.4 | GPIO管脚内部硬件电路原理剖析..... | 194 |
| 5.1.5 | STM32 的GPIO 管脚深入分析..... | 197 |
| 5.1.6 | 在STM32 中如何配置片内外设使用的IO端口 | 201 |
| 5.1.7 | 例程01 单个LED点灯闪烁程序..... | 202 |
| 5.1.8 | 例程02 LED双灯闪烁实验 | 205 |
| 5.1.9 | 例程03 LED三个灯同时亮同时灭 | 206 |
| 5.1.10 | 例程04 LED流水灯程序 | 208 |
| 5.2 | 时钟 | 209 |
| 5.2.1 | 什么是时钟..... | 209 |
| 5.2.2 | STM32 的时钟 | 209 |
| 5.2.3 | STM32 的时钟深入分析 | 210 |
| 5.2.4 | 例程01 STM32 芯片 32MHZ频率下跑点灯程序 | 214 |
| 5.2.5 | 例程02 STM32 芯片 40MHZ频率下跑点灯程序 | 219 |
| 5.2.6 | 例程03 STM32 芯片 72MHZ频率下跑点灯程序 | 220 |
| 5.3 | 独立按键 | 221 |
| 5.3.1 | 键的分类 | 221 |
| 5.3.2 | 按键属性 | 221 |
| 5.3.3 | STM32 的位带操作 | 223 |
| 5.3.4 | 例程01 STM32 芯片按键点灯 (无防抖) | 227 |
| 5.3.5 | 例程02 STM32 芯片按键点灯-增加了防抖的代码 | 232 |
| 5.4 | 串口通信的收与发 | 233 |
| 5.4.1 | 什么是串口通信 | 233 |
| 5.4.2 | 串口通信的属性 | 233 |

| | | |
|----------------------------------|---|------------|
| 5.4.3 | 什么是单片机的TTL电平? | 237 |
| 5.4.4 | 关于NPN和PNP的三极管基础知识? | 238 |
| 5.4.5 | RS-232 电平与TTL电平的转换..... | 239 |
| 5.4.6 | 串口波特率的理解..... | 241 |
| 5.4.7 | STM32 神舟III号独特的USB转串口的TTL电平模块设计..... | 242 |
| 5.4.8 | 例程01 最简单串口打印\$字符..... | 242 |
| 5.4.9 | 例程02 单串口打印www.armjishu.com字符-初级 | 250 |
| 5.4.10 | 例程03 单串口打印www.armjishu.com字符-中级 | 252 |
| 5.4.11 | 例程04 单串口打印www.armjishu.com字符-高级 | 253 |
| 5.4.12 | 例程05 USART-COM1 串口接收与发送实验-初级版..... | 254 |
| 5.4.13 | 例程06 USART-COM1 串口接收与发送实验-中级版..... | 256 |
| 5.4.14 | 例程05 USART-COM1 串口接收与发送实验-高级版..... | 257 |
| 第六章 STM32 库函数架构剖析 | | 259 |
| 6.1 | STM32 库函数到底是什么 | 260 |
| 6.2 | STM32 库函数的好处 | 260 |
| 6.3 | 千人大项目如何分配工作 | 260 |
| 6.4 | STM32 库结构剖析 | 261 |
| 6.4.1 | CMSIS标准 | 261 |
| 6.4.2 | 库目录, 文件简介..... | 262 |
| 6.4.3 | 关于core_cm3.c文件..... | 263 |
| 6.4.4 | system_stm32f10x.c文件..... | 264 |
| 6.4.5 | stm32f10x.c文件..... | 265 |
| 6.4.6 | 启动文件..... | 265 |
| 6.4.7 | STM32F10x_StdPeriph_Driver文件夹 | 267 |
| 6.4.8 | stm32f10x_it.c、stm32f10x_conf.h文件 | 268 |
| 6.4.9 | 库各文件间的关系 | 269 |
| 6.4.10 | 常用官方资料..... | 271 |
| 6.4.11 | 库函数帮助文档使用..... | 271 |
| 第七章 STM32 神舟III号 实战篇..... | | 273 |
| 7.1 | LED流水灯实验 | 273 |
| 7.1.1 | STM32 的最简单外设GPIO分析..... | 273 |
| 7.1.2 | LED具体代码分析 | 276 |
| 7.1.3 | STM32 的时钟系统 | 280 |
| 7.1.4 | STM32 的地址映射 | 283 |
| 7.1.5 | STM32 库对寄存器的封装 | 287 |
| 7.1.6 | 下载与验证 | 289 |
| 7.1.7 | 实验现象 | 289 |
| 7.2 | 蜂鸣器实验 | 289 |
| 7.2.1 | 蜂鸣器的简介 | 289 |
| 7.2.2 | 蜂鸣器的结构 | 290 |
| 7.2.3 | 意义与作用 | 290 |
| 7.2.4 | 实验原理 | 291 |
| 7.2.5 | 硬件设计 | 291 |
| 7.2.6 | 软件设计 | 291 |
| 7.2.7 | 下载与验证 | 294 |
| 7.2.8 | 实验现象 | 295 |
| 7.3 | 按键检测实验 | 295 |
| 7.3.1 | 单片机检测小弹性按键的原理 | 295 |
| 7.3.2 | GPIO的8 种工作模式 | 297 |
| 7.3.3 | 代码分析 | 299 |
| 7.3.1 | 下载与验证 | 301 |
| 7.3.4 | 实验现象 | 301 |
| 7.4 | BITBAND按键扫描检测实验 | 302 |

| | | |
|--------|--|-----|
| 7.4.1 | 什么是位带操作..... | 302 |
| 7.4.2 | 为什么要用位带操作..... | 302 |
| 7.4.3 | 如何设计和实现位带操作的..... | 303 |
| 7.4.4 | STM32 中位带操作的具体部署情况..... | 304 |
| 7.4.5 | 如何用代码来实现位带操作..... | 305 |
| 7.4.6 | 软件设计..... | 307 |
| 7.4.7 | 下载与验证..... | 311 |
| 7.4.8 | 实验现象..... | 311 |
| 7.5 | EXTI外部按键中断实验 | 312 |
| 7.5.1 | 什么是中断..... | 312 |
| 7.5.2 | 什么是单片机的中断? | 312 |
| 7.5.3 | STM32 中断的初步理解..... | 313 |
| 7.5.4 | STM32 中断的初始化过程以及手册的查询..... | 314 |
| 7.5.5 | NVIC中断控制器..... | 316 |
| 7.5.6 | EXTI外部中断..... | 318 |
| 7.5.7 | 硬件设计..... | 320 |
| 7.5.8 | 软件设计..... | 321 |
| 7.5.9 | 下载与验证..... | 325 |
| 7.5.10 | 实验现象..... | 325 |
| 7.6 | SYSTICK系统滴答实验 | 325 |
| 7.6.1 | Systick简介..... | 325 |
| 7.6.2 | 实验原理..... | 327 |
| 7.6.3 | 代码分析..... | 329 |
| 7.6.4 | 实验现象..... | 332 |
| 7.7 | USART串口 1 通信实验 | 333 |
| 7.7.1 | STM32 的异步串口通讯协议 | 333 |
| 7.7.2 | 交叉线和直连线..... | 333 |
| 7.7.3 | 串口工作流程..... | 334 |
| 7.7.4 | 例程 01 USART串口 1 Printf打印输出字符..... | 335 |
| 7.7.5 | 软件设计..... | 335 |
| 7.7.6 | 下载与验证..... | 342 |
| 7.7.7 | 实验现象..... | 342 |
| 7.7.8 | 例程 02 USART串口 1-带SYSTICK 中断Printf()..... | 344 |
| 7.7.9 | 软件设计..... | 344 |
| 7.7.10 | 实验现象..... | 344 |
| 7.7.11 | 例程 03 USART串口 1Printf输出和scanf输入 | 344 |
| 7.7.12 | 软件设计..... | 344 |
| 7.7.13 | 实验现象..... | 345 |
| 7.8 | USART串口 2 通信实验 | 348 |
| 7.8.1 | 意义与作用..... | 348 |
| 7.8.2 | 串口 2 与串口 1 的区别..... | 348 |
| 7.8.3 | 多个串口如何同时接收源源不断发送给每个串口的数据 | 348 |
| 7.8.4 | 两个串口同时来数据了 这个时候只能进入一个中断，会丢失数据吗..... | 349 |
| 7.8.5 | 硬件设计..... | 350 |
| 7.8.6 | 例程 01 USART串口 2Printf () 打印 | 353 |
| 7.8.7 | 软件设计..... | 353 |
| 7.8.8 | 下载与验证..... | 353 |
| 7.8.9 | 实验现象..... | 353 |
| 7.8.10 | 例程 02 USART串口 2-带SYSTICK 中断Printf()..... | 355 |
| 7.8.11 | 例程 03 USART串口 2Printf输出和scanf输入 | 355 |
| 7.9 | UART串口 1 和串口 2 同时格式化输出输入 | 355 |
| 7.9.1 | 下载与验证..... | 355 |
| 7.9.2 | 实验现象..... | 355 |
| 7.10 | 串口高级例程之PRINTF中断接收实验 | 359 |
| 7.10.1 | 意义与作用..... | 359 |

| | | |
|---------|-----------------------------------|-----|
| 7.10.2 | 实验原理..... | 359 |
| 7.10.3 | 硬件设计..... | 359 |
| 7.10.4 | 软件设计..... | 359 |
| 7.10.5 | 下载与验证..... | 362 |
| 7.10.6 | 实验现象..... | 362 |
| 7.11 | 串口高级例程之PRINTF中断收发实验..... | 362 |
| 7.11.1 | 意义与作用..... | 363 |
| 7.11.2 | 实验原理..... | 363 |
| 7.11.3 | 硬件设计..... | 363 |
| 7.11.4 | 软件设计..... | 364 |
| 7.11.5 | 下载与验证..... | 368 |
| 7.11.6 | 实验现象..... | 368 |
| 7.12 | RS-485 总线收发实验..... | 370 |
| 7.12.1 | 485 简介..... | 370 |
| 7.12.2 | RS485 的通信概念..... | 371 |
| 7.12.3 | RS485 的连接方式..... | 372 |
| 7.12.4 | RS485 通信电缆中的信号反射..... | 372 |
| 7.12.5 | RS485 的接地问题..... | 373 |
| 7.12.6 | RS485 的应用..... | 373 |
| 7.12.7 | 原理图的连接..... | 374 |
| 7.12.8 | 实验现象..... | 376 |
| 7.12.9 | 软件设计..... | 376 |
| 7.12.10 | 下载与验证..... | 383 |
| 7.12.11 | 实验现象..... | 383 |
| 7.13 | 产品唯一身份标识（UNIQUE DEVICE ID）实验..... | 388 |
| 7.13.1 | 意义与作用..... | 388 |
| 7.13.2 | 实验原理..... | 388 |
| 7.13.3 | 硬件设计..... | 388 |
| 7.13.4 | 软件设计..... | 389 |
| 7.13.5 | 下载与验证..... | 391 |
| 7.13.6 | 实验现象..... | 391 |
| 7.14 | ADC模数转换实验..... | 392 |
| 7.14.1 | 意义与作用..... | 392 |
| 7.14.2 | 实验原理..... | 392 |
| 7.14.3 | 硬件设计..... | 395 |
| 7.14.4 | 软件设计..... | 396 |
| 7.14.5 | 下载与验证..... | 399 |
| 7.14.6 | 实验现象..... | 399 |
| 7.15 | ADC数据多路采集..... | 401 |
| 7.15.1 | 意义与作用..... | 401 |
| 7.15.2 | 实验原理..... | 401 |
| 7.15.3 | 硬件设计..... | 401 |
| 7.15.4 | 软件设计..... | 401 |
| 7.15.5 | 下载与验证..... | 401 |
| 7.15.6 | 实验现象..... | 401 |
| 7.16 | DAC数模转换实验..... | 401 |
| 7.16.1 | STM32 DAC简介..... | 401 |
| 7.16.2 | 实验原理..... | 402 |
| 7.16.3 | 如何控制正弦波的频率..... | 402 |
| 7.16.4 | 实验现象..... | 403 |
| 7.16.5 | 代码分析..... | 403 |
| 7.16.6 | 下载与验证..... | 408 |
| 7.16.7 | 实验现象..... | 408 |
| 7.17 | RTC实时时钟实验..... | 408 |
| 7.17.1 | 意义与作用..... | 408 |

| | | |
|--------|---------------------------|-----|
| 7.17.2 | 实验原理..... | 409 |
| 7.17.3 | 硬件设计..... | 411 |
| 7.17.4 | 软件设计..... | 412 |
| 7.17.5 | 下载与验证..... | 418 |
| 7.17.6 | 实验现象..... | 418 |
| 7.18 | CALENDAR实时时钟与农历年月日实验..... | 420 |
| 7.18.1 | 意义与作用..... | 421 |
| 7.18.2 | 实验原理..... | 421 |
| 7.18.3 | 硬件设计..... | 423 |
| 7.18.4 | 软件设计..... | 424 |
| 7.18.5 | 下载与验证..... | 429 |
| 7.18.6 | 实验现象..... | 430 |
| 7.19 | EEPROM读写测试实验..... | 432 |
| 7.19.1 | 意义与作用..... | 432 |
| 7.19.2 | I2C的介绍..... | 432 |
| 7.19.3 | 实验原理..... | 433 |
| 7.19.4 | 硬件设计..... | 434 |
| 7.19.5 | 软件设计..... | 435 |
| 7.20 | FLASH模拟EEPROM..... | 448 |
| 7.20.1 | 意义与作用..... | 448 |
| 7.20.2 | STM32 Flash浅析..... | 448 |
| 7.20.3 | 实验原理..... | 450 |
| 7.20.4 | 硬件设计..... | 450 |
| 7.20.5 | 软件设计..... | 450 |
| 7.20.6 | 下载与验证..... | 450 |
| 7.20.7 | 实验现象..... | 451 |
| 7.21 | TIMER定时器中断实验..... | 452 |
| 7.21.1 | 意义与作用..... | 452 |
| 7.21.2 | 实验原理..... | 452 |
| 7.21.3 | 硬件设计..... | 453 |
| 7.21.4 | 软件设计..... | 453 |
| 7.21.5 | 下载与验证..... | 455 |
| 7.21.6 | 实验现象..... | 456 |
| 7.22 | IWDG独立看门狗..... | 457 |
| 7.22.1 | 什么是看门狗..... | 457 |
| 7.22.2 | 为什么是独立看门狗呢..... | 457 |
| 7.22.3 | 怎么使用独立看门狗IWDG..... | 457 |
| 7.22.4 | 启动STM32 的独立看门狗..... | 460 |
| 7.22.5 | 软件设计..... | 461 |
| 7.22.6 | 下载与验证..... | 461 |
| 7.22.7 | 实验现象..... | 462 |
| 7.23 | 窗口看门狗..... | 463 |
| 7.23.1 | 什么是窗口看门狗..... | 463 |
| 7.23.2 | 窗口看门狗的特性..... | 463 |
| 7.23.3 | 窗口看门狗的计算公式: | 464 |
| 7.23.4 | 窗口看门狗要用到哪些寄存器..... | 465 |
| 7.23.5 | 窗口看门狗与独立看门狗的区别..... | 466 |
| 7.23.6 | 怎么使用窗口看门狗..... | 467 |
| 7.23.7 | 硬件设计..... | 467 |
| 7.23.8 | 软件设计..... | 467 |
| 7.23.9 | 实验现象..... | 470 |
| 7.24 | 输入捕获实验..... | 471 |
| 7.24.1 | 意义与作用..... | 471 |
| 7.24.2 | 输入捕获介绍..... | 471 |
| 7.24.3 | 实验原理..... | 471 |

| | | |
|---------|--------------------------------------|-----|
| 7.24.4 | 与输入捕获相关的寄存器..... | 471 |
| 7.24.5 | 输入捕获配置的步骤..... | 475 |
| 7.24.6 | 硬件设计..... | 475 |
| 7.24.7 | 软件设计..... | 476 |
| 7.24.8 | 实验现象..... | 479 |
| 7.25 | 315M无线模块扫描实验..... | 480 |
| 7.25.1 | 意义与作用..... | 480 |
| 7.25.2 | 实验原理..... | 481 |
| 7.25.3 | 硬件设计..... | 481 |
| 7.25.4 | 软件设计..... | 484 |
| 7.25.5 | 下载与验证..... | 490 |
| 7.25.6 | 实验现象..... | 491 |
| 7.26 | EXTI无线315M模块外部中断实验..... | 493 |
| 7.26.1 | 意义与作用..... | 494 |
| 7.26.2 | 实验原理..... | 494 |
| 7.26.3 | 硬件设计..... | 495 |
| 7.26.4 | 软件设计..... | 498 |
| 7.26.5 | 下载与验证..... | 499 |
| 7.26.6 | 实验现象..... | 499 |
| 7.27 | CAN总线实验..... | 501 |
| 7.27.1 | 什么是CAN总线..... | 501 |
| 7.27.2 | CAN总线的特点..... | 502 |
| 7.27.3 | CAN总线技术介绍..... | 502 |
| 7.27.4 | CAN总线关键特性参数..... | 504 |
| 7.27.5 | CAN总线的三种工作模式..... | 505 |
| 7.27.6 | 实验原理..... | 506 |
| 7.27.7 | 硬件设计..... | 510 |
| 7.27.8 | 软件设计..... | 511 |
| 7.27.9 | 下载与测试..... | 519 |
| 7.27.10 | 实验现象..... | 519 |
| 7.28 | 双CAN收发测试实验..... | 519 |
| 7.28.1 | 意义与作用..... | 519 |
| 7.28.2 | 实验原理..... | 519 |
| 7.28.3 | 硬件设计..... | 519 |
| 7.28.4 | 软件设计..... | 520 |
| 7.28.5 | 下载与验证..... | 526 |
| 7.28.6 | 实验现象..... | 526 |
| 7.29 | SPI FLASH读写实验..... | 527 |
| 7.29.1 | SPI FLASH (W25X16) 读写程序实验的意义与作用..... | 527 |
| 7.29.2 | SPI的介绍..... | 528 |
| 7.29.3 | 硬件设计..... | 531 |
| 7.29.4 | 软件设计..... | 532 |
| 7.29.5 | 下载与测试..... | 536 |
| 7.29.6 | 实验现象..... | 537 |
| 7.30 | SRAM访问实验..... | 537 |
| 7.30.1 | SRAM访问实验的意义与作用..... | 537 |
| 7.30.2 | IS62WV25616简介..... | 537 |
| 7.30.3 | STM32处理器FSMC总线简介..... | 540 |
| 7.30.4 | 实验原理..... | 543 |
| 7.30.5 | 硬件设计..... | 544 |
| 7.30.6 | 软件设计..... | 545 |
| 7.30.7 | 下载与测试..... | 549 |
| 7.30.8 | 实验现象..... | 549 |
| 7.31 | 内存管理..... | 549 |
| 7.31.1 | 意义与作用..... | 549 |

| | |
|-------------------------------------|-----|
| 7.31.2 内存管脚简介..... | 549 |
| 7.31.3 实验原理..... | 550 |
| 7.31.4 硬件设计..... | 550 |
| 7.31.5 软件设计..... | 551 |
| 7.31.6 下载与验证..... | 553 |
| 7.31.7 实验现象..... | 553 |
| 7.32 NOR FLASH访问程序实验..... | 555 |
| 7.32.1 Nor Flash与Nand Flash的区别..... | 555 |
| 7.32.2 FSMC扩展Nor Flash配置..... | 557 |
| 7.32.3 Nor Flash访问实验的意义与作用..... | 558 |
| 7.32.4 实验原理..... | 558 |
| 7.32.5 硬件设计..... | 559 |
| 7.32.6 软件设计..... | 560 |
| 7.32.7 下载与测试..... | 566 |
| 7.32.8 实验现象..... | 566 |
| 7.33 NAND FLASH访问实验..... | 567 |
| 7.33.1 Nand Flash访问实验的意义与作用..... | 567 |
| 7.33.2 实验原理..... | 567 |
| 7.33.3 硬件设计..... | 569 |
| 7.33.4 软件设计..... | 571 |
| 7.33.5 下载与测试..... | 578 |
| 7.33.6 实验现象..... | 578 |
| 7.34 2.4G无线模块实验..... | 579 |
| 7.34.1 2.4G无线模块通讯实验的意义与作用..... | 579 |
| 7.34.2 试验原理..... | 579 |
| 7.34.3 硬件设计..... | 580 |
| 7.34.4 软件设计..... | 582 |
| 7.34.5 下载与测试..... | 585 |
| 7.34.6 实验现象..... | 586 |
| 7.35 收音机实验..... | 587 |
| 7.35.1 收音机实验的意义与作用..... | 587 |
| 7.35.2 实验原理..... | 588 |
| 7.35.3 硬件设计..... | 593 |
| 7.35.4 软件设计..... | 594 |
| 7.35.5 下载与测试..... | 599 |
| 7.35.6 实验现象..... | 600 |
| 7.36 TFT彩色液晶屏只显示红色..... | 602 |
| 7.36.1 术语解释..... | 602 |
| 7.36.2 液晶彩屏原理简介..... | 602 |
| 7.36.3 液晶彩屏图像像素分析..... | 602 |
| 7.36.4 控制器命令分析..... | 604 |
| 7.36.5 FSMC介绍..... | 606 |
| 7.36.6 FSMC函数初始化..... | 611 |
| 7.36.7 硬件设计..... | 613 |
| 7.36.8 软件分析..... | 614 |
| 7.36.9 下载与测试..... | 619 |
| 7.36.10 实验现象..... | 619 |
| 7.37 TFT彩色液晶屏显示蓝色..... | 620 |
| 7.37.1 硬件设计..... | 620 |
| 7.37.2 软件分析..... | 620 |
| 7.37.3 下载与测试..... | 621 |
| 7.37.4 实验现象..... | 621 |
| 7.38 TFT彩色液晶屏如何显示一个点..... | 622 |
| 7.38.1 扫描的简要分析..... | 622 |
| 7.38.2 硬件设计..... | 622 |

| | |
|---------------------------------|-----|
| 7.38.3 软件分析..... | 622 |
| 7.38.4 下载与测试..... | 624 |
| 7.38.5 实验现象..... | 624 |
| 7.39 TFT彩色液晶屏显示一个数字 1 | 625 |
| 7.39.1 扫描的简要分析..... | 625 |
| 7.39.2 什么是字模..... | 625 |
| 7.39.3 ASCII码的字符解释..... | 626 |
| 7.39.4 硬件设计..... | 628 |
| 7.39.5 软件分析..... | 628 |
| 7.39.6 下载与测试..... | 632 |
| 7.39.7 实验现象..... | 632 |
| 7.40 TFT彩色液晶屏显示变异的数字 1 | 633 |
| 7.40.1 简要分析..... | 633 |
| 7.40.2 软件分析..... | 633 |
| 7.40.3 软件分析..... | 634 |
| 7.41 TFT彩色液晶屏显示数字 9 | 634 |
| 7.41.1 简要分析..... | 634 |
| 7.41.2 软件分析..... | 634 |
| 7.41.3 下载测试..... | 635 |
| 7.41.4 实验现象..... | 636 |
| 7.42 TFT彩色液晶屏显示一个英文 | 636 |
| 7.42.1 简要分析..... | 636 |
| 7.42.2 硬件设计..... | 636 |
| 7.42.3 软件分析..... | 636 |
| 7.42.4 下载与测试..... | 637 |
| 7.42.5 实验现象..... | 637 |
| 7.43 TFT彩色液晶屏显示 26 个英文字母 | 638 |
| 7.43.1 简要分析..... | 638 |
| 7.43.2 硬件设计..... | 638 |
| 7.43.3 软件分析..... | 638 |
| 7.43.4 下载与测试..... | 639 |
| 7.43.5 实验现象..... | 639 |
| 7.44 TFT彩色液晶屏显示图片 | 640 |
| 7.44.1 图片格式介绍..... | 640 |
| 7.44.2 深入了解BMP图片..... | 640 |
| 7.44.3 使用工具将图片转换成二进制码..... | 641 |
| 7.44.4 硬件设计..... | 642 |
| 7.44.5 软件分析..... | 642 |
| 7.44.6 下载与测试..... | 645 |
| 7.44.7 实验现象..... | 645 |
| 7.45 TFT彩色液晶屏显示图片并刷屏【未更新】 | 645 |
| 7.45.1 硬件设计..... | 645 |
| 7.45.2 软件设计..... | 645 |
| 7.45.3 下载与测试..... | 646 |
| 7.45.4 实验现象..... | 646 |
| 7.46 开机TFT液晶屏图片自动左右移动实验 | 647 |
| 7.46.1 简要分析..... | 647 |
| 7.46.2 硬件设计..... | 647 |
| 7.46.3 软件分析..... | 647 |
| 7.46.4 下载与测试..... | 648 |
| 7.46.5 实验现象..... | 648 |
| 7.47 TFT触摸屏实验 | 648 |
| 7.47.1 实验的意义与作用..... | 649 |
| 7.47.2 实验原理..... | 649 |
| 7.47.3 硬件设计..... | 649 |

| | | |
|---------|----------------------------------|-----|
| 7.47.4 | 软件设计..... | 650 |
| 7.47.5 | 下载与现象..... | 654 |
| 7.48 | 图片跟触摸点移动实验 | 655 |
| 7.48.1 | 简要分析..... | 655 |
| 7.48.2 | 硬件设计..... | 655 |
| 7.48.3 | 软件分析..... | 655 |
| 7.48.4 | 下载与测试..... | 655 |
| 7.48.5 | 实验现象..... | 655 |
| 7.49 | 点击图片小图标显示小图标外框实验 | 656 |
| 7.49.1 | 简要分析..... | 656 |
| 7.49.2 | 硬件设计..... | 656 |
| 7.49.3 | 软件分析..... | 656 |
| 7.49.4 | 下载与测试..... | 657 |
| 7.49.5 | 实验现象..... | 657 |
| 7.50 | 点击图片小图标触发事件实验 | 657 |
| 7.50.1 | 简要分析..... | 657 |
| 7.50.2 | 硬件设计..... | 657 |
| 7.50.3 | 软件分析..... | 657 |
| 7.50.4 | 下载与测试..... | 658 |
| 7.50.5 | 实验现象..... | 658 |
| 7.51 | 双击小图标触发实验 | 659 |
| 7.51.1 | 简要分析..... | 659 |
| 7.51.2 | 硬件设计..... | 659 |
| 7.51.3 | 软件分析..... | 659 |
| 7.51.4 | 实验现象..... | 660 |
| 7.52 | TFT彩色液晶屏显示英文字 | 661 |
| 7.52.1 | 什么是字模..... | 661 |
| 7.52.2 | ASCII码的字符解释..... | 661 |
| 7.52.3 | 硬件设计..... | 664 |
| 7.52.4 | 软件分析..... | 664 |
| 7.52.5 | 下载与测试..... | 667 |
| 7.52.6 | 实验现象..... | 668 |
| 7.53 | TFT彩色液晶屏显示中文字 | 668 |
| 7.53.1 | 使用工具将中文字转换成二进制码..... | 668 |
| 7.53.2 | 硬件设计..... | 669 |
| 7.53.3 | 软件分析..... | 669 |
| 7.53.4 | 下载与测试..... | 671 |
| 7.53.5 | 实验现象..... | 671 |
| 7.54 | TFT彩色液晶屏显示中英文字 | 672 |
| 7.54.1 | 硬件设计..... | 672 |
| 7.54.2 | 软件分析..... | 672 |
| 7.54.3 | 下载与测试..... | 672 |
| 7.54.4 | 实验现象..... | 672 |
| 7.55 | SD卡访问实验 | 673 |
| 7.55.1 | SD卡的发展历程..... | 673 |
| 7.55.2 | SD卡的分类..... | 674 |
| 7.55.3 | SD卡的内部结构..... | 674 |
| 7.55.4 | SD卡模式选择..... | 675 |
| 7.55.5 | SD卡SPI模式命令分析..... | 675 |
| 7.55.6 | SD卡SPI模式的读写操作: | 677 |
| 7.55.7 | SD卡SDIO模式命令分析..... | 678 |
| 7.55.8 | SDIO模式下SD卡对host的各种命令的回复称为响应..... | 678 |
| 7.55.9 | SD卡有无文件系统的区别: | 679 |
| 7.55.10 | 实验原理..... | 679 |
| 7.55.11 | 硬件设计..... | 679 |

| | |
|-----------------------------------|-----|
| 7.55.12 软件设计..... | 681 |
| 7.55.13 下载与验证..... | 687 |
| 7.55.14 实验现象..... | 688 |
| 7.56 SD卡 FAT文件系统访问实验..... | 688 |
| 7.56.1 意义与作用..... | 688 |
| 7.56.2 文件系统浅析..... | 688 |
| 7.56.3 实验原理..... | 689 |
| 7.56.4 硬件设计..... | 690 |
| 7.56.5 软件设计..... | 690 |
| 7.56.6 下载与验证..... | 692 |
| 7.56.7 实验现象..... | 693 |
| 7.57 SD卡读卡器实验..... | 693 |
| 7.57.1 SD卡读卡器实验的意义与作用..... | 693 |
| 7.57.2 试验原理..... | 693 |
| 7.57.3 硬件设计..... | 693 |
| 7.57.4 软件设计..... | 699 |
| 7.57.5 下载与测试..... | 703 |
| 7.57.6 实验现象..... | 704 |
| 7.58 音频播放试验..... | 706 |
| 7.58.1 实验原理..... | 706 |
| 7.58.2 硬件设计..... | 706 |
| 7.58.3 软件设计..... | 709 |
| 7.58.4 下载与测试..... | 713 |
| 7.58.5 试验现象..... | 713 |
| 7.59 硬件CRC循环冗余检验实验..... | 714 |
| 7.59.1 意义与作用..... | 714 |
| 7.59.2 实验原理..... | 714 |
| 7.59.3 硬件设计..... | 715 |
| 7.59.4 软件设计..... | 715 |
| 7.59.5 下载与验证..... | 717 |
| 7.59.6 实验现象..... | 717 |
| 7.60 PVD电源电压监测实验..... | 719 |
| 7.60.1 意义与作用..... | 719 |
| 7.60.2 实验原理..... | 719 |
| 7.60.3 硬件设计..... | 720 |
| 7.60.4 软件设计..... | 720 |
| 7.60.5 下载与验证..... | 723 |
| 7.60.6 实验现象..... | 723 |
| 7.61 STM32 低功耗之--STANDBY待机模式..... | 724 |
| 7.61.1 意义与作用..... | 724 |
| 7.61.2 STM32 低功耗模式介绍..... | 724 |
| 7.61.3 待机模式..... | 724 |
| 7.61.4 进入待机模式..... | 725 |
| 7.61.5 待机模式寄存器介绍..... | 725 |
| 7.61.6 待机唤醒步骤..... | 727 |
| 7.61.7 硬件设计..... | 728 |
| 7.61.8 软件设计..... | 728 |
| 7.61.9 下载与验证..... | 733 |
| 7.61.10 实验现象..... | 734 |
| 7.62 STM32 低功耗之--STOP停止模式实验..... | 734 |
| 7.62.1 意义与作用..... | 734 |
| 7.62.2 实验原理..... | 734 |
| 7.62.3 硬件设计..... | 735 |
| 7.62.4 软件设计..... | 735 |
| 7.62.5 下载与验证..... | 743 |

| | | |
|---------|-------------------------|-----|
| 7.62.6 | 实验现象..... | 743 |
| 7.63 | DMA 传输实验..... | 743 |
| 7.63.1 | STM32 DMA 简介..... | 743 |
| 7.63.2 | 实验原理..... | 744 |
| 7.63.3 | 硬件设计..... | 745 |
| 7.63.4 | 软件设计..... | 745 |
| 7.63.5 | 下载验证..... | 748 |
| 7.64 | STM32 内部温度传感器实验..... | 749 |
| 7.64.1 | STM32 内部温度传感器简介..... | 749 |
| 7.64.2 | 实验原理..... | 749 |
| 7.64.3 | 硬件设计..... | 750 |
| 7.64.4 | 软件设计..... | 751 |
| 7.64.5 | 下载验证..... | 754 |
| 7.65 | 温度传感器 (DS18B20) 实验..... | 754 |
| 7.65.1 | 意义与作用..... | 754 |
| 7.65.2 | 温度传感器的介绍..... | 755 |
| 7.65.3 | DS18B20 温度传感器的使用..... | 758 |
| 7.65.4 | 软件设计..... | 761 |
| 7.65.5 | 硬件设计与实验现象..... | 769 |
| 7.66 | DHT11 数字温湿度传感器实验..... | 770 |
| 7.66.1 | 简介..... | 770 |
| 7.66.2 | DHT11 的工作原理..... | 770 |
| 7.66.3 | 硬件连接与温度协议分析..... | 771 |
| 7.66.4 | 实验现象..... | 773 |
| 7.67 | 三轴加速度模块实验..... | 774 |
| 7.67.1 | 三轴加速传感器有什么用..... | 774 |
| 7.67.2 | ADXL345 简介..... | 774 |
| 7.67.3 | ADXL345 工作原理..... | 775 |
| 7.67.4 | ADXL345 的通信方式实验原理..... | 776 |
| 7.67.5 | ADXL345 的寄存器说明..... | 777 |
| 7.67.6 | ADXL345 内部结构与管脚连接..... | 778 |
| 7.67.7 | 实验原理..... | 780 |
| 7.67.8 | 硬件设计..... | 780 |
| 7.67.9 | 软件设计..... | 780 |
| 7.67.10 | 实验现象..... | 785 |
| 7.68 | 串口IAP在线升级实验..... | 786 |
| 7.68.1 | 意义与作用..... | 786 |
| 7.68.2 | 关于什么是IAP..... | 786 |
| 7.68.3 | 各种烧录方式的比较..... | 786 |
| 7.68.4 | 串口IAP的内部实现流程..... | 787 |
| 7.68.5 | 实验现象..... | 793 |
| 7.68.6 | 代码分析..... | 793 |
| 7.69 | MP3(VS1003)音频模块实验..... | 799 |
| 7.69.1 | MP3 介绍..... | 799 |
| 7.69.2 | VS1003 解码芯片介绍..... | 800 |
| 7.69.3 | 硬件设计..... | 808 |
| 7.69.4 | 软件设计..... | 809 |
| 7.69.5 | 下载与测试..... | 812 |
| 7.69.6 | 实验现象..... | 812 |
| 7.70 | 红外遥控器实验..... | 813 |
| 7.70.1 | 什么是红外? 红外的历史..... | 813 |
| 7.70.2 | 红外的特点和用途..... | 814 |
| 7.70.3 | 红外的功能参数..... | 814 |
| 7.70.4 | 红外的工作原理..... | 814 |
| 7.70.5 | 实验原理..... | 817 |

| | |
|--|-----|
| 7.70.6 硬件设计..... | 817 |
| 7.70.7 代码分析..... | 818 |
| 7.70.8 下载与测试..... | 821 |
| 7.70.9 实验现象..... | 821 |
| 7.71 口程序实验 | 822 |
| 7.71.1 ENC28J60 网口程序实验的意义与作用..... | 822 |
| 7.71.2 实验原理 | 822 |
| 7.71.3 硬件设计..... | 824 |
| 7.71.4 软件设计..... | 826 |
| 7.71.5 下载与测试..... | 830 |
| 7.71.6 实验现象..... | 831 |
| 7.72 UCOSII操作系统之单任务运行 | 832 |
| 7.72.3 UCOSII简单介绍..... | 832 |
| 7.72.4 UCOSII的学习方法..... | 833 |
| 7.72.5 神舟官方团队点破操作系统的学习精妙之处..... | 834 |
| 7.72.6 实验原理..... | 834 |
| 7.72.7 硬件设计..... | 834 |
| 7.72.8 软件设计..... | 835 |
| 7.72.9 下载与测试..... | 837 |
| 7.73 UCOSII操作系统多任务运行 | 837 |
| 7.73.3 SYSTICK时钟介绍..... | 837 |
| 7.73.4 实验原理..... | 837 |
| 7.73.5 硬件设计..... | 837 |
| 7.73.6 软件设计..... | 838 |
| 7.73.7 下载与测试..... | 839 |
| 7.74 UCOS_UCGUI_DEMO实验 | 839 |
| 7.75 1602 液晶屏模块 | 839 |
| 7.75.3 7.80.1 1602 字符型液晶屏简介..... | 839 |
| 7.75.4 1602 液晶屏显示的基本原理..... | 841 |
| 7.75.5 如何控制 1602 液晶屏（寄存器的介绍） | 841 |
| 7.75.6 硬件连接原理..... | 850 |
| 7.75.7 7.80.5 代码分析..... | 852 |
| 7.75.8 7.80.6 下载与验证..... | 854 |
| 7.75.9 7.80.7 实验现象..... | 854 |
| 7.76 4.3 寸液晶屏模块 | 855 |
| 7.76.3 神舟III号_刷屏测试(4.3') 实验现象..... | 855 |
| 7.76.4 神舟III号_触摸测试(4.3') 实验现象..... | 855 |
| 7.66.1 神舟III号_TFT绘图API(4.3')实验现象..... | 856 |
| 7.66.5 神舟III号_HZK16(4.3'卡在屏上) 实验现象..... | 856 |
| 7.66.4 神舟III号_电子相框实验现象..... | 857 |
| 7.77 7 寸液晶屏模块 | 858 |
| 7.77.1 神舟III号_TFT刷屏测试(7 寸) 实验现象..... | 858 |
| 7.77.2 神舟III号_TFT绘图API(7 寸) 实验现象..... | 858 |
| 7.77.3 神舟III号_触摸屏测试(7 寸) 实验现象..... | 859 |
| 7.77.4 神舟III号_电子相框_BIN(7'卡在屏上) 实验现象..... | 860 |
| 附件 1：如何从单片机零基础到嵌入式高手 | 861 |
| 附件 2：STM32 神舟系列其他开发板介绍 | 864 |
| 1.神舟 51+ARM (STM32F103C8T) 单片机开发板..... | 864 |
| 2. STM32 神舟I号 (STM32F103RBT) 开发板..... | 865 |
| 3. STM32 神舟III号 (STM32F103ZET) 开发板..... | 867 |
| 4. STM32 神舟III号 (STM32F103ZET) 开发板..... | 871 |

| | |
|--|-----|
| 5. STM32 神舟王 103 开发板 (STM32F103ZET) | 873 |
| 6. STM32 神舟王 207 开发板 (STM32F207ZGT) | 876 |
| 7. STM32 神舟王 407 开发板 (STM32F407ZGT) | 878 |
| 8. STM32 神舟王 407IGT 开发板+130W 摄像头 (STM32F407IGT) | 879 |
| 9. STM32 神舟王 417IGT 开发板+130W 摄像头 (STM32F417IGT) | 880 |
| 10. STM32 神舟王 439IGT 开发板+130W 摄像头 (STM32F439IGT) | 881 |
| 11. STM32F103/207/407/439 多个核心板共同神舟王底板---真正一板变多板 | 884 |
| 附件 3: STM32 神舟团队 20 年工作经验心得总结 | 884 |
| 1) 需求定义..... | 885 |
| 2) 处理器的选择..... | 887 |
| 3) 开发成本的预测和估计..... | 892 |
| 4) 产品开发设计文档 (需要包括硬件和软件两个方面) | 892 |
| 5) 嵌入式高手对技术的理解 (含辛茹苦这么多年的精华体验) | 894 |
| 附件 4: PCB 设计建议 | 895 |
| 1. PCB 设计干扰的相关基础知识 | 895 |
| 2. 电磁干扰三要素 | 895 |
| 3. 印制电路板 | 896 |
| 3. 器件位置 | 897 |
| 4. 接地和供电 (VSS, VDD) | 897 |
| 5. 数字电路与模拟电路的供地处理 | 897 |
| 6. 信号线布在电或地层上 | 897 |
| 7. 焊盘与产品良品率质量的关系 | 898 |
| 8. 其他信号的注意事项 | 898 |
| 9. 未用到的 I/O 管脚 | 898 |
| 附件 5: 项目合作与技术支持联系方式 | 898 |

第一篇 硬件篇

本章将详细介绍神舟III号STM32开发板的硬件，从神舟III号的整体硬件资源，产品特点到各个硬件模块，进行详细的介绍。通过本章节，使大家对该开发板的整个功能特点有所了解，同时熟悉各个功能模块的硬件实现方案。

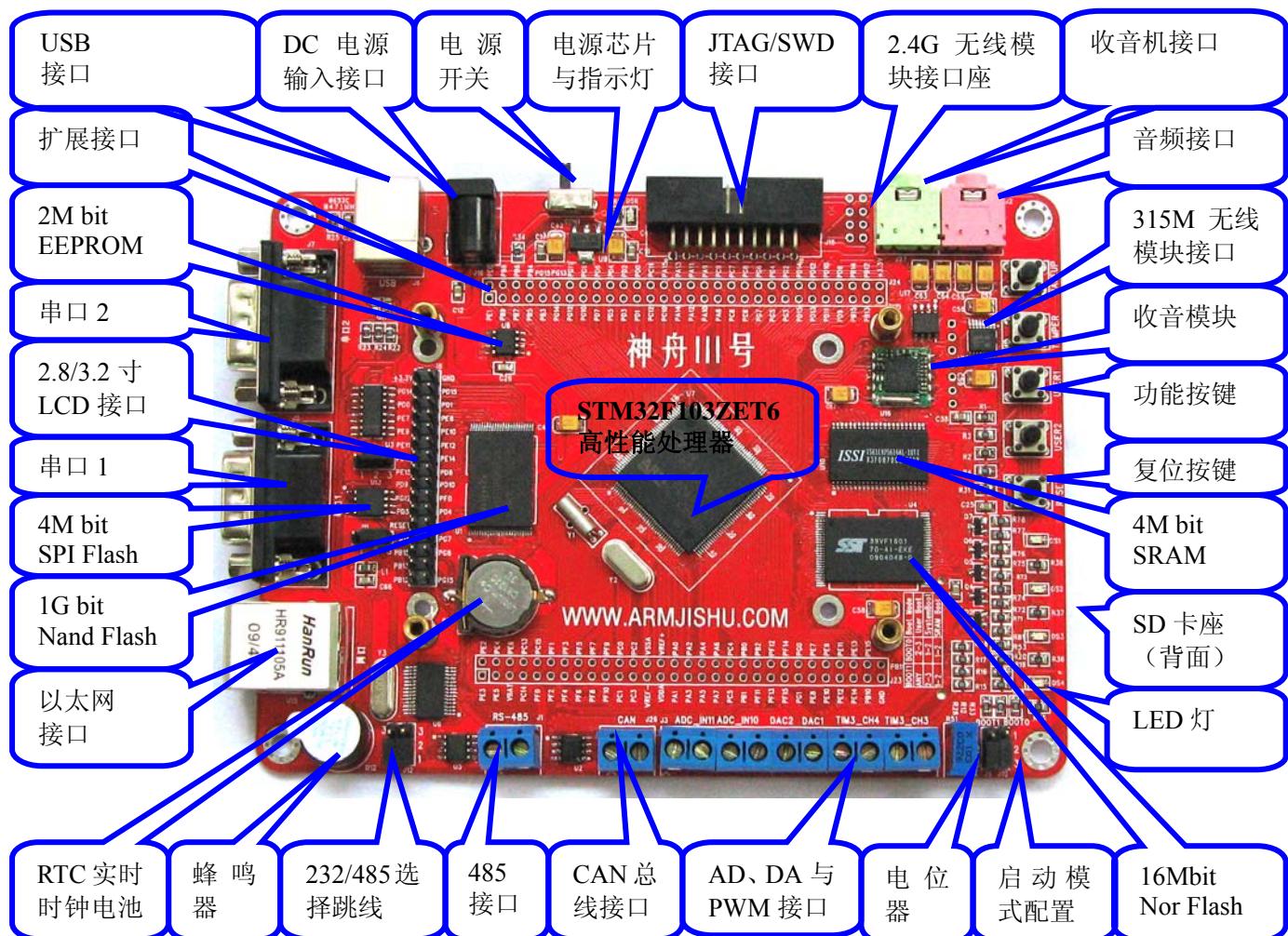
本章主要分为如下几个部分：

- 1.1. 神舟III号STM32开发板简介；
- 1.2. 神舟III号STM32开发板硬件详解；
- 1.3. 神舟III号STM32开发板使用注意事项；

1.1. 神舟III号STM32开发板简介

神舟III号是一款基于STM32F103ZET6处理器的STM32开发板，面向广大的企业客户和STM32爱好者。开发板功能强大，外围资源齐全。此外，还预留了丰富的扩展接口，可以灵活的扩展各种功能。而整板尺寸仅仅110mm*150mm，非常小巧，方便携带。

神舟III号STM32开发板的产品外观及对应各功能模块说明如图表1所示：



图表 1 神舟 III 号开发板外观与功能

神舟 III 号开发板板载硬件资源如下：

- ◆ STM32F103系列最高端配置芯片STM32F103ZET6。Cortex-M3内核32位处理器,72M主频, LQFP144封装, 片内Flash容量:512K,片内SRAM容量:64K
- ◆ 标配1G比特容量的Nand Flash
- ◆ 标配16M比特容量的Nor Flash, 最大支持128M比特容量Nor Flash
- ◆ 标配4M比特容量的SRAM
- ◆ 标配2M比特容量的I2C接口的EEPROM芯片
- ◆ 标配16M比特容量的SPI Flash芯片
- ◆ 采用主流收音机模块, 提供收音机功能
- ◆ 采用专用I2S音频DA芯片, 提供音频播放功能
- ◆ 1个USB SLAVE接口, 支持USB过流保护与USB接口静电防护, 符合ESD防护标注 IEC61000-4-2(ESD 15kV air, 8kV Contact)
- ◆ 1个10M以太网接口, 用于以太网通信
- ◆ 1个SD卡接口
- ◆ 1个标准的2.8/3.2寸LCD接口, 支持触摸屏, 分辨率320X240, 26万色
- ◆ 1个复位按钮, 控制整板硬件复位
- ◆ 4个功能按钮, 包括WAKEUP与TAMPER按键
- ◆ 4个状态指示灯 (DS1~DS4: 绿色)
- ◆ 1个电源指示灯 (绿色)
- ◆ 1个5V外部电源输入接口 (内正外负)
- ◆ 1个电源开关, 控制整个板的电源开关
- ◆ 1个标准的JTAG/SWD调试下载口, 支持JLINK供电
- ◆ 2个串行接口 (DB9 公头)
- ◆ 1个485总线接口
- ◆ 1个CAN总线接口
- ◆ 1个电位器接口
- ◆ 1路蜂鸣器
- ◆ 1个启动模式选择配置接口
- ◆ 支持2.4G无线通信模块
- ◆ 支持315M无线通信模块
- ◆ 1个RTC后备电池座, 支持CR1220纽扣电池
- ◆ 除晶振占用的IO口外, 其余所有IO口全部引出
- ◆ 预留AD, DC, PWM接口

从上面的板载资源可以看出, 神舟III号开发板板载资源非常丰富, 即包括STM32爱好者常用的硬件资源, 又包括各种工业接口。此外, 开发板还将处理器所有GPIO接口通过双排插针接口引出, 非常方便产品的功能扩展和其他功能模块调试, 让你的开发变得更加简单。

神舟III号开发板的特点包括:

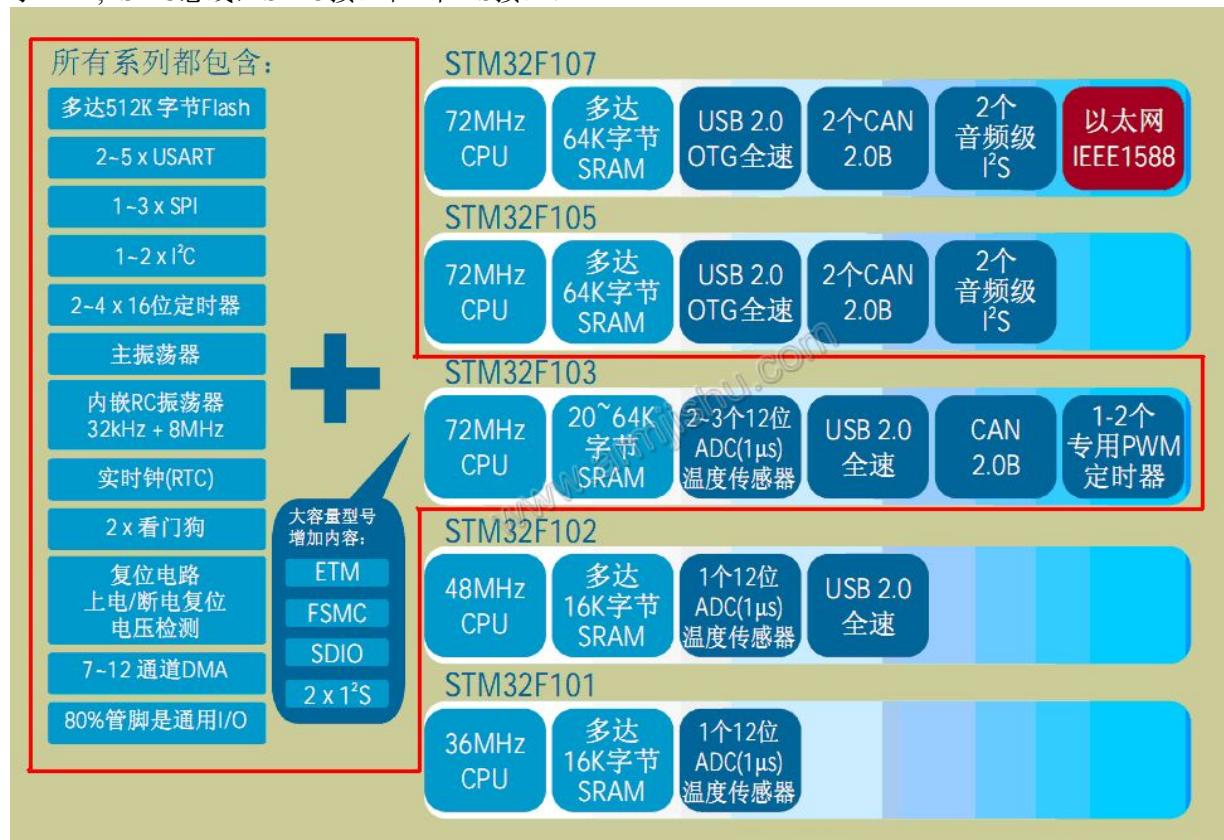
- 1) 外观大气。整个板子尺寸为110mm*150mm*20mm (包括液晶, 但不计算铜柱的高度)。
- 2) 设计灵活。板上除晶振外的所有的IO口全部引出, 可以极大的方便大家扩展及使用。
- 3) 资源丰富。板载十多种外设及接口, 让你畅游STM32。
- 4) 调试方便。和主流调试仿真工具JLINK V8完美结合, 让您快速找到代码的BUG。
- 5) 触摸彩屏。320X240分辨率, 26万色TFT LCD, 带触摸功能, 让您设计出迷人的GUI。
- 6) 畅游网络。支持10M以太网, 支持Ping、HTTP等, 缩短您的开发周期, 而且支持网络固件更新, 降低您的维护成本。
- 7) 教程齐全。共计近二十个实例, 使用ST标准库, 方便用户修改升级。

接下来我们详细介绍神舟III号开发板的各个功能模块。

1) STM32F103ZET6 处理器

开发板使用了STMF103系列中的最高配置的Cortex-M3内核32位处理器STM32F103ZET6, 72M主频, LQFP144封装, 片内Flash容量:512K, 片内SRAM容量:64K。

STM32家族主要产品系列家谱如下图所示, STM32F103ZET6与同系列的其他处理器相比, 增加了ETM,FSMC总线, SDIO接口和2个I2S接口。



图表 2 STM32 家族主要产品系列家谱

STM32F103的产品列表出下图所示, 神舟III号开发板选用的是外设资源和管脚资源最丰富的144脚LQFP封装的STM32F103ZET6芯片, 充分满足企业和广大爱好者的评估开发需求。

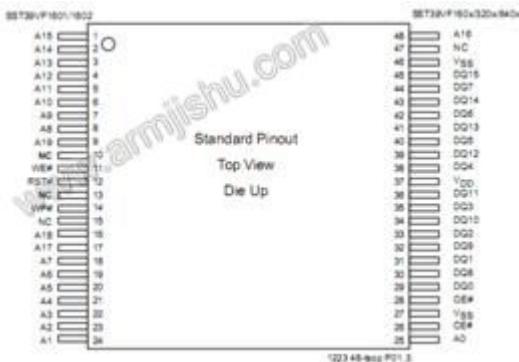
| 型号 | CPU 频率 (MHz) | 程序 空间 (字节) | RAM (字节) | FSMC | 定时器功能 ^① | | | | 串行通信接口 | | | | | | 模拟端口 | | I/O 端口 | 封装 |
|------|--------------------|------------------|-------------|------|--------------------|--------------------|-----------|-----|------------------|--------------------|-----------|------------|---------|----|--------|-------------|-------------|-----------------|
| | | | | | 16位普通 (OC/OPWM) | 16位高级 (OC/OPWM) | 16位 基本 | SPI | I ² C | USART ^② | USB 全速 | CAN 2.0 | 以太 网 | FS | SDIO | ADC (通道) | DAC (通道) | |
| 36脚 | STM32F103T4 | 72 | 16K | 6K | 2(8/8/8) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(10) | 26 | VQFPN36 |
| | STM32F103T6 | 72 | 32K | 10K | 2(8/8/8) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(10) | 26 | VQFPN36 |
| | STM32F103T8 | 72 | 64K | 20K | 3(12/12/12) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(10) | 26 | VQFPN36 |
| 48脚 | STM32F103C4 | 72 | 16K | 6K | 2(8/8/8) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(10) | 37 | LQFP48 |
| | STM32F103C6 | 72 | 32K | 10K | 2(8/8/8) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(10) | 37 | LQFP48 |
| | STM32F103C8 | 72 | 64K | 20K | 3(12/12/12) | 1(4/4/6) | | 2 | 2 | 3 | 1 | 1 | | | | 2/(10) | 37 | LQFP48 |
| | STM32F103CB | 72 | 128K | 20K | 3(12/12/12) | 1(4/4/6) | | 2 | 2 | 3 | 1 | 1 | | | | 2/(10) | 37 | LQFP48 |
| 64脚 | STM32F103R4 | 72 | 16K | 6K | 2(8/8/8) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(16) | 51 | LOFP64/TFBGA64 |
| | STM32F103R6 | 72 | 32K | 10K | 2(8/8/8) | 1(4/4/6) | | 1 | 1 | 2 | 1 | 1 | | | | 2/(16) | 51 | LOFP64/TFBGA64 |
| | STM32F103R8 | 72 | 64K | 20K | 3(12/12/12) | 1(4/4/6) | | 2 | 2 | 3 | 1 | 1 | | | | 2/(16) | 51 | LOFP64/TFBGA64 |
| | STM32F103RB | 72 | 128K | 20K | 3(12/12/12) | 1(4/4/6) | | 2 | 2 | 3 | 1 | 1 | | | | 2/(16) | 51 | LOFP64/TFBGA64 |
| | STM32F103RC | 72 | 256K | 48K | 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(16) | 1(2) | 51 | LOFP64/WLCP64 |
| | STM32F103RD | 72 | 384K | 64K | 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(16) | 1(2) | 51 | LOFP64/WLCP64 |
| | STM32F103RE | 72 | 512K | 64K | 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(16) | 1(2) | 51 | LOFP64/WLCP64 |
| 100脚 | STM32F103V8 | 72 | 64K | 20K | 3(12/12/12) | 1(4/4/6) | | 2 | 2 | 3 | 1 | 1 | | | | 2/(16) | 80 | LOFP100/LBGA100 |
| | STM32F103VB | 72 | 128K | 20K | 3(12/12/12) | 1(4/4/6) | | 2 | 2 | 3 | 1 | 1 | | | | 2/(16) | 80 | LOFP100/LBGA100 |
| | STM32F103VC | 72 | 256K | 48K | ● 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(16) | 1(2) | 80 | LOFP100/LBGA100 |
| | STM32F103VD | 72 | 384K | 64K | ● 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(16) | 1(2) | 80 | LOFP100/LBGA100 |
| | STM32F103VE | 72 | 512K | 64K | ● 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(16) | 1(2) | 80 | LOFP100/LBGA100 |
| 144脚 | STM32F103ZC | 72 | 256K | 48K | ● 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(21) | 1(2) | 112 | LOFP144/LBGA144 |
| | STM32F103ZD | 72 | 384K | 64K | ● 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(21) | 1(2) | 112 | LOFP144/LBGA144 |
| | STM32F103ZE | 72 | 512K | 64K | ● 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(21) | 1(2) | 112 | LOFP144/LBGA144 |

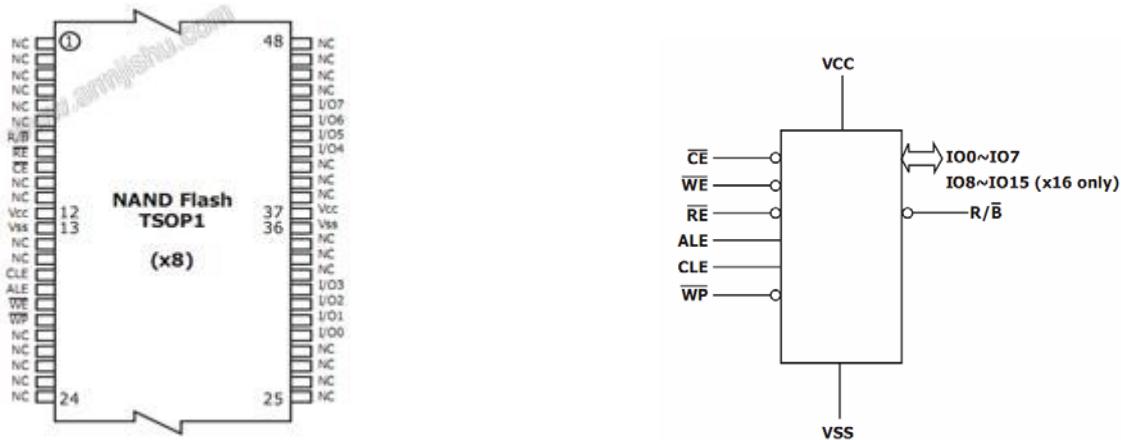
图表 3 STM32F103 产品列表

2) Nor Flash

Nor的特点是芯片内执行，这样应用程序可以直接在flash闪存内运行，不必再把代码读到系统RAM中。另外，Nor的传输效率很高，在1~4MB的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

神舟III号通过FSMC总线扩展了16M比特容量的Nor Flash，最大支持128M比特容量Nor Flash。可用于保存操作系统及一些重要数据。

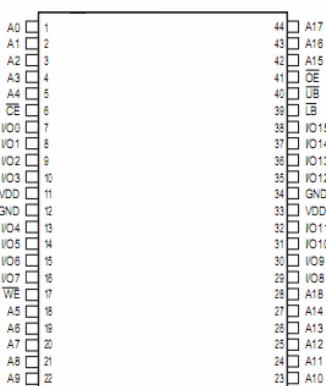




4) SRAM

SRAM是Static Random Access Memory的缩写，中文含义为静态随机访问存储器，它是一种类型的半导体存储器。“静态”是指只要不掉电，存储在SRAM中的数据就不会丢失。这一点与动态RAM (DRAM) 不同，DRAM需要进行周期性的刷新操作。

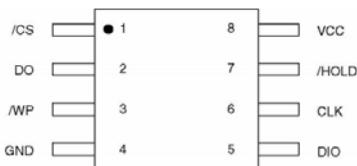
神舟III号通过FSMC总线扩展了8M比特容量的SRAM，可用于系统或程序运行过程中的临时数据的存取。



5) SPI Flash

SPI Flash存储器它具有掉电数据不丢失、快速数据存取速度、电可擦除、容量大、在线可编程、价格低廉以及足够多的擦写次数（一百万次）和较高的可靠性等诸多优点，在嵌入式应用得到广泛引用。

神舟III号板载了一片16M比特容量的SPI Flash芯片W25X16，非常适合我们存储一些不常修改的数据。

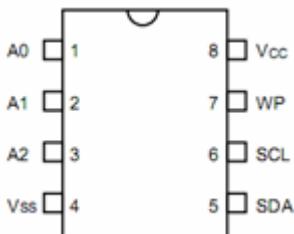


6) SD 卡接口

除了前面提到的SPI Flash以外，神舟III号还自带SD卡接口。SD卡作最常见的存储设备，是很多数码设备的存储媒介，比如数码相框、数码相机、MP5等。有了它，我们的开发板就相当于拥有了一个大容量的外部存储器，不单可以用来提供数据，也可以用来存储数据，使得我们的板子可以完成更多的功能。

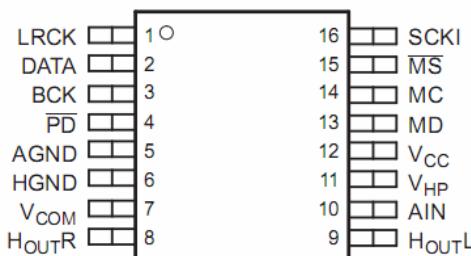
7) I2C EEPROM

用于掉电数据保存，因为STM32内部没有EEPROM，我们这里外扩了24C02，可用于设备的一些配置数据，或一些不需要经常修改的数据保存。



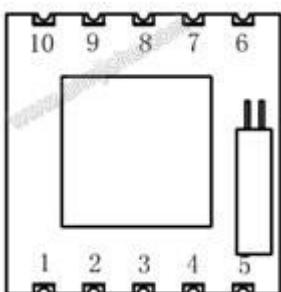
8) 音频播放电路

神舟III号STM32开发板具有音频播放功能，可以将保存在Flash中的音频文件通过专用的解码芯片播放出来，在STM32上也可以听到美妙的歌声，或者给产品播放提示音。



9) 收音机

采用主流的收音机模块，TEA5767实现收音机功能，通过这个模块，可了解收音机的基本原理与TEA5767芯片的配置与访问。



10) 以太网口

神舟III号开发板提供了一个标准的10M以太网接口。通过这个接口，我们可以将神舟III号也接到以太网上，与其他网络设备进行通信。

11) 2.8/3.2LCD 接口

该接口是一个目前比较通用的LCD液晶触摸屏接口，一个32芯LCD接口引出了LCD控制器和触摸屏的全部信号，它的线序兼容市面上在售的主流触摸屏模块，比如ARMJISHU.COM推出的液晶模块、红牛开发板使用的液晶模块、STMSKY开发板使用的液晶模块等。

320 X 240的显示分辨率64万色可以逼真的显示图片、文字和菜单等，配合触摸功能实现灵活的控制，我们提供已经调试成功的LCD液晶屏和触摸屏的示例代码。

该接口是一个目前比较通用的LCD接口，可以用来接很多市面上在售的液晶模块，比如ARMJISHU.COM推出的液晶模块、红牛开发板使用的液晶模块、STMSKY开发板使用的液晶模块等都可以使用这个接口。

12) BOOT 选择跳帽

STM32F10x处理器一共有三种启动模式，可以通过这一跳线进行选择，具体的跳帽设置与启动模式选择关系如下表。

| BOOT1 (J9) | BOOT0 (J10) | 功能 |
|------------|-------------|---------------|
| ANY | 2-3 | User Boot(默认) |
| 2-3 | 1-2 | System Boot |
| 1-2 | 1-2 | SRAM Boot |

13) RTC 实时时钟

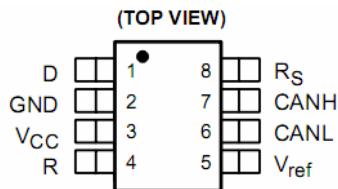
STM32内部RTC功能非常实用，它的供电和时钟是独立于内核的，可以说是STM32内部独立的外设模块，RTC内部寄存器不受系统复位掉电的影响。

在神舟III号STM32开发板上，我们采用外部电池供电和32768表振晶体来实现真正RTC（实时时钟）功能。同时，如果RTC电池没有安装的情况下，可以通过跳线帽来去掉RTC时钟功能，而不影响系统正常运行。

14) CAN 总线接口

CAN 是Controller Area Network 的缩写（以下称为CAN），是ISO国际标准化的串行通信协议。它的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。CAN总线当今自动化领域技术发展的热点之一，被誉为自动化领域的计算机局域网。它的出现为分布式控制系统实现各节点之间实时、可靠的数据通信提供了强有力的技术支持。

在神舟III号STM32开发板中，通过VP230这一款3.3VCAN总线收发器将STM32的CAN总线接口引出，VP230芯片的管脚视图如下：



15) 蜂鸣器

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。用于产品的声音提醒或者告警。

神舟III号STM32开发板，板载一个蜂鸣器。当接通蜂鸣器两端的电源时，蜂鸣器发出固定频率的声音。

16) 电位器

神舟III号STM32开发板，提供了一路电位器，通过这一路电位器可以学习STM32处理器的AD功能。

17) LED 灯

神舟III号提供了4路LED指示灯，分别与GPIOF6-9四个管脚连接，当管脚为低电平时，对应的LED指示灯亮。4路LED灯可以用于STM32开发板的状态指示。

18) 复位按钮

该按钮是神舟III号的整板硬件复位按钮，当按键按下时，STM32处理器，液晶，以太网，以及音频解码芯片都将复位。

19) 通用按钮

神舟III号，除了前面提到的复位按键这一特殊功能按键外，还提供四个按键，其中两个按键分别为WAKEUP唤醒按键，以及TAMPER按键，另两个为用户可自定义功能的通用按键USER1和USER2。（说明：这四个按键，可依据实际产品，定义成产品的功能按键，包括WAKEUP按键与TAMPER按键）

20) JTAG/SWD 接口

标准的20针JTAG，直接可以和ULINK或者JLINK连接的，同时支持SWD（因为STM32支持SWD）。可以用于调试STM32，更方便的开发软件。

21) 电源开关

控制3.3V供电，如果通过选择由USB供电或外部电源输入，实现电源的通断。

22) 电源芯片与电源指示灯

由于STM32F103ZET6是3.3V的工作电压，因此，我们需要将外部电源输入或者USB提供的5V电

源转换成3.3V的工作电压。在神舟III号这一块开发板上，采用了ASM1117-3.3V这一常用电源转换芯片来实现这一功能。另外，为了方便用户确认电源是否已经正常提供，开发板还提供了一个绿色的电源指示灯，当开发板上3.3V电源正常时，指示灯亮。

23) 5V 直流电源输入接口

DC电源座符合常见电源适配器接口标准，DC电源座的里面为电源正极，外面为电源负极。

24) USB 接口

标准的USB SLAVE接口，其他主设备，如PC电脑可通过此接口与神舟III号通讯，另外，该接口也可以作为神舟III的电源输入，由PC或其他USB主设备提供电源。为了保证设备可以正常工作，USB接口具有一个500自恢复保险丝，当工作电流大于500mA时，以及ESD防护器件，符合ESD防护标注IEC61000-4-2(ESD 15kV air, 8kV Contact)。

25) 串口 1

此处将STM32F103ZET6的USART1通过MAX3232转换成232电平的串口，接口方式为DB9公头，实现了一个3线串口。管脚定义如下：

| 管脚 | 信号 |
|----|-----|
| 2 | RXD |
| 3 | TXD |
| 5 | GND |

26) 串口 2

串口2与串口1的实现方式相同，主要的差别在于，我们可以通过跳帽选择处理器的USART2实现的是串口2还是485总线接口。

27) RS-485 接口

RS-485总线接口，采用SP3485VP230接口芯片实现。

28) RS-232 与 RS-485 选择跳帽

神舟III号STM32开发板串口2可以选择实现RS-232接口或者RS-485接口。具体选择关系如下表。

| J12 | J14 | 功能 |
|-----|-----|----------|
| 1-2 | 1-2 | RS-485接口 |
| 2-3 | 2-3 | RS-232接口 |

29) 特殊功能接口

神舟III号STM32开发板，将STM32处理器部分具有特殊功能的管脚连接到预留接口，提供一些可扩展的功能。预留接口的管脚定义如下。

| 管脚号 | 功能 | 管脚号 | 功能 |
|-----|----------|-----|----------|
| 1 | GND | 6 | DAC1 |
| 2 | ADC_IN11 | 7 | GND |
| 3 | ADC_IN10 | 8 | TIM3_CH4 |
| 4 | GND | 9 | TIM3_CH3 |
| 5 | DAC2 | 10 | GND |

30) 扩展接口

神舟III号将所有的GPIO的使用标准双排插针引出，方便大家的实验和测试，调试其他模块或功能扩展。

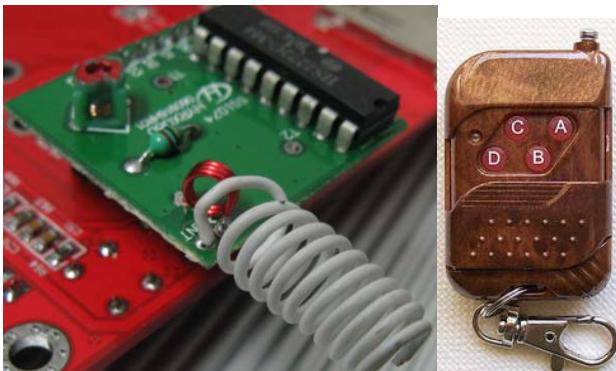
31) 2.4G 无线模块（可选配件）

神舟III号提供了1个2.4G模块接口，可以直接支持深圳云佳科技的NRF24L01模块。本模块作为选配件。产品图片如下。



32) 315M 无线模块（可选配件）

神舟III号提供了1个315M模块接口，可通过遥控控制进行远程控制，相关产品和模块如下所示。



1.2. 神舟III号开发板硬件详解

详细介绍神舟III号各功能模块的硬件实现与原理。

1.2.1. 处理器最小系统

神舟III号处理器选择的是STM32F103ZET6，STM32F103的型号众多，作为一款中高端开发板，我们选择了其中最高配置的型号，让用户可以使用STM32F103系列的所有外设，和体验它的强悍功能。MCU部分原理图如下：

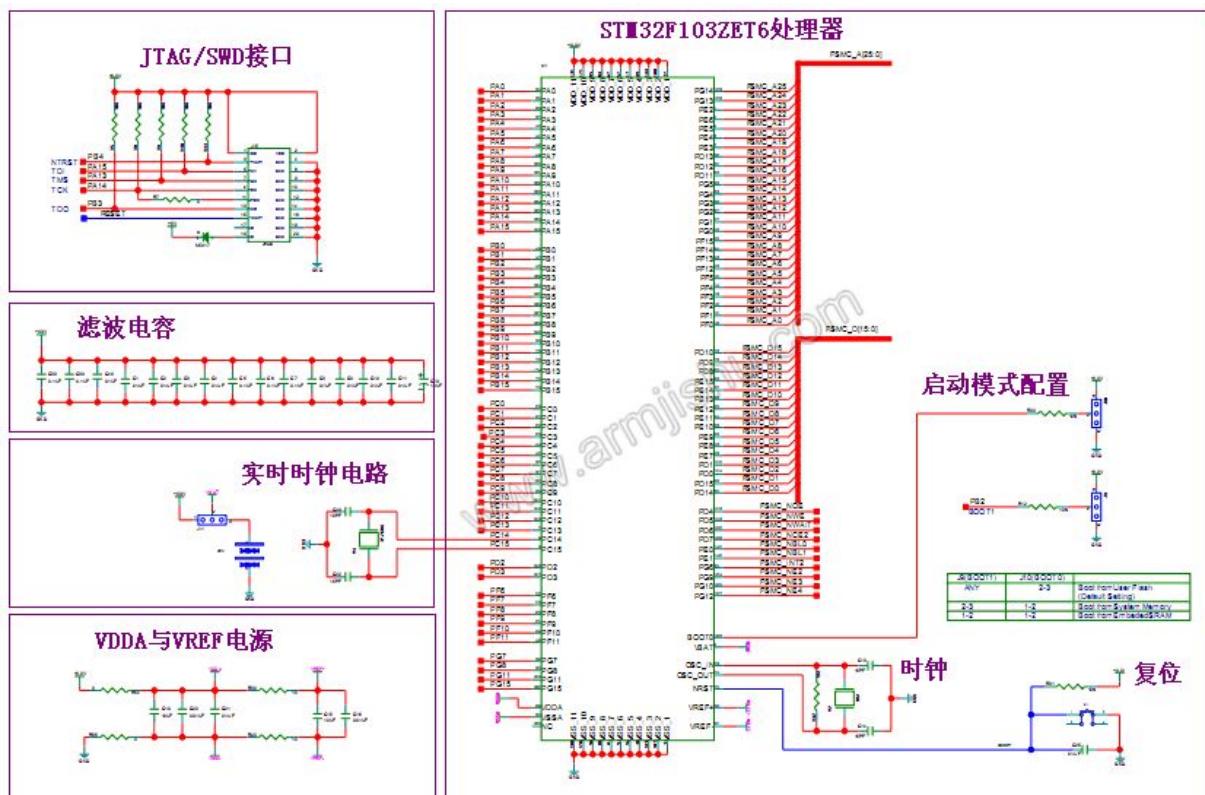
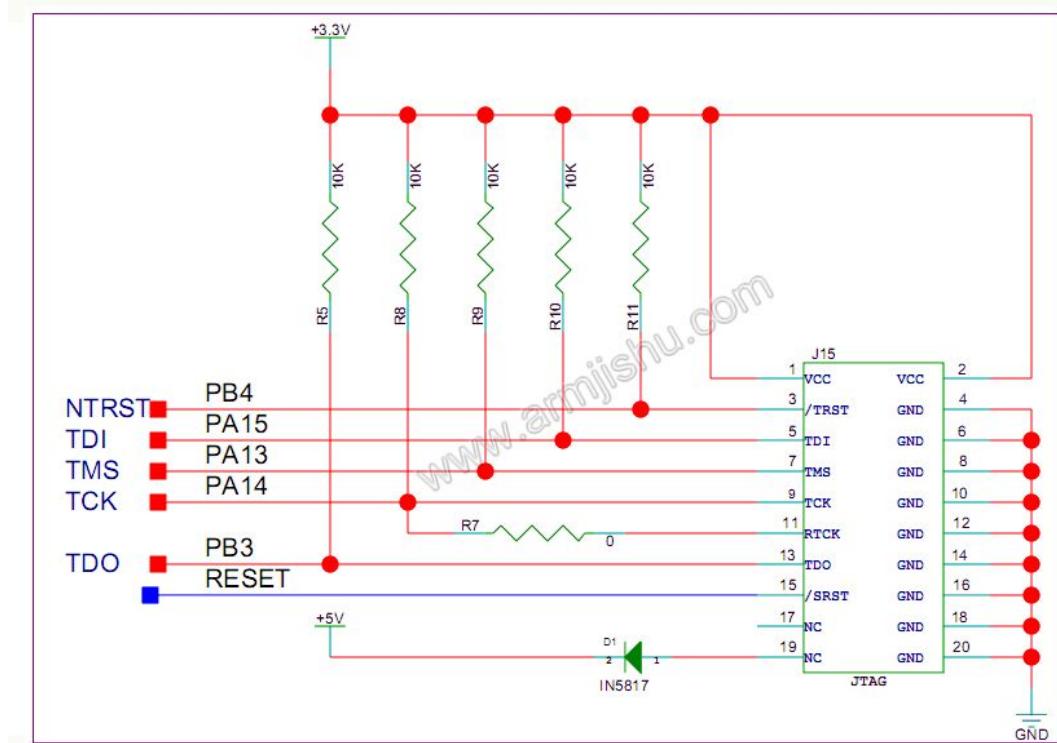


图1.2.1.1 MCU部分原理图

◆ JTAG/SWD接口

JTAG(Joint Test Action Group)联合测试行动小组)是一种国际标准测试协议(IEEE 1149.1兼容)。标准的JTAG接口包括TMS、TCK、TDI和TDO，通过JTAG接口，我们可以烧录和调试程序，神舟III号的JTAG接口的硬件连接如下图所示，可以与目前主流的JLINK V8仿真器配合使用。另外STMM32还有SWD接口，SWD只需要最少2跟线(SWCLK和SWDIO)就可以下载并调试代码了，它与JTAG接口是共用的，只要接上JTAG，你就可以使用SWD模式了。

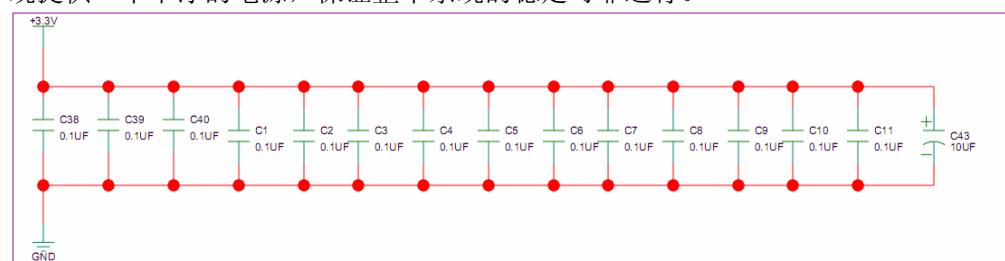


JTAG/SWD接口的信号定义如下：

| SWJ-DP pin name | JTAG debug port | | SW debug port | | Pin assignment |
|-----------------|-----------------|--------------------------|---------------|------------------------------------|----------------|
| | Type | Description | Type | Debug assignment | |
| TMS/SWDIO | I | JTAG Test Mode Selection | I/O | Serial Wire Data Input/Output | PA13 |
| TCK/SWCLK | I | JTAG Test Clock | I | Serial Wire Clock | PA14 |
| TDI | I | JTAG Test Data Input | - | - | PA15 |
| TDO/TRACESWO | O | JTAG Test Data Output | - | TRACESWO if async trace is enabled | PB3 |
| NTRST | I | JTAG Test nReset | - | - | PB4 |

◆ 滤波电容

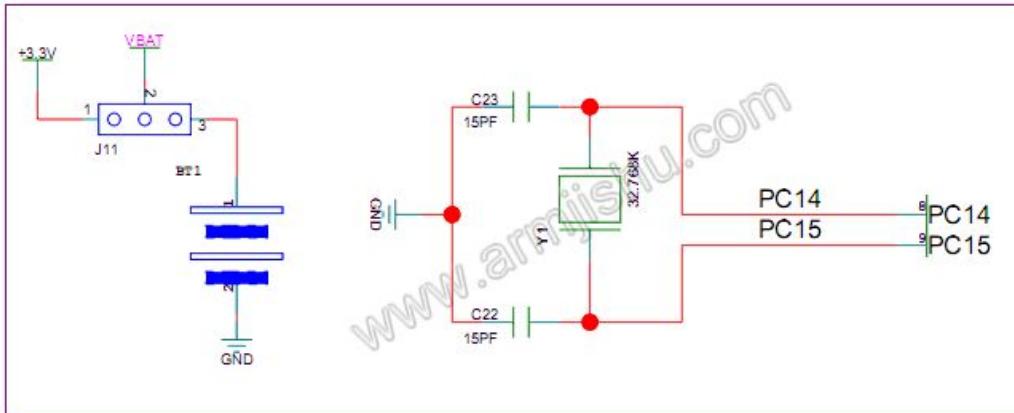
为STM32F103ZET6及神舟III号开发板上其他芯片提供必要的储能电容，滤波和退耦。为整个系统提供一个干净的电源，保证整个系统的稳定可靠运行。



◆ 实时时钟

STM32的VBAT采用CR1220纽扣电池和VCC3.3混合供电的方式，在有外部电源(VCC3.3)的时候，CR1220不给VBAT供电，而在外部电源断开的时候，则由CR1220给VBAT供电。这样，VBAT总

是有电的，以保证RTC的走时以及后备寄存器的内容不丢失。相关电路如下：



当安装了电池后，将J11的2, 3脚使用跳线帽短接。VBAT管脚由电池供电，如没有安装电池，将J11的1, 2脚使用跳线帽短接，VBAT管脚由+3.3V系统电源供电。

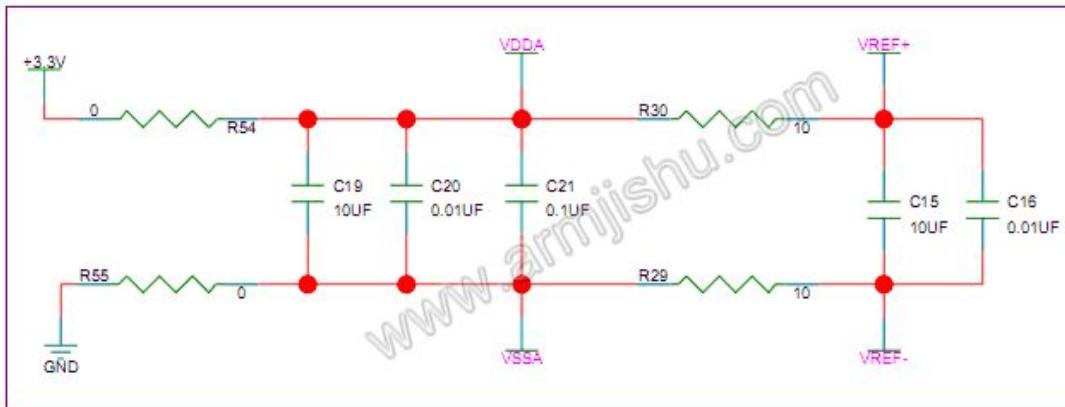
◆ VDDA与VREF电源

VDDA与VREF是STM32处理器数字/模拟转换(ADC)电路需要使用到的模拟参考电源和模拟电源输入。

相关管脚定义如下：

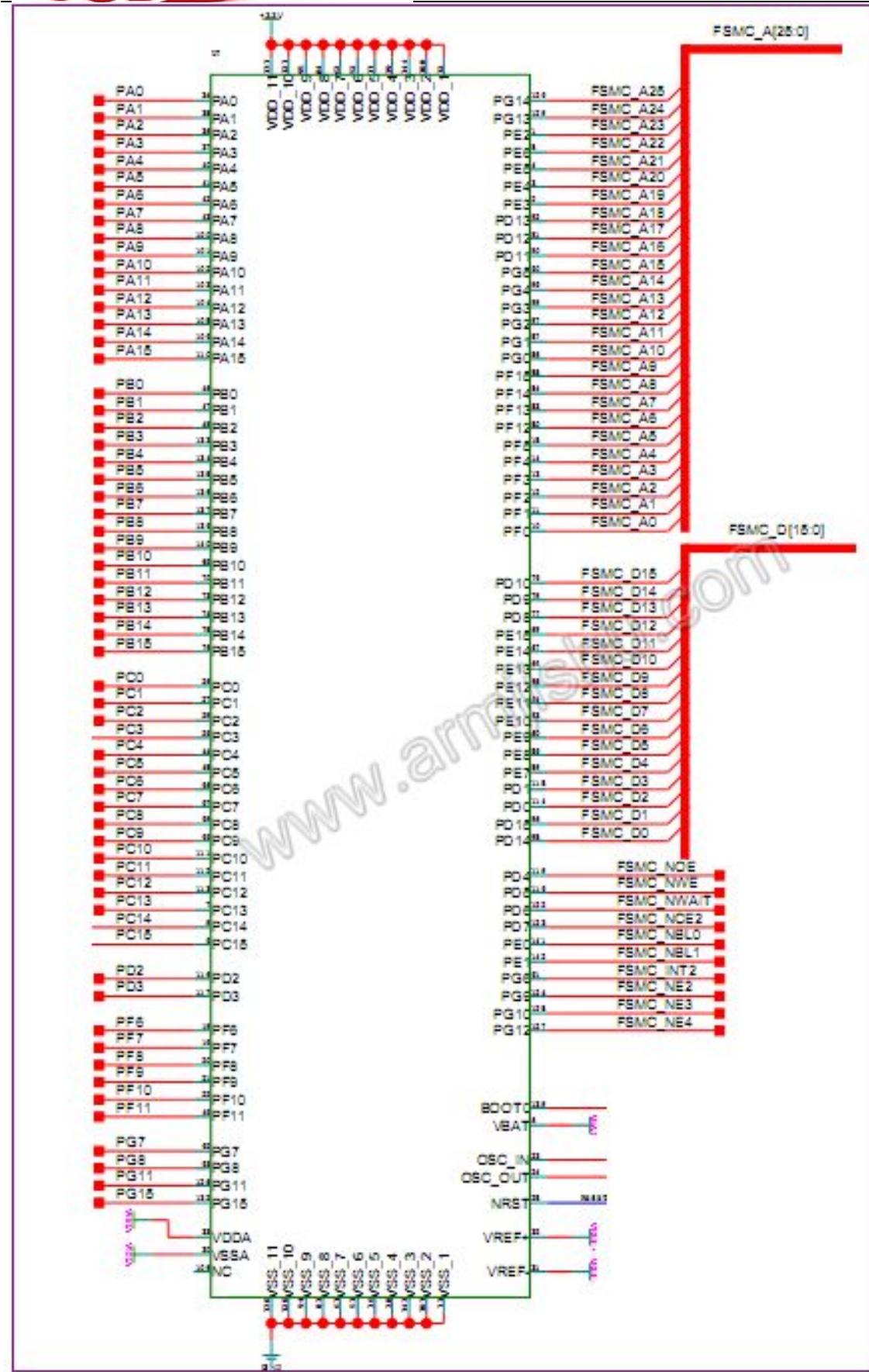
| 名称 | 信号类型 | 注解 |
|------------|-----------|--|
| V_{REF+} | 输入，模拟参考正极 | ADC使用的高端/正极参考电压， $2.4V \leq V_{REF+} \leq V_{DDA}$ |
| V_{DDA} | 输入，模拟电源 | 等效于 V_{DD} 的模拟电源且： $2.4V \leq V_{DDA} \leq V_{DD}(3.6V)$ |
| V_{REF-} | 输入，模拟参考负极 | ADC使用的低端/负极参考电压， $V_{REF-} = V_{SSA}$ |
| V_{SSA} | 输入，模拟电源地 | 等效于 V_{SS} 的模拟电源地 |

在神舟III号板上，VREF和VDDA电源相关电路如下图所示：



◆ STM32F103ZET6处理器

STM32F103ZET6处理器的GPIO口和其他的功能接口分别与神舟III号板的各功能模块连接，处理器部分电路如下。



STM32F103ZET6的资源与管脚分配如下表所示。

| 管脚号 | 信号名 | 功能接口 | | |
|-----|----------|-------------|-----------|------|
| 10 | FSMC_A0 | RS 信号 | Nor Flash | |
| 11 | FSMC_A1 | | | |
| 12 | FSMC_A2 | | | |
| 13 | FSMC_A3 | | | |
| 14 | FSMC_A4 | | | |
| 15 | FSMC_A5 | | | |
| 50 | FSMC_A6 | | | |
| 53 | FSMC_A7 | | | |
| 54 | FSMC_A8 | | | |
| 55 | FSMC_A9 | | | |
| 56 | FSMC_A10 | | | |
| 57 | FSMC_A11 | | | |
| 87 | FSMC_A12 | | | |
| 88 | FSMC_A13 | | | |
| 89 | FSMC_A14 | | | |
| 90 | FSMC_A15 | | | |
| 80 | FSMC_A16 | | | |
| 81 | FSMC_A17 | | | |
| 82 | FSMC_A18 | | | |
| 2 | FSMC_A19 | | | |
| 3 | FSMC_A20 | | | |
| 4 | FSMC_A21 | | | |
| 5 | FSMC_A22 | | | |
| 1 | FSMC_A23 | | | |
| 128 | FSMC_A24 | | | |
| 129 | FSMC_A25 | | | |
| 85 | FSMC_D0 | 2.8/3.2 LCD | Nor Flash | SRAM |
| 86 | FSMC_D1 | | | |
| 114 | FSMC_D2 | | | |
| 115 | FSMC_D3 | | | |
| 58 | FSMC_D4 | | | |
| 59 | FSMC_D5 | | | |
| 60 | FSMC_D6 | | | |
| 63 | FSMC_D7 | | | |
| 64 | FSMC_D8 | | | |
| 65 | FSMC_D9 | | | |
| 66 | FSMC_D10 | | | |
| 67 | FSMC_D11 | | | |
| 68 | FSMC_D12 | | | |
| 77 | FSMC_D13 | | | |
| 78 | FSMC_D14 | | | |
| 79 | FSMC_D15 | | | |

| | | | | | | |
|-----|------------|------------|--|------|--|------------|
| 92 | FSMC_INT3 | | | | | |
| 127 | FSMC_NE4 | | | | | |
| 73 | SPI2_NSS | | | | | |
| 74 | SPI2_SCK | | | | | |
| 75 | SPI2_MISO | | | | | |
| 76 | SPI2_MOSI | | | | | |
| 35 | PA1 | LCD 背光 | | | | |
| 93 | PG8 | BUSY 信号 | | | | |
| 126 | PG11 | Flash CS | | | | |
| 132 | PG15 | SD 卡 CS | | | | |
| 69 | PB10 | | | | | |
| 70 | PB11 | | | | | |
| 44 | PC4 | SPI 片选 | | | | 2.4G 模块片选 |
| 45 | PC5 | USB 上拉 | | | | 2.4G SPICS |
| 96 | PC6 | SD 卡 | | | | 2.4G 模块中断 |
| 118 | FSMC_NOE | | | | | |
| 119 | FSMC_NWE | | | | | |
| 122 | FSMC_NWAIT | | | | | |
| 123 | FSMC_NCE2 | | | | | |
| 91 | FSMC_INT2 | | | | | |
| 124 | FSMC_NE2 | | | | | |
| 125 | FSMC_NE3 | | | | | |
| 141 | FSMC_NBL0 | | | | | |
| 142 | FSMC_NBL1 | | | | | |
| 34 | WKUP | WAKE UP 按键 | | | | |
| 7 | TAMPER-RTC | TAMPER 按键 | | | | |
| 100 | PA8 | USER1 按键 | | | | |
| 117 | PD3 | USER2 按键 | | | | |
| 22 | PF10 | | | | | |
| 101 | USART1_TX | | | | | |
| 102 | USART1_RX | | | | | |
| 36 | USART2_TX | | | | | |
| 37 | USART2_RX | | | | | |
| 49 | PF11 | | | | | 485 接口 |
| 40 | SPI1_NSS | | | | | 485 方向 |
| 41 | SPI1_SCK | | | | | DAC_OUT1 |
| 42 | SPI1_MISO | | | | | DAC_OUT2 |
| 43 | SPI1_MOSI | | | | | |
| 28 | PC2 | | | 中断输入 | | |
| 103 | USB_DM | | | | | |
| 104 | USB_DP | | | | | |
| 135 | I2S_SD | | | | | |
| 105 | TMS-SWDIO | JTAG 接口 | | | | |

| | | | | | |
|-----|-------------|-----------|---------|-----|--|
| 109 | TCK-SWCLK | | | | |
| 110 | TDI | | I2S3_WS | | |
| 133 | TDO | | I2S_CK | | |
| 134 | NTRST | | | | |
| 46 | TIM3_CH3 | PWM 接口 | | | |
| 47 | TIM3_CH4 | | | | |
| 26 | ADC123_IN10 | | ADC 输入 | | |
| 27 | ADC123_IN11 | | | | |
| 29 | ADC123_IN13 | 电位器 | | | |
| 48 | BOOT1 | 启动模式 | | | |
| 138 | BOOT0 | | | | |
| 136 | I2C_SCL | I2C 24C02 | | 收音机 | |
| 137 | I2C_SDA | | | | |
| 139 | CAN_RX | CAN 总线 | | | |
| 140 | CAN_TX | | | | |
| 97 | I2S3_MCK | SD 卡 | | | |
| 98 | SDIO_D0 | | | | |
| 99 | SDIO_D1 | | | | |
| 111 | SDIO_D2 | | | | |
| 112 | SDIO_D3 | | | | |
| 113 | SDIO_CK | | | | |
| 116 | SDIO_CMD | | | | |
| 18 | PF6 | LED 灯 | | | |
| 19 | PF7 | | | | |
| 20 | PF8 | | | | |
| 21 | PF9 | | | | |

说明：功能接口中同一填充颜色表示相同的接口，如上表中的淡绿色表示2.8/3.2寸LCD接口。

◆ 启动模式设置

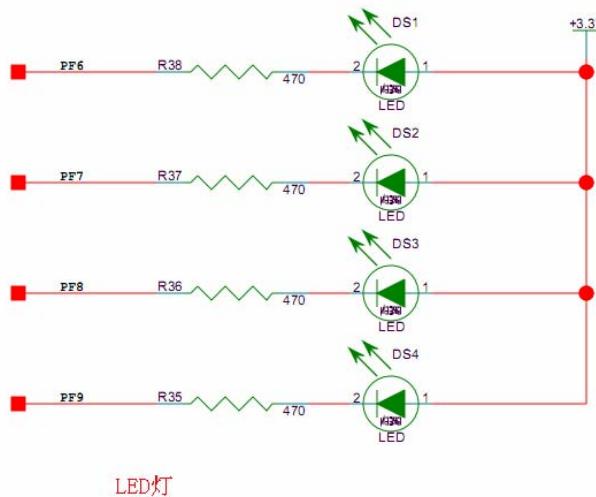
上图中右下角的BOOT0, BOOT1用于设置STM32的启动方式，其对应启动模式如下表所示：

| BOOT1 (J9) | BOOT0 (J10) | 功能 | 说明 |
|------------|-------------|---------------|--------------------|
| ANY | 2-3 | User Boot(默认) | 用主闪存存储器，即Flash启动 |
| 2-3 | 1-2 | System Boot | 系统存储器启动，用于串口下载 |
| 1-2 | 1-2 | SRAM Boot | SRAM启动，用于SRAM中调试代码 |

- ◆ 从主闪存存储器启动：主闪存存储器被映射到启动空间（0x0000 0000），但仍然能够在它原来的地址（0x0800 0000）访问它，即闪存存储器的内容可以在两个地址区域访问，0x0000 0000或0x0800 0000。
- ◆ 从系统存储器启动：系统存储器被映射到启动空间(0x0000 0000)，但仍然能够在它原有的地址(0x1FFF F000)访问它。
- ◆ 从内置SRAM启动：只能在0x2000 0000开始的地址区访问SRAM。

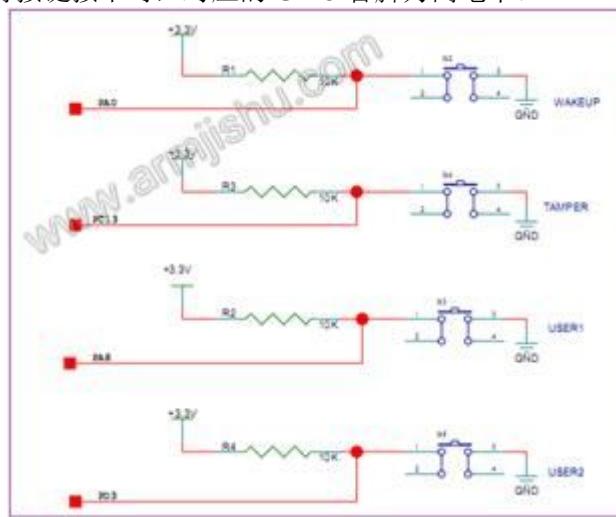
1.2.2. LED指示灯

在神舟 III 号中，除了电源指示灯外，还有 4 个 LED 指示灯，由 GPIO 管脚控制 LED 灯的亮灭，当 GPIO 管脚输出低电平时，LED 指示灯亮。反之，当 GPIO 管脚输出高电平时，LED 指示灯灭。这四个 LED 指示灯分别由 PF6~9 控制。



1.2.3. 按键

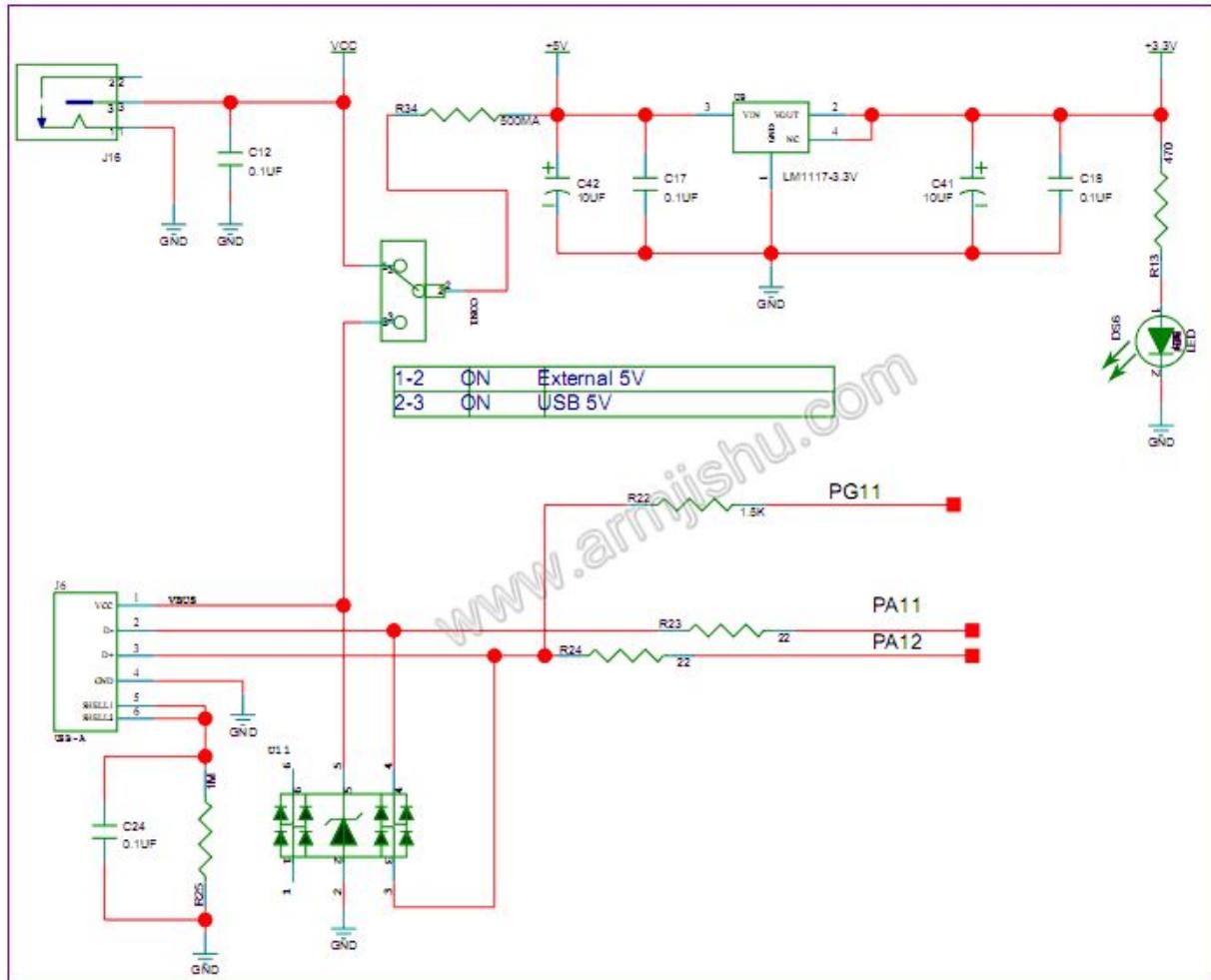
神舟 III 号 STM32 开发板板载 4 个功能按键，分别是 WAKEUP 按键和 TAMPER 按键及两个用于自定义功能按键，在不使用第二功能的情况下，这四个按键都可以作为通用的按键，由用户自定义其功能。按键分别与 PA0, PC13, PA8 和 PD3 四个 GPIO 管脚连接，当按键按下时，对应的 GPIO 管脚为低电平，反之，当没有按键按下时，对应的 GPIO 管脚为高电平。



1.2.4. USB接口与电源

神舟 III 号 STM32 开发板支持的供电方式主要有三种，分别是：

- USB 接口供电，最大 500mA
- 外部直流 5V 供电
- JLINK V8 供电，包括 5V 和 3.3V 两种方式



如上图所示，为神舟III号的USB接口与电源部分电路。

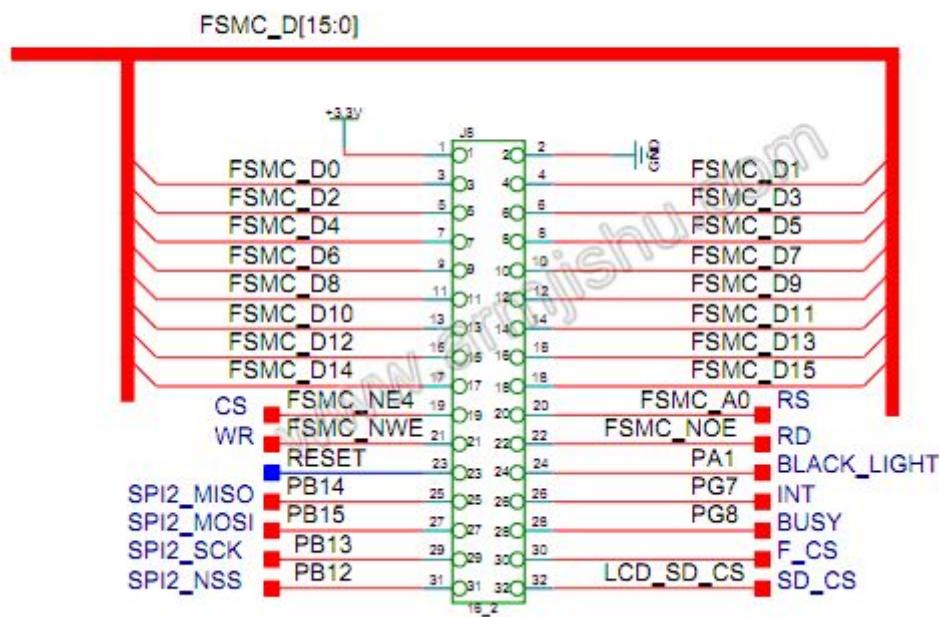
当CON1电源开关拨到1-2位置时，神舟III号由外部接口供电。板上的电源转换芯片将USB接口输入的5V电源转换成3.3V的电源，作为处理器和相关外围电路的工作电源。。

当CON1电源开关拨到2-3位置时，神舟III号由USB接口供电。板上的电源转换芯片将USB接口输入的5V电源转换成3.3V的电源，作为处理器和相关外围电路的工作电源。在上图中，U11为USB接口的ESD防护电路，满足ESD防护标准IEC61000-4-2(ESD 15kV air, 8kV Contact)。

1.2.5. 液晶显示模块

神舟III号开发板载有目前比较通用2.8/3.2寸液晶显示模块接口。其原理图如下：

2.8/3.2 LCD



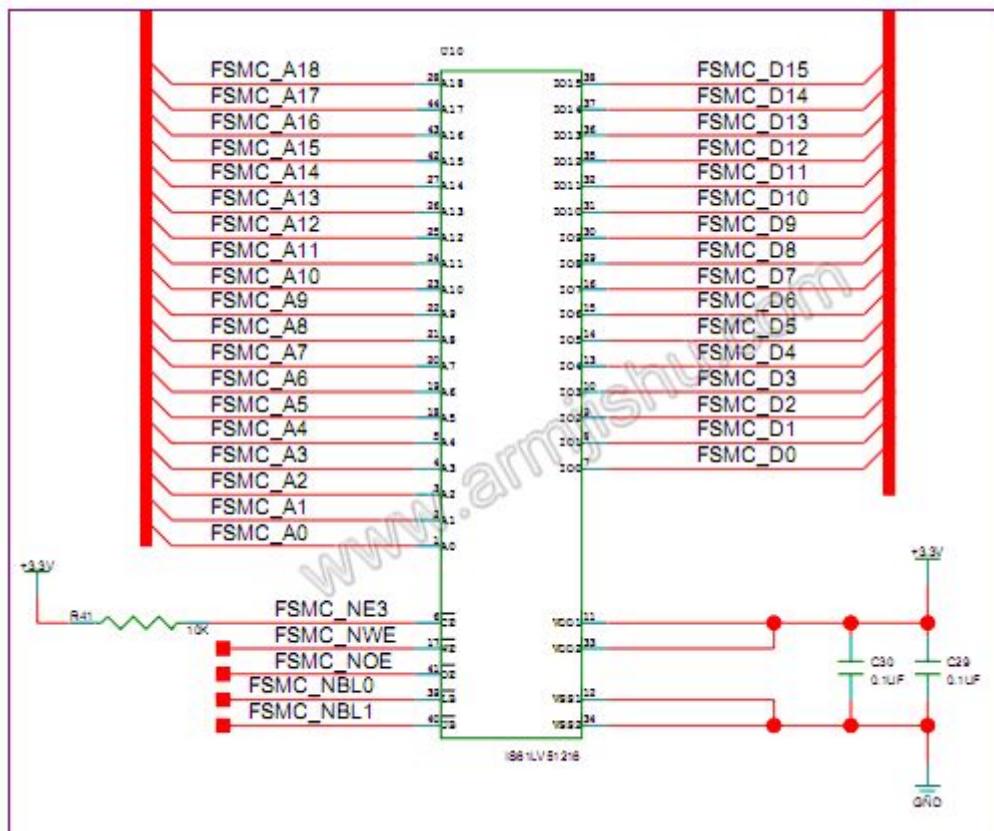
图表 4 液晶显示模块原理图

神舟III号通过FSMC总线对屏进行访问操作实现图形界面的显示。神舟系列的2.8/3.2寸LCD触摸屏支持触摸功能，LCD模块上有触摸芯片，将电阻式触摸屏的模拟信号转化为数字信号，处理器通过SPI接口读取芯片转换后的数字，支持查询方式和中断方式。

此外，2.8/3.2寸LCD屏模块上还集成了SPI Flash和SPI接口的SD卡座（神舟III号板载了一个SDIO接口的SD卡座），在这里由于资源有限，神舟III号只支持2.8/3.2寸屏和SD卡，不支持2.8/3.2寸LCD屏模块上的SPI Flash。

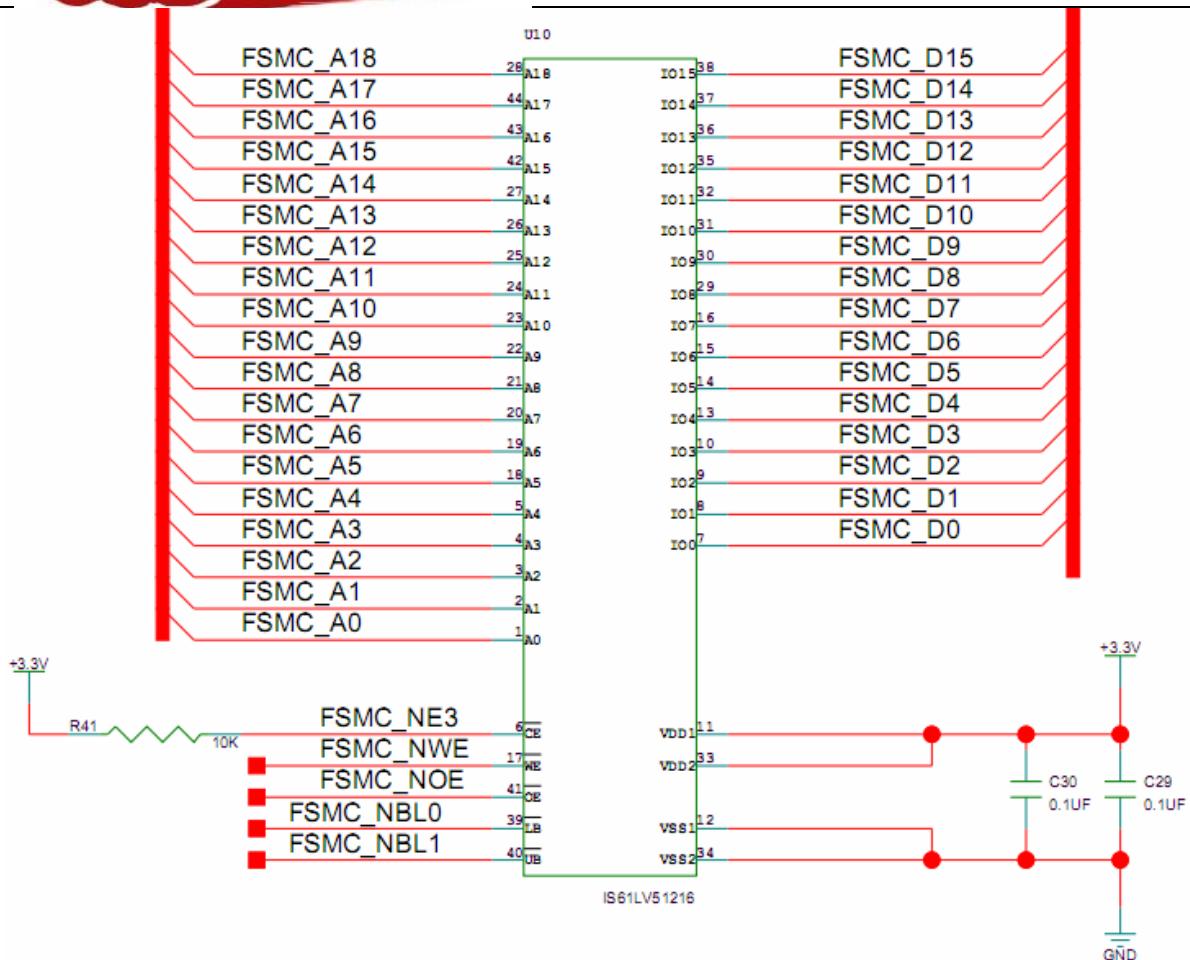
1.2.6. Nor Flash

神舟III号板载了16M比特的Nor Flash，最大支持128M比特容量Nor Flash。如下图所示，为128M Nor Flash的连接原理图，STM32F103ZET通过FSMC访问Nor Flash



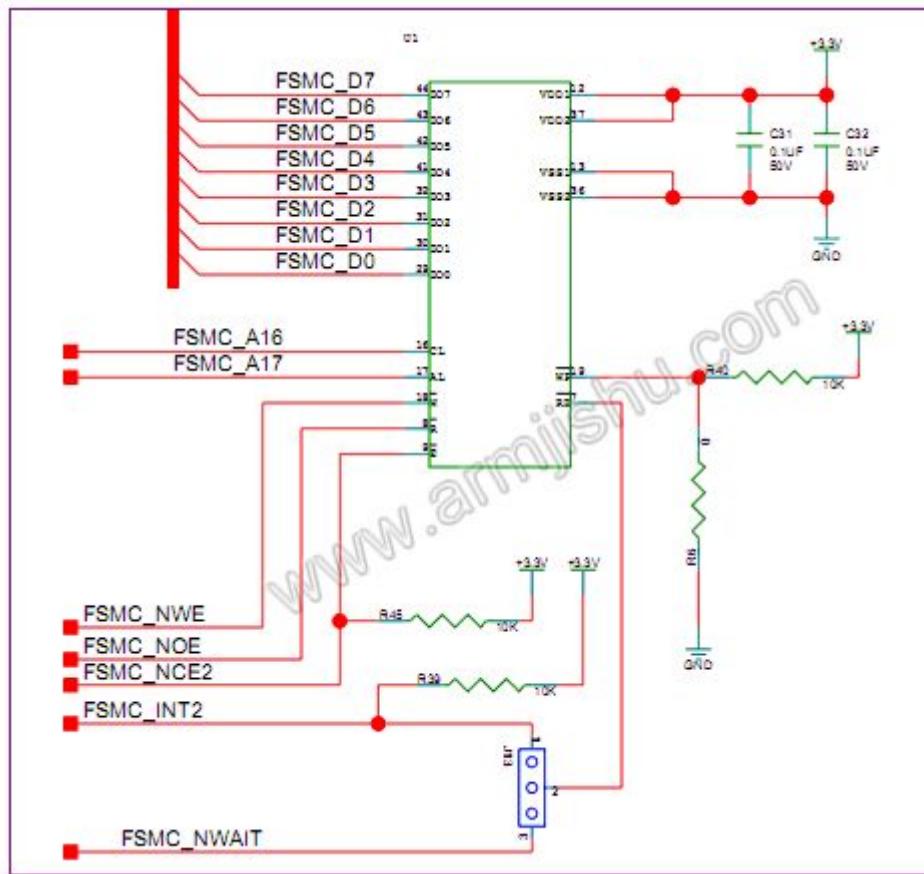
1.2.7. SRAM

神舟III号板载了4M比特的SRAM,如下图所示,为SRAM部分理图, STM32F103ZET通过FSMC访问SRAM。



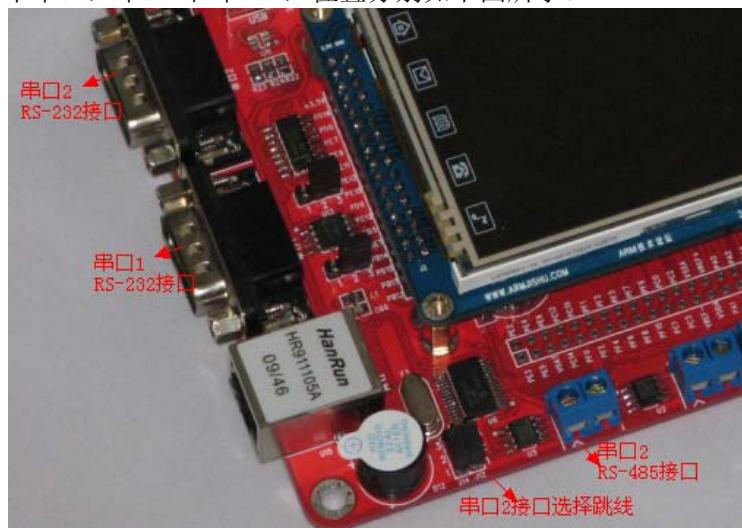
1.2.8. Nand Flash

神舟III号板载了1G比特的Nand Flash,如下图所示,为Nand Flash部分的原理图, STM32F103ZET通过FSMC访问Nand Flash。



1.2.9. 串口与RS485接口

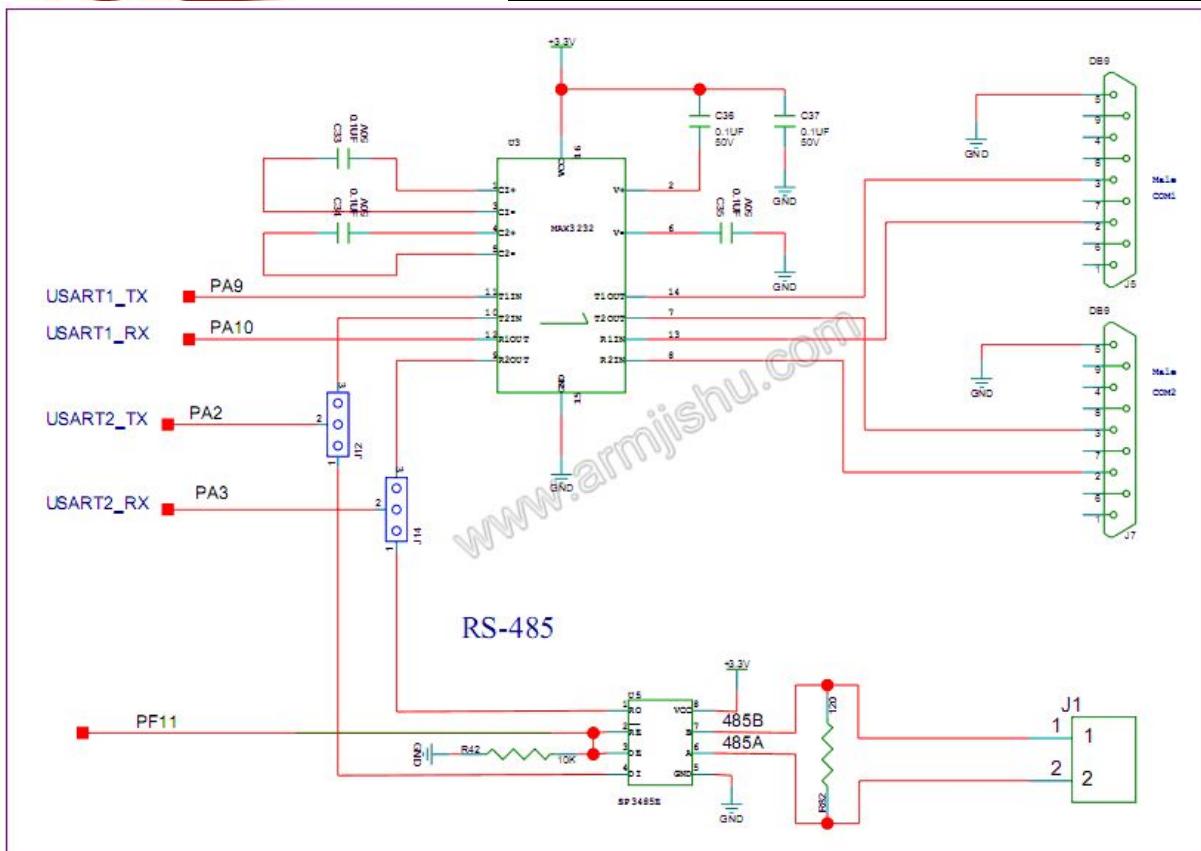
神舟III号板载了2个串口，串口1和串口2，位置分别如下图所示。



其中串口2可通过跳线选择支持RS-232接口或RS-485接口，跳线定义如下：

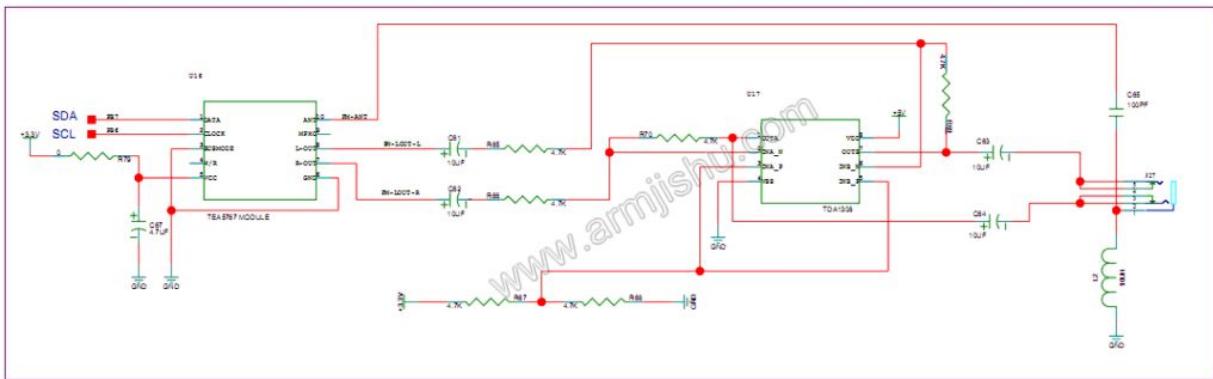
| | | |
|-----|-----|--------------|
| J14 | J12 | 串口2功能选择 |
| 1-2 | 1-2 | 串口2 RS-485接口 |
| 2-3 | 2-3 | 串口2 RS-232接口 |

请按照上表设置跳线，选择串口2作为RS-232串口或RS-485接口。串口1和串口2的原理图如下图所示，串口1和串口2都可作为3线RS-232串口使用，其中串口2还可作为RS-485接口使用。注意，当串口2作为RS-485使用时，神舟III号默认是安装了RS-485接口的120欧终端匹配电阻。对应下图的R82，请依据实际应用选择是否安装此匹配电阻。



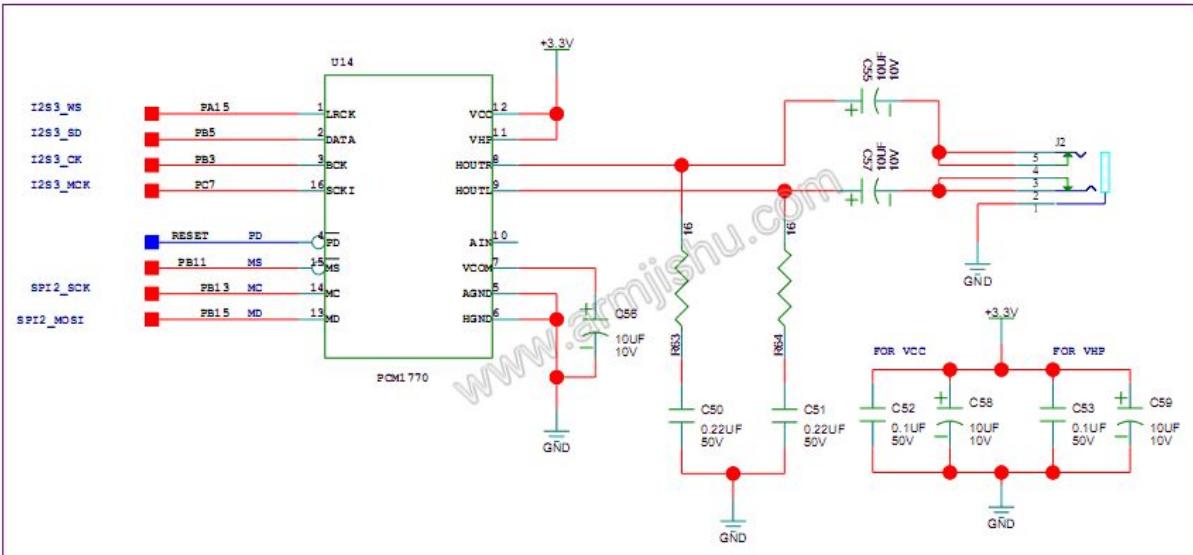
1.2.10. 收音机模块

神舟III号STM32开发板上使用目前主流的TEA5767收音机模块提供收音功能，STM32F103ZET6处理器通过I2C接口访问和配置TEA5767模块。



1.2.11. 音频解码电路

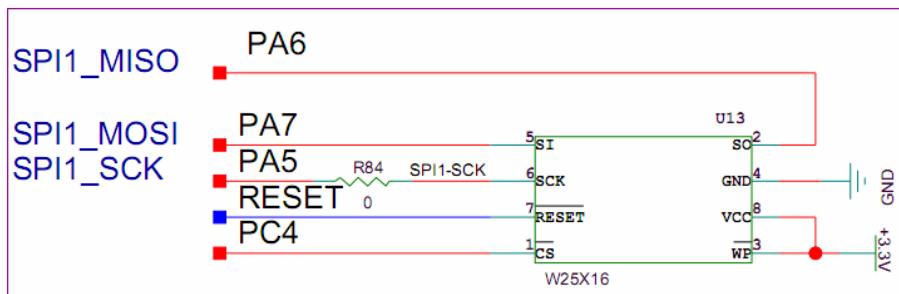
神舟III号STM32开发板上使用了PCM1770这款具有耳机放大器的24位低功耗立体声音频DAC芯片提供音频播放功能。处理器通过I2S3传送音频信号到PCM1770，由它进行解码输出到J2音频座。而PCM1770的配置接口与处理器的SPI2连接，处理器通过SPI接口访问PCM1770。



1.2.12. SPI Flash

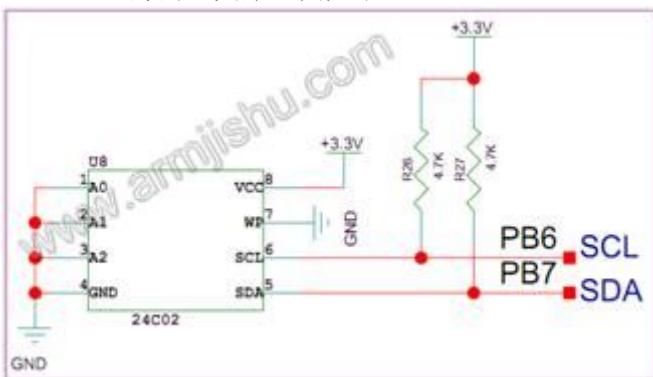
神舟III号开发板载有SPI Flash芯片W25X16，该芯片的容量为2M字节，与AT45DB161属于同一级别，其原理图如下：

W25X16也是与处理器的SPI1接口相连，注意由于神舟III号上有W25X16和网口共用了SPI1，通过不同的CS进行区分，请勿同时使这两个接口的CS有效，否则，可能引起芯片读写访问失败。



1.2.13. EEPROM

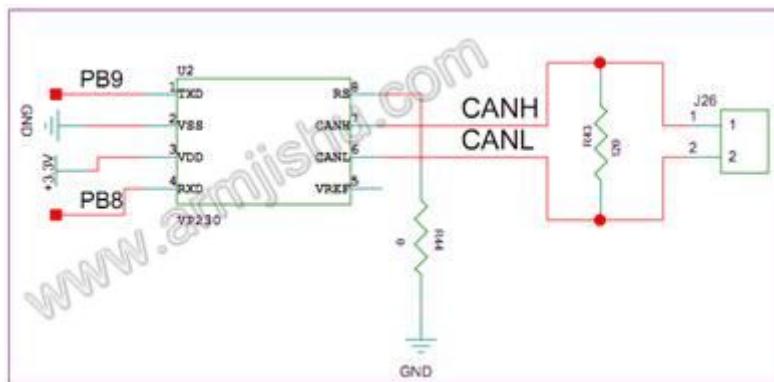
神舟III号自带了24C02的EEPROM芯片，该芯片的容量为2K比特，也就是256个字节，对于我们普通应用来说是足够了的。你也可以选择换大的芯片，因为在原理上是兼容24C02~24C256全系列的EEPROM芯片的。其原理图如下：



这里我们把A0~A2均接地，对24C02来说也就是把地址位设置成0了，在进行程序设计时，请确认EEPROM的地址与之对应。

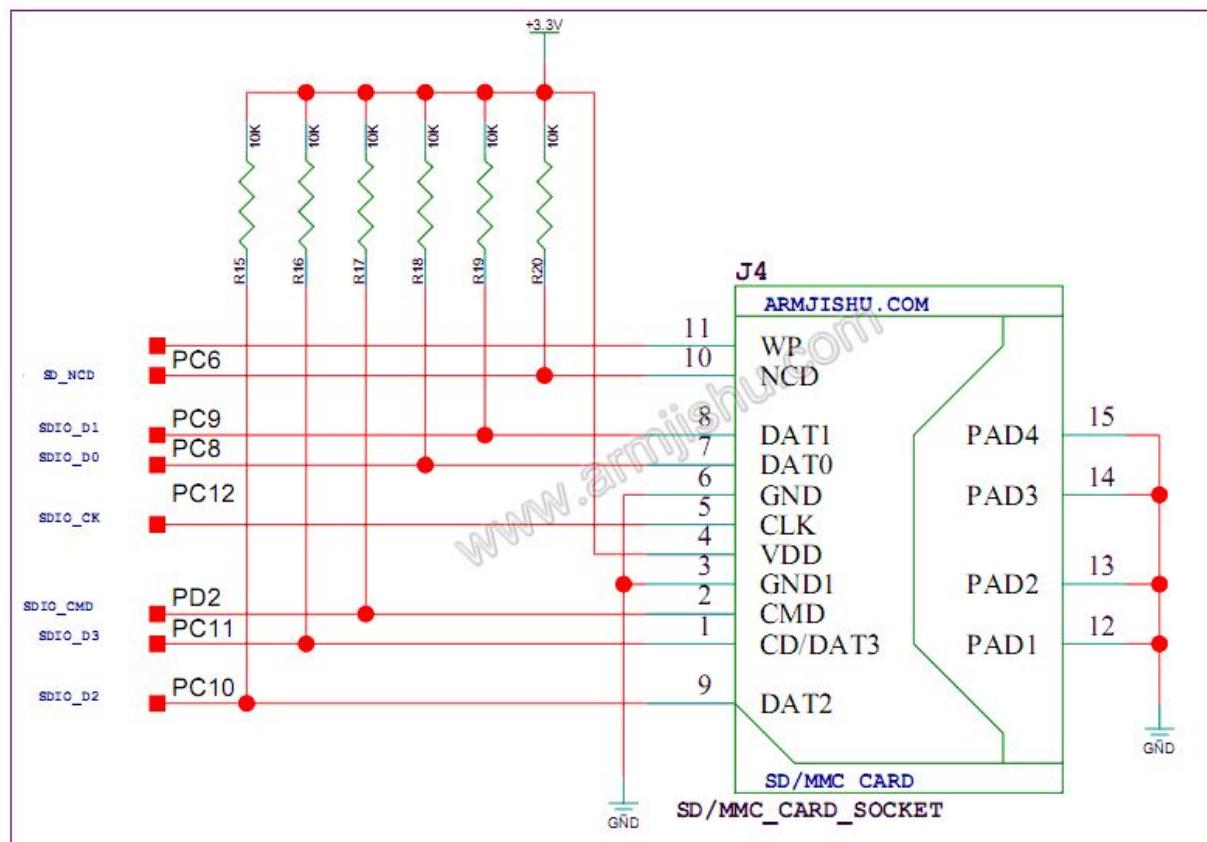
1.2.14. CAN总线接口

STM32F103ZET6 这一款处理器集成了 CAN 总体接口，在开发板上我们使用了 TI 公司的 3.3V 电压的 CAN 总线收发器来实现 CAN 总线，如下图所示。CAN 总线收发器型号为 VP230。



1.2.15. SD卡

神舟III号开发板载标准的SD卡接口，有了这个接口，我们就可以外扩容量存储设备，可以用 来记录数据。其原理图如下：



一般SD卡可通过SPI接口或者SDIO接口来通信，其中SDIO模式通信速率更高，在这里我们即是使用的SDIO模式进行通信。

1.2.16. 无线模块

神舟III号开发板板载了2款无线模块的接口，分别是315M无线模块和NRF24L01+这一款2.4G无线模块。

其中315M无线模块，可以接受遥控器的信号，当遥控的一个按键按下时，对应的无线模块的D0~3管脚变为有效。需要指出的是，无线模块当输出为高电平有效。而神舟III号通过三极管将315M无线模块与板上的按键进行了资源复用。

当无线模块的VT脚有效（低电平）时，表示无线模块接收到遥控的按键信号；当VT管脚无效（高电平）时，表示无线模块没有接收到遥控的按键信号，与无线模块连接的几个管脚的电平变化是按键引起的。

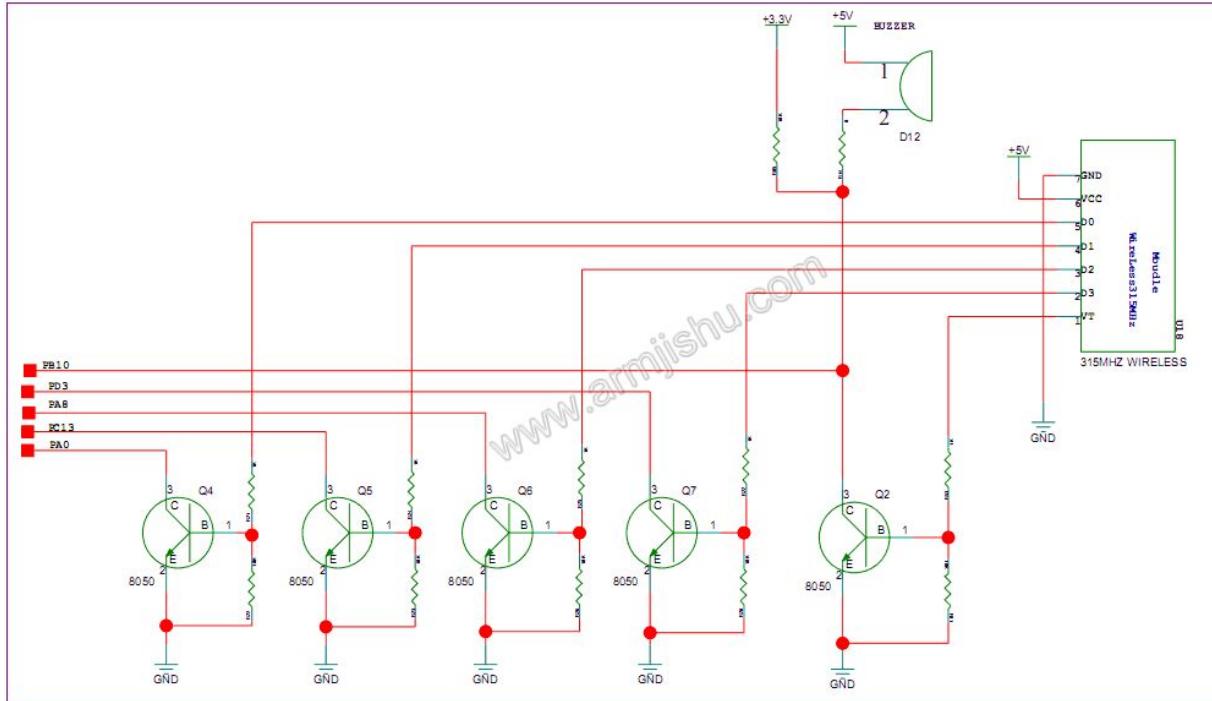
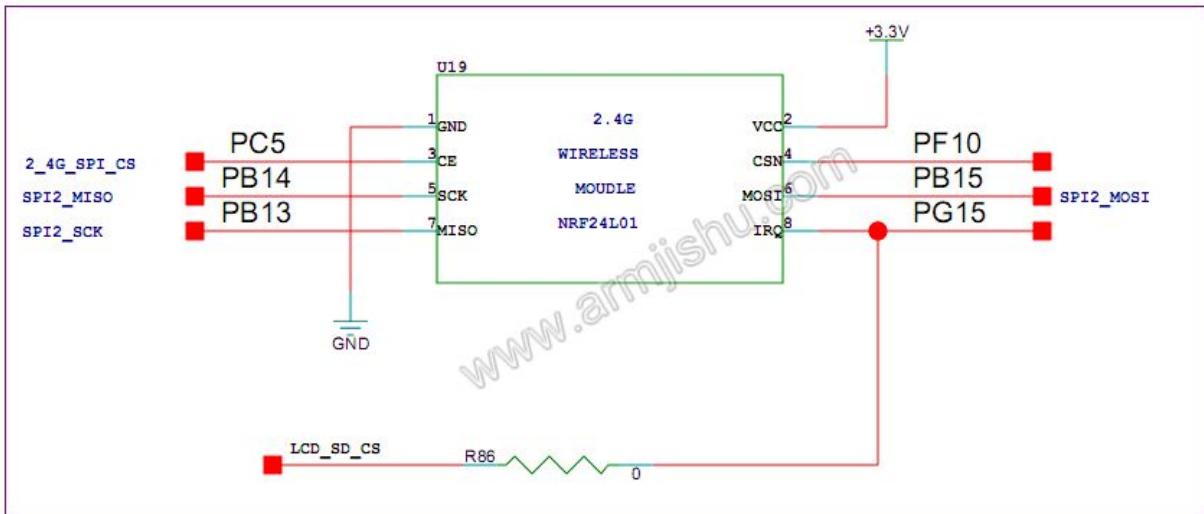


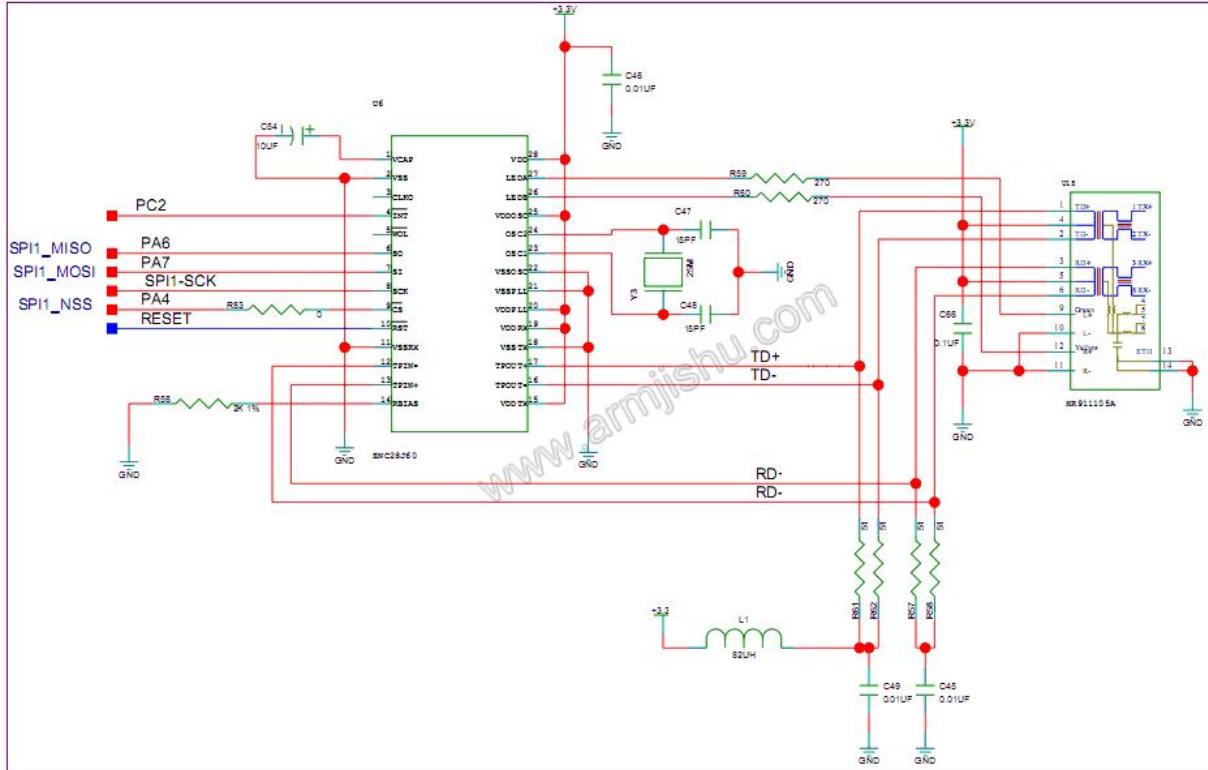
图1.2.10.1 无线模块接口原理图

下图为NRF24L01+模块的连接示意图，NRF24L01+无线模块通过SPI2与STM32F103ZET6相连。注意，NRF24L01+的IRQ管脚与3.2寸屏模块上集成的SD卡CS信号进行了复用，当需要使用3.2寸屏模块上集成的SD卡时，不能使用NRF24L01+模块。



1.2.17. 以太网

神舟三号选用的是 ENC28J60 这一款 SPI 接口的以太网控制器实现 10M 以太网，如下图所示，STM32F103 通过 SPI1 访问 ENC28J60，包括内部寄存器的读写访问。HR911105A 是 HANRUN 公司推出的一款集成变压器的 RJ45 接口，两者配合实现了 10M 以太网。



1.3. STM32神舟III号开发板目前可以支持的外接模块

1.3.1 DS18B20温度传感器

1.3.2 DH11温度传感器

1.3.3 2.4G无线模块（WIFI也属于2.4G范畴之内）

1.3.4 315M无线模块收发

1.3.5 三轴加速度感应模块

1.3.6 MP3音乐播放模块

1.3.7 2.8寸液晶屏模块

1.3.8 3.2寸液晶屏模块

1.3.9 4.3寸液晶屏模块

1.3.10 7寸液晶屏模块

第二章 软件篇

本章以目前最新的RVMDK 4.72版本为例，详细介绍RVMDK（以下简称**MDK**）这一主流的开发环境的使用。包括MDK工程的建立，代码的编译和调试，以及程序下载等。通过这一章节，我们将了解MDK的基本操作，熟悉基于MDK软件的STM32开发流程。

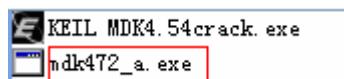
2.1 RVMDK简介

RVMDK源自德国的KEIL公司，是RealView MDK的简称，RealView MDK集成了业内最领先的技术，包括μ Vision4集成开发环境与 RealView编译器。支持ARM7、ARM9和最新的Cortex-M3核处理器，自动配置启动代码，集成Flash烧写模块，强大的Simulation设备模拟，性能分析等功能，与ARM之前的工具包ADS等相比，RealView编译器的最新版本可将性能改善超过20%。

2.2 如何安装RVMDK软件

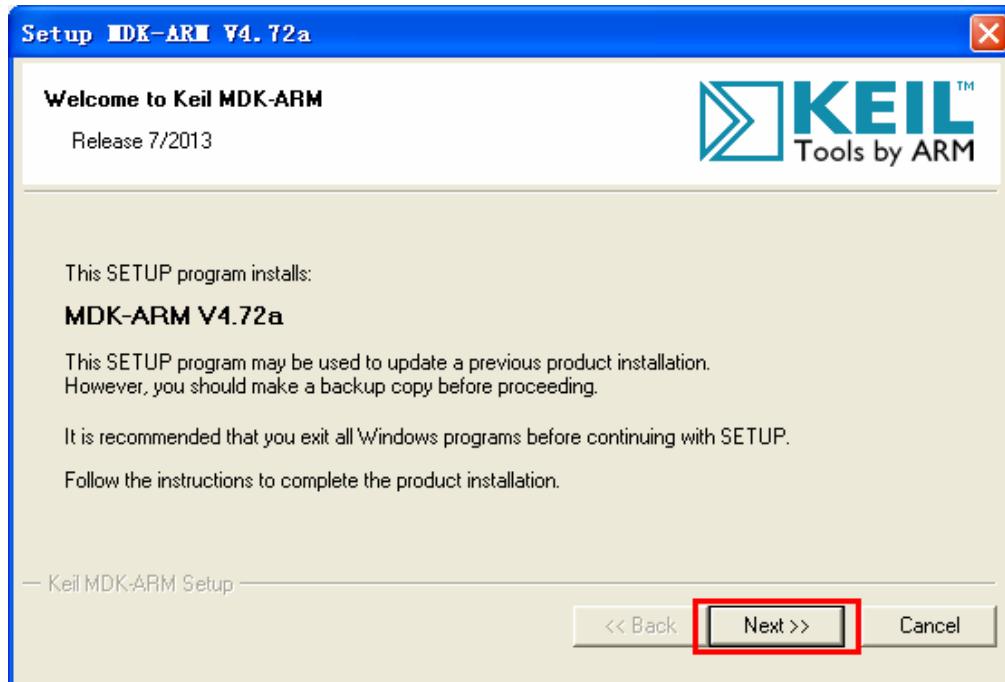
在我们编写代码之前需要先要把 MDK 这个软件安装好，MDK 有很多的版本，神舟 III 号开发板的主芯片是 STM32F103ZET6，该芯片是 M3 的核。M3 核的芯片使用 4.12 或 4.22 版本的 MDK 软件即可，MDK 软件的各个版本的安装方法是一样的。我们这里以 4.72 版本的 MDK 软件给大家说明安装过程。

在光盘资料目录下:\神舟 III 号光盘\神舟 III 号光盘资料(不包含源码和手册)\4.软件工具\MDK 编译器\MDK\keil4.72



双击 mdk472_a.exe，在弹出的 MDK 安装界面后，按如下步骤操作。如果之前安装了，其它低于 4.72 版本的 MDK 软件的话，先将原来的版本卸载，再安装 4.72 的 MDK 版本。

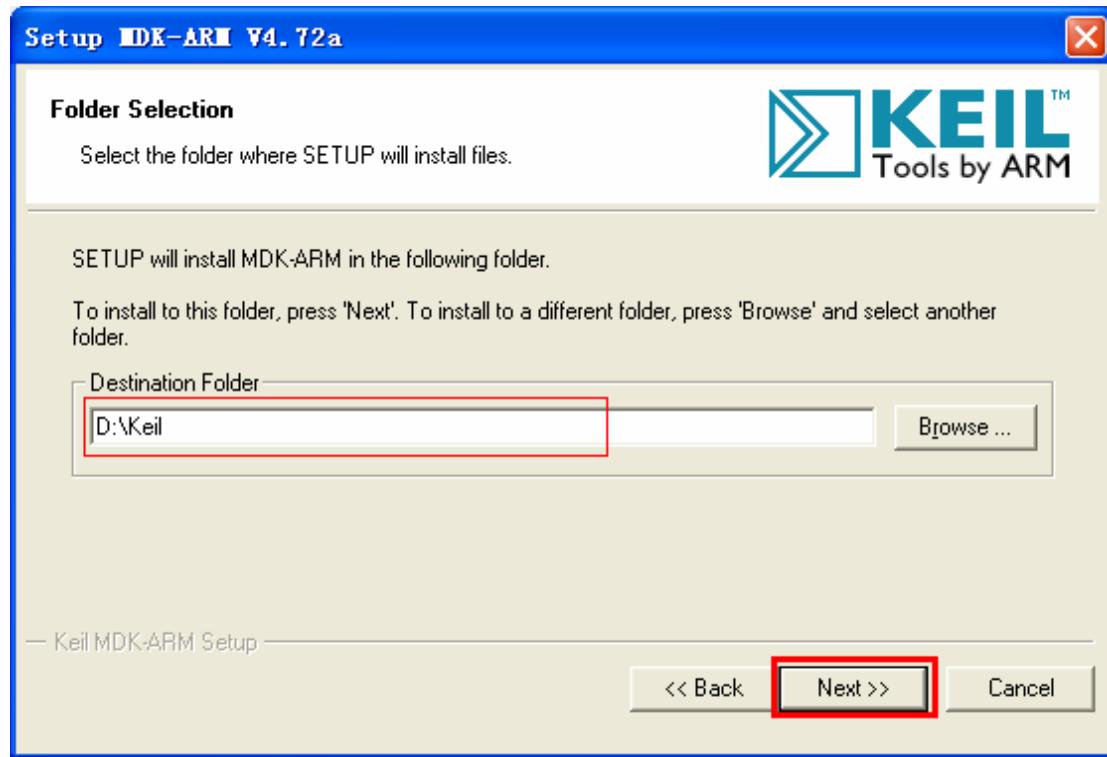
- 点击 Next



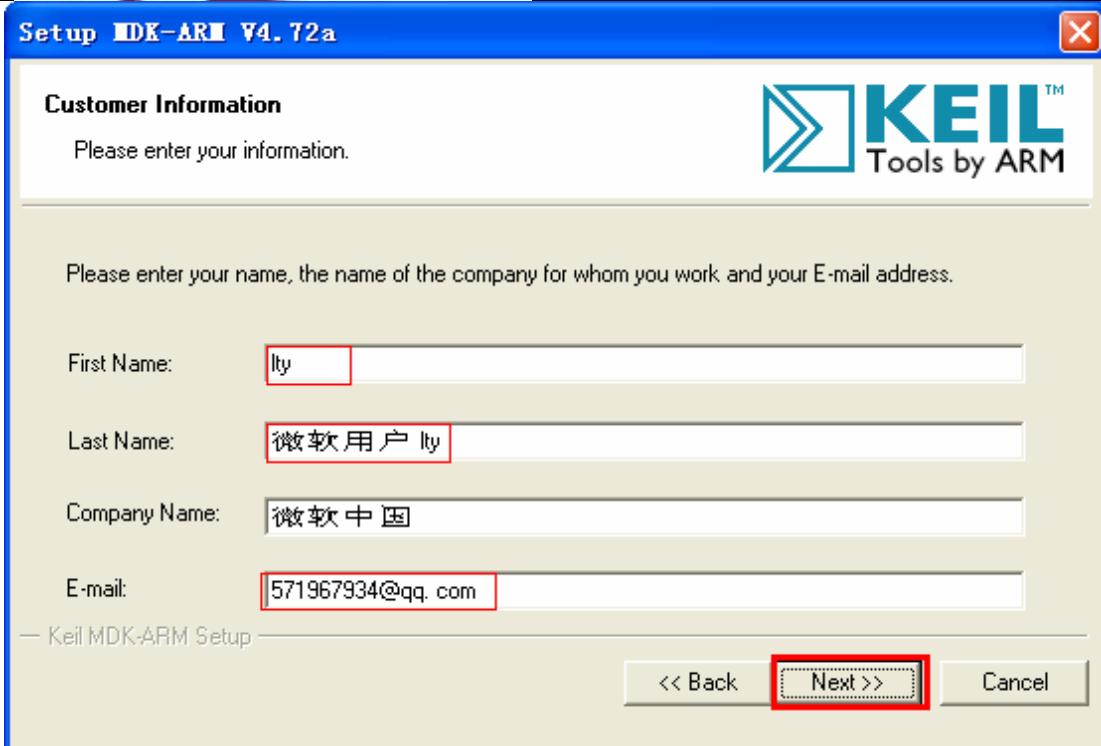
- 把勾勾起，点击 Next



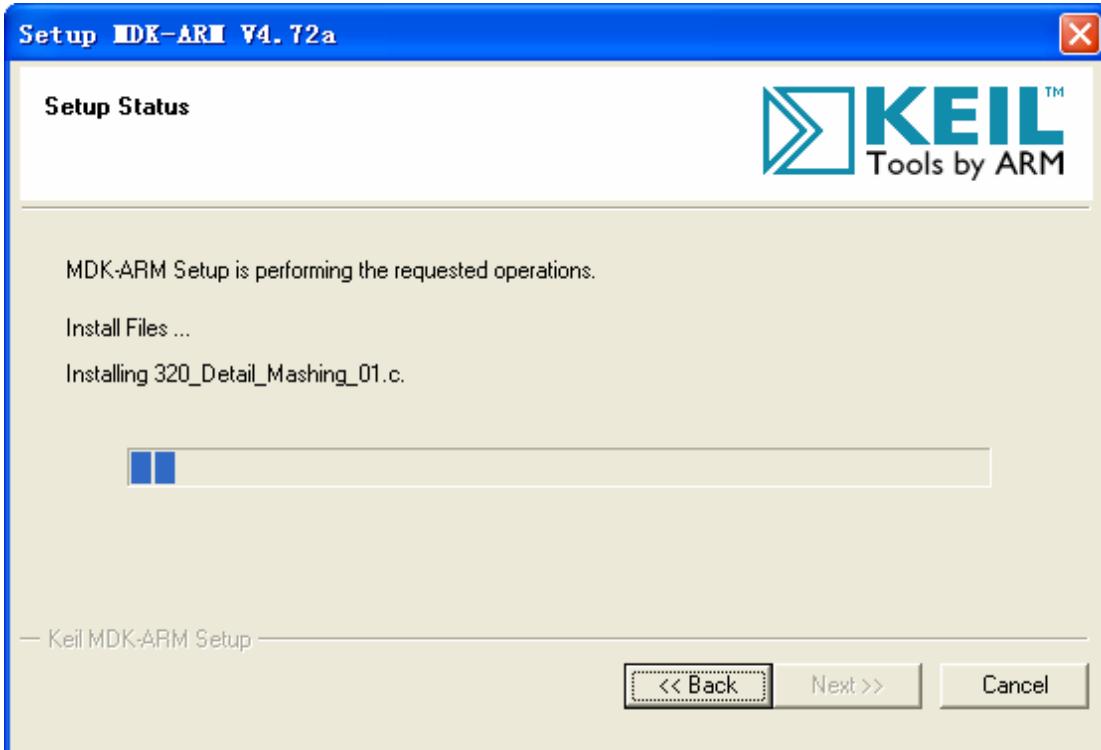
- 点击 Next，默认安装是在 C:\Keil 目录下，可以根据自己的需要安装在其它的路径。个人电脑的话，一般没有往 C 盘装软件的习惯，我们这里安装在 D:\Keil。



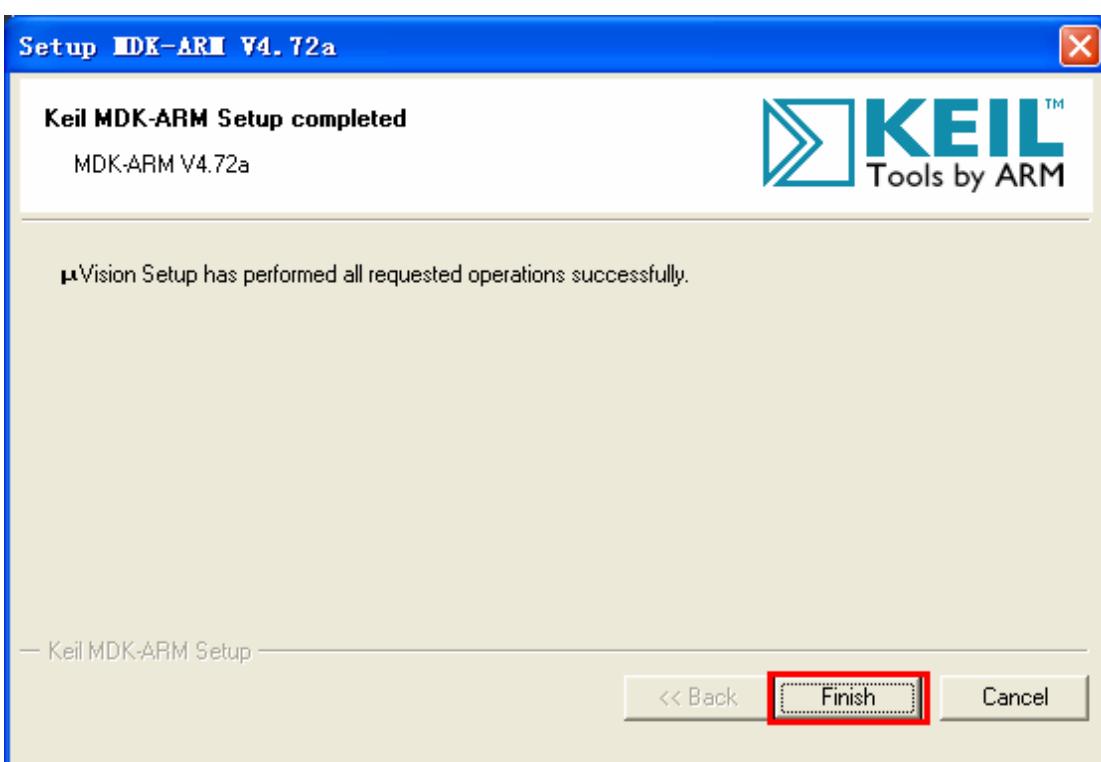
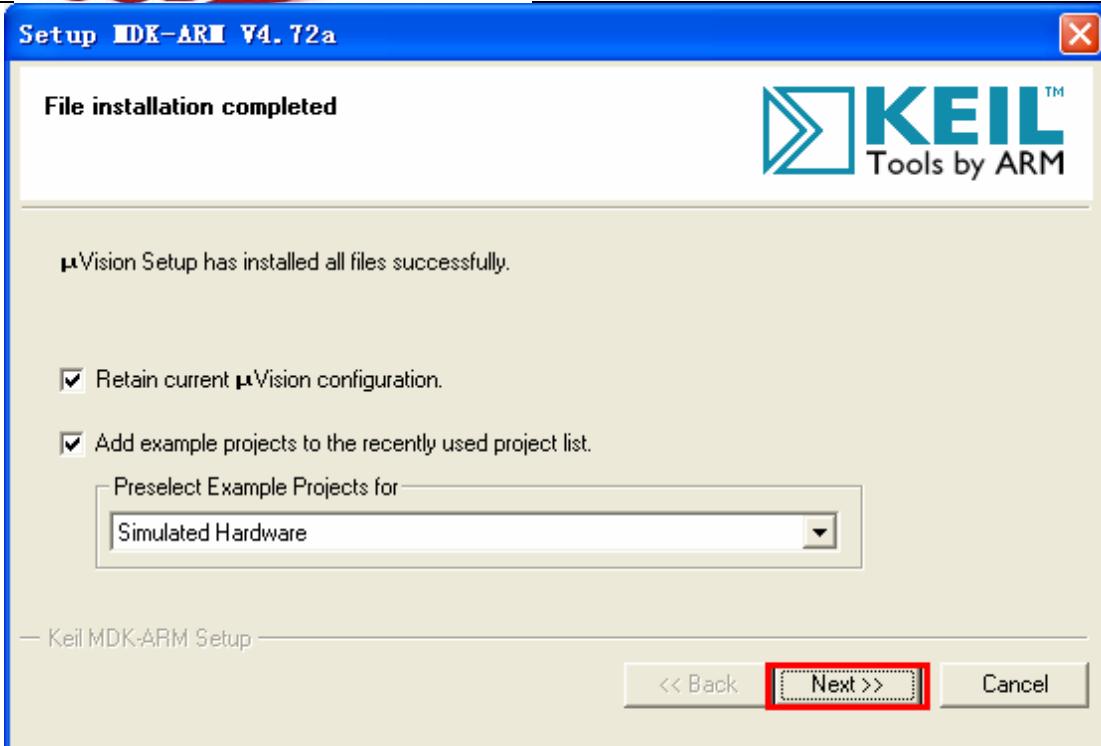
- 在用户名中填入名字（随便写），在邮箱地址中填写个人邮箱，点击 Next。



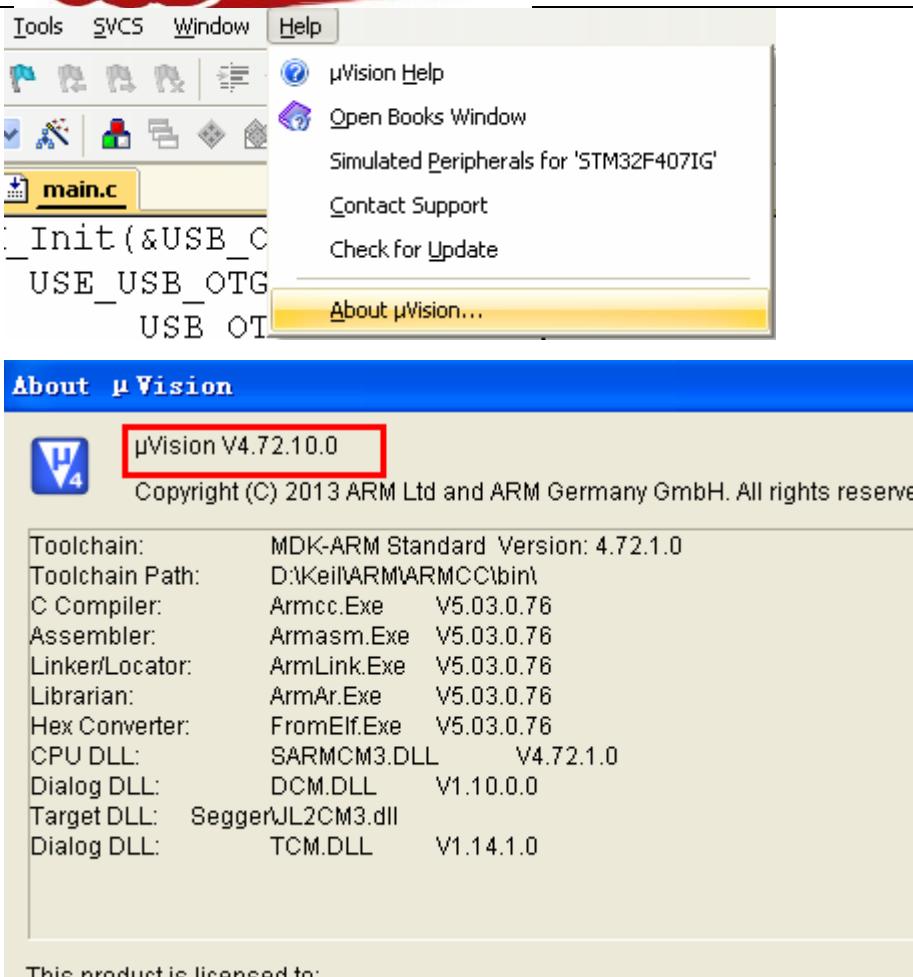
- 安装要需要点时间，耐心等待。



- 依次点击 Next，点击 Finish，完成安装。



- 安装完成，可以在桌面看到 MDK 的图标。安装完成之后可以在工具栏 help->about uVision 选项卡中查看到版本信息。



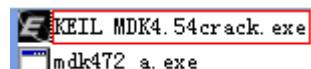
This product is licensed to:

4.72 版本的 MDK 软件，较低版本多了个语法检查的功能，非常人性化，当你写代码的时候，想

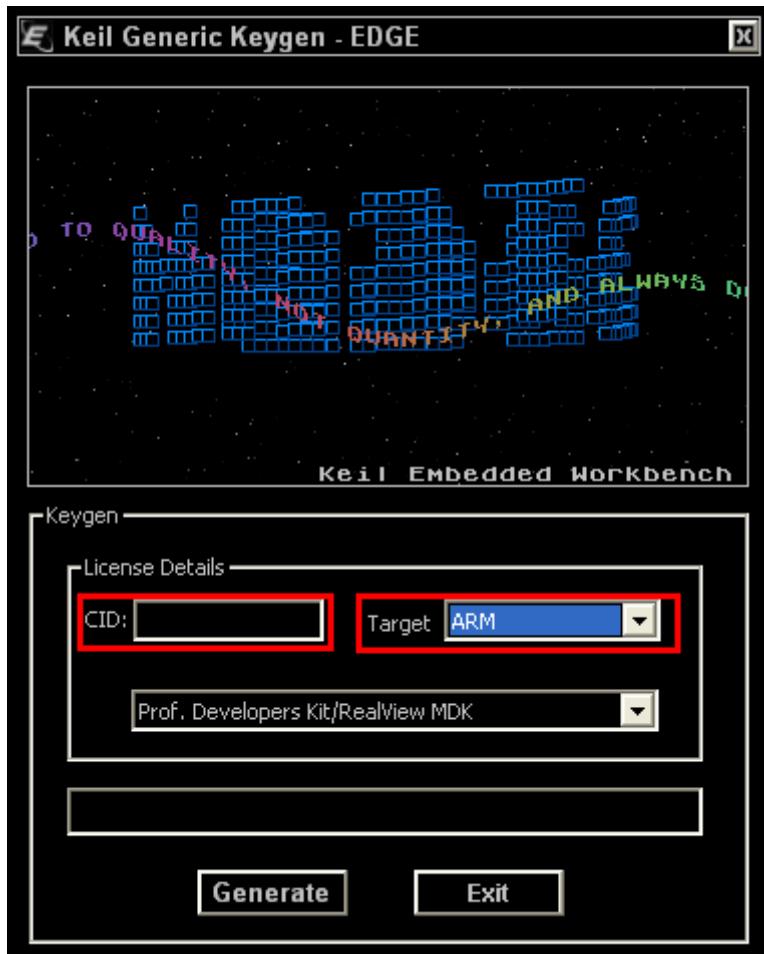
```
12 int main(void)
13 {
14     int v
15     GPIO_InitTypeDef GPIO_InitStructure;
16
17     /* 使能LED对应GPIO的Clock时钟 */
18     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO
19
20     /* 初始化LED的GPIO管脚，配置为推挽输出 */
21     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 |
22                                     //设置对端口的
23     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_O
```

2.3 注册MDK软件

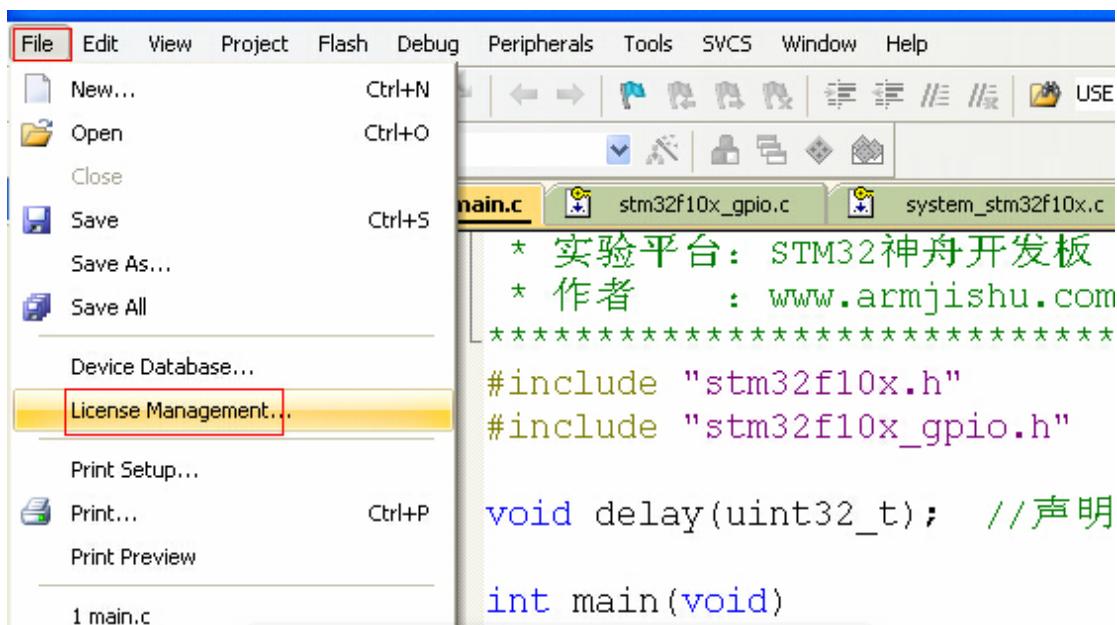
MDK 软件安装完成之后，我们需要注册一下。在光盘资料目录下：\神舟 III 号光盘\神舟 III 号光盘资料(不包含源码和手册)\4.软件工具\MDK 编译器\MDK\keil4.72 下

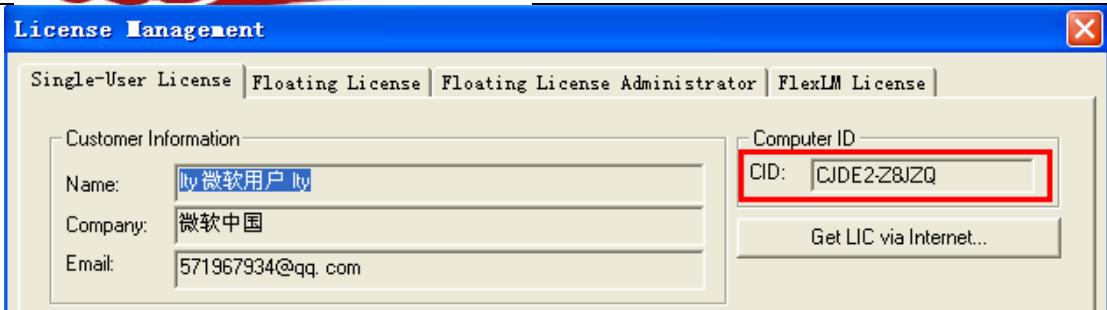


- 双击 KEIL MDK4.54crack.exe，在打开的界面中的 CID 选项框填入 MDK 的 CID。在 Target 选项框中选择 ARM。

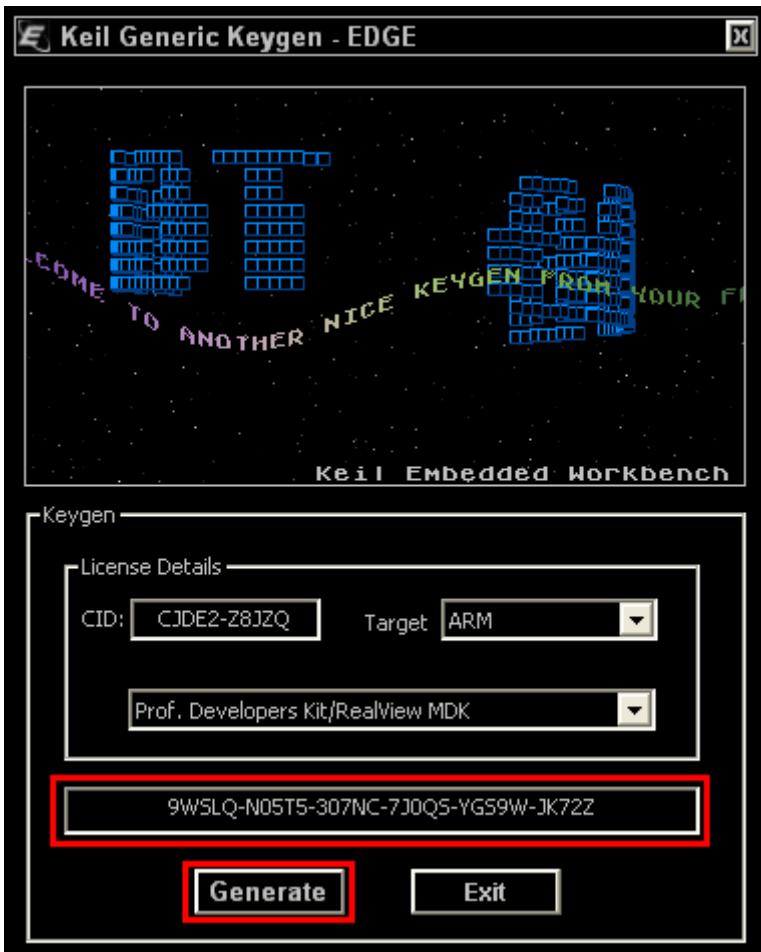


- MDK 的 CID 在软件的菜单栏 File\License Management 中获得。

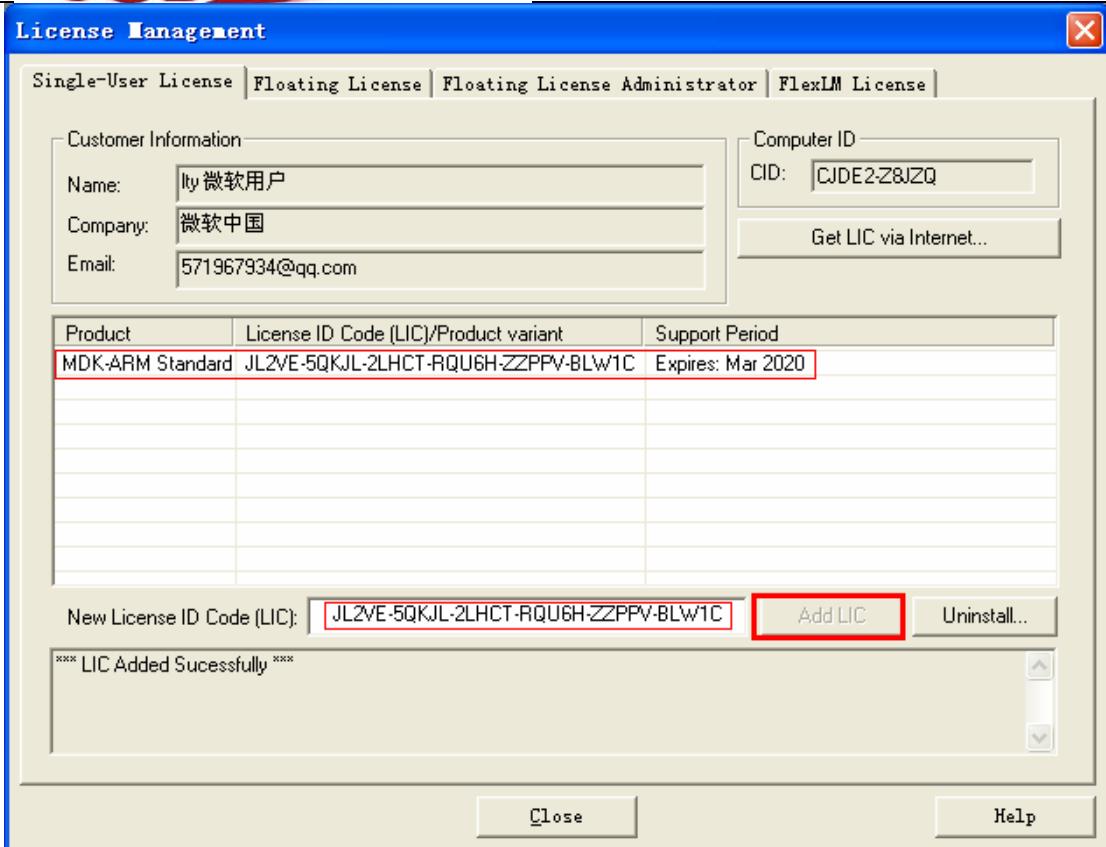




- 将复制的 CID 码，填入 CID 选项框。点击 Generate 按钮，生成 CID Code。

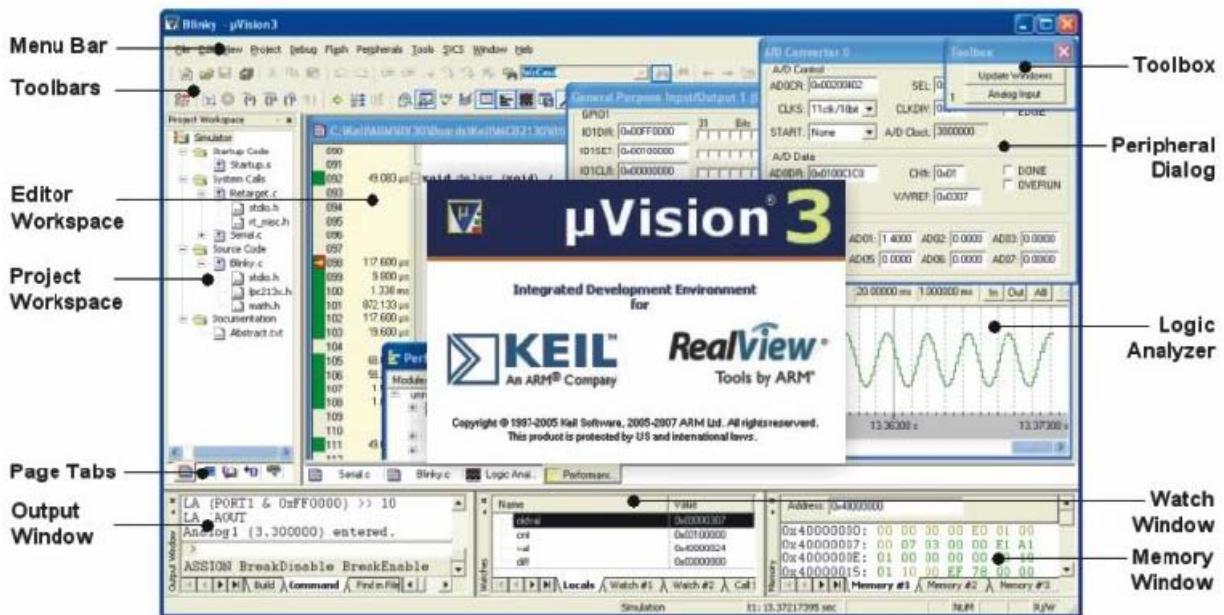


- 回到 MDK，菜单栏 File\License Management 中，把复制的 CID Code 粘贴到 New License ID Code (LIC): 方框中。点击 Add LIC，显示可以用到 2020 年，点击 Close，注册完成。关闭 MDK 软件，再打开，即可正常使用。



2.4 MDK 4.12集成开发环境的组成

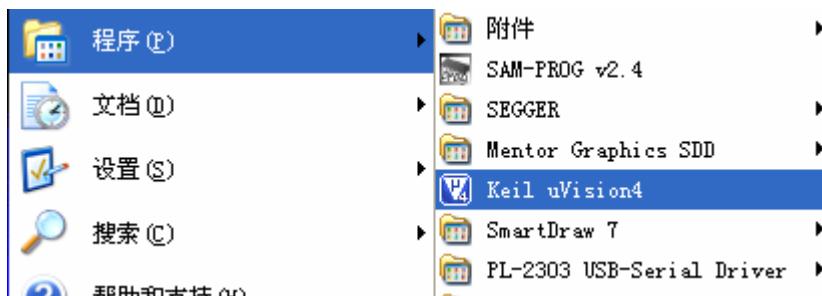
MDK又称叫RVMDK，源自德国的KEIL公司，是RealView MDK的简称，RealView MDK集成了业内最领先的技术，包括μ Vision4集成开发环境与 RealView编译器。支持ARM7、ARM9和最新的Cortex-M3核处理器，自动配置启动代码，集成Flash烧写模块，强大的Simulation设备模拟，性能分析等功能，与ARM之前的工具包ADS等相比，RealView编译器的最新版本可将性能改善超过20%。



2.5 MDK工程的编辑

2.5.1 新建RVMKD工程

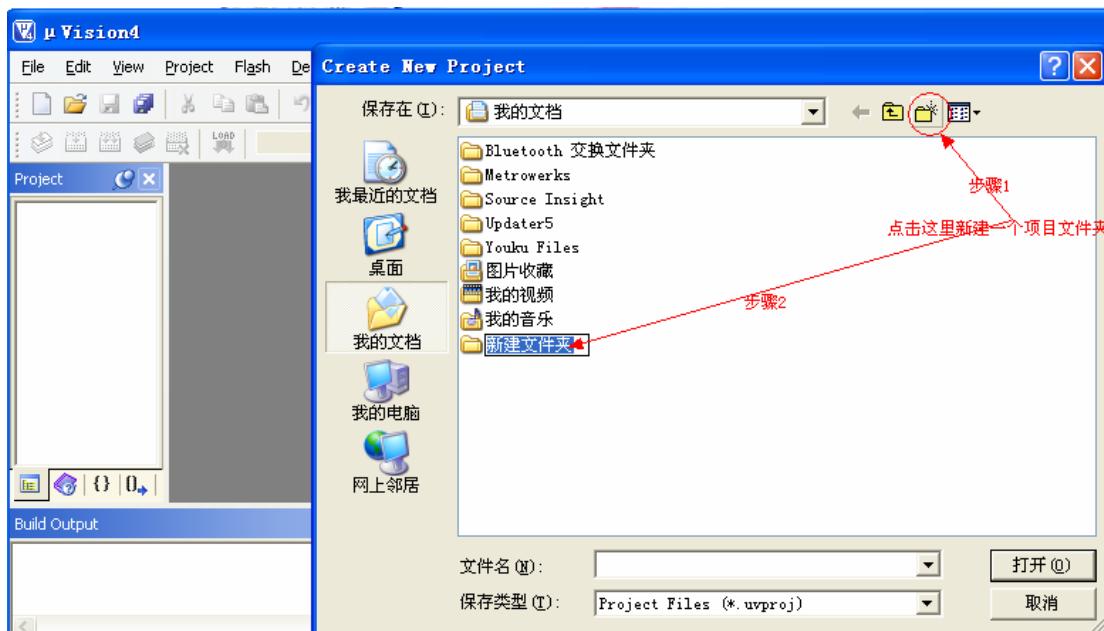
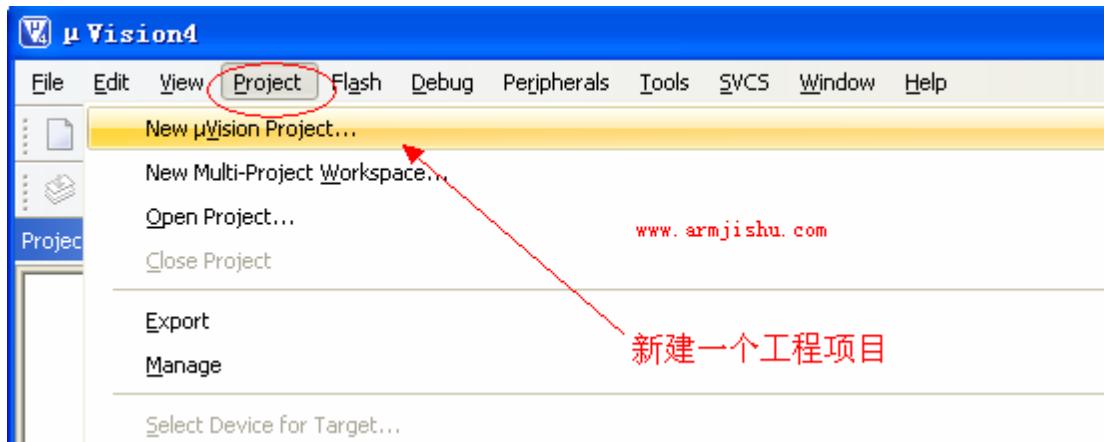
1) 点击 WINDOWS 操作系统的【开始】→【程序】→【Keil uVision4】启动 Keil uVision 或在桌面双击【Keil uVision4】快捷方式启动。启动 MDK4.12 如图所示：



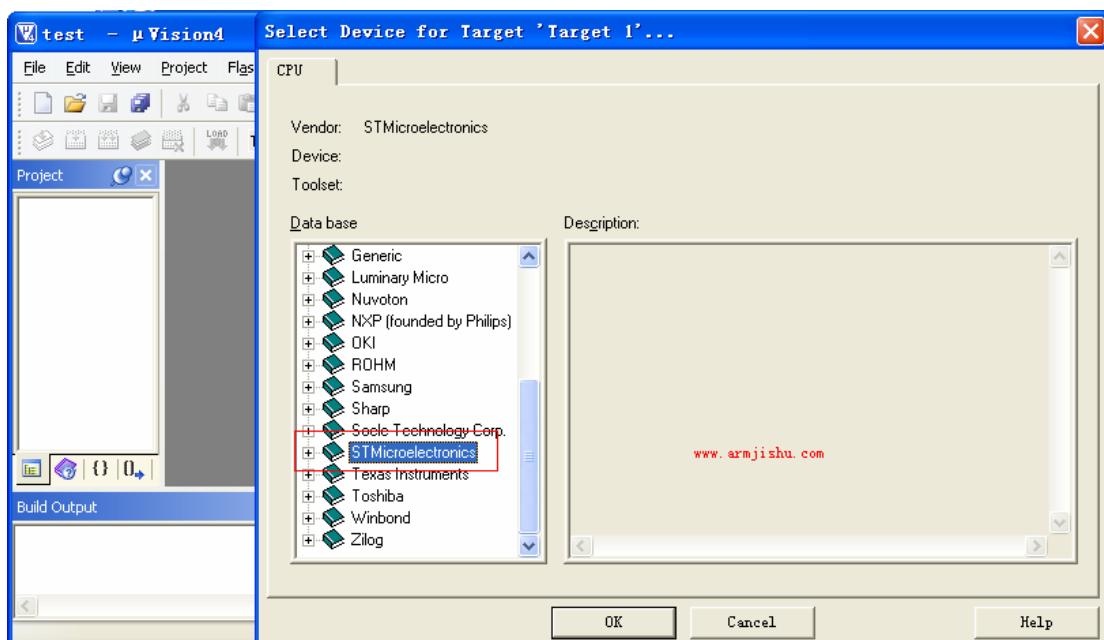
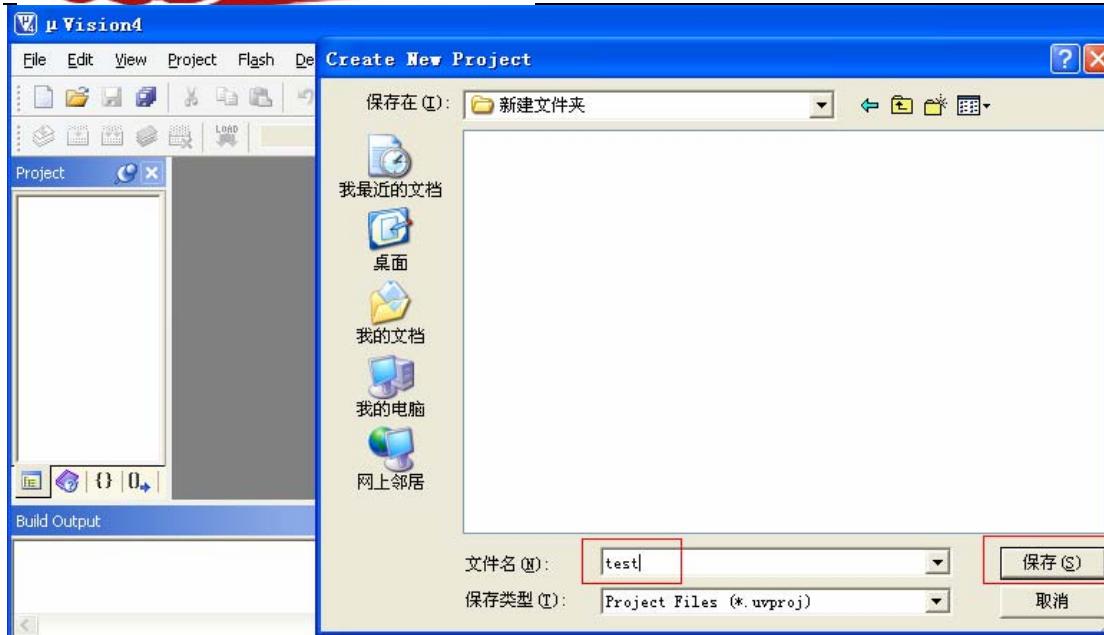
2) 点击之后，出现启动画面：



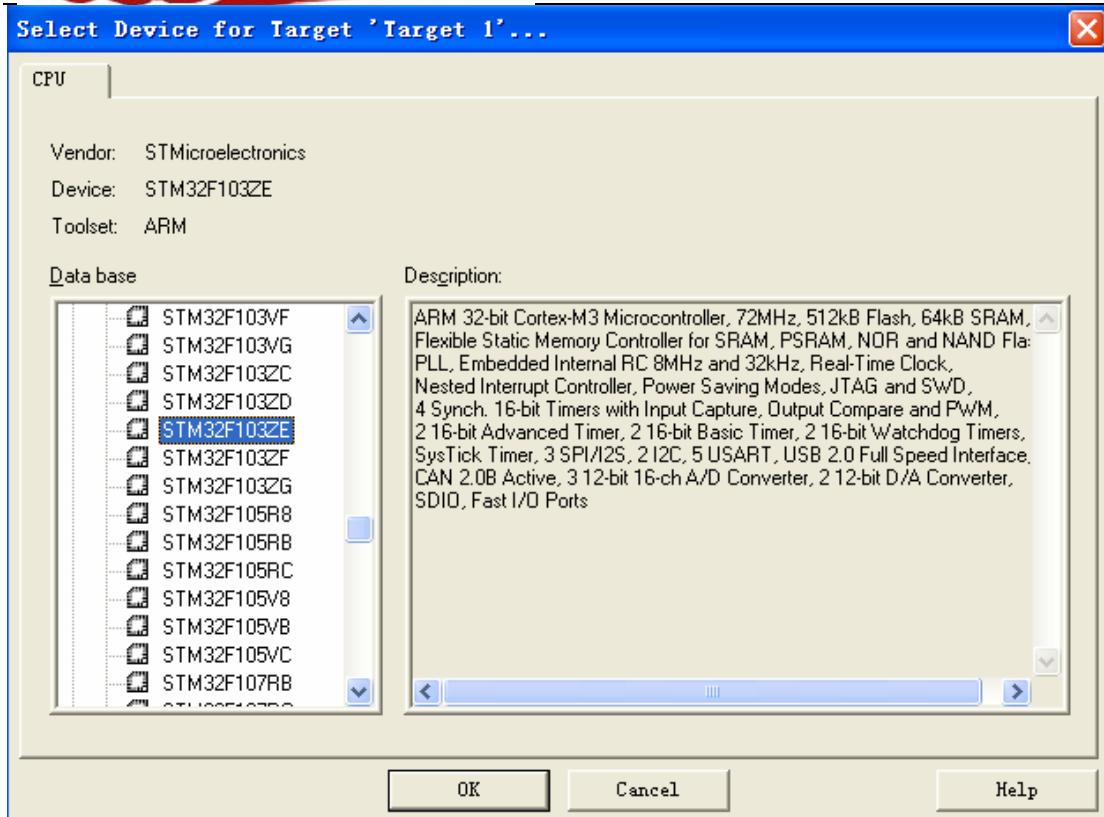
1) 点击“project---New uVision Project”新建一个工程



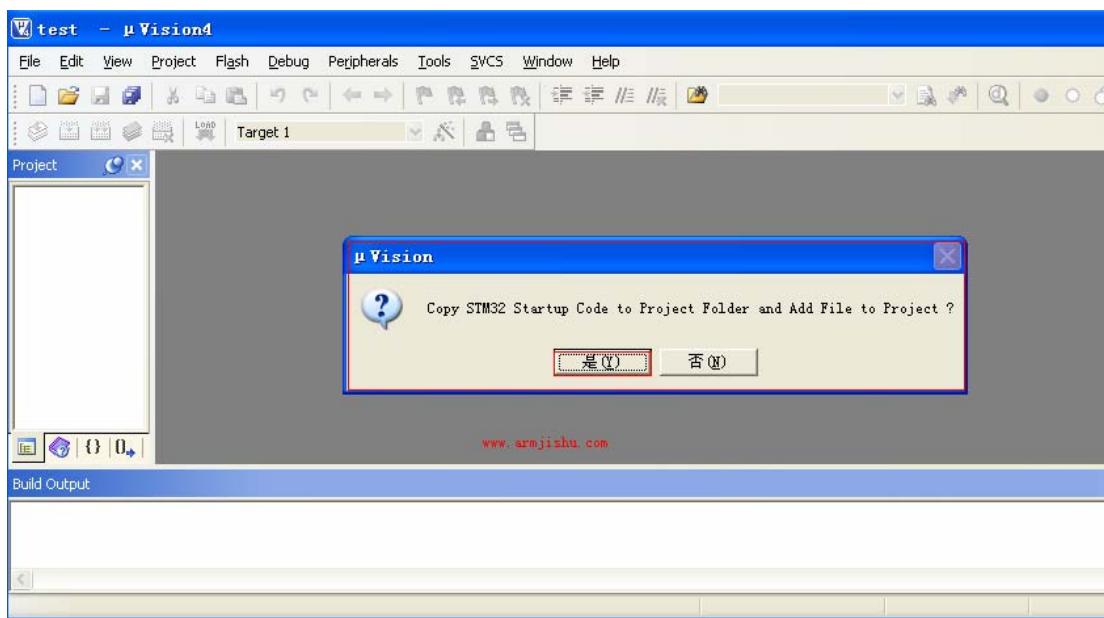
2) 在对话框，选择放在刚才建立的“新建文件夹”下，给这个工程取个名后保存，不需要填后缀：

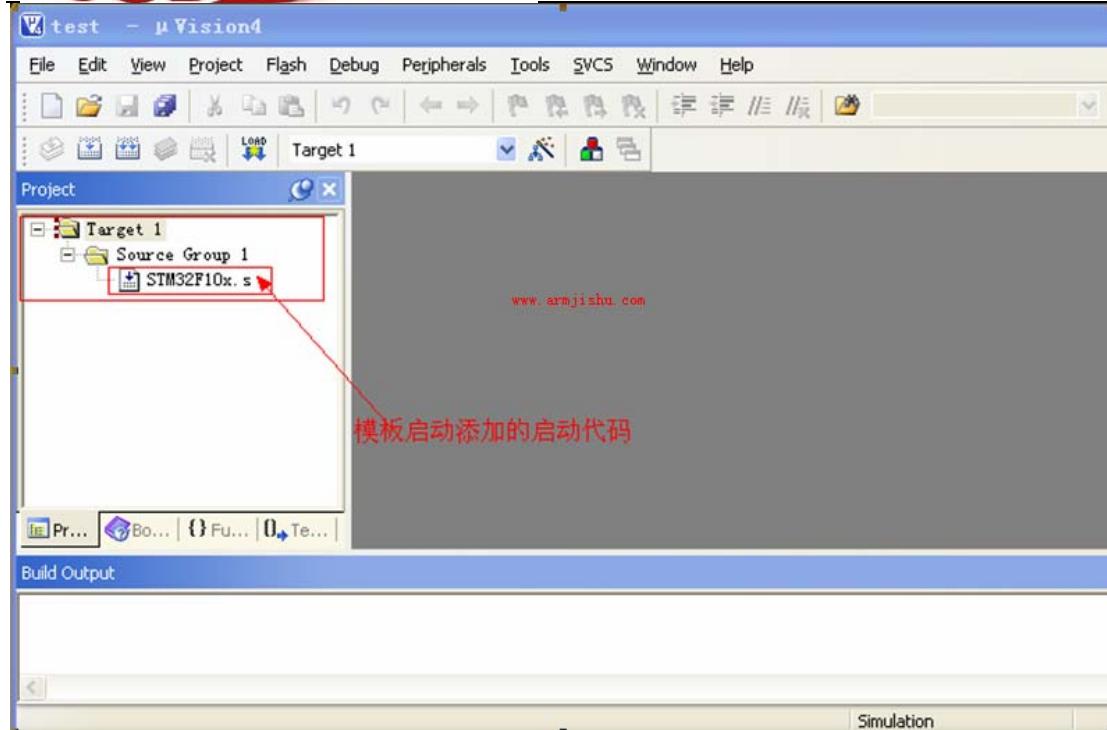


3) 弹出一个框，在 CPU 类型下我们找到并选中 “STMicroelectronics” 下的 STM32F103ZET:



- 4) 出现一个提示框，是否复制 STM32 启动代码到工程文件夹，我们选择【是】，就可以看到 STM32 的启动代码自动添加进来了：

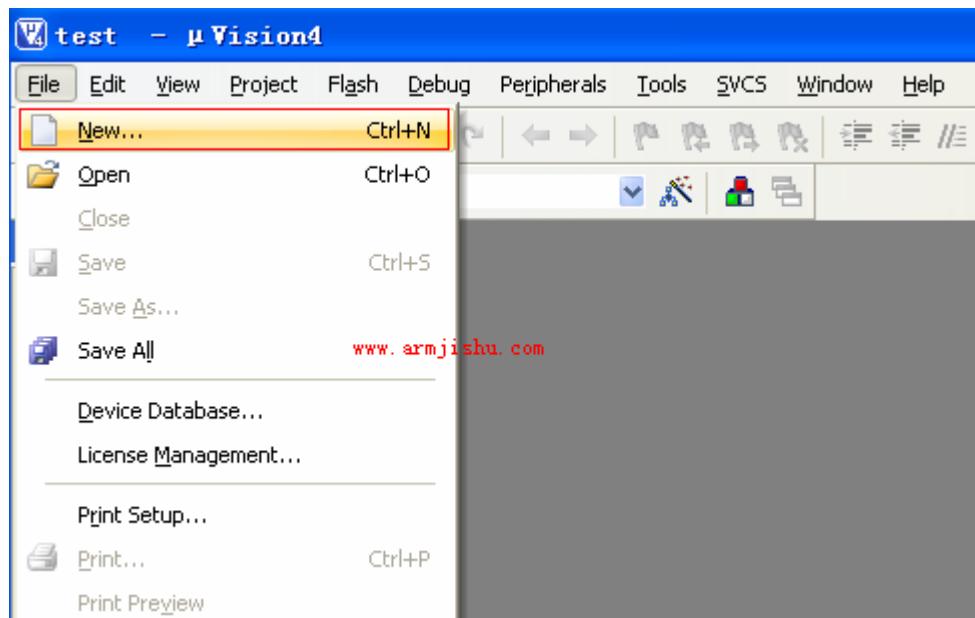




5) 到这里工程全部建立完毕

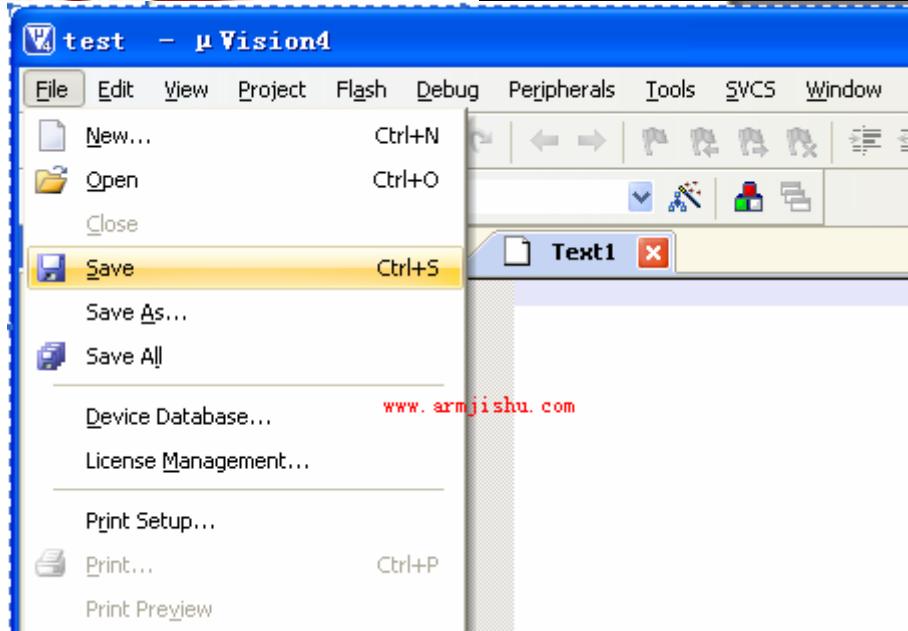
2.5.2 建立文件

建立一个文本文件，以便输入用户程序。点击【File】→【New...】图标按钮，



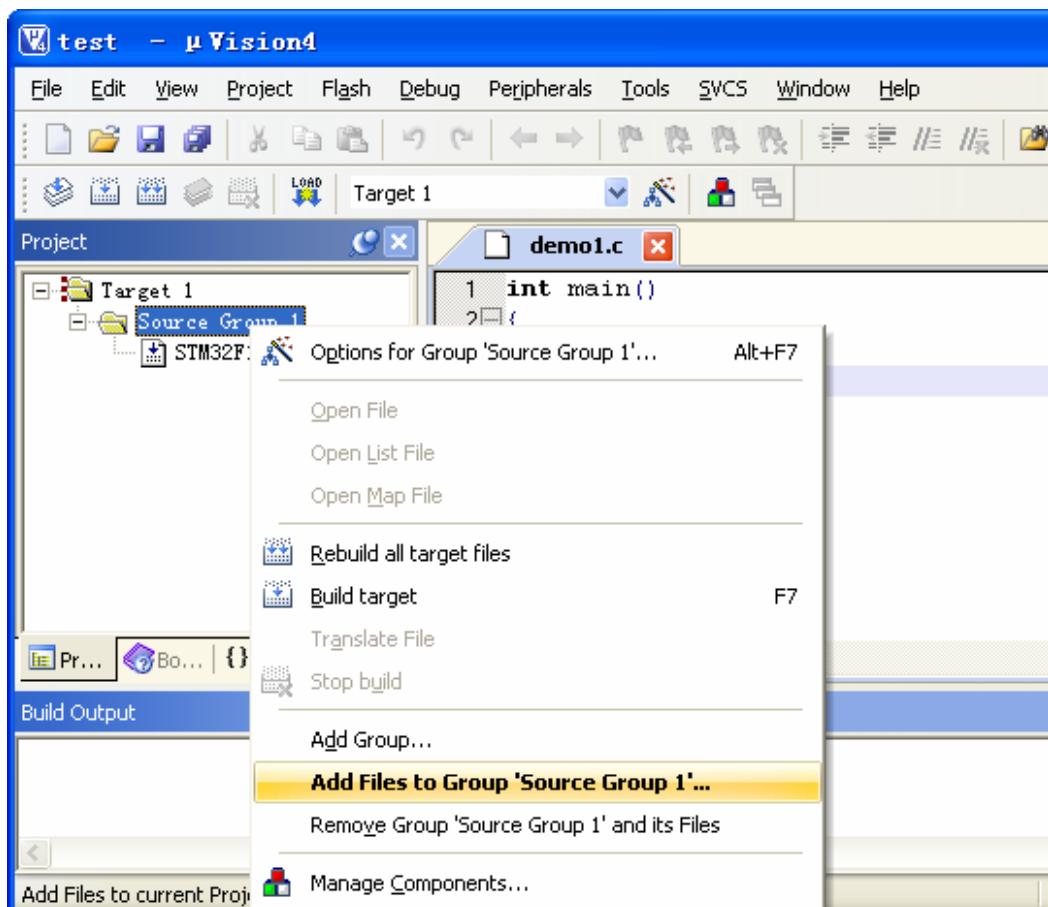
然后在新建的文件中编写程序，点击“Save”图标按钮将文件存盘，输入文件全名，如 demo.c。注意，请将文件保存到相应工程的目录下，以便于管理和查找。

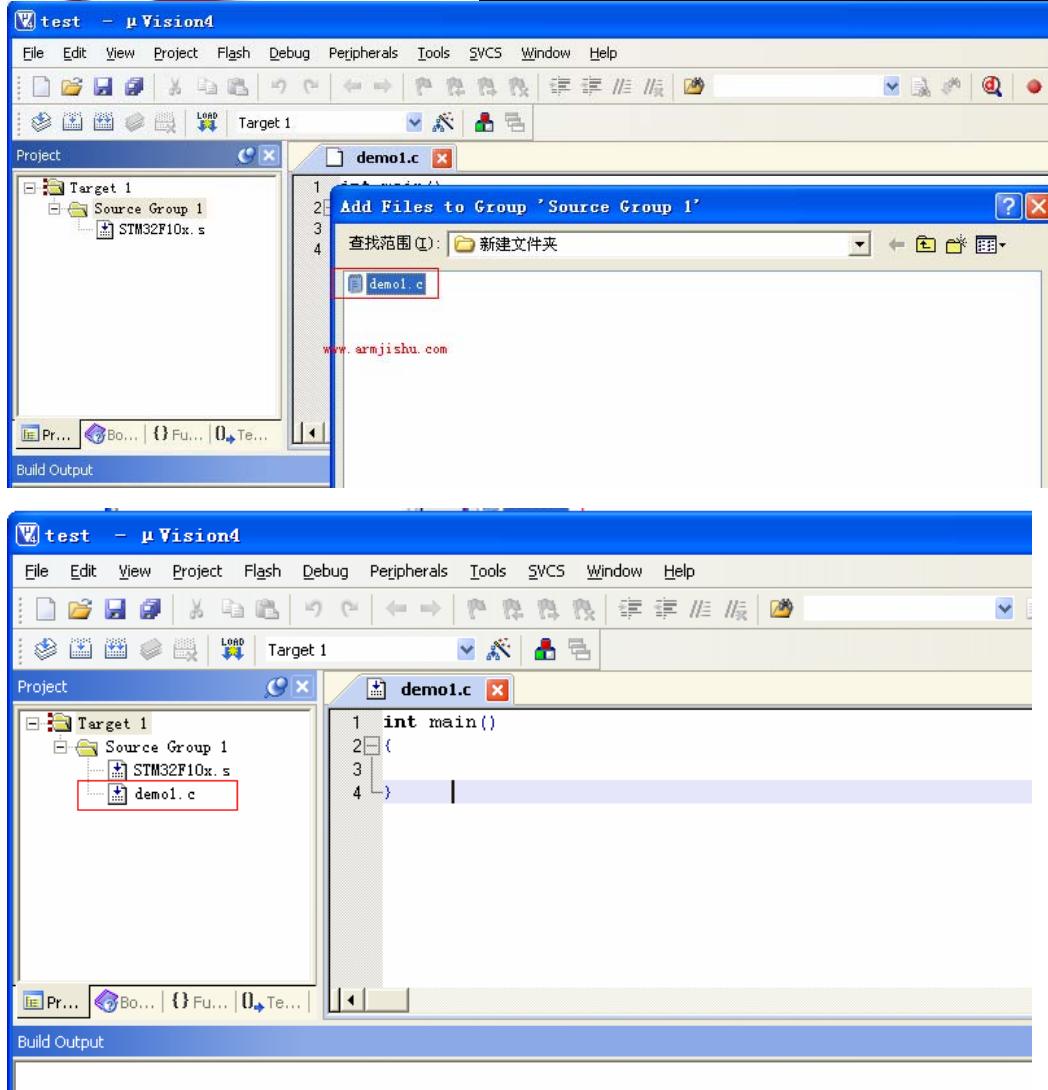
当然，您也可以使用其他文本编辑器建立或编辑源文件。



2.5.3 添加文件到工程

选择 Target1 下面的组，这里是“Source Group1”，单击右键，选择【Add File to Group....】，选定想要添加到文件，确认即可。

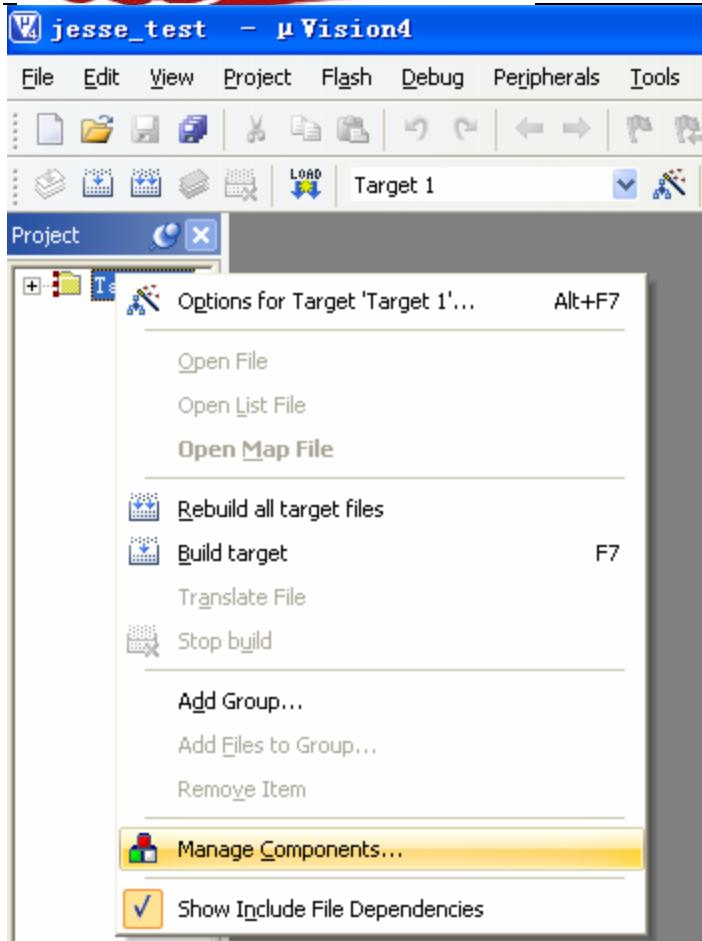




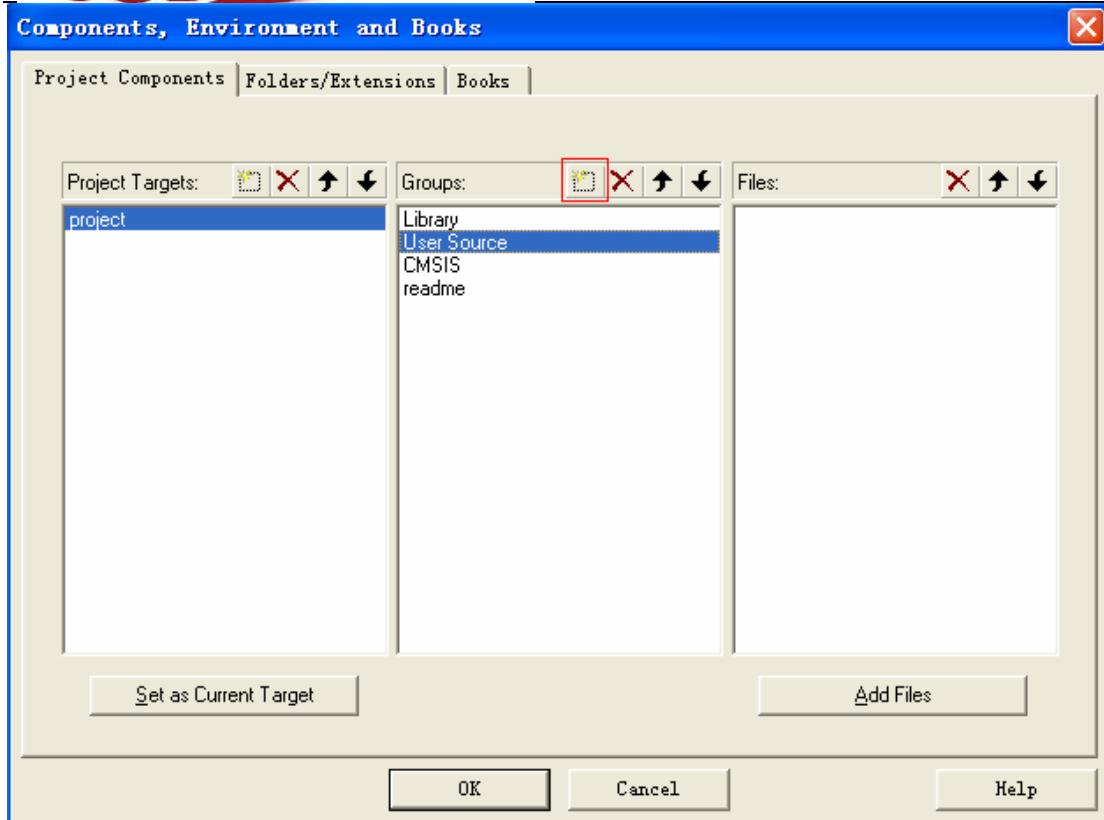
2.5.4 管理工程目录以及源文件

当我们的项目比较庞大时，我们就需要一个管理平台来管理我们的工程文件，以及将一些同类型的工程文件分好类，放入相对应的文件夹中，而这个管理工程文件以及目录的平台就是下面介绍到的：

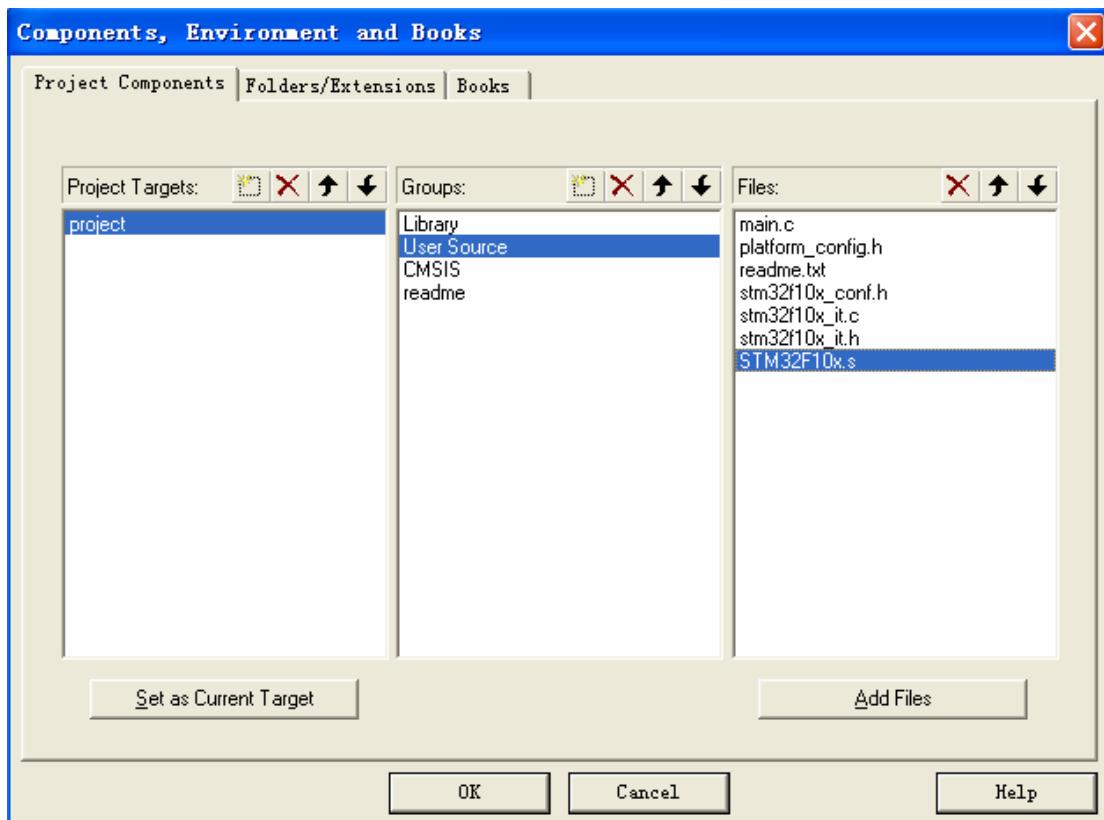
1. 右击工程窗口中的 Target1 选择“Manage Components……”

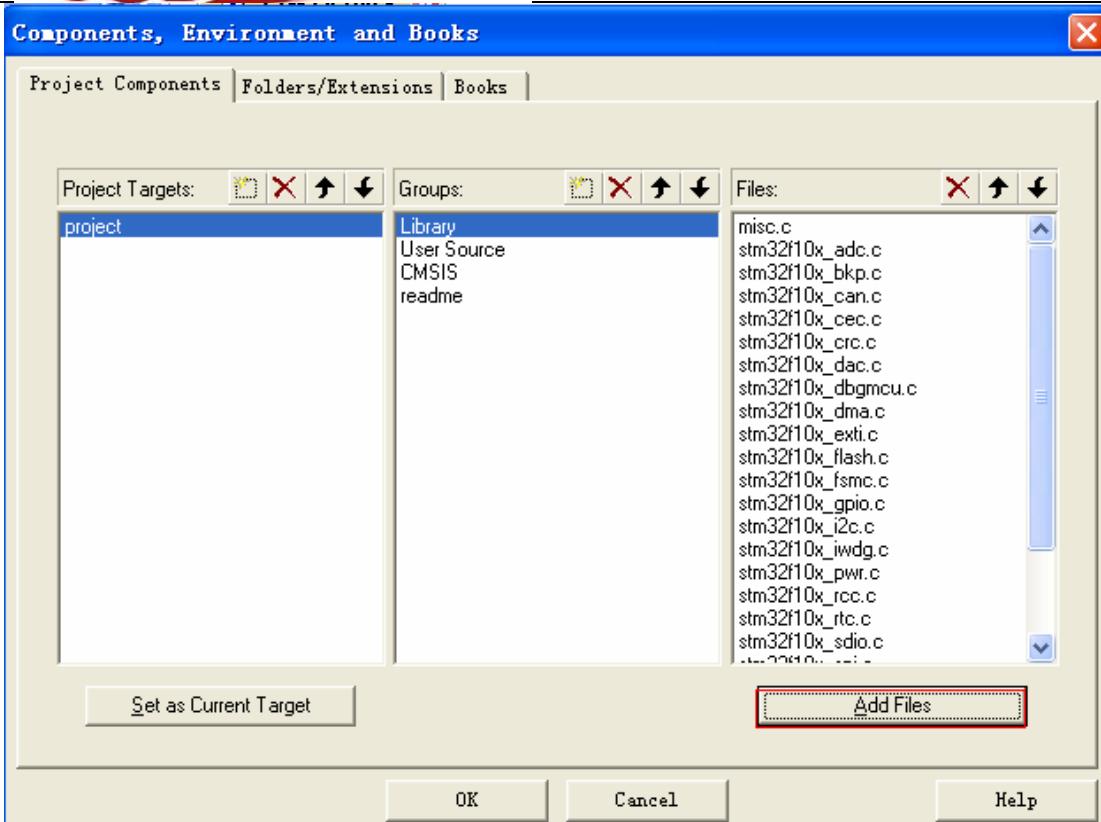


2. 在 Project Target 中把 Target1 改成你想要的名字，可以不改，这里改为 project，然后在 groups 单击新建按钮，这些组对应我们实实在在的文件夹，方便源文件的分类和管理，例如我们这里新建 User Source, Library, CMSIS, 和 readme 四个组：

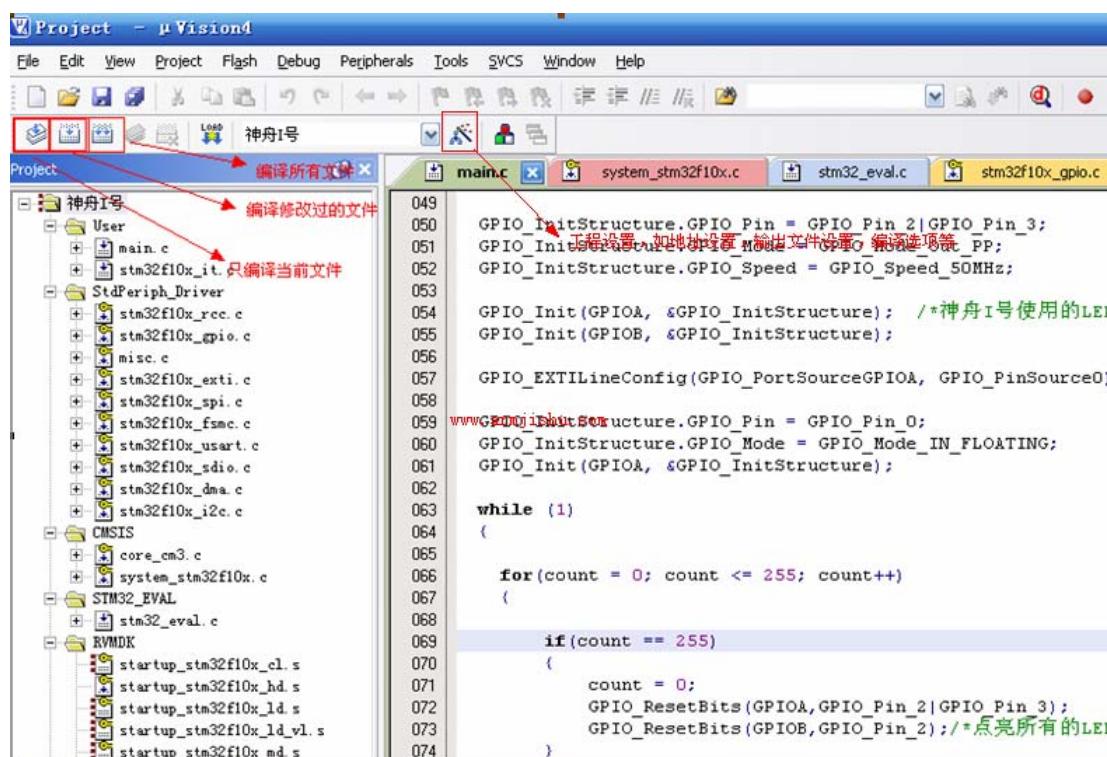


3. 先选中一个组之后，在 Files 点击 Add Files，例如在 User Source 组添加 User 文件夹中的源文件，添加完成该文件就成功添加到工程中了：





2.5.5 编译和连接工程



1. 编译的几个选项介绍：

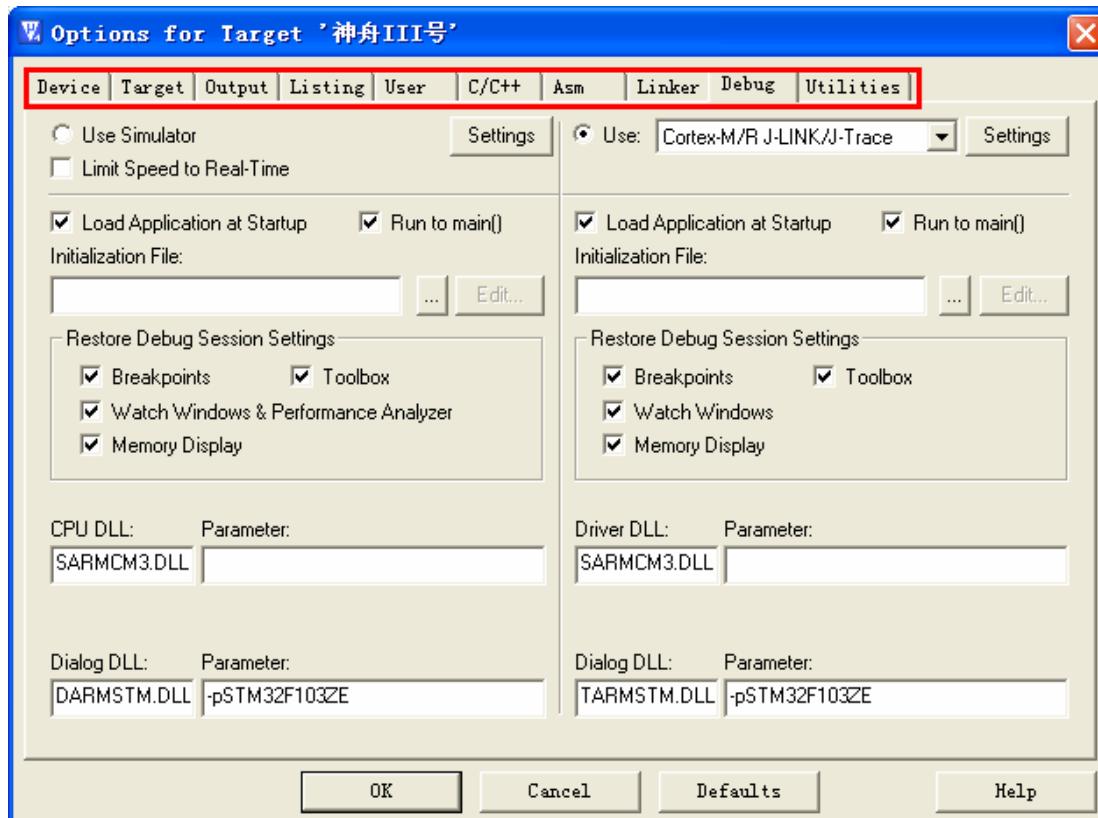
下面是编译程序的三种情况，具体请看上图：

- 编译当前文件：如果只是确认当前文件是否存在问题，可以选择这个按键

- 编译修改过的文件并链接：如果只是确认最近的修改是否存在问题，可以选择这个按键
- 编译所有文件：选择这个按键将重新编译链接整个工程文件。

1、关于工程设置框：

上面还有个红框是工程设置相关的，例如工程设置，如地址设置，输出文件设置，编译选项等；如果我们点击它，就会出现下面的图框，我们可以在这里根据需要进行一系列详细的设置。



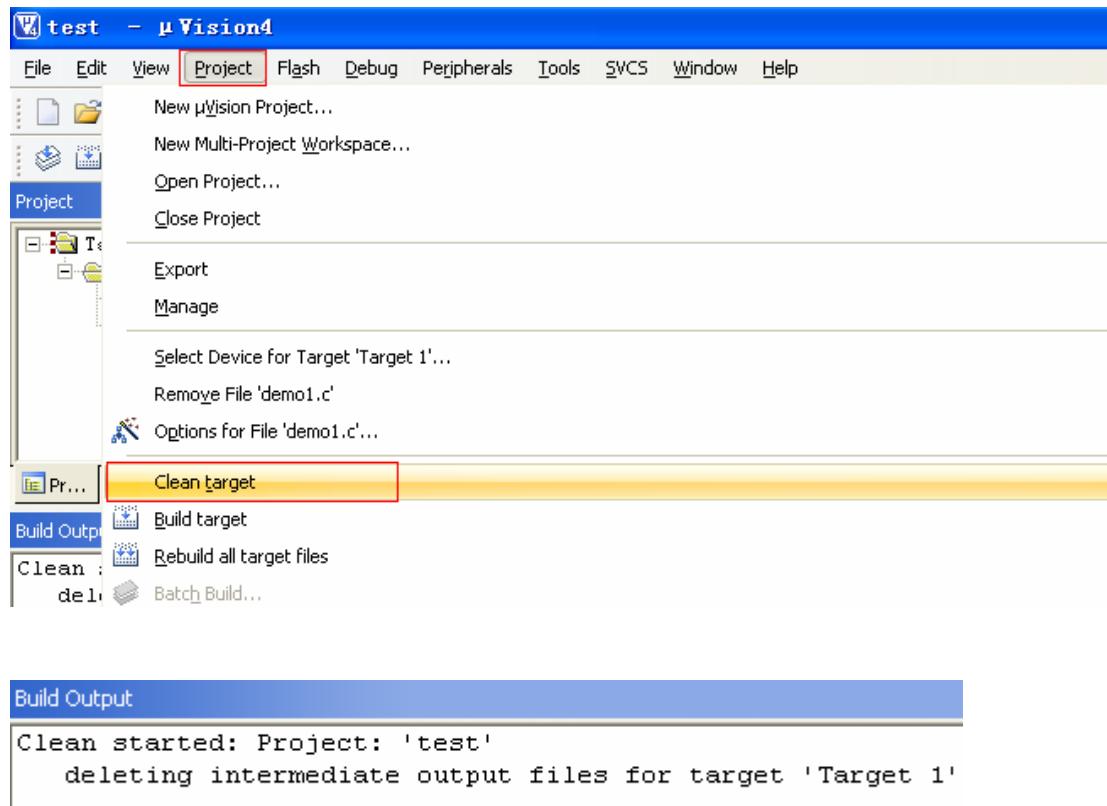
2、调试时输出的错误：

对于简单的软件调试，可以不进行连接地址的设置，直接点击工程窗口的“build”图标按钮，即可完成编译连接。若编译出错，会有相应的出错提示，双击出错提示行信息，编辑窗即会使用光标指出当前出错的源代码行。

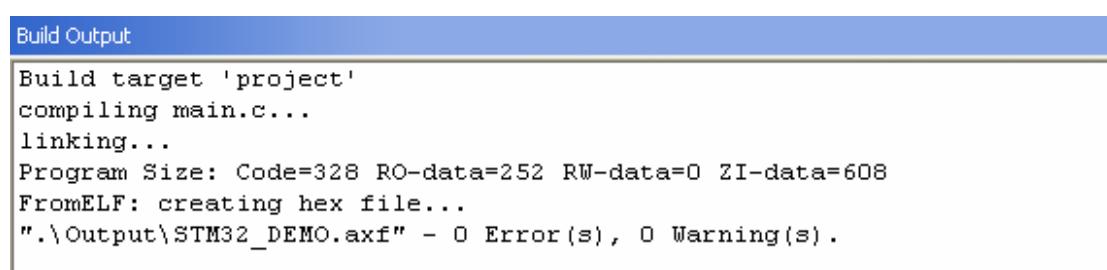
```
Build Output
Build target 'Target 1'
assembling STM32F10x.s...
compiling demo1.c...
demo1.c(4): warning: #1-D: last line of file ends without a newline
demo1.c: }
demo1.c: ^
demo1.c(3): error: #20: identifier "ddd" is undefined
demo1.c:         ddd
demo1.c: ^
demo1.c(4): error: #65: expected a ";""
demo1.c: }
demo1.c: ^
demo1.c: demo1.c: 1 warning, 2 errors
Target not created
```

4. 清除编译之后的垃圾：

重新编译之前，建议将原来生成的目标文件都删除，方法如下，点选“project”下拉选择“Clean Target”，删除所有旧目标文件后再进行编译：



5. 按 键，编译工程，得到结果如下图所示（不同工程显示内容大同小异）：



可以看到没有错误，也没有警告。从编译信息可以看出，我们的代码占用 FLASH 大小为：580 字节 (328+252)

这里我们解释一下，编译结果里面的几个数据的意义：

Code: 表示程序所占用 FLASH 的大小 (FLASH)

RO-data: 即 Read Only-data，表示程序定义的常量 (FLASH)

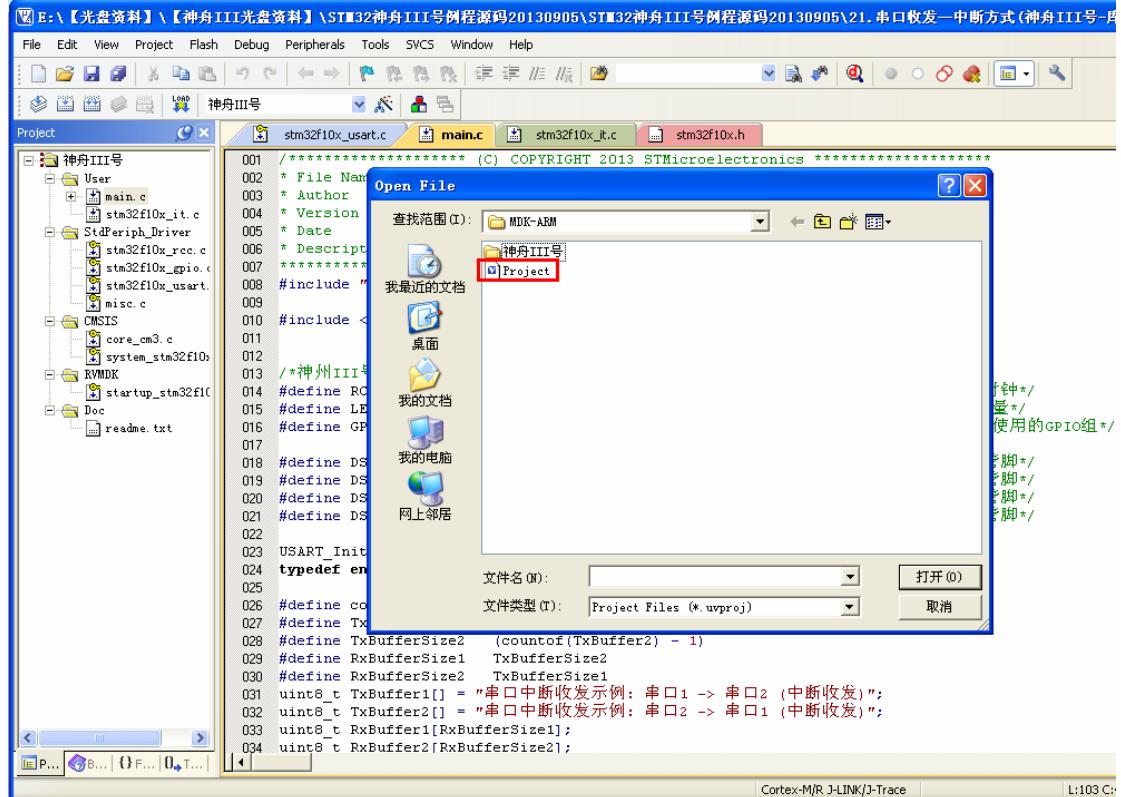
RW-data: 即 Read Write-data，表示已可以读写的变量 (SRAM)

ZI-data: 即 Zero Init-data，表示已被初始化为 0 的变量 (SRAM)

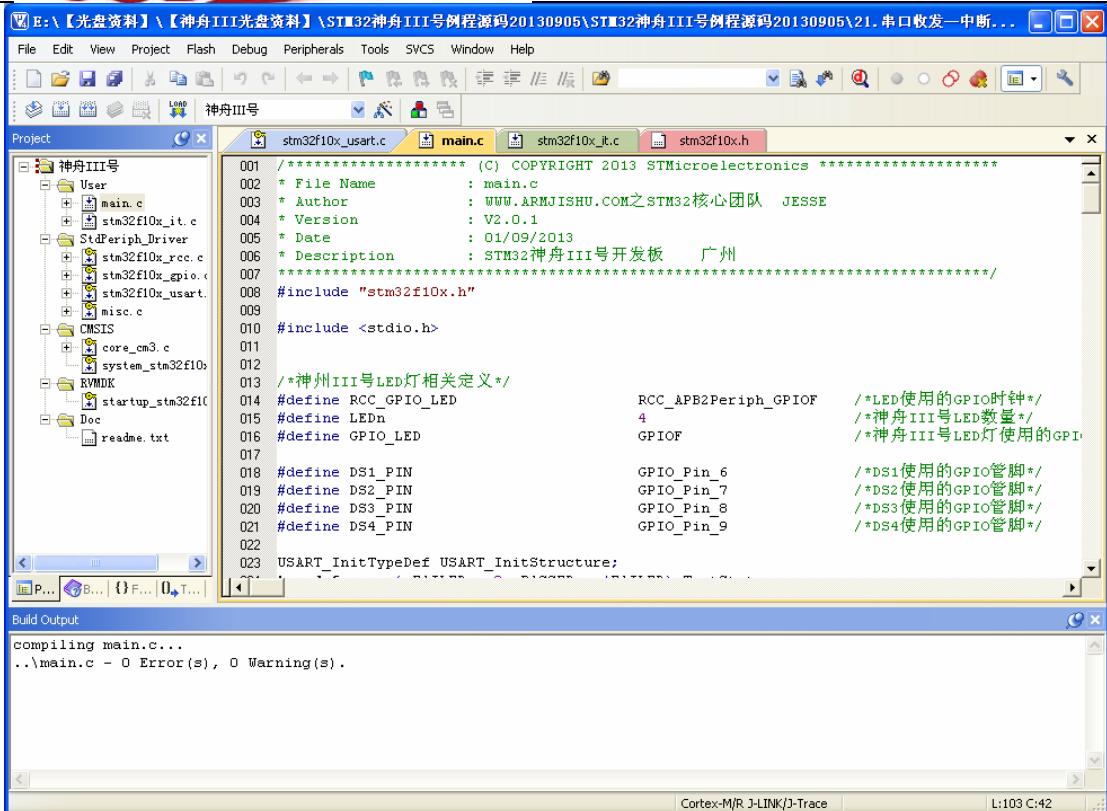
有了这个就可以知道当前使用的 flash 和 sram 大小了，所以，大家不要误解程序的大小就是.hex 文件的大小，这是不对的，程序的大小是编译后的 Code 和 RO-data 之和。

2.5.6 打开旧工程

点击【File】菜单，选择【Open】即弹出“打开”对话框，找到相应的工程文件 (*.uvproj)，单击【打开】即可：



在工程窗口中，双击源程序的文件名即可打开该文件进行编辑：



2.6 RVMDK使用技巧

前面介绍了 RVMDK 的基本使用，接下来简单的介绍一下 RVMDK 的几个使用技巧。

2.6.1 快速定位函数/变量被定义的地方

你在调试代码或编写代码的时候，一定有想看看某个函数是在那个地方定义的，具体里面的内容是怎么样的，也可能想看看某个变量或数组是在哪个地方定义的等。尤其在调试代码或者看别人代码的时候，如果编译器没有快速定位的功能的时候，你只能慢慢的自己找，代码量比较少还好，如果代码量一大，那就郁闷了，有时候要花很久的时间来找这个函数到底在哪里。幸好 MDK 提供了这样的快速定位的功能。只要你把光标放到这个函数/变量的上面，然后右键，如下图所示：

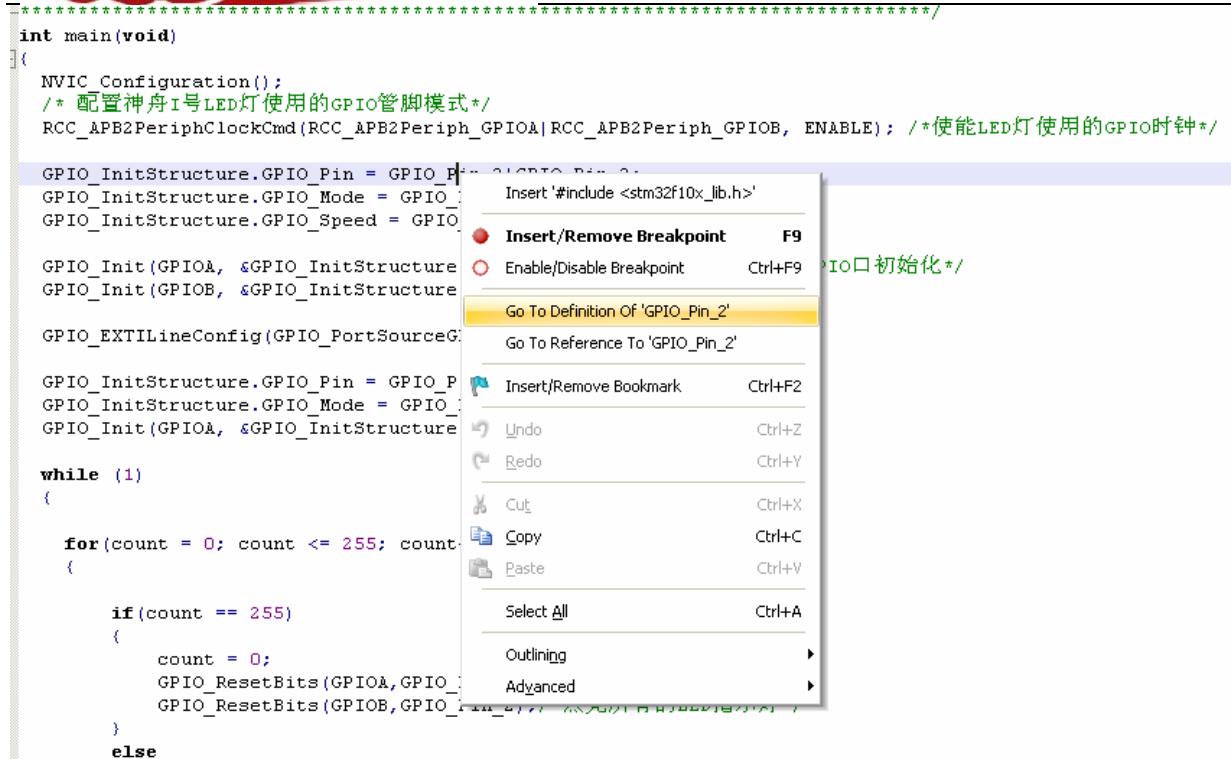


图 2.8.2.3 快速定位

在上图，我们点击 Go To Definition of 'GPIO_Pin_2'，跳转查看 GPIO_Pin_2 是在何处，如何被定义的。如下图所示

```

126 #define GPIO_Pin_0 ((uint16_t)0x0001) /*!< Pin 0 selected */
127 #define GPIO_Pin_1 ((uint16_t)0x0002) /*!< Pin 1 selected */
128 #define GPIO_Pin_2 ((uint16_t)0x0004) /*!< Pin 2 selected */
129 #define GPIO_Pin_3 ((uint16_t)0x0008) /*!< Pin 3 selected */
130 #define GPIO_Pin_4 ((uint16_t)0x0010) /*!< Pin 4 selected */
131 #define GPIO_Pin_5 ((uint16_t)0x0020) /*!< Pin 5 selected */
132 #define GPIO_Pin_6 ((uint16_t)0x0040) /*!< Pin 6 selected */
133 #define GPIO_Pin_7 ((uint16_t)0x0080) /*!< Pin 7 selected */
134 #define GPIO_Pin_8 ((uint16_t)0x0100) /*!< Pin 8 selected */
135 #define GPIO_Pin_9 ((uint16_t)0x0200) /*!< Pin 9 selected */
136 #define GPIO_Pin_10 ((uint16_t)0x0400) /*!< Pin 10 selected */
137 #define GPIO_Pin_11 ((uint16_t)0x0800) /*!< Pin 11 selected */
138 #define GPIO_Pin_12 ((uint16_t)0x1000) /*!< Pin 12 selected */
139 #define GPIO_Pin_13 ((uint16_t)0x2000) /*!< Pin 13 selected */
140 #define GPIO_Pin_14 ((uint16_t)0x4000) /*!< Pin 14 selected */
141 #define GPIO_Pin_15 ((uint16_t)0x8000) /*!< Pin 15 selected */
142 #define GPIO_Pin_All ((uint16_t)0xFFFF) /*!< All pins selected */

```

上面是演示的是一个变量的定义的查找，对于函数，我们也可以按这样的操作快速来定位函数被定义的地方，大大缩短了你查找代码的时间。

2.6.2 快速注释与快速消注释

接下来，我们介绍一下快速注释与快速消注释的方法。在调试代码的时候，你可能会想注释某一片的代码，来看看执行的情况，MDK 提供了这样的快速注释/消注释块代码的功能。也是通过右键实现的。这个操作比较简单，首先选中你要注释的代码区，然后右键，选择 Advanced->Comment Selection 就可以了。

以 Turn_On_LED 函数为例，比如我要注释掉下图中所选中区域的代码，如下图所示：

```
087 /*点亮对应灯*/
088 void Turn_On_LED (u8 LED_NUM)
089 {
090     switch(LED_NUM)
091     {
092         case 0:
093             GPIO_ResetBits(GPIOA,GPIO_Pin_2); /*点亮DS5灯*/
094             break;
095         case 1:
096             GPIO_ResetBits(GPIOB,GPIO_Pin_2); /*点亮DS3灯*/
097             break;
098         case 2:
099             GPIO_ResetBits(GPIOA,GPIO_Pin_3); /*点亮DS4灯*/
100             break;
101     default:
102         GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);
103         GPIO_ResetBits(GPIOB,GPIO_Pin_2); /*点亮所有的LED指示灯*/
104         break;
105 }
```

我们只要在选中了之后，选择右键，再选择 Advanced->Comment Selection 就可以把这段代码注释掉了。执行这个操作以后的结果如下图所示：

```
087 /*点亮对应灯*/
088 void Turn_On_LED (u8 LED_NUM)
089 {
090     switch(LED_NUM)
091     {
092     //         case 0:
093     //             GPIO_ResetBits(GPIOA,GPIO_Pin_2); /*点亮DS5灯*/
094     //             break;
095     //         case 1:
096     //             GPIO_ResetBits(GPIOB,GPIO_Pin_2); /*点亮DS3灯*/
097     //             break;
098     //         case 2:
099     //             GPIO_ResetBits(GPIOA,GPIO_Pin_3); /*点亮DS4灯*/
100     //             break;
101     //
102     default:
103         GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);
104         GPIO_ResetBits(GPIOB,GPIO_Pin_2); /*点亮所有的LED指示灯*/
105 }
```

这样就快速的注释掉了一片代码，而在某些时候，我们又希望这段注释的代码能快速的取消注释，MDK 也提供了这个功能。与注释类似，先选中被注释掉的地方，然后通过右键->Advanced，不过这里选择的是 Uncomment Selection。

2.6.3 快速打开头文件

将光标放到要打开的引用头文件上，然后右键选择 Open Document“XXX”，就可以快速打开这个文件
嵌入式专业技术论坛（www.arm{jishu}.com）出品

了 (XXX 是你要打开的头文件名字)。如下图所示:

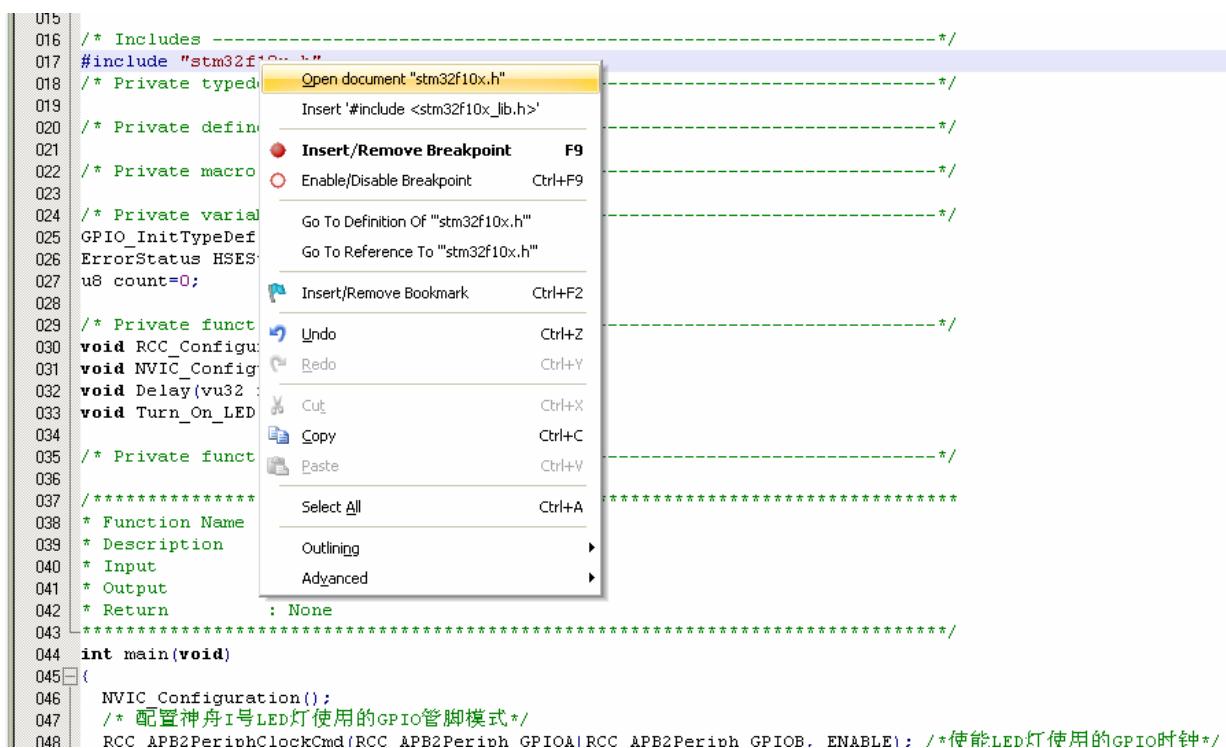


图 2.8.2.7 快速打开头文件

关于 MDK 软件的使用就介绍到此，如需更深入和全面的了解请用户查看 KEIL 软件用户使用手册。

2.7 JLINK V8仿真器的安装与应用

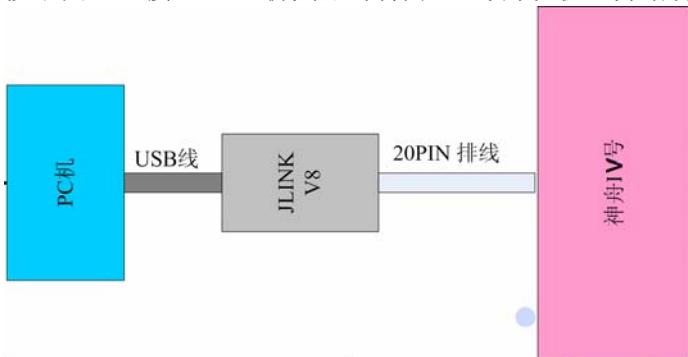
在代码编译通过后，我们需要验证程序是否与我们的设计预期相符合，一般有几种方式来验证。

第一种，软件仿真。在没有硬件环境的情况下，我们可以使用 MDK 的软件仿真功能来对代码的功能进行初步验证。关于软件仿真功能的使用，在本文档中不在详细描述，有兴趣的朋友，可以常看 MDK 相关手册和说明文档。

第二种，在线仿真。如果硬件环境允许的情况下，建议使用在线仿真来进行调试，完全与实际应用相符。神舟 III 号所以实验程序都使用 JLINK 仿真器仿真调试通过。

JLINK 是一款主流的支持 ARM 内核芯片的 JTAG 仿真器，配合 IAR, KEIL, WINARM, RealView 等集成开发环境，支持所有 ARM7/ARM9/Cortex-M3 内核芯片的仿真。在这里，我们以神舟 III 号的入门程序为例，说明如何在 MDK 开发环境中，搭配 JLINK 仿真器，在线仿真调试程序。

首先按下图，连接 JLINK 仿真器与神舟 III 号开发板。并给神舟 III 号上电。



2.7.1 JLINK V8仿真器简介

J-LINK 是 SEGGER 公司为支持仿真 ARM 内核芯片推出的 JTAG 通用仿真器。配合 IAR EWARM, ADS, KEIL, WINARM, RealView 等集成开发环境，支持所有 ARM7/ARM9/ARM11 和 Cortex-M0/M1/M3 核内核芯片的仿真，通过 RDI 接口和 IAR EWARM, ADS, KEIL, WINARM, RealView 等各集成开发环境无缝连接，操作方便、连接方便、简单易学，是学习开发 ARM 最好最实用的开发工具。

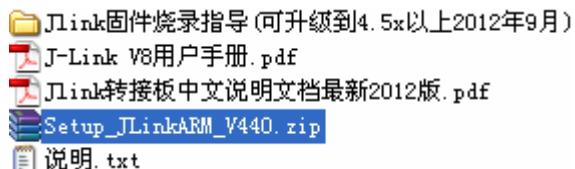
2.7.2 JLINK ARM主要特点

- ◆ USB 接口符合 USB2.0 规范
- ◆ 标准 20 芯 JTAG 接口
- ◆ 支持全系列 ARM 7/9/11, Cortex_M0/M1/M3 ARM 核，包括 Thumb 模式
- ◆ IAR EWARM 集成开发环境无缝连接的 JTAG 仿真器
- ◆ USB 接口供电，无需外接电源
- ◆ J-LINK 支持对目标板 5V (300mA), 3.3V(400mA)供电
- ◆ 带 USB 连接线和 20 芯扁平电缆
- ◆ 支持 RDI 接口，J-LINK 可用于具有 RDI 接口的开发环境，支持主流的开发环境，包括 ADS,IAR,KEIL,WINARM,REALVIEW 等。
- ◆ 下载速度高达 ARM7:600kB/s, ARM9:550kB/s, 通过 DCC 最高可达 800 kB/s

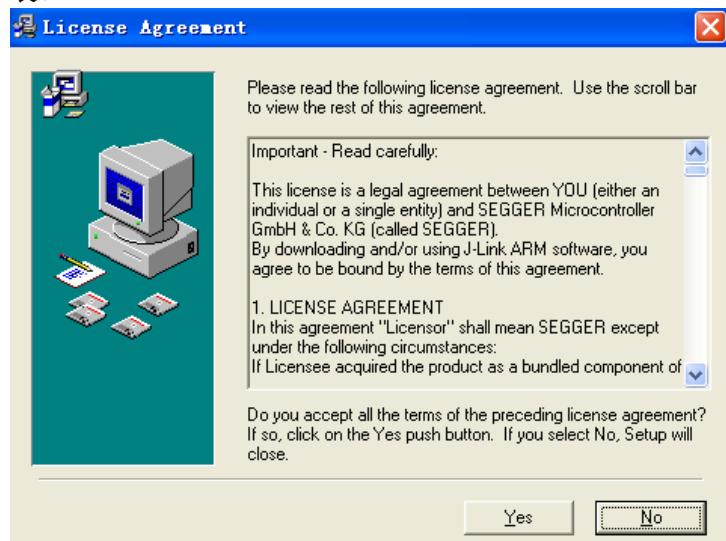
- ◆ 最高 JTAG 速度 12 MHz
- ◆ 目标板电压范围 1.2V – 3.3V
- ◆ 自动速度识别功能
- ◆ 监测所有 JTAG 信号和目标板电压
- ◆ 完全即插即用
- ◆ 支持多 JTAG 器件串行连接
- ◆ 支持两种下载仿真调试接口：JTAG 接口方式和 SWD 两线接口方式

2.7.3 如何安装JLINK软件

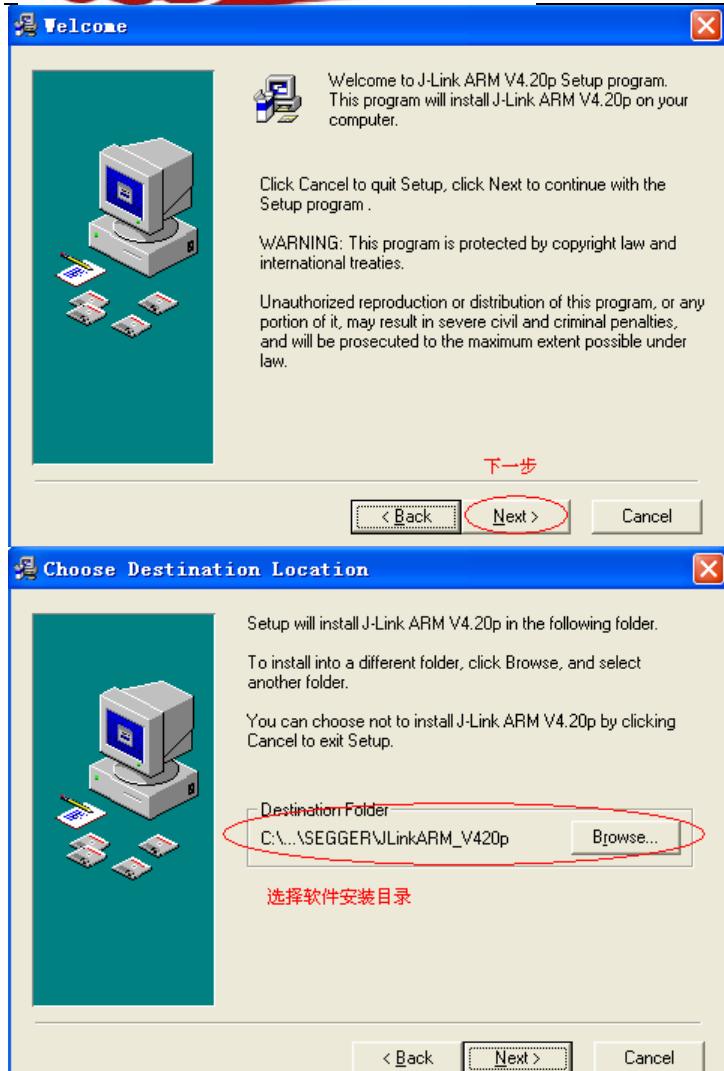
在用 JLINK 下载和调试程序之前，我们需要在电脑上安装 JLINK 驱动，如果电脑上已经安装 JLINK 驱动，则可跳过这一步。JLINK 仿真器驱动程序可以从配套光盘获得。（文件路径：\神舟 III 号光盘\神舟 III 号光盘资料(不包含源码和手册)\4.软件工具\JLINK 驱动）



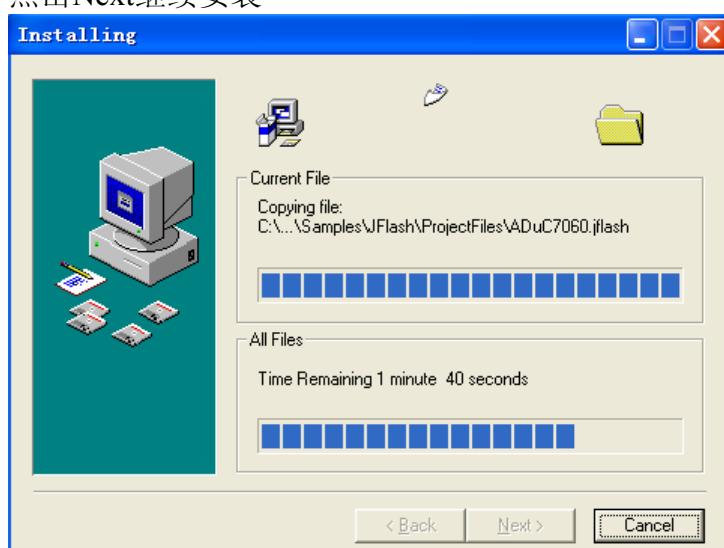
也可以在SEGGER 网站(<http://www.segger.com/cms/jlink-software.html>)获取最新的安装文件，解压后双击执行，出现下图所示的安装界面，根据界面安装向导的提示，完成JLINK仿真器驱动程序的安装。



点击Yes继续安装



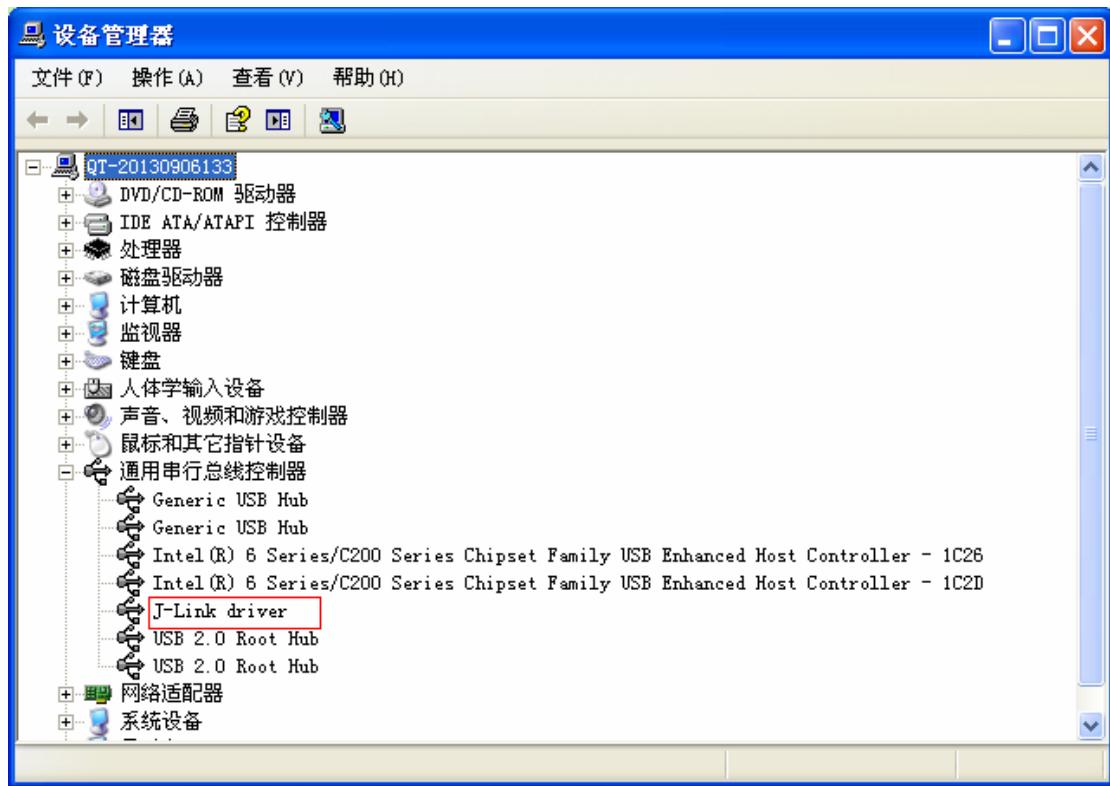
点击Next继续安装



如果你还在以前装了 IAR 的集成开发环境，提示你要选择更新 IAR JLINK 仿真器的动态链接库文件，一般选上后按 OK 完成 JLINK 仿真器的安装。

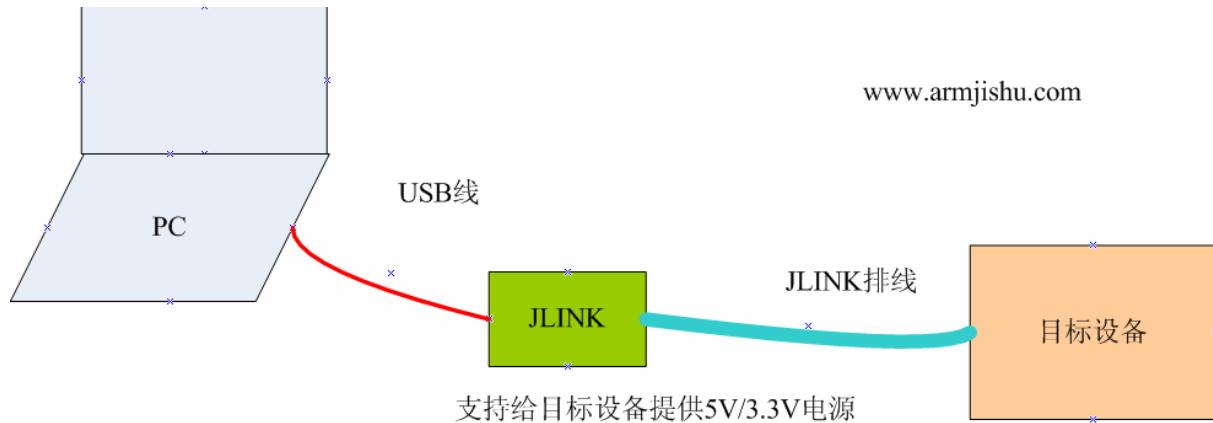
安装完成后，我们将 JLINK 通过 USB 方口线连接到电脑 USB 口。即可在“我的电脑\设备管理器\通用串行总线控制器”中看到一个 J-link driver。要注意的是，没有连接 JLINK

时在“设备管理器”中是看不到 J-link driver 的。



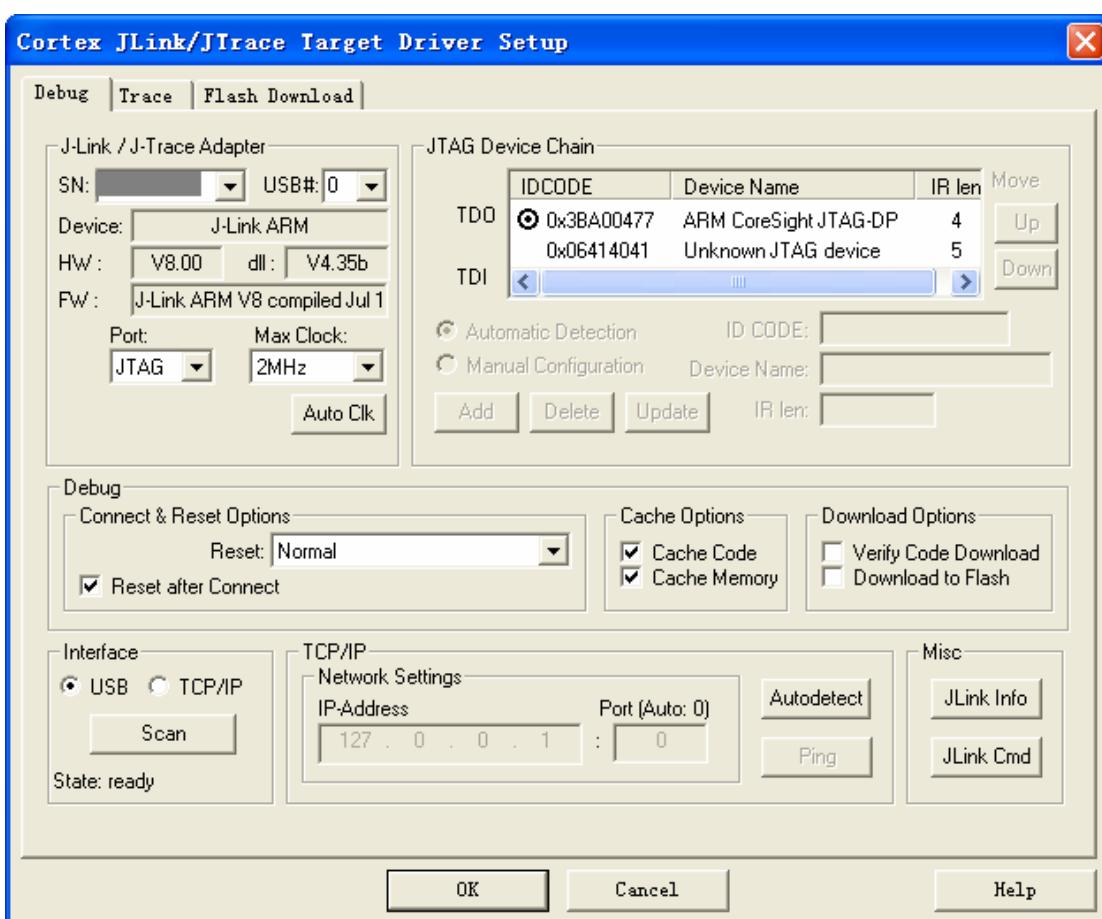
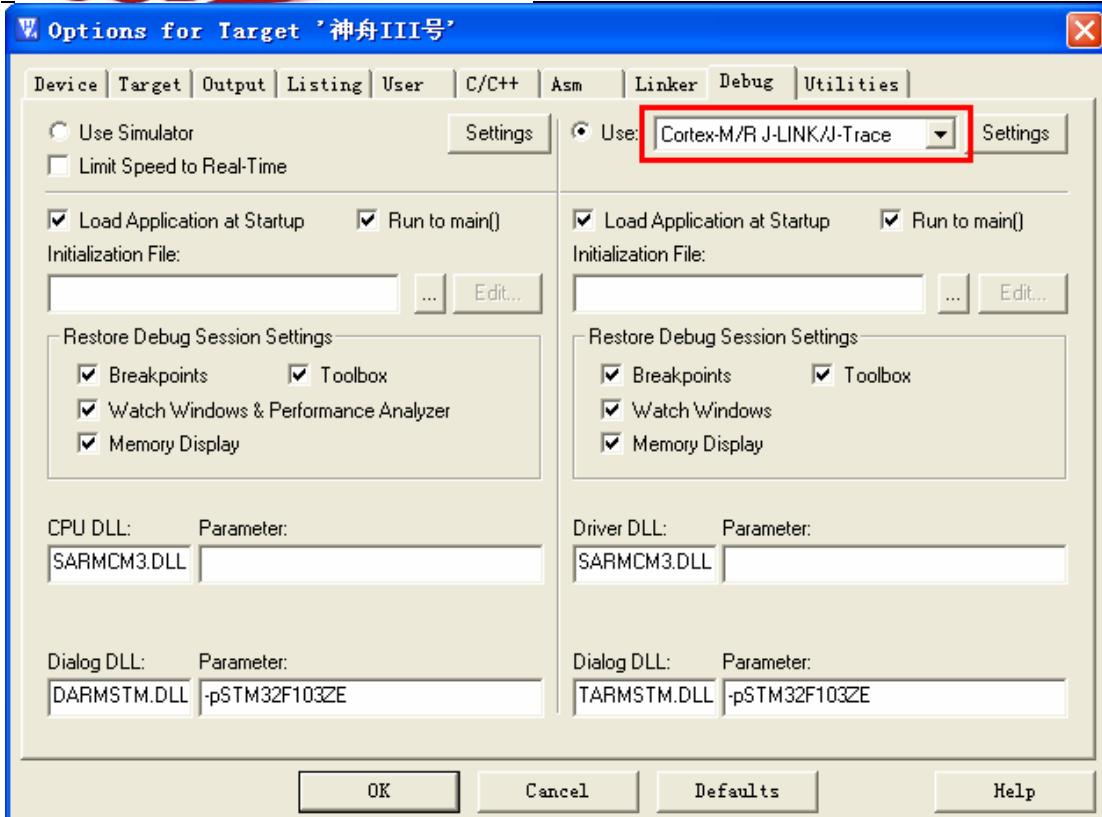
2.7.4 JLINK V8仿真器配置（MDK KEIL环境）

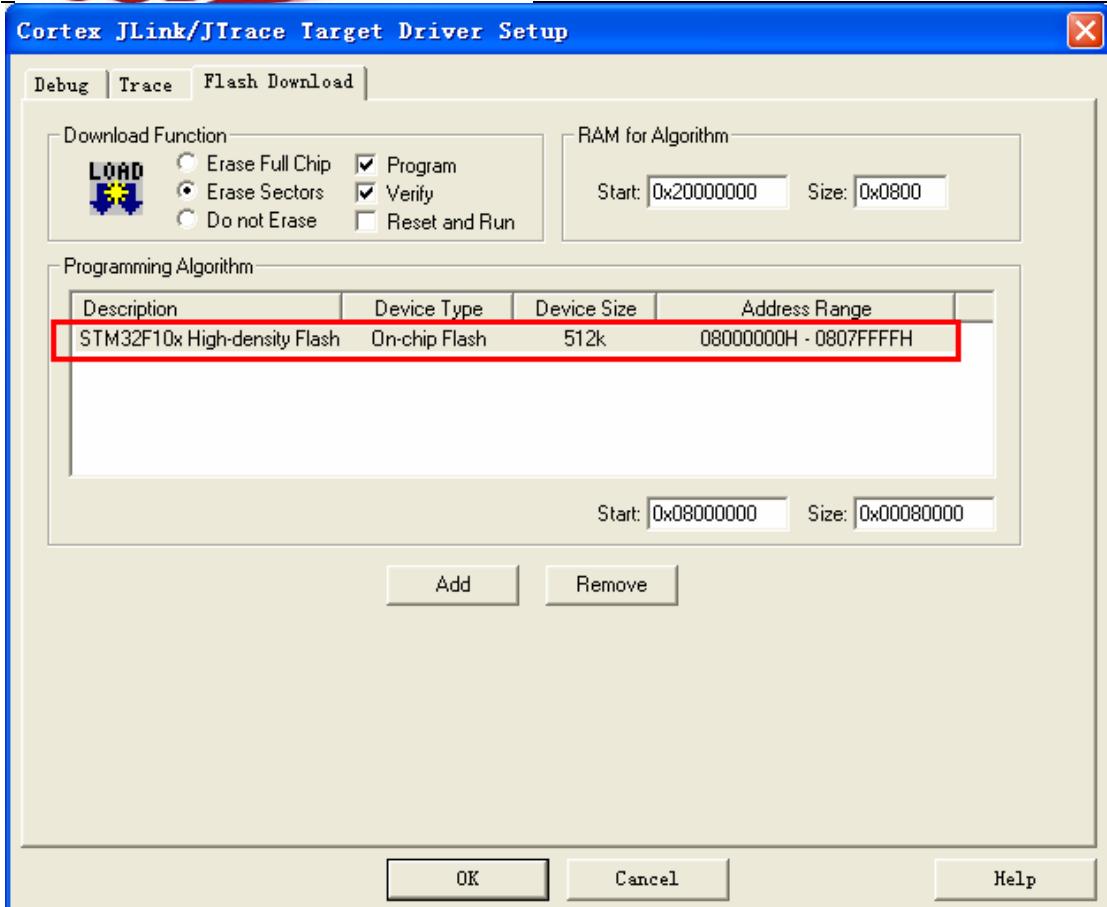
按下图所示，连接 JLINK 和目标设备。



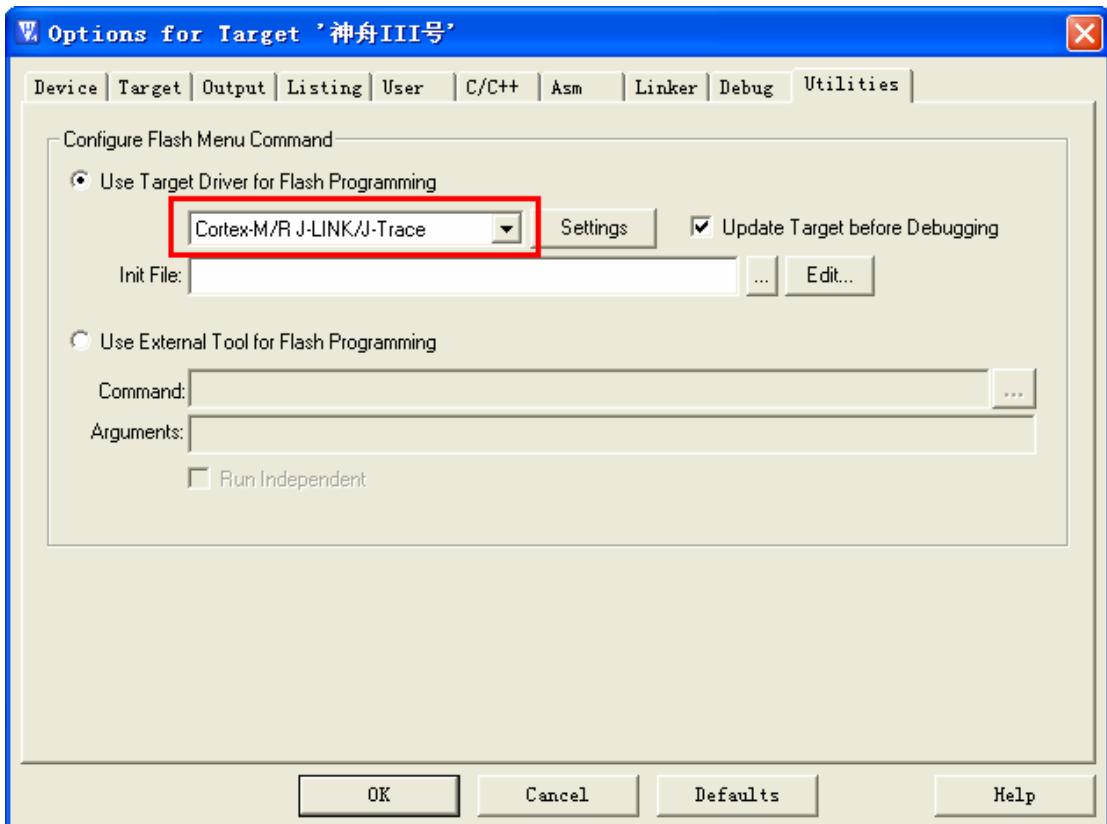
使用 J-LINK 进行 DEBUG 的设置

打开 KEIL 工程后，选择项目设置的 Debug 菜单。如下图选择“Cortex-M/R J-LINK/J-Trace”，点击“Settings”





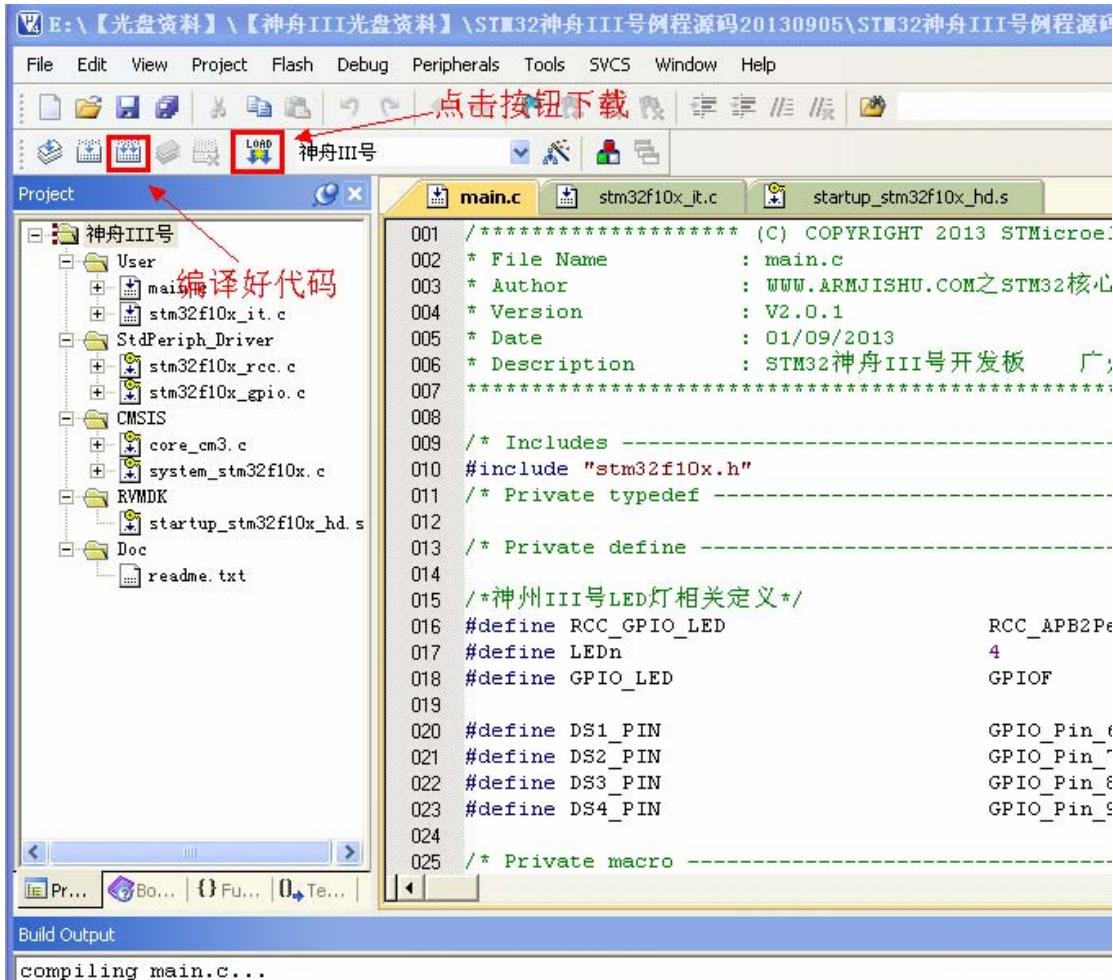
因为 STM32F103ZET 的内部 ROM 大小是 512K 的，所以这里设置 512K 的大小。



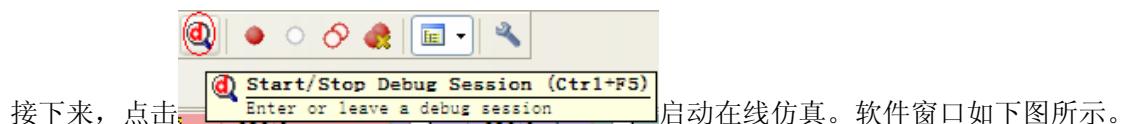
最后设置好之后，点击‘OK’按钮即可

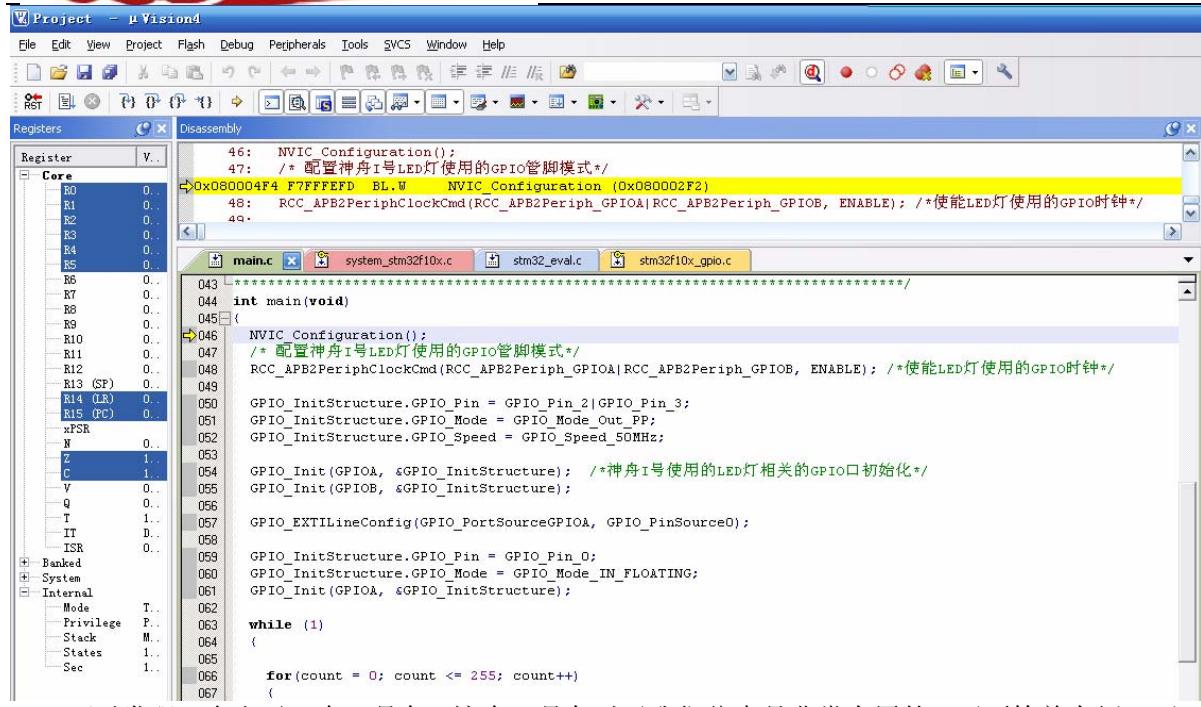
2.7.5 使用KEIL的DOWNLOAD功能

如果要使用 KEIL 提供的 即“DOWNLOAD”功能则在完成 [前一步](#) 的设置外，还需要在“Utilities”菜单里面进行和“Debug”一样的设置：



2.8 在MDK开发环境中JLINK V8的调试技巧





可以发现，多出了一个工具条，这个工具条对于我们仿真是非常有用的，下面简单介绍一下工具条相关按钮的功能，工具条部分按钮的功能如下图所示：



- **复位:** 其功能等同于硬件上按复位按钮。相当于实现了一次硬复位。代码重新执行。
- **执行到断点处:** 该按钮用来快速执行到断点处，有时候你并不需要观看每步是怎么执行的，而是想快速的执行到程序的某个地方看结果，这个按钮就可以实现这样的功能。
- **挂起:** 此按钮在程序一直执行的时候会变为有效，通过按该按钮，就可以使程序停下来，进入到单步调试状态。
- **执行一条指令:** 该按钮用来实现执行到某个函数里面去的功能，在没有函数的情况下，是等同于执行当前行执行过去按钮的。
- **执行当前行:** 在碰到有函数的地方，通过该按钮就可以单步执行过这个函数，而不进入这个函数单步执行。
- **跳出当前函数:** 该按钮是在进入了函数单步调试的时候，有时候你可能不必再执行该函数的剩余部分了，通过该按钮就直接一步执行完函数余下的部分，并跳出函数，回到函数被调用的位置。
- **执行到光标处:** 该按钮可以迅速的使程序运行到光标处，其实是挺像执行到断点处按钮功能，但是两者是有区别的，断点可以有多个，但是光标所在处只有一个。
- **汇编窗口:** 通过该按钮，就可以查看汇编代码，这对分析程序很有用。
- **观看变量窗口:** 该按钮按下，会弹出一个显示变量的窗口，在里面可以查看各种你想要看的变量值，也是很常用的一个调试窗口。
- **串口打印窗口:** 该按钮按下，会弹出一个串口调试助手界面的窗口，用来显示从串口打印出来的内容。

其他几个按钮用的比较少，以上是比较常用的，当然也不是每次都用得着这么多，具体看你程序调试的时候有没有必要观看这些东西，来决定要不要看。

关于如何一步一步进行仿真调试，请查看 MDK 相关资料。

2.9 在MDK开发环境中调试

单片机软件开发过程中，软件调试遇到的各种问题常令初学者感到不知所措。实际上。各种仿真开发软件的程序调试基本方法和技巧大同小异，掌握正确的程序调试基本技巧。对于排查这些程序错误问题可以起到举一反三、事半功倍的效果。软件调试是单片机技术人员必须掌握的重要基本技能。

在这一节里，我们将学习 STM32 在 MDK 下的软件仿真与利用 JLINK 对 STM32 进行在线调试。

2.9.1 KEIL仿真的应用

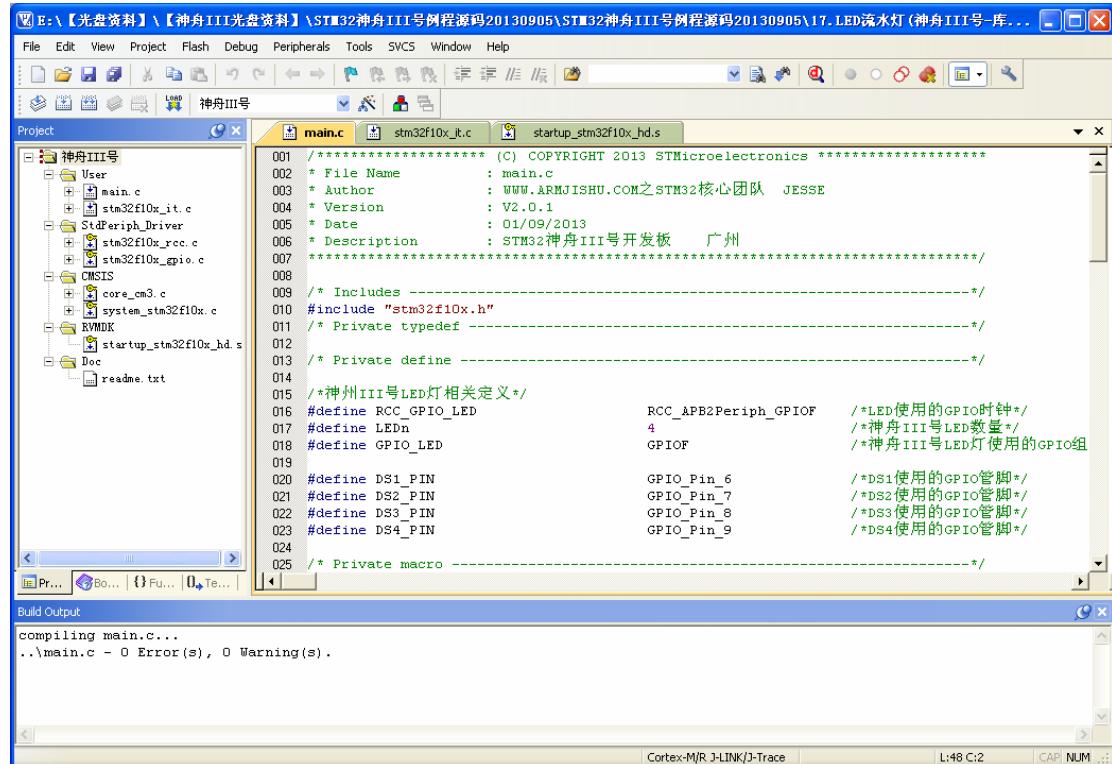
MDK 的一个强大的功能就是仿真功能，分为硬件仿真与软件仿真。通过软件仿真，我们可以发现很多将要出现的问题，避免了下载到 STM32 里面来查这些错误，这样最大的好处是能很方便的检查程序存在的问题，因为在 MDK 的仿真下面，你可以查看很多硬件相关的寄存器，通过观察这些寄存器，你可以知道代码是不是真正有效。而且软件仿真是不必频繁的刷机，从而延长了 STM32 的 FLASH 寿命（STM32 的 FLASH 寿命 $\geq 1W$ 次）。

而硬件仿真是使用附加的硬件来替代用户系统的单片机并完成单片机全部或大部分的功能。使用了附加硬件后用户就可以对程序的运行进行控制，例如单步，全速，查看资源断点等。硬件仿真是开发过程中所必须的。

下面我们通过实际的例子来详细来了解下软件仿真与硬件仿真的使用。

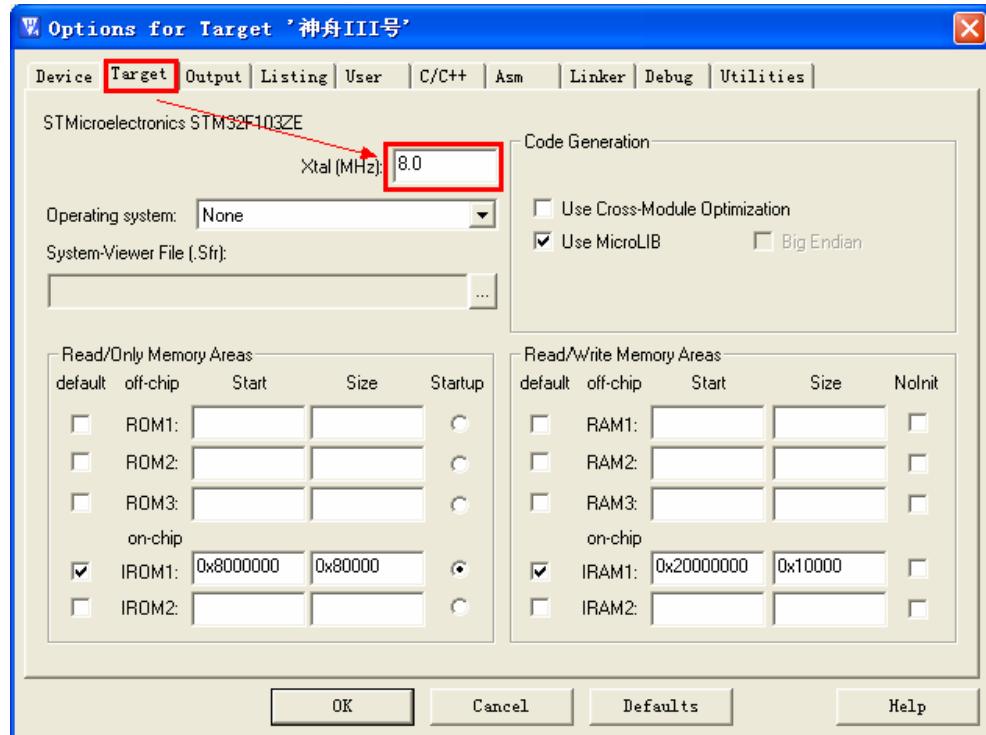
2.9.2 KEIL软件仿真

在做软件仿真的时候，我们先打开一个我们的工程，验证我们的工程是否是有问题的，在这里我们就使用神舟 III 号是最简单的 LED 流水灯的工程来做我们的例子说明。如下图为我们将打开工程后的页面

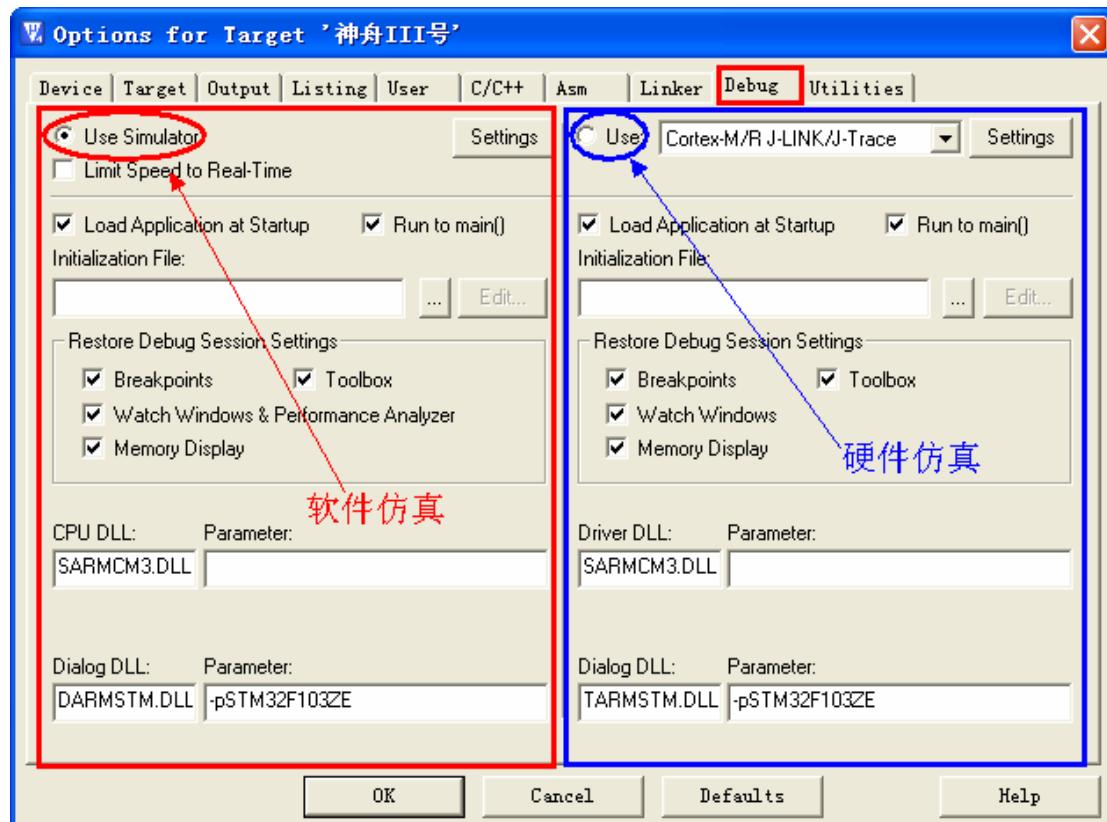


在开始软件仿真之前，先检查一下配置是不是正确，在project里面点击我们的工程选项“Options for Target ‘神舟III号’”或者是我们的快捷按钮 ，在“Device”和“Target”选项卡中先看下处理器与晶振是否是和我们的板子晶振相符的，神舟III号使用的处理器是STM32F103ZET，而外部

晶振是25MHz的，所以我们要设置为25MHz的，其他的默认即可，如下图所示：

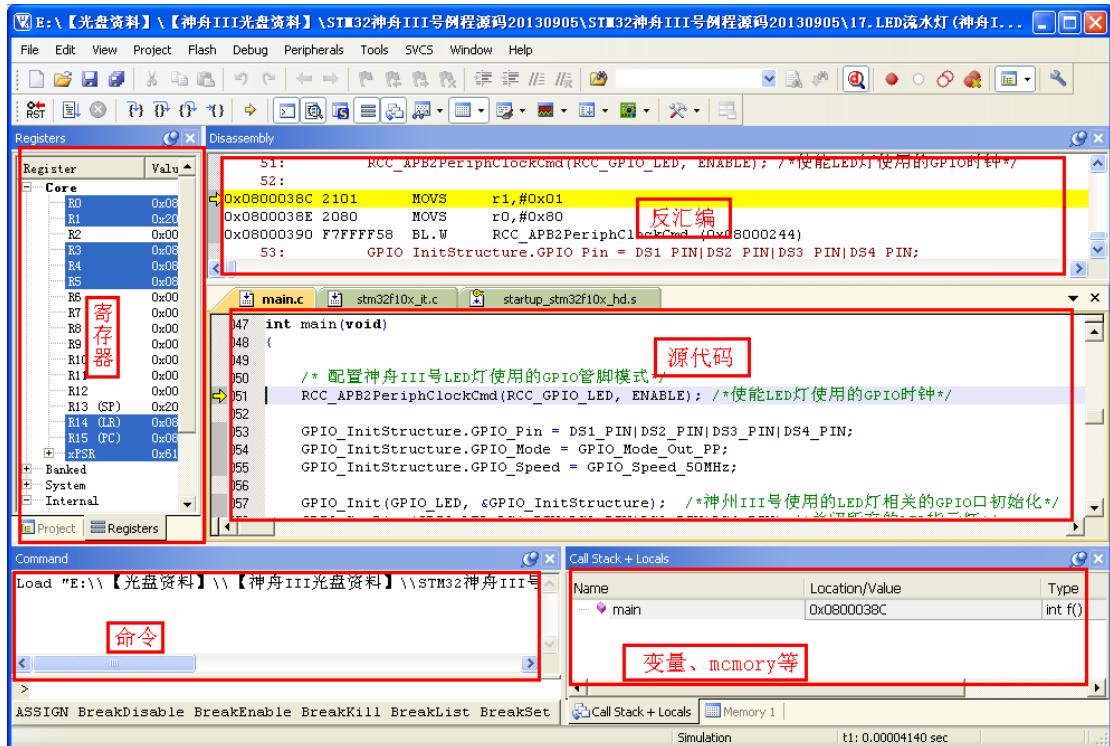


接下来，我们在这个配置选项中点击Debug选项卡，该选项卡是用来配置调试的，调试有两种，一种是基于软件的模拟调试，另一种是通过仿真器在线调试。我们这节主要就是使用这个选项卡。如下图所示，这里分左右两部分，左边为软件仿真，右边为硬件仿真，而我们这里要用的是软件仿真，所以选择左部分的“Use Simulator”模拟仿真。其他默认就行，点击OK退出Options for Target对话框。

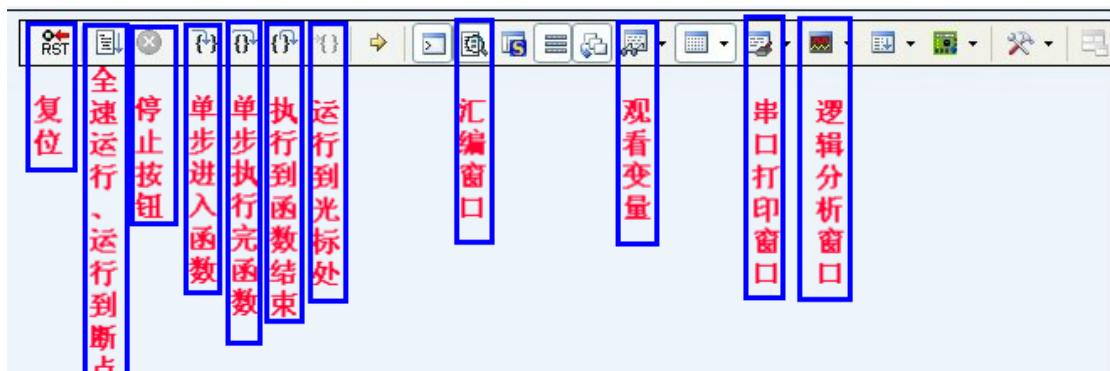


退出之后就能进入我们的软件模拟仿真的功能了，接下来，我们点击快捷按钮 (开始/停止) 嵌入式专业技术论坛 (www.armjishu.com) 出品 第 82 页，共 900 页

仿真按钮), 开始仿真, 出现如下图所示界面:



图中我们可以看到, 快捷按钮比我们仿真之前多了一个工具条, 这个就是我们Debug仿真的时候用到的工具条, 下面我们简单的介绍一下Debug工具条各个按钮的功能。如下图所示:



复位: 其功能等同于硬件上按复位按钮。相当于实现了一次硬复位。按下该按钮之后, 代码会重新从头开始执行。

全速运行、运行到断点: 该按钮用来快速执行到断点处, 有时候你并不需要观看每步是怎么执行的, 而是想快速的执行到程序的某个地方看结果, 这个按钮就可以实现这样的功能, 前提是你在查看的地方设置了断点。

停止: 此按钮在程序一直执行的时候会变为有效, 通过按该按钮, 就可以使程序停下来, 进入到单步调试状态。

单步进入函数: 该按钮用来实现执行到某个函数里面去的功能

单独执行完函数: 在碰到有函数的地方, 通过该按钮就可以单步执行过这个函数, 而不进入这个函数单步执行。

执行到函数结束: 该按钮是在进入了函数单步调试的时候, 有时候你可能不必再执行该函数的剩余部分了, 通过该按钮就直接一步执行完函数余下的部分, 并跳出函数, 回到函数被调用的位置。

运行到光标处: 该按钮可以迅速的使程序运行到光标处, 其实是挺像执行到断点处按钮功能, 但是两者是有区别的, 断点可以有多个, 但是光标所在处只有一个。

汇编窗口: 通过该按钮, 就可以查看汇编代码, 这对分析程序很有用。

观看变量/堆栈窗口: 该按钮按下，会弹出一个显示变量的窗口，在里面可以查看各种你想要看的变量值，也是很常用的一个调试窗口。

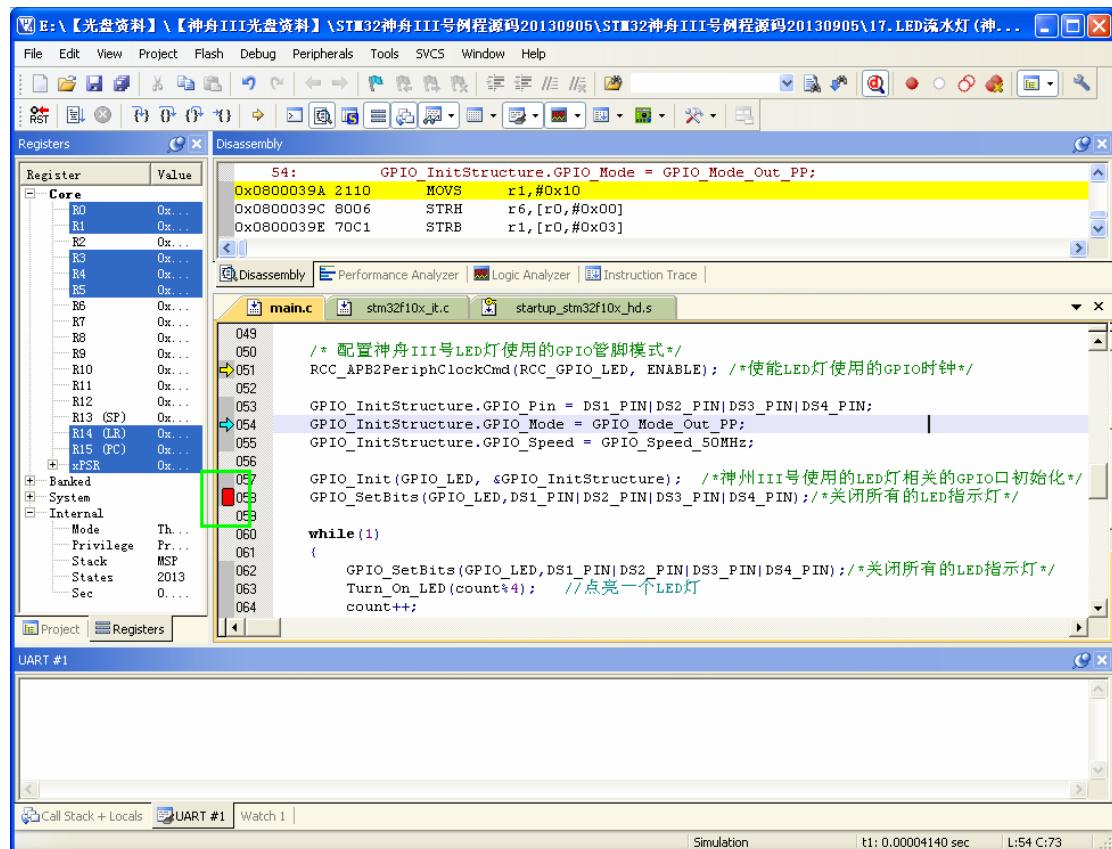
串口打印窗口: 该按钮按下，会弹出一个类似串口调试助手界面的窗口，用来显示从串口打印出来的内容。

逻辑分析窗口: 按下该按钮会弹出一个逻辑分析窗口，通过SETUP按钮新建一些IO口，就可以观察这些IO口的电平变化情况，以多种形式显示出来，比较直观。

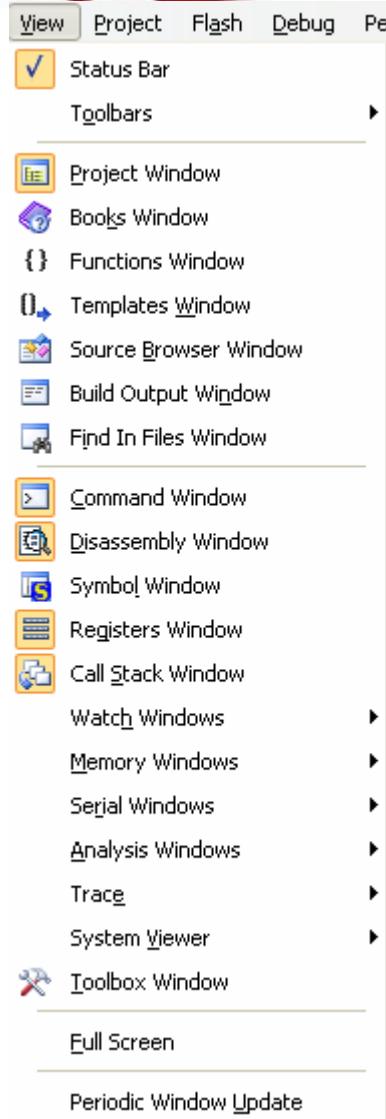
Debug工具条上的其他几个按钮用的比较少，我们这里就不介绍了。以上介绍的是比较常用的，当然也不是每次都用得着这么多，具体看你程序调试的时候有没有必要观看这些东西，来决定要不要看。

断点设置

在要设置断点的地方双击下鼠标左键即可，可以看到在左边会出现了一个红框，即表示设置了一个断点（也可以通过鼠标右键弹出菜单来加入），再次双击则取消）。然后我们点击 ，执行到该断点处，如下图所示：



调试的时候，可以看到对寄存器的修改，但是在实际操作的时候会需要修改某些外设控制寄存器或者 memory 中的数据，那么还需要其他的窗口来支持，这些窗口都在 view 菜单下，如下图：



这里重点介绍下面几个窗口： Command windows 命令窗口

Disassembly windows 反汇编窗口

Symbol windows 符号窗口

Register windows 寄存器窗口

Call Stack windows 函数调用栈显示窗口

Watch windows 变量观察窗口

Memory windows 存储器观察窗口

Serial windows 串行数据输出观察窗口

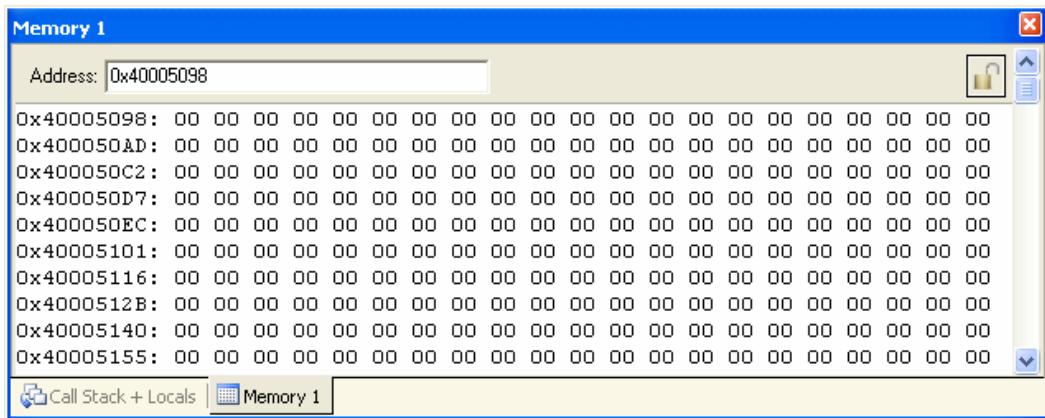
Analysis windows 分析窗口

Trace 轨迹跟踪观察窗口

这里有这么多的窗口，实际上比较常用的是 Watch 窗口和 Memory 窗口。如下图：

| Name | Location/Value | Type |
|----------|----------------|---------|
| ... main | 0x08000212 | int f() |

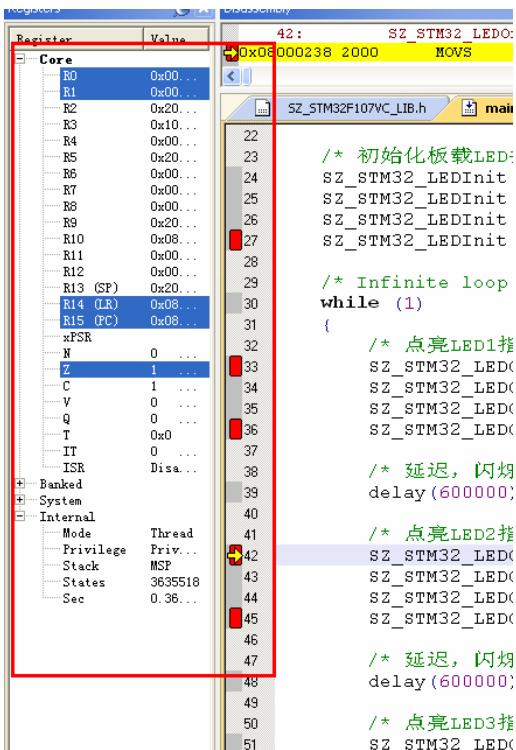
在这个 memory 观察窗口下，只要输入地址就可以看到从这个地址之后的若干个地址的数据。



当然观察窗口同样是支持修改对应的值的。

调试是对代码执行的验证过程，所以具体的调试方法每个人有不同的想法，而且实际的问题不同操作也会不同。

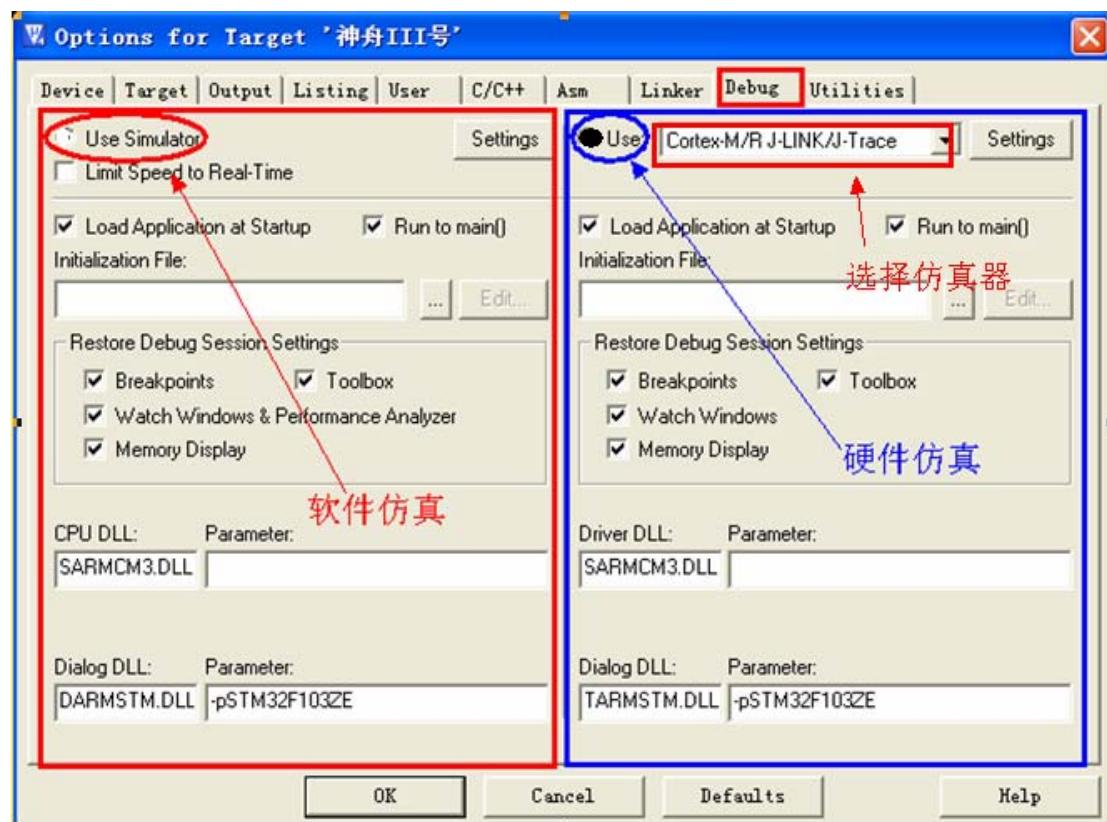
而我们在程序运行的时候，在左边的窗口可以看到寄存器的变化以及运行的时间，方便分析程序



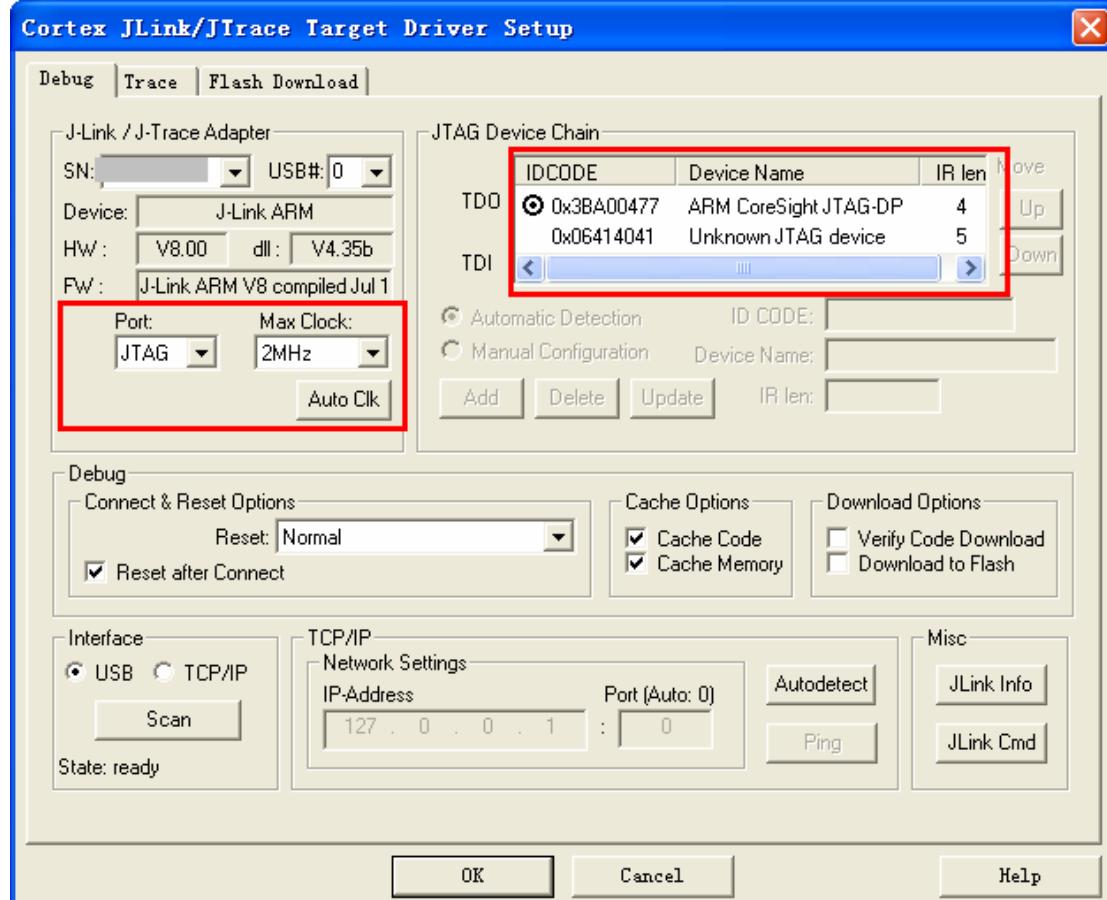
2.9.3 硬件仿真

硬件仿真的话需要用到我们的仿真器进行一个硬件仿真的操作，在这里我使用的是 J-Link 仿真器，打嵌入式专业技术论坛（www.armjishu.com）出品

开我们刚才的 Debug 选项，选择右边的硬件仿真，选择你的仿真器，我这里用的是 J-link 仿真器。



点击 Settings 进入我们的仿真器设置参数，如下图所示



这里我们能设置仿真模式为 JTAG 仿真还是 SW 模式仿真调试, JTAG 需要占用比 SW 模式多很多的 IO 口, 这里我们设置的调试速度为 2Mhz, 单击 OK, 完成此部分设置, 接下来我们还需要在 Utilities 选项卡里面设置下载时的目标编程器, 如图



我们选择 J-LINK 来调试 Cortex M3, 然后点击 Settings, 设置如图



这里要根据不同的MCU选择FLASH的大小，因为我们开发板使用的是STM32F103ZET，其FLASH大小为512KB，所以我们点击Add，并在Programming Algorithm里面选择512K型号的STM32。然后选中Reset and Run选项，以实现在编程后自动启动，其他默认设置即可。在设置完之后，点击OK，然后

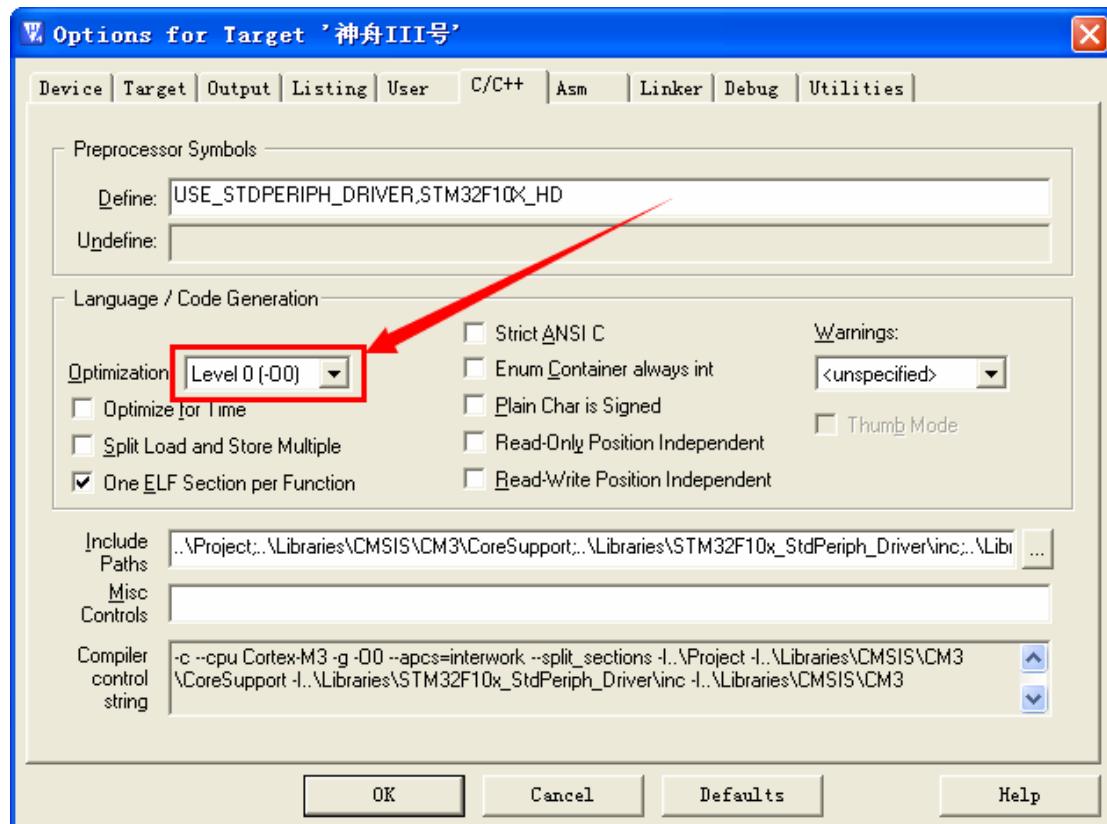
再点击OK，回到IDE界面，编译一下工程。再点击，开始仿真（如果开发板的代码没被更新过，则会先更新代码，再仿真，你也可以通过按，只下载代码，而不进入仿真）。

其他的则和软件仿真差不多了，只不过硬件仿真能在设备上运行我们执行程序的效果，方便我们的调试。

2.9.4 DEBUG模式下不能watch的解决办法

感谢网友王里十一提供解决办法，花了不少时间：

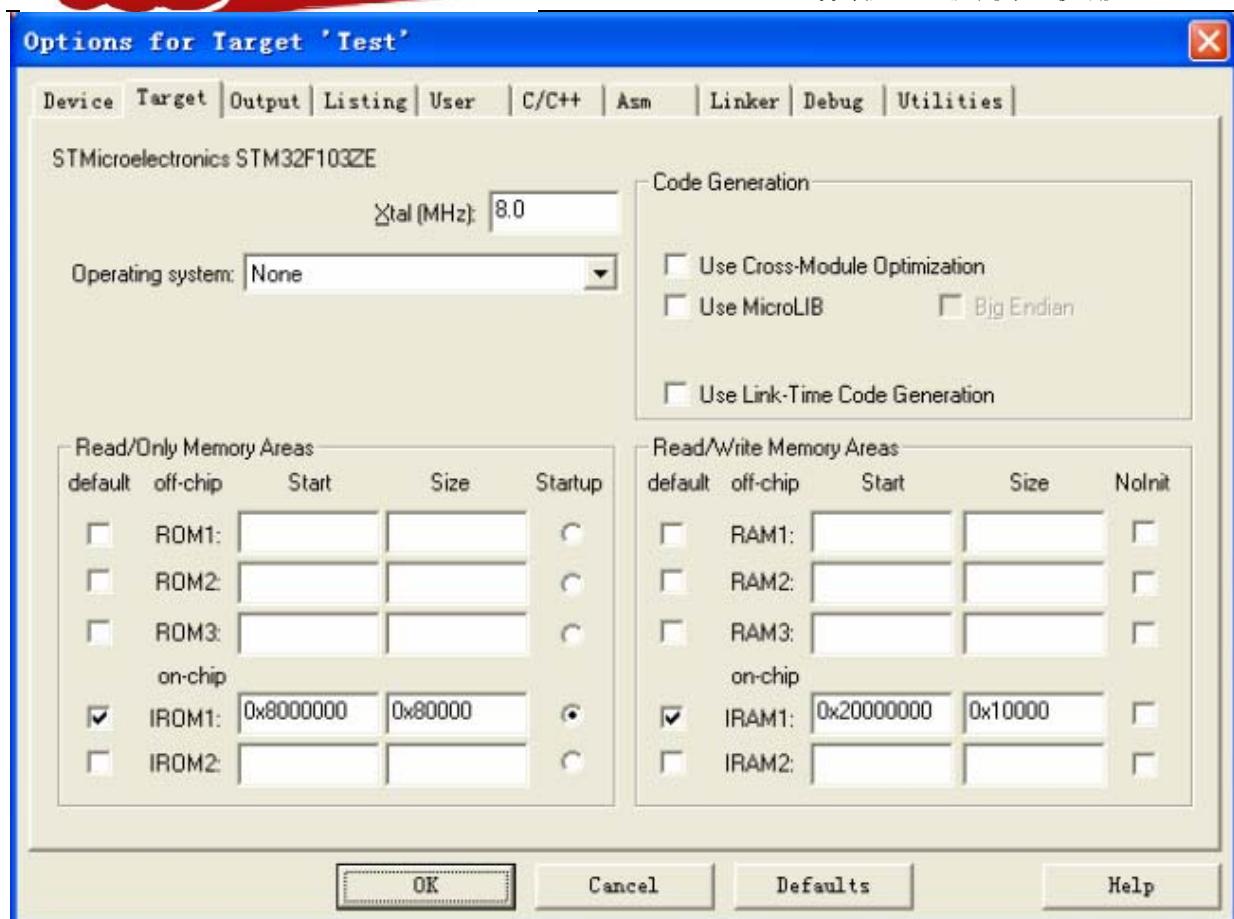
打开该窗口，尝试设置此选项里的 level0 或 level1、level2、level3，再尝试 watch 参数的设置，来检测 watch 变量：



2.10 如何设置程序空间在CPU内部Flash，变量空间在CPU内部RAM运行

将程序空间定位在 CPU 内部 Flash，变量空间定位在 CPU 内部 RAM。这是最通用的设置，一般最终发布的软件就是按照这种设置。

关键设置界面为：



IROM1 设置为 0x80000000, IRAM1 设置为 0x20000000。.

该设置的程序可以用于下载到 Flash，脱离仿真器运行。也可以直接使用仿真器进行调试跟踪

2.11 如何设置程序空间和变量空间都在CPU内部RAM运行

该设置的程序只能使用仿真器进行调试。

优点：

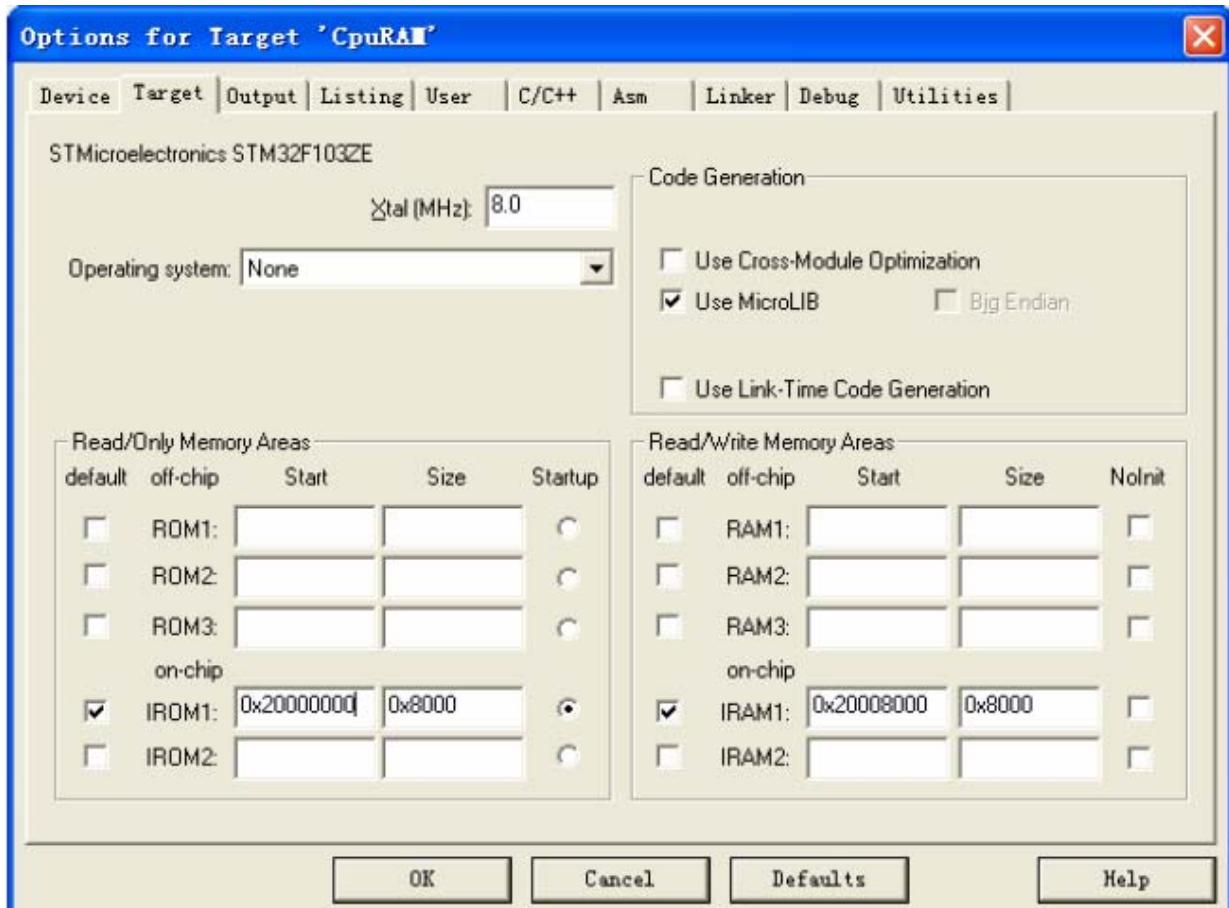
- 1) 下载速度快；
- 2) 不改写 CPU 内部 Flash 已有的程序；
- 3) 无需拨动启动模式选择开关（即拨打 CPU 内部 Flash 启动也可以下载到 RAM 进行调试）；
- 4) 程序执行速度和在 CPU 内部 Flash 一样快。

缺点：

- 1) 开发板掉电会丢失程序；
- 2) 暂时无法使用调试界面的复位按钮进行复位。
- 3) 程序空间最大 32K 字节，变量空间最大限制在 32K 字节。（用户也可以自行调整）

内部 Flash 有效。请直接点击按钮  启动调试即可。IDE 会自动将程序装入 CPU 内部 RAM。

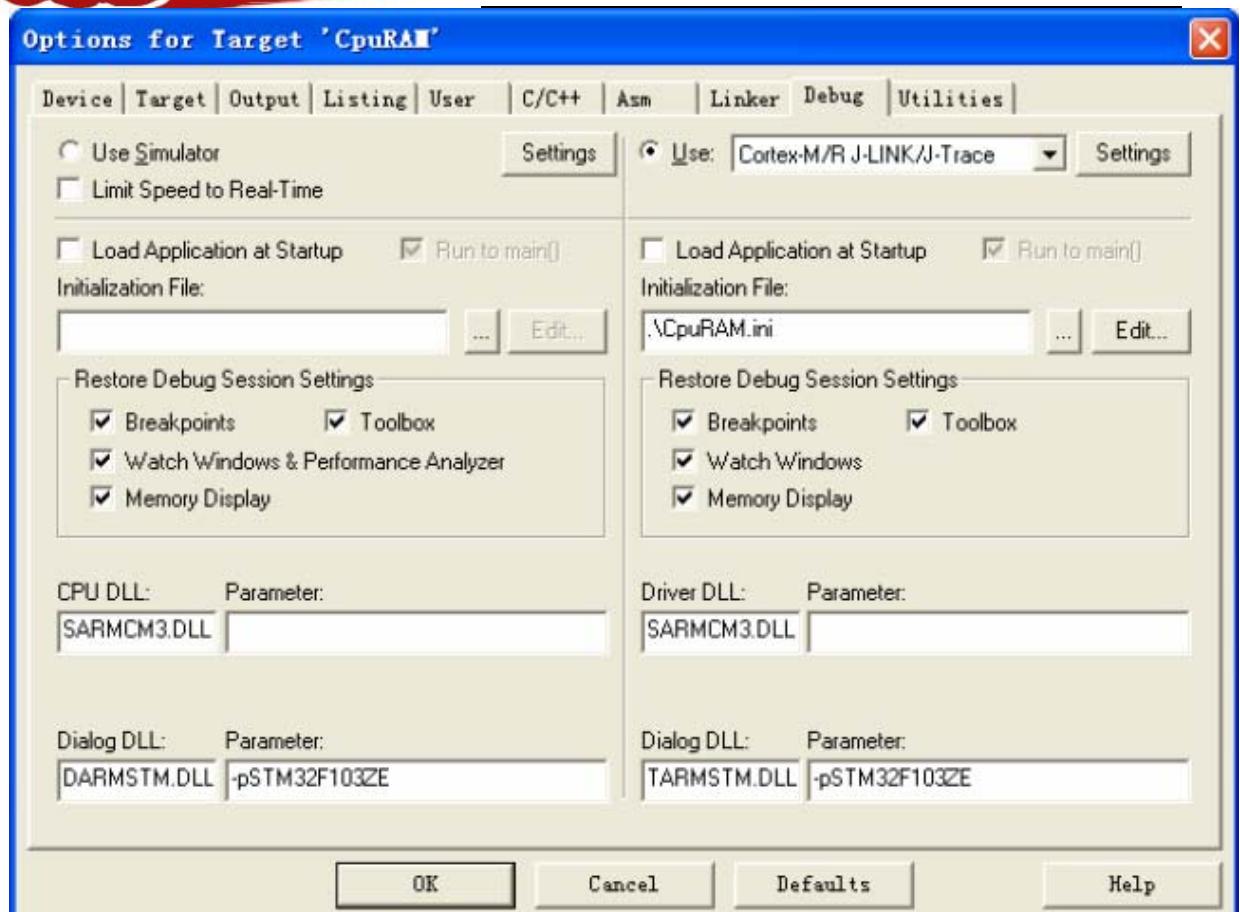
首先设置 Target 定位地址，设置界面如下：



IROM1 = 0x20000000, Size = 0x8000, 这是 CPU 内部 RAM 区的前 32K 字节空间；

IRAM1 = 0x20008000, Size = 0x8000, 这是 CPU 内部 RAM 区的后 32K 字节空间。

然后设置 Debug 调试接口，界面如下：



- 取消“Load Application at Startup”前面的钩。在 CpuRAM.ini 初始化脚本中自动装入程序。
- 在“Initialization Files:”“编辑框指定”.\CpuRAM.ini “。 \ 表示工程文件所在的当前目录。
- 请将 CpuRAM.ini 文件和工程文件放在同一个文件夹下。CpuRAM.ini 文件是一个文本格式的初始化脚本文件。当启动调试时，IDE 会执行这个脚本中的命令。

初始化脚本文件说明：

- 新建一个 CpuRAM.ini 的空文件，然后使用记事本将如下内容复制到这文件保存即可。
- CpuRAM.ini 的内容如下：

本脚本完成的功能是

- (1) 装载目标程序到 CPU 内部 RAM
- (2) 设置堆栈指针 SP
- (3) 修改 PC 指针

脚本的语法：

```
FUNC void Setup (void)
```

```
{
```

```
SP = _RDWORD(0x20000000) + 4; // 设置堆栈指针
```

```
PC = _RDWORD(0x20000004); // 设置 PC 指针
```

```
}
```

```
LOAD obj\output.axf INCREMENTAL // 先装载代码到 CPU 内部 RAM
```

```
Setup(); // 再调用 Setup 函数修改堆栈和 PC 指针
```

```
g, main // 运行到 main()函数
```

IDE 控制调试器将程序装入 CPU 内部 RAM 后，0x20000000 单元存储了程序的堆栈初值（堆栈区域的最大值+4，向下增长）。0x20000004 单元存储了复位向量地址。

2.12 如何设置程序空间和变量空间都在外部SRAM运行

该设置的程序只能使用仿真器进行调试。

优点：

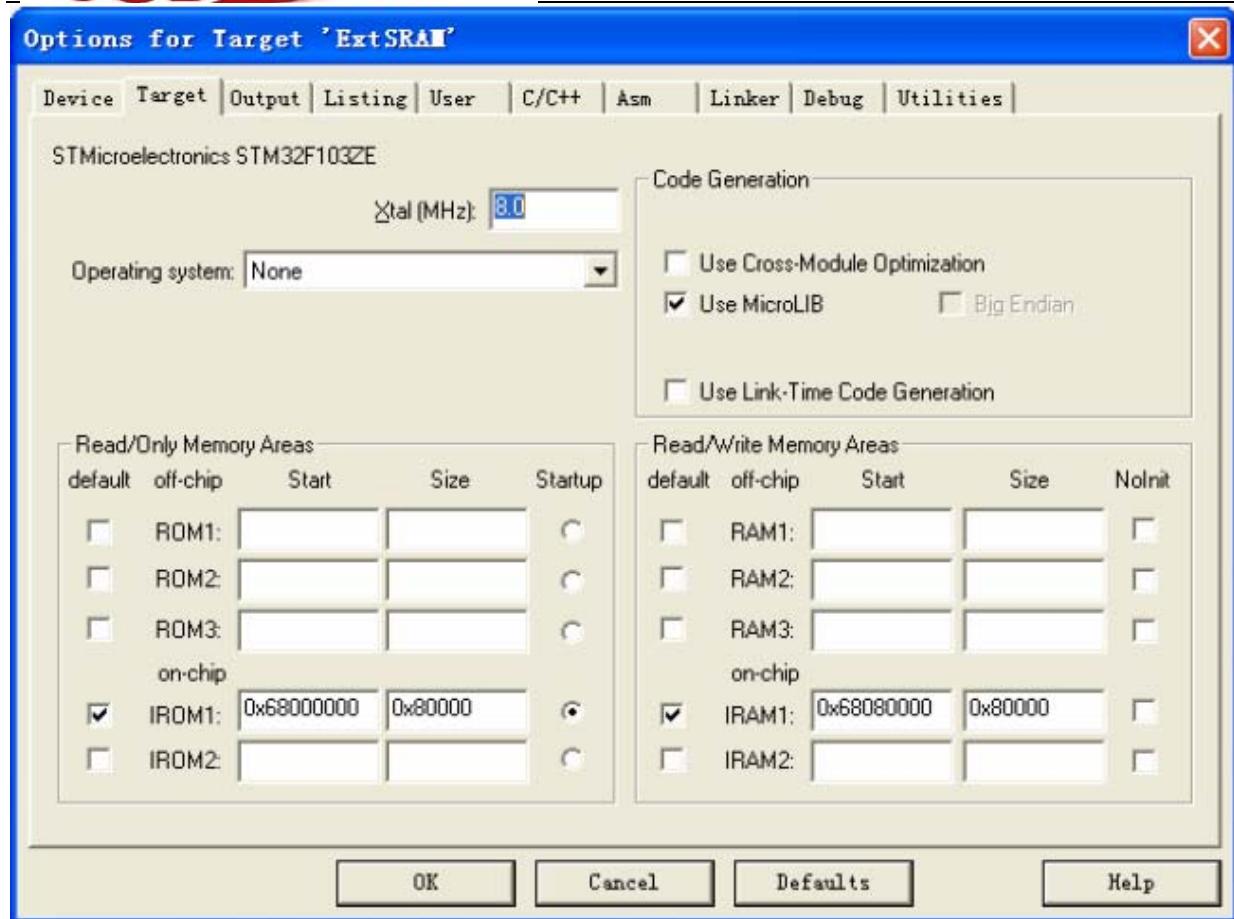
- 1) 下载速度快；
- 2) 不改写 CPU 内部 Flash 已有的程序；
- 3) 无需拨动启动模式选择开关(即拨打 CPU 内部 Flash 启动也可以下载到外部 RAM 进行调试)。

缺点：

- 1) 开发板掉电会丢失程序；
- 2) 无法使用调试界面的复位按钮进行复位。
- 3) 空间最大 512K 字节，变量空间最大 512K 字节（用户也可以自行调整）；
- 4) 程序执行速度比 CPU 内部 Flash 慢很多。

要说明：编译、连接完毕后，不要使用  按钮下载程序。因为这个按钮只针对下载程序到 CPU 内部 Flash 有效。请直接点击  按钮启动调试即可。IDE 会自动将程序装入 CPU 外部 RAM。

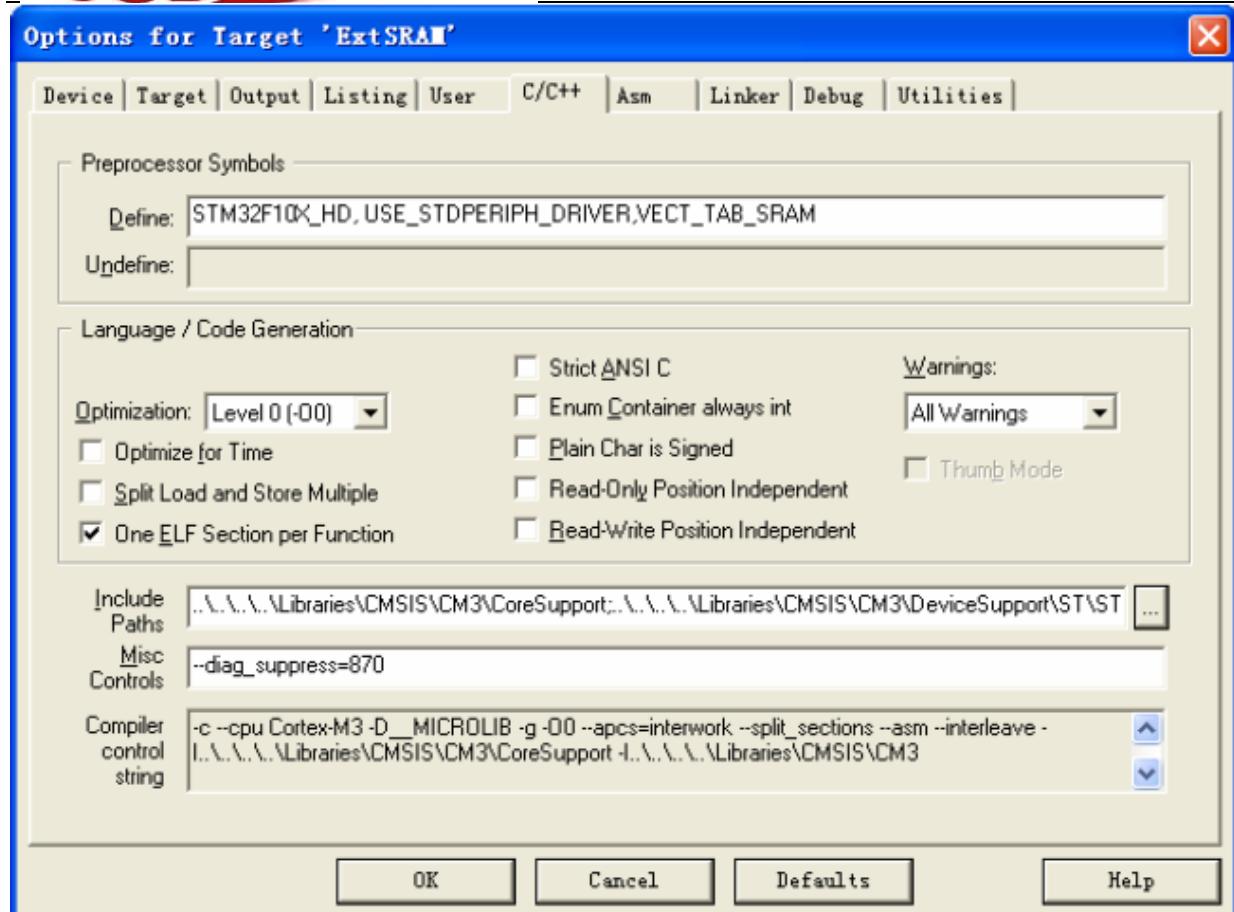
首先设置 Target 定位地址，设置界面如下：



IROM1 = 0x68000000, Size = 0x80000, 这是 CPU 外部 RAM 区的前 512K 字节空间;

IRAM1 = 0x68080000, Size = 0x80000, 这是 CPU 外部 RAM 区的后 512K 字节空间。

然后设置 C/C++ 编译器编译选项，界面如下：



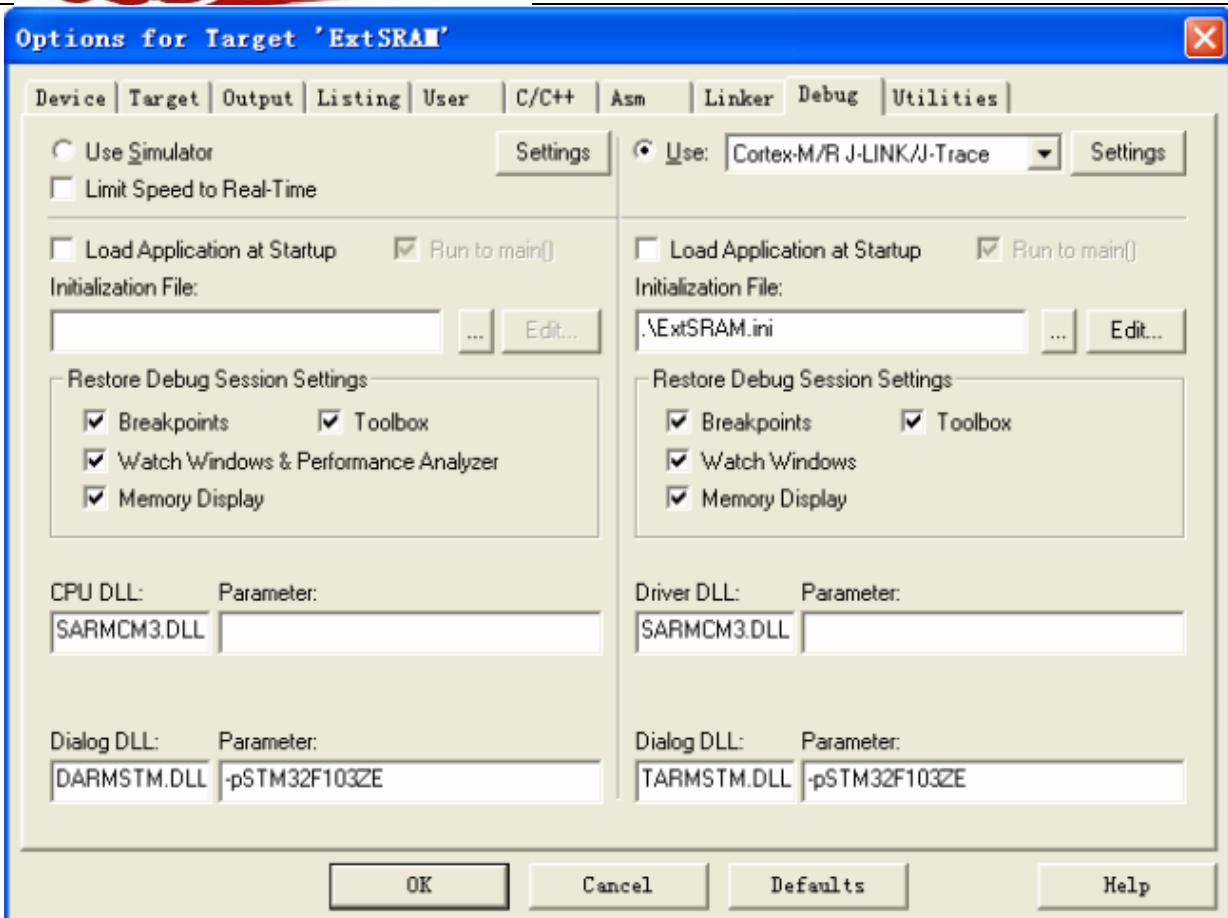
由于 STM32 这款 CPU 只能将中断向量表定位在 CPU 内部 Flash 或者 CPU 内部 RAM，不支持将中断向量表定位在 CPU 外部存储器（外部 SRAM 或外部 NOR Flash）。因此，我们需要通过调试脚本将外部 RAM 的中断向量表复制到 CPU 内部 RAM。

“Define”编辑框需要添加宏“VECT_TAB_SRAM”，这个宏表示中断向量表定位在 CPU 内部 RAM。

ST 固件库提供的 system_stm32f10x.c 文件中，void SystemInit (void) 函数会改写中断向量表地址寄存器 SCB->VTOR。函数中的关键代码如下：

```
#ifdef VECT_TAB_SRAM
SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM. */
#else
SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH. */
#endif
```

最后设置“Debug”选项：



- a) 取消“Load Application at Startup”前面的钩。在 CpuRAM.ini 初始化脚本中自动装入程序。
- b) 在“Initialization Files:”“编辑框指定”.\ExtSRAM.ini “。.\ 表示工程文件所在的当前目录。
- c) 请将 ExtSRAM.ini 文件和工程文件放在同一个文件夹下。ExtSRAM.ini 文件是一个文本格式的初始化脚本文件。当启动调试时，IDE 会执行这个脚本中的命令。

初始化脚本文件说明：

a) 新建一个 ExtSRAM.ini 的空文件，然后使用记事本将如下内容复制到这文件保存即可。

b) ExtSRAM.ini 的内容如下：

```
/*
```

本脚本完成的任务是：

- (1) 配置 CPU 的 FSMC 总线，使 CPU 能够访问外部 SRAM，已便于后面装入程序
- (2) 复制外部 SRAM 的中断向量表（0x130 字节）到 CPU 内部 RAM
- (3) 设置堆栈指针 SP
- (4) 修改 PC 指针

注意：工程选项 IRAM1 的起始地址必须是 0x20000200 开始

脚本的语法：

参见 MDK 的 HELP，搜索关键字“uv3 Library Routines”可以看到 uv3 支持的脚本命令

```
FUNC void Setup (void) {
    SP = _RDWORD(0x68000000); // 设置堆栈指针
    PC = _RDWORD(0x68000004); // 设置 PC 指针 (程序计数器)
    _WDWORD(0xE000ED08, 0x20000000); // 设置中断向量表地址寄存器 = 0x20000000
}

// 初始化 FSMC，用于外部 SRAM
FUNC void InitSRAM (void) {
    //InitRCCC
    _WDWORD(0x40021000, 0x00005083);
    _WDWORD(0x40021004, 0x00000000);
    _WDWORD(0x40021004, 0x00000000);
    _WDWORD(0x40021000, 0x00005083);
    _WDWORD(0x40021004, 0x00000000);
    _WDWORD(0x40021008, 0x009F0000);

    _WDWORD(0x40021000, 0x00015083);
    _sleep_(100); // Wait for PLL lock
    _WDWORD(0x40022000, 0x00000030);
    _WDWORD(0x40022000, 0x00000030);
    _WDWORD(0x40022000, 0x00000032);
    _WDWORD(0x40021004, 0x00000000);
    _WDWORD(0x40021004, 0x00000000);
    _WDWORD(0x40021004, 0x00000400);
    _WDWORD(0x40021004, 0x00000400);
    _WDWORD(0x40021004, 0x001D0400);
    _WDWORD(0x40021000, 0x01035083);
    _sleep_(100);
    _WDWORD(0x40021004, 0x001D0400);
    _WDWORD(0x40021004, 0x001D0402);
    _sleep_(100);

    _WDWORD(0x40021014, 0x00000114); /* Enable AHBPeriphClock */
    _WDWORD(0x40021018, 0x000001E0); /* Enable APB2PeriphClock */
    /* GPIO Configuration for FSMC */
    _WDWORD(0x40011400, 0xB8BB44BB);
```

```
_WDWORD(0x40011404, 0xBCCCCCCC);
 _WDWORD(0x40011800, 0xBCCCCC4BB); /* NBL0, NBL1 & address configuration */
 _WDWORD(0x40011804, 0xBCCCCCCC);
 _WDWORD(0x40011C00, 0x33CCCCCCCC);
 _WDWORD(0x40011C04, 0xBBBB3333);
 _WDWORD(0x40012000, 0x48CCCCCCCC);
 _WDWORD(0x40012004, 0x444B4BB4); /* NE3 configuration */
 _WDWORD(0xA0000010, 0x00001010); /* FSMC Configuration */
 _WDWORD(0xA0000014, 0x00000200); /* FSMC_DataSetupTime = 2; */
 _WDWORD(0xA0000010, 0x00001011); /* Enable FSMC Bank1_SRAM Bank */
 _sleep_(200);
}
```

/*
复制中断向量表

中断向量表地址必须是 512 字节的整数倍。

中断向量表实际大小 : 0x00000130

*/

```
FUNC void CopyVectTable(void) {
 _WDWORD(0x20000000, _RDWORD(0x68000000));
 _WDWORD(0x20000004, _RDWORD(0x68000004));
 _WDWORD(0x20000008, _RDWORD(0x68000008));
 _WDWORD(0x2000000C, _RDWORD(0x6800000C));

 _WDWORD(0x20000010, _RDWORD(0x68000010));
 _WDWORD(0x20000014, _RDWORD(0x68000014));
 _WDWORD(0x20000018, _RDWORD(0x68000018));
 _WDWORD(0x2000001C, _RDWORD(0x6800001C));
 _WDWORD(0x20000020, _RDWORD(0x68000020));
 _WDWORD(0x20000024, _RDWORD(0x68000024));
 _WDWORD(0x20000028, _RDWORD(0x68000028));
 _WDWORD(0x2000002C, _RDWORD(0x6800002C));
 _WDWORD(0x20000030, _RDWORD(0x68000030));
 _WDWORD(0x20000034, _RDWORD(0x68000034));
 _WDWORD(0x20000038, _RDWORD(0x68000038));
```

_WDWORD(0x2000003C, _RDWORD(0x6800003C));

```
_WORD(0x20000040, _WORD(0x68000040));
_WORD(0x20000044, _WORD(0x68000044));
_WORD(0x20000048, _WORD(0x68000048));
_WORD(0x2000004C, _WORD(0x6800004C));
_WORD(0x20000050, _WORD(0x68000050));
_WORD(0x20000054, _WORD(0x68000054));
_WORD(0x20000058, _WORD(0x68000058));
_WORD(0x2000005C, _WORD(0x6800005C));
_WORD(0x20000060, _WORD(0x68000060));
_WORD(0x20000064, _WORD(0x68000064));
_WORD(0x20000068, _WORD(0x68000068));
_WORD(0x2000006C, _WORD(0x6800006C));
_WORD(0x20000070, _WORD(0x68000070));
_WORD(0x20000074, _WORD(0x68000074));
_WORD(0x20000078, _WORD(0x68000078));
_WORD(0x2000007C, _WORD(0x6800007C));
_WORD(0x20000080, _WORD(0x68000080));
_WORD(0x20000084, _WORD(0x68000084));
_WORD(0x20000088, _WORD(0x68000088));
_WORD(0x2000008C, _WORD(0x6800008C));
_WORD(0x20000090, _WORD(0x68000090));
_WORD(0x20000094, _WORD(0x68000094));
_WORD(0x20000098, _WORD(0x68000098));
_WORD(0x2000009C, _WORD(0x6800009C));
_WORD(0x200000A0, _WORD(0x680000A0));
_WORD(0x200000A4, _WORD(0x680000A4));
_WORD(0x200000A8, _WORD(0x680000A8));
_WORD(0x200000AC, _WORD(0x680000AC));

_WORD(0x200000B0, _WORD(0x680000B0));
_WORD(0x200000B4, _WORD(0x680000B4));
_WORD(0x200000B8, _WORD(0x680000B8));
_WORD(0x200000BC, _WORD(0x680000BC));
_WORD(0x200000C0, _WORD(0x680000C0));
_WORD(0x200000C4, _WORD(0x680000C4));
```

```
_WDWORD(0x200000C8, _RDWORD(0x680000C8));
_WDWORD(0x200000CC, _RDWORD(0x680000CC));
_WDWORD(0x200000D0, _RDWORD(0x680000D0));
_WDWORD(0x200000D4, _RDWORD(0x680000D4));
_WDWORD(0x200000D8, _RDWORD(0x680000D8));
_WDWORD(0x200000DC, _RDWORD(0x680000DC));
_WDWORD(0x200000E0, _RDWORD(0x680000E0));
_WDWORD(0x200000E4, _RDWORD(0x680000E4));
_WDWORD(0x200000E8, _RDWORD(0x680000E8));
_WDWORD(0x200000EC, _RDWORD(0x680000EC));
_WDWORD(0x200000F0, _RDWORD(0x680000F0));
_WDWORD(0x200000F4, _RDWORD(0x680000F4));
_WDWORD(0x200000F8, _RDWORD(0x680000F8));
_WDWORD(0x200000FC, _RDWORD(0x680000FC));
_WDWORD(0x20000100, _RDWORD(0x68000100));
_WDWORD(0x20000104, _RDWORD(0x68000104));
_WDWORD(0x20000108, _RDWORD(0x68000108));
_WDWORD(0x2000010C, _RDWORD(0x6800010C));
_WDWORD(0x20000110, _RDWORD(0x68000110));
_WDWORD(0x20000114, _RDWORD(0x68000114));
_WDWORD(0x20000118, _RDWORD(0x68000118));
_WDWORD(0x2000011C, _RDWORD(0x6800011C));

_WDWORD(0x20000120, _RDWORD(0x68000120));
_WDWORD(0x20000124, _RDWORD(0x68000124));
_WDWORD(0x20000128, _RDWORD(0x68000128));
_WDWORD(0x2000012C, _RDWORD(0x6800012C));
}

// 从这里开始执行代码，之前的都是函数定义

InitSRAM(); // 配置 FSMC 用于 SRAM
LOAD obj\output.axf INCREMENTAL // 下载程序到外部 SRAM
CopyVectTable(); // 将外部 SRAM 的中断向量表复制到 CPU 内部 RAM
Setup(); // 配置堆栈和 PC 指针
g, main // 运行到 main() 函数后暂停
```

由于 STM32 的 FSMC 总线缺省是不使能的，外部 SRAM 无法由仿真器直接读写，因此我们需要先配置

FSMC 总线以保证 SRAM 可以通过 CPU 直接访问。

IDE 控制调试器将程序装入 CPU 外部 SRAM 后，0x68000000 单元存储了程序的堆栈初值（堆栈区域的最大值+4，向下增长）。0x68000004 单元存储了复位向量地址。

第三章 STM32神舟III号开发板硬件使用基础篇

本章节主要介绍使用 STM32 神舟 III 号开发板进行学习与试验的基本操作，让您快速入门。

开发板使用第一步是供电，所以我们先介绍如何给神舟 STM32 开发板供电；上电后当然是先下载几个示例观察一下实验现象与效果，所以紧接着我们介绍如何使用如何通过高效 ARM 调试仿真下载工具 Jlink V8 下载一个编译好的固件到开发板；当然不是每个用户手头都有 Jlink 工具，我们同时提供其它几种下载方式包括 USB、串口等方式下载固件到开发板等内容。最后我们介绍神舟开发板上跳帽的设置方式。

3.1 如何给神舟III号板供电

神舟 III 号 STM32 开发板一共支持三种供电方式，分别是：

- 使用 USB 供电
- 使用外接电源供电
- 使用 JLINK V8 供电

3.1.1 使用USB供电

神舟 III 号支持 USB 供电方式，板上自带一个 500mA 自恢复保险丝，当电流大于 500mA 时，自恢复保险丝起作用，防止神舟 III 号过大的工作电流损坏 PC 机 USB 接口和神舟 III 号开发板。

使用 USB 供电时，请使用随神舟 III 号配置的 USB 电缆连接开发板的 USB 接口 (J6) 和 PC 机的 USB 接口，将电源开关 (CON1) 拨到右边，选择 USB 供电方式。在正常情况下，电源转换芯片附近的 LED 灯 (DS6)，将变亮，表示神舟 III 号已经正常供电。

3.1.2 使用外接电源供电

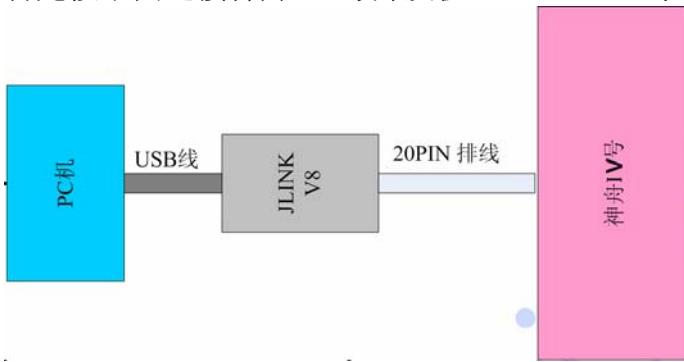
神舟 III 号也支持使用外接电源供电，通过电源开关 CON1 选择使用 USB 供电还是外接电源供电。当电源开关拨向左边时，选择外接电源供电，反之，当电源开关拨向右边时，选择 USB 供电。

神舟 III 号，外接电源电压范围为 4.75V~12V，如使用外接电源供电，建议使用 5V 电压，1A 电流输出的电源适配器进行供电，注意电源适配器必须内芯为电源的正极，外侧为电源的负极。如正负极与之相反，可能损坏神舟 III 号开发板。

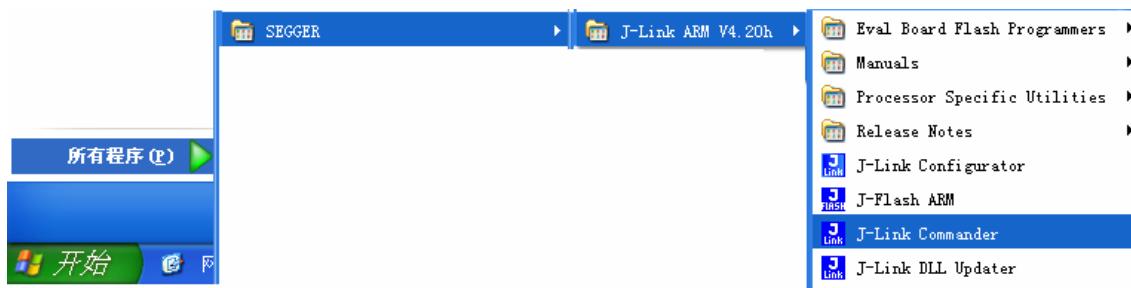
3.1.3 使用JLINK V8供电

除了前面提到的两种供电方式外，神舟 III 号还支持 JLINK V8 供电，以下以 ARMJISHU.COM 推出的 JLINK V8 为例，说明如何使用 JLINK V8 为神舟 III 号供电。

首先按下图连接神舟 III 号开发板，JLINK V8 与 PC 机。



在 PC 机上打开 J-LINK Commander (注：使用 JLINK V8 时，电脑上要求安装相应的驱动，具体驱动安装说明请见 JLINK V8 用户手册说明文档)。



在弹出的界面中，输入 power On 即可使能 JLINK V8 给 USB 供电。

```

J-Link Commander
SEGGER J-Link Commander V4.20h ('?' for help)
Compiled Oct 5 2010 19:11:57
DLL version V4.20h, compiled Oct 5 2010 19:11:41
Firmware: J-Link ARM V8 compiled Oct 5 2010 08:59:59
Hardware: V8.00
S/N: 1234567890
Feature(s): RDI,FlashDL,FlashBP,JFlash,GDBFull
J-Link>power On
J-Link>
  
```

说明：请勿同时使用这三种供电方式进行供电，以免多组电源同时供电，损坏神舟 III 号 STM32 开发板。

3.2 如何使用JLINK软件下载固件到神舟开发板

关于JLINK软件的安装步骤请参阅2.7.3如何安装JLINK软件章节。

3.2.1 如何使用J-FLASH ARM 烧写固件到芯片FLASH

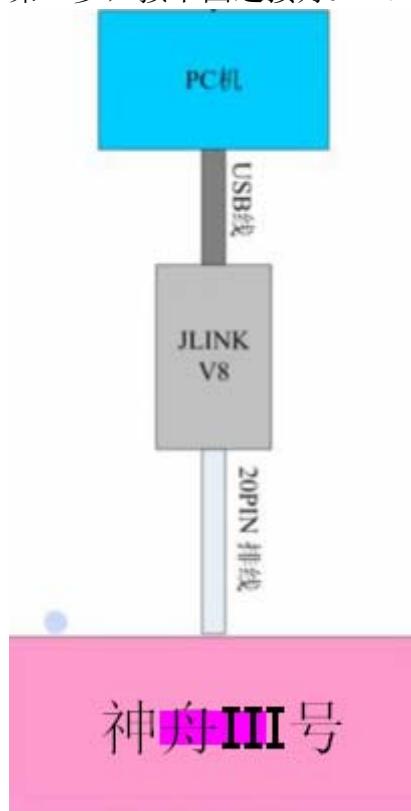
STM32的程序下载有多种方法，可以通过USB、串口、JTAG、SWD等方式下载。这几种方式都可以用来给神舟III号开发板下载程序，这里详细介绍通过JLINK 仿真器下载固件到神舟III号开发板的过程。

JLINK V8是目前主流的JTAG仿真器，支持所有的ARM7/9/11和Cortex-M0/M1/M3处理器。而且与主流的开发环境，如神舟系列STM32开发板采用的IAR,MDK开发环境完美的结合。通过JLINK仿真器，我们可以方便的下载，和在线调试代码。因此，推荐神舟III号开发板与JLINK V8搭配使用。

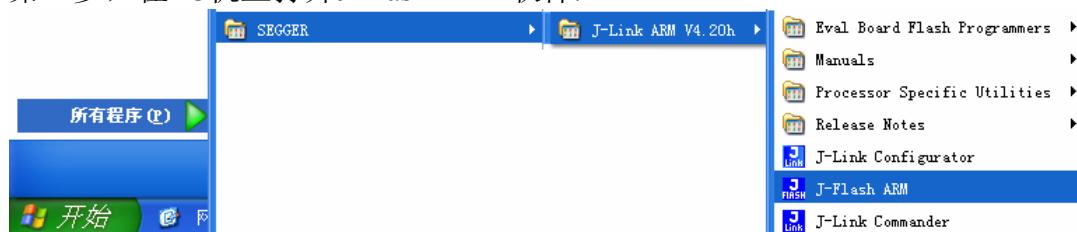
神舟III号提供的官方资源，包括源代码，使用的开发环境为MDK 4.12；使用的仿真调试工具为JLINK V8。

以下详细描述使用JLINK V8下载固件到神舟III号开发板的过程。

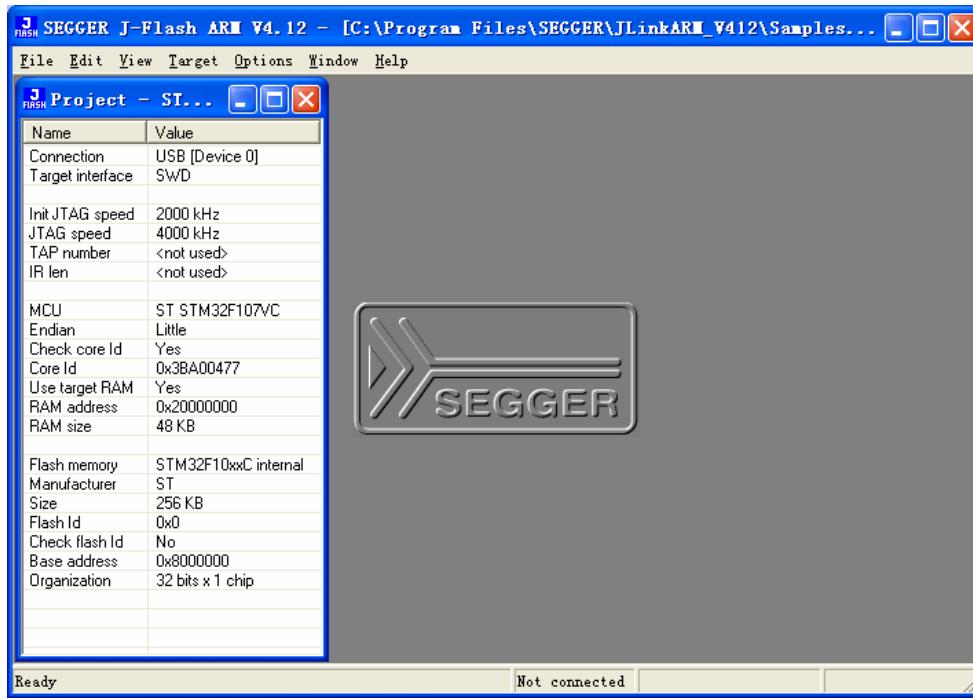
第一步，按下图连接好JLINK V8，神舟III号与PC机。



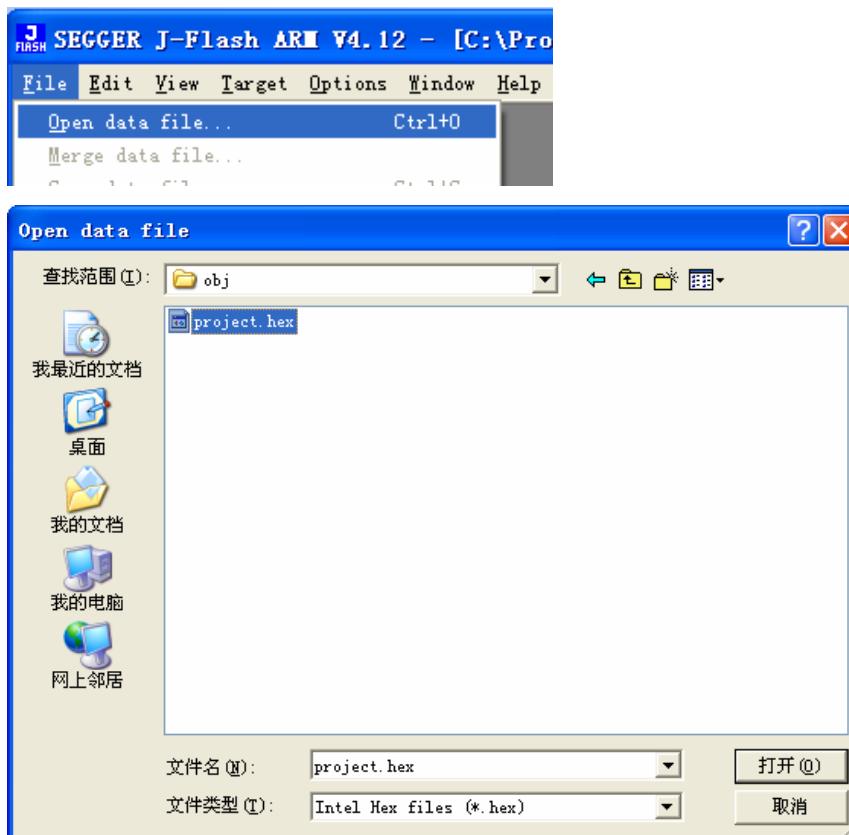
第二步，在PC机上打开J-Flash ARM软件。



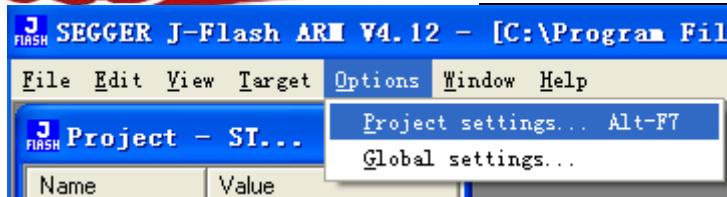
运行 J-Flash ARM，界面如下



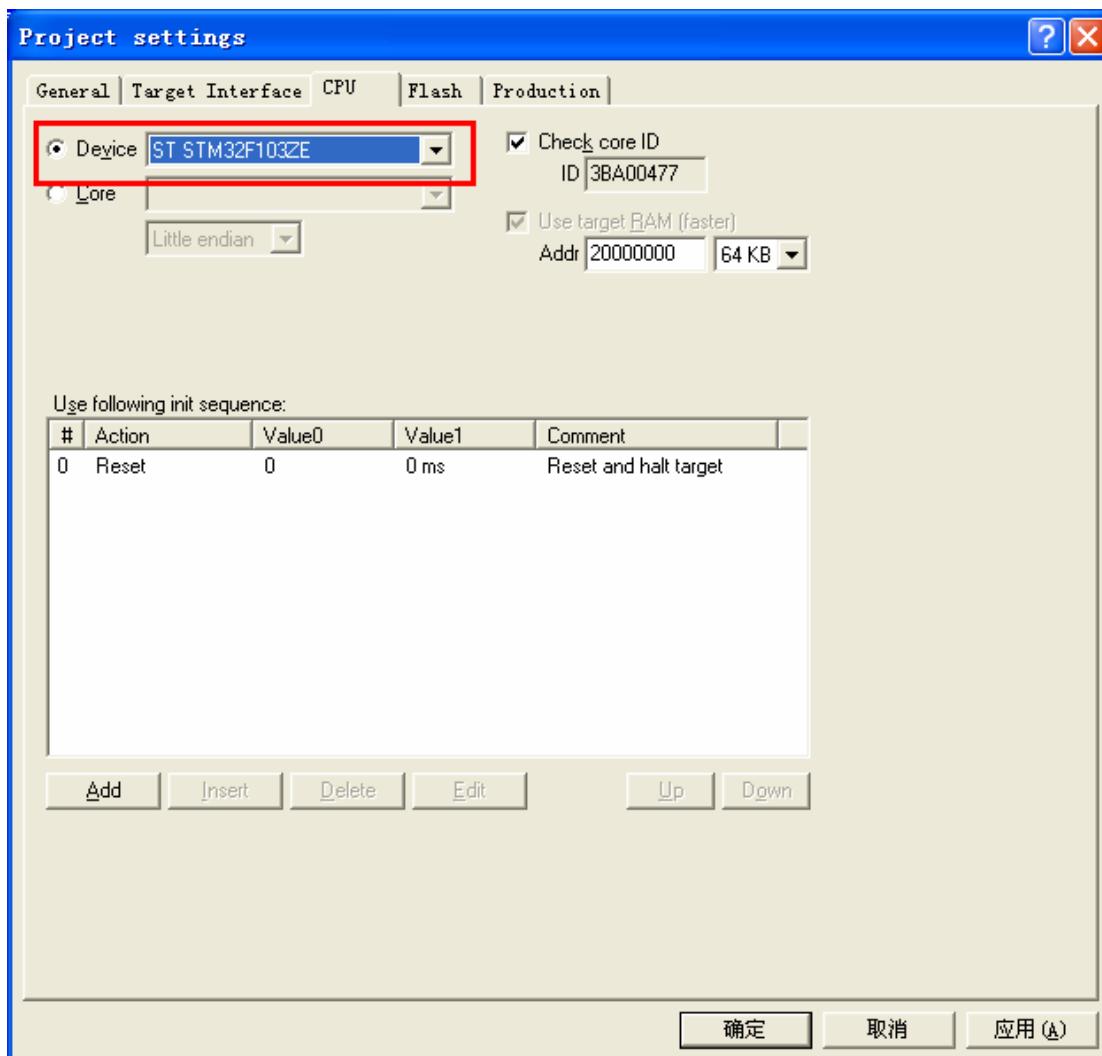
然后通过“File”菜单下的“Open...”来打开需要烧写的文件，可以是.bin 格式，也可以是.hex 格式，甚至可以是.mot 格式。注意起始地址。



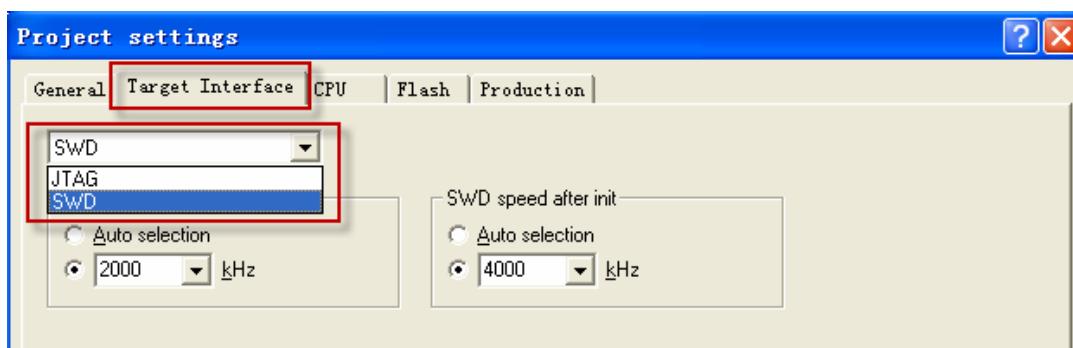
首次使用的时候应该在 File 菜单，选择 Open Project ，选择你的目标芯片：



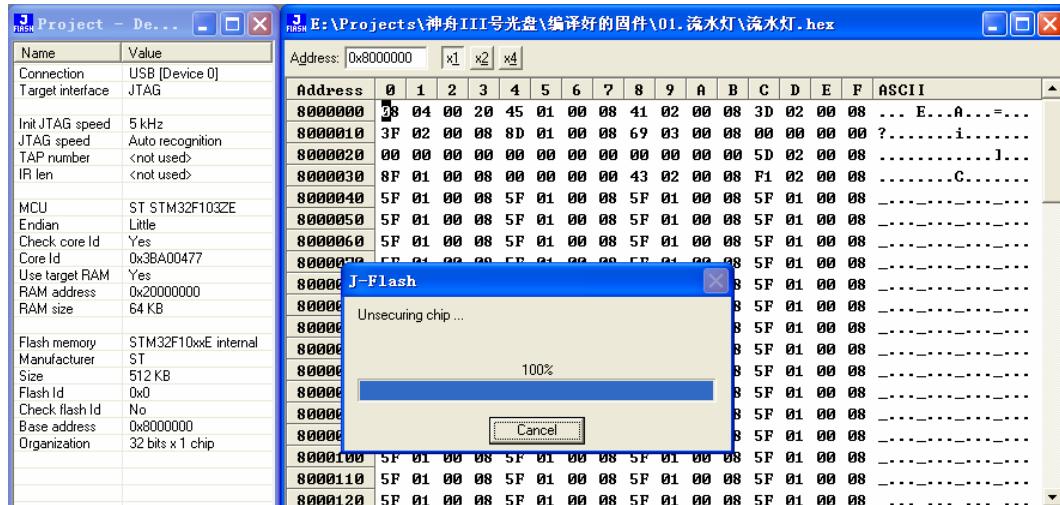
接下来在“Options”选择“Project settings”在弹出的对话框中，点击 CPU 标签，选择神舟 III 号使用的处理器 STM32F103ZE。



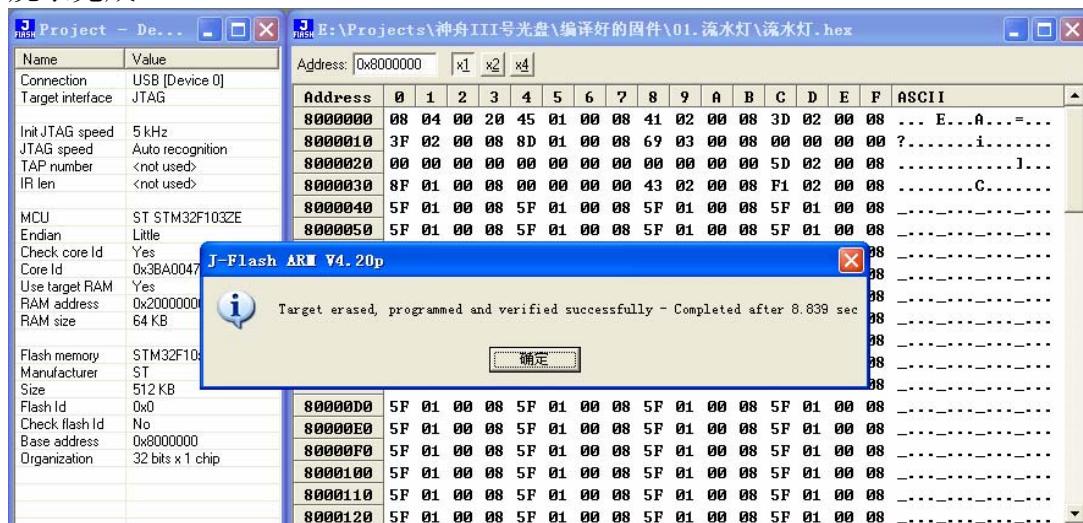
点击 Target Interface 选择使用 JTAG 还是 SWD 接口，JLINK V8 可以选择用 JTAG 或 SWD 接口来下载，在线调试，而这两种接口 STM32 处理器也都是支持的。



设置好之后，就可以到 Target 里面进行操作，一般步骤是先“Connect”，然后“Erase Chip”，然后“Program”。大部分芯片还可以加密，主要的操作都在 Target 菜单下完成。为了方便操作，我们可以直接按 F7 快捷键，自动擦除和烧录固件。



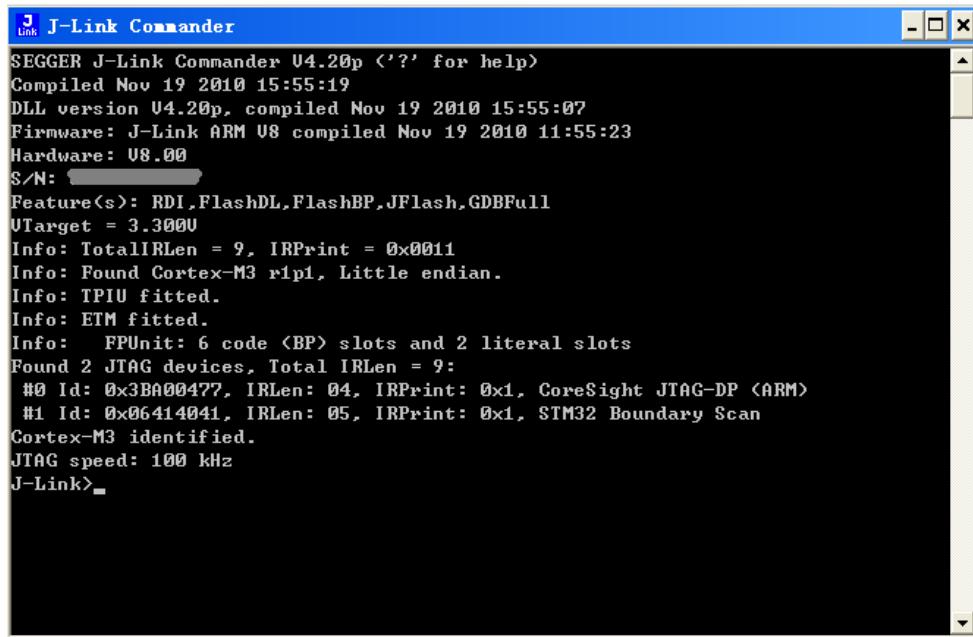
烧录完成



关于J-FLASH ARM更详细的操作请参阅JLINK的用户手册。

3.2.2 使用J-Link command 设置和查看相关调试信息

J-Link command包含了所有设置和查看相关调试信息的命令，它是基于命令行输入方式。打开J-Link command 界面，显示JLINK的相关版本信息，如果连接了目标板，将显示目标板的状态和目标CPU内核信息等。如下



```
J-Link Commander  
SEGGER J-Link Commander V4.20p <'?' for help>  
Compiled Nov 19 2010 15:55:19  
DLL version V4.20p, compiled Nov 19 2010 15:55:07  
Firmware: J-Link ARM V8 compiled Nov 19 2010 11:55:23  
Hardware: U8.00  
S/N: [REDACTED]  
Feature(s): RDI,FlashDL,FlashBP,JFlash,GDBFull  
UTarget = 3.300V  
Info: TotalIRLen = 9, IRPrint = 0x0011  
Info: Found Cortex-M3 r1p1, Little endian.  
Info: TPIU fitted.  
Info: ETM fitted.  
Info: FPUunit: 6 code <BP> slots and 2 literal slots  
Found 2 JTAG devices, Total IRLen = 9:  
#0 Id: 0x3BA00477, IRLen: 04, IRPrint: 0x1, CoreSight JTAG-DP <ARM>  
#1 Id: 0x06414041, IRLen: 05, IRPrint: 0x1, STM32 Boundary Scan  
Cortex-M3 identified.  
JTAG speed: 100 kHz  
J-Link>
```

J-Link command包含丰富的测试、查看等命令，相关命令的详细信息可在 J-Linkcommand 命令行下输入”?”号然后回车有详细的说明，操作非常方便。JLINK的其他软件暂不详细介绍，请用户自行参阅JLINK的用户手册即可得到详细的答案。

通过串口下载固件

(首先推荐使用 Jlink 下载，其次是 USB 下载)

3.3 如何通过串口下载一个固件到神舟III号开发板

STM32的程序下载有多种方法，可以通过USB、串口1、串口2 (remapped), CAN2 (remapped)、USB、JTAG、SWD等方式下载。这几种方式都可以用来给神舟III号开发板下载程序，这里详细介绍通过串口下载固件到神舟III号开发板的过程。

硬件设置

第一步神舟III号启动模式设置为SystemBoot。将跳线JP13短接2↔3，JP15短接1↔2，用于串口下载。此模式下，STM32在复位后不会执行用户代码，而是等待串口更新程序。

跳线与启动模式设置关系如下：

| BOOT1 (JP13) | BOOT0 (JP15) | 功能 |
|--------------|--------------|---------------|
| ANY | 2-3 | User Boot(默认) |
| 2-3 | 1-2 | System Boot |
| 1-2 | 1-2 | SRAM Boot |

STM32的串口下载使用串口1或串口2都可以，一般是通过串口2下载时不需要改变跳线的位置。

第二步，使用神舟III号配套的交叉串口线将神舟开发板与PC连接（或者与USB转串口线连接），连接到神舟III号开发板的串口1（注意使用串口1时将JP3和JP5跳到2↔3）或串口2（注意使用串口2时将JP4跳到2↔3也就是左侧），并为神舟开发板供电。

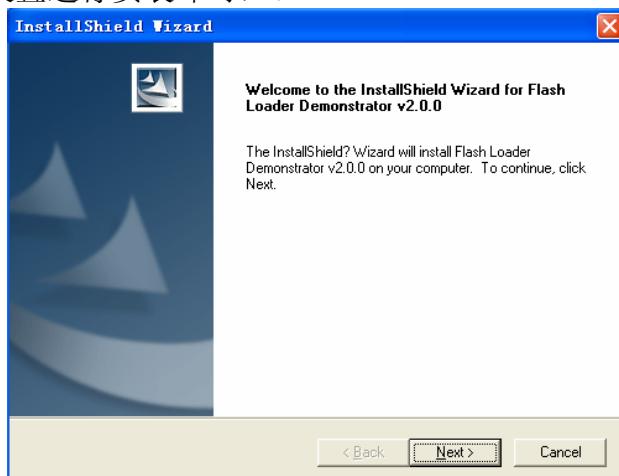
软件使用

接下来，我们介绍如何使用Flash_Loader_Demonstrator软件串口下载过程。

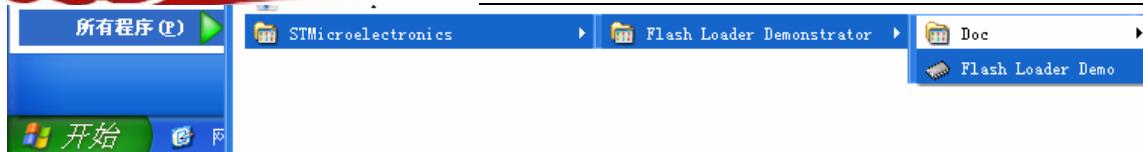
第一步：打开 神舟III号光盘\固件升级工具\目录。解压

um0462_Flash_Loader_Demonstrator_V2.0_Setup.zip文件后，双击

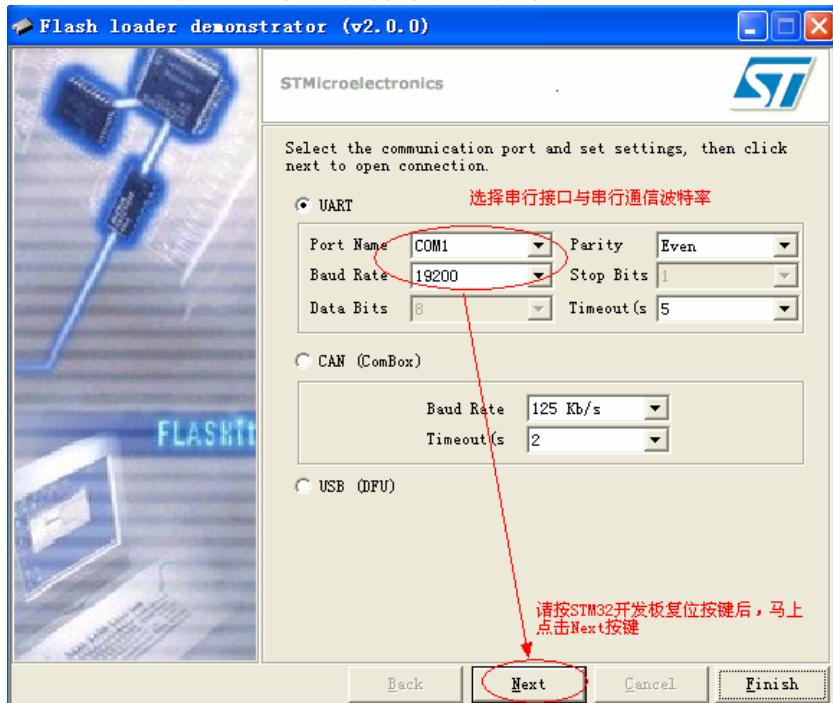
Flash_Loader_Demonstrator_V2.0_Setup.exe安装Flash_Loader_Demonstrator软件（按缺省设置进行安装即可）。



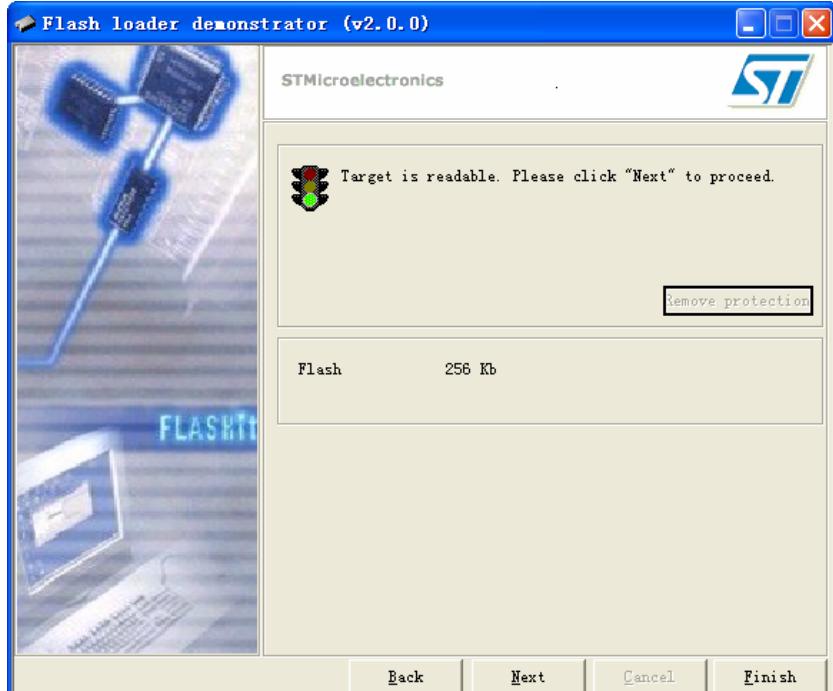
第二步，安装好软件后，运行Flash_Loader_Demonstrator软件，



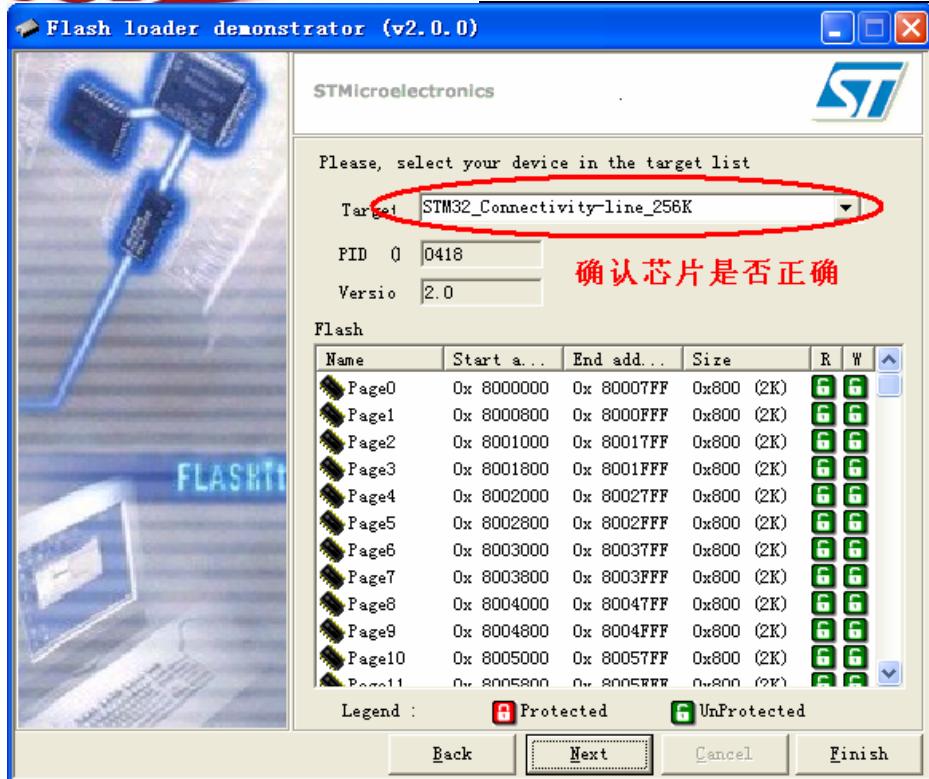
在弹出界面的中，选择正确的PC机使用的串口号，串行通信波特率可以自由选择，神舟III号开发板将自动识别您设置的波特率，当然一般选择高速率的波特率以提高下载速度，但是如果您使用的串口线较长或者使用的是USB转串口线则会导致通信质量下降，此时只能尝试选择低速率的波特率多试几次。



按上图设置好以后，先按开发板的复位按键后，再点击Next按键，此时，如下界面。

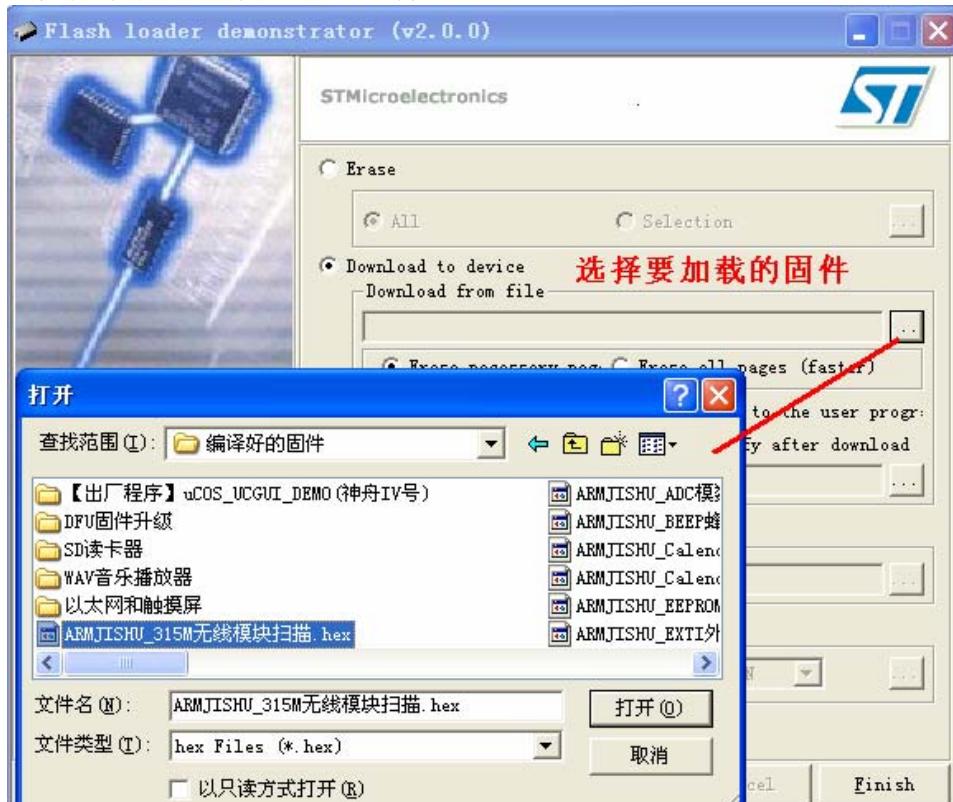


点击Next

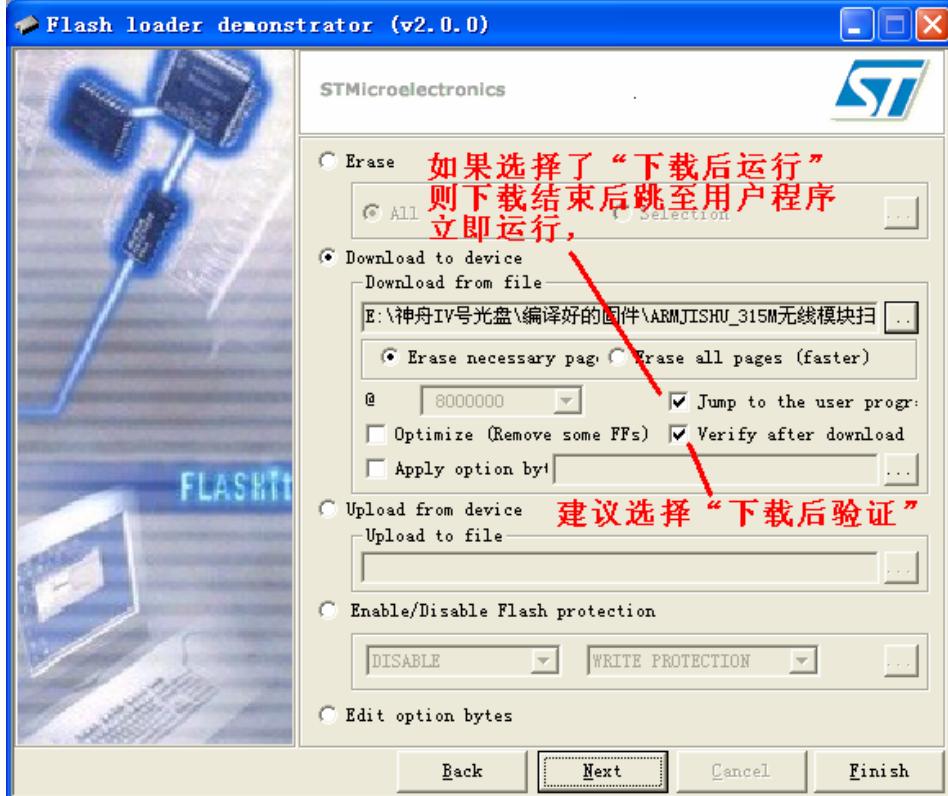


点击Next

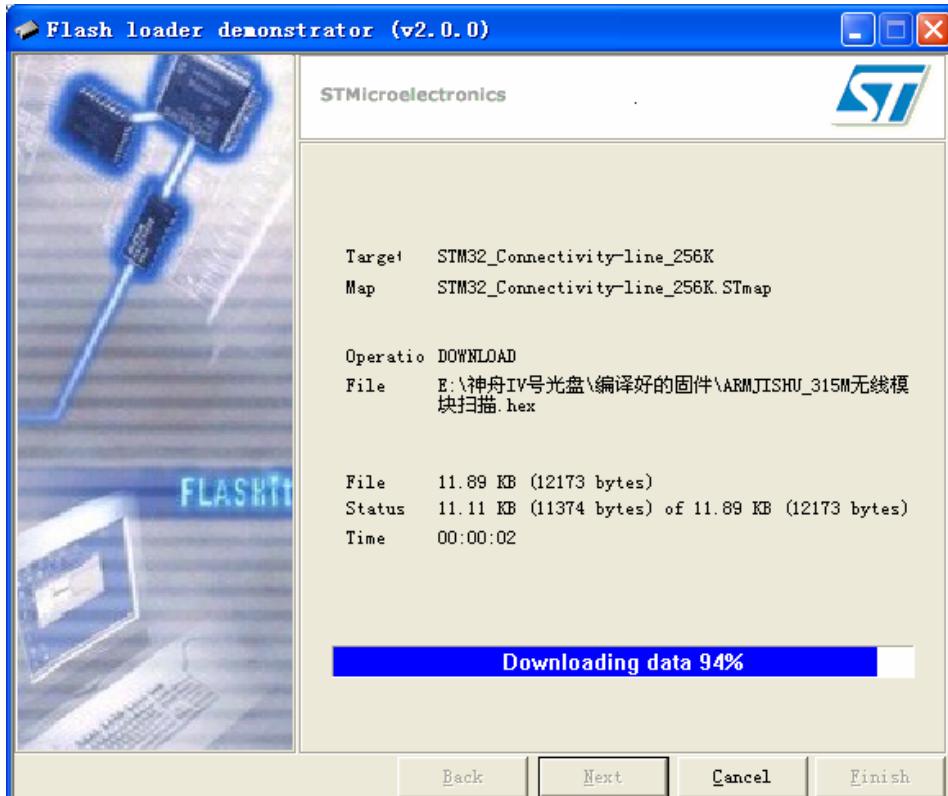
浏览加载需要下载的HEX文件。



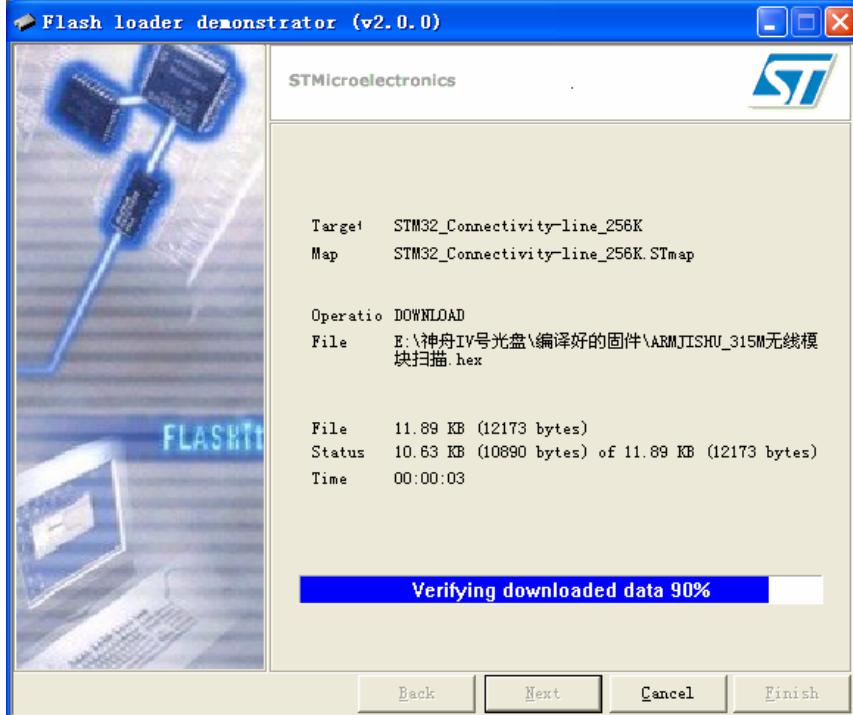
根据需要选择选项：



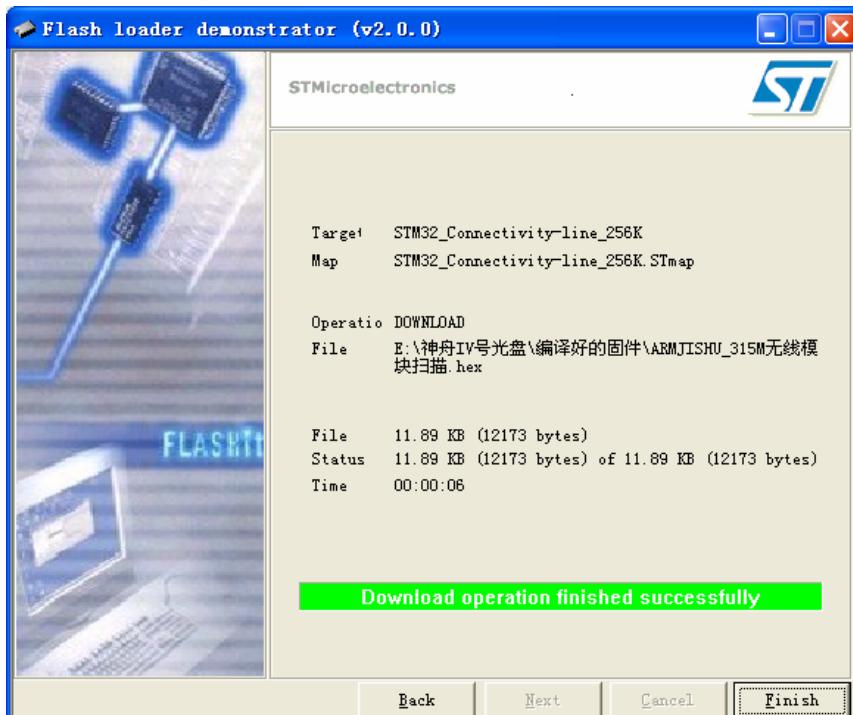
点击Next进行编程。



上图为下载过程



上图为验证过程



上图为下载验证后的界面

至此，HEX 文件已经成功下载了，按照上图如果选择 Jump to the user program：“下载后运行”，则程序已经开始运行；如果没有选择该项，我们需要将启动模式设置为 User Boot 模式，即将 JP15 的 2-3 脚短接。复位神舟 III 号开发板即可看到程序运行的实验现象。

3.4 如何在IAR开发环境中使用JLINK在线调试

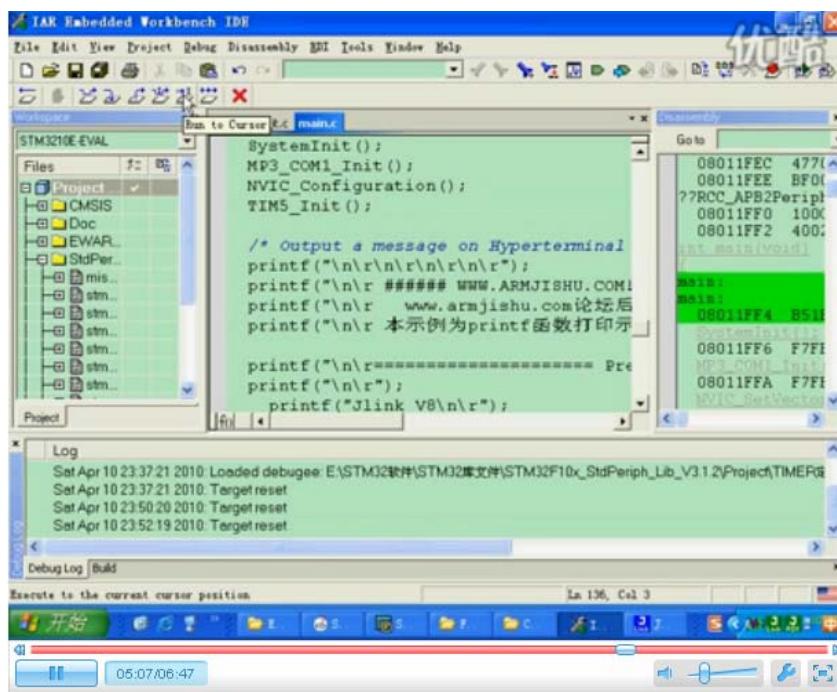
关于如何在 IAR 开发环境中使用 JLINK 在线调试请查阅《错误！未找到引用源。》章节。

更多关于使用 JLINK V8 调试下载工具单步调试 STM32、查看寄存器值、查看变量值等等技巧，请查阅网络上的视频。例如在百度中输入“armjishu youku”关键字即可看到链接：



Jlink V8 仿真调试STM32(Cortex-M3)(www.armjishu.com) - 视频

视频地址：http://v.youku.com/v_show/id_XMTY0MjMzNzg0.html



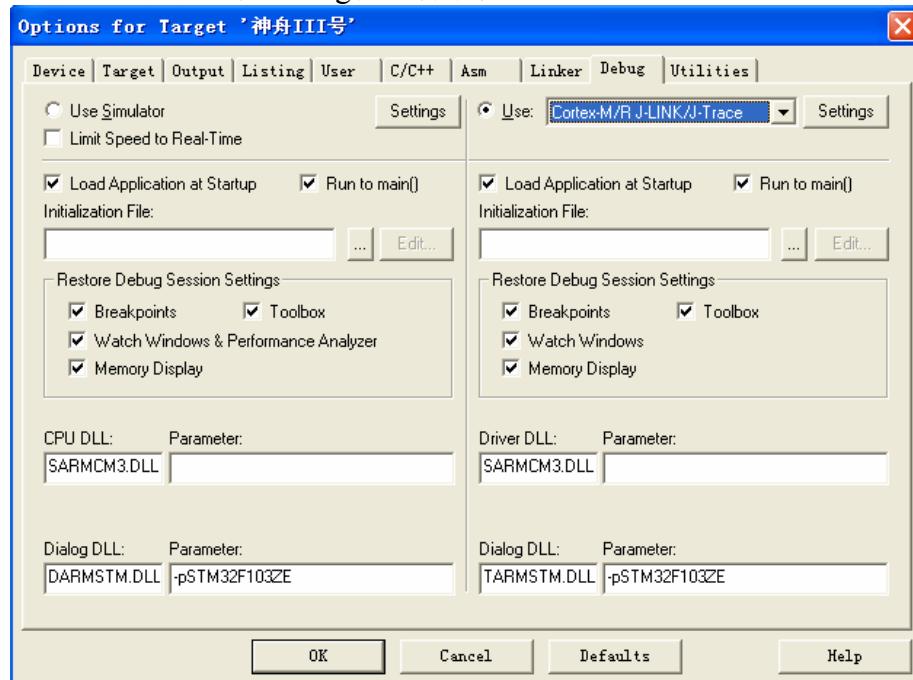
3.5 如何在MDK开发环境中使用JLINK在线调试

利用串口，我们只能下载程序，并不能实时跟踪，而利用调试工具，比如JLINK、ULINK等就可以实时跟踪程序，使你的开发事半功倍。这里我们以JLINK V8为例，说说如何在线调试。

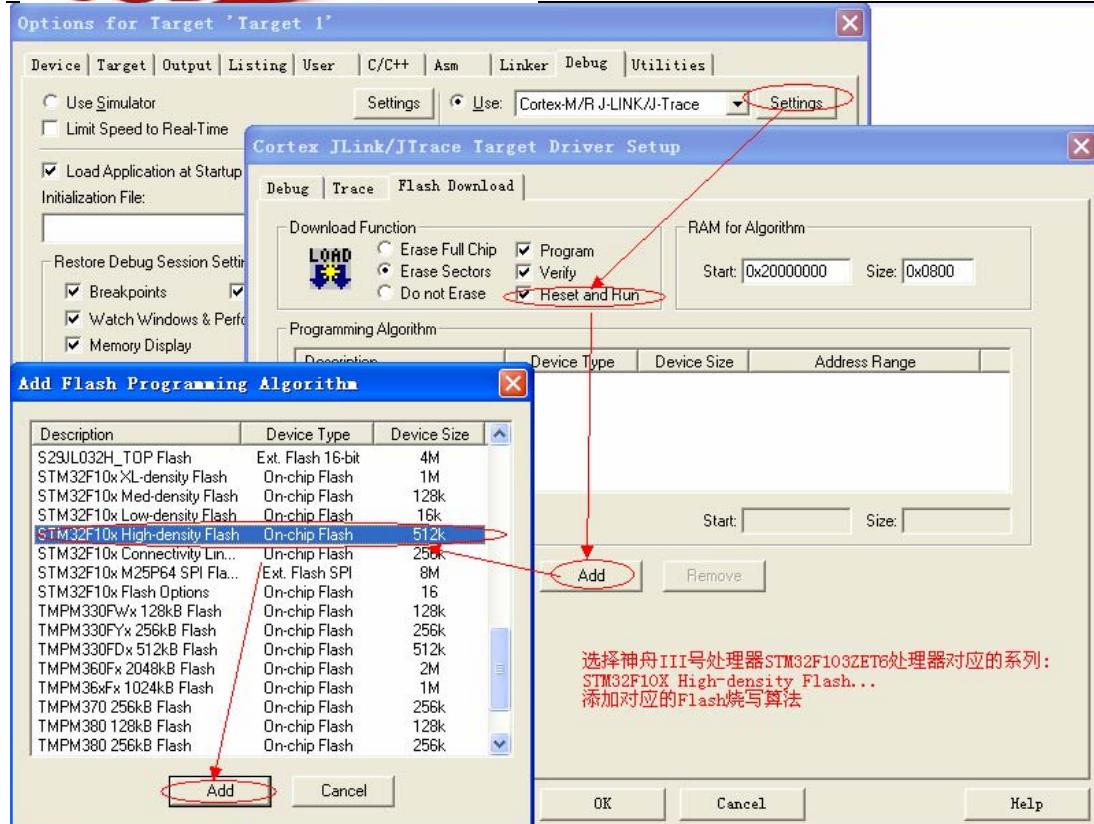
JLINK V8支持JTAG和SWD，而STM32也支持JTAG和SWD。所以，我们有2种方式可以用来调试，JTAG调试的时候，占用的IO线比较多，而SWD调试的时候占用的IO线很少，只需要2跟即可。

JLINKV8的安装我们这里就不说了，JLINK的光盘里面有详细的资料。在安装了JLINK V8之后，我们接上JLINK-V8，并把JTAG口插到神舟III号开发板上，打开[神舟III号光盘下的流水灯例程](#)，点击

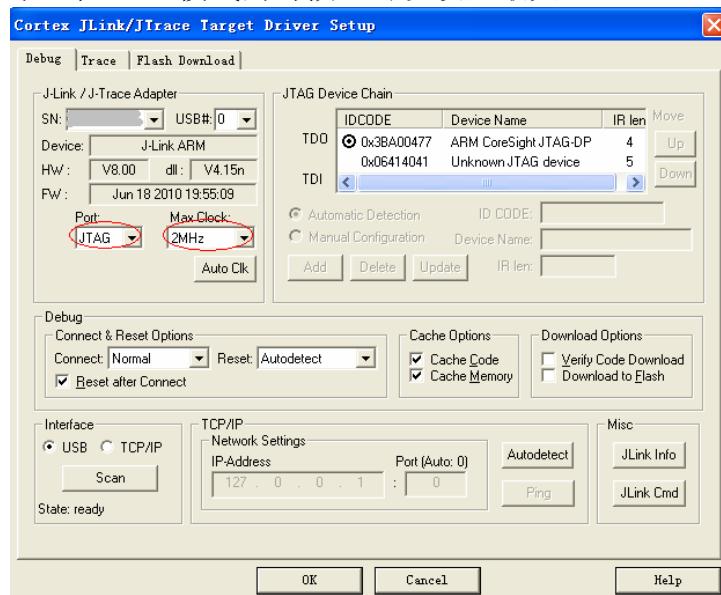
，打开Options for Target ‘神舟III号’选项卡，在Debug栏选择仿真工具为Cortex-M3 J-LINK，如下图所示：



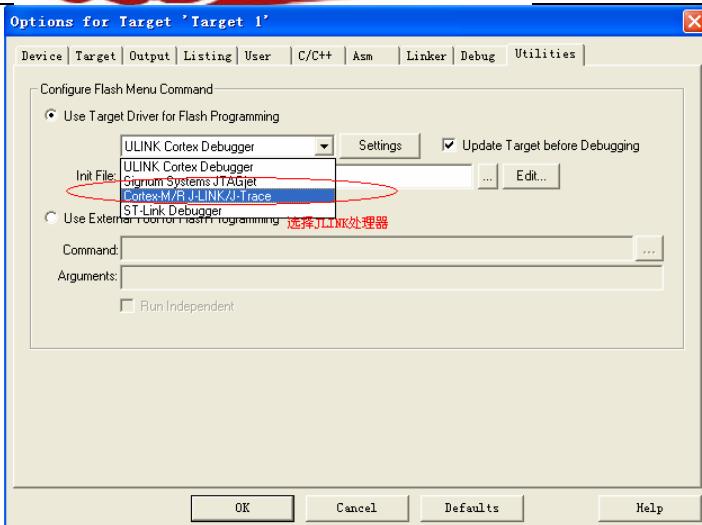
点击右侧的Setting，配置JLINK 下载参数，主要是选择目标处理器的系列，添加目标芯片对应的烧写算法。



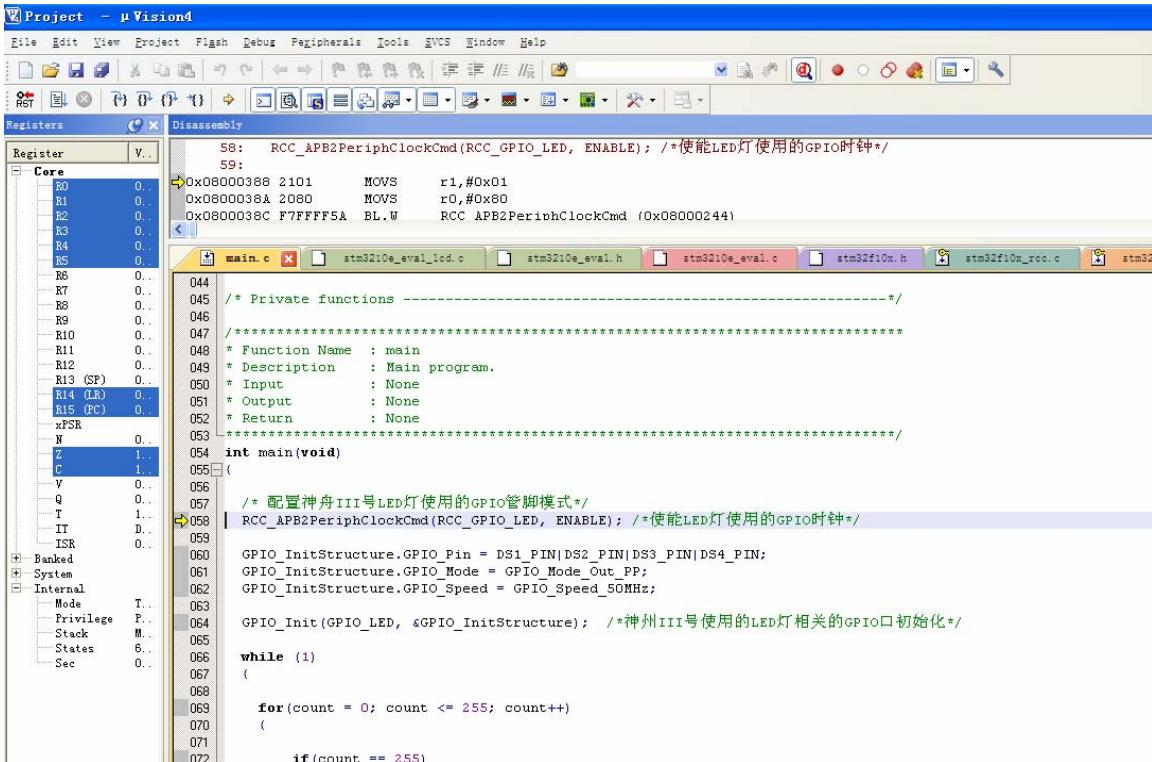
上图中，我们使用J-LINK V8的JTAG模式调试（当然也可以进行SWD模式调试，只要我们在Port处选择SW即可）。Max Clock，可以点击Auto Clk来自动设置，上图中JLINK自动设置最大时钟为2Mhz（注意这里不能设置的太大，如果太大，可能导致JTAG使用不了！但SWD模式的时候，可以设置最大10Mhz）。



单击OK，完成此部分设置。接下来，点击Utilities，进行公共参数设置如下图所示：



在设置完之后，点击OK，然后再点击OK，回到IDE界面，编译一下工程。再点击 ，开始仿真(如果开发板的代码没被更新过，则会先更新代码，再仿真，你也可以通过按 ，只下载代码，而不进入仿真)，如下图所示：



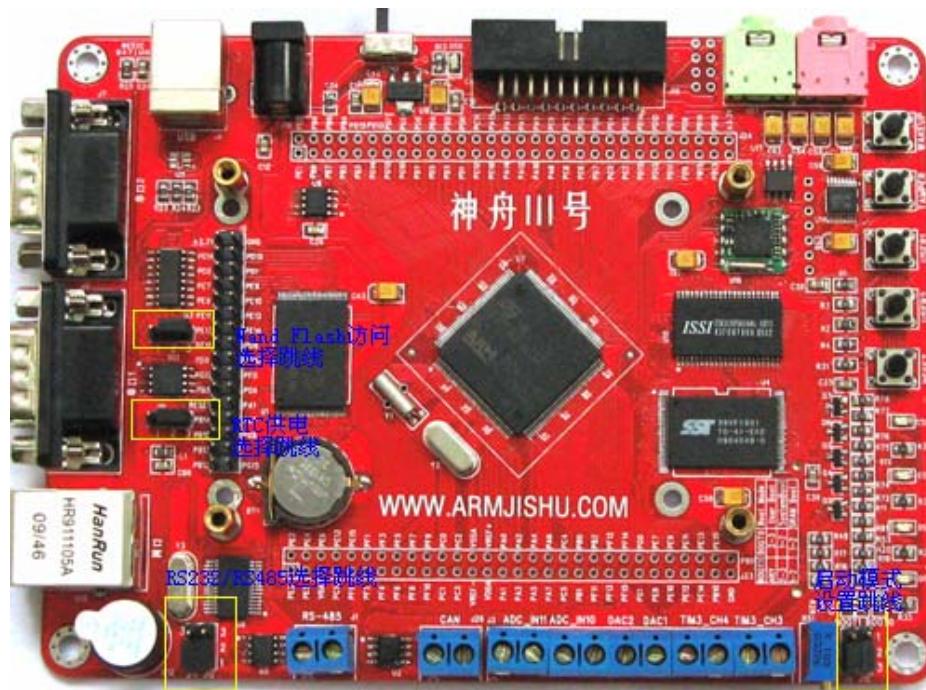
可以看到都是一些汇编码的查看，如果我们要快速运行到main函数，可以在main函数的第一句语句处放入断点，然后点击 ，来快速执行到该处。如下图所示：

```

    69:         for(count = 0; count <= 255; count++)
70:         {
71:             if(count == 255)
72:             {
73:                 /* Private functions -----*/
74:                 /* Function Name : main
75:                  * Description   : Main program.
76:                  * Input          : None
77:                  * Output         : None
78:                  * Return        : None
79: ****
80:             int main(void)
81:             {
82:                 /* 配置神舟III号LED灯使用的GPIO管脚模式*/
83:                 RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE); /*使能LED灯使用的GPIO时钟*/
84:
85:                 GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN;
86:                 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
87:                 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
88:
89:                 GPIO_Init(GPIO_LED, &GPIO_InitStructure); /*神州III号使用的LED灯相关的GPIO口初始化*/
90:
91:                 while (1)
92:                 {
93:
94:                     for(count = 0; count <= 255; count++)
95:                     {
96:
97:                         if(count == 255)
98:                         {
99:                             count = 0;
100:                            GPIO_ResetBits(GPIO_LED, DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN);/*点亮所有的LED指示灯*/
101:                         }
102:                     }
103:                 }
104:             }
105:         }
106:     }
107: 
```

接下来，我们就可以和软件仿真一样的开始仿真了，不过这是真正的在硬件上的仿真，其结果更可信。如何在 MDK 开发环境中使用 JLINK 在线仿真就介绍到这里。

3.6 神舟III号跳线含义



3.6.1 启动模式选择跳线

| BOOT1 (J9) | BOOT0 (J10) | 功能 | 说明 |
|------------|-------------|---------------|--------------------|
| ANY | 2-3 | User Boot(默认) | 用主闪存存储器，即Flash启动 |
| 2-3 | 1-2 | System Boot | 系统存储器启动，用于串口下载 |
| 1-2 | 1-2 | SRAM Boot | SRAM启动，用于SRAM中调试代码 |

- ◆ 从主闪存存储器启动：主闪存存储器被映射到启动空间（0x0000 0000），但仍然能够在它原来的地址（0x0800 0000）访问它，即闪存存储器的内容可以在两个地址区域访问，0x0000 0000或0x0800 0000。
- ◆ 从系统存储器启动：系统存储器被映射到启动空间（0x0000 0000），但仍然能够在它原有的地址（0x1FFF F000）访问它。
- ◆ 从内置SRAM启动：只能在0x2000 0000开始的地址区访问SRAM。

使用JLINK调试下载时，请将启动模式设置为User Boot模式。

3.6.2 RS-232/RS-485选择跳线

串口2可通过跳线选择支持RS-232接口或RS-485接口，跳线定义如下：

| J14 | J12 | 串口2功能选择 |
|-----|-----|--------------|
| 1-2 | 1-2 | 串口2 RS-485接口 |
| 2-3 | 2-3 | 串口2 RS-232接口 |

3.6.3 RTC实时时钟跳线

当安装纽扣电池时，请用跳帽将 J11 的 2-3 脚短接起来，使用 RTC 实时时钟功能；

当没有安装电池时，请用跳帽将 J11 的 1-2 脚短接起来，不使能 RTC 实时时钟功能。

3.6.4 Nand Flash访问选择跳线

当使用中断方式访问 Nand Flash 时，请用跳帽将 J13 的 1-2 脚短接起来；

当使用查询方式访问 Nand Flash 时，请用跳帽将 J13 的 2-3 脚短接起来。

第四章 STM32神舟III号 零基础入门篇

4.1 如何从零开始新建STM32工程模板

4.1.1 如何去官网下载最新的STM32资料的方法

1、 打开网页，在百度搜索网页栏中搜索“ST”，如下图



2、点击 st 意法半导体进入 ST 官网

推荌音乐 – SongTaste 用音乐倾听彼此
你可能会喜欢他/她们 查找所有 大家推荐的歌曲 ST分析器 语言 这些歌曲如何排序? 点击
直接播放 1 2 3 4 5 6 7 8 9 10
www.songtaste.com/music/ 2012-11-2 - 百度快照

st 百度百科
ST, 在股票领域上可以理解为意法半导体企业发布上市的股票代码, ST股票; 在足球领域上可以理解为前锋; 在电脑硬件上可以理解为希捷硬盘的代号; 在...
意法半导体 - 股票 - striker (S - 直捷 - ST系统测试 - 查看全部>>
baike.baidu.com/view/180498.htm 2012-11-24

关于意法半导体st公司
STMicroelectronics... home about ST contacts press login28 Nov ST and JH Cooperate to Develop Digital TV Solutions for Inner Mongolia 27 Nov STMicroelectronics...
www.st.com/ 2012-11-29 - 百度快照

【长光ST】(FIBERNET ST)报价 图片 参数 评测 论坛 长光ST光纤...

3、点击微控制器和存储器



4、点击微控制器



5、选择 STM32-32 位的微控制器中的 F1 系列

The screenshot shows the STM32 product page. At the top, there's a navigation bar with links like '首页' (Home), '关于ST' (About ST), '联系我们' (Contact Us), '新闻中心' (News Center), and '登录' (Login). The '新闻中心' link is highlighted with a red box. Below the navigation is a search bar with placeholder text '搜索...'. The main content area features a large 'STM32' logo. To the right is the ST logo. The page is divided into several sections:

- 第一步**: A section titled '看货' (View Product) with a red arrow pointing to the 'STM32 - 32位微控制器' (STM32 - 32-bit MCUs) section. This section contains a chart showing Flash memory size (bytes) from 2 K to 1 M, with the STM32 32-bit MCUs series highlighted in blue.
- 第二步**: A section titled '直到下机' (Until Off-the-Shelf) with a red arrow pointing to the 'STM32 F3 系列' (STM32 F3 series) section. This section highlights the STM32 F3 series as the world's fastest Cortex™-M4 MCUs.
- 第三步**: A section titled '大放异彩' (Shine Brightly) with a red arrow pointing to the 'STM32 F1 系列' (STM32 F1 series) section. This section highlights the STM32 F1 series as the most cost-sensitive application.

On the right side, there's a sidebar with a '新产品' (New Product) button and a '产品' (Product) dropdown menu. The '新产品' section lists the STM32 F3 series mixed-signal MCUs, featuring Cortex™-M4 performance with a rich analog peripheral set, and offers free technical seminars. It also highlights the STM32 F3 Entry-level MCUs and STM32 DNA at budget price. The '产品' section lists the STM32 F4 series (high-performance) and the STM32 G4 series (cost-sensitive).

6、选择你需要的芯片，比如 STM32F103ZET6、STM32F107VCT6、STM32F207ZGT、STM32F407ZGT 等，这里我们用 STM32F103ZET6 做示范，如下图：

| Part Number | Package | Marketing Status | Cores | Operating Frequency(F) (Processor) | FLASH Size (Prog)(kB) | Internal RAM Size(kB) | 16-bit timer (IC/OC/PW) |
|-------------|-------------------------|------------------|---------------|------------------------------------|-----------------------|-----------------------|-------------------------|
| STM32F103RB | LQFP 48 7x7x1.4 | Active | ARM Cortex-M3 | 48 | 16 | 4 | 2x16-bit |
| STM32F103RC | LQFP 48 7x7x1.4 | Active | ARM Cortex-M3 | 48 | 32 | 8 | 2x16-bit |
| STM32F103CB | LQFP 48 7x7x1.4 | Active | ARM Cortex-M3 | 48 | 64 | 16 | 3x16-bit |
| STM32F103CB | LQFP 48 7x7x1.4 | Active | ARM Cortex-M3 | 48 | 128 | 16 | 3x16-bit |
| STM32F103CA | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 48 | 16 | 4 | 2x16-bit |
| STM32F103CB | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 48 | 32 | 8 | 2x16-bit |
| STM32F103CB | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 48 | 64 | 16 | 3x16-bit |
| STM32F103CB | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 48 | 128 | 16 | 3x16-bit |
| STM32F103CB | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 72 | 16 | 8 | 3x16-bit |
| STM32F103CB | LQFP 48 7x7x1.4; VFQ... | Active | ARM Cortex-M3 | 72 | 32 | 16 | 3x16-bit |
| STM32F103CB | LQFP 48 7x7x1.4 | Active | ARM Cortex-M3 | 72 | 64 | 20 | 4x16-bit |
| STM32F103CB | LQFP 48 7x7x1.4; VFQ... | Active | ARM Cortex-M3 | 72 | 128 | 20 | 4x16-bit |
| STM32F103CA | LQFP 64 10x10x1.4; T... | Active | ARM Cortex-M3 | 72 | 16 | 8 | 3x16-bit |
| STM32F103CB | LQFP 64 10x10x1.4; T... | Active | ARM Cortex-M3 | 72 | 32 | 16 | 3x16-bit |
| STM32F103CB | LQFP 64 10x10x1.4; T... | Active | ARM Cortex-M3 | 72 | 64 | 20 | 4x16-bit |
| STM32F103RB | TFBGA 64 3x3x1.2 | Active | ARM Cortex-M3 | 72 | 128 | 20 | 4x16-bit |
| STM32F103RC | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 72 | 256 | 48 | 8x16-bit |
| STM32F103RD | LQFP 64 10x10x1.4; W... | Active | ARM Cortex-M3 | 72 | 384 | 64 | 8x16-bit |
| STM32F103RE | LQFP 64 10x10x1.4; W... | Active | ARM Cortex-M3 | 72 | 512 | 64 | 8x16-bit |
| STM32F103RF | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 72 | 768 | 96 | 12x16-bit |
| STM32F103RG | LQFP 64 10x10x1.4 | Active | ARM Cortex-M3 | 72 | 1024 | 96 | 12x16-bit |
| STM32F103RA | VFQFN 36 6x6x1.0 | Active | ARM Cortex-M3 | 72 | 16 | 8 | 3x16-bit |
| STM32F103RA | VFQFN 36 6x6x1.0 | Active | ARM Cortex-M3 | 72 | 32 | 16 | 3x16-bit |
| STM32F103RA | VFQFN 36 6x6x1.0 | Active | ARM Cortex-M3 | 72 | 64 | 20 | 4x16-bit |
| STM32F103RA | VFQFN 36 6x6x1.0 | Active | ARM Cortex-M3 | 72 | 128 | 20 | 4x16-bit |
| STM32F103RA | LFBGA 100 10x10x1.7... | Active | ARM Cortex-M3 | 72 | 64 | 20 | 4x16-bit |

7、打开后选择 Design support

STMicroelectronics

HOME ABOUT ST CONTACTS PRESS LOGIN

stm32f103rb

Mainstream Performance line, ARM Cortex-M3 MCU with 128 kbytes Flash, 72 MHz CPU, motor control, USB and CAN

Quick view Design support Orderable products Related information ONLINE SUPPORT

Design support

• Technical Documentation • Models & Simulators • Support

Technical Documentation

• DATASHEET • ERRATA SHEETS • RELEASE NOTES

• TECHNICAL ARTICLES • APPLICATION NOTES • USER MANUALS

• TECHNICAL NOTES • REFERENCE MANUALS

• LICENSE AGREEMENTS • DATA SHEETS

• APPLICATIONS PRACTICE PUBLICATIONS • DEVICE OPTION LEVELS

DATASHEET

Description Version Size

DS5319: Medium-density performance line ARM-based 32-bit MCU with 64 or 128 kB Flash, USB, CAN, 7 timers, 2 ADCs, 9 communication interfaces 13 1414KB

APPLICATION NOTES

Description Version Size

AN2812: Voice demonstration using a Speex audio codec on STM32F101xx and STM32F103xx 2 257KB

AN2548: Using the STM32F101xx and STM32F103xx DMA controller 3 142KB

AN4076: Two or three shunt resistor based current sensing circuit design in 3-phase inverters 1 119KB

AN2945: STM32 and STM32™ MCUs: a consistent 8/32-bit product line for painless migration 1 213KB

AN2606: STM32™ microcontroller system memory boot mode 14 877KB

AN3078: STM32™ in-application programming over the I²C bus 1 671KB

AN3116: STM32™'s ADC modes and their applications 1 256KB

AN4023: STM32 secure firmware upgrade (SPU) overview 1 41KB

AN3429: STM32 proprietary code protection overview 1 61KB

APPLICATION NOTES

| Description | Version | Size |
|--|---------|--------|
| AN2812: Vocoder demonstration using a Speex audio codec on STM32F101xx and STM32F103xx microcontrollers | 2 | 257KB |
| AN2548: Using the STM32F101xx and STM32F103xx DMA controller | 3 | 142KB |
| AN4076: Two or three shunt resistor based current sensing circuit design in 3-phase inverters | 1 | 119KB |
| AN2945: STM32 and STM32™ MCUs: a consistent 8/32-Bit product line for painless migration | 1 | 213KB |
| AN2656: STM32™ microcontroller system memory boot mode | 14 | 877KB |
| AN3078: STM32™ in-application programming over the I²C bus | 1 | 671KB |
| AN3116: STM32™'s ADC modes and their applications | 1 | 236KB |
| AN4023: STM32 secure firmware upgrade (SPU) overview | 1 | 41KB |
| AN3429: STM32 proprietary code protection overview | 1 | 61KB |
| AN3580: STM32 firmware library for dSPIN L6470 | 1 | 774KB |
| AN4013: STM32F1xx, STM32F2xx, STM32F4xx, STM32L1xx, STM32F30/31/37/38x timer overview | 2 | 271KB |
| AN2658: STM32F10xxx LCD glass driver firmware | 2 | 868KB |
| AN2224: STM32F10xxx I²C optimized examples | 4 | 123KB |
| AN2868: STM32F10xxx internal RC oscillator (HSI) calibration | 1 | 169KB |
| AN2557: STM32F10x in-application programming using the USART | 8 | 215KB |
| AN2654: STM32F101xx and STM32F103xx RTC calibration | 1 | 145KB |
| AN2429: STM32F101xx, STM32F102xx and STM32F103xx low-power modes | 2 | 466KB |
| AN3095: STEVAL-15V002V1, STEVAL-15V002V2 3 kW grid-connected PV system, based on the STM32F103xx | 3 | 3676KB |
| AN2639: Soldering recommendations and package information for Lead-Free ECOPACK® microcontrollers | 2 | 209KB |
| AN1015: Software techniques for improving microcontroller EMI performance | 1 | 105KB |
| AN2550: Smartcard interface with the STM32F10x microcontrollers | 3 | 501KB |
| AN2667: Oscillator design guide for STM8S, STM8A, and STM32F1 microcontrollers | 6 | 280KB |
| AN3422: Migration of microcontroller applications from STM32F1 to STM32L1 series | 2 | 312KB |
| AN3364: Migration and compatibility guidelines for STM32 microcontroller applications | 3 | 120KB |
| AN4088: Migrating from STM32F1 to STM32F0 | 1 | 919KB |
| AN3427: Migrating microcontroller application from STM32F1 to STM32F2 series | 1 | 377KB |
| AN2799: Measuring main power consumption with the STM32x and STPM01 | 1 | 376KB |
| AN3070: Managing the Driver Enable signal for RS-485 and I²D-Link communications with the STM32™'s USART | 1 | 192KB |
| AN2666: Improving STM32F101xx and STM32F103xx ADC resolution by oversampling | 1 | 207KB |
| Production programming solutions for the STM32 | 1.0.3 | 155KB |

FIRMWARE

| Description | Version | Size |
|---|---------|---------|
| Using the STM32F101xx and STM32F103xx DMA controller | 2.0.0 | 111KB |
| EEPROM emulation in STM32F101xx and STM32F103xx microcontrollers | 3.1.0 | 1026KB |
| Smartcard interface with the STM32F101xx and STM32F103xx | 1.0 | 1098KB |
| Improving STM32F101xx and STM32F103xx ADC resolution by oversampling | 1.0 | 1129KB |
| Driving bipolar stepper motors using a medium-density STM32F103xx microcontroller | 2.0.0 | 1039KB |
| Clock/calender implementation on the STM32F10xx microcontroller RTC | 1.0 | 1418KB |
| How to achieve 32-bit timer resolution using the link system in STM32F101xx and STM32F103xx microcontrollers | 3.0.0 | 1108KB |
| STM32F101xx and STM32F103xx low-power modes | 2.0.0 | 1294KB |
| STM32F10xxx Speex library firmware STM32_StdPeriph_Lib_speex, audio | 2.0.0 | 1786KB |
| STM32F10xxx DSP library firmware | 2.0.0 | 1392KB |
| Archive for legacy STM32F10xxx Firmware Library V2.0.3 and all related Firmware packages | 2.0.3 | 23428KB |
| CEC (consumer electronic control) C library using the STM32F101xx, STM32F102xx and STM32F103xx microcontrollers | 2.0.0 | 1692KB |
| STM3210B-EVAL demonstration Firmware | 2.0.0 | 2308KB |
| STM32F101xx and STM32F103xx medium- and high-density devices: advanced I²C examples | 4.0 | 1506KB |
| STM32F10x standard peripheral library | 3.5.0 | 21617KB |
| Patch to fix STM32F103xx firmware library v2.0.3 limitations | 2.0.3 | 143KB |
| STM32F10xx motor control firmware library for the L6470 dSPIN IC | 1.0.0 | 675KB |
| Implementing receivers for infrared remote control protocols using STM32F1 microcontrollers | 2.0 | 12984KB |
| STM32 I²C Communication peripheral application library | 1.1.0 | 1969KB |
| STM32F10x motor control firmware for easySPIN L6474 | 1.0.1 | 1333KB |
| STM32F20x and STM32L1xx USB full-speed device library | 3.4.0 | 4030KB |
| STM32 PMSM FOC SDK motor control firmware library (web distribution version) | 3.2 | 36400KB |
| STM32 PMSM FOC SDK motor control firmware library (web distribution, not recommended for new designs) | 3.0 | 26645KB |
| STM32 embedded GUI library | 2.0.0 | 25243KB |
| STM8 and STM32 embedded software solutions | 1.0.1 | 2280KB |
| STM32 motor FOC firmware library | 2.0.1 | 6143KB |

SW DEMOS

以上都是官网提供的文档资料，具体请大家登陆官网查看，比如我要下载一个最新的库，大家看下图，这个库文件我找到了，提示说是 3.5.0 版本的库文件：

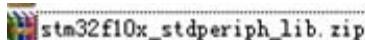
The screenshot shows the 'FIRMWARE' section of the STM32 Firmware Library. The 'STH32F10x standard peripheral library' entry is highlighted with a red box and a green circle. The table lists various firmware packages with their versions and sizes.

| Description | Version | Size |
|---|--------------|----------------|
| Using the STM32F101xx and STM32F103xx DMA controller | 2.0.0 | 1118KB |
| EEROM emulation in STM32F101xx and STM32F103xx microcontrollers | 3.1.0 | 1026KB |
| Smartcard interface with the STM32F101xx and STM32F103xx | 1.0 | 1098KB |
| Improving STM32F101xx and STM32F103xx ADC resolution by oversampling | 1.0 | 1129KB |
| Driving bipolar stepper motors using a medium-density STM32F103xx microcontroller | 2.0.0 | 1036KB |
| Clock/calendar implementation on the STM32F10xxx microcontroller RTC | 1.0 | 1418KB |
| How to achieve 32-bit timer resolution using the link system in STM32F201xx and STM32F103xx microcontrollers | 3.0.0 | 1108KB |
| STM32F101xx and STM32F103xx low-power modes | 2.0.0 | 1294KB |
| STM32F10xxx Speex library firmware STM32_StdPeriph_Lib, speex, audio | 2.0.0 | 1786KB |
| STM32F10xxx DSP Library firmware | 2.0.0 | 1392KB |
| Archive for legacy STM32F10xxx Firmware Library V2.0.3 and all related Firmware packages | 2.0.3 | 2342KB |
| CEC (consumer electronic control) C library using the STM32F101xx, STM32F102xx and STM32F103xx microcontrollers | 2.0.0 | 1692KB |
| STM3210B-EVAL demonstration firmware | 2.0.0 | 2308KB |
| STM32F101xx and STM32F103xx medium- and high-density devices: advanced I ² C examples | 4.0 | 1506KB |
| STH32F10x standard peripheral library | 3.5.0 | 21617KB |
| Patch to fix STM32F10xxx firmware library V2.0.3 limitations | 2.0.3 | 143KB |
| STM32F10xx motor control firmware library for the L6470 dSPIN IC | 1.0.0 | 678KB |
| Implementing receivers for infrared remote control protocols using STM32F1 microcontrollers | 2.0 | 12994KB |
| STM32 I2C Communication peripheral application library | 1.1.0 | 1960KB |
| STM32F10xx motor control firmware for easySPIN L6474 | 1.0.1 | 1393KB |
| STM32F10x and STM32L1xx USB full-speed device library | 3.4.0 | 4030KB |
| STM32 PMSM FOC SDK motor control firmware library (web distribution version) | 3.2 | 36400KB |
| STM32 PMSM FOC SDK motor control firmware library (web distribution, not recommended for new designs) | 3.0 | 26645KB |
| STM32 embedded GUI library | 2.0.0 | 23243KB |
| STM8 and STM32 embedded software solutions | 1.0.1 | 2280KB |
| STM32 motor FOC firmware library | 2.0.1 | 6142KB |

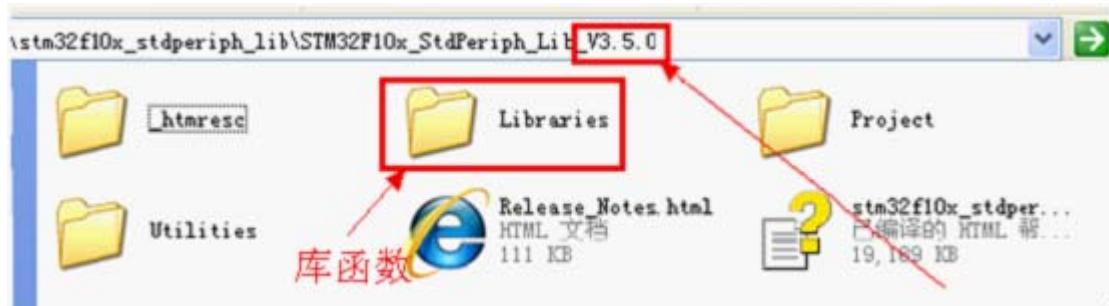
4.1.2 获取ST库源码

在新建工程模板之前，我们首先需要获取到 ST 库的源码，源码从上面的流程就可以从 ST 的官方网站下载到，在这里我们以 V3.5.0 的库来新建我们的工程模板。

可以看到该库的版本为 3.5.0 版本，下载后



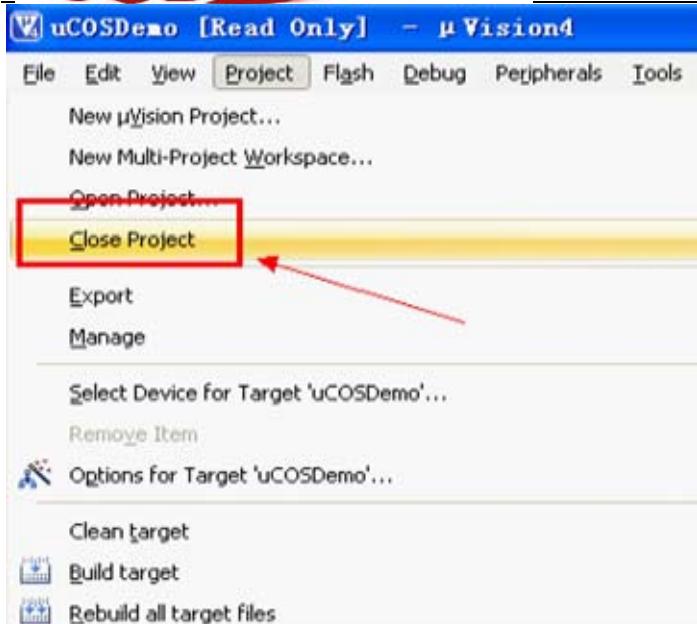
解压缩之后，真正的标准库函数就在 Libraries 文件夹中



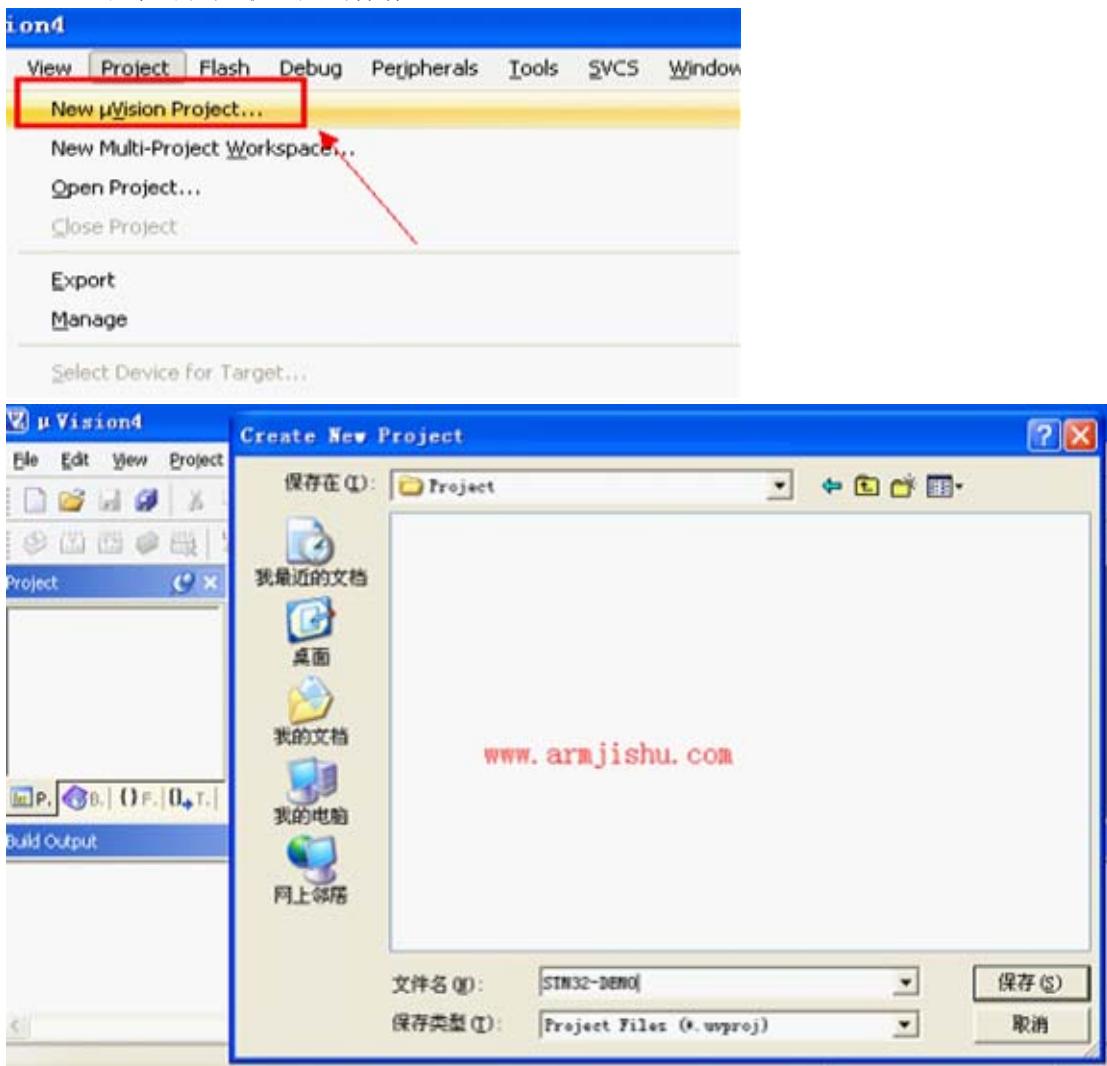
4.1.3 开始新建工程

点击桌面 UVision4 图标，启动软件，如下图。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。

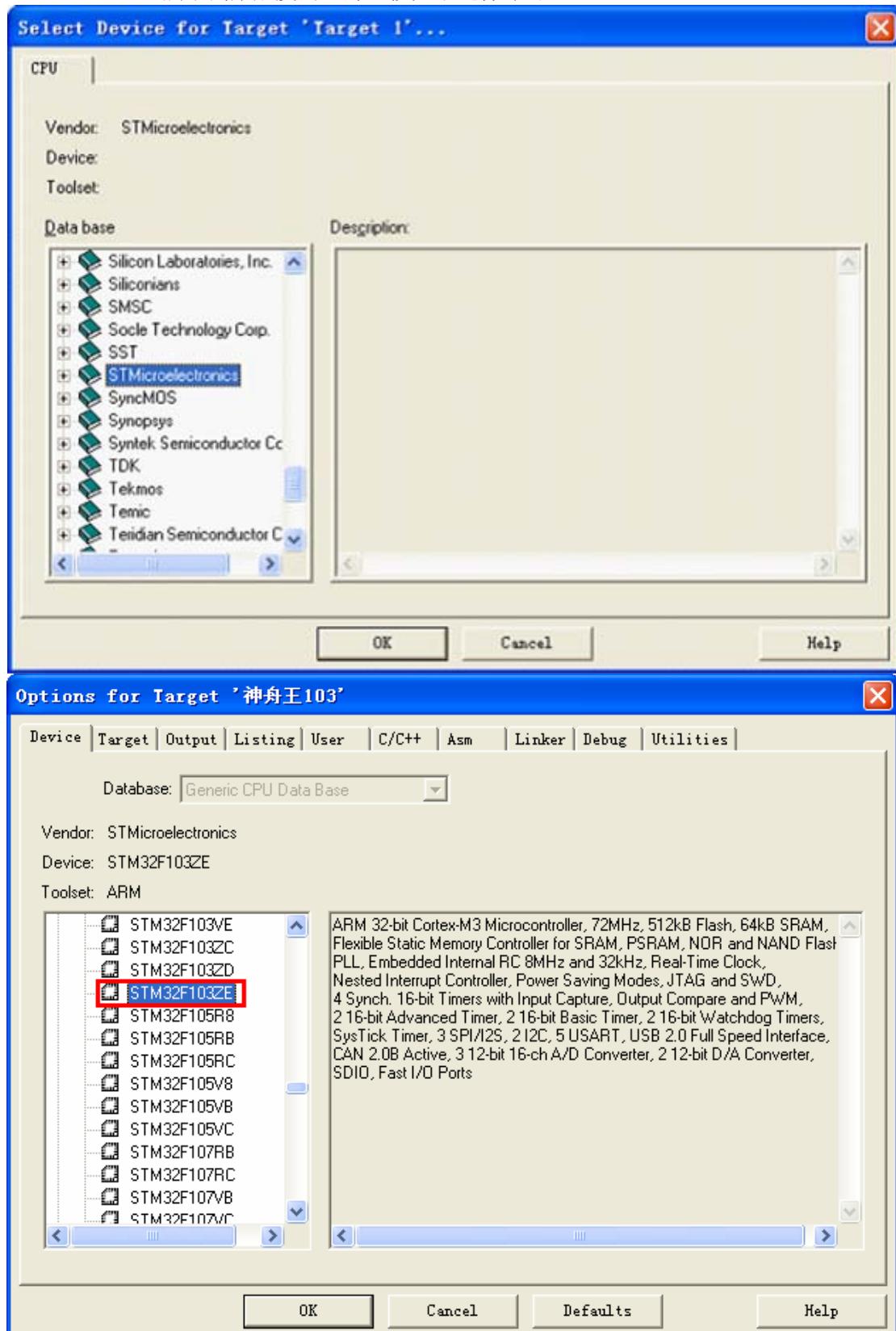




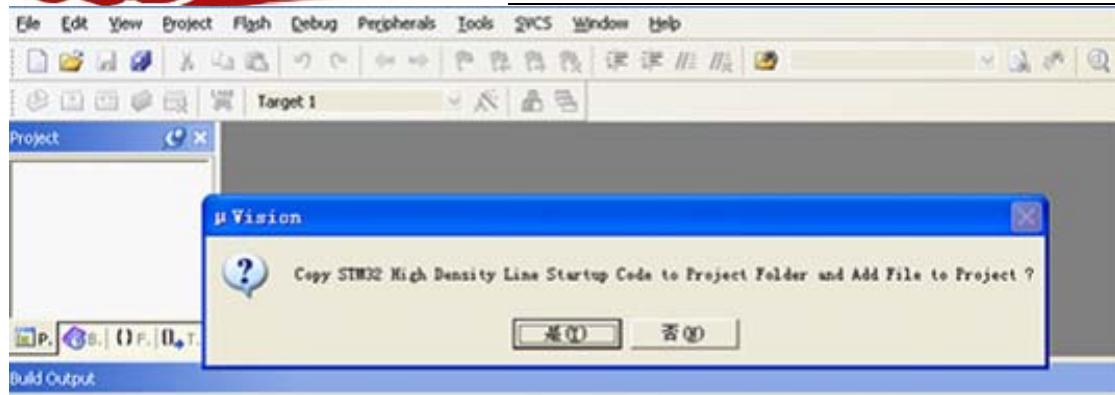
在工具栏 Project->New μVision Project...新建我们的工程文件，我们将新建的工程文件保存在桌面的“STM32 神舟 III 号开发板模板工程”（先在电脑 E 盘上新建一个“STM32 神舟 III 号开发板模板工程”，在该文件夹里面新建一个 Project 文件夹，当然你也可以在其它的盘上建，并不一定要在 E 盘），文件名取为神舟 STM32-DEMO（英文 DEMO 的意思是例子），名字可以随取，点击保存。



接下来的窗口是让我们选择公司跟芯片的型号，我们用 STM32 神舟 4 号的板子做举例说明，因为我们的 STM32 神舟 III 号用的芯片是 ST 公司的 STM32F103ZET6，有 64K SRAM, 512K Flash，属于高集成度的芯片。按如下选择即可。

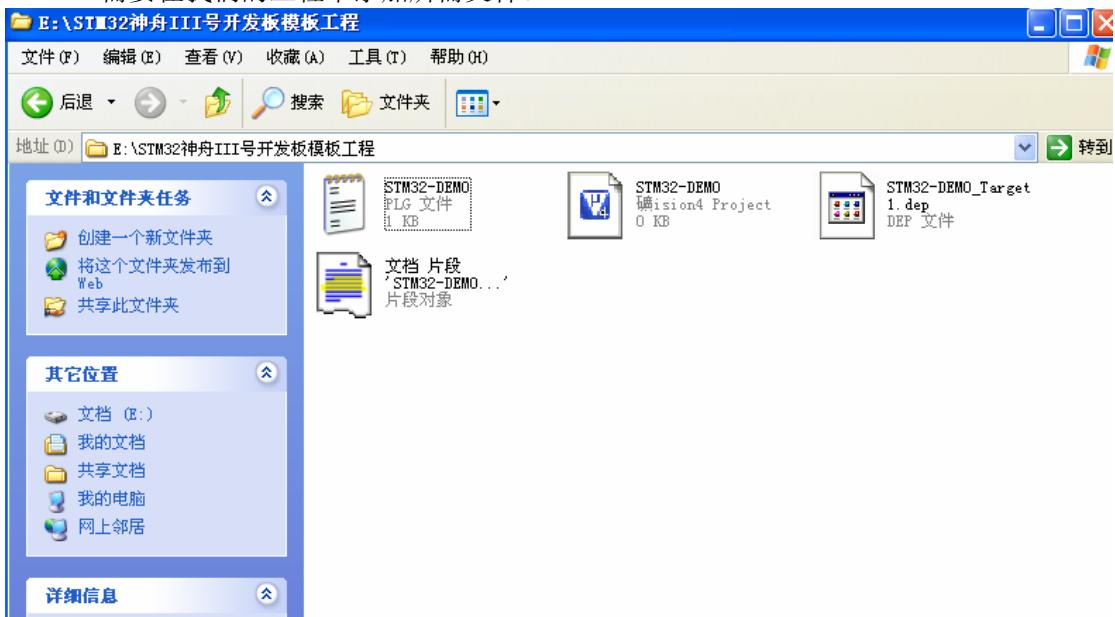


接下来的窗口问我们是否需要拷贝 STM32 的启动代码到工程文件中，这份启动代码在 M3 系列中都是适用的，一般情况下我们都点击是，但我们这里用的是 ST 的库，库文件里面也自带了这一份启动代码，所以为了保持库的完整性，我们就不要开发环境为我们自带的启动代码了，稍后我们自己手动添加，这里我们点击否。

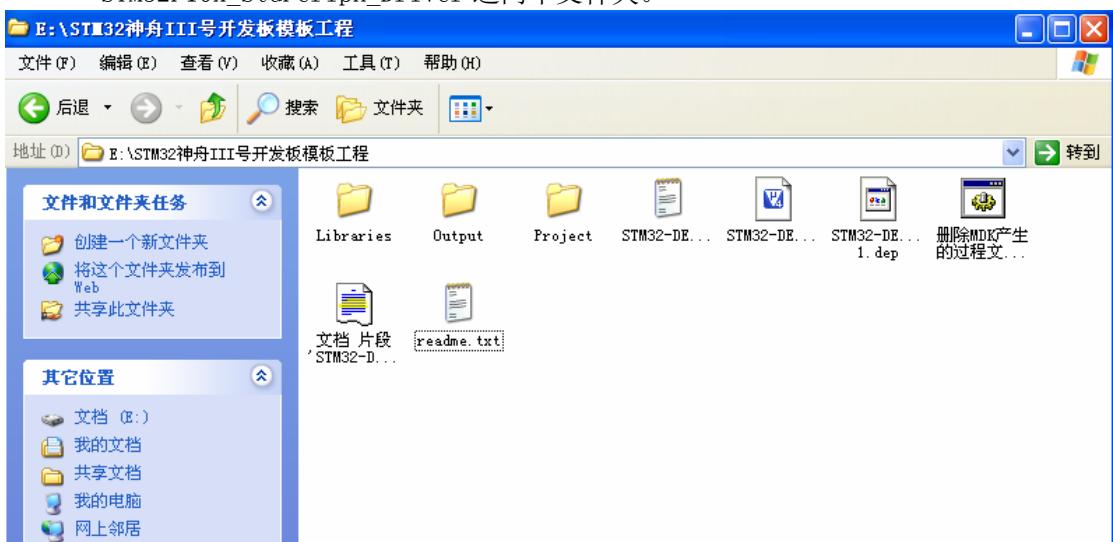


www.armjishu.com

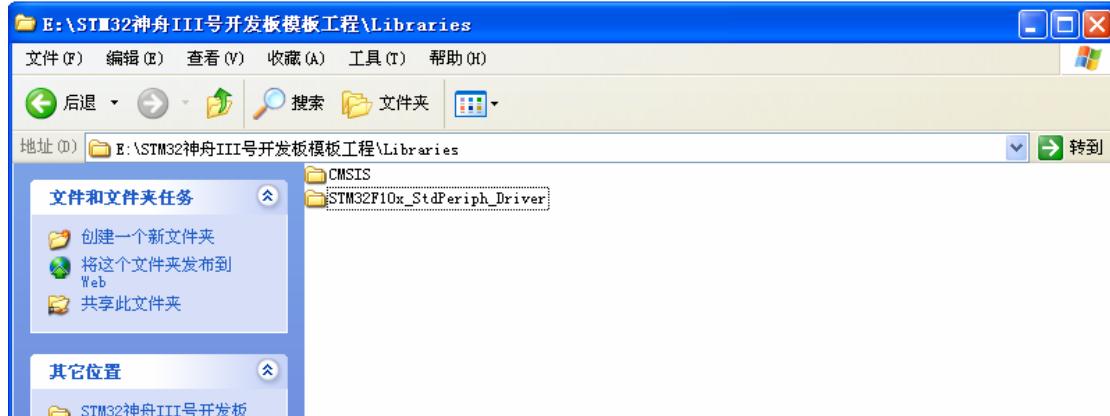
此时我们的工程新建成功，如下图所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。



在“STM32 神舟 III 号开发板模板工程”文件夹下，我们新建 3 个文件夹，分别为 Libraries、Output、Project 文件夹以及“删除 MDK 产生的过程文件.bat”文件、“readme.txt”文件和“stm32f10x_stdperiph_lib.zip”文件；原来新建的 Project 文件夹用来存放工程文件和用户代码，包括主函数 main.c；Libraries 用来存放 STM32 标准库里面的 CMSIS 和 STM32F10x_StdPeriph_Driver 这两个文件夹。



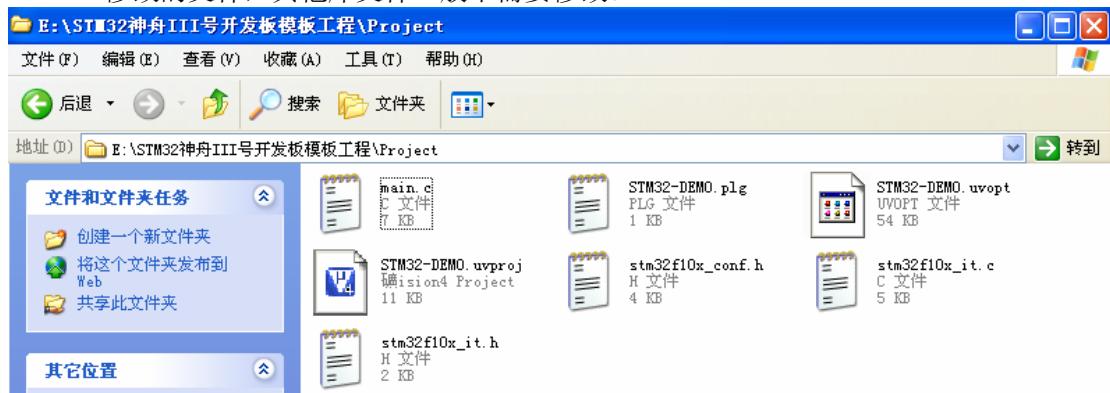
把从 ST 官网下载的 stm32f10x_stdperiph_lib.zip 库函数文件压缩包，解压缩后将 \stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries 的 CMSIS 跟 STM32F10x_StdPeriph_Driver 这两个文件夹拷贝到 STM32 神舟开发板模板工程\Libraries 文件夹中。



把标准库目录下的：

stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template 文件夹下的 main.c、stm32f10x_conf.h、stm32f10x_it.h、stm32f10x_it.c 拷贝到 STM32 神舟 III 号开发板模板工程\Project 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 Libraries\STM32F10x_StdPeriph_Driver\src 文件夹中，Libraries\STM32F10x_StdPeriph_Driver\inc 文件夹里面是它们的头文件。

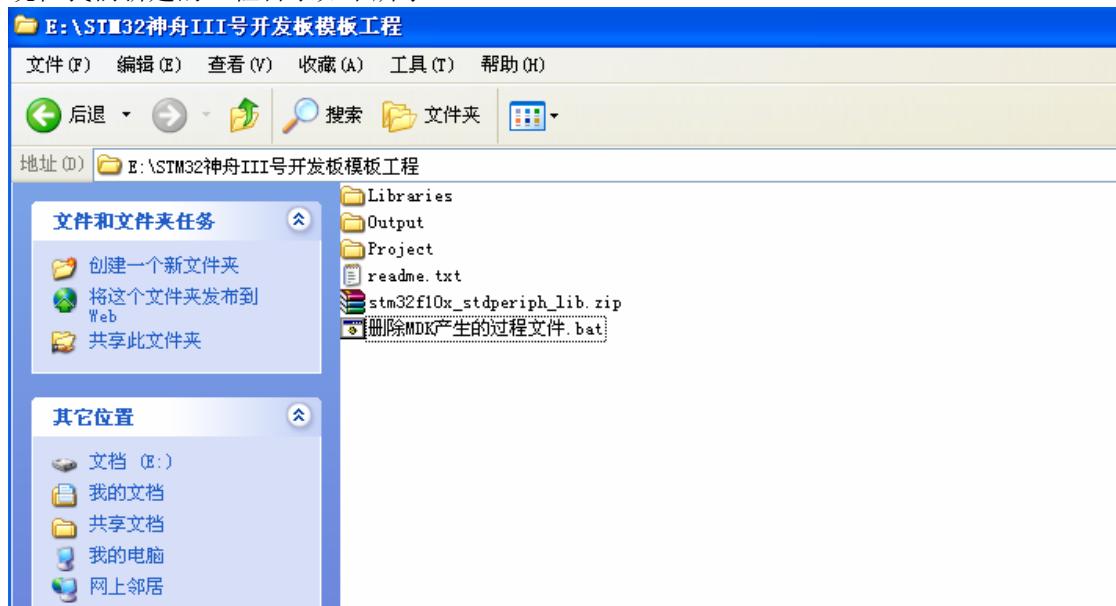
stm32f10x_it.h、stm32f10x_it.c 和 stm32f10x_conf.h 这三个文件是用户在编程时需要修改的文件，其他库文件一般不需要修改。



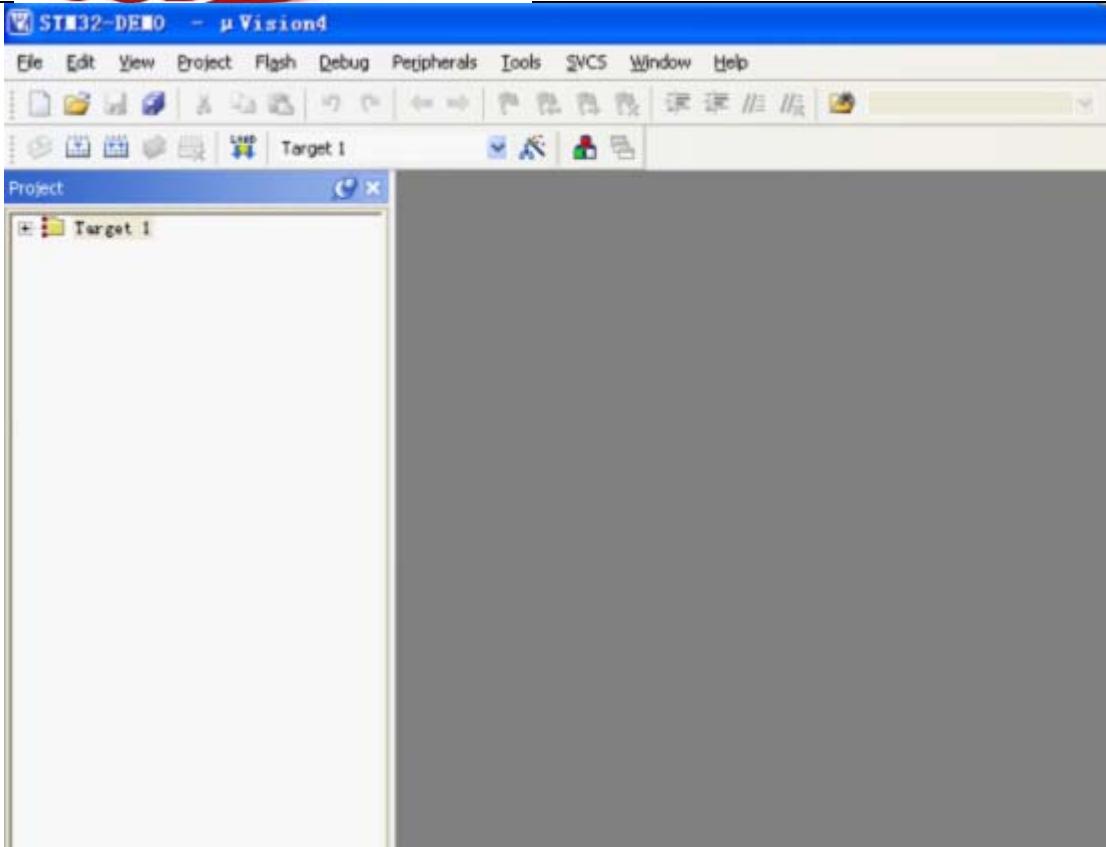
Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm 文件夹下是用汇编写的启动文件。STM32 神舟 III 号开发板用的 CPU 是 STM32F103ZET6，有 512K Flash，属于大容量的，所以等下我们把 startup_stm32f10x_hd.s 添加到我们的工程中。根据 ST 的官方资料：Flash 在 16~32 Kbytes 为小容量，64~128 Kbytes 为中容量，256~512 Kbytes 为大容量，不同大小的 Flash 对应的启动文件不一样，这点要注意。



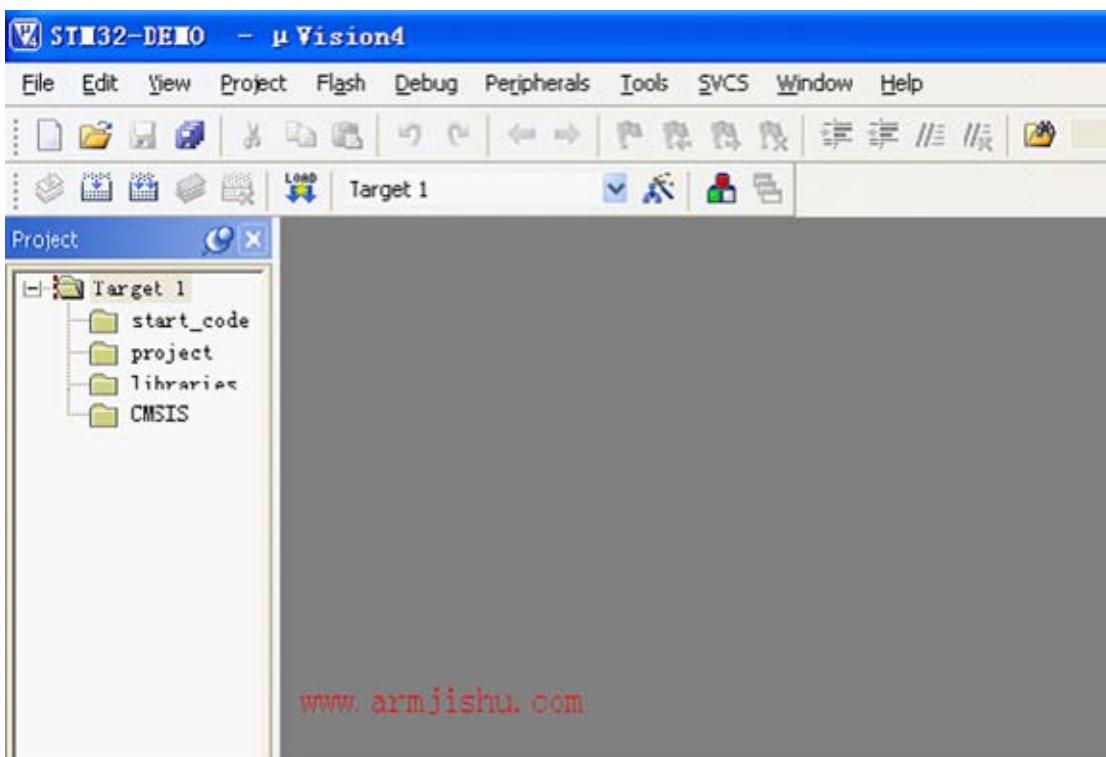
现在我们新建的工程目录如下所示：



回到我们刚刚新建的 MDK 工程中



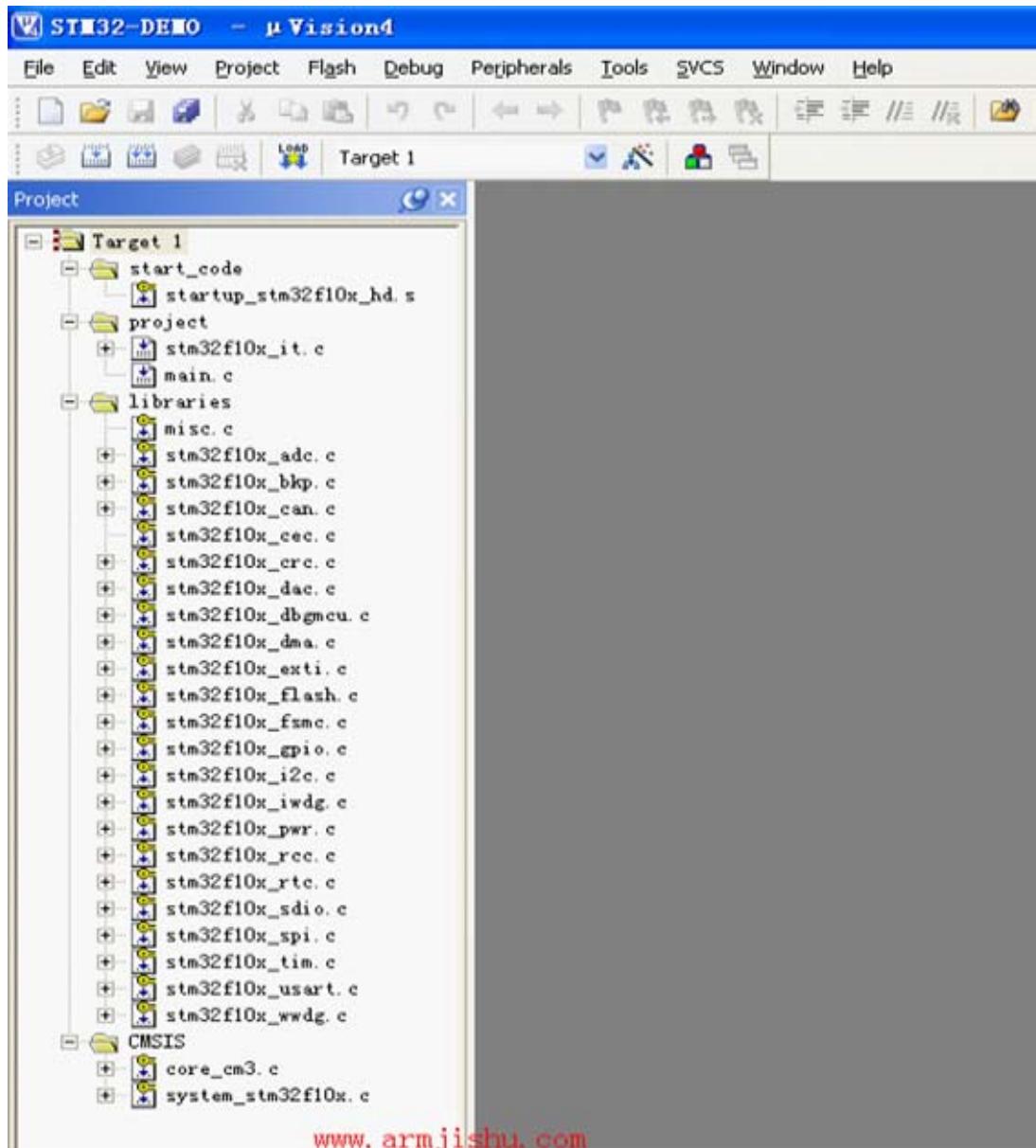
在 STM32-DEMO 上右键选中 Add Group... 选项，新建四个组，分别命名为 start_code、project、libraries、CMSIS。start_code 从名字就可以看得出我们是用它来放我们的启动代码的，project 用来存放用户自定义的应用程序，libraries 用来存放库文件，CMSIS 用来存放 M3 系列单片机通用的文件



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 STARTCODE 里面添加 startup_stm32f10x_hd.s，在 project 组里面添加 main.c 文件和 stm32f10x_it.c 这 2 个文件，在 libraries 组里面添加 “Libraries\STM32F10x_StdPeriph_Driver\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，。

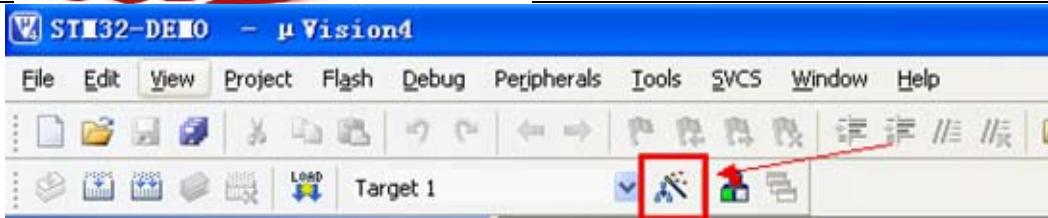
CMSIS 组我们在“Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x”路径下添加 system_stm32f10x.c 文件(system_stm32f10x.c 是 ARM 公司提供的符合 CMSIS 标准的库文件)；然后从 Libraries\CMSIS\CM3\CoreSupport 里添加 core_cm3.c；注意，这些组里面添加的都是汇编文件跟 C 文件，头文件是不需要添加的。最终效果如下图：



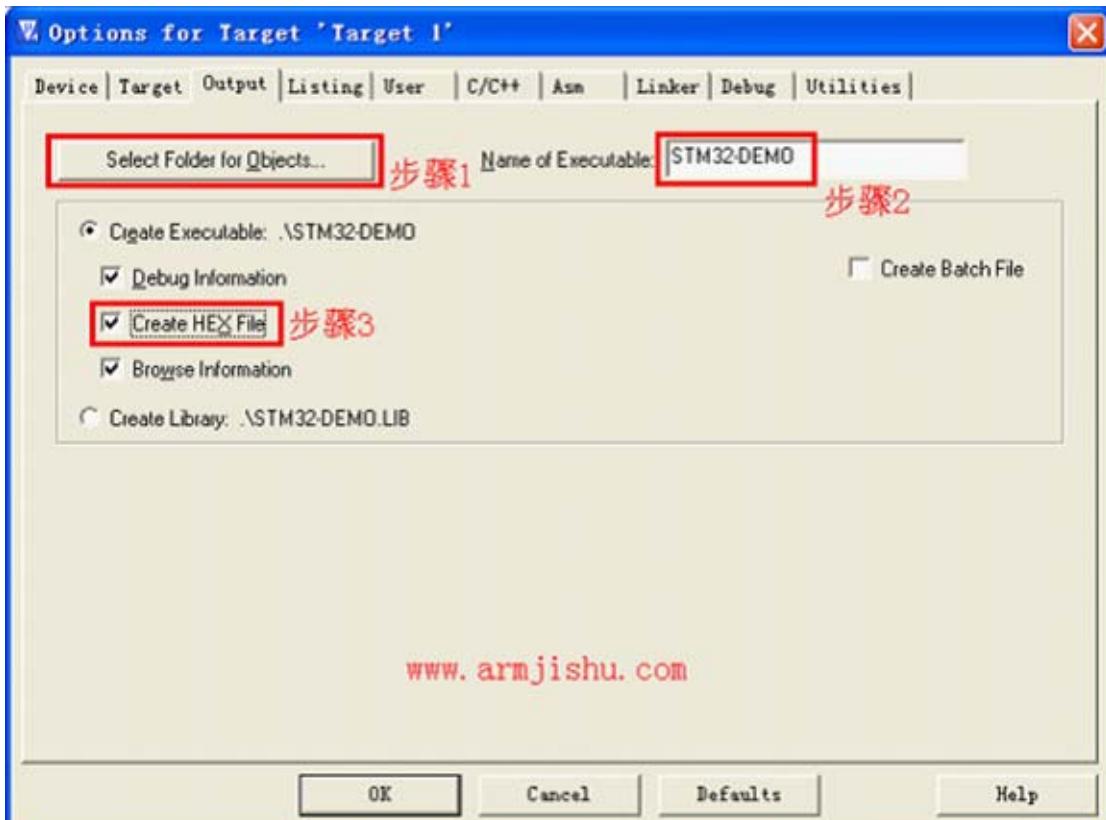
至于有些文件有个锁的图标，是因为这些都是库文件，不需要我们修改，属性为只读。

4.1.4 MDK环境设置

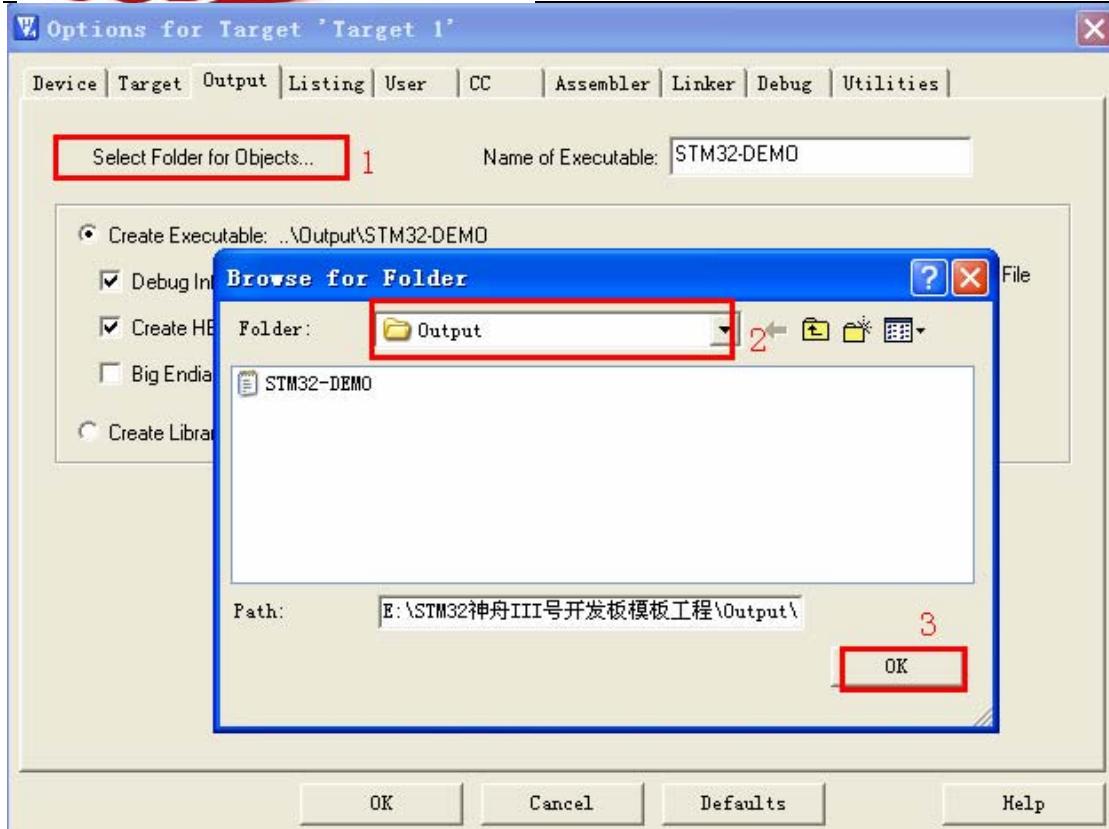
经过以上的一些步骤，我们的工程已经基本建好，下面来配置一下 MDK 的配置选项，点击工具栏中的魔术棒按钮，在弹出来的窗口中选中



然后选择：



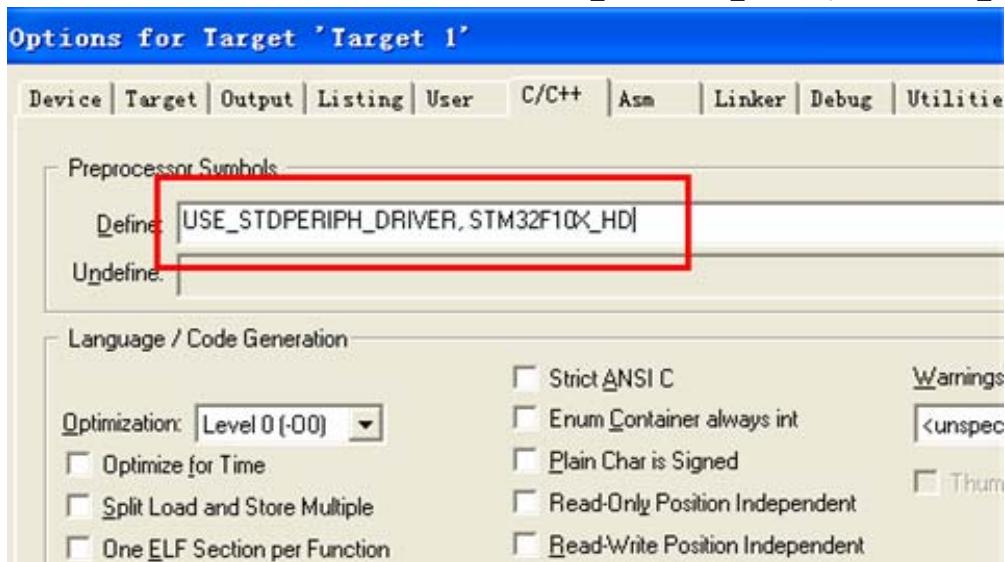
步骤 1：点击 Select Folder for Objects... 设置编译后输出文件保存的位置，我们选择 Output 文件夹。



步骤 2：把编译好的输出文件名定为 STM32-DEMO

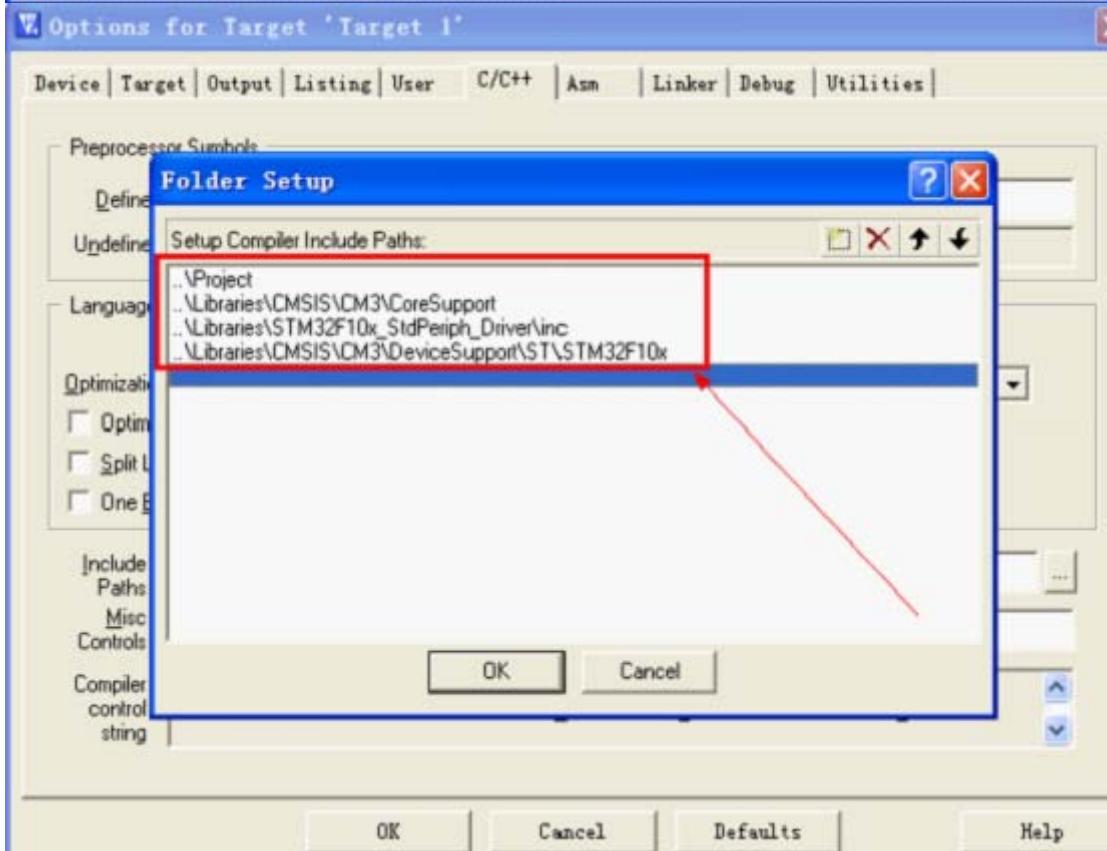
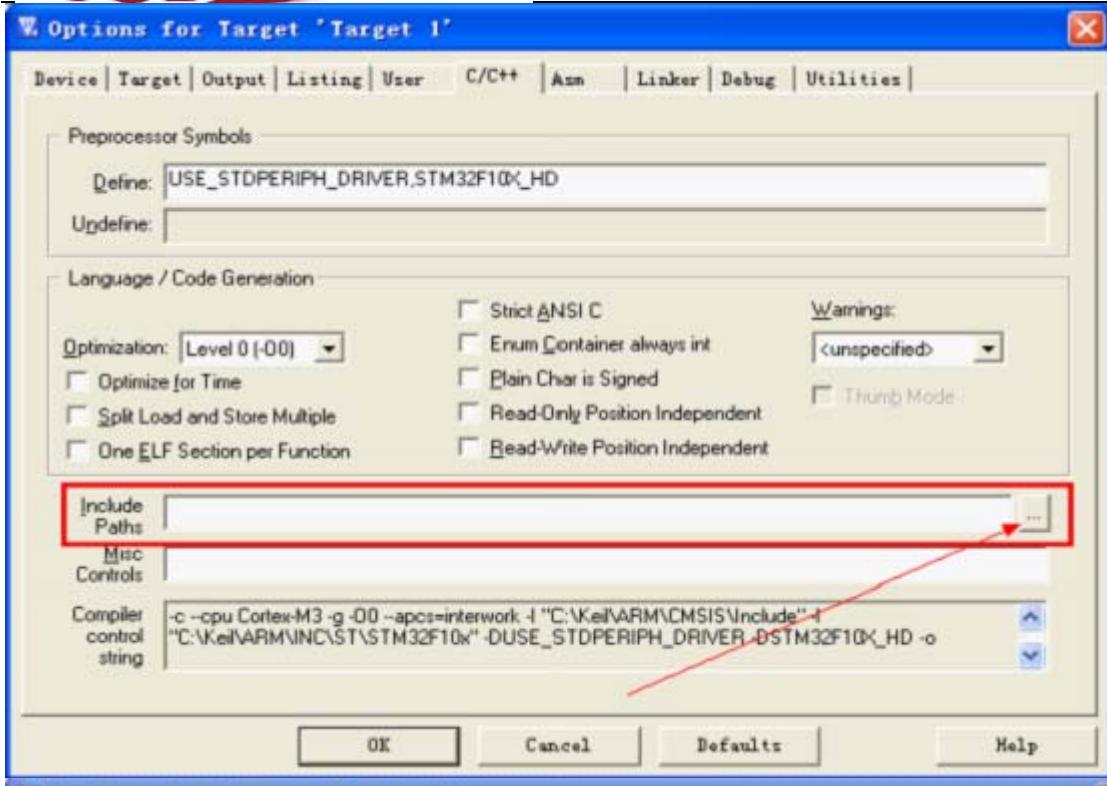
步骤 3：把 Create HEX File 这个选项框也选上，表示编译输出 HEX 文件

选中选项卡，在 Define 里面输入添加 USE_STDPERIPH_DRIVER, STM32F10X_HD。



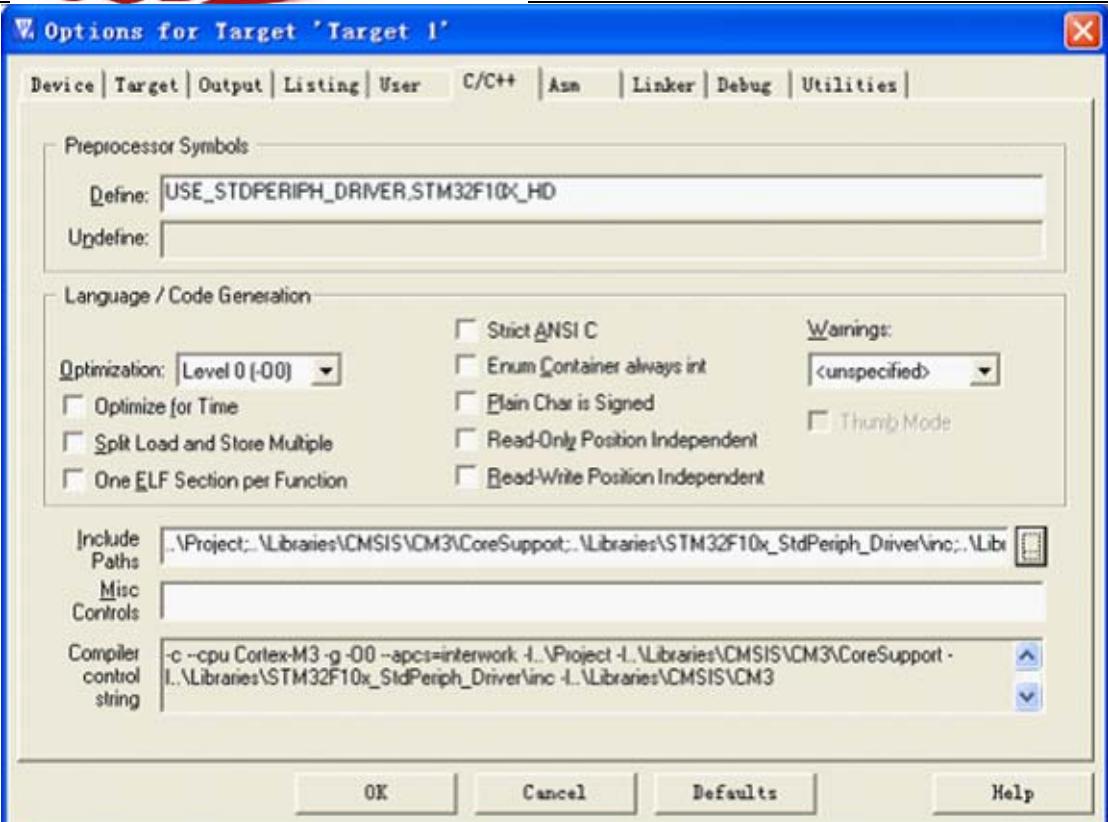
添加 USE_STDPERIPH_DRIVER 是为了屏蔽编译器的默认搜索路径，转而使用我们添加到工程中的 ST 的库，添加 STM32F10X_HD 是因为我们用的芯片是大容量的，添加了 STM32F10X_HD 这个宏之后，库文件里面为大容量定义的寄存器我们就可以用了。芯片是小或中容量的时候宏要换成 STM32F10X_LD 或者 STM32F10X_MD。其实不管是什么容量的，我们只要添加上 STM32F10X_HD 这个宏即可，当你用小或者中容量的芯片时，那些为大容量定义的寄存器我不去访问就是了，反正也访问不了。

在 Include Paths 栏点击 ，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 ANSI C 的库文件，如 stdio.h 、 stdio.h。

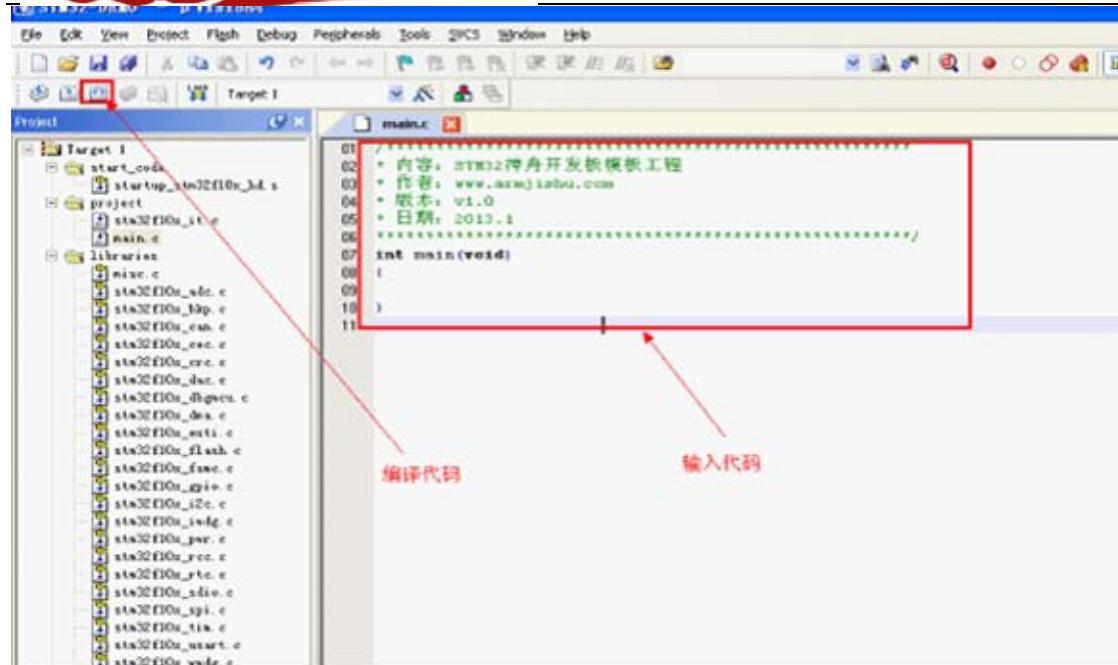
库文件路径修改成功之后如下所示：



至此，我们的工程模板就建成了。学会新建工程，是学习stm32的第一步。

1.4.4 在 main.c 里输入代码，并保存，然后编译代码

```
*****  
* 内容: STM32 神舟开发板模板工程  
* 作者: www.armjishu.com  
* 版本: v1.0  
* 日期: 2013.1  
*****  
int main(void)  
{  
}
```

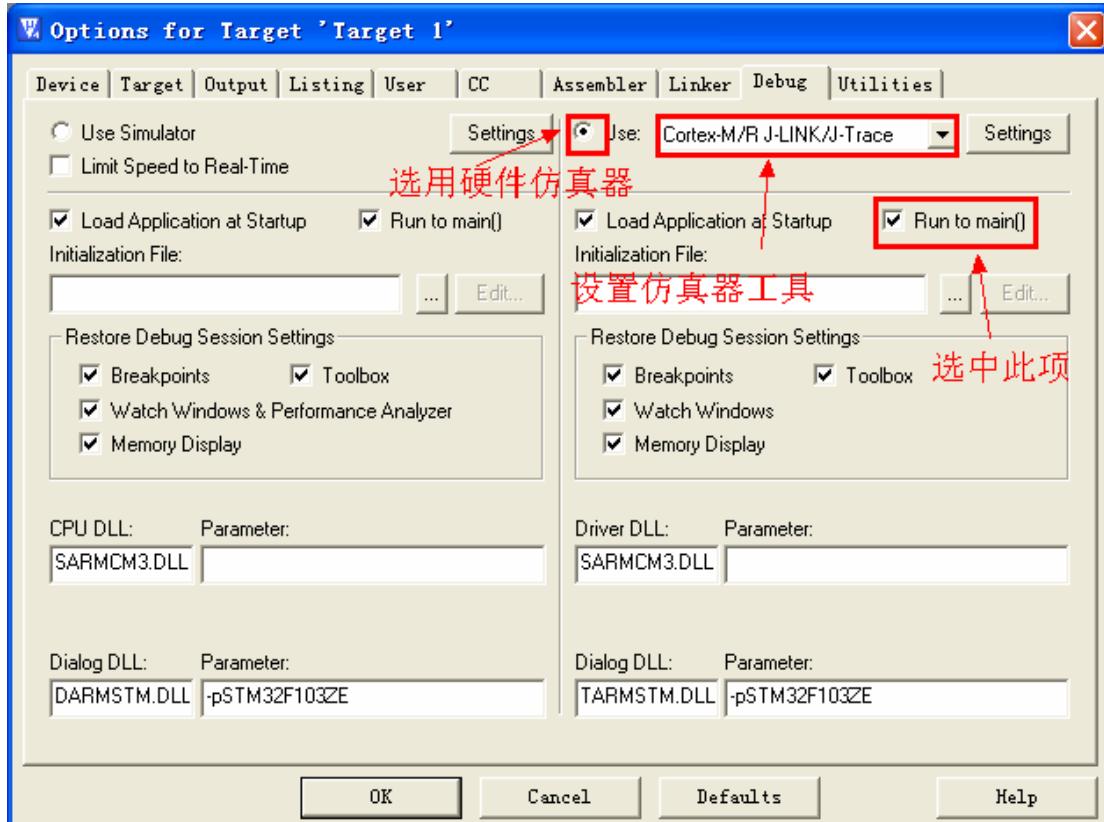


可以看到编译成功，最后产生了 HEX 文件，我们的模板搭建成功！

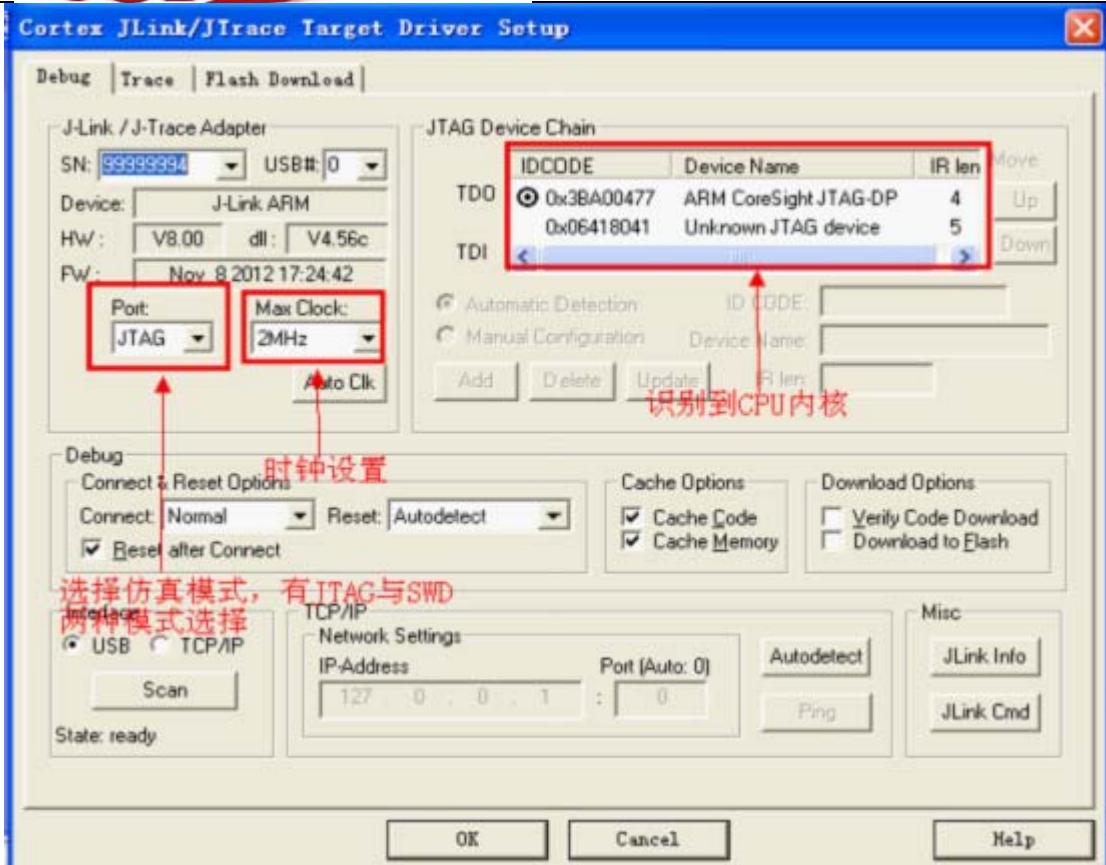
4.1.5 使用JLINK V8仿真器硬件调试配置

这个工程默认的是软件仿真，如果开发板要用 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：点击 ，在 Debug 选项里

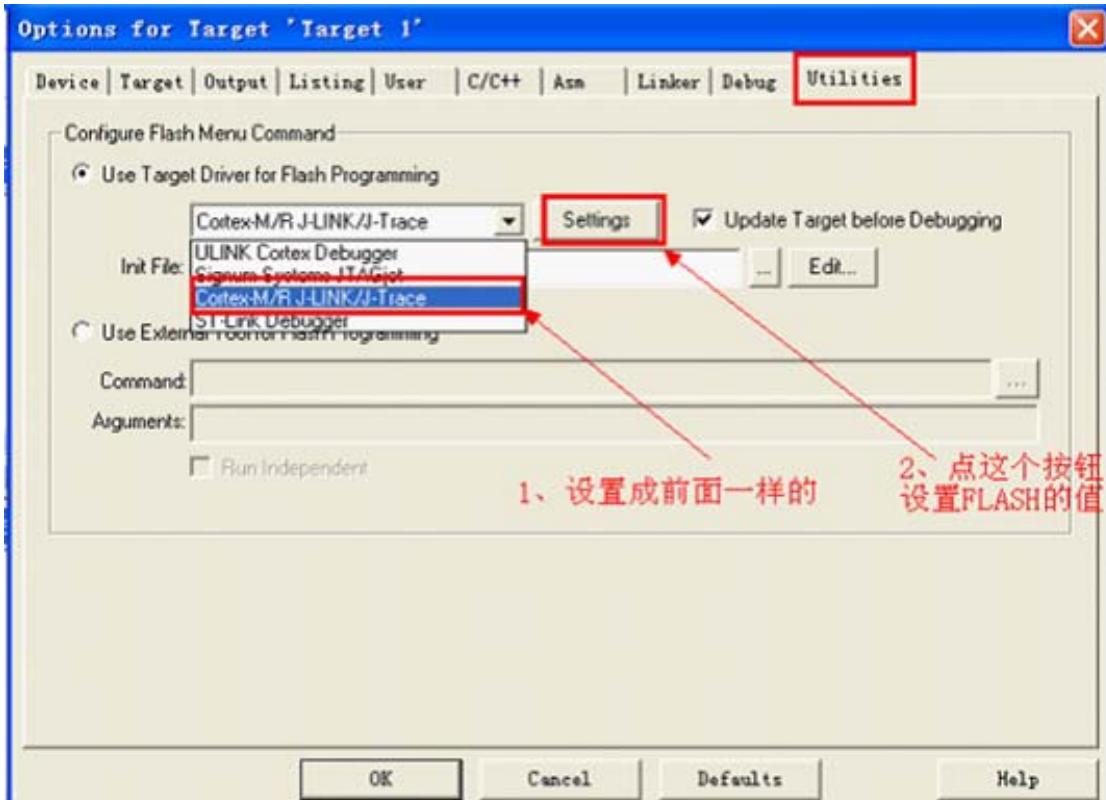


点 Settings 按钮，查看是否识别到了目标板 CPU（注意，此处目标板应该上电，并将 JLINK V8 与模板连接好，JLINK V8 也需要上电）

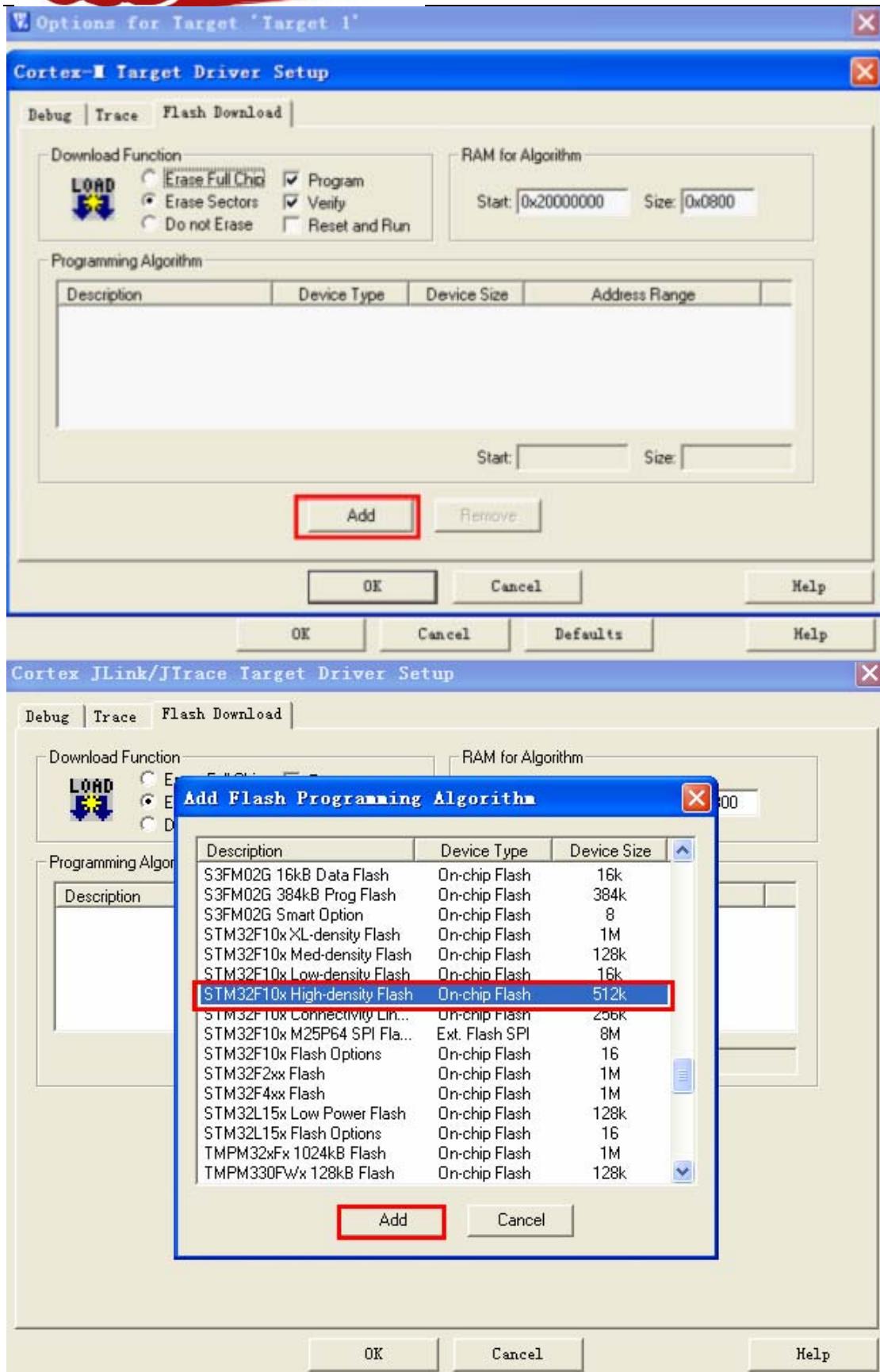


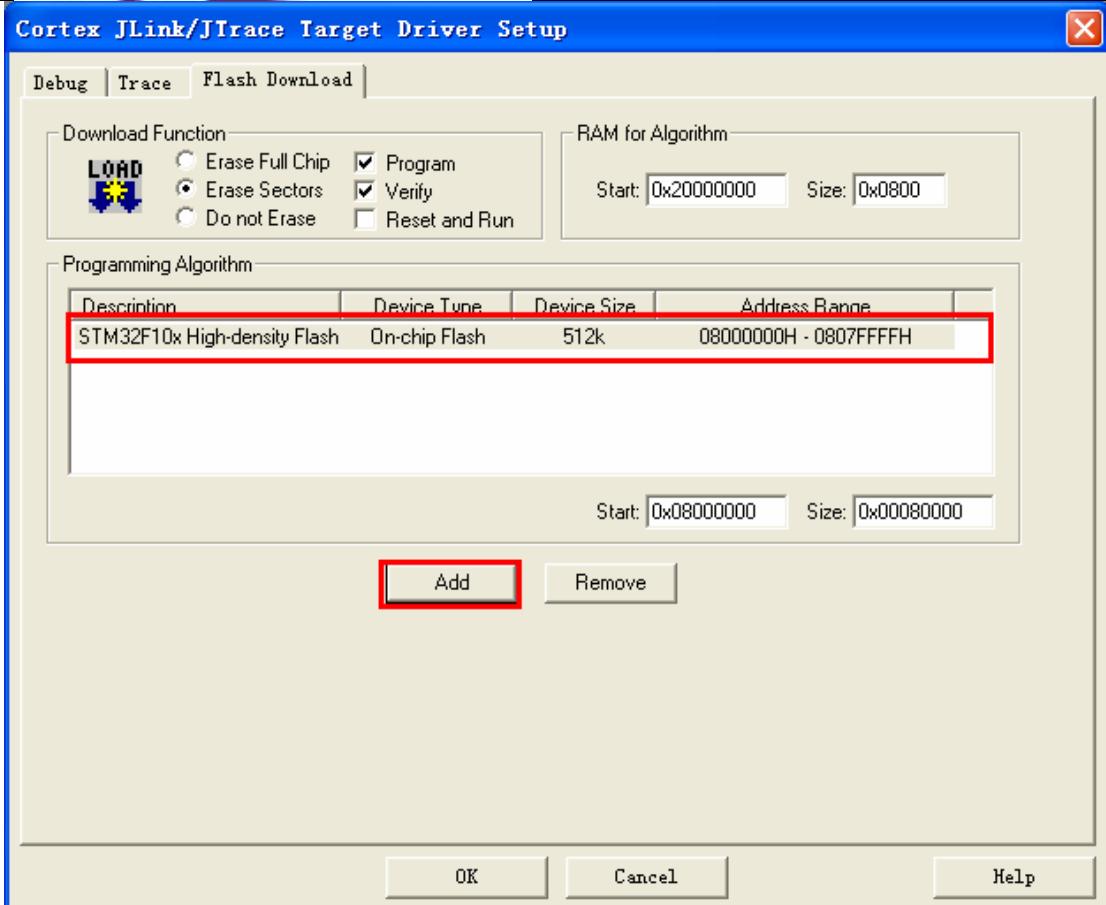
设置完点确定

下面选择 Utilities 选项卡



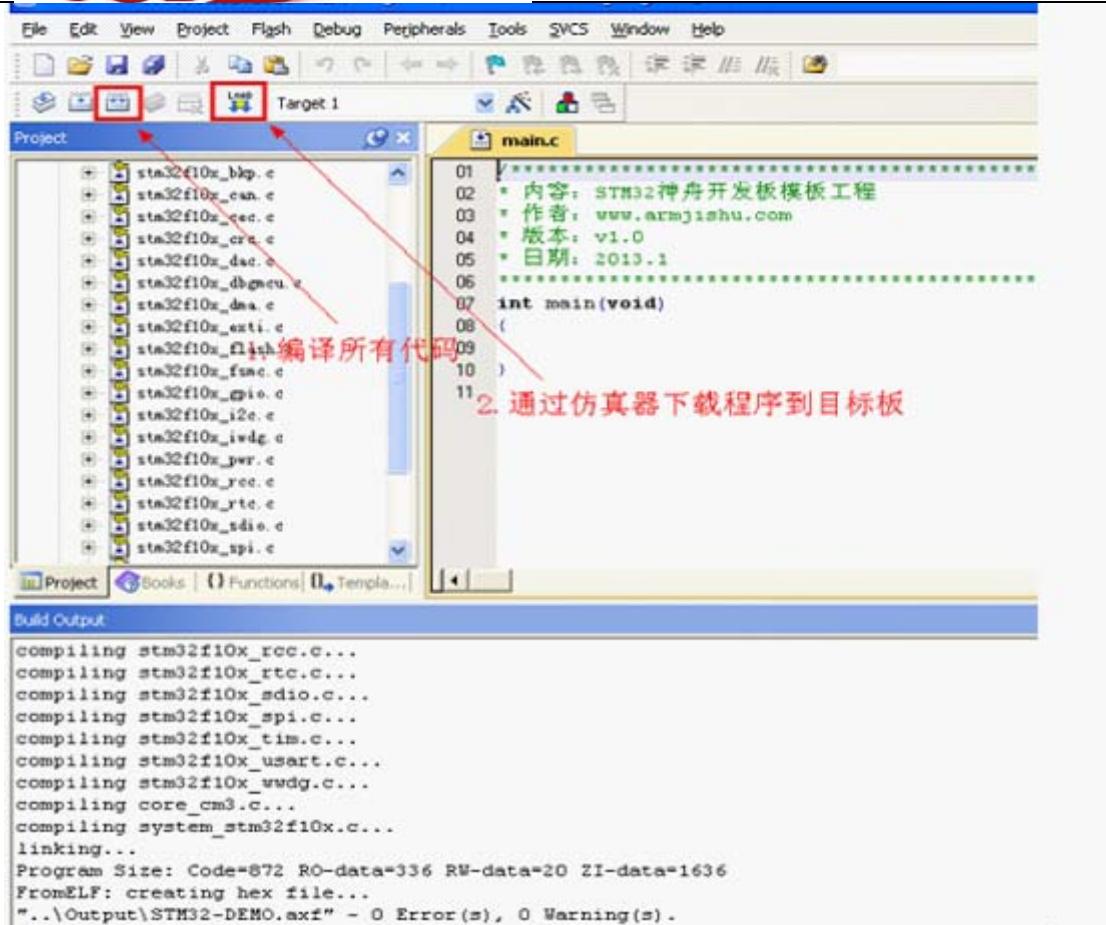
在选项卡 Utilities\Setting\Flash download 中我们设置成如下：





因为 STM32F103ZET6 芯片的内部 ROM 大小是 512K 点击 OK 按钮

编译全部代码，然后点 LOAD 下载程序到目标板



到了这里就算是大功告成了。如果在新建工程中遇到什么问题，先不要急，可先参考 STM32 神舟系列的其他相关资料。

4.2 理解芯片控制的原理

如果说要做单片机很难吗？其实并不难，用 3 句话就可以讲明白：

第 1 句话：芯片管脚不是输入，就是输出。

我们所有的程序，用单片机控制的产品，以及外设，无非就是控制芯片的各个管脚输入或者输出两个状态；例如，芯片发送数据就是输出；芯片驱动一个产品，也是输出；芯片接收数据就是输入；单片机对一个存储芯片写输入，可以理解为单片机与存储芯片连接的管脚输出状态，输出数据到存储芯片的管脚上，而存储芯片此时它的芯片对应管脚被配置成输入，将数据写入到芯片内部。

所以说，芯片管脚不是输入，就是输出，当然，如果你不使用这个管脚，也可以将它配置成某一种中间状态，免得干扰了外界，影响了 PCB 板上的其他元器件状态。

第 2 句话：芯片管脚不是高电平，就是低电平。

芯片管脚不是高电平就是低电平两种状态，当然也有第三种，既不高电平也不是低电平的状态，这样的管脚状态表示没有任何内容和数据；无论管脚是输入还是输出，它的目的都是传输数据、传输信息，所以管脚的高电平我们将它表示为“1”，低电平表示“0”，通过 0 和 1 这样的数据来传输它想传输的内容，这个就是所谓的二进制。

例如：假如复位芯片管脚是低电平进行复位，我们将该管脚一直拉高为高电平“1”的时候，芯片可以正常工作，如果将管脚拉低至低电平“0”的时候，芯片通过检测这个管脚状态为低电平，芯片内部就会自动进行复位；我们通过控制这个管脚拉高和拉低，从而就可以达到控制芯片的工作；其他的管脚也是同样的道理。

第3句话：传输协议。

什么是传输协议，比如与串口芯片通信，那么就要是串口协议的；如果是 I2C 协议的 EERPOM，那么就是 I2C 协议；还有其他一些比如 485 协议，CAN 协议，USB 协议，SD 卡的 SDIO 协议……等等数不胜数。

而这些协议，无非就是按照预先规定的表达方式进行通信，比如举个例子，我约定先连续发 4 个 1，然后再发 4 个 0，就表示芯片 A 要开始发数据给芯片 B 了，即芯片 A 通过它的芯片管脚发‘11110000’给到芯片 B 的时候，那么芯片 B 就知道芯片 A 要给它真正的数据，它就要做好准备工作，准备好之后，芯片 B 就会给芯片 A 一个回应，当芯片 A 收到芯片 B 的回应，就正式开始发数据。

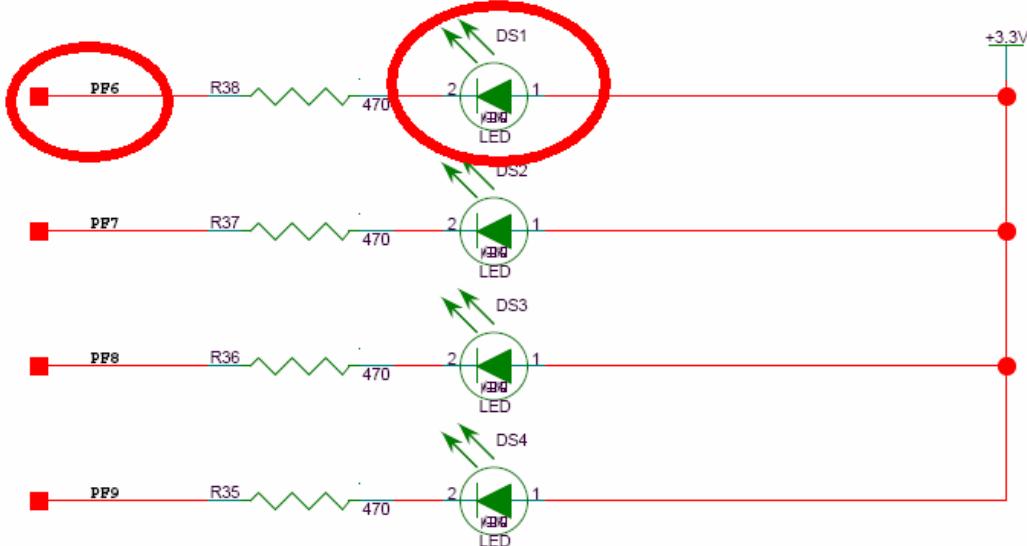
这样通信双方之间的协商规定，就构成了协议，经过这么多年，就形成了我们所常见的串口协议，CAN 协议，USB 协议（像 USB 协议又分为 USB1.0 协议，USB2.0 协议，USB3.0 协议，版本越高，速度就越快，协议进行优化后，通信效率也变高了）。

不知道大家理解了没有呢？所以总结下来，一个芯片最简单的外设莫过于 I/O 口的高低电平控制，我们这里将详细讲解一下如何用一个 I/O 口去控制一个 LED 灯的亮灭。

4.3 芯片管脚控制LED灯原理图解释

下面有个原理图，这个原理图是用 STM32 的 PF6 管脚连 LED 灯的负极，用正 3.3V 电源端连 LED 灯的正极，再串联一个限流电阻限制电流（电阻的作用就是限流、降压；如果线路上电阻很小，那么电压不变的情况下，电流就会变得很大，有可能会烧坏 LED 灯，所以这里我们串联一个电阻进行降压）降压防止 LED 灯被烧掉，这个串联电阻的阻值要计算好，使得在恒定电压的情况下，电流的大小刚好足够驱动 LED 灯点亮，点亮这个 LED 灯大概需要 10ma~20ma（毫安）的电流。

原理图中 STM32 的 PF6 管脚如何控制这个 LED1 的灯呢？可以看到，PF6 输出高电平的时候，LED 灯不会亮；只有当 PF6 输出低电平的时候，LED 灯才会点亮。所以我们想用 STM32 的管脚 PF6 去驱动 LED1 这个 LED 灯亮，只要使得 PF6 输出低电平就可以，这样就知道如何控制这个 LED 灯了。



为什么这么接呢？为什么不让 PF6 管脚接 LED 的正极，而 LED 灯的负极去接 GND 地呢？这样才是最常规的接法不对吗？答案是当然是，但是在这种接法有助于芯片的长久使用，芯片的总体驱动能力是有限的，它可以驱动一个 LED 灯，但驱动不了 100 个，1000 个。

在这里需要重复上面已经说过的内容，首先我们要知道 LED 的发光工作条件，不同的 LED 其

额定电压和额定电流不同，一般而言，红或绿颜色的 LED 的工作电压为 1.7V~2.4V，蓝或白颜色的 LED 工作电压为 2.7~4.2V，直径为 3mm LED 的工作电流 2mA~10mA，在这里采用绿色的 LED；STM32 单片机（如本实验板中所使用的 STM32F103ZET 芯片）的 I/O 口作为输出口时，向外输出电流的能力是 25mA 左右，勉强是可以点亮一个发光二极管，但是如果我们用 STM32 去点亮很多个 LED 灯的时候，就有可能造成芯片本身输出电流不足（因为芯片能输出的总电流大小是恒定的）。

其次，PF6 的这种接法是一种灌电流（要 VCC 往内输入电流）的方式，这种方式使得 STM32 的芯片管脚让一个 LED 灯亮非常轻松，利用灌电流的方式驱动发光二极管是比较常见的一种用法，无论接多少 LED，芯片管脚的负荷都非常轻。当然，现今的一些增强型单片机，是采用拉电流输出的，只要单片机的输出电流能力足够强即可，不过接多了也是不可取的，单片机的总体驱动电流是有限的；上图中的电阻用的是 1K 阻值主要为了限制电流，让发光二极管的工作电流限定在 2mA~10mA。

4.4 STM32 相关的芯片手册有哪些？我们如何阅读这些资料

STM32 神舟 III 号开发板的主芯片 STM32F103ZE 芯片相关的资料和例程都可以在 ST 的官方网站上找到，STM32F103ZET6 芯片的中文网址为“

<http://www.st.com/cn/mcu/product/164487.jsp>”。

首先介绍两个最重要的两个文档，芯片手册和参考手册。

1、芯片手册。芯片手册在网页“DATASHEET”那一栏。芯片手册详细介绍了所选择的芯片型号的功能规格，内核型号，运行主频，外设资源以及其性能，芯片封装种类信息以及各种封装的管脚定义，芯片的电气特性，外设的时序要求，订购信息和器件的机械特。这份文档因为芯片型号的不同而不同，比如 STM32F103ZE 和 STM32F103RB 和 STM32F105 以及 STM32F107 等等，他们的功能外设资源不同，所以芯片手册都不相同。在芯片的选型阶段，这份文档是判断芯片功能和性能是否满足项目需求的关键文档。在原理图阶段这份文档更是尤为重要。后续开发调试阶段这份文档也不可或缺。这份文档将伴随着你从项目开始一直走到项目结束。

2、参考手册。参考手册在网页“REFERENCE MANUALS”那一栏，也称为技术参考手册。这份参考手册是有关如何使用该产品的具体信息，包含各个功能模块的内部结构、所有可能的功能描述、各种工作模式的使用和寄存器配置等详细信息。参考手册不涉及某个具体的芯片，他是将一个系列的芯片。STM32F103ZE 属于 STM32F10x 这一大类，所以我们下载的文档名为“RM0008：STM32F101xx, STM32F102xx、STM32F103xx、STM32F105xx 和 STM32F107xx，ARM 内核 32 位高性能微控制器”。也就是说无论你在 STM32F103RB 和 STM32F103ZE 和 STM32F105 以及 STM32F107 等芯片的网页页面下载的参考手册都是相同的。这也注定这份文档介绍的功能资源是 STM32F10x 这一大类芯片所有功能资源的交集。这份文档包含了 USB 接口和以太网接口的介绍，但并不表示 STM32F103RB 包含这些接口。

这两份文档都很重要，相对来说，硬件开发人员更多关注芯片手册，软件开发人员更多关注参考手册。

其次介绍芯片相关的一些文档资料。

1、处理器内核相关文档。STM32F103ZE 芯片的性能是 ARM 公司的 Cortex-M3。所以如果需要了解内核的资料，可以参考 ARM 公司的“Cortex™-M3 技术参考手册”以及其他 Cortex-M3 的技术书籍，例如“ARM Cortex-M3 权威指南”等等。

2、ST 的应用笔记“APPLICATION NOTES”。充分利用 ST 网页中的资源，可以加快产品设计调试进度。

3、外设资源相关资料。例如 TIM 定时器、UART 串口以及 USB 等等。这些接口的资料可以参考 ST 网页中的 STM32 应用文档以及示例程序。由于很多接口都是各自的协议标准，所以可以查阅这些标准协议的相关资料文献，例如 I2C、SPI、USB 这些都有各自的规范可以查阅。

4、相关外部芯片资料。STM32 的接口对外连接了什么器件，就需要查阅相关的文档资料。同样是 I2C 接口，既可以连接 EEPROM 也可以连接温度传感器或其他；同样是 SPI 接口，既可以连接 DATA FLASH 也可以连接 WIFI 模块或者触摸屏等等。这些已经不属于 STM32 的范围了，所以这块资料在 STM32 的网页一般找不到，或者只能找到对应的参考设计。

5、STM32 神舟系列用户手册。

6、www.armjishu.com 论坛帖子。

7、STM32 神舟系列开发板博客: <http://blog.sina.com.cn/u/1989261580>

8、STM32 神舟系列开发板微博: <http://weibo.com/u/1989261580>

4.5 STM32芯片各个管脚是怎么控制以及被管理的？（如何阅读芯片手册）

那么我们怎么来访问这些寄存器呢？

如果是通过标准库访问，只要调用相关函数即可。

如果是寄存器操作，首先要得到寄存器的地址，寄存器的地址是芯片厂商一开始就定好了的，固定的不能改变，大家看下图：

| | |
|-----------------|---------------------------|
| Reserved | 0x5000 0400 - 0x5FFF FFFF |
| USB OTG FS | 0x5000 0000 - 0x5003 FFFF |
| Reserved | 0x4003 0000 - 0x4FFF FFFF |
| Ethernet | 0x4002 8000 - 0x4002 9FFF |
| Reserved | 0x4002 3400 - 0x4002 7FFF |
| CRC | 0x4002 3000 - 0x4002 33FF |
| Reserved | 0x4002 2400 - 0x4002 2FFF |
| Flash interface | 0x4002 2000 - 0x4002 23FF |
| Reserved | 0x4002 1400 - 0x4002 1FFF |
| RCC | 0x4002 1000 - 0x4002 13FF |
| Reserved | 0x4002 0800 - 0x4002 0FFF |
| DMA2 | 0x4002 0400 - 0x4002 07FF |
| DMA1 | 0x4002 0000 - 0x4002 03FF |
| Reserved | 0x4001 3C00 - 0x4001 FFFF |
| USART1 | 0x4001 3800 - 0x4001 3BFF |
| Reserved | 0x4001 3400 - 0x4001 37FF |
| SPI1 | 0x4001 3000 - 0x4001 33FF |
| TIM1 | 0x4001 2C00 - 0x4001 2FFF |
| ADC2 | 0x4001 2800 - 0x4001 2BFF |
| ADC1 | 0x4001 2400 - 0x4001 27FF |
| Reserved | 0x4001 1C00 - 0x4001 23FF |
| Port E | 0x4001 1800 - 0x4001 1BFF |
| Port D | 0x4001 1400 - 0x4001 17FF |
| Port C | 0x4001 1000 - 0x4001 13FF |
| Port B | 0x4001 0C00 - 0x4001 0FFF |

| | |
|-----------|---------------------------|
| Port A | 0x4001 0800 - 0x4001 0BFF |
| EXTI | 0x4001 0400 - 0x4001 07FF |
| AFIO | 0x4001 0000 - 0x4001 3FFF |
| Reserved | 0x4000 7800 - 0x4000 FFFF |
| DAC | 0x4000 7400 - 0x4000 77FF |
| PWR | 0x4000 7000 - 0x4000 73FF |
| BKP | 0x4000 6C00 - 0x4000 6FFF |
| bxCAN2 | 0x4000 6800 - 0x4000 6BFF |
| bxCAN1 | 0x4000 6400 - 0x4000 67FF |
| Reserved | 0x4000 5C00 - 0x4000 63FF |
| I2C2 | 0x4000 5800 - 0x4000 5BFF |
| I2C1 | 0x4000 5400 - 0x4000 57FF |
| UART5 | 0x4000 5000 - 0x4000 53FF |
| UART4 | 0x4000 4C00 - 0x4000 4FFF |
| USART3 | 0x4000 4800 - 0x4000 4BFF |
| USART2 | 0x4000 4400 - 0x4000 47FF |
| Reserved | 0x4000 4000 - 0x4000 43FF |
| SPI3/I2S3 | 0x4000 3C00 - 0x4000 3FFF |
| SPI2/I2S2 | 0x4000 3800 - 0x4000 3BFF |
| Reserved | 0x4000 3400 - 0x4000 37FF |
| IWDG | 0x4000 3000 - 0x4000 33FF |
| WWDG | 0x4000 2C00 - 0x4000 2FFF |
| RTC | 0x4000 2800 - 0x4000 2BFF |
| Reserved | 0x4000 1800 - 0x4000 27FF |
| TIM7 | 0x4000 1400 - 0x4000 17FF |
| TIM6 | 0x4000 1000 - 0x4000 13FF |
| TIM5 | 0x4000 0C00 - 0x4000 0FFF |
| TIM4 | 0x4000 0800 - 0x4000 0BFF |
| TIM3 | 0x4000 0400 - 0x4000 07FF |
| TIM2 | 0x4000 0000 - 0x4000 03FF |

寄存器的地址是基址+偏移量的和。基址在芯片数据手册“存储器映像”章节，正如上图那样，比如 Port A 的基址是’0x4001 0800’；比如 Port B 的基址是’0x4001 0c00’；比如 SPI1 的基址是’0x4001 3000’；比如 USART3 的基址是’0x4000 4800’；以此类推。

从芯片上电后工作正常开始，所有寄存器都会有一个默认数值，保证处理器处于确定的状态。这个初始值正如上图所描述的这样，这个重要内容我们可以在技术参考手册中获得。

现在我们打开参考手册《STM32F1 中文参考手册》

【中文】 STM32F 系列 ARM 内核 32 位高性能微控制器参考手册 V10_1 , 可以看到这是一个 754 页的参考手册，我们截图看看：

STM32F10xxx 参考手册



参考手册

STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx 和 STM32F107xx, ARM 内核 32 位高性能微控制器

导言

本参考手册针对应用开发，提供关于如何使用 STM32F101xx、STM32F102xx、STM32F103 和 STM32F105xx/STM32F107xx 微控制器的存储器和外设的详细信息。在本参考手册中 STM32F101xx、STM32F102xx、STM32F103 和 STM32F105xx/STM32F107xx 被统称为 STM32F10xxx。

STM32F10xxx 系列拥有不同的存储器容量、封装和外设配置。

关于订货编号、电气和物理性能参数，请参考小容量、中容量和大容量的 STM32F101xx 和 STM32F103xx 的数据手册，小容量和中容量的 STM32F102xx 数据手册和 STM32F105xx/STM32F107xx 互连型产品的数据手册。

关于芯片内部闪存的编程，擦除和保护操作，请参考 [STM32F10xxx 闪存编程手册](#)。

关于 ARM Cortex™-M3 内核的具体信息，请参考 [Cortex™-M3 技术参考手册](#)。

比如我们要访问 GPIO 管脚 PD2，那么首先就要找到端口 D 的位置，即 Port D，我们首先查看文档的目录，找到第 8.2 节 GPIO 寄存器描述章节，可以看到从 113 页开始到 116 页都是描述 GPIO 端口的，我们查看 8.2.1 节查看一下端口配置低寄存器 (GPIOx_CRL)，这个里的 x=A..E 表示，GPIOx_CRL 可以是 GPIOA_CRL、GPIOB_CRL、GPIOC_CRL、GPIOD_CRL、GPIOE_CRL 中的任意一个；在这里我只关心端口 D，就可以把它看成 GPIOD_CRL：

| | |
|--|-----|
| 8.2 GPIO 寄存器描述 | 113 |
| 8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E) | 113 |
| 8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E) | 114 |
| 8.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E) | 114 |
| 8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) | 115 |
| 8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) | 115 |
| 8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E) | 115 |
| 8.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) | 116 |
| 8.3 复用功能 I/O 和调试配置(AFIQ) | 116 |
| 8.3.1 把 OSC32_IN/OSC32_OUT 作为 GPIO 端口 PC14/PC15 | 116 |
| 8.3.2 把 OSC_IN/OSC_OUT 引脚作为 GPIO 端口 PD0/PD1 | 117 |

我们可以找到 PortD 在内存中的基地址的 0x4001 1400，控制端口 D 的各个寄存器的偏移地址应该怎么确定呢？可以查看具体的寄存器描述（PortD 属于 GPIO 端口中的 D 端口）：

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址：0x00

复位值：0x4444 4444

可以看到该文档写了‘偏移地址：0x00’，这表示 GPIOD_CRL 这个寄存器的地址等于 Port D 的位置+偏移地址就可以算出它的地址，即：

GPIOD_CRL 寄存器地址：0x40001 1400 (PortB 的地址) + 0x00 (GPIOx_CRL 的偏移地址) = 0x40001

我们再看一个地址：

8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)

偏移地址：0x04

复位值：0x4444 4444

GPIOD_CRH 寄存器地址：0x4001 1400 (PortB 的地址) + 0x04 (GPIOx_CRH 的偏移地址) = 0x40001 1404

也就是说，当我们访问 0x40001 1404 这个地址所指向的内容时，实际上就是在访问 GPIOD_CRH 这个寄存器了，就这么简单。

4.6 STM32芯片单个管脚是怎么被控制以及被管理的？（如何阅读芯片寄存器）

这里主要是如何去查看 CPU 芯片单个管脚的寄存器表。

实际上，点亮这个 LED 灯，只需要使得我们的 STM32 芯片的 PF6 管脚输出低电平就可以了，那么如何控制一个 PF6 管脚的状态呢？

我们来举个例子，实际上 STM32 的 PF6 管脚的状态是由 STM32 芯片内部的一些寄存器来控制的，通过这些寄存器，可以控制将管脚配置成输出或者输入，拉高还是拉低。芯片通过获取寄存器不同的值，对应我们的 STM32 芯片手册寄存器说明书，就可以知道芯片就相当于获得了不同的命令，获取命令后就开始执行命令，大家可以看下图，还是看这个《STM32F103 中文参考手册》754 页的文档：

（注意：手册可以从 STM32 神舟 III 号开发板的光盘里获得）

STM32F10xxx 参考手册



参考手册

**STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx
和 STM32F107xx, ARM 内核 32 位高性能微控制器**

导言

本参考手册针对应用开发，提供关于如何使用 STM32F101xx、STM32F102xx、STM32F103 和 STM32F105xx/STM32F107xx 微控制器的存储器和外设的详细信息。在本参考手册中 STM32F101xx、STM32F102xx、STM32F103 和 STM32F105xx/STM32F107xx 被统称为

可以看到文档的 8.2 节，注意文档目录红框里的这些寄存器，现在我们就开始仔细来研究一下它们，这是专门描述 GPIO 寄存器的章节，具体内容大家自己打开文档阅读一下：

| | | |
|--------|---------------------------------------|-----|
| 8.1.2 | 单独的位设置或位清除 | 107 |
| 8.1.3 | 外部中断/唤醒线 | 107 |
| 8.1.4 | 复用功能(AF) | 107 |
| 8.1.5 | 软件重新映射I/O复用功能 | 107 |
| 8.1.6 | GPIO锁定机制 | 107 |
| 8.1.7 | 输入配置 | 107 |
| 8.1.8 | 输出配置 | 108 |
| 8.1.9 | 复用功能配置 | 109 |
| 8.1.10 | 模拟输入配置 | 109 |
| 8.1.11 | 外设的GPIO配置 | 110 |
| 8.2 | GPIO寄存器描述 这里就是控制GPIO管脚芯片内部的寄存器 | 113 |
| 8.2.1 | 端口配置低寄存器(GPIOx_CRL) (x=A..E) | 113 |
| 8.2.2 | 端口配置高寄存器(GPIOx_CRH) (x=A..E) | 114 |
| 8.2.3 | 端口输入数据寄存器(GPIOx_IDR) (x=A..E) | 114 |
| 8.2.4 | 端口输出数据寄存器(GPIOx_ODR) (x=A..E) | 115 |
| 8.2.5 | 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) | 115 |
| 8.2.6 | 端口位清除寄存器(GPIOx_BRR) (x=A..E) | 115 |
| 8.2.7 | 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) | 116 |
| 8.3 | 复用功能I/O和调试配置(AFIO) | 116 |
| 8.3.1 | 把OSC32_IN/OSC32_OUT作为GPIO 端口PC14/PC15 | 116 |
| 8.3.2 | 把OSC_IN/OSC_OUT引脚作为GPIO端口PD0/PD1 | 117 |
| 8.3.3 | CAN1复用功能重映射 | 117 |
| 8.3.4 | CAN2复用功能重映射 | 117 |
| 8.3.5 | JTAG/SWD复用功能重映射 | 117 |
| 8.3.6 | ADC复用功能重映射 | 118 |
| 8.3.7 | 定时器复用功能重映射 | 118 |
| 8.3.8 | USART复用功能重映射 | 119 |
| 8.3.9 | I ² C1复用功能重映射 | 120 |

我们进入文档，浏览到第 113 页，我们看其中一个寄存器到底写了些什么：

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址: 0x00

复位值: 0x4444 4444

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|-----------|------------|-----------|------------|-----------|------------|------|--------------------|------|------|------|------|------|------|
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| RW 15 | RW 14 | RW 13 | RW 12 | RW 11 | RW 10 | RW 9 | RW 8 | RW 7 | RW 6 | RW 5 | RW 4 | RW 3 | RW 2 | RW 1 | RW 0 |
| CNF3[1:0] | MODE3[1:0] | CNF2[1:0] | MODE2[1:0] | CNF1[1:0] | MODE1[1:0] | CNF0[1:0] | MODE0[1:0] | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2 | CNFy[1:0]: 端口x配置位(y = 0...7) (GPIO端口模式的配置) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | | | | | | | www.armjishu.com | | | | | | |
| 位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0 | MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式，最大速度10MHz 10: 输出模式，最大速度2MHz 11: 输出模式，最大速度50MHz | | | | | | | | GPIO端口速率的配置 | | | | | | |

我们看到如下内容，我们逐个列出来：

- 1) GPIOx_CRL 这个寄存器一共是 32 个比特，从 0-31bit，有 32 个位
- 2) GPIOx_CRL 寄存器的偏移地址是 0x00；表示在 GPIO 端口的基址加上这个偏移地址 0x00 就可以访问到这个寄存器的内容。
- 3) GPIOx_CRL 寄存器的复位值是 0x4444 4444 总共 8 个 ‘4’
- 4) GPIOx_CRL 寄存器其中第 0、1 位和 4、5 位和 8、9 位和 12、13 位以此类推到 28、29 每两个位为一个组，叫做 MODEy 组；主要功能是设置这个管脚是输入模式，还是输出模式（如果是输出模式，还要确认输出速度是 10MHz、还是 2MHz、还是 50MHz）
- 5) GPIOx_CRL 寄存器其中第 2、3 位和 6、7 位和 10、11 位和 14、15 位以此类推到 30、31 每两个位为一个组，叫做 CNFy 组；主要功能是设置具体哪种输入输出的模式；例如如果是管脚输出，那么要确定是通用推挽输出模式，还是开漏输出模式，还是复用功能推挽输出模式，还是复用功能开漏输出模式；如果是管脚输入，是模拟输入模式还是浮空输入模式，还是上拉/下拉输入模式等。
- 6) 其他的寄存器也是这样查看表和状态，我们寄存器复位值是 0x4444 4444，十六制的 4 转换成二进制是 0100，即 CNF=01，MODE=00，我们可以查表知道 MODE=00 表示这个管脚复位后是输入模式，CNF=01 表示是浮空输入模式。

是不是看得有点晕了，但是没有办法，我们就是通过改变这些寄存器的值来设置芯片管脚的，使得芯片管脚按照寄存器手册里的规定来进行相应的工作；可以是输出或输入，可以是高电平或是低电平（这个是另外一个寄存器来控制，大家可以对应看手册里的寄存器说明），从而达到我们

| | |
|--|-----|
| 8.2 GPIO寄存器描述 | 113 |
| 8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E) | 113 |
| 8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E) | 114 |
| 8.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E) | 114 |
| 8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) | 115 |
| 8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) | 115 |
| 8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E) | 115 |
| 8.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) | 116 |

控制一个

LED 灯亮和灭,像寄存器 GPIOx_CRH、GPIOx_IDR、GPIOx_ODR、GPIOx_BSRR、GPIOx_BRR、GPIOx_LCKR 都是同样的分析和阅读方法, 我们接下来再来举例子详细说明。

4.7 分析一个最简单的例程

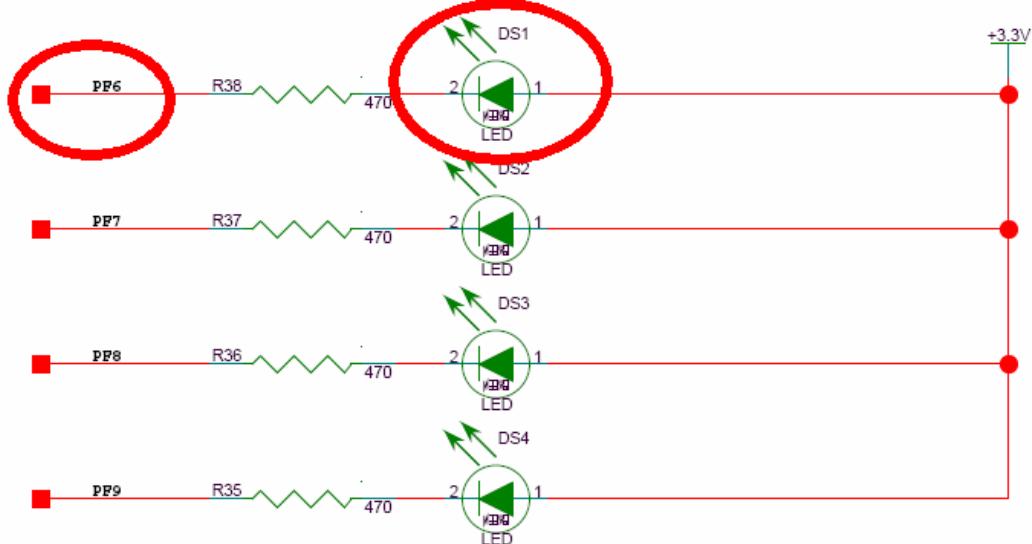
4.7.1 硬件原理图分析

例程硬件原理图说明

现在, 知道可以访问处理器所有的寄存器了, 我们可以通过改写这些寄存器的值, 控制芯片做不同的功能和操作。

下面我们正式写个例程来感受一下. 这个例程用 C 语言来修改这个内存地址的内容, 从而控制寄存器, 通过寄存器控制 STM32 芯片的 PF6 管脚使得一个灯亮和灭的。

原理图如下, 上面已经有介绍:



例程 main.c 源代码 (可以直接运行):

以下是 main.c 的源文件, 读者可以直接粘贴编译:

```
*****代码直接拷贝进去可以直接运行 开始 *****
#define      __IO      volatile
typedef unsigned          int uint32_t;
typedef __IO uint32_t vu32;
typedef unsigned short    int uint16_t;
```

```

#define GPIO_Pin_0           ((uint16_t)0x0001) /*!< Pin 0 selected */
#define GPIO_Pin_1           ((uint16_t)0x0002) /*!< Pin 1 selected */
#define GPIO_Pin_2           ((uint16_t)0x0004) /*!< Pin 2 selected */
#define GPIO_Pin_3           ((uint16_t)0x0008) /*!< Pin 3 selected */
#define GPIO_Pin_4           ((uint16_t)0x0010) /*!< Pin 4 selected */
#define GPIO_Pin_5           ((uint16_t)0x0020) /*!< Pin 5 selected */
#define GPIO_Pin_6           ((uint16_t)0x0040) /*!< Pin 6 selected */
#define GPIO_Pin_7           ((uint16_t)0x0080) /*!< Pin 7 selected */
#define GPIO_Pin_8           ((uint16_t)0x0100) /*!< Pin 8 selected */
#define GPIO_Pin_9           ((uint16_t)0x0200) /*!< Pin 9 selected */
#define GPIO_Pin_10          ((uint16_t)0x0400) /*!< Pin 10 selected */
#define GPIO_Pin_11          ((uint16_t)0x0800) /*!< Pin 11 selected */
#define GPIO_Pin_12          ((uint16_t)0x1000) /*!< Pin 12 selected */
#define GPIO_Pin_13          ((uint16_t)0x2000) /*!< Pin 13 selected */
#define GPIO_Pin_14          ((uint16_t)0x4000) /*!< Pin 14 selected */
#define GPIO_Pin_15          ((uint16_t)0x8000) /*!< Pin 15 selected */
#define GPIO_Pin_All         ((uint16_t)0xFFFF) /*!< All pins selected */

#define RCC_APB2Periph_AFIO   ((uint32_t)0x00000001)
#define RCC_APB2Periph_GPIOA  ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB  ((uint32_t)0x00000008)
#define RCC_APB2Periph_GPIOC  ((uint32_t)0x00000010)
#define RCC_APB2Periph_GPIOD  ((uint32_t)0x00000020)
#define RCC_APB2Periph_GPIOE  ((uint32_t)0x00000040)
#define RCC_APB2Periph_GPIOF  ((uint32_t)0x00000080)

/******************* GPIOB <******/
typedef struct
{
    __IO uint32_t CRL;
    __IO uint32_t CRH;
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint32_t BSRR;
    __IO uint32_t BRR;
    __IO uint32_t LCKR;
} GPIO_TypeDef;

```

```
typedef struct
```

```
{  
    __IO uint32_t CR;  
    __IO uint32_t CFGR;  
    __IO uint32_t CIR;  
    __IO uint32_t APB2RSTR;  
    __IO uint32_t APB1RSTR;  
    __IO uint32_t AHBENR;  
    __IO uint32_t APB2ENR;  
    __IO uint32_t APB1ENR;  
    __IO uint32_t BDCR;  
    __IO uint32_t CSR;  
}  
RCC_TypeDef;  
  
/****** GPIOB 管脚的内存对应地址 *****/  
#define PERIPH_BASE ((uint32_t)0x40000000)  
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)  
#define GPIOF_BASE (APB2PERIPH_BASE + 0x1c00)  
#define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
```

```
/****** RCC 时钟 <*****/  
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)  
#define RCC_BASE (AHBPERIPH_BASE + 0x1000)  
#define RCC ((RCC_TypeDef *) RCC_BASE)
```

```
/****** www.armjishu.com *****/  
void Delay(vu32 nCount);
```

```
int main(void) //main 是程序入口  
{  
    /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */  
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF;  
  
    /*-- GPIO Mode Configuration 速度，输入或输出 -----*/  
    /*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/  
    /*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/  
    GPIOF->CRL &= 0xF0FFFFFF;  
    GPIOF->CRL |= 0x03000000;
```

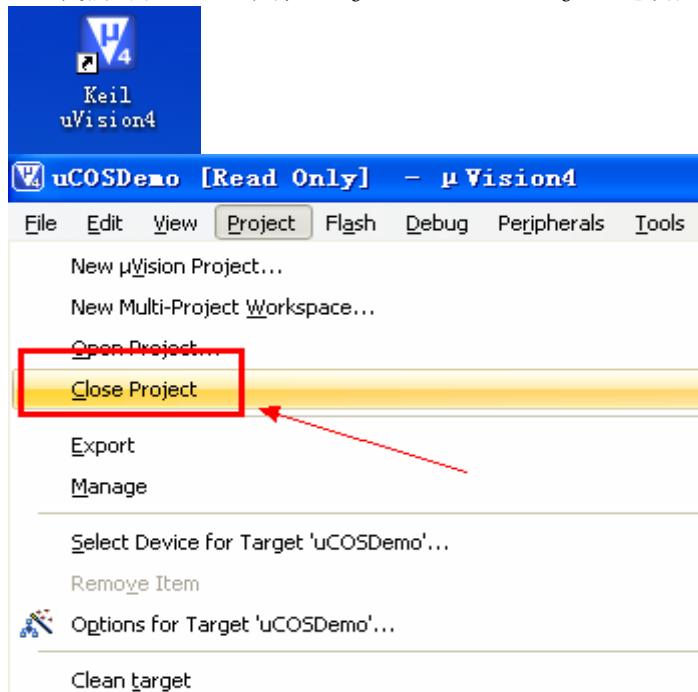
```
while(1)
{
    GPIOF->BRR = GPIO_Pin_6;
    Delay(0xFFFFFFF);
    GPIOF->BSRR = GPIO_Pin_6;
    Delay(0xFFFFFFF);
}

void Delay(vu32 nCount)          //通过不断 for 循环 nCount 次，达到延时的目的口
{
    for(; nCount != 0; nCount--);
}

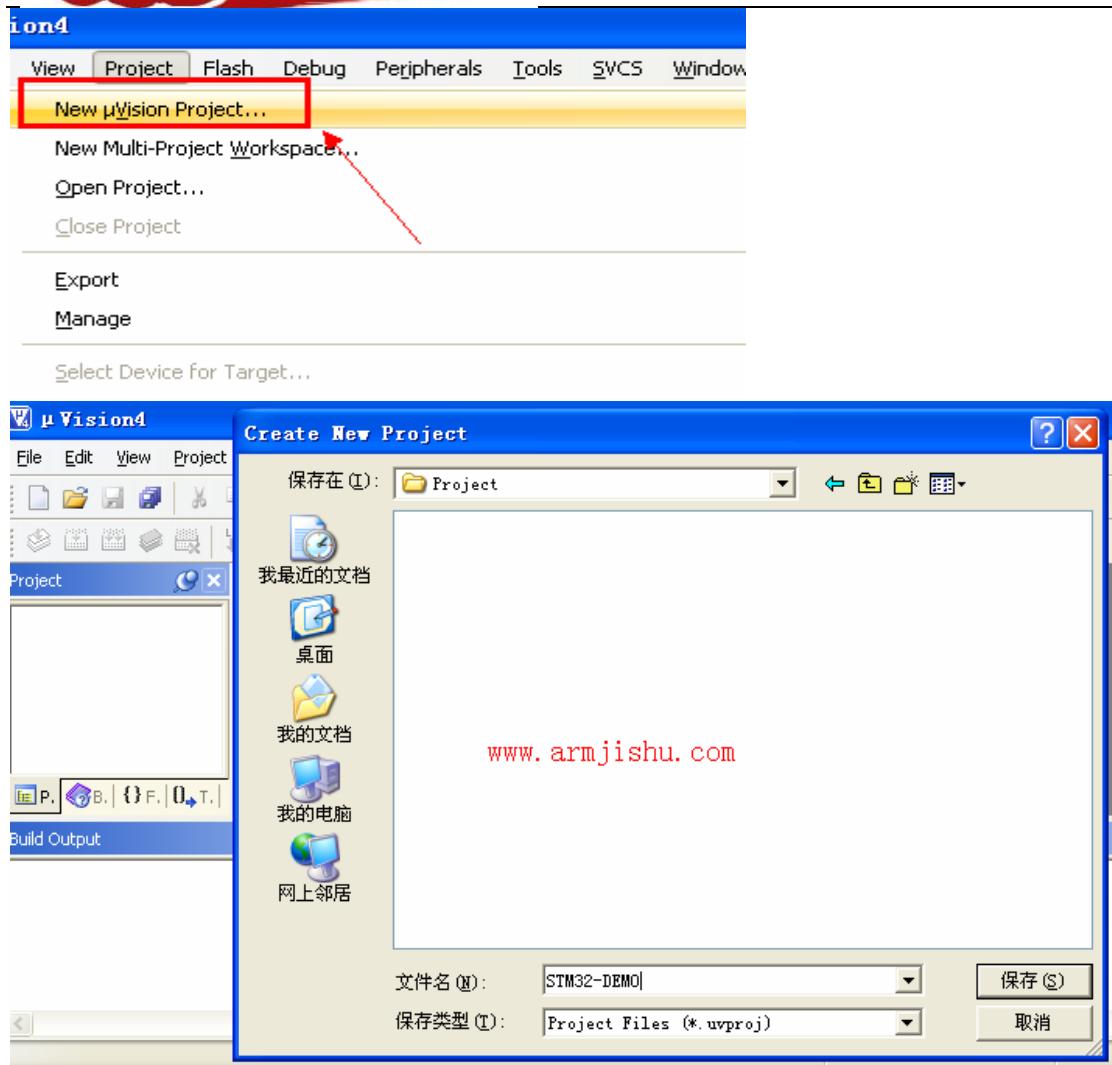
***** 此段代码直接拷贝进去可以直接运行 结束 *****/
```

4.7.2 例程环境搭建

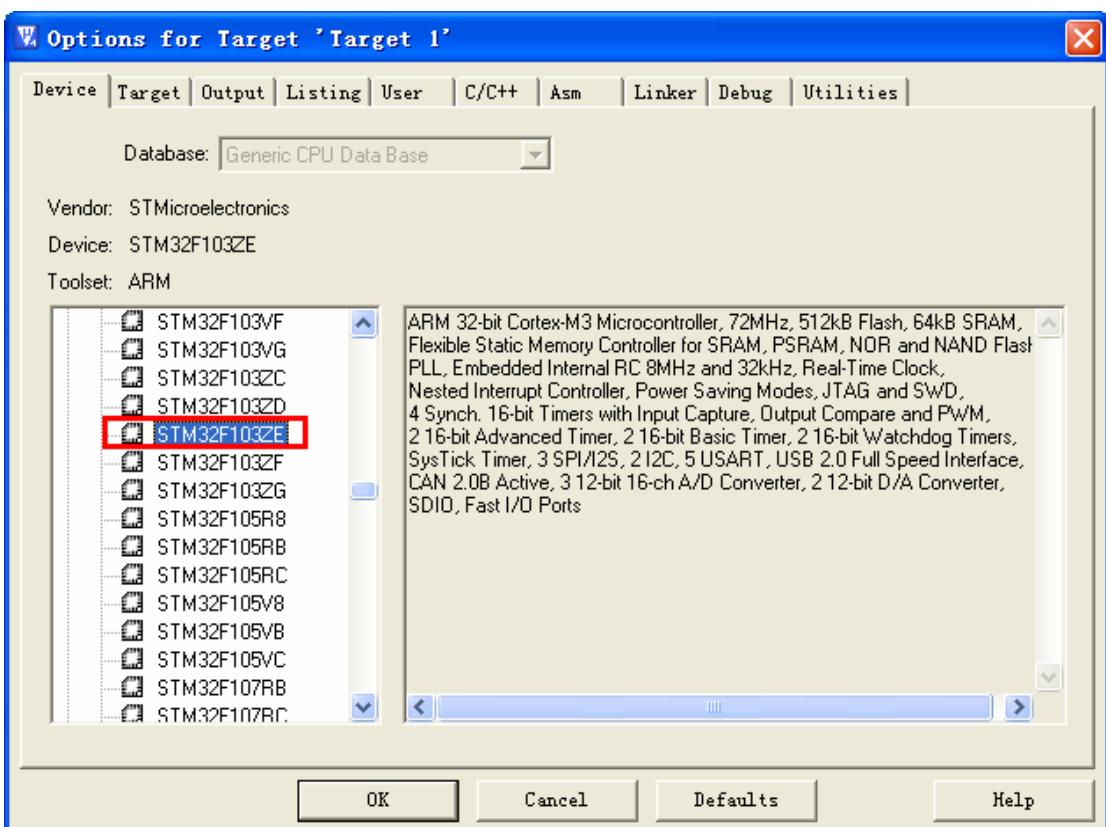
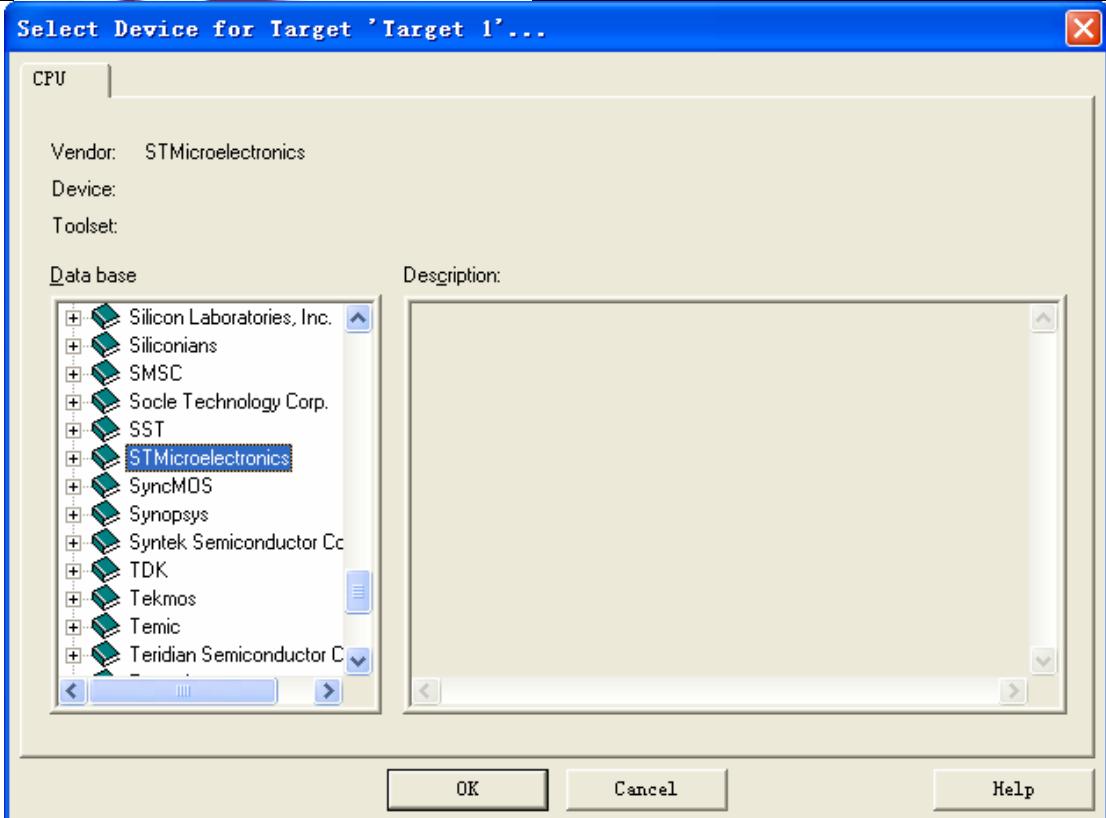
1. 点击桌面UVision4图标，启动软件，如下图。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏Project->Close Project选项把它关掉。



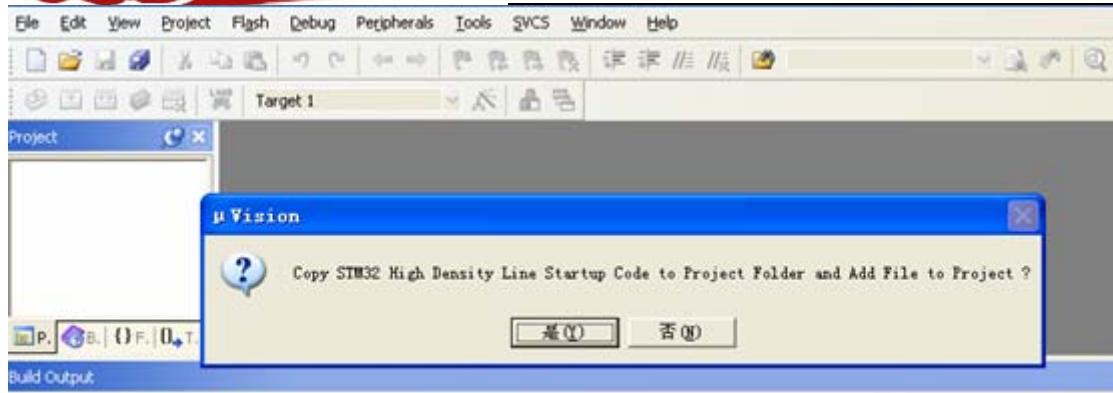
2. 在工具栏Project->New μVision Project...新建我们的工程文件，我们将新建的工程文件保存在“STM32神舟开发板模板工程”（先在电脑硬盘上新建一个“STM32神舟开发板模板工程”，在该文件夹里面新建一个Project文件夹），文件名取为神舟STM32-DEMO（英文DEMO的意思是例子），名字可以随取，点击保存。



3. 接下来的窗口是让我们选择公司跟芯片的型号，我们用的是 STM32 神舟 III 号的板子，因为我们的 STM32 神舟 III 号用的芯片是 ST 公司的 STM32F103ZET6，有 64K SRAM, 512K Flash，属于高集成度的芯片。按如下选择即可。

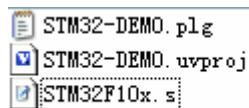


- 接下来的窗口问我们是否需要拷贝STM32的启动代码到工程文件中，这份启动代码在M3系列中都是适用的，一般情况下我们都点击是，但我们这里用的是ST的库，库文件里面也自带了这一份启动代码，这里我们点击否。



www.armjishu.com

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。



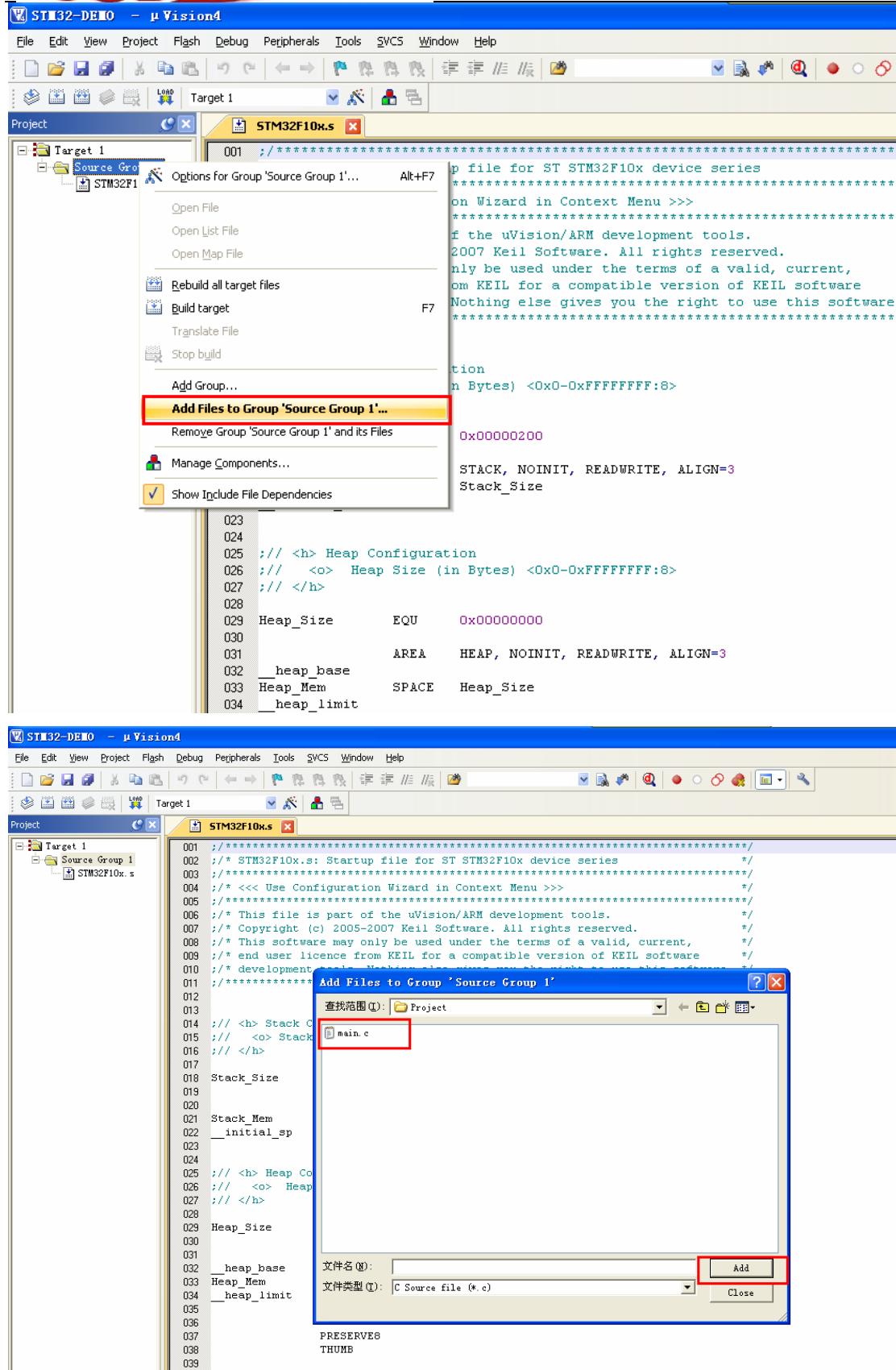
6. 可以看到目前工程里只有一个文件：

```

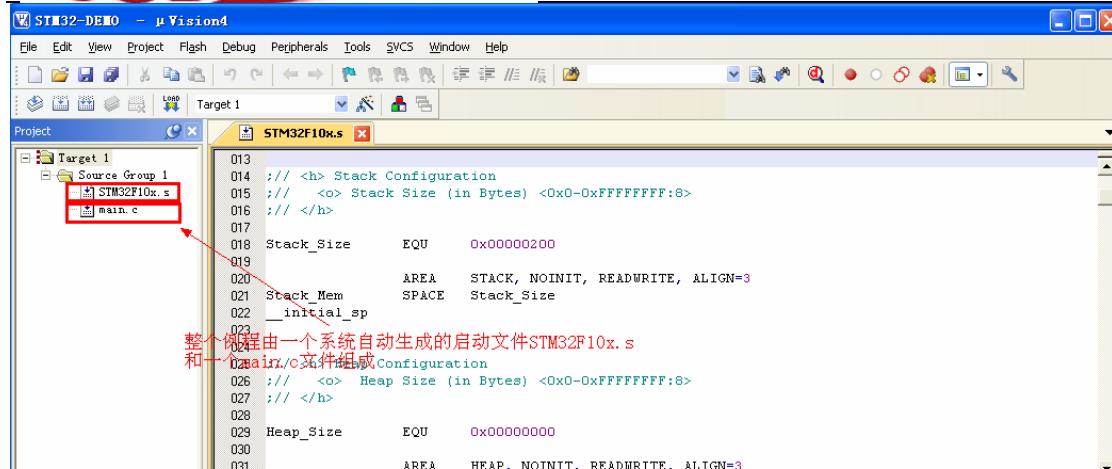
STM32-DEMO - μ Vision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Project Target 1 STM32F10x.s
001 /* ****
002 /* STM32F10x.s: Startup file for ST STM32F10x device series */
003 /* ****
004 /* <<< Use Configuration Wizard in Context Menu >>>
005 /* ****
006 /* This file is part of the wVision/ARM development tools.
007 /* Copyright (c) 2005-2007 Keil Software. All rights reserved.
008 /* This software may only be used under the terms of a valid, current,
009 /* end user licence from KEIL or a compatible version of KEIL software
010 /* development tools. Nothing else gives you the right to use this software. */
011 /* ****
012
013
014 //<h> Stack Configuration
015 //<o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
016 //</h>
017
018 Stack_Size EQU 0x00000200
019
020 AREA STACK, NOINIT, READWRITE, ALIGN=3
021 Stack_Mem SPACE Stack_Size
022 _initial_sp
023
024
025 //<h> Heap Configuration
026 //<o> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
027 //</h>
028
029 Heap_Size EQU 0x00000000
030
031 AREA HEAP, NOINIT, READWRITE, ALIGN=3
032 _heap_base
033 Heap_Mem SPACE Heap_Size
034 _heap_limit
035
036
037 PRESERVE8
038 THUMB
...

```

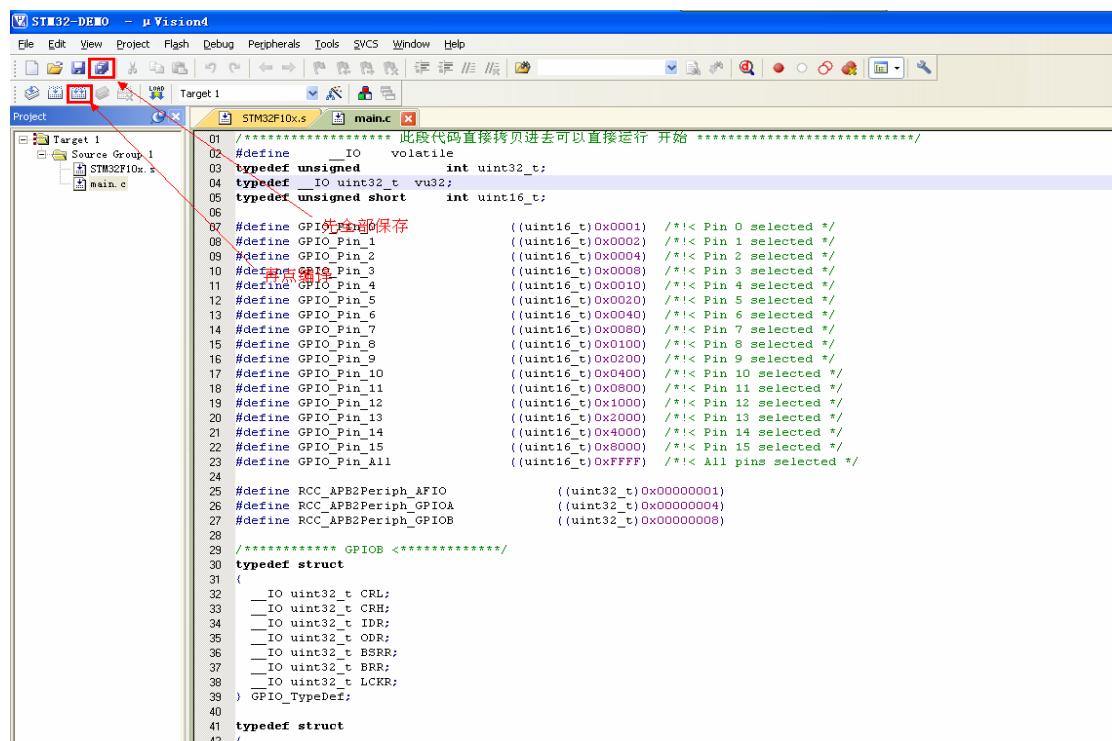
7. 新建一个 main.c 文件存放在路径 STM32 神舟 III 号开发板第 1 个例程\Project\main.c 下，然后按照以下图标操作过程把 main.c 文件添加到工程里：



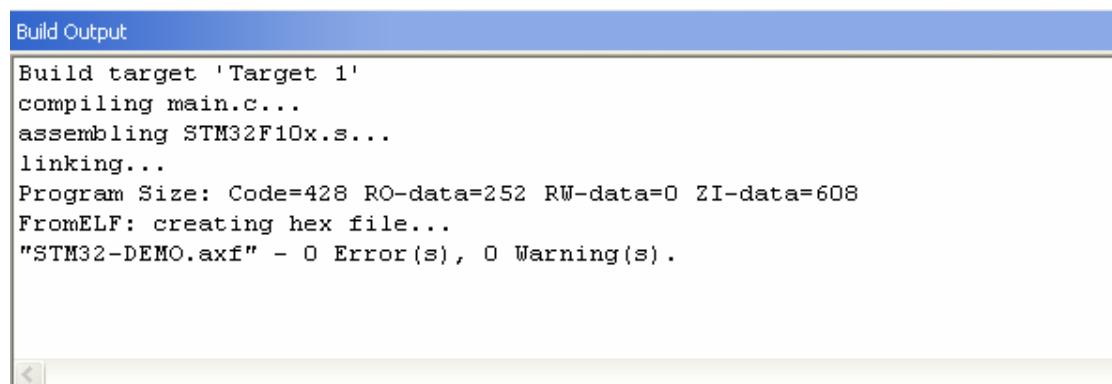
这个例程，我们将所有的代码都写到了一个 main.c 文件，不涉及到任何库函数，也没有包含任何的头文件，下面我们的截图：

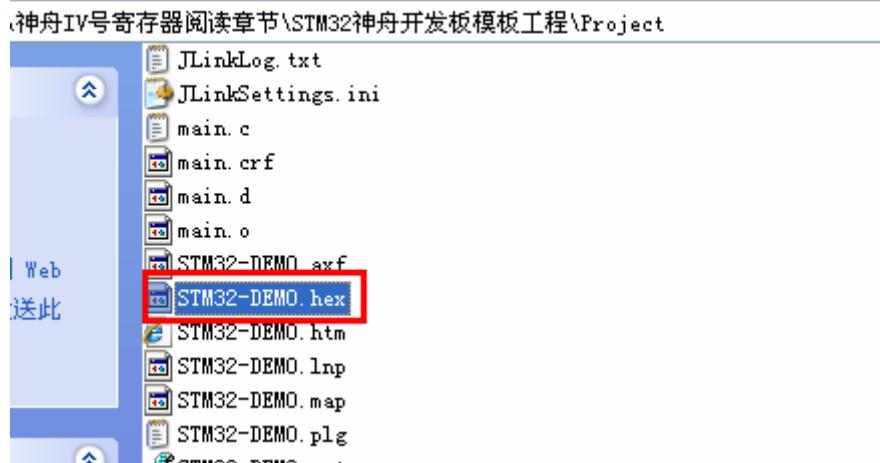


8. 把 5.61 节的代码直接全盘拷贝到 main.c 文件里



9. 编译成功，我们可以看到编译后的 HEX 文件，我们可以直接在光盘中找到这个文件，直接进入 Project 文件夹，打开即可：





10. 该代码可以直接下载到神舟 III 号开发板中，按一下复位按键，可以看到 LED1 灯一亮一灭，具体下载方式我们推荐有三种，具体下载设置请参考手册其他章节：

- 1) JLINK V8 仿真器下载（我们推荐）
- 2) ULINK2 仿真器下载
- 3) 串口下载

11. 或者直接打开我们已经编译好的例程“STM32 神舟III号开发板从零开始建立一个模板工程”，在 http://blog.sina.com.cn/s/blog_7691b90c0101edpx.html 链接中，STM32 神舟III号开发板区域可以找到。

4.7.3 实验现象

可以看到 STM32 神舟 III 号开发板的 LED3 灯一亮一灭的闪烁。

例程软件架构和代码分析（只有一个 main.c 文件）

4.7.4 代码详细分析：

```

01 /***** 此段代码直接拷贝进去可以直接运行 开始 *****/
02 #define __IO volatile
03 typedef unsigned int uint32_t;
04 typedef __IO uint32_t vu32;
05 typedef unsigned short int uint16_t;

```

分析 1：volatile 是什么？怎么用？

答：简单的说，就是不让编译器进行优化，即每次读取或者修改值的时候，都必须重新从内存或者寄存器中读取或者修改，防止从缓存处读取的值是过期了的，所以加了这个 volatile 可以保证每次读的值绝对是实时的：

一般说来，volatile 用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加 volatile；
2. 多任务环境下各任务间共享的标志应该加 volatile；

3. 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义。

我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。搞嵌入式的家伙们经常同硬件、中断、RTOS 等等打交道，所有这些都要求用到 volatile 变量。不懂得 volatile 的内容将会带来灾难。假设被面试者正确地回答了这是问题（嗯，怀疑是否会是这样），我将稍微深究一下，看一下这家伙是不是真正懂得 volatile 完全的重要性。

分析 2：__I、__O、__IO 是什么？

答：如下：

I：输入口。既然是输入，那么寄存器的值就随时会外部修改，那就不能进行优化，每次都要重新从寄存器中读取。也不能写，即只读，不然就不是输入而是输出了。

O：输出口，也不能进行优化，不然你连续两次输出相同值，编译器认为没改变，就忽略了后面那一次输出，假如外部在两次输出中间修改了值，那就影响输出。

IO：输入输出口，同上。

分析 3：为什么加下划线？

答：原因是避免命名冲突，一般宏定义都是大写，但因为这里的字母比较少，所以再添加下划线来区分。这样一般都可以避免命名冲突问题，因为很少人这样命名，这样命名的人肯定知道这些是什么用的。

经常写大工程时，都会发现老是命名冲突，要不是全局变量冲突，要不就是宏定义冲突，所以我们要尽量避免这些问题，不然出问题了都不知道问题在哪里。

分析 4：typedef 是什么意思，怎么使用？

答：typedef 为 C 语言的关键字，作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int, char 等）和自定义的数据类型（struct 等）；在编程中使用 typedef 目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

- 1) typedef 的最简单使用，例如：typedef long byte_4; 表示给已知数据类型 long 起个新名字，叫 byte_4
- 2) typedef 与结构结合使用

例如：typedef struct tagMyStruct

```
{  
    int iNum;  
    long lLength;  
} MyStruct;
```

这语句实际上完成两个操作

操作 1：定义一个新的结构类型 tagMyStruct，struct 关键字和 tagMyStruct 一起，构成了这个结构类型，不论是否有 typedef，这个结构都存在。我们可以用 struct tagMyStruct varName 来定义变量，但要注意，使用 tagMyStruct varName 来定义变量是不对的，因为 struct 和 tagMyStruct 合在一起才能表示一个结构类型。

```
struct tagMyStruct  
{  
    int iNum;
```

```

long lLength;
}

```

操作 2: `typedef` 为这个新的结构起了一个名字, 叫 `MyStruct`。

```
typedef struct tagMyStruct MyStruct;
```

因此, `MyStruct` 实际上相当于 `struct tagMyStruct`, 我们可以使用 `MyStruct varName` 来定义变量。

分析 5: 所以具体的 `typedef` 代码解释如下:

1) 例: `typedef unsigned int uint32_t;`

表示使用 `uint32_t` 符号表示 `unsigned int` 符号

2) 例: `typedef __IO uint32_t vu32;`

表示使用 `vu32` 符号表示 `typedef __IO` 符号

3) 例: `typedef unsigned short int uint16_t;`

表示使用 `uint16_t` 符号表示 `unsigned short int` 符号

2. 初始化宏定义详细分析分解:

```

01
02 #define __IO volatile
03 typedef unsigned int uint32_t;
04 typedef __IO uint32_t vu32;
05 typedef unsigned short int uint16_t;
06
07 #define GPIO_Pin_0 ((uint16_t)0x0001) /*!< Pin 0 selected */
08 #define GPIO_Pin_1 ((uint16_t)0x0002) /*!< Pin 1 selected */
09 #define GPIO_Pin_2 ((uint16_t)0x0004) /*!< Pin 2 selected */
10 #define GPIO_Pin_3 ((uint16_t)0x0008) /*!< Pin 3 selected */
11 #define GPIO_Pin_4 ((uint16_t)0x0010) /*!< Pin 4 selected */
12 #define GPIO_Pin_5 ((uint16_t)0x0020) /*!< Pin 5 selected */
13 #define GPIO_Pin_6 ((uint16_t)0x0040) /*!< Pin 6 selected */
14 #define GPIO_Pin_7 ((uint16_t)0x0080) /*!< Pin 7 selected */
15 #define GPIO_Pin_8 ((uint16_t)0x0100) /*!< Pin 8 selected */
16 #define GPIO_Pin_9 ((uint16_t)0x0200) /*!< Pin 9 selected */
17 #define GPIO_Pin_10 ((uint16_t)0x0400) /*!< Pin 10 selected */
18 #define GPIO_Pin_11 ((uint16_t)0x0800) /*!< Pin 11 selected */
19 #define GPIO_Pin_12 ((uint16_t)0x1000) /*!< Pin 12 selected */
20 #define GPIO_Pin_13 ((uint16_t)0x2000) /*!< Pin 13 selected */
21 #define GPIO_Pin_14 ((uint16_t)0x4000) /*!< Pin 14 selected */
22 #define GPIO_Pin_15 ((uint16_t)0x8000) /*!< Pin 15 selected */
23 #define GPIO_Pin_All ((uint16_t)0xFFFF) /*!< All pins selected */
24
25 #define RCC_APB2Periph_AFIO ((uint32_t)0x00000001)
26 #define RCC_APB2Periph_GPIOA ((uint32_t)0x00000004)
27 #define RCC_APB2Periph_GPIOB ((uint32_t)0x00000008)
28 #define RCC_APB2Periph_GPIOC ((uint32_t)0x00000010)
29 #define RCC_APB2Periph_GPIOD ((uint32_t)0x00000020)
30 #define RCC_APB2Periph_GPIOE ((uint32_t)0x00000040)
31 #define RCC_APB2Periph_GPIOF ((uint32_t)0x00000080)
32
33 /****** GPIOB <******/
34 typedef struct
35 {
36     __IO uint32_t CRL;
37     __IO uint32_t CRH;
38     __IO uint32_t IDR;
39     __IO uint32_t ODR;
40     __IO uint32_t BSRR;
41     __IO uint32_t BRR;

```

分析 1: 可以看出规律, `GPIO_Pin_0`、`GPIO_Pin_1` 到 `GPIO_Pin_15` 总共 16 个 `define` 定义每个都是一个 16 比特(`uint16_t`)的对象。

每个端口都有 16 个 GPIO 管脚，比如 GPIOA, GPIOB, GPIOC 等，我们用 16bit 的位来表示，即 2 个字节，每个 bit 表示 16 个 GPIO 管脚中的一个，可以看下面 2 个寄存器，就是控制 GPIO 对应的具体管脚是高电平还是低电平的，通过对设置对应位为 0 或为 1，就可以使得管脚的电平为高或低。

每个 GPIO_Pin_x 占用这 16 个比特中的 1 个位，其他剩余的 15 个位都是 0，这 16 个 GPIO_Pin_x 就被用来表示芯片各个不同端口的 16 个管脚，比如 PA0、PA1 一直到 PA15 分别对应 GPIO_Pin_0、GPIO_Pin_1 到 GPIO_Pin_15，这样定义好之后具体如何使用我们后面还会再说。

```
#define GPIO_Pin_0      ((uint16_t)0x0001)    0000 0000 0000 0001
#define GPIO_Pin_1      ((uint16_t)0x0002)    0000 0000 0000 0010
#define GPIO_Pin_2      ((uint16_t)0x0004)    0000 0000 0000 0100
#define GPIO_Pin_3      ((uint16_t)0x0008)    0000 0000 0000 1000
#define GPIO_Pin_4      ((uint16_t)0x0010)    0000 0000 0001 0000
#define GPIO_Pin_5      ((uint16_t)0x0020)    0000 0000 0010 0000
#define GPIO_Pin_6      ((uint16_t)0x0040)    0000 0000 0100 0000
#define GPIO_Pin_7      ((uint16_t)0x0080)    0000 0000 1000 0000
#define GPIO_Pin_8      ((uint16_t)0x0100)    0000 0001 0000 0000
#define GPIO_Pin_9      ((uint16_t)0x0200)    0000 0010 0000 0000
#define GPIO_Pin_10     ((uint16_t)0x0400)    0000 0100 0000 0000
#define GPIO_Pin_11     ((uint16_t)0x0800)    0000 1000 0000 0000
#define GPIO_Pin_12     ((uint16_t)0x1000)    0001 0000 0000 0000
#define GPIO_Pin_13     ((uint16_t)0x2000)    0010 0000 0000 0000
#define GPIO_Pin_14     ((uint16_t)0x4000)    0100 0000 0000 0000
#define GPIO_Pin_15     ((uint16_t)0x8000)    1000 0000 0000 0000
#define GPIO_Pin_All   ((uint16_t)0xFFFF)    1111 1111 1111 1111
```

分析 2：这里是定义 GPIO 端口 F 的一些初始化变量，后面那些具体的地址需要查看芯片参考手册

```
/* ***** GPIOF 管脚的内存对应地址 *****/
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
#define GPIOF_BASE            (APB2PERIPH_BASE + 0x1c00)
#define GPIOF                 ((GPIO_TypeDef *) GPIOF_BASE)

/* ***** RCC 时钟 <***** */
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)
#define RCC_BASE               (AHBPERIPH_BASE + 0x1000)
#define RCC                  ((RCC_TypeDef *) RCC_BASE)

/* ***** www.armjishu.com *****/
void Delay(vu32 nCount);
```

| | | | |
|---------------------------|---------|------|-------------|
| 0x4001 2400 - 0x4001 27FF | ADC1 | APB2 | 参见11.12.15节 |
| 0x4001 2000 - 0x4001 23FF | GPIO端口G | | 参见8.5节 |
| 0x4001 2000 - 0x4001 23FF | GPIO端口F | | 参见8.5节 |
| 0x4001 1800 - 0x4001 1BFF | GPIO端口E | | 参见8.5节 |
| 0x4001 1400 - 0x4001 17FF | GPIO端口D | | 参见8.5节 |
| 0x4001 1000 - 0x4001 13FF | GPIO端口C | | 参见8.5节 |
| 0x4001 0C00 - 0x4001 0FFF | GPIO端口B | | 参见8.5节 |
| 0x4001 0800 - 0x4001 0BFF | GPIO端口A | | 参见8.5节 |
| 0x4001 0400 - 0x4001 07FF | EXTI | | 参见9.3.7节 |
| | | | |

1) 定义总线的基地址（这个需要参考手册）

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

2) APB2PERIPH_BASE (APB2 时钟总线) 的地址是在总线基址上加多 0x10000，刚好就是上图的 AFIO 寄存器地址，具体可以看参考手册

```
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
```

3) 定义 GPIO 端口 F 的基地址，该地址是 0x4001 2000。

```
#define GPIOD_BASE (APB2PERIPH_BASE + 0x2000)
```

4) 定义一个 GPIO_TypeDef 的 struct 结构，从 GPIO 端口 F 的基址开始进行覆盖

```
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
```

分析 3：这里是初始化 RCC 时钟总线的基地址，详细分析与上面同原理，具体的地址需要查看芯片参考手册

| | |
|----|--|
| 61 | /***** RCC 时钟 *****/ |
| 62 | #define AHBPERIPH_BASE (PERIPH_BASE + 0x20000) |
| 63 | #define RCC_BASE (AHBPERIPH_BASE + 0x1000) |
| 64 | #define RCC ((RCC_TypeDef *) RCC_BASE) |
| | ----- |
| | CRC |
| | Reserved |
| | Flash interface |
| | Reserved |
| | RCC |
| | Reserved |
| | DMA2 |
| | DMA1 |
| | Reserved |
| | SDIO |
| | Reserved |
| | ADC3 |
| | USART1 |
| | 0x4002 3000 - 0x4002 33FF |
| | 0x4002 2400 - 0x4002 2FFF |
| | 0x4002 2000 - 0x4002 23FF |
| | 0x4002 1400 - 0x4002 1FFF |
| | 0x4002 1000 - 0x4002 13FF |
| | 0x4002 0400 - 0x4002 0FFF |
| | 0x4002 0400 - 0x4002 07FF |
| | 0x4002 0000 - 0x4002 03FF |
| | 0x4001 8400 - 0x4001 FFFF |
| | 0x4001 8000 - 0x4001 83FF |
| | 0x4001 400 - 0x4001 7FFF |
| | 0x4001 3C00 - 0x4001 3FFF |
| | 0x4001 3800 - 0x4001 3BFF |

分析 4：设置端口的偏移量，后面我们会详细解释

```
#define RCC_APB2Periph_AFIO ((uint32_t)0x00000001)
#define RCC_APB2Periph_GPIOA ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB ((uint32_t)0x00000008)
#define RCC_APB2Periph_GPIOD ((uint32_t)0x00000020)
```

3. 定义这两个结构体与芯片参考手册中的寄存器进行对应，芯片参考手册中对应的寄存器都是 32bit 的，所以在这个结构体的各个对象都被定义成 uint32_t 类型，并且是__IO 类型，表示每次操作寄存器都是实时获取数据：

```
40  typedef struct
41  {
42      __IO uint32_t CRL;
43      __IO uint32_t CRH;
44      __IO uint32_t IDR;
45      __IO uint32_t ODR;
46      __IO uint32_t BSRR;
47      __IO uint32_t BRR;
48      __IO uint32_t LCKR;
49  } GPIO_TypeDef;
50
51  typedef struct
52  {
53      __IO uint32_t CR;
54      __IO uint32_t CFGR;
55      __IO uint32_t CIR;
56      __IO uint32_t APB2RSTR;
57      __IO uint32_t APB1RSTR;
58      __IO uint32_t AHBENR;
59      __IO uint32_t APB2ENR;
60      __IO uint32_t APB1ENR;
61      __IO uint32_t BDCR;
62      __IO uint32_t CSR;
63  } RCC_TypeDef;
```

4. 下面是 main 函数的剖析，总共来说分为 4 个步骤，下面一一介绍：

```
int main(void) //main 是程序入口
{
    /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF; // 步骤1

    /*-- GPIO Mode Configuration 速度，输入或输出 --*/
    /*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出） --*/
    /*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出） --*/
    GPIOF->CRL &= 0xFOFFFFFF;
    GPIOF->CRL |= 0x03000000; // 步骤2

    while (1)
    {
        GPIOF->BRR = GPIO_Pin_6;
        Delay(0x2FFFFF);
        GPIOF->BSRR = GPIO_Pin_6;
        Delay(0x2FFFFF); // 步骤3
    }
}
```

步骤 1：使能 APB2 总线的时钟。

对 GPIO 的端口 F 时钟使能，这个是芯片厂家所规定的操作，我们先按照这样来操作就可以，具体实现方式也是将对应 GPIOF 寄存器使能，同时，也有 RCC，串口接口，CAN 接口，485 接口等时钟的使能寄存器，使用前都需要先对时钟总线使能的。

使能操作完毕，就相当于我们对要使用的这个接口功能进行使能和激活，每个接口在使用前都必须要求使能和激活，只有激活后才可以使用。

步骤 2：配置 GPIO 端口的状态。

输入还是输出，速度多少，和大家可以参考对应的芯片寄存器手册，可以看到我们将 PB2 设置成‘00：通用推挽输出模式’并且速度是‘11：输出模式，最大速度 50MHz’

具体可以参考 GPIOF_CRL 寄存器，我们将这个寄存器设置为了 0x0300 0000。

步骤 3：进入 while 死循环

可以使得我们的点灯程序一直不会退出，达到重复一亮一灭的功能。

步骤 4：GPIO 输入和输出使得灯亮灭

GPIOF_BSRR 对应位设置 1，可以使得对应管脚的 ODR 位为 1；GPIO_BRR 的对应位设置 1，可以使得对应管脚的 ODR 位为 0

那么 ODR 位是什么呢？这个就是端口输出数据寄存器 GPIOF_ODR，实际上这个寄存器里的对应位的变化才是真正 GPIO 管脚高低电平的变化；而寄存器 GPIOF_BSRR 和寄存器 GPIOF_BRR 则可以间接影响到它。

当然上面程序，我们也可以改成直接操作 GPIOF_ODR 的代码，这个留个大家来做练习吧。

8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E)

地址偏移: 0Ch

复位值: 0x0000 0000

| | | | | | | | | | | | | | | | |
|------------------|-------|--|-------|-------|-------|------|------|------|------|------|------|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ODR15 | ODR14 | ODR13 | ODR12 | ODR11 | ODR10 | ODR9 | ODR8 | ODR7 | ODR6 | ODR5 | ODR4 | ODR3 | ODR2 | ODR1 | ODR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位31:16 保留，始终读为0。 | | | | | | | | | | | | | | | |
| 位15:0 | | ODRy[15:0]: 端口输出数据(y = 0...15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注：对 GPIOx_BSRR(x = A...E)，可以分别地对各个ODR位进行独立的设置/清除。 | | | | | | | | | | | | | |

| | |
|-------|---|
| 位15:0 | ODRy[15:0]: 端口输出数据(y = 0...15) 这些位可读可写并只能以字(16位)的形式操作。 注：对 GPIOx_BSRR(x = A...E)，可以分别地对各个ODR位进行独立的设置/清除。 |
|-------|---|

4.7.5 程序代码详细说明

代码的定义和声明如何与芯片内部资源挂钩

C 语言程序代码如何真正访问芯片内部寄存器的呢？大家看下面这些定义：

```

/***** GPIOF管脚的内存对应地址 *****/
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
#define GPIOF_BASE            (APB2PERIPH_BASE + 0x1c00)
#define GPIOF                 ((GPIO_TypeDef *) GPIOF_BASE)

/***** RCC时钟 *****/
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)
#define RCC_BASE               (AHBPERIPH_BASE + 0x1000)
#define RCC                  ((RCC_TypeDef *) RCC_BASE)

```

通过这几个 define 可以算出来一下地址：

$\text{GPIOF} = 0x4000\ 0000 + 0x1\ 0000 + 0x1C00 = 0x4001\ 1C00$ 刚好与 PortF 在内存中的位置对应上

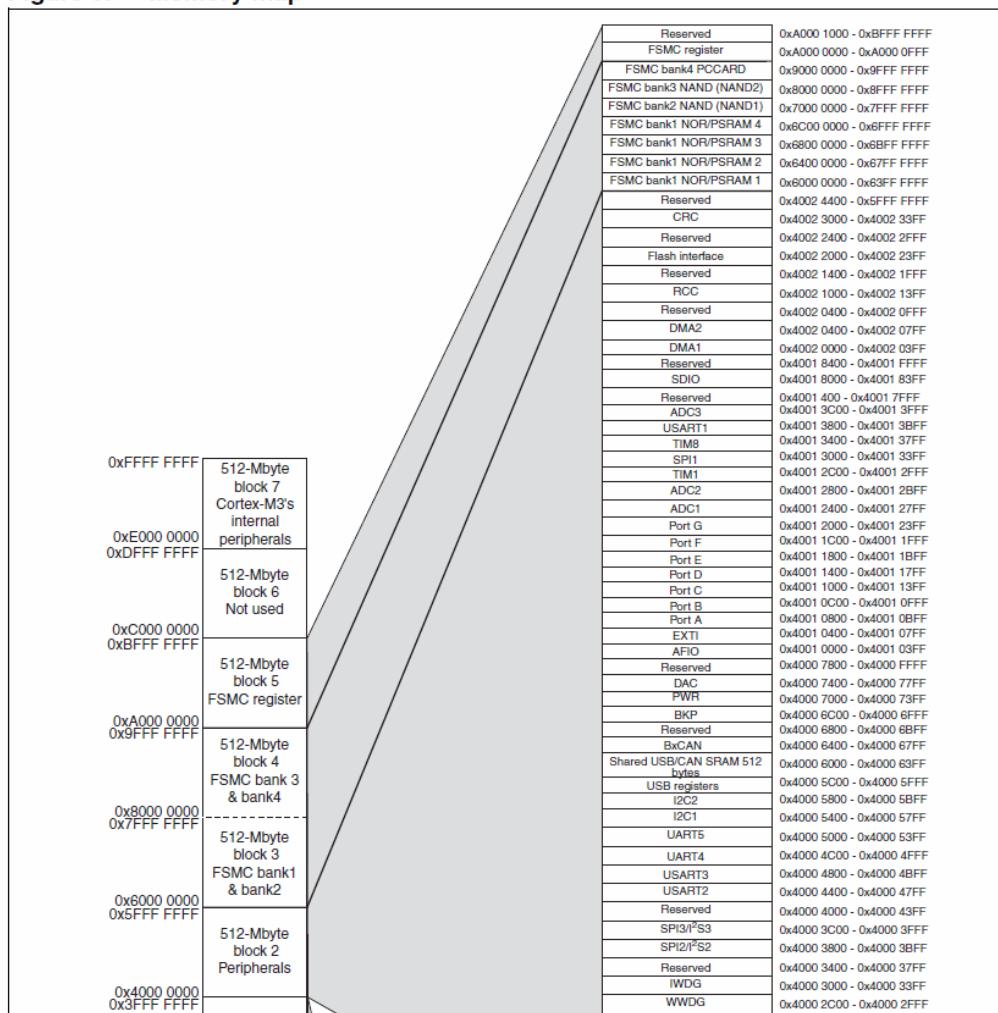
$\text{RCC} = 0x4000\ 0000 + 0x2\ 0000 + 0x1000 = 0x4002\ 1000$ 刚好也与 RCC 在内存中的位置对应上

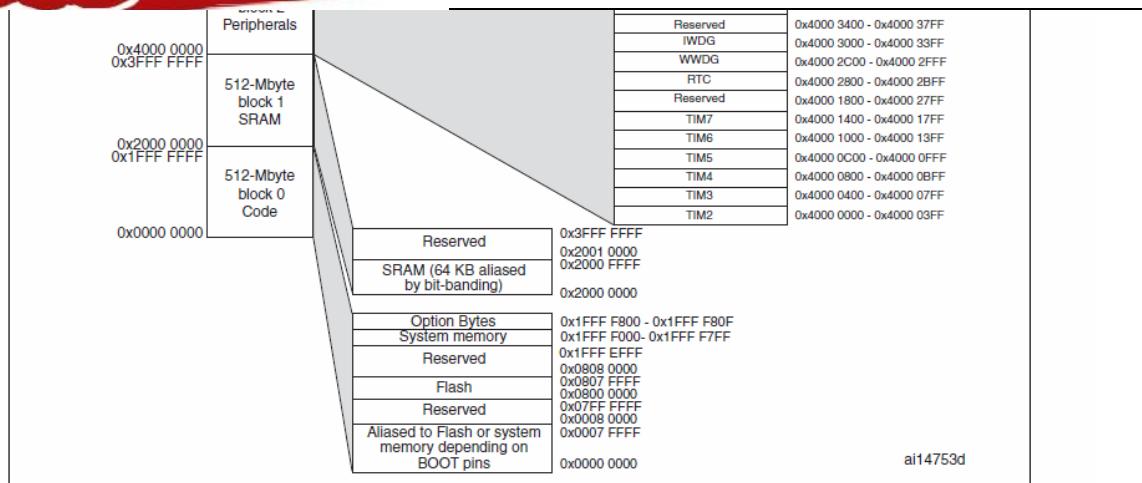
| ADU I | | UX4001 2400 - UX4001 27FF |
|-----------------|--|---------------------------|
| Port G | | 0x4001 2000 - 0x4001 23FF |
| Port F | | 0x4001 1C00 - 0x4001 1FFF |
| Port E | | 0x4001 1800 - 0x4001 1BFF |
| Port D | | 0x4001 1400 - 0x4001 17FF |
| Port C | | 0x4001 1000 - 0x4001 13FF |
| Port B | | 0x4001 0C00 - 0x4001 0FFF |
| Port A | | 0x4001 0800 - 0x4001 0BFF |
| Reserved | | 0x4002 2400 - 0x4002 2FFF |
| Flash interface | | 0x4002 2000 - 0x4002 23FF |
| Reserved | | 0x4002 1400 - 0x4002 1FFF |
| RCC | | 0x4002 1000 - 0x4002 13FF |
| Reserved | | 0x4002 0400 - 0x4002 0FFF |
| DMA2 | | 0x4002 0400 - 0x4002 07FF |
| DMA1 | | 0x4002 0000 - 0x4002 03FF |
| Reserved | | 0x4001 8400 - 0x4001 FFFF |
| SDIO | | 0x4001 8000 - 0x4001 83FF |
| Reserved | | 0x4001 400 - 0x4001 7FFF |
| ADC3 | | 0x4001 3C00 - 0x4001 3FFF |
| IISART1 | | 0x4001 3800 - 0x4001 3BFF |

更多内存映射可以找 STM32F10XXX 的数据手册资料，如下图，可以详细的进行了解了解：



Figure 9. Memory map



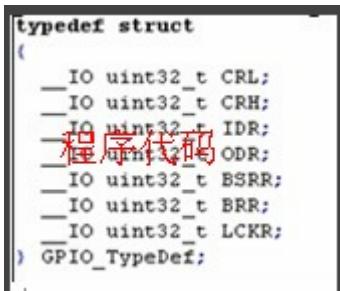


通过以上这个存储器映像图，我们就可以通过代码与存储器地址关联起来了。

4.7.6 代码如何映射到芯片内部的寄存器

这也是从库函数中摘抄出来的一种实现方式，我们通过 struct 结构可以完成对 GPIO, RCC 等外设模块中各个寄存器的管理，比如，一个 GPIO 模块中，有很多个寄存器，我们可以用 C 语言中的 struct 来对应这些寄存器。

在芯片中，一个寄存器连着一个寄存器，每个寄存器都是 32 位的（4 个字节）；我们在 struct 结构中的成员每个也都是 32 位的，一个连着一个，刚好一一对应，大家可以看下图：



8.2 GPIO 寄存器描述

- 8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)
- 8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)
- 8.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E)
- 8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) STM32中文参考手册对应章节
- 8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)
- 8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)
- 8.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E)

可以看到上图，每个寄存器都是 32 位的，左边是而且顺序刚好分别对应，结构体是会分配内存的，这样这些 C 语言中的 struct 结构体中定义的成员会对应映射到对应的寄存器上，那么我们就可以通过操纵程序中的该结构体的对应成员，就相当于操作的是对应的寄存器，这个是 C 语言和单片机软硬件对应上的又一大关键点，请不熟悉的读者好好理解一下，如实在不理解，可以致电 STM32 神舟系列

4.7.7 Main函数寄存器级分析（重点）

1. LED 灯为什么会一亮一灭呢？

```

/*-- GPIO Mode Configuration速度，输入或输出 -----*/
/*-- GPIO CRL Configuration 设置IO端口低8位的模式（输入还是输出）---*/
/*-- GPIO CRH Configuration 设置IO端口高8位的模式（输入还是输出）---*/
GPIOF->CRL &= 0xFOFFFFFF;
GPIOF->CRL |= 0x03000000;

while (1)
{
    GPIOF->BRR = GPIO_Pin_6;          管脚输出低电平，LED亮
    Delay(0x2FFFFFF);
    GPIOF->BSRR = GPIO_Pin_6;        管脚输出高电平，LED灭
    Delay(0x2FFFFFF);
}

```

我们看一下参考手册中的 GPIOx_BRR 和 GPIOx_BSRR 两个寄存器：

| | | |
|-------|----------------------------------|-----|
| 8.2.5 | 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) | 115 |
| 8.2.6 | 端口位清除寄存器(GPIOx_BRR) (x=A..E) | 115 |

8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

地址偏移：0x10

复位值：0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |

| | | |
|--------|--|--------------------------------|
| 位31:16 | BRy: 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。 | 这个16位的功能可以用其它的寄存器完成, 如GPIO_BSR |
| 位15:0 | BSy: 设置端口x的位y (y = 0...15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 设置对应的ODRy位为1 | |

8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)

地址偏移: 0x14

复位值: 0x0000 0000



115/754



参照2009年12月 RM0008 Reference Manual 英文第10版
本译文仅供参考，如有翻译错误，请以英文原稿为准。请读者随时注意在ST网站下载更新版本

通用和复用功能I/O

STM32F10xxx参考手册

| | |
|--------|---|
| 位31:16 | 保留。 |
| 位15:0 | BRy: 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 |

分析1: 从以上两个寄存器里的内容可以知道到:

- ODRy = 1就会输出高电平, 如果是高电平, 我们查看了原理图, 这个LED灯灭, 可以通过操作GPIOx_BSRR寄存器的对应位来改变
- ODRy = 0就会输出低电平, 如果是低电平, 我们查看了原理图, 这个LED灯亮, 可以通过操作GPIOx_BRR寄存器的对应位来改变

分析2: 进一步分析, 我们如何通过代码来改变这个ODRy呢? 大家请看下面:

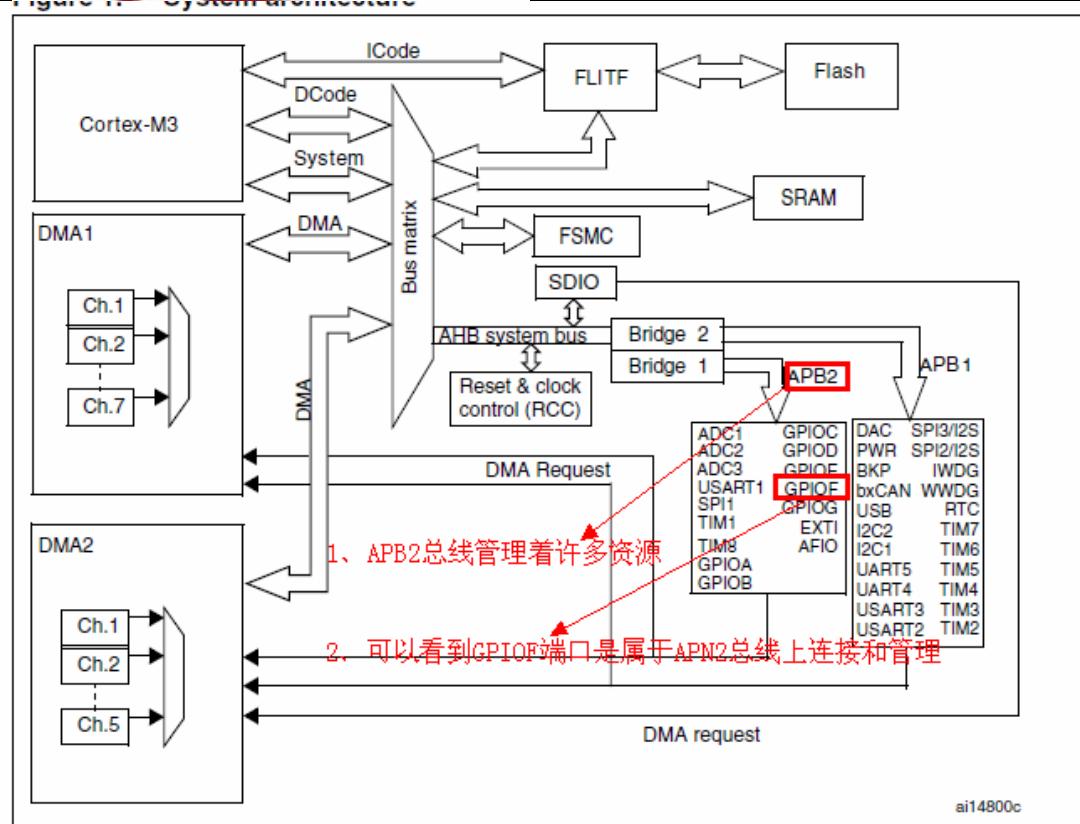
- 代码GPIOF->BRR = GPIO_Pin_6; 可以使得GPIOx_BRR寄存器的BR6位为1, 这样就是的GPIO端口F的对应ODR6=0, 即PF6管脚输出低电平, 使得LED1亮。
- 代码GPIOF->BSRR = GPIO_Pin_6; 可以使得GPIOx_BSRR寄存器的BS6位为1, 这样就是的GPIO端口F的对应ODR6=1, 即PF6管脚输出高电平, 使得LED1灭。

2. 要使用 PB2 管脚需要做哪些初始化的工作呢?

```
/* 使能APB2总线的时钟, 对GPIO的端口F时钟使能 */
RCC->APB2ENR |= RCC_APB2Periph_GPIOF; 初始化GPIOF端口时钟
```

```
/*-- GPIO Mode Configuration速度, 输入或输出 -----*/
/*-- GPIO CRL Configuration 设置IO端口低8位的模式(输入还是输出) ---*/
/*-- GPIO CRH Configuration 设置IO端口高8位的模式(输入还是输出) ---*/
GPIOF->CRL &= 0xFFFFFFFF;
GPIOF->CRL |= 0x03000000; 配置GPIOF端口模式
```

分析1: 使能APB2总线上的GPIO端口F的时钟, 我们可以看下系统图, 可以看到GPIOF是属于APB2总线管理的, 那么如何初始化这个是ST公司要求的, 而不是我们STM32神舟系列开发板官方规定的, 一切都是STM32芯片厂家ST公司制定的。



分析 2：代码 `RCC->APB2ENR |= RCC_APB2Periph_GPIOF;` 是使能 GPIOF 的时钟，我们找到寄存器 `RCC_APB2ENR`，仔细看看是什么操作的

| | |
|---|-----------|
| 7.3 RCC 寄存器 | 85 |
| 7.3.1 时钟控制寄存器(RCC_CR) | 85 |
| 7.3.2 时钟配置寄存器(RCC_CFGR) | 86 |
| 7.3.3 时钟中断寄存器(RCC_CIR) | 88 |
| 7.3.4 APB2外设复位寄存器(RCC_APB2RSTR) | 91 |
| 7.3.5 APB1外设复位寄存器(RCC_APB1RSTR) | 92 |
| 7.3.6 AHB外设时钟使能寄存器(RCC_AHBENR) | 94 |
| 7.3.7 APB2外设时钟使能寄存器(RCC_APB2ENR) | 95 |
| 7.3.8 APB1外设时钟使能寄存器(RCC_APB1ENR) | 97 |
| 7.3.9 备份域控制寄存器(RCC_BDCR) | 99 |
| 7.3.10 控制/状态寄存器(RCC_CSR) | 100 |
| 7.3.11 AHB外设时钟复位寄存器(RCC_AHBRSTR) | 101 |
| 7.3.12 时钟配置寄存器2(RCC_CFGR2) | 101 |
| 7.3.13 RCC寄存器地址映像 | 103 |

点击进入文档第 61 页可以看到 6.3.7 节，可以看到 GPIOF 端口的时钟设置选项：

6.3.7 APB2 外设时钟使能寄存器(RCC_APB2ENR)

偏移地址: 0x18

复位值: 0x0000 0000

访问: 字, 半字和字节访问

通常无访问等待周期。但在APB2总线上的外设被访问时, 将插入等待状态直到APB2的外设访问结束。

注: 当外设时钟没有启用时, 软件不能读出外设寄存器的数值, 返回的数值始终是0x0。

| | | | | | | | | | | | | | | | |
|-----------------------|--------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|----|------------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 GPIOF端口时钟选项 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADC3 EN | USART1 EN | TIM8 EN | SPI1 EN | TIM1 EN | ADC2 EN | ADC1 EN | IOPG EN | IOPF EN | IOPE EN | IOPD EN | IOPC EN | IOPB EN | IOPA EN | 保留 | AFIO EN |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

| | |
|----|--|
| 位8 | IOPGEN: IO端口G时钟使能 (I/O port G clock enable) 由软件置'1'或清'0' 0: IO端口G时钟关闭; 1: IO端口G时钟开启。 |
| 位7 | IOPFEN: IO端口F时钟使能 (I/O port F clock enable) 由软件置'1'或清'0' 0: IO端口F时钟关闭; 1: IO端口F时钟开启。 设置为1表示: GPIOF端口时钟开启 |
| 位6 | IOPEEN: IO端口E时钟使能 (I/O port E clock enable) 由软件置'1'或清'0' 0: IO端口E时钟关闭; 1: IO端口E时钟开启。 |
| 位5 | IOPDEN: IO端口D时钟使能 (I/O port D clock enable) |

我们通过代码 `RCC->APB2ENR |= RCC_APB2Periph_GPIOF;` 来对 RCC_APB2ENR 寄存器的位 3 进行操作置位, 那么 `RCC_APB2Periph_GPIOF` 的值请见:

代码: #define RCC_APB2Periph_GPIOF ((uint32_t)0x00000080)

从这句代码可以知道 0x00000080 的 8 化成二进制是 1000, 刚好是对 RCC->APB2ENR 寄存器也就是 RCC_APB2ENR 寄存器的第 7 位置位, 使得 IOPF EN 为 1, 使得 IO 端口 F 时钟使能。

分析 3: 配置 GPIO 端口 F 的工作模式, 我们这里可以看到它配置改变了 CRL 这个寄存器。

```
GPIOF->CRL &= 0xFFFFFFFF;
GPIOF->CRL |= 0x03000000;
```

我们找到这个 CRL 寄存器完整的名称叫 GPIOx_CRL 寄存器的内容:

| 8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E) | | 113 |
|--|--|-----|
| 8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E) | | 114 |
| 8.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E) | | 114 |
| 8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) | | 115 |
| 8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) | | 115 |
| 8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E) | | 115 |
| 8.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) | | 116 |

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址: 0x00

复位值: 0x4444 4444

| | | | | | | | | | | | | | | | |
|-----------|------------|-----------|------------|-----------|------------|-----------|------------|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF3[1:0] | MODE3[1:0] | CNF2[1:0] | MODE2[1:0] | CNF1[1:0] | MODE1[1:0] | CNF0[1:0] | MODE0[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

我们开始分析一下代码:

- GPIOF->CRL &= 0xF0FFFFFF;

GPIOF->CRL &= 0xF0FFFFFF = 1111 0000 1111 1111 1111 1111 1111 1111

可以看到将 CRL 寄存器的第 24、25、26、27 位清 0，其他位默认值不变，看上表可以知道，这 4 位刚好是管脚 PF6 的配置寄存器

| | | | | | | | | | | | | | | | |
|-----------|------------|-----------|------------|-----------|------------|-----------|------------|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

- GPIOB->CRL |= 0x03000000

GPIOF->CRL |= 0x03000000 = 0000 0011 0000 0000 0000 0000 0000 0000

可以看到将 CRL 寄存器的第 24、25、26、27 位

| | | | | | | | | | | | | | | | |
|-----------|------------|-----------|------------|-----------|------------|-----------|------------|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

分别置成 0、0、1、1，其他位的 GPIO 端口值都清 0，意思就是只配置 PF6 这个管脚，其他 PF 口的管脚配置寄存器都全部变成 0，这样 CNF6=00；MODE6=11；查表可以知道：

11：保留

在输出模式(MODE[1:0]>00):

因为 MODE6 大于 0，所以 CNF6 的配置应该查

00：通用推挽输出模式

看表的这一栏目：

01：通用开漏输出模式

CNF6=00 是通用推挽输出模式

10：复用功能推挽输出模式

11：复用功能开漏输出模式

MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits)

软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。

00：输入模式(复位后的状态)

01：输出模式，最大速度10MHz

10：输出模式，最大速度2MHz

11：输出模式，最大速度50MHz

MODE6=11 是最大输出50MHz

这样配置后 PF6 被配置成输出模式，输出速度为 50MHz，状态是通用推挽输出模式；因为我们这个例程中需要点 LED1 灯，这是一种输出模式。

4.7.8 函数与我们这个例程之间的关系

库函数同我们这个例程的原理是一样，后续的例程，都是源自库函数的，阅读原理一样，只是我们将一些相关功能比较密切的代码封装到一起，变成一个完整的函数。

后续该神舟文档还会升级，库函数分析版本请见下个版本的书籍。

4.8 STM32重映射功能

4.8.1 什么是STM32的重映射

就好像乾坤大挪移，移动穴位，把芯片这个管脚的功能移到另外一个管脚上。

重映射的作用简单的说就是把管脚的外设功能映射到另一个管脚，但是是不是可以随便映射的，具体对应关系参考手册上的管脚说明。比如 USART2_TX 默认在 PA2 管脚，当启用复用功能后就会将 PD5 管脚作为 USART2_TX。

4.8.2 所有的管脚都可以重映射吗

大部分都可以，但不是所有的管脚功能都能重映射，比如 ADC1_IN0 就只能在 PA0，这个具体要看芯片的数据手册。

4.8.3 为什么要有STM32重映射这个功能

我们知道每个内置外设都有若干个输入输出引脚，一般这些引脚的输出脚位都是固定不变的，为了让设计工程师可以更好地安排引脚的走向和功能，在 STM32 中引入了外设引脚重映射的概念，即一个外设的引脚除了具有默认的脚位外，还可以通过设置重映射寄存器的方式，把这个外设的引脚映射到其它的脚位。

比如设计的这个电路板，要 3 个串口接口，并且这 2 个串口要连在一起，那么 STM32 芯片本身的 2 个串口接口的管脚分布到各处的，如果按照实际管脚功能来设计电路板，可能不太好走线，这样可以通过重映射的功能，把另外好连的管脚连在一起，我们通过程序去把原来不是串口的管脚初重映射成串口即可；这样就只需要修改软件就可以，根本不用改变硬件，这样就增加了芯片的灵活性，也增加了硬件设计者以及产品的灵活性。

4.8.4 举例说明

在这里我们随便举一个例子，可能我们的芯片不是这个型号，但是都是同样的原理，就跟九阳神功一样，神舟团队认为，这样举的例子才更有可重用性，你学懂了就是真的懂了，大家可以查看一下 STM32 芯片手册，我们这里举的是 STM32F103xC 中有关 USART3 引脚的摘要片段：

Table 5. High-density STM32F103xx pin definitions (continued)

| Pins | | | | | | Pin name | Type ⁽¹⁾ | I/O Level ⁽²⁾ | Main function ⁽³⁾ (after reset) | Alternate functions ⁽⁴⁾ | |
|----------|----------|----------|--------|---------|---------|-------------------|---------------------|--------------------------|---|------------------------------------|-----------|
| LFBGA144 | LFBGA100 | WL CSP64 | LQFP64 | LQFP100 | LQFP144 | | | | | Default | Remap |
| M9 | J7 | G3 | 29 | 47 | 69 | PB10 | I/O | FT | PB10 | I2C2_SCL/USART3_TX ⁽⁸⁾ | TIM2_CH3 |
| M10 | K7 | F3 | 30 | 48 | 70 | PB11 | I/O | FT | PB11 | I2C2_SDA/USART3_RX ⁽⁸⁾ | TIM2_CH4 |
| H7 | E7 | H2 | 31 | 49 | 71 | V _{SS_1} | S | | V _{SS_1} | | |
| L12 | W0 | F1 | 30 | 54 | 70 | PD10 | I/O | FT | PD10 | TIM1_CH3N ⁽⁹⁾ | |
| L9 | K9 | - | - | 55 | 77 | PD8 | I/O | FT | PD8 | FSMC_D13 | USART3_TX |
| K9 | J9 | - | - | 56 | 78 | PD9 | I/O | FT | PD9 | FSMC_D14 | USART3_RX |

从这里可以看出，USART3_TX 的默认引出脚是 PB10，USART3_RX 的默认引出脚是 PB11；但经过重映射后，可以变更 USART3_TX 的引出脚为 PD8，变更 USART3_RX 的引出脚为 PD9。

STM32 中的很多内置外设都具有重映射的功能，比如 USART、定时器、CAN、SPI、I2C 等，详细请看 STM32 参考手册(RM0008)和 STM32 数据手册。

有些模块(内置外设)的重映射功能还可以有多种选择，下面是 RM0008 上有关 USART3 输入输出引脚的重映射功能表：

Table 44. USART3 remapping

| Alternate function | USART3_REMAP[1:0] = "00" (no remap) | USART3_REMAP[1:0] = "01" (partial remap) ⁽¹⁾ | USART3_REMAP[1:0] = "11" (full remap) ⁽²⁾ |
|--------------------|-------------------------------------|---|--|
| USART3_TX | PB10 | PC10 | PD8 |
| USART3_RX | PB11 | PC11 | PD9 |
| USART3_CK | PB12 | PC12 | PD10 |
| USART3_CTS | | PB13 | PD11 |
| USART3_RTS | PB14 | | PD12 |

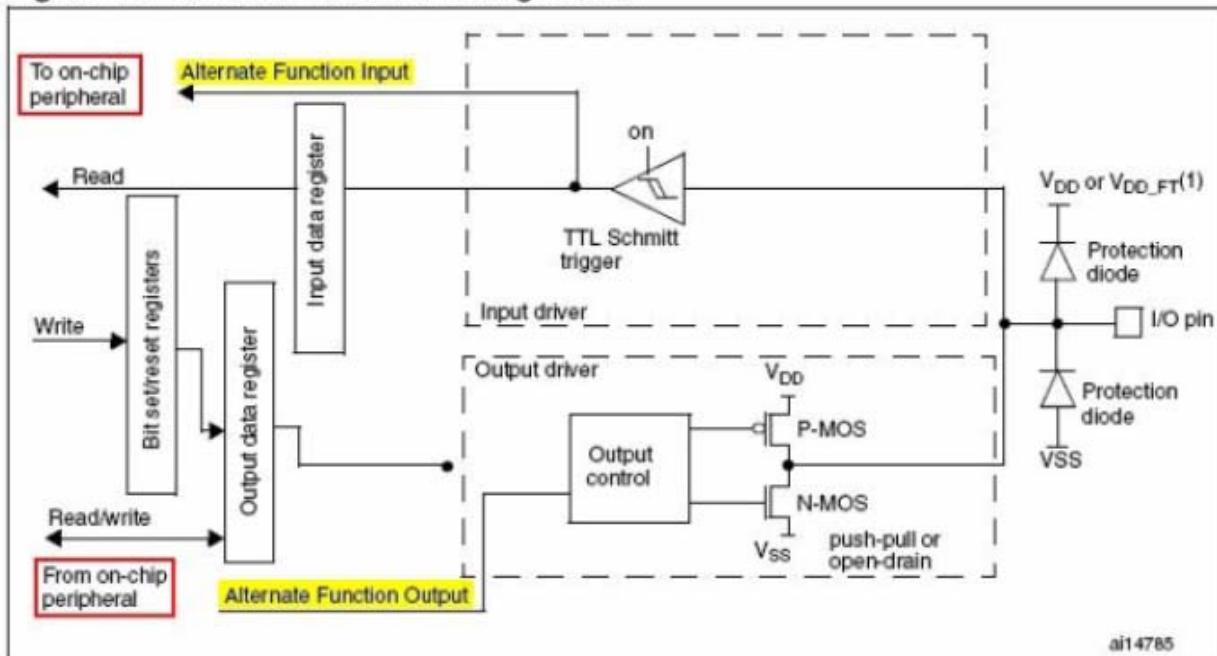
从这个表中可以看出，USART3 的 TX 和 RX 引脚默认的引出脚位是 PB10 和 PB11，根据配置位的设置，可以重映射到 PC10 和 PC11，还可以重映射到 PD8 和 PD9。

4.8.5 深入分析STM32重映射内部架构原理

一个模块的功能引脚不管是从默认的脚位引出还是从重映射的脚位引出，都要通过 GPIO 端口模块实现，相应的 GPIO 端口必须配置为输入(对应模块的输入功能，如 USART 的 RX)或复用输出(对应模块的输出功能，如 USART 的 TX)，对于输出引脚，可以按照需要配置为推挽复用输出或开漏复用输出。

这里就好比，你可以把土地的白菜移到另外一块有土的菜地，它还是可以继续茁壮的成长；但你不能把白菜放在被子里睡觉，让它跟你一样成长，至少要有最基本的土壤；在 STM32 里，一个芯片管脚无非就是输入或者输出，这个基本的属性一定要配对，一个串口打印输出，如果你把管脚配置成输入，那一定是不行的，也不可能配置成功的。

Figure 17. Alternate function configuration



1. V_{DD_FT} is a potential specific to five-volt tolerant I/Os and different from V_{DD} .

上图是 STM32 的 GPIO 端口模块，使用复用功能时的配置。从图中可以看出，配置为复用输出时，该端口对应的 GPIO 输出功能将不起作用。例如当配置 PB10 对应的引脚为复用输出功能时，操作 PB10 对应的输出寄存器将不影响引脚上的信号。

从图中还可以看出，普通的 GPIO 端口输入功能与复用的输入功能的配置方式没有分别，这意味着在使用引脚的复用输入功能时，可以在这个引脚的输入寄存器上读出引脚上的信号。例如在使能了 USART3 模块时，可以读 GPIOB_IDR 寄存器，得到 PB11 信号线上的当前状态。

有不少引脚上配备了来自多个模块的复用功能引出脚，例如本文第一张图中显示的 PB10，默认复用功能就有 I2C2_SCL 和 USART3_TX 两个功能，TIM2 重映射后，TIM2_CH3 也使用 PB10 的复用功能。

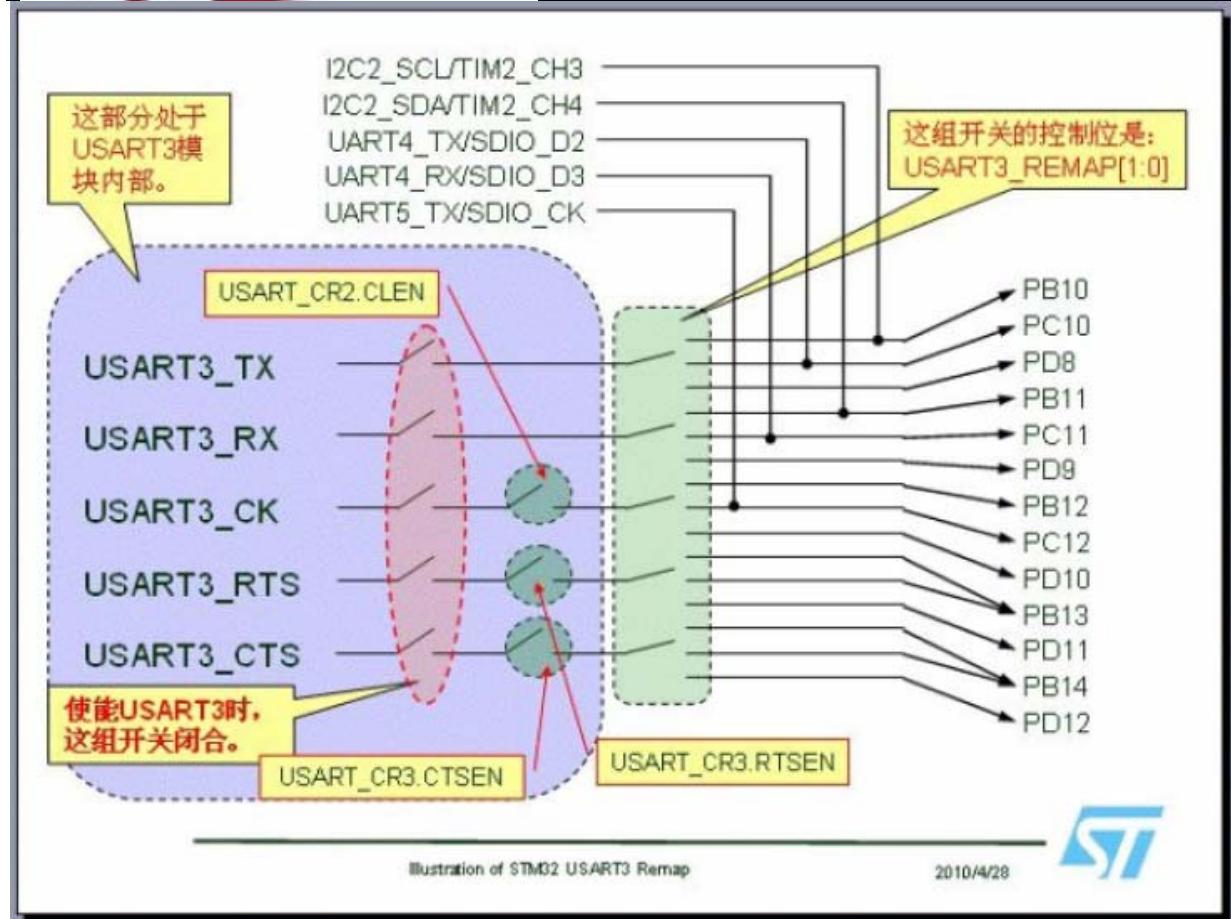
在使用引脚的复用功能时，需要注意在软件上只可以使能一个外设模块，否则在引出脚上可能产生信号冲突。例如，如果使能了 USART3 模块，同时没有对 USART3 进行重映射配置，则不可以使能 I2C2 模块；同理如果需要使用 I2C2 模块，则不能使能 USART3 模块。但是如果配置了 USART3 的引脚重映射，USART3 的 TX 和 RX 信号将从 PC10 和 PC11，或 PD8 和 PD9 引出，避开了 I2C2 使用的 PB10 和 PB11，这时就可以同时使用 I2C2 模块和 USART3 模块了。

USART3 模块共有 5 个信号，分别为 TX、RX、CK、CTS 和 RTS，从上面给出的第二张图中可以看出，重映射是对所有信号同时有效。

这 5 个信号中，在使能了 USART3 模块后，只有 TX 和 RX 是始终与对应的引出脚相连，而其它 3 个信号分别有独立的控制位，控制它们是否与外部引脚相连，如果程序中不使用某个信号的功能，则可以关闭这个信号的功能，对应的引脚可以做为其它功能的引出脚。例如，当关闭了 USART3 的 CK、CTS 和 RTS 功能并且没有重映射 USART3 时，PB12、PB13 和 PB14

可以作为通用输入输出端口使用，也可以作为其它模块的复用功能引出脚。

下面这张图是一个内部控制连接的等效示意图，它并不表示真正的内部连接，但可以有效地帮助理解重映射和复用引脚的概念。图中右边引出的信号，分别连接到了本文第三张图的输入输出模块。



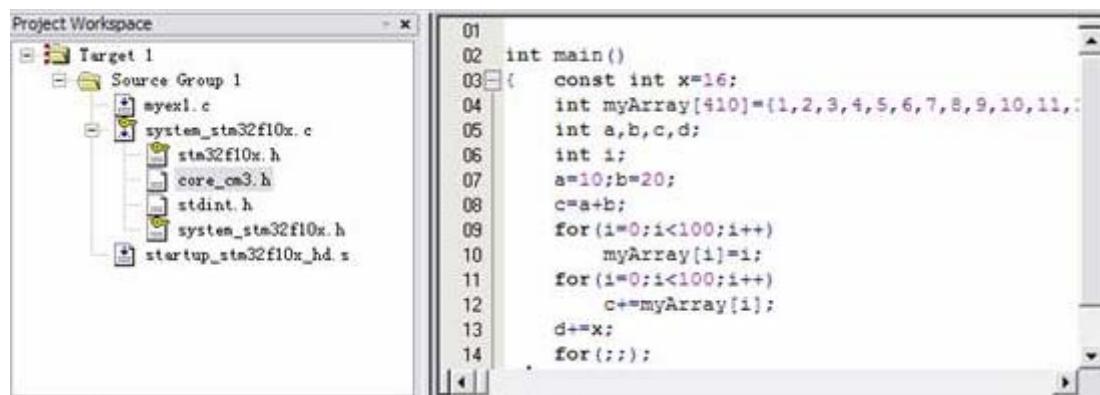
4.8.6 STM32重映射关键指点

这里所说明的原理，如果没看明白没有关系，可以先继续往下学习，这样的概念有个映像就可以，这个知识说明透彻之后，其他的只能在实际的代码中学习；你完全可以打开 STM32 的芯片手册，然后请注意观察那些管脚第一个主功能，以及重映射功能是什么，然后看看代码如何去控制他们的，有疑问再回来查看这一个章节，这才是真正的学习成功之道。

4.9 STM32的内存管理研究（KEIL编程环境下）

4.9.1 研究意义

4.9.2 举例说明并详细分析



非常简单的一个工程，没有用到任何 IO 操作，与 STM32 有关的仅仅只有芯片的选择，即其 SRAM 大小有区别。上图是工程示意图，从图中可以看出，除了自己编写的代码外，仅仅增加了 2 个文件，

即 system_stm32f10x.c 和 startup_stm32f10x_hd.s, 其中为了对 startup_stm32f10x_hd.s 进行修改, 将其从库文件夹复制到了项目文件夹中。

4.9.3 举例分析

代码 1

```
int main()
{
    int a,b,c,d;
    a=10;b=20;
    c=a+b;
    for(;;);
}

myex1.c(3): warning: #550-D: variable "c" was set but never used
linking...
Program Size: Code=796 RO-data=336 RW-data=20 ZI-data=1636
FromELF: creating hex file...
"myex1.axf" - 0 Error(s), 1 Warning(s).
```

代码 2

```
int main()
{
    const int x=16;
    int a,b,c,d;
    a=10;b=20;
    c=a+b;
    for(;;);
}

myex1.c(2): warning: #177-D: variable "x" was declared but never referenced
myex1.c(3): warning: #550-D: variable "c" was set but never used
linking...
Program Size: Code=800 RO-data=336 RW-data=20 ZI-data=1636
FromELF: creating hex file...
"myex1.axf" - 0 Error(s), 2 Warning(s).
```

说明:

- (1) Code 增加了 4 字节
- (2) 其余没有任何变化

代码 3

```
int main()
{
    const int x=16;
```

```
int myArry[100];
int i;
int a,b,c,d;
a=10;b=20;
c=a+b;
for(i=0;i<100;i++)
    myArry[i]=i;
for(;;);
}

myex1.c(2): warning: #177-D: variable "x" was declared but never referenced
myex1.c(3): warning: #550-D: variable "myArry" was set but never used
myex1.c(5): warning: #550-D: variable "c" was set but never used
myex1.c(5): warning: #177-D: variable "d" was declared but never referenced
linking...
```

Program Size: Code=816 RO-data=336 RW-data=20 ZI-data=1636

FromELF: creating hex file...

"myex1.axf" - 0 Error(s), 4 Warning(s).

分析：程序中增加了数组 myArry，Code 增加为 816 字节，但是 RO-data 等仍未变化

代码 4

```
int main()
{
    const int x=16;
    int myArry[100]={1,2,3,4,5,6};
    int i;
    int a,b,c,d;
    a=10;b=20;
    c=a+b;
    for(i=0;i<100;i++)
        myArry[i]=i;
    for(;;);
}

myex1.c(2): warning: #177-D: variable "x" was declared but never referenced
myex1.c(3): warning: #550-D: variable "myArry" was set but never used
myex1.c(5): warning: #550-D: variable "c" was set but never used
myex1.c(5): warning: #177-D: variable "d" was declared but never referenced
linking...
```

Program Size: Code=1024 RO-data=360 RW-data=20 ZI-data=1636

FromELF: creating hex file...

"myex1.axf" - 0 Error(s), 4 Warning(s).

分析：

(1) 由于 myArry 作了初始化，因此 RO-data 增加了 $360-336=24$ 字节。原因是 32 位机中 int 型变量是 32 位的，占 4 字节，所以初始 6 个值后，增加了 24 字节。

(2) 再增加初始化变量的数量，则 RO-data 随之增加，而 Code 不再变化，也就是 Code 由代码 3 的 816 字节增加到 1024 字节，是增加了初始化处理的代码量。

根据以上分析，似乎与已知资料有冲突。

RO 是程序中的指令和常量

RW 是程序中的已初始化变量

ZI 是程序中的未初始化的变量

由以上 3 点说明可以理解为：

RO 就是 readonly,

RW 就是 read/write,

ZI 就是 zero

如果按此说明，增加变量应该增加 RO，但从代码 1 到代码 2 的变化来看，仅是增加了 Code，却没有增加 RO。

初始化变量时，应该增加 RW，但是从代码 2~代码 4，RW 却没有任何变化。

看来这个说法只能适用于 ARM 芯片，即运行时需要将代码调入 RAM 运行的芯片，对于 STM32 这类芯片并不完全适用。

4.9.4 观察堆栈

1) 当使用 int myArray[300]时：

| | |
|---------|--------------------|
| x | 0x00000010 |
| myArray | 0x200001B4 [...] |

2) 当使得 int myArray[100]时：

| | |
|---------|--------------------|
| x | 0x00000010 |
| myArray | 0x200004D4 [...] |

3) 当使得 int myArray[450]时：

当然，执行是错误的，当 int myArray[409]时：正指向 0x2000000，去掉其他变量，对于这个地址没有影响；

| | |
|---------|--------------------|
| x | 0x00000010 |
| myArray | 0x1FFFFF5C [...] |

堆栈应该是向下生成的，而且与芯片无关，无论选择 6K RAM 还是 48K RAM 都是如此，且当数组再大时，就会将地址置于小于 0x2000000 的地址，但编译并不报错。

4) 代码 5

```
int myArray[400]={1,2,3,4,5,6,7,8,9,10,11,12,13,14};
```

```
int main()
```

```
{
```

```
    const int x=16;
```

```

int a,b,c,d;
int i;
a=10;b=20;
c=a+b;
for(i=0;i<100;i++)
    myArray[i]=i;
for(i=0;i<100;i++)
    c+=myArray[i];
d+=x;
for(;;);
}

```

编译结果:

compiling myex1.c...

linking...

Program Size: Code=876 RO-data=336 RW-data=1620 ZI-data=1636

FromELF: creating hex file...

"myex1.axf" - 0 Error(s), 0 Warning(s).

分析:

本段程序将数组作为全局变量来定义，情况立即发生了变化。RW-data 变成了 1620。其中的 1600 应该是这个数组增加的 $4 \times 400 = 1600$ ，而 20 则是代码 1~代码 4 中一直都有有的。

经查验资料，栈的大小应该在启动代码中修改。

```

myex1.c
system_stm32f10x.c
  stm32f10x.h
  core_cm3.h
  stdint.h
  system_stm32f10x.h
  startup_stm32f10x_hd.s

034  Stack_Size      EQU      0x00000400
035
036
037          AREA     STACK, NOINIT, READWRITE, ALIGN=3
038  Stack_Mem      SPACE   Stack_Size
039  __initial_sp
040
041  ; <h> Heap Configuration
042  ; <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
043  ; </h>

```

更改这个：startup_stm32f10x_hd.s 可以更改栈的大小。

RO 是程序中的指令和常量(Code + RO Data); RW 是程序中的已初始化变量; ZI 是程序中的未初始化的变量；由以上 3 点说明可以理解为：

- 1) RO 就是 readonly,
- 2) RW 就是 read/write,
- 3) ZI 就是 zero

简单的说就是在烧写完的时候是：FLASH 中：Code+RO Data+RW Data，运行的时候：RAM: RW Data + ZI Data，当然还要有堆栈的空间。

Program Size: Code=876 RO-data=336 RW-data=1620 ZI-data=1636

FLASH 占 Total ROM Size (Code + RO Data + RW Data) = $876 + 336 + 1620 = 2832 = 2.8\text{KB}$

SRAM 占 Total RW Size (RW Data + ZI Data) = $1620 + 1636 = 3256 = 3.3\text{KB}$

如果你对 ZI-Data 理解还有疑问，可以尝试一下：

int myArray[400]={1,2,3,4,5,6,7,8,9,10,11,12,13,14}; 会增加 RW，但是 int myArray[400]={0}; 就可以增加 ZI 了。

4.10 STM32芯片加密解密

4.10.1 关于芯片加密的定义

芯片解密是指从已经被加密了的芯片里，把存储的代码拷贝出来。嵌入了程序代码的芯片有很多种，而 MCU 只是其中一种。单片机（MCU）一般都有内部 EEPROM/FLASH 供用户存放程序和工作数据。为了防止未经授权访问或拷贝单片机的机内程序，大部分单片机都带有加密锁定位或者加密字节，以保护片内程序。如果在编程时加密锁定位被使能（锁定），就无法用普通编程器直接读取单片机内的程序，这就叫单片机加密或芯片加密。单片机攻击者借助专用设备或者自制设备，利用单片机芯片设计上的漏洞或软件缺陷，通过多种技术手段，就可以从芯片中提取关键信息，获取单片机内程序这就叫芯片解密。

芯片解密又叫单片机解密，单片机破解，芯片破解，IC 解密，但是这严格说来这几种称呼都不科学，但已经成了习惯叫法，我们把 CPLD 解密，DSP 解密都习惯称为芯片解密。单片机只是能装载程序芯片的其中一个类。能烧录程序并能加密的芯片还有 DSP，CPLD，PLD，AVR，ARM 等。也有专门设计有加密算法用于专业加密的芯片或设计验证厂家代码工作等功能芯片，该类芯片也能实现防止电子产品复制的目的。

4.10.2 关于芯片解密方法的理论总结

1. 软件攻击

该技术通常使用处理器通信接口并利用协议、加密算法或这些算法中的安全漏洞来进行攻击。软件攻击取得成功的一个典型事例是对早期 ATMEL AT89C 系列单片机的攻击。攻击者利用了该系列单片机擦除操作时序设计上的漏洞，使用自编程序在擦除加密锁定位后，停止下一步擦除片内程序存储器数据的操作，从而使加过密的单片机变成没加密的单片机，然后利用编程器读出片内程序。

至于在其他加密方法的基础上，可以研究出一些设备，配合一定的软件，来做软件攻击。

近期国内出现了一种泰斗科技 51 芯片解密设备（成都一位高手搞出来的），这种解密器主要针对 SyncMos, Winbond，在生产工艺上的漏洞，利用某些编程器定位插字节，通过一定方法查找芯片中是否有连续空位，也就是说查找芯片中连续的 FF FF 字节，插入的字节能够执行把片内的程序送到片外的指令，然后用解密的设备进行截获，这样芯片内部的程序就被解密完成了。

2. 电子探测攻击

该技术通常以高时间分辨率来监控处理器在正常操作时所有电源和接口连接的模拟特性，并通过监控它的电磁辐射特性来实施攻击。因为单片机是一个活动的电子器件，当它执行不同的指令时，对应的电源功率消耗也相应变化。这样通过使用特殊的电子测量仪器和数学统计方法分析和检测这些变化，即可获取单片机中的特定关键信息。

至于 RF 编程器可以直接读出老的型号的加密 MCU 中的程序，就是采用这个原理。

3. 过错产生技术

该技术使用异常工作条件来使处理器出错，然后提供额外的访问来进行攻击。使用最广泛的过错产生攻击手段包括电压冲击和时钟冲击。低电压和高电压攻击可用来禁止保护电路工作或强制处理器执行错误操作。时钟瞬态跳变也许会复位保护电路而不会破坏受保护信息。电源和时钟瞬态跳变可以

在某些处理器中影响单条指令的解码和执行。

4.探针技术

该技术是直接暴露芯片内部连线，然后观察、操控、干扰单片机以达到攻击目的。

为了方便起见，人们将以上四种攻击技术分成两类，一类是侵入型攻击（物理攻击），这类攻击需要破坏封装，然后借助半导体测试设备、显微镜和微定位器，在专门的实验室花上几小时甚至几周时间才能完成。所有的微探针技术都属于侵入型攻击。另外三种方法属于非侵入型攻击，被攻击的单片机不会被物理损坏。在某些场合非侵入型攻击是特别危险的，这是因为非侵入型攻击所需设备通常可以自制和升级，因此非常廉价。

大部分非侵入型攻击需要攻击者具备良好的处理器知识和软件知识。与之相反，侵入型的探针攻击则不需要太多的初始知识，而且通常可用一整套相似的技术对付宽范围的产品。因此，对单片机的攻击往往从侵入型的反向工程开始，积累的经验有助于开发更加廉价和快速的非侵入型攻击技术。

4.10.3 常规芯片解密过程

侵入型攻击的第一步是揭去芯片封装（简称“开盖”有时候称“开封”，英文为“DECAP”，decapsulation）。有两种方法可以达到这一目的：第一种是完全溶解掉芯片封装，暴露金属连线。第二种是只移掉硅核上面的塑料封装。第一种方法需要将芯片绑定到测试夹具上，借助绑定台来操作。第二种方法除了需要具备攻击者一定的知识和必要的技能外，还需要个人的智慧和耐心，但操作起来相对比较方便，完全家庭中操作。

芯片上面的塑料可以用小刀揭开，芯片周围的环氧树脂可以用浓硝酸腐蚀掉。热的浓硝酸会溶解掉芯片封装而不会影响芯片及连线。该过程一般在非常干燥的条件下进行，因为水的存在可能会侵蚀已暴露的铝线连接（这就可能造成解密失败）。

接着在超声池里先用丙酮清洗该芯片以除去残余硝酸，并浸泡。

最后一步是寻找保护熔丝的位置并将保护熔丝暴露在紫外光下。一般用一台放大倍数至少 100 倍的显微镜，从编程电压输入脚的连线跟踪进去，来寻找保护熔丝。若没有显微镜，则采用将芯片的不同部分暴露到紫外光下并观察结果的方式进行简单的搜索。操作时应用不透明的纸片覆盖芯片以保护程序存储器不被紫外光擦除。将保护熔丝暴露在紫外光下 5~10 分钟就能破坏掉保护位的保护作用，之后，使用简单的编程器就可直接读出程序存储器的内容。

对于使用了防护层来保护 EEPROM 单元的单片机来说，使用紫外光复位保护电路是不可行的。对于这种类型的单片机，一般使用微探针技术来读取存储器内容。在芯片封装打开后，将芯片置于显微镜下就能够很容易的找到从存储器连到电路其它部分的数据总线。由于某种原因，芯片锁定在编程模式下并不锁定对存储器的访问。利用这一缺陷将探针放在数据线的上面就能读到所有想要的数据。在编程模式下，重启读过程并连接探针到另外的数据线上就可以读出程序和数据存储器中的所有信息。

还有一种可能的攻击手段是借助显微镜和激光切割机等设备来寻找保护熔丝，从而寻查和这部分电路相联系的所有信号线。由于设计有缺陷，因此，只要切断从保护熔丝到其它电路的某一根信号线（或切割掉整个加密电路）或连接 1~3 根金线（通常称 FIB：focused ion beam），就能禁止整个保护功能，这样，使用简单的编程器就能直接读出程序存储器的内容。

虽然大多数普通单片机都具有熔丝烧断保护单片机内代码的功能，但由于通用低档的单片机并非定位于制作安全类产品，因此，它们往往没有提供有针对性的防范措施且安全级别较低。加上单片机应用场合广泛，销售量大，厂商间委托加工与技术转让频繁，大量技术资料外泻，使得利用该类芯片的设计漏洞和厂商的测试接口，并通过修改熔丝保护位等侵入型攻击或非侵入型攻击手段来读取单片机的内部程序变得比较容易。

4.10.4 增加芯片解密难度的一些建议总结

任何一款单片机从理论上讲，攻击者均可利用足够的投资和时间使用以上方法来攻破。这是系统

设计者应该始终牢记的基本原则。因此，作为电子产品的设计工程师非常有必要了解当前单片机攻击的最新技术，做到知己知彼，心中有数，才能有效防止自己花费大量金钱和时间辛辛苦苦设计出来的产品被人家一夜之间仿冒的事情发生。根据解密实践提出下面建议：

(1) 在选定加密芯片前，要充分调研，了解芯片解密技术的新进展，包括哪些单片机是已经确认可以破解的。尽量不选用已可破解或同系列、同型号的芯片选择采用新工艺、新结构、上市时间较短的单片机，如可以使用 ATMEGA88PA，这种国内破解的费用一需要 6K 左右，另外相对难解密的有 ST12 系列，dsPIC30F 系列等；其他也可以和 CPLD 结合加密，这样解密费用很高，解密一般的 CPLD 也要 1 万左右。

(2) 尽量不要选用 MCS51 系列单片机，因为该单片机在国内的普及程度最高，被研究得也最透。

(3) 产品的原创者，一般具有产量大的特点，所以可选用比较生僻、偏冷门的单片机来加大仿冒者采购的难度，选用一些生僻的单片机，比如 ATTINY2313,AT89C51RD2,AT89C51RC2, motorola 单片机等比较难解密的芯片，目前国内会开发使用熟悉 motorola 单片机的人很少，所以破解的费用也相当高，从 3000~3 万左右。

(4) 在设计成本许可的条件下，应选用具有硬件自毁功能的智能卡芯片，以有效对付物理攻击；另外程序设计的时候，加入时间到计时功能，比如使用到 1 年，自动停止所有功能的运行，这样会增加破解者的成本。

(5) 如果条件许可，可采用两片不同型号单片机互为备份，相互验证，从而增加破解成本。

(6) 打磨掉芯片型号等信息或者重新印上其它的型号，以假乱真（注意，反面有 LOGO 的也要抹掉，很多芯片，解密者可以从反面判断出型号，比如 51,WINBOND,MDT 等）。

(7) 可以利用单片机未公开,未被利用的标志位或单元,作为软件标志位。

(8) 利用 MCS-51 中 A5 指令加密，其实世界上所有资料,包括英文资料都没有讲这条指令,其实这是很好的加密指令，A5 功能是二字节空操作指令加密方法在 A5 后加一个二字节或三字节操作码,因为所有反汇编软件都不会反汇编 A5 指令,造成正常程序反汇编乱套,执行程序无问题仿制者就不能改变你的源程序。

(9) 你应在程序区写上你的大名单位开发时间及仿制必究的说法,以备获得法律保护；另外写上你的大名的时候，可以是随机的，也就是说，采用某种算法，外部不同条件下，你的名字不同，比如等，这样比较难反汇编修改。

(10) 采用高档的编程器，烧断内部的部分管脚，还可以采用自制的设备烧断金线，这个目前国内几乎不能解密，即使解密，也需要上万的费用，需要多个母片。

(11) 采用保密硅胶（环氧树脂灌封胶）封住整个电路板，PCB 上多一些没有用途的焊盘，在硅胶中还可以掺杂一些没有用途的元件，同时把 MCU 周围电路的电子元件尽量抹掉型号。

(12) 对 SyncMos,Winbond 单片机，将把要烧录的文件转成 HEX 文件,这样烧录到芯片内部的程序空位自动添 00，如果你习惯 BIN 文件，也可以用编程器把空白区域中的 FF 改成 00,这样一般解密器也就找不到芯片中的空位,也就无法执行以后的解密操作。

(13) 比较有水平的加密例如：18F4620 有内部锁相环可以利用 RC 震荡产生高精度的时钟，利用上电时擦除 18F4620 的内部数据，所以导致解密出来的文件根本不能用。

(14) NEC 系列单片机作为日系芯片的代表，单片机中设计了充足的保护措施来保证其程序代码的安全，同时，该系列单片机没有 PROGRAM READ 功能，因此无法利用编程器将程序读出。（注：用编程器给芯片编程时的校验功能并不是将程序读出来进行校验，而是编程器将数据送给芯片，由芯片内核独立完成与存储区数据的比较，然后将比较结果返回给编程器）。

当然，要想从根本上防止单片机被解密，那是不可能的，加密技术不断发展，解密技术也不断发展，至今不管哪个单片机，只要有人肯出钱去做，基本都可以做出来，只不过代价高低和周期长短的问题，编程者还可以从法律的途径对自己的开发作出保护（比如专利）。

4.10.5 STM32加密思路-01 串口ISP设置加密

STM32 的加密，最基本的方法是置读保护，这样可以防止外部工具非法访问，在 STM32 官网发布的 串口 ISP 软件中有置读保护和加密选项，选择一个就可以了，这样外部工具就无法对 FLASH 进行读写操作，但我要重新烧写 FLASH 怎么办？只能清读保护，而清读保护后，芯片内部会自动擦除 FLASH 全部内容。

4.10.6 STM32加密思路-02 软件加密

在软件里做加密，比如利用 CPU 的唯一的 96 位 ID（请见神舟 III 号的例程“产品唯一身份标识（Unique Device ID）实验（96 位唯一 ID 实验）”，或者甚至利用网卡的 MAC，如果说人家反汇编破解你的程序，那是可以做到的，但是成本没法估算，没人愿意出钱尝试，也就起到保护作用了，软件加密，可以看它的闪存编程手册。

利用 STM32 的 UID 加密，具体程序可以做到比如 main 函数开始的时候，可以加一句：

```
if (UID == 正确的 UID) 运行后面代码;  
else 不理会/运行错误的代码;
```

这里还可以迷惑对手，可以运行一段的假的代码，模拟产品正常启动的样子，但就是不运行核心部分的代码，这样可以使得解密人员误认为解密成功了，为对手造成一些经济上的成本。

采用芯片内的唯一 ID 来加密，在程序里识别芯片的 ID，如果 ID 不对，则程序不运行，当然，这样也是有缺陷的，因为每个芯片的 ID 不一样，因此对应的程序也应该不一样，那如何处理呢？有人建议说：采购的时候，产品同批生产的 ID 号应该是连续的，可以通过判别 ID 的范围；还有人说，在烧录工具里做一个算法，读取芯片 ID，再修改相应的二进制文件。当然还会有很多种方法，这里不展开讨论。

4.10.7 STM32加密思路-03 外置ID芯片

可以考虑外置 ID 芯片，如果成本没有问题，可以考虑增加一颗 CPU，两颗 CPU 相互进行验证，才执行真正的程序，增加解密的成本和难度，而且这颗芯片可以是不常用的芯片，或者是型号不同的芯片，这里面很多的想象空间，可以利用产品的特点，来寻找合适的芯片进行处理。

4.10.8 STM32加密思路-04 程序自毁

可以考虑设置某种检测机制，发现异常时程序自毁，比如程序一启动，如果发现芯片某几个管脚的电平出现变化，比如已经封死的 JTAG 管脚出现了其他异常的电平进入，那么就自毁程序，拷贝一段 FF 去刷自己的 FLASH，只要覆盖一小段程序就足以让破解者无法成功破解。

这些预先可以根据自身的产品来设置一些场景，出现这些情况就做相应的急救措施。

4.10.9 STM32加密思路-05 磨IC型号

磨掉 IC 的型号，重新打一个错误的上去；还可以在周围的电阻电容上动手脚，打错的值上去，使得破解成功后，周围的电路有个关键部分的配件的值不对，导致产品运行不正常，增加破解的难度。

4.10.10 STM32加密思路-06 高端硬件加密

目前 STM32 更多高端的芯片增加了硬件的加密，比如 STM32F4XXX 具有个 PDR 寄存器，可以设置三级加密，如果你代码都写好了，直接设置到 LEVE2，那就 stm32 就下载不进去了，直接封锁死，暂时没有听说解决办法，以后有没有不清楚。

还有像 STM32F4XXX 推出的 417 系列带哈希加密的，这些都是 ST 厂商提供的一些解决办法，另外的我们与时俱进进行相应的增加吧。

4.10.11 STM32加密思路-07 AES加密

我们知道，STM32 的内部 FLASH 是用户可编程的，也就是说它支持 IAP，而 IAP 中的 APP 代
嵌入式专业技术论坛（www.armjishu.com）出品

码一般是需要开放的，那么只有保证 BOOT 的代码安全，才能确保不被破解。

前面提到，当 IC 置读保护后，外部工具不能访问内部 FLASH，但 CPU 可以访问，破解者完全可以自己编写一段代码通过 BOOT 下载到 IC 运行，然后在程序中读出你的 BOOT 代码。

所以解决办法就只能加以限制想办法使别人的代码运行不了，才能保证 BOOT 不被读出。

常用的方法是采用加密算法，如 AES；流程如下：

APP 代码加密，下载时，在 BOOT 中解密，这样，只有通过正确加密的 APP 代码才能正常的运行，因此加密的算法就成了你的密钥，而这个是你独有的。

而且这样做好处也可以方便客服人员去客户那升级，也为了保证程序的安全；程序分为两部分，一是 boot 引导程序，二是 app 应用程序部分，产品在出厂时，先用串口 ISP 烧写 boot，烧写的时候置读保护，写完后，可以用 JLINK 测试了一下，确认程序读不出来，说明 FLASH 保护位有效，没有因为后面烧写 APP 而清掉。

关键之处到了，为了安全，这里还需要把 APP 程序用 AES 之类的加密，在 BOOT 中解密写到 APP 区，这样就是说 BOOT 部分需要加入一段解密的代码，这样人家拿到你加密后的 APP 也没用。

ST 官方有一个公开源码版本的 bootloader，很稳定，再找个 AES 在 STM32 进行移植，合体变身，这样一个强大好用的 bootloader 就诞生了；关于 AES 的密码可以由每颗 STM32 都有唯一的 ID 来作为一个唯一输入进行产生，这样可以直接将加密后的文件交给客户自己去下载也是放心的。

这样 BOOT 代码确保不会被读出，APP 也加密了，这样就是目前最安全的模式。

4.11 STM32低功耗经验总结

4.11.1 STM32低功耗实战项目案例故事（转载）

前两个月在公司做了一个低功耗项目，现在功耗最低 10uA 不到，平均功耗 40uA 左右，算是达标了。因为是公司产品，就不方便贴代码、原理图了，该产品是一个小模块，可以方便的嵌入到各种系统里面。跟 NRF2401 类似，是一个读卡器。

做这个项目中间也请了技术支持，因为外围电路芯片的功耗一直降不下来，经过与对方的反复交流，对方提供了低功耗的测试结果、硬件方案、软件方案，经过修改测试，最终成为我们的产品，功耗比较满意。

硬件方案选择的是 STM32，外加某公司的读卡芯片。前期完成了读卡等功能的开发，最后一项开发内容是最艰巨也是最困难的---低功耗。在开发过程中，从硬件设计上不断裁剪元器件，软件上不断精简代码，功耗最低也都保持在 3-4mA 左右，经过许多努力，才解决问题，解决过程如下。

电路设计上，只用到了一个 LED、串口 1、一个模拟 SPI、一个中断线、一个读卡芯片 RESET 线，硬件上就只剩下这么点东西了，这个时候我采用的是待机模式，使用的是读卡芯片的中断接 PA0 唤醒 STM32，在此之前要先使得读卡芯片进入低功耗、然后 STM32 进入低功耗，这一步完成了，貌似没什么问题，功耗确实从几十 mA 骤降到 3mA 左右，开始还挺满意的，但是测试厂商提供的样板，功耗却只有几十 uA，有点郁闷了。为什么会这样？

反复查看硬件、程序，都找不出原因，而且这个时候的效果很烂，根本就不能唤醒，所以我怀疑是读卡芯片一端低功耗有问题，因为我将 PA0 脚直接短接 VCC，这样就可以产生一个边沿触发 STM32 唤醒了，但是用读卡芯片无法唤醒，所以我怀疑是读卡芯片的 RESET 脚电平不对，经检查，确实是因为 RESET 脚加了上拉电阻，读卡芯片是高电平复位，在 STM32 进入待机后，管脚全都浮空了，导致 RESET 被拉高，一直在复位；我去掉上拉电阻，觉得很有希望解决问题了，但是测试结果是：有时候能唤醒，有时候不能，我仔细一想难道是因为 STM32 待机后管脚电平不确定，导致读卡芯片 RESET 脚电平不定，而工作不正常，看样子只有换用其他方案了。后面确实验证了我的想法，使用 STOP 模式后，唤醒问题迎刃而解。

就在关键时刻，芯片原厂火种送炭，送来急需的技术支持资料，一个包含低功耗源代码，赶紧嵌入式专业技术论坛（www.armjishu.com）出品

第 187 页，共 900 页

拿过来测试，先研读下代码，使用的是 STOP 模式，而不是待机模式，使用的是任意外部中断唤醒，这个时候就相当激动啊，赶快下载测试啊，结果功耗确实降了，但还是有 1mA，跟人家一比多了几十倍啊。。。

我第一反应是硬件不对，经过测试修改，首先找到第一个原因，读卡芯片 RESET 管脚上拉电阻又给焊上去了...，拆掉后功耗降到几百 uA，还是不行。。测试过程中，为了去掉 LDO 的干扰，整板采用 3.3V 供电，但是后面经过测试，LDO 的功耗其实也只有 5uA 不到，这 LDO 功耗值得赞一个；虽然结果还是没达到预期，但是看到了希望，胜利就在眼前啊。

为此我反复看了技术支持提供的程序，发现他们的 STM32 的所有管脚的设置都有所考究：(因为公司保密原则，代码中删除掉了关于该读卡芯片的前缀信息等)

```
GPIO_InitTypeDef GPIO_InitStructure;  
/* GPIOA Periph clock enable */  
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);  
/* GPIOB Periph clock enable */  
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);  
/* GPIOC Periph clock enable */  
//RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);  
  
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);  
  
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);  
/////////////////////////////  
//USART1 Port Set  
//TXD  
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  
GPIO_Init(GPIOA, &GPIO_InitStructure);  
//RXD  
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;  
GPIO_Init(GPIOA, &GPIO_InitStructure);  
  
//RST output pushpull mode  
GPIO_InitStructure.GPIO_Pin = TRST;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  
GPIO_Init(PORT1, &GPIO_InitStructure);  
//IRQ input pull-up mode  
GPIO_InitStructure.GPIO_Pin = TIRQ;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;  
GPIO_Init(PORT1, &GPIO_InitStructure);
```

```
//MISO input pull-up mode
GPIO_InitStructure.GPIO_Pin = MISO;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(PORT2, &GPIO_InitStructure);

//NSS,SCK,MOSI output pushpull mode
GPIO_InitStructure.GPIO_Pin = (NSS|SCK|MOSI);
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(PORT2, &GPIO_InitStructure);

#####
//TEST Port set
//TESTO input pushpull mode
GPIO_InitStructure.GPIO_Pin = TESTO;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(TEST_PORT, &GPIO_InitStructure);

#####
//TEST Port set
//TESTI output pushpull mode
GPIO_InitStructure.GPIO_Pin = TESTI;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(TEST_PORT, &GPIO_InitStructure);

#####
//LED Port Set
//LED output pushpull mode
GPIO_InitStructure.GPIO_Pin = LED;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(LED_PORT, &GPIO_InitStructure);

#####
= (GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_8|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_15);
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
GPIO_Init(GPIOA, &GPIO_InitStructure);

= (GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10);
```

```
- GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
```

```
GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
GPIO_InitStructure.GPIO_Pin = (GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15);
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
```

```
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

首先，想 MOSI、SCK、CS、LED、RST 这些管脚应该设置为推挽输出，TXD 设置为复用输出，而 IRQ、RXD、MISO 设置浮空输入，什么都没接的管脚全都设置为下拉输入，而 TESTI、TESO 我一直不解是什么东东，开始就没管，而开始的时候 MISO 我也没怎么注意，设置成上拉输入(而不是浮空输入)，反正大部分按照厂家提供的参考，我并没有照搬，测试效果一样，但功耗确是还有 80-90uA，期间我找了好久没找到原因，给技术支持一看，原来是因为 MISO 没有设置成浮空输入，我是设置成了上拉输入，上拉电阻一直在消耗大约 40uA 的电流。。。

好吧，这是自己不够细心导致的，以后做低功耗的项目管脚配置是个大问题，不能再这么马虎了!!! 我将 MISO 设置成浮空输入之后，最低功耗还是有 40+，离 10uA 的最低功耗还有段距离，到底是为什么呢？

最后我发现，该读卡芯片有个 TESTIN/TESTOUT 管脚，是用来测试用的，出厂后也就用不上了，我也一直以为这两个脚确实没什么用，就没接；可是我发现厂家提供的样板居然接了这两个脚，但是厂商也没说这两个脚接或不接会影响功耗啊，抱着试一试的心态，我把 TESTIN/TESTOUT 两个管脚接到单片机上进行相应的配置，接下来是见证奇迹的时刻了，功耗居然真的、真的降到 10uA 了。。。。。。。 此处省略 n 个字

这时候真的很激动，真的很想骂人啊，坑爹的厂家，为什么不给提示说这两个脚不接单片机会消耗电流呢？（也许是文档里面提到了，但是几百页的文档，还是全英文的，一堆堆的文字，我再看一遍，确实没有提到这两个管脚会有漏电流。）

项目就这样完工了，中间最重要的是技术支持的强力支持，不然项目不能完工了，这个项目低功耗 STM32 方面难度不高，主要是读卡芯片上面的低功耗调试起来问题很多，还是人家原厂的出马才解决了问题，因为众多原因，不能公布该芯片的资料，包括该芯片怎么进入低功耗也无法公开，所以抱歉~~。

关于 STM32 进入低功耗，我简单的总结了一下：

1. 管脚设置，这个很关键，还是跟你电路有关系，外加上拉、下拉电阻切记不能随便加
2. STM32 的 systick clock、DMA、TIM 什么的，能关就全都关掉，STM32 低功耗很简单，关键是外围电路功耗是关键
3. 选择一个低功耗的 LDO，这个项目用到的 LDO 功耗就很不错，静态功耗 10uA 都不到。
4. 确定 STM32 设置没问题，进入低功耗有好几种情况可以选择(睡眠、停机、待机)，我还是推荐选择 STOP 模式，这个我觉的比较好是因为可以任意外部中断都可以唤醒，而且管脚可以保留之前的设置，进入停机模式的代码使用库函数自带的，就一句：

```
PWR_EnterSTOPMode(PWR_Regulator_LowPower, PWR_STOPEntry_WFI);
```

意思是，在进入停机模式之前，也关掉电压调节器，进一步降低功耗，使用 WFI 指令(任意中断唤醒)，但是经过测试，使用 WFE(任事件唤醒)指令效果、功耗一模一样。

最后一步是从 STOP 模式怎么恢复了，恢复其实也很简单，外部中断来了会进入中断函数，然后 STM32 就被唤醒，唤醒还要做一些工作，需要开启外部晶振(当然你也可以选择使用内部自带振荡器)、开启你需要的外设等等。

总之，低功耗关键我觉得还是在于管脚配置，以及你对于外围电路的掌握。

4.11.2 STM32低功耗三种模式

STM32F10xxx 有三种低功耗模式：

- 睡眠模式(Cortex-M3 内核停止，外设仍在运行)
- 停止模式(所有的时钟都以停止)
- 待机模式(1.8V 电源关闭)

此处主要是一些经验的总结，后面带有一些基础例程，大家可以进一步深入进行熟悉，如果遇到看不懂的地方，可以先去学习例程代码，然后再回头来查看此处文档。

4.11.3 STM32低功耗需注意的地方

1) 时钟问题：

STM32 被唤醒以后的时钟自动切换到内部 HIS RC 振荡器，大家都是知道的，RC 振荡器的精度是不高的。而且，休眠前对于时钟的设置都是恢复到复位状态，只是时钟这个地方复位，其他的没有。这也会带来一个问题，可能你休眠前使用的是内部时钟，可是休眠后，时钟却变了，带来的问题就是 UART 和定时器。或许你想不使用 PLL，就是 8M，这样醒来后的时钟 HIS 也是 8M，这样虽然在时钟上没有差别了，但是时钟却不稳定了。UART 波特率肯定不能太高，否则通信会有问题。

2) 醒来时间：这个问题也是个非常大的问题，

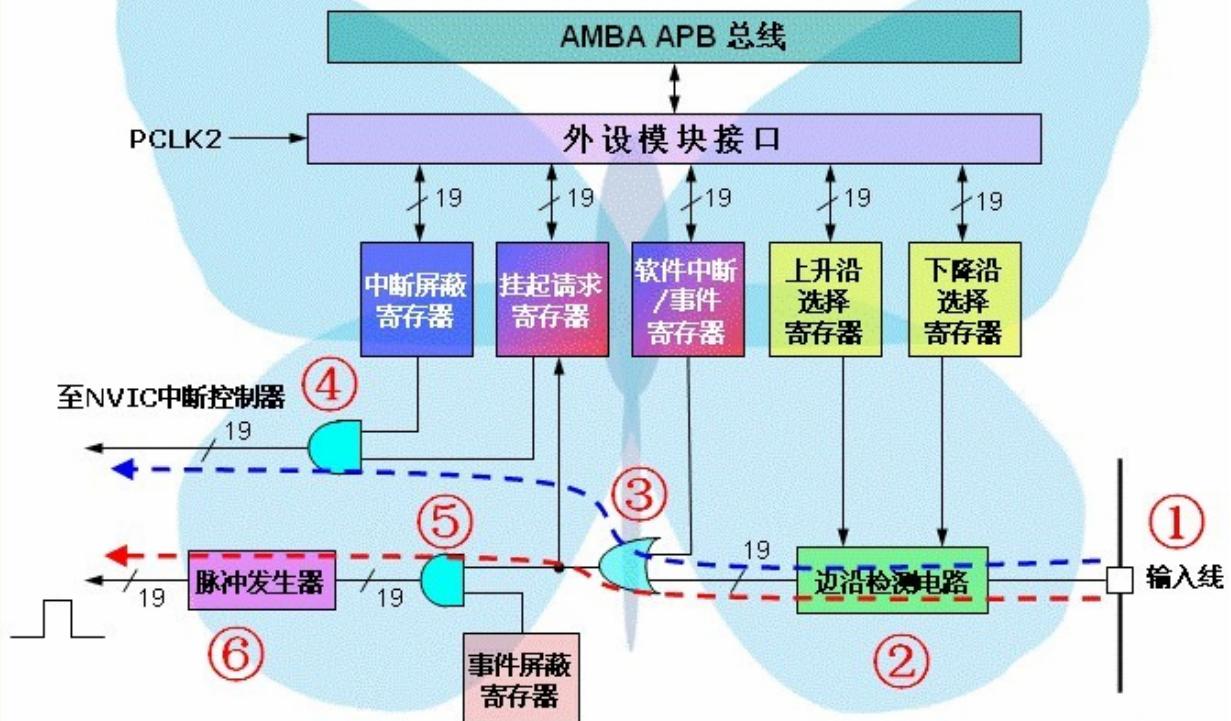
datasheet 上给出的醒来时间是 7us，这个可能真的不假，但是醒来，不能马上干你的活，为什么。初始化 IO，你可能问，我不初始化不行吗，回答应该是否定的。因为，如果你想使用低功耗的话，休眠前 IO 口都应该设置为模拟输入，这样才能达到 datasheet 上的 14uA，但是这样也带来一个问题，那就是初始化 IO，醒来必须要初始化 IO。如果你还想把时钟切换到外部时钟，耗时会更加长，接近 200ms，因为 STM32 会等待外部时钟稳定后才能工作，然后还要在重新初始化所有 IO，这个非常的耗时。可能我只需要醒来 10ms，但是这些活干完就需要 100ms。

3) RTC 唤醒：

RTC 这个也是个问题，为什么？大家需要注意的是 RTC 只能使用报警才能唤醒 MCU，秒中断是不可以唤醒的。并且报警中断必须不停的设置，设置一次只生效一次，中断完了，还需要设置下次中断的时间。并且还有个问题，报警中断必须等到秒中断到了之后才能设置，也就是正好秒寄存器更新了一次的时候设置，这就带来一个问题，等待秒中断。如果睡前还想再能被报警唤醒的话必须重新设置报警中断，而且设置报警中断的时候需要等到秒中断才能设置新的值。这个等待的时间是不定的。可能会几百个毫秒。说以要空空的耗费几百个毫秒等到秒中断标志来设置报警中断。可能我的 MCU 只需要执行 10ms 就需要休眠了。还是要空空的耗费掉几百个毫秒

4.12 STM32的中断与事件关系的区别

外部中断 / 事件控制器框图



这张图是一条外部中断线或外部事件线的示意图，图中信号线上划有一条斜线，旁边标志 19 字样的注释，表示这样的线路共有 19 套。

图中的蓝色虚线箭头，标出了外部中断信号的传输路径，首先外部信号从编号 1 的芯片管脚进入，经过编号 2 的边沿检测电路，通过编号 3 的或门进入中断“挂起请求寄存器”，最后经过编号 4 的与门输出到 NVIC 中断控制器；在这个通道上有 4 个控制选项，外部的信号首先经过边沿检测电路，这个边沿检测电路受上升沿或下降沿选择寄存器控制，用户可以使用这两个寄存器控制需要哪一个边沿产生中断，因为选择上升沿或下降沿是分别受 2 个平行的寄存器控制，所以用户可以同时选择上升沿或下降沿，而如果只有一个寄存器控制，那么只能选择一个边沿了。

接下来是编号 3 的或门，这个或门的另一个输入是“软件中断/事件寄存器”，从这里可以看出，软件可以优先于外部信号请求一个中断或事件，既当“软件中断/事件寄存器”的对应位为“1”时，不管外部信号如何，编号 3 的或门都会输出有效信号。一个中断或事件请求信号经过编号 3 的或门后，进入挂起请求寄存器，到此之前，中断和事件的信号传输通路都是一致的，也就是说，挂起请求寄存器中记录了外部信号的电平变化。外部请求信号最后经过编号 4 的与 NVIC 中断控制器发出一个中断请求，如果中断屏蔽寄存器的对应位为“0”，则该请求信号不能传输到与门的另一端，实现了中断的屏蔽。明白了外部中断的请求机制，很容易理解事件的请求机制了。图中红色虚线箭头，标出了外部事件信号的传输路径，外部请求信号经过编号 3 的或门后，进入编号 5 的与门，这个号 4 的与门，用于引入事件屏蔽寄存器的控制；最后脉冲发生器把一个跳变的信号转变为一个单脉冲，输出到芯片中的其它功能模块。

在这张图上我们也可以知道，从外部激励信号来看，中断和事件是没有分别的，只是在芯片内部分开，一路信号会向 CPU 产生中断请求，另一路信号会向其它功能模块发送脉冲触发信号，其它功能模块如何相应这个触发信号，则由对应的模块自己决定。在图上部的 APB 总线和外设模块接口，是每一个功能模块都有的部分，CPU 通过这样的接口访问各个功能模块，这里就不再赘述了。

第五章 STM32神舟III号 实战篇（寄存器版本）

5.1 通用输入/输出（GPIO）

5.1.1 脚特性

| 型号 | CPU 频率 (MHz) | 程序 空间 (字节) | RAM (字节) | FSMC | 定时器功能 ⁽¹⁾ | | | 串行通信接口 | | | | | 模拟端口 | | | I/O 端口 (通道) | 封装 | | |
|------|--------------------|------------------|-------------|------|-----------------------|-----------------------|-----------|--------|------------------|------------------------------|-----------|------------|---------|------------------|------|-------------------|------|-----|------------------|
| | | | | | 16位普通 (I/C/OC/PWM) | 16位高级 (I/C/OC/PWM) | 16位 基本 | SPI | I ² C | USART ⁽²⁾ UART | USB 全速 | CAN 2.0 | 以太 网 | I ² S | SDIO | ADC | DAC | | |
| 144脚 | STM32F103ZC | 72 | 256K | 48K | ● | 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(21) | 1(2) | 112 | LQFP144/LFBGA144 |
| | STM32F103ZD | 72 | 384K | 64K | ● | 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(21) | 1(2) | 112 | LQFP144/LFBGA144 |
| | STM32F103ZE | 72 | 512K | 64K | ● | 4(16/16/16) | 2(8/8/12) | 2 | 3 | 2 | 3+2 | 1 | 1 | 2 | 1 | 3/(21) | 1(2) | 112 | LQFP144/LFBGA144 |

v

- STM32F103ZET6 总共有 113 个通用输入/输出（GPIO）口
- 每个 I/O 端口位可以自由编程
- I/O 端口寄存器可按 32 位字被访问（不允许半字或字节访问）

5.1.2 GPIO应用领域

- 通用 I/O 口
- 驱动 LED 或其他指示器
- 控制片外器件或片外器件通信
- 检测静态输入

5.1.3 管脚分配

表 1: 端口 A GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PA[15:0] | I/O | 通用输入/输出 PA0 到 PA15 |

表 2: 端口 B GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PB[15:0] | I/O | 通用输入/输出 PB0 到 PB15 |

表 3: 端口 C GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PC[15:0] | I/O | 通用输入/输出 PC0 到 PC15 |

表 4: 端口 D GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PD[15:0] | I/O | 通用输入/输出 PD0 到 PD15 |

表 5: 端口 E GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PE[15:0] | I/O | 通用输入/输出 PE0 到 PE15 |

表 6: 端口 F GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PF[15:0] | I/O | 通用输入/输出 PF0 到 PF15 |

表 7: 端口 G GPIO 管脚描述

| 管脚名称 | 类型 | 描述 |
|----------|-----|--------------------|
| PG[15:0] | I/O | 通用输入/输出 PG0 到 PG15 |

表 8:

| 管脚名称 | 类型 | 描述 |
|------|-----|----|
| NRST | I/O | 复位 |

5.1.4 GPIO管脚内部硬件电路原理剖析

I/O 接口是一颗微控制器必须具备的最基本外设功能。通常在 ARM 里，所有 I/O 都是通用的，称为 GPIO (General Purpose Input/Output)。在 STM32 中，每个 GPIO 端口包含 16 个管脚，如 PA 端口是 PA0~PA15。GPIO 模块支持多种可编程输入/输出管脚，GPIO 模块包含以下特性：

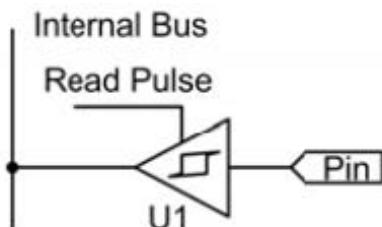
- 1) 可编程控制 GPIO 中断
 - 包括屏蔽中断发生
 - 边沿触发（上升沿、下降沿、双边沿触发）
 - 电平触发（高电平触发、低电平触发）
- 2) 输入/输出管脚最大可承受 5V 电压
- 3) 可通过编程控制 GPIO 管脚配置：
 - 弱上拉或弱下拉电阻
 - 2mA、4mA、8mA 驱动，STM32 芯片管脚驱动最大是 25mA

GPIO 管脚可以被配置为多种工作模式，配置不同的模式实际就是内部的驱动电路有不一样，下面我们分析几种，大家可以从这里了解原理知识：

1. 高阻输入

高阻态是一个数字电路里常见的术语，指的是电路的一种输出状态，既不是高电平也不是低电平，如果高阻态再输入下一级电路的话，对下级电路无任何影响，和没接一样。

电路分析时高阻态可做开路理解。你可以把它看作输出（输入）电阻非常大。他的极限可以认为悬空。也就是说理论上高阻态不是悬空，它是对地或对电源电阻极大的状态。而实际应用上与引脚的悬空几乎是一样的。



如上图所示，为 GPIO 管脚在高阻输入模式下的等效结构示意图， 表示 GPIO 管脚；这是一个管脚的情况，其它管脚的结构也是同样的，输入模式的结构比较简单，就是一个带有施密特触发输入（Schmitt-triggered input）的三态缓冲器（U1），并具有很高的阻抗。施密特触发输入的作用是能将缓慢变化的或者是畸变的输入脉冲信号整形成比较理想的矩形脉冲信号。执行 GPIO 管脚读操作时，在读脉冲（Read Pulse）的作用下会把管脚（Pin）的当前电平状态读到内部总线上（Internal Bus）。

在不执行读操作时，它可以变成高阻抗的状态，使得外部管脚与内部总线之间是隔离的。

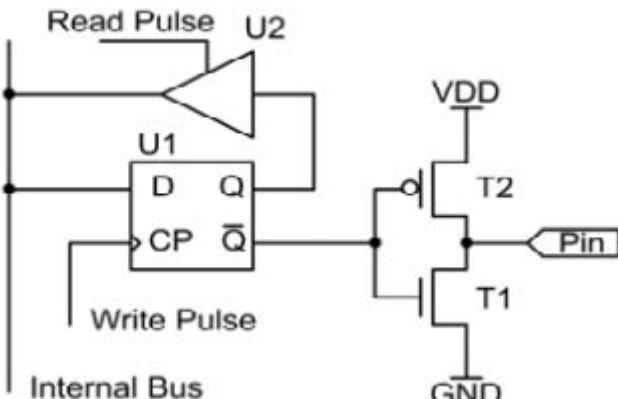
为什么会产生这种高阻抗的管脚设计呢？因为是很多管脚都连在同一根总线上，为了不干扰其他管脚，当某一个管脚在传送数据的适合，其他管脚配置成高阻抗，就不会干扰正在传送数据的管脚了，这样可以很多管脚同时共用一根总线，分时复用。

为减少信息传输线的数目，大多数计算机中的信息传输线采用总线形式，即凡要传输的同类信息都在同一组传输线，且信息是分时传送的。在计算机中一般有三组总线，即数据总线、地址总线和控制总线。为防止信息相互干扰，要求凡挂到总线上的寄存器或存储器等，它的输入输出端不仅能够呈现 0、1 两个信息状态，而且还能产生一种高阻抗状态，即好像它们的输出被开关断开，对总线状态不起作用，此时总线可由其他器件占用。三态缓冲器即可实现上述功能，它除具有输入输出端之外，还有一个控制端，就像一个开关一样，可以控制使其变成高阻抗状态。

2. 推挽输出

推挽输出可以提高输出功率，能够更好驱动外部的设备；推挽输出的原理：在功率放大器电路中大量采用推挽放大器电路，这种电路中用两只三极管构成一级放大器电路，两只三极管分别放大输入信号的正半周和负半周，即用一只三极管放大信号的正半周，用另一只三极管放大信号的负半周，两只三极管输出的半周信号在放大器负载上合并后得到一个完整周期的输出信号。

推挽放大器电路中，一只三极管工作在导通、放大状态时，另一只三极管处于截止状态，当输入信号变化到另一个半周后，原先导通、放大的三极管进入截止，而原先截止的三极管进入导通、放大状态，两只三极管在不断地交替导通放大和截止变化，所以称为推挽放大器。



如上图所示，为 GPIO 管脚在推挽输出模式下的等效结构示意图。U1 是输出锁存器，执行 GPIO 嵌入式专业技术论坛（www.armjishu.com）出品 第 195 页，共 900 页

管脚写操作时，在写脉冲（WritePulse）的作用下，数据被锁存到 Q 和 \bar{Q} 。T1 和 T2 构成 CMOS 反相器，T1 导通或 T2 导通时都表现出较低的阻抗，但 T1 和 T2 不会同时导通或同时关闭，最后形成的是推挽输出。在推挽输出模式下，GPIO 还具有回读功能，实现回读功能的是一个简单的三态门 U2。注意：执行回读功能时，读到的是管脚的输出锁存状态，而不是外部管脚 Pin 的状态。

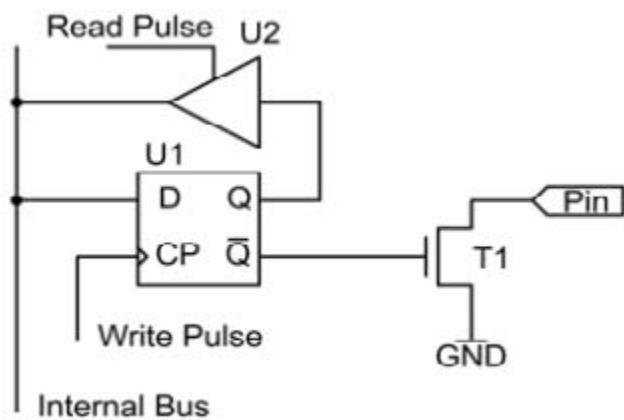
推挽电路是两个参数相同的三极管或 MOSFET，以推挽方式存在于电路中，各负责正负半周的波形放大任务，电路工作时，两只对称的功率开关管每次只有一个导通，所以导通损耗小、效率高。输出既可以向负载灌电流，也可以从负载抽取电流。推拉式输出级既提高电路的负载能力，又提高开关速度。

推挽放大器的输出级有两个“臂”（两组放大元件），一个“臂”的电流增加时，另一个“臂”的电流则减小，二者的状态轮流转换。对负载而言，好像是一个“臂”在推，一个“臂”在拉，共同完成电流输出任务。

3. 开漏输出

开漏输出就是不输出电压，低电平时接地，高电平时不接地。如果外接上拉电阻，则在输出高电平时电压会拉到上拉电阻的电源电压，如果开漏输出的管脚被上拉了，那么这个管脚将一直默认是输出高电平的。

一般来说，开漏是用来连接不同电平的器件，匹配电平用的，因为开漏引脚不连接外部的上拉电阻时，只能输出低电平，如果需要同时具备输出高电平的功能，则需要接上拉电阻，很好的一个优点是通过改变上拉电源的电压，便可以改变传输电平，比如加上上拉电阻就可以提供 TTL/CMOS 电平输出等。

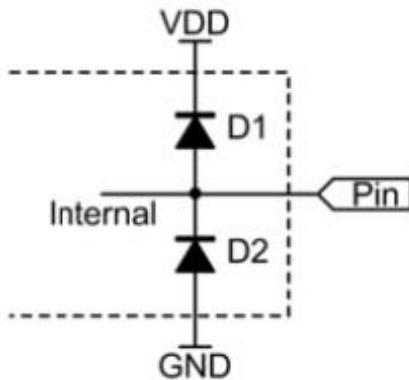


如上图所示，为 GPIO 管脚在开漏输出模式下的等效结构示意图。开漏输出和推挽输出相比结构基本相同，但只有下拉晶体管 T1 而没有上拉晶体管。同样，T1 实际上也是多组可编程选择的晶体管。开漏输出的实际作用就是一个开关，输出“1”时断开、输出“0”时连接到 GND（有一定内阻）。回读功能：读到的仍是输出锁存器的状态，而不是外部管脚 Pin 的状态。因此开漏输出模式是不能用来输入的。

开漏输出的优点是 IC 内部仅需很小的驱动电流就可以了，因为它主要是利用外部电路的驱动能力，这样可以减少 IC 内部的驱动，并且外部需要什么样的电压，就上拉到相应的电压，需要多大的电流，也可以通过改变上拉电阻来调节电流，所以开漏输出是非常灵活的一种输出。

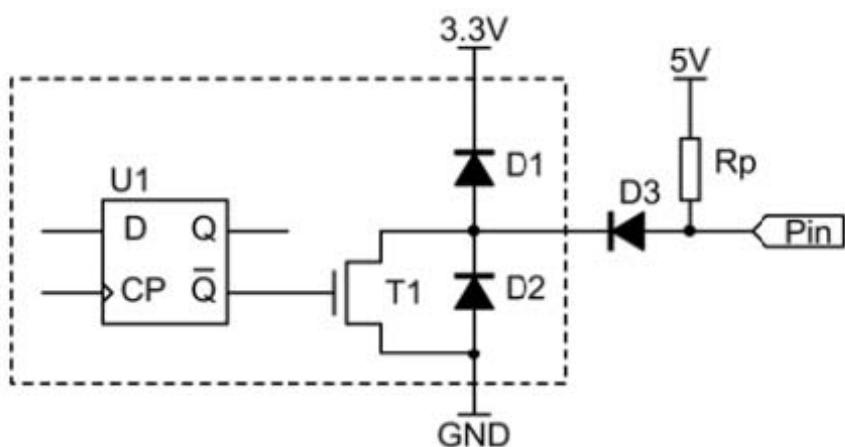
4. 钳位二极管（用来保护 GPIO 管脚）

GPIO 内部具有钳位保护二极管，如下图所示。其作用是防止从外部管脚 Pin 输入的电压过高或者过低。VDD 正常供电是 3.3V，如果从 Pin 输入的信号（假设任何输入信号都有一定的内阻）电压超过 VDD 加上二极管 D1 的导通压降（假定超过 VDD 电压 0.6V 的时候，那么二极管 D1 导通需要 0.6V 的压降），则二极管 D1 导通，这样就会把多余的电流引到 VDD，而真正输入到内部的信号电压不会超过 3.9V ($3.3V + 0.6V = 3.9V$)。同理，如果从 Pin 输入的信号电压比 GND 还低，则由于二极管 D2 的作用，会把实际输入内部的信号电压钳制在 -0.6V 左右。



假设 $VDD=3.3V$, GPIO 设置在开漏模式下, 外接 $10k\Omega$ 上拉电阻连接到 $5V$ 电源, 在输出“1”时, 我们通过测量发现: GPIO 管脚上的电压并不会达到 $5V$, 而是在 $4V$ 上下, 这正是内部钳位二极管在起作用。虽然输出电压达不到满幅的 $5V$, 但对于实际的数字逻辑通常 $3.5V$ 以上就算是高电平了。

如果确实想进一步提高输出电压, 一种简单的方法是先在 GPIO 管脚上串联一只二极管(如 1N4148), 然后再接上拉电阻。参见下图, 框内是芯片内部电路。向管脚写“1”时, T1 关闭, 在 Pin 处得到的电压是 $3.3+VD1+VD3=4.5V$, 电压提升效果明显; 向管脚写“0”时, T1 导通, 在 Pin 处得到的电压是 $VD3=0.6V$, 仍属低电平。



以上这节就是主要介绍芯片管脚的基本驱动原理, 下面开始剖析具体的 STM32 芯片管脚。

5.1.5 STM32的GPIO管脚深入分析

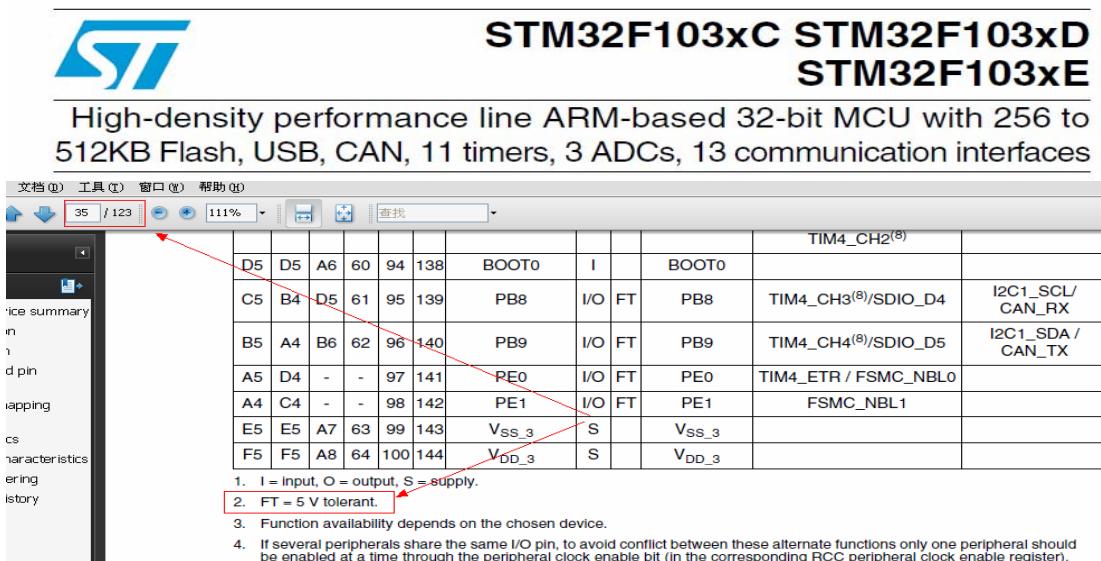
上节我们介绍了芯片管脚实现的硬件原理, 以及一个芯片管脚被配置成不同的模式实际是不同的驱动电路, 这些驱动电路可以适用于不同的场合。STM32 的每个 GPIO 引脚都可以由软件配置成输出(推挽或开漏), 输入(带或不带上拉或下拉)或复用的外设功能端口。多数 GPIO 引脚与数字或模拟的复用外设共用; 除了具有模拟输入(ADC)功能的管脚之外, 其他的 GPIO 引脚都有大电流通过能力, 这些具体如下 8 种模式:

- 1) 输入浮空(这个输入模式, 输入电平必须由外部电路确定, 要根据具体电路, 加外部上拉电阻或下拉电阻, 可以做按键识别)
- 2) 输入上拉(打开 IO 内部上拉电阻)
- 3) 输入下拉(打开 IO 内部下拉电阻)
- 4) 模拟输入(应用 ADC 模拟输入)
- 5) 开漏输出(输出端相当于三极管的集电极. 要得到高电平状态需要上拉电阻才行. 适合于做电流型的驱动, 其吸收电流的能力相对强(一般 20mA 以内). 能驱动大电流和大电压, LED 就

使用这种模式。)

- 6) 推挽式输出 (可以输出高,低电平,连接数字器件。推挽式输出输出电阻小,带负载能力强)
- 7) 推挽式复用功能 (复用是指该引脚打开 remap 功能)
- 8) 开漏复用功能 (复用是指该引脚打开 remap 功能)

每个 IO 口可以自由编程,单 IO 口寄存器必须要按 32 位 bit 被访问。STM32 的很多 IO 口都是 5V 兼容的,这些 IO 口在与 5V 电平的外设连接的时候很有优势,具体哪些 IO 口是 5V 兼容的,可以从该芯片的数据手册管脚描述章节查到(I/O Level 标 FT 的就是 5V 电平兼容的),我们打开 STM32F103ZET 的芯片手册,看到 35 页,对 FT 有详细的解释:



STM32 的每个 IO 端口都有 7 个寄存器来控制 (这里要注意的是关于芯片硬件管脚的信息可以通过芯片手册找到资料,有关控制某个管脚寄存器的说明需要参考《STM32F10xxx 参考手册》);我们可以从手册截图的目录看到 7 个控制 GPIO 管脚的寄存器:



STM32F10xxx 参考手册

翻译说明

本文档是依据 [STM32 Reference Manual \(RM0008\)](#) 翻译的,已经与 2009 年 6 月的英文第 9 版(Doc ID 13902 Rev 9)进行了全面校对,更正了不少以前版本的错误。

在校对即将结束时,ST 于 2009 年 12 月中旬又发布了英文第 10 版(Doc ID 13902 Rev 10),为了与最新的英文版同步,我们按照英文第 10 版结尾的“文档版本历史”中的指示,在翻译的文档中快速地校对更正了对应的部分。由于时间的关系,没有逐字逐句地按照英文第 10 版进行通篇校对,鉴于芯片本身没有改变,我们相信除了“文档版本历史”中指出的差别外,英文第 10 版与英文第 9 版不会再有更多的变化,遂定稿现在这个翻译版本为对应的中文第 10 版文档。

由于我们的水平有限以及文档篇幅的庞大,翻译的过程中难免会有错误和遗漏的地方,希望广大读者们能够及时向我们反馈您在阅读期间所发现的错误和问题,我们会尽快在下一个版本中更正。您可以发邮件到 mcu.china@st.com 向我们提出您的意见和建议,谢谢。

| | |
|--|-----|
| 8.2 GPIO 寄存器描述 | 113 |
| 8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E) | 113 |
| 8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E) | 114 |
| 8.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E) | 114 |
| 8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) | 115 |
| 8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) | 115 |
| 8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E) | 115 |
| 8.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) | 116 |

他们分别是两个端口配置寄存器,一个端口有 0~15 总共 16 个管脚;两个寄存器分别描述输入和

输出的，还有一个端口设置/清除寄存器来负责管脚是输出高电平还是低电平，一个端口清除寄存器和端口配置锁定寄存器。在需要的情况下，I/O 引脚的外设功能可以通过一个特定的操作锁定，以避免意外的写入 I/O 寄存器，这里我们仅介绍常用的几个寄存器，来完成我们的 LED 灯点灯实验的操作；在此，我们可以总结一下 STM32 的 IO 控制寄存器的作用：

- 1) STM32 的 CRL 和 CRH 寄存器主要是用来 **IO 管脚的方向和速率以及何种驱动模式**
- 2) STM32 的 ODR 寄存器是用来控制 **IO 口的输出高电平还是低电平**
- 3) STM32 的 IDR 寄存器主要是用来存储 **IO 口当前的输入状态（高低电平）** 的。
- 4) STM32 的 BSRR 寄存器主要是用来直接对 IO 端某一位直接进行设置和清除操作，通过这个寄存器可以方便的直接修改一个引脚的高低电平
- 5) STM32 的 BRR 寄存器用来清除某端口的某一位位 0，如果该寄存器某位为 0，那么它所对应的那个引脚位不产生影响；如果该寄存器某位为 1，则清除对应的引脚位。
- 6) STM32 的 LCKR 用来锁定端口的配置，当对相应的端口位执行了 LOCK 序列后，在下次系统复位之前将不能再更改端口位的配置。

下面开始分析常用的两个 32 位**配置寄存器** GPIOx_CRL 和 GPIOx_CRH, CRL 和 CRH 控制着每个 IO 口的模式及输出速率，我们下面来介绍一下两个寄存器，接下来我们看看端口低配置寄存器 CRL 的描述，如下图所示：

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-----------|---|-----------|------------|-----------|------------|-----------|------------|-----|-----|-----|-----|-----|-----|-----|-----|
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF3[1:0] | MODE3[1:0] | CNF2[1:0] | MODE2[1:0] | CNF1[1:0] | MODE1[1:0] | CNF0[1:0] | MODE0[1:0] | | | | | | | | |
| r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w | r w |
| 位31:30 | CNFy[1:0]: 端口x配置位(y = 0...7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | | | | | | | | | | | | | |
| 27:26 | | | | | | | | | | | | | | | |
| 23:22 | | | | | | | | | | | | | | | |
| 19:18 | | | | | | | | | | | | | | | |
| 15:14 | | | | | | | | | | | | | | | |
| 11:10 | | | | | | | | | | | | | | | |
| 7:6 | | | | | | | | | | | | | | | |
| 3:2 | | | | | | | | | | | | | | | |
| 位29:28 | MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz | | | | | | | | | | | | | | |
| 25:24 | | | | | | | | | | | | | | | |
| 21:20 | | | | | | | | | | | | | | | |
| 17:16 | | | | | | | | | | | | | | | |
| 13:12 | | | | | | | | | | | | | | | |
| 9:8, 5:4 | | | | | | | | | | | | | | | |
| 1:0 | | | | | | | | | | | | | | | |

该寄存器的复位值为 0X4444 4444 (4 化成二进制为 0100)，从上图可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRL 控制着每个 IO 端口 (A~G) 的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式 (ADC 用)、0X3 表示推挽输出模式 (做输出口用，50M 速率)、0X8 表示上/下拉输入模式 (做输入口用)、0XB 表示复用输出 (使用 IO 口的第二功能)。

STM32 的 IO 口位配置表如下表：

| 配置模式 | | CNF1 | CNF0 | MODE1 | MODE0 | PxODR寄存器 |
|--------|----------------|------|------|----------|---------------|----------|
| 通用输出 | 推挽(Push-Pull) | 0 | 0 | 01 | 0或1 | 0或1 |
| | 开漏(Open-Drain) | | 1 | | | |
| 复用功能输出 | 推挽(Push-Pull) | 1 | 0 | 10 11 | 见表18 | 不使用 |
| | 开漏(Open-Drain) | | 1 | | | 不使用 |
| 输入 | 模拟输入 | 0 | 0 | 00 | 0或1 0 1 | 不使用 |
| | 浮空输入 | | 1 | | | 不使用 |
| | 下拉输入 | 1 | 0 | | | 0 |
| | 上拉输入 | | | | | 1 |

可以看到 CNFX 是上面 CRL 寄存器里的配置位，这里配置位的不同，就会产生不同的 GPIO 管脚模式。

STM32 输出模式配置如下表：

| MODE[1:0] | 意义 |
|-----------|---------------|
| 00 | 保留 |
| 01 | 最大输出速度为 10MHz |
| 10 | 最大输出速度为 2MHz |
| 11 | 最大输出速度为 50MHz |

这里 CRL 寄存器的 MODE 配置位的选项，不同的配置就会产生不同的速率。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口，大家可以自己看 STM32 手册，我们在这里 CRH 就不做详细介绍了。

给个实例，比如我们要设置 PORTB 的 12 位为上拉输入，13 位为推挽输出。代码如下：

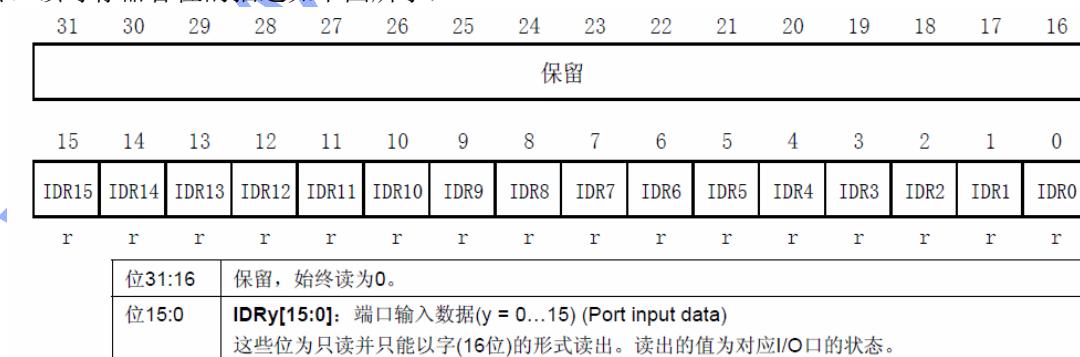
`GPIOB->CRH&=0xFF00FFFF;//清掉这 2 个位原来的设置，同时也不影响其他位的设置`

`GPIOB->CRH|=0X00380000; //PB12 输入， PB13 输出`

`GPIOB->ODR = 1<<12;/PB12 上拉(1 往右移 12 个位，从 PB0 开始，相当于把二进制 1 0000 0000 0000 赋值给 GPIOB_ODR，刚好对应了 PB12)`

通过这 3 句话的配置，我们就设置了 PB12 为上拉输入，PB13 为推挽输出。

IDR 是一个 GPIOx_IDR 的端口输入数据寄存器的简称 (ODR 是输入数据寄存器的简称)，要想知道某个 IO 口的状态，就要读这个寄存器，再从读出的寄存器值分析出某个管脚位的状态，就可以知道这个管脚的状态了；IDR 寄存器只用了低 16 位。该寄存器为只读寄存器，并且只能以 16 位的形式读出。该寄存器各位的描述如下图所示：



ODR 是一个端口输出数据寄存器，其作用就是控制端口的输出，对 ODR 对应寄存器位置 1 即对应的 GPIO 管脚就会输出高电平。该寄存器也只用了低 16 位，并且该寄存器可读可写，如果读的话，从该寄存器读出来的数据都是 0，所以读是没有意义的；只有写是有效的，该寄存器的各位描述如下图所示：

| | | | | | | | | | | | | | | | |
|--|-------|-------|-------|-------|-------|------|------|------|------|------|------|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ODR15 | ODR14 | ODR13 | ODR12 | ODR11 | ODR10 | ODR9 | ODR8 | ODR7 | ODR6 | ODR5 | ODR4 | ODR3 | ODR2 | ODR1 | ODR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位31:16 保留，始终读为0。 | | | | | | | | | | | | | | | |
| 位15:0 ODRy[15:0]: 端口输出数据(y = 0...15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注: 对GPIOx_BSRR(x = A...E), 可以分别地对各个ODR位进行独立的设置/清除。 | | | | | | | | | | | | | | | |

GPIOx_IDR 是端口的输入数据寄存器, GPIOx_ODR 是端口的输出寄存器, 我们配置引脚的输入输出模式是通过 GPIOx_CRL 和 GPIOx_CRH 两个寄存器来配置的, 但是每个端口的 16 个引脚它们有的可能是输出模式, 有的是输入模式, 甚至一会输出一会输入, 而 GPIOx_IDR 和 GPIOx_ODR 两个寄存器是以字模式 (读一次就是访问 2 个字节, 一个字等于 2 个字节) 访问而不能以 bit 模式 (bit 模式表示一次访问一个 bit 位, 一个字节等于 8 个 bit, 一个字等于 16 个 bit) 访问, GPIOx_IDR 只能读, 而 GPIOx_ODR 可以读写。

关于 GPIO 的输出模式下几种速度的区别: 2MHz、10MHz、50MHz; 这个又可以理解为输出驱动电路的不同响应速度 (芯片内部在 I/O 口的输出部分安排了多个响应速度不同的输出驱动电路, 用户可以根据自己的需要选择合适的驱动电路, 通过选择速度来选择不同的输出驱动电路模块, 达到最佳的噪声控制和降低功耗的目的)。

那为什么要几种速率呢? 如果选择了不合适的速率会有什么影响呢? 芯片管脚的速度就好比是信号收发的频率, 速度快就表示信号收发的频率高; 如果信号频率为 10MHz, 而你配置了 2MHz 的带宽, 那么就会丢失很多数据, 很多数据点截取不到, 这个 10MHz 的方波很可能就变成了正弦波。这个就好比是公路的设计时速, 汽车速度低于设计时速时, 可以平稳的运行, 如果超过设计时速就会颠簸, 甚至翻车。

所以芯片管脚的输入输出速率可以理解为, 输出驱动电路的带宽, 即一个驱动电路可以不失真地通过信号的最大频率; 如果一个信号的频率超过了驱动电路的响应速度, 就有可能信号失真。带宽速度高的驱动器耗电大、噪声也大, 带宽低的驱动器耗电小、噪声也小; 比如: 高频的驱动电路, 噪声也高, 当不需要高的输出频率时, 请选用低频驱动电路, 这样非常有利于提高系统的 EMI 性能。当然如果要输出较高频率的信号, 但却选用了较低频率的驱动模块, 很可能会得到失真的输出信号。关键是 GPIO 的引脚速度跟应用匹配, 比如:

- 1) USART 串口, 若最大波特率只需 115.2k, 1M 等于 1000k, 那用 2MHz 的速度就够了, 既省电也噪声小, STM32 最低的速率是 2MHz, 已经可以满足要求了。
- 2) I2C 接口, 若使用 400k 波特率, 也可以选用 2M 的速度够了; 当然若想把余量留大些, 可以选用 10M 的 GPIO 引脚速度。
- 3) SPI 接口, 若使用 18M 或 9M 波特率, 需要选用 50M 的 GPIO 的引脚速度, 我们这里一定要使得 GPIO 的速度大于外部应用的速度, 这好像就是在高速上跑公交车, 而不是在田埂上开赛车。高速上跑公交车, 公交车可以开最快的速度, 而不怕翻车。

这里提到了波特率, 那么什么是波特率呢? 波特率是指数据信号对载波的调制速率, 它用单位时间内载波调制状态改变的次数来表示。波特率一次传输一个数据对象, 而这个数据对象可能是几个比特 (bit), 所以波特率跟比特率是有区别的。

比特率在数字信道中, 比特率是数字信号的传输速率, 它用单位时间内传输的二进制代码的有效位(bit)数来表示, 其单位为每秒比特数 bit/s(bps)、每秒千比特数(Kbps)或每秒兆比特数(Mbps)来表示(此处 K 和 M 分别为 1000 和 1000000)。

波特率与比特率的关系为: 比特率 = 波特率 X 单个调制状态对应的二进制位数。

5.1.6 在STM32中如何配置片内外设使用的IO端口

首先, 一个外设在使用前, 必须先配置和激活启动该外设的时钟, 比如 GPIO 端口 B, 那么就要激

活 GPIOB 的时钟，比如 GPIOA，那么使用 PA2 管脚前，必须要前激活 GPIOA 端口的时钟，只有启动时钟后，这个外设才变得激活可用。

时钟被启动之后，再根据这个具体功能，对这个外设进行相应的设置和配置，这样的好处是可以降低 STM32 芯片的内部功耗，因为需要用到的外设才被激活，激活的外设会消耗芯片的比较多的功耗；不需要使用的外设无需初始化，这样设计可以降低芯片的功耗；好处尤其体现在类似手持设备，功耗比较小，使得电池更加耐用。

那么如何配置管脚采用哪种模式呢？这里粗略总结对应到外设的输入输出功能基本有三种情况：

1) 管脚输出：需要根据外围电路的配置选择对应的管脚为复用功能的推挽输出或复用功能的开漏输出。

2) 管脚输入：则根据外围电路的配置可以选择浮空输入、带上拉输入或带下拉输入。

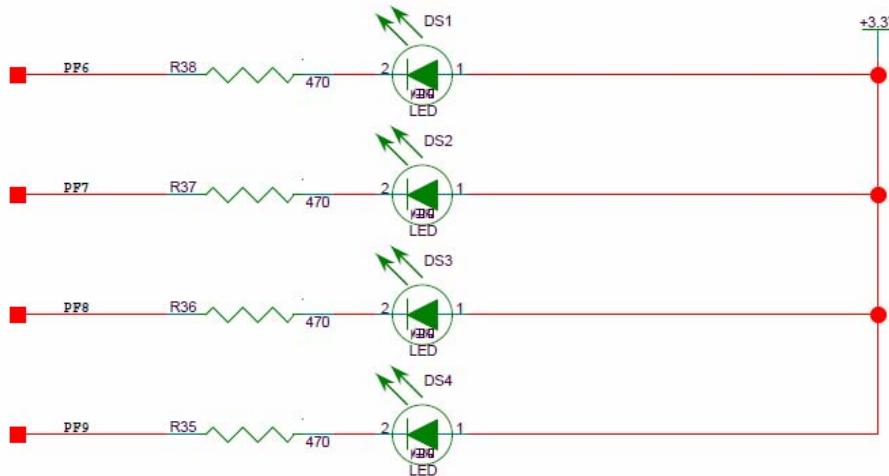
3) ADC 对应的管脚：配置管脚为模拟输入。

值得注意的是，这里如果把端口配置成复用输出功能，则该引脚与它当前连的信号电路断开，和复用功能信号电路连接，所以将管脚配置成复用输出功能后，如果只激活了该引脚的 GPIO 端口的时钟，而忘记把复用功能的时钟激活，那么它的输出将不确定，这样会产生异常的现象。

5.1.7 例程01 单个LED点灯闪烁程序

1. 示例简介

LED 灯的正极接的是 3.3V 电源，所以我们编程让 LED 负极拉低即 GPIO 引脚端口 F 的管脚 6 拉低，即 PF6 拉低，那么 LED 灯就会变亮，相关电路图如下图所示：

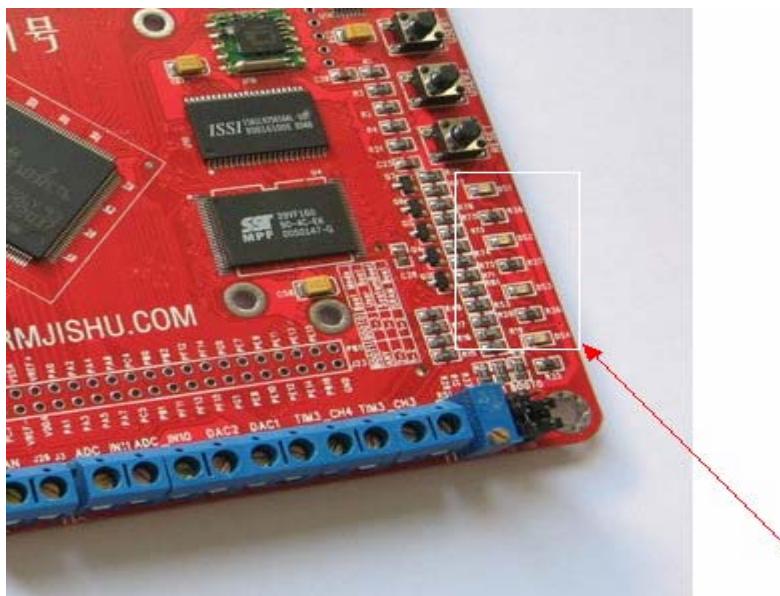


注意到，这里采用GPIO管脚的低电平点灯，原因是：处理器的GPIO管脚只要输出低电平即可点灯，处理器功耗低；如果LED的一端固定接到GND地上，那么对于处理器的GPIO管脚点灯时，就必须输出高电平，这样增加处理器的功耗。同时，大家要注意，在设计LED灯限流时，串接的电阻放置的位置，有人会问，也可以放在LED的右边。一般，我们不会放在右边，主要是LED灯，人手可能会去触摸到，这样可能会将人体上的静电引导板件上，如果将电阻放在左边，静电经过电阻后，会消弱很多，以致不会一下子因为静电就将处理器烧毁。

一般的LED灯需要15~20毫安的电流才可以点亮，我们这里是3.3V的电压，经过1K欧姆的限流电阻，用3.3V除以1000欧姆，理论值是33毫安的电流，足以点亮LED灯了。当然如果电流如果过大，就会使得经过的电流变大，LED灯可能点不亮或者比较暗；如果电阻过小，就会导致电流过大，可能烧掉LED灯，所以这个限流电阻选取也是有个范围的。

2. 调试说明

下载代码，并且按下【复位】键，在神舟 III 号板上找到 DS1，可以看到该 DS1 灯一亮一灭。



3. 关键代码:

相关代码如下图程序清单:

```
***** www.armjishu.com *****/
int main(void) //main 是程序入口
{
    /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF;

    /*-- GPIO Mode Configuration 速度，输入或输出 -----*/
    /*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOF->CRL &= 0xFFFFFFFF;
    GPIOF->CRL |= 0x03000000;

    while (1)
    {
        GPIOF->BRR = GPIO_Pin_6;
        Delay(0x2FFFFF);
        GPIOF->BSRR = GPIO_Pin_6;
        Delay(0x2FFFFF);
    }
}
```

看原理图可以知道，因为 LED 的正极接的是 3.3V 电源端，所以当 PF6 管脚拉低成低电平的时候，LED 灯就会亮起来。

这里要注意的是在配置 STM32 外设的时候，任何时候都要先使能该外设的时钟!!! APB2ENR 寄存器是 APB2 总线上的外设时钟使能寄存器，其各位的描述如下：

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|------------|--------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|----|------------|
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADC3 EN | USART1 EN | TIM8 EN | SPI1 EN | TIM1 EN | ADC2 EN | ADC1 EN | IOPG EN | IOPF EN | IOPE EN | IOPD EN | IOPC EN | IOPB EN | IOPA EN | 保留 | AFIO EN |

图：寄存器 APB2ENR 各位描述

我们要使能的是 PORTF 的时钟使能位，大家可以从上表看得到在 bit7 这个位，只需要将这个位置 1 就可以使能 PORTB 的时钟了，大家可以跟进去看看下面的这句代码，就能看到具体是设置的是这个 RCC 寄存器了

```
#define RCC_APB2Periph_GPIOF ((uint32_t)0x00000080);
```

这句代码 80 化成二进制是 1000 0000 刚好是 bit7 的这个位，然后对这个 APB2ENR 的 GPIOF 端口时钟置位：

```
RCC->APB2ENR |= RCC_APB2Periph_GPIOF; /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */
```

这句代码相当于：RCC->APB2ENR |= 0x00000080；或上以后使得 APB2ENR 寄存器的第 bit7 这个位（从 0 开始，32 位寄存器，bit3 实际是在第 4 位）置 1，刚好是对应的 GPIO 端口 F 的时钟位，那么 GPIOF 的时钟就被使能了。

对 GPIOF->BRR = GPIO_Pin_6; 这句代码来说，我们可以看到 GPIO_Pin_6 的定义如下：

```
#define GPIO_Pin_6 ((uint16_t)0x0040)
```

对 GPIOB_BRR 这个寄存器进行赋值，我们看下图：

8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)

地址偏移：0x14

复位值：0x0000 0000

| | | | | | | | | | | | | | | | |
|------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |

W

W

W

W

W

W

W

W

W

W

W

W

W

W

W

W

W

115/754

参照2009年12月 RM0008 Reference Manual 英文第10版

本译文仅供参考，如有翻译错误，请以英文原稿为准。请读者随时注意在ST网站下载更新版本



通用和复用功能I/O

STM32F10xxx参考手册

| | |
|--------|---|
| 位31:16 | 保留。 |
| 位15:0 | BRy: 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 |

对 GPIOF_BRR 的 BR6 位进行置 1 操作，看寄存器说明可以知道，对 BR6 置 1 后，GPIOF_ODR 寄存器的 ODR6 位就为 0 了，使得 PF6 管脚输出低电平，灯被点亮（前面分析过，LED 灯是低电平点亮，具体原因请见前面的原理图分析）

同样，我们分析一下代码：GPIOF->BSRR = GPIO_Pin_6; 下面看下 GPIOF_BSRR 寄存器：

8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

地址偏移: 0x10

复位值: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|--------|------|--|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 位31:16 | | BRy: 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。 | | | | | | | | | | | | | |
| 位15:0 | | BSy: 设置端口x的位y (y = 0...15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 设置对应的ODRy位为1 | | | | | | | | | | | | | |

因为 GPIO_Pin_6 是等于 0x0040, 所以 GPIOF_BSRR 寄存器的 BS6 被置 1, 可以看到说明, 该位置 1 后, 使得 GPIOF_ODR 寄存器的 ODR6 位就为 1 了, PF6 管脚输出高电平, 灯灭 (前面分析过, LED 灯是低电平点亮, 高电平熄灭, 具体原因请见前面的原理图分析)。

其中 Delay(0xFFFF) 是延时函数, 所以在这个 while 循环里, LED 灯亮一段时间后, 就熄灭一段时间, 周而复始, 交替进行。整段代码就都分析完了, 至于代码中许多 define 的定义, 例如:

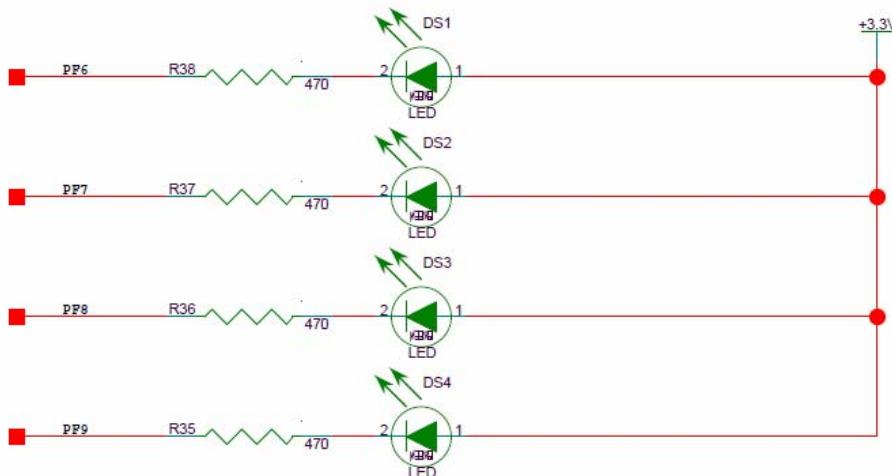
```
***** GPIOB 管脚的内存对应地址 *****/
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB2PERIPH_BASE ((PERIPH_BASE + 0x10000)
#define GPIOF_BASE ((APB2PERIPH_BASE + 0x1c00)
#define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
可以参看前面的基础入门篇, 如何将阅读寄存器的入门章节, 这些都是于 STM32 芯片参考手册里的规定对应的。
```

更多关于这个寄存器的详细说明大家可以看《STM32F10xxx 参考手册》的第 7 章。

5.1.8 例程02 LED双灯闪烁实验

1. 示例简介

LED 灯的正极接的是 3.3V 电源, 所以我们编程让 LED 负极拉低即 GPIO 引脚端口 F 的 Pin6 和 PIN7 拉低, 即 PF6 和 PF7; 那么 LED 灯就会变亮; 同样将 PF6 和 PF7 管脚拉高时, LED 就会灭掉; 亮和灭各经过一段延时, 就会变成闪烁的样子, 这里我们编程增加了延时程序, 相关电路图如下图所示:



2. 调试说明

下载代码，并且按下【复位】键，在神舟 III 号板上找到 DS1 和 DS2，可以这 2 个 LED 灯不停的闪烁。

3. 关键代码

```
***** www.armjishu.com *****
```

```
int main(void) //main 是程序入口
{
    /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF;

    /*-- GPIO Mode Configuration 速度，输入或输出 -----*/
    /*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOF->CRL &= 0x00FFFFFF;
    GPIOF->CRL |= 0x33000000;

    while (1)
    {
        GPIOF->BRR = GPIO_Pin_6;
        GPIOF->BRR = GPIO_Pin_7;
        Delay(0x2FFFFF);
        GPIOF->BSRR = GPIO_Pin_6;
        GPIOF->BSRR = GPIO_Pin_7;
        Delay(0x2FFFFF);
    }
}
```

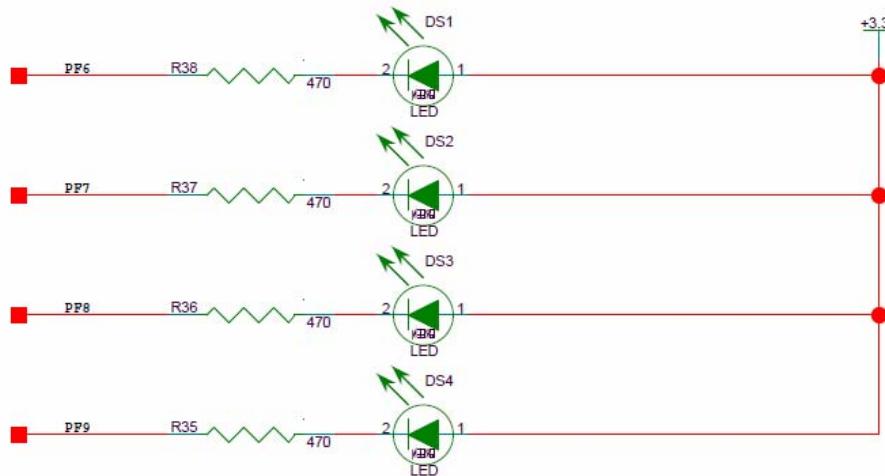
这里和上面程序不同之处是上面使用的是 PF 端口，这里使用的 PF 端口，并且同时控制 PF 端口的 2 个管脚来控制 LED 的亮灭。

5.1.9 例程03 LED三个灯同时亮同时灭

1.示例简介

LED 灯的正极接的是 3.3V 电源，所以我们编程让 PF6、PF7、PF8 三个管脚拉低；那么 LED

灯就会变亮；同样让 PF6、PF7、PF8 三个管脚拉高时，LED 就会灭掉；亮和灭各经过一段延时，就会变成闪烁的样子，这里我们编程增加了延时程序，相关电路图如下图所示：



2. 调试说明

下载代码，并且按下【复位】键，在神舟 III 号板上找到 DS1、DS2 和 DS3，可以这 3 个 LED 灯不停的闪烁。

3. 关键代码

```
***** www.armjishu.com *****/
int main(void) //main 是程序入口
{
    /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF;

    /*-- GPIO Mode Configuration 速度，输入或输出 -----*/
    /*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOF->CRL &= 0x00FFFFFF;
    GPIOF->CRL |= 0x33000000;

    GPIOF->CRH &= 0xFFFFFFFF;
    GPIOF->CRH |= 0x00000003;

    while (1)
    {
        GPIOF->BRR = GPIO_Pin_6; /*熄灭 LED 灯*/
        GPIOF->BRR = GPIO_Pin_7;
        GPIOF->BRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
        GPIOF->BSRR = GPIO_Pin_6; /*点亮 LED 灯*/
        GPIOF->BSRR = GPIO_Pin_7;
        GPIOF->BSRR = GPIO_Pin_8;
        Delay(0x2FFFFF);
    }
}
```

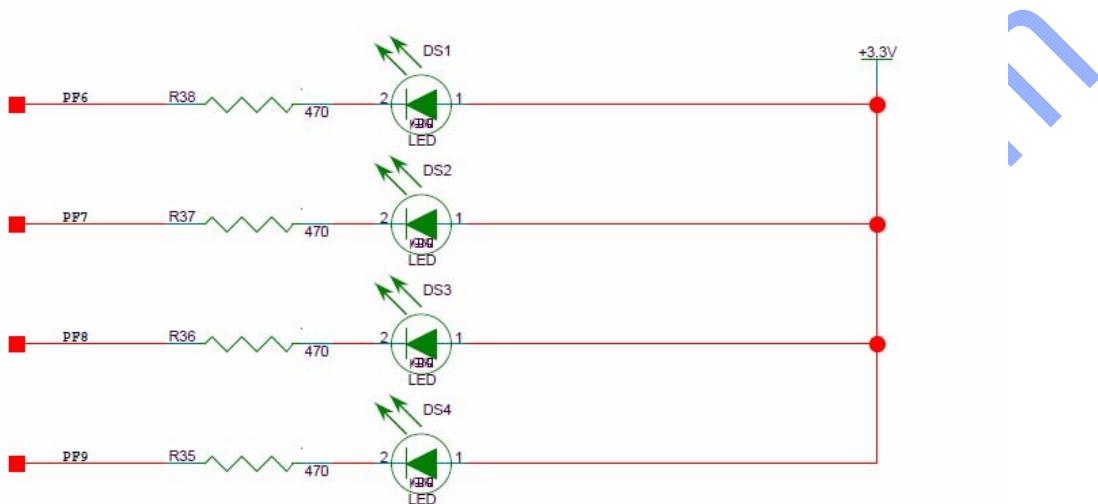
} 这里和上面程序不同之处是这里使用 PF6、PF7、PF8 三个管脚来同时控制 LED 灯的一亮一灭。

5.1.10 例程04 LED流水灯程序

1. 示例简介

在神舟III号STM32开发板中，一共有四个LED指示灯，其中一个是电源指示灯，上电就点灯的；另外三个LED是由三个GPIO管脚控制，当GPIO管脚输出低电平时，对应的LED灯亮；当GPIO管脚输出高电平时，对应的LED灯灭。

下图为 LED 原理图，其中 GPIO 管脚上串的电阻，主要起限流作用。防止电流过大损坏 LED 和 GPIO 管脚：



2. 调试说明

下载代码，并且按下【复位】键，在神舟III号板上找到DS1、DS2、DS3、DS4三个灯，可以看到这三个灯轮流闪烁，流水灯。

3. 关键代码

```
***** www.armjishu.com *****/
int main(void) //main 是程序入口
{
    /* 使能 APB2 总线的时钟，对 GPIO 的端口 F 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF;

    /*-- GPIO Mode Configuration 速度，输入或输出 -----*/
    /*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/
    /*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOF->CRL &= 0x00FFFFFF;
    GPIOF->CRL |= 0x33000000;

    GPIOF->CRH &= 0xFFFFFFF0;
    GPIOF->CRH |= 0x00000033;

    while (1)
    {
        GPIOF->BRR = GPIO_Pin_6; /*熄灭 LED 灯*/
        Delay(0x2FFFF);
        GPIOF->BRR = GPIO_Pin_7; /*熄灭 LED 灯*/
        Delay(0x2FFFF);
    }
}
```

```
GPIOF->BRR = GPIO_Pin_8; /*熄灭 LED 灯*/  
Delay(0xFFFF);  
GPIOF->BRR = GPIO_Pin_9; /*熄灭 LED 灯*/  
Delay(0xFFFF);  
  
GPIOF->BSRR = GPIO_Pin_6; /*点亮 LED 灯*/  
Delay(0xFFFF);  
GPIOF->BSRR = GPIO_Pin_7; /*点亮 LED 灯*/  
Delay(0xFFFF);  
GPIOF->BSRR = GPIO_Pin_8; /*点亮 LED 灯*/  
Delay(0xFFFF);  
GPIOF->BSRR = GPIO_Pin_9; /*点亮 LED 灯*/  
Delay(0xFFFF);  
}  
}
```

程序主要设计思路就是先将所有 LED 灯逐个经过延时后熄灭，然后再逐个被点亮，如此循环，形成 LED 流水灯。

5.2 时钟

5.2.1 什么是时钟

从 CPU 的时钟说起。

计算机是一个十分复杂的电子设备。它由各种集成电路和电子器件组成，每一块集成电路中都集成了数以万计的晶体管和其他电子元件。这样一个十分庞大的系统，要使它能够正常地工作，就必须有一个指挥，对各部分的工作进行协调。各个元件的动作就是在这个指挥下按不同的先后顺序完成自己的操作的，这个先后顺序我们称为时序。时序是计算机中一个非常重要的概念，如果时序出现错误，就会使系统发生故障，甚至造成死机。那么是谁来产生和控制这个操作时序呢？这就是“时钟”。“时钟”可以认为是计算机的“心脏”，如同人一样，只有心脏在跳动，生命才能够继续。不要把计算机的“时钟”等同于普通的时钟，它实际上是由晶体振荡器产生的连续脉冲波，这些脉冲波的幅度和频率是不变的，这种时钟信号我们称为外部时钟。它们被送入 CPU 中，再形成 CPU 时钟。不同的 CPU，其外部时钟和 CPU 时钟的关系是不同的，下表列出了几种不同 CPU 外部时钟和 CPU 时钟的关系。

CPU 时钟周期通常为节拍脉冲或 T 周期，它是处理操作的最基本的单位。

在微程序控制器中，时序信号比较简单，一般采用节拍电位——节拍脉冲二级体制。就是说它只要一个节拍电位，在节拍电位又包含若干个节拍脉冲（时钟周期）。节拍电位表示一个 C P U 周期的时间，而节拍脉冲把一个 C P U 周期划分为几个叫较小的时间间隔。根据需要这些时间间隔可以相等，也可以不等。

指令周期是取出并执行一条指令的时间。

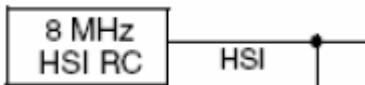
指令周期常常有若干个 C P U 周期，C P U 周期也称为机器周期，由于 C P U 访问一次内存所花费的时间较长，因此通常用内存中读取一个指令字的最短时间来规定 C P U 周期。这就是说，这就是说一条指令取出阶段（通常为取指）需要一个 C P U 周期时间。而一个 C P U 周期时间又包含若干个时钟周期（通常为节拍脉冲或 T 周期，它是处理操作的最基本的单位）。这些时钟周期的总和则规定了一个 C P U 周期的时间宽度。

5.2.2 STM32的时钟

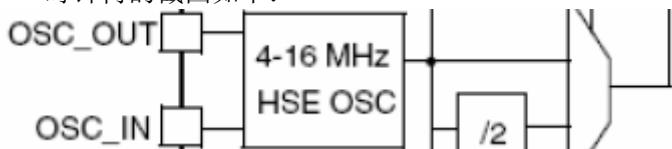
系统时钟的选择是在启动时进行，复位时内部 8MHZ 的 RC 振荡器被选为默认的 CPU 时钟，随后可以选择外部的、具失效监控的 4-16MHZ 时钟；当检测到外部时钟失效时，它将被隔离，系统将自动地切换到内部的 RC 振荡器。

在 STM32 中，有五个时钟源，为 HSI、HSE、LSI、LSE、PLL，它们都是时钟所提供的来源：

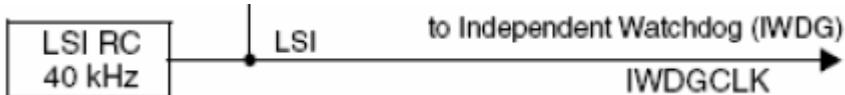
1. HSI 是高速内部时钟，RC 振荡器，频率默认为 8MHz，可以从 STM32 时钟树中看到



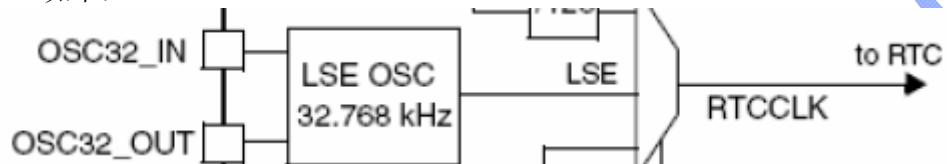
2. HSE 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz~16MHz，时钟树的截图如下：



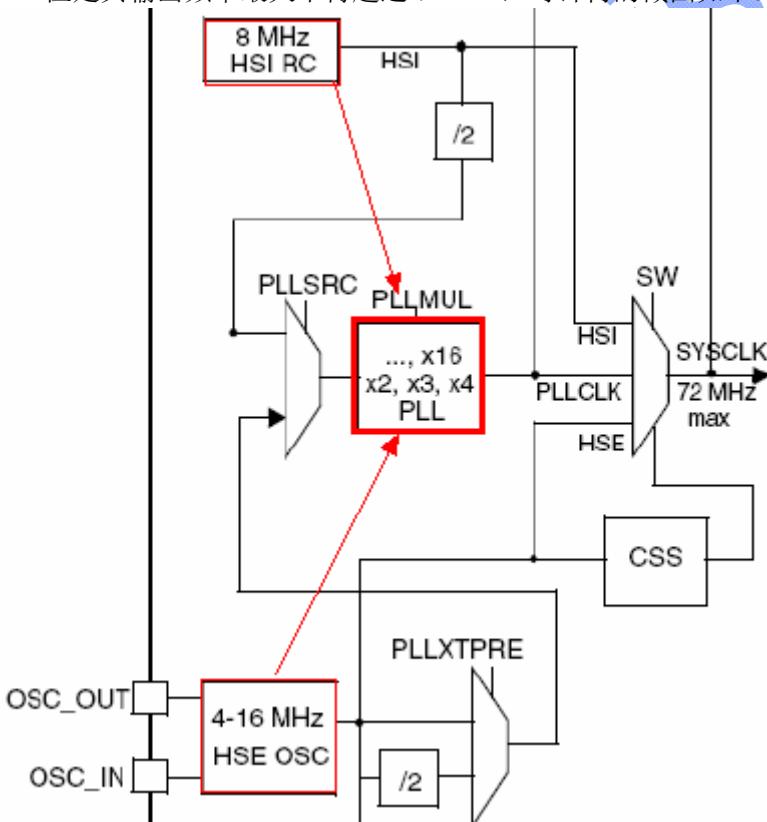
3. LSI 是低速内部时钟，RC 振荡器，频率为 40kHz，可以用于驱动独立看门狗和通过程序选择驱动 RTC (RTC 用于从停机/待机模式下自动唤醒系统)，时钟树的截图如下：



4. LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体，也可以被用来驱动 RTC，时钟树的截图如下：



5. PLL 为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HSE 或者 HSE/2。倍频可选择为 2~16 倍，但是其输出频率最大不得超过 72MHz，时钟树的截图如下：



5.2.3 STM32的时钟深入分析

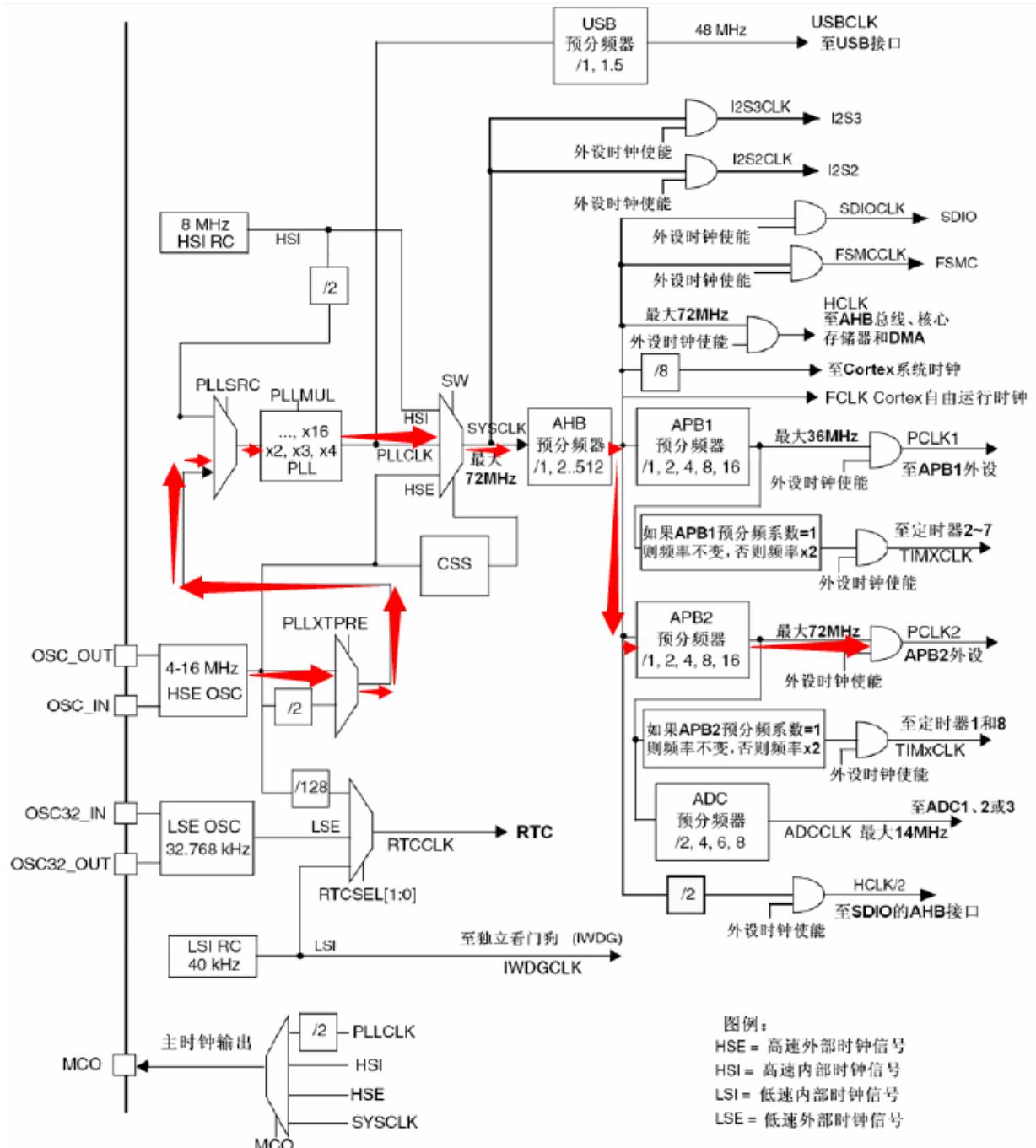
众所周知，微控制器（处理器）的运行必须要依赖周期性的时钟脉冲来驱动——往往由一个外部嵌入式专业技术论坛（www.armjishu.com）出品 第 210 页，共 900 页

晶体振荡器提供时钟输入为始，最终转换为多个外部设备的周期性运作为末，这种时钟“能量”扩散流动的路径，犹如大树的养分通过主干流向各个分支，因此常称之为“时钟树”。在一些传统的低端 8 位单片机诸如 51，AVR，PIC 等单片机，其也具备自身的一个时钟树系统，但其中的绝大部分是不受用户控制的，亦即在单片机上电后，时钟树就固定在某种不可更改的状态（假设单片机处于正常工作的状态）。比如 51 单片机使用典型的 12MHz 晶振作为时钟源，则外设如 IO 口、定时器、串口等设备的驱动时钟速率便已经是固定的，用户无法将此时钟速率更改，除非更换晶振。

STM32 芯片为了实现低功耗，设计了一个功能完善但却非常复杂的时钟系统。我们使用外设的时候是需要开启外部时钟的。STM32 的芯片可以分为小容量产品、中容量产品、大容量产品，互联型产品。前面 3 种类型按芯片的 Flash 大小来分。而 **STM32** 互联系列让设计人员可以在同时需要以太网、**USB**、**CAN** 和音频级**I2S** 接口的产品设计中发挥工业标准的 32 位微处理器的优异性能。按芯片 Flash 大小来分类的 3 种芯片的系统时钟是大同小异的。而互联型产品芯片的系统时钟和它们比较差异较大的。我们神舟 III 号开发板使用的主芯片 STM32F103ZET 是大容量型的。我们下面分析它的系统时钟。

- 小容量型、中容量型、大容量型产品的时钟树&时钟源

首先，从整体上了解 STM32 的时钟系统。



从上图可知，STM32F103ZET有以下4个时钟源：

高速外部时钟（HSE）：以外部晶振作时钟源，晶振频率可取范围为4~16MHz，我们采用8MHz的晶振。

高速内部时钟（HSI）：由内部RC振荡器产生，频率为8MHz，但不稳定。

低速外部时钟(LSE): 以外部晶振作时钟源，主要提供给实时时钟模块，所以一般采用32.768KHz。

低速内部时钟（LSI）：由内部RC振荡器产生，也主要提供给实时时钟模块，频率大约为40KHz。

STM32的时钟走向，从图的左边开始，从时钟源一步步分配到外设时钟。

从时钟频率来说，又分为**高速时钟**和**低速时钟**，高速时钟是提供给芯片主体的主时钟，而低速时钟只是提供给芯片中的RTC（实时时钟）及独立看门狗使用。

从芯片角度来说，时钟源分为**内部时钟**与**外部时钟源**，内部时钟是在芯片内部RC振荡器产生的，起振较快，在芯片刚上电的时候，默认使用内部高速时钟。而外部时钟信号是由外部的晶振输入的，在精度和稳定性上都有很大优势，上电之后我们再通过软件配置，转而采用外部时钟信号。

图例：
 HSE = 高速外部时钟信号
 HSI = 高速内部时钟信号
 LSI = 低速内部时钟信号
 LSE = 低速外部时钟信号

● STM32使用的高速外部时钟（HSE）分析

芯片刚上电的时候，默认使用内部高速时钟。上电之后再通过软件配置，转而采用高速外部时钟信号。系统时钟，在ST官方提供的系统启动文件(startup_stm32f10x_xx.s)中，调用SystemInit()给我们配置好了系统时钟。以神舟III号开发板为例，我们在外部提供的晶振的频率为8MHz。最终配置出来的主频是72MHz。

我们分析一下，如何得到外设 GPIOF 的时钟，下面是一条时钟的“脉络”，其中的标号和时钟树中的标号一一对应。“脉络”也通过箭头标出。

对此条时钟路径做如下解析：

(HSE)，首先我们的外部时钟是 8MHz。经过 PLLXTPRE，直接选择 HSE 为输入，得 8 (MHz)。经过 PLLSRC 选择，还是 8 (MHz)。8MHz 经过 PLLMULL 的 9 倍频， $8 \times 9 = 72$ (MHz)。经过 SW 选择 PLLCLK 做为 SYSCLK (系统时钟) 的输入时钟，那么 SYSCLK 等于输送过来的 72MHz。外设 GPIO 的时钟来源于 APB2 总线，那么经过 AHB 预分频器，不分频。在经过 APB2 预分频器，不分频。最后 APB2 外设 GPIO 就可以得到 72MHz 的时钟源了。

ST 提供的启动文件(startup_stm32f10x_xx.s)中，调用 SystemInit()。在函数 SystemInit() 中，配置时钟的是函数 SetSysClock()。函数 SetSysClock() 又调用函数 SetSysClockTo72()。设置 72MHz 的系统时钟。函数 SetSysClockTo72() 中的时钟设置路径与我们给出的时钟设置“脉络”是一致的。

函数 SetSysClock() 内容如下：

```
static void SetSysClock(void)
{
#ifndef SYSCLK_FREQ_HSE
    SetSysClockToHSE();
#elif defined SYSCLK_FREQ_24MHz
    SetSysClockTo24();
#elif defined SYSCLK_FREQ_36MHz
    SetSysClockTo36();
#elif defined SYSCLK_FREQ_48MHz
    SetSysClockTo48();
#elif defined SYSCLK_FREQ_56MHz
    SetSysClockTo56();
#elif defined SYSCLK_FREQ_72MHz
    SetSysClockTo72();
#endif

/* If none of the define above is
   source (default after reset) */
}
```

可以发现，其实通过定义不同的宏系统时钟，可以配置成不同频率。我们看一下，代码中定义了那个宏。

```
/* #define SYSCLK_FREQ_HSE      HSE_VALUE */
/* #define SYSCLK_FREQ_24MHz    24000000 */
/* #define SYSCLK_FREQ_36MHz    36000000 */
/* #define SYSCLK_FREQ_48MHz    48000000 */
/* #define SYSCLK_FREQ_56MHz    56000000 */
#define SYSCLK_FREQ_72MHz    72000000
```

对宏进行追溯，可以发现，只定义了 SYSCLK_FREQ_72MHz 的宏。其它的都被注销了。至于函数 SetSysClockTo72() 中，系统时钟为 72MHz 的方程，参考我们提供的时钟“脉络”。

系统时钟的配置，不是说想怎么配就怎么配。比如 APB2 总线的时钟，最高是 72MHz，而 APB1 总线最高 36MHz。具体参考 ST 公司提供的参考手册关于系统时钟的说明。

● 常用的HCLK、FCLK、PCLK1、PCLK2时钟说明

从时钟树的分析，看到经过一系列的倍频、分频后得到了几个与我们开发密切相关的时钟。

SYSCLK：系统时钟，STM32大部分器件的时钟来源。主要由AHB预分频器分配到各个部件。

HCLK:由AHB预分频器直接输出得到，它是高速总线AHB的时钟信号，提供给存储器，DMA及cortex内核，是cortex内核运行的时钟，cpu主频就是这个信号，它的大小与STM32运算速度，数据存取速度密切相关。

FCLK: 同样由AHB预分频器输出得到，是内核的“自由运行时钟”。 “自由”表现在它不来自时钟 HCLK，因此在HCLK时钟停止时 FCLK 也继续运行。它的存在，可以保证在处理器休眠时，也能够采样和到中断和跟踪休眠事件，它与HCLK互相同步。

PCLK1: 外设时钟，由APB1预分频器输出得到，最大频率为36MHz，提供给挂载在APB1总线上的外设。

PCLK2: 外设时钟，由APB2预分频器输出得到，最大频率可为72MHz，提供给挂载在APB2总线上的外设。

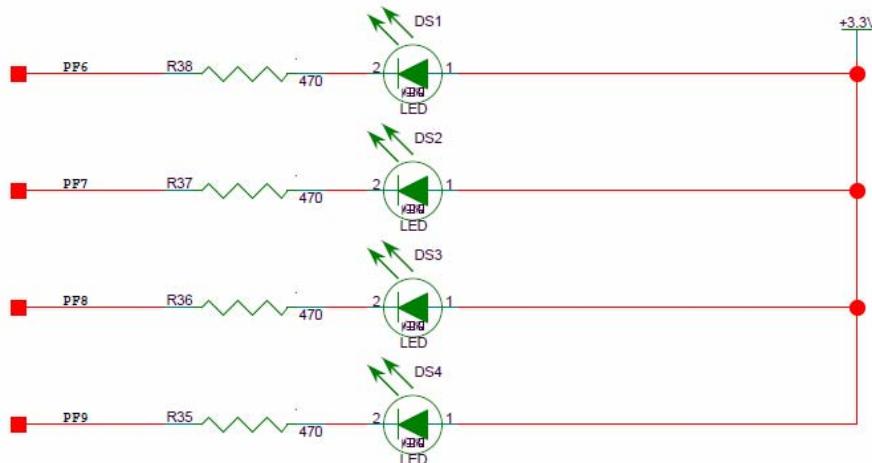
为什么STM32的时钟系统如此复杂，有倍频、分频及一系列的外设时钟的开关。需要倍频是考虑到电磁兼容性，如外部直接提供一个72MHz的晶振，太高的振荡频率可能会给制作电路板带来一定的难度。分频是因为STM32既有高速外设又有低速外设，各种外设的工作频率不尽相同，如同pc机上的南北桥，把高速的和低速的设备分开来管理。最后，每个外设都配备了外设时钟的开关，当我们不使用某个外设时，可以把这个外设时钟关闭，从而降低STM32的整体功耗。

上面是原理的剖析，如果再不明白的，可以接下来看例程代码，理论联系实践是最好的老师。

5.2.4 例程01 STM32芯片32MHZ频率下跑点灯程序

1. 示例简介

让点灯程序在时钟主频 32MHz 下面运行，LED 灯的正极接的是 3.3V 电源，所以我们编程让 LED 负极拉低即 GPIO 引脚端口 F 的管脚 6 拉低，即 PF6 拉低，那么 LED 灯就会变亮，相关电路图如下图所示：



2. 调试说明：

下载代码，并且按下【复位】键，在神舟 III 号板上找到 DS1，可以看到该 DS1 灯一亮一灭。

3. 关键代码：

```
***** www.armjishu.com *****/
int main(void) //main 是程序入口
{
    **** 程序总共 2 部分之第 1 部分 时钟频率的配置 {开始 *****
    ** 以下是关于 RCC 时钟 详细请见《STM32F10XXX 参考手册》6.3 节 RCC 寄存器描述**
    unsigned char sws = 0;
    RCC->CR |= 0X00010000; //使能外部高速时钟 HSEON
```

```
//将 RCC_CR 寄存器的值右移 17 位，等待 HSERDY 就绪，即外部时钟就绪
while(!(RCC->CR>>17));

/* 因为手册有要求 APB1 时钟频率不超过 36MHZ，而在 STM32 中最大为 72MHZ */
/* 为了保证最大速度，我们这里设置成 2 分频 */
/* 设置寄存器 CFGR 里的 8-10 位的值为 100 */
    RCC->CFGGR = 0x00000400;

/* 寄存器 CFGR 的 18-21 四个 bit 位配置成以下值，则 PLL 就会设置成对应的值：
   0000: PLL 2 倍频输出  1000: PLL 10 倍频输出
   0001: PLL 3 倍频输出  1001: PLL 11 倍频输出
   0010: PLL 4 倍频输出  1010: PLL 12 倍频输出
   0011: PLL 5 倍频输出  1011: PLL 13 倍频输出
   0100: PLL 6 倍频输出  1100: PLL 14 倍频输出
   0101: PLL 7 倍频输出  1101: PLL 15 倍频输出
   0110: PLL 8 倍频输出  1110: PLL 16 倍频输出
   0111: PLL 9 倍频输出  1111: PLL 16 倍频输出

   我们在这里，因为 STM32 神舟 III 号上的晶振是 8MHZ 的，配置成 9 倍输出就能达到 STM32
   最大 72MHZ 工作频率*/
    //本例程希望设置成 32MHZ 的工作频率，我们在这里尝试一下
    RCC->CFGGR |= 2<<18;
//2 右移动 18 位，即 0010 使得 PLL 获得 4 倍频输出，外部晶振是 8MHZ 乘以 4 就是 32MHZ
    RCC->CFGGR |= 1<<16; //PLLSRC 设置成 1，使得 HSE 时钟作为 PLL 输入时钟
    RCC->CR |= 1<<24; //将 PLL 使能
    while(!(RCC->CR>>25)); //监控寄存器 CR 的 PLLRDY 位，等待 PLL 时钟就绪
    RCC->CFGGR |= 1<<1; //将时钟切换寄存器配置成用 PLL 输出作为系统时钟
    while(sws != 0x2) //等待 CFGGR 寄存器的 2, 3 位为 10，系统正式切换到了 PLL 输出作为时
        钟
    {
        // 将 CFGGR 寄存器右移 2 位，将 2, 3 位 SWS 状态移出来，详情请见《STM32F10XXX 参考手册》54
        // 页
        sws = RCC->CFGGR>>2;
        //这里的 0x3 为二进制的 11，这个 whlie 循环设计的一个算法，为了判断 sws 是不是为 10
        sws &= 0x3;
    }
/**程序总共 2 部分之第 1 部分 时钟频率的配置 结束} **/}

/** 程序总共 2 部分之第 2 部分 点灯的配置 {开始**/
/* 使能 APB2 总线的时钟，对 GPIO 的端口 B 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOF;

/*-- GPIO Mode Configuration 速度，输入或输出 -----*/
/*-- GPIO CRL Configuration 设置 IO 端口低 8 位的模式（输入还是输出）---*/
/*-- GPIO CRH Configuration 设置 IO 端口高 8 位的模式（输入还是输出）---*/
    GPIOF->CRL &= 0xF0FFFFFF;
    GPIOF->CRL |= 0x03000000;

    while (1)
    {
        GPIOF->BRR = GPIO_Pin_6;
        Delay(0xFFFFFFF);
```

```

GPIOF->BSRR = GPIO_Pin_6;
Delay(0xFFFFFFF);
}
/**************** 程序总共 2 部分之第 2 部分 点灯的配置 结束 */
*/
}

```

这个例程主要是体现在如何设置时钟，点灯的代码和原理图都在通用输入/输出的 GPIO 章节详细说明了。

STM32 的时钟源有几种，有内部的 RC，也有外部的晶振，该选择哪种，代码里通过语句: RCC->CR |= 0X00010000；使能外部的晶振，那么 CR 这个寄存器全称叫做 STM32 的时钟控制寄存器，在《STM32F10xxx 参考手册》的 60 页可以看到 RCC_CR 寄存器：

6.3.1 时钟控制寄存器(RCC_CR)

偏移地址: 0x00

复位值: 0x000 XX83, X代表未定义

访问: 无等待状态, 字, 半字 和字节访问

| | | | | | | | | | | | | | | | |
|-------------|----|--------------|---------|-----|----|---------|--------|---------|---------|--------|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | PLL RDY | PL隆 | 保留 | | CSS ON | HSE BYP | HSE RDY | HSE ON | | | | | |
| r | r | r | r | r | r | r | rw | | | rw | rw | r | r | rw | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| HSICAL[7:0] | | HSITRIM[4:0] | | 保留 | | HSI RDY | HSION | | | | | | | | |
| r | r | r | r | r | r | r | rw | rw | rw | rw | rw | rw | r | rw | |

再分析一下 RCC->CR |= 0X00010000 这个语句 16 进制 0x00010000 换成二进制，是第 17 位为 1，

其他 31 个位都为 0，寄存器是从 0 开始，就是 16 位 HSEON 置位

| | |
|-----|---|
| 位16 | HSEON: 外部高速时钟使能 由软件置'1'或清零。 当进入待机和停止模式时，该位由硬件清零，关闭外部时钟。当外部4-25MHz时钟被用作或被选择将要作为系统时钟时，该位不能被清零。 0: HSE振荡器关闭； 1: HSE振荡器开启。 |
|-----|---|

可以看到对 HSE ON 置位就是把外部的高速时钟使能，CPU 所需要的时钟是从外部获取；然后接下继续看另外一句代码：while(!(RCC->CR>>17))，在 while 循环里，如果为 1，while 就会一直循环，如果为 0 的话 while 就会继续执行；while 循环里还有！取反的操作，所以这句代码的意思三个是让 while 循环等到 CR 的第 17 位变成 1 才往下执行，那么看下第 17 位是什么，翻到《STM32F10xxx 参考手册》的 60 页：

| | |
|-----|--|
| 位17 | HSERDY: 外部高速时钟就绪标志 由硬件置'1'来指示外部4-25MHz时钟已经稳定。在HSEON位清零后，该位需要6个外部4-25MHz时钟周期清零。 0: 外部4-25MHz时钟没有就绪； 1: 外部4-25MHz时钟就绪。 |
|-----|--|

可以看到上图，第 17 位置 1 表示外部的高速时钟就绪好了，晶振起振稳定，CPU 从外部晶振稳定的获取时钟。

接下来，就是操作一个叫做 RCC_CFGR 寄存器，这个寄存器主要负责内部时钟的倍频（倍频就是加入输入进来是 8M 主频，经过 4 倍频后就是乘以 4 倍，等于 32M 了，所以倍频就相当于是乘）和分频（分频就相当于是除法，比如 72M 的主频，4 分频后就是 18M 了），是一个非常重要的寄存器。

6.3.2 时钟配置寄存器(RCC_CFGR)

偏移地址: 0x04

复位值: 0x0000 0000

访问: 0到2个等待周期, 字, 半字 和字节访问

只有当访问发生在时钟切换时, 才会插入1或2个等待周期。

| | | | | | | | | | | | | | | | |
|-------------|----|------------|----|------------|----|------------|-------------|----------|--------------|---------|------------|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | MCO[2:0] | | 保留 | | USB PRE | PLLMUL[3:0] | | PLL XTPRE | | PLL SRC | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADCPRE[1:0] | | PPRE2[2:0] | | PPRE1[2:0] | | HPRE[3:0] | | SWS[1:0] | | SW[1:0] | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | r | r | rw | rw |

这个寄存器主要负责配置 STM32 内部总线上的一些时钟参数, 比如时钟切换状态(可以知道目前哪个时钟作为系统时钟, 比如有内部的时钟, 也有外部的), 这个寄存器还负责 STM32 时钟内部总线的配置, 比如 AHB 时钟总线, APB1 时钟总线, APB2 时钟总线是怎么分频(有几种分频方式, 例如不分频, 2 分频, 4 分频, 8 分频, 16 分频等); 还有 ADC 模数转换的分频; USB 的分频等; 可以看到这个寄存器完成了许多的功能, 下面来分析我们如何操作这个寄存器的:

- 1) RCC->CFGR = 0x00000400;
- 2) RCC->CFGR |= 2<<18;
- 3) RCC->CFGR |= 1<<16;
- 4) RCC->CFGR |= 1<<1;

第 1 句代码是设置 PPRE1 寄存器为 100, 从下图可以知道, 设置 HCLK2 分频, 这里也可以不设置, 默认 HCLK 是不分频的, 我们在这里只是提醒大家, 做个配置师范:

| | |
|-------|--|
| 位10:8 | PPRE1: 低速APB预分频 (APB1) 由软件置'1'或清'0'来控制低速APB1时钟(PCLK1)的预分频系数。 注意: 软件必须保证APB1时钟频率不超过36MHz。 0xx: HCLK不分频 100: HCLK 2分频 101: HCLK 4分频 110: HCLK 8分频 111: HCLK 16分频 |
|-------|--|

第 2 句代码是将 2 左移 18 位, 即将二进制的 10 左边移动 18 位, 是 PLL 4 倍频输出:

| | |
|--------|---|
| 位21:18 | PLLMUL: PLL倍频系数 由软件设置来确定PLL倍频系数。只有在PLL关闭的情况下才可被写入。 注意: PLL的输出频率不能超过72MHz 0000: PLL 2倍频输出 1000: PLL 10倍频输出 0001: PLL 3倍频输出 1001: PLL 11倍频输出 0010: PLL 4倍频输出 1010: PLL 12倍频输出 0011: PLL 5倍频输出 1011: PLL 13倍频输出 0100: PLL 6倍频输出 1100: PLL 14倍频输出 0101: PLL 7倍频输出 1101: PLL 15倍频输出 0110: PLL 8倍频输出 1110: PLL 16倍频输出 0111: PLL 9倍频输出 1111: PLL 16倍频输出 |
|--------|---|

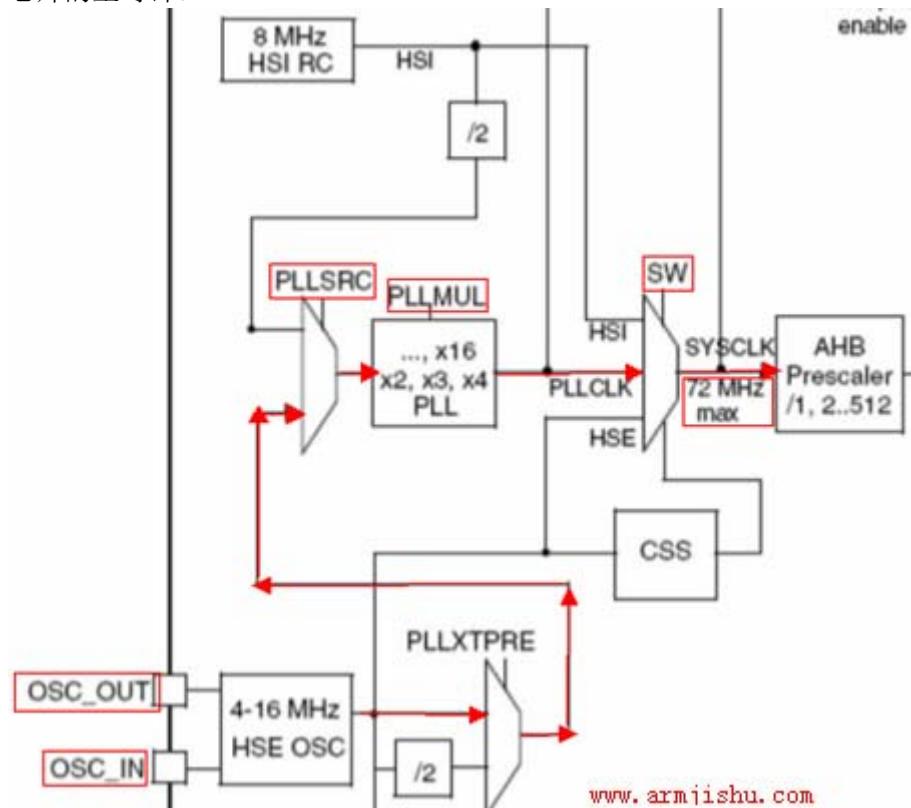
第 3 句代码是左移 16 位后置 1, 表示 HSE 时钟作为 PLL 输入时钟, HSE 就是外部晶振:

| | |
|-----|---|
| 位16 | PLLSRC: PLL输入时钟源 由软件置'1'或清'0'来选择PLL输入时钟源。该位只有在PLL关闭时才可以被写入。 0: HSI时钟2分频后作为PLL输入时钟 1: HSE时钟作为PLL输入时钟 |
|-----|---|

第 4 句代码表示将时钟切换到外部时钟，都设置好之后，通知 STM32 的芯片可以将系统的时钟切换给外部晶振提供时钟了：

| | |
|------|---|
| 位1:0 | SW: 系统时钟切换 由软件置'1'或清'0'来选择系统时钟源。 在从停止或待机模式中返回时或直接或间接作为系统时钟的HSE出现故障时，由硬件强制选择HSI作为系统时钟（如果时钟安全系统已经启动） 00: HSI作为系统时钟； 01: HSE作为系统时钟； 10: PLL输出作为系统时钟； 11: 不可用。 |
|------|---|

可以从下图看到，RCC_CFGR 寄存器里的 PLLSRC 负责切换是内部 HSI 提供时钟，还是外部 HSE 晶振提供时钟，上面程序代码选择的是外部的 HSE 提供时钟；紧接着，RCC_CFGR 寄存器里的 PLLMUL，程序代码里配置的是 4 倍频，如果这里外部晶振是 8MHz 的，4 倍频后就是 32MHz 的频率（STM32 最高可以达到 72MHz）；RCC_CFGR 寄存器里的 SW 负责 HIS、PLLCLK、HSE 三个时钟来源取其中一个，程序里选择的是 PLLCLK，这个来自 HSE 外部晶振然后 4 倍频的时钟做为整个 STM32 芯片的主时钟。



上图上面已经经过详细分析了，接下来就是一个 while(sws != 0x2)循环

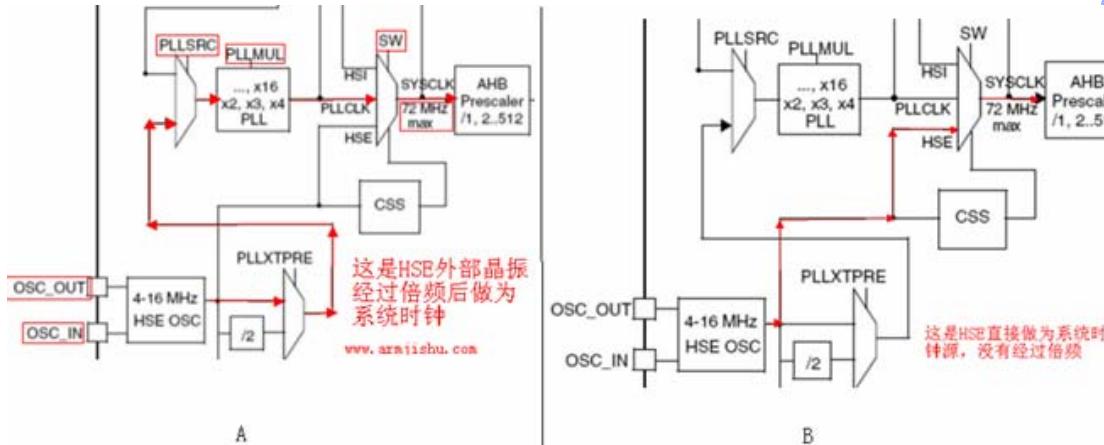
```
while(sws != 0x2)
{
    sws = RCC->CFGR>>2;
    sws &= 0x3;
}
```

上面的代码是将 RCC_CFGR 寄存器右移 2 位后，在与上 0x3 相当于与上二进制的 11，最低两位是 1，其他高位都是 0；这样的意思就是把 SWS 这个系统时钟切换状态的值单独截取出来，然后用 while(sws != 0x2)语句一直等待 SWS 的值为 0x2，0x2 化成二进制是 10，就是 PLL 输出作为系统时钟，

下图可以知道：

| | |
|------|---|
| 位3:2 | SWS: 系统时钟切换状态 由硬件置'1'或清'0'来指示哪一个时钟源被作为系统时钟。 00: HSI作为系统时钟； 01: HSE作为系统时钟； 10: PLL输出作为系统时钟； 11: 不可用。 |
|------|---|

那么这里就奇怪了，为什么不是 HSE 的晶振作为系统时钟呢？而是采用 PLL，因为我们可以看到，我们在代码里设置的就是走左边这张图的配置路线，因为把外部的晶振进行倍频了的，也有右边这种选择，只是这个例程我们在代码里没有这么设计，以后大家可以这样设计：



最后代码配置好了，就开始点灯程序，具体细节请见代码，因为上一章节已经有详细分析。

5.2.5 例程02 STM32芯片40MHz频率下跑点灯程序

1. 示例简介

让点灯程序在时钟主频 40MHz 下面运行，其他不变，同上个例程一样，唯一的改变就是把主频从 32MHz 改成 40MHz 了。

2. 调试说明：

下载代码，并且按下【复位】键，在神舟 III 号板上找到 DS1，可以看到该 DS1 灯一亮一灭。

3. 关键代码：

查看 RCC_CFGR 寄存器的第 18~21 位的 PLLMUL，当外部晶振是 8MHz 时，将 RCC->CFGR |= 3<<18; 将 16 进制的 3 化成二进制是 0011，把 PLL 设置成 5 倍频输出，8MHz 乘以 5 倍频就是 40MHz。

| | | | | | | | | | | | | | | | | | |
|-----------------|---|-----------------|------------------|-----------------|------------------|-----------------|------------------|-----------------|------------------|-----------------|------------------|-----------------|------------------|-----------------|------------------|-----------------|------------------|
| 位21:18 | PLLMUL: PLL倍频系数 由软件设置来确定PLL倍频系数。只有在PLL关闭的情况下才可被写入。 注意：PLL的输出频率不能超过72MHz | | | | | | | | | | | | | | | | |
| | <table style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td style="width: 50%;">0000: PLL 2倍频输出</td> <td style="width: 50%;">1000: PLL 10倍频输出</td> </tr> <tr> <td>0001: PLL 3倍频输出</td> <td>1001: PLL 11倍频输出</td> </tr> <tr> <td>0010: PLL 4倍频输出</td> <td>1010: PLL 12倍频输出</td> </tr> <tr> <td>0011: PLL 5倍频输出</td> <td>1011: PLL 13倍频输出</td> </tr> <tr> <td>0100: PLL 6倍频输出</td> <td>1100: PLL 14倍频输出</td> </tr> <tr> <td>0101: PLL 7倍频输出</td> <td>1101: PLL 15倍频输出</td> </tr> <tr> <td>0110: PLL 8倍频输出</td> <td>1110: PLL 16倍频输出</td> </tr> <tr> <td>0111: PLL 9倍频输出</td> <td>1111: PLL 16倍频输出</td> </tr> </tbody> </table> | 0000: PLL 2倍频输出 | 1000: PLL 10倍频输出 | 0001: PLL 3倍频输出 | 1001: PLL 11倍频输出 | 0010: PLL 4倍频输出 | 1010: PLL 12倍频输出 | 0011: PLL 5倍频输出 | 1011: PLL 13倍频输出 | 0100: PLL 6倍频输出 | 1100: PLL 14倍频输出 | 0101: PLL 7倍频输出 | 1101: PLL 15倍频输出 | 0110: PLL 8倍频输出 | 1110: PLL 16倍频输出 | 0111: PLL 9倍频输出 | 1111: PLL 16倍频输出 |
| 0000: PLL 2倍频输出 | 1000: PLL 10倍频输出 | | | | | | | | | | | | | | | | |
| 0001: PLL 3倍频输出 | 1001: PLL 11倍频输出 | | | | | | | | | | | | | | | | |
| 0010: PLL 4倍频输出 | 1010: PLL 12倍频输出 | | | | | | | | | | | | | | | | |
| 0011: PLL 5倍频输出 | 1011: PLL 13倍频输出 | | | | | | | | | | | | | | | | |
| 0100: PLL 6倍频输出 | 1100: PLL 14倍频输出 | | | | | | | | | | | | | | | | |
| 0101: PLL 7倍频输出 | 1101: PLL 15倍频输出 | | | | | | | | | | | | | | | | |
| 0110: PLL 8倍频输出 | 1110: PLL 16倍频输出 | | | | | | | | | | | | | | | | |
| 0111: PLL 9倍频输出 | 1111: PLL 16倍频输出 | | | | | | | | | | | | | | | | |

关键代码的改变与上面一模一样，唯一的改变就是将 `RCC->CFGR |= 2<<18;` 变成了 `RCC->CFGR |= 3<<18;` 这句不同

5.2.6 例程03 STM32芯片72MHZ频率下跑点灯程序

1.示例简介

让点灯程序在时钟主频 72MHz 下面运行，其他不变，同上个例程一样，唯一的改变就是把主频从 40MHz 改成 72MHz 了。

2.调试说明：

下载代码，并且按下【复位】键，在神舟 III 号板上找到 DS1，可以看到该 DS1 灯一亮一灭。

3.关键代码：

查看 `RCC_CFGR` 寄存器的第 18~21 位的 `PLLMUL`，当外部晶振是 8MHz 时，将 `RCC->CFGR |= 7<<18;`；将 16 进制的 7 化成二进制是 0111，把 PLL 设置成 9 倍频输出，8MHz 乘以 9 倍频就是 72MHz。

| 位21:18 | PLLMUL: PLL倍频系数 |
|------------------------------------|------------------------|
| 由软件设置来确定PLL倍频系数。只有在PLL关闭的情况下才可被写入。 | |
| 注意：PLL的输出频率不能超过72MHz | |
| 0000: PLL 2倍频输出 | 1000: PLL 10倍频输出 |
| 0001: PLL 3倍频输出 | 1001: PLL 11倍频输出 |
| 0010: PLL 4倍频输出 | 1010: PLL 12倍频输出 |
| 0011: PLL 5倍频输出 | 1011: PLL 13倍频输出 |
| 0100: PLL 6倍频输出 | 1100: PLL 14倍频输出 |
| 0101: PLL 7倍频输出 | 1101: PLL 15倍频输出 |
| 0110: PLL 8倍频输出 | 1110: PLL 16倍频输出 |
| 0111: PLL 9倍频输出 | 1111: PLL 17倍频输出 |

关键代码的改变与上面几乎一模一样，改变的是将 `RCC->CFGR |= 3<<18;` 变成了 `RCC->CFGR |= 7<<18;`。

注意：48M 和 72M 之间是 2 等待周期；增加了 FLASH 两个时钟周期的延时，CPU 的运行频率高，而内部 flash 的运行频率低所以 cpu 要从 flash 中取指令当然要有个等待的时间了。怎么协调：cortex 内核“意识”到它的缓冲区没有指令了，它就会去 flash 中预取，当缓冲区中有预取的指令，它自然就不执行预取了，当然 cortex 内核肯定不是当缓冲区中没有指令了才去取指令，而是它认为被 cpu 译码后剩余的预取指令少到一定程度后就去预取指令。关于 FLASH 更多资料请见《STM32F10xxx 闪存编程手册》 在该手册上可以找到：

3.1 Flash access control register (FLASH_ACR)

Address offset: 0x00

Reset value: 0x0000 0030

| | | | | | | | | | | | | | | | |
|----------|----|----|----|----|----|----|----|------------|------------|------------|---------|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | | | | | | | | |
| | | | | | | | | PRFT BS | PRFT BE | HLF CYA | LATENCY | | | | |
| | | | | | | | | r | rw | rw | rw | rw | rw | rw | |

Bits 31:6 Reserved, must be kept cleared.

Bit 5 PRFTBS: Prefetch buffer status

This bit provides the status of the prefetch buffer.

- 0: Prefetch buffer is disabled
- 1: Prefetch buffer is enabled

Bit 4 PRFTBE: Prefetch buffer enable

- 0: Prefetch is disabled
- 1: Prefetch is enabled

Bit 3 HLFCYA: Flash half cycle access enable

- 0: Half cycle is disabled
- 1: Half cycle is enabled

Bits 2:0 LATENCY: Latency

These bits represent the ratio of the SYSCLK (system clock) period to the Flash access time.

- 000 Zero wait state, if $0 < \text{SYSCLK} \leq 24 \text{ MHz}$
- 001 One wait state, if $24 \text{ MHz} < \text{SYSCLK} \leq 48 \text{ MHz}$
- 010 Two wait states, if $48 \text{ MHz} < \text{SYSCLK} \leq 72 \text{ MHz}$

所以在代码中，还要添加 `FLASH->ACR |= 0x32;` FLASH 相关的寄存器的声明定义等。具体看例程：“07. STM32 芯片 72MHz 频率下全速跑 LED 流水灯 (STM32 神舟 III 号)”。

5.3 独立按键

5.3.1 键的分类

目前，按键有多种形式。有机械接触式，电容式，轻触式等。

1. 按制作工艺分：

硬板按键：带弹簧的按键焊接在印刷电路板上

软板键盘：以导电橡胶作为接触材料放在以聚脂薄膜作为基底的印刷电路上所形成的按键。

2. 按工艺原理分：

可以将键盘分为编码键盘和非编码键盘，编码键盘的键盘电路内包含有硬件编码器，当按下某一个键后，键盘电路能直接提供与该键相对应的编码信息，例如 ASCII 码。非编码键盘的键盘电路中只有较简单的硬件，采用软件来识别按下键的位置，并提供与按下键相对应的中间代码送主机，然后由软件将中间代码转换成相应的字符编码，例如 ASCII 码；非编码键盘主要靠软件编程来识别的，在单片机组成的各种系统中，用的较多的是非编码键盘。非编码键盘又分为独立键盘和行列式（又称矩阵式）键盘。

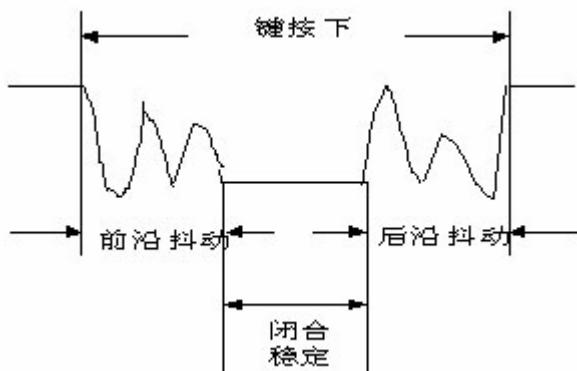
5.3.2 按键属性

键盘实际上就是一组按键，在单片机外围电路中，通常用到的按键都是机械弹性开关，当开关闭合时，线路导通，开关断开时，线路断开，下图是几种单片机系统常见的按键：



弹性小按键被按下时闭合，松手后自动断开；自锁式按键按下时闭合且会自动锁住，只有再次按下时才弹起断开。

单片机的外围输入控制用小弹性按键较好，单片机检测按键的原理是：单片机的 I/O 口既可作为输出也可作为输入使用，当检测按键时用的是它的输入功能，我们把按键的一端接地，另一端与单片机的某个 I/O 口相连，开始时先给该 I/O 口赋一高电平，然后让单片机不断地检测该 I/O 口是否变为低电平，当按键闭合时，即相当于该 I/O 口通过按键与地相连，变成低电平，程序一旦检测到 I / O 口变为低电平则说明按键被按下，然后执行相应的指令。



从上图可看出，理想波形与实际波形之间是有区别的，为什么呢？因为实际波形在按下和释放的瞬间会有抖动现象出现，这是因为通常的按键所用开关为机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开。因而在闭合及断开的瞬间均伴随有一连串的抖动，抖动时间的长短和按键的机械特性有关，一般为 5~10ms。为了不产生这种现象而作的措施就是按键消抖。通常我们手动按下键然后立即释放，这个动作中稳定闭合的时间超过 20ms。因此单片机在检测键盘是否按下时都要加上去抖动操作。



消抖是为了避免在按键按下或是抬起时电平剧烈抖动带来的影响。按键的消抖，可用硬件或软件两种方法，硬件的去抖主要是用专用的去抖动电路，也有专用的去抖动芯片；另外一种方式就是用软件延时的方法就能很容易解决抖动问题，而没有必要再添加多余的硬件电路。所以软件消抖适合按键比较多的情况，而硬件消抖适合按键比较少的情况。

如果按键较多，就用软件方法去抖，即检测出键闭合后执行一个延时程序，5ms~10ms 的延时，让前沿抖动消失后再一次检测键的状态，如果仍保持闭合状态电平，则确认为真正有键按下。当检测到按键释放后，也要给 5ms~10ms 的延时，待后沿抖动消失后才能转入该键的处理程序，这样就靠软件模拟整个按键的过程，控制只取最稳定的那个按键状态。

实现方法：一般来说，软件消抖的方法是不断检测按键值，直到按键值稳定。假设检测到按键按

下之后，为了避免检测到很多的抖动，可以先延时 5ms~10ms，再次检测，如果按键还被检测按下，那么就认为有一次按键输入（因为如果不避开抖动的话，会有很多次按键输入信号出现，通过去抖，模拟人的按下的过程和时间，按下和松开按键实际也占用了 20ms 以上的时间，记录正确的按键次数。

5.3.3 STM32的位带操作

1. 什么是位带操作

还记得 51 单片机吗？单片机 51 中也有位的操作，以一位（BIT）为数据对象的操作；例如 51 单片机可以简单的将 P1 端口的第 2 位独立操作，P1.2=0 或者 P1.2=1，就是这样把 P1 口的第三个脚(bit2)置 0（输出低电平）或者置 1（输出高电平）。

而现在 STM32 的位段、位带别名区这些就是为了实现这样的功能，可以在 SRAM、I/O 外设空间实现对这些区域的某一位的单独直接操作。

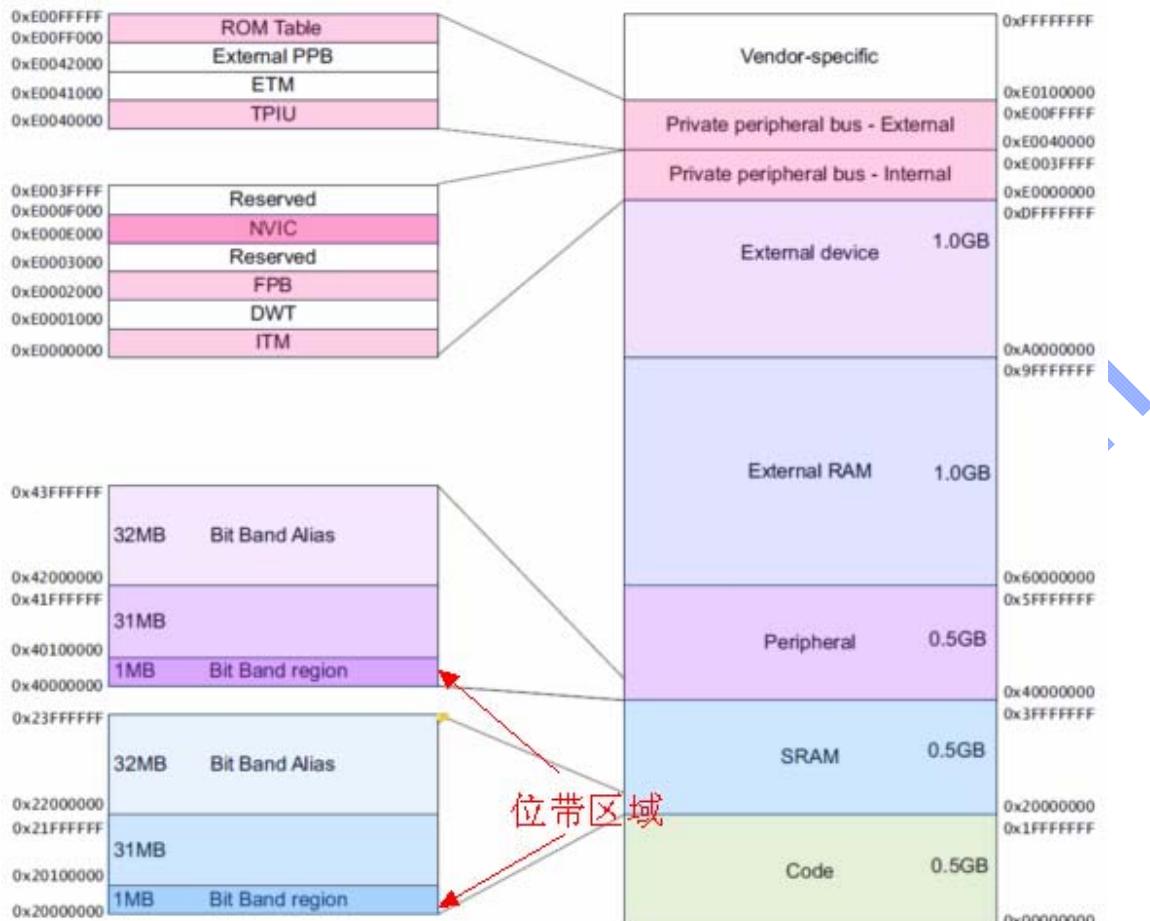
2. 为什么要用位带操作？

那么 51 单片机中间不是有位的操作吗，而 STM32 为什么要提出位带的操作呢？首先，这里不得不提一个事情就是 STM32 的内部区域访问只能是 32 位的字，不能是字节或者半字，这部分 STM32 在神舟开发板手册的 GPIO 章节中提到过；而 51 单片机里一个 bit（一个字节等于 8 个 bit，一个字是 32 个 bit）。这个是 STM32 的特点决定的，所以 STM32 使用一种新型的方式来解决这个问题，设计一个办法来解决用一次访问 32bit 的这样的操作达到 51 单片机那种只访问一个 bit 的效果。

3. 如何设计和实现位带操作的？

从编程者这个角度来说，我们操作的对象是一个一个的 bit 位，而对于 STM32 来说，它内内部只能是 32 位 bit 每次的访问。如果要实现这个技术，必须要做一个映射，也就是从 1 个 bit 映射到 32 个 bit，就是用 STM32 内部的一次访问（32 个 bit）来代表编程者认为的 1 个 bit。

那 STM32 内部是如何解决的呢？它是在支持位带操作的地方，取个别名区空间，而这个别名区空间可以让一次 32 位来进行访问，对这个别名进行操作就相当于对 SRAM 或者 I/O 存储空间中的位（1 个寄存器里的位就是 1 个 bit，1 个 bit 最后对应别名区空间的 32 个位，因为 STM32 芯片内部只能是 32 位去访问）进行操作。

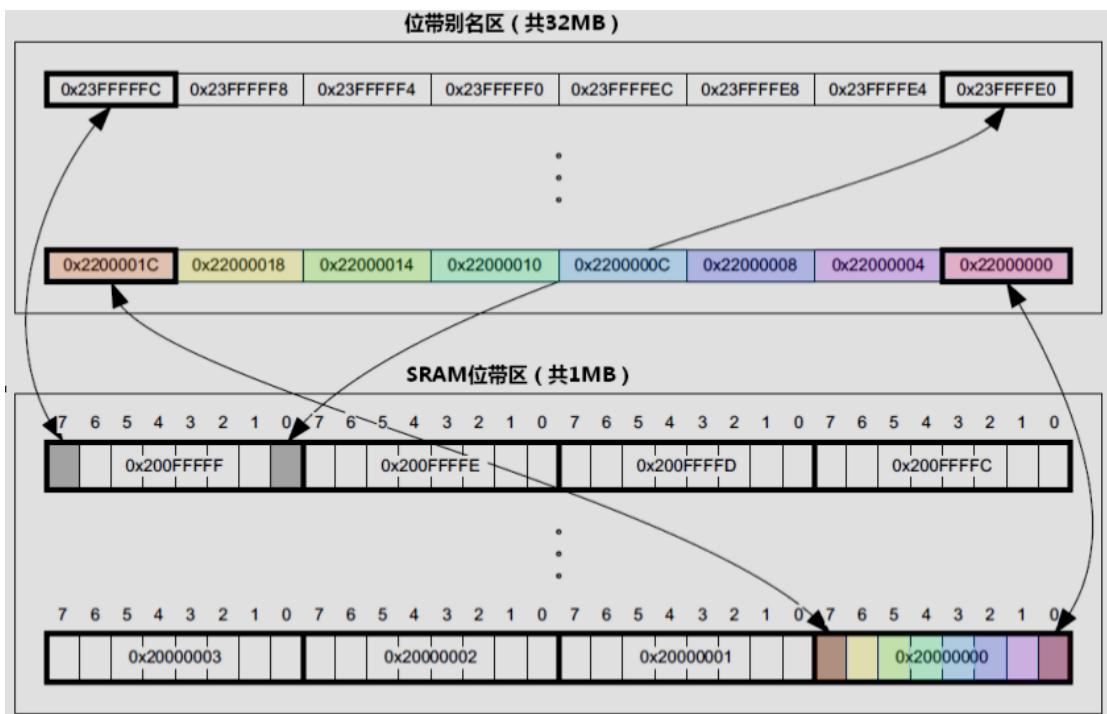


这样呢, 1MB SRAM 就可以 32M 个对应别名区空间, 就是 1 位膨胀到 32 位(1bit 变为 1 个字); 我们对这个别名区空间开始的某一字操作, 置 0 或置 1, 就等于它映射的 SRAM 或 I/O 相应的某地址的某一位的操作。

4. STM32 中位带操作的具体部署情况是 支持位带操作的两个内存区的范围是:

| 序号 | 支持位带操作的两个内存区的范围 | 对应的别名区空间范围 |
|----|--|--|
| 1 | SRAM 区中的最低 1MB: 0x2000_0000-0x200F_FFFF | SRAM 所对应的别名区 32MB 空间: 0x2200_0000-0x23FF_FFFF |
| 2 | 片上外设区中的最低 1MB: 0x4000_0000-0x400F_FFFF | 片上外社区所对应的别名区 32MB 空间: 0x4200_0000-0x43FF_FFFF |

下面是内部空间映射图:



例如：SRAM 区中的最低 1MB 空间中的 0x2000_0000 的 8 个 bit，分别对应如下：

| 地址 | 对应的 bit 位 | 别名空间 | Bit 对应的别名空间 |
|-------------|-----------|-------------|--------------------------|
| 0x2000_0000 | Bit1 | 0x2200_0000 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit2 | 0x2200_0004 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit3 | 0x2200_0008 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit4 | 0x2200_000C | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit5 | 0x2200_0010 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit6 | 0x2200_0014 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit7 | 0x2200_0018 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit8 | 0x2200_001C | 跨度一个字 = 4 个字节 = 32 个 bit |

可以看到上表和上图，0x2000_0000 中的一个 bit 位对应了别名区的一个 32 位的字，也就是说 STM32 芯片的内部寄存器的任意一个位，都其实对应的是别名区的 32 个位。

5. 如何用代码与位带操作挂钩

在 STM32 中，一个寄存器是 32 位的，32 个 bit 中的任意其中一个 bit 所对应的别名空间到底该如何访问呢？首先分为两种情况，一种是在 SRAM，一种是在 FLASH 中，两个别名空间的起地位置是不同的，SRAM 是从 0x2200_0000 开始，而 FLASH 是从 0x4200_0000 开始。

假如在 SRAM 中的一个寄存器的地址是 A，访问寄存器 A 中的第 n 个 bit 位。那么该如何计算呢？我们知道 SRAM 中别名区的起始地址是 0x2200_0000 对应 SRAM 中实际寄存器地址 0x2000_0000，SRAM 中每 1 个 bit，都会对应别名区中的 32 个 bit，那么实际地址的公式应该如下：

$$0x2200_0000 + (\text{SRAM 实际寄存器地址偏移 } 0x2000_0000 \text{ 的 bit 数}) * 4$$

因为 0x2200_0000 这个地址每增加 1，实际上就是增加 8 个 bit（一个地址对应一个字节），实际寄存器中的 1 个 bit 对应 32 个 bit，所以就乘以 4，地址本身增加 1 是 8bit，8bit 乘以 4 倍刚好是 32bit。

那么接下来“SRAM 实际寄存器地址偏移 0x2000_0000 的 bit 数”该如何计算呢？对，用寄存器的地址减去这个基地址，然后在乘以 8（因为一个地址对应 8 个 bit），所以就可

以得到以下的公式：

$$(A - 0x20000000)*8$$

以上这个公式可以知道实际寄存器离基地址有多少个 bit 的距离，访问该寄存器的第 n 个 bit 位还必须加上一个 n，就变成以下的公式：

$$(A - 0x20000000)*8+n$$

好了，最后整理整个换算公式如下，FLASH 与 SRAM 的原理都是想通的：

SRAM :0x22000000 +((A - 0x20000000)*8+n)*4

FLASH :0x42000000 +((A - 0x40000000)*8+n)*4

6. 举例说明：

比如我要访问如下寄存器 GPIOB_BSRR 中的第 14bit 位 BS13，注意因为寄存器内部是从 0 开始计数到 31 截止，所以第 14bit 相当于第 13。

8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

地址偏移：0x10

复位值：0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|--------|------|--|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 位31:16 | | BRy: 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。 | | | | | | | | | | | | | |
| 位15:0 | | BSy: 设置端口x的位y (y = 0...15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 设置对应的ODRy位为1 | | | | | | | | | | | | | |

可以从下图看到，GPIO 端口 B 的起始地址是 x04001_0C00，GPIOB_BSRR 寄存器的偏移地址是 0x10，访问的第 14bit 位的 BS13。

| | |
|----------|---------------------------|
| SPI1 | 0x4001 3000 - 0x4001 33FF |
| TIM1 | 0x4001 2C00 - 0x4001 2FFF |
| ADC2 | 0x4001 2800 - 0x4001 2BFF |
| ADC1 | 0x4001 2400 - 0x4001 27FF |
| Port G | 0x4001 2000 - 0x4001 23FF |
| Port F | 0x4001 1C00 - 0x4001 1FFF |
| Port E | 0x4001 1800 - 0x4001 1BFF |
| Port D | 0x4001 1400 - 0x4001 17FF |
| Port C | 0x4001 1000 - 0x4001 13FF |
| Port B | 0x4001 0C00 - 0x4001 0FFF |
| Port A | 0x4001 0800 - 0x4001 0BFF |
| EXTI | 0x4001 0400 - 0x4001 07FF |
| AFIO | 0x4001 0000 - 0x4001 03FF |
| Reserved | 0x4000 7800 - 0x4000 FFFF |
| DAC | 0x4000 7400 - 0x4000 77FF |

那么通过公式：

FLASH :0x42000000 +((A - 0x40000000)*8+n)*4

换算 0x4200_0000 + ((0x40010c00-0x40000000)*8 + 12)*4 = 实际地址

在这里我们就不具体计算了,SRAM 访问也是同理

7. 如何将理念转化成代码：

由上面几节得出，SRAM 和 FLASH 中别名区的寻址公式如下：

SRAM :0x22000000 +((A - 0x20000000)*8+n)*4

FLASH :0x42000000 +((A - 0x40000000)*8+n)*4

可以看到 0x2200_0000 和 0x4200_0000 都共同有个 x200_0000 这个数值；乘以 8 相当于再加上外面的那个乘以 4，总共是乘以 32，而 n 是乘以 4；乘以 32 相当于是数值向左移 5 位，乘以 4 相当于向左边移 2 位。

尝试将如下公式化成

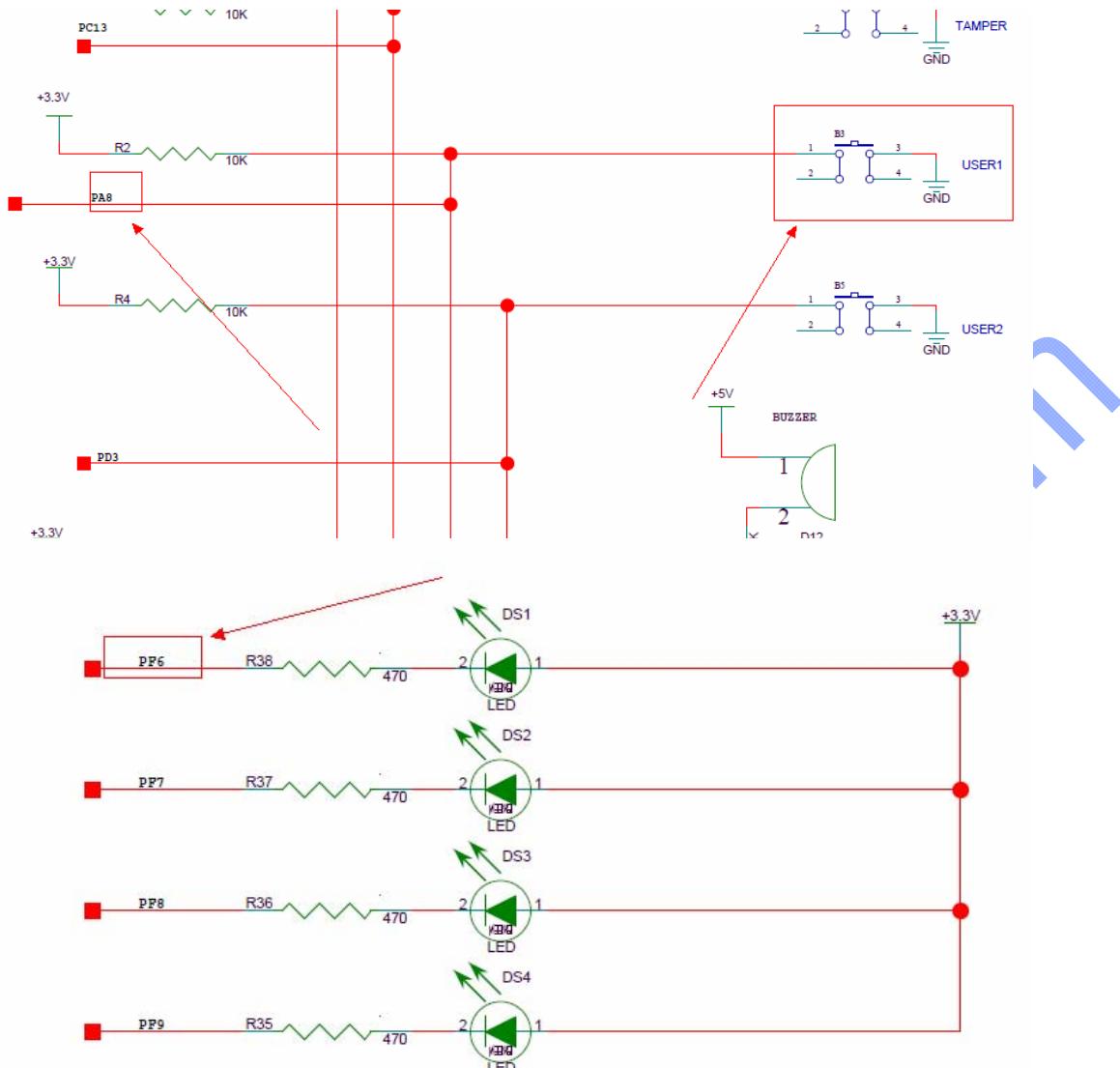
```
#define BITBAND(addr, bitnum)
    ((addr & 0xF0000000)+0x20000000+((addr & 0xFFFF)<<5)+(bitnum<<2))
```

这样就可以用 BITBAND(addr, bitnum) 来表示寄存器里的任何一个 bit 位，大概原理讲到这里，具体使用方法通过例程来体现。

5.3.4 例程01 STM32芯片按键点灯（无防抖）

1. 示例简介

通过连上 PA8 管脚的按键，学会如何从 PA8 管脚读入一个输入，如何配置 GPIO 口成输入状态；当 PA8 获取到输入的数据，STM32 对 LED 灯的状态进行取反。按键 USER1 是一端连接了 GND，一端连接了 PA8 管脚；当按键按下时，PA8 管脚的电平值被拉低，相关电路图如下图所示：



2. 调试说明:

按下 PA8 管脚所连的按键（按钮 USER1），每按一次，LED 灯会由亮变灭，或者又灭变亮，因为没有防抖代码（下个例程会增加），会发现，有时候按下去，灯会亮灭好几次。

3. 关键代码:

```
int main(void) //main 是程序入口
{
    unsigned int key_up =1;
    RCC_init();      //初始化配置时钟频率为 72MHZ
    LED_init();      //LED 初始化配置
    Key_init();      //初始化控制按键的 PA0 端口

    while (1)
    {
        if(key_up)
            LEDON;          // 开灯
        else
            LEDOFF;         // 关灯
    }
}
```

```

    If( KEY0 == 0)
        key_up = !key_up;
    }
}
}

```

STM32 芯片的 GPIO 管脚要采集到按键按下，那么它需要被配置成输入模式，之前有讨论 GPIO 管脚的几种模式，我们接下来还是复习一下，我们看看端口低配置寄存器 CRH 的描述，如下图所示：

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---------------|-------------|--|-------------|------------|-------------|------------|-------------|----|----|----|----|----|----|----|----|
| CNF15[1:0] | MODE15[1:0] | CNF14[1:0] | MODE14[1:0] | CNF13[1:0] | MODE13[1:0] | CNF12[1:0] | MODE12[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF11[1:0] | MODE11[1:0] | CNF10[1:0] | MODE10[1:0] | CNF9[1:0] | MODE9[1:0] | CNF8[1:0] | MODE8[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位31:30 | | CNFy[1:0]: 端口x配置位(y = 8...15) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | | | | | | | | | | | | |
| 位9:28 | | MODEy[1:0]: 端口x的模式位(y = 8...15) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz | | | | | | | | | | | | | |

该寄存器的复位值为 0X4444 4444 (4 化成二进制为 0100)，从上图可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRH 控制着每个 IO 端口 (A~G) 的高 8 位的模式。每个 IO 端口的位占用 CRH 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式 (ADC 用)、0X3 表示推挽输出模式 (做输出口用，50M 速率)、0X8 表示上/下拉输入模式 (做输入口用)、0XB 表示复用输出 (使用 IO 口的第二功能)。

STM32 的 IO 口位配置表如下表：

| 配置模式 | | CNF1 | CNF0 | MODE1 | MODE0 | PxODR寄存器 |
|--------|----------------|------|------|---------------------------|--------------------------|----------------------|
| 通用输出 | 推挽式(Push-Pull) | 0 | 0 | 01 10 11 见表3.1.2 | 0或1 0或1 不使用 不使用 | 不使用 不使用 0 1 |
| | 开漏(Open-Drain) | | 1 | | | |
| 复用功能输出 | 推挽式(Push-Pull) | 1 | 0 | | | |
| | 开漏(Open-Drain) | | 1 | | | |
| 输入 | 模拟输入 | 0 | 0 | 00 | 不使用 不使用 0 1 | 不使用 不使用 0 1 |
| | 浮空输入 | | 1 | | | |
| | 下拉输入 | 1 | 0 | | | |
| | 上拉输入 | | 1 | | | |

可以看到 CNFX 是上面 CRH 寄存器里的配置位，这里配置位的不同，就会产生不同的 GPIO 管脚模式。

STM32 输出模式配置如下表：

| MODE[1:0] | 意义 |
|-----------|--------------|
| 00 | 保留 |
| 01 | 最大输出速度为10MHz |
| 10 | 最大输出速度为2MHz |
| 11 | 最大输出速度为50MHz |

这里 CRH 寄存器的 MODE 配置位的选项，不同的配置就会产生不同的速率。

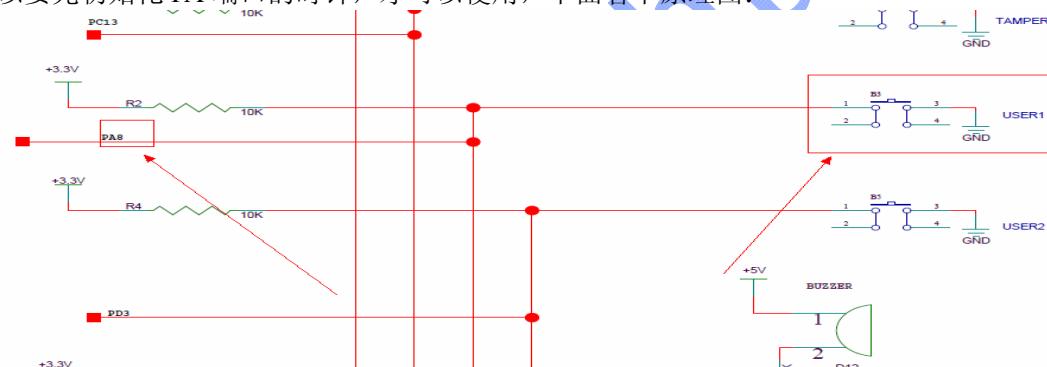
CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口，大家可以自己看 STM32 手册，我们在这里 CRL 就不做详细介绍。

首先看下代码：RCC_init();这里实现的是初始化配置时钟频率为 72MHZ，之前的时钟章节已经将过了，具体细节可以看上一章的内容，有详细的介绍。

下面看下连接按键的 GPIO 管脚的具体设置，这个函数里有 3 句代码：

```
void Key_init()
{
    RCC->APB2ENR |= RCC_APB2Periph_GPIOA; //使能 PORTA 时钟
    GPIOA->CRH &= 0xFFFFFFFF0;
    GPIOA->CRH |= 0X00000008; //PA8 设置成输入,PA8 在按键原理图默认被上拉的
}
```

RCC->APB2ENR |= RCC_APB2Periph_GPIOA 首先使能 PORTA 的时钟，接下来开始使用 PA0 管脚，所以要先初始化 PA 端口的时钟，才可以使用，下面看下原理图：



可以看到 PA0 默认是被上拉到 3.3V 高电平的，GPIOA->CRH &= 0xFFFFFFFF0; 这句代码是将 PA0 的 GPIO_CRH 寄存器的前 4 位清 0，然后 GPIOA->CRH |= 0X00000008 这句话是将 GPIO_CRH 寄存器的前 4 位赋值 0x8，化成二进制就是 1000，通过查表：

8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)

偏移地址: 0x04

复位值: 0x4444 4444

| | | | | | | | | | | | | | | | |
|------------|--|------------|-------------|------------|-------------|------------|-------------|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF15[1:0] | MODE15[1:0] | CNF14[1:0] | MODE14[1:0] | CNF13[1:0] | MODE13[1:0] | CNF12[1:0] | MODE12[1:0] | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF11[1:0] | MODE11[1:0] | CNF10[1:0] | MODE10[1:0] | CNF9[1:0] | MODE9[1:0] | CNF8[1:0] | MODE8[1:0] | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 位31:30 | CNFy[1:0]: 端口x配置位(y = 8...15) (Port x configuration bits) 软件通过这些位配置相应的I/O端口, 请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | | | | | | | | | | | | | |
| 27:26 | | | | | | | | | | | | | | | |
| 23:22 | | | | | | | | | | | | | | | |
| 19:18 | | | | | | | | | | | | | | | |
| 15:14 | | | | | | | | | | | | | | | |
| 11:10 | | | | | | | | | | | | | | | |
| 7:6 | | | | | | | | | | | | | | | |
| 3:2 | | | | | | | | | | | | | | | |
| 位9:28 | MODEy[1:0]: 端口x的模式位(y = 8...15) (Port x mode bits) 软件通过这些位配置相应的I/O端口, 请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz | | | | | | | | | | | | | | |
| 25:24 | | | | | | | | | | | | | | | |
| 21:20 | | | | | | | | | | | | | | | |
| 17:16 | | | | | | | | | | | | | | | |
| 13:12 | | | | | | | | | | | | | | | |
| 9:8, 5:4 | | | | | | | | | | | | | | | |
| 1:0 | | | | | | | | | | | | | | | |

可以知道将 GPIOA_CRL 寄存器的 PA8 管脚配置成 1000, 即上拉/下拉输入模式, 配置好 PA8 管脚之后, 如何才能知道按键按下呢? 这时就需要时刻查看和监听 PA8 管脚是否有电平的变化, 那就要知道 PA8 的管脚的电平值, 而且还要不停的去检测它是否有变化, 因为我们希望当按键按下时候, 我们能以最快的反映速度获取到这一动作。

在代码里, 我们使用 if(KEY0==0) 来判断按键是否按下了, KEY0 就是 PA8, 如果按键按下, 从原理图可以知道 PA8 被拉到 GND 变成低电平, 这样 PA8 就等于 0 即 KEY0 等于 0。

那么 KEY0 如何反映到 PA8 的值的呢? 接下来继续分析代码:

```
(1) #define BITBAND(addr, bitnum)
    ((addr & 0xF0000000)+0x2000000+((addr & 0xFFFF)<<5)+(bitnum<<2))
(2) #define MEM_ADDR(addr)    *((volatile unsigned long *) (addr))
(3) #define BIT_ADDR(addr, bitnum)      MEM_ADDR(BITBAND(addr, bitnum))
(4) #define PAin(n)        BIT_ADDR(GPIOA_IDR_Addr, n)
(5) #define KEY0          PAin(8)
```

代码 (1): 上面有讲解, 这个 BITBAND(addr,bitnum)最后得出的是某寄存器里的某个 bit 位所映射的别名区的地址。

代码 (2): `unsigned long *`表示强制把这个地址变成一个 32 位的长的地址的指针, 或者说是一个指针, 指向 32 位的一个地址; 然后`*((volatile unsigned long *) (addr))`这个表示这个地址, 长达 32 位 bit 的值, 通过 `MEM_ADDR(addr)` 把从 `addr` 地址开始的连续 32 个 bit 里的值给取出来。

代码 (3): `BIT_ADDR(addr, bitnum)`表示某寄存器的地址, 对应的某 bit 位所对应的别名区的空间的值, 也就是说取出某寄存器的某 bit 位的值

代码 (4): 把 PA 的某个管脚, 与 `BIT_ADDR` 进行绑定关联

代码 (5): 把 KEY0 设置成 PA8, 使得 KEY0 能取到 `GPIOA_IDR` 寄存器里第 9 位也就是 PA8 的值

这样, 这个例程的关键是是否取到了 PA8 的值, 取到了值之后, 再进行相关的操作, 这个是大家

可以自己定义的，在这个例程中，按一下按键，我们就将 LED 灯取反，原来是亮的就变成灭的，原来是灭的就变成亮的。

5.3.5 例程02 STM32芯片按键点灯-增加了防抖的代码

1. 示例简介：

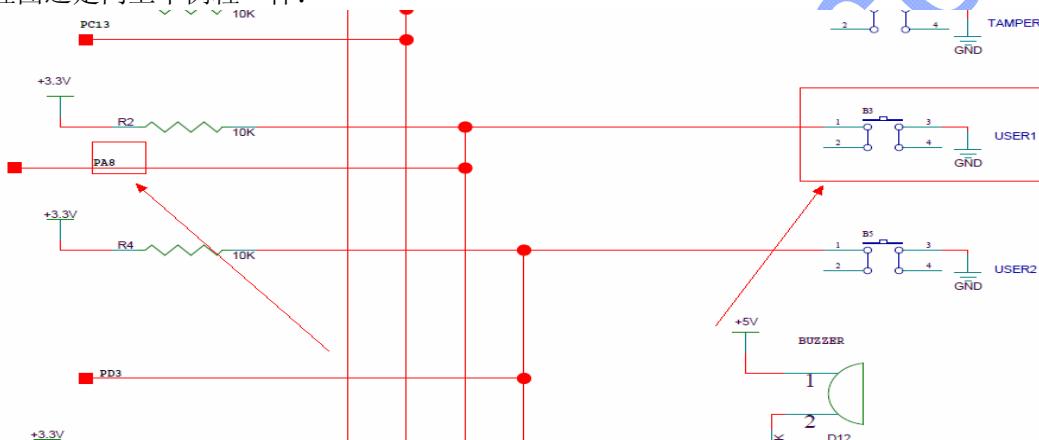
其他都与上个例程相同，唯一不同的就是增加了防抖代码，在这里是用软件防抖

2. 调试说明：

按下 PA8 管脚所连的按键（按钮 USER1），每按一次，LED 灯会由亮变灭，或者又灭变亮，因为增加了防抖代码，基本上可以做到按一次，就采集到一次数据，灯由亮变灭或者又灭变亮，非常的稳定。

3. 关键代码：

原理图还是同上个例程一样：



```

int main(void) //main是程序入口
{
    unsigned int key_up =1;
    RCC_init(); //初始化配置时钟频率为72MHZ
    LED_init(); //LED初始化配置
    Key_init(); //初始化控制按键的PA0端口

    while (1)
    {
        Delay(0xffff); // 增加了防抖功能，如果你按下按键的时候会有抖动
        if(key_up)
            LEDON; // 开灯
        else
            LEDOFF; // 关灯

        if(KEY0==0)
            key_up = !key_up; //取反
    }
}

```

代码Delay(0xffff)就是我们增加的防抖功能，如果你按下按键的时候会有抖动，我们增加一定

时间的时间再来判断按键是否按下来，这样在一定程度上可以起到消抖的作用，但是这里还有一个BUG就是如果长时间按着按键不动，while循环里面就会运行多次灯点亮程序，如果是计数器的话，这个就不准了，按一次键，就计算了许多次数，大家可以尝试一下如何去解决这个问题，按一次键就只亮一次，无论一次按多长时间。

5.4 串口通信的收与发

5.4.1 什么是串口通信

串口通信是指外设和计算机间，通过数据信号线、地线、控制线等，按位进行传输数据的一种通讯方式。这种通信方式使用的数据线少，在远距离通信中可以节约通信成本，但其传输速度比并行传输低。

串口是计算机上一种非常通用的设备通信协议。大多数计算机（不包括笔记本电脑）包含两个基于RS-232的串口。串口同时也是仪器仪表设备通用的通信协议（串口通信协议也可以用于获取远程采集设备的数据）。

当年51单片机内置串口的时候，被认为是微控制器发展史上的重大事件，因为当时的串口是唯一一个微控制器与PC交互的接口。MCU微控制器经过这么多年的发展，串口仍然是其必不可少的接口之一。

5.4.2 串口通信的属性

1. 通信存在的问题

评价一个通信是否优质，主要体现在传输的速度，数据的正确性，功耗是否低，布线成本是否低（例如1根线收发都能满足就比8根线的并行收发要节约成本）；使用是否普及（就好像大家都学英语，世界很大部分的人都可以独立使用英语吗，会英语的人多，就非常普及，可通信面就非常广；如果你学的鸟语，那就只能跟鸟通信，没有人能听懂）。

2. 串口到底有几个标准？（经常听说有3线、5线串口）

传统的串行接口标准有22根线，采用标准25芯D型插头座(DB25)，后来使用简化为9芯D型插座(DB9)，现在应用中25芯插头座已很少采用。

像现在所说的几线串口，一般都是指使用了几根线，最初的RS-232串口是25针的，所有的针脚定义都有用到，后来变成了9针的，所谓全功能串口就是所有的针脚定义都使用上了，例如流量控制，握手信号等都有用到，一般来说国外的产品做产品比较规矩，把所有的串口信号都做上去了。但是国内的技术人员发现，其实RS-232串口最主要使用的就是2,3线，另外的接口如果不使用的话，也不会出现很大的问题，所以，就在9针的基础上做精简，所以就有所谓的2,3,4,5,6,8线的串口出来了。

2线串口只有RXD,TXD两根基本的收发信号线；3线串口除了RXD和TXD，还有GND；所谓4~9线只是在TXD和RXD基础上增加了相应的控制信号线，依据实际需要进行设计。

一般来说，使用5线的232通信，是加了硬件流控的，即RTS,CTS信号，主要是为了保证高速通信时的可靠性，如果你的通信速度不是很高，完全可以不用理会。

3. 串口的速度与距离

RS-232（串口的英文代名词）采取不平衡传输方式，即所谓单端通讯。由于其发送电平与接收电平的差仅为2V至3V左右，所以其共模抑制能力差，再加上双绞线上的分布电容，其传送距离最长为约15米，最高速率为20kb/s。RS-232是为点对点（即只用一对收、发设备）通讯而设计的，其驱动器负载为3~7kΩ。所以RS-232适合本地设备之间的通信。

4. 从串口通信衍生出422与485的通信方式

RS-232、RS-422 与 RS-485 都是串行数据接口标准，最初都是由电子工业协会（EIA）制订并发布的，RS-232 在 1962 年发布，命名为 EIA-232-E，作为工业标准，以保证不同厂家产品之间的兼容。

RS-422 由 RS-232 发展而来，它是为弥补 RS-232 之不足而提出的。为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mb/s，传输距离延长到 4000 英尺（速率低于 100kb/s 时），并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。

为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。

由于 EIA 提出的建议标准都是以“RS”作为前缀，所以在通讯工业领域，仍然习惯将上述标准以 RS 作前缀称谓。

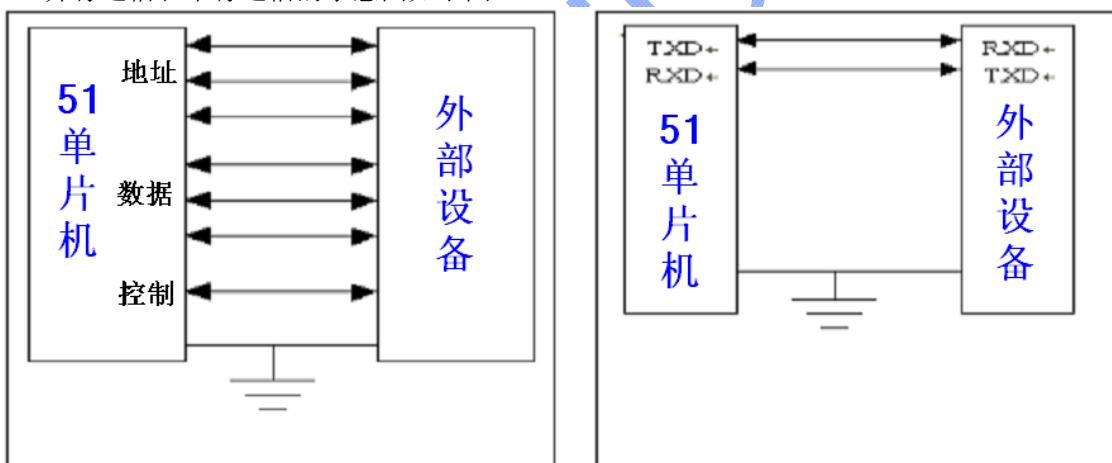
RS-232、RS-422 与 RS-485 标准只对接口的电气特性做出规定，而不涉及接插件、电缆或协议，在此基础上用户可以建立自己的高层通信协议。因此在视频界的应用，许多厂家都建立了一套高层通信协议，或公开或厂家独家使用。如录像机厂家中的 Sony 与松下对录像机的 RS-422 控制协议是有差异的，视频服务器上的控制协议则更多了，如 Louth、Odetis 协议是公开的，而 ProLINK 则是基于 Profile 上的。

5. 串口的通信方式（串口属于串行通信）

（1）并行通信和串行通信

51 单片机与外界通信的基本方式有两种：并行通信和串行通信，并行通信是指利用多条数据传输线将一个数据的各位同时发送或接收。串行通信是指利用一条传输线将数据一位位地顺序发送或接收。

并行通信和串行通信的示意图如下图：



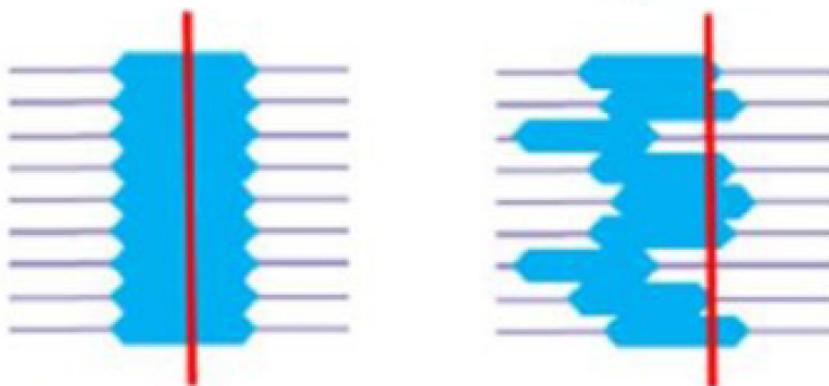
在每一条传输线传输速率相同时，**并行通信的传输速度比串行通信快**。然而当传输距离变长时，**并行通信的缺点就会凸显**，首先是相比于串行通信而言信号易受外部干扰，信号线之间的相互干扰也增加，其次是速率提升之后不能保证每根数据线的数据同时到达接收方而产生接收错误，而且距离越长布线成本越高。

所以并行通信目前主要用在短距离通信，比如处理器与外部的 flash 以及外部 RAM 以及芯片内部各个功能模块之间的通信。串行通信以其通信速率快和成本低等优点成为了远距离通信的首选。RS232C 串口，以及差分串行总线像 RS485 串口、USB 接口、CAN 接口、IEEE-1394 接口、以太网接口、SATA 接口和 PCIE 接口等都属于串行通信的范畴。

下图左侧为每根数据线的数据同时到达接收方，被正确采样的最理想情况；右侧的图为每根数据线的数据不能同时到达接收方而产生接收错误情形。

理想的数据采样点

并行数据经过不等长
线路传输到达接收方
的数据采样点

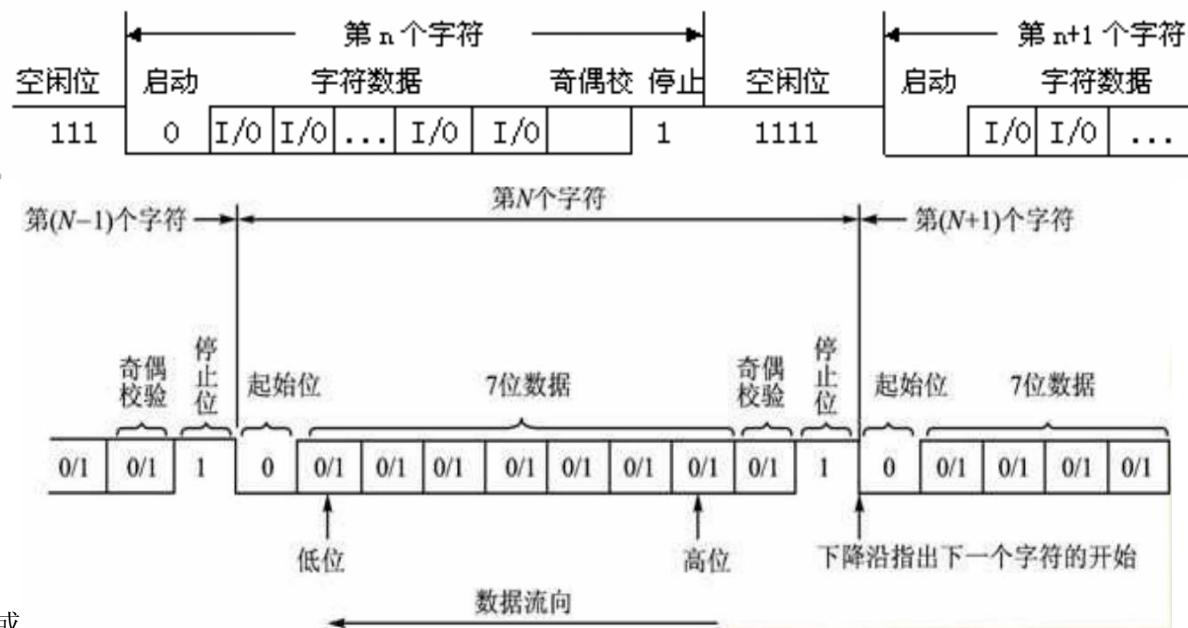


(2) 异步通信与同步通信

串行通信又分为两种方式：异步通信与同步通信。

A、异步通信及其协议

异步通信以一个字符为传输单位，通信中两个字符间的时间间隔不固定可以是任意长的，然而在同一个字符中的两个相邻位代码间的时间间隔是固定的，接收时钟和发送时钟只要相近就可以。通信双方必须使用约定的相同的一些规则（也叫通信协议）。常见的传送一个字符的信息格式规定有起始位、数据位、奇偶校验位、停止位等，其中各位的意义如下：



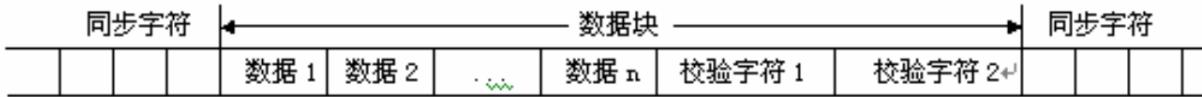
或

- ① 起始位 先发出一个逻辑“0”信号，表示传输字符的开始。
- ② 数据位 紧接着起始位之后。数据位的个数可以是 5、6、7、8 等，构成一个字符。一般采用扩展的 ASCII 码，范围是 0~255，使用 8 位表示。首先传送最低位。
- ③ 奇偶校验位（不是必须） 奇偶校验是串口通信中一种简单的检错方式，当然没有校验位也是可以的。数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)，以此来校验数据传送的正确性。例如，如果数据是 01100000，那么对于偶校验，校验位为 0。
- ④ 停止位 它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。由于数据是在传

输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率同时也越慢。

⑤ 空闲位 处于逻辑“1”状态，表示当前线路上没有数据传送。

B、同步通信是指数据传送是以一个帧（数据块或一组字符）为传输单位，每个帧中包含有多个字符。在通信过程中，字符与字符之间、字符内部的位与位之间都同步，每个字符间的时间间隔是相等的，而且每个字符中各相邻位代码间的时间间隔也是固定的。同步通信的数据格式如图所示



同步通信的特点可以概括为：

- ① 以数据块为单位传送信息。
- ② 在一个数据块（信息帧）内，字符与字符间无间隔。
- ③ 接收时钟与发送进钟严格同步

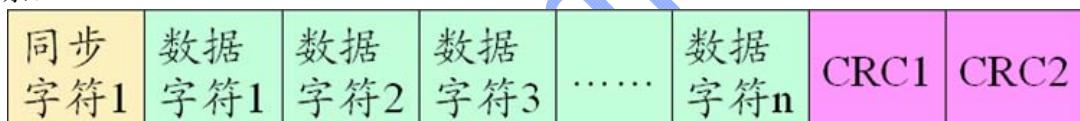
同步串行通信方式中一次连续传输一块数据，开始前使用同步信号作为同步的依据。同步字符的插入可以是单同步字符方式或双同步字符方式，均由同步字符、数据字符和校验字符 CRC 等三部分组成：

同步字符位于帧结构开头，用于确认数据字符的开始。

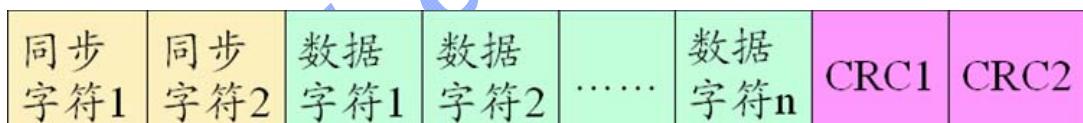
数据字符在同步字符之后，字符个数不受限制，由所需传输的数据块长度决定；

校验字符有 1~2 个，位于帧结构末尾，用于接收端对接收到的数据字符的正确性的校验。

由于连续传输一个数据块，故收发双方时钟必须相当一致，否则时钟漂移会造成接收方数据辨认错误。这种方式下往往是发送方在发送数据的同时也通过一根专门的时钟信号线同时发送时钟信息，接收方使用发送方的时钟来接由数据。同步串行通信方式传输效率高，但对硬件要求高，电路结构复杂。



(a) 单同步字符帧格式



(b) 双同步字符帧格式

图nnn 同步传送的数据格式

所有的串行接口电路都是以并行数据形式与 CPU 接口、而以串行数据形式与外部逻辑接口。所以串口对外应该是串行发送的，速度慢，但是比并行传输要稳定很多。

6. 串口是如何解决干扰以及校验的问题

什么是数据校验？通俗的说，就是为保证数据的完整性，用一种指定的算法对原始数据计算出一个校验值。接收方用同样的算法计算一次校验值，如果和随数据提供的校验值一样，就说明数据是完整的。

为了理解数据校验，什么是最简单的校验呢？最简单的校验就是把原始数据和待比较数据直接进行比较，看是否完全一样这种方法是最安全最准确的。同时这样的比对方式也是效率最低的。只适用于简单的数据量极小的通信。

串口通信使用的是奇偶校验方法，具体实现方法是在数据存储和传输中，字节中额外增加一个比特位，用来检验错误。校验位可以通过数据位异或计算出来；也就是说单片机串口通讯有一模式就是一次发送 8 位的数据通讯，增加一位第 9 位用于放校验值。

奇偶校验是一种校验代码传输正确性的方法。根据被传输的一组二进制代码的数位中“1”的个数是奇数或偶数来进行校验。采用奇数的称为奇校验，反之，称为偶校验。采用何种校验是事先规定好的。通常专门设置一个奇偶校验位，用它使这组代码中“1”的个数为奇数或偶数。若用奇校验，则当接收端收到这组代码时，校验“1”的个数是否为奇数，从而确定传输代码的正确性。

奇偶校验能够检测出信息传输过程中的部分误码(1 位误码能检出,2 位及 2 位以上误码不能检出)，同时，它不能纠错。在发现错误后，只能要求重发。但由于其实现简单，仍得到了广泛使用。

5.4.3 什么是单片机的TTL电平？

单片机是一种数字集成芯片，数字电路中只有两种电平：高电平和低电平；高电平和低电平是通过单片机的管脚进行输入和输出的，我们只要记住一句话，单片机管脚不是输入就是输出，不是高电平就是低电平。

为了让大家在初学的时候对电平特性有一个清晰的认识，我们暂且定义单片机输出与输入为 TTL 电平，其中高电平为+5V，低电平为 0V。计算机的串口出来的为 RS-232C 电平，其中高电平为-5V—-12V，低电平为+5V—+12V。这里要强调的是，RS-232C 电平为负逻辑电平，所以高电平为负的，低电平为正的，大家千万不要认为上面是我写错了，因此当计算机与单片机之间要通信时，需要加电平转换芯片，我们在神舟 51 单片机实验板上所加的电平转换芯片是 MAX3232（在串口 DB9 座附近）。初学者在学习时先掌握上面这点就够了，若有兴趣请大家再看下面的知识点——常用逻辑电平。

知识点：常用逻辑电平

常用的逻辑电平有 TTL、CMOS、LVTTL、ECL、PECL、GTL、RS-232、RS-422、RS-485、LVDS 等。其中 TTL 和 CMOS 的逻辑电平按典型电压可分为四类：5V 系列(5V 的 TTL 和 5V 的 CMOS)、3.3V 系列，2.5V 系列和 1.8V 系列。

5V 的 TTL 和 5V 的 CMOS 是通用的逻辑电平。3.3V 及以下的逻辑电平被称为低电压逻辑电平，常用的为 LVTTL 电平。低电压逻辑电平还有 2.5V 和 1.8V 两种。

那为什么 TTL 电平信号用的最多呢？

原因 1：这是因为大部分数字电路器件都用这个电平标准。就好像我们学英语，国际通用英语这门语言，那大家都用这个语言进行交流和沟通，所以后来的人都要学习英语才能彼此相互能交流。所以使得越来越多的电路器件使用这个电平标准。TTL 电平数据表示通常采用二进制，+5V 等同于逻辑 1，0V 等同于逻辑 0，这被称为 TTL（晶体管—晶体管逻辑电平）信号系统，这是计算机处理器控制的设备内部各部分之间通信的标准技术。TTL 电平信号对于计算机处理器控制的设备内部的数据传输是很理想的，首先计算机处理器控制的设备内部的数据传输对于电源的要求不高，热损耗也较低，另外 TTL 电平信号直接与集成电路连接而不需要价格昂贵的线路驱动器以及接收器电路。

原因 2：TTL 电平的特点适合设备内数据高速的传输。TTL 的通信大多数情况是采用并行数据传输方式，但电平最高为+5V，电压相对比较低，所以传输过程中会有电压损耗和压降，导致 TTL 的传输距离是有限的，一般只适合近距离传输；而且并行数据传输对于超过 10 英尺的距离就可能会有同步偏差，传输距离太远，有可能造成数据不同步；所以 TTL 电平符合近距离（在芯片内部或者计算机内部进行高速数据交互）高速的并行传输，在数字电路要求数据处理速度高的时代来说，选择 TTL 这个标准是正确的，可靠的。

CMOS 电平最高可达 12V，CMOS 电路输出高电平在 3V~12V 之间，而输出低电平接近 0 伏。CMOS 电路中不使用的输入端不能悬空，否则会造成逻辑混乱。另外，CMOS 集成电路因为电源电压可以在较大范围内变化，因而对电源的要求不像 TTL 集成电路那样严格。

TTL 电路和 CMOS 电路的逻辑电平关系如下：

1) CMOS 是场效应管构成，TTL 为双极晶体管构成；因为 TTL 和 CMOS 的高低电平的值不一样，所以互相连接时需要电平的转换。

2) TTL 电路是电流控制器件，而 CMOS 电路是电压控制器件。

3) TTL 电路的速度快，传输延迟时间短(5~10ns)，但是功耗大；CMOS 电路的速度慢，传输延迟时间长(25~50ns)，但功耗低，CMOS 电路本身的功耗与输入信号的脉冲频率有关，频率越高，芯片集越

热，这是正常现象。

4) CMOS 集成电路电源电压可以在较大范围内变化，因而对电源的要求不像 TTL 集成电路那样严格。所以，用 TTL 电平在条件允许下他们就可以兼容。要注意到他们的驱动能力是不一样的，CMOS 的驱动能力会大一些，有时候 TTL 的低电平触发不了 CMOS 电路，有时 CMOS 的高电平会损坏 TTL 电路，在兼容性上需注意。

5) CMOS 的高低电平之间相差比较大、抗干扰性强，TTL 则相差小，抗干扰能力差。

6) CMOS 的工作频率较 TTL 略低。

TTL 电平临界值：

1) TTL 输出电压：逻辑电平 1 = 2.4V，逻辑电平 0 = 0.4V

2) TTL 输入电压：逻辑电平 1 = 2.0V，逻辑电平 0 = 0.8V

CMOS 电平临界值（设电源电压为+5V）

1) CMOS 输出电压：逻辑电平 1 = 4.99V，逻辑电平 0 = 0.01V

2) CMOS 输入电压：逻辑电平 1 = 3.5V，逻辑电平 0 = 1.5V

常用逻辑芯片的特点如下：

74LS 系列：TTL 输入：TTL 输出：TTL

74HC 系列：CMOS 输入：CMOS 输出：CMOS

74HCT 系列：CMOS 输入：TTL 输出：CMOS

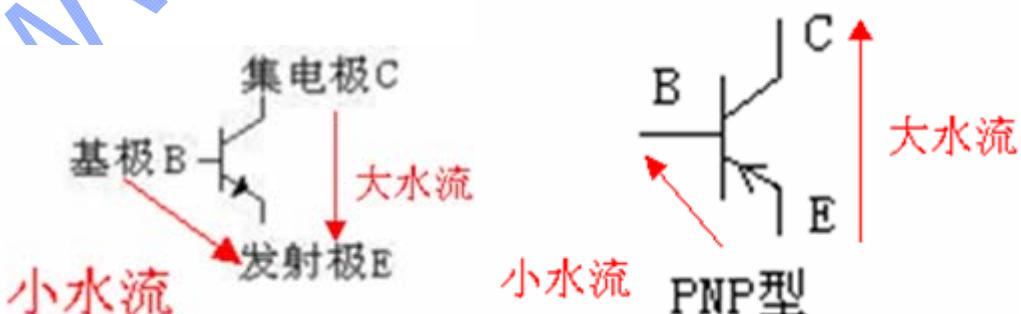
CD4000 系列：CMOS 输入：CMOS 输出：CMOS

通常情况下，单片机、ARM、DSP、FPGA 等各个器件之间引脚能否直接相连要参考以下方法进行判断：一般来说，同电压的是可以相连的，不过最好还是好好查看芯片技术手册上的 VIL（逻辑电平 0 的输入电压）、VIH（逻辑电平 1 的输入电压）、VOL（逻辑电平 0 的输出电压）、VOH（逻辑电平 1 的输出电压）的值，看是否能够匹配。有些情况在一般应用中没有问题，虽然参数上有点不够匹配，但还是在管脚的最大和最小容忍值范围之内，不过有可能在某些情况下可能就不够稳定，所以我们在设计电路的时候要尽量保持匹配，这样是最佳的设计。

5.4.4 关于NPN和PNP的三极管基础知识？

对三极管放大作用的理解，切记一点：能量不会无缘无故的产生，所以，三极管一定不会产生能量，但三极管厉害的地方在于：它可以通过小电流控制大电流。放大的原理就在于：通过小的交流输入，控制大的静态直流。假设三极管是个大坝，这个大坝奇怪的地方是，有两个阀门，一个大阀门，一个小阀门。小阀门可以用人力打开，大阀门很重，人力是打不开的，只能通过小阀门的水力打开。所以，平常的工作流程便是，每当放水的时候，人们就打开小阀门，很小的水流涓涓流出，这涓涓细流冲击大阀门的开关，大阀门随之打开，汹涌的江水滔滔流下。如果不不停地改变小阀门开启的大小，那么大阀门也相应地不停改变，假若能严格地按比例改变，那么，完美的控制就完成了。

在这里，基极 B → 发射极 E 就是小水流，集电极 C → 发射极 E 就是大水流。当然，如果把水流比为电流的话，会更确切，因为三极管毕竟是一个电流控制元件。



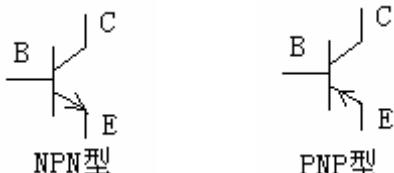
如果某一天，天气很旱，江水没有了，也就是大的水流那边是空的。管理员这时候打开了小阀门，尽管小阀门还是一如既往地冲击大阀门，并使之开启，但因为没有水流的存在，所以，并没有水流出来。这就是三极管中的截止区。

饱和区是一样的，因为此时江水达到了很大很大的程度，管理员开的阀门大小已经没用了。如果不关阀门江水就自己冲开了，这就是二极管的击穿。

在模拟电路中，一般阀门是半开的，通过控制其开启大小来决定输出水流的大小。没有信号的时候，水流也会流，所以，不工作的时候，也会有功耗。

而在数字电路中，阀门则处于开或是关两个状态。当不工作的时候，阀门是完全关闭的，没有功耗。

那么NPN与PNP的三极管到底有些什么区别呢？



NPN和PNP主要就是电流方向和电压正负不同，说得“专业”一点，就是“极性”问题。

NPN 是用 $B \rightarrow E$ 的电流（小水流）控制 $C \rightarrow E$ 的电流（大水流），E极电位最低，且正常放大时通常C极电位最高，即 $VC > VB > VE$ 。

PNP 是用 $E \rightarrow B$ 的电流（小水流）控制 $E \rightarrow C$ 的电流（大水流），E极电位最高，且正常放大时通常C极电位最低，即 $VC < VB < VE$ 。

半导体三极管也称为晶体三极管，可以说它是电子电路中最重要的器件。它最主要的功能是电流放大和开关作用。接下来的一些使用中会用到。

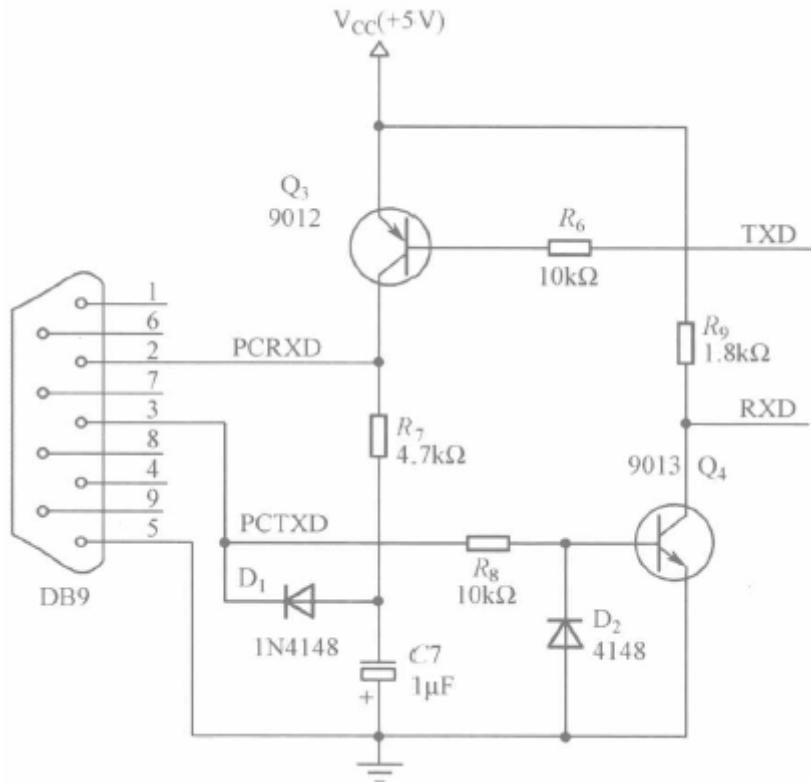
5.4.5 RS-232电平与TTL电平的转换

关于RS-232电平与TTL电平的特性在前面已经讲过，本节主要讲解使用较多的计算机RS-232电平与单片机TTL电平之间的转换方式。MAX232等芯片可实现RS-232电平到TTL电平的转换，但是现在用的较多还有MAX202，HIN232等芯片，它们同时集成了RS-232电平和TTL电平之间的互转。为丰富大家的知识，下面首先讲解在没有MAX3232这种现成电平转换芯片时，如何用二极管、三极管、电阻、电容等分立元件搭建一个简单的RS-232电平与TTL电平之间的转换电路。

1. 用单独的电容 电阻 三极管 实现 RS-232 电平与 TTL 电平转换 电路

集成芯片内部都是由最基本电子元件组成，如电阻、电容、二极管、三极管等元件，为了方便用户使用，制造商把这些具有一定功能的分立元件封装到一个芯片内，这样就制成了我们使用的各种芯片。学会本电路后，我们也就基本搞清了 MAX232 芯片内部的大致结构。

MAX232是把TTL电平从 $0V \sim 5V$ 转换到 $3V \sim 15V$ 或 $-3V \sim -15V$ 之间。如下图所示：



(1) 若发送低电平0，首先TXD（TTL低电平）发送数据时，TXD上是低电平，这时Q3导通（具体请看上节三极管的描述），PCRXD由空闲时的低电平变高电平，满足条件。

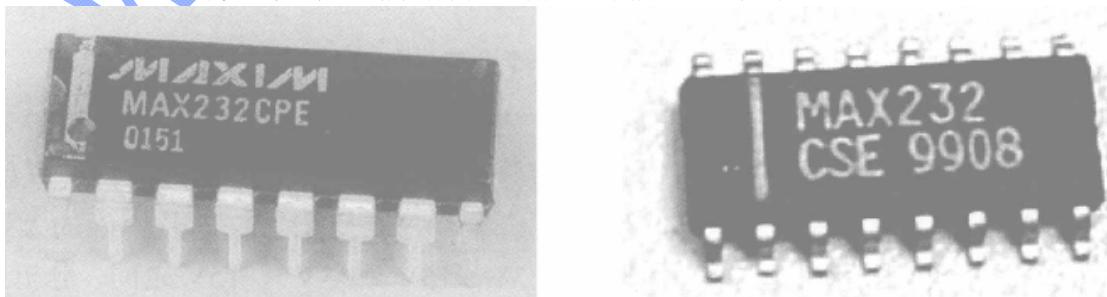
(2) 发送高电平1时，TXD为高电平，Q3截止，由于PCRXD内部高阻，而PCTXD平时是-3~-15V（RS-232的高电平就是负的电压，这点是要注意的，高电平并不是正电压），通过D1和R7将其拉低PCRXD至-3~-15V，此时计算机接收到的就是1。

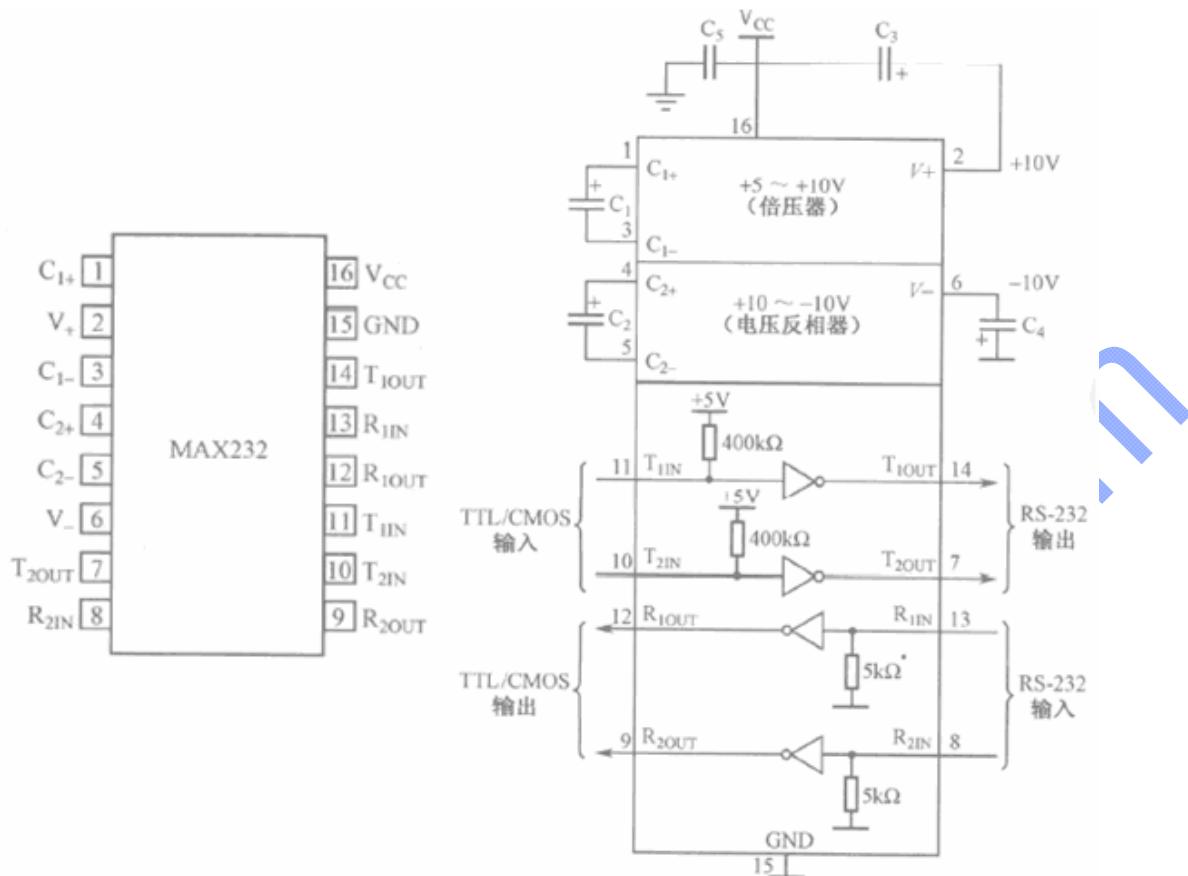
下面再反过来，PC发送信号，由单片机来接收信号。当PCTXD为低电平-3~-15V时，Q4截止，单片机端的RXD被R9拉到5V高电平；当PCTXD变高时，Q4导通，RXD被Q4拉到低电平，这样便实现的双向转换，这是一个很好的电路，值得大家学习。

2. MAX232芯片实现RS-232电平与TTL电平转换

MAX232 芯片是 MAXIM 公司生产的、包含两路接收器和驱动器的 IC 芯片，它的内部有一个电源电压变换器，可以把输入的+5V 电源电压变换成为 RS-232 输出电平所需的+10V 电压。所以，采用此芯片接口的串行通信系统只需单一的+5V 电源就可以了。对于没有+12V 电源的场合，其适应性更强，加之其价格适中，硬件接口简单，所以被广泛采用。

MAX232芯片实物和其引脚结构和外围连接如下图所示：





在上图中上半部分电容C1, C2, C3, C4及V+, V-是电源变换电路部分。在实际应用中，器件对电源噪声很敏感，因此VCC必须要对地加去耦电容C5，其值为0.1uF。按芯片手册中介绍，电容C1, C2, C3, C4应取1.0uF/16V的电解电容，经大量实验及实际应用，这4个电容都可以选用0.1uF的非极性瓷片电容代替1.0uF/16V的电解电容，在具体设计电路时，这4个电容要尽量靠近MAX232芯片，以提高抗干扰能力。

图下半部分为发送和接收部分。实际应用中，T1IN, T2IN可直接连接TTL/CMOS电平的stm32主芯片的串口发送端TXD；R1out, R2out可直接连接TTL/CMOS电平的stm32主芯片的串行接收端RXD；T1out, T2out可直接连接PC机的RS-232串口的接收端RXD；R1IN, R2IN可直接连接PC机的RS-232串口的发送端TXD。

现从MAX232芯片中两路发送、接收中任选一路作为接口。要注意其发送、接收的引脚要对应。如使T1IN连接单片机的发送端TXD，则PC机的RS-232接收端RXD一定要对应接T1out引脚。同时，R1out连接单片机的RXD引脚，PC机的RS-232发送端TXD对应接R1IN小引脚。

5.4.6 串口波特率的理解

在信息传输通道中，携带数据信息的信号单元叫码元，每秒钟通过信道传输的码元数称为码元传输速率，简称波特率。波特率是指数据信号对载波的调制速率，它用单位时间内载波调制状态改变的次数来表示(也就是每秒调制了符号数)，其单位是波特(Baud, symbol/s)。波特率是传输通道频宽的指标。

它是对信号传输速率的一种度量。但是波特率有时候会同比特率混淆，实际上后者是对信息

传输速率（传信率）的度量。当 1 波特等于 1 比特的时候，波特率与比特率才相等；但是如果 1 波特等于 8 比特的时候，那么每秒钟发送的比特率是波特率的 9 倍，波特率可以被理解为单位时间内传输码元符号的个数（传符号率），通过不同的调制方法可以在一个码元上负载多个比特信息。

所以，如果用公式表示，比特率在数值上和波特率有这样的关系：

波特率与比特率的关系为：比特率=波特率 X 单个调制状态对应的二进制位数。

单片机或计算机在串口通信时的速率用波特率表示，它定义为每秒传输二进制代码的位数，即 1 波特=1 位 / 秒，单位是 bps（位 / 秒）。如每秒钟传送 240 个字符，而每个字符格式包含 10 位（1 个起始位、1 个停止位、8 个数据位），这时的波特率为 $10 \text{ 位} \times 240 \text{ 个 / 秒} = 2400 \text{ bps}$ 。

串行接口或终端直接传送串行信息位流的最大距离与传输速率及传输线的电气特性也有关。当传输线使用每 0.3m（约 1 英尺）有 50pF 电容的非平衡屏蔽双绞线时，传输距离随传输速率的增加而减小。当比特率超过 1000 bps 时，最大传输距离迅速下降，如 9600 bps 时最大距离下降到只有 76m（约 250 英尺）。因此我们在做串口通信实验选择较高速率传输数据时，尽量缩短数据线的长度，为了能使数据安全传输，即使是在较低传输速率下也不要使用太长的数据线。

5.4.7 STM32 神舟 III 号独特的 USB 转串口的 TTL 电平模块设计

RS232 接口作为标准外设广泛应用于单片机和嵌入式系统，通用串行总线 USB(Universal Serial Bus) 通信技术以其易插拔、速度快、即插即用和独立供电等特点，已得到更广泛的应用。

STM32 神舟 III 号比较小巧玲珑，采用了一种基于 PL2303 的 RS232 与 USB 转换的设计方案，作为开发板与电脑串口之间的交互接口。PL2303 是高集成度的通用串行总线(USB)与串口的接口转换器，可方便将现有基于 RS232 接口的设备转换为 USB 接口。

PL2303 是 Prolific 公司生产的一种高度集成的 RS232-USB 接口转换器，可提供一个 RS232 全双工异步串行通信装置与 USB 功能接口便利联接的解决方案。该器件内置 USB 功能控制器、USB 收发器、振荡器和带有全部调制解调器控制信号的 UART，只需外接几只电容就可实现 USB 信号与 RS232 信号的转换，能够方便嵌入到手持设备。该器件作为 USB / RS232 双向转换器，一方面从主机接收 USB 数据并将其转换为 RS232 信息流格式发送给外设；另一方面从 RS232 外设接收数据转换为 USB 数据格式传送回主机。这些工作全部由器件自动完成，开发者无需考虑固件设计。

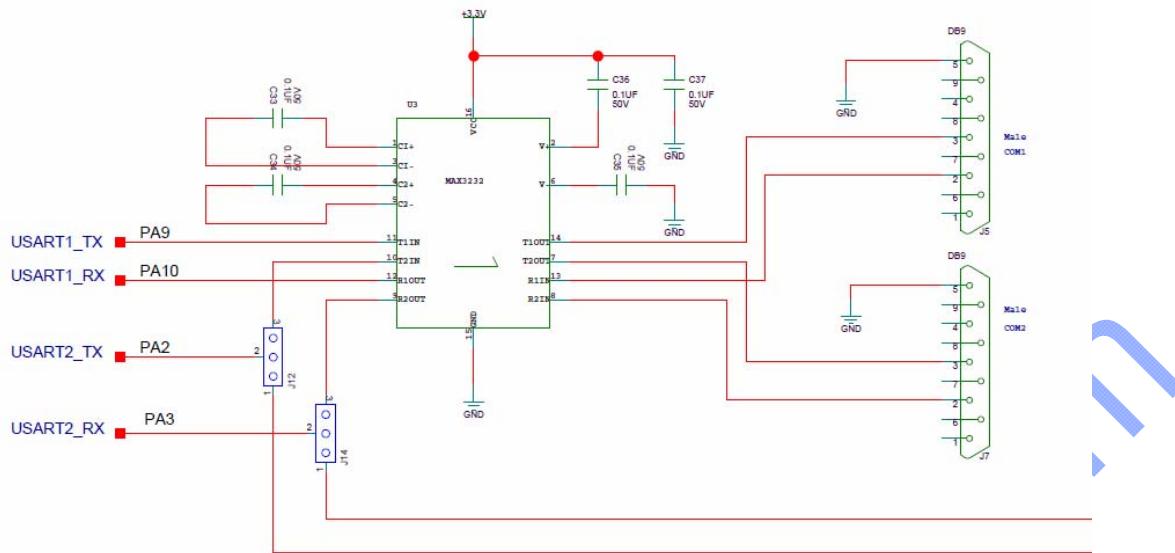
市场上主要的 USB 转串口芯片有 FT232、PL2303、CH340 三种，三个常用的芯片稳定程度和价格是一致的，FT232>CH340>PL2303，PL2303 用的最多，因为最便宜，国内很多开发板板子上，包括 USB 转串口线用的都是这种芯片，几元钱一片，电路也简单，做简单的串口应用可以，但是做嵌入式开发如使用超级终端波特率在 115200 时就有可能出现延迟等现象。CH340 是南京沁恒的芯片，做的还不错，对于普通应用完全能够满足。最好的是 FT232 稳定、可靠，在很多 USB 转串口的下载线、编程器中使用的都是这一种，神舟开发板上目前使用的是 PL2303HX 芯片。

USB 转串口芯片转出来串口电平就是 TTL 电平，高电平一般是 3.3V，如果转出来的电平再经过 MAX232 或 MAX485 芯片再转一下就会输出 RS232 电平或者 485 电平。其实市面上的 USB 转串口线一般都是这样接的。

5.4.8 例程01 最简单串口打印\$字符

1. 例程简介：

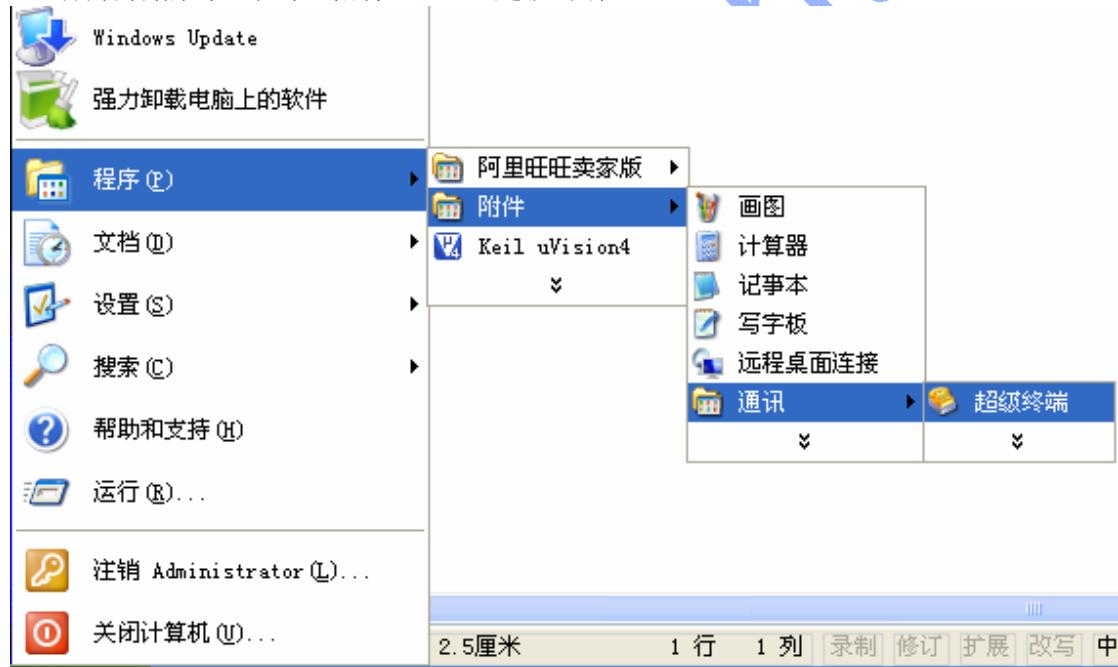
STM32 的 GPIOA 端口的 PA9 和 PA10 位，即串口 1；设置 PA9 为 TX 输出模式，复用功能推挽输出模式；设置 PA10 为 RX 输入模式，模拟输入模式；对超级终端打印输出字符“\$”符号。



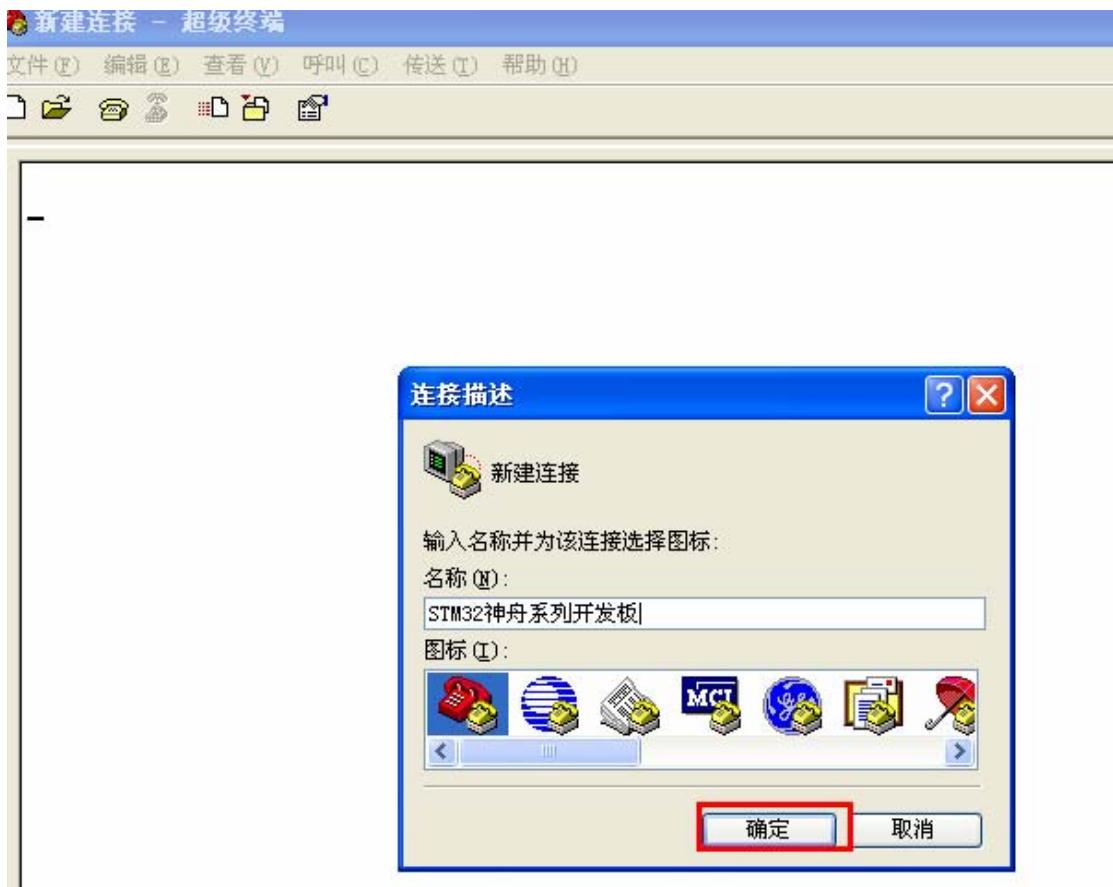
拿串口线将 STM32 神舟 III 号开发板上的串口 1 和电脑的串口连起来，具体硬件电路这里不细说。

2. 调试说明：

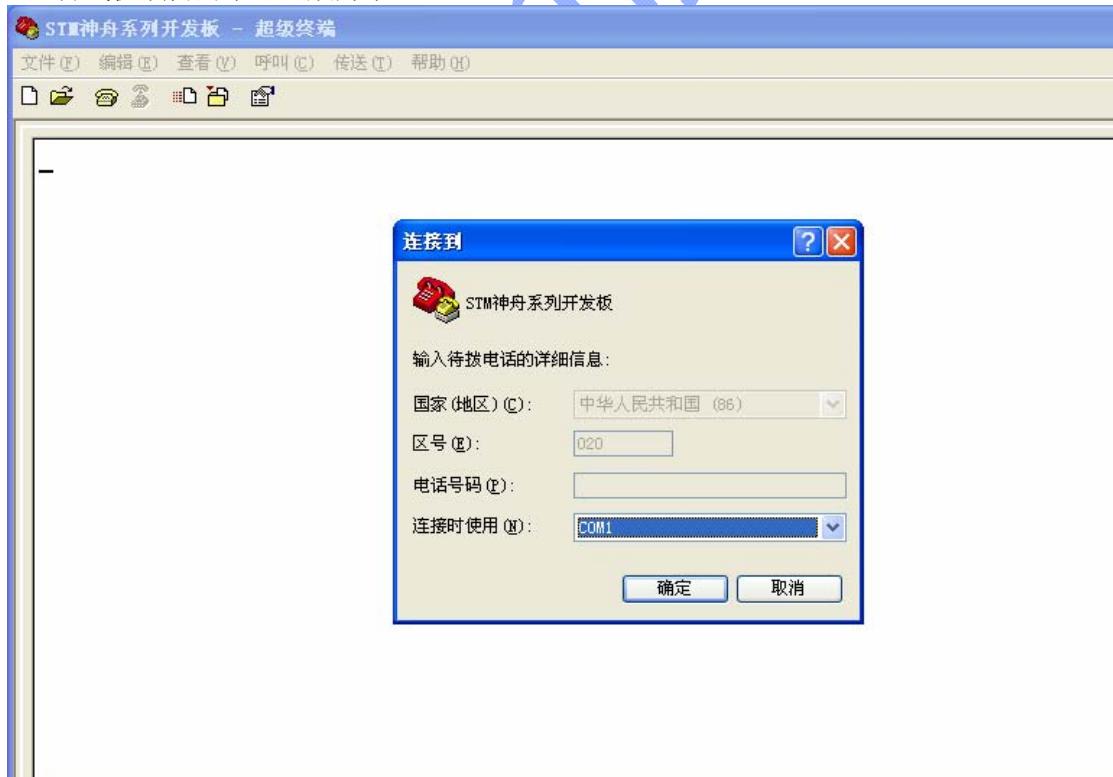
- 1) 打开开始菜单->程序->附件->通讯->超级终端



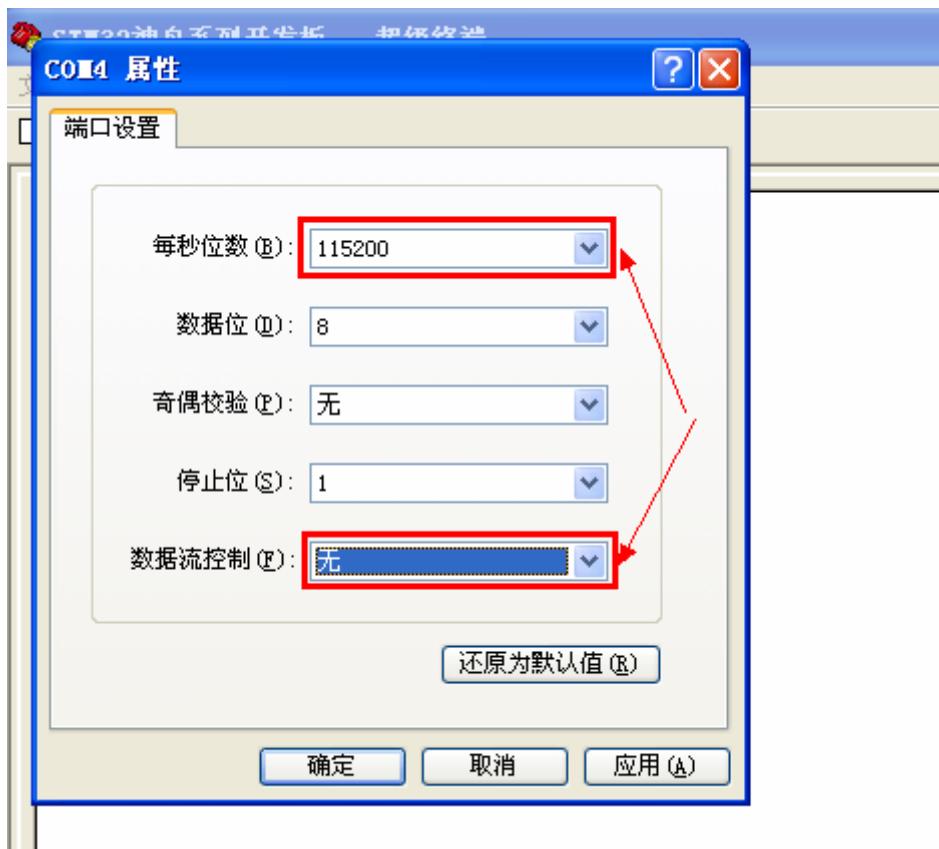
- 2) 输入“STM32 神舟系列开发板”



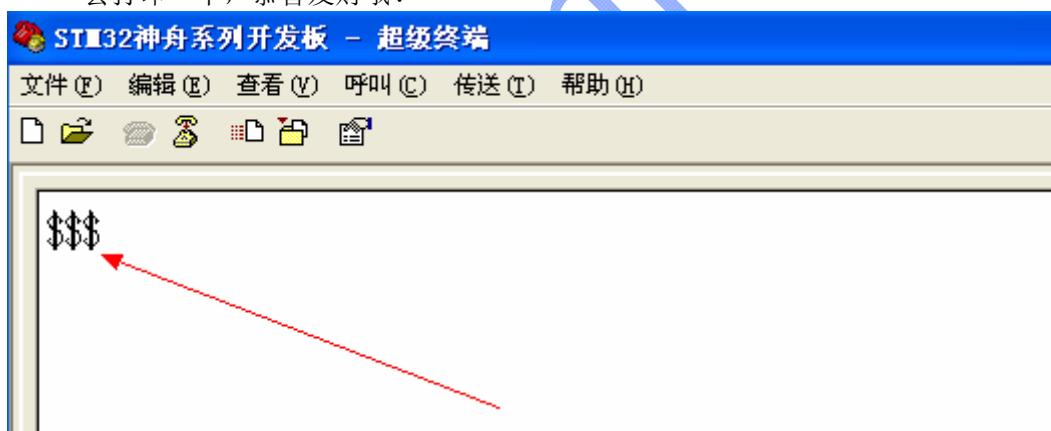
- 3) 紧接着，将神舟 III 号开发板上的串口 1 和电脑的串口用串口线连起来。(我这里用的是台式机，台式机预留的串口一般为串口 1)



- 4) 波特率例程代码中设置的是 115200，数据流控制是无，选择完毕，点确定按钮



- 5) 最后把例程程序下载到开发板里，然后按一下开发板复位或者重新上电，就会打印出“\$”的字符，一会打印一个，恭喜发财哦！



3. 关键代码：

```
int main(void) //main 是程序入口
{
    RCC_init();      //时钟频率的配置
    LED_init();      //LED 初始化配置
    uart_init();      //串口接口初始化，这个部分是按 STM32 芯片手册的要求来做的，比较枯燥，  
细节感兴趣的朋友可以去研究下
    while (1)
    {
        USART1->DR = 0x24; // 打印符号$，0x24 是 ASCII 码
        LEDON;           //点亮 LED 灯
```

```
        Delay(0xFFFF); // 延时
        LEDOFF;           // 熄灭 LED 灯
        Delay(0xFFFF); // 延时
    }
}

void uart_init()
{
    float USARTDIV;

    /* 因为 32 位的 USART_BRR 波特率设置寄存器只有低 16 位有效，所以这里我们定义 16 位寄存器就足够了 */
    u16 USARTDIV_zhengshu; // 这里相当于 u16, 无符号 16 位
    u16 USARTDIV_xiaoshu; // 这里相当于 u16, 无符号 16 位

    RCC->APB2ENR|=1<<2; // 使能 PORTA 口时钟
    RCC->APB2ENR|=1<<14; // 使能串口时钟

    GPIOA->CRH&=0XFFFF00F;
    GPIOA->CRH|=0X000008B0; // IO 状态设置

    USARTDIV = (float)(72*1000000)/(115200*16);
    USARTDIV_zhengshu = USARTDIV;

    USARTDIV_xiaoshu = (USARTDIV - USARTDIV_zhengshu)* 16;
    USARTDIV_zhengshu <<=4;
    USARTDIV_zhengshu += USARTDIV_xiaoshu;

    RCC->APB2RSTR|=1<<14; // 复位串口 1
    RCC->APB2RSTR&=~(1<<14); // 停止复位

    USART1->BRR = USARTDIV_zhengshu;
    USART1->CR1|=0X200C; // 1 位停止, 无校验位.
}
```

代码详细分析：

(1) RCC_init(); 这个函数是负责时钟频率的配置，这里默认配置为 72MHZ，前面章节有详细分析，这里简化，有疑问的可以翻看前面的章节细节。

(2) LED_init() 这个函数是初始化 LED 的配置，这个是附带的，如果串口无法正常打印，只要程序能运行，那 LED 灯就会进行闪烁。

(3) uart_init() 这个函数负责串口的初始化，这个部分是按 STM32 芯片手册的要求来做的，比较枯燥，细节感兴趣的朋友可以继续往下看，不感兴趣的可以跳过这一节，这个函数要把波特率初始化为 115200，下面我们仔细分析一下代码：

(3-1) 初始化一下波特率的值，一个是波特率的整数部分，一个是波特率的小数部分

```
u16 USARTDIV_zhengshu; // 这里相当于 u16, 无符号 16 位, 波特率的整数部分
u16 USARTDIV_xiaoshu; // 这里相当于 u16, 无符号 16 位, 波特率的小数部分
```

(3-2) 初始化串口的时钟，再初始化 PA9 和 PA10 两个管脚的 GPIO 端口 A 的时钟。从原理图可以看到，USART1 的 TX 和 RX 就是 PA9 和 PA10，要使用串口不仅仅要初始化 GPIO 端口 A 的时钟，还要初始化串口的时钟，这里是需要注意的，如果点灯程序或者只做为普通的 GPIO 管脚使用，就不需要初始化串口的时钟。

串口时钟使能。串口作为 STM32 的一个外设，其时钟由外设始终使能寄存器控制，这里我们使用的串口 1 是在 APB2ENR 寄存器的第 14 位。这里需要注意的一点是，除了串口 1 的时钟使能在

APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR。

```
RCC->APB2ENR|=1<<2; //使能 PORTA 口时钟
RCC->APB2ENR|=1<<14; //使能串口时钟
```

(3-3) 设置串口通信 RXD 和 TXD 的配置，一个管脚是接收就要设置为输入模式，一个管脚是输出，就设置成输出模式；寄存器 CRL 是设置 GPIO 的 0~7 位，CRH 是设置 GPIO 的 8~15 位，可以看到这里是设置 GPIOA 端口的 9 和 10 位，即 PA9 和 PA10 设置 PA9 为 TX 输出模式，复用功能推挽输出模式设置 PA10 为 RX 输入模式，模拟输入模式

```
GPIOA->CRH&=0xFFFFF00F;
GPIOA->CRH|=0X000008B0; //IO 状态设置
```

(3-4) 设置波特率，在 CPU 是 72MHZ 的频率下，设置波特率为 115200；STM32 中波特率是如何计算的，首先看下文档：

$$\text{Tx / Rx 波特率} = \frac{f_{PCLKx}}{(16 * USARTDIV)}$$

f_{PCLKx} ($x=1, 2$) 是给外设的时钟 (PCLK1 用于串口 2、3、4、5, PCLK2 用于串口 1)，USARTDIV 是一个无符号的定点数，它的值可以有串口的 USART_BRR 寄存器值得到。而我们更关心的是如何从 USARTDIV 的值得到 USART_BRR 的值，因为一般我们知道的是波特率，和 PCLKx 的时钟，要求的就是 USART_BRR 的值。

25.6.3 波特比率寄存器(USART_BRR)

注意：如果 TE 或 RE 被分别禁止，波特计数器停止计数

地址偏移：0x08

复位值：0x0000

| | | | | | | | | | | | | | | | |
|--------------------|--|----|----|----|----|----|----|----|----|----|----|-------------------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| DIV_Mantissa[11:0] | | | | | | | | | | | | DIV_Fraction[3:0] | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位31:16 | 保留位，硬件强制为0 | | | | | | | | | | | | | | |
| 位15:4 | DIV_Mantissa[11:0]：USARTDIV 的整数部分 这 12 位定义了 USART 分频器除法因子(USARTDIV)的整数部分。 | | | | | | | | | | | | | | |
| 位3:0 | DIV_Fraction[3:0]：USARTDIV 的小数部分 这 4 位定义了 USART 分频器除法因子(USARTDIV)的小数部分。 | | | | | | | | | | | | | | |

可以看到上图波特比率寄存器 USART_BRR 是低 16 位有效，高 16 位是闲置的，最低 4 位用来存放整数部分 DIV_Fraction，[15:4] 这 12 位用来存放小数部分 DIV_Mantissa。高 16 位未使用。这里波特率的计算通过如下公式计算：

假设我们的串口 1 要设置为 115200 的波特率，而 PCLK2 的时钟为 72M。这样，我们根据上面的公式有：

$$\text{波特率} = \frac{f_{PCLKx}}{(16 * USARTDIV)}$$

$$\text{USARTDIV} = \frac{f_{PCLKx}}{\text{波特率} * 16} = (72 * 1000000) / (115200 * 16) = 39.0625$$

我们查看《STM32F10XX 参考手册》中的第 525 页的一个表：

表154 设置波特率时的误差计算

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

1. CPU的时钟频率越低某一特定波特率的误差也越低 www.armjishu.com

2. 只有USART1使用PCLK2(最高72MHz)。其它USART使用PCLK1(最高36MHz)。

USARTDIV 的值被设置为 39.0625，也就是 USART_BRR 寄存器那么得到：

$$\text{DIV_Mantissa} = 39 = 0x27;$$

$$\text{DIV_Fraction} = 16 * 0.0625 = 1 = 0x1;$$

这样，我们就得到了 USART1->BRR 的值为 0x271。只要设置串口 1 的 BRR 寄存器值为 0x271 就可以得到 115200 的波特率。

USARTDIV = (float)(72*1000000)/(115200*16); //算出 USARTDIV 的值

USARTDIV_zhengshu = USARTDIV;

/* 因为波特率设置寄存器是 USARTDIV 整数在 0~3 位，小数在 4~15 位乘以 16 是因为小数点后面是 4 位，将它右移过来取成整数 */

USARTDIV_xiaoshu = (USARTDIV - USARTDIV_zhengshu)* 16;

USARTDIV_zhengshu <<=4;

USARTDIV_zhengshu += USARTDIV_xiaoshu;

(3-5) 当 CPU 刚启动的时候一般都需要重新复位一下外设，确保该外设在正常供电稳定后，能够稳定的工作可以看到复位一下之后就可以了，然后停止复位，让其开始正常工作。

串口复位。当外设出现异常的时候可以通过复位寄存器里面的对应位设置，实现该外设的复位，然后重新配置这个外设达到让其重新工作的目的。一般在系统刚开始配置外设的时候，都会先执行复位该外设的操作。串口 1 的复位是通过配置 APB2RSTR 寄存器的第 14 位来实现的。APB2RSTR 寄存器的各位描述如下：



从上图可知串口 1 的复位设置位在 APB2RSTR 的第 14 位。通过向该位写 1 复位串口 1，写 0 结束复位。其他串口的复位位在 APB1RSTR 里面。

RCC->APB2RSTR|=1<<14; //复位串口 1

RCC->APB2RSTR&=~(1<<14); //停止复位

(3-6) 把 115200 的波特率设置到 USART1->BRR 寄存器中，并且设置一下串口控制的寄存器 USART_CR1。STM32 的每个串口都有 3 个控制寄存器 USART_CR1~3，串口的很多配置都是通过这 3

个寄存器来设置的。这里我们只要用到 USART_CR1 就可以实现我们的功能了，这里主要设置该串口使能，正式启动这个串口功能该寄存器的描述在《STM32F1XX 参考手册》第 542 有更多的详细介绍：

25.6.4 控制寄存器 1(USART_CR1)

地址偏移: 0x0C

复位值: 0x0000

| | | | | | | | | | | | | | | | |
|---|----|----|------|-----|----|------|-------|------|---------|---------|----|----|-----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | UE | M | WAKE | PCE | PS | PEIE | TXEIE | TCIE | RXNE IE | IDLE IE | TE | RE | RWU | SBK | res |
| 保留 | | | | | | | | | | | | | | | |
| 位31:14 保留位, 硬件强制为0。 | | | | | | | | | | | | | | | |
| 位13 UE: USART使能 (USART enable) 当该位被清零, 在当前字节传输完成后USART的分频器和输出停止工作, 以减少功耗。该位由软件设置和清零。 0: USART分频器和输出被禁止; 1: USART模块使能。 | | | | | | | | | | | | | | | |
| 位12 M: 字长 (Word length) 该位定义了数据字的长度, 由软件对其设置和清零 0: 一个起始位, 8个数据位, n个停止位; 1: 一个起始位, 9个数据位, n个停止位。 注意: 在数据传输过程中(发送或者接收时), 不能修改这个位。 | | | | | | | | | | | | | | | |
| 位11 WAKE: 唤醒的方法 (Wake-up method) 这位决定了把USART唤醒的方法, 由软件对该位设置和清零。 0: 被空闲总线唤醒; 1: 被地址标记唤醒。 | | | | | | | | | | | | | | | |

USART1->BRR = USARTDIV zhengshu;
USART1->CR1|=0X200C; //1 位停止,无校验位.

(3-7) 数据发送与接收。STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能，寄存器描述如下：

24.6.2 数据寄存器(USART_DR)

地址偏移: 0x04

复位值: 不确定

| | | | | | | | | | | | | | | | |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 DR[8:0] | | | | | | | | | | | | | | | |
| 位31:9 保留位, 硬件强制为0 | | | | | | | | | | | | | | | |
| 位8:0 DR[8:0]: 数据值 包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)。该寄存器兼具读和写的功能。TDR寄存器提供了内部总线和输出移位寄存器之间的并行接口(参见图236)。RDR寄存器提供了输入移位寄存器和内部总线之间的并行接口。 当使能校验位(USART_CR1中PCE位被置位)进行发送时，写到MSB的值(根据数据的长度不同，MSB是第7位或者第8位)会被后来的校验位取代。 当使能校验位进行接收时，读到的MSB位是接收到的校验位。 | | | | | | | | | | | | | | | |

DR[8:0]为串口数据，可以看出，虽然是一个32位寄存器，但是只用了低9位(DR[8:0])，其他都是保留。

代码 USART1->DR = 0x24 是被用来打印符号 \$，0x24 是 ASCII 码请看下图，通过把这个 0x24 输入送给 USART_DR 寄存器后，就可以打印出 \$ 字符。

| 字符 | 十进制 | 十六进制 | 八进制 | |
|-------|-----|------|-----|-------|
| space | 32 | 20 | 40 | 空格 |
| ! | 33 | 21 | 41 | 无相关信息 |
| . | 34 | 22 | 42 | 无相关信息 |
| # | 35 | 23 | 43 | 无相关信息 |
| \$ | 36 | 24 | 44 | 无相关信息 |
| % | 37 | 25 | 45 | 无相关信息 |
| & | 38 | 26 | 46 | 无相关信息 |
| ' | 39 | 27 | 47 | 无相关信息 |

(4) 进入 while(1) 死循环，不停的让 LED 灯亮和灭，然后每次 LED 亮灭一次，就打印一个 \$ 字符，中间有一些延时，具体代码很简单。

5.4.9 例程02 单串口打印 www.armjishu.com 字符-初级

1. 例程简介：

例程同上个一样，只是打印出 www.armjishu.com 这么多字符来，增加了一些 ASCII 码。

2. 调试说明：

其他都与上个例程一样，下载进去后，打印出来的是如下图所示



3. 关键代码：

```
int main(void) //main 是程序入口
{
    RCC_init(); //时钟频率的配置
    LED_init(); //LED 初始化配置
    uart_init(); //串口接口初始化，这个部分是按 STM32 芯片手册的要求来做的，比较枯燥，细节
感兴趣的朋友可以去研究下
    while (1)
    {
        /* 通过查 ASCII 码表打印 “www.armjishu.com” 的 LOGO 每打印一个字符，都需要延时一下
         * 如果不加延时程序，就无法正确打印出，因为串口的缓冲数据需要一点时间才送出去，所以需
要等待一下 */
        USART1->DR = 0x77; //w
        Delay(0xFFFF); // 延时

        USART1->DR = 0x77; //w
        Delay(0xFFFF); // 延时
```

```
USART1->DR = 0x77; //w
Delay(0xFFFF);      // 延时

USART1->DR = 0x2e; //.
Delay(0xFFFF);      // 延时

USART1->DR = 0x61; //a
Delay(0xFFFF);      // 延时

USART1->DR = 0x72; //r
Delay(0xFFFF);      // 延时

USART1->DR = 0x6d; //m
Delay(0xFFFF);      // 延时

USART1->DR = 0x6a; //j
Delay(0xFFFF);      // 延时

USART1->DR = 0x69; //i
Delay(0xFFFF);      // 延时

USART1->DR = 0x73; //s
Delay(0xFFFF);      // 延时

USART1->DR = 0x68; //h
Delay(0xFFFF);      // 延时

USART1->DR = 0x75; //u
Delay(0xFFFF);      // 延时

USART1->DR = 0x2e; //.
Delay(0xFFFF);      // 延时

USART1->DR = 0x63; //c
Delay(0xFFFF);      // 延时

USART1->DR = 0x6f; //o
Delay(0xFFFF);      // 延时

USART1->DR = 0x6d; // m
Delay(0xFFFF);      // 延时
USART1->DR = 0x20; // 空格

LEDON;              //点亮 LED 灯
Delay(0xFFFFFFFF); // 延时
LEDOFF;             // 熄灭 LED 灯
Delay(0xFFFFFFFF); // 延时
}
```

代码分析：

可以看到，其实就是逐个打印出www.armjishu.com的ASCII码而已。

5.4.10 例程03 单串口打印www.armjishu.com字符-中级

1. 例程简介：

例程同上个一样，只是打印出www.armjishu.com这么多字符来，增加了一些ASCII码，主要体现代码撰写表达有区别。

2. 调试说明：

其他都与上个例程一样，下载进去后，打印出来的是如下图所示



3. 关键代码：

```
int main(void) //main 是程序入口
{
    RCC_init();      //时钟频率的配置
    LED_init();      //LED 初始化配置
    uart_init();
    while(1)
    {
        USART1_Printf(" www.armjishu.com ");
        LEDON;          //点亮 LED 灯
        Delay(0xFFFFFFF); // 延时
        LEDOFF;          // 熄灭 LED 灯
        Delay(0xFFFFFFF); // 延时
    }
}
void USART1_Printf(char *pch)
{
    while(*pch != '\0')
    {
        USART_SendData(USART1,pch);
        pch++;
    }
}
void USART_SendData(USART_TypeDef* USARTx, char *Data)
{
    USARTx->DR = *Data;
    Delay(0xFFF);
}
```

代码分析：

(1) 可以看到，用 USART1_Printf(" www.armjishu.com ")这个函数逐个打印出www.armjishu.com的字符。

(2) USART1_Printf() 函数内部具体实现，就是把*pch指向这个www.armjishu.com其中一个单个字符，然后调用USART_SendData(USART1,pch)将这个字符往串口 1 发送输出，再用while(*pch != '\0')判断下一个字符是不是结束符，不是的话继续打印下一个字符。

(3) 最后调用 USART_SendData(USART_TypeDef* USARTx, char *Data)这个函数，将字符数据的 USARTx->DR = *Data;给到 USART1_DR 寄存器，这里的 USART_SendData()函数中的*Data 等同于

上个函数的 USART1_Printf() 函数中的*pch，而*pch 等同于 USART1_Printf() 函数中的一个字符。

关于 USART1_Printf(" www.armjishu.com ")，这个 www.armjishu.com 在 USART1_Printf() 函数中创造了一个内存空间，这个内存空间是在 main 函数开始调用 USART1_Printf() 函数时分配的，而传入到 USART1_Printf() 中的 pch 是为 www.armjishu.com 所分配的这个内存区域的一个指针的地址，然后继续把这个指针的地址传给 USART_SendData() 中的 Data 变量；换句话说，Data 等于 pch 等于 www.armjishu.com 字符串内存区域的首地址。

5.4.11 例程04 单串口打印www.armjishu.com字符-高级

1. 例程简介：

例程同上个一样，只是打印出 www.armjishu.com 这么多字符来，主要体现增加了一些串口传送校验。

2. 调试说明：

其他都与上个例程一样，下载进去后，打印出来的是如下图所示



3. 关键代码：

```
void USART1_Printf(char *pch)
{
    while(*pch != '\0')
    {
        USART_SendData(USART1, pch);
        while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
        USART_ClearFlag(USART1, USART_FLAG_TXE);
        pch++;
    }
}
void USART_SendData(USART_TypeDef* USARTx, char *Data)
{
    USARTx->DR = (*Data & (uint16_t)0x01FF); // 这里的*Data 就是一个字符
    while((USARTx->SR&0x40) == 0);
}
```

代码分析：

(1) USARTx->DR = (*Data & (uint16_t)0x01FF) 这里的*Data 就是一个字符，ASCII 码是 0~127，并且 DR 是 0~8 个 bit 有效，可以看手册的 USART_DR 寄存器的描述，所以这里实际上我们只需要取二进制的前 9 位即可。

(2) while((USARTx->SR&0x40) == 0), USART_SR 是状态寄存器

24.6.1 状态寄存器(USART_SR)

地址偏移: 0x00

复位值: 0x00C0

| | | | | | | | | | | | | | | | |
|-------|--|----|----|----|-----|-------|-------|----|------|------|-----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | CTS | LBD | TXE | TC | RXNE | IDLE | ORE | NE | FE | PE | |
| rc w0 | | | | | r | rc w0 | rc w0 | r | r | r | r | r | r | r | r |
| 位6 | TC: 发送完成 当包含有数据的一帧发送完成后，由硬件将该位置位。如果USART_CR1中的TCIE为1，则产生中断。由软件序列清除该位(先读USART_SR，然后写入USART_DR)。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。 | | | | | | | | | | | | | | |

判断 TC 是不是已经发送完成，当包含有数据的一帧发送完成后，由硬件将该位置位，之前的代码是通过一定的延时，现在是判断寄存器，判断寄存器可以使得数据在第一时间发出之后，就能够使得开始进行下面的代码，比人工去延时的效率显著提高，所以算是比上个例程性能上的一种优化。

(3) 当发送完一个数据后, while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET)
查看一下寄存器说明：

| | |
|----|--|
| 位7 | TXE: 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。对USART_DR的写操作，将该位清零。 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。 |
|----|--|

判断 USART_SR 寄存器当中的 TXE 是不是为 1，如果是为 1 才能结束这个 while 循环，当等于 1 的时候，数据已经被发送出去了，进入了移位寄存器。

(4) USART_ClearFlag(USART1, USART_FLAG_TXE)这句代码是我们查看《STM32F1xx 参考手册》获得 540 页 25.6.1 节 状态寄存器 (USART_SR) 章节，这里说的就是我们程序代码里的 USART_FLAG_TXE 这个标志位，等数据发送完毕后，再进行标志清空，方便进行下一次传输的时候有效使用。

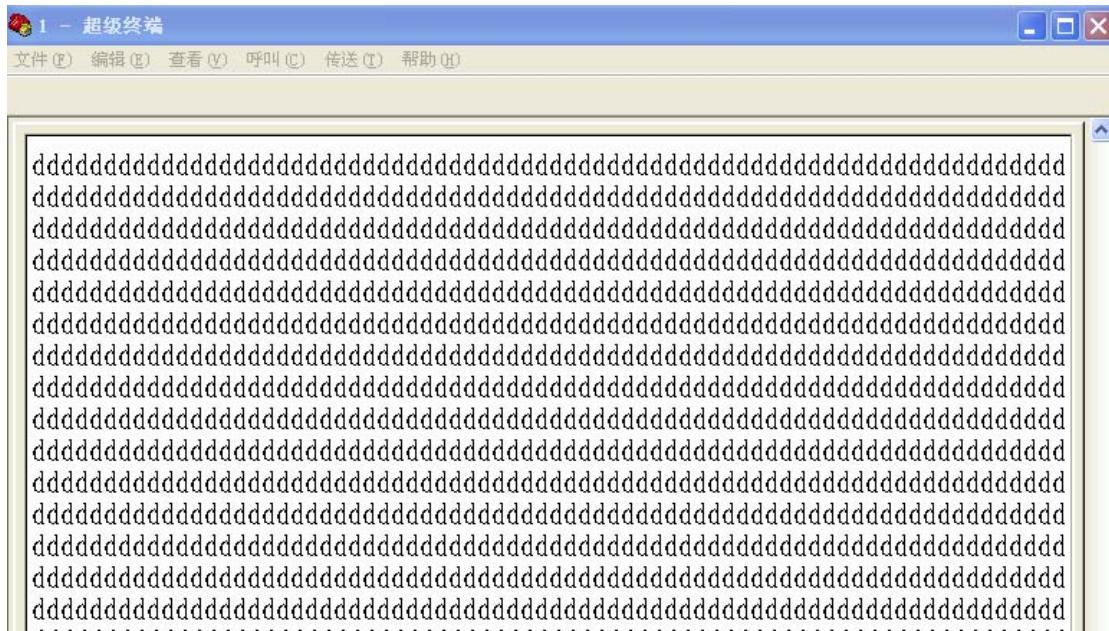
5.4.12 例程05 USART-COM1串口接收与发送实验-初级版

1. 例程简介：

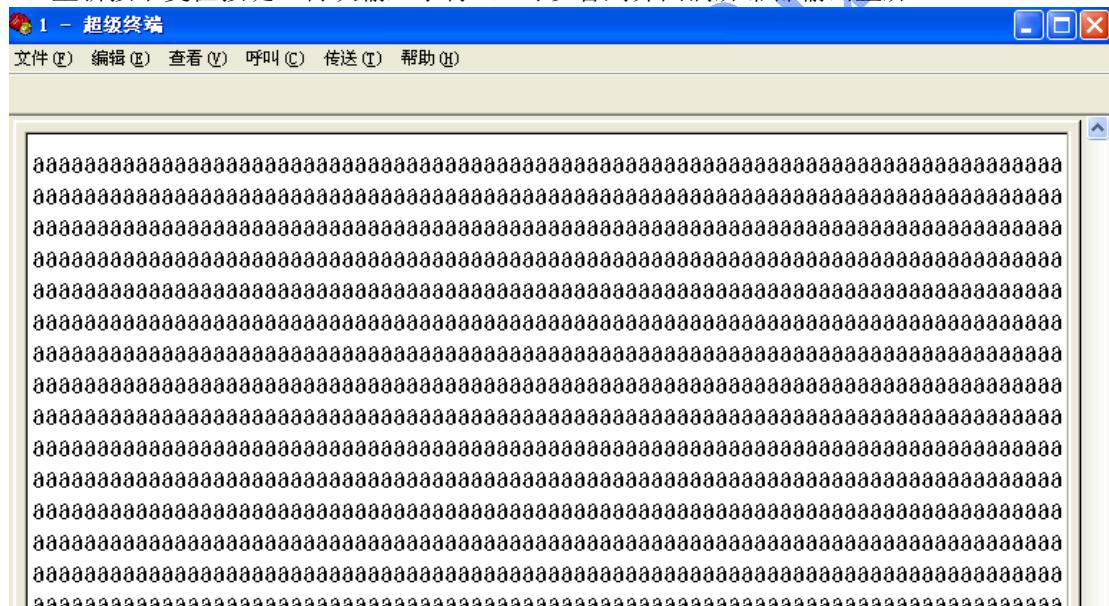
例程主要实现在键盘上敲一个字符，输入这个字符到串口中，然后再通过超级终端打印出来。

2. 调试说明：

1) 下载完程序后，打开超级终端，敲入字符 d，可以看到整个超级终端不停的输出 d 这个字符，表示输入成功。



2) 重新按下复位按键，再次输入字符 a，可以看到界面满屏幕都输出整屏 a



4. 关键代码:

```
int main(void)
{
    uint8_t inputstr[CMD_STRING_SIZE];
    RCC_init();
    uart_init();
    GetInputString(inputstr);
}

void GetInputString (uint8_t * buffP)
{
    uint8_t c = 0;
    do
```

```

{
    c = (uint8_t)USART1->DR;
    USART_SendData(USART1, c);
}
while (1);
}

void USART_SendData(USART_TypeDef* USARTx, uint8_t Data)
{
    USARTx->DR = (Data & (uint16_t)0x01FF);
}

```

代码分析：

- (1) uint8_t inputstr[CMD_STRING_SIZE] 代码声明一个数组，可以看到程序中变量 CMD_STRING_SIZE = 128，就是表示这个数组有 128 个成员，每个成员都是 uint8_t 类型的。
- (2) 接下来就是时钟初始化函数 RCC_init() 和串口初始化函数 uart_init()，之前都有详细介绍，这里就不做具体分析了。
- (3) GetInputString(inputstr) 这句代码表示取 inputstr[] 数组的初地址传入到 GetInputString() 这个函数中，知道了 inputstr[] 数组的首地址后，就可以在函数里操作和修改这个数组的内容。这里主要是熟悉一下指针和数组的一些基础概念，前面章节已经有详细分析。
- (4) 可以看到数据寄存器 USART_DR 包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读写的功能。

24.6.2 数据寄存器(USART_DR)

地址偏移: 0x04

复位值: 不确定

| | | | | | | | | | | | | | | | | | | | | | |
|-------|----|--|----|----|----|----|----|----|----|---------|----|----|----|----|----|--|--|--|--|--|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | | | | | | |
| 保留 | | | | | | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | |
| 保留 | | | | | | | | | | DR[8:0] | | | | | | | | | | | |
| 位31:9 | | 保留位，硬件强制为0 | | | | | | | | | | | | | | | | | | | |
| 位8:0 | | DR[8:0]: 数据值 包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器提供了内部总线和输出移位寄存器之间的并行接口(参见图238)。RDR寄存器提供了输入移位寄存器和内部总线之间的并行接口。 当使能校验位(USART_CR1中PCE位被置位)进行发送时，写到MSB的值(根据数据的长度不同，MSB是第7位或者第8位)会被后来的校验位取代。 当使能校验位进行接收时，读到的MSB位是接收到的校验位。 | | | | | | | | | | | | | | | | | | | |

在 GetInputString() 函数中，用 c = (uint8_t)USART1->DR 这句代码来读取输入到超级终端的键盘字符，如果有输入，那么 c 就会得到输入的初值

- (5) 最后 USART_SendData(USART1, c) 将 c 中的字符输出到超级终端上，细节请见代码。

5.4.13 例程06 USART-COM1串口接收与发送实验-中级版

1. 例程简介：

例程主要实现在键盘上敲一个字符，输入这个字符到串口中，然后再通过超级终端打印出来，例程与上个不同的是“www.armjishu.com 键盘输入 2013 年：”的字符串，而这个字符串输入出来有一个 人机交互界面 的感觉，其他都与上个例程相同

2. 调试说明：

1) 下载完程序后，打开超级终端，按下复位按键，可以看到“www.armjishu.com 键盘输入 2013 年:”的字符串，然后敲入字符 d，可以看到整个超级终端不停的输出 d 这个字符，表示输入成功。

2) 重新按下复位按键，可以再次看到“www.armjishu.com 键盘输入 2013 年:”的字符串，输入字符 a，可以看到界面满屏幕都输出整屏 a

3. 关键代码：

这里代码不做具体分析，之前有详细说

5.4.14 例程05 USART-COM1串口接收与发送实验-高级版

1. 例程简介：

例程主要实现在键盘上敲一个字符，输入这个字符到串口中，而之前的例程输出都是整屏不停的输出，这个例程加入了字符串部分代码的健壮，使得输入一个字符就是一个字符，绝对不会溢出或者数据丢失，或者重复输入。

2. 调试说明：

下载完程序后，打开超级终端，按下复位按键，可以看到“www.armjishu.com 键盘输入 2013 年:”的字符串，每敲一个字符，就会输出一个字符，不会有多余的字符出现。

3. 关键代码：

```
uint32_t SerialKeyPressed(uint8_t *key)
{
    if (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET)
    {
        *key = (uint8_t)USART1->DR;
        return 1;
    }
    else
    {
        return 0;
    }
}

uint8_t GetKey(void)
{
    uint8_t key = 0;
    while (1)
    {
        if (SerialKeyPressed((uint8_t*)&key)) break;
    }
    return key;
}

void GetInputString (uint8_t * buffP)
{
    uint32_t bytes_read = 0;
    uint8_t c = 0;
```

```

do
{
    c = GetKey();
    if (c == 'r')
        break;
    if (c == 'b') /* Backspace */
    {
        if (bytes_read > 0)
        {
            USART1_Printf("\b \b");
            bytes_read--;
        }
        continue;
    }
    if (bytes_read >= (CMD_STRING_SIZE))
    {
        USART1_Printf("Command string size overflow\r\n");
        bytes_read = 0;
        continue;
    }
    if (c >= 0x20 && c <= 0x7E)
    {
        buffP[bytes_read++] = c;
        SerialPutChar(c);
    }
}
while (1);
USART1_Printf("\n\r");
buffP[bytes_read] = '\0';
}

```

代码分析：

(1) SerialKeyPressed(uint8_t *key)函数分析

24.6.1 状态寄存器(USART_SR)

地址偏移: 0x00

复位值: 0x00C0

| | | | | | | | | | | | | | | | |
|----|----|----|----|-----|-----|-----|----|------|------|-----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | CTS | LBD | TXE | TC | RXNE | IDLE | ORE | NE | FE | PE | | |

位5

RXNE: 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。对USART_DR的读操作可以将该位清零。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。

1)用语句 USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET 判断 USART_DR 寄存器中是否收到数据，如果收到了数据就可以读出。

2) *key = (uint8_t)USART1->DR; 第二步就是把数据取出来放到*key 中，*key 的此时的值就是键盘输入的字符的 ASCII 码的值。

- (2) GetKey(void)函数在函数里分配了 key 的内存空间，然后调用了 SerialKeyPressed() 函数；其实这 GetKey 和 SerialKeyPressed 这两个函数在这个例程里可以合并，但是这样的写法有个好处就是，获取键盘的输入不一定是在串口中，也有可能是 can 的数据，或者 485，或者是网口等其他渠道的输入，用 GetKey()这个标准函数，再在里面调用具体的子函数，这样的好处就是增加了代码的可移植性。
- (3) GetInputString() 分析获得一个输入之后，就开始一系列判断，这个输入是不是空格，是不是换行符，如果是，就进行相对应的处理，这里值得注意的是，越详细，越复杂就表示你的输入这部分的代码越健壮，各种异常输入情况都考虑进去了；最后经过一连串的判断之后，合格的输入就调用 SerialPutChar()函数，把用户输入的内容输出到超级终端上。

www.armjishu.com

第六章 STM32库函数架构剖析

本章通过简单介绍 STM32 库的各个文件及其关系，让读者建立 STM32 库的概念，看完后对库有

个总体印象即可，在后期实际开发时接触了具体的库时，再回头看看这一章，相信你对 STM32 库又会有一个更深刻的认识。

6.1 STM32库函数到底是什么

STM32 的库函数是 ST 公司已经封装好一个软件封装库，也就是很多基础的代码，在开发产品的时候只需要直接调用这个 ST 库函数的函数接口就可以完成一系列工作；例如，你原来要自己烧饭洗衣服，现在 ST 库函数就好像一个保姆，你只需要使唤一声就有饭吃，有干净的衣服了。

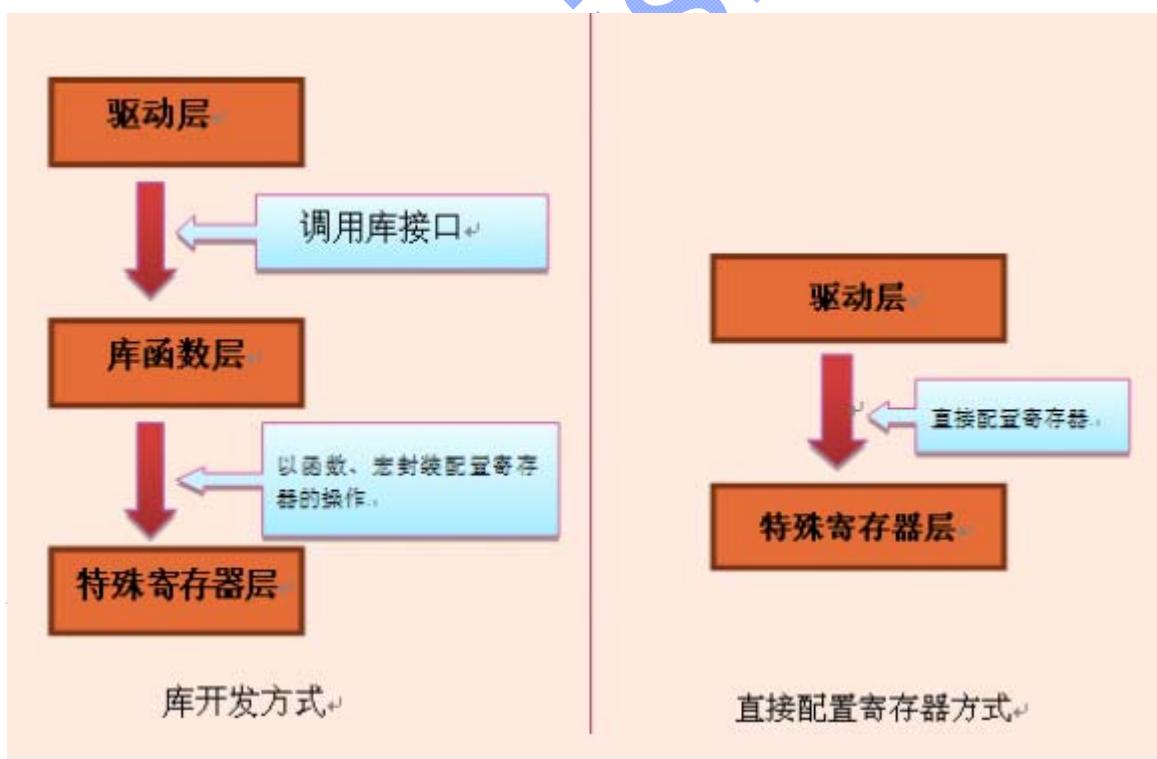
6.2 STM32库函数的好处

在 51 单片机的程序开发中，我们直接配置 51 单片机的寄存器，控制芯片的工作方式，如中断，定时器等。配置的时候，我们常常要查阅寄存器表，看用到哪些配置位，为了配置某功能，该置 1 还是置 0。这些都是很琐碎的、机械的工作，因为 51 单片机的软件相对来说比较简单，而且资源很有限，所以可以直接配置寄存器的方式来开发。

STM32 库是由 ST 公司针对 STM32 提供的函数接口，即 API (Application Program Interface)，开发者可调用这些函数接口来配置 STM32 的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速，易于阅读，维护成本低等优点。

当我们调用库的 API 的时候可以不用挖空心思去了解库底层的寄存器操作，就像当年我们学习 C 语言的时候，用 printf() 函数时只是学习它的使用格式，并没有去研究它的源码实现，如非必要，可以说是老死不相往来。

实际上，库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口。



6.3 千人项目如何分配工作

其实大多数项目产品研发都是用类似库函数的方式进行分解的，如果有 1000 个开发人员来负责开发板 windows 操作系统，那么怎么做？一定是一群人负责最底层的硬件级，寄存器的读写封装，包括显示器的点亮，图形刷写；然后另外一群人根据底层这群人提供的接口，同步做二次开发。

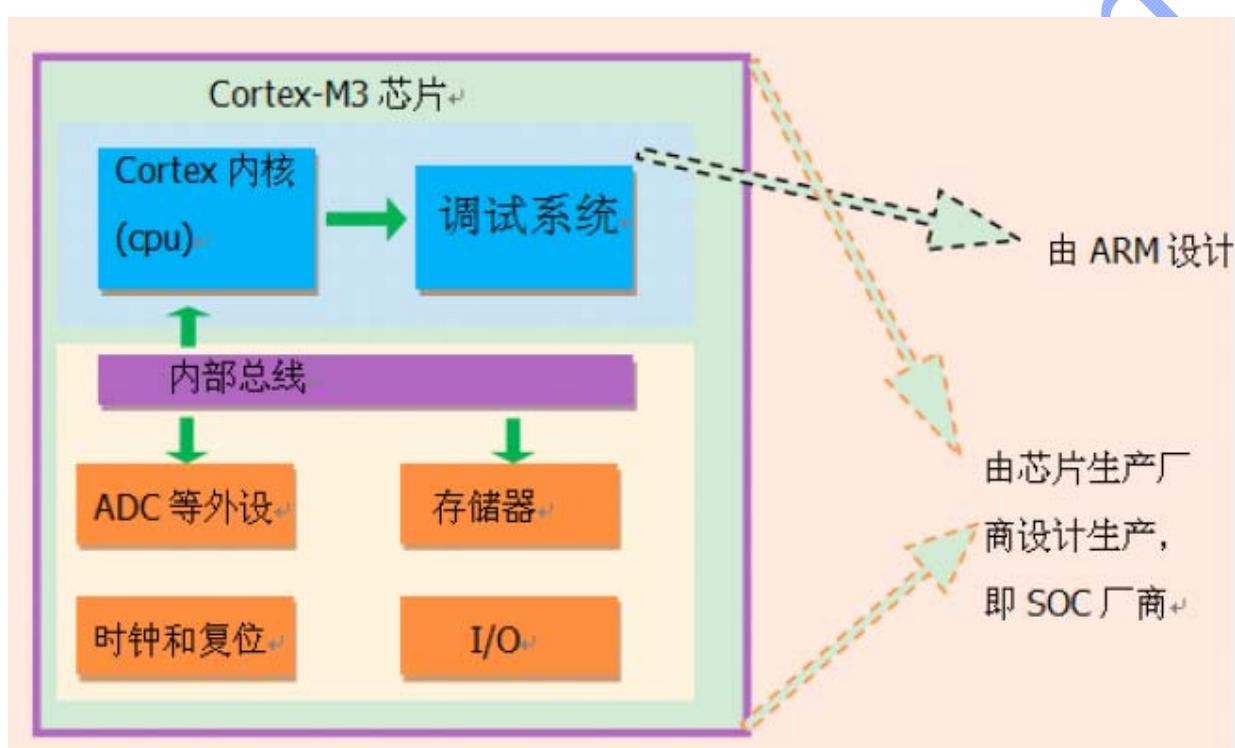
所以库函数的理念完全被广泛应用于各种实际的项目和产品中，因为这样才可以使得多人协同工作。嵌入式专业技术论坛（www.armjishu.com）出品 第 260 页，共 900 页

作，才能做更大的项目产品。

6.4 STM32库结构剖析

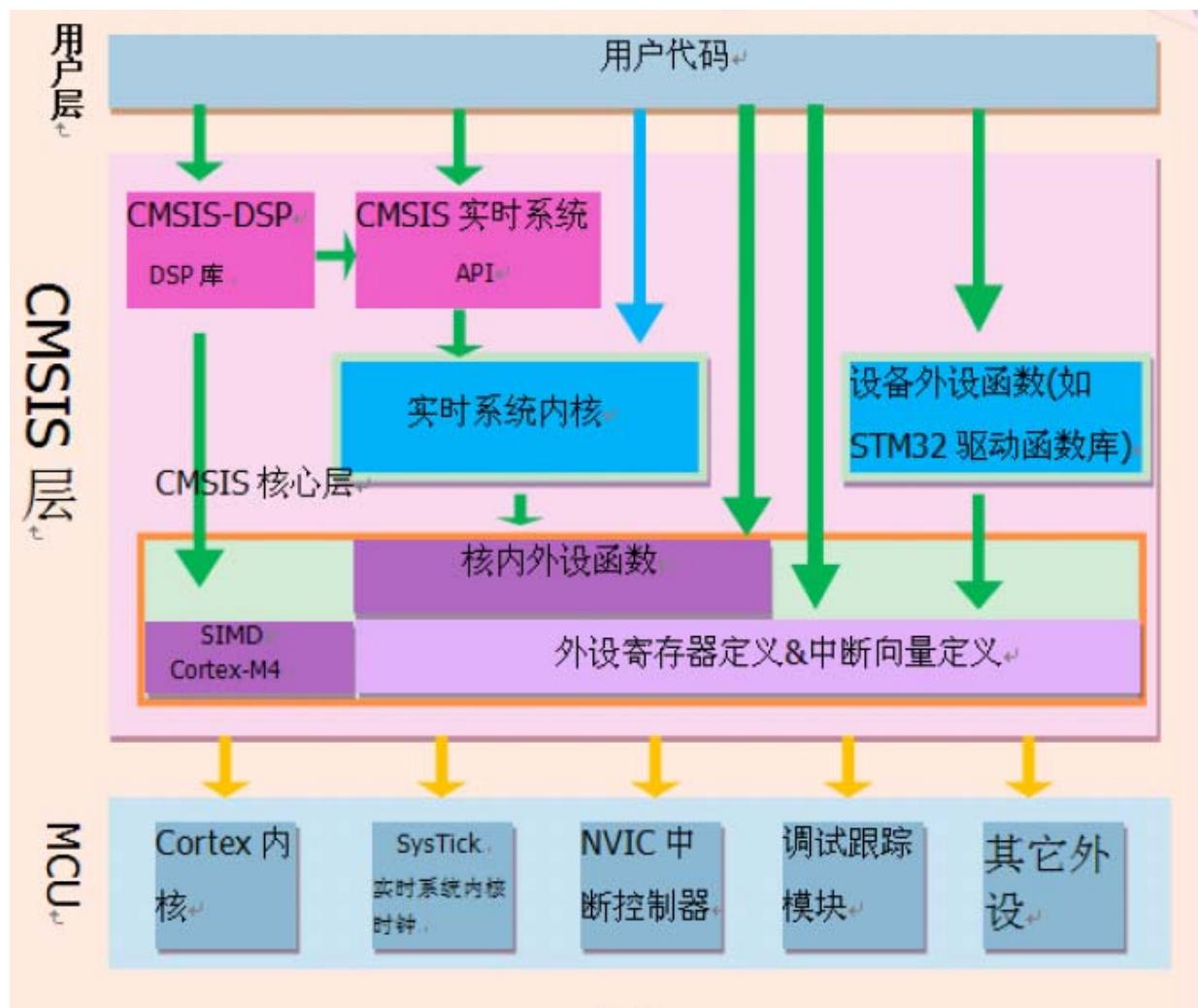
6.4.1 CMSIS标准

我们知道由 ST 公司生产的 STM32 采用的是 Cortex-M3 内核，内核是整个微控制器的 CPU。该内核是 ARM 公司设计的一个处理器体系架构。ARM 公司并不生产芯片，而是出售其芯片技术授权。ST 公司或其它芯片生产厂商如 TI，负责设计的是在内核之外的部件，被称为核外外设或片上外设、设备外设。如芯片内部的模数转换外设 ADC、串口 UART、定时器 TIM 等。内核与外设，如同 PC 上的 CPU 与主板、内存、显卡、硬盘的关系：



因为基于 Cortex 的某系列芯片采用的内核都是相同的，区别主要为核外的片上外设的差异，这些差异却导致软件在同内核，不同外设的芯片上移植困难。为了解决不同的芯片厂商生产的 Cortex 微控制器软件的兼容性问题，ARM 与芯片厂商建立了 CMSIS 标准(Cortex MicroController Software Interface Standard)。

所谓 CMSIS 标准，实际是新建了一个软件抽象层。见下图：



CMSIS 标准中最主要的为 CMSIS 核心层，它包括了：

- 内核函数层：其中包含用于访问内核寄存器的名称、地址定义，主要由 ARM 公司提供
- 设备外设访问层：提供了片上的核外外设的地址和中断定义，主要由芯片生产商提供

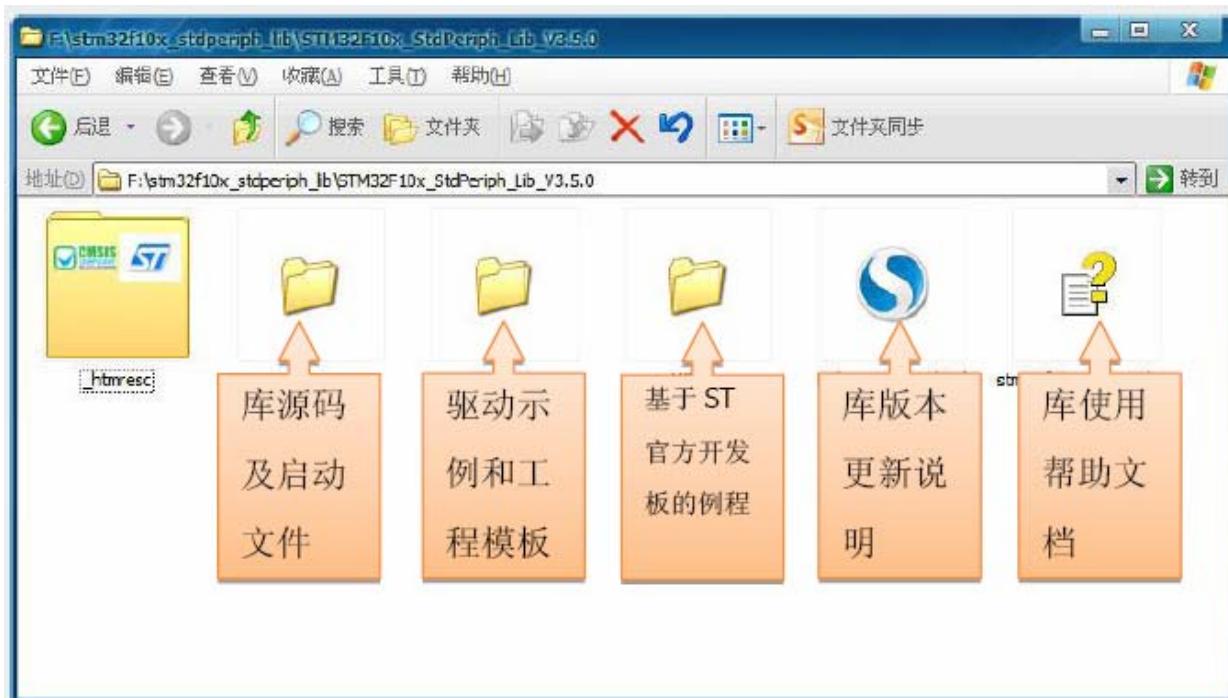
可见 CMSIS 层位于硬件层与操作系统或用户层之间，提供了与芯片生产商无关的硬件抽象层，可以为接口外设、实时操作系统提供简单的处理器软件接口，屏蔽了硬件差异，这对软件的移植是有极大的好处的。STM32 的库，就是按照 CMSIS 标准建立的。

6.4.2 库目录，文件简介

STM32 的 3.5 版库可以从官网获得，也可以直接从本书的附录光盘得到。本书主要采用最新版的 3.5 库文件，在高级篇的章节有部分代码是采用 3.0 的库开发的，因为 3.5 与 3.0 的库文件兼容性很好，对于旧版的代码我们仍然使用用 3.0 版的。

解压后进入库目录：stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0

各文件夹内容说明见图：



Libraries 文件夹下是驱动库的源代码及启动文件。

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，是库帮助文档，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。不幸的是，这个帮助文档是英文的，这对很多英文不好的朋友来说是一个很大的障碍。但要告诉大家，英文仅仅是一种工具，绝对不能让它成为我们学习的障碍。其实这些英文还是很简单的，我们需要的是拿下它的勇气。

网上流传有一份中文版本的库帮助文档，但那个是 2.x 版本的，但 3.x 以上版本的目录结构和库函数接口跟 2.x 版本的区别还是比较大的，这点大家要注意下。

在使用库开发时，我们需要把 libraries 目录下的库函数文件添加到工程中，并查阅库帮助文档来了解 ST 提供的库函数，这个文档说明了每一个库函数的使用方法。

进入 Libraries 文件夹看到，关于内核与外设的库文件分别存放在 CMSIS 和 STM32F10x_StdPeriph_Driver 文件夹中。

Libraries\CMSIS\CM3 文件夹下又分为 CoreSupport 和 DeviceSupport 文件夹。

6.4.3 关于core_cm3.c文件

在 CoreSupport 中的是位于 CMSIS 标准的核内设备函数层 的 M3 核通用的源文件 core_cm3.c 和头文件 core_cm3.h，它们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。这两个文件在其它公司的 M3 系列芯片也是相同的。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，有兴趣的朋友可以深究。

core_cm3.c 文件还有一些与编译器相关条件编译语句，用于屏蔽不同编译器的差异，我们在开发时不用管这部分，有兴趣可以了解一下。里面包含了一些跟编译器相关的信息，如：RealView Compiler (RVMDK)，ICC Compiler (IAR)，GNU Compiler。

```

1. /* define compiler specific symbols */
2. #if defined ( __CC_ARM )
3.   #define __ASM           __asm
4.   #define __INLINE         inline
5.
6. #elif defined ( __ICCARM__ )
7.   #define __ASM           __asm
8.   #define __INLINE         inline
9. #elif defined ( __GNUC__ )
10.  #define __ASM           __asm
11.  #define __INLINE         inline
12. #elif defined ( __TASKING__ )
13.  #define __ASM           __asm

```

使用 RVMDK 编
译器时的嵌入汇编
与内联函数的关键
字形式

使用 IAR 编译器时
的形式

较重要的是在 core_cm3.c 文件中包含了 stdint.h 这个头文件，这是一个 ANSI 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新类型定义，如：

```

1. /* exact-width signed integer types */
2. typedef signed     char int8_t;
3. typedef signed short int int16_t;
4. typedef signed       int int32_t;
5. typedef signed      int64 int64_t;
6.
7. /* exact-width unsigned integer types */
8. typedef unsigned    char uint8_t;
9. typedef unsigned short int uint16_t;
10. typedef unsigned      int uint32_t;
11. typedef unsigned     int64 uint64_t;

```

这些新类型定义屏蔽了在不同芯片平台时，出现的诸如 int 的大小是 16 位，还是 32 位的差异。所以在以后的程序中，都将使用新类型如 int8_t、int16_t,..,

在稍旧版的程序中还可能会出现如 u8、u16、u32 这样的类型，请尽量避免这样使用，在这里提出来是因为初学时如果碰到这样的旧类型让人一头雾水，而且在以新的库建立的工程中是无法追踪到 u8、u16、u32 这些的定义。

core_cm3.c 跟启动文件一样都是底层文件，都是由 ARM 公司提供的，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 芯片的移植工作就得以简化。

6.4.4 system_stm32f10x.c 文件

在 DeviceSupport 文件夹下的是启动文件、外设寄存器定义&中断向量定义层的一些文件，这是由 ST 公司提供的。



system_stm32f10x.c, 是由 ST 公司提供的, 遵守 CMSIS 标准。该文件的功能是设置系统时钟和总线时钟, M3 比 51 单片机复杂得多, 并不是说我们外部给一个 8M 的晶振, M3 整个系统就以 8M 为时钟协调整个处理器的工作。我们还要通过 M3 核的核内寄存器来对 8M 的时钟进行倍频, 分频, 或者使用芯片内部的时钟。所有的外设都与时钟的频率有关, 所以这个文件的时钟配置是很关键的。

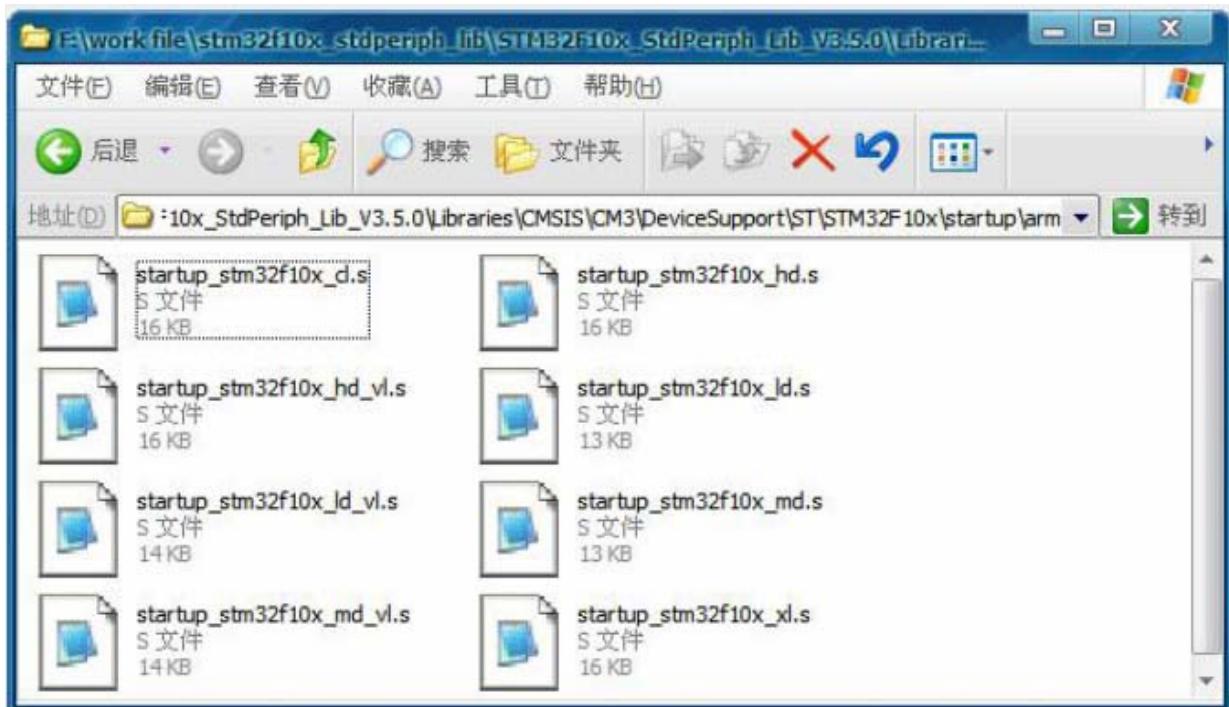
system_stm32f10x.c 在实现系统时钟的时候要用到 PLL (锁相环), 这就需要操作寄存器, 寄存器都是以存储器映射的方式来访问的, 所以该文件中包含了 stm32f10x.h 这个头文件。

6.4.5 stm32f10x.c 文件

stm32f10x..h 这个文件非常重要, 是一个非常底层的文件。所有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。

6.4.6 启动文件

Libraries\CMSIS\Core\CM3\startup\arm 文件夹下是由汇编编写的系统启动文件, 不同的文件对应不同的芯片型号, 在使用时要注意。



文件名的英文缩写的意义如下：

- cl: 互联型产品, stm32f105/103 系列
- vl: 超值型产品, stm32f100 系列
- xl: 超高密度(容量)产品, stm32f101/103 系列
- ld: 低密度产品, FLASH 小于 64K
- md: 中等密度产品, FLASH=64 or 128
- hd: 高密度产品, FLASH 大于 128

神舟 III 号中用的芯片是 STM32F103ZET, 512KROM, 所以启动文件要选择 startup_stm32f10x_hd.s。

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。在我们编写的 C 语言代码运行之前, 需要由汇编为 C 语言的运行建立一个合适的环境, 接下来才能运行我们的程序。所以我们也要把启动文件添加进我们的工程中去; 所以, 总的来说, 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. 设置堆、栈的大小;
4. 设置异常向量表的入口地址;
5. 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);
6. 设置 C 库的分支入口_main(最终用来调用 main 函数);
7. 在 3.5 版的启动文件还调用了在 system_stm32f10x.c 文件中的 SystemIni() 函数配置系统时钟, 在旧版本的工程中要用户进入 main 函数自己调用 SystemIni() 函数。

6.4.7 STM32F10x_StdPeriph_Driver 文件夹

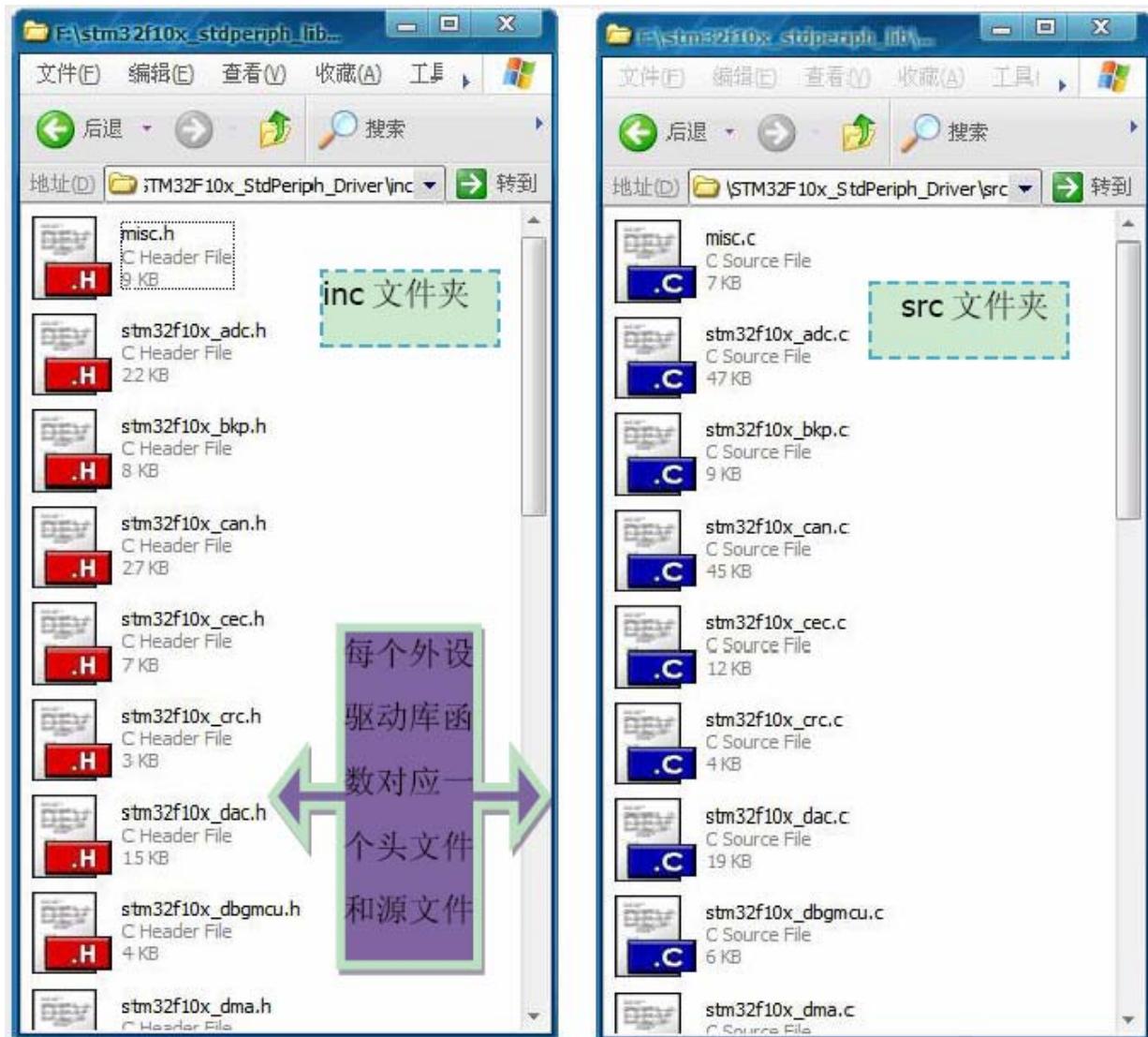
Libraries\STM32F10x_StdPeriph_Driver 文件夹下有 inc (include 的缩写) 跟 src (source 的简写) 这两个文件夹, 这都属于 CMSIS 的设备外设函数 部分。src 里面是每个设备外设的驱动程序, 这些外设是芯片制造商在 Cortex-M3 核外加进去的。

进入 libraries 目录下的 STM32F10x_StdPeriph_Driver 文件夹如下图:



在 src 和 inc 文件夹里的就是 ST 公司针对每个 STM32 外设而编写的库函数文件, 每个外设对应一个 .c 和 .h 后缀的文件。我们把这类外设文件统称为: stm32f10x_ppp.c 或 stm32f10x_ppp.h 文件, PPP 表示外设名称。

如针对模数转换(ADC)外设, 在 src 文件夹下有一个 stm32f10x_adc.c 源文件, 在 inc 文件夹下有一个 stm32f10x_adc.h 头文件, 若我们开发的工程中用到了 STM32 内部的 ADC, 则至少要把这两个文件包含到工程里:



这两个文件夹中，还有一个很特别的 misc.c 文件，这个文件提供了外设对内核中的 NVIC(中断向量控制器)的访问函数，在配置中断时，我们必须把这个文件添加到工程中。

6.4.8 stm32f10x_it.c、stm32f10x_conf.h文件

在库目录的 Project\STM32F10x_StdPeriph_Template 目录下，存放了官方的一个库工程模板，我们在用库建立一个完整的工程时，还需要添加这个目录下的 stm32f10x_it.c、stm32f10x_it.h、stm32f10x_conf.h 这三个文件。

stm32f10x_it.c，是专门用来编写中断服务函数的，在我们修改前，这个文件已经定义了一些系统异常的接口，其它普通中断服务函数由我们自己添加。但是我们怎么知道这些中断服务函数的接口如何写呢？是不是可以自定义呢？答案当然不是的，这些都有可以在汇编启动文件中找到，具体的大家自个看库的启动文件的源码去吧。

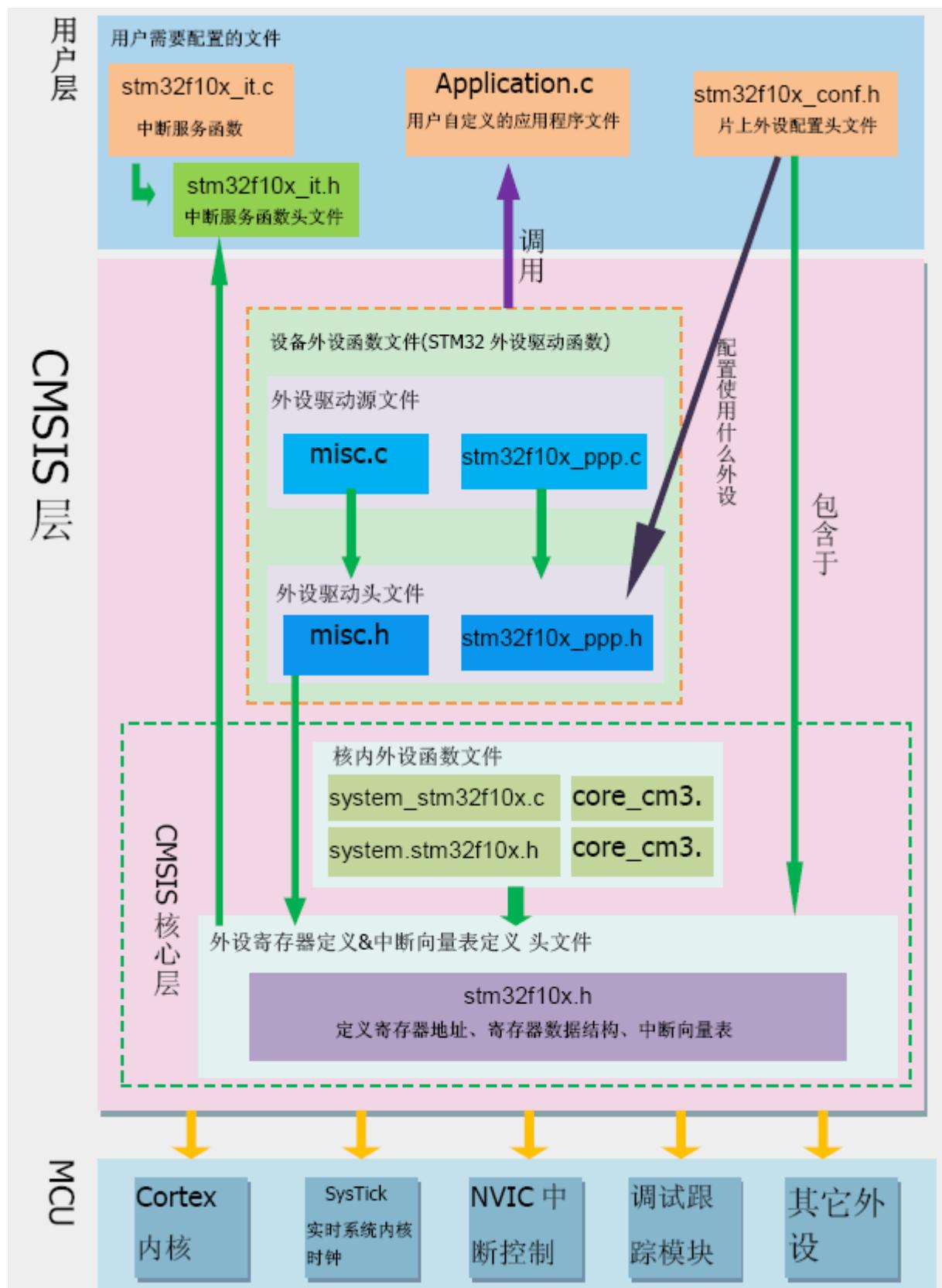
stm32f10x_conf.h，这个文件被包含进 stm32f10x.h 文件。是用来配置使用了什么外设的头文件，用这个头文件我们可以很方便地增加或删除上面 drIer 目录下的外设驱动函数库。如下面的代码配置表示使用了 gpio、rcc、spi、uart 的外设库函数，其它的注释掉的部分，表示没有用到。

```
1. /* Includes -----
   -----*/
2. /* Uncomment/Comment the line below to enable/disable peripheral header file inclusion */
3. //:#include "stm32f10x_adc.h"
4. //:#include "stm32f10x_bkp.h"
5. //:#include "stm32f10x_can.h"
6. //:#include "stm32f10x_cec.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_fsmc.h"
14. #include "stm32f10x_gpio.h"
15. //:#include "stm32f10x_i2c.h"
16. //:#include "stm32f10x_iwdg.h"
17. //:#include "stm32f10x_pwr.h"
18. #include "stm32f10x_rcc.h"
19. //:#include "stm32f10x_rtc.h"
20. //:#include "stm32f10x_sdio.h"
21. #include "stm32f10x_spi.h"
22. //:#include "stm32f10x_tim.h"
23. #include "stm32f10x_usart.h"
24. //:#include "stm32f10x_wwdg.h"
25. //:#include "misc.h" /* High level functions for NVIC and SysTick (add-on to CMSIS functions) */
```

stm32f10x_conf.h 这个文件还可配置是否使用“断言”编译选项，在开发时使用断言可由编译器检查库函数传入的参数是否正确，软件编写成功后，去掉“断言”编译选项可使程序全速运行。可通过定义 USE_FULL_ASSERT 或取消定义来配置是否使用断言。

6.4.9 库各文件间的关系

前面向大家简单介绍了各个库文件的作用，库文件是直接包含进工程即可，丝毫不用修改，而有的文件就要我们在使用的时候根据具体的需要进行配置。接下来从整体上把握一下各个文件在库工程中的层次或关系，这些文件对应到 CMSIS 标准架构上：



上图描述了 STM32 库各文件之间的调用关系，这个图省略了 DSP 核(Cortex-M3 没有 DSP 核)和实时系统层部分的文件关系。在实际的使用库开发工程的过程中，我们把位于 CMSIS 层的文件包含进工程，丝毫不用修改，也不建议修改。

对于位于用户层的几个文件，就是我们在使用库的时候，针对不同的应用对库文件进行增删（用条件编译的方法增删）和改动的文件。

6.4.10 常用官方资料

1. 《STM32 数据手册.pdf》

这个文件相当于 STM32 的 datasheet，管脚定义，内部存储器架构分配，跟芯片硬件相关的定义都可以从这里找得到。

2. 《STM32 数据手册.pdf》

这个文件对 STM32 的寄存器描述，它把 STM32 的时钟、存储器架构、及各种外设都描述得清清楚楚。当我们对 STM32 的库函数的实现方式 感到困惑时，可查阅这个文件，以直接配置寄存器方式开发的话查阅这个文档的频率会更高。

3. 《Cortex-M3 权威指南》 宋岩译。

该手册详细讲解了 Cortex 内核的架构和特性，要深入了解 Cortex-M3 内核，这是首选，经典中的经典呀。

4. 《stm32f10x_stdperiph_lib_um.chm》

这个就是前面提到的库的帮助文档，在使用库函数时，我们最好通过查阅此文件来了解库函数原型，或库函数的调用 的方法。也可以直接阅读源码里面的函数的函数说明。

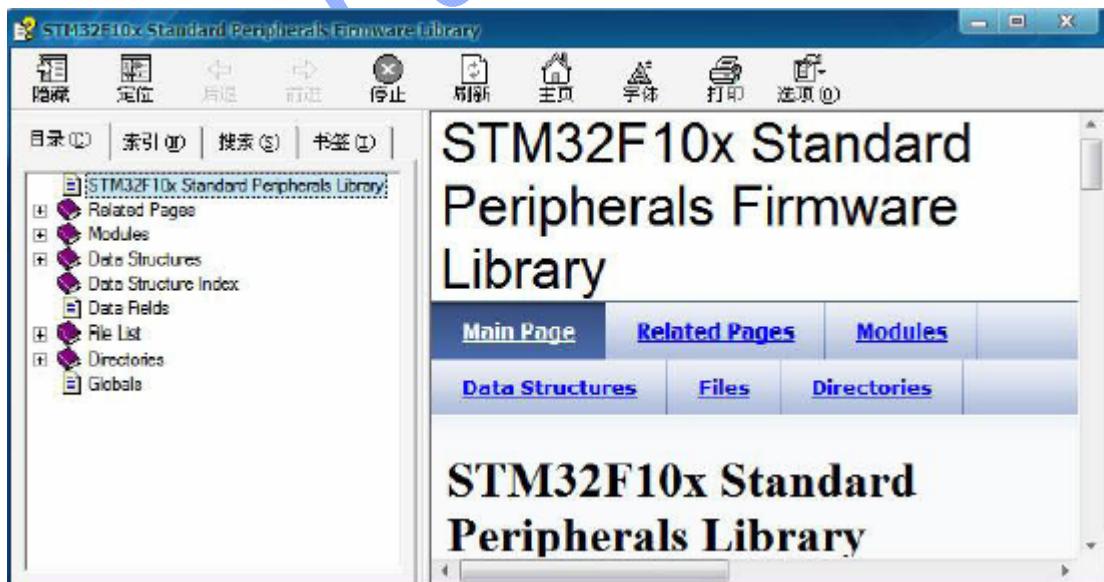
5. 其他文档

6.4.11 库函数帮助文档使用

所谓库函数，就是 STM32 的库文件中为我们编写好的函数接口，我们只要调用这些库函数，就可以对 STM32 进行配置，达到控制目的。我们可以不知道库函数是如何实现的，但我们调用函数必须要知道函数的功能、可传入的参数及其意义、和函数的返回值。

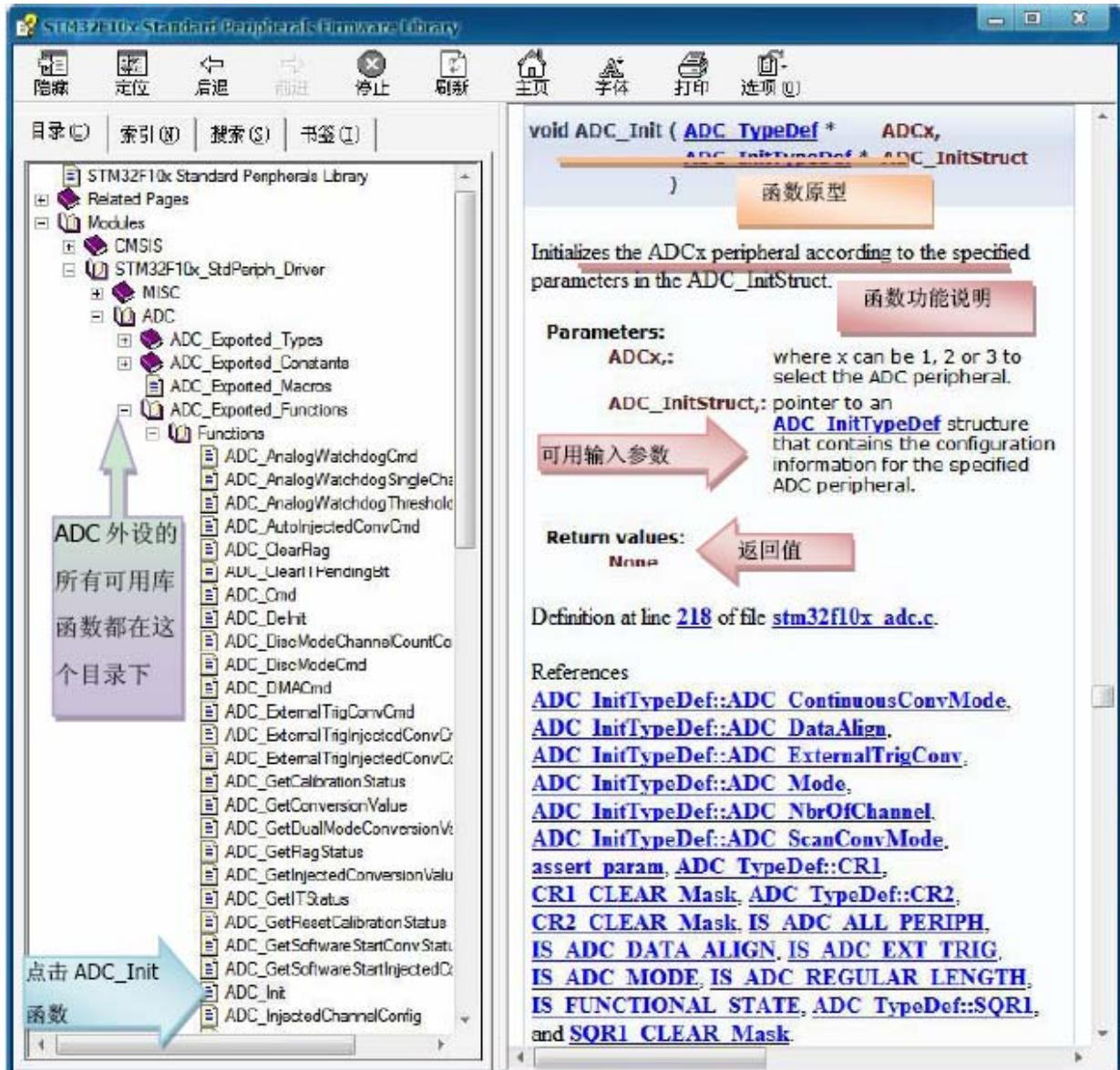
于是，有读者就问那么多函数我怎么记呀？神舟的回答是：会查就行了，哪个人记得了那么多。所以我们学会查阅库帮助文档 是很有必要的。

打开库帮助文档 stm32f10x_stdperiph_lib_um.chm 见下图：



层层打开文档的目录标签 Modules\STM32F10x_StdPeriph_Driver, 可看到

STM32F10x_StdPeriph_Driver 标签下有很多外设驱动文件的名字 MISC、ADC、BKP、CAN 等标签。我们试着查看 ADC 的初始化库函数 (ADC_Init) 看看，继续打开标签 \ADC\ADC_Exported_Functions\Functions\ADC_Init 见下图：



利用这个文档，我们即使没有去看它的具体代码，也知道要怎么利用它了。

如它的功能是：以 ADC_InitStruct 参数配置 ADC，进行初始化。原型为 void ADC_Init(ADC_TypeDef * ADCx , ADC_InitTypeDef * ADC_InitStruct)

其中输入的参数 ADCx 和 ADC_InitSturct 均为库文档中定义的 自定义数据类型，这两个传入参数均为结构体指针。初学时，我们并不知道如 ADC_TypeDef 这样的类型是什么意思，可以点击函数原型中带下划线的 ADC_TypeDef 就可以查看这是什么类型了。

就这样初步了解了一下库函数，读者就可以发现 STM32 的库是写得很优美的。每个函数和数据类型都符合见名知义 的原则，当然，这样的名称写起来特别长，而且对于我们来说要输入这么长的英文，很容易出错，所以在开发软件的时候，在用到库函数的地方，直接把库帮助文档 中函数名称复制粘贴到工程文件就可以了。

第七章 STM32神舟III号 实战篇

经过前两章的学习，我们对神舟III号开发板的硬件，以及其开发环境有了个比较深入的了解了，接下来就是通过实战，来真正开始STM32的开发。通过本章的学习，你将学会STM32的大部分外设的使用。本章将从浅入深向大家介绍如何一步步开发STM32，使你真正能独立用STM32开发自己的东西。本章将通过一系列的实例，向大家介绍STM32的强大功能，及开发实例。

神舟III号STM32开发板实验例程基于STM32F10x_StdPeriph_Lib_V3.5.0库编写，在RVMDK 4.22开发环境中调试运行通过。

7.1 LED流水灯实验

通过前面章节“初始 STM32 库”的内容，读者对库仅仅是有一个模糊的印象。本实验通过控制 IO 管脚的电平高低，使 LED 实现流水灯功能（流水灯也叫跑马灯）。我们对这个例程进行详细的分析，为读者扫清对使用库函数的困惑。**读者利用这个流水灯例程，真正领会了库开发的流程以及原理，再进行其它外设的开发就变得相当简单了。**

本章的内容非常的重要。学习后面的其它外设接口，除了本身接口的知识之外，也是对库开发流程，对库函数的理解不断深化的一个过程。

7.1.1 STM32的最简单外设GPIO分析

控制 LED 灯亮灭，是通过控制 STM32 芯片的 I/O 引脚电平的高低来实现。

芯片的引脚是硬件，芯片被生成出来，它的引脚数就固定了。一个芯片中，不同类型的引脚有不同的功能。对于 STM32 芯片来说，它的引脚大部分是，GPIO 引脚。GPIO 引脚被分为 GPIOA、GPIOB、GPIOC……不同的组，每组端口分为 0~15 个引脚，**共 16 个不同的引脚**。每个 I/O 引脚，可以被软件设置成各种**不同的功能**，如输入或输出，所以被称为 GPIO (General-purpose input/output)。对于不同型号的芯片，端口的组和引脚的数量不同，具体有多少个 GPIO 管脚，参考对应芯片型号的数据手册 (datasheet)。

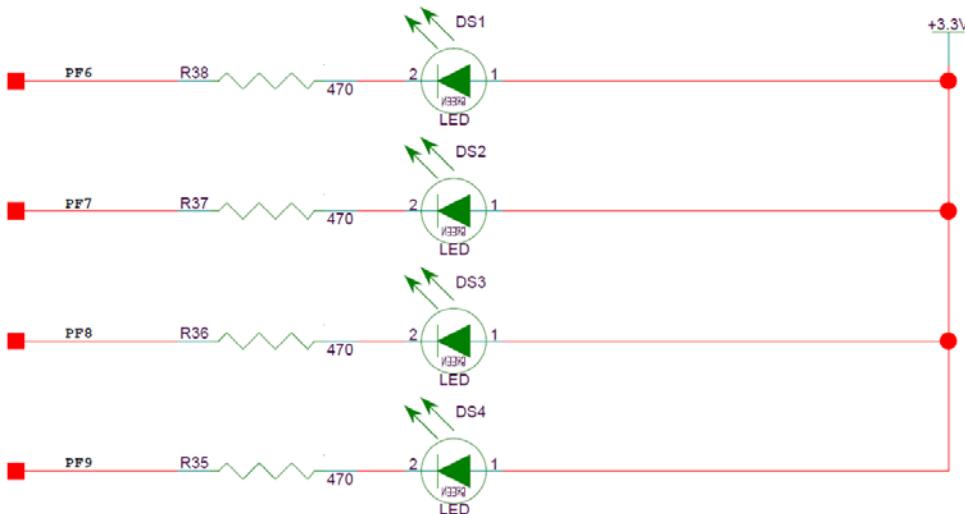
流水灯的关键实际上就是如何控制 STM32 处理器的 GPIO 引脚作为输出引脚，输出指定的高低电平。那怎么让引脚输出高低电平？我们回想一下 51 单片机的管脚控制中，我们使用的控制方式是类似于“P0^1=0”的方式控制管脚。而 STM32 控制 GPIO 管脚的方式，就复杂得多了。要控制 STM32 的 GPIO 管脚，就要涉及到与 GPIO 管脚相关的寄存器。通过对寄存器的配置，使 GPIO 引脚输出高低电平。

STM32 的 GPIO 管脚的配置都可以分为 3 步：

- ① 选择要配置的外设
- ② 配置外设的特定功能
- ③ 实现要达到的目标

我们 STM32 的外设比较多，有 GPIO，ADC，串口等等。每种外设里面还有细分的，比如 GPIO 分为 GPIOA，GPIOB……；串口分为串口 1、串口 2、串口 3 等等。比如流水灯实验中，我们用到的是外设 GPIO 口中的，GPIOF 端口的 PF6、PF7、PF8、PF9 共 4 个引脚。选择好外设后，配置外设的特定功能。比如 GPIO 引脚，可以用做输入引脚，也可以用做输出引脚。管脚的速率有 50MHz 的、10MHz 的、2MHz 的。流水灯实验中我们主要确定它的功能为输出，速率可以自由配置。最后，就是操作了，如流水灯实验的话，就是拉高拉低管脚。配置 GPIO 管脚的 3 个步骤，同样适用于其它的外设。

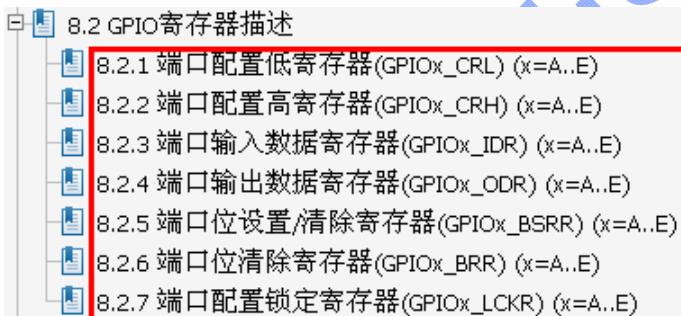
第一步：选择要配置的外设。流水灯实验中 LED 灯使用到了那个 GPIO 管脚，这个通过原理图可以确定。神舟 III 号开发板，实现流水灯的功能的话，使用到的管脚是 GPIOF 端口的 PF6、PF7、PF8、PF9，如下图。配置 GPIO 管脚的 3 个步骤中，这里我们就完成了第一步，确定了要操作管脚具体是那几个。



从原理图中，我们知道 LED 灯的一段连接了 3.3V 的电压，另一点接 470R 的电阻，电阻的另一端接到 STM32 芯片的对应的管脚上。比如，DS1 灯是连接的 PF6。我们只要让 LED 灯对应的管脚输出低电平就可以点亮 LED 灯，输出高电平熄灭 LED 灯。

说明：这个硬件电路图从神舟 III 号开发板的原理图中截取。[开发板的原理图](#)，在我们以后的学习中，也是经常用到的。

第二步：配置外设的特定功能。配置我们要配置 GPIO 管脚，先要了解和 GPIO 管脚相关的寄存器。我们看一下 ST 官方提供的参考手册，在 GPIO 管脚章节中，找到 GPIO 管脚相关的寄存器：



可以看到 STM32 的 GPIO 端口由 7 个寄存器控制。它们分别是：

- 配置 GPIO 管脚模式的 2 个配置寄存器 CRL 和 CRH。
- 2 个数据寄存器 IDR 和 ODR。
- 2 个位控制寄存器 BSRR 和 BRR。
- 1 个锁定寄存器 LCKR。

GPIO 管脚被分成 GPIOA、GPIOB、GPIOC……很多个组。那么对于端口 GPIOA 的 7 个控制寄存器分别为：GPIOA_CRL、GPIOA_CRH、GPIOA_IDR……。GPIOx_CRL 中的 “x” 可以换成 B、C、D……依此类推。也就是说每 7 个寄存器为一组，控制一组 GPIO 端口（比如 GPIOA）。实际上每组寄存器从物理上是分开的，是互不相关的。

注：在我们以后的学习中，会大量的查阅 ST 公司提供的[数据手册](#)和[参考手册](#)。这两个手册，我们在光盘资料中给大家提供。我们学习 ST 公司的 STM32 芯片，要想知道其功能严谨、详细的描述。一定要以 ST 官方提供的手册为准。学会阅读参考手册和数据手册是非常重要的。

实现本实验，最主要的配置是，配置 GPIOF 端口的 PF6、PF7、PF8、PF9 管脚为输出模式。实际上 STM32 的 IO 口可以由软件配置成 8 种模式：

- 模拟输入
- 输入浮空

- 输入下拉
- 输入上拉
- 开漏输出
- 推挽输出
- 复用功能开漏输出
- 复用功能推挽输出

可以看到在输出模式中，有开漏输出模式，有推挽输出模式，有复用功能推挽输出模式。我们要配置 GPIO 管脚的模式为推挽输出模式。

如下图，我们看一下配置 GPIOF 端口的 PF6、PF7 管脚为推挽输出模式的 **GPIOx_CRL** 寄存器。

8.2.1 端口配置低寄存器(**GPIOx_CRL**) (x=A..E)

偏移地址: 0x00

复位值: 0x4444 4444

| | | | | | | | | | | | | | | | |
|-----------|---|-----------|--|-----------|------------|-----------|------------|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF3[1:0] | MODE3[1:0] | CNF2[1:0] | MODE2[1:0] | CNF1[1:0] | MODE1[1:0] | CNF0[1:0] | MODE0[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位31:30 | CNFy[1:0]: 端口x配置位(y = 0...7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | 每4位配置一个引脚。 位0到位3这4位配置第一个管脚, 库函数中表示为: GPIO_Pin_0 | | | | | | | | | | | | |
| 位29:28 | MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz | | 配置一个引脚需要4位, 这4位中又分为两组。组 MODEy[1:0] , 确定管脚是输入模式还是输出模式。是输出模式时还确定了输出速度。确定了什么模式后, 由组 CNFy[1:0] 进一步确定输入/输出模式中具体是哪一种输入/输出模式。 | | | | | | | | | | | | |

对于任意一个GPIO端口，每个端口有16个引脚，每个引脚的模式由寄存器的4个位控制，4个位中又分为两组，每组两位。组**MODEy[1:0]** 控制引脚的模式及最高速度，组**CNFy[1:0]** 进一步控制引脚具体是输入/输出模式中的，什么模式。STM32寄存器是32位的， $32 \div 4 = 8$ 。也就是说它可以控制8个GPIO管脚。

配置GPIO引脚模式一共有两个寄存器，**GPIOx_CRL**寄存器和**GPIOx_CRH**寄存器。**GPIOx_CRL**是低8位寄存器，用来配置低8位引脚：pin0~pin7。**GPIOx_CRH**是高寄存器，用来配置高8位引脚：pin8~pin15。我们需要配置的是GPIOF端口的PF6、PF7、PF8、PF9，这个PF6、PF7管脚是通过**GPIOF_CRL**寄存器配置，PF8、PF9管脚是通过**GPIOF_CRH**寄存器配置的。

举例说明：当给**GPIOF_CRL**寄存器的第24至25位设置为参数“11”，并在第26至27位设置为参数“00”，则把GPIOF端口的PF6的模式配置成了“输出的最大速度为50MHz的通用推挽输出模式”，其它引脚可通过其**GPIOx_CRH**或**GPIOx_CRL**的其它寄存器位来配置。这里我们就学习了7个GPIO管脚相关的寄存器中的2个。

第三步：实现要达到的目标。接下来分析要控制引脚电平高低，实现LED灯的亮灭。我们看一下

寄存器GPIOx_BSRR，如下图：

8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

地址偏移: 0x10

复位值: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |

位31:16 **BRy**: 清除端口x的位y (y = 0...15) (Port x Reset bit y)
这些位只能写入并只能以字(16位)的形式操作。
0: 对对应的ODRy位不产生影响
1: 清除对应的ODRy位
注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。

位15:0 **BSy**: 设置端口x的位y (y = 0...15) (Port x Set bit y)
这些位只能写入并只能以字(16位)的形式操作。
0: 对对应的ODRy位不产生影响
1: 设置对应的ODRy位为1

32位的寄存器GPIOx_BSRR可分为两部分, **BRy**和**BSy**。一个引脚y的输出数据由GPIOx_BSRR寄存器位的2个位来控制分别为**BRy** (**Bit Reset y**)和**BSy** (**Bit Set y**), **BRy**位用于写1清零, 使引脚输出低电平, **BSy**位用来写1置1, 使引脚输出高电平。而对这两个位进行写零都是无效的。寄存器GPIOx_BSRR是通过对寄存器GPIOx_ODR的控制, 实现控制引脚输出电平的。其实也可以直接设置寄存器ODR来控制引脚的输出。

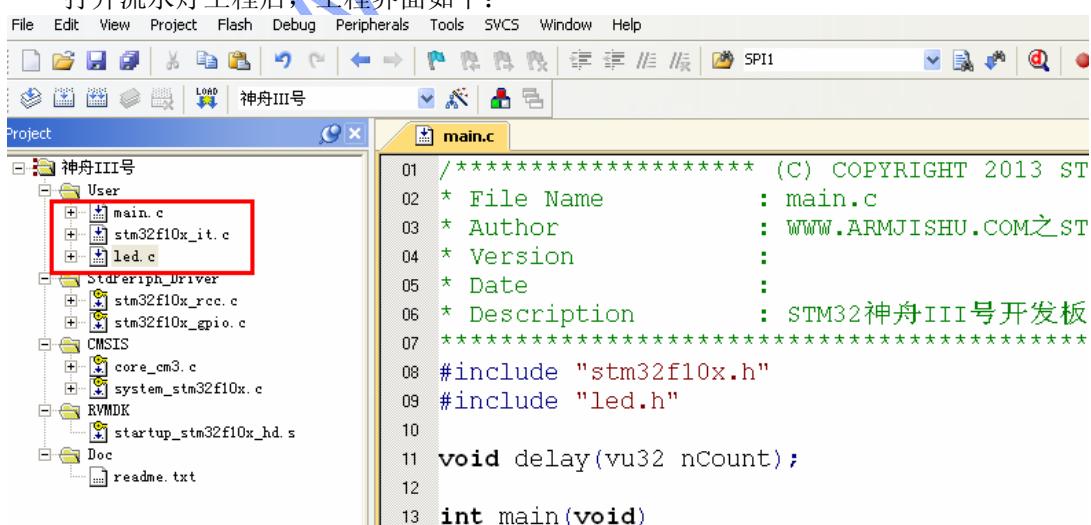
例如: 对F端口的寄存器GPIOF_BSRR的第7位(**BS6**) 进行写1, 则F端口的第6引脚被设置为1, 输出高电平, 若要令第PF6引脚再输出低电平, 则需要向GPIOF_BSRR的第23位(**BR6**) 写1。

理解了寄存器GPIOx_BSRR和GPIOx_ODR, 那么学习对应的寄存器GPIOx_BRR和GPIOx_IDR也没什么问题。这里我们就学习了7个GPIO管脚相关的寄存器中的4个。到此, 我们只剩下最后一个寄存器GPIOx_LCKR没有接触。这个寄存器, 一般我们使用得较少, 用到的时候, 再给大家分析。

7.1.2 LED具体代码分析

● 主要工程文件描述

打开流水灯工程后, 工程界面如下:



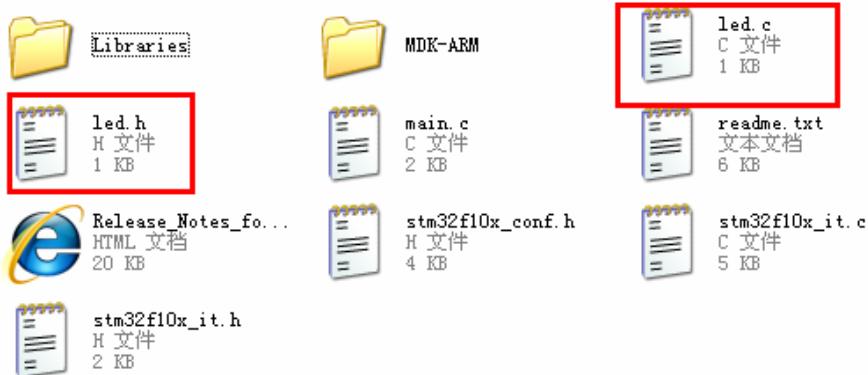
工程中包含5个组, User、StdPeriph_Driver、CMSIS、RVMDK、Doc。其中组CMSIS、RVMDK、Doc里面的内容在其它的工程中, 没有什么区别。

组StdPeriph_Driver中，主要用到的固件库文件有stm32f10x_gpio.c和stm32f10x_rcc.c（注：固件库中每一个.c文件对应一个.h文件）。使用库开发的方式，stm32f10x_rcc.c文件是每个工程都使用到的。因为我们使用外设的时候，需要使用使能外设时钟的函数，这些函数在文件stm32f10x_rcc.c中，同时系统时钟配置函数也在里边。我们这里拉高拉低GPIO管脚，使用到文件stm32f10x_gpio.c中的函数，所以我们将它包含进来。其实，如果嫌麻烦的话，也可以将固件库里面的文件，全部包含进来。就是编译的时候慢一点，也没有多大关系。

组User中，包含了main.c文件，主程序在这个文件中。led.c文件，是我们编写的用户文件。这个文件无论是文件名、还是文件里面的内容，完全由个人决定。

● 用户文件led.c和led.h分析

工程结构和工程的配置和我们在“新建模板工程”章节中的新建的模板工程是一致的。我们看一下，工程包含的文件：



我们在文件夹下新建led.c和led.h，并把led.c添加到工程之中。led.c文件中输入代码如下：

```
08 #include "led.h"
09
10 /*初始化4个LED灯*/
11 void LED_config(void)
12 {
13     GPIO_InitTypeDef GPIO_InitStructure;
14
15     /* 配置神舟III号LED灯使用的GPIO管脚模式*/
16     RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE); /*使能LED灯使用的
17
18     GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN;
19     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
20     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21
22     GPIO_Init(GPIO_LED, &GPIO_InitStructure); /*神州III号使用的LEI
23 }
24 }
```

可以看到代码中，首先包含了头文件“led.h”。这个头文件的内容如下：

```

01 #ifndef __LED_H
02 #define __LED_H
03 #include "stm32f10x.h"
04
05 /*神州III号LED灯相关定义*/
06 #define RCC_GPIO_LED RCC_APB2Periph_GPIOF
07 #define LEDn 4
08 #define GPIO_LED GPIOF
09
10 #define DS1_PIN GPIO_Pin_6
11 #define DS2_PIN GPIO_Pin_7
12 #define DS3_PIN GPIO_Pin_8
13 #define DS4_PIN GPIO_Pin_9
14
15 void LED_config(void);
16
17 #endif

```

#ifndef __LED_H #define __LED_H#endif作用是避免重定义的错误，这个是C语言的知识。
#include "stm32f10x.h"，包含文件stm32f10x.h。这个文件的重要意义，在我们“初始STM32库”章节中进行了介绍。这里就不重复详细说明。这里，特别提的一点是，文件stm32f10x.h中包含了头文件stm32f10x_conf.h。这个文件用来配置使用了什么外设的头文件，流水灯的章节我们需要使用RCC和GPIO的头文件就可以了。但是，一般我们将所有的头文件都包含，就不用每次开发的时候，都要看一下，对应外设的头文件是否包含在里面了。包含的部分头文件如下：

```

#include "stm32f10x_fsmc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_i2c.h"
#include "stm32f10x_iwdg.h"
#include "stm32f10x_pwr.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_rtc.h"

```

“led.h”文件中，最后是“void LED_config(void)”这也是C语言的知识，函数声明。我们要使用函数LED_config(void)时只需要将“led.h”文件包含就可以了。这里.C文件和.h文件是成对出现的。比如：**led.c**和**led.h**这对文件。

函数LED_config()的分析：

分析完了**led.h**文件后，我们回到**led.c**文件，详细分析**LED_config()**函数。函数中，首先定义了一个结构体类型为**GPIO_InitTypeDef**的结构体**GPIO_InitTypeDef**。追踪其定义原型如下：

```

typedef struct
{
    uint16_t GPIO_Pin;
    GPIOSpeed_TypeDef GPIO_Speed;
    GPIOMode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;

```

可以看到**GPIO_InitTypeDef**类型的结构体有三个成员。**uint16_t**类型的**GPIO_Pin**，确定要配置的引脚，可以是**GPIO_Pin_0~GPIO_Pin_15**这16个引脚中的一个或者多个。流水灯实验中用到的是GPIOF的**GPIO_Pin_6**、**GPIO_Pin_7**、**GPIO_Pin_8**和**GPIO_Pin_9**。

GPIOSpeed_TypeDef类型的**GPIO_Speed**确定引脚的输出速率的最高值，可以有G

PIO_Speed_10MHz、GPIO_Speed_2MHz、GPIO_Speed_50MHz三个选择。我们将对应的管脚配置成GPIO_Speed_50MHz的输出速率。GPIOMode_TypeDef类型的**GPIO_Mode**，确定管脚的输入/输出模式。这里我们将管脚都配置成，推挽输出模式（GPIO_Mode_Out_PP）。

GPIO_InitTypeDef 类型结构体的三个成员的这些可配置的数值（如 GPIO_Pin_0，GPIO_Speed_2MHz，GPIO_Mode_Out_PP等），由ST的库文件封装成见名知义的枚举常量。这使我们编写代码变得非常简便。

LED_Init()函数中的最后一行代码是GPIO_Init(GPIOF, &GPIO_InitStructure)。函数GPIO_Init()是ST提供的库函数，我们使用的时候，只要知道函数的功能，输入怎么样的函数参数即可。**这些我们都**可以能通过查找库帮助文档获得。

```
void GPIO_Init ( GPIO_TypeDef * GPIOx,
                  GPIO_InitTypeDef * GPIO_InitStruct
                )
```

Initializes the GPIOx peripheral according to the specified parameters in the GPIO_InitStruct.

Parameters:

GPIOx: where x can be (A..G) to select the GPIO peripheral.

GPIO_InitStruct: pointer to a **GPIO_InitTypeDef** structure that contains the configuration information for the specified GPIO peripheral

可以知道

GPIO_Init()函数有两个参数。第一个参数是**GPIO_TypeDef**型的指针，允许值是GPIOA~GPIOG。第二个参数是**GPIO_InitTypeDef**型的指针，它的值是**GPIO_InitTypeDef**型指针变量。我们的流水灯实验中，第一个参数是GPIOF，第二个参数是&GPIO_InitStructure（我们前面对这个GPIO_InitStructure这个结构体进行了配置）。

对于STM32的外设，一般来说它的初始化方式是类似的，都是通过结构体，设定外设的参数，再调用类似于GPIO_Init()的函数，初始化外设。不同的是，外设不同，对应的结构体不同，调用的初始化函数不同。

● 文件main.c分析

文件内容如下：

```
13 int main(void)
14 {
15     LED_config(); // 调用4个LED灯初始化函数
16
17     while(1)
18     {
19         GPIO_ResetBits(GPIOF,GPIO_Pin_6); // 第一灯亮
20         delay(800000); // 延时
21         GPIO_SetBits(GPIOF,GPIO_Pin_6); // 第一灯灭
22         delay(800000); // 延时
23
24         GPIO_ResetBits(GPIOF,GPIO_Pin_7); // 第二灯亮
25         delay(800000); // 延时
26         GPIO_SetBits(GPIOF,GPIO_Pin_7); // 第二灯灭
27         delay(800000); // 延时
28
29         GPIO_ResetBits(GPIOF,GPIO_Pin_8); // 第三灯亮
30         delay(800000); // 延时
31         GPIO_SetBits(GPIOF,GPIO_Pin_8); // 第三灯灭
32         delay(800000); // 延时
33
34         GPIO_ResetBits(GPIOF,GPIO_Pin_9); // 第四灯亮
35         delay(800000); // 延时
36         GPIO_SetBits(GPIOF,GPIO_Pin_9); // 第四灯灭
37         delay(800000); // 延时
38     }
39
40 }
```

文件中，首先包含了led.h文件（#include "led.h"）。然后声明延时函数。我们主要看一下main函数。在芯片上电(复位)后，经过启动文件中SystemInit()函数配置好了时钟，就进入main函数了。

首先，主程序首先调用了在led.c文件中的LED_config()函数，完成了对GPIOF的GPIO_Pin_6、GPIO_Pin_7、GPIO_Pin_8、GPIO_Pin_9的初始化。紧接着就在while死循环里不断执行在stm32f10x_gpio.c文件中的函数GPIO_ResetBits()和函数GPIO_SetBits()，并加上延时函数，使各盏LED轮流亮灭。

我们看一下，函数GPIO_SetBits()

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Pin));

    GPIOx->BSRR = GPIO_Pin;
}
```

可以发现，这个函数里边实际上，是使用了外设GPIO的寄存器BSRR使电平发生变化。到此，我们整个控制LED灯的工程的讲解就完成了。

7.1.3 STM32的时钟系统

在主函数 main.c 中，我们并没有看到系统函数配置的地方。实际上系统时钟在芯片上电(复位)后，经过启动文件中的 SystemInit() 函数配置好了。

● 什么是时钟

从 CPU 的时钟说起。

计算机是一个十分复杂的电子设备。它由各种集成电路和电子器件组成，每一块集成电路中都集成了数以万计的晶体管和其他电子元件。这样一个十分庞大的系统，要使它能够正常地工作，就必须有一个指挥，对各部分的工作进行协调。各个元件的动作就是在这个指挥下按不同的先后顺序完成自己的操作的，这个先后顺序我们称为时序。时序是计算机中一个非常重要的概念，如果时序出现错误，就会使系统发生故障，甚至造成死机。那么是谁来产生和控制这个操作时序呢？这就是“时钟”。“时钟”可以认为是计算机的“心脏”，如同人一样，只有心脏在跳动，生命才能够继续。不要把计算机的“时钟”等同于普通的时钟，它实际上是由晶体振荡器产生的连续脉冲波，这些脉冲波的幅度和频率是不变的，这种时钟信号我们称为外部时钟。它们被送入 CPU 中，再形成 CPU 时钟。不同的 CPU，其外部时钟和 CPU 时钟的关系是不同的，下表列出了几种不同 CPU 外部时钟和 CPU 时钟的关系。

CPU 时钟周期通常为节拍脉冲或 T 周期，它是处理操作的最基本的单位。

在微程序控制器中，时序信号比较简单，一般采用节拍电位——节拍脉冲二级体制。就是说它只要一个节拍电位，在节拍电位又包含若干个节拍脉冲（时钟周期）。节拍电位表示一个 CPU 周期的时间，而节拍脉冲把一个 CPU 周期划分为几个叫较小的时间间隔。根据需要这些时间间隔可以相等，也可以不等。

指令周期是取出并执行一条指令的时间。

指令周期常常有若干个 CPU 周期，CPU 周期也称为机器周期，由于 CPU 访问一次内存所花费的时间较长，因此通常用内存中读取一个指令字的最短时间来规定 CPU 周期。这就是说，这就是说一条指令取出阶段（通常为取指）需要一个 CPU 周期时间。而一个 CPU 周期时间又包含若干个时钟周期（通常为节拍脉冲或 T 周期，它是处理操作的最基本的单位）。这些时钟周期的总和则规定了一个 CPU 周期的时间宽度。

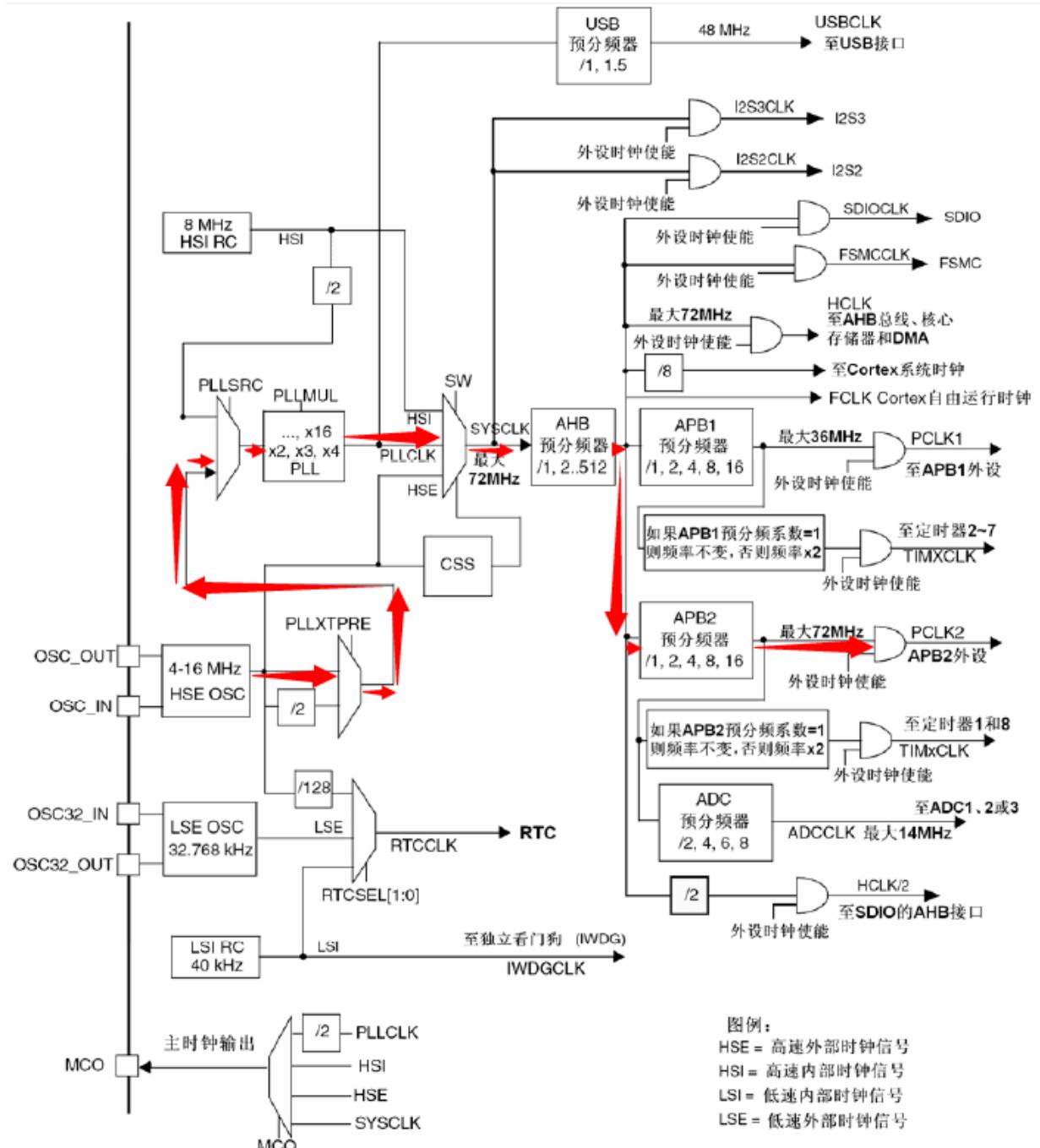
● STM32的系统时钟

STM32芯片为了实现低功耗，设计了一个功能完善但却非常复杂的时钟系统。我们使用外设的时候是需要开启外部时钟的。STM32的芯片可以分为小容量产品、中容量产品、大容量产品，互联型产品。前面3种类型按芯片的Flash大小来分。而STM32互联系列让设计人员可以在同时需要以太网、USB、CAN和音频级I2S接口的产品设计中发挥工业标准的32位微处理器的优异性能。按芯片Flash大小来分类的3种芯片的系统时钟是大同小异的。而互联型产品芯片的系统时钟和它们比较差异较大的。我们神

舟III号开发板使用的主芯片STM32F103ZET是大容量型的。我们下面分析它的系统时钟。

● 小容量型、中容量型、大容量型产品的时钟树&时钟源

首先，从整体上了解STM32的时钟系统。



从上图可知，STM32F103ZET有以下4个时钟源：

高速外部时钟（HSE）：以外部晶振作时钟源，晶振频率可取范围为4~16MHz，我们采用8MHz的晶振。

高速内部时钟（HSI）：由内部RC振荡器产生，频率为8MHz，但不稳定。

低速外部时钟(LSE): 以外部晶振作时钟源，主要提供给实时时钟模块，所以一般采用32.768KHz。

低速内部时钟（LSI）：由内部RC振荡器产生，也主要提供给实时时钟模块，频率大约为40KHz。

STM32的时钟走向，从图的左边开始，从时钟源一步步分配到外设时钟。

从时钟频率来说，又分为**高速时钟**和**低速时钟**，高速时钟是提供给芯片主体的主时钟，而低速时钟只是提供给芯片中的RTC（实时时钟）及独立看门狗使用。

从芯片角度来说，时钟源分为内部时钟与外部时钟源，内部时钟是在芯片内部RC振荡器产生的，起振较快，在芯片刚上电的时候，默认使用内部高速时钟。而外部时钟信号是由外部的晶振输入的，在精度和稳定性上都有很大优势，上电之后我们再通过软件配置，转而采用外部时钟信号。

● STM32使用的高速外部时钟（HSE）分析

芯片刚上电的时候，默认使用内部高速时钟。上电之后再通过软件配置，转而采用高速外部时钟信号。系统时钟，在ST官方提供的系统启动文件(startup_stm32f10x_xx.s)中，调用SystemInit()给我们配置好了系统时钟。以神舟III号开发板为例，我们在外部提供的晶振的频率为8MHz。最终配置出来的主频是72MHz。

我们分析一下，如何得到外设 GPIOF 的时钟，下面是一条时钟的“脉络”，其中的标号和时钟树中的标号一一对应。“脉络”也通过箭头标出。

对此条时钟路径做如下解析：

(HSE)，首先我们的外部时钟是8MHz。经过 PLLXTPRE，直接选择 HSE 为输入，得 8 (MHz)。经过 PLLSRC 选择，还是 8 (MHz)。8MHz 经过 PLLMULL 的 9 倍频， $8 \times 9 = 72$ (MHz)。经过 SW 选择 PLLCLK 做为 SYSCLK (系统时钟) 的输入时钟，那么 SYSCLK 等于输送过来的 72MHz。外设 GPIO 的时钟来源于 APB2 总线，那么经过 AHB 预分频器，不分频。在经过 APB2 预分频器，不分频。最后 APB2 外设 GPIO 就可以得到 72MHz 的时钟源了。

ST 提供的启动文件(startup_stm32f10x_xx.s)中，调用 SystemInit()。在函数 SystemInit() 中，配置时钟的是函数 SetSysClock()。函数 SetSysClock() 又调用函数 SetSysClockTo72()。设置 72MHz 的系统时钟。函数 SetSysClockTo72() 中的时钟设置路径与我们给出的时钟设置“脉络”是一致的。

函数 SetSysClock() 内容如下：

```
static void SetSysClock(void)
{
    #ifdef SYSCLK_FREQ_HSE
        SetSysClockToHSE();
    #elif defined SYSCLK_FREQ_24MHz
        SetSysClockTo24();
    #elif defined SYSCLK_FREQ_36MHz
        SetSysClockTo36();
    #elif defined SYSCLK_FREQ_48MHz
        SetSysClockTo48();
    #elif defined SYSCLK_FREQ_56MHz
        SetSysClockTo56();
    #elif defined SYSCLK_FREQ_72MHz
        SetSysClockTo72();
    #endif

    /* If none of the define above is
       source (default after reset) */
}
```

可以发现，其实通过定义不同的宏系统时钟，可以配置成不同频率。我们看一下，代码中定义了那个宏。

```
/* #define SYSCLK_FREQ_HSE      HSE_VALUE */
/* #define SYSCLK_FREQ_24MHz    24000000 */
/* #define SYSCLK_FREQ_36MHz    36000000 */
/* #define SYSCLK_FREQ_48MHz    48000000 */
/* #define SYSCLK_FREQ_56MHz    56000000 */
#define SYSCLK_FREQ_72MHz    72000000
```

对宏进行追溯，可以发现，只定义了 SYSCLK_FREQ_72MHz 的宏。其它的都被注销了。至于函数 SetSysClockTo72() 中，系统时钟为 72MHz 的方程，参考我们提供的时钟“脉络”。

系统时钟的配置，不是说想怎么配就怎么配。比如 APB2 总线的时钟，最高是 72MHz，而 APB1 总线最高 36MHz。具体参考 ST 公司提供的参考手册关于系统时钟的说明。

● 常用的HCLK、FCLK、PCLK1、PCLK2时钟说明

从时钟树的分析，看到经过一系列的倍频、分频后得到了几个与我们开发密切相关的时钟。

SYSCLK：系统时钟，STM32大部分器件的时钟来源。主要由AHB预分频器分配到各个部件。

HCLK:由AHB预分频器直接输出得到，它是高速总线AHB的时钟信号，提供给存储器，DMA及cortex内核，是cortex内核运行的时钟，cpu主频就是这个信号，它的大小与STM32运算速度，数据存取速度密切相关。

FCLK：同样由AHB预分频器输出得到，是内核的“自由运行时钟”。 “自由”表现在它不来自时钟 HCLK，因此在HCLK时钟停止时 FCLK 也继续运行。它的存在，可以保证在处理器休眠时，也能够采样和到中断和跟踪休眠事件，它与HCLK互相同步。

PCLK1: 外设时钟，由APB1预分频器输出得到，最大频率为36MHz，提供给挂载在APB1总线上的外设。

PCLK2: 外设时钟，由APB2预分频器输出得到，最大频率可为72MHz，提供给挂载在APB2总线上的外设。

为什么STM32的时钟系统如此复杂，有倍频、分频及一系列的外设时钟的开关。需要倍频是考虑到电磁兼容性，如外部直接提供一个72MHz的晶振，太高的振荡频率可能会给制作电路板带来一定的难度。分频是因为STM32既有高速外设又有低速外设，各种外设的工作频率不尽相同，如同pc机上的南北桥，把高速的和低速的设备分开来管理。最后，每个外设都配备了外设时钟的开关，当我们不使用某个外设时，可以把这个外设时钟关闭，从而降低STM32的整体功耗。

7.1.4 STM32的地址映射

首先回顾一下在51单片机上点亮LED是怎样实现的。

```
1. #include<reg52.h>
2. int main (void)
3. {
4.     P0=0;
5.     while(1);
6. }
```

以上代码将P0的引脚拉低。当然，这里省略了头文件。为什么这个P0 =0; 句子就能控制P0端口为低电平？关键之处在于这个代码所包含的头文件<reg52.h>。

在这个文件下有以下的定义：

```
1. /* BYTE Registers */
2. sfr P0      = 0x80;
3. sfr P1      = 0x90;
4. sfr P2      = 0xA0;
5. sfr P3      = 0xB0;
6. sfr PSW    = 0xD0;
7. sfr ACC    = 0xE0;
8. sfr B       = 0xF0;
9. sfr SP      = 0x81;
10. sfr DPL   = 0x82;
11. sfr DPH   = 0x83;
12. sfr PCON  = 0x87;
13. sfr TCON  = 0x88;
14. sfr TMOD  = 0x89;
15. sfr TL0    = 0x8A;
16. sfr TL1    = 0x8B;
17. sfr TH0    = 0x8C;
18. sfr TH1    = 0x8D;
19. sfr IE     = 0xA8;
20. sfr IP     = 0xB8;
21. sfr SCON  = 0x98;
22. sfr SBUF  = 0x99;
```

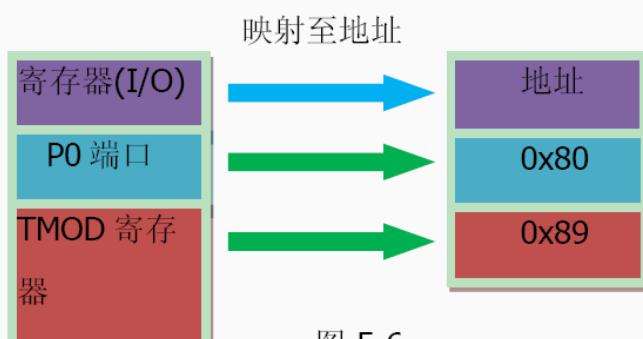


图 5-6

这些定义被称为地址映射。

所谓地址映射，就是将芯片上的存储器 甚至I/O等资源与地址建立一一对应的关系。如果某地址对应着某寄存器，我们就可以运用c语言的指针来寻址并修改这个地址上的内容，从而实现修改该寄存

器的内容。

正是因为`<reg52.h>`头文件中有了对于各种寄存器和I/O端口的地址映射，我们才可以在51单片机程序中方便地使用`P2^0 = 0xFF; TMOD = 0xFF`等赋值句子对寄存器进行配置，从而控制单片机。

Cortex-M3的地址映射也是类似的。Cortex-M3有32根地址线，所以它的寻址空间大小为 2^{32} bit=4GB。ARM公司设计时，预先把这4GB的寻址空间大致地分配好了。它把地址从0x4000 0000至0x5FFF FFFF(512MB)的地址分配给片上外设。通过把片上外设的寄存器映射到这个地址区，就可以简单地以访问内存的方式，访问这些外设的寄存器，从而控制外设的工作。结果，片上外设可以使用C语言来操作。M3存储器映射见下图：

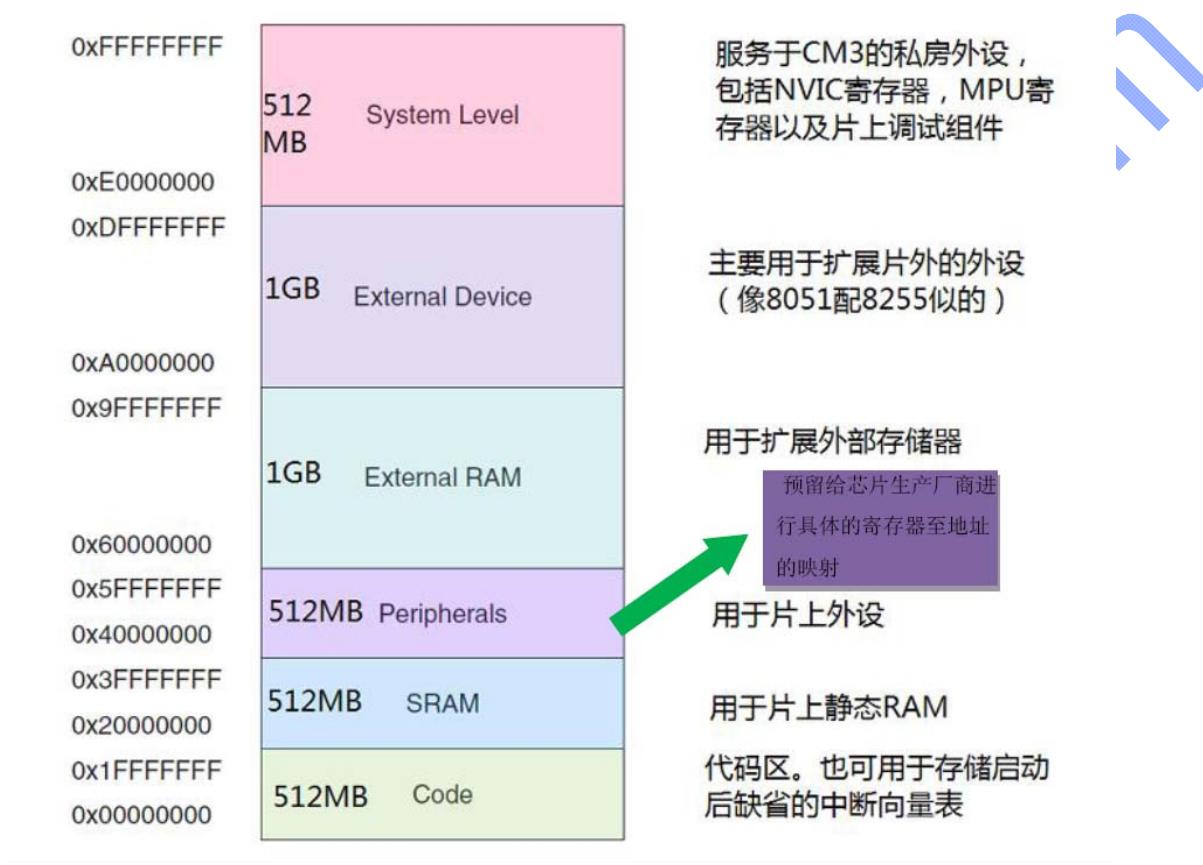


图 5-7

`stm32f10x.h`这个文件中重要的内容就是把STM32的所有寄存器进行地址映射。如同51单片机的`<reg52.h>`头文件一样，`stm32f10x.h`像一个大表格，我们在使用的时候就是通过宏定义进行类似查表的操作，大家想像一下没有这个文件的话，我们要怎样访问STM32的寄存器？有什么缺点？不进行这些宏定义的缺点有：

- 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。

当然，这些工作都是由ST的固件工程师来完成的，只有设计M3的人才是最了解M3的，才能写出完美的库。

在这里我们以外设GPIOF为例，在这个文件中有这样的一系列宏定义：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define GPIOF_BASE (APB2PERIPH_BASE + 0x1C00)
#define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
```

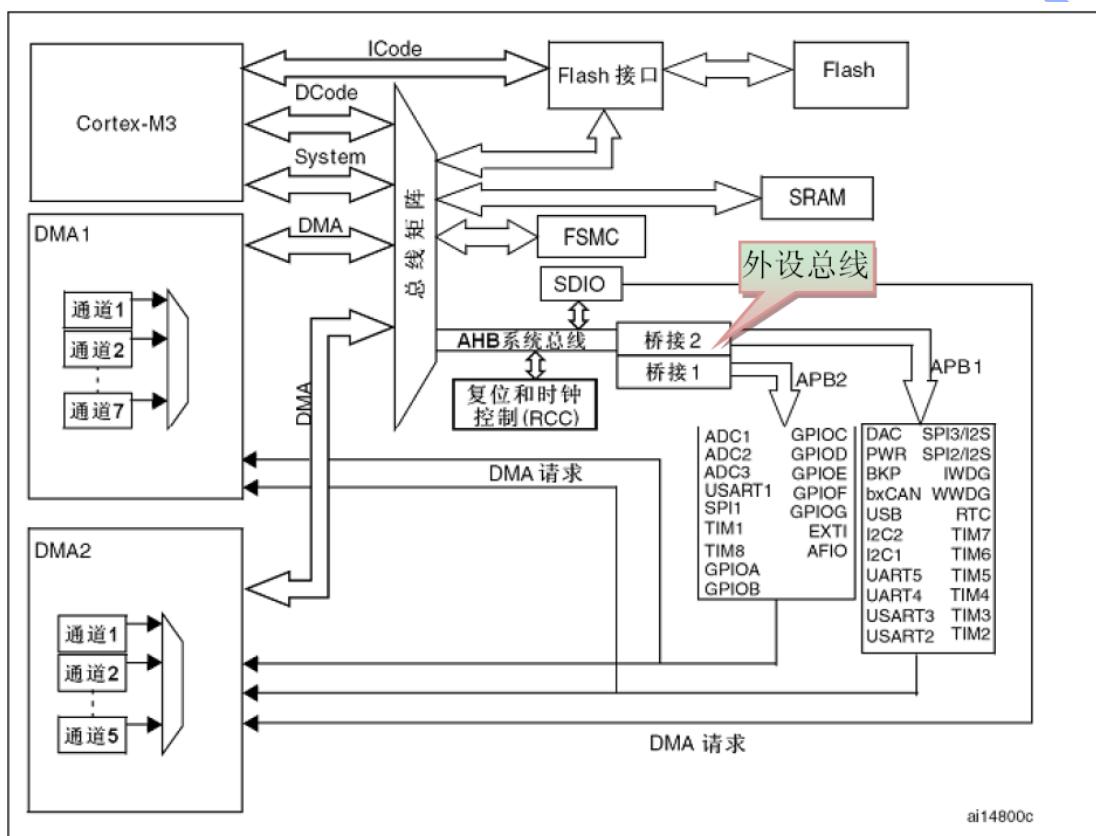
这几个宏定义是从文件中的几个部分抽离出来的，具体的读者可参考[stm32f10x.h源码](#)。

外设基址

首先看到`PERIPH_BASE`这个宏，宏展开为`0x4000 0000`，并把它强制转换为`uint32_t`的32位类型数据，这是因为地STM32的地址是32位的，是不是觉得`0x4000 0000`这个地址很熟？是的，这个是Cortex-M3核分配给片上外设的从`0x4000 0000`至`0x5FFF FFFF`的512MB寻址空间中 的第一个地址，我们把`0x4000 0000`称为外设基址。

总线基址

接下来是宏`APB2PERIPH_BASE`，宏展开为`PERIPH_BASE`（外设基址）加上偏移地址`0x1 0000`，即指向的地址为`0x4001 0000`。这个`APB2PERIPH_BASE`宏是什么地址呢？STM32不同的外设是挂载在不同的总线上的。有AHB总线、APB2总线、APB1总线，挂载在这些总线上的外设有特定的地址范围。



其中像GPIO、串口1、ADC及部分定时器是挂载这个被称为APB2的总线上，挂载到APB2总线上的外设地址空间是从`0x4001 0000`至地址`0x4001 3FFF`。这里的第一个地址，也就是`0x4001 0000`,被称为`APB2PERIPH_BASE` (APB2总线外设的基址)。

而APB2总线基址相对于外设基址的偏移量为`0x1 0000`个地址，即为APB2相对外设基址的偏移地址。

见表：

| 地址范围 | 总线 | 总线基地址 | 总线基地址相对外设基地址(0x4000000)的偏移量 |
|---------------------------|------|-------------|-----------------------------|
| 0x4001 8000 -0x5003 FFFF | AHB | 0x4001 8000 | 0x1 8000 |
| 0x4001 0000 - 0x4001 7FFF | APB2 | 0x4001 0000 | 0x1 0000 |
| 0x4000 0000 - 0x4000FFFF | APB1 | 0x4000 0000 | 0x0 0000 |

由这个表我们可以知道，**stm32f10x.h**这个文件中必然还有以下的宏：

```
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
```

寄存器组基地址

最后到了宏**GPIOF_BASE**，宏展开为**APB2PERIPH_BASE**(APB2总线外设的基地址)加上相对APB2总线基地址的偏移量**0x1C00**得到了GPIOF端口的寄存器组的基地址。这个所谓的寄存器组又是什么呢？它包括什么寄存器？

细看**stm32f10x.h**文件，我们还可以发现以下类似的宏：

```
1249 #define AFIO_BASE          (APB2PERIPH_BASE + 0x0000)
1250 #define EXTI_BASE          (APB2PERIPH_BASE + 0x0400)
1251 #define GPIOA_BASE          (APB2PERIPH_BASE + 0x0800)
1252 #define GPIOB_BASE          (APB2PERIPH_BASE + 0x0C00)
1253 #define GPIOC_BASE          (APB2PERIPH_BASE + 0x1000)
1254 #define GPIOD_BASE          (APB2PERIPH_BASE + 0x1400)
1255 #define GPIOE_BASE          (APB2PERIPH_BASE + 0x1800)
1256 #define GPIOF_BASE          (APB2PERIPH_BASE + 0x1C00) // 这一行被红色框选
1257 #define GPIOG_BASE          (APB2PERIPH_BASE + 0x2000)
```

除了GPIOF寄存器组的地址，还有GPIOA、GPIOB、GPIOC等的地址，并且这些地址是不一样的。

前面提到，每组GPIO都对应着独立的一组寄存器，查看stm32的datasheet，看到寄存器说明如下图：

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

| 偏移地址：0x00 | | | | | | | | | | | | | | | |
|-----------------|------------|-----------|------------|-----------|------------|-----------|------------|----|----|----|----|----|----|----|----|
| 复位值：0x4444 4444 | | | | | | | | | | | | | | | |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF7[1:0] | MODE7[1:0] | CNF6[1:0] | MODE6[1:0] | CNF5[1:0] | MODE5[1:0] | CNF4[1:0] | MODE4[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF3[1:0] | MODE3[1:0] | CNF2[1:0] | MODE2[1:0] | CNF1[1:0] | MODE1[1:0] | CNF0[1:0] | MODE0[1:0] | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

图 5-9

注意到这个说明中有一个偏移地址：0x00，这里的偏移地址的是相对哪个地址的偏移呢？下面进行举例说明。

对于GPIOF组的寄存器，GPIOF含有的 端口配置低8位寄存器(**GPIOF_CRL**) 寄存器地址为：

GPIOF_BASE +0x00。

假如是GPIOA组的寄存器，则GPIOA含有的 端口配置低8位寄存器(GPIOA_CRL)寄存器地址为：GPIOA_BASE+0x00。

也就是说，这个偏移地址，就是该寄存器 相对所在寄存器组基地址的偏移量。

于是，读者可能会想，大概这个文件含有一个类似如下的宏：

```
1. #define GPIOA_CRL (GPIOAF_BASE + 0x00)
```

这个宏，定义了GPIOA_CRL寄存器的具体地址，然而，在stm32f10x.h文件中并没有这样的宏。ST公司的工程师采用了更巧妙的方式来确定这些地址，请看下一小节——STM32库对寄存器的封装。

7.1.5 STM32库对寄存器的封装

ST的工程师用结构体的形式，封装了寄存器组，c语言结构体学的不好的同学，可以在这里补补课了。在stm32f10x.h文件中，有以下代码：

```
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG ((GPIO_TypeDef *) GPIOG_BASE)
```

有了这些宏，我们就可以定位到具体的寄存器地址。像GPIOA、GPIOB等这个的，我们可以直接追溯到它们的地址。可以像GPIOA_CRL这样的寄存器，在stm32f10x.h文件中并没有找到像GPIOA、GPIOB等类似的地址定义。那么它们是怎么定义的呢？

在这里发现了一个陌生的类型**GPIO_TypeDef**，追踪它的定义，可以在stm32f10x.h 文件中找到如下代码：

```
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;
```

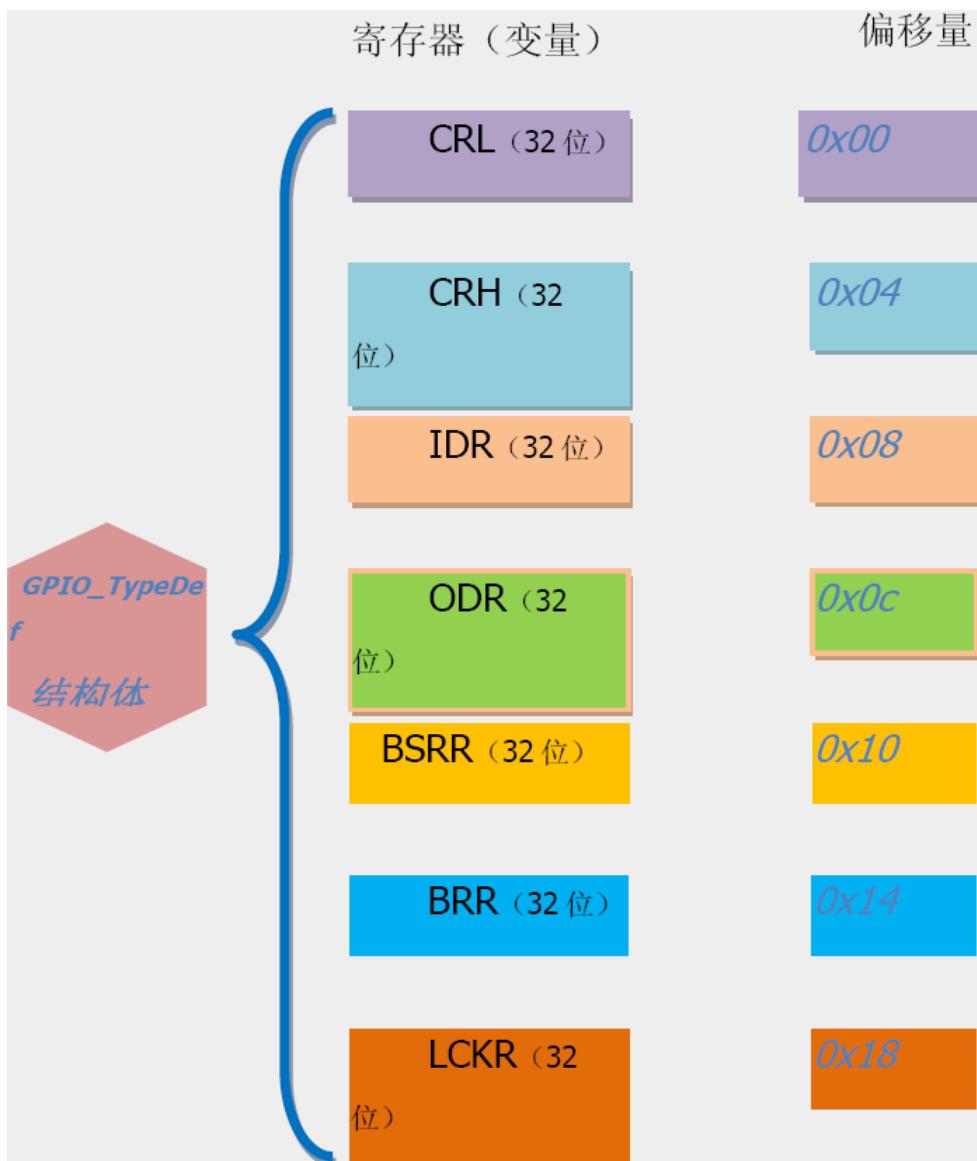
其中 **__IO** 也是一个ST库定义的宏，宏定义如下：

```
1. #define __IO volatile /*!< defines 'write only' permissions */
2. #define __IO volatile /*!< defines 'read / write' permissions */
```

volatile 是c语言的一个关键字，有关**volatile**的用法可查阅相关的C语言书籍。

回到**GPIO_TypeDef** 这段代码，这个代码用**typedef** 关键字声明了名为**GPIO_TypeDef**的结构体类型，结构体内又定义了7个 **__IO uint32_t** 类型的变量。这些变量每个都为32位，也就是每个变量占内存空间4个字节。在c语言中，结构体内变量的存储空间是连续的，也就是说假如我们定义了一个**GPIO_TypeDef**，这个结构体的首地址（变量CRL的地址）若为**0x4001 1000**，那么结构体中第二个变量（CRH）的地址即为**0x4001 1000 +0x04**，正是代表4个字节地址的偏移量。

细心的读者会发现，这个0x04偏移量，正是GPIOx_CRH寄存器相对于所在寄存器组的偏移地址。同理，**GPIO_TypeDef** 结构体内其它变量的偏移量，也和相应的寄存器偏移地址相符。于是，只要我们匹配了结构体的首地址，就可以确定各寄存器的具体地址了。



有了这些准备，就可以分析本小节的第一段代码了：

```
#define GPIOA          ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB          ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC          ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD          ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE          ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF          ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG          ((GPIO_TypeDef *) GPIOG_BASE)
```

GPIOA_BASE 在上一小节已解析，是一个代表GPIOA组寄存器的地址。**(GPIO_TypeDef *)** 在这里的作用则是把**GPIOA_BASE** 地址转换为**GPIO_TypeDef** 结构体指针类型。

有了这样的宏，以后我们写代码的时候，如果要修改GPIO的寄存器，就可以用以下的方式来实现。代码分析见注释。

1. GPIO_TypeDef * GPIOx; // 定义一个 GPIO_TypeDef 型结构体指针 GPIOx
2. GPIOx = GPIOA; // 把指针地址设置为宏 GPIOA 地址
3. GPIOx->CRL = 0xffffffff; // 通过指针访问并修改 GPIOA CRL 寄存器

通过类似的方式，我们就可以给具体的寄存器写上适当的参数，控制STM32了。是不是觉得很巧妙？但这只是库开发的皮毛，而且实际上我们并不是这样使用库的，库为我们提供了更简单的开发方

式。M3的库可谓尽情绽放了c的魅力，如果你是单片机初学者，c语言初学者，那么请你不要放弃与M3库邂逅的机会。是否选择库，就差你一个闪亮的回眸。

7.1.6 下载与验证

如果使用JLINK下载固件，请按[3.2如何使用JLINK软件](#)下载固件到神舟III号开发板小节进行操作。

如果使用USB下载固件，请按[错误！未找到引用源。错误！未找到引用源。](#)小节进行操作。

如果使用串口下载固件，请按[3.3如何通过串口下载一个固件到神舟III号开发板](#)小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.1.7 实验现象

将固件下载在神舟 III 号 STM32 开发板后，可以看到神舟 III 号开发板的四个 LED 灯（LED1~4）轮流闪亮，实现我们所说的流水灯效果。4 个 LED 的具体位置如下图中蓝色区域所示：



7.2 蜂鸣器实验

7.2.1 蜂鸣器的简介

什么是蜂鸣器呢？蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中用作发声器件。

1) 蜂鸣器的详细分析

蜂鸣器可以分为：电磁式和电压式。

压电式蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成。它是以压电陶瓷的压电效应，来带动金属片的振动而发声；

电磁式的蜂鸣器，则是用电磁的原理，通电时将金属振动膜吸下，不通电时依振动膜的弹力弹回。

故压电式蜂鸣器是以方波来驱动，电磁式是 1/2 方波驱动，压电式蜂鸣器需要比较高的电压才能有足够的音压，一般建议为 9V 以上。压电的有些规格，可以达到 120dB 以上，较大尺寸的也很容易达到 100dB。电磁式蜂鸣器：用 1.5V 就可以发出 85dB 以上的音压了，唯消耗电流会大大的高于

压电式蜂鸣器，而在相同的尺寸时，电磁式的蜂鸣器，响应频率可以做的比较低；电磁式蜂鸣器的音压一般最多到 90dB。机械式蜂鸣器是电磁式蜂鸣器中的一个小类别。

蜂鸣器又分为有源蜂鸣器和无源蜂鸣器。这里讲的“源”，指的是：振荡源。

有源蜂鸣器，内部有振荡、驱动电路，加电源就可以响，优点是用起来省事；缺点是频率固定了，就只一个单音。

无源的蜂鸣器与喇叭一样，需要加上交变的音频电压才能发声，也可以发出不同频率的声音。不过，蜂鸣器的声音反正是不好听的，所以经常是加上方波，而不是加正弦。

2) 蜂鸣器和喇叭的对比

蜂鸣器和喇叭主要区别如下：

1、蜂鸣器一般是高阻，直流电阻无限大，交流阻抗也很大，窄带发声器件，通常由压电陶瓷片发声。需要较大的电压来驱动，但电流很小，几 mA 就可以了。功率也很小。

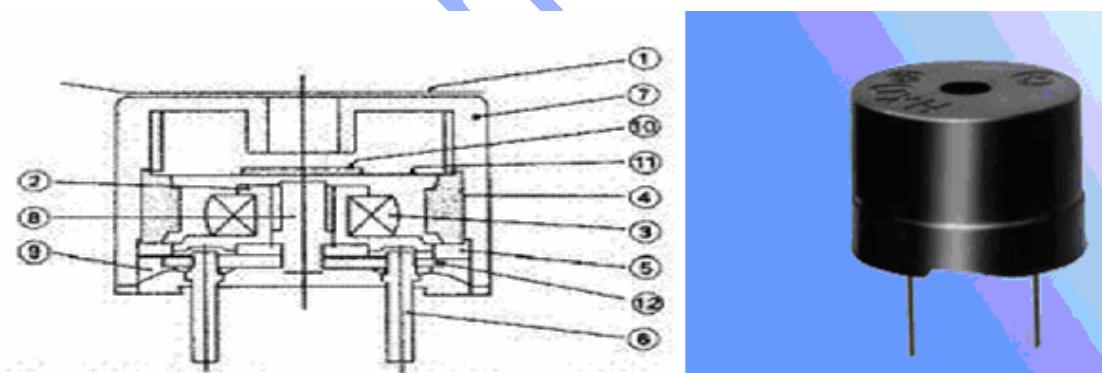
2、喇叭则是低阻，由线圈、磁铁、振膜及外壳组成。直流电阻几乎是 0，交流阻抗一般几欧到十几欧。宽频发声器件，通常由利用线圈的电磁力推动膜片发声。

3、有源蜂鸣器只能用固定电压驱动，发生频率出厂时固定了；而喇叭通过驱动器可以发出各种声音。可讲究音质，我们的音响、MP3、耳机、手机上都是用的喇叭，声音频率是可以改变的。一些报警器只能发出“滴 滴”声响的一般都是蜂鸣器，没什么音质可言。

在实际应用中我们会区分，蜂鸣器是有源还是无源，电磁式还是电压式？我们神舟 51 开发板提供了电磁式无源蜂鸣器，下面我们会详细分析。

7.2.2 蜂鸣器的结构

一般的无源蜂鸣器的结构图，如下：



1: 防水贴纸 2: 线轴 3: 线圈 4: 磁铁 5: 底座 6: 引脚 7: 外壳 8: 铁蕊 9: 封胶 10: 小铁片 11: 振动膜 12: 电路板

蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。接通电源后，外接的振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。

因此要使得这种蜂鸣器发出声音，必须在外部给它接一个震荡发生器。

7.2.3 意义与作用

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、报
嵌入式专业技术论坛 (www.armjishu.com) 出品 第 290 页，共 900 页

警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。用于产品的声音提醒或者告警等。

本实验将介绍，如何通过一个GPIO管脚控制有源直流蜂鸣器。该实验与流水灯实验类似，都是学习如何控制STM32的IO口输出。（有源蜂鸣器直接接上额定电源(新的蜂鸣器在标签上都有注明)就可连续发声;而无源蜂鸣器则和电磁扬声器一样，需要接在音频输出电路中(交流信号)才能发声。）

7.2.4 实验原理

前面流水灯的实验已经介绍过STM32处理器GPIO接口，如何配置它的模式，时钟速率等，在这一节就不再重复讲解GPIO接口的使用。

当蜂鸣器两端的电压大于4V，典型值为5V时，蜂鸣器就会发出固定频率的声音。

当控制蜂鸣器的GPIO管脚的管脚输出高电平时，关闭蜂鸣器，当GPI管脚输出低电平时，蜂鸣器发声鸣响。

7.2.5 硬件设计

在神舟III号STM32开发板中，提供了一个蜂鸣器，器鸣器连接到了处理器的PB10管脚，由处理器的PB10管脚控制，当处理器的PB10管脚输出低电平时蜂鸣器开始鸣响，反之处理器的PB10管脚输出高电平或OD开漏时蜂鸣器停止鸣响，。

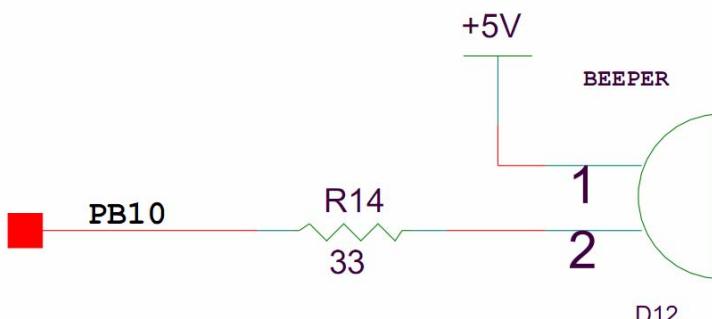
图中PB10的电阻R14的作用为限流作用，防止电流蜂鸣器工作时电流过大，损坏处理器的管脚。查阅STM32F103处理器可知，STM32处理器IO口能够承受的电流为25mA，如下表所示：

| | | |
|----------|---|------|
| I_{IO} | Output current sunk by any I/O and control pin | 25 |
| | Output current source by any I/Os and control pin | - 25 |

而蜂鸣器的的额定工作电流为30mA，因此在此处串一个电阻，保证工作电流即满足蜂鸣器的要求，又不至于太大而损坏处理器的GPIO管脚。

其原理图如下：

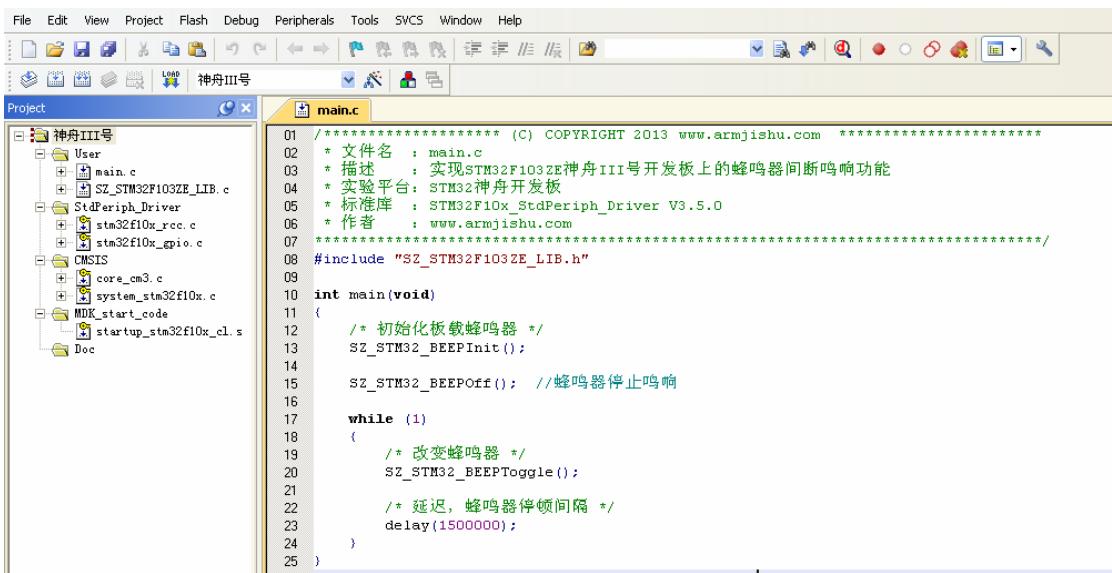
图中R14实际安装22欧姆电阻使蜂鸣器的声音柔和。



图表 5 蜂鸣器电路

7.2.6 软件设计

进入例程的文件夹，然后打开\Project\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```

int main(void)
{
    /* 初始化板载蜂鸣器 */
    SZ_STM32_BEEPInit();

    SZ_STM32_BEEPOff(); //蜂鸣器停止鸣响

    while (1)
    {
        /* 改变蜂鸣器 */
        SZ_STM32_BEEToggle();

        /* 延迟，蜂鸣器停顿间隔 */
        delay(1500000);
    }
}

```

代码分析1：在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了时钟，这里的代码是汇编的，大家可以了解一下就可以，这个文件里的代码是ST官方定制好了的，不需要我们修改，我们只需要知道在这里有这个时钟调用的函数就可以。

```

; Reset handler
Reset_Handler PROC
    EXPORT Reset_Handler [WEAK]
    IMPORT SystemInit
    IMPORT __main
    LDR R0, =SystemInit
    BLX R0
    LDR R0, =__main
    BX R0
ENDP

```

代码分析2：那么在哪里设置的初始化成72MHZ的主频呢？在system_stm32f10x.c文件中，有个宏定义如下，设置了SYSCLK_FREQ_72MHz之后，会在SystemInit()函数里进行调用

```

#ifndef STM32F10X_LD_VL || (defined STM32F10X_MD_VL) || (defined STM32F10X_HD_VL)
/* #define SYSCLK_FREQ_HSE      HSE_VALUE */
#define SYSCLK_FREQ_24MHz 24000000
#else
/* #define SYSCLK_FREQ_HSE      HSE_VALUE */
/* #define SYSCLK_FREQ_24MHz 24000000 */
/* #define SYSCLK_FREQ_36MHz 36000000 */
/* #define SYSCLK_FREQ_48MHz 48000000 */
/* #define SYSCLK_FREQ_56MHz 56000000 */
#define SYSCLK_FREQ 72MHz 72000000

```

进入到SystemInit()函数中→调用SetSysClock()→调用SetSysClockTo72()真正去设置系统为72M

```

static void SetSysClock(void)
{
#ifdef SYSCLK_FREQ_HSE
    SetSysClockToHSE();
#elif defined SYSCLK_FREQ_24MHz
    SetSysClockTo24();
#elif defined SYSCLK_FREQ_36MHz
    SetSysClockTo36();
#elif defined SYSCLK_FREQ_48MHz
    SetSysClockTo48();
#elif defined SYSCLK_FREQ_56MHz
    SetSysClockTo56();
#elif defined SYSCLK_FREQ_72MHz
    SetSysClockTo72();
#endif
}

```

代码分析3：SetSysClockTo72 ()这里实际上与寄存器版本实现的时钟例程是一样的原理，只是这里用的是库函数的方式来实现的。

```

static void SetSysClockTo72(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;

    /* SYSCLK, HCLK, PCLK2 and PCLK1 configuration -----
    /* Enable HSE */
    RCC->CR |= ((uint32_t)RCC_CR_HSEON);

    /* Wait till HSE is ready and if Time out is reached exit */
    do
    {
        HSEStatus = RCC->CR & RCC_CR_HSERDY;
        StartUpCounter++;
    } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));

    if ((RCC->CR & RCC_CR_HSERDY) != RESET)
    {
        HSEStatus = (uint32_t)0x01;
    }
}

```

代码分析4：神舟III号BEEPER蜂鸣器使用的GPIO接口定义：

```

/*蜂鸣器管脚定义*/
#define BEEPER_PIN          GPIO_Pin_10           /*蜂鸣器使用的GPIO管脚*/
#define GPIO_BEEPER         GPIOB                /*神舟III号蜂鸣器使用的GPIO组*/
#define BUZZER_GPIO_CLK     RCC_APB2Periph_GPIOB /*蜂鸣器使用的GPIO时钟*/

```

代码分析 5: SZ_STM32_BEEPInit() 函数初始化蜂鸣器的 GPIO 管脚和配置模式

```
/*蜂鸣器初始化函数*/
void SZ_STM32_BEEPInit(void)
{
    /*使能蜂鸣器使用的GPIO时钟*/
    RCC_APB2PeriphClockCmd(BUZZER_GPIO_CLK, ENABLE);

    /*初始化蜂鸣器使用的GPIO管脚*/
    GPIO_InitStructure.GPIO_Pin = BEEPER_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_BEEPER, &GPIO_InitStructure);
}
```

代码分析 6: 调用 void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin) 函数指定管脚输出高电平, 使蜂鸣器停止鸣响, 通过 **GPIOx_BSRR** 寄存器控制的。

```
GPIO_SetBits(GPIO_BEEPER, BEEPER_PIN); /*关闭蜂鸣器*/

void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Pin));

    GPIOx->BSRR = GPIO_Pin;
}
```

代码分析 7: 调用 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin) 函数指定管脚输出低电平, 使蜂鸣器鸣响, 通过 **GPIOx_BRR** 寄存器控制的。

```
GPIO_ResetBits(GPIO_BEEPER, BEEPER_PIN); /*开启蜂鸣器*/

void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Pin));

    GPIOx->BRR = GPIO_Pin;
}
```

代码分析 8: 在 while 循环中, 加上延时, 使得蜂鸣器发出滴滴的叫声

```
while (1)
{
    /* 改变蜂鸣器 */
    SZ_STM32_BEEToggle();

    /* 延迟, 蜂鸣器停顿间隔 */
    delay(1500000);
}
```

7.2.7 下载与验证

如果使用JLINK下载固件, 请按[3.3如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作
嵌入式专业技术论坛 (www.armjishu.com) 出品 第 294 页, 共 900 页

作。

如果使用串口下载固件，请按[3.4如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.8如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

7.2.8 实验现象

将固件下载在神舟III号STM32开发板后，可以听到看到的蜂鸣器发出间断的鸣响了，蜂鸣器的位置如下图红色框所示。



到这里，我们就完成了蜂鸣器实验的讲解，这个例程只是简单的实现蜂鸣器的鸣响和关闭控制，用蜂鸣器来唱歌，大家相信吗？感兴趣的朋友赶紧去尝试吧！

7.3 按键检测实验

7.3.1 单片机检测小弹性按键的原理

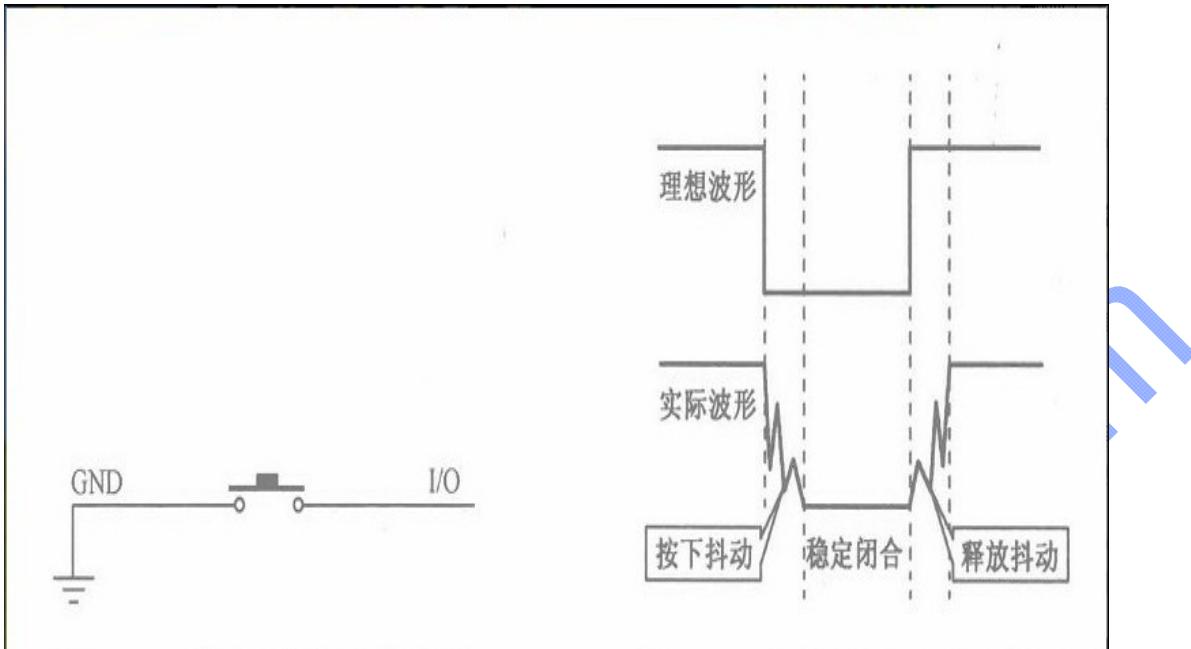
单片机检测按键的原理是：单片机的I/O口既可作为输出也可作为输入使用，当检测按键时用的是它的输入功能，我们把按键的一端接地，另一端与单片机的某个I/O口相连，开始时先给该I/O口赋一高电平，然后让单片机不断地检测该I/O口是否变为低电平，当按键闭合时，即相当于该I/O口通过按键与地相连，变成低电平，程序一旦检测到I/O口变为低电平则说明按键被按下，然后执行相应的指令。

去抖概念：（分为软件去抖和硬件去抖）

按键是机械器件，按下或者松开时有固定的机械抖动。

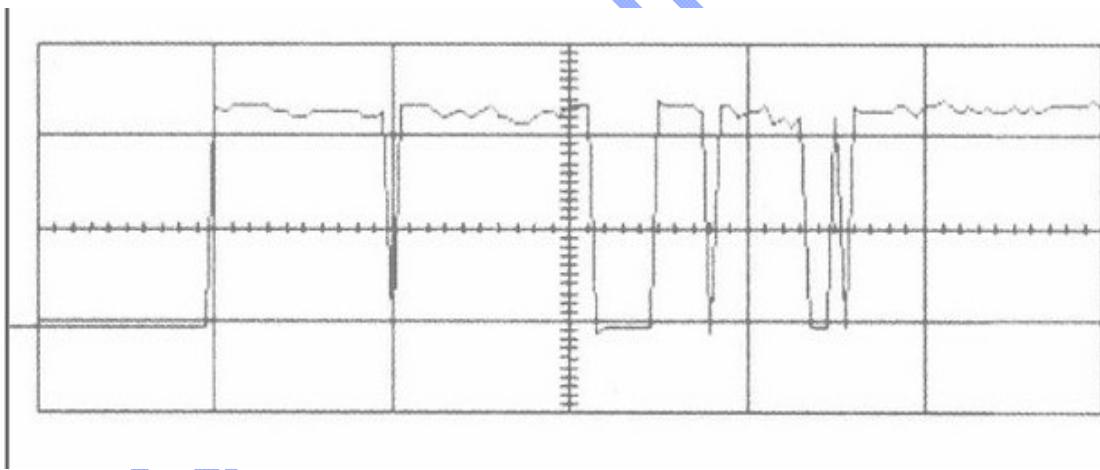
什么是机械抖动？通常的按键开关为机械弹性开关，由于机械触点的弹性作用，一个按键开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开。因而在闭合及断开的瞬间均伴随有一连串的抖动。抖动时间的长短由按键的机械特性决定，一般为5ms~10ms。这是一个很重要的时间参数，在很多场合都要用到；实际上只进行一次按键操作，但有可能执行了多次按键结果，这就是抖动造成的，所以大多数产品实际使用中都使用了按键去抖功能。

按键的连接方法和按键在被按下时其触点电压变化过程下图：



这个抖动时间虽然很短，不同按键抖动不同，但对应单片机来说，很轻松就能检测到，单片机的运行的速度是微秒 us 级别。

用示波器跟踪一个小的按钮开关在闭合时的抖动现象，得到如下图的波形。



观察波形可以帮助我们对抖动现象有一个直观的了解。水平轴2ms/DIII，抖动间隙大约为10ms，在达到稳定状态前一共有6次变化，频率随时间升高。

硬件去抖最简单的就是按键两端并联电容，容量根据实验而定。当然也有专用的去抖动芯片。

软件去抖使用方便不增加硬件成本，容易调试，所以现在大都使用软件去抖。

关于软件去抖，我们在这里也详细讲解一下原理：

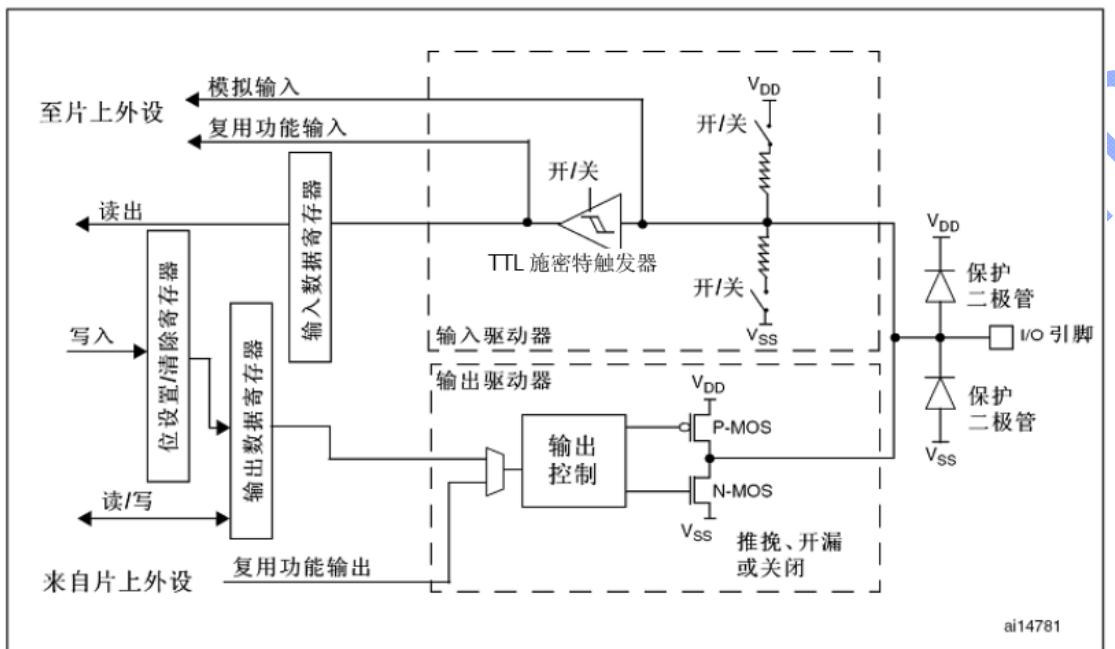
- 1、检测到按键按下后进行 5~10ms 延时，用于跳过这个抖动区域
- 2、延时后再检测按键状态，如果没有按下表明是抖动或者干扰造成，如果仍旧按下，可以认为是真正的按下。并进行对应的操作。

3、同样按键释放后也要进行去抖延时，延时后检测按键是否真正松开。

7.3.2 GPIO的8种工作模式

在初始化 GPIO 的时候，根据我们的使用要求，必须把 GPIO 设置为相应的模式。如 LED 例程中的 GPIO 引脚如果配置为 模拟输入模式 是必然会导致 错误 的。

我们配合 GPIO 结构图，来看看 GPIO 的 8 种模式及其应用场合：



图的最右端为 I/O 引脚，左端的器件位于芯片内部。I/O 引脚并联了两个用于保护的二极管。本图从 ST 提供的参考手册截取。

● 四种输入模式

结构图的上半部分为输入模式结构。

接下来就遇到了两个开关和电阻，与 V_{DD} 相连的为 上拉电阻，与 V_{SS} 相连的为 下拉电阻。再连接到 施密特触发器 就把电压信号转化为 0、1 的数字信号存储在 输入数据寄存器(IDR)。我们可以通过设置 配置寄存器(CRL、CRH)， 控制这两个开关，于是就可以得到 GPIO 的 上拉输入(GPIO_Mode_IPU) 和 下拉输入模式(GPIO_Mode_IPD) 了。

从它的结构我们就可以理解，若 GPIO 引脚配置为上拉输入模式，在 默认状态下 (GPIO 引脚无输入)，读取得的 GPIO 引脚数据为 1，高电平。而下拉模式则相反，在默认状态下其引脚数据为 0，低电平。

而 STM32 的 浮空输入模式(GPIO_Mode_IN_FLOATING) 在芯片内部既没有接上拉，也没有接下拉电阻，经由触发器输入。配置成这个模式直接用电压表测量其引脚电压为 1 点几伏，这是个不确定值。由于其输入阻抗较大，一般把这种模式用于标准的通讯协议如 I2C、USART 的接收端。

模拟输入模式(GPIO_Mode_AIN) 则关闭了施密特触发器，不接上、下拉电阻，经由另一线路把电压信号传送到片上外设模块。如传送至给 ADC 模块，由 ADC 采集电压信号。所以 使用 ADC 外设 的时候，必须设置为 模拟输入模式。

● 四种输出模式

结构图的下半部分为输出模式结构。

线路经过一个由 P-MOS 和 N-MOS 管组成的单元电路。而所谓推挽输出模式，则是根据其工作方式来命名的。在输出高电平时，P-MOS 导通，低电平时，N-MOS 管导通。两个管子轮流导通，一个负责灌电流，一个负责拉电流，使其负载能力和开关速度都比普通的方式有很大的提高。推挽输出的供电平为 0 伏，高电平为 3.3 伏。

在开漏输出模式时，如果我们控制输出为 0，低电平，则使 N-MOS 管导通，使输出接地，若控制输出为 1(无法直接输出高电平)，则既不输出高电平，也不输出低电平，为高阻态。为正常使用时必须在外部接上一个上拉电阻。它具“线与”特性，即很多个开漏模式引脚连接到一起时，只有当所有引脚都输出高阻态，才由上拉电阻提供高电平，此高电平的电压为外部上拉电阻所接的电源的电压。若其中一个引脚为低电平，那线路就相当于短路接地，使得整条线路都为低电平，0 伏。

STM32 的 GPIO 输出模式就分为普通推挽输出(GPIO_Mode_Out_PP)、普通开漏输出(GPIO_Mode_Out_OD)及复用推挽输出(GPIO_Mode_AF_PP)、复用开漏输出(GPIO_Mode_AF_OD)。

普通推挽输出模式一般应用在输出电平为 0 和 3.3 伏的场合。而普通开漏输出一般应用在电平不匹配的场合，如需要输出 5 伏的高电平，就需要在外部接一个上拉电阻，电源为 5 伏，把 GPIO 设置为开漏模式，当输出高阻态时，由上拉电阻和电源向外输出 5 伏的电平。

对于相应的复用模式，则是根据 GPIO 的复用功能来选择的，如 GPIO 的引脚用作串口的输出，则使用复用推挽输出模式。如果用在 IC、SMBUS 这些需要线与功能的复用场合，就使用复用开漏模式。其它不同的复用场合的复用模式引脚配置将在具体的例子中讲解。

在使用任何一种开漏模式，都需要接上拉电阻。

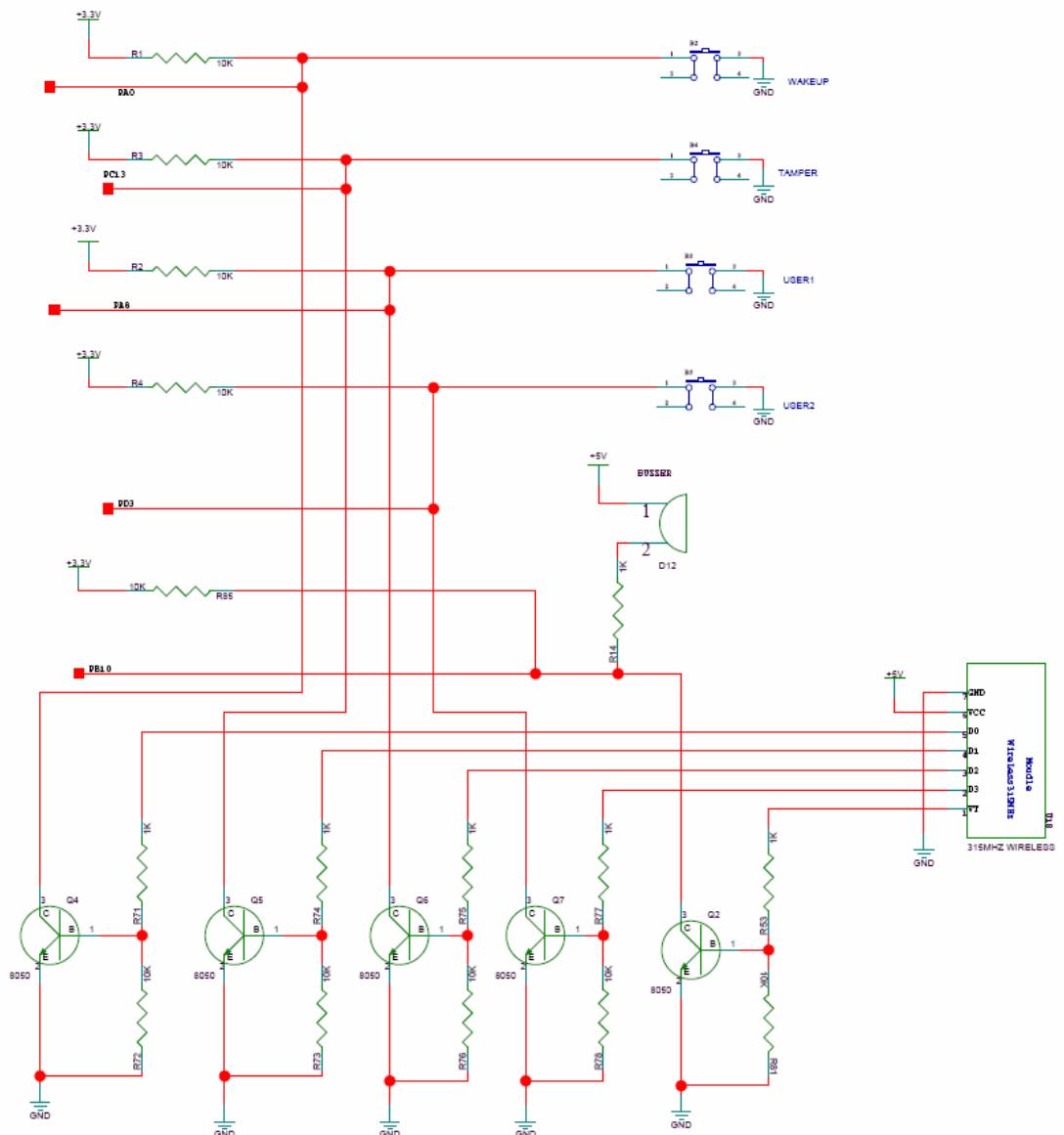
● 按键实验分析

了解了 GPIO 的 8 种工作模式之后，立即进行一下小测。如果采用以下的电路，我们的按键 GPIO 端口应该如何进行配置？

有两个方案可以选择，一是采用上拉输入模式，因为按键在没按下的时候，是默认为高电平的，采用内部上拉模式正好符合这个要求。

第二个方案是直接采用浮空输入模式，因为按照这个硬件电路图，在芯片外部接了上拉电阻，其实就没必要再配置成内部上拉输入模式了，因为在外部上拉与内部上拉效果都是一样的。本实验中，我们将按键都配置为浮空输入模式。

STM32 神舟 III 号开发板按键硬件原理图：



神舟III号STM32开发板总共有4个功能按键，分别是WAKEUP按键和TAMPER按键及两个用于自定义功能按键，在不使用第二功能的情况下，这四个按键都可以作为通用的按键，由用户自定义其功能。这四个按键分别与PD3、PA8、PC13和PA0四个GPIO管脚连接，当按键按下时，对应的GPIO管脚为低电平，反之，当没有按键按下时，对应的GPIO管脚为高电平。其中PA0 (STM32的WKUP引脚)可以作为WAKEUP功能，它除了可以用作普通输入按键外，还可以用作STM32的唤醒输入。PC13可以实现备份区寄存器的入侵功能。本实验中所有的按键均作为普通IO使用。

7.3.3 代码分析

本实验中我们引入了一对新的文件：key.c 和 key.h。我们分析一下 key.c 文件的内容。文件中主要包含了两个函数：SZ_STM32_KEYInit() 函数和 SZ_STM32_KEYScan() 函数。我们代码分析 1：先看一下 SZ_STM32_KEYInit() 函数：

```

void SZ_STM32_KEYInit()
{
    GPIO_InitTypeDef GPIO_Initstructure;

    /* 使能KEY按键对应的Clock时钟 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                           RCC_APB2Periph_GPIOC | RCC_APB2Periph_AFIO, ENABLE);

    /* 初始化KEY按键的GPIO管脚PA0、PB10、PC4、PC13，配置为浮空输入模式 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_Init(GPIOA, &GPIO_Initstructure);

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_Init(GPIOB, &GPIO_Initstructure);

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_13;
    GPIO_Init(GPIOC, &GPIO_Initstructure);
}

```

这个函数是按键初始化函数，这个和 LED 灯的初始化函数类似。首先使能按键对应的 GPIO 的 Clock 时钟。配置 4 个按键对应 GPIO 管脚 PA0、PD3、PA8 和 PC13 为浮空输入模式。调用库函数 GPIO_Init() 完成这 4 个 GPIO 管脚的初始化。

代码分析 2：SZ_STM32_KEYScan() 函数

```

/* 获取KEY按键的输入电平状态，按键按下时为低电平0 */
if(0 == GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_3))
{
    /* 延迟去抖 */
    delay(150000);
    if(0 == GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_3))
    {
        return 1;
    }
}

```

通过调用函数 GPIO_ReadInputDataBit() 判断，按键的 GPIO 管脚是否被拉低。函数 GPIO_ReadInputDataBit() 是 ST 提供的库函数。在帮助文档中有对应的函数说明。实际上这个函数是通过输入数据寄存器 GPIOx_IDR 判断电平的高低。

```

uint8_t GPIO_ReadInputDataBit ( GPIO_TypeDef * GPIOx,
                               uint16_t      GPIO_Pin
)

```

Reads the specified input port pin.

Parameters:

GPIOx,: where x can be (A..G) to select the GPIO peripheral.
 GPIO_Pin,: specifies the port bit to read. This parameter can be GPIO_Pin_x where x can be (0..15).

代码分析 3：主函数

```
int main(void)
{
    uint32_t keynum;

    /* 初始化板载LED指示灯 */
    LED_Init();

    /* 初始化板载按键为GPIO模式(非中断) */
    SZ_STM32_KEYInit();

    /* Infinite loop 主循环 */
    while (1)
    {
        /* 按键按下时为低电平，如果按键按下则改变指示灯状态 */
        keynum = SZ_STM32_KEYScan();
        if(keynum != 0)
        {
            switch(keynum)
            {
                case 1:
                    SZ_STM32_LED1Toggle();
                    break;
            }
        }
    }
}
```

首先调用函数 LED_Init(), 初始化 LED 灯的 GPIO 管脚。然后，调用函数 SZ_STM32_KEYInit() 初始化，按键的 GPIO 管脚。再调用函数 SZ_STM32_KEYScan() 判断那个按键被按下了。如果有按键按下，那么通过调用对应的函数使对应的 LED 灯的状态反转。

7.3.1 下载与验证

如果使用JLINK下载固件，请按[如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作。

如果使用串口下载固件，请按[如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

7.3.4 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，重新打开电源；神舟 III 号开发板上的 4 个 LED 灯同时闪一下，然后都亮；如下图显示



在没有按键按下时，4个LED灯都亮；有按键按下，相关的LED灯随之发生变化，具体实现现象如下表所示：

| LED 灯 | 按键 | 说明 |
|-------|-----------|--------------------|
| DS1 | USER1 按键 | 按下按键 1， DS1 灯取反状态。 |
| DS2 | USER2 按键 | 按下按键 2， DS2 灯取反状态。 |
| DS3 | TAMPER 按键 | 按下按键 3， DS3 灯取反状态。 |
| DS4 | WAKEUP 按键 | 按下按键 4， DS4 灯取反状态。 |

7.4 BitBand按键扫描检测实验

7.4.1 什么是位带操作

还记得 51 单片机吗？单片机 51 中也有位的操作，以一位（BIT）为数据对象的操作；例如 51 单片机可以简单的将 P1 端口的第 2 位独立操作，P1.2=0 或者 P1.2=1，就是这样把 P1 口的第三个脚（bit2）置 0（输出低电平）或者置 1（输出高电平）。

而现在 STM32 的位段、位带别名区这些就是为了实现这样的功能，可以在 SRAM、I/O 外设空间实现对这些区域的某一位的单独直接操作。

7.4.2 为什么要用位带操作

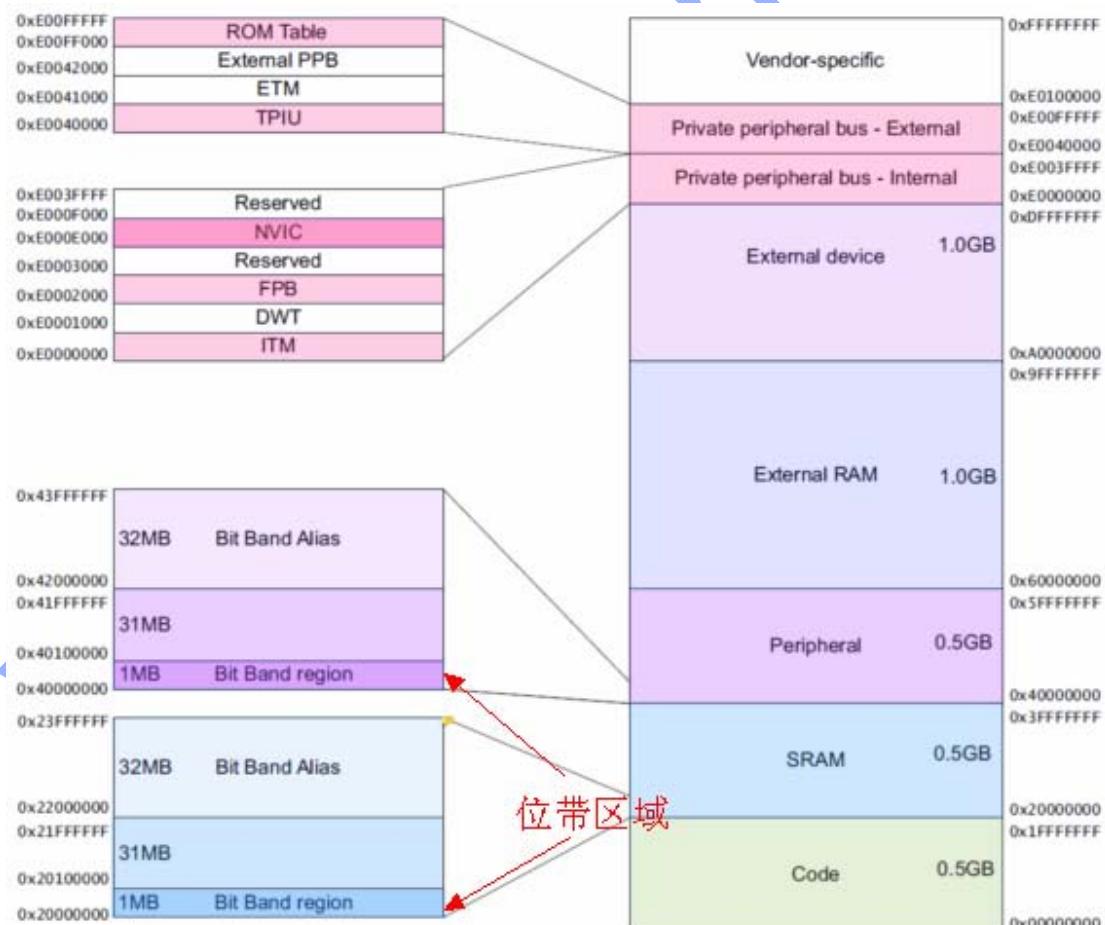
那么 51 单片机中间不是有位的操作吗，而 STM32 为什么要提出位带的操作呢？首先，这里不嵌入式专业技术论坛（www.armjishu.com）出品 第 302 页，共 900 页

得不提一个事情就是 STM32 的内部区域访问只能是 32 位的字，不能是字节或者半字，也更不可能只访问一个 bit，要访问就要一次访问 32 个 bit，这部分 STM32 在神舟开发板手册的 GPIO 章节中提到过；而 51 单片机里一个 bit（一个字节等于 8 个 bit，一个字是 32 个 bit）。而 STM32 中一个寄存器一般是 32 个 bit 的，每一个位都是 1 个 bit，那么如何可以访问一个寄存器中的 1 个位呢？就是 1 个 bit，把一次访问 32bit 的 STM32 经过一系列转换达到只访问其中的一个 bit，所以 STM32 使用一种新型的方式来解决这个问题，设计一个办法来解决用一次访问 32bit 的这样的操作达到 51 单片机那种只访问一个 bit 的效果。

7.4.3 如何设计和实现位带操作的

从编程者这个角度来说，我们操作的对象是一个一个的 bit 位，而对于 STM32 来说，它内部只能是 32 位 bit 每次的访问。如果要实现这个技术，必须要做一个映射，也就是从 1 个 bit 映射到 32 个 bit，就是用 STM32 内部的一次访问（32 个 bit）来代表编程者认为的 1 个 bit。

那 STM32 内部是如何解决的呢？它是在支持位带操作的地方，取个别名区空间，而这个别名区空间可以让一次 32 位来进行访问，对这个别名进行操作就相当于对 SRAM 或者 I/O 存储空间中的位（1 个寄存器里的位就是 1 个 bit，1 个 bit 最后对应别名区空间的 32 位，因为 STM32 芯片内部只能是 32 位去访问）进行操作。



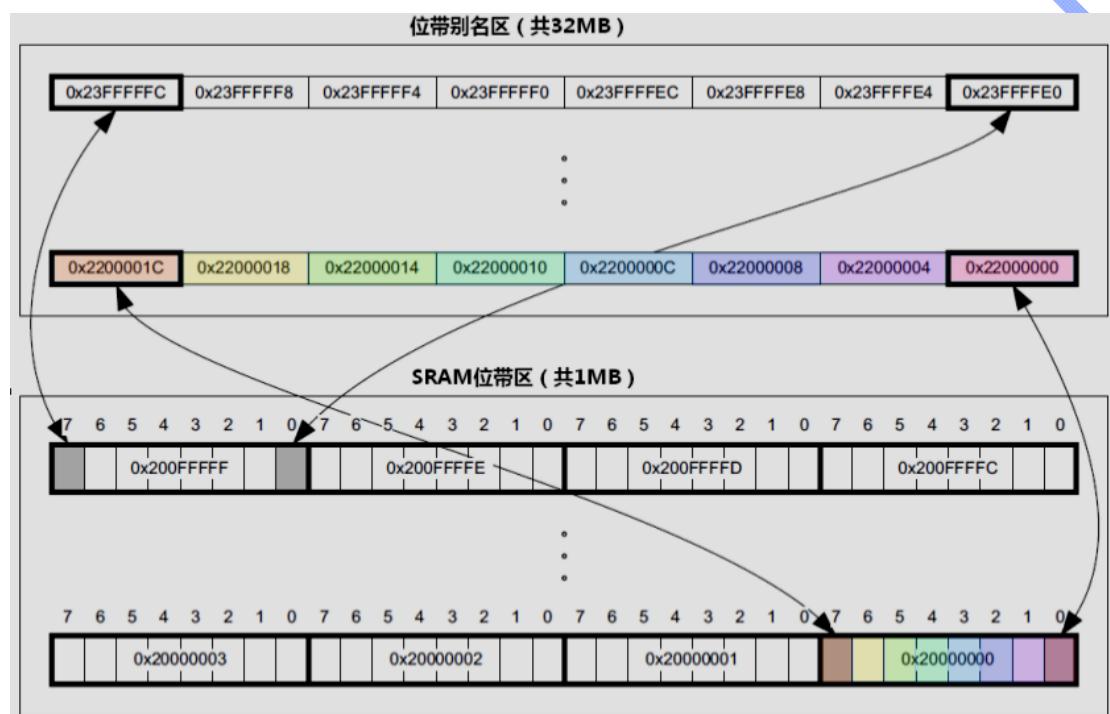
这样呢，1MB SRAM 就可以 32M 个对应别名区空间，就是 1 位膨胀到 32 位（1bit 变为 1 个字）；我们对这个别名区空间开始的某一字操作，置 0 或置 1，就等于它映射的 SRAM 或 I/O 相应的某地址的某一位的操作。

7.4.4 STM32中位带操作的具体部署情况

支持位带操作的两个内存区的范围是：

| 序号 | 支持位带操作的两个内存区的范围 | 对应的别名区空间范围 |
|----|--|--|
| 1 | SRAM 区中的最低 1MB： 0x2000_0000-0x200F_FFFF | SRAM 所对应的别名区 32MB 空间： 0x2200_0000-0x23FF_FFFF |
| 2 | 片上外设区中的最低 1MB： 0x4000_0000-0x400F_FFFF | 片上外社区所对应的别名区 32MB 空间： 0x4200_0000-0x43FF_FFFF |

下面是内部空间映射图：



例如：SRAM 区中的最低 1MB 空间中的 0x2000_0000 的 8 个 bit，分别对应如下：

| 地址 | 对应的 bit 位 | 别名空间 | Bit 对应的别名空间 |
|-------------|-----------|-------------|--------------------------|
| 0x2000_0000 | Bit1 | 0x2200_0000 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit2 | 0x2200_0004 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit3 | 0x2200_0008 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit4 | 0x2200_000C | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit5 | 0x2200_0010 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit6 | 0x2200_0014 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit7 | 0x2200_0018 | 跨度一个字 = 4 个字节 = 32 个 bit |
| | Bit8 | 0x2200_001C | 跨度一个字 = 4 个字节 = 32 个 bit |

可以看到上表和上图，0x2000_0000 中的一个 bit 位对应了别名区的一个 32 位的字，也就是说嵌入式专业技术论坛（www.armjishu.com）出品 第 304 页，共 900 页

STM32 芯片的内部寄存器的任意一个位，都其实对应的是别名区的 32 个位。

7.4.5 如何用代码来实现位带操作

在 STM32 中，一个寄存器是 32 位的，32 个 bit 中的任意其中一个 bit 所对应的别名空间到底该如何访问呢？首先分为两种情况，一种是在 SRAM，一种是在 FLASH 中，两个别名空间的起地位置是不同的，SRAM 是从 0x2200_0000 开始，而 FLASH 是从 0x4200_0000 开始。

假如在 SRAM 中的一个寄存器的地址是 A，访问寄存器 A 中的第 n 个 bit 位。那么该如何计算呢？我们知道 SRAM 中别名区的起始地址是 0x2200_0000 对应 SRAM 中实际寄存器地址 0x2000_0000，SRAM 中每 1 个 bit，都会对应别名区中的 32 个 bit，那么实际地址的公式应该如下：

$$0x2200_0000 + (\text{SRAM 实际寄存器地址偏移 } 0x2000_0000 \text{ 的 bit 数}) * 4$$

因为 0x2200_0000 这个地址每增加 1，实际上就是增加 8 个 bit(一个地址对应一个字节)，实际寄存器中的 1 个 bit 对应 32 个 bit，所以就乘以 4，地址本身增加 1 是 8bit，8bit 乘以 4 倍刚好是 32bit。

那么接下来“SRAM 实际寄存器地址偏移 0x2000_0000 的 bit 数”该如何计算呢？

对，用寄存器的地址减去这个基地址，然后在乘以 8 (因为一个地址对应 8 个 bit)，所以就可以得到以下的公式：

$$(A - 0x20000000) * 8$$

以上这个公式可以知道实际寄存器离基地址有多少个 bit 的距离，访问该寄存器的第 n 个 bit 位还必须加上一个 n，就变成以下的公式：

$$(A - 0x20000000) * 8 + n$$

好了，最后整理整个换算公式如下，FLASH 与 SRAM 的原理都是想通的：

$$\text{SRAM : } 0x22000000 + ((A - 0x20000000) * 8 + n) * 4$$

$$\text{FLASH : } 0x42000000 + ((A - 0x40000000) * 8 + n) * 4$$

1. 举例说明：

比如我要访问如下寄存器 GPIOB_BSRR 中的第 14bit 位 BS13，注意因为寄存器内部是从 0 开始计数到 31 截止，所以第 14bit 相当于第 13。

7.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

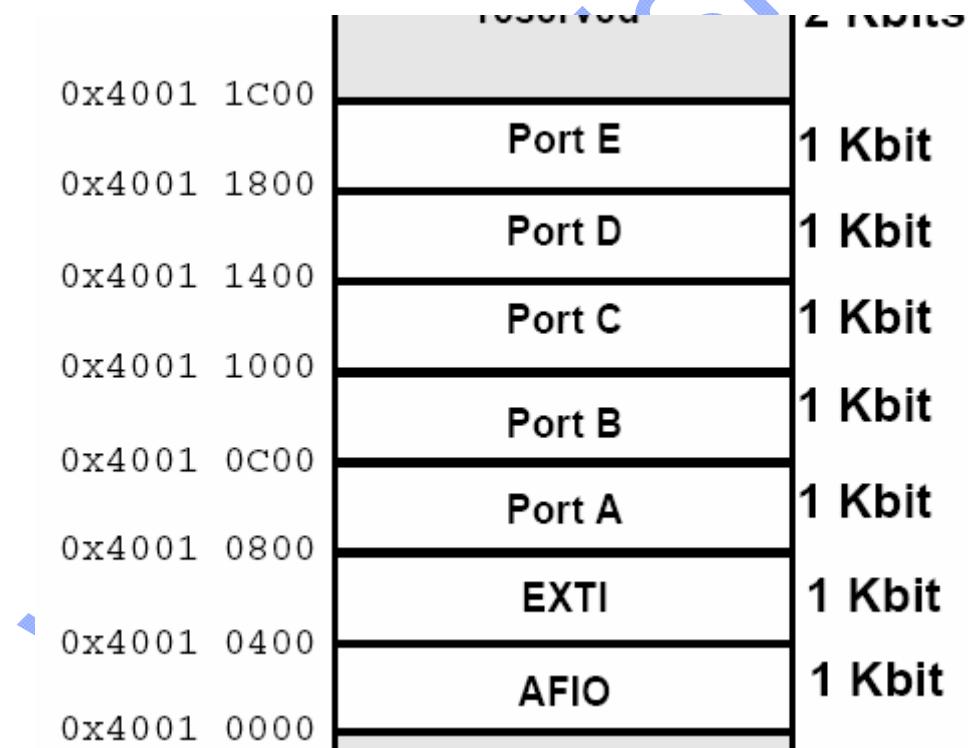
地址偏移: 0x10

复位值: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |

| | |
|--------|---|
| 位31:16 | BRy: 清除端口x的位y (y = 0...15) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。 |
| 位15:0 | BSy: 设置端口x的位y (y = 0...15) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 设置对应的ODRy位为1 |

可以从下图看到, GPIO 端口 B 的起始地址是 x04001_0C00, GPIOB_BSRR 寄存器的偏移地址是 0x10, 访问的第 14bit 位的 BS13。



那么通过公式:

$$\text{FLASH :} 0x42000000 + ((A - 0x40000000) * 8 + n) * 4$$

$$\text{换算 } 0x4200_0000 + ((0x40010c00-0x40000000) * 8 + 12) * 4 = \text{实际地址}$$

在这里我们就不具体计算了, SRAM 访问也是同理

2. 如何将理念转化成代码:

由上面几节得出，SRAM 和 FLASH 中别名区的寻址公式如下：

SRAM : $0x22000000 + ((A - 0x20000000) * 8 + n) * 4$

FLASH : $0x42000000 + ((A - 0x40000000) * 8 + n) * 4$

可以看到 $0x2200_0000$ 和 $0x4200_0000$ 都共同有个 $x200_0000$ 这个数值；乘以 8 相当于再加上外面的那个乘以 4，总共是乘以 32，而 n 是乘以 4；乘以 32 相当于是数值向左移 5 位，乘以 4 相当于向左边移 2 位。

尝试将如下公式化成

```
#define BITBAND(addr, bitnum)
```

```
((addr & 0xF0000000)+0x2000000+((addr &0xFFFF)<<5)+(bitnum<<2))
```

这样就可以用 BITBAND(addr, bitnum) 来表示寄存器里的任何一个 bit 位，大概原理讲到这里，具体使用方法通过例程来体现。

7.4.6 软件设计

进入例程的文件夹，然后打开\Project\Project.uvproj 文件

```

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Project 神舟III号 main.c
001 / ****文件名：main.c **** (C) COPYRIGHT 2013 www.armjishu.com ****
002 * 描述：实现STM32F103ZE神舟III号开发板上按键BitBand方式扫描检测控制LED功能
003 * 实验平台：STM32神舟开发板
004 * 标准库： STM32F10x_StdPeriph_Driver V3.5.0
005 * 作者： www.armjishu.com
006 ****
007 ****
008
009 #include "SZ_STM32F103ZE_LIB.h"
010
011 int main(void)
012 {
013     /*!< 在系统启动文件 (startup_stm32f10x_xx.s) 中已经调用SystemInit() 初始化了时钟,
014     所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
015     SystemInit() 函数的实现位于system_stm32f10x.c文件中。 */
016
017     /* 初始化板载LED指示灯 */
018     SZ_STM32_LEDInit(LED1);
019     SZ_STM32_LEDInit(LED2);
020     SZ_STM32_LEDInit(LED3);
021     SZ_STM32_LEDInit(LED4);
022
023     /* 延迟 */
024     delay(2000000);
025
026     /* BitBand方式1 管脚输出高电平熄灭指示灯 */
027     LED1OBB = 1; //PF.06
028     LED2OBB = 1; //PF.07
029     LED3OBB = 1; //PF.08
030     LED4OBB = 1; //PF.09

```

可以看到工程已经被打开，下面开始具体分析程序代码：

```

int main(void)
{
    /*!< 在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了时钟,
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
       SystemInit()函数的实现位于system_stm32f10x.c文件中。 */

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    /* 延迟 */
    delay(2000000);

    /* BitBand方式1 管脚输出高电平熄灭指示灯 */
    LED1OBB = 1; //PF.06
    LED2OBB = 1; //PF.07
    LED3OBB = 1; //PF.08
    LED4OBB = 1; //PF.09

    /* 延迟 */
    delay(2000000);

    /* BitBand方式1 管脚输出低电平点亮指示灯 */
    LED1OBB = 0; //PF.06
    LED2OBB = 0; //PF.07
    LED3OBB = 0; //PF.08
    LED4OBB = 0; //PF.09

```

代码分析1：在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了72MHZ时钟，最开始的例程已经对此分析过了，还有不明白的可以看下前面的例程。

代码分析2：调用SZ_STM32_LEDInit()函数初始化板载LED指示灯，前面已经有介绍

代码分析3：开始使用位带的BitBand方式，使得管脚输出高电平熄灭指示灯，这里可以看到，一个变量就可以独立操作一个GPIO管脚，这究竟是如何做到的呢？下面继续看代码分析：

```

/* BitBand方式1 管脚输出高电平熄灭指示灯 */
LED1OBB = 1; //PF.06
LED2OBB = 1; //PF.07
LED3OBB = 1; //PF.08
LED4OBB = 1; //PF.09

```

1) 首先看如下这行代码：

```
#define LED1OBB Periph_BB((uint32_t)&LED1_GPIO_PORT->ODR, LED1_PIN_NUM)
```

LED1OBB希望变成PF端口的第6个GPIO管脚，那就是PF.06，那么LED1_PIN_NUM=6，而LED1_GPIO_PORT->ODR则变成了GPIOF->ODR

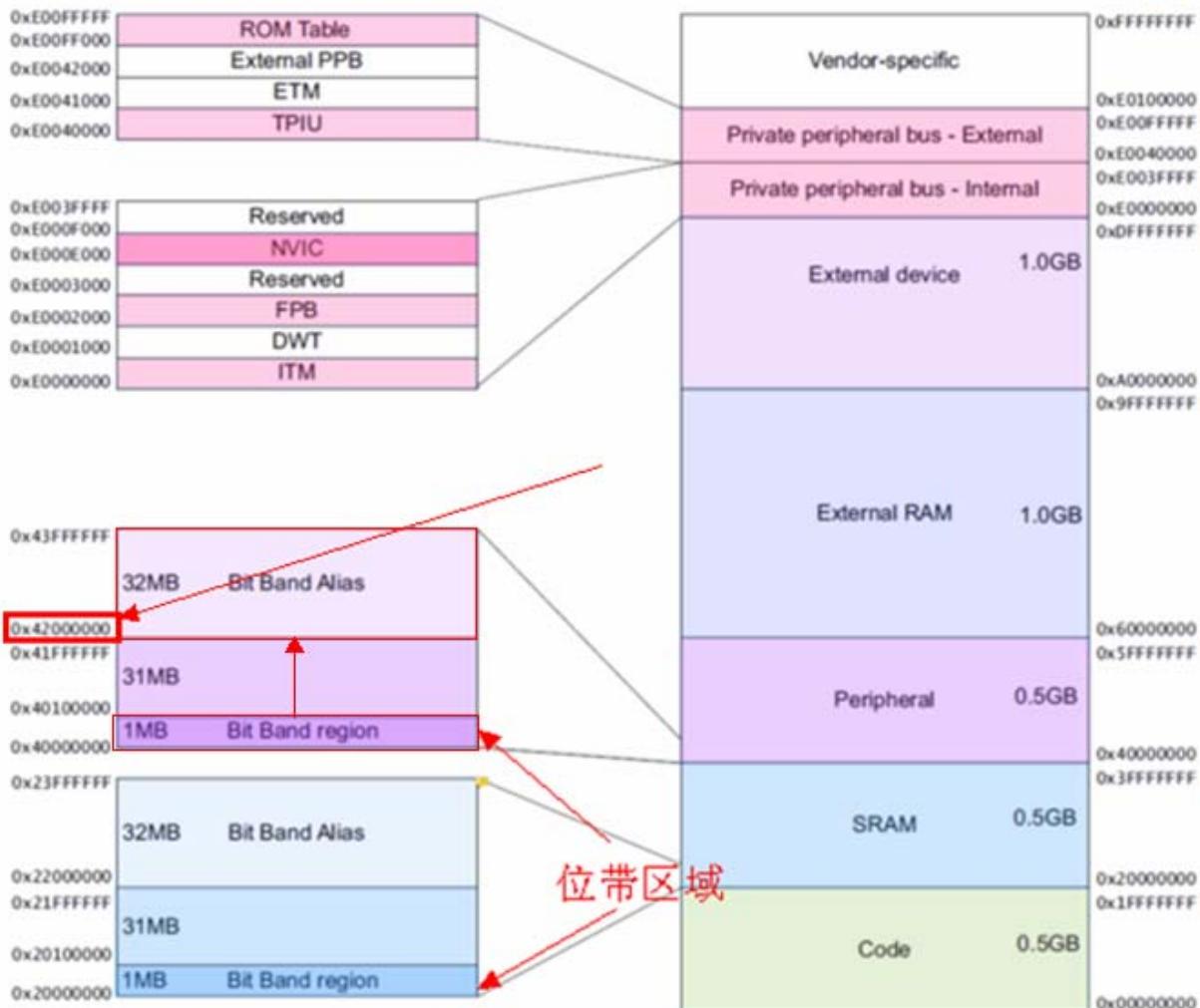
2) 继续分析一下，把PF端口和PF端口的第6管脚这两个参数传入到Periph_BB()就可以访问到PF.06的这个GPIO管脚了吗？还得分析一下Periph_BB()函数

代码分析4：开始详细分析和解析Periph_BB()函数，这里就比较复杂，如果感兴趣的朋友可以认真研究和计算，如果不感兴趣，也不会影响后面的学习，等有兴趣的时候再来研究也都可以，这是STM32神舟团队专门提供的建议：

```
#define Periph_BB(PeriphAddr, BitNumber) \
    *(_IO uint32_t *) (Periph_BB_BASE | ((PeriphAddr - Periph_BASE) << 5) | ((BitNumber) << 2))
```

1) Periph_BB_BASE是外设别名区基址

```
#define Periph_BASE      0x40000000 // 外设基地址 Peripheral
#define Periph_BB_BASE   0x42000000 // 外设别名区基址 Peripheral bitband
```



从图中可以看到从0x40000000到0x41000000这1M的Bit Band区域对应了从0x42000000~0x43FFFFFF这32MB的区域，也就是说Bit Band区域中的1个bit对应了Bit Band Alias区域32个bit的空间。

2) 在 STM32 中，一个寄存器是 32 位的，32 个 bit 中的任意其中一个 bit 所对应的别名空间到底该如何访问呢？首先分为两种情况，一种是在 SRAM，一种是在 FLASH 中，两个别名空间的起地位置是不同的，FLASH 是从 0x4200_0000 开始。

假如在 FLASH 中的一个寄存器的地址是 A，访问寄存器 A 中的第 n 个 bit 位。那么该如何计算呢？我们知道 FLASH 中别名区的起始地址是 0x4200_0000 对应 SRAM 中实际寄存器地址 0x4000_0000，FLASH 中每 1 个 bit，都会对应别名区中的 32 个 bit，那么实际地址的公式应该如下：

$$0x4200_0000 + (\text{FLASH 实际寄存器地址偏移 } 0x4000_0000 \text{ 的 bit 数}) * 4$$

因为 0x4200_0000 这个地址每增加 1，实际上就是增加 8 个 bit(一个地址对应一个字节)，实际寄存器中的 1 个 bit 对应 32 个 bit，所以就乘以 4，地址本身增加 1 是 8bit，8bit 乘以 4 倍刚好是 32bit。

那么接下来“FLASH 实际寄存器地址偏移 0x4000_0000 的 bit 数”该如何计算呢？

对，用寄存器的地址减去这个基地址，然后在乘以 8 (因为一个地址对应 8 个 bit)，所以就可以得到以下的公式：

$$(A - 0x40000000)*8$$

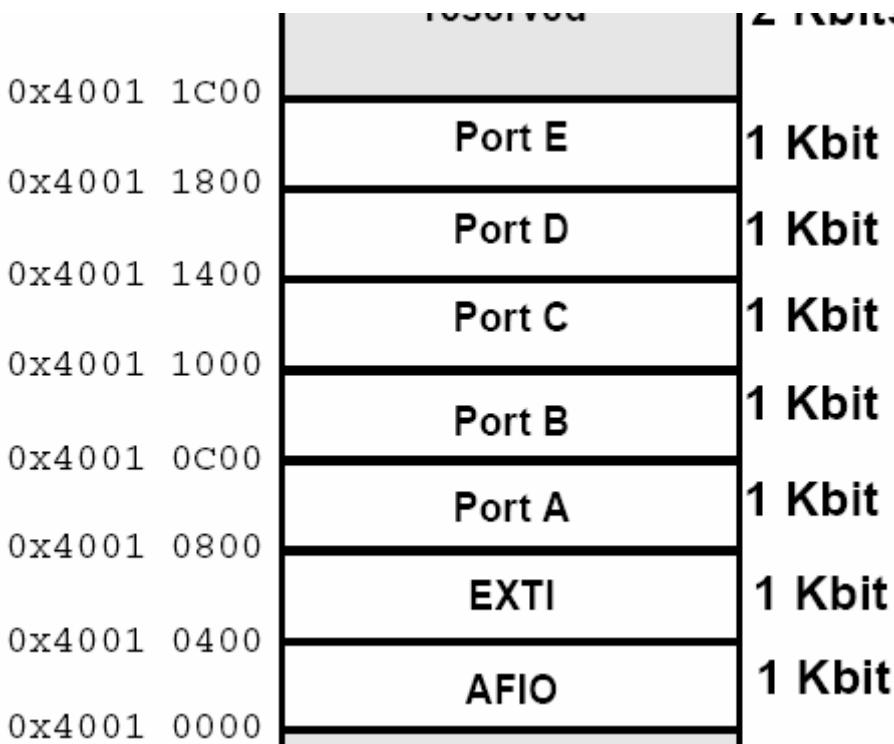
以上这个公式可以知道实际寄存器离基地址有多少个 bit 的距离，访问该寄存器的第 n 个 bit 位还必须加上一个 n, 就变成以下的公式：

$$(A - 0x40000000)*8+n = \text{实际寄存器的地址距基地址的实际 bit 数}$$

好了，最后整理整个换算公式如下，FLASH 与 SRAM 的原理都是想通的：

FLASH :0x42000000 +((A - 0x40000000)*8+n)*4 (实际寄存器的地址距基地址的实际 bit 数 * 4 个字节 (4 个字节是 32 个 bit) 再加上 bitband 的别名区的基地址，就得出实际的别名区 bitband 的地址了)

3) 可以从下图看到，GPIO 端口 F 的起始地址是 x04001_1C00，访问的是 GPIO 管脚 6



那么通过公式：

$$\text{FLASH :}0x42000000 +((A - 0x40000000)*8+n)*4$$

$$\text{换算 } 0x4200_0000 + ((0x40011C00-0x40000000)*8 + 12)*4 = \text{实际地址}$$

4) FLASH :0x42000000 +((A - 0x40000000)*8+n)*4 这个公式化成另外一种计算方式就变成：

$0x42000000 + (A - 0x40000000)*32+n*4$, 所以乘以32实际上是左移5位，乘以4实际上就是左移2位，最后3个地址相加：

```
*(_IO uint32_t *) Periph_BB_BASE | ((PeriphAddr - Periph_BASE) << 5) | ((BitNumber) << 2)
```

这三个地址，一个是最基础的地址0x42000000，一个是寄存器对应bitband区域的地址，最后一个寄存器里面的位对应的bitband区域的地址。

代码分析5：PFSetBit(6)函数就是表示设置PF.06管脚，使得管脚管脚依次输出高电平

```
#define PFSetBit(n) (PFOutBit(n)=1)
#define PFOutBit(n) Periph_BB((uint32_t)&GPIOF->ODR,n)
```

可以看到，最终还是调用了Periph_BB()函数了，在这里只需要传入所对应寄存器的地址，以及对应

的GPIO管脚号，就可以访问到最终的bitband区域。

7.4.7 下载与验证

如果使用JLINK下载固件，请按[3.2如何使用JLINK软件下载固件到神舟III号开发板小节进行操作](#)。

如果使用串口下载固件，请按[3.3如何通过串口下载一个固件到神舟III号开发板小节进行操作](#)。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作](#)。

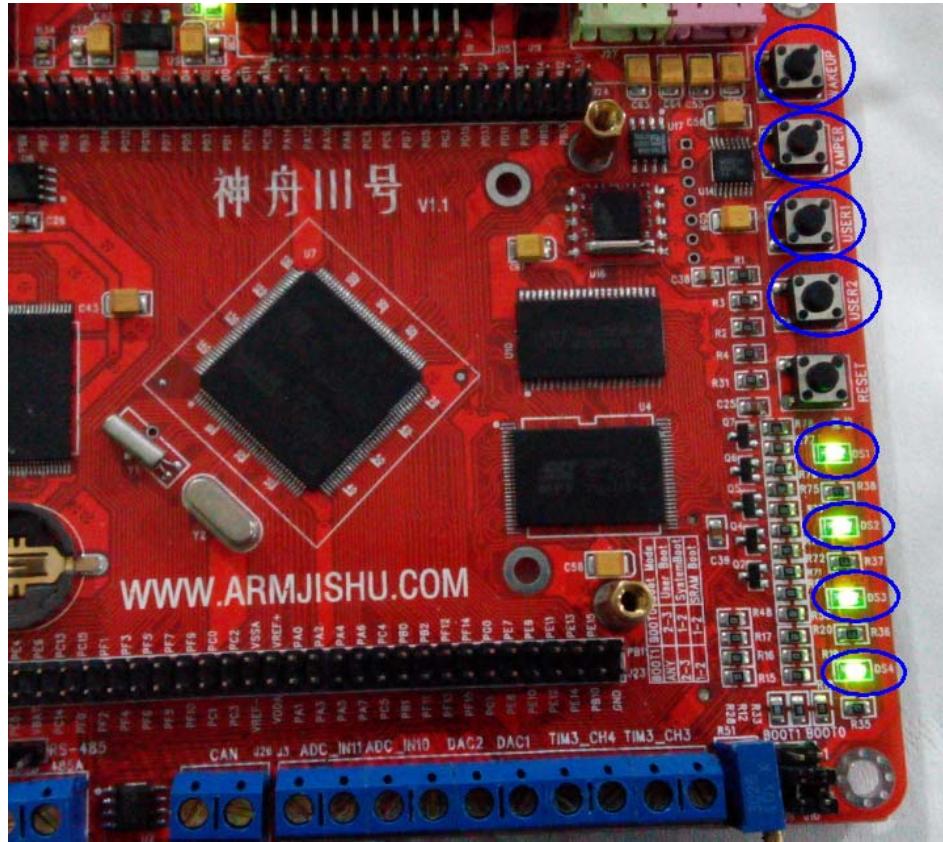
7.4.8 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，重新打开电源；神舟 III 号开发板上的 4 个 LED 灯闪烁两次，然后从上往下熄灭 LED 灯；

在没有按键按下时，4 个 LED 灯都是熄灭；有按键按下，相关的 LED 灯随之发生变化，具体实现现象如下表所示：

| LED 灯 | 按键 | 说明 |
|-------|-----------|-----------------------------------|
| DS1 | USER 1 按键 | 按一下 USER 1 对应 DS 1 灯亮，再按一下 DS 1 灭 |
| DS2 | USER2 按键 | 按一下 USER2 对应 DS2 灯亮，再按一下 DS 2 灭 |
| DS3 | TAMPER 按键 | 按一下 TAMPER 对应 DS 3 灯亮，再按一下 DS 3 灭 |
| DS4 | WAKEUP 按键 | 按一下 WAKEUP 对应 DS 4 灯亮，再按一下 DS 4 灭 |

神舟 III 号开发板按键位置，如下图



7.5 EXTI外部按键中断实验

前面我们学习了，LED 灯和按键。实际上对于 STM32 来说，我们是学习了它的外设 GPIO。这一节我们前面学习的内容，学习 STM32 的 EXTI (External interrupt)，即外部中断。

前面的按键章节中，我们检测按键是否被按下的方式是轮询检测的方式，这里我们改为使用中断检测的方式，提高 CPU 的效率。

7.5.1 什么是中断

单片机中断系统的概念：什么是中断，我们从一个生活中的例程引入。比如说你在做A事，但是突然间来了你想起来了更重要的B事，所以你马上去做B事了，做完之后再回来继续做A事，这个就是中断。

7.5.2 什么是单片机的中断？

当CPU正在执行一个任务，但突然又发生了一个更高级的任务，CPU必须立即去执行的任务，所以CPU必须中断当前的任务，并保存该任务已经执行的状态和相关信息，然后转而去执行那个更

加高级的任务，因此就引入了“中断”这个概念。

中断是指计算机在执行程序的过程中，当出现异常情况或特殊请求时，计算机停止现行程序的运行，转向对这些异常情况或特殊请求的处理，处理结束后再返回现行程序的间断处，继续执行原程序。中断是单片机实时地处理内部或外部事件的一种内部机制。当某种内部或外部事件发生时，单片机的中断系统将迫使CPU暂停正在执行的程序，转而去进行中断事件的处理，中断处理完毕后，又返回被中断的程序处，继续执行下去。

在程序里面也是一样的。举个例子可能会容易懂点，定时中断：比如你定时1ms，主程序在运行，每当1ms时间到后，就跑到定时中断子程序里面执行，执行完后再回到主程序（中断程序是1ms中断一次）。那对于整个系统来说中断能实现什么好处呢？下面我们给以说明：

1) 提高了CPU的效率

CPU是计算机的指挥中心，它与外围设备（如按键、显示器等）通讯的方法有查询和中断2种：查询的方法是无论外围IO是否需要服务，CPU每隔一段时间都要依次查询一遍，这种方法CPU需要花费一些时间在做查询服务工作。

中断则是在外围设备需要通讯服务时主动告诉CPU，这个时候CPU才停下当前工作去处理中断程序，不需要占用CPU主动去查询的时间，CPU可以在没有中断请求来临之前一直做自己的工作，从而提高了CPU效率。

2) 可以实现实时处理

外设任何时候都可能发出请求中断信号，CPU接到请求后及时处理，以满足实时系统的需要。

3) 可以及时处理故障

计算机系统运行过程中难免会出现故障，有许多事情是无法预料的，如电源掉电、存储器出错、外围设备工作不正常等，这时可以通过中断系统向中断源CPU发送中断请求，由CPU及时转到相应的出错处理程序，从而提高计算机的可靠性。

7.5.3 STM32中断的初步理解

神舟 III 号开发板的主芯片是STM32F103ZET6，它采用的是ARM公司的Cortex-M3内核。Cortex-M3内核支持256个中断，具有256级的可编程中断设置。但STM32并没有使用M3内核的全部东西，而是只用了它的一部分。STM32只有84个中断（16个内核+68个外部），具有16级可编程的中断优先级。

Cortex-M3内核具有强大的异常响应系统，它能够打断当前代码执行流程的事件分为异常(exception)和中断(interrupt)，并把它们用一个表管理起来，这个表就称为中断向量表。而STM32对中断向量表重新进行了编排，把编号从-3至6的中断向量定义为系统异常，编号为负的内核异常不能被设置优先级，如复位(Reset)、不可屏蔽中断(NMI)、硬错误(Hardfault)。从编号7开始的外部中断，这些中断的优先级都是可以自行设置的。

STM32的部分中断向量表如下图，详细内容请参考ST公司提供的参考手册相关中断的章节。

表55 其它STM32F10xxx产品(小容量、中容量和大容量)的向量表

| 位置 | 优先级 | 优先级类型 | 名称 | 说明 | 地址 |
|----|-----|--------------------|----------------------------------|----|-----------------------------|
| - | - | - | 保留 | | 0x0000_0000 |
| -3 | 固定 | Reset | 复位 | | 0x0000_0004 |
| -2 | 固定 | NMI | 不可屏蔽中断 RCC时钟安全系统(CSS)联接到NMI向量 | | 0x0000_0008 |
| -1 | 固定 | 硬件失效(HardFault) | 所有类型的失效 | | 0x0000_000C |
| 0 | 可设置 | 存储管理(MemManage) | 存储器管理 | | 0x0000_0010 |
| 1 | 可设置 | 总线错误(BusFault) | 预取指失败, 存储器访问失败 | | 0x0000_0014 |
| 2 | 可设置 | 错误应用(UsageFault) | 未定义的指令或非法状态 | | 0x0000_0018 |
| - | - | - | 保留 | | 0x0000_001C ~0x0000_002B |
| 3 | 可设置 | SVCall | 通过SWI指令的系统服务调用 | | 0x0000_002C |
| 4 | 可设置 | 调试监控(DebugMonitor) | 调试监控器 | | 0x0000_0030 |
| - | - | - | 保留 | | 0x0000_0034 |
| 5 | 可设置 | PendSV | 可挂起的系统服务 | | 0x0000_0038 |
| 6 | 可设置 | SysTick | 系统滴答定时器 | | 0x0000_003C |

图 中断向量表

STM32F103ZET6 的芯片属于大容量的产品。虽然都是 STM32 的芯片，但是不同类型的产品中断向量表是有区别的。ST 公司的参考手册中针对不同类型的产品，提供有不同的中断向量表。实际应用的时候我们可以从启动文件 `startup_stm32f10x_hd.s` 中查找，在启动文件中，已经有相应芯片可用的全部中断向量。我们编写代码的时候，在写中断服务函数时，可以从启动文件中定义的中断向量表查找中断服务函数名。

7.5.4 STM32中断的初始化过程以及手册的查询

关于中断的更加深入的理解查阅手册之后，ST 明确给出需要查看另外的几个文档

STM32微控制器产品中大多数功能模块都是在多个产品(或所有产品)中共有的并且是相同的，因此只有一份STM32微控制器产品的技术参考手册对应所有这些产品。技术参考手册对每种功能模块都有专门的一个章节对应，每章的开始申明了这个功能模块的适用范围：例如第5章“备份寄存器”适用于整个STM32微控制器系列，第27章“以太网”只适用于STM32F107xx互联型产品。

为了方便阅读，下一页的表格列出了每个产品子系列所对应功能模块在技术参考手册中的章节一览。

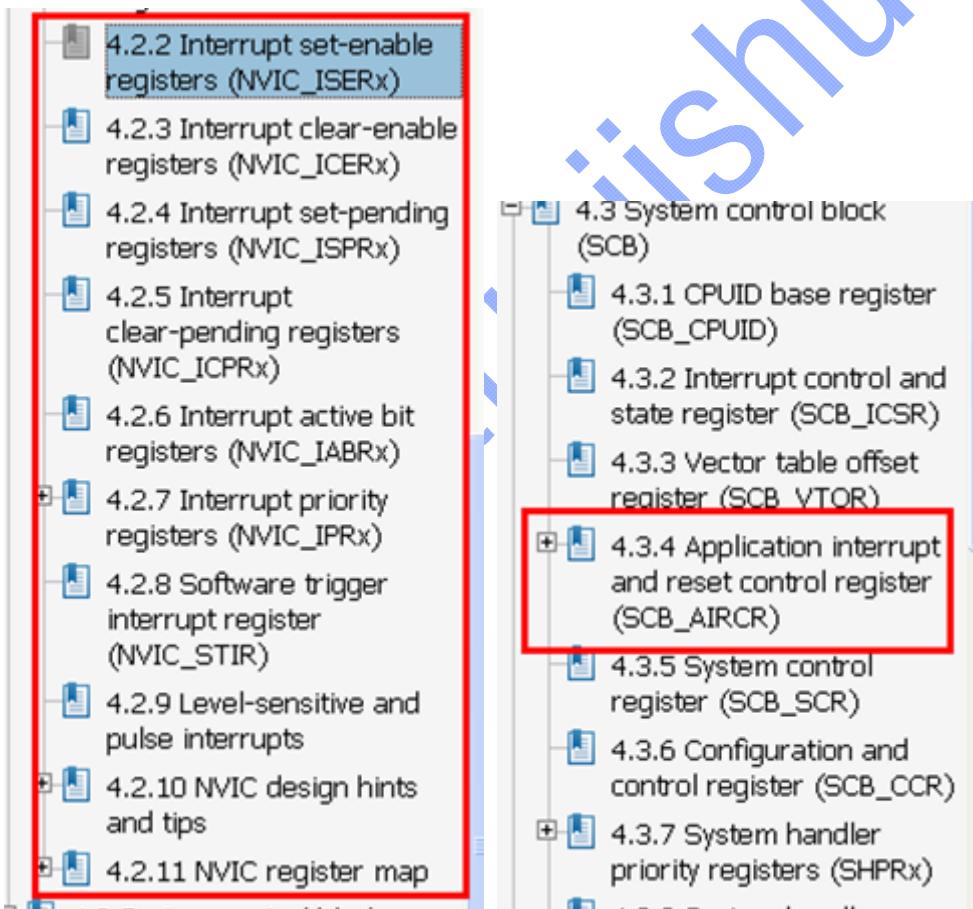
通常在芯片选型的初期，首先要看数据手册以评估该产品是否能够满足设计上的功能需求；在基本选定所需产品后，需要察看技术参考手册以确定各功能模块的工作模式是否符合要求；在确定选型进入编程设计阶段时，需要详细阅读技术参考手册获知各项功能的具体实现方式和寄存器的配置使用。在设计硬件时还需参考数据手册以获得电压、电流、管脚分配、驱动能力等信息。

关于 Cortex-M3 核心、SysTick 定时器和 NVIC 的详细说明，请参考另一篇 ST 的文档和一篇 ARM 的文档：
[《STM32F10xxx Cortex-M3 编程手册》](#) 和 [《Cortex™-M3 技术参考手册》](#)。

这两个手册都在光盘资料里可以找到，《STM32F10XXX 参考手册》中主要介绍的是关于中断的屏蔽，哪根中断线进行上升沿或者下降沿触发等，如下手册目录截图：

- 9.3 EXTI 寄存器描述
 - 9.3.1 中断屏蔽寄存器 (EXTI_IMR)
 - 9.3.2 事件屏蔽寄存器 (EXTI_EMR)
 - 9.3.3 上升沿触发选择寄存器(EXTI_RTSR)
 - 9.3.4 下降沿触发选择寄存器(EXTI_FTSR)
 - 9.3.5 软件中断事件寄存器(EXTI_SWIER)
 - 9.3.6 挂起寄存器 (EXTI_PR)
 - 9.3.7 外部中断/事件寄存器映像

这里没有涉及到优先级的设定，优先级的设定具体细节在另外一个手册中就是《STM32F10xxx Cortex-M3 编程手册》，如下手册目录截图：



最后第三个手册是《CM3 技术参考手册》主要是介绍理论上的，介绍整个 NVIC 的编程器模型，有兴趣的可以详细去研究一下：

| |
|----------------|
| 第8章 嵌套向量中断控制器 |
| 8.1 NVIC 概述 |
| 8.2 NVIC 编程器模型 |
| 8.3 电平中断与脉冲中断 |
| 第9章 存储器保护单元 |

这么用户手册，那么外部中断初始化的一般要经过那些步骤呢？请见下面：

- 1) 初始化IO口为输入。
- 2) 开启IO口复用时钟，设置IO口与中断线的映射关系。
- 3) 初始化线上中断，设置触发条件等（需参考《STM32F10XXX参考手册》）
- 4) 配置中断分组(NVIC)，并使能中断（需参考《STM32F10xxx Cortex-M3编程手册》）
- 5) 编写中断服务函数。

通过以上几个步骤的设置，我们就可以正常使用外部中断了，关于IO口初始化中断之后，唯一复杂的就只剩下中断分组的优先级了，这个优先级原本在51单片机中是比较简单的，但在M3中就被设计得比较复杂一些，优先级无非就是这些任务按先后顺序进行切换，那么我们接来下做一些详细的分析：

7.5.5 NVIC中断控制器

中断的数量到达一定的数量，控制起来变得麻烦，因此Cortex-M3给我们提供了NVIC (Nested Vectored Interrupt Controller)，即向量中断控制器。ST提供的参考手册中，并没有关于NVIC的描述。STM32的NVIC描述在手册《STM32F10xxx的Cortex-M3编程手册.pdf》中，特别是对相关寄存器的说明。

STM32采用Cortex-M3内核。但是STM32并没有使用Cortex-M3内核全部的东西，因此它的NVIC是Cortex-M3内核NVIC的子集。

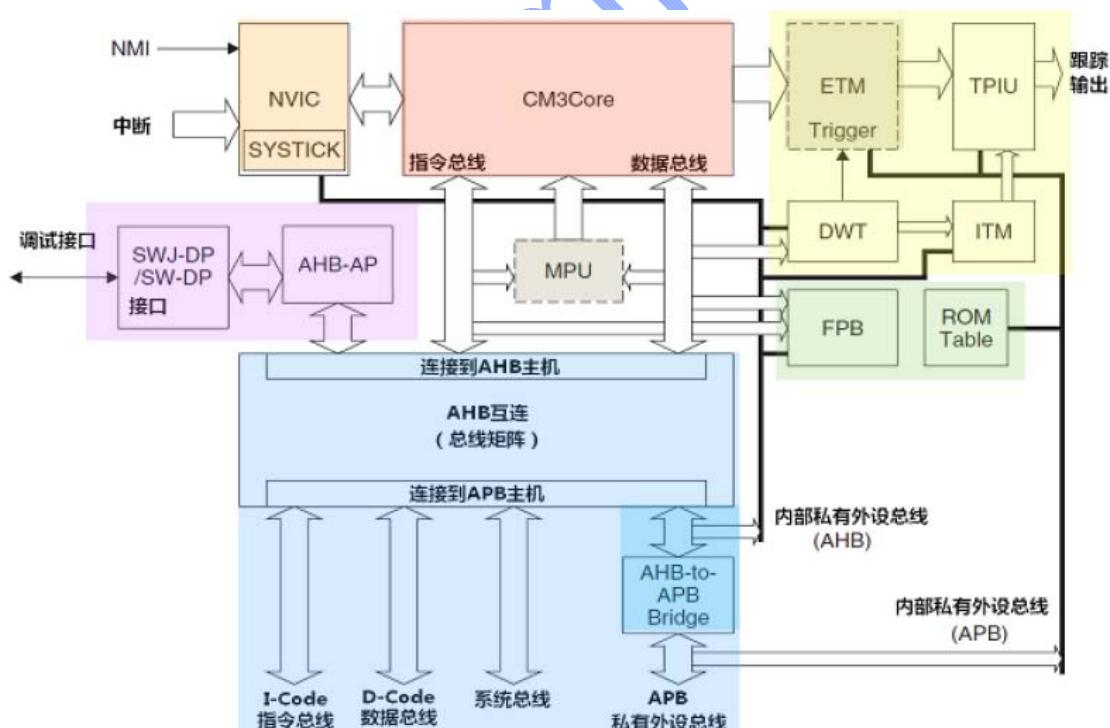


图 NVIC 在内核中的位置

我们对 NVIC (向量中断控制器) 进行操作的时候，一般要调用两个函数，中断分组函数 NVIC_PriorityGroupConfig() 和初始化函数 NVIC_Init()。我们先看一下，这两个函数怎么使用，然后嵌入式专业技术论坛（www.armjishu.com）出品

再对重要的概念进行分析。

中断分组函数 `NVIC_PriorityGroupConfig()`，对于该函数只有一个参数，ST 给我们提供了 5 个可用的参数，我们只需要直接调用即可。NVIC 初始化函数 `NVIC_Init()`，它有一个参数。它的参数是一个 `NVIC_InitTypeDef` 类型的结构体。这个结构体参数一般由我们进行设定。

我们看一下该结构体的成员：

| | |
|--|------------------|
| <code>NVIC_IRQChannel</code> | 需要配置的中断向量 |
| <code>NVIC_IRQChannelCmd</code> | 使能或关闭相应中断向量的中断响应 |
| <code>NVIC_IRQChannelPreemptionPriority</code> | 配置中断向量抢先优先级 |
| <code>NVIC_IRQChannelSubPriority</code> | 配置中断向量的响应优先级 |

首先要用 `NVIC_IRQChannel` 参数来选择将要配置的中断向量，用 `NVIC_IRQChannelCmd` 参数来进行使能(ENABLE)或关闭 (DISABLE) 该中断。这两个比较好理解。就是STM32的中断那么多，你要确认使用的是那一个，然后使能一下。

`NVIC_IRQChannelPreemptionPriority` 成员配置中断向量的抢先优先级，在 `NVIC_IRQChannelSubPriority` 配置中断向量的响应优先级。这两个优先级也是蛮好理解的。中断可以停下CPU当前正在执行的任务，转去执行其它的任务。当一个系统中，有许多的中断时。大家都是中断，一个中断正在运行时，另一个中断发生，后一个中断是否能够将前一个中断停止，从而去执行其它的任务？当多个中断同时发送时，要先执行那个，后执行那个？这个就是优先级要解决的问题。优先级的作用就是制定“规则”。优先级只有在有多个中断的时候，才有意义。一个系统中，只有一个中断的话，优先级就没有什么意义了。

下面我们详细分析一下，中断分组和中断优先级。

● 抢先优先级和响应优先级

STM32 的中断具有两个属性，一个为抢先属性，另一个为响应属性，其属性编号越小，表明它的优先级别越高。

抢先，是指打断其它中断的属性，即因为具有这个属性，会出现嵌套中断(在执行中断服务函数 A 的过程中被中断 B 打断，执行完中断服务函数 B 再继续执行中断服务函数 A)，抢先属性由 `NVIC_IRQChannelPreemptionPriority` 的参数配置。

而响应属性则应用在抢先属性相同的情况下，当两个中断向量的抢先优先级相同时，如果两个中断同时到达，则先处理响应优先级高的中断，响应属性由 `NVIC_IRQChannelSubPriority` 的参数配置。

例如，现在有三个中断向量：

| 中断向量 | 抢先优先级 | 响应优先级 |
|------|-------|-------|
| A | 0 | 0 |
| B | 1 | 0 |
| C | 1 | 1 |

若内核正在执行 C 的中断服务函数，则它能被抢先优先级更高的中断 A 打断，由于 B 和 C 的抢先优先级相同，所以 C 不能被 B 打断。但如果 B 和 C 中断是同时到达的，内核就会首先响应响应优先级更高的 B 中断。

● NVIC的优先级组

在配置优先级的时候，还有一个**优先级分组**。这个分组，对前面我们提到的**抢先优先级**和**响应优先级**进行分组。就是说，一个中断系统中，有多少个**抢先优先级**，多少个**响应优先级**，可以由用户自行设定。

实际上，中断分组是由 STM32 的寄存器 **AIRC** 中 **PRIGROUP** 的值设定。寄存器 **AIRC** 在 ST 官方提供的参考手册中并没有提到，而是在《STM32F10xxx 的 Cortex-M3 编程手册.pdf》手册中。在底层的寄存器配置中，中断分组由 4 个位来决定。 $2^4=16$ ，那么 NVIC 可以配置 16 种 中断向量的优先级。对于这 4bit 的中断优先级控制位还分成 2 组。第一组是定义抢先式优先级的位，第二组是定义子优先级的位。4bit 的分组组合可以有以下 5 种形式：

| 组别 | 4 个 bit 的分配 | 优先级状况 |
|-------|-------------|-------------------|
| 第 0 组 | 0:4 | 无抢先式优先级，16 个子优先级 |
| 第 1 组 | 1:3 | 2 个抢先式优先级，8 个子优先级 |
| 第 2 组 | 2:2 | 4 个抢先式优先级，4 个子优先级 |
| 第 3 组 | 3:1 | 8 个抢先式优先级，2 个子优先级 |
| 第 4 组 | 4:0 | 16 个抢先式优先级，无子优先级 |

第 0 组中，4 个 bit 都用来配置抢先优先级，即 NVIC 配置的 $2^4=16$ 种中断向量都是只有抢先属性，没有响应属性。

第1组：最高1位用来配置抢先优先级，低3位用来配置响应优先级。表示有 $2^1=2$ 种级别的抢先优先级(0级，1级)，有 $2^3=8$ 种响应优先级，即在16种中断向量之中，有8种中断，其抢先优先级都为0级，而它们的响应优先级分别为0~7，其余8种中断向量的抢先优先级则都为1级，响应优先级别分别为0~7。

依此类推，第4组中，所有4位用来配置响应优先级。即16种中断向量具有都不相同的响应优先级。

这5种中断分组ST官方已经给我们定义好，使用的时候非常的简单。我们在调用库函数 **NVIC_PriorityGroupConfig()** 的时候，可输入ST公司给我们定义好的参数：
NVIC_PriorityGroup_0 ~ NVIC_PriorityGroup_4。

于是，有读者觉得疑惑了，STM32有几十个中断，NVIC只能配置16种中断向量，这怎么够用？注意这里是**16种**，而不是**16个**。两个中断向量可以是同样的中断种类。

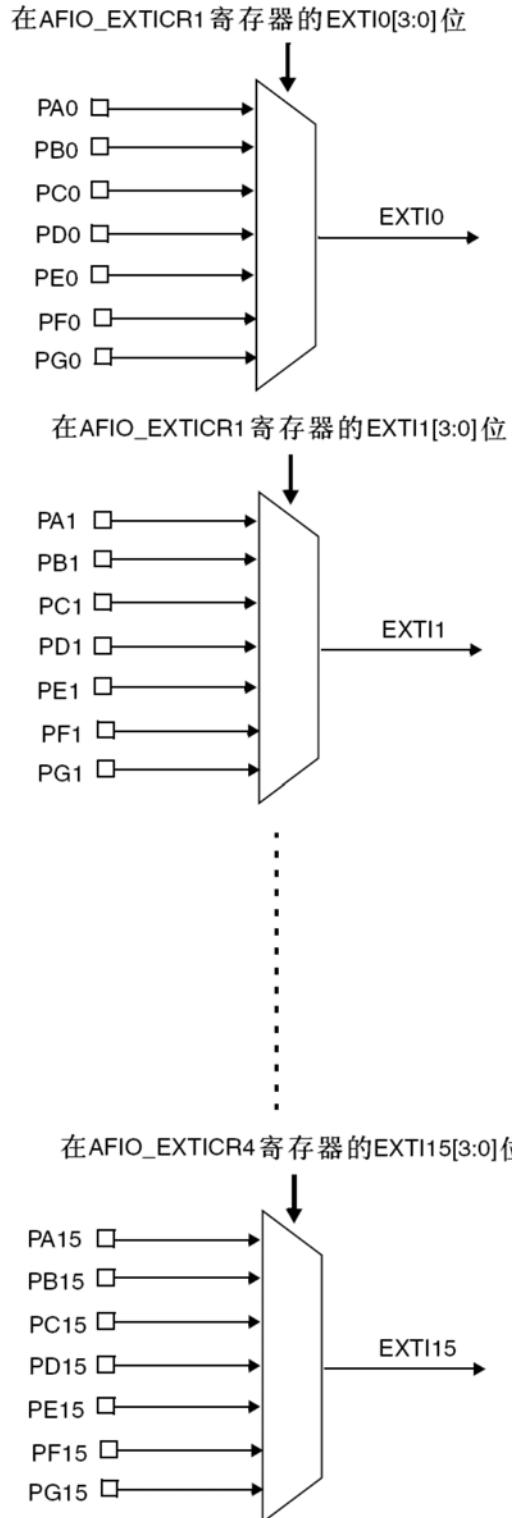
同时这里我们引入“**中断通道**”这个概念。比如一个STM2 单片机的所有I/O管脚可以配置为下降沿中断，上升沿中断和上升下降沿中断这三种模式。定时器（比如TIME2）本身能够引起中断的中断源或事件有好多，比如更新事件（上溢/下溢）、输入捕获、输出匹配、DMA申请等。所有TIME2的中断事件都是通过一个TIME2的中断通道向STM32内核提出中断申请。

外围设备通常具备若干个可以引起中断的中断源或中断事件。而该设备的所有的中断都只能通过该指定的“中断通道”向内核申请中断。当该中断通道的优先级确定后，也就确定了该外围设备的中断优先级，并且该设备所能产生的所有类型的中断，都享有相同的通道中断优先级。至于该设备本身产生的多个中断的执行顺序，则取决于用户的中断服务程序。

7.5.6 EXTI外部中断

STM32F103ZET一共有7组GPIO，每组GPIO口都有16个管脚，分别是PA[15:0]、PB[15:0]、PC[15:0]、PD[15:0]、PE[15:0]、PF[15:0]、PG[15:0]。STM32的所有GPIO管脚都可以作为中断输入源，但是如果每个GPIO都是一个独立的中断源，则需要112个中断源，这是不科学的，所以**通过复用的方式**使其对处理器来说来自GPIO的一共有16个中断Px[15:0]。具体实现是PA[0]、PB[0]、PC[0]、

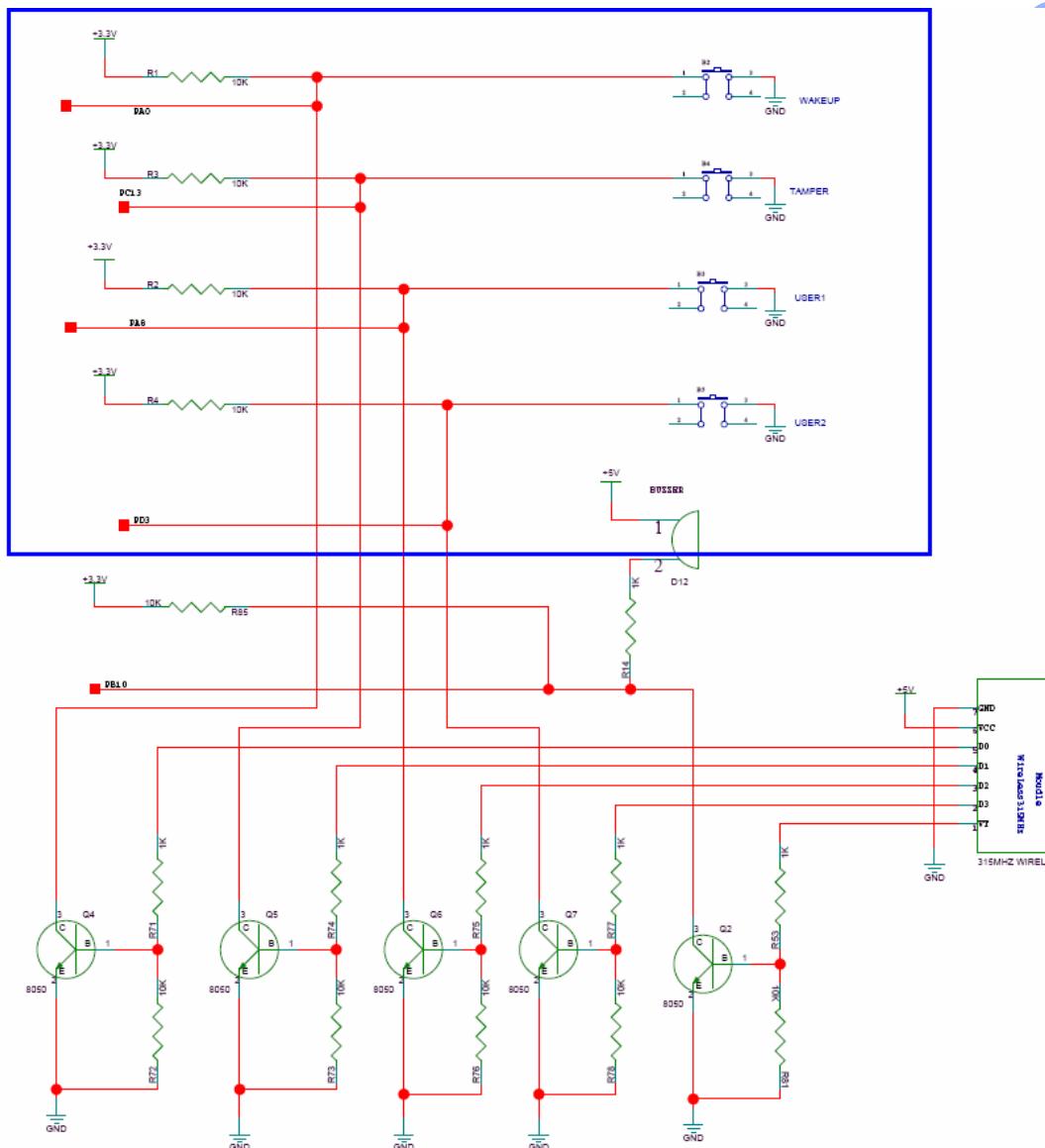
PD[0] 和PE[0]共享一个GPIO中断； PA[1]、PB[1]、PC[1]、PD[1] 和PE[1]共享一个GPIO中断； …… PA[15]、PB[15]、PC[15]、PD[15] 和PE[15]共享一个GPIO中断，如下图所示：



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 GPIO 端口的事件，但可以分时复用。它可以配置为上升沿触发，下降沿触发或双边沿触发。EXTI 最普通的应用就是接上一个按键，设置为下降沿触发，用中断来检测按键。

7.5.7 硬件设计

神舟III号STM32开发板总共有4个功能按键，分别是WAKEUP按键和TAMPER按键及两个用于自定义功能按键，在不使用第二功能的情况下，这四个按键都可以作为通用的按键，由用户自定义其功能。这四个按键分别与PD3、PA8、PC13和PA0四个GPIO管脚连接，当按键按下时，对应的GPIO管脚为低电平，反之，当没有按键按下时，对应的GPIO管脚为高电平。其中PA0 (STM32的WKUP引脚)可以作为WK_UP功能，它除了可以用作普通输入按键外，还可以用作STM32的唤醒输入。PC13可以实现备份区寄存器的入侵功能。本实验中所有的按键均作为普通IO使用。



图表 6 按键输入电路

GPIO 管脚与按键对应关系

| 按键 | 按键对应的GPIO |
|-------|-----------|
| USER1 | PA8 |

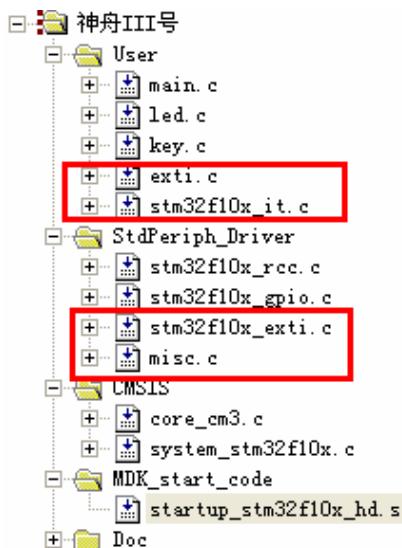
| | |
|-------------|------|
| USER2 | PD3 |
| KEY3/TAMPER | PC13 |
| KEY4/WAKEUP | PA0 |

本实验需要使用的 GPIO 管脚与对应的 LED 灯关系如下：

| LED灯 | LED灯对应的GPIO |
|------|-------------|
| DS1 | PF6 |
| DS2 | PF7 |
| DS3 | PF8 |
| DS4 | PF9 |

7.5.8 软件设计

我们在目录 Project 文件夹下增加了 exti.c 和 exti.h 文件。在工程中 User 组下添加 exti.c 文件和 stm32f10x_it.c 文件。同时在组 StdPeriph_Driver 中，添加 stm32f10x_exti.c 文件和 misc.c 文件。如下图：



我们从 main.c 主函数开始分析。

```
int main(void)
{
    /* 初始化板载LED指示灯 */
    LED_Init();

    /* 初始化板载按键为GPIO模式 */
    SZ_STM32_KEYInit();

    /* 配置NVIC中断优先级分组 */
    NVIC_GroupConfig();

    EXTIx_Init();

    /* Infinite loop 主循环 */
    while (1)
    {
```

主程序中，首先调用函数 LED_Init() 初始化板载的 LED 灯。函数 SZ_STM32_KEYInit() 初始化板载的按键。然后调用函数 NVIC_GroupConfig() 和函数 EXTIx_Init() 对按键中断进行配置。最后进入 while 循环。当没有按键中断发生的时候，板载的 4 个 LED 灯，呈流水灯的现象。按键中断发生时，执行相应的中断服务函数。而后再回过头来，执行流水灯的程序。

LED 灯和按键，这两部分我们前面已经讲过，这里就不重复。这里我们主要分析关于中断的内容。

代码分析 1：函数 NVIC_GroupConfig()，优先级分组。

```
/* 配置NVIC中断优先级分组：
- 2比特表示主优先级 主优先级合法取值为 0、1、2、3
- 2比特表示次优先级 次优先级合法取值为 0、1、2、3
- 数值越低优先级越高，取值超过合法范围时取低bit位 */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
```

本实验中我们将，2 比特表示主优先级，主优先级合法取值为 0、1、2、3 共 4 个等级。2 比特表示次优先级，次优先级合法取值为 0、1、2、3。数值越低优先级越高。

代码分析 2：函数 void EXTIx_Init()，外部中断配置，中断优先级配置。

```
039 void EXTIx_Init(void)
040 {
041     EXTI_InitTypeDef EXTI_InitStructure;
042     NVIC_InitTypeDef NVIC_InitStructure;
043
044     /* 开启AFIO的时钟 */
045     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
046
047     /* 将KEY1 按键对应的管脚连接到内部中断线 */
048     GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource3);
049
050     /* 将KEY1 按键配置为中断模式，下降沿触发中断 */
051     EXTI_InitStructure EXTI_Line = EXTI_Line3;           // 中断
052     EXTI_InitStructure EXTI_Mode = EXTI_Mode_Interrupt; // 中
053     EXTI_InitStructure EXTI_Trigger = EXTI_Trigger_Falling; // 下
054     EXTI_InitStructure EXTI_LineCmd = ENABLE;
055
056     EXTI_Init(&EXTI_InitStructure);
057
058     /* 将KEY1 按键的中断优先级配置为最低 */
059     NVIC_InitStructure NVIC_IRQChannel = EXTI3_IRQn;      // 使能按
060     NVIC_InitStructure NVIC_IRQChannelPreemptionPriority = 0x02;
061     NVIC_InitStructure NVIC_IRQChannelSubPriority = 0x03;
062     NVIC_InitStructure NVIC_IRQChannelCmd = ENABLE;        // 根据NVIC_Init
063
064     NVIC_Init(&NVIC_InitStructure);
```

代码中，首先定义了 EXTI_InitTypeDef 型的结构体 EXTI_InitStructure，和 NVIC_InitTypeDef 型的结构体 NVIC_InitStructure。这两个结构体在后边我们会用到。这是 C 语言的知识。

- 使用代码 `RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE)` 开启 AFIO 的时钟。
AFIO (alternate-function I/O)，指 GPIO 端口的复用功能，GPIO 除了用作普通的 GPIO 输入输出功能外，还可以作为片上外设的管脚。比如在 STM32F103ZET 芯片中，PA9 和 PA10 可以作为普通的 GPIO 管脚，但是它两通过复用可以用做串口的数据发送接收引脚。

当 GPIO 管脚用做 EXTI 外部中断时，必须打开 AFIO 时钟。

- EXTI 的初始化配置。

使用代码 `GPIO_EXTILineConfig(GPIO_PortSourceGPIOC, GPIO_PinSource4)`，把 GPIOC 的 Pin4 设置为 EXTI 的输入线。

给 EXTI_InitStructure 结构体赋值，该结构体有 4 个成员。

```
EXTI_InitStructure EXTI_Line = EXTI_Line3;           // 中断线
EXTI_InitStructure EXTI_Mode = EXTI_Mode_Interrupt; // 中断模式
EXTI_InitStructure EXTI_Trigger = EXTI_Trigger_Falling; // 下降沿触发
EXTI_InitStructure EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

给 `EXTI_Line` 成员赋值。选择 `EXTI_Line3` 线进行配置，因为按键的 PD3 连接到了 `EXTI_Line3`。

给 `EXTI_Mode` 成员赋值。把 `EXTI_Line3` 的模式设置为为中断模式 `EXTI_Mode_Interrupt`。这个结构体成员也可以赋值为事件模式 `EXTI_Mode_Event`，这个模式不会立刻触发中断，而只是在寄存器上把相应的事件标志位置 1，应用这个模式要不停地查询相应的寄存器。

给 `EXTI_Trigger` 成员赋值。把中断触发方式 (`EXTI_Trigger`) 设置为下降沿触发
嵌入式专业技术论坛（www.armjishu.com）出品 第 323 页，共 900 页

(EXTI_Trigger_Falling)。

给EXTI_LineCmd成员赋值。把EXTI_LineCmd设置为使能。最后调用EXTI_Init()把EXTI初始化结构体的参数写入寄存器。

● NVIC中断优先级配置。

```
NVIC_InitStructure.NVIC_IRQChannel = EXTI3 IRQn; //使能外部中断通道  
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02; //抢占优先级  
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x03; //子优先级  
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //使能外部中断通道  
NVIC_Init(&NVIC_InitStructure);
```

给NVIC_IRQChannel成员赋值。选择EXTI3_IRQn线进行配置，因为按键的PD3使用的是外部通道EXTI3_IRQn。

给NVIC_IRQChannelPreemptionPriority成员赋值。这里是设置抢先优先级。

给NVIC_IRQChannelSubPriority成员赋值。这里是设置响应优先级。

给NVIC_IRQChannelCmd成员赋值。设置为ENABLE，使能外部中断通道。调用函数NVIC_Init()把NVIC初始化结构体的参数写入寄存器。

代码分析3：中断服务函数。

在这个EXTI设置中我们把PD3连接到内部的EXTI3，GPIO配置为上拉输入，工作在下降沿中断。在外围电路上我们将PD3接到了key1上。当按键没有按下时，PD3始终为高，当按键按下时PD3变为低，从而PD3上产生一个下降沿跳变，EXTI3会捕捉到这一跳变，并产生相应的中断，对应到的中断函数为EXTI3_IRQHandler()，可以在stm32f10x_it.c中找到这个对应的函数。

```
159 void EXTI3_IRQHandler(void) /* Key 1 */  
160 {  
161     if(EXTI_GetITStatus(EXTI_Line3) != RESET)  
162     {  
163         /* 延迟去抖 */  
164         delay(150000);  
165         if(0 == GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_3))  
166         {  
167             SZ_STM32_LED1Toggle();  
168         }  
169         while(0 == GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_3))  
170         delay(150000);  
171         /* 清除中断挂起标志位，否则会被认为中断没有被处理而循  
172         EXTI_ClearITPendingBit(EXTI_Line3);  
173     }  
174 }
```

中断服务函数的名称是不能随意定义的，必须与启动文件startup_stm32f10x_hd.s中的中断向量表定义一致。

● 这里需要说明一下，STM32的IO口外部中断函数只有6个，分别为：

```
EXPORT EXTI0_IRQHandler  
EXPORT EXTI1_IRQHandler  
EXPORT EXTI2_IRQHandler  
EXPORT EXTI3_IRQHandler  
EXPORT EXTI4_IRQHandler  
EXPORT EXTI9_5_IRQHandler  
EXPORT EXTI15_10_IRQHandler
```

中断线0-4每个中断线对应一个中断函数，中断线5-9共用中断函数EXTI9_5_IRQHandler，中断线10-15共用中断函数EXTI15_10_IRQHandler。

● 关于函数EXTI_GetITStatus(EXTI_Line3)和EXTI_ClearITPendingBit(EXTI_Line3)做进一步分析在编写中断服务函数的时候会经常使用到两个函数。

第一个函数是判断某个中断线上的中断是否发生（标志位是否置位）。这个函数一般使用在中断服务函数的开头判断中断是否发生。

第二个函数是清除某个中断线上的中断标志位，这个函数一般应用在中断服务函数结束之前，清除中断标志位。

中断服务函数中，其它的内容就是我们讲按键的章节的内容一致，延时去抖，反转 LED 灯的状态。

7.5.9 下载与验证

如果使用 JLINK 下载固件，请按[3.2 如何使用JLINK软件](#)下载固件到神舟III号开发板小节进行操作。

7.5.10 实验现象

上电运行。在没有按键按下时，板载的4个LED灯，呈流水灯功能。有按键按下，相关的LED灯会随之发生变化，并且流水灯功能停止。按键松开，流水灯功能恢复。

7.6 SysTick系统滴答实验

我们学习 NVIC 向量中断控制器的时候，ST 官方提供的参考手册中几乎没有涉及 NVIC 的内容。我们要了解它里面的寄存器的话还要参考《STM32F10xxx 的 Cortex-M3 编程手册.pdf》。Systick 定时器的情况和 NVIC 类似。

STM32 采用的是 Cortex-M3 的核，NVIC 和 Systick 在 Cortex-M3 的手册中都有详细的介绍。要注意的是 STM32 中的 NVIC、Systick 和 Cortex-M3 中的 NVIC、Systick 是有一些差异的。

7.6.1 Systick简介

● 什么是 SysTick 定时器

SysTick，系统滴答滴答。系统滴答滴答很形象地表示了它是一个系统节拍器。SysTick 是一个 24 位的倒计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。

● SysTick 定时器的作用

(1) 产生操作系统的时钟节拍

SysTick 定时器被捆绑在 NVIC 中，用于产生 SYSTICK 异常（异常号：15）。在以前，大多操作系统需要一个硬件定时器来产生操作系统需要的滴答中断，作为整个系统的时基。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。SysTick 的最大使命，就是定期地产生异常请求，作为系统的时基。OS 都需要这种“滴答”来推动任务和时间的管理。

(2) 便于不同处理器之间程序移植。

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，软嵌入式专业技术论坛 (www.armjishu.com) 出品 第 325 页，共 900 页

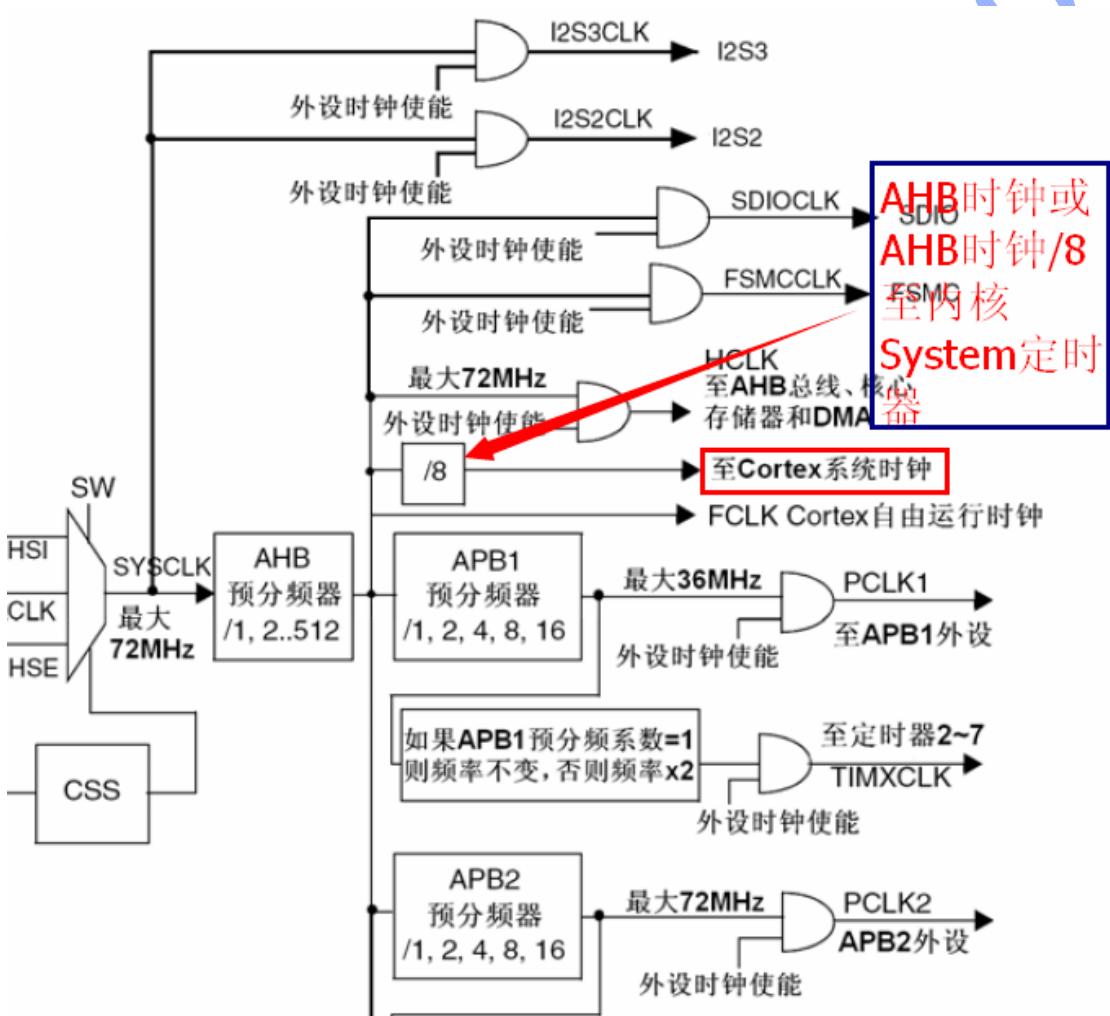
件在不同 CM3 器件间的移植工作得以化简。该定时器的时钟源可以是内部时钟 (FCLK, CM3 上的自由运行时钟), 或者是外部时钟 (CM3 处理器上的 STCLK 信号)。不过, STCLK 的具体来源则由芯片设计者决定, 因此不同产品之间的时钟频率可能会大不相同, 你需要检视芯片的器件手册来决定选择什么作为时钟源。SysTick 定时器能产生中断, CM3 为它专门开出一个异常类型, 并且在向量表中有它的一席之地。它使操作系统和其它系统软件在 CM3 器件间的移植变得简单多了, 因为在所有 CM3 产品间对其处理都是相同的。

(3) 作为一个闹铃测量时间。

SysTick 定时器还可以用作闹钟, 作为启动一个特定任务的时间依据。它作为一个闹铃, 用于测量时间。要注意的是, 当处理器在调试期间被喊停 (halt) 时, 则 SysTick 定时器亦将暂停运作。

● SysTick 定时器的时钟源

如下图:



Systick 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, 需要阅读芯片的使用手册来确定选择什么作为时钟源。

STM32 中, 我们可以通过代码配置 Systick 定时器的时钟等于 AHB 时钟或者是 AHB 时钟/8。

7.6.2 实验原理

本实验我们通过 Systick，进行精确的延时。

在前面的学习中，我们使用的延时函数 delay ()：

```
void delay(__IO uint32_t nCount)
{
    for ( ; nCount != 0; nCount--);
}
```

延时是通过给 nCount 设置值，CPU 循环执行变量 nCount 自减实现。对于 STM32 系列微处理器来说，很难计算出延时的精确值。当我们需要精确延时时，可以利用 Systick 来设计。

SysTick其实就是一个24位的倒计数定时器，当计到0时，将从STK_LOAD寄存器中自动重装载定时初值。只要不把它在SysTick控制及状态寄存器中的使能位清除，就永不停息。SysTick相关的几个寄存器的寄存器不算多，相关的说明《STM32F10xxx的Cortex-M3编程手册.pdf》中有详细的介绍：

● SysTick相关的4个寄存器名称和地址如下：

| | | |
|---------------|------------|-----------|
| SysTick_CTRL | 0xE000E010 | -- 控制寄存器 |
| SysTick_LOAD | 0xE000E014 | -- 重载寄存器 |
| SysTick_VAL | 0xE000E018 | -- 当前值寄存器 |
| SysTick_CALIB | 0xE000E01C | -- 校准值寄存器 |

◆ SysTick_CTRL SysTick控制与状态寄存器

4.4.1 SysTick control and status register (STK_CTRL)

Address offset: 0x00

Reset value: 0x0000 0004

Required privilege: Privileged

The SysTick CTRL register enables the SysTick features.

| | | | | | | | | | | | | | | | | |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | COUNT FLAG |
| Reserved | | | | | | | | | | | | | | | | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Reserved | | | | | | | | | | | | | | | | rw |
| CLKSO URCE | | | | | | | | | | | | | | | | TICK INT |
| rw | | | | | | | | | | | | | | | | ENABLE |
| rw | | | | | | | | | | | | | | | | rw |

可以看

到实际使用只用到了其中的4位，如下表所示：

| 域 | 名称 | 定义 |
|------|-----------|--|
| [16] | COUNTFLAG | 从上次读取定时器开始，如果定时器计数到 0，则返回 1。读取时清零。 |
| [2] | CLKSOURCE | 0= 外部参考时钟 1= 内核时钟 如果没有提供参考时钟，那么该位保持为 1，并且因此赋予和内核时钟一样多的时间。内核时钟比参考时钟至少要快 2.5 倍。否则计数值将不可预测。 |
| [1] | TICKINT | 1= 向下计数至 0 会导致挂起 SysTick 处理器 0= 向下计数至 0 不会导致挂起 SysTick 处理器。软件可以使用 COUNTFLAG 来判断是否计数到 0。 |
| [0] | ENABLE | 1= 计数器工作在连拍模式 (multi-shot)。即计数器装载重装值后接着开始往下计数。到计数到 0 时将 COUNTFLAG 设为 1，此时根据 TICKINT 的值可以选择是否挂起 SysTick 处理器。接着又再次装载重装值，并重新开始计数。 0= 禁能计数器 |

位CLKSOURCE用于选择Systick定时器时钟。当为1时，Systick定时器时钟为AHB。当为0时，Systick定时器时钟为AHB/8。

◆ SysTick_LOAD 重装值寄存器

SysTick一个递减的计数器，当计数器递减到“0”时，重装寄存器中的值就会被重装。

SysTick_LOAD重装值寄存器是一个24位宽的寄存器，如下图所示：



SysTick 重装值寄存器的位分配

| 域 | 名称 | 定义 |
|--------|--------|------------------------|
| [23:0] | RELOAD | 当计数器到达 0 时装载“当前值寄存器”的值 |

◆ SysTick_VAL 当前值寄存器

使用SysTick当前值寄存器来查找寄存器中的当前值。具体寄存器的位分配如下所示：

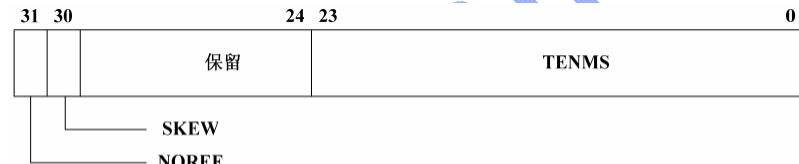


SysTick 当前值寄存器的位分配

| 域 | 名称 | 定义 |
|--------|---------|--|
| [23:0] | CURRENT | 访问寄存器时的当前值。没有提供读-修改-写保护，所以在修改时要特别小心。 该寄存器是写-清除。向该寄存器写入任意值都可以将其清除变为 0。清零该寄存器还会导致“SysTick 控制与状态寄存器”的 COUNTFLAG 位清零。 |

◆ SysTick_CALIB 校准寄存器

使用校准寄存器通过乘法或是除法运算可以将寄存器调节成任意所需的时钟速率，如下所示：



SysTick 校验值寄存器的位分配

| 域 | 名称 | 定义 |
|--------|-------|---|
| [31] | NOREF | I= 没有提供参考时钟 |
| [30] | SKEW | I= 由于时钟频率的原因，校验值不是精确的 10ms。这会影响其作为一个软件实时钟的适合性。 |
| [23:0] | TENMS | 该值是用于 10ms 定时的重装值。其值取决于 SKEW，它可以是精确的 10ms，也可以是最接近 10ms 的值。 如果为 0，那么检验值就未知。这很可能是因为参考时钟是系统的一个未知输入或者因为参考时钟可以动态调节。 |

● Systick定时器定时时间的计算

本实验中，我们给Systick定时器，配置的时钟频率是72MHz。1Hz表示1秒震荡1次，72MHz表示1秒震荡72M次，那么每次振荡时间是1/72MHz。现在需要振荡72M/1000000次，总共所花的时间用以下公式来算（72M=72000000）：

需要振荡的次数 * 每次的振荡时间 = $72M/10000000 * 1/72M = 1/1000000 = 0.000001$ 秒 = 1微秒=1us。

也就是说，systick 定时器每隔 1us 时钟就会产生一次 systick 的中断。

7.6.3 代码分析

我们从主函数开始。

```
17 int main(void)
18 {
19     /* 初始化板载LED指示灯 */
20     LED_Init();
21
22     //配置Systick定时器, 1us中断一次
23     SZ_STM32_SysTickInit(1000000);
24
25     while (1)
26     {
27         GPIO_ResetBits(GPIOB, GPIO_Pin_2); //拉低
28         Delay_us(10); //延时10微秒
29         Delay_ms(10); //延时10毫秒
30         Delay_s(1); //延时1秒
31         GPIO_SetBits(GPIOB, GPIO_Pin_2); //拉高
32         Delay_us(10); //延时10微秒
33         Delay_ms(10); //延时10毫秒
34         Delay_s(1); //延时1秒
35     }
36 }
```

调用函数 LED_Init() 初始化板载 LED 灯，调用函数 SZ_STM32_SysTickInit() 配置 Systick，进入 while 循环，点亮 4 个 LED 灯，延时 1 秒时间，熄灭 LED 灯，再延时 1 秒时间。如此循环。我们这里主要分析的是 SZ_STM32_SysTickInit() 函数。

代码分析 1：函数 SZ_STM32_SysTickInit() 分析。

```
void SZ_STM32_SysTickInit(uint32_t HzPreSecond)
{
    if (SysTick_Config(SystemCoreClock / HzPreSecond))
    {
        while (1);
    }
}
```

本函数实际上只是调用了 SysTick_Config() 函数，它是属于内核层的 Cortex-M3 通用函数，位于 core_cm3.h 文件中，若调用 SysTick_Config() 配置 SysTick 不成功，则进入死循环。

代码分析 2： SysTick_Config() 函数详解

```

static __INLINE uint32_t SysTick_Config(uint32_t ticks)
{
    if (ticks > SysTick_LOAD_RELOAD_Msk)  return (1);

    SysTick->LOAD  = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;
    NVIC_SetPriority (SysTick_IRQn, (1<<_NVIC_PRIO_BITS) - 1);
    SysTick->VAL   = 0;
    SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                      SysTick_CTRL_TICKINT_Msk |
                      SysTick_CTRL_ENABLE_Msk;

    return (0);
}

```

这个函数配置 24 位的倒计数 Systick 定时器，当计到 0 时引起中断，并从 STK_LOAD 寄存器中自动重装载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。输入的参数 ticks 经过处理送 STK_LOAD 寄存器，为两个中断之间的脉冲数（定时初值），即相隔 ticks 个时钟周期会引起一次中断；配置 SysTick 成功时返回 0，出错进返回 1。

第一行代码 “if (ticks > SysTick_LOAD_RELOAD_Msk) return (1)” 检查输入参数 ticks，因为 ticks 是定时初值，要被保存到重载寄存器 STK_LOAD 寄存器中，再由硬件把 STK_LOAD 值加载到当前计数值寄存器 STK_VAL 使用的，STK_LOAD 和 STK_VAL 都是 24 位的，所以当输入参数 ticks 大于其可存储的最大值时，将由这行代码检查出错误返回。

第二行代码 “SysTick->LOAD = (ticks & SysTick_LOAD_RELOAD_Msk) - 1” 稍稍处理一下把 ticks-1 赋值给 STK_LOAD 寄存器，要注意的是减 1，若 STK_VAL 从 ticks-1 向下计数至 0，实际上就经过了 ticks 个脉冲。

这句赋值代码中使用到了宏 SysTick_LOAD_RELOAD_Msk，与其它库函数类似，这样子的宏是用来指示寄存器的**特定位置** 或进行位屏蔽用的。类似的宏定义如下：

```

/* SysTick Reload Register Definitions */
#define SysTick_LOAD_RELOAD_Pos      0
#define SysTick_LOAD_RELOAD_Msk     (0xFFFFFFFFul << SysTick_LOAD_RELOAD_Pos)

/* SysTick Current Register Definitions */
#define SysTick_VAL_CURRENT_Pos      0
#define SysTick_VAL_CURRENT_Msk     (0xFFFFFFFFul << SysTick_VAL_CURRENT_Pos)

/* SysTick Calibration Register Definitions */
#define SysTick_CALIB_NOREF_Pos      31
#define SysTick_CALIB_NOREF_Msk     (1ul << SysTick_CALIB_NOREF_Pos)

```

这里 0xFFFFFFFFul，ul 表示无符号长整型数值。0xFFFFFFFF6 位的 16 进制刚好是 24 位的 2 进制。0xFFFFFFFFul 左移 SysTick_LOAD_RELOAD_Pos 位，SysTick_LOAD_RELOAD_Pos 等于 0。即 SysTick_LOAD_RELOAD_Msk 等于 0xFFFFFFFFul 左移 0 位。像这样子的操作，在控制寄存器的代码（大部分库函数）中用得十分广泛。

第三行代码 “NVIC_SetPriority (SysTick_IRQn, (1<<_NVIC_PRIO_BITS) - 1)” 配置中断向量及重置 STK_VAL 寄存器。配置了 SysTick 中断，这就是为什么我们在外部没有再使用 NVIC 配置 SysTick 中断的原因。

第四行代码 “SysTick->VAL = 0” 把 STK_VAL 寄存器重新赋值为 0（在使能 SysTick 时，硬件会把存储在 STK_LOAD 寄存器中的 ticks 值加载给它）。

最后一行代码设置了 Systick 的 CTRL 寄存器，配置 SysTick 定时器时钟频率为 AHB（72MHz）。同时使能 Systick 定时器和 Systick 定时器中断。

代码分析 3：中断服务函数详解

```
void SysTick_Handler(void)
{
    TimingDelay_Decrement();
}
```

调用 TimingDelay_Decrement() 函数

```
void TimingDelay_Decrement(void)
{
    if (TimingDelay != 0x00)
    {
        TimingDelay--;
    }
}
```

Systick 定时器每中断一次 TimingDelay 减 1，就表示时间过了 1 微秒。

代码分析 4：延时函数 Delay_us()

```
void Delay_us(__IO u32 nTime)
{
    TimingDelay = nTime;
    // 使能滴答定时器
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;

    while(TimingDelay != 0);

    // 关闭滴答定时器
    SysTick->CTRL &= ~ SysTick_CTRL_ENABLE_Msk;
}
```

延时函数有一个参数 nTime，函数中首先通过 nTime 对 TimingDelay 进行赋值。然后，对 Systick 定时器 CTRL 寄存器的 ENABLE 位进行置 1，重载计数值，使能 Systick 定时器。“`while(TimingDelay != 0)`” 代码，等待 TimingDelay 变为 0。Systick 定时器每中断一次 TimingDelay 减 1，即时间过了 1 微秒。最后对 Systick 定时器 CTRL 寄存器的 ENABLE 位进行置 0，关闭 Systick 定时器，完成一次延时。

通过对参数 nTime 的设定，确定延时时间。延时函数和 Systick 定时器中断的关联如下图：

```

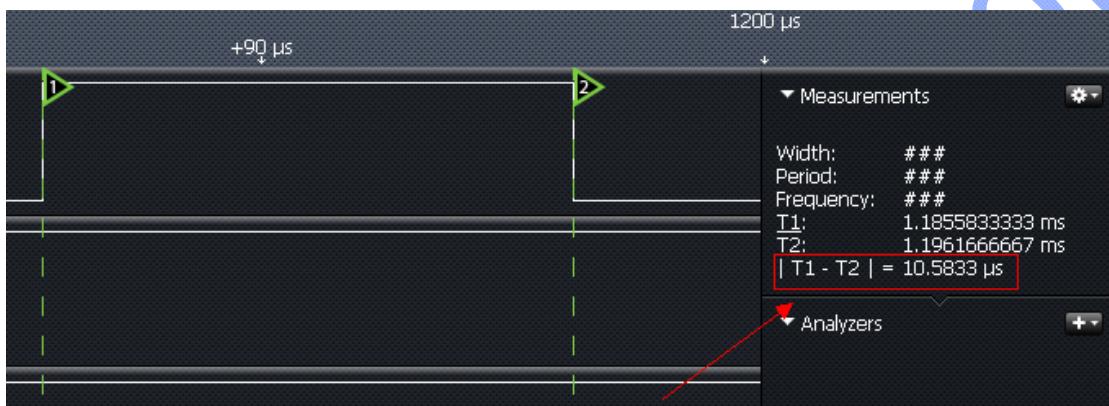
void sysTick_Handler(void)
{
    TimingDelay_Decrement();
}

void TimingDelay_Decrement(void)
{
    if (TimingDelay != 0x00)
    {
        TimingDelay--;
    }
}

void Delay_us(__IO u32 nTime)
{
    TimingDelay = nTime;
    // 使能滴答定时器
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
    while(TimingDelay != 0);
    // 关闭滴答定时器
    SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;
}

```

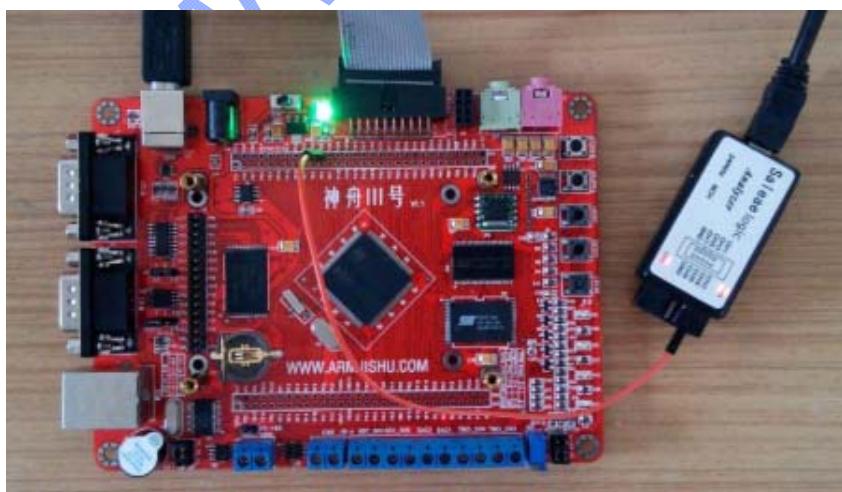
调用 Delay_us() 函数，对一个 LED 灯的 GPIO 管脚的进行拉高 10us 在拉低 10us。通过仪器测量得出波形如下：



实际高电平持续的时间是 10.5833 微秒，注意这个测得的时间是高电平持续的时候，而不延时函数的延时时间。这个多出来的 0.5833 微秒是因为，延时函数中调用了赋值语句，设置寄存器语句，同时调用库函数将 GPIO 管脚的电平拉低也消耗一定的时候。这样子的话，延时函数的误差是非常的小的。微秒级的延时函数就有那么高的精确度，那么当我们调用 Delay_ms() 函数，进行毫秒级的延时的时候，0.5833 微秒的差异几乎等于没有差异。

7.6.4 实验现象

本实验，通过 Systick 定时器，进行精确的 1 秒延时。将代码下载到开发板，按下复位按键使用逻辑分析仪测量管脚电平变化的时间，从而得初延时的时间。如下图：



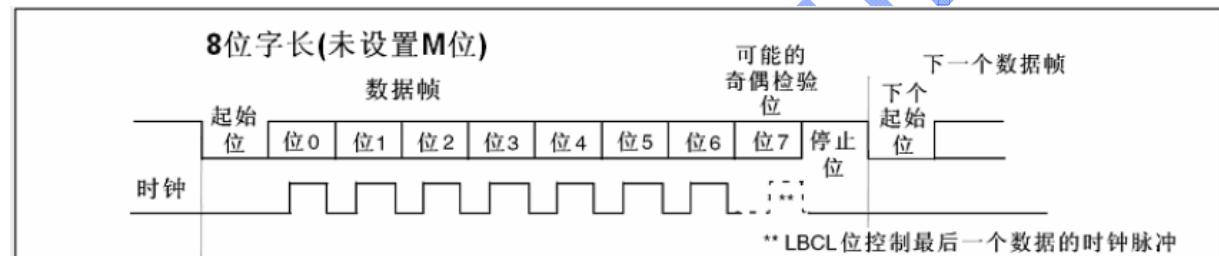
7.7 USART串口1通信实验

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。

前面我们已经通过寄存器的方式学习过串口了，下面主要是更多介绍一下 STM32 库函数的串口实现方式，再进一步补充一点串口的更深入的知识点。

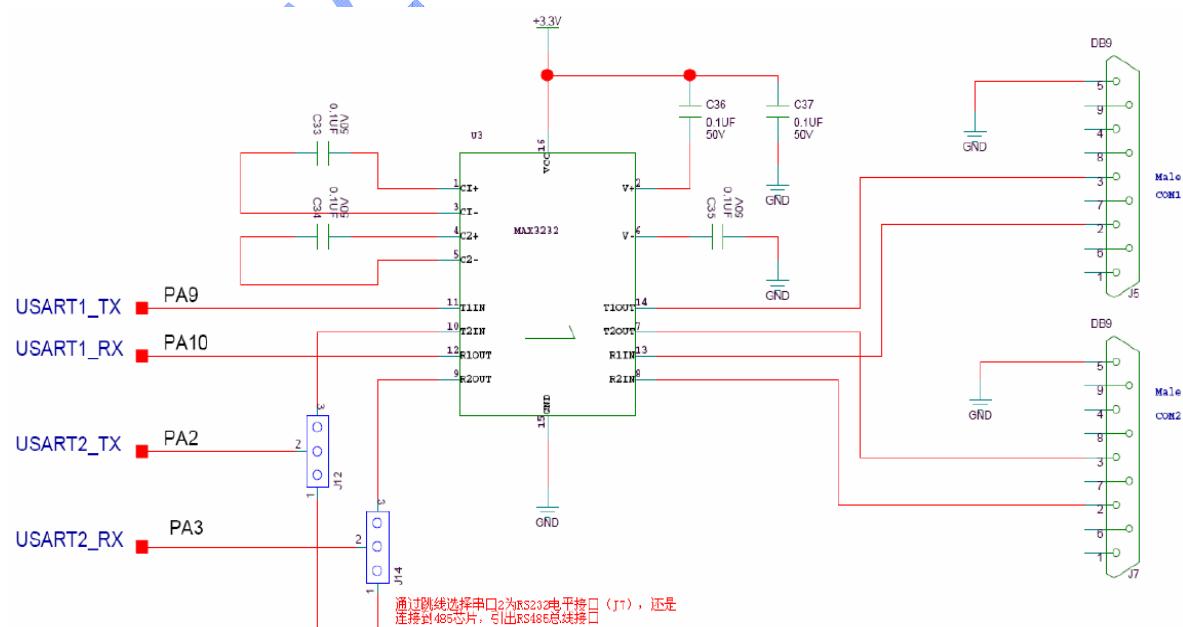
7.7.1 STM32的异步串口通讯协议

阅读过《STM32 中文参考手册》的读者会发现，STM32 的串口非常强大，它不仅支持最基本的通用串口同步、异步通讯，还具有 LIN 总线功能(局域互联网)、IRDA 功能(红外通讯)、SmartCard 功能。为实现最迫切的需求，利用串口来帮助我们调试程序，本章介绍的为串口最基本、最常用的方法，全双工、异步通讯方式。下图为串口异步通讯协议。



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

7.7.2 交叉线和直连线



见上图，这是神舟 III 号 STM32 开发板的接线图，使用的为 MAX3232 芯片，把 STM32 的 PA10 引脚(复用功能为 USART1 的 Rx)接到了 DB9 接口的第 2 针脚，把 PA9 引脚(复用功能为 USART 的 Tx)连接到了 DB9 接口的第 3 针脚。

Tx (发送端) 接第 3 针脚，Rx (接收端) 接第 2 针脚。这种接法是跟 PC 的串口接法一样的，如果要实现 PC 跟神舟 STM32 开发板通讯，就要使用两头都是母的交叉线。

7.7.3 串口工作流程

1.串口传输速度控制（波特率控制）

前面的寄存器版本有提到过，这里再细讲一下，波特率，即每秒传输的二进制位数，用 b/s (bps) 表示，通过对时钟的控制可以改变波特率。在配置波特率时，我们向波特比率寄存器 USART_BRR 写入参数，修改了串口时钟的分频值 USARTDIV。USART_BRR 寄存器包括两部分，分别是 DIV_Mantissa(USARTDIV 的整数部分)和 DIVFraction(USARTDIV 的小数)部分，最终，计算公式为 $\text{USARTDIV} = \text{DIV_Mantissa} + (\text{DIVFraction}/16)$ 。

USARTDIV 是对串口外设的时钟源进行分频的，对于 USART1，由于它是挂载在 APB2 总线上的，所以它的时钟源为 fPCLK2；而 USART2、3 挂载在 APB1 上，时钟源则为 fPCLK1，串口的时钟源经过 USARTDIV 分频后分别输出作为发送器时钟及接收器时钟，控制发送和接收的时序。

2.收和发控制

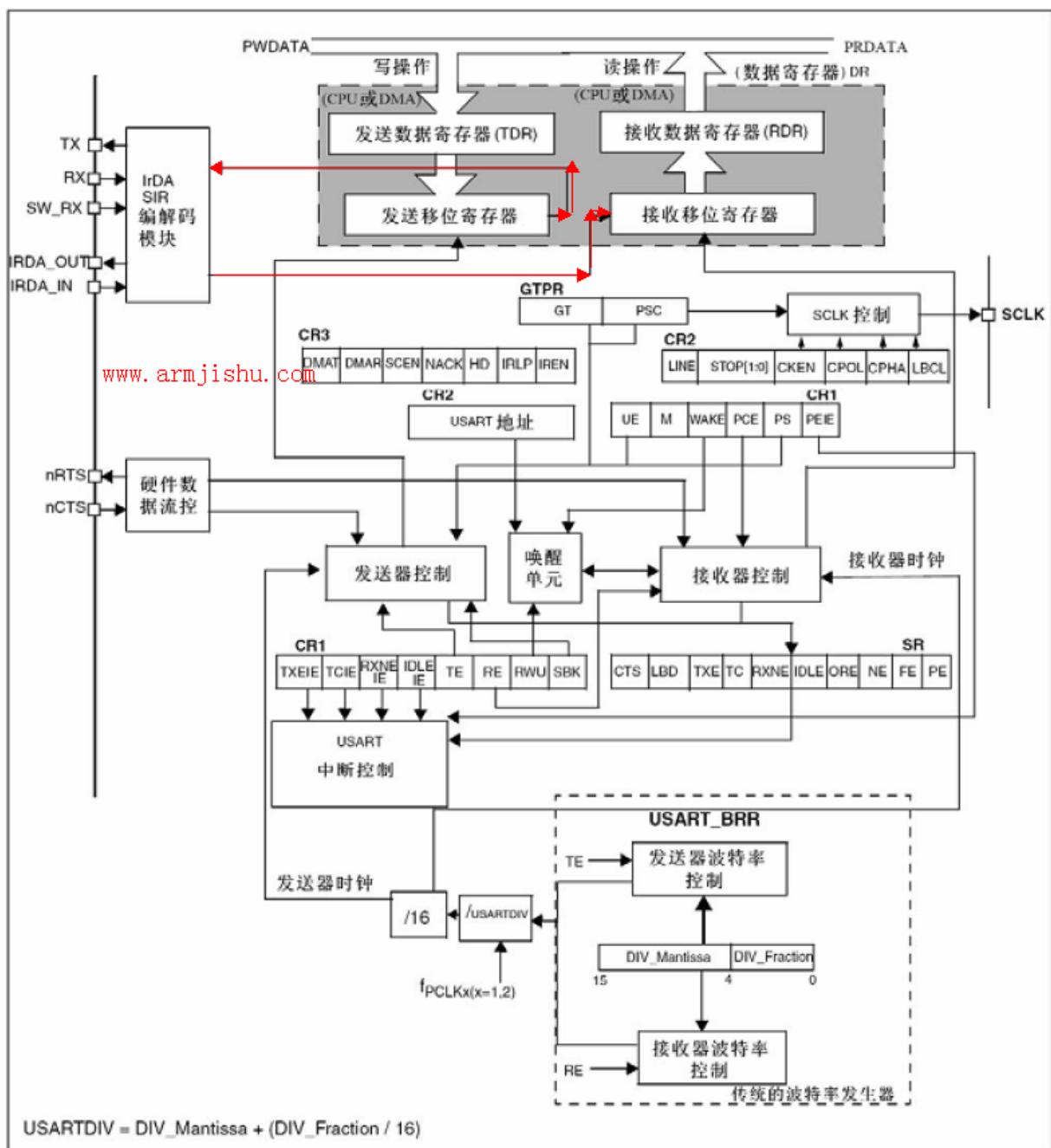
围绕着发送器和接收器控制部分，有好多个寄存器：CR1、CR2、CR3、SR，即 USART 的三个控制寄存器(Control Register)及一个状态寄存器(Status Register)。通过向寄存器写入各种控制参数，来控制发送和接收，如奇偶校验位，停止位等，还包括对 USART 中断的控制；串口的状态在任何时候都可以从状态寄存器中查询得到。具体的控制和状态检查，我们都是使用库函数来实现的，在此就不具体分析这些寄存器位了。

3.传输过程中的数据存储

收发控制器根据我们的寄存器配置，对数据存储转移部分的移位寄存器进行控制。当我们需要发送数据时，内核或 DMA 外设把数据从内存(变量)写入到发送数据寄存器 TDR 后，发送控制器将适时地自动把数据从 TDR 加载到发送移位寄存器，然后通过串口线 Tx，把数据一位一位地发送出去，在数据从 TDR 转移到移位寄存器时，会产生发送寄存器 TDR 已空事件 TXE，当数据从移位寄存器全部发送出去时，会产生数据发送完成事件 TC，这些事件可以在状态寄存器中查询到。

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

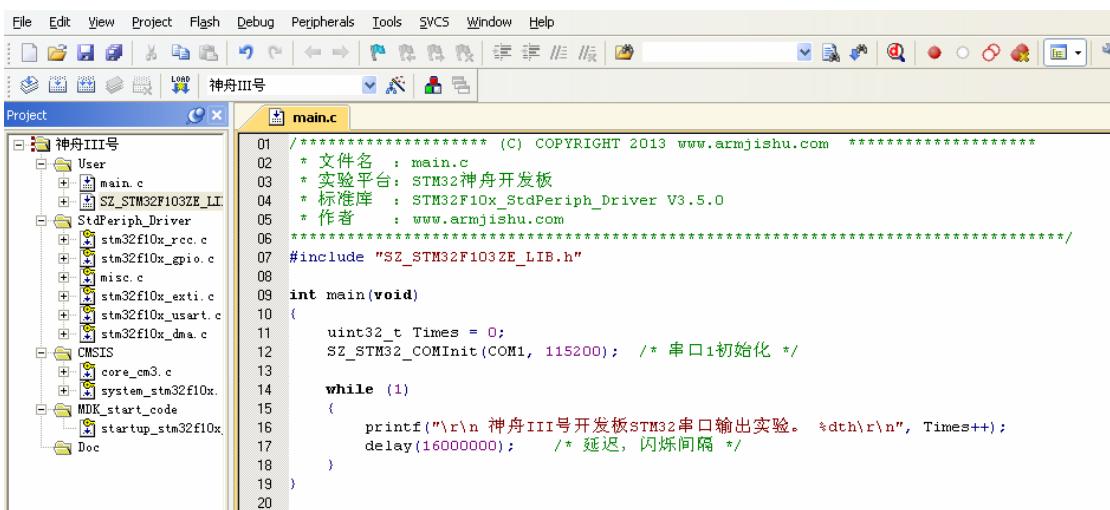
4.串口整体架构图



7.7.4 例程 01 UART串口1 Printf打印输出字符

7.7.5 软件设计

进入主函数；



代码分析 1: SZ_STM32_COMInit(COM1, 115200); 初始化串口 1，并且设置波特率为 115200，并对串口设置了许多参数

```

void SZ_STM32_COMInit(COM_TypeDef COM, uint32_t BaudRate)
{
    USART_InitTypeDef USART_InitStructure;

    USART_InitStructure.USART_BaudRate = BaudRate; // 串口的波特率，例如115200 最高达4.5Mbps/s
    USART_InitStructure.USART_WordLength = USART_WordLength_8b; // 数据字长度(8位或9位)
    USART_InitStructure.USART_StopBits = USART_StopBits_1; // 可配置的停止位-支持1或2个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No; // 无奇偶校验
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None; // 无硬件流控制
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; // 双工模式，使能发送和接收

    __SZ_STM32_COMInit(COM, &USART_InitStructure); // 调用STM32的USART初始化底层函数

    printf("\r\n ##### WWW.ARmjishu.COM! ##### \r\n");
}

```

代码分析 2: __SZ_STM32_COMInit(COM, &USART_InitStructure); 开始真正的对 USART 进行初始化的工作，这里与前面寄存器版本的初始化串口一样，唯一不同的是这里是库函数来做的。这里主要过程分为几步，我们分开来阐述：

代码分析 3: 第一步是 GPIO 初始化

1.首先先对 GPIO 初始化

GPIO 具有默认的复用功能，在使用它的复用功能的时候，我们首先要把相应的 GPIO 进行初始化。此时我们使用的 GPIO 的复用功能为串口，但为什么是 PA9 和 PA10 用作串口的 Tx 和 Rx，而不是其它 GPIO 引脚呢？这是从《STM32F103 数据手册》的引脚功能定义中查询到的。

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

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

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

复用功能模式设置

代码分析 4：第二步是 USART 的初始化

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructureUSART_BaudRate = BaudRate; //串口的波特率，例如115200 最高达4.5Mbps
USART_InitStructureUSART_WordLength = USART_WordLength_8b; //数据字长度(8位或9位)
USART_InitStructureUSART_StopBits = USART_StopBits_1; //可配置的停止位-支持1或2个停止位
USART_InitStructureUSART_Parity = USART_Parity_No; //无奇偶校验
USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None; //无硬件流控制
USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx; //双工模式，使能发送和接收

/* USART configuration */
/* 根据传入的参数初始化STM32的USART配置 */
USART_Init(COM_USART[COM], USART_InitStruct);

/* Enable USART */
/* 使能STM32的USART功能模块 */
USART_Cmd(COM_USART[COM], ENABLE);
```

初始化好 GPIO 之后，就要进行 USART1 的初始化，在库函数里就是填充 USART 的初始化结构体。这部分内容，是根据串口通讯协议来设置的。

1. USART_BaudRate = 115200;

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

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

2. `.USART_WordLength = USART_WordLength_8b;`

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

3. `.USART_StopBits = USART_StopBits_1;`

配置停止位。把通讯协议中的停止位设置为 1 位。

4. `.USART_Parity = USART_Parity_No ;`

配置奇偶校验位。本例程不设置奇偶校验位。

5. `.USART_HardwareFlowControl= USART_HardwareFlowControl_None;`

配置硬件流控制。不采用硬件流。

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

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

6. `USART_Mode = USART_Mode_Rx | USART_Mode_Tx;`

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

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

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

代码分析 5：第三步就是将 `printf()` 进行重定向，使得串口打印的数据可以从 `printf` 直接输出，这样就比较方便调试产品。

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

```
while (1)
{
    printf("\r\n 神舟III号开发板STM32串口输出实验。 %dth\r\n", Times++);
    delay(16000000); /* 延迟，闪烁间隔 */
}
```

```
##### WWW.ARMJISHU.COM! #####
```

神舟III号开发板STM32串口输出实验。 0th

神舟III号开发板STM32串口输出实验。 1th

神舟III号开发板STM32串口输出实验。 2th

神舟III号开发板STM32串口输出实验。 3th

神舟III号开发板STM32串口输出实验。 4th

神舟III号开发板STM32串口输出实验。 5th

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

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

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

```
/* 重定义fputc函数 如果使用MicroLIB只需要重定义fputc函数即可 */
int fputc(int ch, FILE *f)
{
    /* Place your implementation of fputc here */
    /* Loop until the end of transmission */
    while (USART_GetFlagStatus(SZ_STM32_COM1, USART_FLAG_TC) == RESET)
        {}

    /* e.g. write a character to the USART */
    USART_SendData(SZ_STM32_COM1, (uint8_t) ch);

    return ch;
}
```

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

这个代码中调用了两个 ST 库函数。USART_SendData() 和 USART_GetFlagStatus() 其说明见图下图：

```
void USART_SendData ( USART_TypeDef * USARTx,  
                      uint16_t          Data  
)
```

通过串口 x 发送一个数据

Transmits single data through the USARTx peripheral.

Parameters:

USARTx,: Select the USART or the UART peripheral. This parameter can be one of the following values: USART1, USART2, USART3, USART4 or USART5.

Data,: the data to transmit.

Return values:

None

Data 参数为将要发送的数据

```
FlagStatus USART_GetFlagStatus ( USART_TypeDef * USARTx,  
                                uint16_t          USART_FLAG  
                            )
```

检查 USART 的标志位

Checks whether the specified USART flag is set or not.

Parameters:

USARTx,: Select the USART or the UART peripheral. This parameter can be one of the following values: USART1, USART2, USART3, USART4 or USART5.
USART_FLAG,: specifies the flag to check. This parameter can be one of the following values:

这些为可输入的标志位参数
如 **USART_FLAG_TXE** 表示发送区是否为空的标志位。
USART_FLAG_TC 表示是否发送完成的标志位。

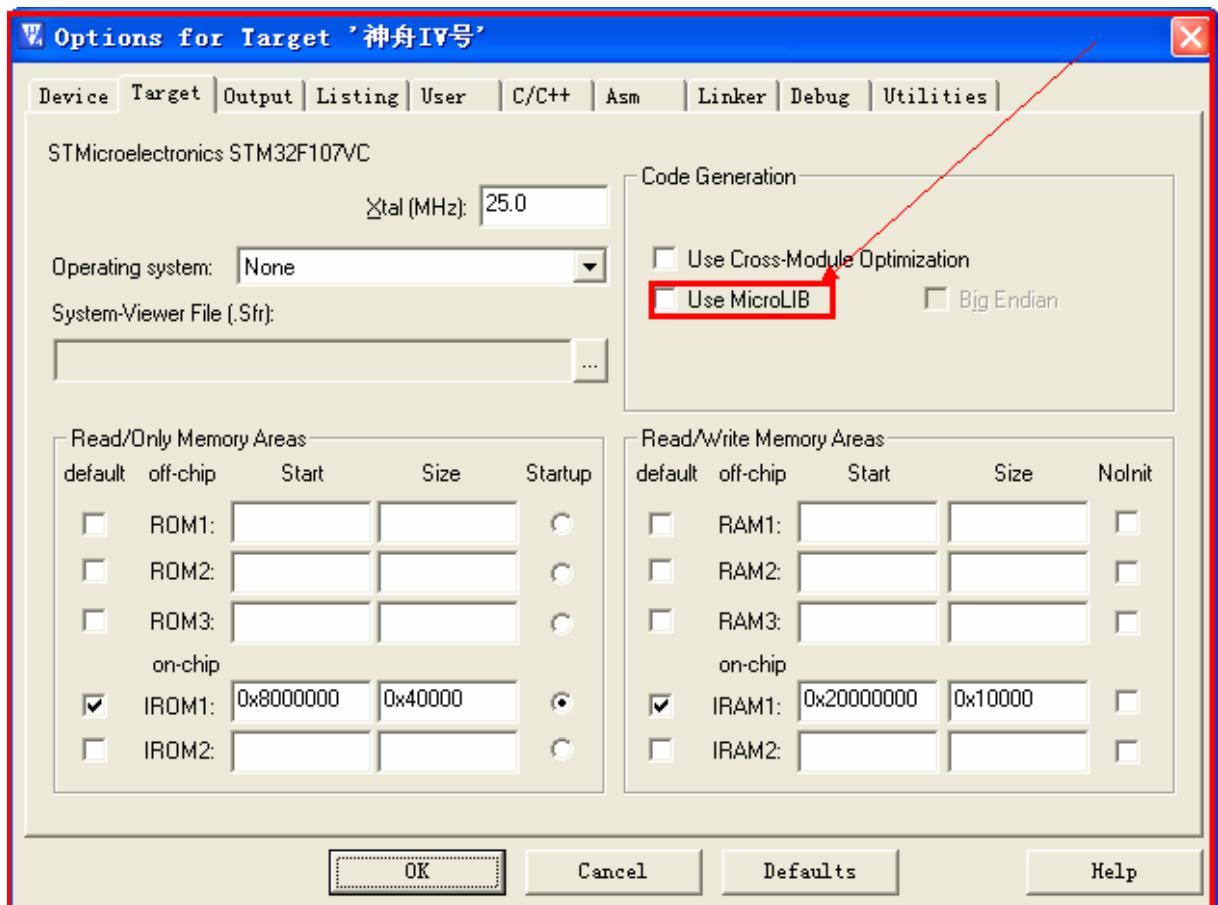
- USART_FLAG_CTS: CTS Change flag (not available for USART4 and USART5)
- USART_FLAG_LBD: LIN Break detection flag
- USART_FLAG_TXE: Transmit data register empty flag
- USART_FLAG_TC: Transmission Complete flag
- USART_FLAG_RXNE: Receive data register not empty flag
- USART_FLAG_IDLE: Idle Line detection flag
- USART_FLAG_ORE: OverRun Error flag
- USART_FLAG_NE: Noise Error flag
- USART_FLAG_FE: Framing Error flag
- USART_FLAG_PE: Parity Error flag

返回标志位检查结果。

Return values:

The new state of USART_FLAG (SET or RESET).

注意：这里有个 USR MicroLIB 库，如果没有效果可以对此处打钩，一般是不需要的。



7.7.6 下载与验证

如果使用JLINK下载固件，请按[3.3如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作。

如果使用串口下载固件，请按[3.4如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

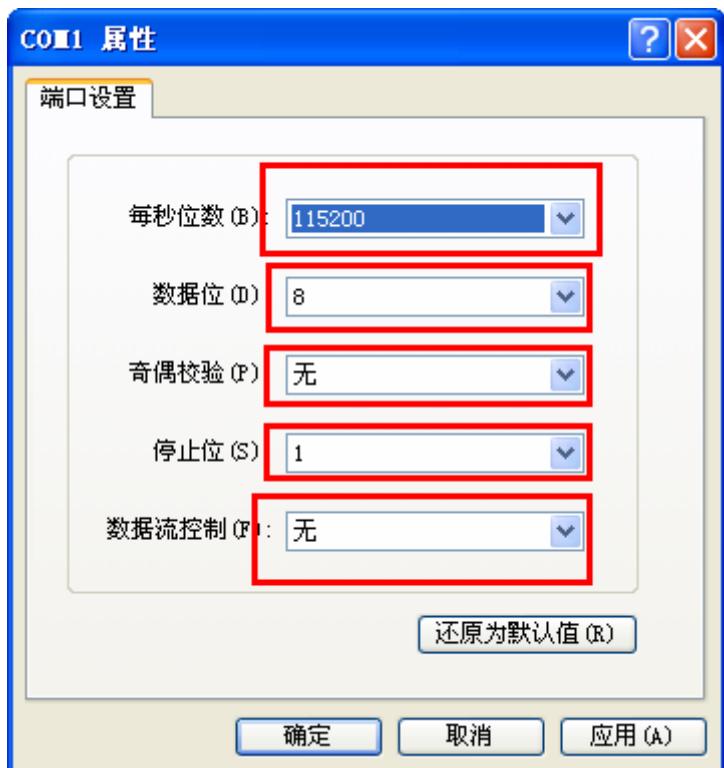
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.8如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

7.7.7 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 1 与电脑连接，并打开超级终端，按以下设置，如下图：

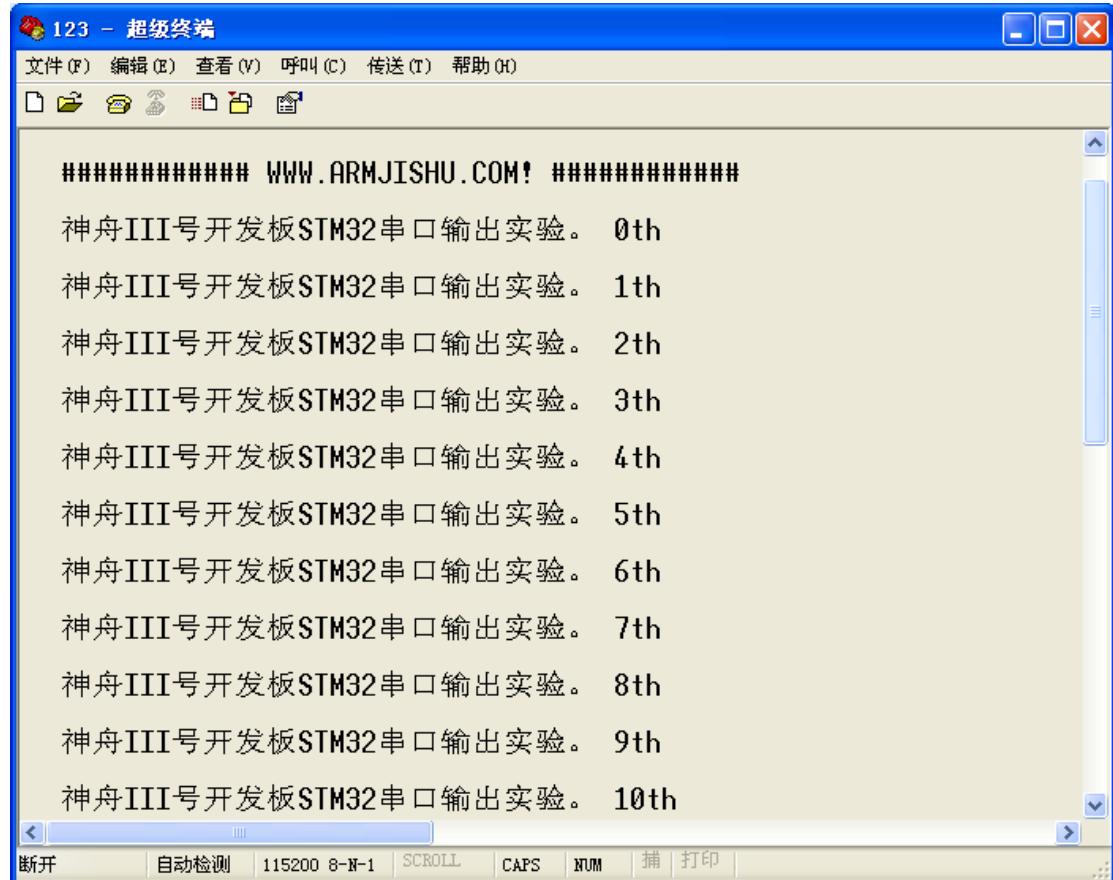


选择 COM1；按确定



再按确定，完成超级终端设置，重新打开电源

超级终端窗口显示信息，如下图



7.7.8 例程 02 UART串口1-带SYSTICK中断Printf()

7.7.9 软件设计

此处仅仅多了一个 SYSTICK 定时器来产生中断，在串口打印的同时还可以点亮 LED 灯。

7.7.10 实验现象

实验现象同上，不同的是可以看到板子上多两个 LED 灯在不停的闪烁。

7.7.11 例程 03 UART串口1Printf输出和scanf输入

7.7.12 软件设计

关键代码分析：

```
while (1)
{
    /* LED1指示灯状态取反 */
    LED1OBB = !LED1OBB;

    printf("\n\n\r 神舟III号开发板STM32串口输出实验。 %dth\r\n", Times++);

    printf("\n\r 请输入字符串，以空格或者回车结尾：");
    /* 注意串口1使用scanf时"SZ_STM32F103ZE_LIB.c"文件中fgetc定义中设备改为sz_STM32_COM1 */
    scanf("%s", ucStr);
    printf("\n\r 您输入的字符串是：<%s>", ucStr);

    printf("\n\r 请输入一个十进制的数字，以空格或者回车结尾：");
    num = 0;
    /* 注意串口1使用scanf时"SZ_STM32F103ZE_LIB.c"文件中fgetc定义中设备改为sz_STM32_COM1 */
    scanf("%d", &num);
    printf("\n\r 您输入的数字是：<%d>=<0x%X>", num, num);

    /* 此处可以添加用户的程序 */
}
```

代码分析 1: `scanf()`是一个 C 语言的标准库函数

`scanf` 函数，与 `printf` 函数一样，都被定义在 `stdio.h` 里，因此在使用 `scanf` 函数时要加上 `#include<stdio.h>`。它是格式输入函数，即按用户指定的格式从键盘上把数据输入到指定的变量之中，其关键字最末一个字母 f 即为“格式”(format)之意。

代码分析 2: `scanf("%s", ucStr);` 表示敲打键盘输入的字符会被自动转化成字符串和 `scanf("%d", &num);` 表示敲打键盘输入的字符会被自动转化数字。

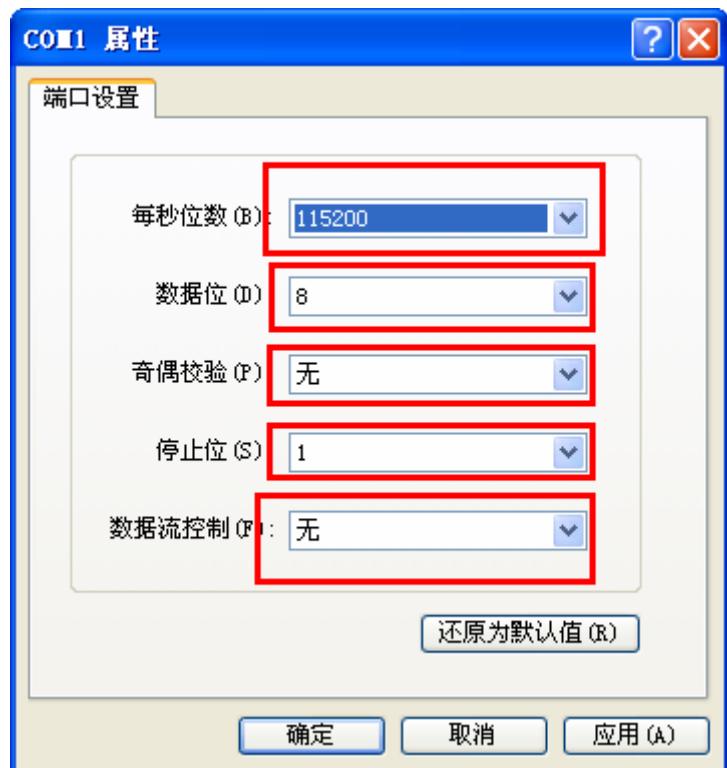
代码分析 3: 当执行到 `scanf()` 函数的时候，程序就会被阻塞在这里，当 `scanf()` 函数收到回车键的时候，就会把回车键之前输入的字符，都会输入到 `scanf()` 函数的变量中，比如这里是 `ucStr` 和 `num` 这 2 个变量参数，接下来的任务就把输入的值用作用户交互的一些事务了。

7.7.13 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 1 与电脑连接，并打开超级终端，按以下设置，如下图：



选择 COM1；按确定



再按确定，完成超级终端设置。

重新打开电源；神舟 III 号 STM32 开发板上 4 个 LED 灯闪一下，LED 2 和 LED 4 灯常亮，LED 3 在闪烁，完成一次数据输入后 LED 1 灯取反一次。

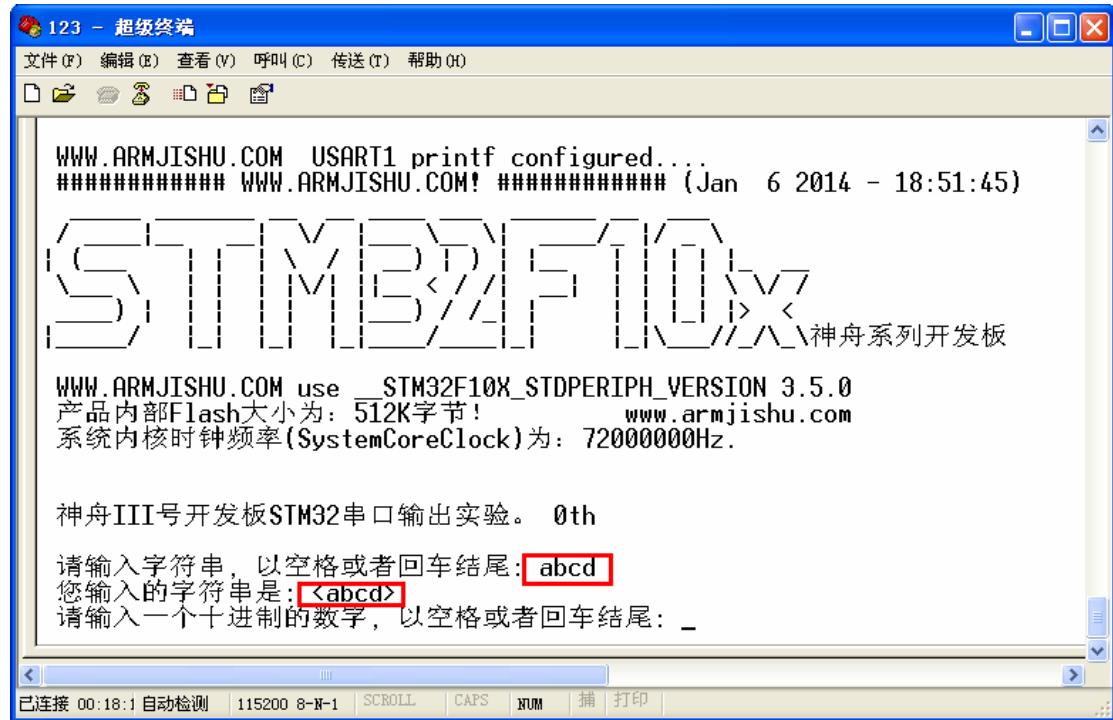
超级终端窗口显示信息，如下图



并按相关提示信息输入内容。

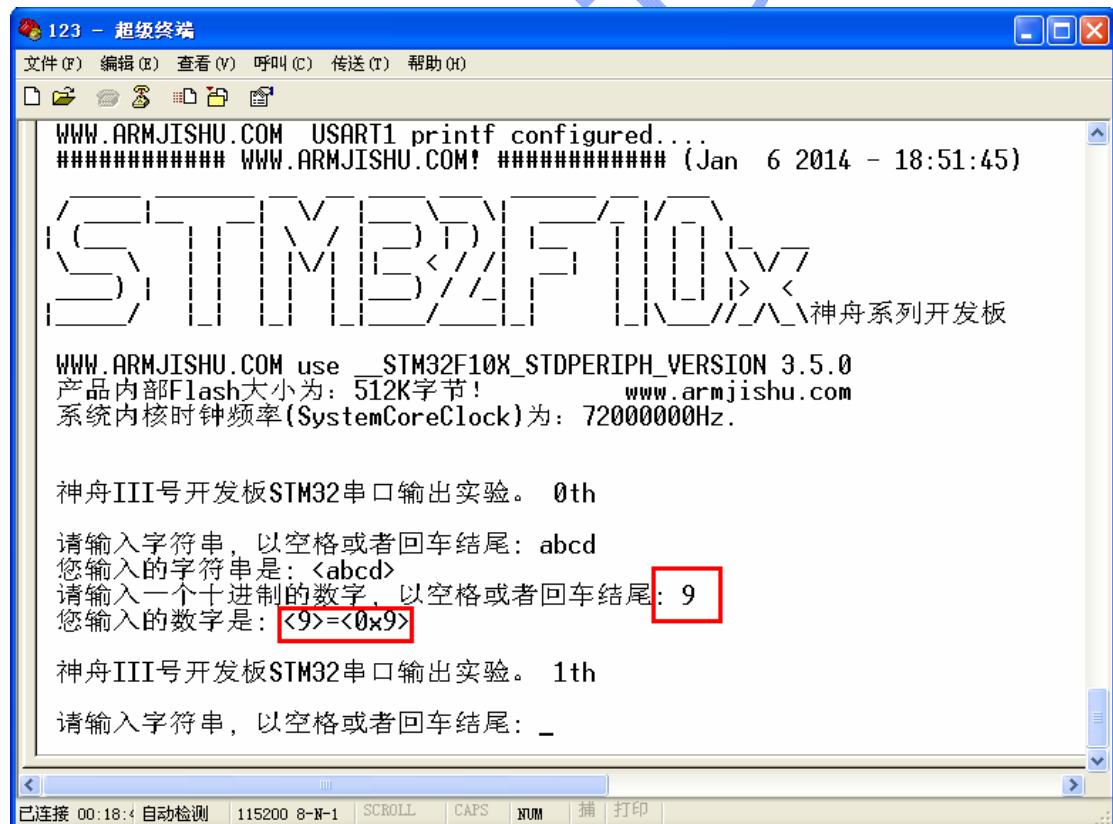
如输入字符串：abcd 并以空格或回车键结束；

显示结果，如图：



再输入一个数字：如输入 9；并以空格或回车键结束；

显示结果，如下图



以上就是串口 1 的输入输出实验。

7.8 USART串口2通信实验

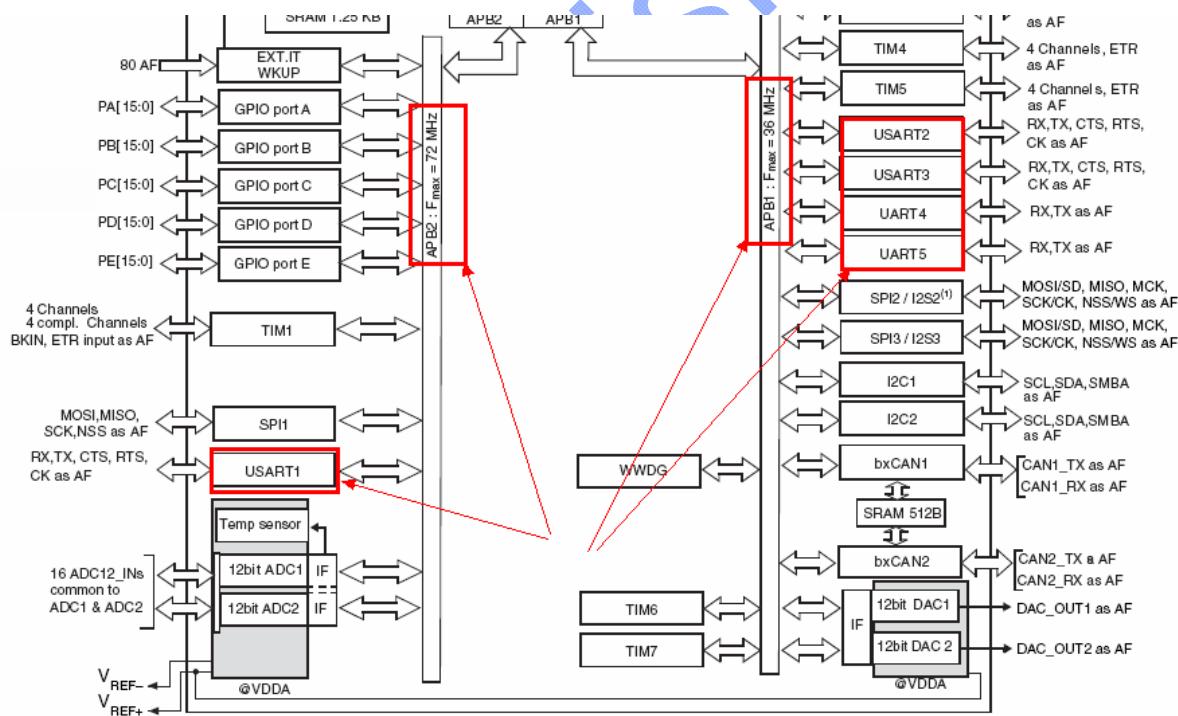
7.8.1 意义与作用

STM32 支持 5 个串口，有的设备有需要支持多个串口进行通信，比如一个串口不停发数据，一个串口不停接收数据；或者一个设备同时接收多路数据。

串口的使用对于我们开发调试过程中的作用是非常之大，可以用来查看，打印以及输入相关信息，是我们在嵌入式开发中最先与中央处理器通信的接口，学习好串口的功能，对于后续神舟III号的各个例程的调试具有至关重要作用。

STM32 的通用同步异步收发器(USART)是相当丰富的，功能也很强劲。最多可提供 5 路串口（神舟III号开发板使用的是 STM32F103ZET6，具有 5 个串口）。该 USART 有分数波特率发生器，发送和接收共用的可编程波特率，最高达 4.5Mbits/s；可编程数据字长度(8位或9位)；可配置的停止位-支持 1 或 2 个停止位；支持单线半双工通讯；支持 LIN；支持智能卡协议和 IrDA SIR ENDEC 规范（仅串口 3 支持）、具有 DMA 等。通过本节的学习，你将了解到 STM32 串口的基本使用方法、以及使用中断模式来实现串口收发的过程。

7.8.2 串口2与串口1的区别



这个图是从 STM32F103 数据手册中摘抄出来的，也就是说 USART1 是由 APB2 总线提供时钟，而 USART2~USART5 都是由 APB1 提供时钟，APB1 的时钟是 35MHz，比 APB2 的时钟要慢一半，所以 USART1 实际上可以在高速一点的波特率传输的时候要比 USART2~USART5 要稳定。

7.8.3 多个串口如何同时接收源源不断地发送给每个串口的数据

使用中断

7.8.4 两个串口同时来数据了 这个时候只能进入一个中断，会丢失数据吗

可以做到不丢失数据，原因如下：

- 1) 串口是低速设备，115200 的波特率对 stm32 来说也是挺慢的速度
- 2) 中断与数据传输覆盖的时间差

USART 串口数据传输是一种很慢的传输方式，以波特率为 38400 计算，假定数据帧格式为：8 个数据位、1 个停止位、没有奇偶检验位，则每个字节的传输时间至少为 260us，中断处理程序只要在这个时间之内从接受寄存器读出收到的数据，就不会造成数据丢失。

串口数据在下一个自己接收完成之前，接收 BUF 里的数据是不会被覆盖的。所以实际中断相应只要不超过 1 个字节的传输时间就没事。适当调整下各个中断处理程序，完全来得及

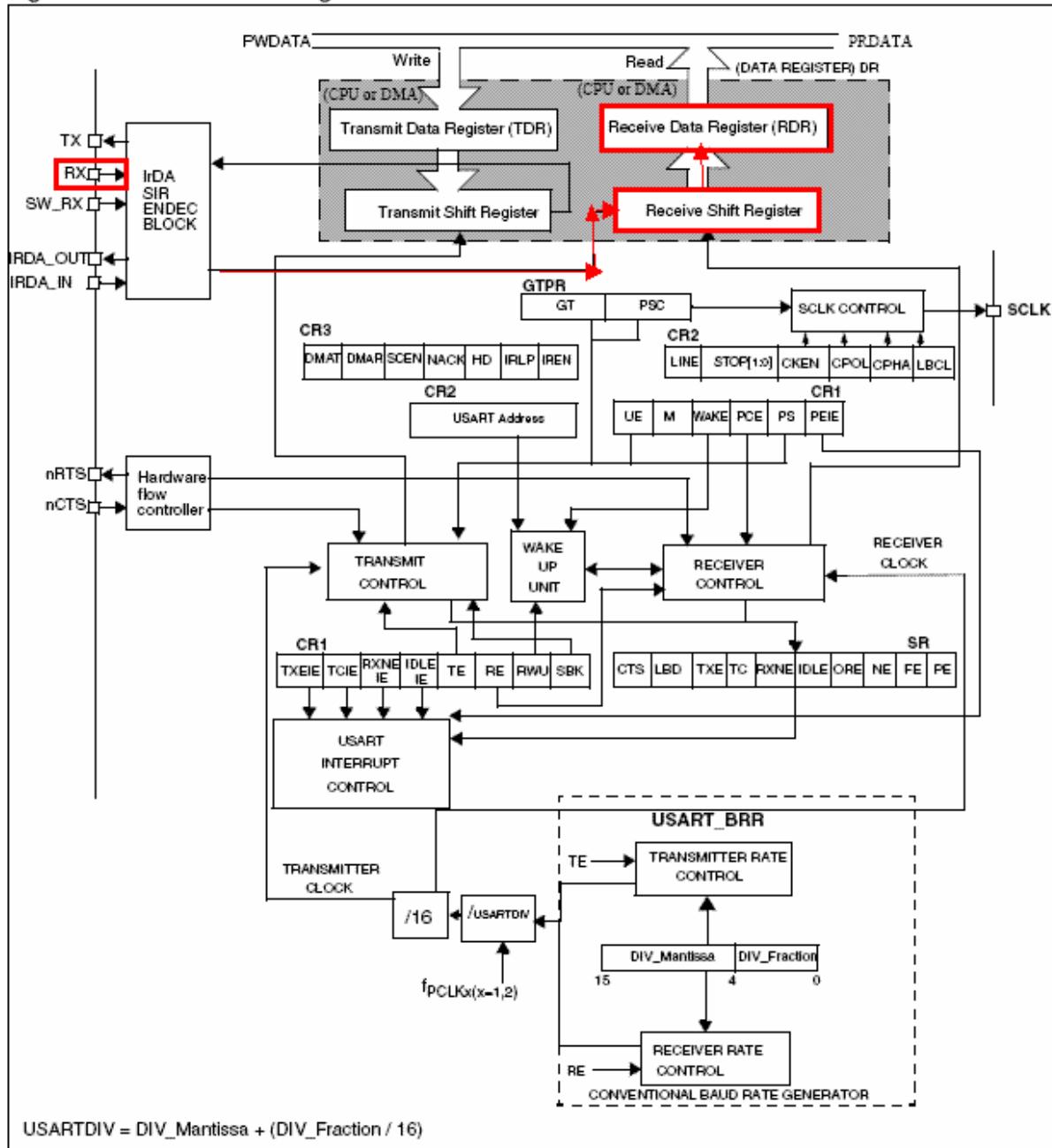
如果你有 2 个串口同时传输数据，则每个串口的接收中断处理程序只要能在 130us 内完成，就不会造成数据丢失。串口是低速设备，115200 的波特率对 stm32 来说也是挺慢的速度，同时中断，也就是微秒级延时，不至于丢数据。

- 3) STM32 的 USART 中移位寄存器和数据寄存器是分开的

下图是 STM32 USART 的框图，在图的右上方可以看到，从 USART_RX 到来的数据通过 Receive Shift Register 被转换为并行格式，只有当收到一个完整的字节后，Receive Shift Register 的内容才被送到 Receive Data Register (RDR)，程序可以通过 RDR 读出收到的数据字节。

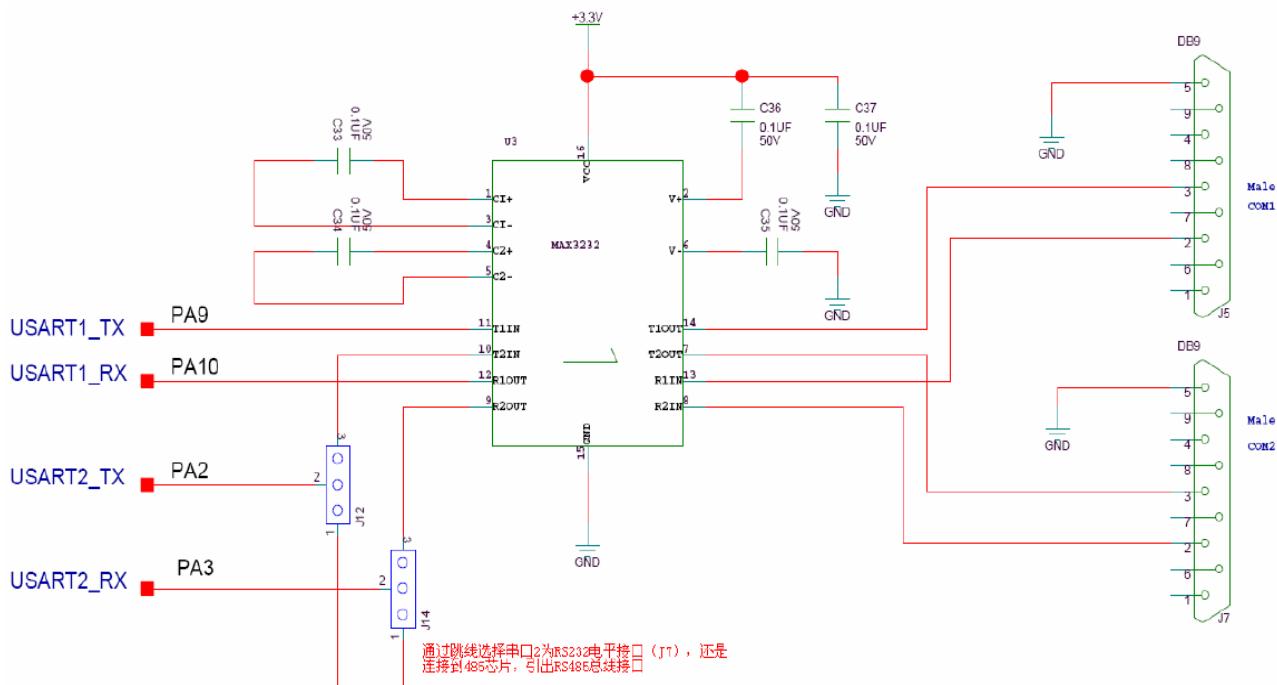
只要你的程序在下一个数据字节到来之前读出 Receive Data Register (RDR)，就不会产生数据丢失。

Figure 237. USART block diagram



7.8.5 硬件设计

由于处理器输出的是 TTL/COMS 电平，而 PC 串口为 RS-232 电平，所以硬件需要使用一颗电平转换芯片 MAX3232 实现双向电压转换。原理图如错误！未找到引用源。所示。由于串口外围电路很简单也很常见，再此就不做深入讲解了。



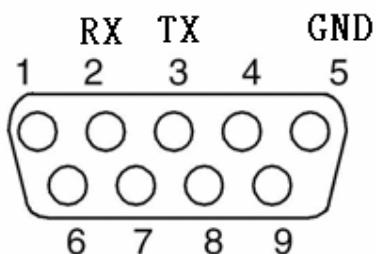
串口数据的输入和输出是站在处理器角度看的：

USART2_RX：接收数据串行输入。通过过采样技术来区别数据和噪音从而恢复数据。

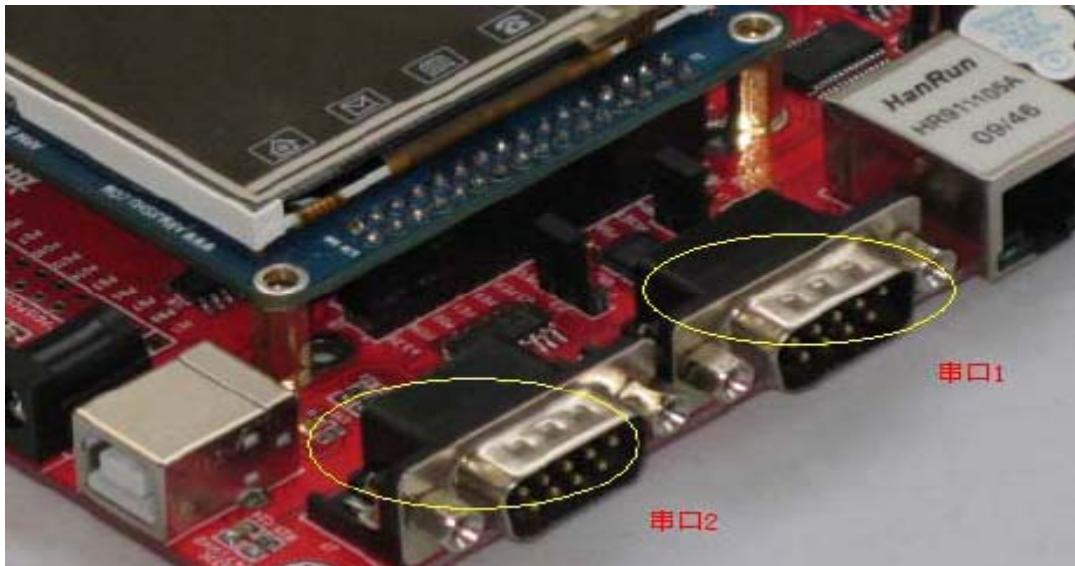
USART2_TX：发送数据输出。当发送器被禁止时，输出引脚恢复到它的I/O端口配置。当发送器被激活，并且不发送数据时，TX管脚处于高电平。

GND：数字地。为接收和发送信号提供参考地。

神舟III号开发板上有2个RS-232C串口，如图表8 神舟III号开发板的2个RS-232C串口所示。这两个RS-232C串口为DB9公头插针，其线序与PC电脑上的DB9公头插针相同，都为2脚输入到开发板，3脚输出，如所图表7 DB9公头线序及信号定义示。所以只要一根母到母交叉串口线就可以方便的将这两个串口连接起来，或者将开发板上的串口与PC机连接起来。



图表7 DB9公头线序及信号定义



图表 8 神舟 III 号开发板的 2 个 RS-232C 串口

| DB9管脚 | 功能描述 | DB9管脚 | 功能描述 |
|-------|-------------------|-------|------|
| 1 | NC | 6 | NC |
| 2 | USART1_PA10 开发板接收 | 7 | NC |
| 3 | USART1_PA9 开发板发送 | 8 | NC |
| 4 | NC | 9 | NC |
| 5 | GND | | |

表格 1 串口 1 (CN5) 信号定义

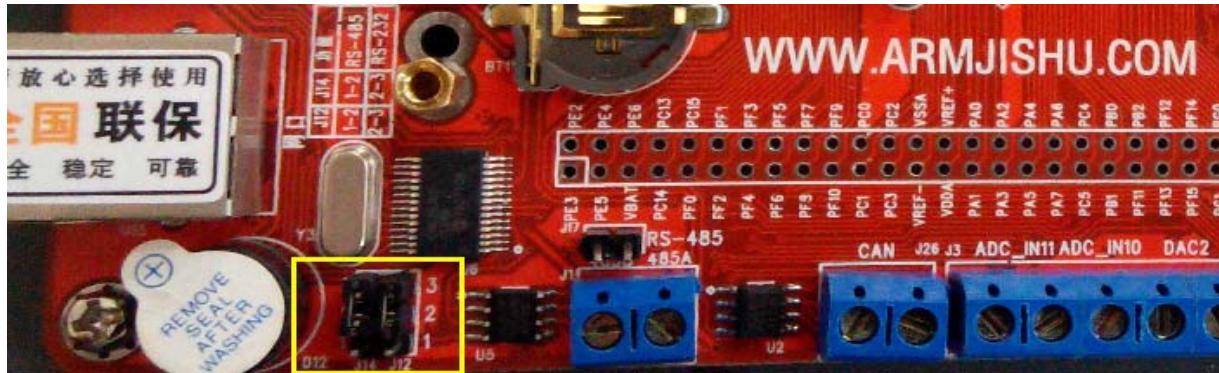
| DB9管脚 | 功能描述 | DB9管脚 | 功能描述 |
|-------|------------|-------|------|
| 1 | NC | 6 | NC |
| 2 | USART2_PA3 | 7 | NC |
| 3 | USART2_PA2 | 8 | NC |
| 4 | NC | 9 | NC |
| 5 | GND | | |

表格 2 串口 2 (CN6) 信号定义

注：由于串口1与USB的ID和VBUS信号复用，使用时需要改变跳线位置，所以我们推荐使用串口2，我们的示例也大都是在串口2输出信息。

由于 STM32 的管脚数量有限，所以其串口管脚为多功能复用管脚，为了不同功能硬件不相互影响，开发板上使用跳线来选择对应功能，在做此次试验前需要检查神舟 III 号开发板上的跳线位置是否正确。

如果连接串口 2（推荐， 默认出厂跳线既连接好了串口 2）， 则需要将 JP14 的 2↔3 用跳帽短接， JP12 的 2↔3 用跳帽短接， 串口 2 跳线位置图所示。



跳线连接正确后使用交叉母对母串口线连接神舟III号开发板的串口2（推荐）到PC或USB转串口线，下面可以开始软件设计了。

7.8.6 例程01 UART串口2Printf() 打印

大部分原理都通串口1相同，只有少部分不一样，实验现象也是相同

7.8.7 软件设计

主要有 3 点不同之处：

1) 在 __SZ_STM32_COMInit() 函数中，初始化的串口接收和发送的管脚是 PD5 和 PD6

```
#define __SZ_STM32_COM2_TX_GPIO_PORT GPIOA
```

2) 串口端口由 USART1 变成了 USART2

```
/* 串口2初始化 */
__SZ_STM32_COMInit(COM2, 115200);
```

3) 串口 USART2 的时钟总线变成了 APB1 总线

```
/* 使能STM32的USART的Clock时钟 */
RCC_APB1PeriphClockCmd(COM_USART_CLK[COM], ENABLE);
```

7.8.8 下载与验证

如果使用JLINK下载固件，请按3.2如何使用JLINK软件下载固件到神舟III号开发板小节进行操作。

如果使用串口下载固件，请按3.3如何通过串口下载一个固件到神舟III号开发板小节进行操作。

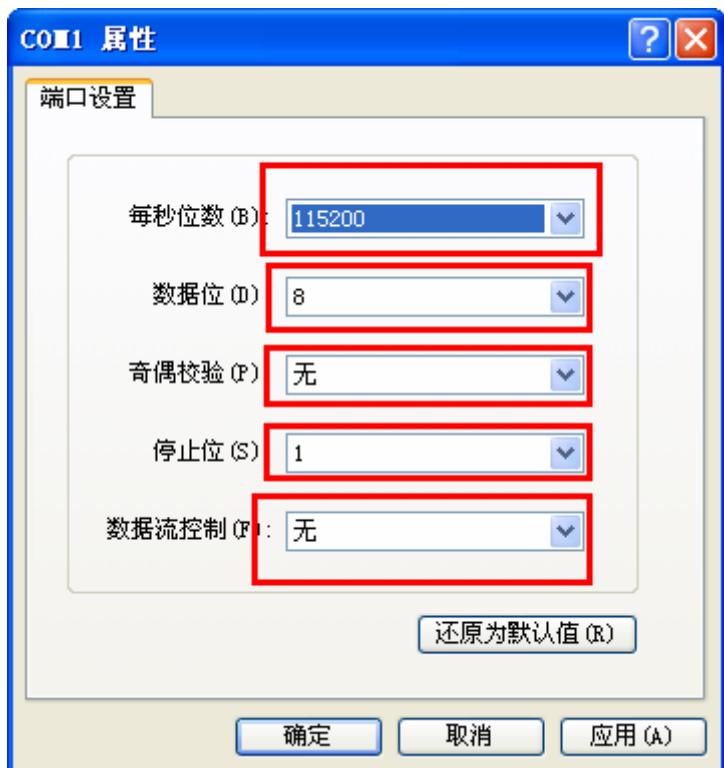
如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 3.5 如何在 MDK 开发环境中使用 JLINK 在线调试小节 进行操作。

7.8.9 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 2 与电脑连接，并打开超级终端，按以下设置，如下图：

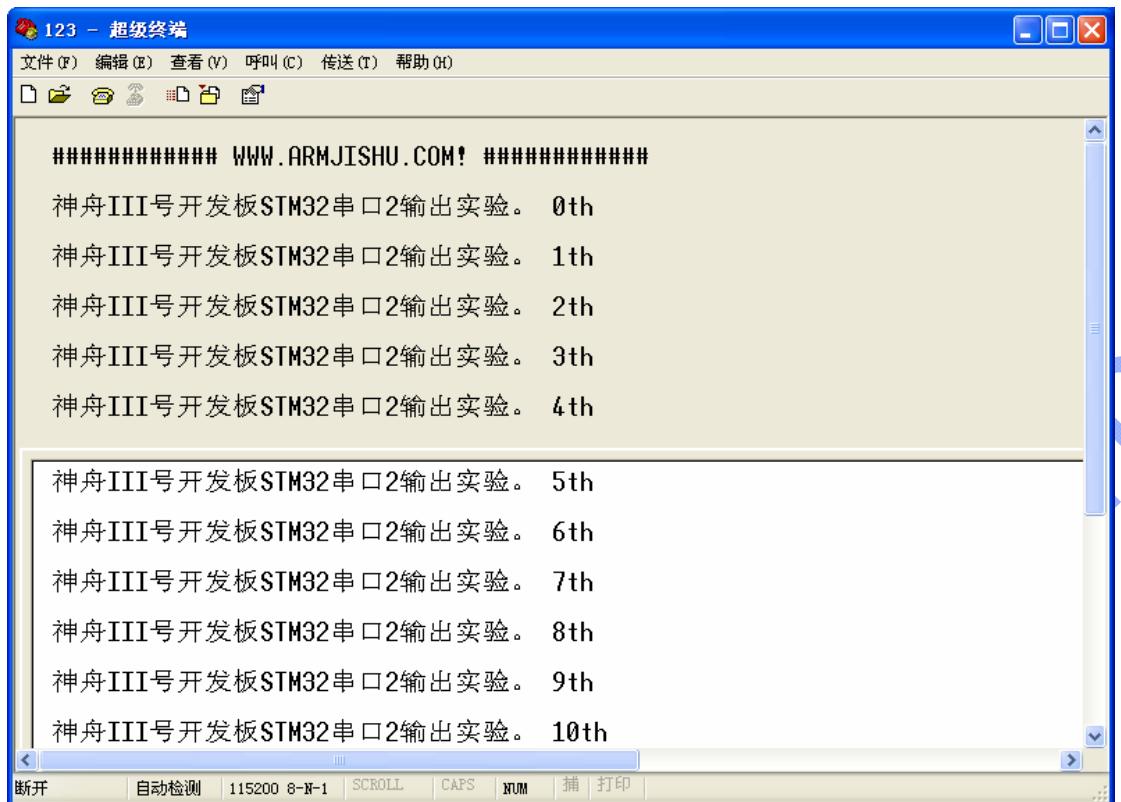


选择 COM1；按确定



再按确定，完成超级终端设置。

重新打开电源，神舟 III 号 STM32 开发板。超级终端窗口显示信息，如下图



7.8.10 例程 02 UART串口2-带SYSTICK中断Printf()

同串口 1 对应章节

7.8.11 例程 03 UART串口2Printf输出和scanf输入

同串口 1 对应章节

7.9 UART串口1和串口2同时格式化输出输入

7.9.1 下载与验证

如果使用JLINK下载固件，请按[3.2如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作。

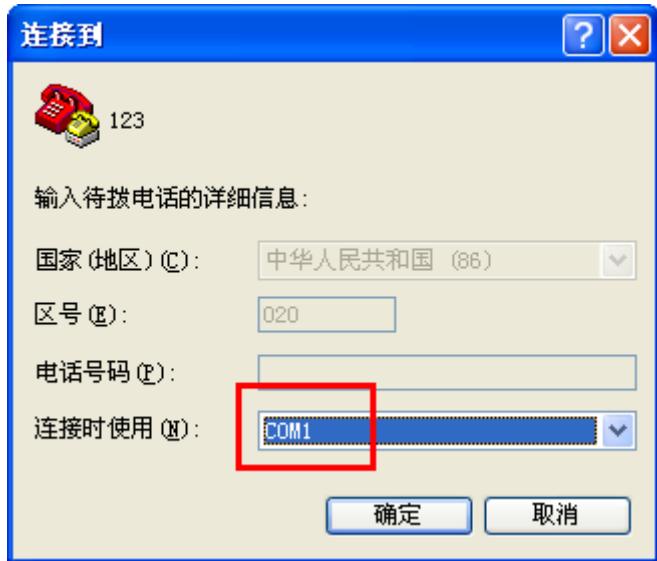
如果使用串口下载固件，请按[3.3如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

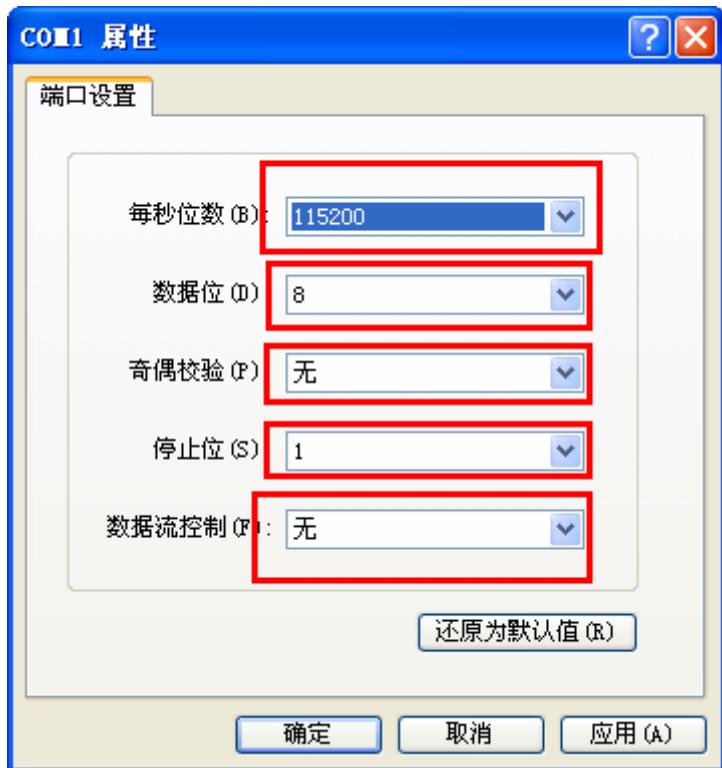
7.9.2 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 2 与电脑连接，

并打开超级终端，按以下设置，如下图：

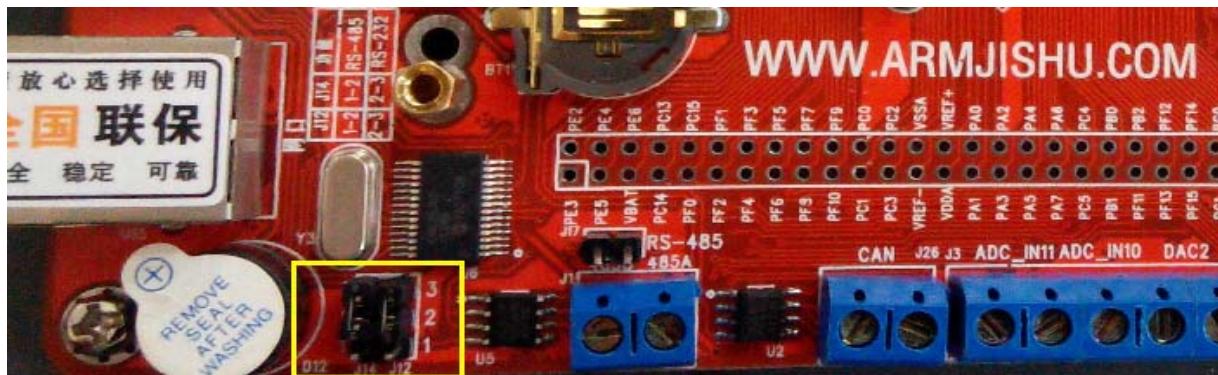


选择 COM1；按确定



再按确定，完成超级终端设置。

注意神舟 III 号开发板上的跳线短连接；

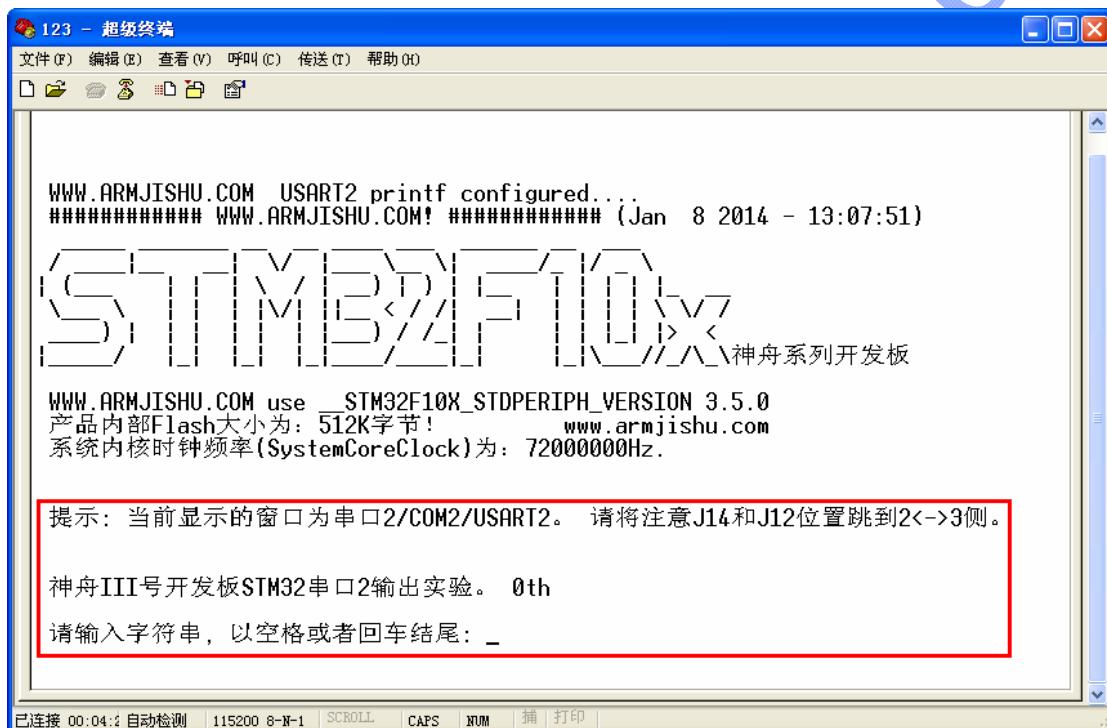


串口线连接开发板串口 2，重新打开电源，超级终端窗口显示详细信息：

提示：当前显示的窗口为串口 2/COM2/USART2。请将注意 J14 和 J12 位置跳到 2<->3 侧。

神舟 III 号开发板 STM32 串口 2 输出实验。

请输入字符串，以空格或者回车结尾：

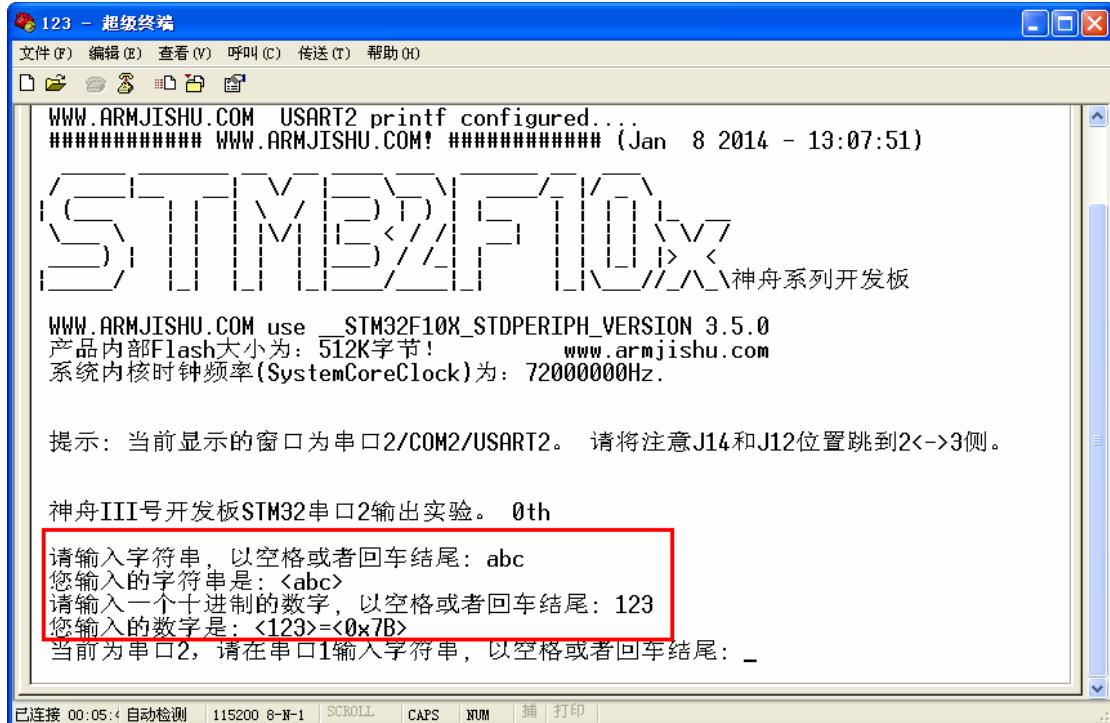


根据提示输入信息：

例如输入：abc 回车键

再输入：123 回车键

超级终端显示，以下图：

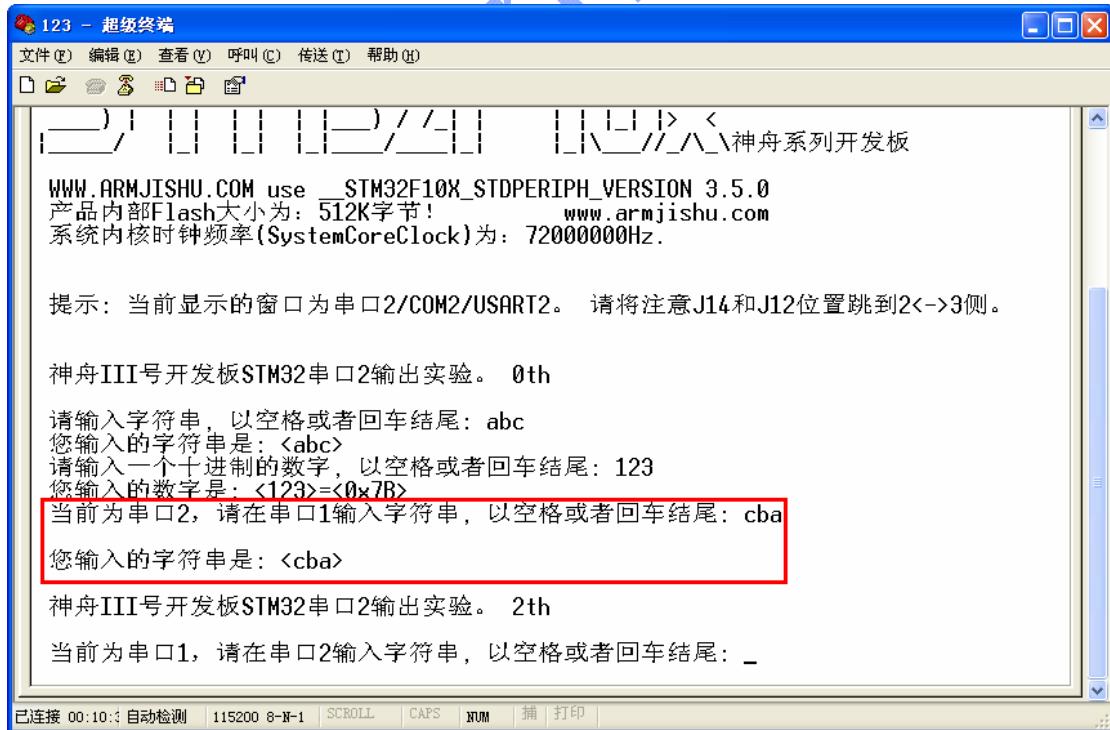


根据提示“当前为串口 2，请在串口 1 输入字符串，以空格或者回车结尾：”

接下来，切换串口 1；把串口线与开发板串口 1 连接；

并输入：cba 回车键

超级终端显示，以下图：



以上是 UART 串口 1 和串口 2 同时格式化输出输入实验。

7.10 串口高级例程之Printf中断接收实验

前面两节学习了STM32的常用接口UART串口的基本使用。之前的讲解都是串口的基本使用方法，效率比较低。本章节我们学习在神舟系列STM32开发板上**使用串口接收中断**实例。

7.10.1 意义与作用

目前串口仍然是MCU微控制器必不可少的接口之一。串口的使用对于我们开发调试过程中的作用是非常之大，可以用来查看，打印以及输入相关信息，是我们在嵌入式开发中最先与中央处理器通信的接口，学习好串口的功能，对于后续神舟III号的各个例程的调试具有至关重要作用。

鉴于前面章节已经对串口基础知识有了讲解，相关知识请参考实验4和实验5。

本章节我们学习**使用串口接收中断**实例。我们一起来研究如何使Printf函数实现0等待以及如何实现缩短串口接收中断处理速度。

现讲一段经历：本人曾经在调试板件系统时，需要将某一接口（标记为接口A）的数据消息打印出来以Debug。接口A的特点是数据收发具有突发性和不持续性，当时计算过串口在115200波特率下根本来不及打印接口A的突发数据，而且很担心使用Printf函数对系统造成不良影响甚至BUG不能重现。带着怀疑的态度添加了Printf函数来Debug，事实证明系统的运行几乎没有受到影响，而且串口打印的数据几乎没有丢失！！！！！我感到很是惊讶，直到一次调试中我断开了接口A的连接后，串口依然打印接口A的消息数据持续了十几秒，我对VxWorks操作系统的串口相关实现方法产生了浓厚的兴趣。相比之前调试Linux驱动时由于串口输出导致其他接口中断处理不及时而丢失产生了鲜明的对比。**该方法在后续的USB调试和以太网口程序调试时很实用。**

7.10.2 实验原理

本实验通过中断的方式接收数据。

7.10.3 硬件设计

请参阅 错误！未找到引用源。错误！未找到引用源。章节

7.10.4 软件设计

我们自顶向下剖析分解，首先是**主程序MAIN函数**：

```
17 int main(void)
18 {
19     /* USART1 config 115200 8-N-1 */
20     USART1_Config();
21     NVIC_Configuration();
22
23     printf("\r\n this is a USART Interrupt demo \r\n");
24
25     printf("\r\n*****串口接受中断实验*****\r\n");
26     printf("\r\n请开始输入字符串:\r\n");
27
28     for(;;)
29     {
30
31     }
32 }
```

主程序MAIN函数首先1、配置中断向量表，然后2、初始化串口，接着打印提示消息，最后进入While循环等待串口接收数据完成或缓存区溢出，将其打印。

代码分析1：初始化串口

根据宏定义初始化串口1参数，并使能串口1接收中断：

```
11 void USART1_Config(void)
12 {
13     GPIO_InitTypeDef GPIO_InitStructure;
14     USART_InitTypeDef USART_InitStructure;
15
16     /* config USART1 clock */
17     RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);
18
19     /* USART1 GPIO config */
20     /* Configure USART1 Tx (PA.09) as alternate function push-pull */
21     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
22     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
23     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
24     GPIO_Init(GPIOA, &GPIO_InitStructure);
25     /* Configure USART1 Rx (PA.10) as input floating */
26     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
27     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
28     GPIO_Init(GPIOA, &GPIO_InitStructure);
29
30     /* USART1 mode config */
31     USART_InitStructureUSART_BaudRate = 115200;
32     USART_InitStructureUSART_WordLength = USART_WordLength_8b;
33     USART_InitStructureUSART_StopBits = USART_StopBits_1;
34     USART_InitStructureUSART_Parity = USART_Parity_No;
35     USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
36     USART_InitStructureUSART_Mode = USART_Mode_RX | USART_Mode_TX;
37     USART_Init(USART1, &USART_InitStructure);
38     USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); // 高亮显示
39
40     USART_Cmd(USART1, ENABLE);
41 }
```

代码分析2：配置中断向量表

根据宏定义来使能串口1或串口2的中断：

```
44 void NVIC_Configuration(void)
45 {
46     NVIC_InitTypeDef NVIC_InitStructure;
47     /* Configure the NVIC Preemption Priority Bits */
48     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
49
50     /* Enable the USARTy Interrupt */
51     NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
52     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
53     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
54     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
55     NVIC_Init(&NVIC_InitStructure);
56 }
```

代码分析3：串口Printf发送的重定向

printf函数默认是输出到屏幕上，我们要让它从串口1/USART1输出，所以我们需要对fputc进行改写。

```

60  /*
61   * 函数名: fputc
62   * 描述  : 重定向c库函数printf到USART1
63   * 输入  : 无
64   * 输出  : 无
65   * 调用  : 由printf调用
66   */
67 int fputc(int ch, FILE *f)
68 {
69     /* 将Printf内容发往串口 */
70     USART_SendData(USART1, (unsigned char) ch);
71     while (!(USART1->SR & USART_FLAG_TXE));
72
73     return (ch);
74 }

```

通过 USART_SendData() 函数，将数据发送出去。其实现如下：

```

0584 /**
0585  * @brief Transmits single data through the USARTx peripheral.
0586  * @param USARTx: Select the USART or the UART peripheral.
0587  *   This parameter can be one of the following values:
0588  *   USART1, USART2, USART3, USART4 or USART5.
0589  * @param Data: the data to transmit.
0590  * @retval None
0591 */
0592 void USART_SendData(USART_TypeDef* USARTx, uint16_t Data)
0593 {
0594     /* Check the parameters */
0595     assert_param(IS_USART_ALL_PERIPH(USARTx));
0596     assert_param(IS_USART_DATA(Data));
0597
0598     /* Transmit Data */
0599     USARTx->DR = (Data & (uint16_t)0x01FF);
0600 }

```

通过对应的标志位判断发送是否完成：

```
while !(USART1->SR & USART_FLAG_TXE);
```

4、串口中断接收的实现，中断服务函数如下：

```

146 void USART1_IRQHandler(void)
147 {
148     u8 c;
149     if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
150     {
151         c=USART1->DR;
152         printf("%c", c);      //将接受到的数据直接返回打印
153     }
154 }
155 }

```

以发送，中断服务函数中首先通过函数 USART_GetITStatus(USART1, USART_IT_RXNE) 对中断的相关标志位进行判断，然后通过串口1的DR寄存器 (USART1->DR)，得到数据。最终通过Printf 函数输出到串口。

7.10.5 下载与验证

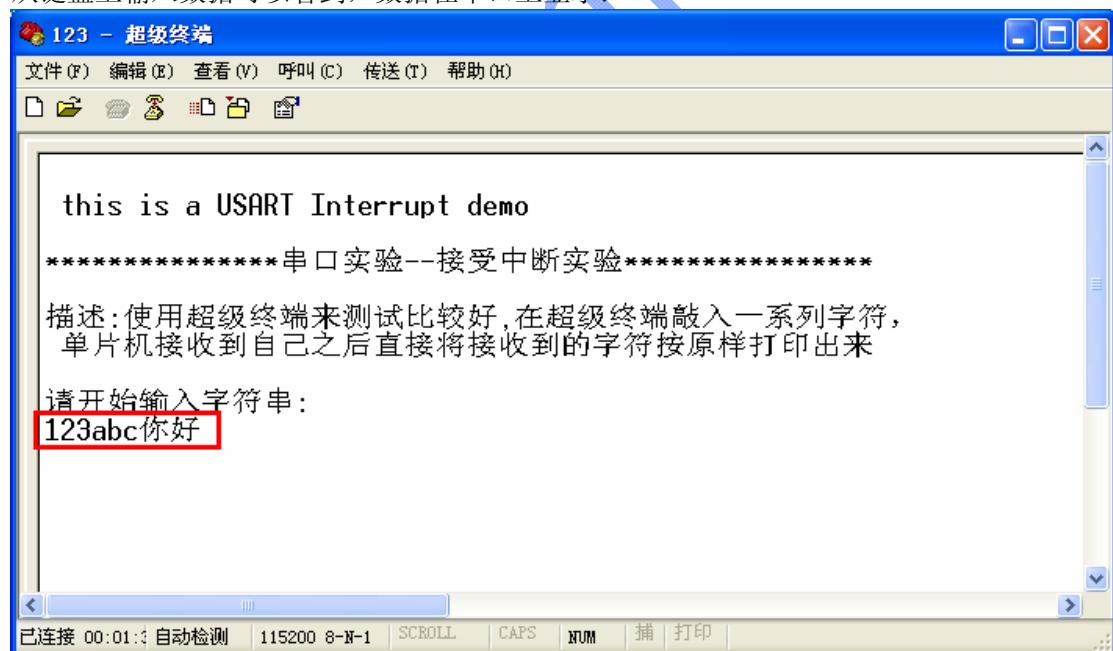
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.10.6 实验现象

将固件下载在神舟III号STM32开发板后，用随板配置的串口线连接神舟III号串口2与电脑的串口，打开超级终端，并按如下参数配置串口。



从键盘上输入数据可以看到，数据在串口上显示：



7.11 串口高级例程之Printf中断收发实验

前面两节学习了STM32的常用接口UART串口的基本使用。之前的讲解都是串口的基本使用方法，

效率比较低。本章节我们学习在神舟系列STM32开发板上**使用串口接收中断和发送中断以及FIFO缓存来实现高效率的串口数据收发**实例。

本次串口实例借鉴了著名的操作系统VxWorks操作系统的串口相关实现方法。做过驱动的人都知道，在驱动中频繁的使用Printf函数不但会降低系统的效率，而且可能会使系统崩溃。原因是Printf在串口打印输出是需要等待，中断中的长时间等待是致命的。所以**本章节我们一起来研究如何使Printf函数实现0等待，以及如何实现缩短串口接收中断处理速度**。

7.11.1 意义与作用

目前串口仍然是MCU微控制器必不可少的接口之一。串口的使用对于我们开发调试过程中的作用是非常之大，可以用来查看，打印以及输入相关信息，是我们在嵌入式开发中最先与中央处理器通信的接口，学习好串口的功能，对于后续神舟III号的各个例程的调试具有至关重要作用。

鉴于前面章节已经对串口基础知识有了讲解，相关知识请参考实验4和实验5。

本章节我们学习**使用串口接收中断和发送中断以及FIFO缓存来实现高效率的串口数据收发**实例。我们一起来研究如何使Printf函数实现0等待以及如何实现缩短串口接收中断处理速度。

现讲一段经历：本人曾经在调试板件系统时，需要将某一接口（标记为接口A）的数据消息打印出来以Debug。接口A的特点是数据收发具有突发性和不持续性，当时计算过串口在115200波特率下根本来不及打印接口A的突发数据，而且很担心使用Printf函数对系统造成不良影响甚至BUG不能重现。带着怀疑的态度添加了Printf函数来Debug，事实证明系统的运行几乎没有受到影响，而且串口打印的数据几乎没有丢失！！！！！我感到很是惊讶，直到一次调试中我断开了接口A的连接后，串口依然打印接口A的消息数据持续了十几秒，我对VxWorks操作系统的串口相关实现方法产生了浓厚的兴趣。相比之前调试Linux驱动时由于串口输出导致其他接口中断处理不及时而丢失产生了鲜明的对比。本章节我们以类似的方法在神舟系列STM32开发板上实现**串口接收中断和发送中断以及FIFO缓存来实现高效率的串口数据收发**。

该方法在后续的USB调试和以太网口程序调试时很实用。

7.11.2 实验原理

本实验的核心是：

对于**串口 Printf 输出**，为了做到使 Printf 函数实现 0 等待，我们在底层使用 Buffer 缓存来自 Printf 的数据，Buffer 按照 FIFO 先入先出的结构组织。当 Printf 有数据发送时，其底层的发送函数并不是真正的串口发送，而只是将数据写入 FIFO 并使能串口发送中断后便退出，这个操作时 RAM 的读写操作，执行速度很快，所以不需要等待，唯一可能影响速度的是 FIFO 的深度，如果 FIFO 较小而实际要打印的数据量很大则 FIFO 填满以后需要等待，所以可以根据实际情况设置 FIFO 的深度。本次在神舟系列 STM32 开发板上的实验中发送缓存 FIFO “**USART_Tx_Buffer**”的深度为 256 字节，用户可以自己修改其大小。而实际的发送数据是在串口发送中断中完成的。

对于**串口输入**，我们在在中断中使用 Buffer 缓存串口收到的数据，默认是收到回车符时或者收到的数据填满缓冲区时通过全局变量“**USART_Rx_Done**”通知前台程序。这样中断里的程序简单则处理速度快，中断可以快速返回。本次在神舟系列 STM32 开发板上的实验中接收缓存区**USART_Rx_Buffer**的大小为 20 字节，用户可以自己修改其大小。

7.11.3 硬件设计

请参阅 错误！未找到引用源。错误！未找到引用源。章节

由于 STM32 的管脚数量有限，所以其串口管脚为多功能复用管脚，为了不同功能硬件不相互影响，开发板上使用跳线来选择对应功能，在做此次试验前需要检查神舟 III 号开发板上的跳线位置是否正确。

如果连接串口 2（推荐， 默认出厂跳线既连接好了串口 2）， 则需要将 J12 和 J14 的 2 \leftrightarrow 3 用跳帽短接；

跳线连接正确后使用交叉母对母串口线连接神舟III号开发板的串口2（推荐）到PC或USB转串口线， 下面可以开始软件设计了。

7.11.4 软件设计

这一章节我们学习**使用串口接收中断和发送中断以及FIFO缓存来实现高效率的串口数据收发**实例。我们一起来研究如何使Printf函数实现0等待以及如何实现缩短串口接收中断处理速度。

本实验设计灵活，可以支持串口1或者串口2，只需修改“**Printf.h**”文件的以下语句：

```
/* WWW.ARmjishu.COM 使用#define USE_USART2则打印输出在神舟开发板的串口2*/
#define USE_USART2
#ifndef USE_USART2
```

以上定义编译结果为使用神舟系列STM32开发板上的串口2（推荐）。

```
/* WWW.ARmjishu.COM 使用#define USE_USART2则打印输出在神舟开发板的串口2*/
#ifndef USE_USART2
#define USE_USART2
```

以上定义编译结果为使用神舟系列STM32开发板上的串口1。

关于 STM32 的 USART 的库函数实现，主要是 在 STM32F10x_StdPeriph_Driver 库的“stm32f10x_usart.c”和“stm32f10x_usart.h”两个文件里。以V3.5.0版本的库为例，它们位于“STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\Stm32F10x_StdPeriph_Driver”目录的“src”和“inc”文件夹里，前面章节已经有所讲解，本章节不再讲述。

我们自顶向下剖析分解，首先是**主程序MAIN函数**：

```
int main (void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
    */
    /* 配置中断向量表 */
    NVIC_Configuration();

    /* 初始化串口 */
    /* ARMJISHU.COM提示修改"Printf.h"中宏定义选择串口2或者串口1 */
    Printf_Init();

    /* 初始化LED指示灯 */
    STM_EVAL_LEDInit(LED1);
    /* Turn on LED1 */
    STM_EVAL_LEDOn(LED1);

    printf ("\r\n WWW.ARmjishu.COM 神舟系列STM32开发板 串口printf输出中断接收实验");
    printf ("\r\n ARMJISHU.COM 经典高效率的串口实例，串口收发均在后台中断中完成");
    printf ("\r\n 等待的Printf串口打印函数将数据写入Buffer后立即返回(Buffer未满时)");
    printf ("\r\n 输入字符串后按回车或者输入超过20个字符显示刚才的输入,支持中文!\r\n");
    Delay_ARMJISHU(200);

    while (1)
    {
        /* 等待串口接收数据完成或缓存区溢出 */
        if (USART_Rx_Done == 1)
        {
            printf ("\n\r欢迎使用神舟系列STM32开发板 You input string is:\n\r");
            printf ("%s\n\r", USART_Rx_Buffer);
            USART_Rx_Buffer_Clear();
        }
    }
} ? end main ?
```

主程序MAIN函数首先1、配置中断向量表，然后2、初始化串口，接着打印提示消息，最后进入While循环等待串口接收数据完成或缓存区溢出，将其打印。

1、配置中断向量表

根据宏定义来使能串口1或串口2的中断：

```

void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Set the Vector Table base address at 0x08000000 */
    NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x00000);

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

#ifdef USE_USART2
    NVIC_InitStructure.NVIC_IRQChannel = EVAL_COM1_IRQn;
#else
    NVIC_InitStructure.NVIC_IRQChannel = EVAL_COM2_IRQn;
#endif
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

    NVIC_Init(&NVIC_InitStructure);
} ? end NVIC_Configuration ?

```

2、初始化串口

根据宏定义初始化串口1或串口2的参数，并使能接收和发送中断：

```

#ifdef USE_USART2
    #define EVAL_COMX           EVAL_COM1
    #define EVAL_COMX_STR        EVAL_COM1_STR
#else
    #define EVAL_COMX           EVAL_COM2
    #define EVAL_COMX_STR        EVAL_COM2_STR
#endif

void Printf_Init(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
    */

    /* USARTx configured as follow:
       - BaudRate = 115200 baud
       - Word Length = 8 Bits
       - One Stop Bit
       - No parity
       - Hardware flow control disabled (RTS and CTS signals)
       - Receive and transmit enabled
    */
    USART_InitStructureUSART_BaudRate = 115200;
    USART_InitStructureUSART_WordLength = USART_WordLength_8b;
    USART_InitStructureUSART_StopBits = USART_StopBits_1;
    USART_InitStructureUSART_Parity = USART_Parity_No;
    USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx;

#ifdef USE_USART2
    STM_EVAL_COMInit(COM1, &USART_InitStructure);
#else
    STM_EVAL_COMInit(COM2, &USART_InitStructure);
#endif

    /* Enable the EVAL_COM1 Transmtoit interrupt: this interrupt is generated when the
       EVAL_COM1 transmit data register is empty */
    USART_ITConfig(EVAL_COMx, USART_IT_TXE, ENABLE);

    /* Enable the EVAL_COM1 Receive interrupt: this interrupt is generated
       when the EVAL_COM1 receive data register is not empty */
    USART_ITConfig(EVAL_COMx, USART_IT_RXNE, ENABLE);

```

3、串口Printf发送的实现

首先，完成串口发送单个字节的函数，在“**Printf.c**”文件的有如下的宏定义，这是为了兼容不同的编译平台。

```

#ifdef __GNUC__
    /* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
       set to 'Yes') calls __io_putchar() */
    #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
    #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

```

图表 9 串口 PUTCHAR_PROTOTYPE 宏定义

其实现如下：

```
PUTCHAR_PROTOtYPE
{
    /* ARMJISHU 等待的Printf串口打印函数将数据
       写入buffer后立即返回(buffer未满时) */
    USART_StoreBufferData((uint8_t) ch);

    return ch;
}
```

其“USART_StoreBufferData”函数的实现如下：

```
/**
 * @brief put char to the Tx Buffer
 * 将来自printf的数据写入buffer,
 * @param None
 * @retval None
 * @by WWW.ARMJISHU.COM
 */
void USART_StoreBufferData(uint8_t ch)
{
    while (TxCounter == USART_TX_DATA_SIZE);
    if (TxCounter < USART_TX_DATA_SIZE)
    {
        /* Write one byte to the transmit data register */
        USART_Tx_Buffer[USART_Tx_ptr_Store++] = ch;
        USART_Tx_ptr_Store = USART_Tx_ptr_Store & 0xFF;
        TxCounter++;
    }
    USART_ITConfig(EVAL_COMx, USART_IT_TXE, ENABLE);
}
```

原理：对于串口 Printf 输出，为了做到使 Printf 函数实现 0 等待，我们在底层使用 Buffer 缓存来自 Printf 的数据，Buffer 按照 FIFO 先入先出的结构组织。当 Printf 有数据发送时，其底层的发送函数并不是真正的串口发送，而只是将数据写入 FIFO 并使能串口发送中断后便退出，这个操作时 RAM 的读写操作，执行速度很快，所以不需要等待，唯一可能影响速度的是 FIFO 的深度，如果 FIFO 较小而实际要打印的数据量很大则 FIFO 填满以后需要等待，所以可以根据实际情况设置 FIFO 的深度。本次在神舟系列 STM32 开发板上的实验中发送缓存 FIFO “USART_Tx_Buffer”的深度为 256 字节，用户可以自己修改其大小。而实际的发送数据是在串口发送中断中完成的。

```
#define USART_TX_DATA_SIZE      0xFF

uint8_t USART_Tx_Buffer [USART_TX_DATA_SIZE+1];
__IO uint32_t USART_Tx_ptr_Out = 0;
__IO uint32_t USART_Tx_ptr_Store = 0;
__IO uint32_t TxCounter = 0;
```

4、串口接收的实现

串口接收相关主要是在串口接收中断中调用“USART_GetInputString”函数接收并缓存串口数据，支持退格键：

```

/** 
 * @brief Get Input string from the HyperTerminal
 * 将串口收到的数据写入Buffer,
 * 收到回车或长度超范围通知前台程序
 * @param None
 * @retval None
 * @by WWW.ARmjishu.COM
 */
void USART_GetInputString(void)
{
    uint8_t mychar = 0;

    mychar = USART_ReceiveData(EVAL_COMx);
    if (USART_Rx_Done == 0)
    {
        if (mychar == '\r')
        {
            USART_Rx_Buffer[USART_Rx_ptr_in] = '\0';
            USART_Rx_Done = 1;
        }
        else if (mychar == '\b') /* Backspace */
        {
            if (USART_Rx_ptr_in > 0)
            {
                printf("\b \b");
                USART_Rx_ptr_in--;
            }
        }
        else
        {
            USART_Rx_Buffer[USART_Rx_ptr_in++] = mychar;
            printf("%c", mychar); /* 回显字符 */
        }

        if (USART_Rx_ptr_in >= (USART_RX_DATA_SIZE - 1))
        {
            /**
             USART_Rx_Buffer[USART_Rx_ptr_in] = '\0';
             USART_Rx_Done = 1;
            */
        }
    }
}

```

原理：对于**串口输入**，我们在在中断中使用 Buffer 缓存串口收到的数据，默认是收到回车符时或者收到的数据填满缓冲区时通过全局变量“**USART_Rx_Done**”通知前台程序。这样中断里的程序简单则处理速度快，中断可以快速返回。本次在神舟系列 STM32 开发板上的实验中接收缓存区 **USART_Rx_Buffer** 的大小为 20 字节，用户可以自己修改其大小。

注意上述函数中变量“**USART_Rx_Done**”与前台 MAIN 函数中语句的对应：

```

while (1)
{
    /* 等待串口接收数据完成或缓存区溢出 */
    if (USART_RX_Done == 1)
    {
        printf("\n\r欢迎使用神舟系列STM32开发板 You input string is:\n\r");
        printf("[%s]\n\r", USART_Rx_Buffer);
        USART_RX_Buffer_Clear();
    }
}

```

5、串口中断处理的实现

首先，根据宏定义决定是使用串口 1 中断还是或串口 2 中断

```

#ifdef USE_USART2
#define USARTx_IRQHandler USART2_IRQHandler
#else
#define USARTx_IRQHandler USART1_IRQHandler
#endif

#ifdef USE_USART2
#define USARTx USART2
#else
#define USARTx USART1
#endif

```

串口1中断还是或串口2中断统一如下：

```
void USARTx_IRQHandler(void)
{
    if (USART_GetITStatus(USARTx, USART_IT_RXNE) != RESET)
    {
        /* received data */
        USART_GetInputString();
    }

    /* If overrun condition occurs, clear the ORE flag
       and recover communication */
    if (USART_GetFlagStatus(USARTx, USART_FLAG_ORE) != RESET)
    {
        (void)USART_ReceiveData(USARTx);
    }

    if (USART_GetITStatus(USARTx, USART_IT_TXE) != RESET)
    {
        /* Write one byte to the transmit data register */
        USART_SendBufferData();
    }
} // end USARTx_IRQHandler
```

前面已经讲解了在串口接收中断中调用的“USART_GetInputString”函数。

对于发送：前面“[3、串口Printf发送的实现](#)”讲解时说：而实际的发送数据是在串口发送中断中完成的。如果发送缓冲区非空则发送数据，否则如果发送缓冲区已经为空则禁止串口发送中断（发送中断将在缓冲区非空时由“USART_StoreBufferData”打开），“USART_SendBufferData”函数的具体实现如下：

```
/** 
 * @brief put char to the HyperTerminal in the interrupt
 *        在串口中断中发送Buffer中的数据, Buffer空时关闭中断
 * @param None
 * @retval None
 * @by   WWW.ARMJISHU.COM
 */
void USART_SendBufferData(void)
{
    if (TxCounter > 0)
    {
        /* Write one byte to the transmit data register */
        USART_SendData(EVAL_COMx, USART_Tx_Buffer[USART_Tx_ptr_Out++]);
        TxCounter--;
        USART_Tx_ptr_Out = USART_Tx_ptr_Out & 0xFF;
    }
    else
    {
        /* Disable the EVAL_COM1 Transmit interrupt */
        USART_ITConfig(EVAL_COMx, USART_IT_TXE, DISABLE);
    }
}
```

7.11.5 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

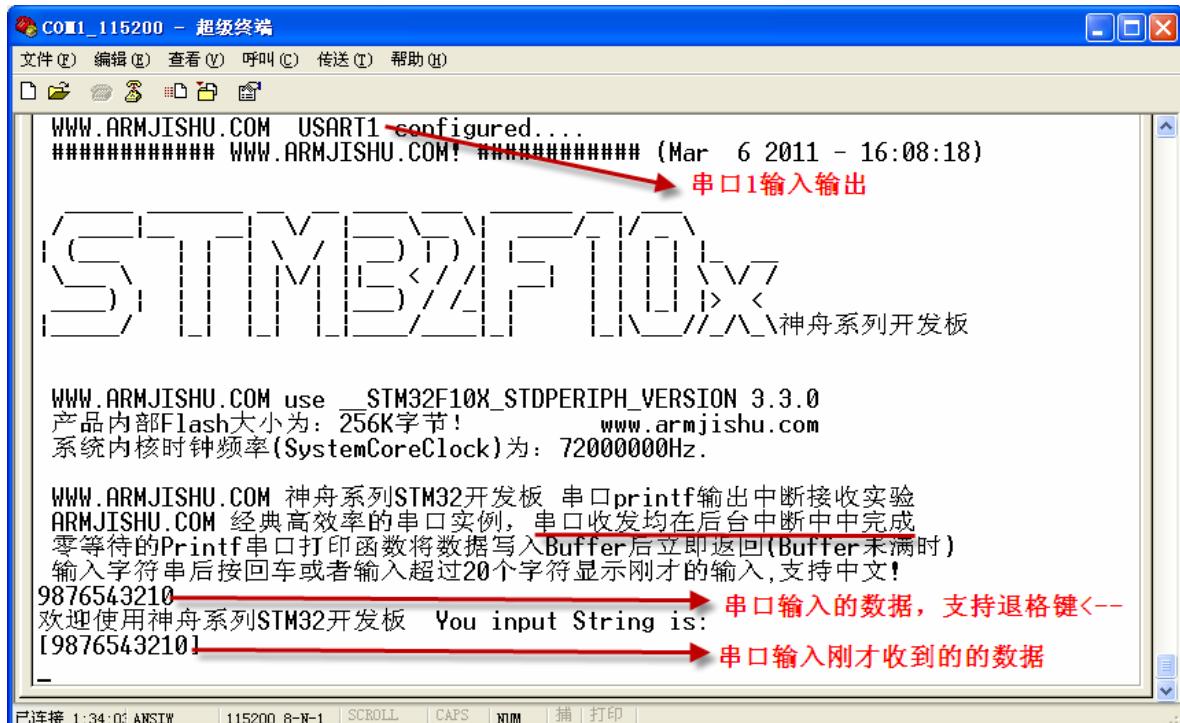
7.11.6 实验现象

将固件下载在神舟III号STM32开发板后，用随板配置的串口线连接神舟III号串口2与电脑的串口，打开超级终端，并按如下参数配置串口。

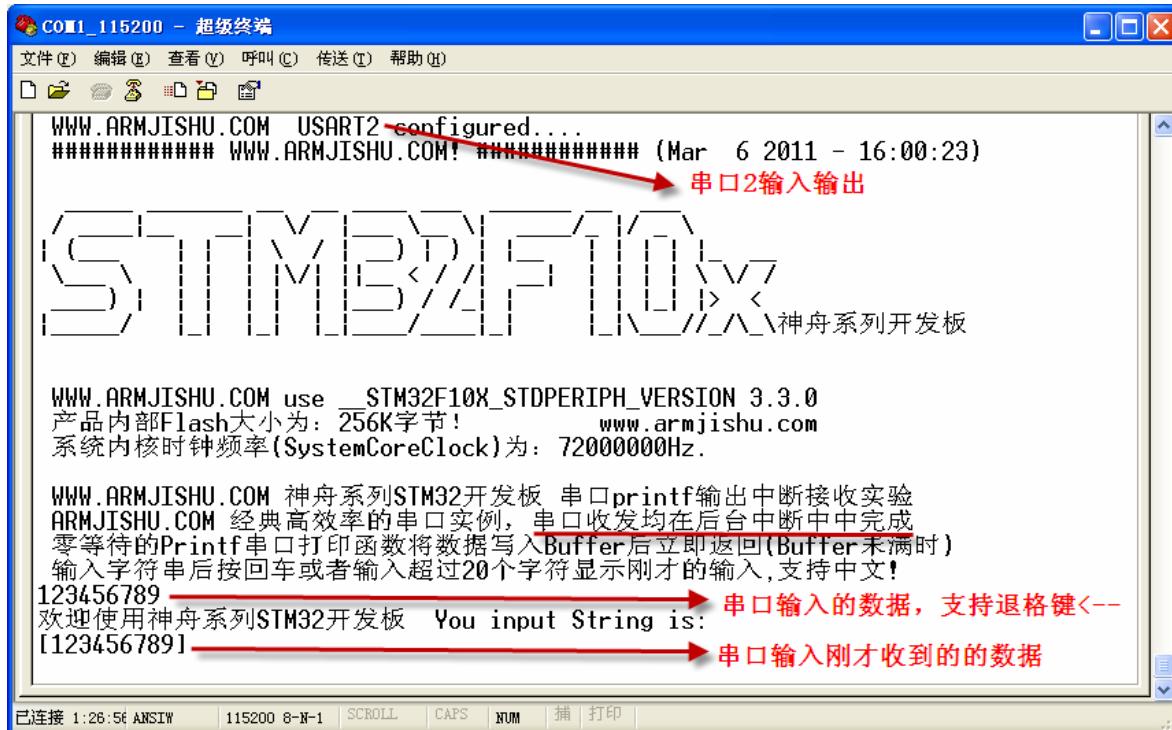


本实验使用串口接收中断和发送中断以及FIFO缓存来实现高效率的串口数据收发实例。下载固件后，可以在串口输入信息后按回车键，其实现效果如下图所示。

如果选择的是串口1，实验效果如下：



如果选择的是串口2，实验效果如下：



7.12 RS-485总线收发实验

7.12.1 485简介

485（一般称作RS485/EIA-485）是隶属于OSI（OSI：开放系统互连基本参考模型。开放，是指非垄断的。系统是指现实的系统中与互联有关的各部分。）模型物理层的电气特性规定为2线，半双工，多点通信的标准。它的电气特性和RS-232大不一样。用缆线两端的电压差值来表示传递信号。RS485仅仅规定了接受端和发送端的电气特性。它没有规定或推荐任何数据协议。

RS-232在1962年发布，命名为EIA-232-E，作为工业标准，以保证不同厂家产品之间的兼容。RS-422由RS-232发展而来，它是为弥补RS-232之不足而提出的。为改进RS-232通信距离短、速率低的缺点，RS-422定义了一种平衡通信接口，将传输速率提高到10Mb/s，传输距离延长到4000英尺（速率低于100kb/s时），并允许在一条平衡总线上连接最多10个接收器。RS-422是一种单机发送、多机接收的单向、平衡传输规范，被命名为TIA/EIA-422-A标准。为扩展应用范围，EIA又于1983年在RS-422基础上制定了RS-485标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为TIA/EIA-485-A标准。

RS485的特点包括：

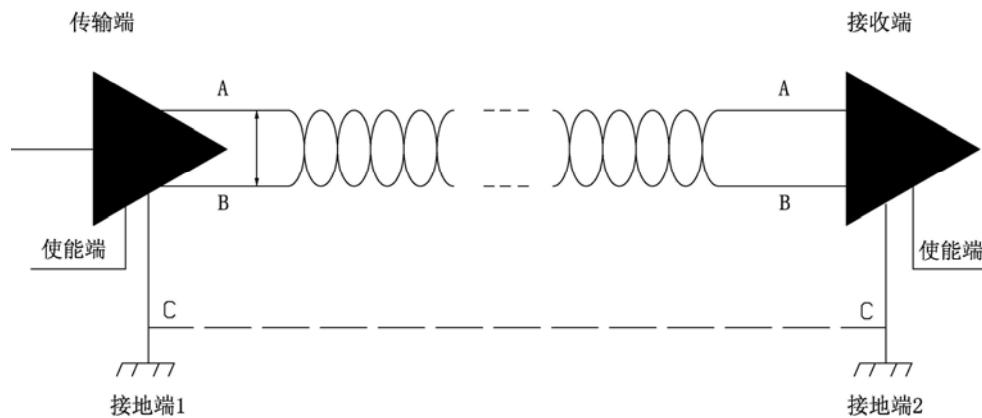
- 1) 接口电平低，不易损坏芯片。RS485的电气特性：逻辑“1”以两线间的电压差为+(2~6)V表示；逻辑“0”以两线间的电压差为-(2~6)V表示。接口信号电平比RS232降低了，不易损坏接口电路的芯片，且该电平与TTL电平兼容，可方便与TTL 电路连接。
- 2) 传输速率高。10米时，RS485的数据最高传输速率可达35Mbps，在1200m时，传输速度可达100Kbps。
- 3) 抗干扰能力强。RS485接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。
- 4) 传输距离远，支持节点多。RS485 总线最长可以传输 1200m 以上（速率≤100Kbps）一般最大支持 32 个节点，如果使用特制的 485 芯片，可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

7.12.2 RS485的通信概念

RS-485 是一个电气接口规范它只规定了平衡驱动器和接收器的电特性而没有规定接插件传输电缆和通信协议。

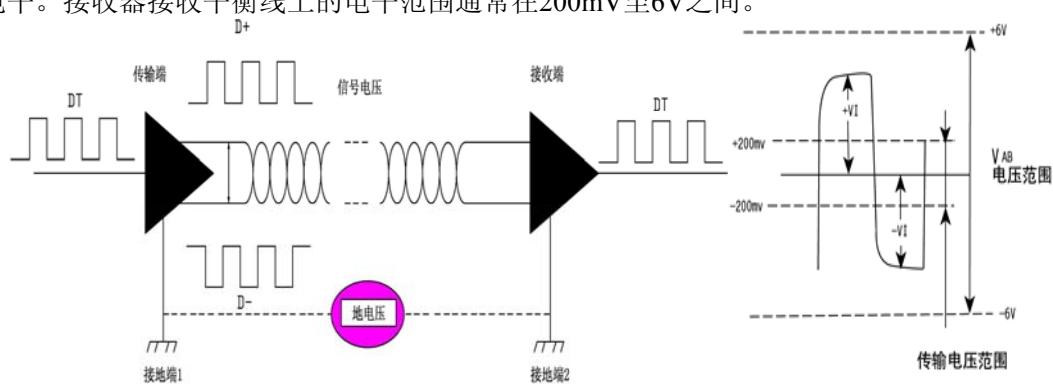
RS-485建议性标准作为一种多点差分数据传输的电气规范，现已成为业界应用最为广泛的标准通信接口之一，这种通信接口允许在简单的一对双绞线上进行多点双向通信，它所具有的噪声抑制能力、数据传输速率、电缆长度及可靠性是其他标准无法比拟的，因此许多不同领域都采用RS-485作为数据传输链路，它是一种极为经济并具有相当高的噪声抑制、传输速率、传输距离和宽共模范围的通信平台。

RS-485是一种在工业上作为数据交换的手段而广泛使用的串行通信方式，数据信号采用差分传输方式，也称作平衡传输，因此具有较强的抗干扰能力。它使用一对双绞线，将其中一线定义为A，另一线定义为B。如下图所示：



通常情况下，RS-485的信号在传送出去之前会先分解成正负对称的两条线路（即我们常说的A、B信号线），当到达接收端后，再将信号相减还原成原来的信号。发送驱动器A、B之间的正电平在+2~+6V，是一个逻辑状态；负电平在-2~-6V，是另一个逻辑状态；另有一个信号地C，在RS-485中还有一个“使能”端。“使能”端是用于控制发送驱动器与传输线的切断与连接。当“使能”端起作用时，发送驱动器处于高阻状态，称作“第三态”，即它是有别于逻辑“1”与“0”的第三态。

接收器也与发送端相对的电平逻辑规定，收、发端通过平衡双绞线将AA与BB对应相连，当在接收端AB之间(DT)=(D+) - (D-)有大于+200mV的电平时，输出正逻辑电平，小于-200mV时，输出负逻辑电平。接收器接收平衡线上的电平范围通常在200mV至6V之间。



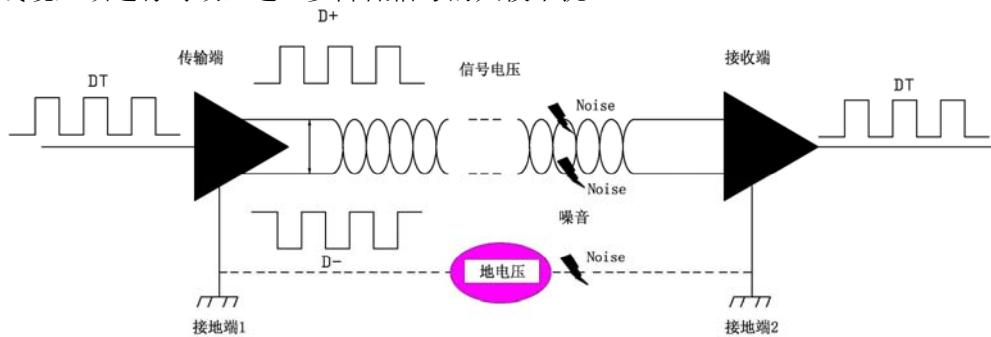
RS-485的信号在传送出去之前会先分解成正负对称的两条线路（即我们常说的A、B信号线），当到达接收端后，再将信号相减还原成原来的信号。如果将原来的信号标注为(DT)，而被分解后的信号分别标注为(D+)和(D-)，则原始信号与分解后的信号在由传输端传送出去时的运算关系如下：

$$(DT) = (D+) - (D-)$$

同样地，接收端在接收到信号后，也按上式的关系将信号还原成原来的样子。如果此线路受到干扰时，在两条传输线上的信号会分别成为(D+) + Noise 和 (D-) + Noise，如果接收端接收此信号，它必须按照一定的方式将其合成，合成的方程式如下：

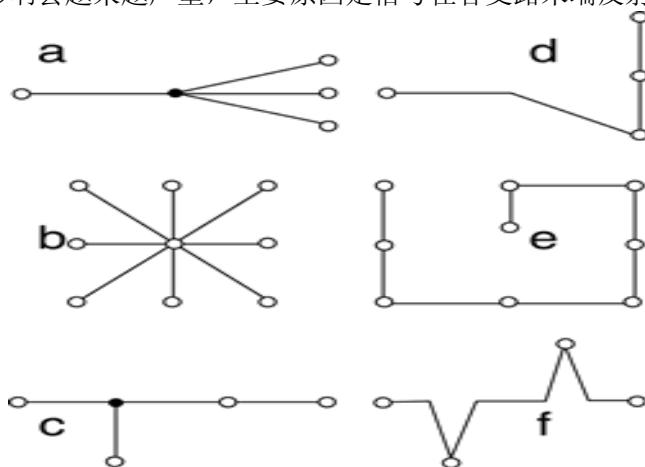
$$(DT) = [(D+ \text{ Noise})] - [(D- \text{ Noise})] = (D+) - (D-)$$

此方程与前一方程式的结果是一样的，干扰信号被抵消。因此在RS-485网络传输中要求两根信号线缆必须进行对绞，进一步降低信号的共模干扰。



7.12.3 RS485的连接方式

在每一个分支中采用一条双绞线电缆作总线，将各个通讯设备手拉手串接起来，从总线到每个通讯设备的引出线长度应尽量短，以便使引出线中的反射信号对总线信号的影响最低。如图3所示为实际应用中常见的一些错误连接方式(a, b, c)和正确的连接方式(d, e, f)。a、b、c这三种网络连接尽管不正确，在短距离、低速率仍可能正常工作，但随着通信距离的延长或通信速率的提高，其不良影响会越来越严重，主要原因是信号在各支路末端反射后与原信号叠加，会造成信号质量下降。



7.12.4 RS485通信电缆中的信号反射

什么信号反射：信号沿传输线向前传播时，每时每刻都会感受到一个瞬态阻抗，这个阻抗可能是传输线本身的，也可能是中途或末端其他元件的。对于信号来说，它不会区分到底是什么，信号所感受到的只有阻抗。如果信号感受到的阻抗是恒定的，那么他就会正常向前传播，只要感受到的阻抗发生变化，不论是什么引起的（可能是中途遇到的电阻，电容，电感，过孔，PCB 转角，接插件），信号都会发生反射。

反射的影响：如果负载阻抗小于传输线阻抗，反射电压为负，反之，如果负载阻抗大于传输线阻抗，反射电压为正。实际问题中，PCB 上传输线不规则的几何形状，不正确的信号匹配，经过连接器的传输及电源平面不连续等因素均会导致反射情况发生，而表现出诸如过冲/下冲以及振荡等信号失真的现象。

注意总线特性阻抗的连续性，在阻抗不连续和阻抗不匹配就会发生信号的反射（如图1所示）下列几种情况易产生这种不连续性：总线的不同区段采用了不同电缆，或某一段总线上有过多收发器紧靠在一起安装，再者是过长的分支线引出到总线。总之，应该提供一条单一、连续的信号通道作为总线。

消除这种反射的方法，就必须在电缆的末端跨接一个与电缆的特性阻抗同样大小的终端电阻，使

电缆的阻抗连续。由于信号在电缆上的传输是双向的，因此，在通讯电缆的另一端可跨接一个同样大小的终端电阻，如图2所示。

从理论上分析，在传输电缆的末端只要跨接了与电缆特性阻抗相匹配的终端电阻，就再也不会出现信号反射现象。但是，在实现应用中，由于传输电缆的特性阻抗与通讯波特率等应用环境有关，特性阻抗不可能与终端电阻完全相等，因此或多或少的信号反射还会存在。

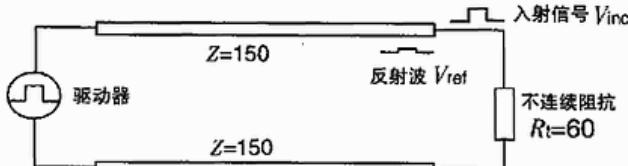


图 1 由于阻抗不连续引起的信号反射

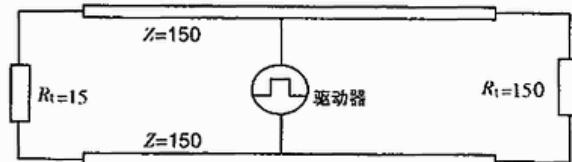


图 2 终端电阻的正确连接

7.12.5 RS485的接地问题

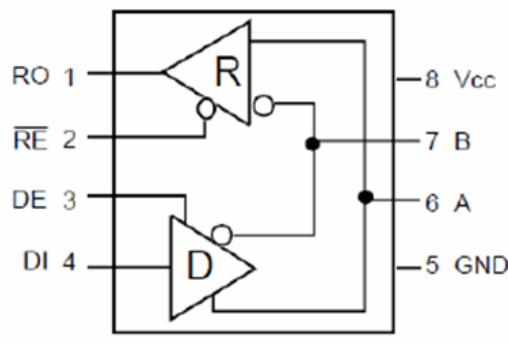
接地处理不当往往会导致电子系统不能稳定工作甚至危及系统安全。RS-485传输网络的接地同样也是很重要的，因为接地系统不合理会影响整个网络的稳定性，尤其是在工作环境比较恶劣和传输距离较远的情况下，对于接地的要求更为严格，否则接口损坏率较高。很多情况下，连接RS-485通信链路时只是简单地用一对双绞线将各个接口的“A”、“B”端连接起来。而忽略了信号地的连接，这种连接方法在许多场合是能正常工作的，但却埋下了很大的隐患。

7.12.6 RS485的应用

由于RS485具有传输距离远、传输速度快、支持节点多和抗干扰能力更强等特点，所以RS485有很广泛的应用。

SP3485芯片介绍

STM32神舟III号开发板采用SP3485作为收发器，该芯片支持3.3V供电，最大传输速度可达10Mbps，支持多达32个节点，并且有输出短路保护。该芯片的框图如下图所示：



SP3485框图

- A、B——485总线接口
- RO——接收输出端
- DI——发送数据收入端
- RE——接收使能信号（低电平有效）
- DE——发送使能信号（高电平有效）

➤ VCC、GND——电源与地

/RE和DE管脚控制RS-485的收发使能控制。在神舟III号中，这两个管脚与处理器的PF11管脚连接，由PF11管脚控制神舟III号STM32开发板的RS485作为发送端还是接收端。

查看《SP3485 RS-485收发器.pdf》可知，当PF11输出高电平，此时SP3485芯片的2脚（/RE），3脚（DE）都为高电平。SP3485工作与发送模式。逻辑关系如下表所示。

| INPUTS | | | LINE CONDITION | OUTPUTS | |
|------------------------|----|----|----------------|---------|---|
| $\overline{\text{RE}}$ | DE | DI | | B | A |
| X | 1 | 1 | No Fault | 0 | 1 |
| X | 1 | 0 | No Fault | 1 | 0 |
| X | 0 | X | X | Z | Z |

Table 1. Transmit Function Truth Table

当PF11输出低高电平，此时SP3485芯片的2脚（/RE），3脚（DE）都为低电平。SP3485工作与接收模式。逻辑关系如下表所示。

| INPUTS | | A - B | OUTPUTS | |
|------------------------|----|-------------|---------|--|
| $\overline{\text{RE}}$ | DE | | R | |
| 0 | 0 | +0.2V | 1 | |
| 0 | 0 | -0.2V | 0 | |
| 0 | 0 | Inputs Open | 1 | |
| 1 | 0 | X | Z | |

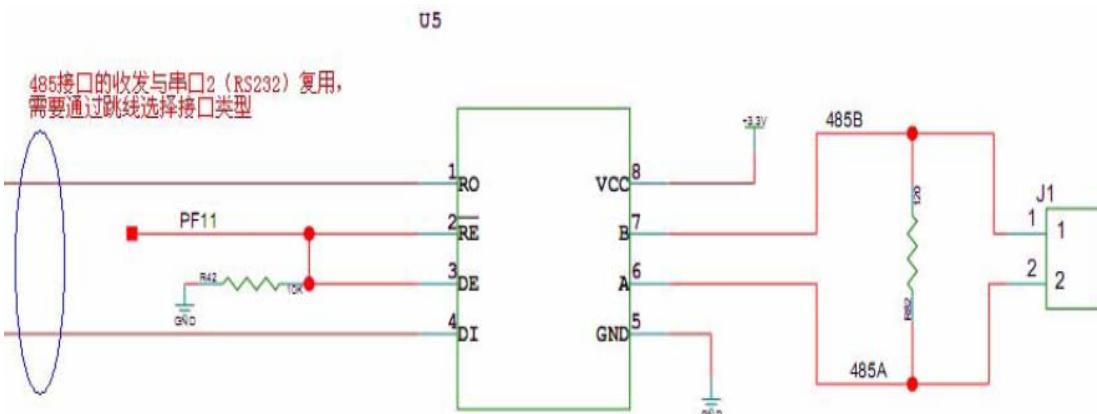
Table 2. Receive Function Truth Table

另外，上图中的R82的作用是作用RS485的终端匹配电阻，在RS-485总线网络中，终端匹配电阻主要作用是使总线的阻抗连续，减小信号的反射，提高信号的传输质量，一般RS485网络的终端匹配电路只需要在总线的最远端的节点并一个即可，其他接点不需要安装此匹配电阻。

在神舟III号STM32开发板中，默认终端匹配电阻都是安装的(对应原理图的R82)，请依据实际情况，选择安装或去掉此电阻。

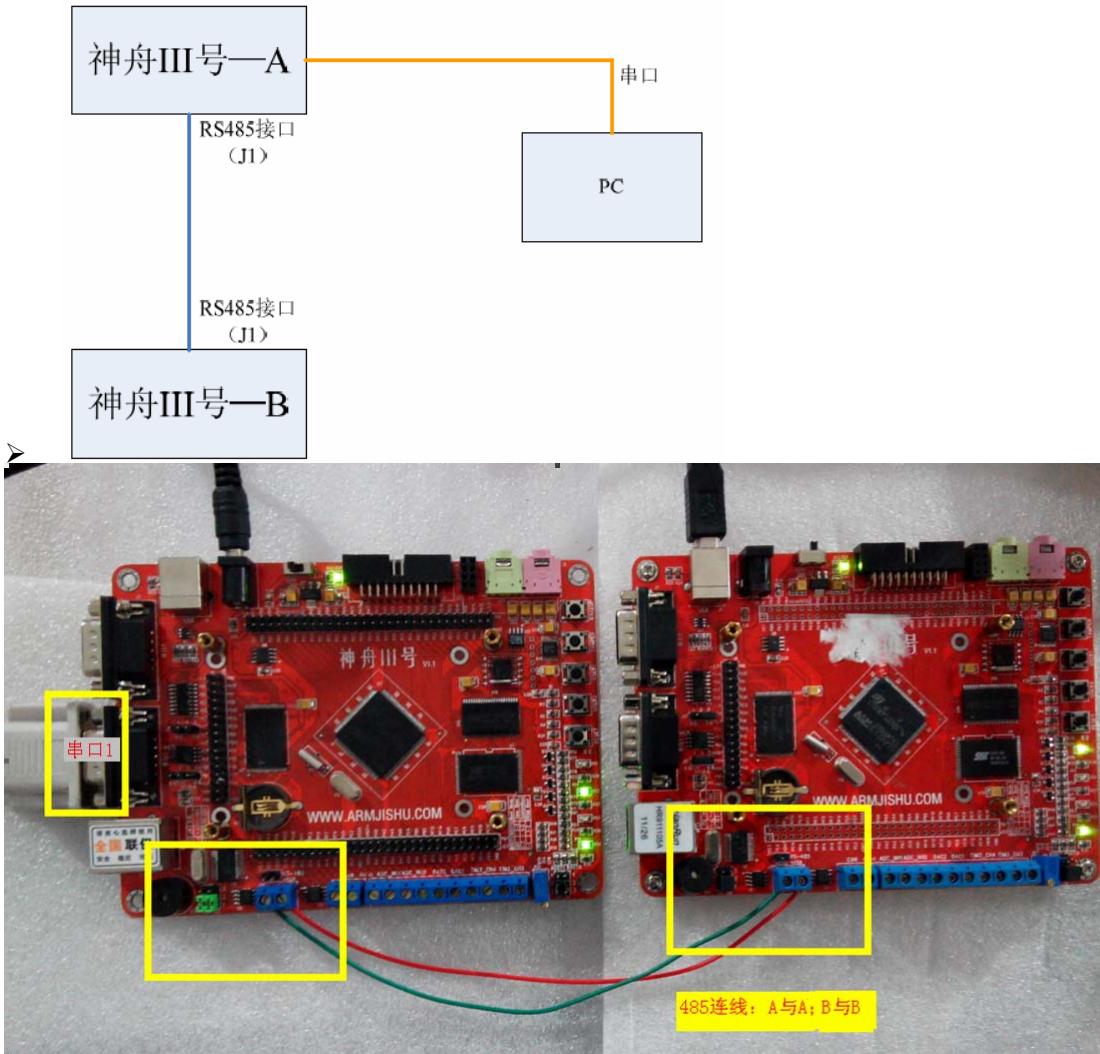
7.12.7 原理图的连接

我们通过该芯片连接STM32的串口2，实现两个开发板之间的485通信。



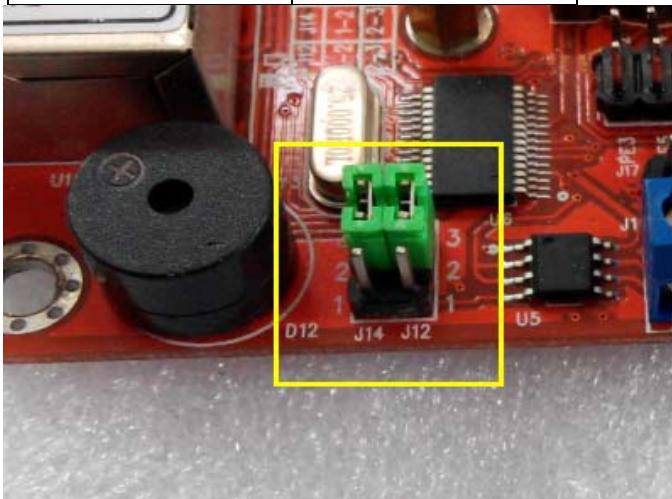
连接两个神舟STM32开发板的RS485接口，然后由KEY0控制发送，当按下一个开发板的KEY0的时候，就发送5个数据给另外一个开发板，并在电脑终端上分别显示发送的值和接收到的值。

➤ 产品连接情况



神舟III号开发板载有RS485物理芯片，它与处理器的UART2连接，与串口2复用，可通过跳线选择支持RS-232接口或RS-485接口，跳线定义如下：

| | |
|---------|--------------|
| J14和J12 | 串口2功能选择 |
| 1-2 | 串口2 RS-485接口 |
| 2-3（默认） | 串口2 RS-232接口 |



神舟III号默认是安装了RS-485接口的120欧终端匹配电阻。对应上图的R82，请依据实际应用选择是否安装此匹配电阻。

7.12.8 实验现象

8 发送数据现象



9 接收数据现象



7.12.9 软件设计

根据实验现象初步分析代码

嵌入式专业技术论坛 (www.armjishu.com) 出品

第 376 页, 共 900 页

在本实验中，程序运行以后，首先通过串口1打印提示信息，提示通过板上的USER1和USER2按键设置神舟III号为发送端或者接收端，设置完成后，发送端周期性的发送数据到RS-485网络上（神舟III号的RS485收发器与处理器的串口2连接），而接收端等待RS-485网络上的数据，并将接收到的完整数据通过串口1打印出来。因此本实验需要用的资源有串口1，串口2，按键，LED，RS485收发器。

在本实现中，按键，LED灯以及串口，RS-485收发器方向控制等都是有处理器GPIO连接的，因此在使用之前，我们需要对相关的GPIO初始化。

初始化部分

```
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
    */
    /*!< 在系统启动文件 (startup_stm32f10x_xx.s) 中已经调用SystemInit() 初始化了时钟,
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
       SystemInit() 函数的实现位于system_stm32f10x.c文件中。
    */
    uint8_t RcvCh;
    uint8_t TxBuffer[] = "www.armjishu.com STM32神舟系列开发板 RS485总线收发实验\r\n";
    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    /* 初始化RS485控制方向管脚PF11 */
    SZ_STM32_RS485();

    /* 默认配置RS485的方向为接收，以免引起总线冲突 */
    RS485_SET_RX_Mode();

    /* 初始化系统定时器SysTick, 每秒中断1000次 */
    SZ_STM32_SysTickInit(1000);

    /* 注意串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为sz_STM32_COM2 */
    /* 串口2初始化 请将注意J14和J12跳到1<->2侧 */
    SZ_STM32_COMInit(COM2, 9600);

    /* 串口1初始化 */
    xPrintf_Init();

    xprintf("\r-----\n");
    xprintf("\r      STM32神舟系列开发板RS485总线收发实验\n");
    . . . . .
```

分别初始化 LED、RS485、系统定时器与串口，这样我们才能正常使用这些功能

串口打印数据，初始化完后，我们让串口打印我们这次的实验内容与提示信息，方便我们的操作

```
xprintf("\n\r-----\n");
xprintf("\n\r      STM32神舟III号开发板RS485总线收发实验\n");
xprintf("\n\r--按USER1按键设置神舟开发板设置为RS485发送端\n");
xprintf("\n\r--按USER2按键设置神舟开发板设置为RS485接收端\n");
xprintf("\n\r-----\n");
xprintf("\n\r 提示：当前显示的窗口为串口1/COM1/USART1。 \r\n");
xprintf("\n\r      串口2/COM2/USART2作为RS485，请将注意J14和J12跳到1<->2侧。 \r\n");
```

STM32神舟III开发板RS485总线收发实验

- 按USER1按键设置神舟开发板设置为RS485发送端
- 按USER2按键设置神舟开发板设置为RS485接收端

提示：当前显示的窗口为串口1/COM1/USART1。

串口2/COM2/USART2作为RS485，请将注意J14和J12跳到1<->2侧。

其中蓝色方框的为串口初始化的时候，串口打印的数据

往下是我们的选择接收的时候，对数据的一个处理与给串口发送接收到的数据部分

```
/* Infinite loop 主循环 */
while (1)
{
    if (RS485_Mode == RX_MODE) //RX模式
    {
        /* 接收连接在USART2上的RS485的数据 */
        while (USART_GetFlagStatus(SZ_STM32_COM2, USART_FLAG_RXNE) == RESET)
        {
        }

        RcvCh = (int)SZ_STM32_COM2->DR & 0xFF;

        xprintf("%c", RcvCh);
    }
}
```

串口打印接收到的数据“RcvCh”本次实验中，我们接收的数据为：

提示：当前显示的窗口为串口1/COM1/USART1。

串口2/COM2/USART2作为RS485，请将注意J14和J12跳到1<->2侧。

RS485 接收模式设置成功
等待接收数据

www.armjishu.com STM32神舟系列开发板 RS485总线收发实验
www.armjishu.com STM32神舟系列开发板 RS485总线收发实验

这个是接收数据，那么我们看下发送的数据

```
else if (RS485_Mode == TX_MODE)          //TX模式
{
    xprintf("\n\r正在发送数据: %s", TxBuffer);
    /* 通过printf函数即可将数据通过连接在USART2上的RS485发送出去 */
    printf("%s", TxBuffer);

    delay(60000000);
}
```

这里我们发送出去的数据为“TxBuffer”，而这个“TxBuffer”就是我们前面定义的一个数组

```
/*
uint8_t RcvCh;
uint8_t TxBuffer[] = "www.armjishu.com STM32神舟系列开发板 RS485总线收发实验\r\n";
... ...
... ...
```

那么也就是说我们现在发送的数据就是把"www.armjishu.com STM32 神舟系列开发板 RS485 总线收发实验\r\n"通过 485 发送出去,我们看下实验现象中发送的数据

提示：当前显示的窗口为串口1/COM1/USART1。

串口2/COM2/USART2作为RS485，请将注意J14和J12跳到1<->2侧。

RS485 发送模式设置成功
 正在发送数据: www.armjishu.com STM32神舟系列开发板 RS485总线收发实验
 正在发送数据: www.armjishu.com STM32神舟系列开发板 RS485总线收发实验
 正在发送数据: www.armjishu.com STM32神舟系列开发板 RS485总线收发实验

如果都不是发送或者接收的话，那么我们继续等待按键的按下再执行相对应的程序操作

```
}
else
{
    RS485_MODE_SET();
}
```

这个就是我们的主函数的内容，通过按键确定是接收还是发送数据，通过串口打印把数据显示出来

深入分析例程代码

初始化RS485控制方向管脚PF11

```
void SZ_STM32_RS485(void)
{
    /* 配置神舟III号RS485方向控制管脚*/
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin = RS485_PIN;           //与PF11管脚连接
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(RS485_GPIO_PORT, &GPIO_InitStructure);
}
```

LED初始化，配置控制LED的I/O口为推挽输出

```
void SZ_STM32_LEDInit(Led_TypeDef Led)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable the GPIO_LED Clock */
    /* 使能LED对应GPIO的Clock时钟 */
    RCC_APB2PeriphClockCmd(GPIO_CLK[Led], ENABLE);

    /* Configure the GPIO_LED pin */
    /* 初始化LED的GPIO管脚，配置为推挽输出 */
    GPIO_InitStructure.GPIO_Pin = GPIO_PIN[Led];
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

    GPIO_Init(GPIO_PORT[Led], &GPIO_InitStructure);
}
```

SZ_STM32_KEYInit()函数完成与按键连接的GPIO初始化。

```
void SZ_STM32_KEYInit(Button_TypeDef Button, ButtonMode_TypeDef Button_Mode)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable the BUTTON Clock */
    /* 使能KEY按键对应GPIO的clock时钟 */
    RCC_APB2PeriphClockCmd(BUTTON_CLK[Button] | RCC_APB2Periph_AFIO, ENABLE);

    /* Configure Button pin as input floating */
    /* 初始化KEY按键的GPIO管脚，配置为带上拉的输入 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = BUTTON_PIN[Button];
    GPIO_Init(BUTTON_PORT[Button], &GPIO_InitStructure);

    /* 初始化KEY按键为中断模式 */
    if (Button_Mode == BUTTON_MODE_EXTI)
    {
        /* Connect Button EXTI Line to Button GPIO Pin */
        /* 将KEY按键对应的管脚连接到内部中断线 */
        GPIO_EXTILineConfig(BUTTON_PORT_SOURCE[Button], BUTTON_PIN_SOURCE[Button]);

        /* Configure Button EXTI line */
        /* 将KEY按键配置为中断模式，下降沿触发中断 */
        EXTI_InitStructure EXTI_Line = BUTTON_EXTI_LINE[Button];
        EXTI_InitStructure EXTI_Mode = EXTI_Mode_Interrupt;
        EXTI_InitStructure EXTI_Trigger = EXTI_Trigger_Falling;
        EXTI_InitStructure EXTI_LineCmd = ENABLE;
        EXTI_Init(&EXTI_InitStructure);

        /* Enable and set Button EXTI Interrupt to the lowest priority */
        /* 将KEY按键的中断优先级配置为最低 */
        NVIC_InitStructure.NVIC_IRQChannel = BUTTON_IRQn[Button];
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F;
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
        NVIC_Init(&NVIC_InitStructure);
    }
}
```

在实验中，串口1用于打印程序提示信息和RS-485接收/发送的数据，因此，我们初始化串口1，作为printf函数输出使用。关于printf的实现可参见《串口1 printf实验》详细说明。配置串口的基本参数，具体如下所示：

```
void xPrintf_Init(void)
{
    USART_InitTypeDef USART_InitStructure;

    /* USARTx configured as follow:
       - BaudRate = 115200 baud
       - Word Length = 8 Bits
       - One Stop Bit
       - No parity
       - Hardware flow control disabled (RTS and CTS signals)
       - Receive and transmit enabled
    */
    USART_InitStructureUSART_BaudRate = 115200;
    USART_InitStructureUSART_WordLength = USART_WordLength_8b;
    USART_InitStructureUSART_StopBits = USART_StopBits_1;
    USART_InitStructureUSART_Parity = USART_Parity_No;
    USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    __SZ_STM32_COMInit(COM1, &USART_InitStructure);
    xdev_out(xUSART1_putchar);
    xdev_in(xUSART1_getchar);
}
```

除了串口1外，我们还能使用串口2作为printf函数输出使用，串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为SZ_STM32_COM2。在使用串口2之前，我们首先需要对串口参数进行配置，主要是配置串口的波特率，数据位，奇偶校验位等信息等，具体代码如下。

```
void SZ_STM32_COMInit(COM_TypeDef COM, uint32_t BaudRate)
{
    USART_InitTypeDef USART_InitStructure;

    /* USARTx 默认配置:
       - BaudRate = 115200 baud
       - Word Length = 8 Bits
       - One Stop Bit
       - No parity
       - Hardware flow control disabled (RTS and CTS signals)
       - Receive and transmit enabled
    */
    USART_InitStructureUSART_BaudRate = BaudRate; //串口的波特率，例如115200 最高达4.5Mb
    USART_InitStructureUSART_WordLength = USART_WordLength_8b; //数据字长度(8位或9位)
    USART_InitStructureUSART_StopBits = USART_StopBits_1; //可配置的停止位-支持1或2个停止位
    USART_InitStructureUSART_Parity = USART_Parity_No; //无奇偶校验
    USART_InitStructureUSART_HardwareFlowControl = USART_HardwareFlowControl_None; //无硬件流控制
    USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx; //双工模式，使能发送和接收

    SZ_STM32_COMInit(COM, &USART_InitStructure); // 调用STM32的USART初始化底层函数
}
```

在完成了初始化以后，主程序中，首先等待用于通过STM32按键设置神舟III号RS485作为接收或发送端设置，如果作为RS485接收端，则将PF11输出低电平，控制SP3485收发器在接收模式，通过串口1打印提示信息。如果作为RS485发送端，则将PF11输出高电平，控制SP3485收发器在发送模式，并通过串口1打印提示信息。
这主要是通过RS485_MODE_SET() 函数实现的。

```
/*
 * @函数名 RS485_MODE_SET
 * @功能 通过按键设置RS485为收发模式
 * @参数 无
 * @返回值 无
*/
void RS485_MODE_SET(void)
{
    uint32_t KeyNum = 0;

    /* 等待按键 */
    while(! (KeyNum = SZ_STM32_KEYScan()));

    if(1 == KeyNum)
    {
        RS485_Mode = TX_MODE;
        xprintf("\n\rRS485 发送模式设置成功");
        RS485_SET_TX_Mode();
        SZ_STM32_LEDOn(LED1);
        SZ_STM32_LEDOff(LED2);
    }
    else if(2 == KeyNum)
    {
        RS485_Mode = RX_MODE;
        xprintf("\n\rRS485 接收模式设置成功");
        xprintf("\n\r等待接收数据\n\r");
        SZ_STM32_LEDOff(LED1);
        SZ_STM32_LEDOn(LED2);
        RS485_SET_RX_Mode();
    }
    else
    {
        RS485_Mode = IDLE;
        SZ_STM32_LEDOff(LED1);
        SZ_STM32_LEDOff(LED2);
        xprintf("\n\r退出发送接收，请重新设置工作模式");
        xprintf("\n\r USER1按键：设置RS485为接收模式");
        xprintf("\n\r USER2按键：设置RS485为发送模式");
    }
}
```

设置完神舟III号的RS485工作模式后，依据实际的工作模式后，程序依据模式执行相关的代码。按键1按下的时候，使“RS485_Mode = TX_MODE;”配置为发送模式，按键2按下的时候，使“RS485_Mode = RX_MODE;”配置为接收模式，而当按键3按下的时候，使“RS485_Mode = IDLE;”退出发送接收，重新设置工作模式。

如果为接收模式，则接收来自485接收到的数据并保存接收到的数据。接收完完整的一串数据后，将接收到的数据打印出来。

```
if(RS485_Mode == RX_MODE) //RX模式
{
    /* 接收连接在USART2上的RS485的数据 */
    while(USART_GetFlagStatus(SZ_STM32_COM2, USART_FLAG_RXNE) == RESET)
    {

        RcvCh = (int)SZ_STM32_COM2->DR & 0xFF;
        xprintf("%c", RcvCh);
    }
}
```

如果为发送模式，则把“TxBuffer”里面的数据通过485发送出去给接收端，发送完完整的一串数据。并通过串口1打印发送的数据。延迟一段时间后，重复发送。

```

else if(RS485_Mode == TX_MODE)          //TX模式
{
    xprintf("\n\r正在发送数据: %s", TxBuffer);
    /* 通过printf函数即可将数据通过连接在USART2上的RS485发送出去 */
    printf("%s", TxBuffer);

    delay(60000000);
}

```

发送的数据为

```
uint8_t TxBuffer[] = "www.armjishu.com STM32神舟系列开发板 RS485总线收发实验\r\n";
```

7.12.10 下载与验证

如果使用JLINK下载固件，请按[3.3如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作。

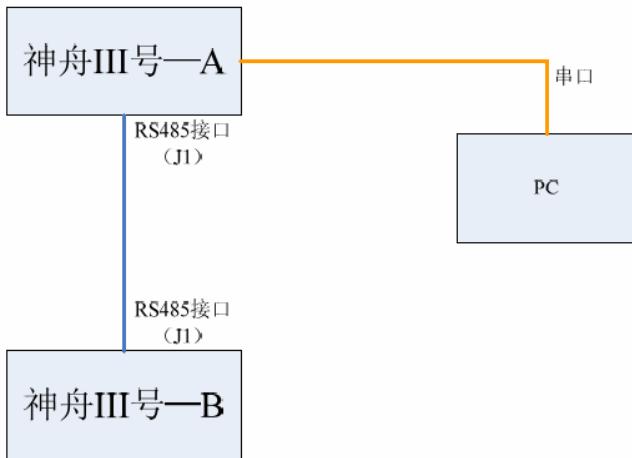
如果使用串口下载固件，请按[3.4如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.8如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

7.12.11 实验现象

在实验中，一块神舟III号STM32开发板作为RS485发送端，一块神舟III号STM32开发板作用RS485接收端。因此，进行本实验需要准备两块神舟III号STM32开发板。

首先将固件分别下载到两块神舟III号STM32开发板，按如下连接网络。



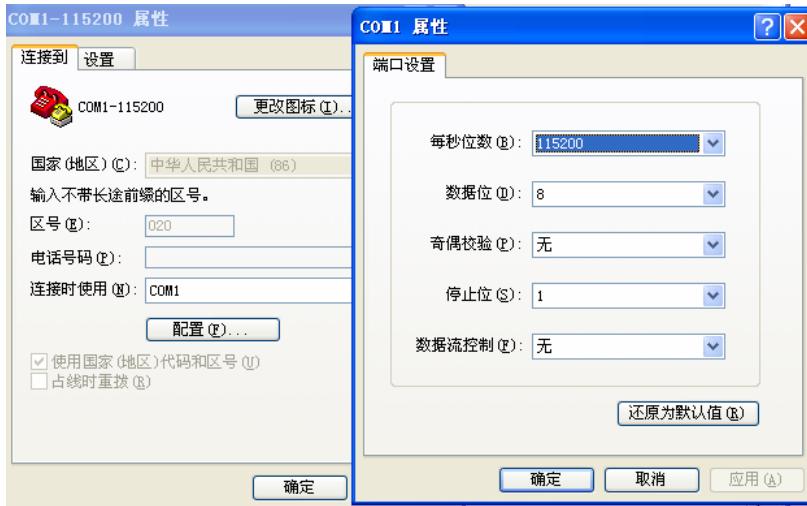
用两个电线连接两块神舟III号的RS-485接口（J1位置），注意两块神舟III号的RS-485接口（J1位置）的第1脚和第1脚连接，第2脚和第2脚连接。

由于485与USART2共用JPIO，所以首先请确认将J14和J12跳帽跳至1←→2(下侧)，选择串口2为RS485接口类型。

串口2可通过跳线选择接口定义如下：

| J14和J12 | 串口2功能选择 |
|---------|--------------|
| 1←→2 | 串口2 RS-485接口 |
| 2←→3 | 串口2 RS-232接口 |

用随板配带的串口线连接神舟III号和PC机，打开超级终端，按如下配置超级终端参数。



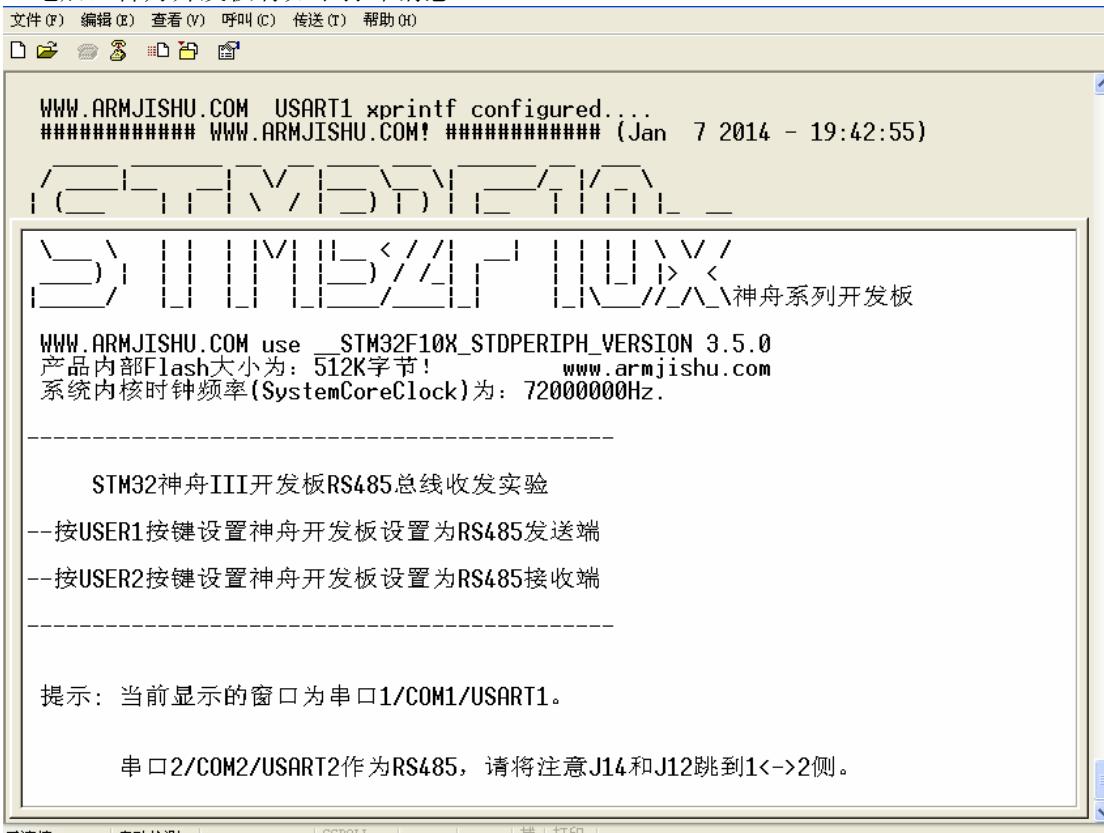
上电运行神舟III号，在串口提示信息指示下，首先设置设置神舟III号—A为RS-485接收端，然收设置神舟III号—B为RS-485发送端。具体设置方法如下：

| 操作 | 说明 |
|----------|------------------|
| 按USER1按键 | 神舟III号作为RS485发送端 |
| 按USER2按键 | 神舟III号作为RS485接收端 |

设置成功后，神舟III号的LED指示灯也将指示具体的工作模式

| 现象 | 含义操作 |
|------|----------------------------------|
| DS1亮 | 神舟III号正常运行（上电以后，DS1点亮，在发送或接收时闪烁） |
| DS2亮 | 神舟III号作为RS485发送端，周期性发送数据 |
| DS3亮 | 神舟III号作为RS485发送端，接收RS485的数据 |

上电后，神舟开发板有如下打印消息：



首先设置设置神舟III号—A为RS-485接收端，按其“**USER2**”按键；然收设置神舟III号—B为RS-485发送端，按其“**USER1**”按键。神舟III号—A接收端会通过串口1打印接收或发送的数据。以下是RS485接收端的串口1打印信息。

提示：当前显示的窗口为串口1/COM1/USART1。

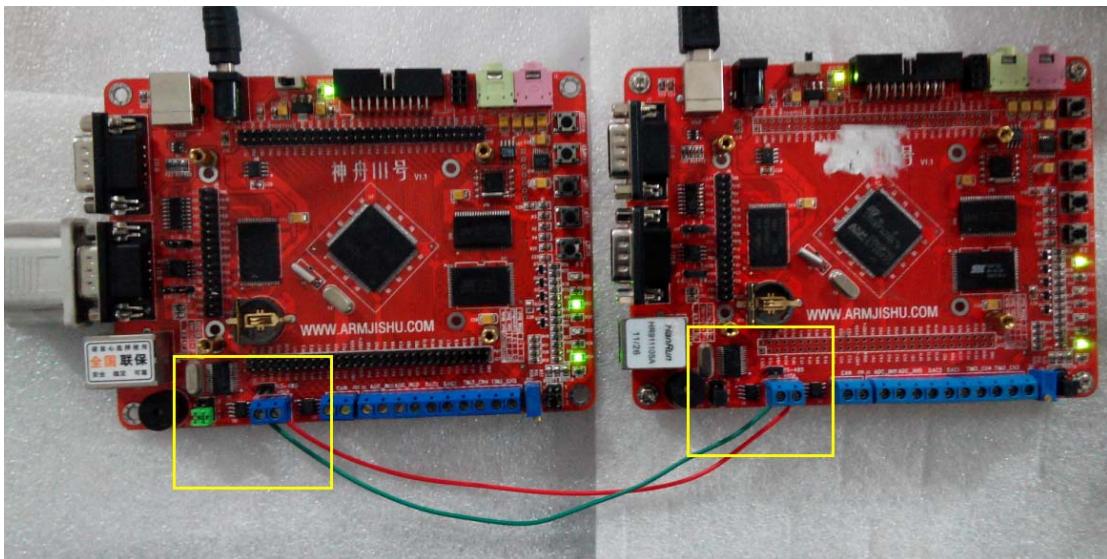
串口2/COM2/USART2作为RS485，请将注意J14和J12跳到1<->2侧。

RS485 接收模式设置成功
等待接收数据

实物连接图如下：

两块神舟 III 号开发板串口 1，用两条串口线与两台电脑一对一连好；

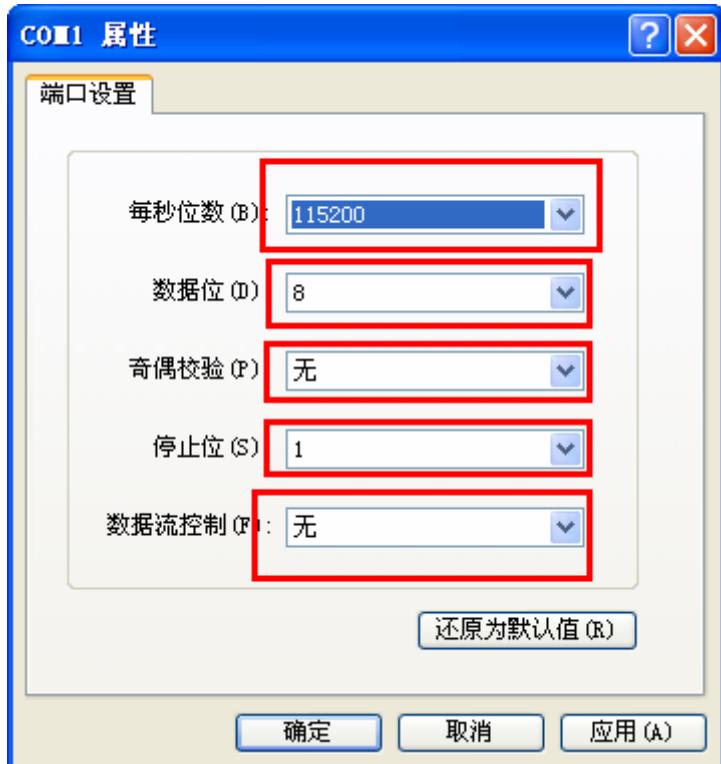
两块神舟 III 号开发板的 485 总线座子（开发板有显示标号）用两条杜邦线相连，485A 与 485A 连接，485B 与 485B 连接；以下图：



两台电脑都打开超级终端，按以下设置，如下图：



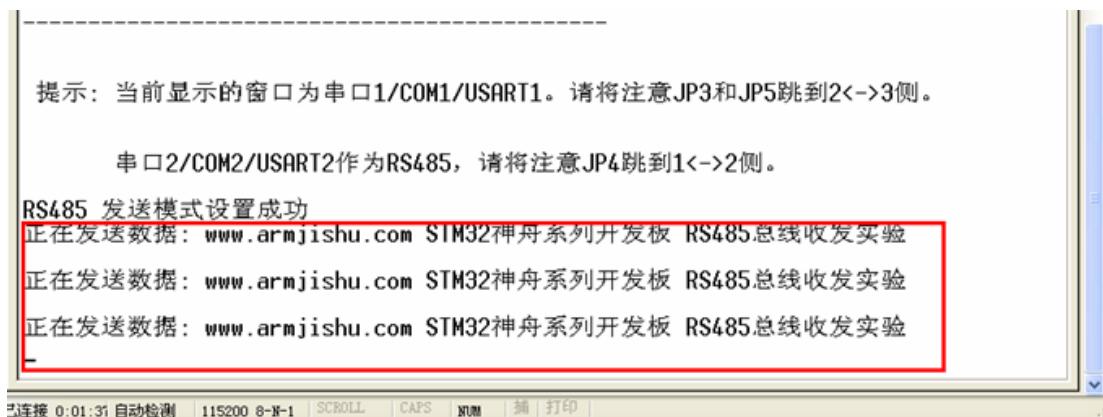
选择 COM1；按确定



再按确定，完成超级终端设置。

两块开发板重新打开电源，一块开发板设置为发送按下 USART1 按键，另一块开发板设置为接收按下 KEY2 按键；看超级终端显示：

发送那台的超级终端窗口显示信息：



接收那台的超级终端窗口显示信息：

提示：当前显示的窗口为串口1/COM1/USART1。

串口2/COM2/USART2作为RS485，请将注意J14和J12跳到1<->2侧。

RS485 发送模式设置成功

正在发送数据: www.armjishu.com STM32神舟系列开发板 RS485总线收发实验

正在发送数据: www.armjishu.com STM32神舟系列开发板 RS485总线收发实验

正在发送数据: www.armjishu.com STM32神舟系列开发板 RS485总线收发实验

以上就是RS-485的实验现象。

7.13 产品唯一身份标识（Unique Device ID）实验

7.13.1 意义与作用

产品唯一的身份标识(Unique Device ID)非常适合：

- 用来作为序列号(例如 USB 字符序列号或者其他终端应用)
- 用来激活带安全机制的自举过程
- 用来作为密码，在编写闪存时，将此唯一标识与软件加解密算法结合使用，提高代码在闪存存储器内的安全性。

96 位的产品唯一身份标识所提供的参考号码对任意一个 STM32 微控制器，在任何情况下都是唯一的。用户在何种情况下，都不能修改这个身份标识。

7.13.2 实验原理

这个 96 位的产品唯一身份标识，按照用户不同的用法，可以以字节(8 位)为单位读取，也可以以半字(16 位)或者全字(32 位)读取。

96 位的独特 ID 位于地址 0x1FFFF7E8 ~ 0x1FFFF7F3 的系统存储区，用户可以以字节、半字、或字的方式单独读取其间的任一地址，其中 0x1FFFFF3 中为最高字节，0x1FFFF7E8 中为最低字节。

本次试验以 ARMJISHU 的神舟系列开发板为硬件平台，通过调用系统的 Printf 函数来打印出唯一标识 DeviceSerial。

作为扩展本次试验顺带读取位于 0x1FFF F7E0 地址的“闪存容量寄存器”获得开发板内部集成 Flash 的大小信息，通过调用系统的 Printf 函数来打印出产品内部 Flash 大小。

7.13.3 硬件设计

产品唯一身份标识(Unique Device ID)为处理器内部组件，这部分不需要硬件电路，这里仅在串口中输出产品唯一身份标识(Unique Device ID)即可。

7.13.4 软件设计

程序中定义全局变量 IntDeviceSerial 存放读到的设备 ID, 96 位的独特 ID 位于地址 0x1FFFF7E8 ~ 0x1FFFF7F3 的系统存储区，程序如下：

```
uint32_t IntDeviceSerial[3]; /* 全局变量IntDeviceSerial存放读到的设备ID */

void Get_ChipSerialNum(void)
{
    IntDeviceSerial[0] = *(_IO uint32_t*) (0x1FFFF7E8);
    IntDeviceSerial[1] = *(_IO uint32_t*) (0x1FFFF7EC);
    IntDeviceSerial[2] = *(_IO uint32_t*) (0x1FFFF7E0);
}
```

Delay_ARMJISHU 函数流水灯使用的延时函数，用简单的 For 循环实现

```
static void Delay_ARMJISHU(_IO uint32_t nCount)
{
    for (; nCount != 0; nCount--);
```

下来看看 main 函数，其中涉及的子程序如果之前的章节已有介绍（如串口相关），则此处不再讲述。本示例涉及的程序都添加了较为详细的注释。

在 MAIN 主函数中调用 Get_ChipSerialNum() 以后，就可以使用 printf 来打印，然后作为扩展，本次试验顺带读取位于 0x1FFF F7E0 地址的“闪存容量寄存器” 使用 printf 来打印出来。

```
/** 
 * @brief Main program
 * @param None
 * @retval None
 */
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
    */

    /* USARTx configured as follow:
       - BaudRate = 115200 baud
       - Word Length = 8 Bits
       - One Stop Bit
       - No parity
       - Hardware flow control disabled (RTS and CTS signals)
       - Receive and transmit enabled
    */
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    STM_EVAL_COMInit(COM1, &USART_InitStructure);

    /* Output a message on Hyperterminal using printf function */
    printf("\n\rUSART Printf Example: retarget the C library printf function to the USART\n\r");
    printf("\r\n\r\n\r WWW.ARmjishu.COM is configured....", EVAL_COM1_STR);
    printf("\n\r ##### WWW.ARmjishu.COM! ##### (%_DATE_ " "- "%_TIME_ ")");
    printf("\n\r www.armjishu.com论坛后续还会有更多精彩示例，欢迎访问论坛交流与学习\n\r\n\r");
}
```


7.13.5 下载与验证

如果使用JLINK下载固件，请按3.2如何使用JLINK软件下载固件到神舟III号开发板小节进行操作。

如果使用串口下载固件，请按3.3如何通过串口下载一个固件到神舟III号开发板小节进行操作。

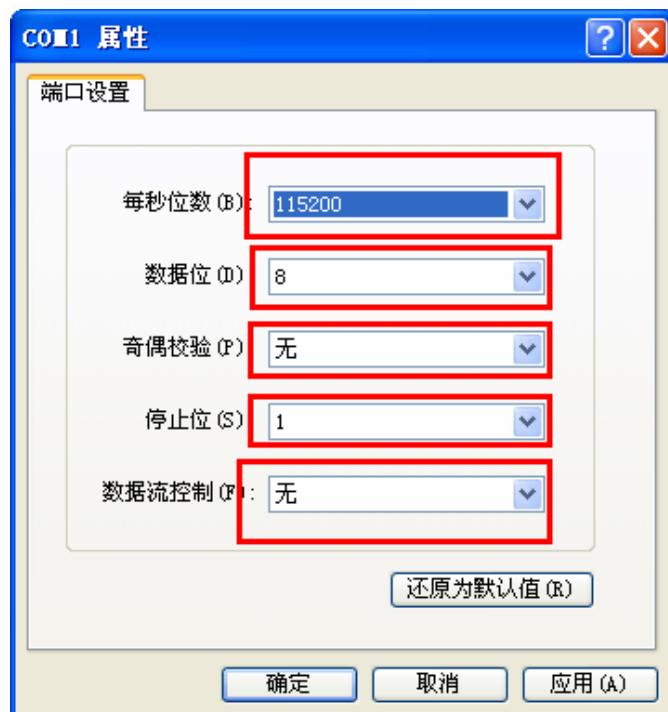
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.13.6 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 2 与电脑连接(注意开发板上 J14 和 12 位置都短接 2=3 为串口模式)，并打开超级终端，按以下设置，如下图：



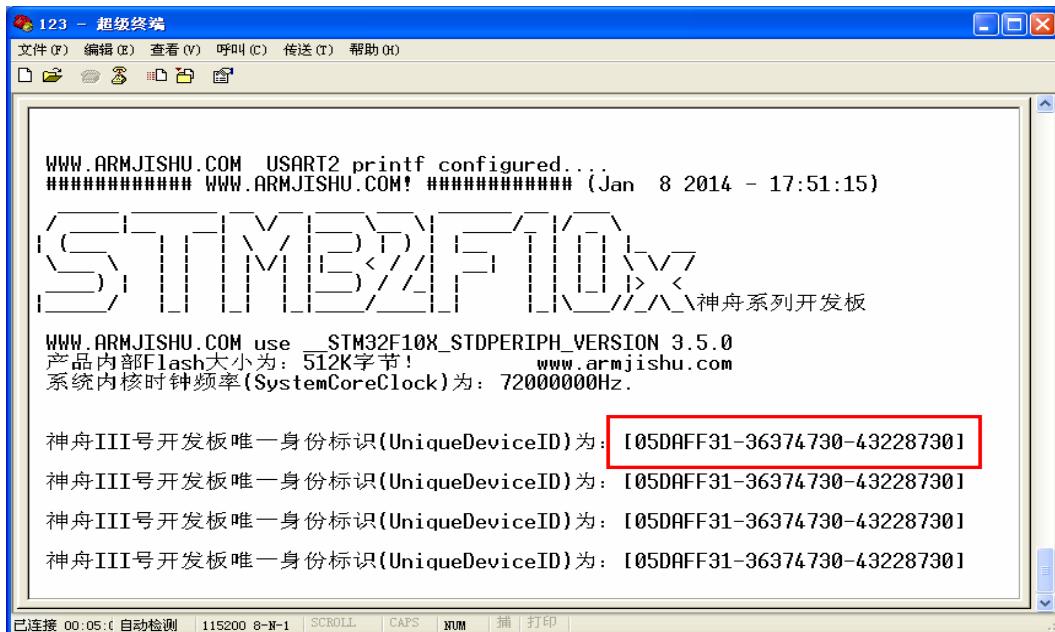
选择 COM1；按确定



再按确定，完成超级终端设置。

重新打开电源；神舟 III 号 STM32 开发板上 4 个 LED 灯闪一下，LED 2 和 LED 3 灯亮，LED 1 和 LED 4 在闪烁。

超级终端窗口显示信息，如下图，可以看到神舟 III 号开发板的串口打印出了产品唯一身份标识(Unique Device ID)



图上红框的为产品唯一身份标识。

7.14 ADC模数转换实验

ADC：Analog-to-Digital Converter（模数转换器），顾名思意就是将模拟量转换成数字量的设备或模块。

前面的例程，我们大家一起学习了STM32处理器的GPIO操作，以及串行接口的使用，在这一节，我们将一起来简单学习STM32的ADC的使用。神舟III号将电位器上采集到的数据通过ADC转换后，通过串口将转换的结果数据打印出来。

7.14.1 意义与作用

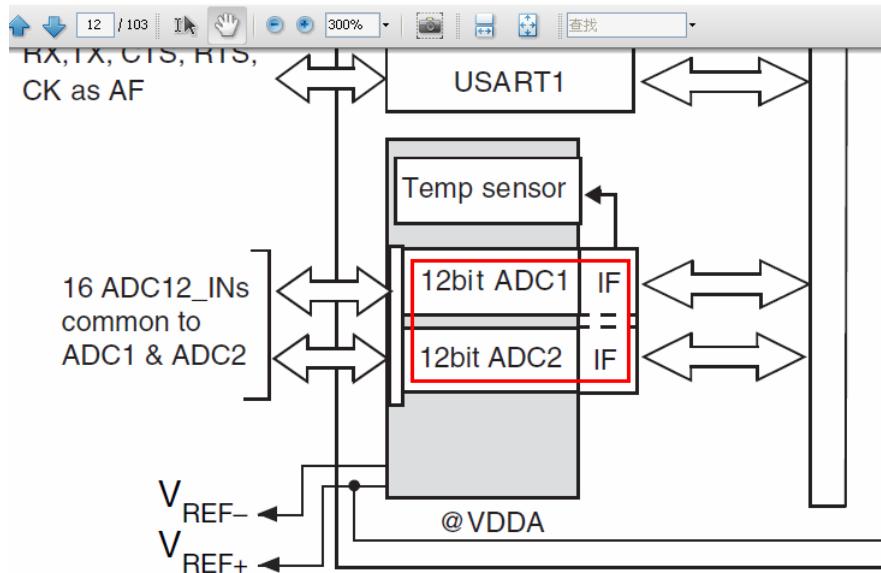
日常中，我们直接接触的都是一些模拟的设备，电位器、传感器、语音和视频等等，那么如何将这些设备采集到的数据进行传输呢？模拟数据在传输过程中，数据量大，占用带宽大，受干扰严重，直接限制影响到我们的通信传输质量。因此，在数据进行传输的第一步都需要进行模拟/数字的转换。将模拟信号进行采样、量化、编码等一系列操作后，再传到通信信道中进行通信。

那么第一步的模拟/数字的转换便是最基础的，所以，这节先让我们简单了解模拟/数字转换的操作，以及对数据进行处理，然后从串口打印计算结果。

7.14.2 实验原理

ADC通过读取PC3管脚输入的模拟信号输出ADC转换值，然后将ADC转换值换算成电压值并通过串口显示。

神舟III号开发板STM32F103ZET6有2个ADC。一般STM32拥有1~3个ADC。STM32F101/102系列有1个ADC，STM32F103系列一般有2个ADC。STM32F207/407一般有3个ADC。这个在ST官方提供的数据手册中可以找到。例如在STM32F103的数据手册的第12页我们就可以看到芯片的ADC资源：



神舟系列开发板ADC的转换结果是12位二进制数，最小转换结果为0x0000，最大转换结果为0xFFFF，所以我们定义一个16位的变量（ADCConvertedValue）来保存转换结果。

STM32处理器的ADC是一个12位的逐次逼近型模拟数字转换器。它有多达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者右对齐方式存储在16位数据寄存器中。需要强调的是ADC的输入时钟不得超过14MHz，并有PCLK2经分频产生。神舟系列开发板的ADC是一种逐次逼近型模拟数字转换器，它的转换结果是12位二进制数，最快转换速度1uS，这么优秀的性能不是一般的MCU所能具有的。

STM32将ADC的转换分为2个通道组：规则通道组和注入通道组。

注入通道的转换可以打断规则通道的转换，在注入通道被转换完成之后，规则通道才得以继续转换。这个和中断是类似的，在实际应用中给我们提供方便。

我们通过例子说明：假如在蔬菜大棚外放了几个温度湿度传感器，大棚内也放了几个温度湿度传感器。我们主要监控的是大棚内的温度/湿度情况，需要时刻监控。但偶尔也需要看看大棚外的温度湿度情况；因此我们可以使用规则通道组循环扫描大棚内的传感器并显示AD转换结果，当你想看大棚外温度湿度情况时，通过一个按钮启动注入转换组并暂时显示大棚外的信息，当你放开这个按钮后，系统又会回到规则通道组继续检测大棚内的信息。

从系统设计上，测量并显示大棚外温度/湿度的过程中断了测量并显示大棚内温度/湿度的过程，但程序设计上可以在初始化阶段分别设置好不同的转换组，系统运行中不必再变更循环转换的配置，从而达到两个任务互不干扰和快速切换的结果。可以设想一下，如果没有规则组和注入组的划分，当你按下按钮后，需要从新配置AD循环扫描的通道，然后在释放按钮后需再次配置AD循环扫描的通道。

对于ADC中涉及的几个寄存器，如ADC控制器（ADC_CR）、ADC的采样事件寄存器（ADC_SMPR）、ADC规则序列寄存器（ADC_SQR）以及ADC规则数据寄存器（ADC_DR），这几个寄存器的使用在此就不展开描述，大家有兴趣可以查阅《【中文】STM32F系列ARM内核32位高性能微处理器参考手册V10_1》资料155页开始的关于ADC的描述。此部分涉及到的寄存器在ST标准库的外设驱动“stm32f10x_adc.c”中有相关的定义以及初始化。ADC的工程项目中，直接调用相关函数与定义。

为什么需要DMA？

谈到ADC时，我们就有必要提及DMA方式。因为规则通道转换的值存储在一个仅有的数据寄存器中，所有当转换多个规则通道时，就需要使用DMA，否则将导致已经存储在ADC_DR寄存器中的数据丢失。同时ADC采集数据时采集一次数据，结果是不准确的，所以要多采几次。那么数据量就大了。这里就用到DMA。

只有在规则通道的转换结束时，才产生DMA请求，并将转换的数据从ADC_DR寄存器传输到用户指定的目的地址。

DMA: Direct Memory Access（存储器直接访问）是指一种高速的数据传输操作，提供在外设和存储器之间或是存储器和存储器之间的高速数据传输。CPU除了在数据传输开始和结束时做一点处理外，在传输过程中CPU可以进行其他的工作。这样，在大部分时间里，CPU和输入输出都处于并行操作状态。因此，使整个系统的效率大大提高。这样便为快速，高性能的ADC提供了通道。简单了解DMA
嵌入式专业技术论坛（www.armjishu.com）出品

具有多个channel即可。详见《【中文】STM32F系列ARM内核32位高性能微处理器参考手册V10_1》资料142页关于DMA的描述。

下面讲解ADC的设置步骤：

1) 开启PC口时钟，设置PC3为模拟输入模式。

本实验我们用到的是ADC1的通道13。神舟III号开发板的主芯片是STM32F103ZET6，通道13对应的引脚是PC3。各个通道对应的管脚大家参考数据手册。例如：

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

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

2) 开启DMA时钟，对DMA进行配置。

为什么使用DMA，这里就不再重复。

3) 开启ADC1时钟，对ADC1参数进行配置。

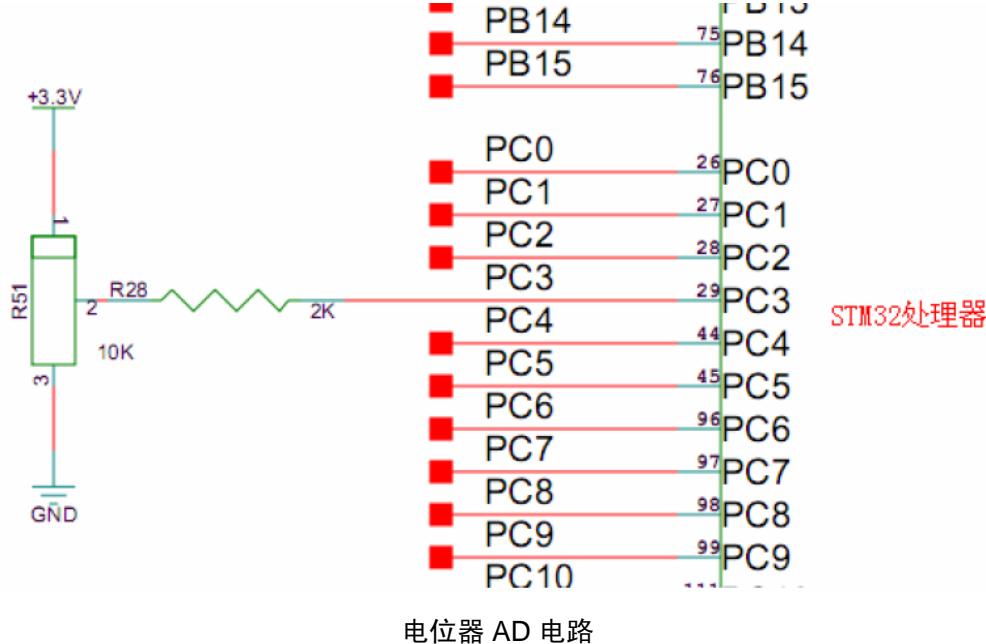
4) 对ADC1使用的通道组进行设置，对采用顺序，转换时间进行设置。

5) 使能ADC并校准

6) 读取ADC的值

7.14.3 硬件设计

神舟III号开发板载有电位器与STM32处理器的PC3管脚相连，可以实现AD模数转换实现。外部模拟信号由电位器（可调电阻）对3.3V电压分压得到，其硬件原理图如下图所示：



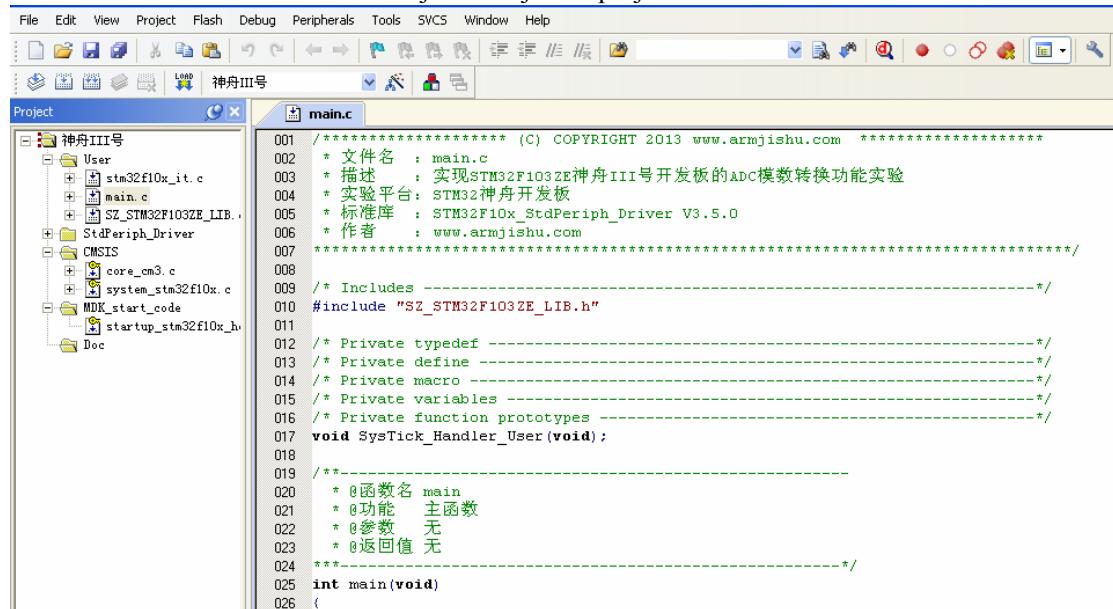
R51 为 10K 欧的电位器（类似于滑动变阻器），其管脚 2 为滑片的输出，根据分压原理，其输出电压为：

$$V_{out} = V_{cc} * (R_{top} / R_{v1})$$

其中 V_{cc} 为 3 脚的输入电压，本次是 3.3V； R_{top} 为 2 脚与 3 脚的电阻值，与滑片当前的位置有关； R51 为电位器 1 脚与 3 脚之间的电阻，本次为 10K。

7.14.4 软件设计

进入例程的文件夹，然后打开Project\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
    */
    /*!< 在系统启动文件 (startup_stm32f10x_xx.s) 中已经调用 SystemInit() 初始化了时钟,
       所以 main 函数不需要再次重复初始化时钟。默认初始化系统主时钟为 72MHz。
       SystemInit() 函数的实现位于 system_stm32f10x.c 文件中。
    */

    uint16_t ADCConvertedValueLocal, Precent = 0, Voltage = 0;

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED4);

    /* 注意串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为sz_STM32_COM2 */
    /* 串口2初始化 */
    SZ_STM32_COMInit(COM2, 115200);

    printf("\n\r\n");
    printf("\n\r www.armjishu.com 论坛后续还会有很多精彩的示例，欢迎访问论坛交流与学习.");
    printf("\n\r 本示例为AD转换示例，串口输出转换结果，模拟信号来自板上的电位器！\n\r");
    printf("\n\r 开发板上的电位器VR1连接到了STM32的PC0，调节电位器观察串口输出的AD转换结果\n\r");
    printf("\n\r=====\n\r");
    printf("\n\r");

    SZ_STM32_ADC_Configuration();

    /* Infinite loop 主循环 */
    while (1)
}
```

代码分析1：在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了72MHZ时钟，最开始的例程已经对此分析过了，还有不明白的可以看下前面的例程。

代码分析2：调用SZ_STM32_LEDInit()函数初始化板载LED指示灯，前面已经有介绍

代码分析3：调用SZ_STM32_COMInit()函数初始化串口，并向超级终端打印信息。前面已经有介绍。

代码分析4：调用SZ_STM32_ADC_Configuration()函数初始化ADC。

```

void SZ_STM32_ADC_Configuration(void)
{
    ADC_InitTypeDef ADC_InitStructure;          //ADC初始化结构体声明
    DMA_InitTypeDef DMA_InitStructure;          //DMA初始化结构体声明

    ADC_GPIO_Configuration();

    /* Enable DMA1 clock */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //使能DMA时钟

    /* DMA1 channel1 configuration -----*/
    DMA_DeInit(DMA1_Channel1); //开启DMA1的第一通道
    DMA_InitStructure.DMA_PeripheralBaseAddr = DR_ADDRESS; //DMA对应的外设地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADCConvertedValue; //内存存储地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //DMA的转换模式为SRC模式，由外设搬移到内存
    DMA_InitStructure.DMA_BufferSize = 1; //DMA缓存大小，1个
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //接收一次数据后，设备地址禁止后移
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; //关闭接收一次数据后，目标内存地址后移
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //定义外设数据宽度为16位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord; //DMA搬移数据尺寸，HalfWord就是为16位
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; //转换模式，循环缓存模式。
    DMA_InitStructure.DMA_Priority = DMA_Priority_High; //DMA优先级高
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; //M2M模式禁用
    DMA_Init(DMA1_Channel1, &DMA_InitStructure);
    /* Enable DMA1 channel1 */
    DMA_Cmd(DMA1_Channel1, ENABLE);

    /* Enable ADC1 and GPIOC clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); //使能ADC时钟
}

```

代码分析5：进入while循环，通过GetADCConvertedValue()函数，获取ADC模数转换的结果，并将结果转换为电压值。最后通过串口显示信息。到此我们的代码流程基本走完。

```

/* Infinite loop 主循环 */
while (1)
{
    /* LED1指示灯状态取反 */
    SZ_STM32_LEDToggle(LED1);

    ADCConvertedValueLocal = GetADCConvertedValue();
    Precent = (ADCConvertedValueLocal*100/0x1000); //算出百分比
    Voltage = Precent*33; // 3.3v的电平，计算等效电平

    printf("\r\n 当前AD转换结果为: 0x%X, 百分比为: %d%, 电压值: %d.%d%dV.\n\r",
        ADCConvertedValueLocal, Precent, Voltage/1000, (Voltage%1000)/100, (Voltage%100)/10);

    /* 延迟，打印间隔 */
    delay(16000000);

    /* 此处可以添加用户的程序 */
}

```

代码分析6：下面我们详细分析一下初始化ADC函数SZ_STM32_ADC_Configuration()。

```

void SZ_STM32_ADC_Configuration(void)
{
    ADC_InitTypeDef ADC_InitStructure;          //ADC初始化结构体声明
    DMA_InitTypeDef DMA_InitStructure;          //DMA初始化结构体声明

    ADC_GPIO_Configuration();

    /* Enable DMA1 clock */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //使能DMA时钟

    /* DMA1 channel1 configuration -----*/
    DMA_DeInit(DMA1_Channel1); //开启DMA1的第一通道
    DMA_InitStructure.DMA_PeripheralBaseAddr = DR_ADDRESS; //DMA对应的外设地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADCConvertedValue; //内存存储地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //DMA的转换模式为SRC模式，由外设搬移到内存
    DMA_InitStructure.DMA_BufferSize = 1; //DMA缓存大小，1个
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //接收一次数据后，设备地址禁止后移
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; //关闭接收一次数据后，目标内存地址后移
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //定义外设数据宽度为16位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord; //DMA搬移数据尺寸，HalfWord就是为16位
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; //转换模式，循环缓存模式。
    DMA_InitStructure.DMA_Priority = DMA_Priority_High; //DMA优先级高
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; //M2M模式禁用
    DMA_Init(DMA1_Channel1, &DMA_InitStructure);
    /* Enable DMA1 channel1 */
    DMA_Cmd(DMA1_Channel1, ENABLE);
}

```

1) 我们进入初始化ADC函数后，它调用的第一个函数是ADC_GPIO_Configuration()。这个函数将GPIO管脚PC0配置成模拟输入模式(GPIO_Mode_AIN)。每个ADC通道都对应着一个GPIO引脚端口，GPIO的引脚在设置为模拟输入模式后可用于模拟电压的输入端。

```
void ADC_GPIO_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable GPIOC clock */
    /* 使能GPIOC时钟 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);      //

    /* Configure PA.01 (ADC Channel12) as analog input -----*/
    //PC3 作为模拟通道13输入引脚
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;          //管脚3
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;        //输入模式
    GPIO_Init(GPIOC, &GPIO_InitStructure);            //GPIO组
}
```

2) 配置完PC3引脚后，对DMA结构体的使用配置。

```
/* Enable DMA1 clock */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);      //使能DMA时钟

/* DMA1 channel1 configuration -----*/
DMA_DeInit(DMA1_Channel1);           //开启DMA1的第一通道
DMA_InitStructure.DMA_PeripheralBaseAddr = DR_ADDRESS;    //DMA对应的外设地址
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADCConvertedValue; //内存存储地址
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //DMA的转换模式为SRC模式，由外设搬到内存
DMA_InitStructure.DMA_BufferSize = 1;           //DMA缓存大小，1个
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //接收一次数据后，设备地址禁止后移
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; //关闭接收一次数据后，目标内存地址后移
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //定义外设数据宽度为16位
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord; //DMA搬移数据尺寸，HalfWord就是为16位
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; //转换模式，循环缓存模式。
DMA_InitStructure.DMA_Priority = DMA_Priority_High; //DMA优先级高
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;           //M2M模式禁用
DMA_Init(DMA1_Channel1, &DMA_InitStructure);
/* Enable DMA1 channel1 */
DMA_Cmd(DMA1_Channel1, ENABLE);
```

DMA结构体配置的内容包括。使用DMA1的第一个通道DMA1_Channel1，包括时钟设置、目标存储地址、转换模式、数据宽度和优先级等。

数据从ADC外设的数据寄存器(DR_Address)转移到内存(ADCConvertedValue变量)，内存、外设地址都固定，每次传输的数据大小为半字(16位)，使用DMA循环传输模式。DMA传输的外设地址DR_Address是一个自定义的宏：

```
/* STM32芯片ADC转换结果DR寄存器地址 */
#define DR_ADDRESS          ((uint32_t)0x4001244C)
/* 存放ADC为12位模数转换器结果的变量，只有ADCConvertedValue的低12位有效 */
__IO uint16_t ADCConvertedValue; //
```

ADC_DR数据寄存器保存了ADC转换后的数值，以它作为DMA的传输源地址。它的地址是由ADC1外设的地址(0x40012400)加上ADC数据寄存器(ADC_DR)的地址偏移(0x4c)计算得到的。

3) 电位器使用的ADC通道初始化及其重新校准。

```

/* Enable ADC1 and GPIOC clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); //使能ADC时钟

/* ADC1 configuration -----*/
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; //独立的转换模式
ADC_InitStructure.ADC_ScanConvMode = ENABLE; //开启扫描模式
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE; //开启连续转换模式
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None; //ADC外部开关, 关闭状态
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //对齐方式, ADC为12位中, 右对齐方式
ADC_InitStructure.ADC_NbrOfChannel = 1; //开启通道数, 1个
ADC_Init(ADC1, &ADC_InitStructure);

/* ADC1 regular channel10 configuration */
ADC-RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_55Cycles5);
//ADC通道组, 第13个通道 采样顺序1, 转换时间

/* Enable ADC1 DMA */
ADC_DMACmd(ADC1, ENABLE); //ADC命令, 使能
/* Enable ADC1 */
ADC_Cmd(ADC1, ENABLE); //开启ADC1

/* Enable ADC1 reset calibration register */
ADC_ResetCalibration(ADC1); //重新校准
/* Check the end of ADC1 reset calibration register */
while(ADC_GetResetCalibrationStatus(ADC1)); //等待重新校准完成
/* Start ADC1 calibration */
ADC_StartCalibration(ADC1); //开始校准

```

PC3管脚是AD的第13输入通道，在代码中我们完全可以找到对应的配置。ADC通道和GPIO管脚的关系可以从《【英文】STM32F103ZE数据手册V6.pdf》中找到。

代码分析7：函数SZ_STM32_ADC_Configuration()完成初始化以后，ADC将开始转换数据，同时ADC通过DMA方式不断的更新，最新的ADC值，由于DMA通道传输无须CPU干预，由硬件自己完成，因此。程序只需要打印读取的最新的ADC值即可，如下所示程序，重复打印ADC转换的值。

```

while (1)
{
    /* LED1指示灯状态取反 */
    SZ_STM32_LEDToggle(LED1);

    ADCConvertedValueLocal = GetADCConvertedValue();
    Precent = (ADCConvertedValueLocal*100/0x1000); //算出百分比
    Voltage = Precent*33; // 3.3V的电平, 计算等效电平

    printf("\r\n 当前AD转换结果为: 0x%X, 百分比为: %d%%, 电压值: %d.%d%dV.\n\r",
           ADCConvertedValueLocal, Precent, Voltage/1000,
           (Voltage%1000)/100, (Voltage%100)/10);
    /* 延迟, 打印间隔 */
    delay(16000000);
}

```

既然数据从ADC外设的数据寄存器(DR_Address)转移到内存(ADCConvertedValue变量)，那么我们使用函数GetADCConvertedValue()获取ADC模数转换的结果。我们需要转换为电压值。下面计算电压值，计算电压值的公式：

实际电压值=ADC 转换值*Vdd/Vdd_convert_value (0xFFFF)

Vdd 是参考电压，3.3V。Vdd_convert_value 是 4096，因为我们这个 STM32 它的转换的精度是 12 位的，0x1000，2 的 12 次方是 4096。有了这些数据就可以求出我们需要的电压值了。

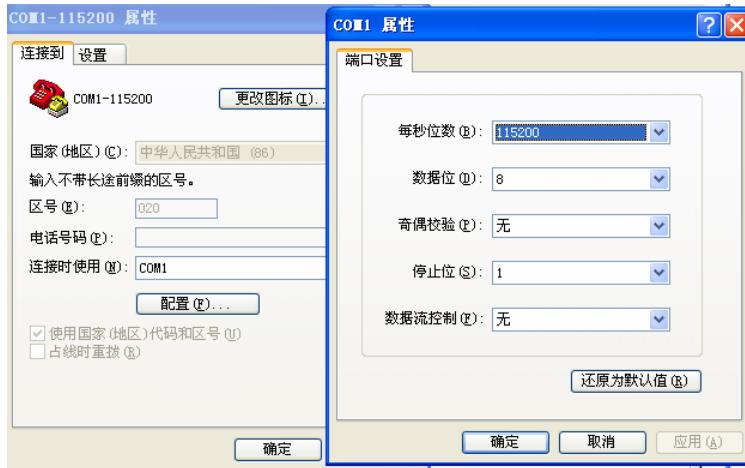
7.14.5 下载与验证

如果使用JLINK下载固件，请按3.2如何使用JLINK软件下载固件到神舟III号开发板小节进行操作。

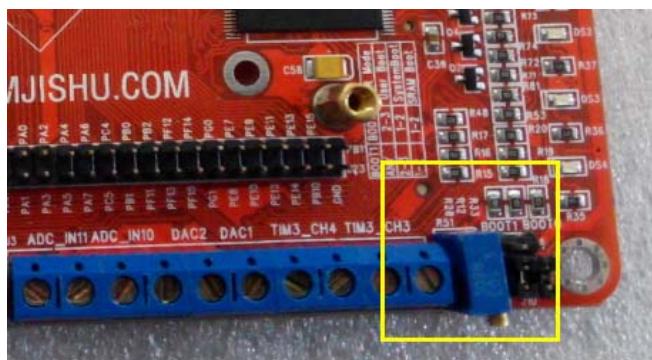
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.14.6 实验现象

将固件下载在神舟III号STM32开发板后，用随板配置的串口线连接神舟III号串口2与电脑的串口，打开超级终端，并按如下参数配置串口。



上电运行神舟III号，串口将打印如下信息，旋转神舟III号板载的电位器，如下图所示，



串口打印的ADC转换电压值随之发生变化，如下图所示，调节电位器时转换结果随之变化，停止调节电位器时其值稳定不变：



以上就是ADC实验现象。

7.15 ADC数据多路采集

7.15.1 意义与作用

7.15.2 实验原理

7.15.3 硬件设计

7.15.4 软件设计

7.15.5 下载与验证

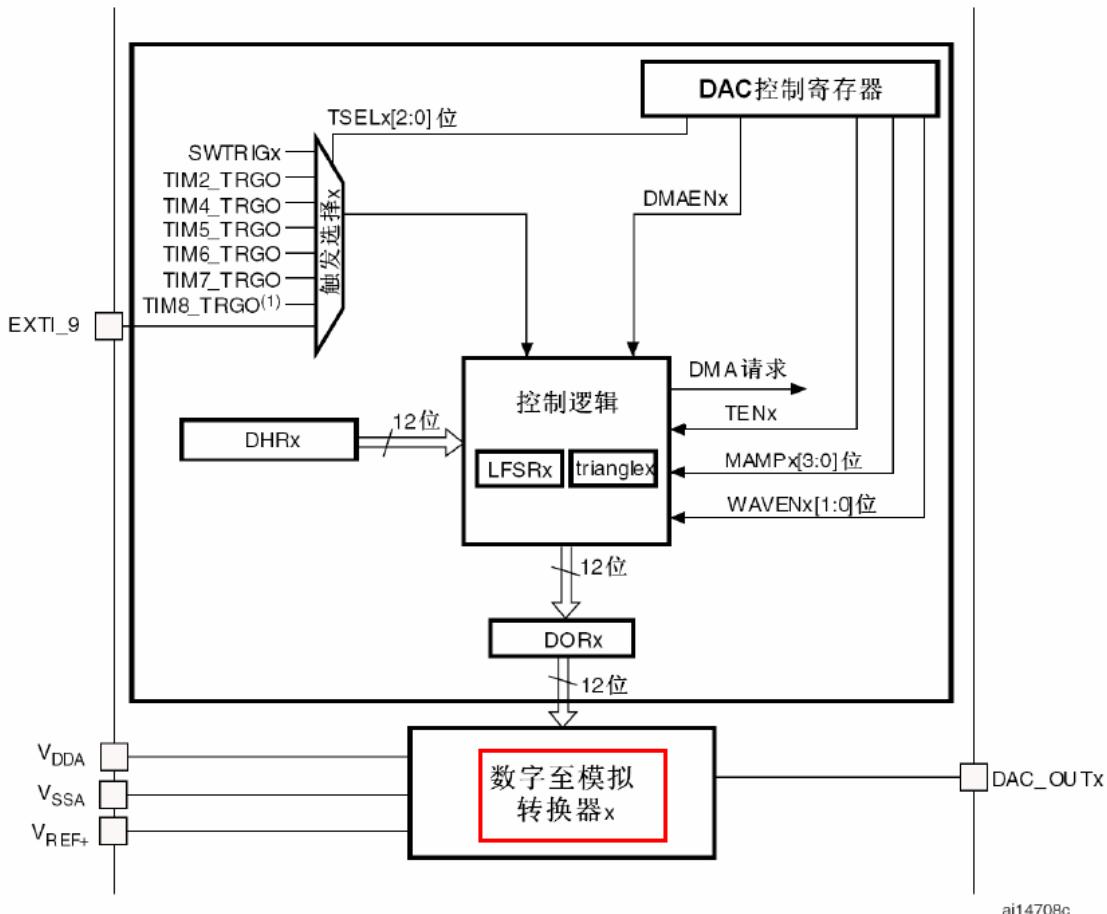
7.15.6 实验现象

7.16 DAC数模转换实验

这一节我们对 DAC 进行学习，DAC: Digital-to-Analog Converter（数模转换器）。它的作用是数字输入，用模拟变化来表示输出。举简单例子：MP3 播放音乐是通过数字信号转换成声音的模拟信号。

7.16.1 STM32 DAC简介

STM32 的数字/模拟转换模块(DAC)是 12 位数字输入，电压输出的数字/模拟转换器。DAC 可以配置为 8 位或 12 位模式，也可以与 DMA 控制器配合使用。DAC 工作在 12 位模式时，数据可以设置成左对齐或右对齐。DAC 模块有 2 个输出通道，每个通道都有单独的转换器。在双 DAC 模式下，2 个通道可以独立地进行转换，也可以同时进行转换并同步地更新 2 个通道的输出。DAC 可以通过引脚输入参考电压 VREF+以获得更精确的转换结果。DAC 通道模块框图如下：



ai14708c

可以看出，DAC 的转换，围绕着部件“**数字至模拟转换器**”进行。DAC 输出是受 DORx 寄存器直接控制的，但是我们不能直接往 DORx 寄存器写入数据，而是通过 DHRx 间接的传给 DORx 寄存器，实现对 DAC 输出的控制。

图中 V_{DDA} 和 V_{SSA} 为模块模拟部分的供电，而 V_{ref+} 则是 DAC 模块的参考电压。DAC_OUTx 是 DAC 的输出通道了（对应 PA4 或者 PA5 引脚）。如下表：

| 名称 | 型号类型 | 注释 |
|-------------------|------------|--|
| V _{REF+} | 输入，正模拟参考电压 | DAC使用的高端/正极参考电压， 2.4V ≤ V _{REF+} ≤ V _{DDA} (3.3V) |
| V _{DDA} | 输入，模拟电源 | 模拟电源 |
| V _{SSA} | 输入，模拟电源地 | 模拟电源的地线 |
| DAC_OUTx | 模拟输出信号 | DAC通道x的模拟输出 |

7.16.2 实验原理

DAC 的运行过程，定时器触发 DAC 转换数据，每当出现定时器更新事件时，由 DMA 运送新的数据到达 DAC 的寄存器，DAC 输出新的数据，而由于这些数据正是一个周期的正弦波的数字形式，经过一个周期的 DAC 转换，就能输出一个连续的模拟正弦波数据。

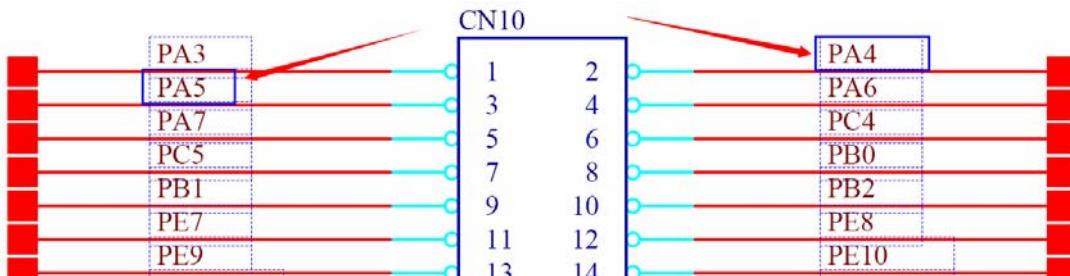
实验中，我们使用双 DAC，产生两路波形。使用到了 DAC、定时器、DMA 。

7.16.3 如何控制正弦波的频率

DAC 数据周期乘以单周期的数据点数就是该正弦波信号的周期。为此我们先要求取 DAC 的数据更新速率，它是由定时器的更新周期决定的。

7.16.4 实验现象

本实验使用到了 DAC、定时器、DMA，它们都属于 STM32 的外设。我们通过 PA4 和 PA5 两个管脚，获得 DAC 输出的波形。



7.16.5 代码分析

我们从主函数开始分析。

```
25 int main(void)
26 {
27     uint32_t Idx = 0;
28
29     DAC_DMA_Config();
30     DAC_TIM_Config();
31     DAC_Config();
32
33     /* 填充正弦波形数据，双通道右对齐*/
34     for (Idx = 0; Idx < 32; Idx++)
35     {
36         DualSine12bit[Idx] = (sine12bit[Idx] << 16) + (sine12bit[Idx]);
37     }
38
39     while(1);
40 }
```

本实验，DAC1 数字输入，电压输出。数据输入使用 DMA 方式。DAC 进行转换的时候需要进行触发，代码中，函数 `DAC_Config()` 配置 DAC，函数 `DAC_TIM_Config()` 配置定时器 2 触发 DMA 数据传输，函数 `DAC_DMA_Config()` DMA 传输数据。`for` 循环。

代码分析 1：函数 `DAC_Config()`。

```

019 void DAC_Config(void)
020 {
021     GPIO_InitTypeDef GPIO_Initstructure;
022     DAC_InitTypeDef DAC_Initstructure;
023
024     /* 使能GPIOA、DAC时钟 */
025     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
026     RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
027
028     /* DAC的GPIO配置，模拟输入 */
029     GPIO_Initstructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
030     GPIO_Initstructure.GPIO_Mode = GPIO_Mode_AIN;
031     GPIO_Init(GPIOA, &GPIO_Initstructure);
032
033     /* 配置DAC 通道1、通道2 */
034     DAC_Initstructure.DAC_Trigger = DAC_Trigger_T2_TRGO; //使用T
035     DAC_Initstructure.DAC_WaveGeneration = DAC_WaveGeneration_None; //不使
036     DAC_Initstructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_LFSRUnmask_Bit0;
037     DAC_Initstructure.DAC_OutputBuffer = DAC_OutputBuffer_Disable; //不使
038     DAC_Init(DAC_Channel_1, &DAC_Initstructure);
039     DAC_Init(DAC_Channel_2, &DAC_Initstructure);
040
041     /* 使能通道1 由PA4输出，通道2 由PA5输出 */
042     DAC_Cmd(DAC_Channel_1, ENABLE);
043     DAC_Cmd(DAC_Channel_2, ENABLE);
044
045     /* 使能DAC的DMA请求 */
046     DAC_DMACmd(DAC_Channel_2, ENABLE);
047 }

```

- 神舟 III 号开发板的主芯片 STM32F103ZET 有 2 个 DAC 转换器，每个 DAC 转换器对应一个输出通道。DAC 输出通道 1 在 PA4 上，输出通道 2 在 PA5 上，如下图所示：

Table 5. Pin definitions (continued)

| Pins | | | Pin name | Type ⁽¹⁾ | I/O Level ⁽²⁾ | Main function ⁽³⁾ (after reset) | Alternate functions ⁽⁴⁾ | |
|--------|--------|---------|----------|---------------------|--------------------------|---|---|------------------|
| BGA100 | LQFP64 | LQFP100 | | | | | Default | Remap |
| G3 | 20 | 29 | PA4 | I/O | | PA4 | SPI1_NSS ⁽⁷⁾ / DAC_OUT1 / USART2_CK ⁽⁷⁾ / ADC12_IN4 | SPI3_NSS/I2S3_WS |
| H3 | 21 | 30 | PA5 | I/O | | PA5 | SPI1_SCK ⁽⁷⁾ / DAC_OUT2 / ADC12_IN5 | |

- 我们先要使能GPIOA端口的时钟，然后设置PA4、PA5为模拟输入。DAC本身是输出，但是为什么端口要设置为模拟输入模式呢？因为一但使能DACx通道之后，相应的GPIO引脚（PA4或者PA5）会自动与DAC的模拟输出相连，设置为输入，是为了避免额外的干扰。

- 配置完GPIO管脚，然后对DAC的结构体成员进行配置。

1) .DAC_Trigger

- DAC工作的时候，将数字量转换为模拟量。它什么时候开始转换，可以通过触发决定。这个触发信号它是有很多种的，本实验我们选择DAC_Trigger_T2_TRGO（定时器2触发）。
- 在ST的参考手册中，我们可以看到触发源如下：

| 触发源 | 类型 | TSELx[2:0] |
|---|--------------|------------|
| 定时器6 TRGO事件 | | 000 |
| 互联型产品为定时器3 TRGO事件 或大容量产品为定时器8 TRGO事件 | | 001 |
| 定时器7 TRGO事件 | 来自片上定时器的内部信号 | 010 |
| 定时器5 TRGO事件 | | 011 |
| 定时器2 TRGO事件 | | 100 |
| 定时器4 TRGO事件 | | 101 |
| EXTI线路9 | 外部引脚 | 110 |
| SWTRIG(软件触发) | 软件控制位 | 111 |

2) .DAC_WaveGeneration

波形发生器。DAC 内部有个波形发生器，可以产生噪声波、三角波。本实验我们选择 DAC_WaveGeneration_None (不使用波形发生器)。

3) .DAC_LFSRUnmask_TriangleAmplitude

用来设置屏蔽/幅值选择器，这个变量只在使用波形发生器的时候才有用，这里我们设置为 0 即可，值为 DAC_LFSRUnmask_Bit0。

4) .DAC_OutputBuffer

输出缓冲，减少输出通道的阻抗，输出通道的阻抗减少了，那么驱动能力就增加了。我们是两个通道输出正弦波，选择 DAC_OutputBuffer_Disable (不使用 DAC 输出缓冲)。

初始化完结构体，调用函数 DAC_Init() 和 DAC_Cmd() 初始化和使能 DAC 通道 DAC_Channel_1、DAC_Channel_2。

代码分析 2：函数 DAC_TIM_Config()

```

055 void DAC_TIM_Config(void)
056 {
057     TIM_TimeBaseInitTypeDef      TIM_TimeBaseStructure;
058
059     /* 使能TIM2时钟，TIM2CLK 为72M */
060     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
>061
062     /* TIM2基本定时器配置 */
063     TIM_TimeBaseStructure.TIM_Period = 19;
064     TIM_TimeBaseStructure.TIM_Prescaler = 0x0;
065     TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
066     TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
067     TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
068
069     /* 配置TIM2触发源 */
070     TIM_SelectOutputTrigger(TIM2, TIM_TriggerSource_Update);
071
072     /* 使能TIM2 */
073     TIM_Cmd(TIM2, ENABLE);
074 }
```

本实验我们在配置 DAC 的时候，配置了定时器 2 触发。这里我们对定时器 2 进行配置，定时器嵌入式专业技术论坛（www.armjishu.com）出品 第 405 页，共 900 页

结构体的各个成员的意义在定时器章节进行了介绍，这里不再重复。我们这里给定时器 2 配置为向上计数，给重载寄存器赋值 19。

该函数中通过调用函数 `TIM_SelectOutputTrigger()` 选择定时器触发方式。最终调用函数 `TIM_Cmd()` 使能定时器。

配置完成，定时器从 0 向上计数，记到我们给定时器设定重载值，定时器重新开始计数，并触发 DAC 模数转换。

代码分析 3：函数 `DAC_DMA_Config()`

```
081 void DAC_DMA_Config(void)
082 {
083     DMA_InitTypeDef DMA_InitStructure;
084
085     /* 使能DMA2时钟 */
086     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA2, ENABLE);
087
088     /* 配置DMA2 */
089     DMA_InitStructure.DMA_PeripheralBaseAddr = DAC_DHR12RD_Address;
090     DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&DualSine12bit;
091     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;
092     DMA_InitStructure.DMA_BufferSize = 32;
093     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
094     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
095     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
096     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
097     DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
098     DMA_InitStructure.DMA_Priority = DMA_Priority_High;
099     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
100     DMA_Init(DMA2_Channel4, &DMA_InitStructure);
101
102     /* 使能DMA2-14通道 */
103     DMA_Cmd(DMA2_Channel4, ENABLE);
104 }
```

DAC 数模转换器，将数字量转换为模拟量，我们需要将数字量传输到 DAC 的数据保持寄存器中。传输的方式可以使用 CPU 运送数据，也可以使用 DMA 传输。本实验我们使用 DMA 传输。

我们分析 DMA 的重要成员配置：

1) .DMA_PeripheralBaseAddr

外设地址，本实验我们赋值 `DAC_DHR12RD_Address`。其定义如下：

```
#define DAC_DHR12RD_Address 0x40007420
```

这里的 `0x40007420` 是怎么来的呢？实际上该值是双 DAC 的 12 位右对齐数据保持寄存器 (`DAC_DHR12RD`) 的地址。寄存器的地址=基址+寄存器的偏移量。这里基址是 `0x40007420`，寄存器的偏移量是 `0x20`。

2) .DMA_MemoryBaseAddr

内存地址，本实验我们赋值(`uint32_t`)`&DualSine12bit`。该地址是我们自行定义：

```
uint32_t DualSine12bit[32];
```

3) .DMA_DIR

数据的传输方向。本实验我们赋值 `DMA_DIR_PeripheralDST`，即方向是由内存到外设（由 `(uint32_t)&DualSine12bit` 到 `DAC_DHR12RD_Address`）。

4) .DMA_BufferSize

缓存大小，我们赋值 32，单位的话由后面的成员决定。

5) .DMA_PeripheralInc 和.DMA_MemoryInc

这两个成员分别设置外设地址和内存地址是否递增。本实验设置外设地址不变，即地址一直是 DAC 的寄存器 **DAC_DHR12RD** 的地址。而内存地址递增，内存是用来存放数据的，传递完一个数据后，内存地址递增，传输另一个数据。单位由后面的成员决定。

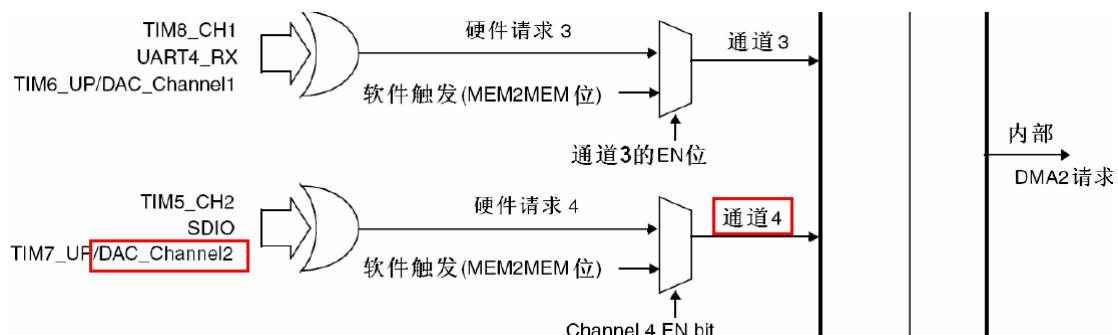
6) .DMA_PeripheralDataSize 和.DMA_MemoryDataSize

外设数据单位和内存数据单位，本实验这两个成员都赋值 16 位的数据宽度。这两个成员决定上面的单位，比如说内存递增，递增的单位的话是字，即每次递增 16 位。

7) .DMA_Mode

本实验我们赋值 **DMA_Mode_Circular**（循环模式）。传输完一次之后，启动下一次传输，不停的循环。

对 DMA 结构体进行赋值完成后，调用 **DMA_Init()** 函数使能 DMA2 的通道 **DMA2_Channel4**。我们 **DAC_Channel2** 使用的是 **DMA2_Channel4**，如下图：



初始化完成，调用函数 **DMA_Cmd(DMA2_Channel4, ENABLE)**，使能 DMA 通道。我们使用了 **DAC_Channel1** 和 **DAC_Channel2** 但是 DMA 配置的代码中我们只配置了 **DAC_Channel2**，这是为什么呢？这个在 ST 给我们提供的手中有说明，如下图：

12.3.7 DMA请求

任一DAC通道都具有DMA功能。2个DMA通道可分别用于2个DAC通道的DMA请求。

如果DMAENx位置'1'，一旦有外部触发(而不是软件触发)发生，则产生一个DMA请求，然后 DAC_DHRx 寄存器的数据被传送到 DAC_DORx 寄存器。

在双DAC模式下，如果2个通道的DMAENx位都为'1'，则会产生2个DMA请求。如果实际只需要一个DMA传输，则应只选择其中一个DMAENx位置'1'。这样，程序可以在只使用一个DMA请求，一个DMA通道的情况下，处理工作在双DAC模式的2个DAC通道。

DAC的DMA请求不会累计，因此如果第2个外部触发发生在响应第1个外部触发之前，则不能处理第2个DMA请求，也不会报告错误。

代码分析 4：填充正弦波形数据。

```
for (Idx = 0; Idx < 32; Idx++)
{
    DualSine12bit[Idx] = (Sine12bit[Idx] << 16) + (Sine12bit[Idx]);
}
```

我们 DMA 传输的时候，设定了内存地址是(**uint32_t**)&**DualSine12bit**。**DualSine12bit** 的定义如下：

```
uint32_t DualSine12bit[32];
```

这里我们通过 `for` 循环对它进行赋值。到此，我们的代码分析完成

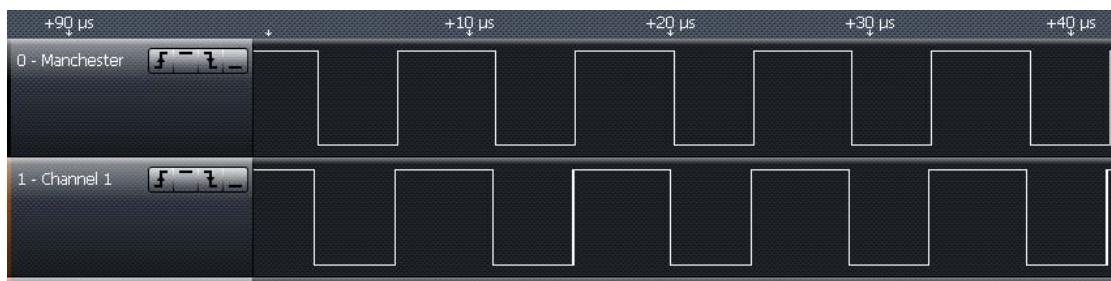
7.16.6 下载与验证

如果使用 JLINK 下载固件，请按[3.2如何使用JLINK软件](#)下载固件到神舟III号开发板小节进行操作。

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.16.7 实验现象

将代码下载到开发板，用杜邦线将开发板上的 PA4, PA5 连接到逻辑分析仪的通道 0 和通道 1。按下复位按键，测得波形如下图：



7.17 RTC实时时钟实验

本章节我们讲解 STM32 的 RTC 原理并通过每秒显示当前实时时间的例程来将掌握 RTC 实时时钟功能及用法。

7.17.1 意义与作用

RTC (Real-time clock) 是实时时钟的意思。神舟 III 号开发板的处理器 STM32F103 集成了 RTC (Real-time clock) 实时时钟，在处理器复位或系统掉电但有实时时钟电池的情况下，能维持系统当前的时间和日期的准确性。实时时钟是一个独立的定时器。RTC 实时时钟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。

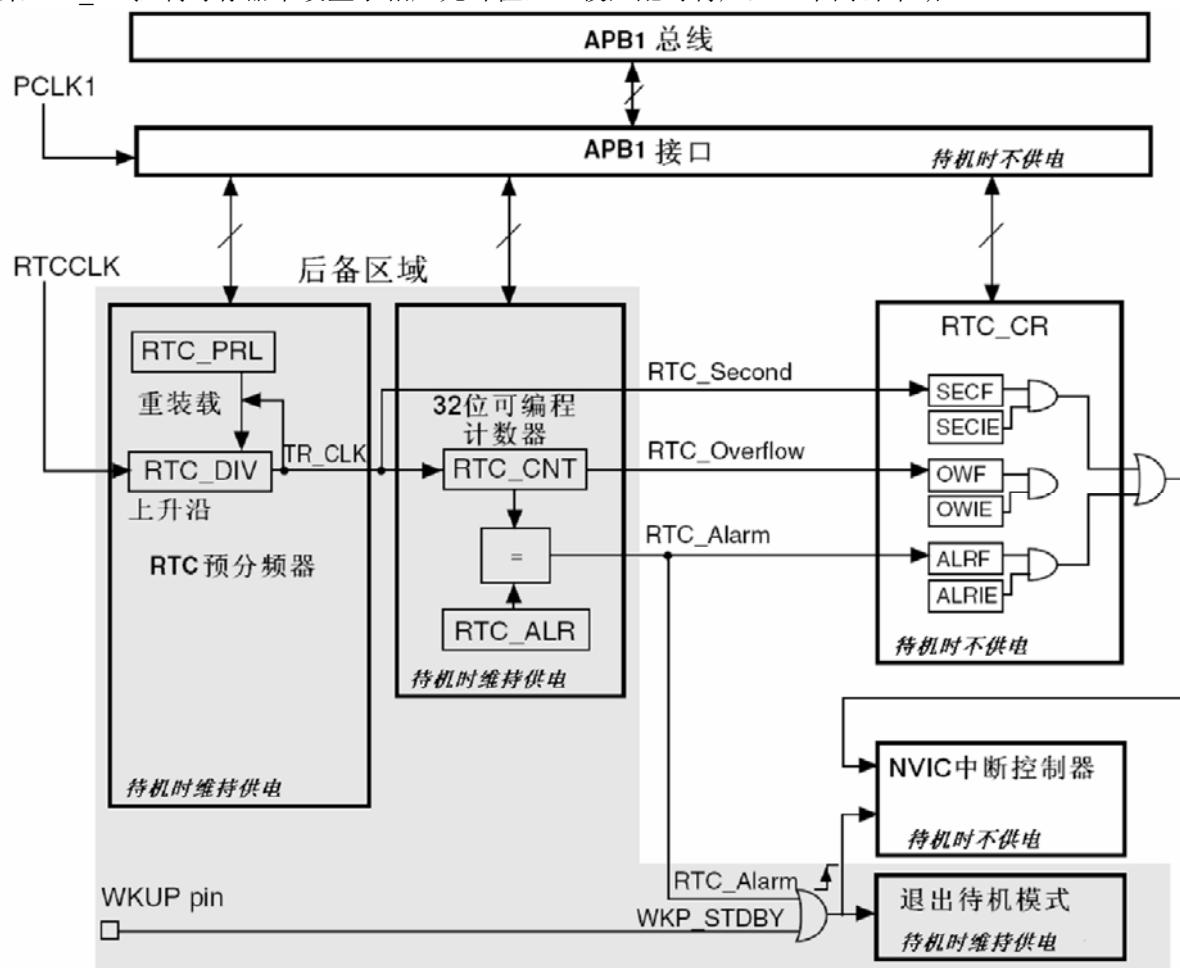
STM32 的 RTC 主要特性为：

- 可编程的预分频系数：分频系数最高为 220。
- 32 位的可编程计数器，可用于较长时间段的测量。
- 2 个分离的时钟：用于 APB1 接口的 PCLK1 和 RTC 时钟(RTC 时钟的频率必须小于 PCLK1 时钟频率的四分之一以上)。
- 可以选择以下三种 RTC 的时钟源：
 - HSE 时钟除以 128；
 - LSE 振荡器时钟；
 - LSI 振荡器时钟。
- 2 个独立的复位类型：

- APB1 接口由系统复位;
- RTC 核心(预分频器、闹钟、计数器和分频器)只能由后备域复位。
- 3 个专门的可屏蔽中断:
 - 闹钟中断，用来产生一个软件可编程的闹钟中断。
 - 秒中断，用来产生一个可编程的周期性中断信号(最长可达 1 秒)。
 - 溢出中断，指示内部可编程计数器溢出并回转为 0 的状态

7.17.2 实验原理

RTC由两个主要部分组成(参见下图)。第一部分(APB1接口)用来和APB1总线相连。此单元还包含一组16位寄存器，可通过APB1总线对其进行读写操作(参见16.4节)。APB1接口由APB1总线时钟驱动，用来与APB1总线接口。另一部分(RTC核心)由一组可编程计数器组成，分成两个主要模块。第一个模块是RTC的预分频模块，它可编程产生最长为1秒的RTC时间基准TR_CLK。RTC的预分频模块包含了一个20位的可编程分频器(RTC预分频器)。如果在RTC_CR寄存器中设置了相应的允许位，则在每个TR_CLK周期中RTC产生一个中断(秒中断)。第二个模块是一个32位的可编程计数器，可被初始化为当前的系统时间。系统时间按TR_CLK周期累加并与存储在RTC_ALR寄存器中的可编程时间相比较，如果RTC_CR控制寄存器中设置了相应允许位，比较匹配时将产生一个闹钟中断。



图表 10 简化的 RTC 框图

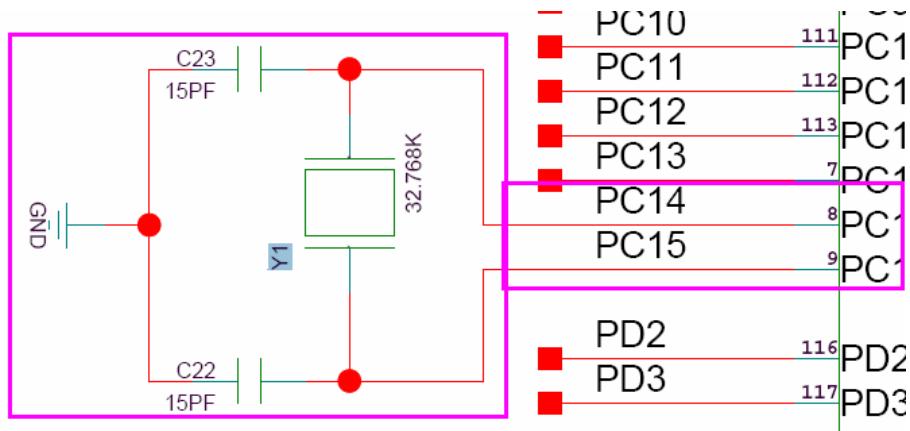
RTC 模块和时钟配置系统(RCC_BDCR 寄存器)处于后备区域，即在系统复位或从待机模式唤醒后，RTC 的设置和时间维持不变。系统复位后，对后备寄存器和 RTC 的访问被禁止，这是为了防止对后备区域(BKP)的意外写操作。执行以下操作将使能对后备寄存器和 RTC 的访问：

- 设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPen 位，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。

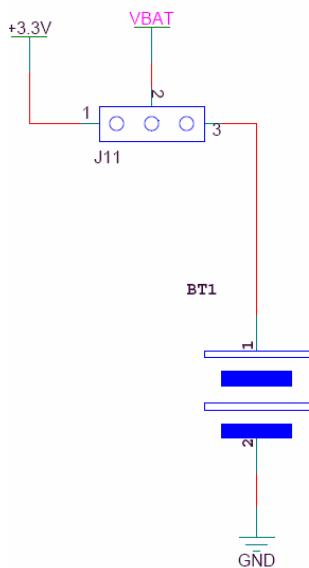
7.17.3 硬件设计

神舟系列开发板的RTC的硬件设计非常简单，其主要硬件都集成在了处理器内部，外围电路只需要一个32.768KHz的晶振和VBAT供电电池即可。

STM32F103内部已经包含了40kHz低速内部RC振荡电路LSE，但是其精准度不是很高，为此我们在外部增加了32.768KHz的晶振电路驱动RTC实时时钟。



STM32的VBAT采用CR1220纽扣电池和VCC3.3混合供电的方式，在有外部电源（VCC3.3）的时候，BT1不给处理器的VBAT供电，而在外部电源断开的时候，则由BT1给VBAT供电。这样，VBAT总是有电的，以保证RTC的持续运行以及后备寄存器的内容不丢失。相关电路如下：



当安装了电池后，VBAT管脚由+3.3V系统电源供电。

7.17.4 软件设计

我们先从main函数讲起，其中涉及的LED指示灯和串口等相关子程序如果之前的章节已有介绍，则此处不再讲述。本示例涉及的程序都添加了较为详细的注释。

```
/**  
 * @brief Main program.  
 * @param None  
 * @retval None  
 */  
  
int main(void)  
{  
    /*!< At this stage the microcontroller clock setting is already configured,  
    this is done through SystemInit() function which is called from startup  
    file (startup_stm32f10x_xx.s) before to branch to application main.  
    To reconfigure the default setting of SystemInit() function, refer to  
    system_stm32f10x.c file  
*/  
  
    /* 初始化神舟开发板上的 LED 指示灯 详见ARMJISHU神舟开发板LED实验章节 */  
    STM_EVAL_LEDInit(LED1);  
    STM_EVAL_LEDInit(LED2);  
    STM_EVAL_LEDInit(LED3);  
    STM_EVAL_LEDInit(LED4);  
  
    /* 关闭其中两个指示灯 */  
    STM_EVAL_LEDOff(LED2);  
    STM_EVAL_LEDOff(LED4);  
  
    /* 配置神舟开发板串口 详见ARMJISHU神舟开发板UART串口输入输出实验章节 */  
    //串口配置与串口输入输出章节的配置方法相同，不是本示例的重点，请参考相关章节。  
    printf("\r\n\r\n\r\n\r WWW.ARmjishu.COM USART2 configured....");  
    |  
    STM_EVAL_LEDOn(LED3);  
  
    InterruptConfig();  
  
    /* NVIC configuration */  
    NVIC_Configuration();  
  
    /* 以下if...else.... if判断系统时间是否已经设置，判断RTC后备寄存器1的值  
    是否为事先写入的0xA5A5，如果不是，则说明RTC是第一次上电，需要配置RTC，  
    提示用户通过串口更改系统时间，把实际时间转化为RTC计数值写入RTC寄存器，  
    并修改后备寄存器1的值为0XA5A5。  
    else表示已经设置了系统时间，打印上次系统复位的原因，并使能RTC秒中断  
*/  
    if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)  
    {  
        /* 以下if...else.... if判断系统时间是否已经设置，判断RTC后备寄存器1的值  
        是否为事先写入的0XA5A5，如果不是，则说明RTC是第一次上电，需要配置RTC，  
        提示用户通过串口更改系统时间，把实际时间转化为RTC计数值写入RTC寄存器，  
        并修改后备寄存器1的值为0XA5A5。  
        else表示已经设置了系统时间，打印上次系统复位的原因，并使能RTC秒中断  
*/  
        if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)  
        {  
            /* Backup data register value is not correct or not yet programmed (when  
            the first time the program is executed) */  
            printf("\r\n\r RTC not yet configured....");  
  
            /* RTC Configuration */  
            RTC_Configuration();  
  
            printf("\r\n RTC configured....");  
  
            /* Adjust time by values entred by the user on the hyperterminal */  
            Time_Adjust();  
            /* 修改后备寄存器1的值为0XA5A5 */  
            BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);  
        }  
    }  
}
```

```

{
    /* Check if the Power On Reset flag is set */
    if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
    {
        printf("\r\n\n Power On Reset occurred....");
    }
    /* Check if the Pin Reset flag is set */
    else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
    {
        printf("\r\n\n External Reset occurred....");
    }

    printf("\r\n No need to configure RTC....");
    /* Wait for RTC registers synchronization */
    RTC_WaitForSynchro();

    /* Enable the RTC Second */
    RTC_ITConfig(RTC_IT_SEC, ENABLE);

    /* Wait until last write operation on RTC registers has finished */
    RTC_WaitForLastTask();
} ? end else ?
上面的函数已经打印了复位的原因，此处RCC_ClearFlag()函数的目的是清除上次复位原因的记录。
/* Clear reset flags */
RCC_ClearFlag();

/* Display time in infinite loop */
printf("\r\n\r\n WWW.ARmjishu.COM Display time in infinite loop....\r\n");
Time_Show();
} ? end main ?

```

函数Time_Show()是一个while死循环，每秒打印一次系统时间。全局变量TimeDisplay是否为1，如果为1则打印时间并将全局变量TimeDisplay清0以便它在"stm32f10x_it.c"文件的"void RTC_IRQHandler(void)"函数中当发生秒中断时再次置为1，这样便实现了精准的每秒输出时间一次，其实现如下：

```

/**
 * @brief Shows the current time (HH:MM:SS) on the Hyperterminal.
 * @param None
 * @retval None
 * 等待并每秒按照(时:分:秒)的格式输出时间一次
 */
void Time_Show(void)
{
    printf("\n\r");

    /* Infinite loop */
    while (1)
    {
        /* If 1s has passed */
        /* 判断全局变量TimeDisplay是否为1，如果为1则时间并将全局变量TimeDisplay清0
           TimeDisplay变量在"stm32f10x_it.c"文件的"void RTC_IRQHandler(void)"函数中当发生秒中断时为置为1，这样便实现了精准的每秒输出时间一次 */
        if (TimeDisplay == 1)
        {
            /* Display current time */
            Time_Display(RTC_GetCounter());
            TimeDisplay = 0;
        }
    }
}

```

Time_Show()函数中又调用了两个子函数Time_Display()和RTC_GetCounter()，其中RTC_GetCounter()函数只是简单的读取RTC的Counter寄存器的值，其实现如下：

```

/**
 * @brief Gets the RTC counter value.
 * @param None
 * @retval RTC counter value.
 */
uint32_t RTC_GetCounter(void)
{
    uint16_t tmp = 0;
    tmp = RTC->CNTL;
    return ((uint32_t)RTC->CNTH << 16) | tmp;
}

```

而Time_Display()函数是实现将RTC_GetCounter()读到RTC的Counter寄存器的值幻化为时分秒的时间信息并打印，其实现如下：

```
/**  
 * @brief Displays the current time.  
 * @param TimeVar: RTC counter value.  
 * @retval None  
 * 把RTC寄存器的数值转化为时间为并通过串口打印  
 */  
void Time_Display(uint32_t TimeVar)  
{  
    uint32_t THH = 0, TMM = 0, TSS = 0;  
  
    /* Compute hours */  
    THH = (TimeVar / 3600) % 24;  
    /* Compute minutes */  
    TMM = (TimeVar % 3600) / 60;  
    /* Compute seconds */  
    TSS = (TimeVar % 3600) % 60;  
  
    /* 通过串口打印时间，注意此处的'\r'保证在串口同一行覆盖输出 */  
    printf("Time: %0.2d:%0.2d:%0.2d\r", THH, TMM, TSS);  
}
```

其中的中断相关函数解释如下：

“InterruptConfig()”函数告诉处理器中断向量表存放的起始地址，STM32支持中断向量表起始地址动态设置，这个特性在SRAM调试和DFU固件升级时很有用，以为这些情况下中断向量表起始地址已经不是0x0000处。此处将中断向量表起始地址设置为内部Flash的起始地址0x08000000处。

```
/* Private functions -----*/  
void InterruptConfig(void)  
{  
    /* Set the Vector Table base address at 0x08000000 */  
    /* 设置中断向量表的位置为内部FLASH的0x08000000，详见ARMJISHU神舟DFU实验章节 */  
    NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0000);  
}
```

NVIC_Configuration 函数实现配置嵌套向量中断中断优先级并使能中断。其中的 NVIC_PriorityGroupConfig 函数配置中断优先级的组织方式，STM32 的嵌套向量中断控制器可以配置 16 个可编程的优先等级，使用了 4 位表示中断优先级（2 的 4 次方就是 16），16 个可编程的优先等级又可以分为主优先级和次优先级，例如参数 NVIC_PriorityGroup_1 表示 1bit 主优先级（pre-emption priority）3 bits 次优先级（subpriority）。

```
/**  
 * @brief Configures the nested vectored interrupt controller.  
 * @param None  
 * @retval None  
 */  
void NVIC_Configuration(void)  
{  
    NVIC_InitTypeDef NVIC_InitStructure;  
  
    /* Configure one bit for preemption priority */  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);  
  
    /* Enable the RTC Interrupt 使能RTC中断 */  
    NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;  
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;  
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;  
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;  
    NVIC_Init(&NVIC_InitStructure);  
}
```

以下为中断处理函数，请看代码注释。注意其中的对全局变量TimeDisplay的置一与Time_Show()函数中的清零操作相呼应，是实现精确1秒延时的关键。

```
/***
 * @brief This function handles RTC global interrupt request.
 * @param None
 * @retval None
 */
void RTC_IRQHandler(void)
{
    /* 判断RTC是否发生了秒中断（也有可能是溢出或者闹钟中断） */
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
    {
        /* Clear the RTC Second interrupt */
        /* 清除秒中断标志 */
        RTC_ClearITPendingBit(RTC_IT_SEC);

        /* Toggle LED1 改变LED1的亮灭状态(取反) */
        STM_EVAL_LEDToggle(LED1);

        /* Enable time update */
        /* 置全局变量为1，通知主程序打印时间 */
        TimeDisplay = 1;

        /* Wait until last write operation on RTC registers has finished */
        /* 等待上一次对RTC寄存器的写操作完成 */
        RTC_WaitForLastTask();

        /* Reset RTC Counter when Time is 23:59:59 */
        /* 如果时间达到23:59:59则下一刻时间为00:00:00 */
        if (RTC_GetCounter() == 0x000015180)
        {
            RTC_SetCounter(0x0);

            /* Wait until last write operation on RTC registers has finished */
            /* 等待上一次对RTC寄存器的写操作完成 */
            RTC_WaitForLastTask();
        }
    }
    /* end if RTC_GetITStatus(RTC_I... ? */
} /* end RTC_IRQHandler ? */
}
```

-----RTC_Configuration-----

其中的中断相关函数解释如下：

```
/***
 * @brief Configures the RTC.
 * @param None
 * @retval None
 */
void RTC_Configuration(void)
{
    /* Enable PWR and BKP clocks */
    /* PWR时钟（电源控制）与BKP时钟（RTC后备寄存器）使能 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

    /* Allow access to BKP Domain */
    /* 使能RTC和后备寄存器访问 */
    PWR_BackupAccessCmd(ENABLE);

    /* Reset Backup Domain */
    /* 将外设BKP的全部寄存器重设为缺省值 */
    BKP_DeInit();

    /* Enable LSE */
    /* 使能LSE（外部32.768KHz低速晶振） */
    RCC_LSEConfig(RCC_LSE_ON);

    /* Wait till LSE is ready */
    /* 等待外部晶振震荡稳定输出 */
    while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
    {}
}
```

```
/* Select LSE as RTC Clock Source */
/* 使用外部32.768KHz晶振作为RTC时钟 */
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

/* Enable RTC Clock */
/* 使能 RTC 的时钟供给 */
RCC_RTCCLKCmd(ENABLE);

/* Wait for RTC registers synchronization */
/* 等待RTC寄存器同步 */
RTC_WaitForSynchro();

/* Wait until last write operation on RTC registers has finished */
/* 等待上一次对RTC寄存器的写操作完成 */
RTC_WaitForLastTask();

/* Enable the RTC Second */
/* 使能RTC的秒中断 */
RTC_ITConfig(RTC_IT_SEC, ENABLE);

/* Wait until last write operation on RTC registers has finished */
/* 等待上一次对RTC寄存器的写操作完成 */
RTC_WaitForLastTask();

/* Set RTC prescaler: set RTC period to 1sec */
/* 32.768KHz晶振预分频值是32767,如果对精度要求很高可以修改此分频值来校准晶振 */
RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */

/* Select LSE as RTC Clock Source */
/* 使用外部32.768KHz晶振作为RTC时钟 */
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

/* Enable RTC Clock */
/* 使能 RTC 的时钟供给 */
RCC_RTCCLKCmd(ENABLE);

/* Wait for RTC registers synchronization */
/* 等待RTC寄存器同步 */
RTC_WaitForSynchro();

/* Wait until last write operation on RTC registers has finished */
/* 等待上一次对RTC寄存器的写操作完成 */
RTC_WaitForLastTask();

/* Enable the RTC Second */
/* 使能RTC的秒中断 */
RTC_ITConfig(RTC_IT_SEC, ENABLE);

/* Wait until last write operation on RTC registers has finished */
/* 等待上一次对RTC寄存器的写操作完成 */
RTC_WaitForLastTask();

/* Set RTC prescaler: set RTC period to 1sec */
/* 32.768KHz晶振预分频值是32767,如果对精度要求很高可以修改此分频值来校准晶振 */
RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */

/* Wait until last write operation on RTC registers has finished */
/* 等待上一次对RTC寄存器的写操作完成 */
RTC_WaitForLastTask();
} ? end RTC_Configuration ?
```

上述多个程序中都调用了以下等待的子函数调用：

```
/* Wait until last write operation on RTC registers has finished */
RTC_WaitForLastTask();
```

为什么要不停的等待呢？

RTC核完全独立于RTC APB1接口。 软件通过APB1接口访问RTC的预分频值、计数器值和闹钟值。但是，相关的可读寄存器只在与RTC APB1时钟进行重新同步的RTC时钟的上升沿被更新。RTC标志也是如此的。这意味着，如果APB1接口曾经被关闭，而读操作又是在刚刚重新开启APB1之后，则在第一次的内部寄存器更新之前，从APB1上读出的RTC寄存器数值可能被破坏了(通常读到0)。因此，若在读取RTC寄存器时，RTC的APB1接口曾经处于禁止状态，则软件首先必须等待RTC_CRL寄存器中的RSF位(寄存器同步标志)被硬件置‘1’。(注：RTC的APB1接口不受WFI和WFE等低功耗模式的影响。)

----- Time_Adjust-----

将用户输入的时分秒信息转换为计数器的计数值并对RTC计数器赋值：

```
/*
 * @brief Adjusts time.
 * @param None
 * @retval None
 * 把时间转化为RTC计数值写入RTC寄存器
 */
void Time_Adjust(void)
{
    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();

    /* Change the current time */
    /* 把时间转化为RTC计数值写入RTC寄存器 */
    RTC_SetCounter(Time_Regulate());
}

/* Wait until last write operation on RTC registers has finished */
/* 等待上一次对RTC寄存器的写操作完成 */
RTC_WaitForLastTask();
```

Time_Regulate函数多次调用用户输入函数，请求用户输入时分秒信息并将其转化为计数数值。

```
/*
 * @brief Returns the time entered by user, using Hyperterminal.
 * @param None
 * @retval Current time RTC counter value
 */
uint32_t Time_Regulate(void)
{
    uint32_t Tmp_HH = 0xFF, Tmp_MM = 0xFF, Tmp_SS = 0xFF;

    printf("\r\n=====Time Settings=====");
    printf("\r\n Please Set Hours");

    while (Tmp_HH == 0xFF)
    {
        Tmp_HH = USART_Scanf(23);
    }
    printf(": %d", Tmp_HH);
    printf("\r\n Please Set Minutes");
    while (Tmp_MM == 0xFF)
    {
        Tmp_MM = USART_Scanf(59);
    }
    printf(": %d", Tmp_MM);
    printf("\r\n Please Set Seconds");
    while (Tmp_SS == 0xFF)
    {
        Tmp_SS = USART_Scanf(59);
    }
    printf(": %d", Tmp_SS);

    /* Return the value to store in RTC counter register */
    return ((Tmp_HH*3600 + Tmp_MM*60 + Tmp_SS));
} ? end Time_Regulate ?
```

用户输入函数USART_Scanf，请求用户输入小于等于输入参数的数值并将该值返回。

```
uint8_t USART_Scanf(uint32_t value)
{
    uint32_t index = 0;
    uint32_t tmp[2] = {0, 0};

    while (index < 2)
    {
        /* Loop until RXNE = 1 */
        while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_RXNE) == RESET)
        {}
        tmp[index++] = (USART_ReceiveData(EVAL_COM1));
        if ((tmp[index - 1] < 0x30) || (tmp[index - 1] > 0x39))
        {
            printf("\r\nPlease enter valid number between 0 and 9");
            index--;
        }
        /* Calculate the corresponding value */
        index = (tmp[1] - 0x30) + ((tmp[0] - 0x30) * 10);
        /* Checks */
        if (index > value)
        {
            printf("\r\nPlease enter valid number between 0 and %d", value);
            return 0xFF;
        }
    }
    return index;
} ? end USART_Scanf ?
```

7.17.5 下载与验证

如果使用JLINK下载固件,请按[3.2如何使用JLINK软件](#)下载固件到神舟III号开发板小节进行操作。

如果使用USB下载固件,请按[错误!未找到引用源。错误!未找到引用源。](#)小节进行操作。

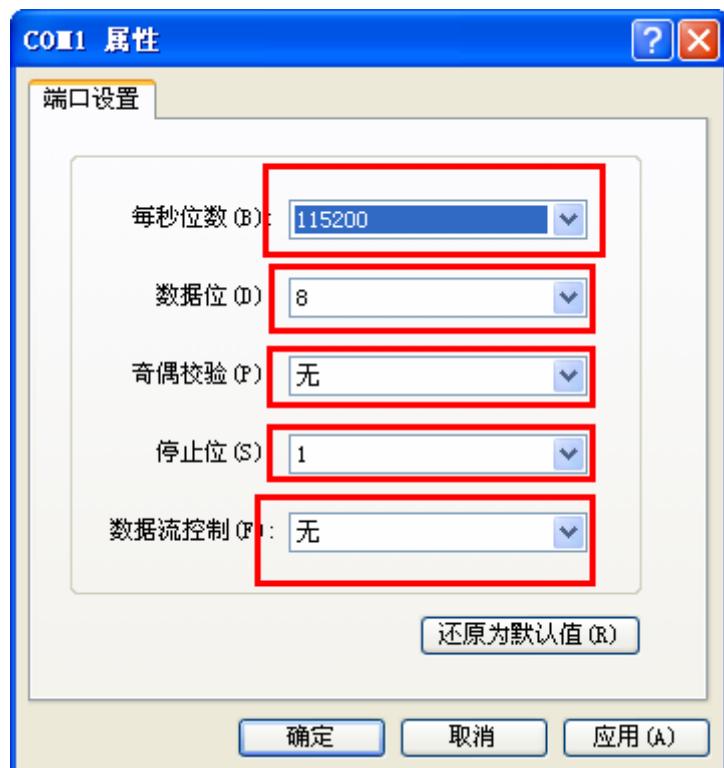
如果在MDK开发环境中,下载编译好的固件或者在线调试,请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.17.6 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后,关闭电源,用串口线神舟 III 号串口 2 与电脑连接,并打开超级终端,按以下设置,如下图:



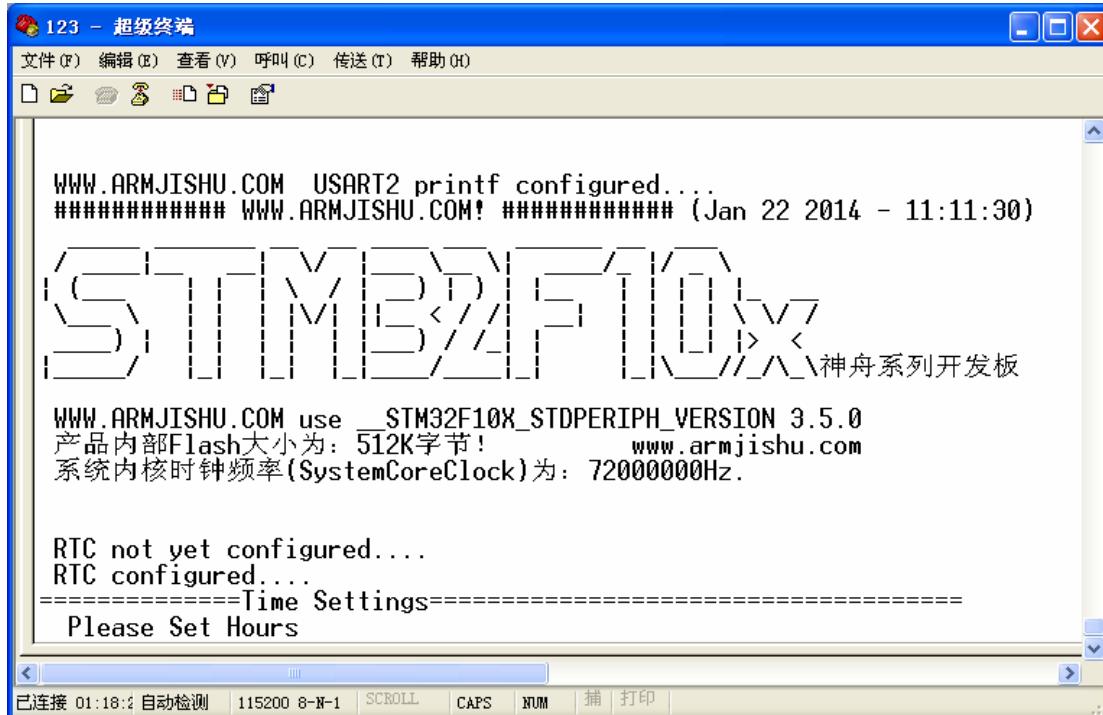
选择 COM1; 按确定



再按确定，完成超级终端设置。

重新打开电源；神舟 III 号 STM32 开发板上 LED 2 和 LED 3 常亮，LED 1 和 LED 4 间断亮灭。

下载后串口会有打样信息输出，如果是第一次运行该程序，则串口会输出提示“RTC not yet configured....”然后会要求用户输入时分秒信息，然后就是每秒刷新一次系统时间。如下图



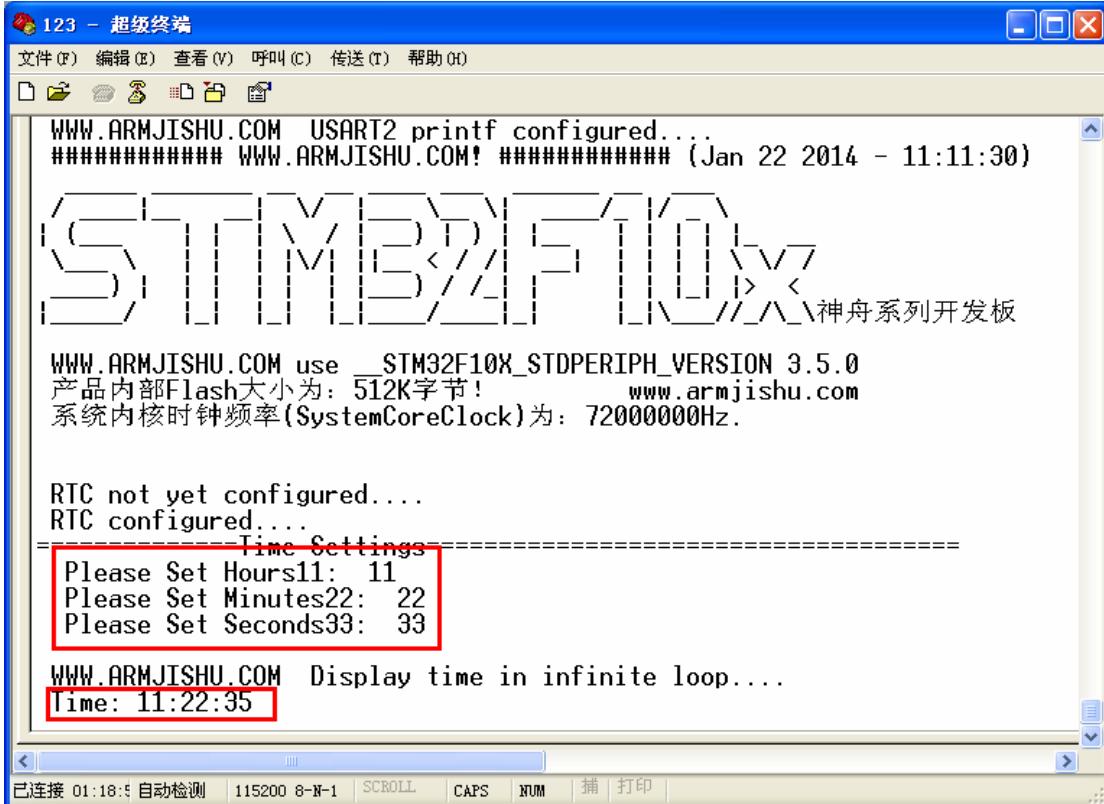
按照超级终端窗口显示信息提示输入时间信息，

如输入小时 11

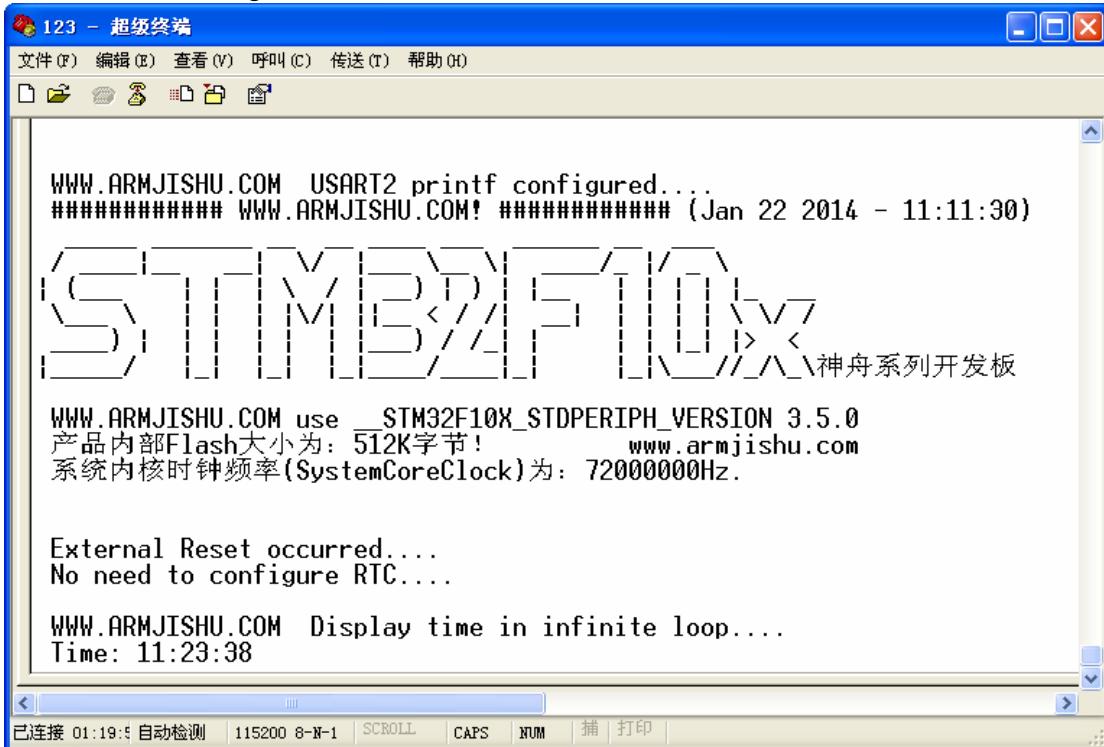
如输入分钟 22

如输入秒钟 33

超级终端窗口显示，以下图：



如果上次运行时已经设置了系统时间，在复位重启或者有RTC电池时断电重启后，串口会输出提示“*No need to configure RTC....*”然后每秒刷新一次系统时间。如下图所示：



7.18 Calendar实时时钟与农历年月日实验

上一章节我们讲解了 STM32 的 RTC 原理并介绍了通过每秒显示当前实时时间的例程。本章节我们在上一章节的继续上继续讲解 STM32 的 RTC 的高级应用，不仅实现当前时分秒的现实，而且实现 **Calendar** 农历年月日与节气和公历日历年月日时分秒的计算和显示，在乐趣中让大家更深入的掌握嵌入式专业技术论坛（www.armjishu.com）出品

RTC 实时时钟功能及用法。

7.18.1 意义与作用

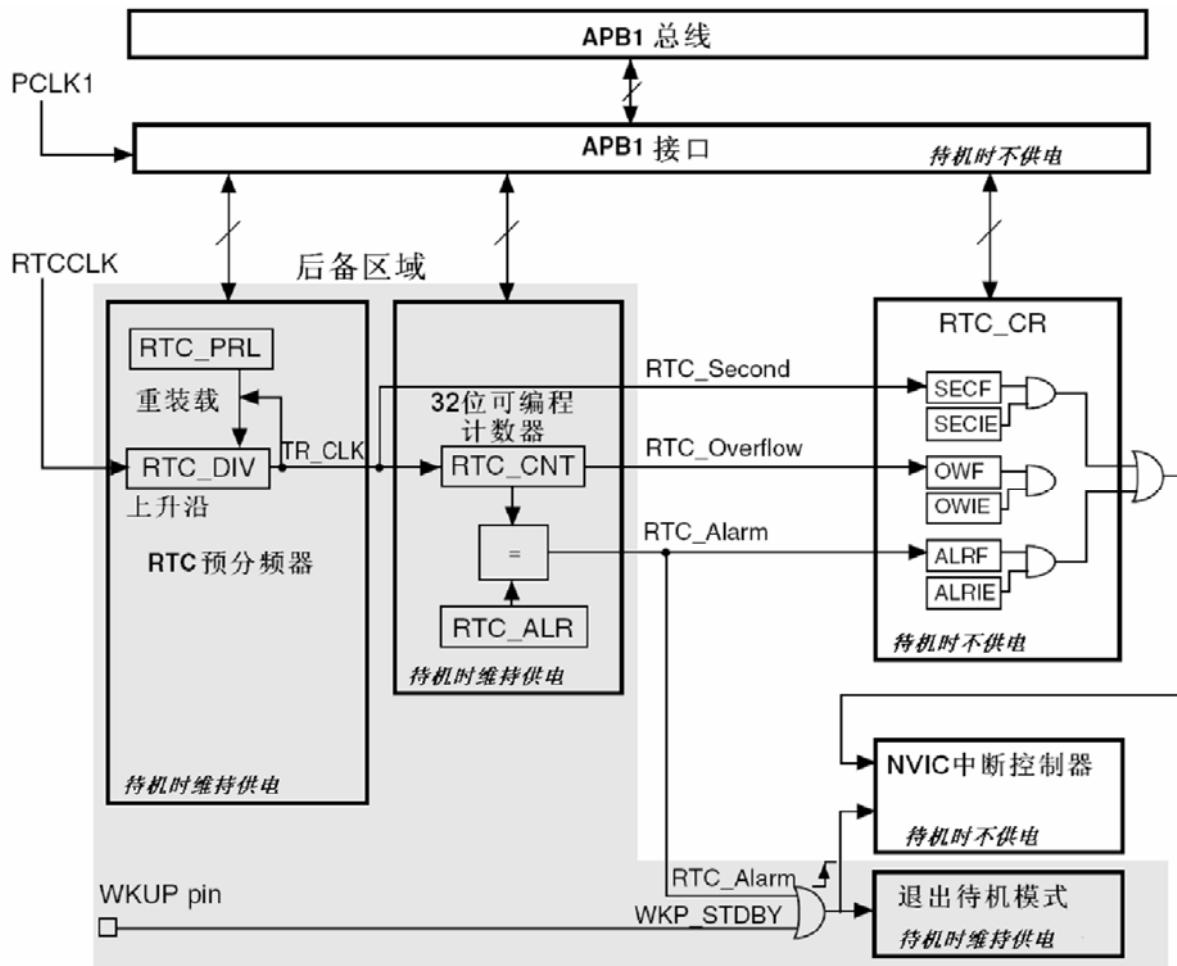
RTC (Real-time clock) 是实时时钟的意思。神舟 III 号开发板的处理器 STM32F103 集成了 RTC (Real-time clock) 实时时钟，在处理器复位或系统掉电但有实时时钟电池的情况下，能维持系统当前的时间和日期的准确性。实时时钟是一个独立的定时器。RTC 实时时钟模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。本次实验不仅实现当前时分秒的现实，而且实现 **Calendar** 农历年月日与节气和公历年月日时分秒的计算和显示，在乐趣中让大家更深入的掌握 RTC 实时时钟功能及用法。

STM32 的 RTC 主要特性为：

- 可编程的预分频系数：分频系数最高为 220。
- 32 位的可编程计数器，可用于较长时间段的测量。
- 2 个分离的时钟：用于 APB1 接口的 PCLK1 和 RTC 时钟(RTC 时钟的频率必须小于 PCLK1 时钟频率的四分之一以上)。
- 可以选择以下三种 RTC 的时钟源：
 - HSE 时钟除以 128;
 - LSE 振荡器时钟;
 - LSI 振荡器时钟。
- 2 个独立的复位类型：
 - APB1 接口由系统复位；
 - RTC 核心(预分频器、闹钟、计数器和分频器)只能由后备域复位。
- 3 个专门的可屏蔽中断：
 - 闹钟中断，用来产生一个软件可编程的闹钟中断。
 - 秒中断，用来产生一个可编程的周期性中断信号(最长可达 1 秒)。
 - 溢出中断，指示内部可编程计数器溢出并回转为 0 的状态

7.18.2 实验原理

RTC由两个主要部分组成(参见下图)。第一部分(APB1接口)用来和APB1总线相连。此单元还包含一组16位寄存器，可通过APB1总线对其进行读写操作(参见16.4节)。APB1接口由APB1总线时钟驱动，用来与APB1总线接口。另一部分(RTC核心)由一组可编程计数器组成，分成两个主要模块。第一个模块是RTC的预分频模块，它可编程产生最长为1秒的RTC时间基准TR_CLK。RTC的预分频模块包含了一个20位的可编程分频器(RTC预分频器)。如果在RTC_CR寄存器中设置了相应的允许位，则在每个TR_CLK周期中RTC产生一个中断(秒中断)。第二个模块是一个32位的可编程计数器，可被初始化为当前的系统时间。系统时间按TR_CLK周期累加并与存储在RTC_ALR寄存器中的可编程时间相比较，如果RTC_CR控制寄存器中设置了相应允许位，比较匹配时将产生一个闹钟中断。



图表 11 简化的 RTC 框图

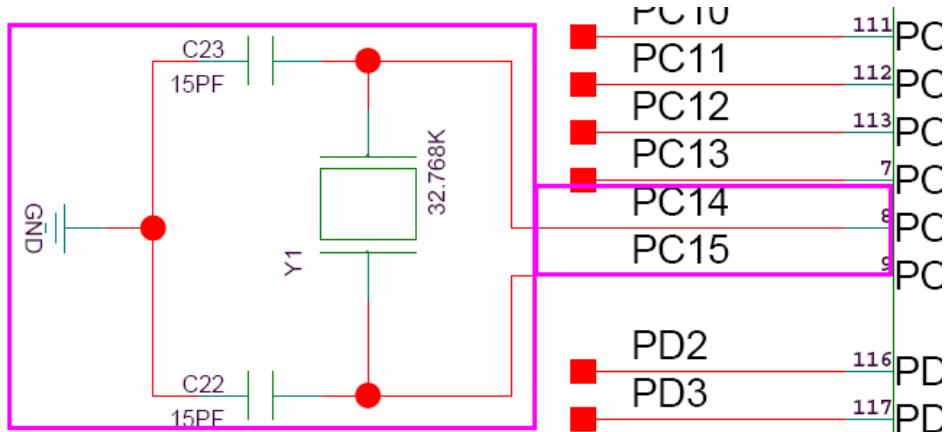
RTC 模块和时钟配置系统(RCC_BDCR 寄存器)处于后备区域，即在系统复位或从待机模式唤醒后，RTC 的设置和时间维持不变。系统复位后，对后备寄存器和 RTC 的访问被禁止，这是为了防止对后备区域(BKP)的意外写操作。执行以下操作将使能对后备寄存器和 RTC 的访问：

- 设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPREN 位，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。

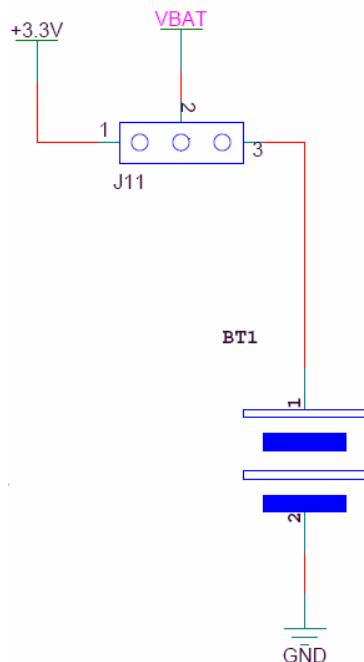
7.18.3 硬件设计

神舟系列开发板的RTC的硬件设计非常简单，其主要硬件都集成在了处理器内部，外围电路主要需要一个32.768KHz的晶振和VBAT供电电池即可。

STM32F103内部已经包含了40kHz低速内部RC振荡电路LSE，但是其精准度不是很高，为此我们在外部增加了32.768KHz的晶振电路驱动RTC实时时钟。



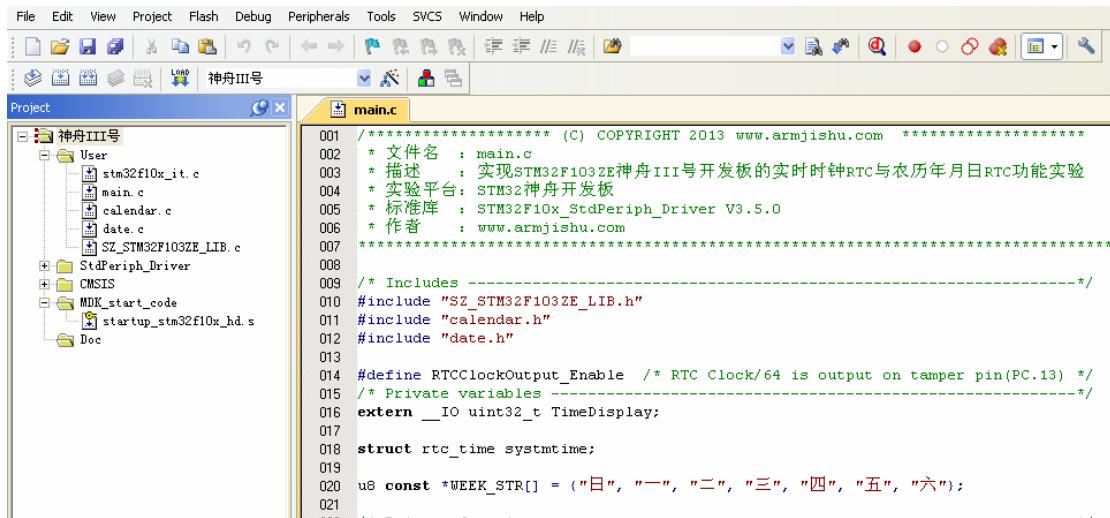
STM32的VBAT采用CR1220纽扣电池和VCC3.3混合供电的方式，在有外部电源（VCC3.3）的时候，BT1不给处理器的VBAT供电，而在外部电源断开的时候，则由BT1给VBAT供电。这样，VBAT总是有电的，以保证RTC的持续运行以及后备寄存器的内容不丢失。相关电路如下：



当安装了电池后，VBAT管脚由+3.3V系统电源供电。

7.18.4 软件设计

进入例程的文件夹，然后打开Project\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    /*!< 在系统启动文件(startup_stm32f1
     所以main函数不需要再次重复初始
     SystemInit()函数的实现位于syst
     */

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    /* 初始化系统定时器SysTick,每秒中断
    SZ_STM32_SysTickInit(1000);

    /* 配置NVIC中断优先级分组函数 */
    NVIC_GroupConfig();

    /* 注意串口2使用Printf时"SZ_STM32F1
    /* 串口2初始化 */
    SZ_STM32_COMInit(COM2, 115200);

    RTC_NVIC_Configuration();
}
```

从main函数开始分析，其中涉及的LED指示灯和串口等相关子程序如果之前的章节已有介绍，则此处不再讲述。同时本示例涉及的程序我们添加了详细的注释。

代码分析 1：SZ_STM32_LEDInit()函数初始化 LED 灯，前面我们已经讲解，我们就不再重复。

代码分析 2：SZ_STM32_SysTickInit()函数初始化系统定时器 SysTick，前面我们已经讲解，我们就不在重述。

代码分析 3：SZ_STM32_COMInit()函数初始化串口 2，本实验通过串口 2 打印 Calendar 时钟。串

口配置不是本示例的重点。前面我们已经讲解，我们就不再重复。

代码分析 4: NVIC_GroupConfig()为配置中断优先级分组函, RTC_NVIC_Configuration()函数是对 RTC 实时时钟的中断参数配置。

代码分析 5: 以下 if...else...判断系统时间是否已经设置，并进行对应的处理。

```
if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
{
    /* Backup data register value is not correct
    the first time the program is executed */
    printf("\r\n\n RTC not yet configured....");

    /* RTC Configuration */
    RTC_Configuration();

    printf("\r\n RTC configured....");

    /* Adjust time by values entred by the user */
    Time_Adjust();
    /* 修改后备寄存器1的值为0XA5A5 */
    BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
}
else
{
```

这里面通过 BKP_ReadBackupRegister(BKP_DR1) 函数读取 RTC 后备寄存器 1 (BKP_DR1) 的值，判断是否事先写入的 0XA5A5，如果不是，则说明 RTC 是第一次上电，需要配置 RTC，提示用户通过串口更改系统时间，通过函数 Time_Adjust() 将实际时间转化为 RTC 计数值写入 RTC 寄存器，然后调用 BKP_WriteBackupRegister () 函数修改后备寄存器 1(BKP_DR1) 的值为 0XA5A5。

else 表示已经设置了系统时间，打印上次系统复位的原因，并使能 RTC 秒中断。

代码分析 6: 上面 if...else...判断系统中，else 部分打印系统复位的原因。我们再一次进入这个判断之前，要清除上次复位原因的记录。否则再一次进入判断，进入 else 产生的复位的原因不一样的话，它串口打印的复位原因是不变的。

```
/* Clear reset flags */
RCC_ClearFlag();
```

代码分析 7: while 循环显示当前系统时钟。

```

while (1)
{
    /* 秒中断函数中会将TimeDisplay置为1 */
    /* 判断全局变量TimeDisplay是否为1，如果为1则时间才输出 */
    TimeDisplay变量在"stm32f10x_it.c"文件的"void RTC_IRQHandler(void)"函数中当发生秒中断时为置为1，这样便实现了精准的每秒输出一次
    if (TimeDisplay == 1) /* If 1s has passed */
    {
        /* Display current time */
        Time_Display(RTC_GetCounter());
        TimeDisplay = 0;

        /* LED1指示灯状态取反 */
        SZ_STM32_LEDToggle(LED1);
    }
    /* 此处可以添加用户的程序 */
}

```

在本次循环中，调用 Time_Display() 函数显示当前的时钟，每秒打印一次系统时间。全局变量 TimeDisplay 是否为 1，如果为 1 则打印时间并将清 TimeDisplay 为 0 以便它在"SZ_STM32F1_LIB.c"文件的"void RTC_IRQHandler(void)"函数中当发生秒中断时再次置为 1，这样便实现了精准的每秒输出时间一次。我们可以发现 Time_Display() 函数调用了 RTC_GetCounter() 函数读取 RTC 的 Counter 寄存器的值，其实现如下：

```

/***
 * @brief Gets the RTC counter value.
 * @param None
 * @retval RTC counter value.
 */
uint32_t RTC_GetCounter(void)
{
    uint16_t tmp = 0;
    tmp = RTC->CNTL;
    return (((uint32_t)RTC->CNTH << 16) | tmp);
}

```

注意：此实验的 Time_Display() 函数与上一个实验有所不同。此处的 Time_Display() 函数使用静态局部变量 FirstDisplay 保证在第一次调用该函数时计算并打印当前时间对应的农历年月日与农历节气，并且将 RTC_GetCounter() 读到 RTC 的 Counter 寄存器的值转换为年月日时分秒的时间信息并打印，其实现如下：

```

void Time_Display(uint32_t TimeVar)
{
    static uint32_t FirstDisplay = 1;
    u8 str[15]; // 字符串暂存

    to_tm(TimeVar, &systmtime);

    if ((!systmtime.tm_hour && !systmtime.tm_min && !systmtime.tm_sec) || (FirstDisplay))
    {
        /* 神舟系列开发板 计算农历 */
        GetChinaCalendar((u16)systmtime.tm_year, (u8)systmtime.tm_mon, (u8)systmtime.tm_mday, str);
        printf("\n\r\n\r 今天农历: %0.2d%0.2d,%0.2d,%0.2d", str[0], str[1], str[2], str[3]);
        GetChinaCalendarStr((u16)systmtime.tm_year, (u8)systmtime.tm_mon, (u8)systmtime.tm_mday, str);
        printf(" %s", str);
        if (GetJieqiStr((u16)systmtime.tm_year, (u8)systmtime.tm_mon, (u8)systmtime.tm_mday, str))
            printf(" %s\n\r", str); /* 神舟系列开发板 计算农历节气 */
    }
    FirstDisplay = 0;
}

```

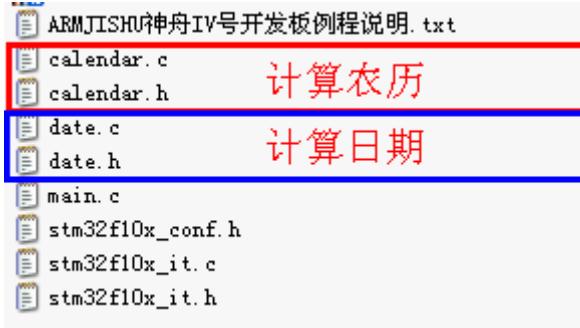
此处的 Time_Display() 函数调用了计算农历日期与节气的三个函数，这三个函数位于 calendar.c 中，在 calendar.h 中的声明如下图所示：

```

u8 GetChinaCalendar(u16 year, u8 month, u8 day, u8 *p);
void GetChinaCalendarStr(u16 year, u8 month, u8 day, u8 *str);
u8 GetJieQi(u16 year, u8 month, u8 day, u8 *JQdate);
u8 GetJieQistr(u16 year, u8 month, u8 day, u8 *str);

```

本例程与上一章节的例程在程序文件上多了以下四个文件：



其中“date.c”实现了时间信息与计数值之间的相互转换功能。

其中“calendar.c”实现了超强的日历功能，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。

到此，我们主函数部分框架流程基本完成，下面我们对一些比较重要的函数进行分析。

代码分析8：我们来看一下有关中断函数。NVIC_GroupConfig()和，RTC_NVIC_Configuration()。

```

void NVIC_GroupConfig(void)
{
    /* 配置NVIC中断优先级分组：
     - 1比特表示主优先级 主优先级合法取值为 0 或 1
     - 3比特表示次优先级 次优先级合法取值为 0..7
     - 数值越低优先级越高，取值超过合法范围时取低bit位
    */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
}

```

NVIC_Configuration 函数实现配置嵌套向量中断中断优先级并使能中断。其中的 NVIC_PriorityGroupConfig 函数配置中断优先级的组织方式，STM32 的嵌套向量中断控制器可以配置 16 个可编程的优先等级，使用了 4 位表示中断优先级（2 的 4 次方就是 16），16 个可编程的优先等级又可以分为主优先级和次优先级，例如参数 NVIC_PriorityGroup_1 表示 1bit 主优先级（pre-emption priority）3 bits 次优先级（subpriority）。

RTC_NVIC_Configuration()函数使能RTC中断。

```

void RTC_NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Configure one bit for preemption priority */
    //NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

    /* Enable the RTC Interrupt */
    NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

以下为中断处理函数，请看代码注释。注意其中的对全局变量TimeDisplay的置一与Time_Show()函数中的清零操作相呼应，是实现精确1秒延时的关键。

```
void RTC_IRQHandler(void)
{
    /* 判断RTC是否发生了秒中断（也有可能是溢出或者闹钟中断） */
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
    {
        /* Clear the RTC Second interrupt */
        /* 清除秒中断标志 */
        RTC_ClearITPendingBit(RTC_IT_SEC);

        /* Toggle LED1 改变LED1的亮灭状态(取反) */
        SZ_STM32_LEDToggle(LED1);

        /* Enable time update */
        /* 置全局变量为1，通知主程序打印时间 */
        TimeDisplay = 1;
    }
}
```

RTC配置中有关中断相关函数解释如下：

```
void RTC_Configuration(void)
{
    /* Enable PWR and BKP clocks */
    /* PWR时钟（电源控制）与BKP时钟（RTC后备寄存器）使能 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

    /* Allow access to BKP Domain */
    /* 使能RTC和后备寄存器访问 */
    PWR_BackupAccessCmd(ENABLE);

    /* Reset Backup Domain */
    /* 将外设BKP的全部寄存器重设为缺省值 */
    BKP_DeInit();

    /* Enable LSE */
    /* 使能LSE（外部32.768KHz低速晶振） */
    RCC_LSEConfig(RCC_LSE_ON);

    /* Wait till LSE is ready */
    /* 等待外部晶振震荡稳定输出 */
    while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
    {}

    /* Select LSE as RTC Clock Source */
    /* 使用外部32.768KHz晶振作为RTC时钟 */
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

    /* Enable RTC Clock */
    /* 使能 RTC 的时钟供给 */
    RCC_RTCCLKCmd(ENABLE);

    /* Wait for RTC registers synchronization */
    /* 等待RTC寄存器同步 */
    RTC_WaitForSynchro();

    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();

    /* Enable the RTC Second */
    /* 使能RTC的秒中断 */
    RTC_ITConfig(RTC_IT_SEC, ENABLE);

    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();

    /* Set RTC prescaler: set RTC period to 1sec */
    /* 32.768KHz晶振预分频值是32767,如果对精度要求很高可以修改此分频值来校准晶振 */
    RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */
}
```

上述多个程序中都调用了以下等待的子函数调用：

```
/* Wait until last write operation on RTC registers has finished */
RTC_WaitForLastTask();
```

为什么要不停的等待呢？

RTC核完全独立于RTC APB1接口。 软件通过APB1接口访问RTC的预分频值、计数器值和闹钟值。但是，相关的可读寄存器只在与RTC APB1时钟进行重新同步的RTC时钟的上升沿被更新。RTC标志也是如此的。这意味着，如果APB1接口曾经被关闭，而读操作又是在刚刚重新开启APB1之后，则在第一次的内部寄存器更新之前，从APB1上读出的RTC寄存器数值可能被破坏了(通常读到0)。因此，若在读取RTC寄存器时，RTC的APB1接口曾经处于禁止状态，则软件首先必须等待RTC_CRL寄存器中的RSF位(寄存器同步标志)被硬件置‘1’。

(注： RTC的 APB1接口不受WFI和WFE等低功耗模式的影响。)

代码分析9：实验操作中，我们可以输入时间信息。最后我们看一下，时间调整有关的函数。

```
void Time_Adjust(void)
{
    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();
    /* Get time entered by the user on the hyperterminal */
    Time_Regulate(&systmtime);
    /* Get wday */
    GregorianDay(&systmtime);
    /* Change the current time */
    /* 把时间转化为RTC计数值写入RTC寄存器 */
    RTC_SetCounter(mktimev(&systmtime));
    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();
}
```

Time_Adjust函数实现时间调整相关功能，首先调用Time_Regulate(&systmtime)函数请求用户输入年月日时分秒等信息，然后使用mktimev函数将用户输入的年月日时分秒等信息转换为计数器的计数值并对RTC计数器赋值保存。

Time_Regulate函数多次调用用户输入函数，要求用户输入年月日时分秒等信息：

```
void Time_Regulate(struct rtc_time *tm)
{
    uint32_t DataIn = 0xFF;

    printf("\r\n=====www.armjishu.com=====

    printf("\r\n 请输入年份(Please Set Years):  20");
    while(DataIn == 0xFF)
    {
        DataIn = USART_Scanf(99);
    }
    printf("\r\n  年份被设置为:  20%0.2d\r", DataIn);
    tm->tm_year = DataIn+2000;

    DataIn = 0xFF;
    printf("\r\n 请输入月份(Please Set Months):  ");
```

7.18.5 下载与验证

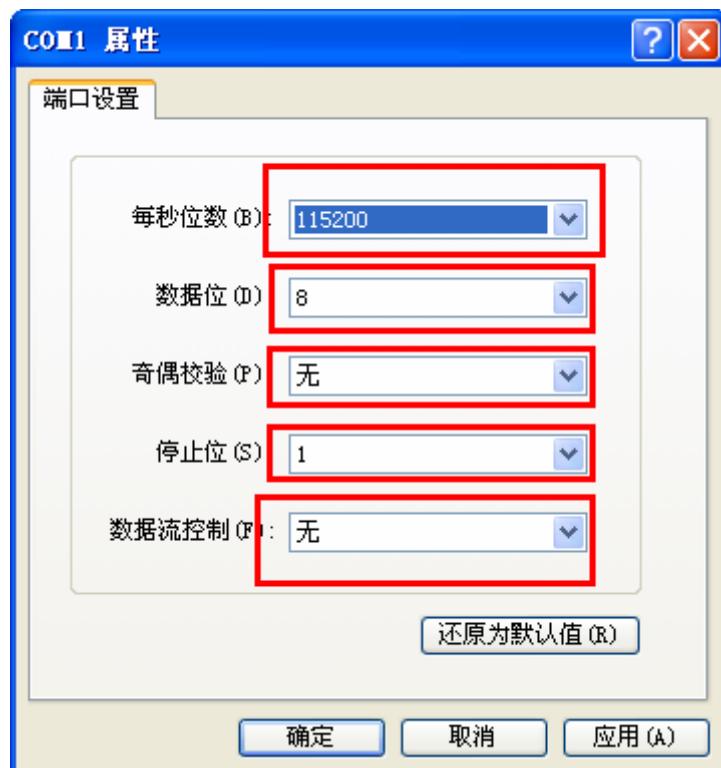
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.18.6 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 2 与电脑连接，并打开超级终端，按以下设置，如下图：



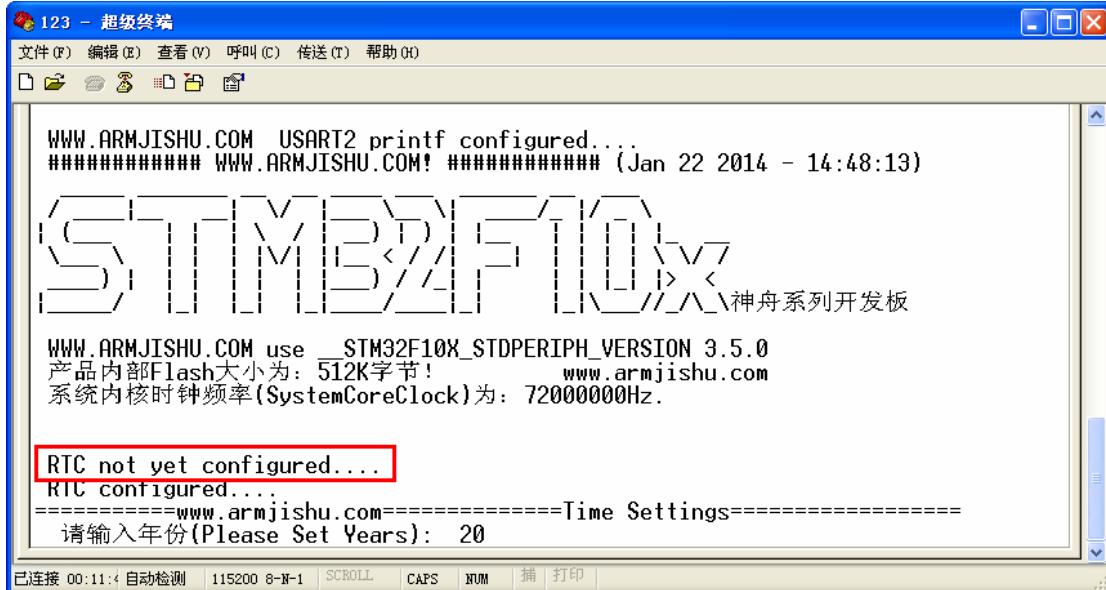
选择 COM1；按确定



再按确定，完成超级终端设置。

重新打开电源；神舟 II 号 STM32 开发板上 LED 2 和 LED 3 常亮，LED 1 和 LED 4 间断亮灭。

超级终端窗口显示信息，如果是第一次运行该程序，则串口会输出提示“RTC not yet configured....”然后会要求用户输入公历的年月日和时分秒信息如下图所示，如下图



提示输入相关的时间信息，

请输入年份： 15

请输入月份： 10

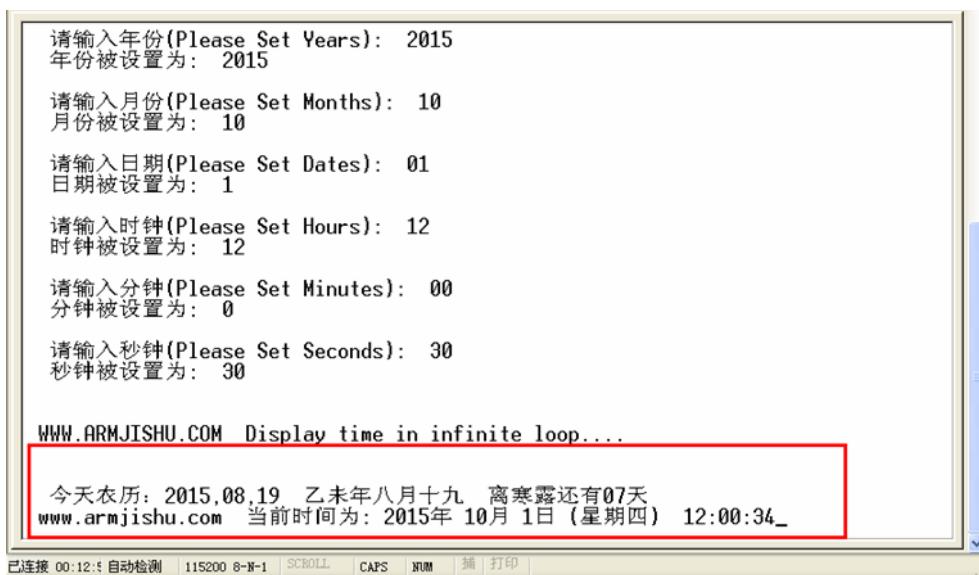
请输入日期： 01

请输入时钟： 12

请输入分钟： 00

请输入秒钟： 30

完成以上相关信息输入后，超级终端窗口显示，以下图：



7.19 EEPROM读写测试实验

7.19.1 意义与作用

EEPROM是一种电可擦可编程只读存储器，掉电后数据不丢失。是单片机应用系统中经常会用到的存储器。EEPROM掉电后数据不会丢失，而且可以用电信号直接清除存储数据和再编程，正是由于它的这一特性，EEPROM在嵌入式设备中应用广泛，用于产品出厂数据的保存，产品运行过程中一些数据量不大的重要数据保存等。

在本章节，我们以最常见的I2C接口的24CXX芯片为例进行学习研究。它采用PHILIPS公司开发的两线式串行总线(I2C总线)。通过本章节实验，我们将对I2C总线有一个深入的了解，进而掌握如何读写访问24CXX这一系列的I2C接口EEPROM。

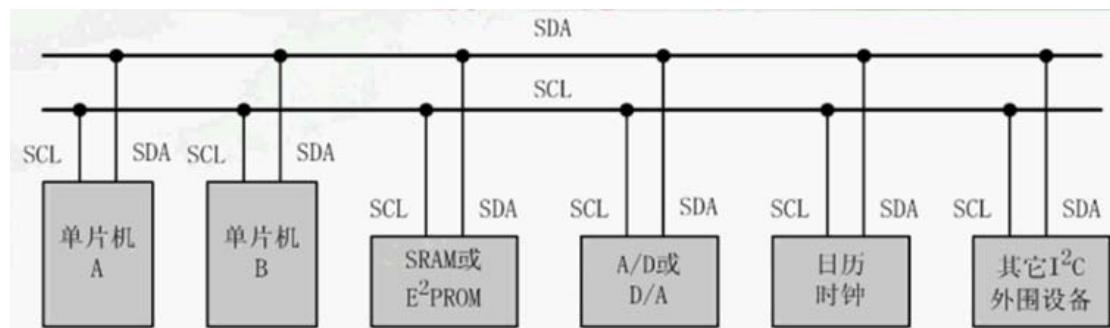
7.19.2 I2C的介绍

I2C总线的定义

I2C(Inter—Integrated Circuit)总线是一种由PHILIPS公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线SDA和时钟SCL构成的串行总线，可发送和接收数据，读写访问简单。在CPU与被控IC之间、IC与IC之间进行双向传送，高速I2C总线一般可达400kbps以上。

I2C总线特点

I2C总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此I2C总线占用的空间非常小，减少了电路板的空间和芯片管脚的数量，降低了互联成本。总线的长度可高达25英尺，并且能够以10Kbps的最大传输速率支持40个组件。I2C总线的另一个优点是，它支持多主控(multimastering)，其中任何能够进行发送和接收的设备都可以成为总线。一个主控能够控制信号的传输和时钟频率。当然，在任何时间点上只能有一个主控。



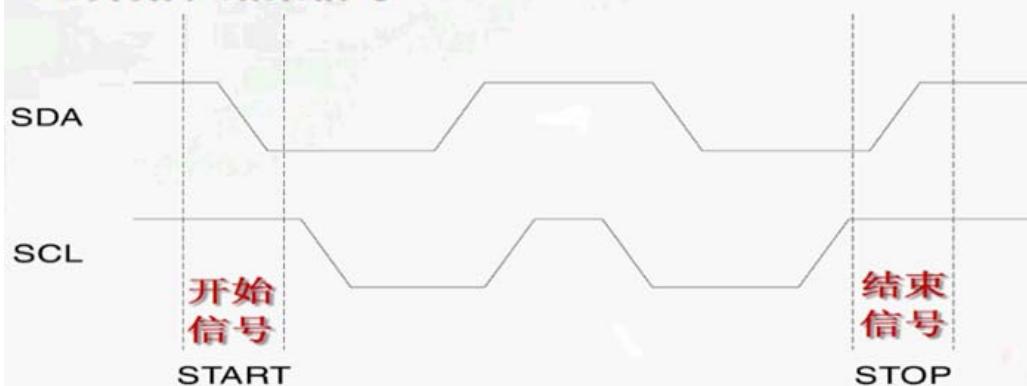
I2C总线工作状态

I2C总线在传送数据过程中共有三种特殊类型信号，它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL为高电平期间，SDA由高电平向低电平跳变，开始传送数据。

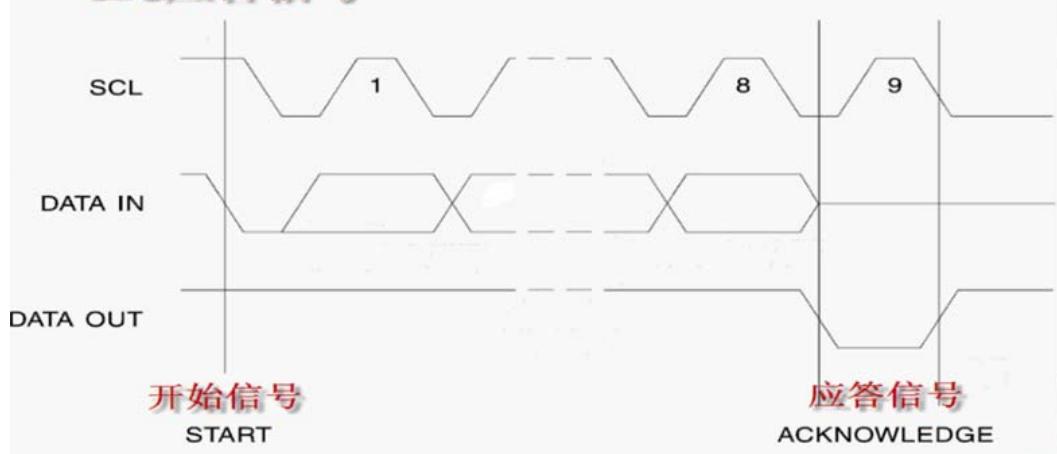
结束信号：SCL为高电平期间，SDA由低电平向高电平跳变，结束传送数据。

I2C开始和结束信号

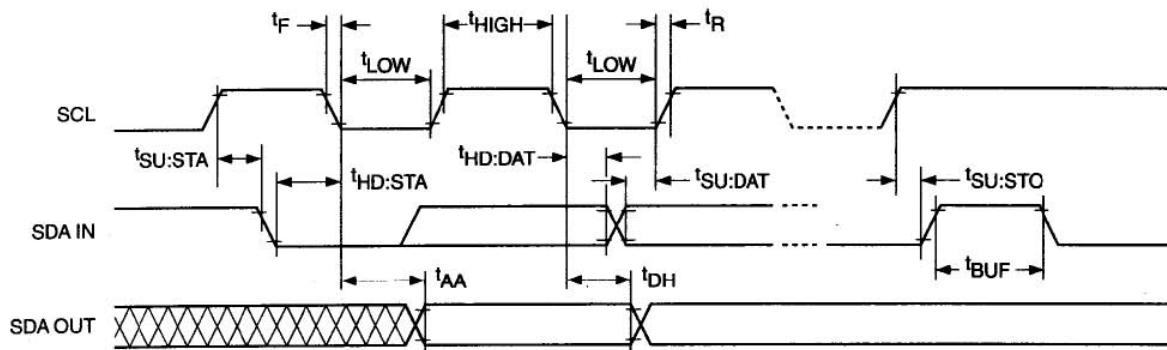


应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

I2C应答信号



I2C 总线三种特殊类型信号时序图为：



7.19.3 实验原理

神舟III号开发板板载的EEPROM芯片型号为24C02，该芯片的总容量是256个字节。
本节实验的基本原理：神舟III号通过STM32F103ZET6处理器本身自带的硬件I2C接口与24C02相连，我们首先往EEPROM中写入一连串的有规律的数据，然后顺序读出，通过串口打印读出的数据，判断读出的数据是否正确，从而得知EEPROM是否可以正常访问。

7.19.4 硬件设计

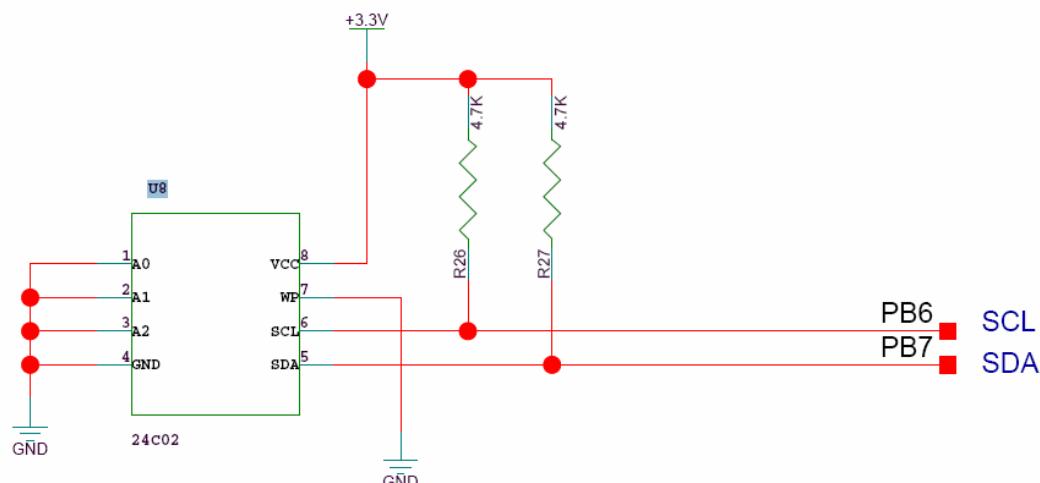
本实验需要用到的硬件资源：

- 串口 2：串口的输入输出实验在前面已经进行了详细的讲解，在这里就不在重复。具体串口输入输出实验。
- I2C EEPROM 24C02

STM32F103ZET6处理器具有两个I2C接口，I2C接口与管脚对应关系如下表所示。

| I2C接口 | 管脚名 | 对应GPIO | 功能描述 |
|-------|----------|--------|-----------|
| I2C1 | I2C1_SCL | PB6 | I2C1接口的时钟 |
| | I2C1_SDA | PB7 | I2C1接口的数据 |
| I2C2 | I2C2_SCL | PB10 | I2C2接口的时钟 |
| | I2C2_SDA | PB11 | I2C2接口的数据 |

神舟III号自带了24C02的EEPROM芯片，该芯片的容量为2Kbit，也就是256个字节，对于我们普通应用来说是足够的了。你也可以选择换大的芯片，因为在原理上是兼容24C02~24C512全系列的EEPROM芯片的。神舟III号通过处理器自带硬件I2C1接口与EEPROM 24C02连接，电路图如下：



EEPROM 原理图

AT24C02是美国ATMEL公司的低功耗CMOS串行EEPROM，它是内含 256×8 位存储空间，具有工作电压宽（2.5~5.5V）、擦写次数多（大于10000次）、写入速度快（小于10ms）等特点。

AT24C02的1、2、3脚是三条地址线，用于确定芯片的硬件地址。如上图中，A0~A2全部接地，对应I2C的硬件地址为0xA0。

第4脚为电源输入端，第8脚为GND管脚。

第5脚SDA为串行数据输入/输出，数据通过这条双向I2C总线串行传送，在神舟III号中，与处理器的PB7管脚连接。

第6脚SCL为串行时钟输入线，在神舟III号中，与处理器的PB6管脚连接。

SDA和SCL都需要和正电源间各接一个5.1K的电阻上拉。

第7脚为AT24C02的写保护端，如果需要禁止对AT24C02进行读写，需要上拉到电源。在神舟III号中，为了方便随时对AT24C02进行访问操作，将WP管脚接地，禁止AT24C02的写保护功能。

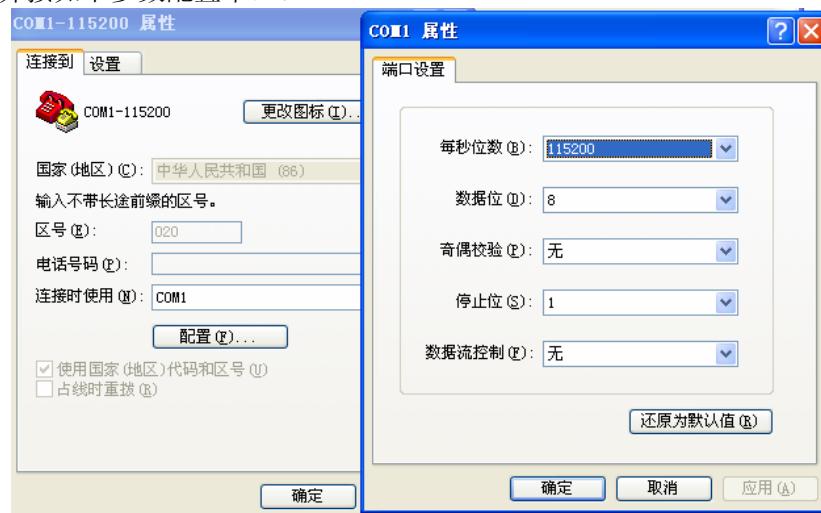
24C02的特性如下：

- 存储器组织结构
 - 24C02, 256 X 8 (2K bits)
 - 24C04, 512 X 8 (4K bits)
 - 24C08, 1024 X 8 (8K bits)
 - 24C16, 2048 X 8 (16K bits)
 - 24C32, 4096 X 8 (32K bits)
 - 24C64, 8192 X 8 (64K bits)
- 2线串行接口, 完全兼容I²C总线
- I²C时钟频率为1 MHz (5V), 400 kHz (1.8V, 2.5V, 2.7V)
- 施密特触发输入噪声抑制
- 硬件数据写保护
- 内部写周期 (最大5 ms)
- 可按字节写
- 页写: 8字节页 (24C02), 16字节页(24C04/08/16), 32字节页(24C32/64)
- 可按字节, 随机和序列读
- 自动递增地址

7.19.5 软件设计

神舟 III 号 EEPROM 读写试验位于神舟 III 号光盘\源码\ EEPROM 读写测试实验\Project 目录

把代码下载到神舟 III 号板子上后, 用随板配置的串口线连接神舟 III 号串口 2 与电脑的串口, 打开超级终端, 并按如下参数配置串口。



超级终端串口参数配置

上电运行神舟 III 号, 串口将打印信息, 如下图所示:

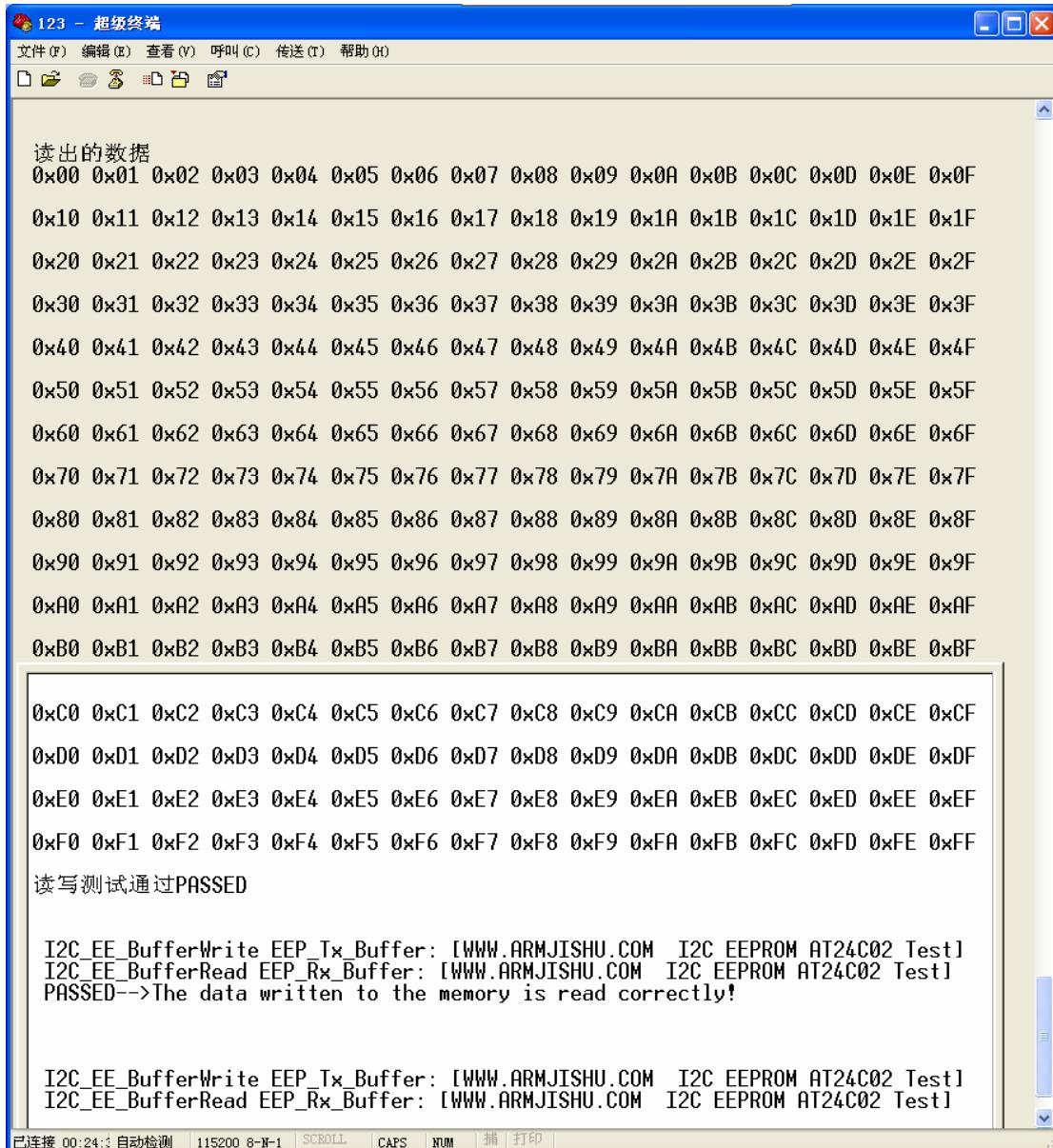
123 - 超级终端

WWW.ARmjishu.COM USART2 printf configured.
WWW.ARmjishu.COM! ##### (Jan 22 2014 - 13:23:51)

STM32F10X_STDPERIPH_VERSION 3.5.0
产品内部Flash大小为: 512K字节! www.armjishu.com
系统内核时钟频率(SystemCoreClock)为: 72000000Hz.

I2C_Configuration----
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 00:23:: 自动检测 | 115200 8-N-1 | SCROLL | CAPS | NUM | 插 | 打印 |



实验现象：往 24C02 写入数据，然后读出 24C02 里面的数据，写入写出 2 个数据进行比较，相同的话测试通过，反之失败。

初步根据实验现象查看代码（大体的思路）

以下为工程文件中主要代码的解释与说明。

1)、地址数据的定义

```
/* Private variables ----- */
USART_InitTypeDef USART_InitStructure;

u8 EEP_Tx_Buffer[] = "WWW.ARMJISHU.COM I2C EEPROM AT24C02 Test"; /*输入数据*/
#define EEPTx_BUFSIZE      sizeof(EEP_Tx_Buffer) / sizeof(EEP_Tx_Buffer[0])
#define EEP_Firstpage      0x00
#define EEP_Randompage     0x06
u8 EEP_Rx_Buffer[EEPTx_BUFSIZE];

TestStatus TransferStatus;
```

2)、初始化

```
/* 初始化板载LED指示灯 */
SZ_STM32_LEDInit(LED1);
SZ_STM32_LEDInit(LED2);
SZ_STM32_LEDInit(LED3);
SZ_STM32_LEDInit(LED4);

/* 注意串口2使用Printf时"sz_STM32F103ZE_LIB.c"文件中fputc定义中设备改为sz_STM32_COM2 */
/* 串口2初始化 */
SZ_STM32_COMInit(COM2, 115200);

I2C_EE_Init();

I2C_Test();
```

3)、写入与读出数据的对比与结果

比较 1:

```
/* 如果上述测试已经通过则下面的测试可以不需要，以下是另一种方式的测试 */
// Write on I2C EEPROM from EEPROM_WriteAddress0
printf("\r\n\n I2C_EE_BufferWrite EEP_Tx_Buffer: [%s]", EEP_Tx_Buffer);
I2C_EE_BufferWrite(EEP_Tx_Buffer, EEP_Firstpage, EEPTx_BUFSIZE-1);
// Read from I2C EEPROM from EEPROM_ReadAddress0
I2C_EE_BufferRead(EEP_Rx_Buffer, EEP_Firstpage, EEPTx_BUFSIZE-1);
printf("\r\n I2C_EE_BufferRead EEP_Rx_Buffer: [%s]", EEP_Rx_Buffer);
// Check if the data written to the memory is read correctly
TransferStatus = Buffercmp(EEP_Tx_Buffer, EEP_Rx_Buffer, EEPTx_BUFSIZE);
if(FAILED == TransferStatus)
{
    /*失败—>数据写入内存读取错误!*/
    printf("\r\n FAILED-->The data written to the memory is read Error!\r\n");
}
else
{
    /*通过了—>数据写入内存读取正确!*/
    printf("\r\n PASSED-->The data written to the memory is read correctly! \r\n");
}
printf("\r\n ");
```

比较 2:

```

//从 I2C_eepm 写在 EEPROM_WriteAddress2
printf("\r\n\n I2C_EE_BufferWrite EEP_Tx_Buffer: [%s]", EEP_Tx_Buffer);
I2C_EE_BufferWrite(EEP_Tx_Buffer, EEP_Randompage, EEPTx_BUFSIZE-1);
// 从 EEPROM_ReadAddress2 读取 I2C_eepm
I2C_EE_BufferRead(EEP_Rx_Buffer, EEP_Randompage, EEPTx_BUFSIZE-1);
printf("\r\n I2C_EE_BufferRead EEP_Rx_Buffer: [%s]", EEP_Rx_Buffer);
// 检查数据写入内存读取正确
TransferStatus &= Buffercmp(EEP_Tx_Buffer, EEP_Rx_Buffer, EEPTx_BUFSIZE);

if(FAILED == TransferStatus)
{
    /*失败-->数据写入Randompage内存读取错误!*/
    printf("\r\n FAILED-->The data written to the Randompage memory is read Error!\n\r");
}
else
{
    /*通过了-->数据写入Randompage内存读取正确!*/
    printf("\r\n PASSED-->The data written to the Randompage memory is read correctly! \n\r");
}

printf("\r\n\n I2C_EEPROM TEST Finished!!!");

```

最后是 LED 状态

```

while (1)
{
    /* LED1 指示灯状态取反 */
    SZ_STM32_LEDToggle(LED1);

    /* 延迟，闪烁间隔 */
    delay(16000000);

    /* 此处可以添加用户的程序 */
}
}

```

以上为主函数的整体框架

进一步分析程序代码

1)、初始化部分

LED 初始化

```

/* 初始化板载 LED 指示灯 */
SZ_STM32_LEDInit(LED1);
SZ_STM32_LEDInit(LED2);
SZ_STM32_LEDInit(LED3);
SZ_STM32_LEDInit(LED4);

```

神舟III号I2C1接口初始化。

```

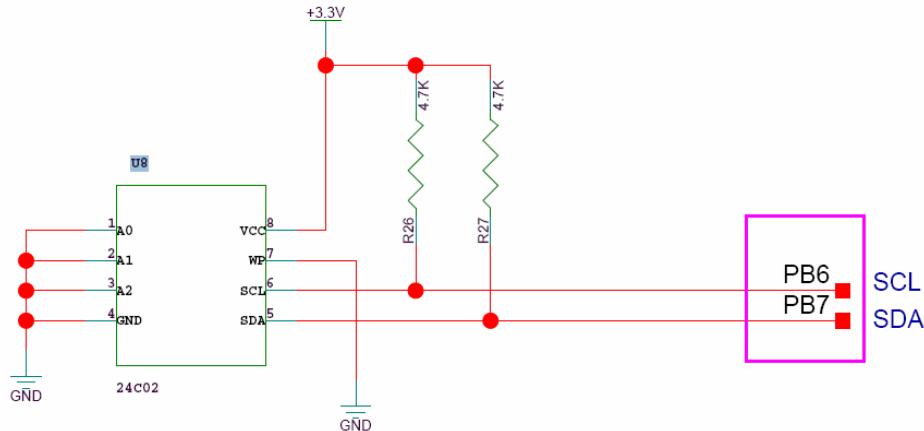
/* 注意串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为SZ_STM32_COM2 */
/* 串口2初始化 */
SZ_STM32_COMInit(COM2, 115200);

I2C_EE_Init();

I2C_Test();

```

我们首先将I2C1接口初始化为复用功能开漏输出，从原理图上可以看到，我们的24C02的SCL与SDA接的是PB6和PB7管脚，所以我们要对它们进行一个配置。



```

void GPIO_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    /* Configure I2C1 pins: SCL and SDA */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
}

```

然后，对 I2C1 接口进行初始化，具体为设置 I2C 接口的模式（由于 I2C 接口可工作在 I2C 模式或者 SMBus 模式，因此在初始化时，需要设置其具体工作模式），I2C 主机地址，以及 I2C 速率等等。

```

*****
void I2C_Configuration(void)
{
    I2C_InitTypeDef I2C_InitStructure;

    /* I2C configuration */
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStructure.I2C_OwnAddress1 = I2C1_SLAVE_ADDRESS7;
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = I2C_Speed;

    /* I2C Peripheral Enable */
    I2C_Cmd(I2C1, ENABLE);
    /* Apply I2C configuration after enabling it */
    I2C_Init(I2C1, &I2C_InitStructure);

    /* 允许1字节1应答模式 */
    I2C_AcknowledgeConfig(I2C1, ENABLE);

    printf("\n\r I2C_Configuration----\n\r");
}

```

在完成了I2C接口初始化以后，我们就可以正常的使用I2C1接口了，下面是神舟III号EEPROM读写访问程序。

```
/* 串口2初始化 */
SZ_STM32_COMInit(COM2, 115200);

I2C_EE_Init();

I2C_Test(); //如果上述测试已经通过则下面的测试可以不需要，以下是另一种方式的测试
// Write on I2C EEPROM from EEPROM_WriteAddress0
// Read on I2C EEPROM to EEPROM_ReadAddress0

void I2C_Test(void)
{
    u16 i;
    u8 I2c_Buf_Write[256];
    u8 I2c_Buf_Read[256];
    ——
    printf("写入的数据\n\r");

    for(i=0;i<=255;i++) //填充缓冲
    {
        I2c_Buf_Write[i]=i;
        printf("0x%02X ",I2c_Buf_Write[i]);
        if(i%16 == 15)
        {
            printf("\n\r");
        }
    }

    //将I2c_Buf_Write中顺序递增的数据写入EEPROM中
    I2C_EE_BufferWrite(I2c_Buf_Write,EEP_Firstpage,256);

    printf("\n\r读出的数据\n\r");
    //将EEPROM读出数据顺序保持到I2c_Buf_Read中
    I2C_EE_BufferRead(I2c_Buf_Read,EEP_Firstpage,256);

    //将I2c_Buf_Read中的数据通过串口打印
    for(i=0;i<256;i++)
    {
        if(I2c_Buf_Read[i]!=I2c_Buf_Write[i])
        {
            printf("0x%02X ", I2c_Buf_Read[i]);
            printf("错误:I2C EEPROM写入与读出的数据不一致\n\r");
            return;
        }
        printf("0x%02X ", I2c_Buf_Read[i]);
        if(i%16 == 15)
        {
            printf("\n\r");
        }
    }
    printf("读写测试通过PASSED\n\r");
}
```

这段代码主要是将0x00~0xFF顺序写入到EEPROM中，然后再依次从EEPROM中读出这些数据，并通过串口打印。

24C02中带有片内地址寄存器。每写入或读出一个数据字节后，该地址寄存器自动加1，以实现对下一个存储单元的读写。所有字节均以单一操作方式读取。为降低总的写入时间，一次操作可写入多达8个字节的数据。

其中I2C_WriteS_24C函数，是24CXX EEPROM的页写入函数，在神舟III号，我们使用的24C02的页大小为8，也就是说，可以单次连续8个字节写入。具体的函数实现如下：

```
//将I2c_Buf_Write中顺序递增的数据写入EEPROM中
I2C_EE_BufferWrite(I2c_Buf_Write,EEP_Firstpage,256);

printf("\n\r读出的数据\n\r");
//将EEPROM读出数据顺序保持到I2c_Buf_Read中
I2C_EE_BufferRead(I2c_Buf_Read,EEP_Firstpage,256);

void I2C_EE_BufferWrite(u8* pBuffer, u8 WriteAddr, u16 NumByteToWrite)
{
    u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0;

    Addr = WriteAddr % I2C_PageSize; //0x00%8=0
    count = I2C_PageSize - Addr; //8-0=8
    NumOfPage = NumByteToWrite / I2C_PageSize; //256/8 =32
    NumOfSingle = NumByteToWrite % I2C_PageSize; //256%8 =0

    /* If WriteAddr is I2C_PageSize aligned */
    if(Addr == 0)
    {
        /* If NumByteToWrite < I2C_PageSize */
        if(NumOfPage == 0)
        {
            I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
            I2C_EE_WaitEepromStandbyState();
        }
        /* If NumByteToWrite > I2C_PageSize */
        else
        {
            while(NumOfPage--) //一次写8个数据
            {
                I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
                I2C_EE_WaitEepromStandbyState();
                WriteAddr += I2C_PageSize;
                pBuffer += I2C_PageSize;
            }

            if(NumOfSingle!=0) //除8个以外的
            {
                I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
                I2C_EE_WaitEepromStandbyState();
            }
        }
    }
    /* If WriteAddr is not I2C_PageSize aligned */
    else
    {
        /* If NumByteToWrite < I2C_PageSize */
        if(NumByteToWrite < I2C_PageSize)
        {
            I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
            I2C_EE_WaitEepromStandbyState();
        }
        /* If NumByteToWrite > I2C_PageSize */
        else
        {
            while(NumOfPage--) //一次写8个数据
            {
                I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
                I2C_EE_WaitEepromStandbyState();
                WriteAddr += I2C_PageSize;
                pBuffer += I2C_PageSize;
            }

            if(NumOfSingle!=0) //除8个以外的
            {
                I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
                I2C_EE_WaitEepromStandbyState();
            }
        }
    }
}
```

```
if(NumOfPage== 0)
{
    I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
    I2C_EE_WaitEepromStandbyState();
}
/* If NumByteToWrite > I2C.PageSize */
else
{
    NumByteToWrite == count;
    NumOfPage = NumByteToWrite / I2C.PageSize;
    NumOfSingle = NumByteToWrite % I2C.PageSize;

    if(count != 0)
    {
        I2C_EE_PageWrite(pBuffer, WriteAddr, count);
        I2C_EE_WaitEepromStandbyState();
        WriteAddr += count;
        pBuffer += count;
    }

    while(NumOfPage--)
    {
        I2C_EE_PageWrite(pBuffer, WriteAddr, I2C.PageSize);
        I2C_EE_WaitEepromStandbyState();
        WriteAddr += I2C.PageSize;
        pBuffer += I2C.PageSize;
    }
}
}
```

+

其中I2C_ReadS_24C函数，是24CXX EEPROM的页读出函数，在神舟III号，我们使用的24C02的页大小为8，也就是说，可以单次连续8个字节读出。具体的函数实现如下：

```
//将I2c_Buf_Write中顺序递增的数据写入EEPROM中
I2C_EE_BufferWrite(I2c_Buf_Write,EEP_Firstpage,256);

printf("\n\r读出的数据\n\r");
//将EEPROM读出数据顺序保持到I2c_Buf_Read中
I2C_EE_BufferRead(I2c_Buf_Read,EEP_Firstpage,256);

//将I2c_Buf_Read中的数据通过串口打印
for(i=0;i<256;i++)
{
    //从I2C_Buf_Read中读取一个字节
    //将I2C_Buf_Write中写入一个字节
}
```

```
void I2C_EE_BufferRead(u8* pBuffer, u8 ReadAddr, u16 NumByteToRead)
{
    /*((u8 *)0x4001080c) |=0x80;
    while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)); // Added by Najoua 27/08/2008

    /* Send START condition */
    I2C_GenerateSTART(I2C1, ENABLE);
    /*((u8 *)0x4001080c) &=~0x80;

    /* Test on EV5 and clear it */
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

    /* Send EEPROM address for write */
    I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);

    /* Test on EV6 and clear it */
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

    /* Clear EV6 by setting again the PE bit */
    I2C_Cmd(I2C1, ENABLE);

    /* Send the EEPROM's internal address to write to */
    I2C_SendData(I2C1, ReadAddr);

    /* Test on EV8 and clear it */
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

    /* Send STRAT condition a second time */
    I2C_GenerateSTART(I2C1, ENABLE);

    /* Test on EV5 and clear it */
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

    /* Send EEPROM address for read */
    I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Receiver);

    /* Test on EV6 and clear it */
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));

    /* While there is data to be read */
    while(NumByteToRead)
    {
        if(NumByteToRead == 1)
        {
            /* Disable Acknowledgement */
            I2C_AcknowledgeConfig(I2C1, DISABLE);

            /* Send STOP Condition */
            I2C_GenerateSTOP(I2C1, ENABLE);
        }

        /* Test on EV7 and clear it */
        if(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED))
        {
            /* Read a byte from the EEPROM */
            *pBuffer = I2C_ReceiveData(I2C1);

            /* Point to the next location where the byte read will be saved */
            pBuffer++;

            /* Decrement the read bytes counter */
            NumByteToRead--;
        }
    }

    /* Enable Acknowledgement to be ready for another reception */
    I2C_AcknowledgeConfig(I2C1, ENABLE);
}
```

下面我们根据程序代码来了解下 IIC 主要的特殊类型信号

- 首先我们看下起始信号

```

/* 从这里开始 */
I2C_GenerateSTART(I2C1, ENABLE);

void I2C_GenerateSTART(I2C_TypeDef* I2Cx, FunctionalState NewState)
{
    /* Check the parameters */
    assert_param(IS_I2C_ALL_PERIPH(I2Cx));
    assert_param(IS_FUNCTIONAL_STATE(NewState));
    if (NewState != DISABLE)
    {
        /* Generate a START condition */
        I2Cx->CR1 |= CR1_START_Set;
    }
    else
    {
        /* Disable the START condition generation */
        I2Cx->CR1 &= CR1_START_Reset;
    }
}

```

函数中，只要 NewState 与 DISABLE 不相等的话，在这里我们用到了一个 I2C 中的 CR1 寄存器，只要不相等，就为这个寄存器赋一个值，相等的话也赋一个值，下面我们看下 NewState 与 DISABLE 是怎么样的，而 CR1 的寄存器赋值后呢？下面我们进入 NewState 与 DISABLE 看下

```

typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
#define IS_FUNCTIONAL_STATE(STATE) (((STATE) == DISABLE) || ((STATE) == ENABLE))

```

只要不等于的话，我们给 CR1 赋值 CR1_START_Set，否则赋值 CR1_START_Reset

下面我们分别看下这 2 个值是什么

```

/* I2C START mask */
#define CR1_START_Set ((uint16_t)0x0100)
#define CR1_START_Reset ((uint16_t)0xFEFF)

```

分别是 0x0100 和 0xFEFF，那么我们看下把它们赋值给 CR1 寄存器后会怎么样，打开我们的 STM32F103ZE 的参考手册，找到 CR1 的寄存器

24.6.1 控制寄存器 1(I2C_CR1)

地址偏移: 0x00

复位值: 0x0000

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--|-------|-----|-------|-----|-----|-----|------|-------|------------|-----|-------|-------|----------|-----|-------|----|
| | SWRST | 保留 | ALERT | PEC | POS | ACK | STOP | START | NO STRETCH | ENG | ENPEC | ENARP | SMB TYPE | 保留 | SMBUS | PE |
| | rw | res | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | res | rw | rw |

赋的值中，0x0100 那么也就是第 8 位为 1，其他为 0，我们找到第八位

位8**START:** 起始条件产生 (Start generation)

软件可以设置或清除该位，或当起始条件发出后或PE=0时，由硬件清除。

在主模式下：

0: 无起始条件产生；

1: 重复产生起始条件。

在从模式下：

0: 无起始条件产生；

1: 当总线空闲时，产生起始条件。

根据第八位的描述，该位为 1 时，即可产生一个起始条件。把 0xFEFF 赋值给 CR1 时，因为是相与的，那么也就是第 8 位为 0，我们可以看到，当第八位为 0 时，是没有起始条件产生的。这个便是我们的产生起始信号的例程代码。

➤ 我们再看下**停止信号**

```
/* Send STOP condition */
I2C_GenerateSTOP(I2C1, ENABLE);

void I2C_GenerateSTOP(I2C_TypeDef* I2Cx, FunctionalState NewState)
{
    /* Check the parameters */
    assert_param(IS_I2C_ALL_PERIPH(I2Cx));
    assert_param(IS_FUNCTIONAL_STATE(NewState));
    if (NewState != DISABLE)
    {
        /* Generate a STOP condition */
        I2Cx->CR1 |= CR1_STOP_Set;
    }
    else
    {
        /* Disable the STOP condition generation */
        I2Cx->CR1 &= CR1_STOP_Reset;
    }
}
```

函数中，只要 NewState 与 DISABLE 不相等的话，为 CR1 寄存器赋一个 CR1_STOP_Set 的值，相等的话赋一个 CR1_STOP_Reset 的值，我们看下这两个值是多少的

```
/* I2C STOP mask */
#define CR1_STOP_Set          ((uint16_t)0x0200)
#define CR1_STOP_Reset         ((uint16_t)0xFDFF)
```

分别是 0x0200 和 0xFDFF，那么我们看下把它们赋值给 CR1 寄存器后会怎么样，赋的值中，0x0200 那么也就是第 9 位为 1，其他为 0，我们找到第九位

| | |
|----|--|
| 位9 | <p>STOP: 停止条件产生 (Stop generation)</p> <p>软件可以设置或清除该位；或当检测到停止条件时，由硬件清除；当检测到超时错误时，硬件将其置位。</p> <p>在主模式下：</p> <p>0：无停止条件产生；</p> <p>1：在当前字节传输或在当前起始条件发出后产生停止条件。</p> <p>在从模式下：</p> <p>0：无停止条件产生；</p> <p>1：在当前字节传输或释放SCL和SDA线。</p> |
|----|--|

注：当设置了STOP、START或PEC位，在硬件清除这个位之前，软件不要执行任何对I2C_CR1的写操作；否则有可能会第2次设置STOP、START或PEC位。

当第九位为1时，主模式下是产生一个停止条件，从模式下，是释放SCL和SDA线。相等时，与上0xFDFF，那么第九位为0，也就是无停止条件产生。

其他的信号查看方法和起始信号与停止信号一样。

以下是MAIN函数中，完成数据写入读出比对情况：

```
/* 如果上述测试已经通过则下面的测试可以不需要，以下是另一种方式的测试 */
// Write on I2C EEPROM from EEPROM_WriteAddress0
printf("\r\n\r\n I2C_EE_BufferWrite EEP_Tx_Buffer: [%s]", EEP_Tx_Buffer);
I2C_EE_BufferWrite(EEP_Tx_Buffer, EEP_Firstpage, EEPTX_BUFSIZE-1);
// Read from I2C EEPROM from EEPROM_ReadAddress0
I2C_EE_BufferRead(EEP_Rx_Buffer, EEP_Firstpage, EEPTX_BUFSIZE-1);
printf("\r\n I2C_EE_BufferRead EEP_RX_Buffer: [%s]", EEP_RX_Buffer);
// Check if the data written to the memory is read correctly
TransferStatus = Buffercmp(EEP_Tx_Buffer, EEP_RX_Buffer, EEPTX_BUFSIZE);
if(FAILED == TransferStatus)
{
    /*失败-->数据写入内存读取错误*/
    printf("\r\n FAILED-->The data written to the memory is read Error!\r\n");
}
else
{
    /*通过了-->数据写入内存读取正确*/
    printf("\r\n PASSED-->The data written to the memory is read correctly! \r\n");
}
printf("\r\n ");

//从I2C eepm写在EEPROM_WriteAddress2
printf("\r\n\r\n I2C_EE_BufferWrite EEP_Tx_Buffer: [%s]", EEP_Tx_Buffer);
I2C_EE_BufferWrite(EEP_Tx_Buffer, EEP_Randompage, EEPTX_BUFSIZE-1);
// 从EEPROM_ReadAddress2读取I2C eepm
I2C_EE_BufferRead(EEP_Rx_Buffer, EEP_Randompage, EEPTX_BUFSIZE-1);
printf("\r\n I2C_EE_BufferRead EEP_RX_Buffer: [%s]", EEP_RX_Buffer);
// 检查数据写入内存读取正确
TransferStatus &= Buffercmp(EEP_Tx_Buffer, EEP_RX_Buffer, EEPTX_BUFSIZE);

if(FAILED == TransferStatus)
{
    /*失败-->数据写入Randompage内存读取错误*/
    printf("\r\n FAILED-->The data written to the Randompage memory is read Error!\r\n");
}
else
{
    /*通过了-->数据写入Randompage内存读取正确*/
    printf("\r\n PASSED-->The data written to the Randompage memory is read correctly!
}
printf("\r\n\r\n I2C_EEPROM TEST Finished!!!");
```

所以，我们的神舟 III 号执行我们的这个 MAIN 函数后就可以得到我们上面所达到的实验现象了。

7.20 FLASH模拟EEPROM

7.20.1 意义与作用

神舟 III 号的主芯片是 STM32F103ZET6，它本身没有自带 EEPROM，STM32 具有 IAP（在应用编程）功能，我们可以把它的 FLASH 当成 EEPROM 来使用。我们是将数据直接存放在 STM32 内部，而不是存放在 W25x16 里。

7.20.2 STM32 Flash浅析

不同型号的 STM32，其 FLASH 容量也有所不同，最小的只有 16K 字节，最大的则达到了 1024K 字节。神舟 III 号的主芯片是 STM32F103ZET6 的 FLASH 容量为 512K 字节，属于大容量产品（另外还有中容量和小容量产品），大容量产品的闪存模块组织如图所示：

| 块 | 名称 | 地址范围 | 长度(字节) |
|----------------|---------------|---------------------------|--------|
| 主存储器 | 页0 | 0x0800 0000 – 0x0800 07FF | 2K |
| | 页1 | 0x0800 0800 – 0x0800 0FFF | 2K |
| | 页2 | 0x0800 1000 – 0x0801 17FF | 2K |
| | 页3 | 0x0800 1800 – 0x0801 FFFF | 2K |
| | . | . | . |
| | . | . | . |
| | 页255 | 0x0807 F800 – 0x0807 FFFF | 2K |
| 信息块 | 启动程序代码 | 0x1FFF F000 – 0x1FFF F7FF | 2K |
| | 用户选择字节 | 0x1FFF F800 – 0x1FFF F80F | 16 |
| 闪存存储器 接口寄存器 | FLASH_ACR | 0x4002 2000 – 0x4002 2003 | 4 |
| | FLASH_KEYR | 0x4002 2004 – 0x4002 2007 | 4 |
| | FLASH_OPTKEYR | 0x4002 2008 – 0x4002 200B | 4 |
| | FLASH_SR | 0x4002 200C – 0x4002 200F | 4 |
| | FLASH_CR | 0x4002 2010 – 0x4002 2013 | 4 |
| | FLASH_AR | 0x4002 2014 – 0x4002 2017 | 4 |
| | 保留 | 0x4002 2018 – 0x4002 201B | 4 |
| | FLASH_OBR | 0x4002 201C – 0x4002 201F | 4 |
| | FLASH_WRPR | 0x4002 2020 – 0x4002 2023 | 4 |

STM32 的闪存模块由：主存储器、信息块和闪存存储器接口寄存器等 3 部分组成。

主存储器，该部分用来存放代码和数据常数（如 const 类型的数据）。对于大容量产品，其被划分为 256 页，每页 2K 字节。注意，小容量和中容量产品则每页只有 1K 字节。从上图可以看出主存储器的起始地址就是 0X08000000，B0、B1 都接 GND 的时候，就是从 0X08000000 开始运行代码的。

信息块，该部分分为 2 个小部分，其中启动程序代码，是用来存储 ST 自带的启动程序，用于串口下载代码，当 B0 接 V3.3，B1 接 GND 的时候，运行的就是这部分代码。用户选择字节，则一般用于配置写保护、读保护等功能。

闪存存储器接口寄存器，该部分用于控制闪存读写等，是整个闪存模块的控制机构。

对主存储器和信息块的写入由内嵌的闪存编程/擦除控制器(FPEC)管理；编程与擦除的高电压由内部产生。

在执行闪存写操作时，任何对闪存的读操作都会锁住总线，在写操作完成后读操作才能正确地进行；既在进行写或擦除操作时，不能进行代码或数据的读取操作。

闪存的读取：

内置闪存模块可以在通用地址空间直接寻址，任何 32 位数据的读操作都能访问闪存模块的内容并得到相应的数据。读接口在闪存端包含一个读控制器，还包含一个 AHB 接口与 CPU 衔接。这个接口的主要工作是产生读闪存的控制信号并预取 CPU 要求的指令块，预取指令块仅用于在 I-Code 总线上的取指操作，数据常量是通过 D-Code 总线访问的。这两条总线的访问目标是相同的闪存模块，访

问 D-Code 将比预取指令优先级高。

这里要特别留意一个闪存等待时间，因为 CPU 运行速度比 FLASH 快得多，STM32F103 的 FLASH 最快访问速度 $\leq 24\text{MHz}$ ，如果 CPU 频率超过这个速度，那么必须加入等待时间，比如我们一般使用 72MHz 的主频，那么 FLASH 等待周期就必须设置为 2，该设置通过 FLASH_ACR 寄存器设置。

闪存的编程和擦除：

STM32 的闪存编程是由 FPEC（闪存编程和擦除控制器）模块处理的，这个模块包含 7 个 32 位寄存器，它们分别是：

- FPEC 键寄存器(FLASH_KEYR)
- 选择字节键寄存器(FLASH_OPTKEYR)
- 闪存控制寄存器(FLASH_CR)
- 闪存状态寄存器(FLASH_SR)
- 闪存地址寄存器(FLASH_AR)
- 选择字节寄存器(FLASH_OBR)
- 写保护寄存器(FLASH_WRPR)

其中 FPEC 键寄存器总共有 3 个键值：

RDPRT 键=0X000000A5

KEY1=0X45670123

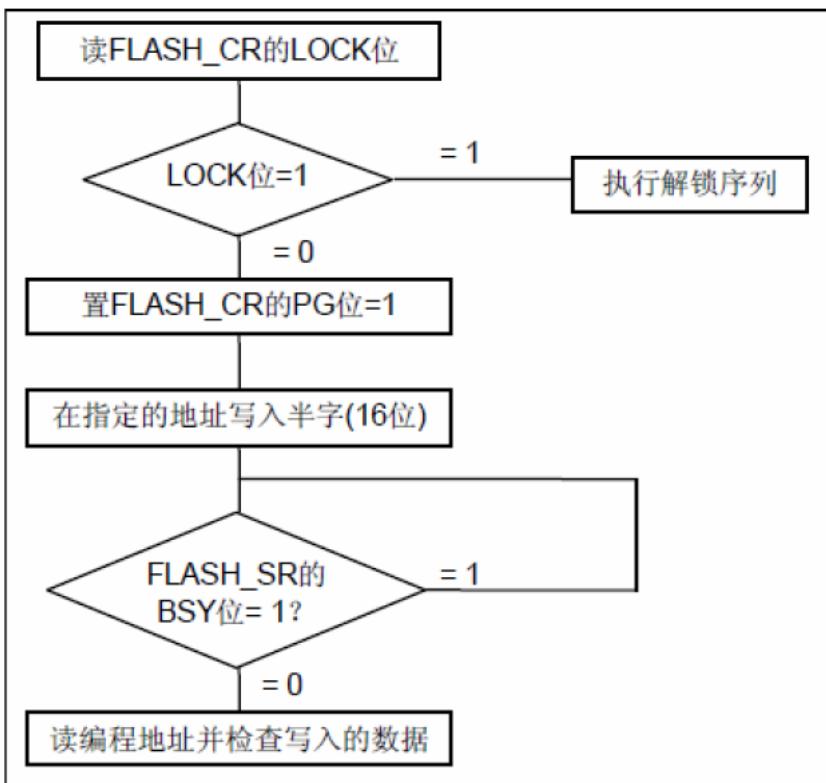
KEY2=0XCDEF89AB

STM32 复位后，FPEC 模块是被保护的，不能写入 FLASH_CR 寄存器；通过写入特定的序列到 FLASH_KEYR 寄存器可以打开 FPEC 模块（即写入 KEY1 和 KEY2），只有在写保护被解除后，我们才能操作相关寄存器。

STM32 闪存的编程每次必须写入 16 位（不能单纯的写入 8 位数据哦！），当 FLASH_CR 寄存器的 PG 位为‘1’时，在一个闪存地址写入一个半字将启动一次编程；写入任何非半字的数据，FPEC 都会产生总线错误。在编程过程中(BSY 位为‘1’)，任何读写闪存的操作都会使 CPU 暂停，直到此次闪存编程结束。

同样，STM32 的 FLASH 在编程的时候，也必须要求其写入地址的 FLASH 是被擦除了的（也就是其值必须是 0xFFFF），否则无法写入，在 FLASH_SR 寄存器的 PGERR 位将得到一个警告。

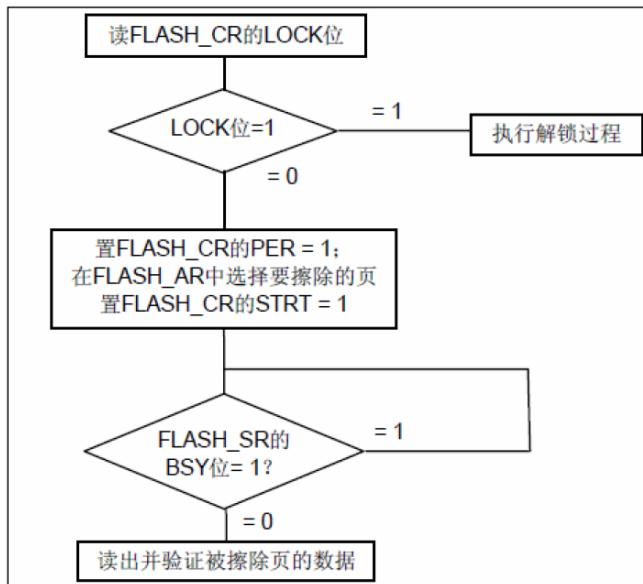
STM32 的 FLASH 编程过程如图所示：



从上图可以得到闪存的编程顺序如下：

- 检查FLASH_CR的LOCK是否解锁，如果没有则先解锁
- 检查FLASH_SR寄存器的BSY位，以确认没有其他正在进行的编程操作
- 设置FLASH_CR寄存器的PG位为'1'
- 在指定的地址写入要编程的半字
- 等待BSY位变为'0'
- 读出写入的地址并验证数据

前面提到，我们在 STM32 的 FLASH 编程的时候，要先判断缩写地址是否被擦除了，所以，我们有必要再绍一下 STM32 的闪存擦除，STM32 的闪存擦除分为两种：页擦除和整片擦除。页擦除过程如图所示：



从上图可以看出，STM32的页擦除顺序为：

- 检查FLASH_CR的LOCK是否解锁，如果没有则先解锁
- 检查FLASH_SR寄存器的BSY位，以确认没有其他正在进行的闪存操作
- 设置FLASH_CR寄存器的PER位为'1'
- 用FLASH_AR寄存器选择要擦除的页
- 设置FLASH_CR寄存器的STRT位为'1'
- 等待BSY位变为'0'
- 读出被擦除的页并做验证

7.20.3 实验原理

本实验写入数据到 Flash，然后读取出来，并再串口上显示读写出来的数据。可以通过数据的对比，判断读出数据是否一致。

7.20.4 硬件设计

本实验我们用到串口和 TFT 彩屏。相关的硬件这里前面已经涉及也可以直接参考开发板原理图，不再重复。

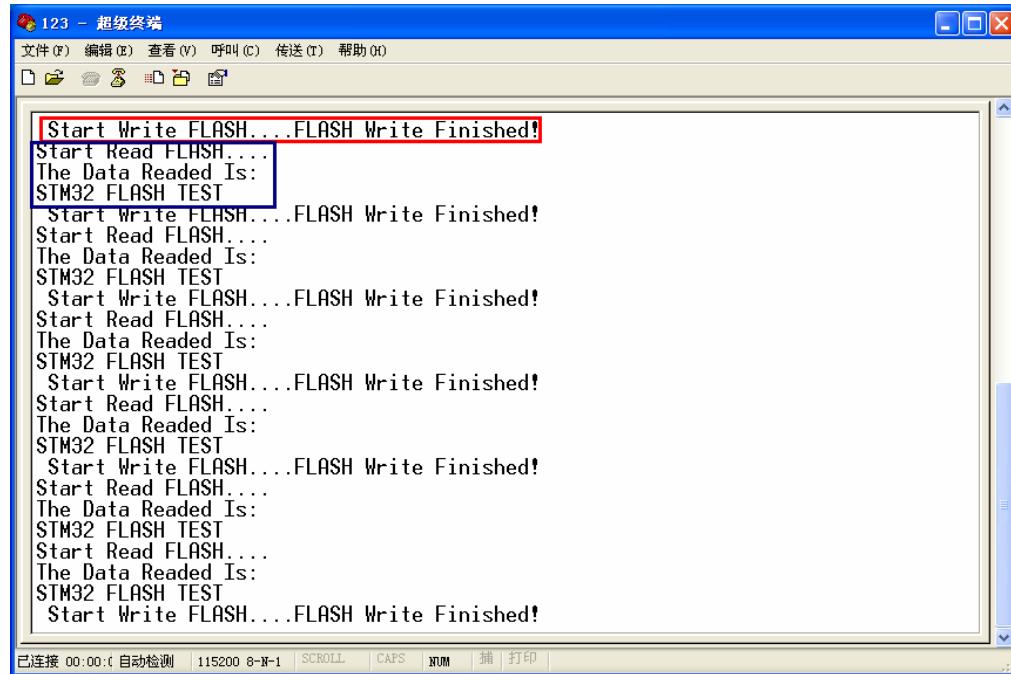
7.20.5 软件设计

7.20.6 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.20.7 实验现象

将程序烧录到开发板，按下复位按键，串口打印信息如下：



```
Start Write FLASH....FLASH Write Finished!
Start Read FLASH....
The Data Readed Is:
STM32 FLASH TEST
Start Write FLASH....FLASH Write Finished!
Start Read FLASH....
The Data Readed Is:
STM32 FLASH TEST
Start Write FLASH....FLASH Write Finished!
Start Read FLASH....
The Data Readed Is:
STM32 FLASH TEST
Start Write FLASH....FLASH Write Finished!
Start Read FLASH....
The Data Readed Is:
STM32 FLASH TEST
Start Write FLASH....FLASH Write Finished!
Start Read FLASH....
The Data Readed Is:
STM32 FLASH TEST
Start Write FLASH....FLASH Write Finished!
```

7.21 TIMER定时器中断实验

7.21.1 意义与作用

对于TIMER定时器大家应该不会陌生，51单片机内部就有定时器，作为高性能的STM32开发板自然少不了对其介绍。定时器属于STM32处理器的内部资源。神舟系列的本例程通过TIMER定时器，产生的中断定时在串口输出打印信息。我们按照以下几个部分对STM32处理器的TIMER进行学习。

7.21.2 实验原理

神舟III开发板使用的处理器是ARM CORTEX-M3系列的STM32F103ZET，其内部含有多达10个定时器，包括：

- 多达4个16位定时器，每个定时器有多达4个用于输入捕获/输出比较/PWM或脉冲计数的通道和增量编码器输入
- 1个16位带死区控制和紧急刹车，用于电机控制的PWM高级控制定时器
- 2个看门狗定时器(独立的和窗口型的)
- 系统时间定时器：24位自减型计数器
- 2个16位基本定时器用于驱动DAC

按功能分为：

- 2个高级控制定时器(两个高级控制定时器TIM1和TIM8，可以被看成是分配到6个通道的三相PWM发生器，它具有带死区插入的互补PWM输)
- 4个普通定时器(达4个可同步运行的标准定时器TIM2、TIM3、TIM4和TIM5)
- 2个基本定时器(这2个定时器主要是用于产生DAC触发信号，也可当成通用的16位时基计数器)
- 2个看门狗定时器
- 1个系统嘀嗒定时器SysTick。

下表比较了高级控制定时器、普通定时器和基本定时器的功能：

表4 定时器功能比较

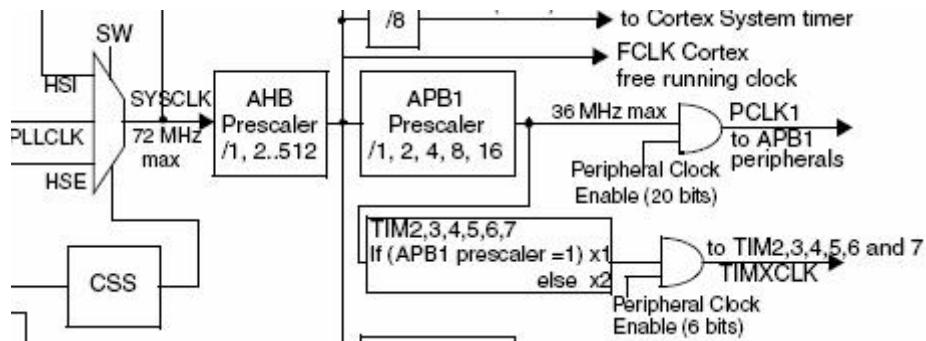
| 定时器 | 计数器分辨率 | 计数器类型 | 预分频系数 | 产生DMA请求 | 捕获/比较通道 | 互补输出 |
|------------------------------|--------|-----------------|--------------------|---------|---------|------|
| TIM1 TIM8 | 16位 | 向上, 向下, 向上/下 | 1~65536之间 的任意整数 | 可以 | 4 | 有 |
| TIM2 TIM3 TIM4 TIM5 | 16位 | 向上, 向下, 向上/下 | 1~65536之间 的任意整数 | 可以 | 4 | 没有 |
| TIM6 TIM7 | 16位 | 向上 | 1~65536之间 的任意整数 | 可以 | 0 | 没有 |

前面我们已经介绍了系统嘀嗒定时器SysTick的使用，本次示例以通用定时器TIM5为例来完成一个一秒中断一次的基本功能。通用定时器介绍如下：

通用定时器(TIMx) STM32F103xC系列产品中，内置了多达4个可同步运行的标准定时器(TIM2、TIM3、TIM4和TIM5)。每个定时器都有一个16位的自动加载递加/递减计数器、一个16位的预分频器和4个独立的通道，每个通道都可用于输入捕获、输出比较、PWM和单脉冲模式输出，在最大的封装配置中可提供最多16个输入捕获、输出比较或PWM通道。它们还能通过定时器链接功能与高级控制定时器共同工作，提供同步或事件链接功能。在调试模式下，计数器可以被冻结。任一标准定时器都能用于产生PWM输出。每个定时器都有独立的DMA请求机制。这些定时器还能够处理增量编码器的信号，也能处理1至3个霍尔传感器的数字输出。

STM32定时器定时时间的计算

首先我们来看一下如何计算定时器的时间来完成一个一秒中断一次的基本功能。



假如系统时钟是 72Mhz, 如上图所示 TIM2-7 是由 PCLK1 得到, 一般的配置包括我们本次的示例中 $PCLK2 = HCLK$ (72MHz)。也就是说本次示例中 TIM5 模块入口时钟是 72Mhz。

TIM5 模块入口时钟经过一个预分频器后的时钟才是 TIM5 计数器的时钟 (用 CK_CNT 表示)。预分频器的系数为: $TIMx_PSC$, 当 $TIMx_PSC=0$ 时表示不分频, 则 TIM5 计数器的时钟用 CK_CNT =模块入口时钟 72Mhz; 当 $TIMx_PSC=1$ 时表示不分频, 则 TIM5 计数器的时钟用 CK_CNT =模块入口时钟 36Mhz; 以此类推。

公式为: $CK_CNT = fCK_PSC / (PSC[15:0] + 1)$, 其中 PSC 最大为 65535。

其次是 TIM5 计数器的计数值的设置, TIM5 计数器以 CK_CNT 为时钟来计数, 向下计数到 0 或向上计数到设定值 ($TIMx_ARR$) 则产生中断。我们以向上计数为例, 从 0 开始计数到设定值 $TIMx_ARR$ 时产生中断。要产生一秒一次中断则要使计数器的值乘以预分频值=系统时钟 72MHz, 其中计数器的值和预分频值都必须小于 65535。我们使预分频值为 7200, 计数器值为 10000, 则 $7200 * 10000 = 7200000$ 即 72M。其中拆分法有很多, 比如 $35000 * 2000 = 7200000$, 只要注意计数器的值和预分频值都必须小于 65535 即可。

当然既然是一秒一次的重复中断, 当然中断要是自动装载的。

7.21.3 硬件设计

TIMER 定时器为 STM32 处理器内部组件, 这部分不需要硬件电路, 这里仅在中断产生时, 进行串口打印操作。

说明: 本实例需要用到使用GPIO管脚控制LED的相关知识和串口相关知识, 关于LED的其它函数请查看“LED跑马灯实验”, 关于串口的函数请查看“串口输入输出实验”, 此处不再介绍。本历程中主要是针对STM32定时器示例中新增添的代码进行说明。

7.21.4 软件设计

进入例程的文件夹, 然后打开 Project \ MDK-ARM \ Project.uvproj 文件

```

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Project 神舟III号 main.c
01 /***** (C) COPYRIGHT 2013 www.armjishu.com *****/
02 * 文件名 : main.c
03 * 描述 : 实现STM32F103ZE神舟III号开发板的TIM定时器功能实验
04 * 实验平台: STM32神舟开发板
05 * 标准库 : STM32F10x_StdPeriph_Driver V3.5.0
06 * 作者 : www.armjishu.com
07 *****/
08
09 /* Includes */
10 #include "SZ_STM32F103ZE_LIB.h"
11
12 /* Private typedef */
13 /* Private define */
14 /* Private macro */
15 /* Private variables */
16 /* Private function prototypes */
17 void SysTick Handler User(void);

```

可以看到工程已经被打开, 下面开始具体分析程序代码:

```
int main(void)
{
    /*!< 在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了时钟,
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
       SystemInit()函数的实现位于system_stm32f10x.c文件中。 */

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    /* 注意串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为SZ_STM32_COM2 */
    /* 串口2初始化 */
    SZ_STM32_COMInit(COM2, 115200);
    printf(" ARMJISHU.COM-->TIM5定时器实验\r\n");

    TIM5_Init();

    /* Infinite loop 主循环 */
    while (1)
    {
        /* 此处可以添加用户的程序 */
    }
}
```

代码分析 1: SZ_STM32_LEDInit(LED1)函数初始化 LED 灯，前面我们已经讲解，我们就不再重复。

代码分析 2: SZ_STM32_COMInit () 函数初始化串口 2，前面我们已经讲解，我们就不再重复。

代码分析 3: SZ_STM32_SysTickInit ()函数初始化系统定时器 SysTick，前面我们已经讲解，我们就不再重复。

代码分析 4: TIM5_Init()函数对 TIM5 进行初始化。

```
void TIM5_Init(void)
{
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;

    /* TIM5 clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);

    /* -----
     * TIM4 Configuration: Output Compare Timing Mode:
     * TIM2CLK = 36 MHz, Prescaler = 7200, TIM2 counter clock = 7.2 MHz
     * ----- */

    /* Time base configuration */
    //这个就是自动装载的计数值，由于计数是从0开始的，计数10000次后为9999
    TIM_TimeBaseStructure.TIM_Period = (10000 - 1);
    // 这个就是预分频系数，当由于为0时表示不分频所以要减1
    TIM_TimeBaseStructure.TIM_Prescaler = (7200 - 1);
    // 高级应用本次不涉及。定义在定时器时钟(CK_INT)频率与数字滤波器(ETR,TIX)
    // 使用的采样频率之间的分频比例
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    //向上计数
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    //初始化定时器5
    TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure);

    /* Clear TIM5 update pending flag[清除TIM5溢出中断标志] */
    TIM_ClearITPendingBit(TIM5, TIM_IT_Update);

    /* TIM IT enable */ //打开溢出中断
    TIM_ITConfig(TIM5, TIM_IT_Update, ENABLE);
}
```

本函数配置定时器参数。前面实验原理章节的“STM32定时器定时时间的计算”部分已经介绍了定时时间的计算方法，从 TIM5_Init函数中我们可以看出TIM5的参数配置流：

1) TIM5时钟使能。调用RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE)函数现实。

2) 初始化定时器参数, 设置自动重装值, 分频系数, 计数方式等。调用 `TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure)` 函数实现。函数中用到结构体, 结构体类型是 `TIM_TimeBaseInitTypeDef`。

```
typedef struct
{
    uint16_t TIM_Prescaler;
    uint16_t TIM_CounterMode;
    uint16_t TIM_Period;
    uint16_t TIM_ClockDivision;
    uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef
```

大家翻开其它的工程, 我们初始化配置其它的接口的话, 也常常用到类似的函数。这样的结构体包含了这个接口的信息。比如我们配置GPIO管脚的时候, 要确定是那个管脚, 管脚的速率, 输入输出模式等。这里 `TIM_Prescaler` 用来配置分频系数。`TIM_CounterMode` 用来设置计数方式。`TIM_Period` 用来配置自动重载计数周期。`TIM_RepetitionCounter` 用来设置时钟分频因子。

3) 经过 `TIM_TimeBaseInit()` 函数初始化 `TIM5` 之后, 我们清除 `TIM5` 溢出中断标志, 打开 `TIM5` 的溢出中断, 使能 `TIM5`, 当然还要配置 `TIM5` 中断参数。这里的每一个功能都调用了对应的函数实现。到此我们的初始化函数基本完成。

代码分析 5: `TIM5` 中断触发的时候, 进入中断服务函数。我们看一下中断服务函数 `TIM5_IRQHandler()`。

```
void TIM5_IRQHandler(void)
{
    /* www.armjishu.com ARM技术论坛 */
    static u32 counter = 0;

    if (TIM_GetITStatus(TIM5, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM5, TIM_IT_Update);
        /* LED1指示灯状态取反 */
        SZ_STM32_LEDToggle(LED1);

        /* armjishu.com 提心您: 不建议在中断中使用Printf, 此示例只
         * print("n\rarmjishu.com 提心您: 不建议在中断中使用Printf,
         * printf("ARMJISHU.COM-->TIM5:%d\n\r", counter++);
    }
}
```

中断服务函数中, 我们调用 `SZ_STM32_LEDToggle(LED1)` 函数控制 LED1 灯闪烁, 并且通过串口打印提示信息, 信息中的数值加1。`armjishu.com 提心您: 不建议在中断中使用Printf, 此示例只是演示。`

为什么使用静态变量 `counter`? 目的是每次进入定时器中断函数数静态变量 `counter` 保持上次退出时的值, 这样才能达到计数的目的。否则, 如果去掉 `static` 关键字则每次进入中断函数时 `counter` 为0, 串口打印的数值也为0, 就达不到每次打印的数值加1的目的。

由上述分析可知每产生一次定时器中断, 进入定时器中断服务程序, 中断服务程序中, 都调用 `SZ_STM32_LEDToggle(LED1)` 函数, 而我们配置的是每秒中断一次, 那么 `SZ_STM32_LEDToggle(LED1)` 函数每秒中会被调用1次。同时使用静态变量 `counter` 将中断服务函数执行的次数记录下来。

至此神舟系列 TIMER 定时器中断相关软件程序介绍完毕!

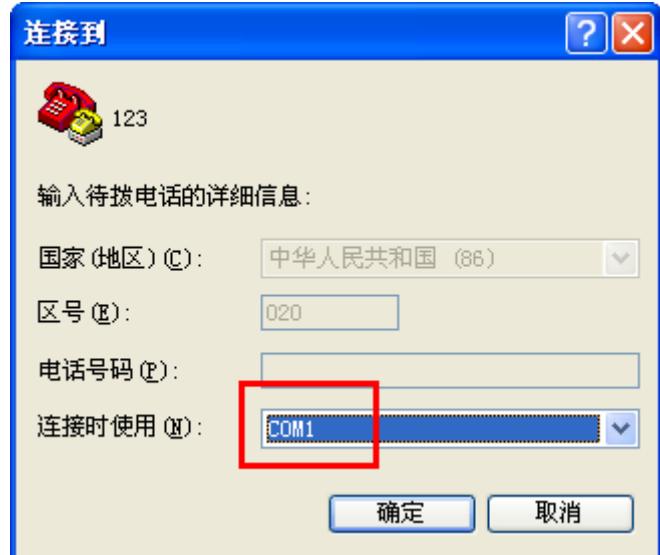
7.21.5 下载与验证

如果在 MDK 开发环境中, 下载编译好的固件或者在线调试, 请按 3.5 如何在 MDK 开发环境中使用 JLINK 在线调试小节进行操作。

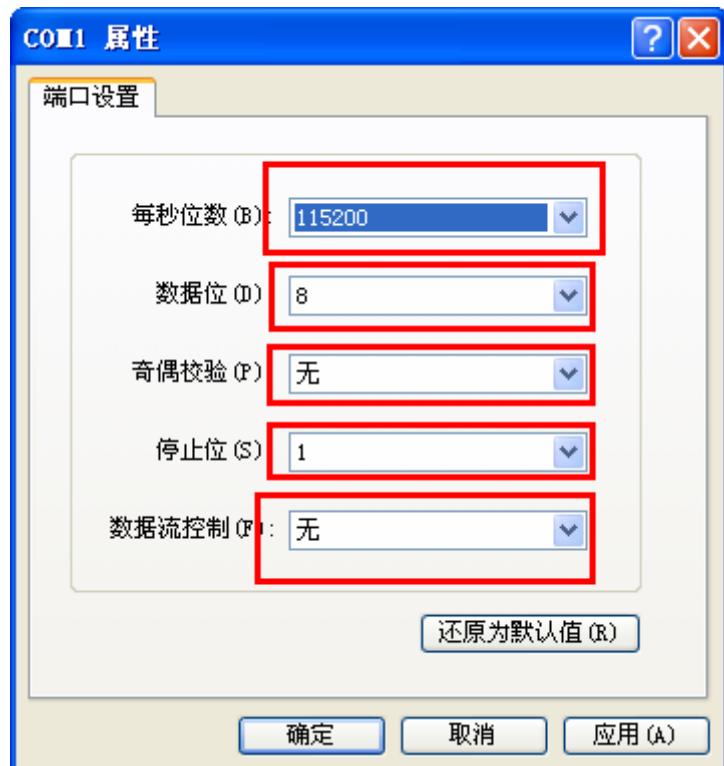
7.21.6 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，

用串口线神舟 III 号串口 2 与电脑连接，并打开超级终端，按以下设置，如下图：



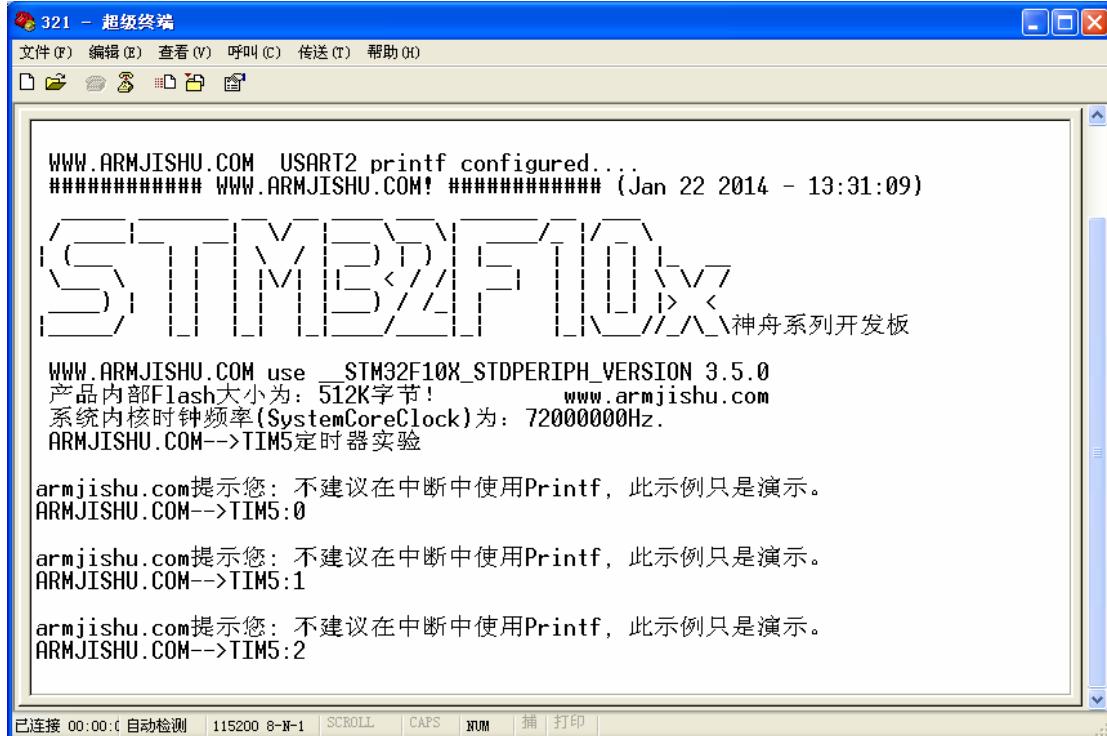
选择 COM1；按确定



再按确定，完成超级终端设置。

重新打开电源；神舟III号STM32开发板上LED 1灯间断亮灭，LED2至LED4都常亮。
并且串口打印的数值每秒加1：

armjishu.com 提心您：不建议在中断中使用Printf，此示例只是演示。



7.22 IWDG独立看门狗

STM32 内部自带了 2 个看门狗：独立看门狗（IWDG）和窗口看门狗（WWDG）。这一章，介绍如何使用 STM32 的独立看门狗（以下简称 IWDG）。

7.22.1 什么是看门狗

在由单片机构成的微型计算机系统中，由于单片机的工作常常会受到来自外界电磁场的干扰，造成程序的跑飞，而陷入死循环，程序的正常运行被打断，由单片机控制的系统无法继续工作，会造成整个系统的陷入停滞状态，发生不可预料的后果，所以出于对单片机运行状态进行实时监测的考虑，便产生了一种专门用于监测单片机程序运行状态的模块或者芯片，俗称“看门狗”（watchdog）。

其实就是相当一个装满时间数值的倒计时器一样，启动之后就开始倒计时，倒计时为零传送一个复位信号到 CPU，又重新开始倒计时。

7.22.2 为什么是独立看门狗呢

就是在芯片内部独立的一部分，独立的看门狗是基于一个 12 位的递减计数器和一个 8 位的预分频器，它由一个内部独立的 40kHz 的 RC 振荡器提供时钟；因为这个 RC 振荡器独立于主时钟，所以它可运行于停机和待机模式。它可以被当成看门狗用于在发生问题时复位整个系统，或作为一个自由定时器为应用程序提供超时管理。通过选项字节可以配置成是软件或硬件启动看门狗。在调试模式下，计数器可以被冻结。

7.22.3 怎么使用独立看门狗IWDG

前面已经介绍了，独立看门狗相当于一个独立的倒计时器，那要用之前就要给这个倒计时设置一个启动开关才能用，而启动开关这里就是用 Key Register 键值寄存器；有了开关那可以用，但倒计数嵌入式专业技术论坛（www.armjishu.com）出品 第 457 页，共 900 页

的值不可能是无穷大，所以在这里也要设置一个 Reload Register 重装载寄存器用来控制我们想要时间；控制启动开关、时间长短都有了，还有一个倒计时节奏速度，也就是频率，这里就用到了 Prescaler Register 预分频寄存器；好，有了这三个寄存器，我们就可以进行对它控制操作。对这几个寄存器进行进一步的了解：

Key Register 键值寄存器的作用是：

- (1) 控制独立看门狗的启用功能；
- (2) 改变另外两个寄存器 (Prescaler Register 预分频寄存器) 和 (Reload Register 重装载寄存器) 作用状态功能；
- (3) 改变看门狗计数器为 0 时，复位的产生或重新再计数。

Prescaler Register 预分频寄存器的作用是：用来设置看门狗时钟的分频系数。

Reload Register 重装载寄存器的作用是：用来重新加载为倒计数计数器的初值。

Status Register 状态寄存器的作用是：设置 (Prescaler Register 预分频寄存器) 和 (Reload Register 重装载寄存器) 两个寄存器读操作的有效性。

其实使用独立看门狗就是在控制和设置几个寄存器。接下来一步一步详细操作怎么使用独立看门狗 IWDG。

键值寄存器 IWDG_KR (Key Register):

首先是键值寄存器 IWDG_KR，该寄存器功能上面所说，接下是各位的描述，如图：

| | | | | | | | | | | | | | | | |
|-----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| KEY[15:0] | | | | | | | | | | | | | | | |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |

| | |
|----------|--|
| 位 31: 16 | 保留，始终读为 0。 |
| 位 15: 0 | KEY[15:0]: 键值（只写寄存器，读出值为 0x0000） |
| | 软件必须以一定的间隔写入 0xAAAA，否则，当计数器为 0 时，看门狗会产生复位。 |
| | 写入 0x5555 表示允许访问 IWDG_PR 和 IWDG_RLR 寄存器。 |
| | 写入 0xCCCC，启动看门狗工作（若选择硬件看门狗则不受此命令字限制）。 |

在键值寄存器 (IWDG_KR) 中写入 0xCCCC，开始启用独立看门狗；此时计数器开始从其复位值 0xFFFF 递减计数。当计数器计数到末尾 0x000 时，会产生一个复位信号(IWDG_RESET)。无论何时，只要键寄存器 IWDG_KR 中被写入 0xAAAA，IWDG_RLR 中的值就会被重新加载到计数器中从而避免产生看门狗复位，这里相当于喂狗，每一定时间内喂一次，防止看门狗自动复位。

IWDG_PR 和 IWDG_RLR 寄存器具有写保护功能。要修改这两个寄存器的值，必须先向 IWDG_KR 寄存器中写入 0x5555。将其他值写入这个寄存器将会打乱操作顺序，寄存器将重新被保护。重装载操作(即写入 0xAAAA)也会启动写保护功能。

预分频寄存器 IWDG_PR (Prescaler Register):

接下来，介绍预分频寄存器 (IWDG_PR)，该寄存器用来设置看门狗时钟的分频系数，最低为 4，最高位 256，该寄存器是一个 32 位的寄存器，但是我们只用了最低 3 位，其它都是保留位。预分频寄存器各位定义如图：

| | | | | | | | | | | | | | | | |
|----------|----|----|----|----|----|----|----|----|----|----|----|---------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | | | | | PR[2:0] | | | |
| rw rw rw | | | | | | | | | | | | | | | |

| | |
|---------|--|
| 位 31: 3 | 保留, 始终读为 0。 |
| 位 2: 0 | <p>PR[2:0]: 预分频因子。</p> <p>这些位具有写保护设置, 上面已有介绍。通过设置这些位来选择计数器的预分频因子。要改变预分频因子, IWDG_SR 寄存器的 PVN 位必须位 0。</p> <p>000: 预分频因子=4 001: 预分频因子=8 010: 预分频因子=16 011: 预分频因子=32 100: 预分频因子=64 101: 预分频因子=128 110: 预分频因子=256 111: 预分频因子=256</p> <p>注意: 对此寄存器进行读操作, 将从 VDD 电压域返回预分频值。如果写操作正在进行, 则读回的值可能是无效的因此, 只有当 IWDG_SR 寄存器的 PVU 位为 0 时, 读出的值才有效。</p> |

重装载寄存器 IWDG_RLR (Reload Register):

介绍一下重装载寄存器 IWDG_RLR。该寄存器用来保存重装载到计数器中的值。该寄存器也是一个 32 位寄存器, 但是只有低 12 位是有效的, 该寄存器的各位描述如图:

| | | | | | | | | | | | | | | | |
|--|----|----|----|----------|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | RL[11:0] | | | | | | | | | | | |
| rw | | | | | | | | | | | | | | | |

| | |
|----------|--|
| 位 31: 12 | 保留, 始终读为 0。 |
| 位 11: 0 | <p>RL[11:0]: 看门狗计数器重装载值, 这些位具有写保护功能。前面已有介绍。</p> <p>用于定义看门狗计数器的重装载值, 每当向 IWDG_KG 寄存器写入 0xFFFF 时, 重装载值会被传送到计数器中。随后计数器从这个值开始递减计数。</p> <p>看门狗超时周期可通过此重装载值和时钟预分频值来计算。</p> <p>只有当 IWDG_SR 寄存器中的 RVU 位为 0 时, 才能对此寄存器进行修改。</p> <p>注意: 对此寄存器进行读操作, 将从 VDD 电压域返回预分频。如果写操作正在进行, 则读回的值可能是无效的。因此, 只有当 IWDG_SR 寄存器的 RVU 位为 0 时, 读出的值才有效。</p> |

状态寄存器 IWDG_SR (Status Register) :

| | | | | | | | | | | | | | | | |
|---------|---|----|----|----|----|----|----|----|----|----|----|----|----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | RVU | PVU |
| | | | | | | | | | | | | | | r | r |
| 位 31: 2 | 保留。 | | | | | | | | | | | | | | |
| 位 1 | RVU: 看门狗计数器重装载值更新此位由硬件置‘1’用来指示重装载值的更新正在进行中。当在 VDD 域中的重装载更新结束后，此位由硬件清‘0’(最多需 5 个 40KHz 的 RC 周期)。重装载值只有在 RVU 位被清‘0’后才可更新。 | | | | | | | | | | | | | | |
| 位 0 | PVU: 看门狗预分频值更新此位由硬件置‘1’用来预分频值的更新正在进行中。当在 VDD 域中的预分频值更新结束后，此位由硬件清‘0’(最多需 5 个 40KHz 的 RC 周期)。预分频值只有在 PVU 位被清‘0’后才可更新。 | | | | | | | | | | | | | | |

注：如果在应用程序中使用了多个重装载值或预分频值，则必须在 RVU 位被清除后才能重新改变预装载值，在 PVU 位被清除后才能重新改变预分频值。然而，在预分频和/或重装值更新后，不必等待 RVU 或 PVU 复位，可继续执行下面的代码。(即是在低功耗模式下，此写操作仍会被继续执行完成。) 以上就是对独立看门狗有关几个寄存器的功能操作等介绍。

7.22.4 启动STM32的独立看门狗

STM32 的独立看门狗由内部专门的 40Khz 低速时钟驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 40Khz，而是在 30~60Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 40Khz 的频率计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

接下来就可以启动 STM32 的独立看门狗，启动过程可以按以下步骤实现：

(1) 向 IWDG_KR 写入 0x5555。

通过这步，取消 IWDG_PR 和 IWDG_RLR 的写保护，使后面可以操作这两个寄存器。

设置 IWDG_PR 和 IWDG_RLR 的值。

这两步设置看门狗的分频系数，和重装载的值。

怎么计算看门狗溢出时间呢？

由此，就可以知道看门狗的喂狗时间（也就是看门狗溢出时间），该时间的计算方式为：

$$Tout = ((4 \times 2^{prer}) \times RLR) / 40KHz$$

其中 Tout 为看门狗溢出时间（单位为 ms）；prer 为看门狗时钟预分频值 (IWDG_PR 值)，范围为 0~7(十进制，转二进制 000~111 以上面预分频对应)；RLR 为看门狗的重装载值 (IWDG_RLR 的值低 12 位)。

看门狗复位时间=(预分频×重装载值)/40KHz

根据公式推算就可以知道重装载值、预分频系数与溢出时间的关系，做出一个简单参考表格：

| 重装载值 (RLR 寄存器低 12 位) | 预分频 | 内部 RC | 看门狗复位时间 (ms) |
|----------------------|-----|--------|-----------------------|
| 625 | 64 | 40 KHz | 1000 ms |
| 1 (最小值) | 64 | 40 KHz | 1.6 ms (在预分频 64 最小时间) |

| | | | |
|------------|----|--------|-------------------------|
| 4096 (最大值) | 64 | 40 KHz | 6553.6 ms(在预分频 64 最大时间) |
| 2500 | 16 | 40 KHz | 1000 ms |
| 1 (最小值) | 16 | 40 KHz | 0.4 ms (在预分频 16 最小时间) |
| 4096 (最大值) | 16 | 40 KHz | 1638.4 ms(在预分频 16 最大时间) |

假设：预分频系数设定 prer 值为 4，重装载值 RLR 值为 625，那么就可以得到

$Tout=64 \times 625/40=1000\text{ms}$ ，这样，看门狗的溢出时间就是 1s，只要你在一秒钟之内，有一次写入 0XAAAA 到 IWDG_KR，就不会导致看门狗复位（当然写入多次也是可以的）。这里需要提醒大家的是，看门狗的时钟不是准确的 40Khz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

(2) 向 IWDG_KR 写入 0XAAAA。

通过这句，将使 STM32 重新加载 IWDG_RLR 的值到看门狗计数器里面。即实现独立看门狗的喂狗操作。

(3) 向 IWDG_KR 写入 0XCCCC。

通过这句，来启动 STM32 的看门狗。注意 IWDG 在一旦启用，就不能再被关闭！想要关闭，只能重启，并且重启之后不能打开 IWDG，否则问题依旧，所以在这里提醒大家，如果不用 IWDG 的话，就不要去打开它，免得麻烦。

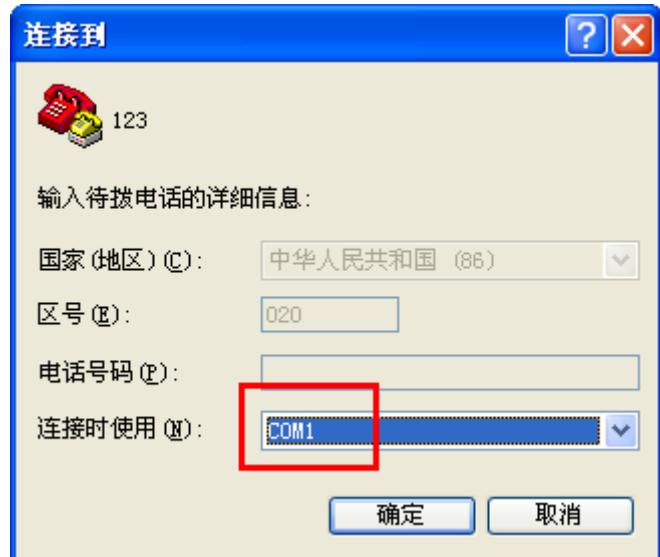
7.22.5 软件设计

7.22.6 下载与验证

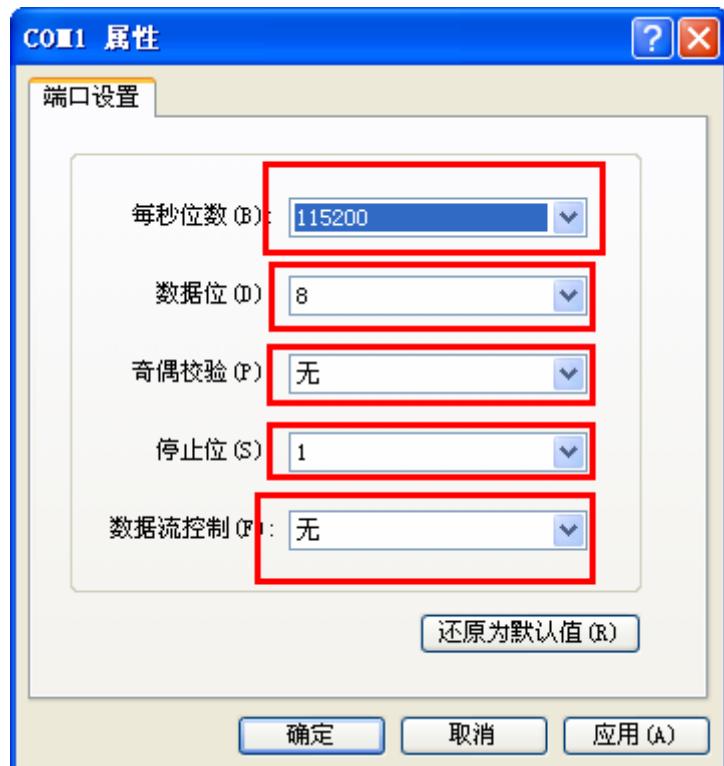
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.22.7 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 2 与电脑连接，并打开超级终端，按以下设置，如下图：



选择 COM1；按确定



再按确定，完成超级终端设置。

重新打开电源；神舟 III 号 STM32 开发板上 LED 1 和 LED3 灯亮慢慢的暗又慢慢的亮，又轮到 LED2 和 LED4 灯亮慢慢的暗又慢慢的亮，一直循环。

超级终端窗口显示信息，如下图



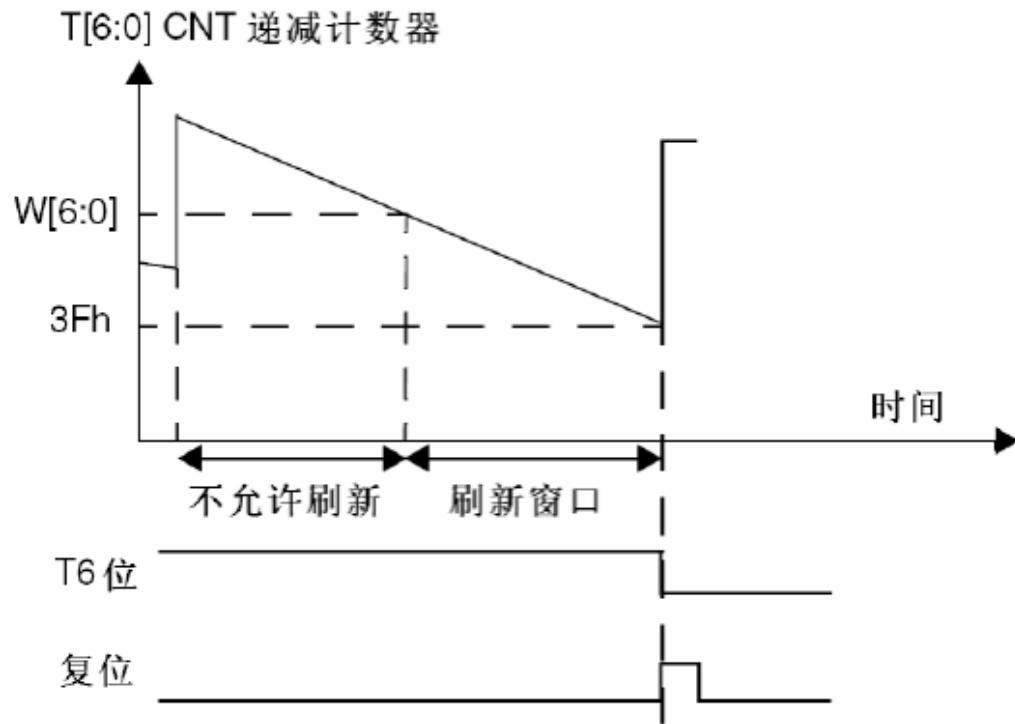
7.23 窗口看门狗

7.23.1 什么是窗口看门狗

窗口看门狗(WWDG)通常被用来监测，由外部干扰或不可预见的逻辑条件造成应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位变成 0 前被刷新，看门狗电路在达到预置的时间周期时，会产生一个 MCU 复位。在递减计数器达到窗口寄存器数值之前，如果 7 位的递减计数器数值(在控制寄存器中)被刷新，那么也将产生一个 MCU 复位。这表明递减计数器需要在一个有限的时间窗口中被刷新。也就是在独立看门狗的递减时间上设置一个有效的“窗口”只有在这“窗口”内才能进行有效的“喂狗操作”。在“窗口”之外的喂狗操作和“窗口”的上线到下限中未进行喂狗操作都将默认未错误操作，将会产生一个 MCU 复位。我们将之称为窗口看门狗。

7.23.2 窗口看门狗的特性

- 可编程的自由运行递减计数器
- 条件复位
 - 当递减计数器的值小于 0x40，(若看门狗被启动)则产生复位。
 - 当递减计数器在窗口外被重新装载，(若看门狗被启动)则产生复位。见 0。
- 如果启动了看门狗并且允许中断，当递减计数器等于 0x40 时产生早期唤醒中断(EWI)，它可以被用于重装载计数器以避免 WWDG 复位，在了解了窗口看门狗的特性后我们在来看窗口看门狗的时序图。



窗口看门狗时序图

时序图中，T[6:0]就是 WWDG_CR 的低七位，W[6:0]即是 WWDG->CFR（控制寄存器）的低七位。T[6:0]就是窗口看门狗的计数器，而 W[6:0]则是窗口看门狗的上窗口（上窗口的值不是固定的用户可以根据自己的需要进行设置，因为 WWDG->CFR 这个寄存器只有低 7 位提供给用户来设置窗口的上限所以我们最大只能设置到 7F，最小不能小于下窗口值），下窗口值是固定的（0X40）。当窗口看门狗的计数器在上窗口值之外被刷新，或者低于下窗口值都会产生复位。

7.23.3 窗口看门狗的计算公式：

窗口看门狗的超时公式如下：

$$T_{wdg} = (4096 \times 2^{WDGTB} \times (T[5:0]+1)) / F_{pclk1};$$

其中：

T_{wdg} : WWDG 超时时间（单位为 ms）

F_{pclk1} : APB1 的时钟频率（单位为 KHz）

WDGTB: WWDG 的预分频系数

T[5:0]: 窗口看门狗的计数器低 6 位

根据上面的公式，那么可以得到最早喂狗时间-最晚喂狗时间表如表所示：

| WDGTB | 最小超时值 | 最大超时值 |
|-------|-------|---------|
| 0 | 113us | 7.28ms |
| 1 | 227us | 14.56ms |
| 2 | 455us | 29.12ms |
| 3 | 910us | 58.25ms |

最小超时值为窗口看门狗计数周期时间=1s/FPCK1X(4096×2^WDGTB)列假设 Fpclk1=36Mhz, 预分频系数为 1, 看门狗的计数周期时间

$us = 1000000(1S=1000ms=1000000us)/36000000(1MHz=1000KHz=1000000ZH) \times (4096 \times 1) \approx 113us$

最大超时值为窗口的最大值=1S/FPCK1× (4096×2*WDGTB) × (7F-3F)

注意: 7F 为窗口上限最大值 40 为窗口下限 STM 系统默认值是固定的 3F 将报错

7.23.4 窗口看门狗要用到哪些寄存器

在了解了窗口看门狗的一些基本特性后我们来介绍下窗口看门狗 3 个寄存器, 首先介绍控制寄存器 (WWDG_CR), 该寄存器的各位描述如下图所示:

| | | | | | | | | | | | | | | | |
|---|---|----|----|----|----|----|----|------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | WDGA | T6 | T5 | T4 | T3 | T2 | T1 | T0 |
| RS RW | | | | | | | | | | | | | | | |
| 位 31: 8 | 保留 | | | | | | | | | | | | | | |
| 位 7 | WDGA:激活位 此位由软件置 1, 但仅能由硬件在硬件复位后清零, 当 WDGA=1 时, 看门狗可以产生复位 0: 禁止看门狗 1: 启用看门狗 | | | | | | | | | | | | | | |
| 位 6: 0 | T[6:0]: 7 位计数器(MSB 至 LSB) (7-bit counter) 这些位用来存储看门狗的计数器值。每(4096x2^WDGTB)个 PCLK1 周期减 1。当计数器值从 40h 变为 3Fh 时(T6 变成 0), 产生看门狗复位 | | | | | | | | | | | | | | |

可以看出, 这里我们的 WWDG_CR 只有低八位有效, T[6: 0]用来存储看门狗的计数器值, 随时更新的, 每个看窗口看门狗计数周期(4096×2^ WDGTB)减 1。当该计数器的值从 0X40 变为 0X3F 的时候, 将产生看门狗复位。

WDGA 位则是看门狗的激活位, 该位由软件置 1, 以启动看门狗, 并且一定要注意的是该位一旦设置, 就只能在硬件复位后才能清零了。

窗口看门狗的第二个寄存器是配置寄存器 (WWDG_CFR), 该寄存器的各位及其描述如下图所示:

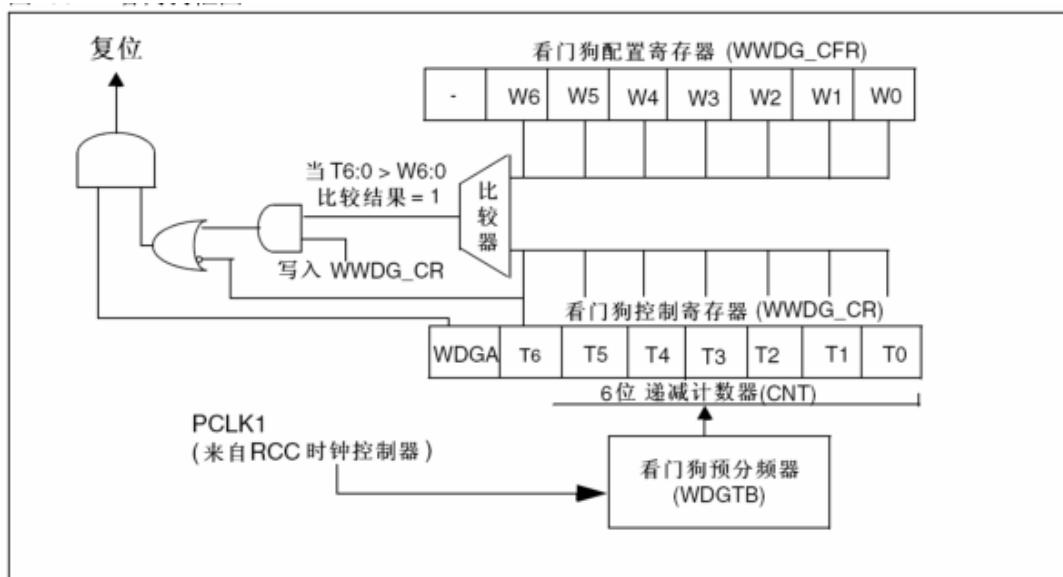
| | | | | | | | | | | | | | | | | | |
|--|----|----|----|----|----|----|----|-----|---------|---------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | | |
| 保留 | | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| 保留 | | | | | | | | EWI | WDG TB1 | WDG TBO | W6 | W5 | W4 | W3 | W2 | W1 | W0 |
| RS RW | | | | | | | | | | | | | | | | | |

| | |
|-------|--|
| 位31:8 | 保留。 |
| 位9 | EWI: 提前唤醒中断 此位若置1，则当计数器值达到40h，即产生中断。 此中断只能由硬件在复位后清除。 |
| 位8:7 | WDGTB[1:0]: 时基 预分频器的时基可根据如下修改： 00: CK计时器时钟(PCLK1除以4096)除以1 01: CK计时器时钟(PCLK1除以4096)除以2 10: CK计时器时钟(PCLK1除以4096)除以4 11: CK计时器时钟(PCLK1除以4096)除以8 |
| 位6:0 | W[6:0]: 7位窗口值 这些位包含了用来与递减计数器进行比较用的窗口值。 |

该位中的 EWI 是提前唤醒中断，也就是在快要产生复位的前一段时间 (T[6:0]=0X40) 来提醒我们，需要进行喂狗了，否则将复位！因此，我们一般用该位来设置中断，当窗口看门狗的计数器值减到 0X40 的时候，如果该位设置，并开启了中断，则会产生中断，我们可以在中断里面向 WWDG_CR 重新写入计数器的值，来达到喂狗的目的。注意这里在进入中断后，必须在不大于 1 个窗口看门狗计数周期的时间（在 PCLK1 频率为 36M 且 WDGTB 为 0 的条件下，该时间为 113us）内重新写 WWDG_CR，否则，看门狗将产生复位！

最后我们要介绍的是状态寄存器 (WWDG_SR)，该寄存器用来记录当前是否有提前唤醒的标志。该寄存器仅有位 0 有效，其他都是保留位。当计数器值达到 40h 时，此位由硬件置 1。它必须通过软件写 0 来清除。对此位写 1 无效。即使中断未被使能，在计数器的值达到 0X40 的时候，此位也会被置 1。

下面我们来看下窗口看门狗的架构图：



如果看门狗被启动(WWDG_CR 寄存器中的 WDGA 位被置'1')，并且当 7 位(T[6:0])递减计数器从 0x40 翻转到 0x3F(T6 位清零)时，则产生一个复位。如果软件在计数器值大于窗口寄存器中的数值时重新装载计数器，将产生一个复位。这里我要讲的是窗口看门狗它不像独立看门狗内部有一个独立的时钟它是用来自与 RCC 的外部时钟驱动的，也就是说当外部时钟受到影响窗口看门狗也会受到相应的影响，这样做有什么好处那？它的好处在于它不光更严格的监视了系统喂狗操作，而且它关联到外部时钟当外部时钟不走了，这狗也将失去作用。

7.23.5 窗口看门狗与独立看门狗的区别

独立看门狗 Iwdg——有独立时钟（内部低速时钟 LSI---40KHz），所以不受系统硬件影响的系统
嵌入式专业技术论坛（www.armjishu.com）出品

故障探测器。主要用于监视硬件错误。

窗口看门狗 wwdg——时钟与系统相同。如果系统时钟不走了，这个狗也就失去作用了，主要用于监视软件错误。

对于一般的看门狗，程序可以在它产生复位前的任意时刻刷新看门狗，但这有一个隐患，有可能程序跑乱了又跑回到正常的地方，或跑乱的程序正好执行了刷新看门狗操作，这样的情况下一般的看门狗就检测不出来了；如果使用窗口看门狗，程序员可以根据程序正常执行的时间设置刷新看门狗的一个时间窗口，保证不会提前刷新看门狗也不会滞后刷新看门狗，这样可以检测出程序没有按照正常的路径运行非正常地跳过了某些程序段的情况。

7.23.6 怎么使用窗口看门狗

介绍完窗口看门狗的原理、窗口看门狗将要用到的特殊功能寄存器以及窗口看门狗与独立看门狗的区别后我们在来介绍要如何使用窗口看门狗，这里我们介绍的是用中断来喂狗，具体步骤如下：

1) 使能 WWDG 时钟

前面我们介绍独立看门狗和窗口看门狗的区别时特别说明了独立看门狗使用的是内部低速时钟，不存在使能的问题。而 WWDG 是使用的 PCLK1 的时钟，在使用前需要先使能时钟的。

2) 设置 WWDG_CFR 和 WWDG_CR 两个寄存器

在使能好时钟过后，我们设置 WWDG 的 CFR 和 CR 两个寄存器，对 WWDG 进行配置。包括使能窗口看门狗、开启中断、设置计数器的初始值、设置窗口值并设置分频数 WDGTB 等。这里要注意的是用户根据自己的需要设定时间，先要通过上面的看门狗超时公式和外部晶振算出我们需要设定的时间那个分频段，在根据预分频时基算出 WWDG_CR 的计数器值。

如：假设我们使用的是 36M 的外部晶振，我们将要设定的时间是 3.41ms，我们不难算出应该设定的 WDGTB[0:0]然后我们可以 $3.41\text{ms} = 1\text{s}/36\text{M} \times (4096 \times 1) \times \text{计算器值}$ 。可以算出大概为 30，也就表示在 WWDG_CR 的计数器次数在加上 3F 得出 WWDG_CR 的计数值为 5D

3) 开启 WWDG 中断并分组

在设置完了 WWDG 后，需要配置该中断的分组及使能。这点通过我们之前所编写的 MY_NVIC_Init 函数实现就可以了。

4) 编写中断服务函数

在最后，还是要编写窗口看门狗的中断服务函数，通过该函数来喂狗，喂狗要快，否则当窗口看门狗计数器值减到 0X3F 的时候，就会引起软复位了。在中断服务函数里面也要将状态寄存器的 EWIF 位清空。

7.23.7 硬件设计

本实验中，我们只需要用到 2 个 LED 灯，分别是 LED1 和 LED2，通过 STM32 的窗口看门狗的功能在看门狗中断的时候对我们的 LED 进行一个取反闪烁的效果，通过串口把数据打印出去。

7.23.8 软件设计

我们先来看下主函数，开始的时候，我们因为需要使用 LED 观察看门狗的状态，所以先是对 LED 进行了一个初始化，并让 LED 都熄灭，防止影响到下面。

```
int main(void)
{
    /*!< 在系统启动文件 (startup_stm32f10x_xx.s) 中已经调用SystemInit() 初始化了时钟,
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
       SystemInit() 函数的实现位于system_stm32f10x.c文件中。*/
    /* 配置NVIC中断优先级分组 */
    NVIC_GroupConfig();

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    SZ_STM32_LEDOff(LED1);
    SZ_STM32_LEDOff(LED2);
    SZ_STM32_LEDOff(LED3);
    SZ_STM32_LEDOff(LED4);
```

初始化串口，并让串口输出打印我们这次的实验目的

```
SZ_STM32_COMInit(COM2, 115200);

printf("ARMJISHU.COM-->窗口看门狗实验\n\r");
```

先是点亮我们的 LED1，延时一段时间

```
LED1OBB = 0;
delay(6000000);
```

对窗口看门狗进行一个配置，如寄存器、中断的配置函数

```
/*计数器值为7F, 窗口寄存器为5F, 分频数为8*/
WWDG_Init();
```

熄灭 LED1，在这里等待窗口看门狗的中断

```
while (1)
{
    LED1OBB = 1;
    /* 此处可以添加用户的程序 */
}
```

深入分析程序代码

- 使能时钟，窗口看门狗使用的是 PCLK1 的时钟，需要先使能时钟。方法是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE); // WWDG时钟使能
```

- 设置窗口值和分频数

设置窗口值的函数是：

```
WWDG_SetWindowValue(0x5f); //设置窗口值
```

这个函数就一个入口参数为窗口值

设置分频数的函数是：

```
WWDG_SetPrescaler(WWDG_Prescaler_8); //设置IWDG预分频值
```

这个函数同样只有一个入口参数就是分频值。

- 开启窗口看门狗中断

开启 WWDG 中断的函数为：

```
WWDG_EnableIT(); //开启窗口看门狗中断
```

- 窗口看门狗配置分组

配置 NVIC 中断优先级分组

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
```

中断 NVIC 配置，中断函数 WWDG_IRQHandler。

```
WWDG_NVIC_Init(); //初始化窗口看门狗 NVIC
```

```
void WWDG_NVIC_Init()
{
    NVIC_InitTypeDef NVIC_InitStructure;

    NVIC_InitStructure.NVIC_IRQChannel = WWDG_IRQn;      //WWDG中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure); //NVIC初始化
}
```

我们看下看门狗的寄存器配置是怎么样的，首先是使能看门狗，并设置一个计数值。在这个函数里面，是用一个 0x80 与上 0x7f，得到一个新的数值给到寄存器 WWDG_CR,使得第七位为 1，使能看门狗。

```
WWDG_Enable(WWDG_CNT); //使能看门狗，设置 counter .
```

```
void WWDG_Enable(uint8_t Counter)
{
    /* Check the parameters */
    assert_param(IS_WWDG_COUNTER(Counter));
    WWDG->CR = CR_WDGA_Set | Counter;
}
```

我们再来看下这个函数，它的主要作用是让窗口看门狗配置为中断模式，计数值递减到 0x40 的时候产生中断，只能是复位清除。

WWDG_EnableIT(); //开启窗口看门狗中断

```
void WWDG_EnableIT(void)
{
    *(__IO uint32_t *) CFR_EWI_BB = (uint32_t)ENABLE;
}
```

位9 EWI: 提前唤醒中断 (Early wakeup interrupt)
此位若置'1'，则当计数器值达到40h，即产生中断。
此中断只能由硬件在复位后清除。

我们下面看下中断函数是怎么样的，当看门狗计数值到 0x40 产生中断时会怎么样操作的，首先进入中断之后先是重新给寄存器重新载入计数值，然后清除中断标志位，清除完中断标志位后对 LED2 进行一个状态转换的功能，让串口输出打印一串字符。也就是说我们的窗口看门狗每一次执行中断就会点亮或者熄灭一次 LED，并打印一次数据

```
//窗口看门狗中断服务程序
void WWDG_IRQHandler(void)
{
    WWDG_SetCounter(0x7f); //当禁掉此句后，窗口看门狗将产生复位
    WWDG_ClearFlag(); //清除提前唤醒中断标志位
    LED2OBB = !LED2OBB; //LED状态翻转
    printf("ARMJISHU.COM-->喂狗\r\n");
}
```

这样就能看到我们前面做实验现象中，LED2 进行一个闪烁的效果了

7.23.9 实验现象

板子串口 2 连接串口线到电脑打印串口数据，为板子供电，观察 LED1 与 LED2 的状态

板子上电时，先会对 LED1 点亮，延时一段时间后熄灭，等待看门狗中断，运行中断函数，改变 LED2 的状态，产生一次中断改变一次，达到闪烁的效果。看门狗在计数器递减到 0x40 的时候产生中断，避免递减到 0x3f 的时候产生的复位信号。

下面为串口打印现象



先是打印串口初始化的信息，然后打印我们窗口看门狗中断函数里面的数据。

7.24 输入捕获实验

7.24.1 意义与作用

通过该实验，我们能够通过定时器捕获我们引脚上的高低电平脉冲宽度或者测量频率，对定时器有了一个更全面的认识与运用。在这里介绍了通用定时器作为输入捕获的使用，我们将用 TIM5 的通道 1 (PA0) 来做输入捕获，捕获 PA0 上低电平的脉宽（用 KEY4 按键输入低电平），通过串口打印低电平脉宽时间。

7.24.2 输入捕获介绍

输入捕获模式可以用来测量脉冲宽度或者测量频率。STM32 的定时器，除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。STM32 的输入捕获，简单的说就是通过检测 `TIMx_CHx` 上的边沿信号，在边沿信号发生跳变（比如上升沿/下降沿）的时候，将当前定时器的值（`TIMx_CNT`）存放到对应的通道的捕获/比较寄存器（`TIMx_CCRx`）里面，完成一次捕获。同时还可以配置捕获时是否触发中断/DMA 等。

7.24.3 实验原理

输入捕获就是用定时器 TIMx（除了 TIM6 和 TIM7）来捕获高低电平的脉宽，这里我们就也捕获通用定时器 TIM5 的高电平脉宽来讲解，也就是要先设置输入捕获为上升沿检测，记录发生上升沿的时候 TIM5_CNT 的值。然后配置捕获信号为下降沿捕获，当下降沿到来时，发生捕获，并记录此时的 TIM5_CNT 值。这样，前后两次 TIM5_CNT 之差，就是高电平的脉宽，同时 TIM5 的计数频率我们是知道的，从而可以计算出高电平脉宽的准确时间。

7.24.4 与输入捕获相关的寄存器

了解了输入捕获的基本原理后，我们来简要的分析下输入捕获将要用到的寄存器，在上面原理讲解的时候我们知道输入捕获就是用定时器来测量边沿的宽度或者频率，那么就会用到定时器，因此会

用到 TIMx_ARR、TIMx_PSC 这两个寄存器在此我就不做过多的说明相信看来我们前面知识的朋友都会明白。在通用定时器每个定时器都会有 4 个通道来进行输入捕获因此就会涉及到通道的选择所以会用到 TIMx_CCMR1 和 TIMx_CCMR2 这两个寄存器，应为本次实验只用到定时器 5 的通道 1 所以我们就只用到 TIMx_CCMR1 这个寄存器。我们需要用到中断来处理捕获数据，所以必须开启通道 1 的捕获比较中断，即 CC1IE 设置为 1 我们要将捕获计数器的值到捕获寄存器中就要用到 TIMx_CCER 寄存器，最后再来看看捕获/比较寄存器 1：TIMx_CCR1，该寄存器用来存储捕获发生时，TIMx_CNT 的值，我们从 TIMx_CCR1 就可以读出通道 1 捕获发生时刻的 TIMx_CNT 值，通过两次捕获（一次下降沿捕获，一次上升沿捕获）的差值，就可以计算出低电平脉冲的宽度。

下面我们在来看看各个寄存器各位的功能说明：

首先 TIMx_ARR 和 TIMx_PSC，这两个寄存器用来设自动重装载值和 TIMx 的时钟分频，用法同前面介绍的，我们这里不再介绍。

再来看看捕获/比较模式寄存器 1：TIMx_CCMR1，这个寄存器在此必须注意，同一个位在输出模式和输入模式下的功能是不同的。有必要重新介绍，该寄存器的各位描述如图所示：

| | | | | | | | | | | | | | | | |
|-----------|-------------|-----------|-------------|-----------|-------|-----------|-------|-------|-----------|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OC2CE | OC2M[2:0] | OC2PE | OC2FE | CC2S[1:0] | OC1CE | OC1M[2:0] | OC1PE | OC1FE | CC1S[1:0] | | | | | | |
| IC2F[3:0] | IC2PSC[1:0] | IC1F[3:0] | IC1PSC[1:0] | | | | | | | | | | | | |

| | |
|--------|--|
| 位15:12 | IC2F[3:0]: 输入捕获2滤波器 (Input capture 2 filter) |
| 位11:10 | IC2PSC[1:0]: 输入捕获2预分频器 (input capture 2 prescaler) |

| | |
|------|--|
| 位9:8 | CC2S[1:0]: 捕获/比较2选择 (Capture/compare 2 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00: CC2通道被配置为输出； 01: CC2通道被配置为输入，IC2映射在TI2上； 10: CC2通道被配置为输入，IC2映射在TI1上； 11: CC2通道被配置为输入，IC2映射在TRC上。此模式仅工作在内部触发器输入被选中时(由 TIMx_SMCR 寄存器的TS位选择)。 注：CC2S仅在通道关闭时(TIMx_CCER寄存器的CC2E='0')才是可写的。 |
| | |
| 位7:4 | IC1F[3:0]: 输入捕获1滤波器 (Input capture 1 filter) 这几位定义了TI1输入的采样频率及数字滤波器长度。数字滤波器由一个事件计数器组成，它记录到N个事件后会产生一个输出的跳变： 0000: 无滤波器，以 f_{DTS} 采样 0001: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=2 0010: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=4 0011: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=8 0100: 采样频率 $f_{SAMPLING}=f_{DTS}/2$, N=6 0101: 采样频率 $f_{SAMPLING}=f_{DTS}/2$, N=8 0110: 采样频率 $f_{SAMPLING}=f_{DTS}/4$, N=6 0111: 采样频率 $f_{SAMPLING}=f_{DTS}/4$, N=8 注：在现在的芯片版本中，当ICxF[3:0]=1、2或3时，公式中的 f_{DTS} 由 CK_INT 替代。 |
| | |
| 位3:2 | IC1PSC[1:0]: 输入/捕获1预分频器 (Input capture 1 prescaler) 这2位定义了CC1输入(IC1)的预分频系数。 一旦 $CC1E=0$ (TIMx_CCER寄存器中)，则预分频器复位。 00: 无预分频器，捕获输入口上检测到的每一个边沿都触发一次捕获； 01: 每2个事件触发一次捕获； 10: 每4个事件触发一次捕获； 11: 每8个事件触发一次捕获。 |
| | |

| | |
|------|---|
| 位1:0 | CC1S[1:0]: 捕获/比较1选择 (Capture/Compare 1 selection) 这2位定义通道的方向(输入/输出), 及输入脚的选择: 00: CC1通道被配置为输出; 01: CC1通道被配置为输入, IC1映射在TI1上; 10: CC1通道被配置为输入, IC1映射在TI2上; 11: CC1通道被配置为输入, IC1映射在TRC上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 <small>注: CC1S仅在通道关闭时(TIMx_CCER寄存器的CC1E='0')才是可写的。</small> |
|------|---|

当在输入捕获模式下使用的时候, 对应图的第二行描述, 从图中可以看出, TIMx_CCMR1 显然是针对 2 个通道的配置, 低八位[7: 0]用于捕获/比较通道 1 的控制, 而高八位[15: 8]则用于捕获/比较通道 2 的控制, 因为 TIMx 还有 CCMR2 这个寄存器, 所以可以知道 CCMR2 是用来控制通道 3 和通道 4。

在次我们用到的是定时器 5 的通道 1 我们重点介绍 TIMx_CMMR1 的[7:0]位 (其实高 8 位配置类似), TIMx_CMMR1 和 TIMx_CMMR2 也是大同小异的有兴趣的朋友可以去 STM32 手册上做深入了解。其中 CC1S[1:0], 这两个位用于 CCR1 的通道配置, 这里我们设置 IC1S[1:0]=01, 也就是配置 IC1 映射在 TI1 上关于 IC1, 即 CC1 对应 TIMx_CH1。输入捕获 1 预分频器 IC1PSC[1:0], 这个比较好理解。设置我们是 几 次边沿就触发 1 次捕获, 我们这里是 1 次边沿就触发 1 次捕获所以选择 00 就是了。输入捕获 1 滤波器 IC1F[3:0], 这个用来设置输入采样频率和数字滤波器长度。其中, fck_int 是定时器的输入频率 (TIMxCLK), 一般为 72Mhz, 而 Fdtc 则是根据 TIMx_CR1 的 CKD[1:0] 的设置来确定的, 如果 CKD[1:0]设置为 00, 那么 Fck_int=Fdtc。N 值就是滤波长度, 举个简单的例子: 假设 IC1F[3:0]=0011, 并设置 IC1 映射到通道 1 上, 且为上升沿触发, 那么在捕获到上升沿的时候, 再以 fck_int 的频率, 连续采样到 8 次通道 1 的电平, 如果都是高电平, 则说明却是一个有效的触发, 就会触发输入捕获中断 (如果开启了的话)。这样可以滤除那些高电平脉宽低于 8 个采样周期的脉冲信号, 从而达到滤波的效果。这里, 我们不做滤波处理, 所以设置 IC1F[3:0]=0000, 只要采集到上升沿, 就触发捕获。

再来看看捕获/比较使能寄存器: TIMx_CCER, 本章我们要用到这个寄存器的最低 2 位, CC1E 和 CC1P 位。这两个位的描述如图所示:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|------|------|-------|-------|------|------|-------|-------|------|------|-------|-------|------|------|----|
| 保留 | CC4P | CC4E | CC3NP | CC3NE | CC3P | CC3E | CC2NP | CC2NE | CC2P | CC2E | CC1NP | CC1NE | CC1P | CC1E | |
| | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

| | |
|----|--|
| 位1 | CC1P: 输入/捕获1输出极性 (Capture/Compare 1 output polarity) CC1通道配置为输出: 0: OC1高电平有效; 1: OC1低电平有效。 CC1通道配置为输入: 该位选择是IC1还是IC1的反相信号作为触发或捕获信号。 0: 不反相: 捕获发生在IC1的上升沿; 当用作外部触发器时, IC1不反相。 1: 反相: 捕获发生在IC1的下降沿; 当用作外部触发器时, IC1反相。 <small>注: 一旦LOCK级别(TIMx_BDTR寄存器中的LOCK位)设为3或2, 则该位不能被修改。</small> |
|----|--|

| | |
|----|---|
| 位0 | CC1E: 输入/捕获1输出使能 (Capture/Compare 1 output enable) CC1通道配置为输出: 0: 关闭— OC1禁止输出, 因此OC1的输出电平依赖于MOE、OSSI、OSSR、OIS1、OIS1N 和CC1NE位的值。 1: 开启— OC1信号输出到对应的输出引脚, 其输出电平依赖于MOE、OSSI、OSSR、OIS1、OIS1N 和CC1NE位的值。 CC1通道配置为输入: 该位决定了计数器的值是否能捕获入TIMx_CCR1寄存器。 0: 捕获禁止; 1: 捕获使能。 |
|----|---|

所以，要输入捕获使能位开启，必须设置 CC1E=1，而 CC1P 输入捕获 1 极性设置位用户可以根据自己的需要来配置（0：捕获到上升沿触发 1：捕获到下降沿触发）。

接下来我们再看看 DMA/中断使能寄存器：TIMx_DIER，我们需要用到中断来处理捕获数据，所以必须开启通道 1 的捕获比较中断，即 CC1IE 设置为 1。

| | | | | | | | | | | | | | | | |
|----|-----|----|-------|-------|-------|-------|-----|----|-----|----|-------|-------|-------|-------|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | TDE | 保留 | CC4DE | CC3DE | CC2DE | CC1DE | UDE | 保留 | TIE | 保留 | CC4IE | CC3IE | CC2IE | CC1IE | UIE |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

| | |
|----|---|
| 位1 | CC1IE: 允许捕获/比较1中断 (Capture/Compare 1 interrupt enable) |
| | 0: 禁止捕获/比较1中断; |
| | 1: 允许捕获/比较1中断。 |

当我们开启 CC1IE DMA/中断使能寄存器后将 CC1 配置为输出模式，当捕获事件发生时该位由硬件置 1，它由软件清零或通过读 TIM_CCR1 清零。

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|----|-------|----|-------|-------|-------|-------|-------|-------|-------|-------|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | CC4OF | CC3OF | CC2OF | CC1OF | 保留 | TIF | 保留 | CC4IF | CC3IF | CC2IF | CC1IF | UIF | | | |
| rc w0 | | rc w0 | | rc w0 |

| | |
|----|---|
| 位1 | CC1IF: 捕获/比较1 中断标记 (Capture/Compare 1 interrupt flag) |
| | 如果通道 CC1 配置为输出模式： |
| | 当计数器值与比较值匹配时该位由硬件置'1'，但在中心对称模式下除外(参考TIMx_CR1寄存器的CMS位)。它由软件清'0'。 |
| | 0: 无匹配发生； |
| | 1: TIMx_CNT 的值与TIMx_CCR1 的值匹配。 |
| | 如果通道 CC1 配置为输入模式： |
| | 当捕获事件发生时该位由硬件置'1'，它由软件清'0'或通过读TIMx_CCR1清'0'。 |
| | 0: 无输入捕获产生； |
| | 1: 计数器值已被捕获(拷贝)至TIMx_CCR1(在IC1上检测到与所选极性相同的边沿)。 |

控制寄存器：TIMx_CR1，我们只用到了它的最低位，也就是用它来使能定时器的，这里前面两章都有介绍，请大家参考前面的章节

| | | | | | | | | | | | | | | | |
|----|----------|------|----------|-----|-----|-----|------|-----|---|---|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | CKD[1:0] | ARPE | CMS[1:0] | DIR | OPM | URS | UDIS | CEN | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | | | | | | | rw |

| | |
|----|--|
| 位0 | CEN: 使能计数器 |
| | 0: 禁止计数器； |
| | 1: 使能计数器。 |
| | 注：在软件设置了CEN位后，外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 |
| | 在单脉冲模式下，当发生更新事件时，CEN被自动清除。 |

最后再来看看捕获/比较寄存器 1：TIMx_CCR1，该寄存器用来存储捕获发生时，TIMx_CNT 的值，我们从 TIMx_CCR1 就可以读出通道 1 捕获发生时刻的 TIMx_CNT 值，通过两次捕获（一次下降沿捕获，一次上升沿捕获）的差值，就可以计算出低电平脉冲的宽度。

| | | | | | | | | | | | | | | | | |
|------------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| CCR1[15:0] | | | | | | | | | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | |
| 位15:0 | | CCR1[15:0]: 捕获/比较1的值 (Capture/Compare 1 value) 若CC1通道配置为输出: CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载特性, 写入的数值会被立即传输至当前寄存器中。否则只有当更新事件发生时, 此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较, 并在OC1端口上产生输出信号。 若CC1通道配置为输入: CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。 | | | | | | | | | | | | | | |

至此, 我们把本章要用的几个相关寄存器都介绍完了, 本章要实现通过输入捕获, 来获取 TIM5_CH1(PA0)上面的高电平脉冲宽度, 并从串口打印捕获结果。

7.24.5 输入捕获配置的步骤

下面我们介绍输入捕获的配置步骤:

1) 开启 TIM5 时钟, 配置 PA0 为上拉输入。

首先我们要用到那个定时器就必须先使能它的时钟。这里要使用 TIM5, 我们必须先开启 TIM5 的时钟(通过 APB1ENR 设置)。这里我们还要配置 PA0 为上拉输入, 因为我们要捕获 TIM5_CH1 上面的低电平脉宽, 而 TIM5_CH1 是连接在 PA0 上面的。

2) 设置 TIM5 的 ARR 和 PSC。

在开启了 TIM5 的时钟之后, 我们要设置 ARR 和 PSC 两个寄存器的值来设置输入捕获的自动重装载值和计数频率(也就是初始化定时器, 这样我们才知道定时器的计数频率和定时器的溢出值)。

3) 设置 TIM5 的 CCMR1

TIM5_CCMR1 寄存器控制着输入捕获 1 和 2 的模式, 包括映射关系, 滤波和分频等。这里我们需要设置通道 1 为输入模式, 且 IC1 映射到 TI1(通道 1)上面, 并且不使用滤波(提高响应速度)器。

4) 设置 TIM5 的 CCER, 开启输入捕获, 并设置为上升沿捕获。

TIM5_CCER 寄存器是定时器的开关, 并且可以设置输入捕获的边沿。只有 TIM5_CCER 寄存器使能了输入捕获, 我们的外部信号, 才能被 TIM5 捕获到, 否则一切白搭。同时要设置好捕获边沿, 才能得到正确的结果。

5) 设置 TIM5 的 DIER, 使能捕获和更新中断, 并编写中断服务函数

因为我们要捕获的是高电平信号的脉宽, 所以, 第一次捕获是上升沿, 第二次捕获时下降沿, 必须在捕获上升沿之后, 设置捕获边沿为下降沿, 同时, 如果脉宽比较长, 那么定时器就会溢出, 对溢出必须做处理, 否则结果就不准了。这两件事, 我们都在中断里面做, 所以必须开启捕获中断和更新中断。设置了中断必须编写中断函数, 否则可能导致死机。我们需要在中断函数里面完成数据处理和捕获设置等关键操作, 从而实现高电平脉宽统计。

6) 设置 TIM5 的 CR1, 使能定时器

最后, 必须打开定时器的计数器开关, 通过设置 TIM5_CR1 的最低位为 1, 启动 TIM5 的计数器, 开始输入捕获。通过以上 6 步设置, 定时器 5 的通道 1 就可以开始输入捕获了, 同时因为还用到了串口输出结果, 所以还需要配置一下串口。

7.24.6 硬件设计

本次实验中, 我们只需要使用到 KEY4 的按键资源, 该按键接到处理器的 PA0 端口上, 是一个上嵌入式专业技术论坛 (www.armjishu.com) 出品

拉输入的按键，当按键按下去的时候，使得 PA0 得到一个低电平的电压，实现捕获低电平持续时间的测试实验，通过我们的串口线，把得到的数据传输到我们的串口工具上显示出来。

7.24.7 软件设计

开始的时候，先是定义了输入捕获的状态与捕获值，为下面的函数所使用

```
void SysTick_Handler_User(void);
extern uint8_t TIM5CH1_CAPTURE_STA;           //输入捕获状态
extern uint16_t TIM5CH1_CAPTURE_VAL;          //输入捕获值
```

主函数的时候，先是对 LED 进行了一个初始化

```
/*
 * @函数名 main
 * @功能 主函数
 * @参数 无
 * @返回值 无
 */
int main(void)
{
    uint32_t temp=0;
    /*!< 在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了时钟,
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
       SystemInit()函数的实现位于system_stm32f10x.c文件中。*/
    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);
```

初始化函数，分别对串口与定时器做了一个初始化，串口初始化完了后输出打印我们这次实验内容“ARMJISHU.COM-->输入捕获实验”其中红色方框上的数据是在串口初始化的时候串口输出的数据，而我们的蓝色方框的数据则是我们这句代码发出的数据。

```
SZ_STM32_COMInit(COM2, 115200);
printf("ARMJISHU.COM-->输入捕获实验\r\n");

TIM5_Init();
TIM3_PWM_Init(); //不分频。 PWM频率=72000/(899+1)=80Khz
```



大，所以我们的捕获时间精度为1us。

主函数通过 TIM5CH1_CAPTURE_STA 的第 7 位，来判断有没有成功捕获到一次低电平，如果成功捕获，则将高电平时间通过串口输出到电脑。

```
while (1)
{
    SysTick_Handler_User();
    delay(1200000);
    TIM_SetCompare2(TIM3, TIM_GetCapture2(TIM3)+1);
    if (TIM_GetCapture2(TIM3) == 300) TIM_SetCompare2(TIM3, 0);
    if (TIM5CH1_CAPTURE_STA&0x80)//成功捕获到了一次上升沿
    {
        temp = TIM5CH1_CAPTURE_STA&0x3F;
        temp *= 65536;//溢出时间总和
        temp += TIM5CH1_CAPTURE_VAL;//得到总的低电平时间
        printf ("HIGH: %d us\r\n", temp); //打印总的低电平时间
        TIM5CH1_CAPTURE_STA = 0;//开启下一次捕获
    }
    /* 此处可以添加用户的程序 */
}
```

详细分析代码

TIM5_Init 函数中，开启 **TIM5** 时钟和 **GPIOA** 时钟，配置 **PA0** 为上拉输入，要使用 TIM5，我们必须先开启 TIM5 的时钟。这里我们还要配置 PA0 为上拉输入，因为我们要捕获 TIM5_CH1 上面的低电平脉宽，而 TIM5_CH1 是连接在 PA0 上面的。

```
void TIM5_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* TIM5 clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能GPIOA时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //PA0 清除之前设置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //PA0 输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_ResetBits(GPIOA, GPIO_Pin_0); //PA0 上拉
}
```

使用定时器的时候需要对定时器进行一个配置使用，配置完后还要对我们的输入捕获进行一个参数的设置

```
/* Time base configuration */
//这个就是自动装载的计数值，由于计数是从0开始的，计数10000次后为9999
TIM_TimeBaseStructure.TIM_Period = 9999;
// 这个就是预分频系数，当由于为0时表示不分频所以要减1
TIM_TimeBaseStructure.TIM_Prescaler = (7200 - 1);
// 高级应用本次不涉及。定义在定时器时钟(CK_INT)频率与数字滤波器(ETR,TIX)
// 使用的采样频率之间的分频比例
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
//向上计数
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
//初始化定时器5
TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure);

/* Clear TIM5 update pending flag[清除TIM5溢出中断标志] */

    //初始化TIM5输入捕获参数
TIM5_ICInitStructure.TIM_Channel = TIM_Channel_1; //CC1S=01      选择输入端 IC1映射到TI1上
TIM5_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling; //下升沿捕获
TIM5_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //映射到TI1上
TIM5_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频
TIM5_ICInitStructure.TIM_ICFilter = 0x00; //IC1F=0000 配置输入滤波器 不滤波
TIM_ICInit(TIM5, &TIM5_ICInitStructure);
```

配置 TIM5 中断向量参数函数

```
/**-
 * @函数名 NVIC_TIM5Configuration
 * @功能 配置TIM5中断向量参数函数
 * @参数 无
 * @返回值 无
**/-
static void NVIC_TIM5Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Set the Vector Table base address at 0x08000000 */
    //NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0000);

    //中断分组初始化
    NVIC_InitStructure.NVIC_IRQChannel = TIM5_IRQn; //TIM3中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; //先占优先级2级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; //从优先级0级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ通道被使能
    NVIC_Init(&NVIC_InitStructure); //根据NVIC_InitStruct中指定的参数初始化外设NVIC寄存器

    TIM_ITConfig(TIM5, TIM_IT_Update|TIM_IT_CC1, ENABLE); //允许更新中断，允许CC1IE捕获中断
}
```

当定时器 TIM5 产生中断的时候，进行的处理函数，该函数中，因为我们要捕获的是低电平信号的脉宽，所以，第一次捕获是下降沿，第二次捕获时上升沿，必须在捕获下降沿之后，设置捕获边沿为上升沿，同时，如果脉宽比较长，那么定时器就会溢出，对溢出必须做处理，否则结果就不准了。这两件事，我们都在中断里面做，所以必须开启捕获中断和更新中断。

```
/*
 * @函数名 TIM5_IRQHandler
 * @功能   TIM5中断处理函数，每秒中断一次
 * @参数   无
 * @返回值 无
 */
uint8_t  TIM5CH1_CAPTURE_STA=0; //输入捕获状态
uint16_t    TIM5CH1_CAPTURE_VAL; //输入捕获值
void TIM5_IRQHandler(void)
{
    /* www.armjishu.com ARM技术论坛 */
    if((TIM5CH1_CAPTURE_STA&0X80)==0)//还未成功捕获
    {
        if (TIM_GetITStatus(TIM5, TIM_IT_Update) != RESET)

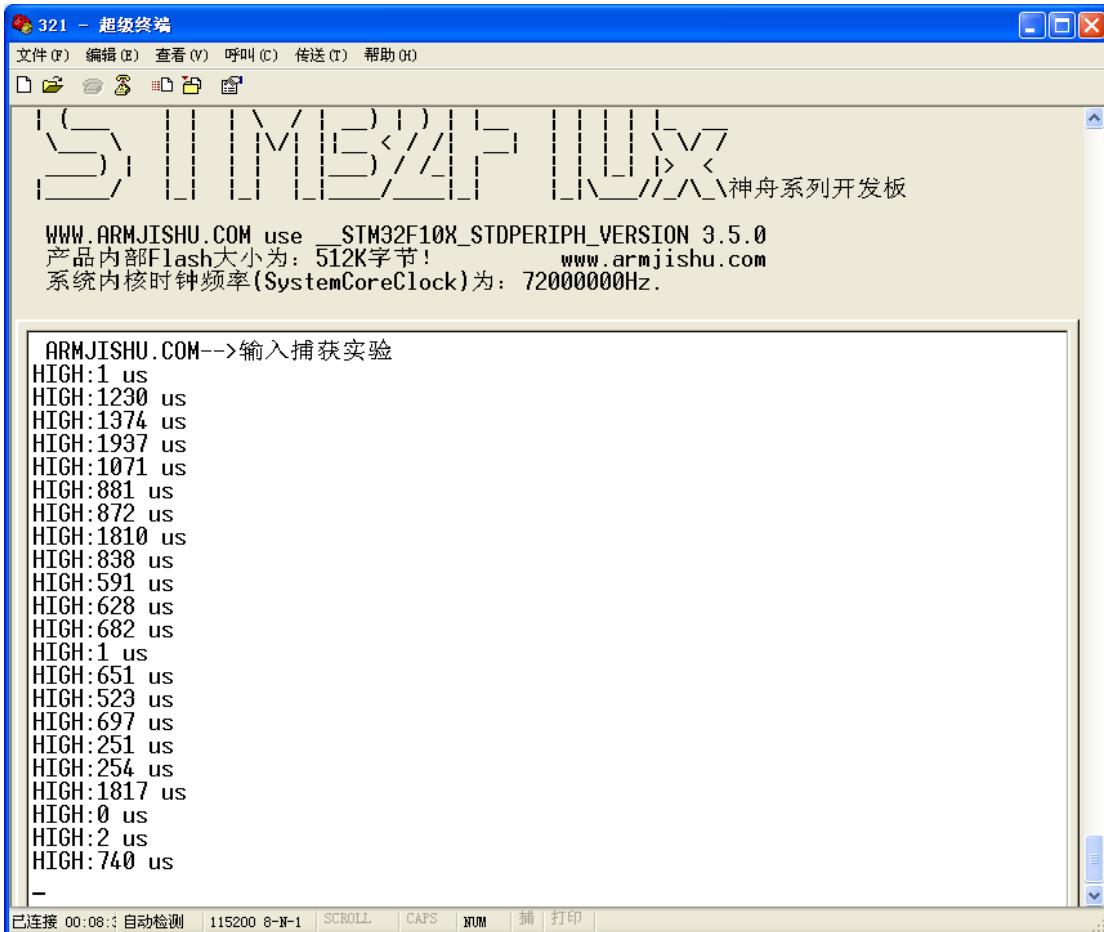
        {
            if (TIM5CH1_CAPTURE_STA&0X40)//已经捕获到低电平了
            {
                if ((TIM5CH1_CAPTURE_STA&0X3F)==0X3F)//低电平太长了
                {
                    TIM5CH1_CAPTURE_STA|=0X80;//标记成功捕获了一次
                    TIM5CH1_CAPTURE_VAL=0xFFFF;
                }else TIM5CH1_CAPTURE_STA++;
            }
        }

        if (TIM_GetITStatus(TIM5, TIM_IT_CC1) != RESET)//捕获1发生捕获事件
        {
            if (TIM5CH1_CAPTURE_STA&0X40)           //捕获到一个上升沿
            {
                TIM5CH1_CAPTURE_STA|=0X80;           //标记成功捕获到一次低电平脉宽
                TIM5CH1_CAPTURE_VAL=TIM_GetCapture1(TIM5);
                TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Falling); //CC1P=0 设置为下降沿捕获
            }else                                //还未开始,第一次捕获上升沿
            {
                TIM5CH1_CAPTURE_STA=0;             //清空
                TIM5CH1_CAPTURE_VAL=0;
                TIM_SetCounter(TIM5,0);
                TIM5CH1_CAPTURE_STA|=0X40;         //标记捕获到了上升沿
                TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Rising); //CC1P=1 设置为上升沿捕获
            }
        }
    }

    TIM_ClearITPendingBit(TIM5, TIM_IT_CC1|TIM_IT_Update); //清除中断标志位
}
```

7.24.8 实验现象

程序下载到我们的神舟 III 号开发板上后，连接我们的串口线到串口 2 上，波特率设置为 115200，具体串口设置可查看我们前面的串口介绍，复位后即可打印出我们串口初始化时的数据，然后通过 KEY4 按键得到低电平的脉宽时间，通过实验现象可看到捕获到的数据有 2 个是低于 10us 的，这种就是按键按下时发生的抖动。这就是为什么我们按键输入的时候，一般都需要做防抖处理，防止类似的情况干扰正常输入。



7.25 315M无线模块扫描实验

前面“按键检测”章节我们介绍了STM32的IO口作为输入功能的使用方法，本章节我们以**315M无线模块**扫描为例继续讲解IO相关知识，通过本节的学习，你将了解到STM32的IO口作为输入实现无线控制的功能。本节分为如下几个小节：

- 4.14.1 315M无线模块实验的意义与作用
- 4.14.2 实验原理
- 4.14.3 硬件设计
- 4.14.4 软件设计
- 4.14.5 下载与验证
- 4.14.6 实验现象

7.25.1 意义与作用

STM32的IO口在前面的流水灯实验、蜂鸣器实验和按键检测实验中已经有了详细的介绍，这一节我们讲结合STM32的库，描述如何设置STM32的GPIO口与315M无线模块的连接与使用。

这一节，我们将通过神舟 III 号板载的 315M 无线模块上 4 个按键，来控制板上的蜂鸣器和 4 个 LED (LED1~4)，按下任一个按钮，对应的 LED1~4 点亮，同时蜂鸣器会鸣响，还会在串口输出按键或无线控制的提示信息。

7.25.2 实验原理

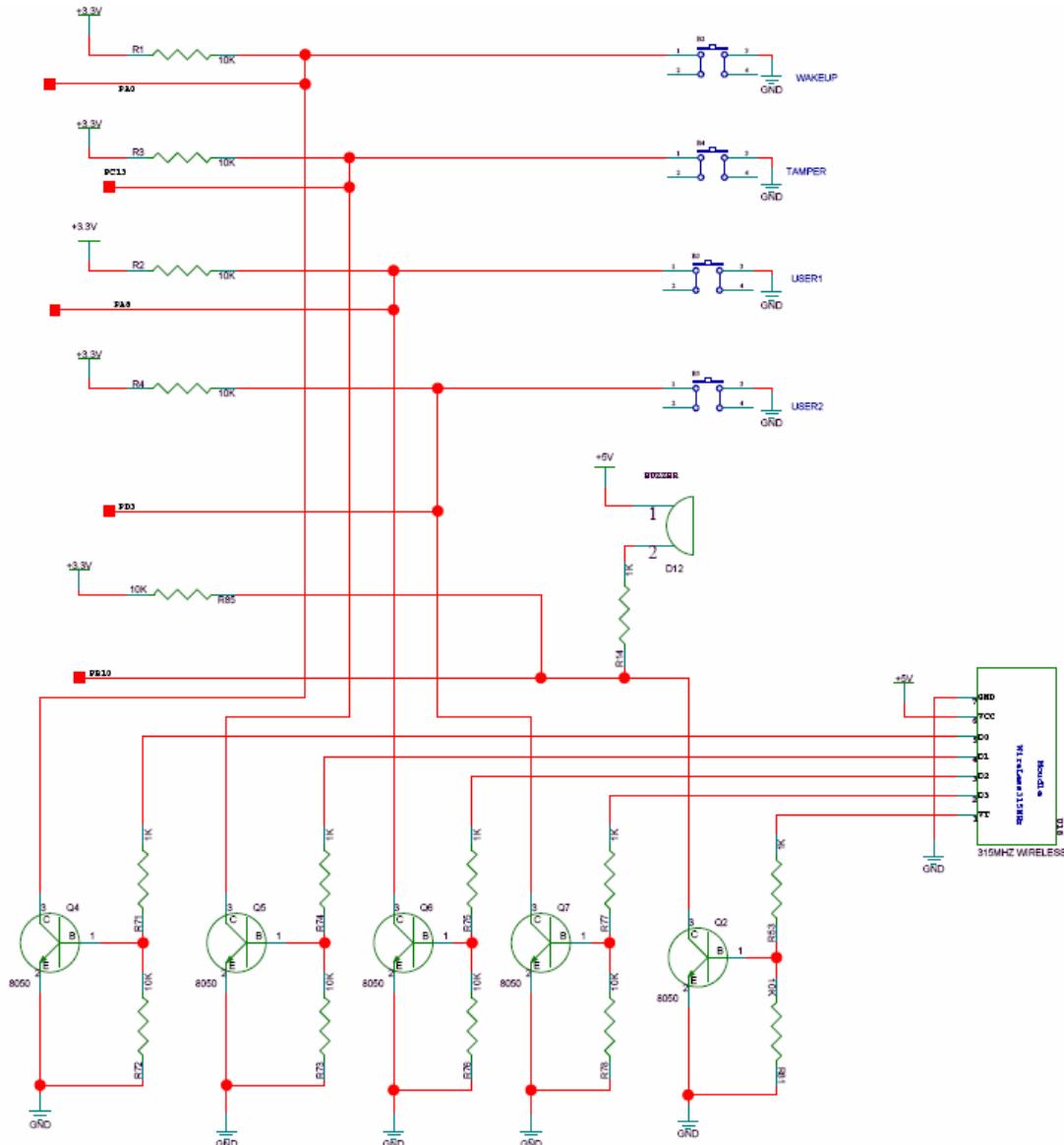
这个例程的实验原理主要是通过神舟III号上的315无线模块，接收315M无线遥控上的4个按钮（遥控上的A、B、C、D键），控制蜂鸣器和4个LED灯的点亮和关闭状态。

具体的对应关系如下：

| 现象 | 操作 | 备注 |
|-------------|--|------------|
| LED1亮其它LED灭 | 神舟III号板载KEY1按键被按下 或者315M无线遥控的按键A被按下 | 同时串口会有打印提示 |
| LED2亮其它LED灭 | 神舟III号板载KEY2按键被按下 或者315M无线遥控的按键C被按下 | |
| LED3亮其它LED灭 | 神舟III号板载KEY3/TAMPER按键被按下， 或者315M无线遥控的按键B被按下 | |
| LED4亮其它LED灭 | 神舟III号板载KEY4/WAKEUP按键被按下， 或者315M无线遥控的按键D被按下 | |
| 蜂鸣器鸣响 | 315M无线遥控的按键任意键被按下 | |

7.25.3 硬件设计

神舟III号开发板板载了315M无线模块，可以接受遥控器的信号，当遥控的一个按键按下时，对应的无线模块的D0~3管脚变为有效。需要指出的是，无线模块当输出为高电平有效，经过三极管放大反向以后并为低电平有效，再将这些送给神舟开发板的STM32。



图表 12 315M 无线接口原理图

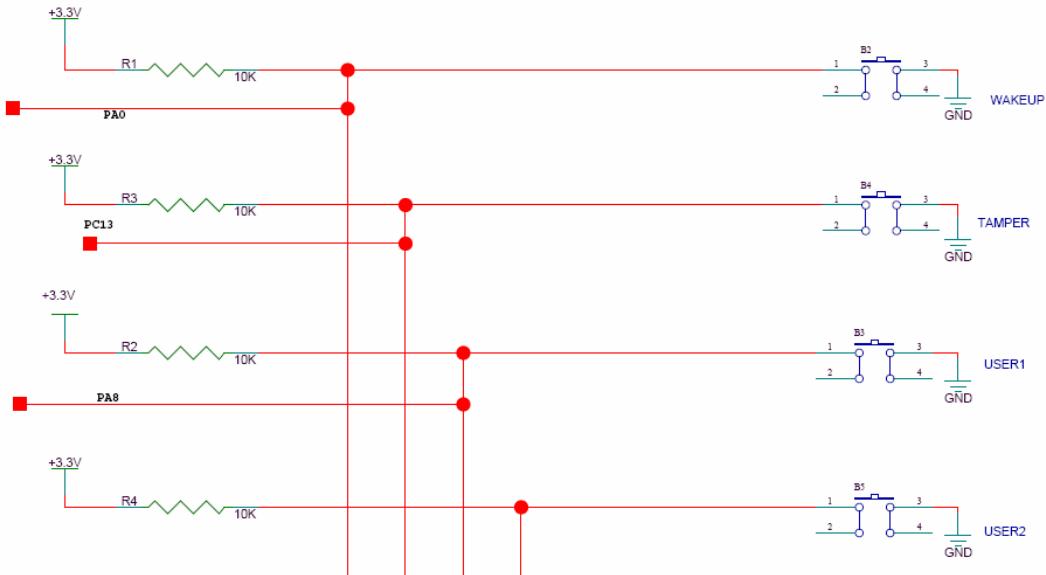
当无线模块的VT脚有效（低电平）时，表示无线模块接收到遥控的按键信号；当VT管脚无效（高电平）时，表示无线模块没有接收到遥控的按键信号，与无线模块连接的几个管脚的电平变化是按键引起的。

315M无线收发模块如下所示：



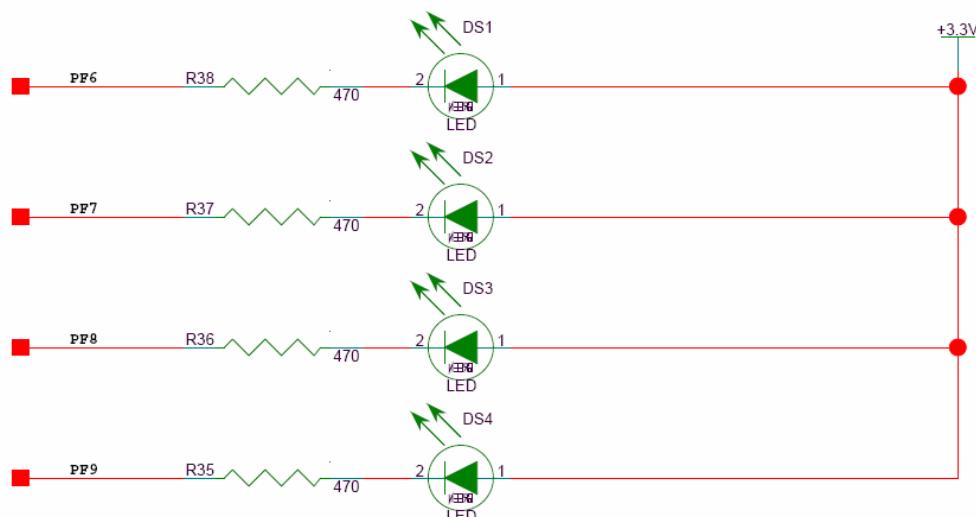
而神舟III号通过三极管将315M无线模块与板上的按键进行了资源复用。神舟III号STM32开发板总共有4个功能按键，分别是WAKEUP按键和TAMPER按键及两个用于自定义功能按键，在不使用第二功
嵌入式专业技术论坛 (www.armjishu.com) 出品 第 482 页，共 900 页

能的情况下,这四个按键都可以作为通用的按键,由用户自定义其功能。这四个按键分别与PC4、PB10、PC13和PA0四个GPIO管脚连接,当按键按下时,对应的GPIO管脚为低电平,反之,当没有按键按下时,对应的GPIO管脚为高电平。其中PA0 (STM32的WKUP引脚)可以作为WK_UP功能,它除了可以用作普通输入按键外,还可以用作STM32的唤醒输入。PC13可以实现备份区寄存器的入侵功能。本实验中所有的按键均作为普通IO使用。



图表 13 按键输入电路

该实验需要使用到神舟III号开发板上的LED灯,按键,相关硬件电路如下:



图表 14 LED 指示灯电路

GPIO 管脚与无线模块对应关系

| 无线模块 | 按键对应的GPIO |
|------|------------|
| D3 | PD3 |
| D2 | PA8 |
| D1 | PC13 |
| D0 | PA0 |
| VT | PB10 (蜂鸣器) |

GPIO 管脚与按键对应关系

| 按键 | 按键对应的GPIO |
|-------------|-----------|
| KEY1 | PA8 |
| KEY2 | PC3 |
| KEY3/TAMPER | PC13 |
| KEY4/WAKEUP | PA0 |

GPIO 管脚与对应的 LED 灯关系如下：

| LED灯 | LED灯对应的GPIO |
|------|-------------|
| LED1 | PD6 |
| LED2 | PD7 |
| LED3 | PD8 |
| LED4 | PD9 |

7.25.4 软件设计

在分析例程的代码之前，我们想回顾一下关于 STM32 GPIO 的使用。

STM32 GPIO 的使用与配置

STM32 的 IO 口可以由软件配置成 8 种模式：

- 模拟输入
- 输入浮空
- 输入下拉
- 输入上拉
- 开漏输出
- 推挽输出
- 复用功能开漏输出
- 复用功能推挽输出

对应到STM32库文件中的定义如下

```
typedef enum
{
    GPIO_Mode_AIN = 0x0,
    GPIO_Mode_IN_FLOATING = 0x04,
    GPIO_Mode_IPD = 0x28,
    GPIO_Mode_IPU = 0x48,
    GPIO_Mode_Out_OD = 0x14,
    GPIO_Mode_Out_PP = 0x10,
    GPIO_Mode_AF_OD = 0x1C,
    GPIO_Mode_AF_PP = 0x18
} GPIOMode_TypeDef;
```

在我们使用一个GPIO之前，我们一般需要对GPIO管脚的时钟，和GPIO管脚模式以及速率进行设定。

STM32的GPIO端口在作为输出时，可以软件配置端口最大支持的时钟速率，有以下几种：

- 输出模式，最大时钟速率10MHz
- 输出模式，最大时钟速率2MHz
- 输出模式，最大时钟速率50MHz

对应到STM32库中的定义如下：

```

typedef enum
{
    GPIO_Speed_10MHz = 1,
    GPIO_Speed_2MHz,
    GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;

```

速率主要针对GPIO作为输出时使用，作为输入时可以不关注。

神舟III号的“315M无线模块扫描”位于神舟III号光盘\编译好的固件\315M无线模块扫描目录中。

为程序便于大家修改和移植，此处使用宏定义来定义指示灯的管脚等：

```

#define LEDn          4
/** LED指示灯管脚资源定义 输出低电平点亮指示灯 ***/
#define LED1_PIN_NUM      6 /* bitband别名区使用宏定义 */
#define LED1_PIN          GPIO_Pin_6
#define LED1_GPIO_PORT    GPIOF
#define LED1_GPIO_CLK     RCC_APB2Periph_GPIOF
#define LED1OBB          Periph_BB((uint32_t)&LED1_GPIO_PORT->ODR, LED1_PIN_NUM)//等

#define LED2_PIN_NUM      7
#define LED2_PIN          GPIO_Pin_7
#define LED2_GPIO_PORT    GPIOF
#define LED2_GPIO_CLK     RCC_APB2Periph_GPIOF
#define LED2OBB          Periph_BB((uint32_t)&LED2_GPIO_PORT->ODR, LED2_PIN_NUM)

#define LED3_PIN_NUM      8
#define LED3_PIN          GPIO_Pin_8
#define LED3_GPIO_PORT    GPIOF
#define LED3_GPIO_CLK     RCC_APB2Periph_GPIOF
#define LED3OBB          Periph_BB((uint32_t)&LED3_GPIO_PORT->ODR, LED3_PIN_NUM)

#define LED4_PIN_NUM      9
#define LED4_PIN          GPIO_Pin_9
#define LED4_GPIO_PORT    GPIOF
#define LED4_GPIO_CLK     RCC_APB2Periph_GPIOF
#define LED4OBB          Periph_BB((uint32_t)&LED4_GPIO_PORT->ODR, LED4_PIN_NUM)

```

下面的LED_config函数是初始化神舟III号STM32开发板的4个LED灯对应的GPIO端口初始化的子函数。

```

void SZ_STM32_LEDInit(Led_TypeDef Led)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* Enable the GPIO LED Clock */
    /* 使能LED对应GPIO的Clock时钟 */
    RCC_APB2PeriphClockCmd(GPIO_CLK[Led], ENABLE);

    /* Configure the GPIO LED pin */
    /* 初始化LED的GPIO管脚，配置为推挽输出 */
    GPIO_InitStruct.GPIO_Pin = GPIO_PIN[Led];
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;

    GPIO_Init(GPIO_PORT[Led], &GPIO_InitStruct);
}

```

关于LED的其它函数请查看“LED跑马灯实验”章节，关于串口相关知识请“串口Printf输出实现”章节，此处主要介绍无线控制和按键相关程序。

为程序便于大家修改和移植，此处使用宏定义来定义按键和315无线模块对应的管脚：

```
#define BUTTONn          4
/** KEY按键管脚资源定义 按键按下时输入低电平 按键释放时输入高电平 **/
/** KEY1按键管脚 ***/
#define KEY1_BUTTON_PIN_NUM      8
#define KEY1_BUTTON_PIN          GPIO_Pin_8
#define KEY1_BUTTON_GPIO_PORT    GPIOA
#define KEY1_BUTTON_GPIO_CLK     RCC_APB2Periph_GPIOA
#define KEY1_BUTTON_EXTI_LINE    EXTI_Line8
#define KEY1_BUTTON_EXTI_PORT_SOURCE  GPIO_PortSourceGPIOA
#define KEY1_BUTTON_EXTI_PIN_SOURCE  GPIO_PinSource8
#define KEY1_BUTTON_EXTI_IRQn    EXTI9_5_IRQn
#define KEY1IBB                  Periph_BB((uint32_t)&KEY1_BUTTON_GPIO_PORT->IDR, KEY1_BUTTON_PIN_NUM) /

/** KEY2按键管脚 ***/
#define KEY2_BUTTON_PIN_NUM      3
#define KEY2_BUTTON_PIN          GPIO_Pin_3
#define KEY2_BUTTON_GPIO_PORT    GPIOD
#define KEY2_BUTTON_GPIO_CLK     RCC_APB2Periph_GPIOD
#define KEY2_BUTTON_EXTI_LINE    EXTI_Line3
#define KEY2_BUTTON_EXTI_PORT_SOURCE  GPIO_PortSourceGPIOD
#define KEY2_BUTTON_EXTI_PIN_SOURCE  GPIO_PinSource3
#define KEY2_BUTTON_EXTI_IRQn    EXTI3_IRQn
#define KEY2IBB                  Periph_BB((uint32_t)&KEY2_BUTTON_GPIO_PORT->IDR, KEY2_BUTTON_PIN_NUM)

/** KEY3按键同时也是Tamper管脚 ***/
#define KEY3_BUTTON_PIN_NUM      13
#define KEY3_BUTTON_PIN          GPIO_Pin_13
#define KEY3_BUTTON_GPIO_PORT    GPIOC
#define KEY3_BUTTON_GPIO_CLK     RCC_APB2Periph_GPIOC
#define KEY3_BUTTON_EXTI_LINE    EXTI_Line13
#define KEY3_BUTTON_EXTI_PORT_SOURCE  GPIO_PortSourceGPIOC
#define KEY3_BUTTON_EXTI_PIN_SOURCE  GPIO_PinSource13
#define KEY3_BUTTON_EXTI_IRQn    EXTI15_10_IRQn
#define KEY3IBB                  Periph_BB((uint32_t)&KEY3_BUTTON_GPIO_PORT->IDR, KEY3_BUTTON_PIN_NUM)

/** KEY4按键同时也是Wakeup管脚 ***/
#define KEY4_BUTTON_PIN_NUM      0
#define KEY4_BUTTON_PIN          GPIO_Pin_0
#define KEY4_BUTTON_GPIO_PORT    GPIOA
#define KEY4_BUTTON_GPIO_CLK     RCC_APB2Periph_GPIOA
#define KEY4_BUTTON_EXTI_LINE    EXTI_Line0
#define KEY4_BUTTON_EXTI_PORT_SOURCE  GPIO_PortSourceGPIOA
#define KEY4_BUTTON_EXTI_PIN_SOURCE  GPIO_PinSource0
#define KEY4_BUTTON_EXTI_IRQn    EXTI0_IRQn
#define KEY4IBB                  Periph_BB((uint32_t)&KEY4_BUTTON_GPIO_PORT->IDR, KEY4_BUTTON_PIN_NUM)
```

神舟III号按键使用的GPIO的接口初始化，由于按键或无线模块上的按键按下时会使相应的GPIO口变为低电平，因此，在这里我们配置按键使用的GPIO为输入上拉模式。没有收到有效信号，保持这些GPIO口为高电平。

```
void SZ_STM32_KEYInit(Button_TypeDef Button, ButtonMode_TypeDef Button_Mode)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable the BUTTON Clock */
    /* 使能KEY按键对应的Clock时钟 */
    RCC_APB2PeriphClockCmd(BUTTON_CLK[Button] | RCC_APB2Periph_AFIO, ENABLE);

    /* Configure Button pin as input floating */
    /* 初始化KEY按键的GPIO管脚，配置为带上拉的输入 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = BUTTON_PIN[Button];
    GPIO_Init(BUTTON_PORT[Button], &GPIO_InitStructure);

    /* Initialize KEY按键为中断模式 */
    if (Button_Mode == BUTTON_MODE_EXTI)
    {
        /* Connect Button EXTI Line to Button GPIO Pin */
        /* 将KEY按键对应的管脚连接到内部中断线 */
        GPIO_EXTILineConfig(BUTTON_PORT_SOURCE[Button], BUTTON_PIN_SOURCE[Button]);

        /* Configure Button EXTI line */
        /* 将KEY按键配置为中断模式，下降沿触发中断 */
        EXTI_InitStructure EXTI_Line = BUTTON_EXTI_LINE[Button];
        EXTI_InitStructure EXTI_Mode = EXTI_Mode_Interrupt;
        EXTI_InitStructure EXTI_Trigger = EXTI_Trigger_Falling;
        EXTI_InitStructure EXTI_LineCmd = ENABLE;
        EXTI_Init(&EXTI_InitStructure);

        /* Enable and set Button EXTI Interrupt to the lowest priority */
        /* 将KEY按键的中断优先级配置为最低 */
        NVIC_InitStructure.NVIC_IRQChannel = BUTTON IRQn[Button];
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F;
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
        NVIC_Init(&NVIC_InitStructure);
    }
}
```

无线遥控按键和开发板上轻触开关检测函数，当有无线遥控按键或开发板上轻触开关按下时，函数返回对应的键值。

```
uint32_t SZ_STM32_KEYScan(void)
{
    /* 获取KEY按键的输入电平状态，按键按下时为低电平 */
    if(O == KEY1IBB)
    {
        /* 延迟去抖 */
        delay(150000);
        if(O == KEY1IBB)
        {
            return 1;
        }
    }
    /* 获取KEY按键的输入电平状态，按键按下时为低电平 */
    if(O == KEY2IBB)
    {
        /* 延迟去抖 */
        delay(150000);
        if(O == KEY2IBB)
        {
            return 2;
        }
    }
    /* 获取KEY按键的输入电平状态，按键按下时为低电平 */
    if(O == KEY3IBB)
    {
        /* 延迟去抖 */
        delay(150000);
        if(O == KEY3IBB)
        {
            return 3;
        }
    }
    /* 获取KEY按键的输入电平状态，按键按下时为低电平 */
    if(O == KEY4IBB)
    {
        /* 延迟去抖 */
        delay(150000);
        if(O == KEY4IBB)
        {
            return 4;
        }
    }
    return 0;
}
```

说明：在本例程中，由于蜂鸣器和315M无线接收模块的VT使用了相同的GPIO管脚，只要无线遥控上的任意按键按下，无线接收模块的VT信号就为低电平，经过三极管放大取反后的低电平被Check_315M_Wireless程序检测到，同时驱动蜂鸣器鸣响。

介绍完基本的函数，我们再来分析一下main主程序：

```
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
    */
    /*!< 在系统启动文件 (startup_stm32f10x_xx.s) 中已经调用SystemInit()初始化了时钟,
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
       SystemInit() 函数的实现位于system_stm32f10x.c文件中。
    */
    uint8_t KeyNum = 0;

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    /* 初始化板载按键 */
    SZ_STM32_KEYInit(KEY1, BUTTON_MODE_GPIO);
    SZ_STM32_KEYInit(KEY2, BUTTON_MODE_GPIO);
    SZ_STM32_KEYInit(KEY3, BUTTON_MODE_GPIO);
    SZ_STM32_KEYInit(KEY4, BUTTON_MODE_GPIO);
    W315M_VTConfig();

    /* 注意串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为sz_STM32_CC
     * 串口2初始化 */
    SZ_STM32_COMInit(COM2, 115200);

    printf("\n\r www.armjishu.com STM32神舟开发板315M无线实验");
    printf("\n\r ======\r\n");

    /* Infinite loop 主循环 */
    while (1)
    {
        /* 315M无线接收模块的VT信号被三极管电路取反后为低说明315M有按键按下 */
        if (!W315M_VT)
        {
            以上主程序首先显示初始化串口，打印提示信息。然后是初始化LED和无线按键的GPIO，如下：

        /* 初始化板载LED指示灯 */
        SZ_STM32_LEDInit(LED1);
        SZ_STM32_LEDInit(LED2);
        SZ_STM32_LEDInit(LED3);
        SZ_STM32_LEDInit(LED4);

        /* 初始化板载按键 */
        SZ_STM32_KEYInit(KEY1, BUTTON_MODE_GPIO);
        SZ_STM32_KEYInit(KEY2, BUTTON_MODE_GPIO);
        SZ_STM32_KEYInit(KEY3, BUTTON_MODE_GPIO);
        SZ_STM32_KEYInit(KEY4, BUTTON_MODE_GPIO);
        W315M_VTConfig();

        /* 注意串口2使用Printf时"SZ_STM32F103ZE_LIB.c"文件中fputc定义中设备改为sz_STM32_COM2 */
        /* 串口2初始化 */
        SZ_STM32_COMInit(COM2, 115200);
    }
}
```

最后主程序进入while死循环，不停的检测来自无线模块的控制信号和按键的状态信息。程序按以下顺序执行

```
/* 315M无线接收模块的VT信号被三极管电路取反后为低说明315M有按键按下 */
if(!W315M_VT)
{
    do{
        /* 延迟，去抖 */
        delay(200000);
        /* 315M无线接收模块的VT信号被三极管电路取反后为高说明315M没有按键按下 */
        if(W315M_VT)
        {
            break;
        }
        printf("\n\r ^_^ WWW.ARmjishu.COM ^_^ "
               "Get signal from The 315M wireless Module!");

        KeyNum = SZ_STM32_KEYScan();
        if(0 == KeyNum)
        {
            printf("\n\r WWW.ARmjishu.COM Probably "
                   "315M wireless Hardware is install Error,");
            printf("\n\r Please check The 315M wireless Hardware!\n\r");
            while(!W315M_VT);
        }
        else
        {
            Led_Turn_on(KeyNum);
            printf("\n\r WWW.ARmjishu.COM 315M wireless "
                   "Key %d is Pressed! \n\r", KeyNum);
            /* 等待315M无线按键释放 */
            while(W315M_VT);
            printf("\n\r WWW.ARmjishu.COM 315M wireless "
                   "Key %d is Free! \n\r", KeyNum);
            /* 延迟，去抖 */
            delay(200000);
        }
    }while(0);
}

else
{
    KeyNum = SZ_STM32_KEYScan();
    if(0 != KeyNum)
    {
        /* 延迟，去抖 */
        delay(200000);

        KeyNum = SZ_STM32_KEYScan();
        if(0 != KeyNum)
        {
            Led_Turn_on(KeyNum);
            printf("\n\r Key %d On board is Pressed! \n\r", KeyNum);
            while(SZ_STM32_KEYScan() == KeyNum);
            printf("\n\r Key %d On board is Free! \n\r", KeyNum);
            /* 延迟，去抖 */
            delay(200000);
        }
    }
}
```

说明：在本例程中，由于按键和315M无线模块使用了相同的GPIO管脚，我们主要是利用前面提到的方法二，即蜂鸣器是否发声来判断具体是按键（开发板上的轻触开关）被按下还是无线模块接收到遥控信号。

7.25.5 下载与验证

如果使用JLINK下载固件，请按3.3如何使用JLINK软件下载固件到神舟III号开发板小节进行操作
嵌入式专业技术论坛（www.armjishu.com）出品

作。

如果使用USB下载固件，请按3.5如何通过USB接口下载固件到神舟III号开发板小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.8如何在MDK开发环境中使用JLINK在线调试小节进行操作。

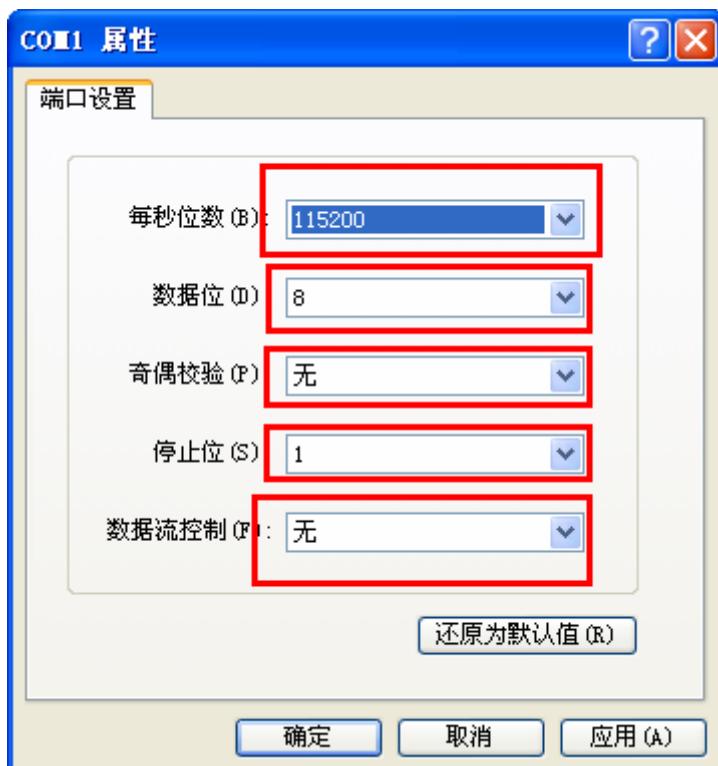
7.25.6 实验现象

下载固件后上电运行。遥控有按键按下时蜂鸣器会鸣响，对应的LED灯会随之发生点亮，同时串口会有提示信息；在开发板上的轻触开关被按下时蜂鸣器不会鸣响，对应的LED灯会随之发生点亮，同时串口会有提示信息。

用串口线神舟 III 号串口 2 与电脑连接，并打开超级终端，按以下设置，如下图：

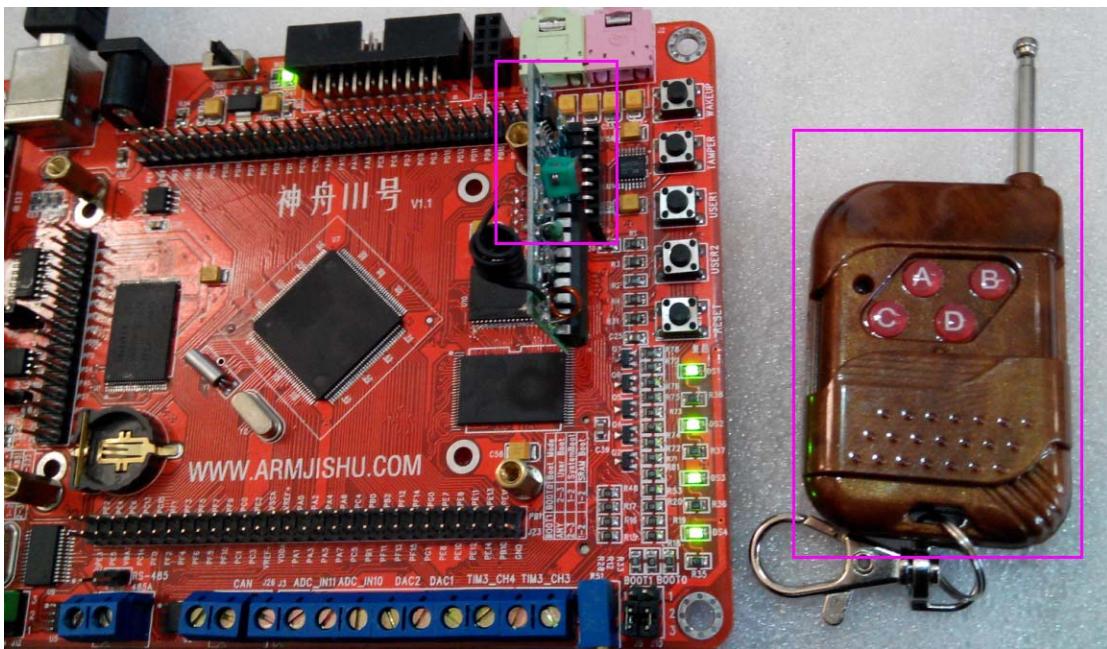


选择 COM1；按确定

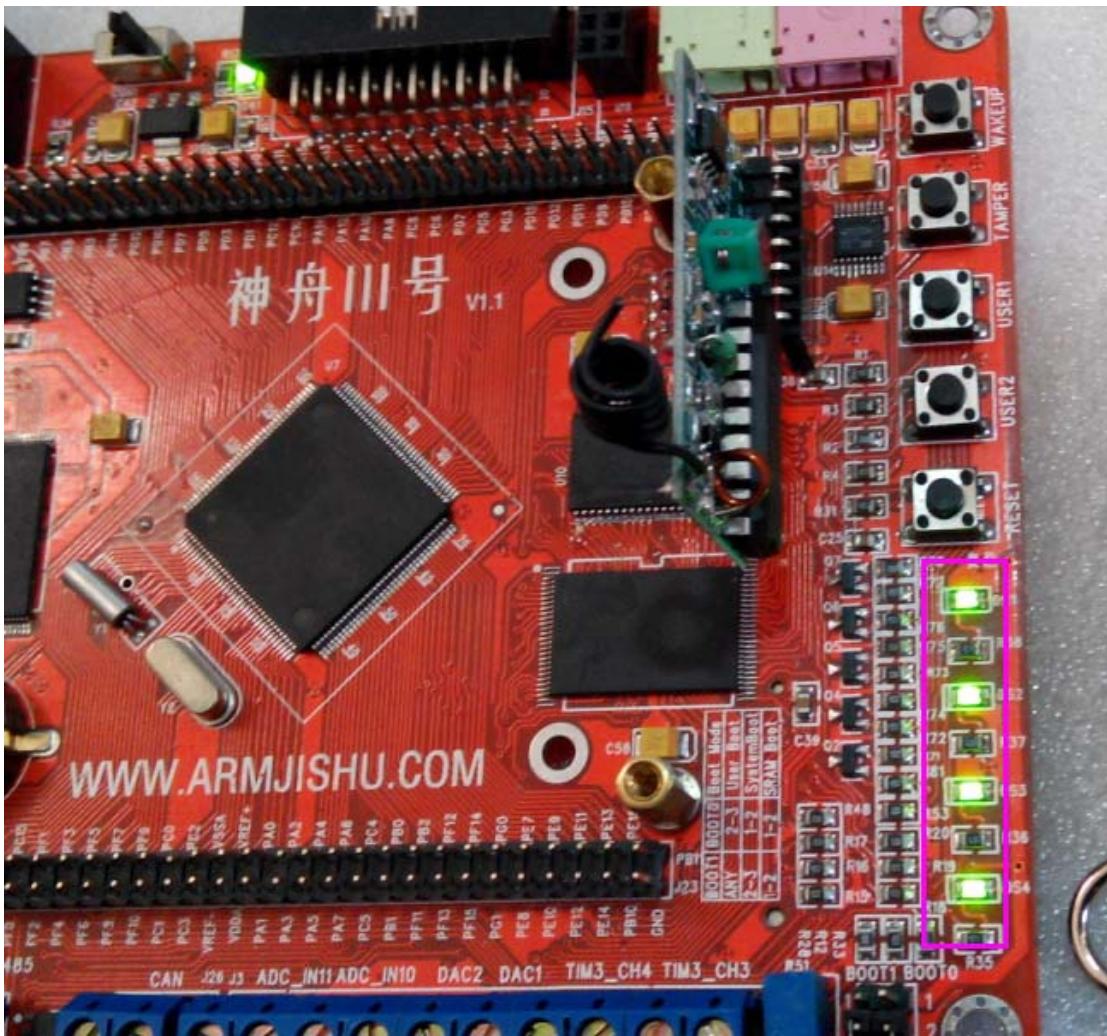


再按确定，完成超级终端设置。

外设器件：315M 无线模块，315M 无线遥控器；将 315M 无线模块插在神舟 III 号开发板上，



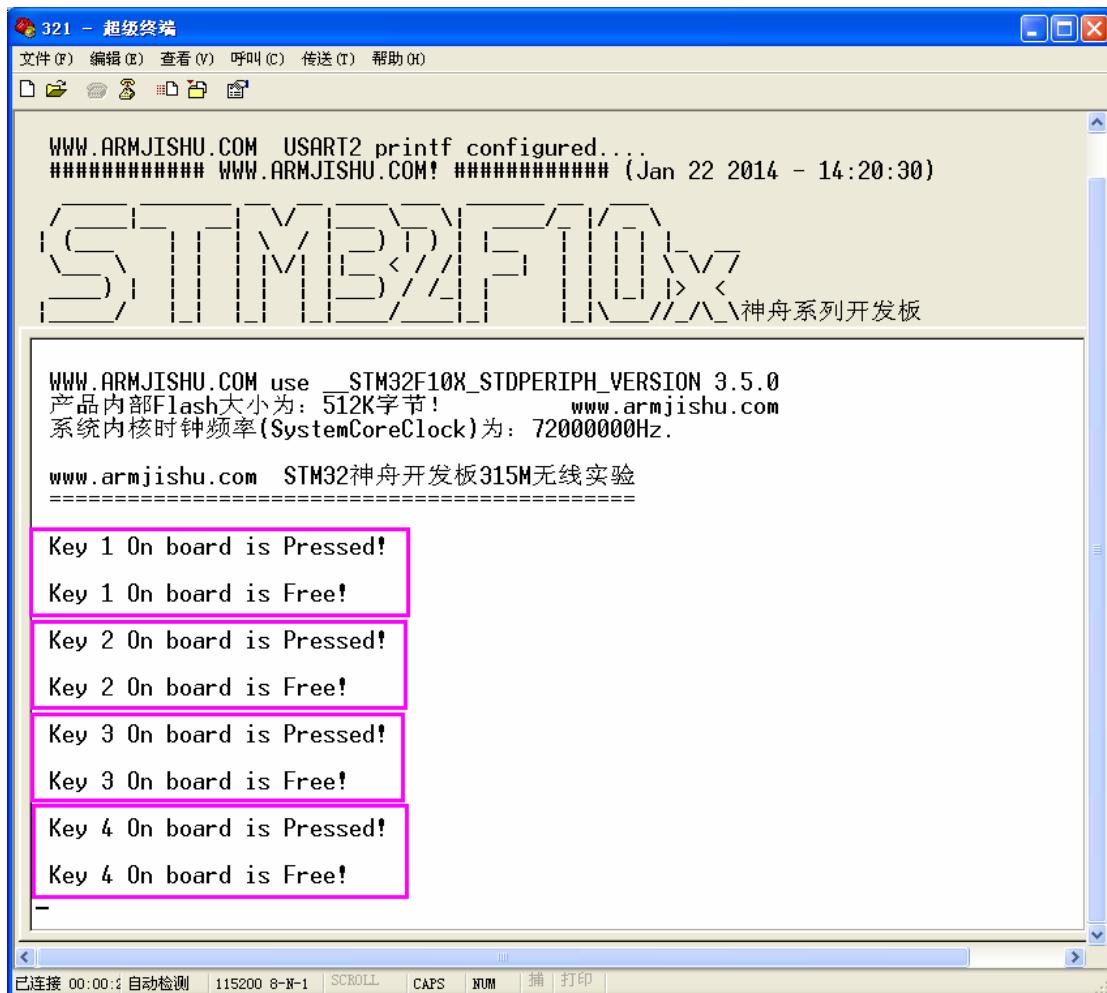
重新打开电源；可以看到神舟 III 号开发板上的 4 个 LED 灯（LED1——LED4）都亮，



315M 遥控器按键和神舟 III 号板按键操作说明，以下表：

| LED 灯现象 | 按键操作 |
|------------------|--------------------------------------|
| LED1 亮, 其它 LED 灭 | 神舟 III 号板 KEY1 按下或 315M 无线遥控的按键 A 按下 |
| LED2 亮, 其它 LED 灭 | 神舟 III 号板 KEY2 按下或 315M 无线遥控的按键 C 按下 |
| LED3 亮, 其它 LED 灭 | 神舟 III 号板 KEY3 按下或 315M 无线遥控的按键 D 按下 |
| LED4 亮, 其它 LED 灭 | 神舟 III 号板 KEY4 按下或 315M 无线遥控的按键 B 按下 |
| 蜂鸣器响 | 315M 无线遥控按任意按键按下 |

如用 315M 无线遥控器按下：A、B、C、D 按键；超级终端窗口显示信息，以下图：



7.26 EXTI无线315M模块外部中断实验

前面我们介绍了STM32的IO口作为输出输入功能的使用并介绍了我们以按键的中断扫描实验，这一节，我们将向大家介绍如何使用STM32的IO口作为**EXTI外部中断**的输入功能，我们以**315M无线模块**为例讲解，通过本节的学习，你将了解到STM32的IO口作为**EXTI外部中断**输入实现无线控制的功能的方法。本节分为如下几个小节：

- 1 EXTI无线315M模块外部中断实验的意义与作用
- 2 实验原理
- 3 硬件设计
- 4 软件设计
- 5 下载与验证
- 6 实验现象

7.26.1 意义与作用

STM32的IO口在前面的流水灯实验和按键的输入扫描实验中已经有了详细的介绍，这里我们细讲，这一节我们将向大家介绍如何结合STM32的库使STM32的GPIO口作为**EXTI外部中断**的输入功能使用。

Cortex-M3的NVIC（嵌套向量中断控制器）是一大特色，而STM32的所有GPIO管脚都可以作为中断输入源，所以掌握使STM32的GPIO口作为**EXTI外部中断**的硬件与软件设计思想对理解其NVIC有很大帮助。

本实验章节，我们将通过 **315M 无线发射遥控**控制神舟 III 号板的 **315M 无线接收模块**产生中断来控制串口打印信息，同时蜂鸣器会鸣响。

7.26.2 实验原理

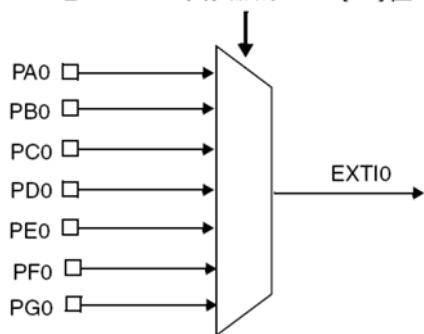
这个例程的实验原理主要是通过神舟III号上的315无线模块，接收315M无线遥控上的4个按钮（遥控上的A、B、C、D键）并产生中断，控制串口打印信息，同时蜂鸣器会鸣响。

具体的对应关系如下：

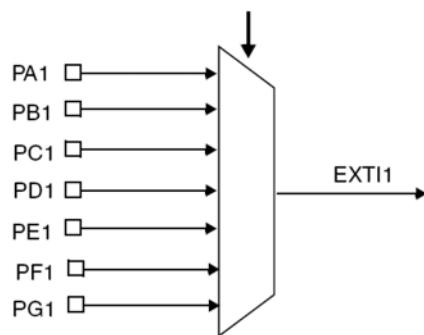
| 现象 | 操作 | 备注 |
|-------|-------------------|------------|
| 蜂鸣器鸣响 | 315M无线遥控的按键A被按下 | 同时串口会有打印提示 |
| 蜂鸣器鸣响 | 315M无线遥控的按键B被按下 | |
| 蜂鸣器鸣响 | 315M无线遥控的按键C被按下 | |
| 蜂鸣器鸣响 | 315M无线遥控的按键D被按下 | |
| 蜂鸣器鸣响 | 315M无线遥控的按键任意键被按下 | |

STM32F103ZET一共有5组GPIO，分别是PA[15:0]、PB[15:0]、PC[15:0]、PD[15:0] 和PE[15:0]。STM32的所有GPIO管脚都可以作为中断输入源，但是如果每个GPIO都是一个独立的中断源，则需要90个中断源，这是不科学的，所以**通过复用的方式**使其对处理器来说来自GPIO的一共有16个中断Px[15:0]。具体实现是PA[0]、PB[0]、PC[0]、PD[0] 和PE[0]共享一个GPIO中断；PA[1]、PB[1]、PC[1]、PD[1] 和PE[1]共享一个GPIO中断；……PA[15]、PB[15]、PC[15]、PD[15] 和PE[15]共享一个GPIO中断，如下图所示：

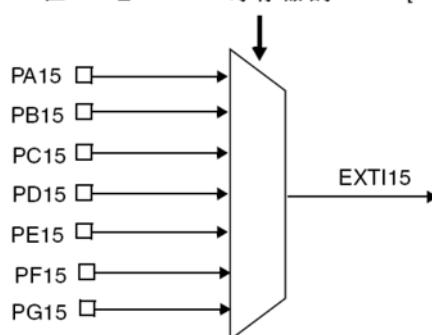
在AFIO_EXTICR1 寄存器的EXTI0[3:0]位



在AFIO_EXTICR1 寄存器的EXTI1[3:0]位



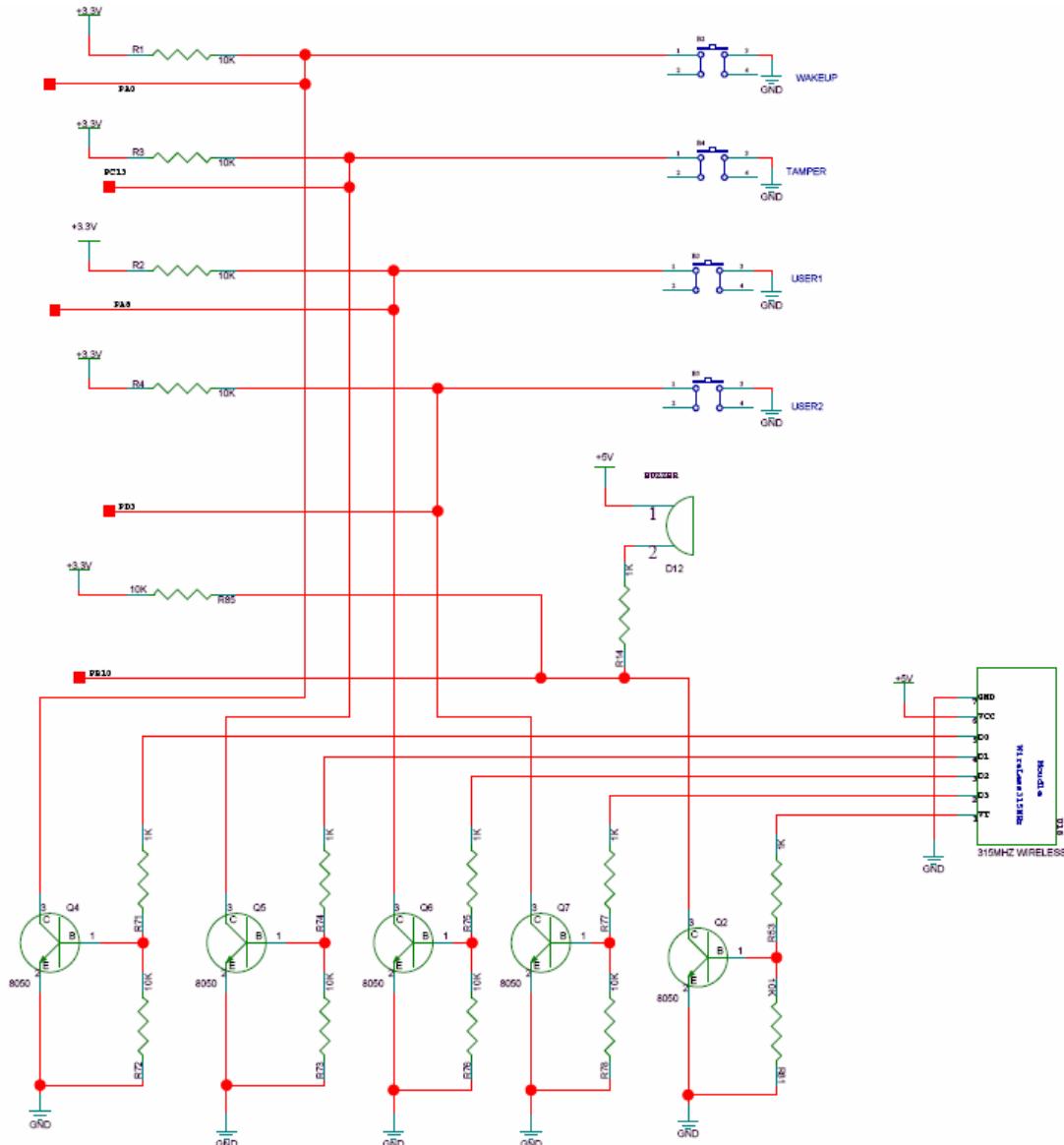
在AFIO_EXTICR4 寄存器的EXTI15[3:0]位



所以在硬件设计时要注意，不要将外部中断连接到PA[1]、PB[1]、PC[1]和PE[1]，这样的话处理器只能选择一个作为中断源比如PA[1]，那么其它中断将无法到达处理器。

7.26.3 硬件设计

神舟III号开发板板载了315M无线模块，可以接受遥控器的信号，当遥控的一个按键按下时，对应的无线模块的D0~3管脚变为有效。需要指出的是，无线模块当输出为高电平有效，经过三极管放大取反以后并为低电平有效，再将这些送给神舟开发板的STM32。



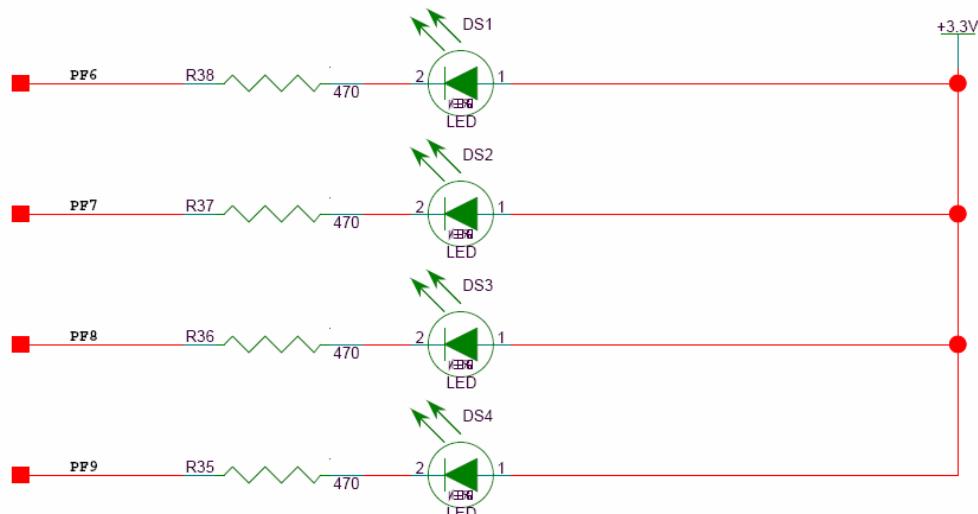
图表 15 315M 无线接口原理图

当无线模块的VT脚有效（低电平）时，表示无线模块接收到遥控的按键信号；当VT管脚无效（高电平）时，表示无线模块没有接收到遥控的按键信号，与无线模块连接的几个管脚的电平变化是按键引起的。

315M无线收发模块如下所示：



该实验需要使用到神舟III号开发板上的LED灯，按键，相关硬件电路如下：



图表 16 LED 指示灯电路

GPIO 管脚与无线模块对应关系

| 无线模块 | 按键对应的GPIO |
|------|------------|
| D3 | PD3 |
| D2 | PA8 |
| D1 | PC13 |
| D0 | PA0 |
| VT | PB10 (蜂鸣器) |

GPIO 管脚与对应的 LED 灯关系如下：

| LED灯 | LED灯对应的GPIO |
|------|-------------|
| LED1 | PD6 |
| LED2 | PD7 |
| LED3 | PD8 |
| LED4 | PD9 |

7.26.4 软件设计

我们从主函数开始分析：

```
020 int main(void)
021 {
022     /* 初始化板载LED指示灯 */
023     SZ_STM32_LEDInit(LED1);
024     SZ_STM32_LEDInit(LED2);
025     SZ_STM32_LEDInit(LED3);
026     SZ_STM32_LEDInit(LED4);
027
028     /* 初始化板载按键 */
029     SZ_STM32_KEYInit(KEY1, BUTTON_MODE_GPIO);
030     SZ_STM32_KEYInit(KEY2, BUTTON_MODE_GPIO);
031     SZ_STM32_KEYInit(KEY3, BUTTON_MODE_GPIO);
032     SZ_STM32_KEYInit(KEY4, BUTTON_MODE_GPIO);
033
034     W315M_VTConfig();
035     W315M_EXIT_VTConfig();
036
037     /* 串口2初始化 */
038     SZ_STM32_COMInit(COM1, 115200);
039
040     printf("\n\r www.armjishu.com STM32神舟开发板315M无线实验");
041     printf("\n\r ======\r\n");
042
043     /* Infinite loop 主循环 */
044     while (1)
```

函数中首先，通过 SZ_STM32_LEDInit() 函数初始化 LED 灯。通过函数 SZ_STM32_KEYInit() 初始化按键。这些前面都有涉及，不知这里重复。

代码分析 1：W315M_VTConfig() 函数初始化，315M 的 VT 引脚。

```
060 void W315M_VTConfig(void)
061 {
062     GPIO_InitTypeDef GPIO_InitStructure;
063
064     /* Configure 315M Button */
065     RCC_APB2PeriphClockCmd(W315M_GPIO_CLK, ENABLE);
066
067     GPIO_InitStructure.GPIO_Pin = W315M_VT_PIN;
068     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
069     GPIO_Init(W315M_GPIO_PORT, &GPIO_InitStructure);
070 }
```

我们将它配置为输入模式，具体引脚通过原理图确认。

代码分析 2：W315M_EXIT_VTConfig(), 315M 中断配置。

```

078 void W315M_EXIT_VTConfig(void)
079 {
080     EXTI_InitTypeDef EXTI_InitStructure;
081     NVIC_InitTypeDef NVIC_InitStructure;
082
083     /* Connect Button EXTI Line to Button GPIO Pin */
084     GPIO_EXTILineConfig(W315M_VT_EXTI_PORT_SOURCE, W315M_VT_EXTI_PIN_SOURCE);
085
086     /* Configure W315M_VT EXTI line */
087     EXTI_InitStructure.EXTI_Line = W315M_VT_EXTI_LINE;
088     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
089     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
090     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
091     EXTI_Init(&EXTI_InitStructure);
092
093     /* Enable the 315M VT EXTI Interrupt */
094     NVIC_InitStructure.NVIC IRQChannel = W315M_VT_EXTI_IRQn;
095     NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
096     NVIC_InitStructure.NVIC IRQChannelSubPriority = 0;
097     NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
098     NVIC_Init(&NVIC_InitStructure);
099 }

```

函数 GPIO_EXTILineConfig(), 连接 315M 中断线。下面对 EXTI_InitStructure 的各个成员进行配置。如：

1) .EXTI_Line 确认中断线为 EXTI_Line10。

| | |
|---------------------------------------|-----------------------|
| 211 /** 315M模块VT信号管脚 **/ | 10 |
| 212 #define W315M_VT_PIN_NUM | GPIO_Pin_10 |
| 213 #define W315M_VT_PIN | GPIOB |
| 214 #define W315M_GPIO_PORT | RCC_APB2Periph_GPIOB |
| 215 #define W315M_GPIO_CLK | |
| ►216 #define W315M_VT_EXTI_LINE | EXTI_Line10 |
| 217 #define W315M_VT_EXTI_PORT_SOURCE | GPIO_PortSourceGPIOB |
| 218 #define W315M_VT_EXTI_PIN_SOURCE | GPIO_PinSource10 |
| 219 #define W315M_VT_EXTI_IRQn | EXTI15_10_IRQn |
| 220 #define W315MVTIBB | Periph_BB((uint32_t)& |

2) .EXTI_Trigger

该成员我们赋值 EXTI_Trigger_Falling。下降沿触发中断。

最后通过 NVIC_InitStructure 的各个成员，配置中断优先级。

7.26.5 下载与验证

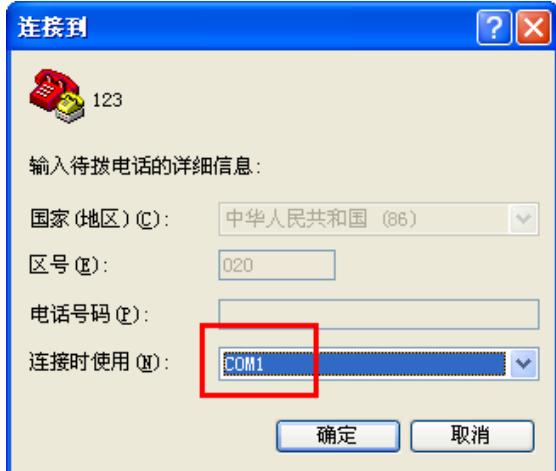
如果使用JLINK下载固件，请按[3.2如何使用JLINK软件](#)下载固件到神舟III号开发板小节进行操作。

如果使用USB下载固件，请按[错误！未找到引用源。错误！未找到引用源。](#)小节进行操作。

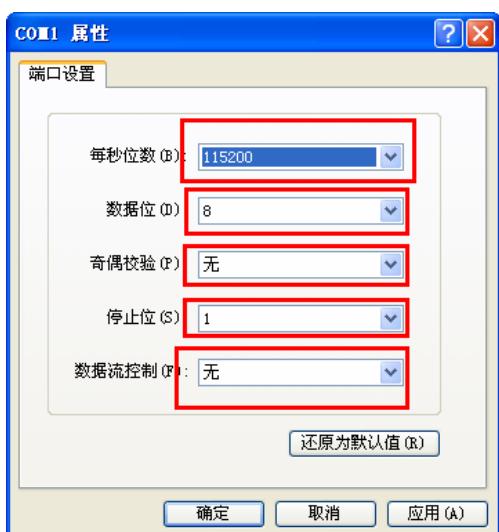
如果使用串口下载固件，请按[3.3如何通过串口下载一个固件到神舟III号开发板](#)小节进行操作。

7.26.6 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，关闭电源，用串口线神舟 III 号串口 1 与电脑连接，并打开超级终端，按以下设置，如下图：

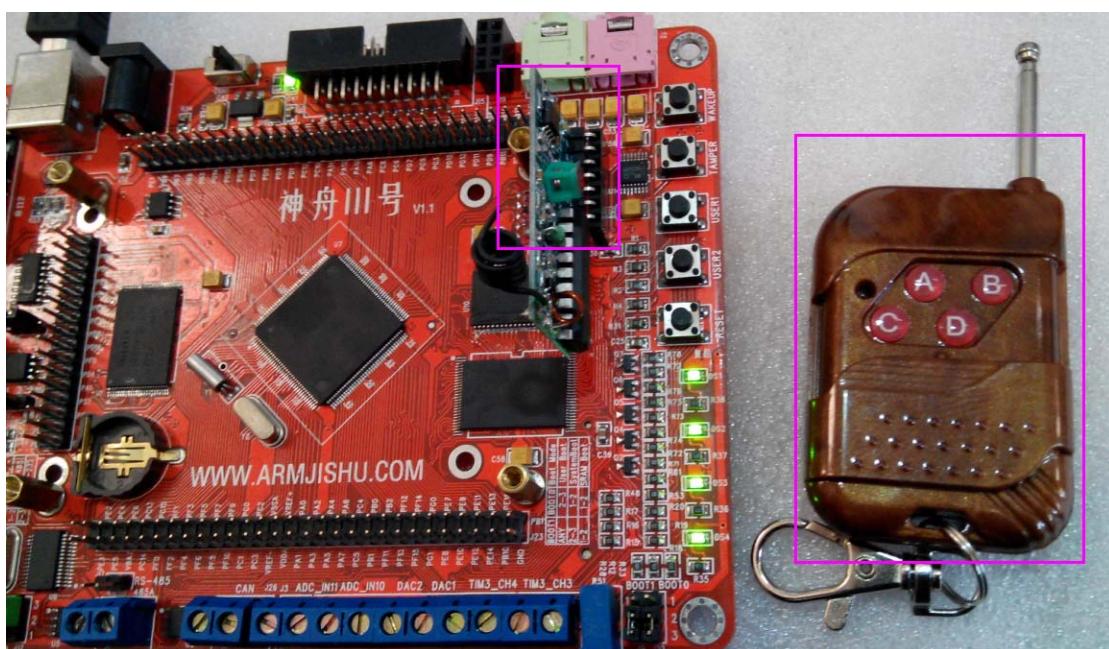


选择 COM1；按确定



再按确定，完成超级终端设置。

外设器件：315M 无线模块，315M 无线遥控器；将 315M 无线模块插在神舟 III 号开发板上，



重新打开电源；可以看到神舟 III 号开发板上的 4 个 LED 灯（LED1——LED4）都亮，串口打印数据如下：



按下 315M 无线遥控器，触发中断，串口打印信息。



7.27 CAN总线实验

这一节我们将向大家介绍STM32的CAN总线的基本使用。在本小节，我们初始化CAN总线，分别测试轮询模式和中断模式下的CAN总线环回，并通过神舟III号的LED等指示CAN环回的数据传送结果。

7.27.1 什么是CAN总线

CAN，全称“**Controller Area Network**”，即**控制器局域网**，是国际上应用最广泛的工业级现场总线之一。它是一种具有国际标准而且性能价格比又较高的现场总线，当今自动控制领域的发展中能发挥重要的作用。最初 CAN 被设计作为汽车环境中的微控制器通讯，在车载各电子控制装置 ECU 之间交换信息，形成汽车电子控制网络。比如：发动机管理系统、变速箱控制器、仪表装备、电子主干系统中，均嵌入 CAN 控制装置。

CAN 控制器局部网是 BOSCH 公司为现代汽车应用领先推出的一种多主机局部网，由于其卓越性能现已广泛应用于工业自动化、多种控制设备、交通工具、医疗仪器以及建筑、环境控制等众多部门。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。控制器局部网将在我国迅速普及推广。

由于 CAN 为愈来愈多不同领域采用和推广，导致要求各种应用领域通信报文的标准化。为此，1991 年 9 月 PHILIPS SEMICONDUCTORS 制订并发布了 CAN 技术规范（VERSION 2.0）。该技术规范包括 A 和 B 两部分。2.0A 给出了曾在 CAN 技术规范版本 1.2 中定义的 CAN 报文格式，而 2.0B 给出了标准的和扩展的两种报文格式。此后，1993 年 11 月 ISO 正式颁布了道路交通运输工具--数字信息交换--高速通信控制器局部网（CAN）国际标准（ISO11898），为控制器局部网标准化、规范化推广铺平了道路。

7.27.2 CAN总线的特点

CAN 总线是一种串行数据通信协议，它是一种多主总线，通信介质可以是双绞线、同轴电缆或光导纤维。通信速率可达 1MBPS。CAN 总线通信接口中集成了 CAN 协议的物理层和数据链路层功能，可完成对通信数据的成帧处理，包括位填充、数据块编码、循环冗余检验、优先级判别等项工作。

CAN 协议的一个最大特点是废除了传统的站地址编码，而代之以对通信数据块进行编码。采用这种方法的优点可使网络内的节点个数在理论上不受限制，数据块的标识码可由 11 位或 29 位二进制数组成，因此可以定义 211 或 229 个不同的数据块，这种按数据块编码的方式，还可使不同的节点同时接收到相同的数据，这一点在分布式控制系统中非常有用。数据段长度最多为 8 个字节，可满足通常工业领域中控制命令、工作状态及测试数据的一般要求。同时，8 个字节不会占用总线时间过长，从而保证了通信的实时性。CAN 协议采用 CRC 检验并可提供相应的错误处理功能，保证了数据通信的可靠性。CAN 卓越的特性、极高的可靠性和独特的设计，特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。

另外，CAN 总线采用了多主竞争式总线结构，具有多主站运行和分散仲裁的串行总线以及广播通信的特点。CAN 总线上任意节点可在任意时刻主动地向网络上其它节点发送信息而不分主次，因此可在各节点之间实现自由通信。CAN 总线协议已被国际标准化组织认证，技术比较成熟，控制的芯片已经商品化，性价比高，特别适用于分布式测控系统之间的数据通讯。CAN 总线插卡可以任意插在 PC AT XT 兼容机上，方便地构成分布式监控系统。

7.27.3 CAN总线技术介绍

1 位仲裁

要对数据进行实时处理，就必须将数据快速传送，这就要求数据的物理传输通路有较高的速度。在几个站同时需要发送数据时，要求快速地进行总线分配。实时处理通过网络交换的紧急数据有较大的不同。一个快速变化的物理量，如汽车引擎负载，将比类似汽车引擎温度这样相对变化较慢的物理量更频繁地传送数据并要求更短的延时。

CAN 总线以报文为单位进行数据传送，报文的优先级结合在 11 位标识符中，具有最低二进制数的标识符有最高的优先级。这种优先级一旦在系统设计时被确立后就不能再被更改。总线读取中的冲突可通过位仲裁解决。当几个站同时发送报文时，站 1 的报文标识符为 011111；站 2 的报文标识符为 0100110；站 3 的报文标识符为 0100111。所有标识符都有相同的两位 01，直到第 3 位进行比较时，站 1 的报文被丢弃，因为它的第 3 位为高，而其它两个站的报文第 3 位为低。站 2 和站 3 报文的 4、5、6 位相同，直到第 7 位时，站 3 的报文才被丢弃。注意，总线中的信号持续跟踪最后获得总线读取权的站的报文。在此例中，站 2 的报文被跟踪。这种非破坏性位仲裁方法的优点在于，在网络最终确定哪一个站的报文被传送以前，报文的起始部分已经在网络上传送了。所有未获得总线读取权的站都成为具有最高优先权报文的接收站，并且不会在总线再次空闲前发送报文。

CAN 具有较高的效率是因为总线仅仅被那些请求总线悬而未决的站利用，这些请求是根据报文在整个系统中的重要性按顺序处理的。这种方法在网络负载较重时有很多优点，因为总线读取的优先级已被按顺序放在每个报文中了，这可以保证在实时系统中较低的个体隐伏时间。

对于主站的可靠性，由于 CAN 协议执行非集中化总线控制，所有主要通信，包括总线读取（许可）控制，在系统中分几次完成。这是实现有较高可靠性的通信系统的唯一方法。

2 CAN 与其它通信方案的比较

在实践中,有两种重要的总线分配方法:按时间表分配和按需要分配。在第一种方法中,不管每个节点是否申请总线,都对每个节点按最大期间分配。由此,总线可被分配给每个站并且是唯一的站,而不论其是立即进行总线存取或在一特定时间进行总线存取。这将保证在总线存取时有明确的总线分配。在第二种方法中,总线按传送数据的基本要求分配给一个站,总线系统按站希望的传送分配(如:Ethernet CSMA/CD)。因此,当多个站同时请求总线存取时,总线将终止所有站的请求,这时将不会有任何一个站获得总线分配。为了分配总线,多于一个总线存取是必要的。

CAN 实现总线分配的方法,可保证当不同的站申请总线存取时,明确地进行总线分配。这种位仲裁的方法可以解决当两个站同时发送数据时产生的碰撞问题。不同于 Ethernet 网络的消息仲裁,CAN 的非破坏性解决总线存取冲突的方法,确保在不传送有用消息时总线不被占用。甚至当总线在重负载情况下,以消息内容为优先的总线存取也被证明是一种有效的系统。虽然总线的传输能力不足,所有未解决的传输请求都按重要性顺序来处理。在 CSMA/CD 这样的网络中,如 Ethernet,系统往往由于过载而崩溃,而这种情况在 CAN 中不会发生。

3 CAN 的报文格式

在总线中传送的报文,每帧由 7 部分组成。CAN 协议支持两种报文格式,其唯一的不同是标识符(ID)长度不同,标准格式为 11 位,扩展格式为 29 位。

在标准格式中,报文的起始位称为帧起始(SOF),然后是由 11 位标识符和远程发送请求位 (RTR) 组成的仲裁场。RTR 位标明是数据帧还是请求帧,在请求帧中没有数据字节。

控制场包括标识符扩展位(IDE),指出是标准格式还是扩展格式。它还包括一个保留位 (ro),为将来扩展使用。它的最后四个字节用来指明数据场中数据的长度(DLC)。数据场范围为 0~8 个字节,其后有一个检测数据错误的循环冗余检查(CRC)。

应答场(ACK)包括应答位和应答分隔符。发送站发送的这两位均为隐性电平(逻辑 1),这时正确接收报文的接收站发送主控电平(逻辑 0)覆盖它。用这种方法,发送站可以保证网络中至少有一个站能正确接收到报文。

报文的尾部由帧结束标出。在相邻的两条报文间有一很短的间隔位,如果这时没有站进行总线存取,总线将处于空闲状态。

4 数据错误检测

不同于其它总线,CAN 协议不能使用应答信息。事实上,它可以将发生的任何错误用信号发出。CAN 协议可使用五种检查错误的方法,其中前三种为基于报文内容检查。

4.1 循环冗余检查(CRC)

在一帧报文中加入冗余检查位可保证报文正确。接收站通过 CRC 可判断报文是否有错。

4.2 帧检查

这种方法通过位场检查帧的格式和大小来确定报文的正确性,用于检查格式上的错误。

4.3. 应答错误

如前所述,被接收到的帧由接收站通过明确的应答来确认。如果发送站未收到应答,那么表明接收站发现帧中有错误,也就是说,ACK 场已损坏或网络中的报文无站接收。CAN 协议也可通过位检查的方法探测错误。

4.4 总线检测

有时,CAN 中的一个节点可监测自己发出的信号。因此,发送报文的站可以观测总线电平并探测发送位和接收位的差异。

4.5 位填充

一帧报文中的每一位都由不归零码表示,可保证位编码的最大效率。然而,如果在一帧报文中有太多相同电平的位,就有可能失去同步。为保证同步,同步沿用位填充产生。在五个生。在五个连续相等

位后,发送站自动插入一个与之互补的补码位;接收时,这个填充位被自动丢掉。例如,五个连续的低电平位后,CAN 自动插入一个高电平位。CAN 通过这种编码规则检查错误,如果在一帧报文中有 6 个相同位,CAN 就知道发生了错误。

如果至少有一个站通过以上方法探测到一个或多个错误,它将发送出错标志终止当前的发送。这可以阻止其它站接收错误的报文,并保证网络上报文的一致性。当大量发送数据被终止后,发送站会自动地重新发送数据。作为规则,在探测到错误后 23 个位周期内重新开始发送。在特殊场合,系统的恢复时间为 31 个位周期。

但这种方法存在一个问题,即一个发生错误的站将导致所有数据被终止,其中包括正确的数据。因此,如果不采取自监测措施,总线系统应采用模块化设计。为此,CAN 协议提供一种将偶然错误从永久错误和局部站失败中区别出来的办法。这种方法通过对出错站统计评估来确定一个站本身的错误并进入一种不会对其他站产生不良影响的运行方法来实现,即站可以通过关闭自己来阻止正常数据因被错误地当成不正确的数据而被终止。

4.6 CAN 可靠性

为防止汽车在使用寿命期内由于数据交换错误而对司机造成危险,汽车的安全系统要求数据传输具有较高的安全性。如果数据传输的可靠性足够高,或者残留下来的数据错误足够低的话,这一目标不难实现。从总线系统数据的角度看,可靠性可以理解为,对传输过程产生的数据错误的识别能力。

残余数据错误的概率可以通过对数据传输可靠性的统计测量获得。它描述了传送数据被破坏和这种破坏不能被探测出来的概率。残余数据错误概率必须非常小,使其在系统整个寿命周期内,按平均统计时几乎检测不到。计算残余错误概率要求能够对数据错误进行分类,并且数据传输路径可由一模型描述。如果要确定 CAN 的残余错误概率,我们可将残留错误的概率作为具有 80~90 位的报文传送时位错误概率的函数,并假定这个系统中有 5~10 个站,并且错误率为 1/1000,那么最大位错误概率为 10—13 数量级。例如,CAN 网络的数据传输率最大为 1Mbps,如果数据传输能力仅使用 50%,那么对于一个工作寿命 4000 小时、平均报文长度为 80 位的系统,所传送的数据总量为 9×10^{10} 。在系统运行寿命期内,不可检测的传输错误的统计平均小于 10—2 量级。换句话说,一个系统按每年 365 天,每天工作 8 小时,每秒错误率为 0.7 计算,那么按统计平均,每 1000 年才会发生一个不可检测的错误。

4.应用举例

某医院现有 5 台 16T/H 德国菲斯曼燃气锅炉,向洗衣房、制剂室、供应室、生活用水、暖气等设施提供 5kg/cm² 的蒸汽,全年耗用天然气 1200 万 m³,耗用 20 万吨自来水。医院采用接力式方式供热,对热网进行地域性管理,分四大供热区。其中冬季暖气的用气量很大,据此设计了基于 CAN 现场总线的分布式锅炉蒸汽热网智能监控系统。现场应用表明:该楼宇自动化系统具有抗干扰能力强,现场组态容易,网络化程度高,人机界面友好等特点。

7.27.4 CAN 总线关键特性参数

CAN 总线有如下基本特点:

- ◎ 废除传统的站地址编码,代之以对通信数据块进行编码,可以多主方式工作;
- ◎ 采用非破坏性仲裁技术,当两个节点同时向网络上传送数据时,优先级低的节点主动停止数据发送,而优先级高的节点可不受影响继续传输数据,有效避免了总线冲突;
- ◎ 采用短帧结构,每一帧的有效字节数为 8 个,数据传输时间短,受干扰的概率低,重新发送的时间短;
- ◎ 每帧数据都有 CRC 校验及其他检错措施,保证了数据传输的高可靠性,适于在高干扰环境下使用;
- ◎ 节点在错误严重的情况下,具有自动关闭总线的功能,切断它与总线的联系,以使总线上其他操作不受影响;
- ◎ 可以点对点,一对多及广播集中方式传送和接受数据。

CAN 总线的**优点**

- 具有实时性强、传输距离较远、抗电磁干扰能力强、成本低等优点；
- 采用双线串行通信方式，检错能力强，可在高噪声干扰环境中工作；
- 具有优先权和仲裁功能，多个控制模块通过 CAN 控制器挂到 CAN-bus 上，形成多主机局部网络；
 - 可根据报文的 ID 决定接收或屏蔽该报文；
 - 可靠的错误处理和检错机制；
 - 发送的信息遭到破坏后，可自动重发；
 - 节点在错误严重的情况下具有自动退出总线的功能；
 - 报文不包含源地址或目标地址，仅用标志符来指示功能信息、优先级信息。

STM32 处理器的 CAN 总线，支持 CAN 协议 2.0A 和 2.0B。具有以下特点：

- ◆ 支持CAN协议2.0A和2.0B主动模式
- ◆ 波特率最高可达1兆位/秒
- ◆ 支持时间触发通信功能
- 发送
 - ◆ 3个发送邮箱
 - ◆ 发送报文的优先级特性可软件配置
 - ◆ 记录发送SOF时刻的时间戳
- 接收
 - ◆ 3级深度的2个接收FIFO
 - ◆ 14个位宽可变的过滤器组—由整个CAN共享标识符列表
 - ◆ FIFO溢出处理方式可配置
 - ◆ 记录接收SOF时刻的时间戳
 - ◆ 可支持时间触发通信模式
 - ◆ 禁止自动重传模式
 - ◆ 16位自由运行定时器
 - ◆ 定时器分辨率可配置
 - ◆ 可在最后2个数据字节发送时间戳
- 管理
 - ◆ 中断可屏蔽
 - ◆ 邮箱占用单独1块地址空间，便于提高软件效率

7.27.5 CAN总线的三种工作模式

STM32 的 CAN 一共有 3 个主要的工作模式，分别是初始化、正常和睡眠模式。

● 初始化模式

软件通过对CAN_MCR寄存器的INRQ位置1，来请求bxCAN进入初始化模式，然后等待硬件对CAN_MSR寄存器的INAK位置1来进行确认软件通过对CAN_MCR寄存器的INRQ位清0，来请求

bxCAN退出初始化模式，当硬件对CAN_MSR寄存器的INAK位清0就确认了初始化模式的退出。

当bxCAN处于初始化模式时，报文的接收和发送都被禁止，并且CANTX引脚输出隐性位（高电平）。

● 正常模式

在初始化完成后，软件应该让硬件进入正常模式，以便正常接收和发送报文。软件可以通过对CAN_MCR寄存器的INRQ位清0，来请求从初始化模式进入正常模式，然后要等待硬件对CAN_MSR寄存器的INAK位置1的确认。在跟CAN总线取得同步，即在CANRX引脚上监测到11个连续的隐性位（等效于总线空闲）后，bxCAN才能正常接收和发送报文。

● 睡眠模式（低功耗）

软件通过对CAN_MCR寄存器的SLEEP位置1，来请求进入这一模式。在该模式下，bxCAN的时钟停止了，但软件仍然可以访问邮箱寄存器。当bxCAN处于睡眠模式，软件想通过对CAN_MCR寄存器的INRQ位置1，来进入初始化模式，那么软件必须同时对SLEEP位清0才行。

有2种方式可以唤醒（退出睡眠模式）bxCAN：通过软件对SLEEP位清0，或硬件检测CAN总线的活动。

7.27.6 实验原理

◆ CAN 发送报文介绍

应用程序选择1个空发送邮箱；设置标识符，数据长度和待发送数据；然后对CAN_TIxR寄存器的TXRQ位置1，来请求发送。TXRQ位置1后，邮箱就不再是空邮箱；而一旦邮箱不再为空，软件对邮箱寄存器就不再有写的权限。TXRQ位置1后，邮箱马上进入挂号状态，并等待成为最高优先级的邮箱，参见发送优先级。一旦邮箱成为最高优先级的邮箱，其状态就变为预定发送状态。一旦CAN总线进入空闲状态，预定发送邮箱中的报文就马上被发送（进入发送状态）。一旦邮箱中的报文被成功发送后，它马上变为空邮箱；硬件相应地对CAN_TSR寄存器的RQCP和TXOK位置1，来表明一次成功发送。

如果发送失败，由于仲裁引起的就对CAN_TSR寄存器的ALST位置1，由于发送错误引起的，就对TERR位置1。

原来发送的优先级可以由标识符和发送请求次序决定：

◆ 由标识符决定

当有超过1个发送邮箱在挂号时，发送顺序由邮箱中报文的标识符决定。根据CAN协议，标识符数值最低的报文具有最高的优先级。如果标识符的值相等，那么邮箱号小的报文先被发送。

◆ 由发送请求次序决定

通过对CAN_MCR寄存器的TXFP位置1，可以把发送邮箱配置为发送FIFO。在该模式下，发送的优先级由发送请求次序决定。该模式对分段发送很有用。

◆ CAN 接收报文介绍

接收到的报文，被存储在3级邮箱深度的FIFO中。FIFO完全由硬件来管理，从而节省了CPU的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取FIFO输出邮箱，来读取FIFO中最先收到的报文。

根据CAN协议，当报文被正确接收（直到EOF域的最后1位都没有错误），且通过了标识符过滤，那么该报文被认为是有效报文。

◆ 接收相关的中断条件

- 一旦往FIFO存入1个报文，硬件就会更新FMP[1:0]位，并且如果CAN_IER寄存器的FMPIE位为1，那么就会产生一个中断请求。
- 当FIFO变满时（即第3个报文被存入），CAN_RXR寄存器的FULL位就被置1，并且如果CAN_IER寄存器的FFIE位为1，那么就会产生一个满中断请求。
- 在溢出的情况下，FOVR位被置1，并且如果CAN_IER寄存器的FOVIE位为1，那么就会产生一个溢出中断请求

◆ 标识符过滤

在CAN协议里，报文的标识符不代表节点的地址，而是跟报文的内容相关的。因此，发送者以广播的形式把报文发送给所有的接收者。（注：不是一对一通信，而是多机通信）节点在接收报文时一根据标识符的值一决定软件是否需要该报文；如果需要，就拷贝到SRAM里；如果不需要，报文就被丢弃且无需软件的干预。

为满足这一需求，bxCAN为应用程序提供了14个位宽可变的、可配置的过滤器组（13~0），以便只接收那些软件需要的报文。硬件过滤的做法节省了CPU开销，否则就必须由软件过滤从而占用一定的CPU开销。每个过滤器组x由2个32位寄存器，CAN_FxR0和CAN_FxR1组成。

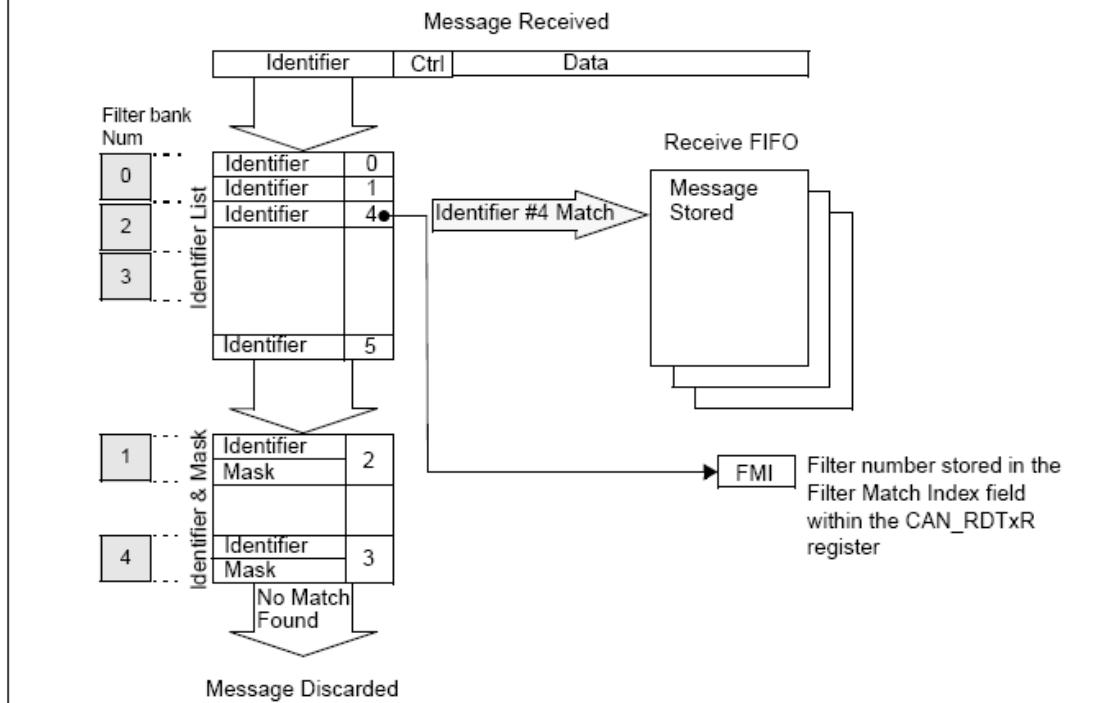
过滤器的模式的设置

- 通过设置CAN_FM0R的FBMx位，可以配置过滤器组为标识符列表模式或屏蔽位模式。
- 为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。
- 为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。
- 应用程序不用的过滤器组，应该保持在禁用状态。

过滤器优先级规则

- 位宽为32位的过滤器，优先级高于位宽为16位的过滤器
- 对于位宽相同的过滤器，标识符列表模式的优先级高于屏蔽位模式
- 位宽和模式都相同的过滤器，优先级由过滤器号决定，过滤器号小的优先级高

Example of 3 filter banks in 32-bit Unidentified List mode and the remaining in 32-bit Identifier Mask mode



上面的例子说明了bxCAN的过滤器规则：在接收一个报文时，其标识符首先与配置在标识符列表模式下的过滤器相比较；如果匹配上，报文就被存放到相关联的FIFO中，并且所匹配的过滤器的序号被存入过滤器匹配序号中。如同例子中所显示，报文标识符跟#4标识符匹配，因此报文内容和FMI4被存入FIFO。

如果没有匹配，报文标识符接着与配置在屏蔽位模式下的过滤器进行比较。

如果报文标识符没有跟过滤器中的任何标识符相匹配，那么硬件就丢弃该报文，且不会对软件有任何打扰。

◆ 接收邮箱（FIFO）

在接收到一个报文后，软件就可以访问接收FIFO的输出邮箱来读取它。一旦软件处理了报文（如把它读出来），软件就应该对CAN_RFxR寄存器的RFOM位进行置1，来释放该报文，以便为后面收到的报文留出存储空间。

中断

bxCAN占用4个专用的中断向量。通过设置CAN中断允许寄存器(CAN_IER)，每个中断源都可以单独允许和禁用。

发送中断可由下列事件产生：

- 发送邮箱0变为空，CAN_TSR寄存器的RQCP0位被置1。
- 发送邮箱1变为空，CAN_TSR寄存器的RQCP1位被置1。
- 发送邮箱2变为空，CAN_TSR寄存器的RQCP2位被置1。

FIFO0中断可由下列事件产生：

- FIFO0接收到一个新报文，CAN_RF0R寄存器的FMP0位不再是‘00’。
- FIFO0变为满的情况，CAN_RF0R寄存器的FULL0位被置1。
- FIFO0发生溢出的情况，CAN_RF0R寄存器的FOVR0位被置1。

FIFO1中断可由下列事件产生：

- FIFO1接收到一个新报文，CAN_RF1R寄存器的FMP1位不再是‘00’。
- FIFO1变为满的情况，CAN_RF1R寄存器的FULL1位被置1。
- FIFO1发生溢出的情况，CAN_RF1R寄存器的FOVR1位被置1。

错误和状态变化中断可由下列事件产生：

- 出错情况，关于出错情况的详细信息请参考CAN错误状态寄存器(CAN_ESR)。
- 唤醒情况，在CAN接收引脚上监视到帧起始位(SOF)。
- CAN进入睡眠模式。

波特率设定

CAN总线通信的难点在于波特率的设定，当然如果是近距离通信，只有几十米那就可以忽略了，如果几公里 波特率 位序 时序 都需要好好计算 不然通信是不成功的。

CAN控制器器只需要进行少量的设置就可以进行通信,就可以像RS232/485那样使用。其中较难设置的部分就是通信波特率的计算。CAN总线能够在一定的范围内容忍总线上CAN节点的通信波特率的偏差，这种机能使得CAN总线有很强的容错性，同时也降低了对每个节点的振荡器精度。

实际上，CAN总线的波特率是一个范围。假设定义的波特率是250KB/S，但是实际上根据对寄存器的设置，实际的波特率可能为200~300KB/S（具体值取决于寄存器的设置）,简单介绍一个波特率的计算，在CAN的底层协议里将CAN数据的每一位时间(TBit)分为许多的时间段(Tscl)，这些时间段包括：

- A. 位同步时间(Tsync)
- B. 时间段1(Tseg1)
- C. 时间段2(Tseg2)

其中位同步时间占用1个Tscl；时间段2占用(Tseg1+1)个Tscl；时间段2占用(Tseg2+1)个Tscl,所以CAN控制器的位时间(TBit)就是： $TBit = Tseg1 + Tseg2 + Tsync = (TSEG1 + TSEG2 + 3) * Tscl$ ，那么CAN的波特率(CANbps)就是 $1/TBit$ 。

但是这样计算出的值是一个理论值。在实际的网络通信中由于存在传输的延时、不同节点的晶体的误差等因素，使得网络CAN的波特率的计算变得复杂起来。CAN在技术上便引入了重同步的概念，以更好的解决这些问题。这样重同步带来的结果就是要么时间段1(Tseg1)增加TSJW（同步跳转宽度SJW+1），要么时间段减少TSJW，因此CAN的波特率实际上有一个范围：

$$1/(Tbit+Tsjw) \leq CANbps \leq 1/(Tbit-Tsjw)$$

CAN有波特率的值由以下几个元素决定：

- A. 最小时间段Tscl;
- B. 时间段1 TSEG1;
- C. 时间段2 TSEG2;
- D. 同步跳转宽度 SJW

那么Tscl又是怎么计算的呢？这是总线时序寄存器中的预分频寄存器BRP派上了用场， $Tscl = (BRP+1) / FVBP$ 。FVBP为微处理器的外设时钟。

而TSEG1与TSEG2又是怎么划分的呢？TSEG1与TSEG2的长度决定了CAN数据的采样点，这种方式允许宽范围的数据传输延迟和晶体的误差。其中TSEG1用来调整数据传输延迟时间造成的误差，而TSEG2则用来调整不同节点晶体频率的误差。

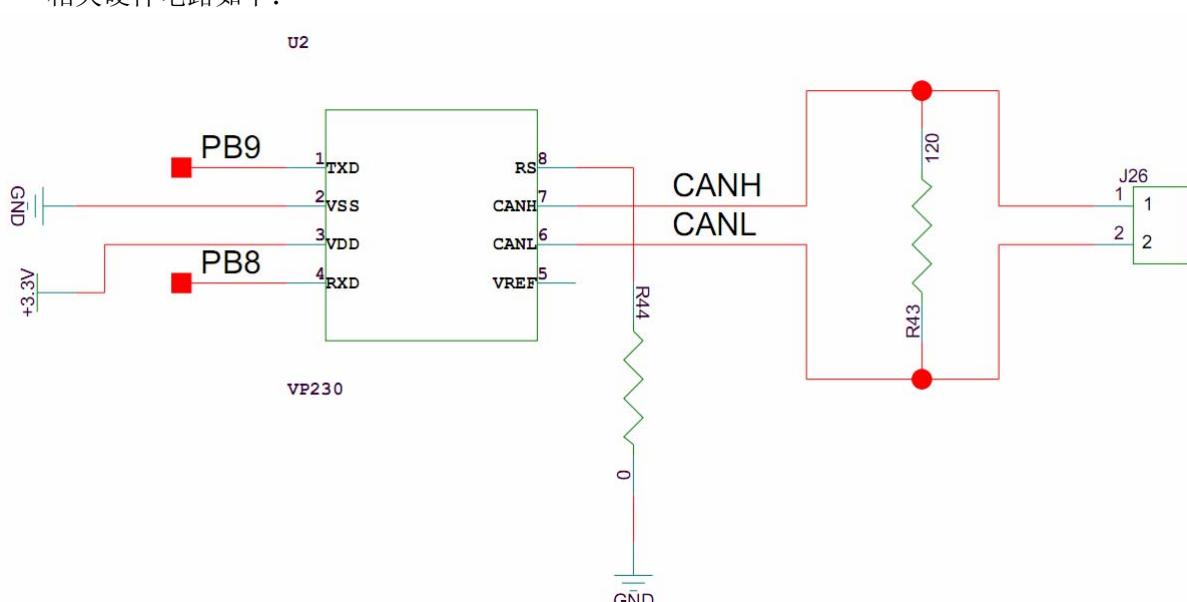
总的来说，对于CAN的波特率计算问题，把握一个大的方向就行了，其计算公式可简化为：

$$\text{BitRate} = \text{Fpclk} / ((\text{BRP}+1) * (\text{Tseg1}+1) + (\text{Tseg2}+1) + 1)$$

7.27.7 硬件设计

在神舟III号STM32开发板中，我们使用了处理器STM32F103ZET6的CAN外扩TI的VP230 CAN总线收发芯片来实现CAN总线接口。

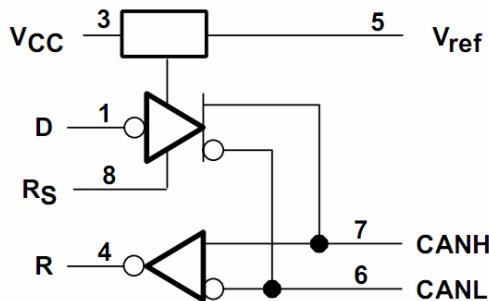
相关硬件电路如下：



其中 VP230 是 TI 公司推出的 3.3V CAN 总线收发器。它具有如下特性：

- ◆ 工作电压为 3.3V
- ◆ 满足 HBM 模式 16KV 的 ESD 防护
- ◆ 允许总线上最多到 120 个节点
- ◆ 符合 ISO 11898 标准要求
- ◆ 具有过热关断保护功能

VP230 的逻辑框图如下：



管脚功能如下：

| TERMINAL NAME | NO. | DESCRIPTION |
|------------------|-----|-----------------------|
| CANL | 6 | Low bus output |
| CANH | 7 | High bus output |
| D | 1 | Driver input |
| GND | 2 | Ground |
| R | 4 | Receiver output |
| RS | 8 | Standby/slope control |
| VCC | 3 | Supply voltage |
| Vref | 5 | Reference output |

7.27.8 软件设计

时钟使能配置

在本实验中，我们分别通过轮询和中断两种方式，测试CAN总线的环回，并通过LED灯只是CAN总线环回结果。因此，我们使用到的硬件资源有LED灯，CAN1总线接口，在使用这些资源之前，我们需要先使能这么资源模块的时钟，具体代码如下：

```
/*使能CAN1总线的时钟*/
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);

/* 使能LED灯使用的GPIO管脚时钟*/
RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE);
```

◆ CAN 总线中断配置

在本实验中，我们通过分别利用轮询和中断方式进行CAN的环回（loopback）实验，因此，我们需要设置CAN1接口的接收终端及其优先级，在本例程中，设置中断组1个，CAN1的子优先级别为0，相关代码如下：

```
void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable CAN1 RX0 interrupt IRQ channel */
#ifdef STM32F10X_CL
    NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
#else
    NVIC_InitStructure.NVIC_IRQChannel = CAN1_RX0_IRQn;
#endif /* STM32F10X_CL */
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

◆ LED 灯配置

在神舟III号STM32开发板中，4个LED灯由GPIOF6~9这四个GPIO管脚控制，当GPIO输出低电平时，LED灯亮，反之，LED灯熄灭，为了使用LED灯，我们需要将神舟III号LED占用的GPIO初始化为推挽输出模式，相关代码如下：

神舟III号LED灯使用的GPIO接口定义

```
/*神州III号LED灯相关定义*/
#define RCC_GPIO_LED          RCC_APB2Periph_GPIOF /*LED使用的GPIO时钟*/
#define LEDn                  4                      /*神舟III号LED数量*/
#define GPIO_LED               GPIOF                 /*神舟III号LED灯使用的GPIO组*/
#define DS1_PIN                GPIO_Pin_6           /*DS1使用的GPIO管脚*/
#define DS2_PIN                GPIO_Pin_7           /*DS2使用的GPIO管脚*/
#define DS3_PIN                GPIO_Pin_8           /*DS3使用的GPIO管脚*/
#define DS4_PIN                GPIO_Pin_9           /*DS4使用的GPIO管脚*/
```

神舟III号LED灯使用的GPIO接口初始化

```
GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

GPIO_Init(GPIO_LED, &GPIO_InitStructure); /*神州III号使用的LED灯相关的GPIO口初始化*/
GPIO_SetBits(GPIO_LED,DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN);/*关闭所有的LED指示灯*/
```

轮询方式CAN总线环回测试

在本实验中，CAN环回实验，分成中断和轮询两种模式进行测试，其中轮询方式的CAN接口初始化和测试主要是由TestStatus CAN_Polling(void)函数实现的。

在这个函数中，我们首先对CAN1接口进行配置，包括bus-off管理，wake-up模式设置等等，其中CAN_Init函数根据CAN_InitStruct中指定的参数初始化CAN寄存器，相关代码如下：

```
/* CAN cell init */
CAN_InitStructure.CAN_TTCM=DISABLE; //禁止时间触发通信
CAN_InitStructure.CAN_ABOM=DISABLE; //离线退出是在中断置位清零后退出
CAN_InitStructure.CAN_AWUM=DISABLE; //自动唤醒模式:清零SLEEP
CAN_InitStructure.CAN_NART=DISABLE; //自动重新传送报文，直到发送成功
CAN_InitStructure.CAN_RFLM=DISABLE; //FIFO没有锁定，新报文覆盖旧报文
CAN_InitStructure.CAN_TXFP=DISABLE; //发送报文优先级确定：标识符
CAN_InitStructure.CAN_Mode=CAN_Mode_LoopBack; //CAN模式选择
CAN_InitStructure.CAN_SJW=CAN_SJW_1tq;
CAN_InitStructure.CAN_BS1=CAN_BS1_8tq;
CAN_InitStructure.CAN_BS2=CAN_BS2_7tq;
CAN_InitStructure.CAN_Prescaler=5;
CAN_Init(CAN1, &CAN_InitStructure);
```

CAN_InitStruct结构体的成员说明见下表

| | |
|---------------|---|
| CAN_TTCM | CAN_TTCM用来使能或去使能时间触发通讯模式 |
| CAN_ABOM | CAN_ABOM用来使能或者去使能自动离线管理 |
| CAN_AWUM | CAN_AWUM用来使能或者去使能自动唤醒模式 |
| CAN_NART | CAN_NART用来使能或者去使能非自动重传模式 |
| CAN_RFLM | CAN_RFLM用来使能或者去使能FIFO锁定模式 |
| CAN_TXFP | CAN_TXFP用来使能或者去使能发送FIFO优先级 |
| CAN_MODE | 设置CAN工作模式，可设置为正常工作模式(CAN_Mode_Normal) 静默模式(CAN_Mode_Silent),环回模式(CAN_Mode_LoopBack) 和静默环回模式(CAN_Mode_Silent_LoopBack) |
| CAN_SJW | 定义重新同步跳跃宽度，即每位中可以延长或缩短多少个时间单位的上限，可设置为1~4个时间单位(CAN_SJW_1tq~CAN_SJW_4tq) |
| CAN_BS1 | 设定时间段1的时间单位数目，可设置为1~16个时间单位(CAN_BS1_1tq~CAN_BS1_16tq)。 |
| CAN_BS2 | 设定时间段2的时间单位数目，可设置为1~16个时间单位(CAN_BS1_1tq~CAN_BS1_16tq)。 |
| CAN_Prescaler | 设定一个时间单位的长度，它的范围是1~1024 |

在完成CAN的基本参数配置以后，我们还需要对滤波器的参数进行配置，这个主要是通过CAN_FilterInit函数实现的，相关代码如下：

```
/* CAN filter init */
CAN_FilterInitStructure.CAN_FilterNumber=0;
CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;
CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000;
CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterFIFOAssignment=0;
CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;
CAN_FilterInit(&CAN_FilterInitStructure);
```

| | |
|--------------------------|---|
| CAN_FilterNumber | 指定待初始化的过滤器，它的范围是1到13 |
| CAN_FilterMode | 指定过滤器将被初始化的模式，可设置为标识符屏蔽位模式（CAN_FilterMode_IdMask）或者标识符列表模式（CAN_FilterMode_IdList） |
| CAN_FilterScale | 过滤器位宽，可设置为2个16位过滤器（CAN_FilterScale_Two16bit）或者1个32位过滤器（CAN_FilterScale_One32bit）。 |
| CAN_FilterIDHigh | 用来设定过滤器标识符（32位位宽时为其高段位，16位宽时为第一个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterIDLow | 用来设定过滤器标识符（32位位宽时为其低段位，16位宽时为第二个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterMaskIDHigh | 用来设定过滤器屏蔽标识符或者过滤器标识符（32位位宽时为其高段位，16位宽时为第一个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterMaskIDLow | 用来设定过滤器屏蔽标识符或者过滤器标识符（32位位宽时为其低段位，16位宽时为第二个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterFIFOAssignment | 设置指向过滤器的FIFO（0或者1） |
| CAN_FilterActivation | 使能或者去使能过滤器 |

前面，我们完成了CAN总线接口的参数配置和滤波器配置，正常情况下，我们就可以使用CAN总线接口了，在接下来，我们来看一下另两个非常重要的函数CAN_Transmit（）和CAN_Receive（），如何发送和接收一个正确格式的CAN帧。

在我们的例程中，CAN_Transmit（）相关的代码为

```
/* transmit */
TxMessage.StdId=0x11;
TxMessage.RTR=CAN_RTR_DATA;
TxMessage.IDE=CAN_ID_STD;
TxMessage.DLC=2;
TxMessage.Data[0]=0xCA;
TxMessage.Data[1]=0xFE;

TransmitMailbox=CAN_Transmit(CAN1, &TxMessage);
```

它的相关参数说明如下：

| | |
|---------|---|
| StdId | 用来设定标准标识符。可设置为0到0x7FF |
| ExtId | 用来设定扩展标识符，可设置为0到0x3FFFF |
| IDE | 用来设定消息标识符的类型，可设置为使用标准标识符（CAN_ID_STD）或者使用标准标识符+扩展标识符（CAN_ID_EXT） |
| RTR | 设定待传输消息的帧类型，可设置为数据帧（CAN_RTR_DATA）或者远程帧（CAN_RTR_REMOTE） |
| DLC | 用来设定带传输消息的帧长度。取值范围0到0x8 |
| Data[8] | 待传输的数据 |

常传输，待进入传输状态以后，由于是采用LoopBack方式，数据应该马上环回接收侧，因此程序接下来又调用CAN_MessagePending()检查CAN接口的挂号信息数据，大于等于1表示接收到的新的数据，调用CAN_Receive函数进行处理。相关代码为

```
i = 0;
//CAN_TransmitStatus 检查消息传输状态，如果没有在传输，重复检查，最大255次
while( (CAN_TransmitStatus(CAN1, TransmitMailbox) != CANTXOK) && (i != 0xFF) )
{
    i++;
}

i = 0;
//CAN_MessagePending 返回挂号的信息数目，如果数目小于1，则重复检查，最大255次。
while( (CAN_MessagePending(CAN1, CAN_FIFO0) < 1) && (i != 0xFF) )
{
    i++;
}

/* receive */
RxMessage.StdId=0x00;
RxMessage.IDE=CAN_ID_STD;
RxMessage.DLC=0;
RxMessage.Data[0]=0x00;
RxMessage.Data[1]=0x00;
CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);

/* receive */
RxMessage.StdId=0x00;
RxMessage.IDE=CAN_ID_STD;
RxMessage.DLC=0;
RxMessage.Data[0]=0x00;
RxMessage.Data[1]=0x00;
CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
```

它的相关参数说明如下：

| | |
|---------|---|
| StdId | 用来设定标准标识符。可设置为0到0x7FF |
| ExtId | 用来设定扩展标识符，可设置为0到0x3FFFF |
| IDE | 用来设定消息标识符的类型，可设置为使用标准标识符(CAN_ID_STD)或者使用标准标识符+扩展标识符(CAN_ID_EXT) |
| RTR | 设定待传输消息的帧类型，可设置为数据帧(CAN_RTR_DATA)或者远程帧(CAN_RTR_REMOTE) |
| DLC | 用来设定带传输消息的帧长度。取值范围0到0x8 |
| Data[8] | 待传输的数据 |
| FMI | 设定消息将要通过的过滤器索引 |

至此，就实现了一个CAN总线的LoopBack模式的数据收发，在例程中，还对接收的数据的正确性进行了一系列的判断，最终确认接收到的数据是否出现错误。并返回对应的状态。

```
if (RxMessage.StdId!=0x11)
{
    return FAILED;
}

if (RxMessage.IDE!=CAN_ID_STD)
{
    return FAILED;
}

if (RxMessage.DLC!=2)
{
    return FAILED;
}

if ((RxMessage.Data[0]<<8|RxMessage.Data[1])!=0xCAFE)
{
    return FAILED;
}

return PASSED; /* Test Passed */
```

中断方式CAN总线环回测试

前面，我们分析了CAN总线环回实验的轮询模式的具体实现过程，接下来，我们一起分析一下，中断模式下的CAN总线的环回具体实现过程。这个主要是在CAN_Interrupt(void)函数实现的。

首先，和轮询模式一样，我们也要通过利用CAN_Init函数和CAN_FilterInit函数初始化CAN接口和滤波器参数。

主要的不同是，在晚上上述初始化以后，我们需要使用CAN FIFO的中断，并完成对应的中断服务程序。

```
/* CAN FIFO0 message pending interrupt enable */
CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
```

在主程序中，开启了FIFO的中断响应使能以后，程序调用了CAN_Transmit函数发送CAN数据帧，由于CAN工作与环回模式，发送的数据帧环回回CAN接口，在中断服务程序中，对接收的帧进行处理。

```
/* CAN FIFO0 message pending interrupt enable */
CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);

/* transmit 1 message */
TxMessage.StdId=0x00;
TxMessage.ExtId=0x1234;
TxMessage.IDE=CAN_ID_EXT;
TxMessage.RTR=CAN_RTR_DATA;
TxMessage.DLC=2;
TxMessage.Data[0]=0xDE;
TxMessage.Data[1]=0xCA;
CAN_Transmit(CAN1, &TxMessage);

/* initialize the value that will be returned */
ret = 0xFF;

/* receive message with interrupt handling */
i=0;
while( (ret == 0xFF) && (i < 0xFFFF) )
{
    i++;
}

if (i == 0xFFFF)
{
    ret=0;
}

/* disable interrupt handling */
CAN_ITConfig(CAN1, CAN_IT_FMP0, DISABLE);
```

注意函数CAN_Interrupt(void)的最后要关中断呢？

因为一旦往FIFO存入1个报文，硬件就会更新FMP[1:0]位，并且如果CAN_IER寄存器的FMPIE位为1，那么就会产生一个中断请求。所以中断函数执行完后就要清除FMPIE标志位。

中断服务程序位于stm32f10x_it.c文件中，具体的函数实现为

```
#ifndef STM32F10X_CL
void USB_LP_CAN1_RX0_IRQHandler(void)
#else
void CAN1_RX0_IRQHandler(void)
#endif
{
    CanRxMsg RxMessage;

    RxMessage.StdId=0x00;
    RxMessage.ExtId=0x00;
    RxMessage.IDE=0;
    RxMessage.DLC=0;
    RxMessage.FMI=0;
    RxMessage.Data[0]=0x00;
    RxMessage.Data[1]=0x00;

    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);

    if( (RxMessage.ExtId==0x1234) && (RxMessage.IDE==CAN_ID_EXT)
        && (RxMessage.DLC==2) && ((RxMessage.Data[1] | RxMessage.Data[0]<<8)==0xDECA) )
    {
        ret = 1;
    }
    else
    {
        ret = 0;
    }
} ? end USB_LP_CAN1_RX0_IRQHandler ?
```

当FIFO产生中断时，处理器将自动进入中断服务程序。并对数据正确性进行判断，并返回对应的状态。

7.27.9 下载与测试

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.27.10 实验现象

首先将固件分别下载到神舟III号STM32开发板，上电运行开发板，正常情况下，LED指示灯指示程序运行结果。

| 现象 | 含义操作 |
|------|---------------|
| DS1亮 | 轮询模式CAN总线接收成功 |
| DS2亮 | 中断模式CAN总线接收成功 |
| DS3亮 | 轮询模式CAN总线接收失败 |
| DS4亮 | 中断模式CAN总线接收失败 |

7.28 双CAN收发测试实验

这一节我们将向大家介绍STM32的CAN总线的基本使用。有了STM32，CAN总线将变得简单，俗话说“百闻不如一见”，应当再加上“百见不如一试”。**神舟III号开发板上板载有1个CAN总线接口，在本小节，需要2块开发板，来进行CAN通信。可以通过神舟III号的串口指示CAN总线数据收发结果。**本节分为如下几个部分：

- 1 双CAN收发测试实验的意义与作用
- 2 实验原理
- 3 软件设计
- 4 硬件设计
- 5 下载与验证
- 6 实验现象

7.28.1 意义与作用

关于CAN的相关知识请参阅 [错误！未找到引用源。](#) 章节。不同之是本实验是两个物理的CAN接口之间真实的通信。

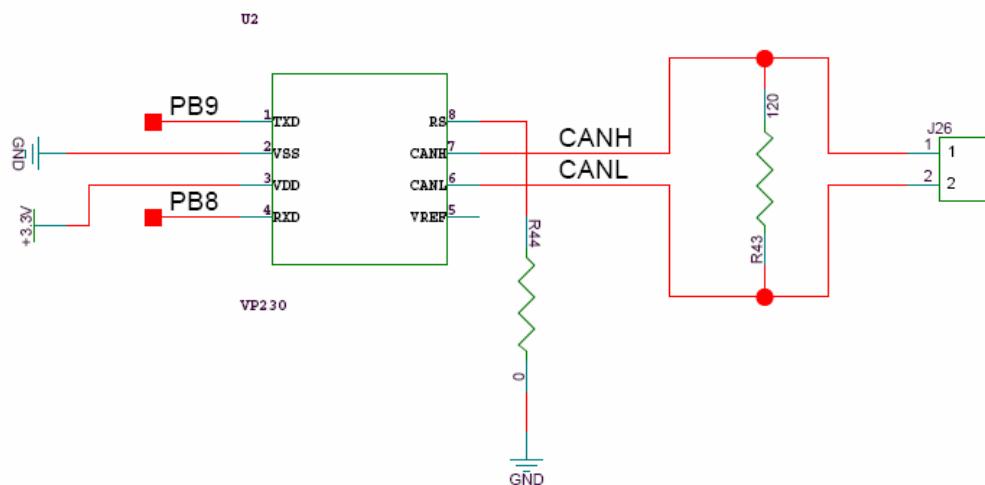
7.28.2 实验原理

关于CAN的实验原理相关知识请参阅 [错误！未找到引用源。](#) 章节。不同之是本实验是两个物理的CAN接口之间真实的通信。

7.28.3 硬件设计

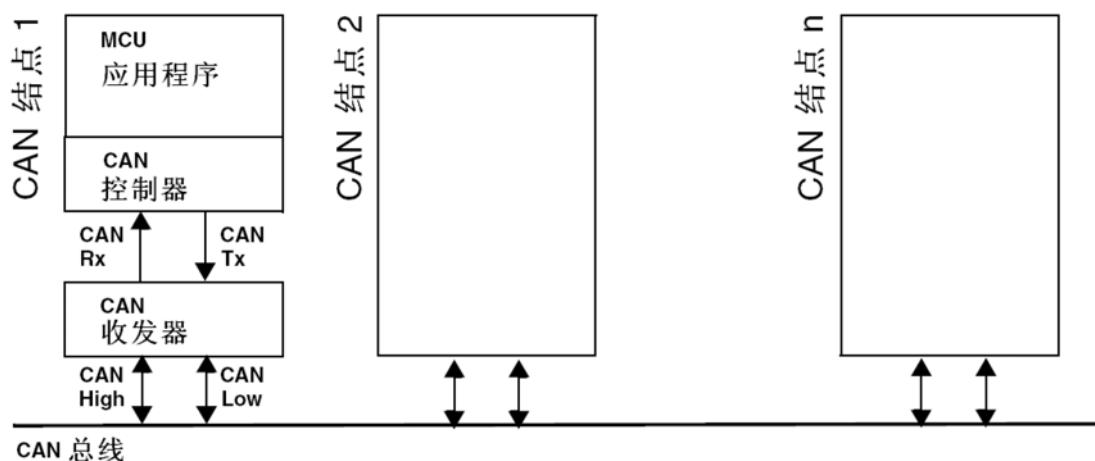
关于CAN的实验硬件电路相关知识请参阅 [错误！未找到引用源。](#) 章节。不同之是本实验是两个物理的CAN接口之间真实的通信。

在神舟 III 号开发板中，我们使用了处理器 CAN 外扩 TI 的 VP230 CAN 总线收发芯片来实现 CAN 总线接口。



STM32F103ZET6 这款处理器集成了 CAN 总线接口，在开发板上我们使用了 TI 公司的 3.3V 电压的 CAN 总线收发器来实现 CAN 物理层，如下图所示。CAN 总线收发器型号为 VP230。

CAN 网拓扑结构：



7.28.4 软件设计

本实验有 2 个工程源码，分别对应 2 块开发板，2 块开发板分别是主机、从机。我们这里分析主机的程序。看一下 main 函数：

```

21 int main(void)
22 {
23     /* 初始化串口模块 */
24     USART1_Config();
25
26     /* 配置CAN模块 */
27     CAN_Config();
28
29     printf( "\r\n***** 这是一个双CAN通讯实验***** \r\n");
30     printf( "\r\n这是“主机端”的反馈信息: \r\n");
31
32     /* 设置要通过CAN发送的信息 */
33     CAN_SetMsg();
34
35     printf("\r\n将要发送的报文内容为: \r\n");
36     printf("\r\n 扩展ID号ExtId: 0x%x",TxMessage.ExtId);
37     printf("\r\n 数据段的内容:Data[0]=0x%x , Data[1]=0x%x \r\n",TxMessage.Data[0],
38
39     /*发送消息 “ABCD”*/
40     CAN_Transmit(CAN1, &TxMessage);
41
42
43     while( flag == 0xff );                                //flag =0 ,success
44
45     printf( "\r\n 成功接收到“从机”返回的数据\r\n ");
46     printf(" 接收到的报文为: \r\n");
47     printf(" 扩展ID号ExtId: 0x%x",RxMessage.ExtId);
48     printf(" 数据段的内容:Data[0]= 0x%x , Data[1]=0x%x \r\n",RxMessage.Data[0],
49
50     while(1);
51
52

```

主函数中，首先是对串口进行了配置。然后，调用函数 CAN_Config()，配置 CAN 模块。然后调用 printf 打印信息。通过 CAN_SetMsg()函数设置要发送的信息。通过函数 CAN_Transmit(CAN1, &TxMessage)向从机发送数据。

代码分析 1:: CAN_Config()

```

132 void CAN_Config(void)
133 {
134     CAN_GPIO_Config();
135     CAN_NVIC_Config();
136     CAN_Mode_Config();
137     CAN_Filter_Config();
138 }

```

本次实验与上一个实验不同之处在于，本次实验是硬件物理上实实在在的连接。所以需要配置初始化对应的硬件管脚如下：

```

024 static void CAN_GPIO_Config(void)
025 {
026     GPIO_InitTypeDef GPIO_InitStructure;
027
028     /*外设时钟设置*/
029     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOB, ENABLE);
030     RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
031
032     /*IO设置*/
033     GPIO_PinRemapConfig(GPIO_Remap1_CAN1, ENABLE);                                         // PB8
034     /* Configure CAN pin: RX */                                                               // 上拉输入
035     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
036     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
037     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
038     GPIO_Init(GPIOB, &GPIO_InitStructure);
039     /* Configure CAN pin: TX */                                                               // PB9
040     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
041     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;                                         // 复用推挽输出
042     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
043     GPIO_Init(GPIOB, &GPIO_InitStructure);
044
045 }

```

对应的管脚和功能硬件时钟使能配置如下：

```
/*外设时钟设置*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOB, ENABLE);
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
```

CAN_Mode_Config() 函数中，我们首先对 CAN 接口进行配置，其中 CAN_Init 函数根据 CAN_InitStruct 中指定的参数初始化 CAN 寄存器，相关代码如下：

```
074 static void CAN_Mode_Config(void)
075 {
076     CAN_InitTypeDef      CAN_InitStructure;
077     /******CAN通信参数设置*****
078     /*CAN寄存器初始化*/
079     CAN_DeInit(CAN1);
080     CAN_StructInit(&CAN_InitStructure);
081     /*CAN单元初始化*/
082     CAN_InitStructure.CAN_TTCM=DISABLE;
083     CAN_InitStructure.CAN_ABOM=ENABLE;
084     CAN_InitStructure.CAN_AWUM=ENABLE;
085     CAN_InitStructure.CAN_NART=DISABLE;
086     CAN_InitStructure.CAN_RFLM=DISABLE;
087     CAN_InitStructure.CAN_TXFP=DISABLE;
088     CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
089     CAN_InitStructure.CAN_SJW=CAN_SJW_2tq;
090     CAN_InitStructure.CAN_BS1=CAN_BS1_6tq;
091     CAN_InitStructure.CAN_BS2=CAN_BS2_3tq;
092     CAN_InitStructure.CAN_Prescaler =4;           // 
093     CAN_Init(CAN1, &CAN_InitStructure);
094 }
```

CAN_InitStruct 结构体的成员说明见下表：

| | |
|---------------|--|
| CAN_TTCM | CAN_TTCM 用来使能或去使能时间触发通讯模式 |
| CAN_ABOM | CAN_ABOM 用来使能或者去使能自动离线管理 |
| CAN_AWUM | CAN_AWUM 用来使能或者去使能自动唤醒模式 |
| CAN_NART | CAN_NART 用来使能或者去使能非自动重传模式 |
| CAN_RFLM | CAN_RFLM 用来使能或者去使能 FIFO 锁定模式 |
| CAN_TXFP | CAN_TXFP 用来使能或者去使能发送 FIFO 优先级 |
| CAN_MODE | 设置 CAN 工作模式，可设置为正常工作模式 (CAN_Mode_Normal) 静默模式 (CAN_Mode_Silent), 环回模式 (CAN_Mode_LoopBack) 和静默环回模式 (CAN_Mode_Silent_LoopBack) |
| CAN_SJW | 定义重新同步跳跃宽度，即每位中可以延长或缩短多少个时间单位的上限，可设置为 1~4 个时间单位 (CAN_SJW_1tq~ CAN_SJW_4tq) |
| CAN_BS1 | 设定时间段 1 的时间单位数目，可设置为 1~16 个时间单位 (CAN_BS1_1tq~ CAN_BS1_16tq)。 |
| CAN_BS2 | 设定时间段 2 的时间单位数目，可设置为 1~16 个时间单位 (CAN_BS1_1tq~ CAN_BS1_16tq)。 |
| CAN_Prescaler | 设定一个时间单位的长度，它的范围是 1~1024 |

在完成 CAN 的基本参数配置以后，我们还需要对滤波器的参数进行配置，这个主要是通过 CAN_Filter_Config() 函数实现的，相关代码如下：

```

103 static void CAN_Filter_Config(void)
104 {
105     CAN_FilterInitTypeDef  CAN_FilterInitStructure;
106
107     /*CAN过滤器初始化*/
108     CAN_FilterInitStructure.CAN_FilterNumber=0;
109     CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
110     CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;
111     /* 使能报文标志符过滤器按照标志符的内容进行比对过滤，扩展ID不支持 */
112
113     CAN_FilterInitStructure.CAN_FilterIdHigh= (((u32)0x1314<<3)&0x
114     CAN_FilterInitStructure.CAN_FilterIdLow= (((u32)0x1314<<3)|CAN
115     CAN_FilterInitStructure.CAN_FilterMaskIdHigh= 0xFFFF;
116     CAN_FilterInitStructure.CAN_FilterMaskIdLow= 0xFFFF;
117     CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO1;
118     CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;
119     CAN_FilterInit(&CAN_FilterInitStructure);
120     /*CAN通信中断使能*/
121     CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
122 }

```

| | |
|--------------------------|---|
| CAN_FilterNumber | 指定待初始化的过滤器，它的范围是1到13 |
| CAN_FilterMode | 指定过滤器将被初始化的模式，可设置为标识符屏蔽位模式（CAN_FilterMode_IdMask）或者标识符列表模式（CAN_FilterMode_IdList） |
| CAN_FilterScale | 过滤器位宽，可设置为2个16位过滤器（CAN_FilterScale_Two16bit）或者1个32位过滤器（CAN_FilterScale_One32bit）。 |
| CAN_FilterIDHigh | 用来设定过滤器标识符（32位位宽时为其高段位，16位宽时为第一个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterIDLow | 用来设定过滤器标识符（32位位宽时为其低段位，16位宽时为第二个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterMaskIDHigh | 用来设定过滤器屏蔽标识符或者过滤器标识符（32位位宽时为其高段位，16位宽时为第一个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterMaskIDLow | 用来设定过滤器屏蔽标识符或者过滤器标识符（32位位宽时为其低段位，16位宽时为第二个）。它的范围是0x0000到0xFFFF。 |
| CAN_FilterFIFOAssignment | 设置指向过滤器的FIFO（0或者1） |
| CAN_FilterActivation | 使能或者去使能过滤器 |

CAN 总线中断配置

在本实验中，通过中断方式进行CAN总线的数据收发实验。这个是主机，主机发送数据，也可以接受从机发出的数据。我们需要设置CAN接口的接收终端及其优先级，相关代码如下：

```

054 static void CAN_NVIC_Config(void)
055 {
056     NVIC_InitTypeDef NVIC_InitStructure;
057     /* Configure one bit for preemption priority */
058     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
059     /*中断设置*/
060     NVIC_InitStructure.NVIC IRQChannel = USB_LP_CAN1_RX0_IRQn;
061     NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
062     NVIC_InitStructure.NVIC IRQChannelSubPriority = 0;
063     NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
064     NVIC_Init(&NVIC_InitStructure);
065 }

```

CAN1的中断接收服务程序具体实现为：

```

148 void USB_LP_CAN1_RX0_IRQHandler(void)
149 {
150
151 /*从邮箱中读出报文*/
152 CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
153
154 /* 比较ID和数据是否为0x1314及DCBA */
155 if((RxMessage.ExtId==0x1314) && (RxMessage.IDE==CAN_ID_EXT)
156 && (RxMessage.DLC==2) && ((RxMessage.Data[1]|RxMessage.Data[0]<<8)==0xDCBA))
157 {
158     flag = 0;                                //接收成功
159 }
160 else
161 {
162     flag = 0xff;                            //接收失败
163 }
164 }

```

当 CAN 收到数据时产生中断，中断服务程序被自动调用。经过判断，接收完成则将 flag 赋值为 0；接收失败则将 flag 赋值为 0xff。

代码分析2：

前面，我们完成了CAN总线接口的参数配置和滤波器配置，正常情况下，我们就可以使用CAN总线接口了，最后初始化发送数据的部分内容，**本次实验者这些内容在每次发送时都相同，只留下Data部分在发送的时候填充。**

```

148 void CAN_SetMsg(void)
149 {
150     //TxMessage.StdId=0x00;
151     TxMessage.ExtId=0x1314;
152     TxMessage.IDE=CAN_ID_EXT;
153     TxMessage.RTR=CAN_RTR_DATA;
154     TxMessage.DLC=2;
155     TxMessage.Data[0]=0xAB;
156     TxMessage.Data[1]=0xCD;
157 }

```

它的相关参数说明如下：

| | |
|---------|---|
| StdId | 用来设定标准标识符。可设置为0到0x7FF |
| ExtId | 用来设定扩展标识符，可设置为0到0x3FFF |
| IDE | 用来设定消息标识符的类型，可设置为使用标准标识符（CAN_ID_STD）或者使用标准标识符+扩展标识符（CAN_ID_EXT） |
| RTR | 设定待传输消息的帧类型，可设置为数据帧（CAN_RTR_DATA）或者远程帧（CAN_RTR_REMOTE） |
| DLC | 用来设定带传输消息的帧长度。取值范围0到0x8 |
| Data[8] | 待传输的数据 |

代码分析3：CAN_Transmit(CAN1, &TxMessage)

该函数是 ST 给我们提供的库函数，详细内容参考提供的帮助文档。

代码分析4：当正确接收到从机的数据时，通过串口显示。

```

while( flag == 0xff );                                //flag =0 ,success

printf( "\r\n 成功接收到“从机”返回的数据\r\n" );
printf(" \r\n 接收到的报文为: \r\n");
printf("\r\n 扩展ID号ExtId: 0x%x", RxMessage.ExtId);
printf("\r\n 数据段的内容:Data[0]= 0x%x , Data[1]=0x%x \r\n",)
while(1);

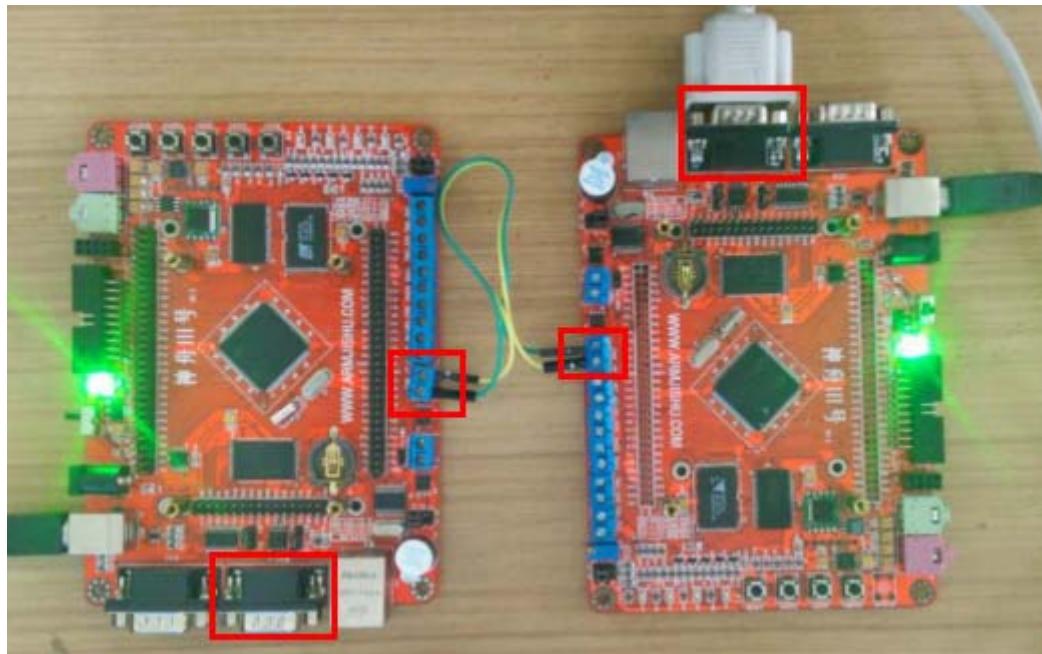
```

7.28.5 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.28.6 实验现象

本实验使用到 2 个开发板。用 1 根连根杜邦线，将开发板 1 的 CANH 接开发板 2 的 CANH，另一根将将开发板 1 的 CANL 接开发板 2 的 CANL，如下图：



本例程分，主机程序，和从机程序。分别将 2 个程序下载到 2 个开发板上。

将串口接到从机上，按下从机复位键：



按下主机复位键：



将串口接到到主机上，按一下从机的复位键，再按一下主机的复位键。



7.29 SPI FLASH读写实验

本节将利用SPI来实现对神舟III号板载的FLASH（W25X16）的读写，并将结果通过串口显示在PC机上。后续升级，便将显示结果从LCD屏上显示出来。

7.29.1 SPI FLASH（W25X16）读写程序实验的意义与作用

SPI总线是Motorola公司推出的三线同步接口，主要应用在FLASH, EEPROM以及一些数字通信中。

神舟III号硬件上使用到SPI接口的有：触摸屏，音频DA芯片PCM1770，W25X16，ENC28J60以太网芯片以及2.4G无线模块。

SPI总线接口作为一种非常基本的外设接口，但是其应用却是很广泛。通过本例程SPI对W25X16的读写实验，让大家简单了解SPI的通信原理。

7.29.2 SPI的介绍

什么是 SPI

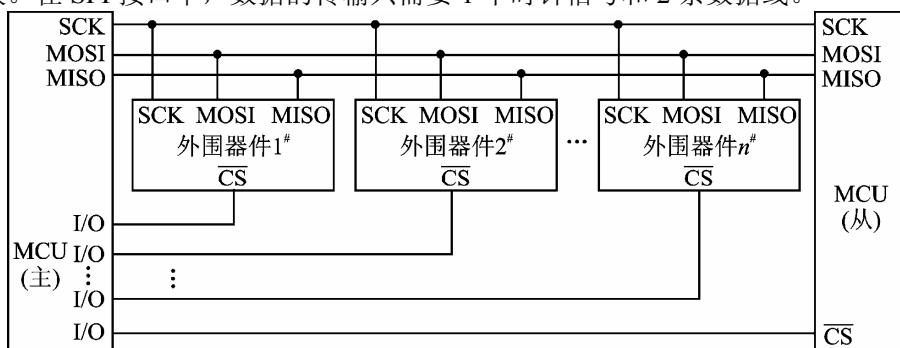
SPI(Serial Peripheral Interface——串行外设接口)总线是 Motorola 公司推出的一种同步串行外设接口，它用于 MCU 与各种外围设备以串行方式进行通信。

SPI 的通信方式

SPI 用于 MCU 与各种外围设备以串行方式进行通信（8 位数据同时同步地被发送和接收），系统可配置为主或从操作模式。是一种高速的，全双工，同步的通信总线

SPI 的连接方式

SPI 是一个环形总线结构，在芯片的管脚上只占用四根线，节约了芯片的管脚，由串行时钟线（SCK）、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 CS(NSS)。构成，其时序其实很简单，主要是在 SCK 的控制下，两个双向移位寄存器进行数据交换。在 SPI 接口中，数据的传输只需要 1 个时钟信号和 2 条数据线。



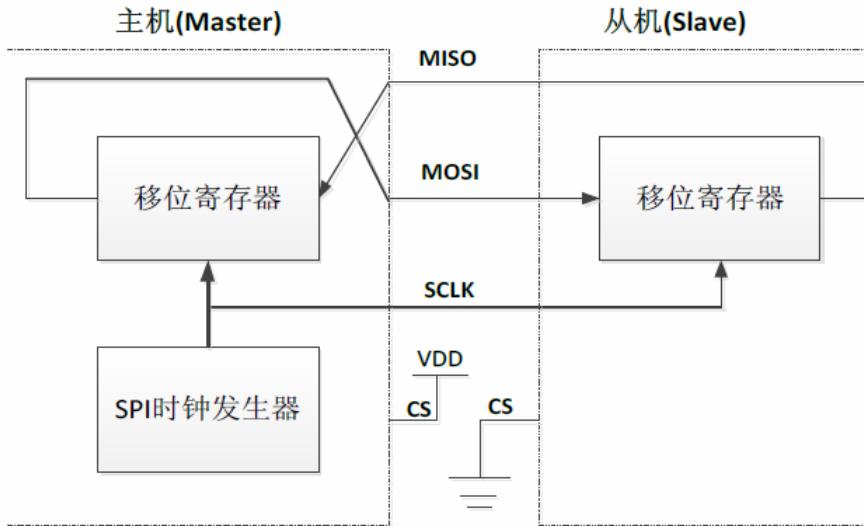
MISO: 主设备数据输入，从设备数据输出。

MOSI: 主设备数据输出，从设备数据输入。

SCK: 时钟信号，由主设备产生。

CS (NSS): 从设备选择。这是一个可选的引脚，用来选择主/从设备。它的功能是用来作为“片选引脚”，让主设备可以单独地与特定从设备通讯，避免数据线上的冲突。从设备的NSS引脚可以由主设备的一个标准I/O引脚来驱动。一旦被使能(SSOE位)，NSS引脚也可以作为输出引脚，并在SPI处于主模式时拉低；此时，所有的SPI设备，如果它们的NSS引脚连接到主设备的NSS引脚，则会检测到低电平，如果它们被设置为NSS硬件模式，就会自动进入从设备状态。当配置为主设备、NSS配置为输入引脚(MSTR=1, SSOE=0)时，如果NSS被拉低，则这个SPI设备进入主模式失败状态：即MSTR位被自动清除，此设备进入从模式。

SPI 的内部结构



SPI 内部结构简明图

从图中可以看出，主机和从机都有一个串行移位寄存器，主机通过向它的 SPI 串行寄存器写入一个字节来发起一次传输。寄存器通过 MOSI 信号线将字节传送给从机，从机也将自己的移位寄存器中的内容通过 MISO 信号线返回给主机。这样，两个移位寄存器中的内容就被交换。外设的写操作和读操作是同步完成的。如果只进行写操作，主机只需忽略接收到的字节；反之，若主机要读取从机的一个字节，就必须发送一个空字节来引发从机的传输。

SPI 总线的特点

SPI 主要特点有：

可以同时发出和接收串行数据；可以当作主机或从机工作；提供频率可编程时钟；发送结束中断标志；写冲突保护；总线竞争保护等。

由于 SPI 系统总线只需 3~4 位数据线和控制线即可扩展具有 SPI 的各种 I/O 器件，而并行总线扩展方法需 8 根数据线、8~16 位地址线、2~3 位控制线，因而 SPI 总线的使用可以简化电路设计，省掉了很多常规电路中的接口器件，提高了设计的可靠性。

SPI 总线的特性

STM32 集成的 SPI 接口特征如下：

- 3 线全双工同步传输
- 带或不带第三根双向数据线的双线单工同步传输
- 8 或 16 位传输帧格式选择
- 主或从操作
- 支持多主模式
- 8 个主模式波特率预分频系数(最大为 fPCLK/2)
- 从模式频率 (最大为 fPCLK/2)
- 主模式和从模式的快速通信
- 主模式和从模式下均可以由软件或硬件进行 NSS 管理：主/从操作模式的动态改变
- 可编程的时钟极性和相位
- 可编程的数据顺序，MSB 在前或 LSB 在前
- 可触发中断的专用发送和接收标志
- SPI 总线忙状态标志
- 支持可靠通信的硬件 CRC
 - 在发送模式下，CRC 值可以被作为最后一个字节发送
 - 在全双工模式中对接收到的最后一个字节自动进行 CRC 校验
- 可触发中断的主模式故障、过载以及 CRC 错误标志
- 支持 DMA 功能的 1 字节发送和接收缓冲器：产生发送和接受请求

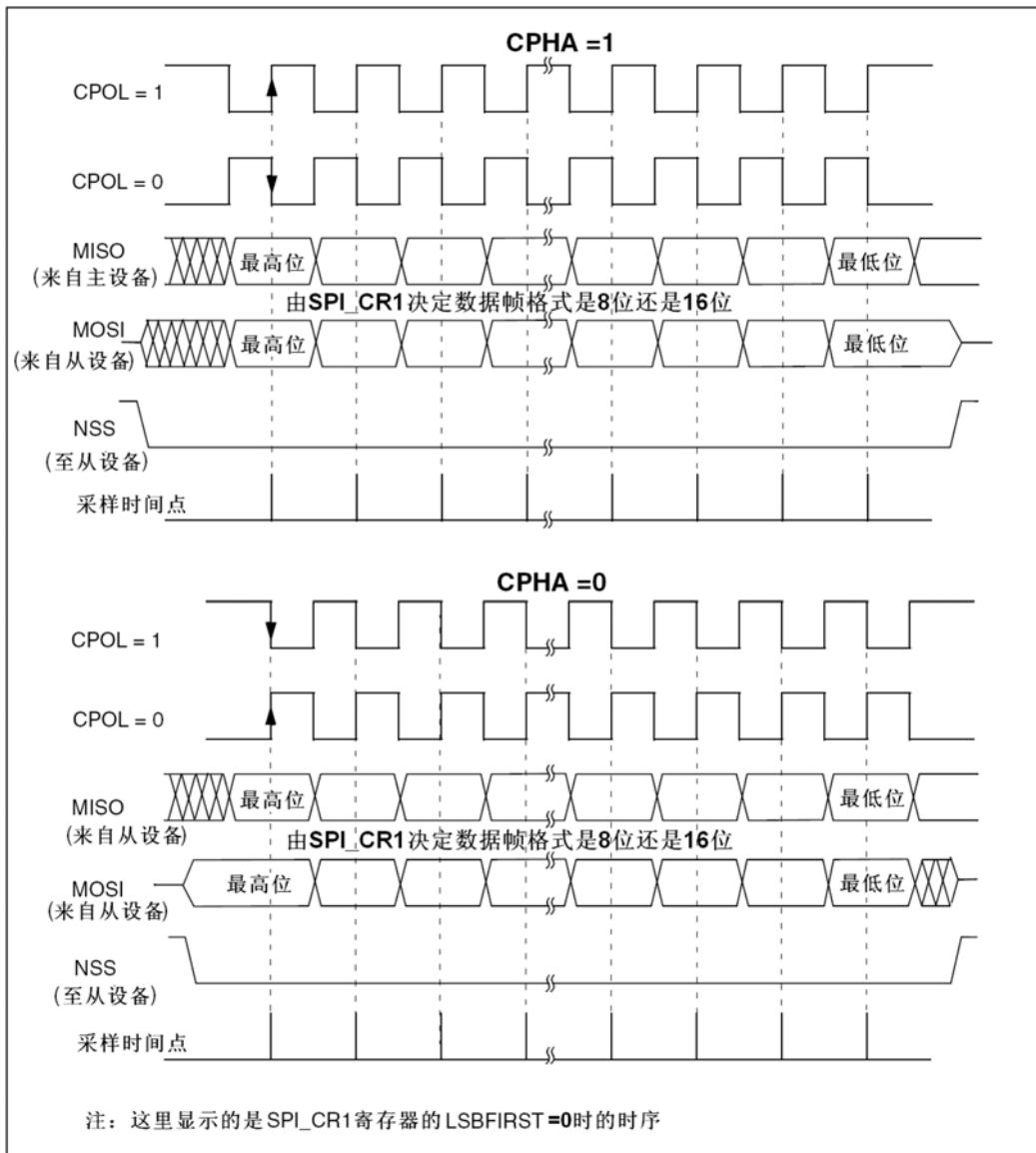
时钟信号的相位和极性

SPI_CR寄存器的CPOL和CPHA位，能够组合成四种可能的时序关系。CPOL(时钟极性)位控制在没有数据传输时时钟的空闲状态电平，此位对主模式和从模式下的设备都有效。如果CPOL被清'0'，SCK引脚在空闲状态保持低电平；如果CPOL被置'1'，SCK引脚在空闲状态保持高电平。

如果CPHA(时钟相位)位被置'1'，SCK时钟的第二个边沿(CPOL位为0时就是下降沿，CPOL位为'1'时就是上升沿)进行数据位的采样，数据在第二个时钟边沿被锁存。如果CPHA位被清'0'，SCK时钟的第一边沿(CPOL位为'0'时就是上升沿，CPOL位为'1'时就是下降沿)进行数据位采样，数据在第一个时钟边沿被锁存。

SPI总线工作方式

根据CPOL时钟极性和CPHA时钟相位的组合，选择数据捕捉的时钟边沿，SPI接口有四种不同的数据传输时序。下图显示了SPI传输的4种CPHA和CPOL位组合。此图可以解释为主设备和从设备的SCK脚、MISO脚、MOSI脚直接连接的主或从时序图。



CPOL时钟极性和CPHA时钟相位的组合选择数据捕捉的时钟边沿。上图显示了SPI传输的4种CPHA和CPOL位组合。此图可以解释为主设备和从设备的SCK脚、MISO脚、MOSI脚直接连接的主或从时序图。

注意：

1. 在改变CPOL/CPHA位之前，必须清除SPE位将SPI禁止。
2. 主和从必须配置成相同的时序模式。
3. SCK的空闲状态必须和SPI_CR1寄存器指定的极性一致(CPOL为'1'时，空闲时应上拉SCK为高

电平；CPOL为'0'时，空闲时应下拉SCK为低电平)。

4. 数据帧格式(8位或16位)由SPI_CR1寄存器的DFF位选择，并且决定发送/接收的数据长度。

数据帧格式

根据SPI_CR1寄存器中LSBFIRST位，输出数据位时可以MSB在先也可以LSB在先。根据SPI_CR1寄存器的DFF位，每个数据帧可以是8位或是16位。所选择的数据帧格式对发送和/或接收都有效。

关于STM32的SPI详细资料请详见《【中文】STM32F系列ARM内核32位高性能微控制器参考手册V10_1.pdf》一文的第457页。

本例程中，我们采用STM32的SPI1作为主模式来读取外部SPI FLASH芯片（W25X16），实现读写功能。下面简单说明SPI1部分的配置情况：

在主配置时，在SCK脚输出串行时钟。

配置步骤：

- a) 配置 SPI 串行时钟波特率；
- b) 定义数据传输和串行时钟间的相位关系。
- c) 设置 8 位或 16 位数据帧格式；
- d) 如果需要 NSS 引脚工作在输入模式，硬件模式下，在整个数据帧传输期间应把 NSS 脚连接到高电平，在软件模式下，需设置 SPI_CR1 寄存器的 SSM 位和 SSI 位。如果 NSS 引脚工作在输出模式，则只需要设置 SSOE 位；
- e) 必须设置 MSTR 位和 SPE 位（只当 NSS 脚被连接到高电平，这些位才能保持置位）。

在这个配置中，MOSI引脚是数据输出，而MISO引脚是数据输入。

SPI芯片W25X16的介绍

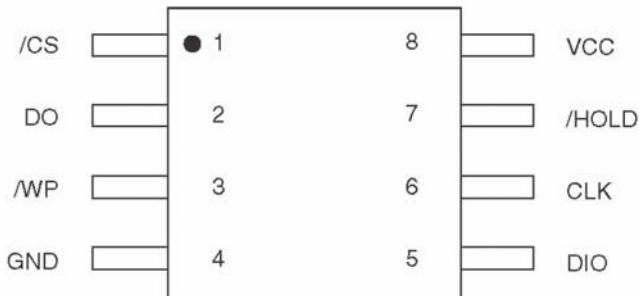
W25X16是华邦公司推出的容量为16Mb，也就是2M字节的芯片，容量大小跟AT45DB161是一样的。

W25X16芯片将2M的容量分为32个块（Block），每个块大小为64K字节，每个块又分为16个扇区（Sector），每个扇区4K个字节。W25X16的最少擦除单位为一个扇区，也就是每次必须擦除4K个字节。这样我们需要给W25X16开辟一个至少4K的缓存区。

W25X16的擦写周期为10000次，具有20年的数据保存期限，支持电压为2.7~3.6V，W25X16支持标准的SPI，还支持双输出的SPI，最大SPI时钟可以到75Mhz(双输出时相当于150Mhz)，详细的W25X16的介绍，请参考“...\\神舟III号光盘\\外围器件数据手册”文件下的《W25X16 SPI Flash数据手册.pdf》

7.29.3 硬件设计

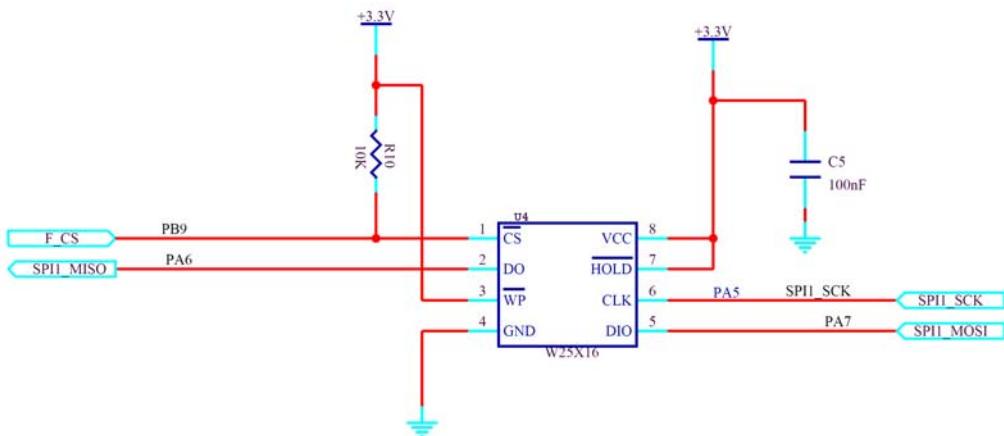
神舟III号开发板载有SPI FLASH芯片W25X16，该芯片的容量为2M字节(16MBit)，与AT45DB161属于同一级别，W25X16的管脚定义如下：



其管脚含义如下：

| PAD NO. | PAD NAME | I/O | FUNCTION |
|---------|----------|-----|---------------------|
| 1 | /CS | I | Chip Select Input |
| 2 | DO | O | Data Output |
| 3 | /WP | I | Write Protect Input |
| 4 | GND | | Ground |
| 5 | DIO | I/O | Data Input / Output |
| 6 | CLK | I | Serial Clock Input |
| 7 | /HOLD | I | Hold Input |
| 8 | VCC | | Power Supply |

根据以上定义，神舟III号开发板与W25X16理解的原理图如下：



SPI FLASH

SPI FLASH电路原理图

7.29.4 软件设计

本例程在库文件的基础上，我们根据实际使用，还需要增加两个文件，一个是SPI外设驱动文件，另外一个则是W25X16芯片的驱动代码。

1、我们先看SPI外设驱动文件，包括spi.c文件和spi.h头文件。对于spi.h文件，主要是头文件的声明，以及SPI1初始化和读写一个字节的函数。而spi.c则是相关函数的具体实现，其中SPIx_init初始化SPI1接口为主模式读取FLASH芯片。

```

void SPIx_Init(void)
{
    SPI_InitTypeDef SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable SPI1 and GPIOA clocks */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1|RCC_APB2Periph_AFIO, ENABLE);

    /* Configure SPI1 pins: NSS, SCK, MISO and MOSI */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    //SPI1 NSS
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_SetBits(GPIOC, GPIO_Pin_4);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA, GPIO_Pin_4);

    /* SPI1 configuration */
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //SPI1设置为两线全双工
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //设置SPI1为主模式
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; //SPI发送接收8位帧结构
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High; //串行时钟在不操作时，时钟为高电平
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge; //第二个时钟沿开始采样数据
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //NSS信号由软件（使用SSI位）管理
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8; //定义波特率预分频的值：波特率预分频值为8
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //数据传输从MSB位开始
    SPI_InitStructure.SPI_CRCPolynomial = 7; //CRC值计算的多项式
    SPI_Init(SPI1, &SPI_InitStructure);
    /* Enable SPI1 */
    SPI_Cmd(SPI1, ENABLE); //使能SPI1外设
} ? end SPIx_Init ?

```

SPI接口读写FLASH的一个字节的函数，如下所示：

```

//SPIx 读写一个字节
//返回值：读取到的字节
u8 SPIx_ReadWriteByte(u8 TxData)
{
    u8 retry=0;
    while((SPI1->SR&1<<1)==0) //等待发送区空
    {
        retry++;
        if(retry>200) return 0;
    }
    SPI1->DR=TxData; //发送一个byte
    retry=0;
    while((SPI1->SR&1<<0)==0) //等待接收完一个byte
    {
        retry++;
        if(retry>200) return 0;
    }
    return SPI1->DR; //返回收到的数据
}

```

2、我们接着介绍W25X16芯片的驱动文件，包括flash.c和flash.h两个文件。其中flash.h文件的内容，主要包括FLASH的CS，W25X16相关的一些指令设置以及flash.c文件中包括的一些函数声明，而flash.c文件的主要内容便是关于SPI对FLASH的读操作，写操作函数，其中包括SPI读取flash ID的函数。其中主要的几个函数是描述如下：

◆ 获取SPI FLASH ID

```
u16 SPI_Flash_ReadID (void)
{
    u16 Temp = 0;
    SPI_FLASH_CS=0;
    SPIx_WriteByte (0x90); //发送读取ID命令
    SPIx_WriteByte (0x00);
    SPIx_WriteByte (0x00);
    SPIx_WriteByte (0x00);
    Temp|=SPIx_WriteByte (0xFF)<<8;
    Temp|=SPIx_WriteByte (0xFF);
    SPI_FLASH_CS=1;
    return Temp;
}
```

◆ SPI FLASH 读函数

在指定地址开始读取指定长度的数据。

```
void SPI_Flash_Read (u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
{
    u16 i;
    SPI_FLASH_CS=0; //使能器件
    SPIx_WriteByte (W25X_ReadData); //发送读取命令
    SPIx_WriteByte ((u8)((ReadAddr)>>16)); //发送24bit地址
    SPIx_WriteByte ((u8)((ReadAddr)>>8));
    SPIx_WriteByte ((u8)ReadAddr);
    for (i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPIx_WriteByte (0xFF); //循环读数
    }
    SPI_FLASH_CS=1; //取消片选
}
```

◆ SPI FLASH 写函数

往指定地址写入指定长度的数据。

```
void SPI_Flash_Write(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
    u32 secpos;
    u16 secoff;
    u16 secremain;
    u16 i;
    secpos=WriteAddr/4096; //扇区地址 0~511 for w25x16
    secoff=WriteAddr%4096; //在扇区内的偏移
    secremain=4096-secoff; //扇区剩余空间大小
    if(NumByteToWrite<=secremain) secremain=NumByteToWrite; //不大于4096个字节
    while(1)
    {
        SPI_Flash_Read(SPI_FLASH_BUF, secpos*4096, 4096); //读出整个扇区的内容
        for(i=0;i<secremain;i++) //校验数据
        {
            if(SPI_FLASH_BUF[secoff+i]!=0xFF) break; //需要擦除
        }
        if(i<secremain) //需要擦除
        {
            SPI_Flash_Erase_Sector(secpos); //擦除这个扇区
            for(i=0;i<secremain;i++) //复写
            {
                SPI_FLASH_BUF[i+secoff]=pBuffer[i];
            }
            SPI_Flash_Write_NoCheck(SPI_FLASH_BUF, secpos*4096, 4096); //写入整个扇区
        }
        else SPI_Flash_Write_NoCheck(pBuffer, WriteAddr, secremain); //写已经擦除了的,直接写入扇区剩余区间.
        if(NumByteToWrite==secremain) break; //写入结束了
        else //写入未结束
        {
            secpos++; //扇区地址增1
            secoff=0; //偏移位置为0
            pBuffer+=secremain; //指针偏移
            WriteAddr+=secremain; //写地址偏移
            NumByteToWrite-=secremain; //字节数递减
            if(NumByteToWrite>4096) secremain=4096; //下一个扇区还是写不完
            else secremain=NumByteToWrite; //下一个扇区可以写完了
        }
    }
} ? end while 1 ?
} ? end SPI_Flash_Write ?
```

3、下面将对主函数进行简单分析。本例程还是将从串口打印信息，点灯操作，按键配置，按键扫描，这些在之前例程中已经讲解了，在此不赘述。那么主程序中，上电初始化操作后，包括前面提到的一些配置后，将开始读取Flash的ID，作为Flash是否存在，或是其他异常问题的判断。如果读取到ID，说明Flash在位，否则，不在位。当Flash准备好后，我们将通过按钮1，将由SPI1接口，把提前设置好的字符组“const u8 TEXT_Buffer[]={"神舟III号 SPI 读写访问程序"}”中的数据写到Flash W25X16中；然后，我们通过按钮2的检测，将写到W25X16的字符组读取出来，并在串口显示出来。下面简单了解一下代码，详细的代码设计，请详阅main()函数。

上电后，读取Flash的ID号，作为在位判断，如果不在位，则循环打印“check failed”等。否则准备就绪，并提示写flash或是读flash操作。

```
while(SPI_Flash_ReadID()!=FLASH_ID) //检测不到W25X16
{
    i=SPI_Flash_ReadID();
    printf("\n\r ID: %d", i);
    printf("\n\r 没有读到正确的W25X16芯片ID, 请检查硬件连接");
    Delay(0xFFFF);
    Delay(0xFFFF);
    GPIO_ResetBits(GPIOF, GPIO_Pin_7);
    GPIO_SetBits(GPIOF, GPIO_Pin_7);
}
```

循环检测按键是否按下，根据不同的按键值，进行不同的操作。同时启动LED1灯的闪烁，表示系统正在运行中。

```
while(1)
{
    key=ReadKeyDown();
    if(key==KEY1)//KEY1按下,写入SPI FLASH
    {
        printf("\n\r开始写入W25X16 SPI FLASH芯片....");
        SPI_Flash_Write((u8*)TEXT_Buffer,1000,SIZE); //从1000字节处开始,写入SIZE长度的数据
        printf("\n\r写入完成!");
    }
    if(key==KEY2)//KEY2按下,读取写入的字符传字符串并显示
    {
        printf("\n\r开始从W25X16 SPI FLASH芯片读取数据.... ");
        SPI_Flash_Read(datatemp,1000,SIZE); //从1000地址处开始,读出SIZE个字节
        printf("\n\r读取完成, 读出的数据为: %s ",datatemp); //提示传送完成
    }

    i++;
    Delay(0xFFFF);
    if(i>0&&i<100)
    {
        GPIO_SetBits(GPIOF, GPIO_Pin_6); //提示系统正在运行
    }
    else if(i >= 100 && i < 200)
    {
        GPIO_ResetBits(GPIOF, GPIO_Pin_6); //提示系统正在运行
    }
    i = i % 200;
} ? end while 1 ?
```

7.29.5 下载与测试

如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.29.6 实验现象

下载固件后，连接串口到PC机，按照要求设置波特率为115200，上电运行神舟III号，串口将输出提示信息，按下USER2键，往SPI FLASH写入一串数据，按下USER1按键，将从SPI FLASH读出之前写入的一串数据，在正常情况下，数据为“**神舟III号 SPI 读写访问程序**”，具体串口提示信息如下，请按提示信息按下USER1或USER2键进行SPI FLASH读写操作。

神舟III号

www.ARmjishu.COM! ##### (Dec 29 2010 - 22:08:26)



www.ARmjishu.COM use __STM32F10X_STDPERIPH_VERSION 3.3.0

产品内部Flash大小为：512K字节！ www.armjishu.com

--DS1闪烁：程序正常运行

--按USER2键：往SPI FLASH (W25x16) 写入一串字符

--按USER1键：从SPI FLASH (W25x16) 读出之前写入的字符

开始写入W25x16 SPI FLASH芯片.... 按USER2按键写入数据到SPI FLASH
写入完成！

开始从W25x16 SPI FLASH芯片读取数据.... 按USER1按键从SPI FLASH读出数据
读取完成，读出的数据为：神舟III号 SPI 读写访问程序

7.30 SRAM 访问实验

7.30.1 SRAM访问实验的意义与作用

SRAM 是英文 Static RAM 的缩写，它是一种具有静止存取功能的内存，不需要刷新电路即能保存它内部存储的数据。SRAM 的速度非常快，在快速读取和刷新时能够保持数据完整性。它的用途广泛，用于 CPU 内部的一级缓存以及内置的二级缓存，以及一些嵌入式设备，如网络服务器以及路由器等。

而 STM32 处理器具有 FSMC (Flexible Static Memory Controller) 总线，通过合适的参数配置，可支持不同的外部存储器类型。包括 SRAM, NorFlash, Nand Flash 等。

在本实验，将介绍如何通过 STM32 处理器的 FSMC 总线访问 SRAM。

STM32F103ZET6 自带了 64K 字节的 SRAM，对一般应用来说，已经足够了，不过在一些对内存要求高的场合，STM32 自带的这些内存就不够用了。比如跑算法或者跑 GUI 等，就可能不太够用，所以战舰 STM32 开发板板载了一颗 1M 字节容量的 SRAM 芯片：IS62WV25616，满足大内存使用的需求。

本章，我们将使用 STM32 来驱动 IS62WV25616，实现对 IS62WV25616 的访问控制，并测试其容量。

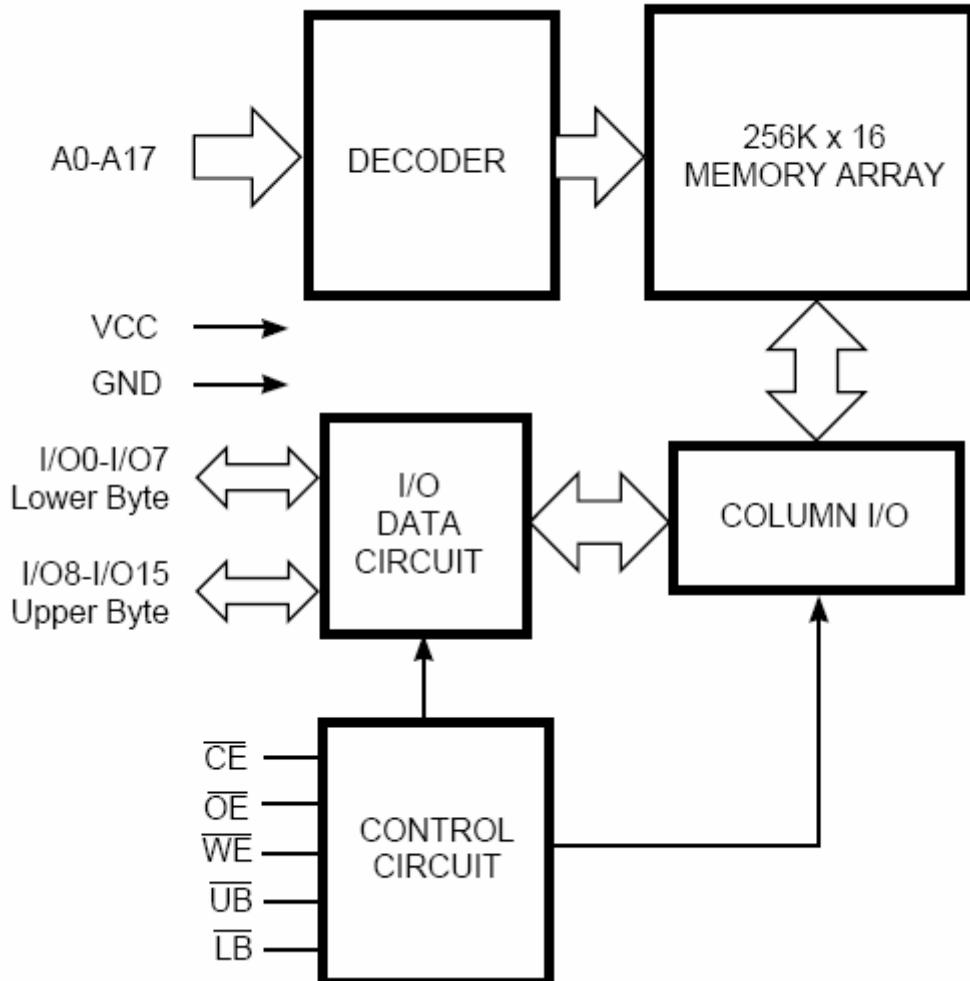
7.30.2 IS62WV25616简介

S62WV25616 是 ISSI (Integrated Silicon Solution, Inc) 公司生产的一颗 16 位宽 256K (256*16，即 512K 字节) 容量的 CMOS 静态内存芯片。该芯片具有以下几个特点：

- 高速。具有 45ns/55ns 访问速度
- 低功耗

- TTL 电平兼容
- 全静态操作。不需要刷新和时钟电路
- 三态输出
- 字节控制功能。支持高/低字节控制

IS62WV25616 的功能框图如图所示：



图中 A0~A17 为地址线，总共 18 根地址线（即 $2^{18}=256K$ ， $1K=1024$ ）；IO0~15 为数据线，总共 16 根数据线。CS2 和 CS1 都是片选信号，不过 CS2 是高电平有效 CS1 是低电平有效；OE 是输出使能信号（读信号）；WE 为写使能信号；UB 和 LB 分别是高字节控制和低字节控制信号；

在 STM32 神舟 III 号中，使用的 SRAM 的 IS61LV25616 这一有 ISSI 公司推出的 SRAM 芯片。IS61LV25616 容量为 4M 比特(如果需要配置更大容量的 SRAM，可以直接替换为 IS61LV51216，IS61LV51216 与 IS61LV25616 完全 pin-to-pin 兼容，而容量确是原来的一倍，达到 8M 比特)。

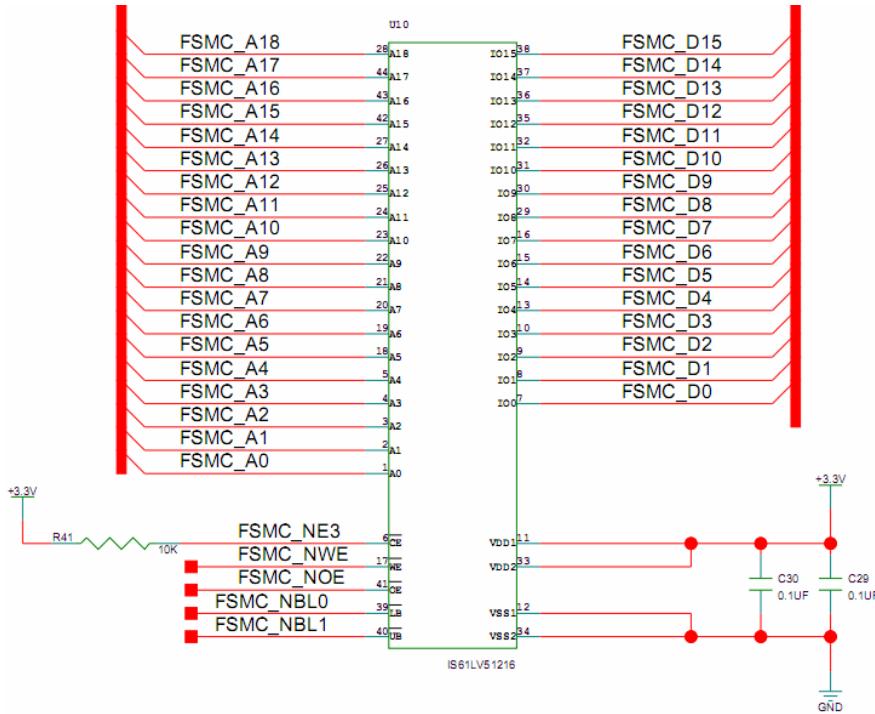
IS61LV25616 的管脚定义如下：

| | | | |
|------|----|----|------|
| A0 | 1 | 44 | A17 |
| A1 | 2 | 43 | A16 |
| A2 | 3 | 42 | A15 |
| A3 | 4 | 41 | OE |
| A4 | 5 | 40 | UB |
| CE | 6 | 39 | LB |
| I/O0 | 7 | 38 | VO15 |
| I/O1 | 8 | 37 | VO14 |
| I/O2 | 9 | 36 | VO13 |
| I/O3 | 10 | 35 | VO12 |
| Vcc | 11 | 34 | GND |
| GND | 12 | 33 | Vcc |
| I/O4 | 13 | 32 | VO11 |
| I/O5 | 14 | 31 | VO10 |
| I/O6 | 15 | 30 | VO9 |
| I/O7 | 16 | 29 | VO8 |
| WE | 17 | 28 | NC |
| A5 | 18 | 27 | A14 |
| A6 | 19 | 26 | A13 |
| A7 | 20 | 25 | A12 |
| A8 | 21 | 24 | A11 |
| A9 | 22 | 23 | A10 |

| | |
|------------|---------------------|
| A0-A17 | Address Inputs |
| I/O0-I/O15 | Data Inputs/Outputs |
| CE | Chip Enable Input |
| OE | Output Enable Input |
| WE | Write Enable Input |

| | |
|-----|---------------------------------|
| LB | Lower-byte Control (I/O0-I/O7) |
| UB | Upper-byte Control (I/O8-I/O15) |
| NC | No Connection |
| Vcc | Power |
| GND | Ground |

在神舟III号中，通过FSMC总线与IC61LV25616LL SRAM连接，具体电路如下：



在上图中，我们可以看到以下信息

从原理图可以看出，IS62WV25616同STM32的连接关系：

A[0:18]接FSMC_A[0:18]

D[0:15]接FSMC_D[0:15]

UB接FSMC_NBL1

LB接FSMC_NBL0

OE接FSMC_NOE

WE接FSMC_NWE

CE接FSMC_NE3

本章，我们使用FSMC的BANK1 区域3来控制IS62WV25616，关于FSMC的详细介绍，在之前的LCD章节

已经有过详细介绍，我们采用的是读写不同的时序来操作TFTLCD模块（因为TFTLCD模块读的速度比写的速度慢很多），但是在本章，因为IS62WV25616的读写时间基本一致，所以，我们设置读写相同的时序来访问FSMC。关于FSMC的详细介绍，请大家看《STM32参考手册》。

IS62WV25616就介绍到这，最后，我们来看看实现IS62WV25616的访问，需要对FSMC进行哪些配置。FSMC的详细配置介绍在之前的LCD实验章节已经有详细讲解，这里就做一个概括性的讲解。步骤如下：

1) 使能FSMC时钟，并配置FSMC相关的I0及其时钟使能。

要使用FSMC，当然首先得开启其时钟。然后需要把FSMC_D0~15, FSMCA0~18等相关I0口，全部配置为复用输出，并使能各I0组的时钟。使能FSMC时钟的方法：

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC, ENABLE);
```

对于其他I0口设置的方法前面讲解很详细，这里不做过多的讲解。

2) 设置FSMC BANK1区域3。

此部分包括设置区域3的存储器的工作模式、位宽和读写时序等。本章我们使用模式A、16位宽，读写共用一个时序寄存器。使用的函数是：

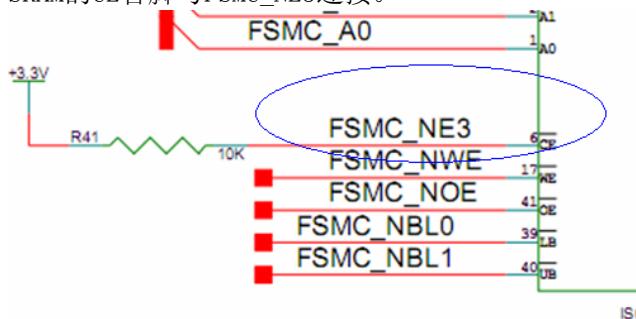
```
void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct)
```

3) 使能BANK1区域3。

使能BANK的方法跟前面LCD实验也是一样的，这里也不做详细讲解，函数是：

```
void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);
```

通过以上几个步骤，我们就完成了FSMC的配置，可以访问IS62WV25616了，这里还需要注意，在上图中，SRAM的CE管脚与FSMC_NE3连接。



查看STM参考手册可知，SRAM连接到STM32F103ZET6处理器的BANK 1 SRAM3。

因为我们使用的是BANK1的区域3，所以HADDR[27:26]=10，故外部内存的首地址为0X68000000。

● SRAM 数据宽度

从上图中可以看出，与SRAM连接的数据线为FSMC_D0~D15,一共16位。也就是说神舟III号上使用的SRAM为16位数据宽度的SRAM。

7.30.3 STM32处理器FSMC总线简介

FSMC(Flexible Static Memory Controller, 可变静态存储控制器)是STM32系列中内部集成256KB以上Flash，后缀为xC、xD和xE的高存储密度微控制器特有的存储控制机制。之所以称为“可变”，是由于通过对特殊功能寄存器的设置，FSMC能够根据不同的外部存储器类型，发出相应的数据/地址/控制信号类型以匹配信号的速度，从而使得STM32系列微控制器不仅能够应用各种不同类型、不同速度的外部静态存储器，而且能够在不增加外部器件的情况下同时扩展多种不同类型的静态存储器，满足系统设计对存储容量、产品体积以及成本的综合要求。

FSMC优点

- ①. 支持多种静态存储器类型。STM32通过FSMC可以与SRAM、ROM、PSRAM、Nor Flash嵌入式专业技术论坛（www.armjishu.com）出品

和 NandFlash 存储器的引脚直接相连。

②. 支持丰富的存储操作方法。FSMC 不仅支持多种数据宽度的异步读 / 写操作，而且支持对 Nor / PSRAM / Nand 存储器的同步突发访问方式。

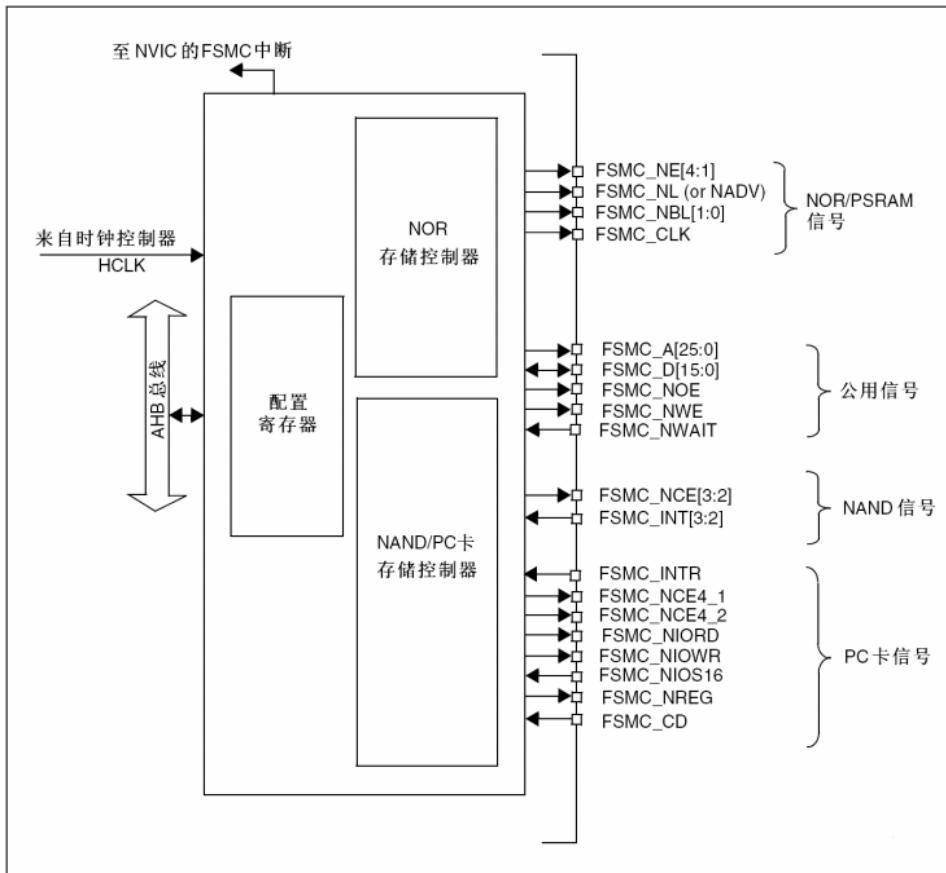
③. 支持同时扩展多种存储器。FSMC 的映射地址空间中，不同的 BANK 是独立的，可用于扩展不同类型的存储器。当系统中扩展和使用多个外部存储器时，FSMC 会通过总线悬空延迟时间参数的设置，防止各存储器对总线的访问冲突。

④. 支持更为广泛的存储器型号。通过对 FSMC 的时间参数设置，扩大了系统中可用存储器的速度范围，为用户提供了灵活的存储芯片选择空间。

⑤. 支持代码从 FSMC 扩展的外部存储器中直接运行。不需要首先调入内部 SRAM。

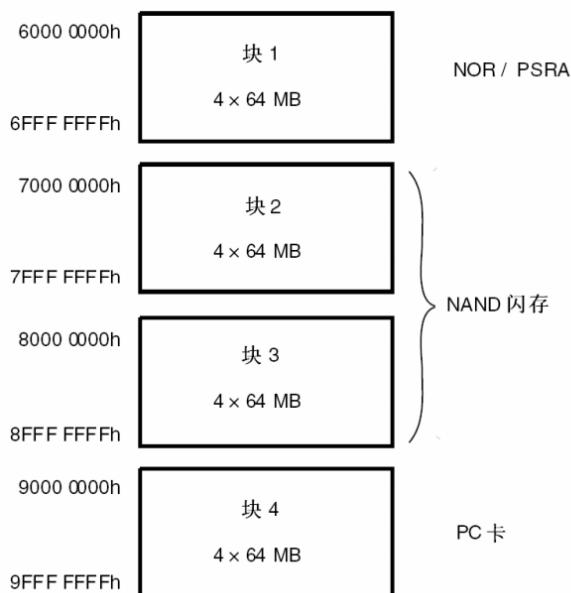
FSMC内部结构

STM32 微控制器之所以能够支持Nor Flash/SRAM和Nand Flash这两类访问方式完全不同的存储器扩展，是因为FSMC内部实际包括Nor Flash和Nand / PC Card两个控制器，分别支持两种截然不同的存储器访问方式。在STM32 内部，FSMC的一端通过内部高速总线AHB连接到内核Cortex—M3，另一端则是面向扩展存储器的外部总线。内核对外部存储器的访问信号发送到AHB总线后，经过FSMC转换为符合外部存储器通信规约的信号，送到外部存储器的相应引脚，实现内核与外部存储器之间的数据交互。FSMC起到桥梁作用，既能够进行信号类型的转换，又能够进行信号宽度和时序的调整，屏蔽掉不同存储类型的差异，使之对内核而言没有区别。



FSMC映射地址空间

FSMC管理1 GB的映射地址空间。该空间划分为4个大小为256 MB的BANK，每个BANK又划分为4个64 MB的子BANK，如下表所列。FSMC的2个控制器管理的映射地址空间不同。Nor Flash控制器管理第1个BANK，Nand / PC Card控制器管理第2~4个BANK。由于两个控制器管理的存储器类型不同，扩展时应根据选用的存储设备类型确定其映射位置。其中，BANK1的4个子BANK拥有独立的片选线和控制寄存器，可分别扩展一个独立的存储设备，而BANK2~BANK4只有一组控制寄存器。



FSMC对每个存储块分配一个唯一的片选信号NE[4: 1]。

7.30.4 实验原理

在本实验中，将SRAM挂到STM32F103ZET6的FSMC总线的Bank 3上(具体分析见4.12.4 硬件设计此节)。

在本实验中，首先按照SRAM芯片要求，初始化好FSMC总线后，往SRAM芯片内固定地址写入一串确定的值，然后程序读回之前写入的数据，判断写入与读出的值是否一致，通过神舟III号的LED灯指示程序执行结果。各种灯的指示含义如下：

| LED指示灯 | 含义 |
|--------|---------------------------------------|
| DS1闪烁 | 程序正在执行过程中 |
| DS2亮 | 写入神舟III号SRAM的数据与读出的数据一致，也就是说访问SRAM成功 |
| DS3亮 | 写入神舟III号SRAM的数据与读出的数据不一致，也就是说访问SRAM失败 |

7.30.5 硬件设计

SRAM访问实验要用到的硬件资源有：

- 串口 1：串口 1 在本实验中用于打印 SRAM 访问提示信号和显示 SRAM 访问程序运行结果。
 串口的输入输出实验在前面已经 进行了详细的讲解，在这里就不在重复。
 具体见 4.5 串口 1 的发送与接收实验。
- LED 指示灯：LED 指示灯主要用于指示 SRAM 访问程序运行状态和 SRAM 访问结果。
- IC61LV25616LL SRAM

7.30.6 软件设计

神舟III号 EEPROM 读写试验位于 [神舟III号光盘\源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\Project\12.SRAM访问实验\(神舟II号\)](#) 目录。

进入 [12.SRAM访问实验\(神舟II号\)\MDK-ARM](#) 目录后，双击 Project.uvproj 可以打开工程，以下为工程文件中主要代码的解释与说明。

FSMC的初始化

在使用SRAM之前，我们需要对SRAM使用的FSMC BANK进行参数配置，使之与SRAM芯片的要求相符合。

在前面的硬件设计原理部分，我们可以得到一下参数。

| | 参数 | 说明 |
|---------------|--------------|------------------------------|
| 存储器类型 | SRAM | |
| 储存器数据宽度 | 16 位 | 使用的数据线一共 16 位，FSMC_D[0:15] |
| 占用的 FSMC BANK | BANK1 的子板块 3 | 硬件上与 STM32F103 的 FSMC_NE3 相连 |

在STM32技术参考手册中，针对FSMC引脚的GPIO模式配置，已经进行了说明。具体FSMC总线的配置如下：

| FSMC引脚 | GPIO配置 |
|---|------------|
| FSMC_A[25:0] FSMC_D[15:0] | 推挽复用输出 |
| FSMC_CK | 推挽复用输出 |
| FSMC_NOE FSMC_NWE | 推挽复用输出 |
| FSMC_NE[4:1] FSMC_NCE[3:2] FSMC_NCE4_1 FSMC_NCE4_2 | 推挽复用输出 |
| FSMC_NWAIT FSMC_CD | 浮空输入或带上拉输入 |
| FSMC_NIOS16 FSMC_INTR FSMC_INT[3:2] | 浮空输入 |
| FSMC_NL FSMC_NBL[1:0] | 推挽复用输出 |
| FSMC_NIORD FSMC_NIOWR FSMC_NREG | 推挽复用输出 |

说明，上表是整个FSMC接口的引脚的初始化配置说明，实际初始化时可只初始化具体存储器使用的FSMC引脚。

如下为神舟III号访问SRAM时，FSMC引脚的初始化，只初始化了与SRAM连接的相关引脚。

```
/*-----FSMC GPIO配置-----*/
/*FSMC总线使用的GPIO组时钟使能*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOG | RCC_APB2Periph_GPIOE |
RCC_APB2Periph_GPIOF, ENABLE);

/*FSMC数据线FSMC_D[0:15]初始化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_8 | GPIO_Pin_9 |
GPIO_Pin_10 | GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOD, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 |
GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 |
GPIO_Pin_15;
GPIO_Init(GPIOE, &GPIO_InitStructure);

/*FSMC地址线FSMC_A[0:17]初始化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 |
GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_12 | GPIO_Pin_13 |
GPIO_Pin_14 | GPIO_Pin_15;
GPIO_Init(GPIOF, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 |
GPIO_Pin_4 | GPIO_Pin_5;
GPIO_Init(GPIOG, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13;
GPIO_Init(GPIOD, &GPIO_InitStructure);

/*FSMC NOE和NWE初试化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
GPIO_Init(GPIOD, &GPIO_InitStructure);

/*FSMC NOE和NE3初试化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_Init(GPIOG, &GPIO_InitStructure);

/*FSMC NBL0和NBL1初试化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
GPIO_Init(GPIOE, &GPIO_InitStructure);
```

在完成FSMC引脚的初始化以后，我们还需要配置FSMC可编程的存储器的参数。包括访问时序，是否支持非对齐访问和等待周期管理（只针对突发模式访问PSRAM和NOR闪存）。

针对NOR 控制器，我们需要初始化下表中的参数。

可编程的NOR/PSRAM访问参数

| 参数 | 功能 | 访问方式 | 单位 | 最小 | 最大 |
|--------|-----------------------------|----------|---------------|----|-----|
| 地址建立时间 | 地址建立阶段的时间 | 异步 | AHB时钟周期(HCLK) | 1 | 16 |
| 地址保持时间 | 地址保持阶段的时间 | 异步，复用I/O | AHB时钟周期(HCLK) | 2 | 16 |
| 数据建立时间 | 数据建立阶段的时间 | 异步 | AHB时钟周期(HCLK) | 2 | 256 |
| 总线恢复时间 | 总线恢复阶段的时间 | 异步或同步读 | AHB时钟周期(HCLK) | 1 | 16 |
| 时钟分频因子 | 存储器访问的时钟周期(CLK)与 AHB时钟周期的比例 | 同步 | AHB时钟周期(HCLK) | 1 | 16 |
| 数据产生时间 | 突发模式下产生第一个数据所需的时钟数目 | 同步 | 存储器时钟周期(CLK) | 2 | 17 |

对应神舟III号SRAM访问程序中，对应的存储器的参数配置程序如下。

```
/*-----FSMC 总线 存储器参数配置-----*/
p.FSMC_AddressSetupTime = 0; //地址建立时间
p.FSMC_AddressHoldTime = 0; //地址保持时间
p.FSMC_DataSetupTime = 2; //数据建立时间
p.FSMC_BusTurnAroundDuration = 0; //总线恢复时间
p.FSMC_CLKDivision = 0; // 时钟分频因子
p.FSMC_DataLatency = 0; //数据产生时间
p.FSMC_AccessMode = //FSMC_AccessMode_A; //FSMC_NOR控制器时序
```

由于STM32处理器的FSMC总线支持多种存储器类型，因此，我们还需要对FSMC总线的工作模式和参数进行设置。为了支持神舟III号的SRAM，对FSMC总线的配置程序如下：

```
/*-----FSMC 总线 参数配置-----*/
FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM3; //使用了FSMC的BANK的子板块3
FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable; //禁止地址数据线复用
FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_SRAM; //存储器类型为SRAM
FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b; //存储器数据宽度为16位
FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable; //关闭突发模式访问
//等待信号优先级，只有在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
//关闭Wrapped burst access mode，只有在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
//等待信号设置，只有在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_BeforeWaitState;
FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable; //使能这个BANK的写操作
//使能/关闭等待信息设置，只在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable; //关闭Extend Mode
FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable; //关闭Write Burst Mode
FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p; //读操作时序参数
FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p; //写操作时序参数

FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);

/*-----使能BANK1的子板块3-----*/
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM3, ENABLE);
```

至此，我们完成了FSMC总线的初始化。

SRAM访问程序

在完成FSMC的初始化以后，我们就可以对SRAM进行读写操作了。在本实验中，首先按照SRAM芯片要求，初始化好FSMC总线，往SRAM芯片内固定地址写入一串确定的值，程序再从这一地址，读回之前写入的数据。判断写入与读出的值是否一致，通过串口和神舟III号的LED灯指示程序执行结果。

```
/*使能FSMC时钟 */
RCC_AHBPeriphClockCmd (RCC_AHBPeriph_FSMC, ENABLE); //使能FSMC时钟
/*配置与SRAM连接的FSMC BANK1 NOR/SRAM3*/
FSMC_SRAM_Init();

/*将写SRAM的数据BUFFER填充为从0x1234开始的连续递增的一串数据 */
Fill_Buffer (TxBuffer, BUFFER_SIZE, 0x1234);
/*将数据写入到SRAM中。WRITE_READ_ADDR:写入的起始地址*/
FSMC_SRAM_WriteBuffer (TxBuffer, WRITE_READ_ADDR, BUFFER_SIZE);

/*从SRAM中读回刚写入的数据。 WRITE_READ_ADDR:读出数据的起始地址*/
FSMC_SRAM_ReadBuffer (RxBuffer, WRITE_READ_ADDR, BUFFER_SIZE);

/*判断读回的数据与写入的数据是否一致*/
for (Index = 0x00; ((Index < BUFFER_SIZE) && (WriteReadStatus == 0)); Index++)
{
    if (RxBuffer[Index] != TxBuffer[Index])
    {
        WriteReadStatus = Index + 1;
    }
}
printf ("\n\r SRAM读写访问程序运行结果: ");
if (WriteReadStatus == 0)
{
    printf ("\n\r SRAM读写访问成功");
    GPIO_ResetBits (GPIO_LED, DS2_PIN);
}
else
{
    printf ("\n\r SRAM读写访问失败");
    GPIO_ResetBits (GPIO_LED, DS3_PIN);
}
while (1)
{
    GPIO_ResetBits (GPIO_LED, DS1_PIN);
    Armjishu_Delay (0x3FFFFF);
    GPIO_SetBits (GPIO_LED, DS1_PIN);
    Armjishu_Delay (0x3FFFFF);
}
```

7.30.7 下载与测试

在 [神舟III号光盘编译好的固件\12.SRAM访问程序实验](#) 目录下的SRAM访问.hex文件即为前面我们分析的SRAM访问试验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟III号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.30.8 实验现象

将程序下载到神舟III号后，重现上电运行，正常情况下，串口打印如下信息。

神舟III号 SRAM读写程序
--DS1闪烁表示神舟III号正常运行
--DS2--亮， 表示读写SRAM成功
--DS3--亮， 表示读写SRAM失败

SRAM读写访问程序运行结果：
SRAM读写访问成功

同时神舟 III 号的 LED 指示灯也会指示运行结果，具体 LED 指示灯的状态及其含义如下：

| LED指示灯 | 含义 |
|--------|---------------------------------------|
| DS1闪烁 | 程序正在执行过程中 |
| DS2亮 | 写入神舟III号SRAM的数据与读出的数据一致，也就是说访问SRAM成功 |
| DS3亮 | 写入神舟III号SRAM的数据与读出的数据不一致，也就是说访问SRAM失败 |

7.31 内存管理

7.31.1 意义与作用

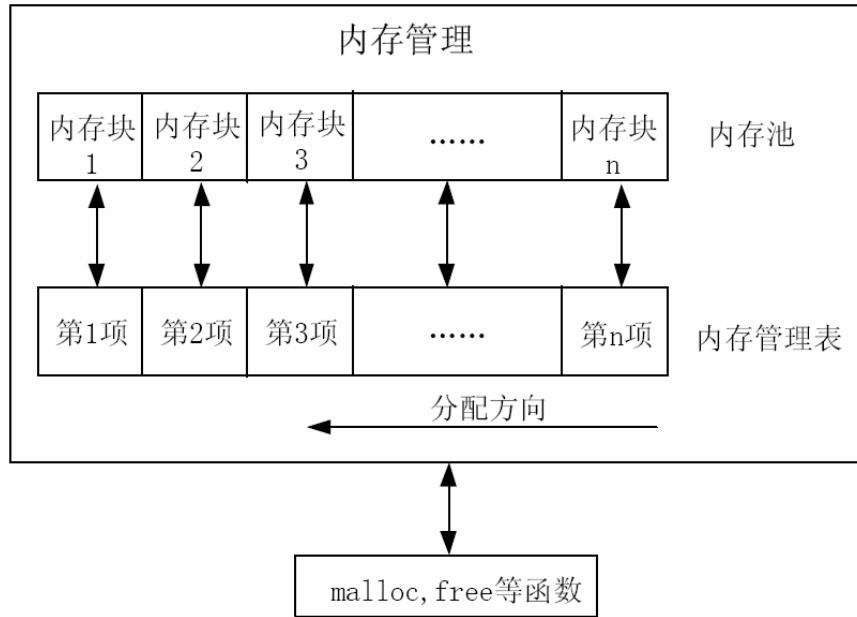
上一节，我们学会了使用STM32驱动外部SRAM，以扩展STM32的内存，加上STM32本身自带的64K字节内存，我们可供使用的内存还是比较多的。如果我们所用的内存都像上一节的testsram那样，定义一个数组来使用，显然不是一个好办法。

本章，我们将学习内存管理，实现对内存的动态管理。

7.31.2 内存管脚简介

内存管理，是指软件运行时对计算机内存资源的分配和使用的技术。其最主要的目的是高效，快速的分配，并且在适当的时候释放和回收内存资源。内存管理的实现方法有很多种，其实最终都是要实现 2 个函数：malloc 和 free；malloc 函数用于内存申请，free 函数用于内存释放。

本章，我们介绍一种比较简单的办法来实现：分块式内存管理。下面我们介绍一下该方法的实现原理，如图：



从上图可以看出，分块式内存管理由**内存池**和**内存管理表**两部分组成。内存池被等分为 n 块，对应的内存管理表，大小也为 n ，内存管理表的每一个项对应内存池的一块内存。

内存管理表的项值代表的意义为：当该项值为 0 的时候，代表对应的内存块未被占用，当该项值非零的时候，代表该项对应的内存块已经被占用，其数值则代表被连续占用的内存块数。比如某项值为 10，那么说明包括本项对应的内存块在内，总共分配了 10 个内存块给外部的某个指针。

内存分配方向如图所示，是从顶→底的分配方向。即首先从最末端开始找空内存。当内存管理刚初始化的时候，内存表全部清零，表示没有任何内存块被占用。

分配原理：

当指针 p 调用 `malloc` 申请内存的时候，先判断 p 要分配的内存块数 (m)，然后从第 n 项开始，向下查找，直到找到 m 块连续的空内存块（即对应内存管理表项为 0），然后将这 m 个内存管理表项的值都设置为 m （标记被占用），最后，把最后的这个空内存块的地址返回指针 p ，完成一次分配。注意，如果当内存不够的时候（找到最后也没找到连续的 m 块空闲内存），则返回 `NULL` 给 p ，表示分配失败。

释放原理：

当 p 申请的内存用完，需要释放的时候，调用 `free` 函数实现。`free` 函数先判断 p 指向的内存地址所对应的内存块，然后找到对应的内存管理表项目，得到 p 所占用的内存块数目 m （内存管理表项目的值就是所分配内存块的数目），将这 m 个内存管理表项目的值都清零，标记释放，完成一次内存释放。

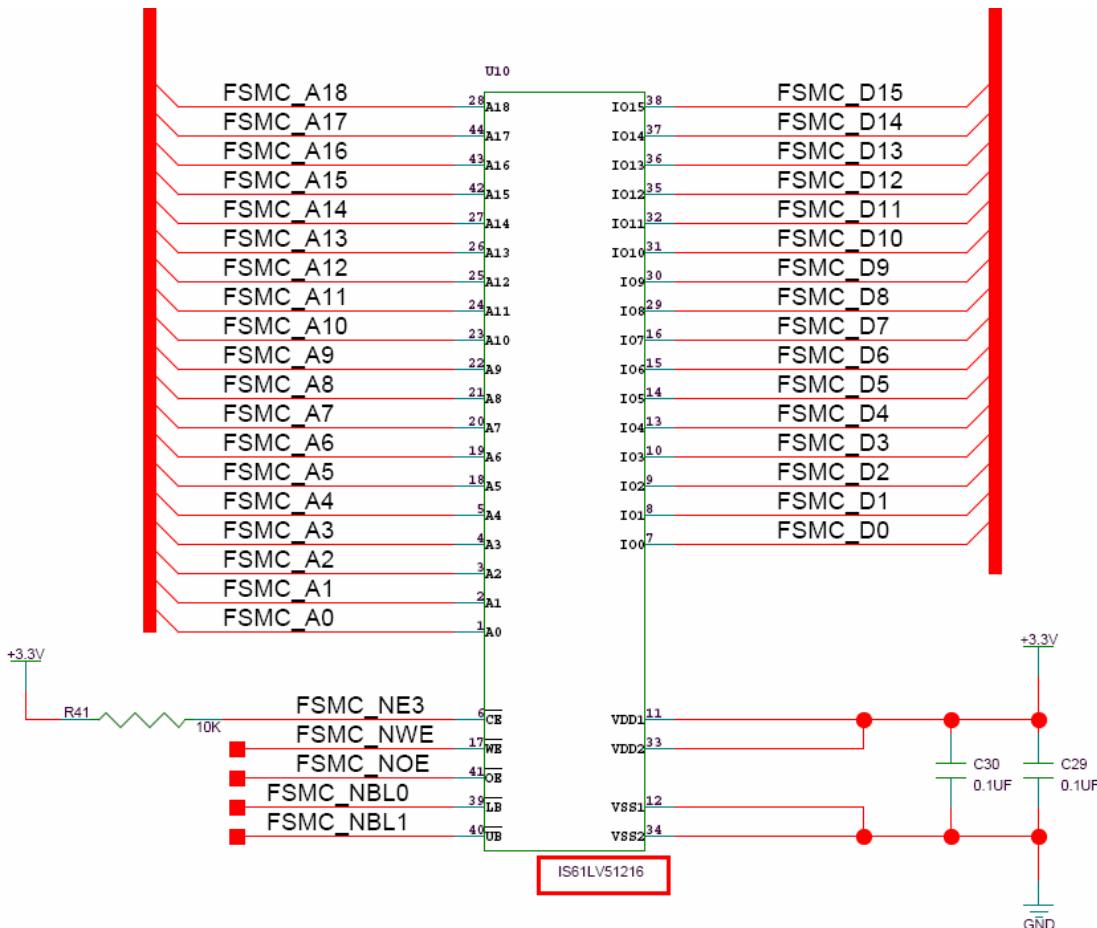
关于分块式内存管理的原理，我们就介绍到这里。

7.31.3 实验原理

本实验我们通过按下按键 USER1 申请内存；按下按键 TAMPER 释放内存；按下按键 WAKEUP 是使用外部 SRAM 还是使用内部 SRAM。

7.31.4 硬件设计

本实验我们使用到按键、串口、TFT 彩屏、外部 SRAM、内部 SRAM。外部 SRAM 的原理图如下：



7.31.5 软件设计

我们从主函数开始分析

```

12 int main(void)
13 {
14     u8 key;
15     u8 i=0;
16     u8 *p=0;
17     u8 *tp=0;
18     u8 paddr[18];           //存放P Addr:+p地址的ASCII值
19     u8 sramx=0;             //默认为内部sram
20
21     delay_init();           //延时函数初始化
22     NVIC_Configuration();  //设置NVIC中断分组2:2位抢占优先级,
23     uart_init(9600);        //串口初始化为9600
24     LCD_Init();             //按键初始化
25     KEY_Init();             //按键初始化
26
27     usmart_dev.init(72);    //初始化USMART
28     FSMC_SRAM_Init();      //初始化外部SRAM
29     mem_init(SRAMIN);       //初始化内部内存池
30     mem_init(SRAMEX);       //初始化外部内存池
31
32     POINT_COLOR=RED;//设置字体为红色
33     LCD_ShowString(20,80,200,16,16,"MALLOC TEST");
34     LCD_ShowString(20,100,200,16,16,"USER1:Malloc TAMPER:Free");
35     LCD_ShowString(20,120,200,16,16,"WAKEUP:SRAMX USER2:Read");
36
37     POINT_COLOR=BLUE;//设置字体为蓝色
38     LCD_ShowString(60,170,200,16,16,"SRAMIN");
39     LCD_ShowString(60,190,200,16,16,"SRAMIN USED: %");
40     LCD_ShowString(60,210,200,16,16,"SRAMEX USED: %");

```

本实验中，按键、TFT 彩屏、外部 SRAM 都是重要的。代码中，通过函数 LCD_Init() 初始化 TFT 彩屏。通过函数 KEY_Init() 初始化按键。通过函数 FSMC_SRAM_Init() 初始化，外部 SRAM。

代码分析1：我们主要看和内存管理相关的

```
//内存池(4字节对齐)
__align(4) u8 mem1base[MEM1_MAX_SIZE];
__align(4) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0x68000000)));
//内存管理表
u16 mem1mapbase[MEM1_ALLOC_TABLE_SIZE];
u16 mem2mapbase[MEM2_ALLOC_TABLE_SIZE] __attribute__((at(0x68000000+MEM2_MAX_SIZE)));
//内存管理参数
```

我们可以看到外部内存池指定的地址为 0x68000000，这个是外部 SRAM 的首地址。__align(4)定义内存池为 4 字节对齐，这个非常重要！如果不加这个限制，在某些情况下（比如分配内存给结构体指针），可能出现错误，所以一定要加上这个。

代码分析2：我们看一下，malloc.h文件

```
01 #ifndef __MALLOC_H
02 #define __MALLOC_H
03 #include "sys.h"
04
05 #ifndef NULL
06 #define NULL 0
07 #endif
08
09 #define SRAMIN 0 //内部内存池
10 #define SRAMEX 1 //外部内存池
11
12
13 //mem1内存参数设定.mem1完全处于内部SRAM里面
14 #define MEM1_BLOCK_SIZE 32 //内存块大小为32字节
15 #define MEM1_MAX_SIZE 40*1024 //最大管理内存 45K
16 #define MEM1_ALLOC_TABLE_SIZE MEM1_MAX_SIZE/MEM1_BLOCK_SIZE //内存表大小
17
18 //mem2内存参数设定.mem2的内存池处于外部SRAM里面,其他的处于内部SRAM里面
19 #define MEM2_BLOCK_SIZE 32 //内存块大小为32字节
20 #define MEM2_MAX_SIZE 192*1024 //最大管理内存 192K
21 #define MEM2_ALLOC_TABLE_SIZE MEM2_MAX_SIZE/MEM2_BLOCK_SIZE //内存表大小
22
23
24 //内存管理控制器
25 struct _m_malloc_dev
26 {
27     void (*init)(u8); //初始化
28     u8 (*perused)(u8); //内存使用率
29     u8 *membase[2]; //内存池 管理2个区域的内存
30     u16 *memmap[2]; //内存管理状态表
31     u8 memrady[2]; //内存管理是否就绪
32 };
```

这部分代码，定义了很多关键数据，比如内存块大小的定义：MEM1_BLOCK_SIZE 和 MEM2_BLOCK_SIZE，都是32字节。内存池总大小，内部为40K，外部为200K（最大支持到近1M字节，不过为了方便演示，这里只管理200K内存）。MEM1_ALLOC_TABLE_SIZE 和 MEM2_ALLOC_TABLE_SIZE，则分别代表内存池1和2的内存管理表大小。

从这里可以看出，如果内存分块越小，那么内存管理表就越大，当分块为2字节1个块的时候，内存管理表就和内存池一样大了（管理表的每项都是u16类型）。显然是不合适的，我们这里取32字节，比例为1:16，内存管理表相对就比较小了。

代码分析3：mymalloc()函数myfree()函数的应用。

```
121 void myfree(u8 memx,void *ptr)
122 {
123     u32 offset;
124     if(ptr==NULL) return;//地址为0.
125     offset=(u32)ptr-(u32)mallco_dev.membase[memx];
126     mem_free(memx,offset);//释放内存
127 }
128 //分配内存(外部调用)
129 //memx:所属内存块
130 //size:内存大小(字节)
131 //返回值:分配到的内存首地址.
132 void *mymalloc(u8 memx,u32 size)
133 {
134     u32 offset;
135     offset=mem_malloc(memx,size);
136     if(offset==0xFFFFFFFF) return NULL;
137     else return (void*)((u32)mallco_dev.membase[memx]+offset);
138 }
```

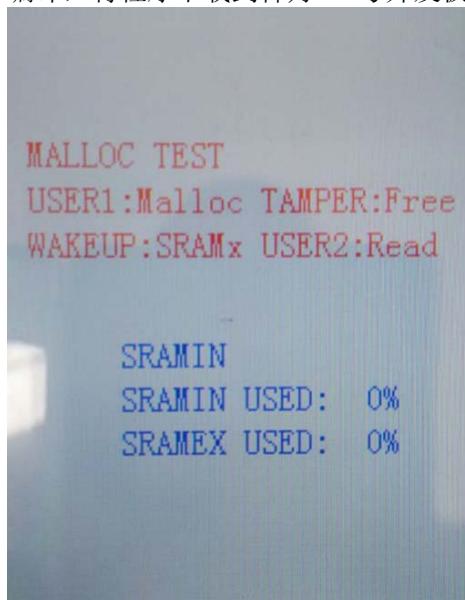
这两个函数的使用方法在“内存管理简介”中我们进行了介绍。不过这里提醒大家，如果对一个指针进行多次内存申请，而之前的申请又没释放，那么将造成“内存泄露”，这是内存管理所不希望发生的，久而久之，可能导致无内存可用的情况！所以，在使用的时候，请大家一定记得，申请的内存在用完以后，一定要释放。

7.31.6 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

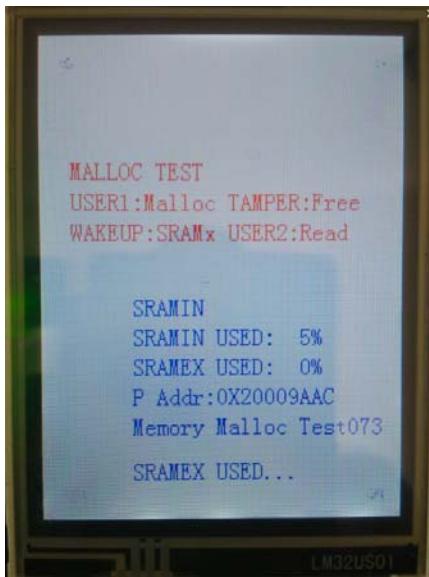
7.31.7 实验现象

编译，将程序下载到神舟 III 号开发板上，按下复位按键，TFT 彩屏显示如下：



可以看到，内外内存的使用率均为 0%，说明还没有任何内存被使用，此时我们按下 USER1，就可以看到内部内存被使用 5% 了，同时看到下面提示了指针 p 所指向的地址（其实就是被分配到的内存地址）和内容。多按几次 USER1，可以看到内存使用率持续上升（注意对比 p 的值，可以发现是递减的，说明是从顶部开始分配内存！），此时如果按下按键 TAMPER，可以发现内存使用率降低了 5%，但是再按按键 TAMPER 将不再降低，说明“内存泄露”了。这就是前面提到的对一个指针多次申请内存，而之前申请的内存又没释放，导致的“内存泄露”。

按下 USER1 按键，TFT 彩屏显示如下：



7.32 Nor Flash访问程序实验

在提到Nor Flash时候，我们就不得不提到Nand Flash。Nor和Nand Flash是现在市场上两种主要的非易失闪存技术。其中，Intel于1988年首先开发出Nor flash技术，彻底改变了原先由EPROM和EEPROM一统天下的局面。紧接着，1989年，东芝公司发表了Nand flash结构，强调降低每比特的成本，更高的性能，并且象磁盘一样可以通过接口轻松升级。

经过了十多年之后，仍然有相当多的硬件工程师分不清Nor和Nand闪存。在这里，我们首先对Nor Flash和Nand Flash从应用领域和特性等方面来认识Nor Flash与Nand Flash。

7.32.1 Nor Flash与Nand Flash的区别

4.13.1.1 性能比较

Flash闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何flash器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。Nand 器件执行擦除操作是十分简单的，而Nor则要求在进行擦除前先要将目标块内所有的位都写为0。由于擦除Nor器件时是以64~128KB 的块进行的，执行一个写入/擦除操作的时间为5s，与此相反，擦除Nand器件是以8~32KB的块进行的，执行相同的操作最多只需要4ms。执行擦除时块尺寸的不同进一步拉大了 Nor 和 NAND之间的性能差距。

Nor Flash与Nand Flash的主要性能区别在于：

- Nor的读速度比Nand稍快一些。
- Nand的写入速度比Nor快很多。
- Nand的4ms擦除速度远比Nor的5s快。
- 大多数写入操作需要先进行擦除操作。
- Nand的擦除单元更小，相应的擦除电路更少。

4.13.1.2 接口差别

Nor flash带有SRAM接口，有足够的地址引脚来寻址，可以轻松地挂接在CPU的地址、数据总线上，对CPU的接口要求低。可以很容易地存取其内部的每一个字节。

Nand Flash器件使用复杂的I/O口来串行地存取数据，各个产品或厂商的方法可能各不相同。8个引脚用来传送控制、地址和数据信息。由于时序较为复杂，所以一般CPU最好集成Nand控制器。另外由于NandFlash没有挂接在地址总线上，所以如果想用NandFlash作为系统的启动盘，就需要CPU具备特殊的功能，如s3c2410在被选择为NandFlash启动方式时会在上电时自动读取NandFlash的4k数据到地址0的SRAM中。如果CPU不具备这种特殊功能，用户不能直接运行NandFlash上的代码，那可以采取其他方式，比如好多使用NandFlash的开发板除了使用NandFlash以外，还用上了一块小的NorFlash来运行启动代码。

Nand 读和写操作采用 256 字节的板，这一点有点像硬盘管理此类操作，很自然地，基于 Nand 的存储器就可以取代硬盘或其他块设备。

4.13.1.3 容量和成本

Nand flash 的单元尺寸几乎是 Nor 器件的一半，由于生产过程更为简单，Nand 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

Nor flash占据了容量为1~16MB闪存市场的大部分，而Nand flash只是用在8~128MB的产品当中，这也说明Nor主要应用在代码存储介质中，Nand适合于数据存储，Nand 在CompactFlash、Secure Digital、PC Cards和MMC 存储卡市场上所占份额最大。

相比起NandFlash来说，NorFlash的容量要小，一般在1~16MByte左右，一些新工艺采用了芯片叠加技术可以把NorFlash的容量做得大一些。在价格方面，NorFlash相比NandFlash来说较高。

NandFlash生产过程更为简单，Nand结构可以在给定的模具尺寸内提供更高的容量，这样也就相应地降低了价格。

4.13.1.4 可靠性和耐用性

采用flash介质时一个需要重点考虑的问题是可靠性。对于需要扩展MTBF的系统来说，Flash 是非常合适的存储方案。可以从寿命(耐用性)、位交换和坏块处理三个方面来比较Nor和Nand的可靠性。

➤ 寿命(耐用性)

在Nand 闪存中每个块的最大擦写次数是一百万次，而Nor的擦写次数是十万次。Nand存储器除了具有10比1的块擦除周期优势，典型的Nand块尺寸要比Nor器件小8倍，每个Nand存储器块在给定的时间内的删除次数要少一些。

➤ 位交换

所有flash 器件都受位交换现象的困扰。在某些情况下(很少见，Nand发生的次数要比Nor多)，一个比特位会发生反转或被报告反转了。一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。当然，如果这个位真的改变了，就必须采用错误探测/错误更正(EDC/ECC)算法。位反转的问题更多见于Nand闪存，Nand的供应商建议使用Nand闪存的时候，同时使用EDC/ECC算法。这个问题对于用 Nand 存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用EDC/ECC系统以确保可靠性。

➤ 坏块处理

Nand器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。 Nand 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

➤ 易于使用

可以非常直接地使用基于Nor的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。由于需要I/O接口，Nand要复杂得多。各种Nand器件的存取方法因厂家而异。 在使用Nand器件时，必须先写入驱动程序，才能继续执行其他操作。向Nand 器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在Nand器件上自始至终都必须进行虚拟映射。

➤ 软件支持

当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。在Nor器件上运行代码不需要任何的软件支持，在Nand器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，Nand和Nor器件在进行写入和擦除操作时都需要MTD。 使用 Nor 器件时所需要的 MTD 要相对少一些，许多厂商都提供用于 Nor 器件的更高级软件，这其中包括 M-System的TrueFFS驱动，该驱动被Wind River System、Microsoft、QNX Software System、Symbian和Intel等厂商所采用。

驱动还用于对DiskOnChip产品进行仿真和Nand闪存的管理，包括纠错、坏块处理和损耗平衡。

Nand Flash内部结构是用与非门组成存储单元的。有非易失性，读写速度快，而且比较容易做到大容量。目前单片Nand Flash存储容量可以达到8Gbit (1GByte)。Nor Flash有易失性，掉电不保存数据。随机存储速度比Nand Flash 快得多。所以一般用Nor Flash 用做内存片，或者叫做数据缓冲。而Nand Flash则一般用来做存储数据用。比方说，U盘.MP3等。

4.13.1.5 应用领域

Flash在嵌入式行业应用广泛，但是由于Nor Flash与Nand Flash他们的容量，性能和价格等方面的差异，在不同的市场， Nor 的传输效率很高，在小容量时具有很高的成本效益，更加安全，不容易出现数据故障，因此，主要应用以代码存储为主，多与运算相关，而Nand Flash由于容量可以很大，而成本又低，它的市场份额处于不断的增长中。

在手机市场，Nor Flash主要用于存储代码和数据；而Nand Flash随着技术的发展，高速手机平台也开始使用Nand Flash+ SDRAM架构。

在便携式消费类电子产品市场，处于成本考虑，基本上没有厂家使用Nor Flash存储器。

在PC市场，Nor Flash主要用于PC的BIOS部分，并且多为4Mb-16Mb小容量的。

7.32.2 FSMC 扩展 Nor Flash 配置

SRAM / ROM、Nor Flash 和 PSRAM 类型的外部存储器都是由 FSMC 的 Nor Flash 控制器管理的，扩展方法基本相同，其中 Nor Flash 最为复杂。通过 FSMC 扩展外部存储器时，除了传统存储器扩展所需要的硬件电路外，还需要进行 FSMC 初始化配置。FSMC 提供大量、细致的可编程参数，以便能够灵活地进行各种不同类型、不同速度的存储器扩展。外部存储器能否正常工作的关键在于：

用户根据选用的存储器型号，对配置寄存器进行合理的初始化配置。

(1) 确定映射地址空间

根据选用的存储器类型确定扩展使用的映射地址空间。Nor Flash 只能选用 BANK1 中的 4 个子 BANK。选定映射子 BANK 后，即可确定以下 2 方面内容：

- ① 硬件电路中用于选中该存储器的片选线 FSMC_NE*i*(*i* 为子 BANK 号，*i*=1, ..., 4);
- ② FSMC 配置中用于配置该外部存储器的特殊功能寄存器号。

表 1 FSMC 映射地址空间

| 内部控制器 | BANK 号 | 映射地址范围 | 支持设备类型 | 特殊功能寄存器 |
|-------------------------|--------|-----------------------|--------------------------------|---|
| NOR Flash 控制器 | BANK1 | 6000 0000H~6FFF FFFFH | SRAM/ROM NOR Flash PSRAM | FSMC_BCR1~4 FSMC_BTR1~4 FSMC_BWTR1~4 |
| NAND/ PC Card 控制器 | BANK2 | 7000 0000H~7FFF FFFFH | NAND Flash | FSMC_PCR2~4 FSMC_SR2~4 FSMC_PMEM2~4 FSMC_PATT2~4 |
| | BANK3 | 8000 0000H~8FFF FFFFH | | FSMC_PIO4 |
| | BANK4 | 9000 0000H~9FFF FFFFH | PC Card | |

(2) 配置存储器基本特征

根据选用的存储器芯片确定需要配置的存储器特征，FSMC 根据不同存储器特征可灵活地进行工作方式和信号的调整。主要包括以下方面：

- ① 存储器类型(MTYPE)是 SRAM / ROM、PSRAM，还是 Nor Flash；
- ② 存储芯片的地址和数据引脚是否复用(MUXEN)，FSMC 可以直接与 AD0~AD15 复用的存储器相连，不需要增加外部器件；
- ③ 存储芯片的数据线宽度(MWID)，FSMC 支持 8 位 / 16 位两种外部数据总线宽度；
- ④ 对于 Nor Flash(PSRAM)，是否采用同步突发访问方式(BURSTEN)；
- ⑤ 对于 Nor Flash(PSRAM)，NWAIT 信号的特性说明(WAITEN、WAITCFG、WAITPOL)；
- ⑥ 对于该存储芯片的读 / 写操作，是否采用相同的时序参数来确定时序关系(EXTMOD)。

(3) 配置存储器时序参数

FSMC 通过使用可编程的存储器时序参数寄存器，拓宽了可选用的外部存储器的速度范围。FSMC 的 NorFlash 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要的设置的时间参数有 2 个：

- ① HCLK 与 FSMC_CLK 的分频系数(CLKDIV)，可以为 2~16 分频；
- ② 同步突发访问中获得第 1 个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。FSMC 综合了 SRAM / ROM、PSRAM 和 Nor Flash 产品的信号特点，定义了 4 种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数。在实际扩展时，根据选用存储器的特征确定时序模型，从而确定各时间参数与存储器读 / 写周期参数指标之间的计算关系；利用该计算关系和存储芯片数据手册中给定的参数指标，可计算出 FSMC 所需要的各时间参数，从而对时间参数寄存器进行合理的配置。

表 2 NOR Flash 控制器支持的时序模型

| 时序模型 | 简单描述 | 时间参数 |
|------|---------|--|
| 异步突发 | Mode1 | SRAM/CRAM 时序 DATAST、ADDSET |
| | ModeA | SRAM/CRAM OE 选通型时序 DATAST、ADDSET |
| | Mode2/B | NOR Flash 时序 DATAST、ADDSET |
| | ModeC | NOR Flash OE 选通型时序 DATAST、ADDSET |
| | ModeD | 延长地址保持时间的异步时序 DATAST、ADDSET ADDHLD |
| | 同步突发 | 根据 FSMC_CLK，同步读取多个顺序单元的数据 CLKDIV、DATLAT |

7.32.3 Nor Flash访问实验的意义与作用

在前面，我们比较了Nor Flash和Nand Flash的区别，Nand Flash虽然在容量和成本方面比较有优势，但是在易用性和数据安全方面，Nor Flash还是有不可替代的优势，因此Nor Flash一般用于程序存储。

Nor flash带有SRAM接口，有足够的地址引脚来寻址，可以轻松地挂接在CPU的地址、数据总线上，对CPU的接口要求低。在本章节，我们将一起分析，如何通过STM32的FSMC接口访问Nor Flash。

7.32.4 实验原理

在本实验中，将Nor Flash挂到STM32F103ZET6的FSMC总线的板块1的子板块2上(与前一节描述的，将SRAM挂在板块1的子板块3上类似)。

在本实验中，首先按照Nor Flash芯片要求，初始化好FSMC总线后，往Nor Flash芯片内固定地址写入一串确定的值，然后程序读回之前写入的数据，判断写入与读出的值是否一致，通过神舟III号的LED灯指示程序执行结果。各种灯的指示具体含义如下：

| LED指示灯 | 含义 |
|--------|---------------------------------------|
| DS1闪烁 | 程序正在执行过程中 |
| DS2亮 | 写入神舟III号SRAM的数据与读出的数据一致，也就是说访问SRAM成功 |
| DS3亮 | 写入神舟III号SRAM的数据与读出的数据不一致，也就是说访问SRAM失败 |

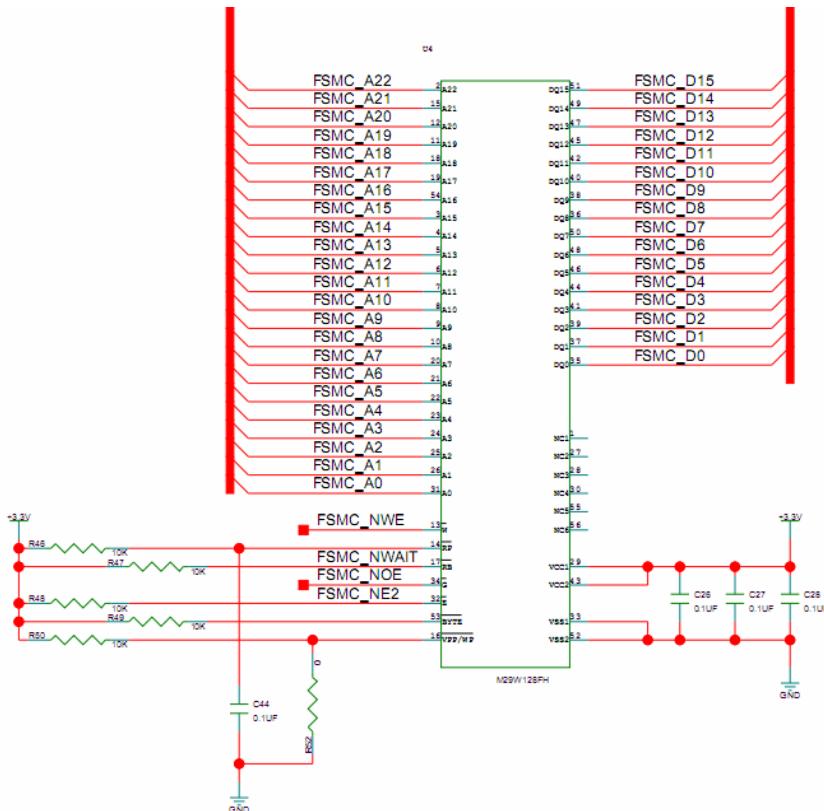
7.32.5 硬件设计

NOR FLASH访问实验要用到的硬件资源有：

- 串口 1：串口 1 在本实验中用于打印 Nor Flash 访问提示信号和显示 Nor Flash 访问运行结果。
 串口的输入输出实验在前面已经 进行了详细的讲解，在这里就不在重复。
 具体见 4.5 串口 1 的发送与接收实验。
- LED 指示灯：LED 指示灯主要用于指示 Nor Flash 访问程序运行状态和 Nor Flash 访问结果。
- 39VF1601 NOR FLASH

4.12.4.1. 硬件原理

在神舟III号中，通过FSMC总线与39VF1601 NOR Flash连接，具体电路如下：



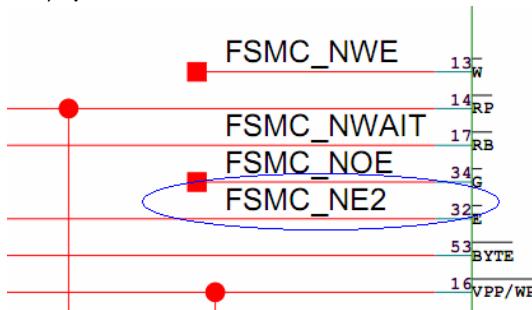
在上图中，我们可以看到以下信息

- **NOR FLASH 数据宽度**

从上图中可以看出，与 NOR FLASH 连接的数据线为 FSMC_D0~D15,一共 16 位。也就是说神舟 III 号上使用的 Nor Flash 为 16 位数据宽度的 Nor Flash。

- **Nor Flash 的地址范围**

在上图中，Nor Flash 的 CE 管脚与 FSMC_NE2 连接。将 Nor Flash 扩展到 Nor Flash 控制器管理的 BANK1 的第 2 个子 BANK。

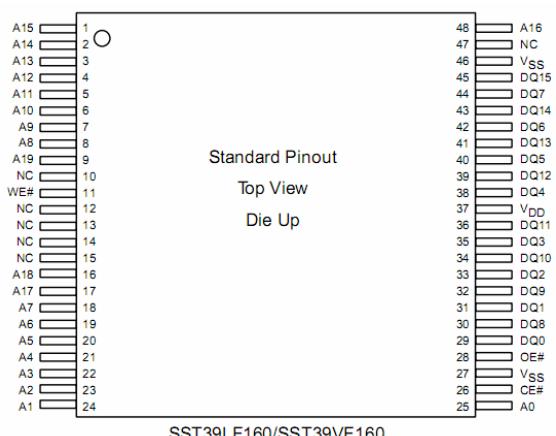


查看 STM 参考手册可知，SRAM 连接到 STM32F103ZET6 处理器的 BANK 1 子板块 2。

4.12.4.2.Nor Flash 芯片说明

在神舟 III 号中，使用的 Nor Flash 是 SST 公司推出的 SST39VF160。它的容量为 16M 比特(如果需要配置更大容量的 Nor Flash，可以直接替换为 SST39VF3201, SST39VF6401 等大容量的 Nor Flash，他们与 SST39VF160 完全 pin-to-pin 兼容，而容量却更大，达到 64M 比特)。

SST39VF160 的管脚定义如下：



SST39LF160/SST39VF160

PIN DESCRIPTION

| Symbol | Pin Name | Functions |
|-----------------------------------|-------------------|--|
| A ₁₉ -A ₀ | Address Inputs | To provide memory addresses. During Sector-Erase A ₁₉ -A ₁₁ address lines will select the sector. During Block-Erase, A ₁₉ -A ₁₅ address line will select the block. |
| DQ ₁₅ -DQ ₀ | Data Input/output | To output data during Read cycles and receive input data during Write cycles. Data is internally latched during a Write cycle. The outputs are in tri-state when OE# or CE# is high. |
| CE# | Chip Enable | To activate the device when CE# is low |
| OE# | Output Enable | To gate the data output buffers |
| WE# | Write Enable | To control the Write operations |
| V _{DD} | Power Supply | To provide power supply voltage: 3.0-3.6V for SST39LF160 2.7-3.6V for SST39VF160 |
| V _{SS} | Ground | |
| NC | No Connection | Unconnected pins |

7.32.6 软件设计

神舟 III 号 Nor Flash 读写试验位于 [神舟III号光盘\源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar](#)

\Project\13.Nor Flash 访问实验 (神舟III号) 目录。

进入 13.Nor Flash 访问实验 (神舟III号)\MDK-ARM 目录后，双击 Project.uvproj 可以打开工程，以下为工程文件中主要代码的解释与说明。

4.12.5.1. 应用STM32固件对FSMC进行初始化配置

ST 公司为用户开发提供了完整、高效的工具和固件库，其中使用 C 语言编写的固件库提供了覆盖所有标准外设的函数，使用户无需使用汇编操作外设特性，从而提高了程序的可读性和易维护性。

在使用Nor Flash之前，我们需要对使用的FSMC BANK进行参数配置，使之与Nor Flash芯片的要求相符合。

在前面的硬件设计原理部分，我们可以得到一下参数。

| | 参数 | 说明 |
|---------------|--------------|------------------------------|
| 存储器类型 | Nor | |
| 储存器数据宽度 | 16 位 | 使用的数据线一共 16 位，FSMC_D[0:15] |
| 占用的 FSMC BANK | BANK1 的子板块 2 | 硬件上与 STM32F103 的 FSMC_NE2 相连 |

在STM32技术参考手册中，针对FSMC引脚的GPIO模式配置，已经进行了说明。具体FSMC总线的配置如下：

| FSMC引脚 | GPIO配置 |
|---|------------|
| FSMC_A[25:0] FSMC_D[15:0] | 推挽复用输出 |
| FSMC_CK | 推挽复用输出 |
| FSMC_NOE FSMC_NWE | 推挽复用输出 |
| FSMC_NE[4:1] FSMC_NCE[3:2] FSMC_NCE4_1 FSMC_NCE4_2 | 推挽复用输出 |
| FSMC_NWAIT FSMC_CD | 浮空输入或带上拉输入 |
| FSMC_NIOS16 FSMC_INTR FSMC_INT[3:2] | 浮空输入 |
| FSMC_NL FSMC_NBL[1:0] | 推挽复用输出 |
| FSMC_NIORD FSMC_NIOWR FSMC_NREG | 推挽复用输出 |

说明，上表是整个FSMC接口的引脚的初始化配置说明，实际初始化时可只初始化具体存储器使用的FSMC引脚。

如下为神舟 III 号访问 Nor Flash，FSMC 引脚的初始化，只初始化了与 Nor Flash 连接的相关引脚。

```
/*FSMC总线使用的GPIO组时钟使能*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOE |
                        RCC_APB2Periph_GPIOF | RCC_APB2Periph_GPIOG, ENABLE);

/*FSMC数据线FSMC_D[0:15]初始化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_8 | GPIO_Pin_9 |
                               GPIO_Pin_10 | GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOD, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 |
                               GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13 |
                               GPIO_Pin_14 | GPIO_Pin_15;
GPIO_Init(GPIOE, &GPIO_InitStructure);

/*FSMC地址线FSMC_A[0:17]初始化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 |
                               GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_12 | GPIO_Pin_13 |
                               GPIO_Pin_14 | GPIO_Pin_15;
GPIO_Init(GPIOF, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 |
                               GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5;
GPIO_Init(GPIOG, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13;
GPIO_Init(GPIOD, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6;
GPIO_Init(GPIOE, &GPIO_InitStructure);

/*FSMC_NOE和NWE初试化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
GPIO_Init(GPIOD, &GPIO_InitStructure);

/*FSMC_NB2初试化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_Init(GPIOG, &GPIO_InitStructure);
```

STM32 固件库中提供的FSMC的Nor Flash控制器操作固件，主要包括 2 个数据结构和 3 个函数。数据结构FSMC_NorSRAMTimingInitTypeDef对应时间参数寄存器FSMC_BTR和FSMC_BWTR的结构定义。

在完成FSMC引脚的初始化以后，我们还需要配置FSMC可编程的存储器的参数。包括访问时序，是否支持非对齐访问和等待周期管理（只针对突发模式访问PSRAM和NOR闪存）。

针对NOR 控制器，我们需要初始化下表中的参数。

可编程的NOR/PSRAM访问参数

| 参数 | 功能 | 访问方式 | 单位 | 最小 | 最大 |
|--------|-----------------------------|----------|---------------|----|-----|
| 地址建立时间 | 地址建立阶段的时间 | 异步 | AHB时钟周期(HCLK) | 1 | 16 |
| 地址保持时间 | 地址保持阶段的时间 | 异步，复用I/O | AHB时钟周期(HCLK) | 2 | 16 |
| 数据建立时间 | 数据建立阶段的时间 | 异步 | AHB时钟周期(HCLK) | 2 | 256 |
| 总线恢复时间 | 总线恢复阶段的时间 | 异步或同步读 | AHB时钟周期(HCLK) | 1 | 16 |
| 时钟分频因子 | 存储器访问的时钟周期(CLK)与 AHB时钟周期的比例 | 同步 | AHB时钟周期(HCLK) | 1 | 16 |
| 数据产生时间 | 突发模式下产生第一个数据所需的时钟数目 | 同步 | 存储器时钟周期(CLK) | 2 | 17 |

对应神舟III号Nor Flash访问程序中，对应的存储器的参数配置程序如下。

```
FSMC_NORSRAMTimingInitTypeDef p;  
/*-----FSMC 总线 存储器参数配置-----*/  
p.FSMC_AddressSetupTime = 0x05; //地址建立时间  
p.FSMC_AddressHoldTime = 0x00; //地址保持时间  
p.FSMC_DataSetupTime = 0x07; //数据建立时间  
p.FSMC_BusTurnAroundDuration = 0; //总线恢复时间x00  
p.FSMC_CLKDivision = 0x00; // 时钟分频因子  
p.FSMC_DataLatency = 0x00; //数据产生时间  
p.FSMC_AccessMode = FSMC_AccessMode_B; //FSMC NOR控制器时序
```

FSMC_NorSRAMInitTypeDef 对应特征配置寄存器 FSMC_BCR 的结构定义，并包含 2 个指向对应 BANK 的 FSMC_BTR 和 FSMC_BWTR 寄存器的 FSMC_NorSRAMTimingInitTypeDef 结构指针。

由于 STM32 处理器的 FSMC 总线支持多种存储器类型，因此，我们还需要对 FSMC 总线的工作模式和参数进行设置。为了支持神舟 III 号的 Nor Flash, SRAM，针对上述 SST39VF160 芯片扩展要求，利用固件库对 FSMC 总线的配置程序如下：

```
FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM2; // 使用了 FSMC 的 BANK1 的子模块2
FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable; // 禁止地址数据线复用
FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_NOR; // 存储器类型为 NOR
FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b; // 存储器数据宽度为 16 位
FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable; // 关闭突发模式访问
// 等待信号优先级，只有在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
// 关闭 wrapped burst access mode，只有在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
// 等待信号设置，只有在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_BeforeWaitState;
// 使能这个 BANK 的写操作
FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
// 使能/关闭等待信息设置，只在使能突发访问模式才有效
FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable; // 关闭 Extend Mode
// FSMC_NORSRAMInitStructure.FSMC_AsyncWait = FSMC_AsyncWait_Disable;
FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable; // 关闭 Write Burst Mode
FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p; // 读操作时序参数
FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p; // 写操作时序参数

FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);

/*-----使能 BANK1 的子模块2-----*/
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM2, ENABLE);
```

至此，我们完成了 FSMC 总线的初始化。

4.12.5.2. Nor Flash 访问程序

在完成FSMC的初始化以后，我们就可以对Nor Flash进行读写操作了。在本实验中，首先按照Nor Flash芯片要求，初始化好FSMC总线，往Nor Flash芯片内固定地址写入一串确定的值，程序再从这一地址，读回之前写入的数据。判断写入与读出的值是否一致，通过串口和神舟III号的LED灯指示程序执行结果。

```
/*使能FSMC时钟 */
RCC_AHBPeriphClockCmd (RCC_AHBPeriph_FSMC, ENABLE);
/*配置与SRAM连接的FSMC BANK1 NOR/SRAM2*/
FSMC_NOR_Init ();

/*读取Nor Flash ID并打印*/
FSMC_NOR_ReadID (&NOR_ID);
printf ("\n\r Nor Flash ID:0x%x 0x%x", NOR_ID.Manufacturer_Code, NOR_ID.Device_Code1);
/*返回写模式*/
FSMC_NOR_ReturnToReadMode ();
/*擦除NOR FLASH中，将要写入数据的块*/
FSMC_NOR_EraseBlock (WRITE_READ_ADDR);

/*将写Nor Flash的数据BUFFER填充为从0x1234开始的连续递增的一串数据 */
Fill_Buffer (TxBuffer, BUFFER_SIZE, 0x1234);
/*将数据写入到Nor Flash中。 WRITE_READ_ADDR: 写入的起始地址*/
FSMC_NOR_WriteBuffer (TxBuffer, WRITE_READ_ADDR, BUFFER_SIZE);

/*从NOR FLASH中读回刚写入的数据。 WRITE_READ_ADDR: 读出数据的起始地址*/
FSMC_NOR_ReadBuffer (RxBuffer, WRITE_READ_ADDR, BUFFER_SIZE);

/*判断读回的数据与写入的数据是否一致*/
for (Index = 0x00; (Index < BUFFER_SIZE) && (WriteReadStatus == 0); Index++)
{
    if (RxBuffer[Index] != TxBuffer[Index])
    {
        WriteReadStatus = Index + 1;
    }
}
printf ("\n\r Nor Flash读写访问程序运行结果: ");
if (WriteReadStatus == 0)
{
    printf ("\n\r Nor Flash读写访问成功");
    GPIO_ResetBits (GPIO_LED, DS2_PIN);
}
else
{
    printf ("\n\r Nor Flash读写访问失败");
    GPIO_ResetBits (GPIO_LED, DS3_PIN);
}

while (1)
{
    GPIO_ResetBits (GPIO_LED, DS1_PIN);
    Armjishu_Delay (0x3FFFFF);
    GPIO_SetBits (GPIO_LED, DS1_PIN);
    Armjishu_Delay (0x3FFFFF);
}
```

7.32.7 下载与测试

在 [神舟III号光盘编译好的固件13.Nor Flash访问试验](#) 目录下的 Nor Flash 读写.hex 文件即为前面我们分析的 Nor Flash 访问试验编译好的固件，我们可以直接通过 JLINK V8 将固件下载到神舟 III 号开发板中，观察运行效果。

如果使用 JLINK 下载固件，请按 [如何使用 JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.32.8 实验现象

将程序下载到神舟 III 号后，重现上电运行，正常情况下，串口打印如下信息。

神舟III号 Nor Flash 读写程序
--DS1闪烁表示神舟III号正常运行
--DS2--亮，表示读写Nor Flash成功
--DS3--亮，表示读写Nor Flash失败

Nor Flash ID:0xbf 0x2782
Nor Flash 读写访问程序运行结果：
Nor Flash 读写访问成功

同时神舟 III 号的 LED 指示灯也会指示运行结果，具体 LED 指示灯的状态及其含义如下：

| LED指示灯 | 含义 |
|--------|--|
| DS1闪烁 | 程序正在执行过程中 |
| DS2亮 | 写入神舟III号SRAM的数据与读出的数据一致，也就是说访问Nor Flash成功 |
| DS3亮 | 写入神舟III号SRAM的数据与读出的数据不一致，也就是说访问Nor Flash失败 |

7.33 Nand Flash访问实验

7.33.1 Nand Flash访问实验的意义与作用

在Nor Flash访问实验一节，我们已经比较Nor Flash和Nand Flash的区别。Nand Flash由于在容量和成本方面上有明显的优势，虽然它的数据安全和可靠性方面不及Nor Flash，但是这些很大程度上，可以在软件和驱动层来避免由于Nand Flash本身的问题导致数据的丢失。

Nand Flash主要用于数据数据存储。

Nand Flash和SRAM，Nor Flash不一样，它的数据线和地址线是复用的。操作Nand Flash，需要特别的访问控制协议，在本章节，我们将一起分析，如何通过STM32的FSMC接口访问Nand Flash，以及了解Nand Flash的访问操作。

7.33.2 实验原理

FSMC扩展Nand Flash配置

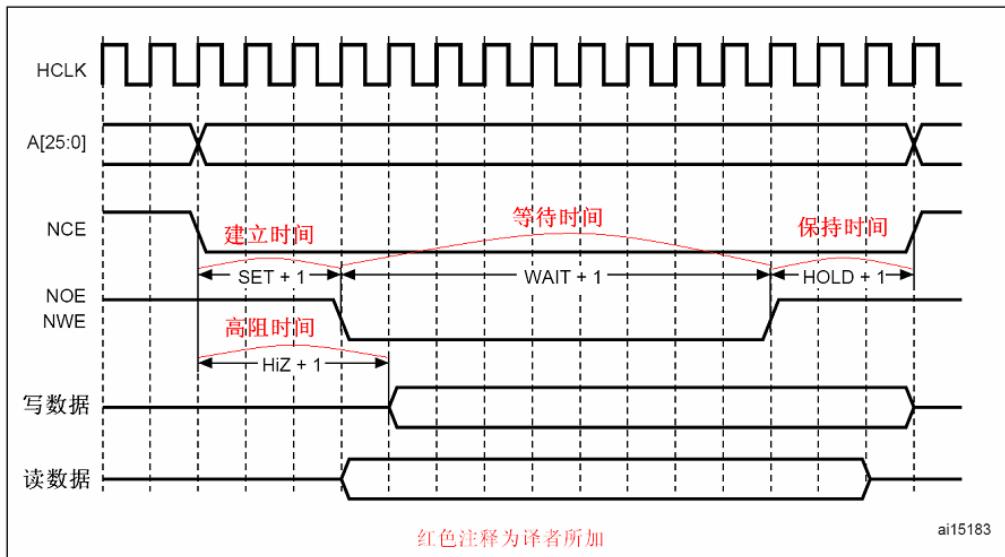
SRAM / ROM、Nor Flash和PSRAM类型的外部存储器都是由FSMC的Nor Flash控制器管理的，通过STM32处理器的FSMC访问Nand Flash之前，需要根据NAND闪存的特性初始化FSMC的NAND闪存控制器功能、时序、数据总线宽度等参数，才能访问控制NAND闪存存储器：

- 开启或关闭存储器就绪/繁忙(Ready/Busy)信号作为FSMC的输入等待。
- 开启或关闭存储器就绪/繁忙(Ready/Busy)信号作为FSMC的中断输入源：
中断可以以下述3种方式产生：
 - 在就绪/繁忙信号的上升沿产生中断：存储器刚刚完成一个操作，新的状态已经就绪。
 - 在就绪/繁忙信号的下降沿产生中断：存储器开始一个新的操作。
 - 在就绪/繁忙信号为高电平时产生中断：存储器已经就绪。
- 选择NAND存储器的数据总线宽度：8或16位。
- 开启或关闭ECC计算逻辑。
- 指定ECC计算的页面大小：可以是256、256、1024、2048、4096或8192字节/页。

用户可以配置FSMC的时序分别满足NAND闪存的不同段的操作：公共段和属性段。可配置的时序是：

- 建立时间：这是发送命令字之前地址的建立时间(以HCLK为单位)，即从地址有效至开始读写操作之间的时间。(译注：这里讲的读写操作是指对NAND内控制单元的读写，不一定是对NAND中存储单元的操作)
- 等待时间：这是发送命令字所需要的时间(以HCLK为单位)，即从NOE和NWE信号下降至上升之间的时间。
- 保持时间：这是发送命令字后地址保持的时间(以HCLK为单位)，即从NOE和NWE信号下降至上升至整个操作周期结束的时间。
- 数据总线高阻时间：这个参数只在写操作时有效，它是在开始写操作后数据总线保持高阻状态的时间(以HCLK为单位)，即从地址有效至FSMC驱动数据总线的时间。

下图显示了一个典型的NAND存储器访问的不同时序。

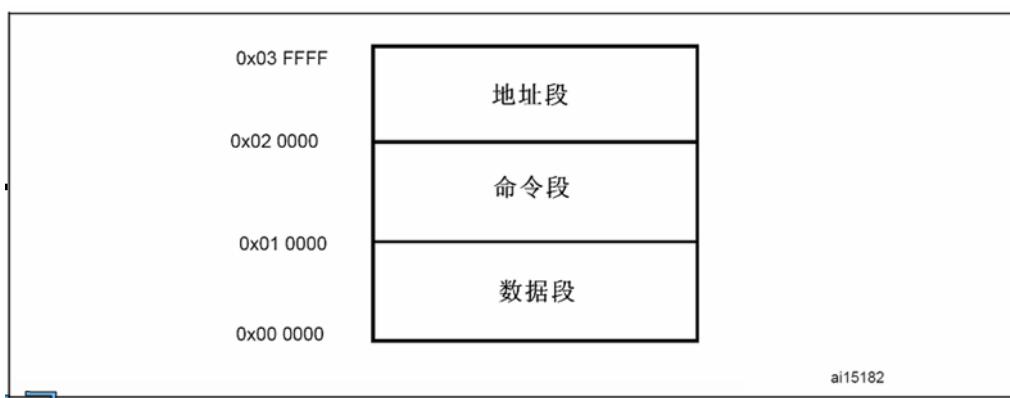


NAND FLASH操作

操作NAND闪存存储器，需要使用特别的访问协议，所有的读写操作，需要有下述步骤：

1. 向NAND闪存存储器发送一个命令
2. 发送读或写的地址
3. 读出或写入数据

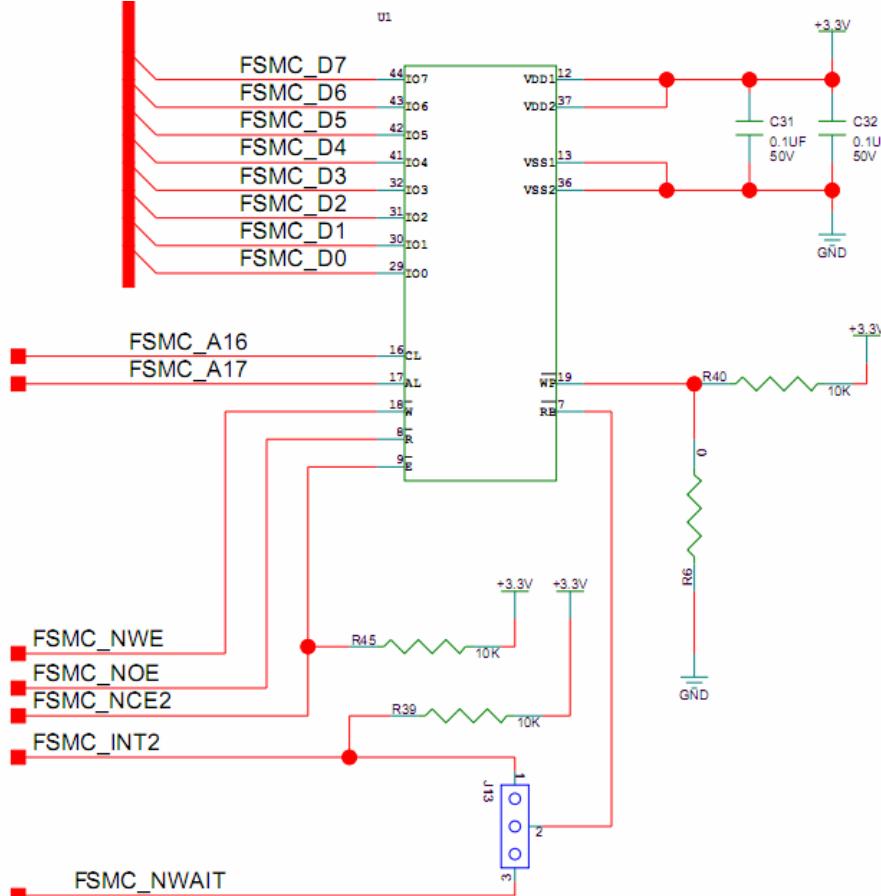
为了使用户可以方便地操作NAND闪存，FSMC的NAND存储块被划分为3个段：数据段、地址段和命令段。



实际上，这3个段的划分反映了真实的NAND闪存存储器的结构。写入命令段的任何地址，结果都是向NAND闪存写入命令。写入地址段的任何地址，结果都是向NAND闪存写入读写操作的地址；根据所用NAND闪存的构造，通常需要4~5个写入地址段才能写入一个读写操作的地址。写入或读出数据段的任何地址，结果都是写入或读出NAND的内部单元，该单元的地址是之前在地址段写入的那个地址。

7.33.3 硬件设计

在神舟 III 号中，通过 FSMC 总线与 HY27UF081G2A NAND Flash 连接，具体电路如下：



HY27UF081G2A芯片与FSMC管脚的对应关系

| NAND FLASH信号 | FSMC信号 | 管脚/端口分配 | 信号说明 |
|--------------|------------|---------|----------|
| AL | ALE/A17 | PD11 | 地址锁存信号 |
| CL | CLE/A16 | PD12 | 命令锁存使能 |
| I/O0~7 | D0~7 | 端口D/端口E | 数据总线D0~7 |
| /E | NCE2 | PD7 | 片选使能 |
| /R | NOR | PD4 | 输出使能 |
| /W | NWE | PD5 | 写使能 |
| R/B | NWAIT/INT2 | PD6/PG6 | 就绪/繁忙信号 |

神舟III号中NAND的就绪/繁忙信号可选择连接至FSMC_NWAIT信号，或者FSMC_INT2信号。

如果NAND的就绪/繁忙信号连接至FSMC_NWAIT管脚，就初始化时需要使用等待功能管理NAND闪存的操作。

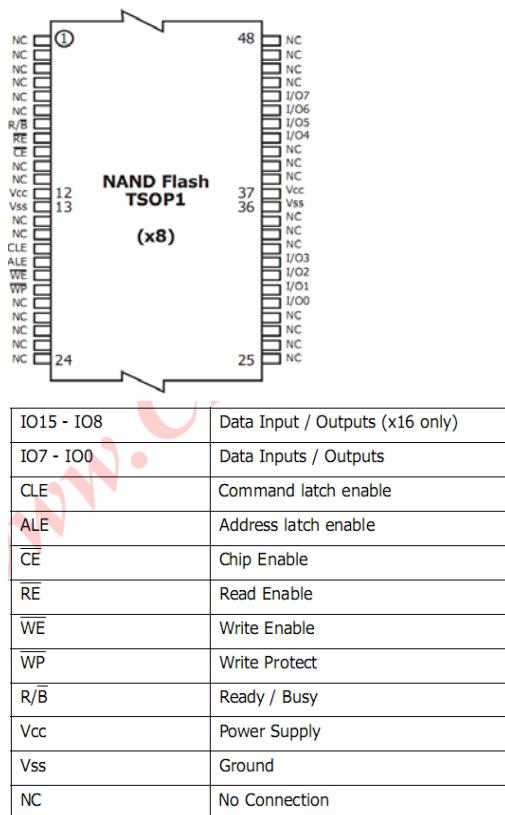
如果NAND的就绪/繁忙信号连接到INT2，作为一个中断源使用，这样CPU可以在NAND闪存操作的等待周期内执行其他的任务。把这个信号作为中断源使用时有3种用法，可以选择就绪/繁忙信号的上升沿、下降沿或高电平触发中断。

.NAND Flash 芯片说明

在神舟III号中，使用的NAND Flash是Hynix公司推出的HY27UF081G2A NAND Flash。它的容量为1G比特。HY27UF081G2A的特性如下：

- NAND接口：8位总线宽度，复用的地址/数据线。
- 页大小：(2K + 64)字节
- 页读/编程时序：
 - 随机访问：25 μ s (最大)
 - 顺序访问：30ns (最小)
 - 页编程时间：200 μ s(典型值)

HY27UF081G2A 的管脚定义如下：



7.33.4 软件设计

神舟III号Nand Flash读写试验位于*神舟III号光盘\源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\Project\14.Nand Flash访问实验(神舟III号)*目录。

进入*13.Nand Flash访问实验(神舟III号)\MDK-ARM* 目录后，双击Project.uvproj可以打开工程，以下为工程文件中主要代码的解释与说明。

应用STM32固件对FSMC进行初始化配置

ST 公司为用户开发提供了完整、高效的工具和固件库，其中使用 C 语言编写的固件库提供了覆盖所有标准外设的函数，使用户无需使用汇编操作外设特性，从而提高了程序的可读性和易维护性。

在使用Nand Flash之前，我们需要对Nand Flash使用的FSMC BANK进行参数配置，使之与Nand Flash芯片的要求相符合。

在前面的硬件设计原理部分，我们可以得到以下参数。

| | 参数 | 说明 |
|---------------|------------|------------------------------|
| 存储器类型 | Nand Flash | |
| 占用的 FSMC BANK | BANK2 | 硬件上与 STM32F103 的 FSMC_NE2 相连 |
| 储存器数据宽度 | 8 位 | 一共 8 位数据线，D0~7 |
| ECC 页大小 | 2048 字节 | |

◆ FSMC管脚初始化

Nand Flash里面占用的GPIO管脚如下表所示：

| NAND FLASH信号 | FSMC信号 | 管脚/端口分配 | 信号说明 |
|--------------|------------|---------|----------|
| AL | ALE/A17 | PD11 | 地址锁存信号 |
| CL | CLE/A16 | PD12 | 命令锁存使能 |
| I/O0~7 | D0~7 | 端口D/端口E | 数据总线D0~7 |
| /E | NCE2 | PD7 | 片选使能 |
| /R | NOR | PD4 | 输出使能 |
| /W | NWE | PD5 | 写使能 |
| R/B | NWAIT/INT2 | PD6/PG6 | 就绪/繁忙信号 |

在STM32技术参考手册中，针对FSMC引脚的GPIO模式配置，已经进行了说明。具体FSMC总线的配置如下：

| FSMC引脚 | GPIO配置 |
|---------------|------------|
| FSMC_A[25:0] | 推挽复用输出 |
| FSMC_D[15:0] | |
| FSMC_CK | 推挽复用输出 |
| FSMC_NOE | 推挽复用输出 |
| FSMC_NWE | |
| FSMC_NE[4:1] | |
| FSMC_NCE[3:2] | 推挽复用输出 |
| FSMC_NCE4_1 | |
| FSMC_NCE4_2 | |
| FSMC_NWAIT | 浮空输入或带上拉输入 |
| FSMC_CD | |
| FSMC_NIOS16 | |
| FSMC_INTR | 浮空输入 |
| FSMC_INT[3:2] | |
| FSMC_NL | 推挽复用输出 |
| FSMC_NBL[1:0] | |
| FSMC_NIORD | |
| FSMC_NIOWR | 推挽复用输出 |
| FSMC_NREG | |

说明，上表是整个FSMC接口的引脚的初始化配置说明，实际初始化时可只初始化具体存储器使用的FSMC引脚。

如下为神舟 III 号访问 Nand Flash，FSMC 引脚的初始化，只初始化了与 Nand Flash 连接的相关引脚。

```
/*FSMC总线使用的GPIO组时钟使能*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOE |
RCC_APB2Periph_GPIOF | RCC_APB2Periph_GPIOG, ENABLE);

/*FSMC CLE, ALE, D0->D3, NOE, NWE and NCEB初始化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_14 | GPIO_Pin_15 |
GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_4 | GPIO_Pin_5 |
GPIO_Pin_7;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;

GPIO_Init(GPIOD, &GPIO_InitStructure);

/*FSMC数据线FSMC_D[4:7]初始化，推挽复用输出*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10;

GPIO_Init(GPIOE, &GPIO_InitStructure);

/*FSMC_NWAIT初始化，输入上拉*/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;

GPIO_Init(GPIOD, &GPIO_InitStructure);
/*FSMC_INT2初始化，输入上拉*/

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_Init(GPIOG, &GPIO_InitStructure);
```

◆ 配置FSMC存储器时间参数

在完成FSMC引脚的初始化以后，我们还需要配置FSMC可编程的存储器的参数。STM32 固件库中提供的 FSMC 的 Nand Flash 的初始化主要包括 2 个数据结构。数据结构 `FSMC_NAND_PCCARDTimingInitTypeDef` 对应时间参数寄存器定义。

针对NAND控制器，我们需要初始化下表中的参数。

| 参数 | 功能 | 操作模式 | 单位 | 最小 | 最大 |
|-------------|---|------|----------------|----|-----|
| 存储器建立时间 | 发出命令之前建立地址的 (HCLK) 时钟周期数目 | 读/写 | AHB时钟周期 (HCLK) | 1 | 256 |
| 存储器等待时间 | 发出命令的最短持续时间 (HCLK周期数目) | | | 1 | 256 |
| 存储器保持时间 | 在发送命令结束后保持地址的 (HCLK) 时钟周期数目，写操作时也是数据的保持时间 | | | 1 | 255 |
| 存储器数据总线高阻时间 | 启动写操作之后保持数据总线为高阻态的时间 | | | 0 | 255 |

这些参数需要根据NAND存储器的特性和STM32F10xxx的HCLK时钟计算。

根据Nand Flash存储器数据手册中访问时序参数，可以得到下述公式：

读或写访问时间是NAND存储器的片选信号，从下降沿至上升沿之间的时间，这是FSMC时序参数的函数：

$$\text{读/写访问时间} = ((SET + 1) + (WAIT + 1) + (HOLD + 1)) \times HCLK$$

等待时间是读/写使能信号，从下降沿至上升沿之间的时间：

$$\text{读/写使能信号低至高时间} = (WAIT + 1) \times HCLK$$

对于读操作，数据总线高阻时间(HiZ)是由片选建立时间和数据建立时间衡量：

$$\text{片选建立时间} - \text{数据建立时间} = HiZ \times HCLK$$

保持时间参数可以在第一个公式中获得。实际上，NAND存储器的数据手册给出了写操作中片选低至写使能高的时序，保持时间可以由此计算得出：

$$\text{片选低至写使能高时间} = ((SET + 1) + (WAIT + 1)) \times HCLK$$

为了保证正确地配置FSMC的时序，下述因素应加以考虑：

- 最大读/写访问时间
- FSMC内部各部分的延迟
- 存储器内部各部分的延迟

因此，我们得到下述公式：

$$(WAIT + 1) \times HCLK = \max(t_{WP}, t_{RP})$$

$$((SET + 1) + (WAIT + 1)) \times HCLK = \max(t_{CS}, t_{ALS}, t_{CLS})$$

$$HOLD = \max(t_{CH}, t_{ALH}, t_{CLH}) / HCLK$$

还需要满足下述公式的验证：

$$((SET + 1) + (WAIT + 1) + (HOLD + 1)) \times HCLK = \max(t_{RC}, t_{WC})$$

$$HiZ = (\max(t_{CS}, t_{ALS}, t_{CLS}) - t_{DS}) / HCLK - 1$$

考虑FSMC和存储器内部各部分的延迟，这些公式变为如下形式：

- WAIT需要满足：

$$(WAIT+1+ SET + 1) = ((t_{CEA} + t_{su(Data_NE)} + t_{v(A_NE)}) / HCLK)$$

$$WAIT = ((t_{CEA} + t_{su(Data_NE)} + t_{v(A_NE)}) / HCLK) - SET - 2$$

- SET需要满足

$$(SET + 1) = \max((t_{CS}, t_{ALS}, t_{CLS}) - \max(t_{WP}, t_{RP})) / HCLK - 1$$

$$SET = (\max(t_{CS}, t_{ALS}, t_{CLS}) - \max(t_{WP}, t_{RP})) / HCLK - 1$$

下表列出了NAND存储器各项参数的意义和时序。

| Parameter | Symbol | 3.3Volt | | Unit |
|--|---------------------|---------|-------------------------|------|
| | | Min | Max | |
| CLE Setup time | tCLS | 15 | | ns |
| CLE Hold time | tCLH | 5 | | ns |
| CE setup time | tCS | 20 | | ns |
| CE hold time | tCH | 5 | | ns |
| WE pulse width | tWP | 15 | | ns |
| ALE setup time | tALS | 15 | | ns |
| ALE hold time | tALH | 5 | | ns |
| Address to Data Loading | tADL ⁽⁴⁾ | 100 | | ns |
| Data setup time | tDS | 5 | | ns |
| Data hold time | tDH | 5 | | ns |
| Write Cycle time | tWC | 30 | | ns |
| WE High hold time | tWH | 10 | | ns |
| Data Transfer from Cell to register | tR | | 25 | us |
| ALE to RE Delay | tAR | 15 | | ns |
| CLE to RE Delay | tCLR | 15 | | ns |
| Ready to RE Low | tRR | 20 | | ns |
| RE Pulse Width | tRP | 15 | | ns |
| WE High to Busy | tWB | | 100 | ns |
| Read Cycle Time | tRC | 30 | | ns |
| RE Access Time | tREA | | 20 | ns |
| RE High to Output High Z | tRHZ | | 50 | ns |
| CE High to Output High Z | tCHZ | | 50 | ns |
| RE High Hold Time | tREH | 10 | | ns |
| Output High Z to RE low | tZR | 0 | | ns |
| CE Access Time | tCEA | | 25 | ns |
| WE High to RE low | tWHR | 60 | | ns |
| RE or CE High to Output Hold | tOH | 10 | | ns |
| Device Resetting Time (Read / Program / Erase) | tRST | | 5/10/500 ⁽¹⁾ | us |
| Write Protection time | tWW ⁽³⁾ | 100 | | ns |

使用上述公式、存储器时和STM32F10xxx参数，我们得到：

- 地址建立时间：0x1
- 地址保持时间：0x3
- 数据建立时间：0x2
- 数据总线高阻时间：0x1

对应神舟III号Nor Flash访问程序中，对应的存储器的参数配置程序如下。

```
/*-----FSMC 总线 存储器参数配置-----*/
p.FSMC_SetupTime = 0x1;           //建立时间
p.FSMC_WaitSetupTime = 0x3;        //等待时间
p.FSMC_HoldSetupTime = 0x2;         //保持时间
p.FSMC_HIZSetupTime = 0x1;          //高阻建立时间
```

◆ 配置FSMC存储器时间参数

FSMC_NorSRAMInitTypeDef 对应特征配置寄存器 FSMC_BCR 的结构定义，并包含 2 个指向对应 BANK 的 FSMC_BTR 和 FSMC_BWTR 寄存器的 FSMC_NorSRAMTimingInitTypeDef 结构指针。对应神舟 III 号中的初始化代码如下：

```
FSMC_NANDInitTypeDef FSMC_NANDInitStructure;

FSMC_NANDInitStructure.FSMC_Bank = FSMC_Bank2_NAND; // 使用FSMC_BANK2
FSMC_NANDInitStructure.FSMC_Waitfeature = FSMC_Waitfeature_Enable; // 使能FSMC的等待功能
FSMC_NANDInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_8b; // NAND Flash的数据宽度为8位
FSMC_NANDInitStructure.FSMC_ECC = FSMC_ECC_Enable; // 使能ECC特性
FSMC_NANDInitStructure.FSMC_ECCPageSize = FSMC_ECCPageSize_2048Bytes; // ECC页大小2048
FSMC_NANDInitStructure.FSMC_TCLRSetupTime = 0x00;
FSMC_NANDInitStructure.FSMC_TARSetupTime = 0x00;
FSMC_NANDInitStructure.FSMC_CommonSpaceTimingStruct = &p;
FSMC_NANDInitStructure.FSMC_AttributeSpaceTimingStruct = &p;

FSMC_NANDInit(&FSMC_NANDInitStructure);

/*! 使能FSMC_BANK2 */
FSMC_NANDCmd(FSMC_Bank2_NAND, ENABLE);
```

至此，我们完成了 FSMC 总线的初始化。

4.14.4.2. Nand Flash 访问程序

在完成FSMC的初始化以后，我们就可以对Nand Flash进行读写操作了。在本实验中，首先按照Nand Flash芯片要求，初始化好FSMC总线，读取Nand Flash ID并打印，如果Nand Flash的ID读取成功，则往Nand Flash芯片内固定地址写入一串确定的值，程序再从这一地址，读回之前写入的数据。判断写入与读出的值是否一致，通过串口和神舟III号的LED灯指示程序执行结果。

```
/*使能FSMC时钟 */
RCC_AHBPerrifClockCmd(RCC_AHBPeriph_FSMC, ENABLE);

/*配置与SRAM连接的FSMC BANK2 NAND*/
NAND_Init();

/*读取Nand Flash ID并打印*/
NAND_ReadID(&NAND_ID);
printf("\n\r Nand Flash ID:0x%x\t 0x%x\t 0x%x\t 0x%x",NAND_ID.Maker_ID,NAND_ID.Device_ID,
      NAND_ID.Third_ID,NAND_ID.Fourth_ID);

/*校验Nand Flash 的ID是否正确*/
if((NAND_ID.Maker_ID == NAND_HY_MakerID) && (NAND_ID.Device_ID == NAND_HY_DeviceID))
{
    /*设置NAND FLASH的写地址*/
    WriteReadAddr.Zone = 0x00;
    WriteReadAddr.Block = 0x00;
    WriteReadAddr.Page = 0x00;

    /*擦除待写入数据的块*/
    status = NAND_EraseBlock(WriteReadAddr);

    /*将写Nand Flash的数据BUFFER填充为从0x25开始的连续递增的一串数据 */
    Fill_Buffer(TxBUFFER, BUFFER_SIZE, 0x25);
    /*将数据写入到Nand Flash中。WriteReadAddr:读写的起始地址*/
    status = NAND_WriteSmallPage(TxBUFFER, WriteReadAddr, PageNumber);

    /*从Nand Flash中读回刚写入的数据。WriteReadAddr:读写的起始地址*/
    status = NAND_ReadSmallPage(RxBUFFER, WriteReadAddr, PageNumber);

    /*判断读回的数据与写入的数据是否一致*/
    for(j = 0; j < BUFFER_SIZE; j++)
    {
        if(TxBUFFER[j] != RxBUFFER[j])
        {
            WriteReadStatus++;
        }
    }
    printf("\n\r Nand Flash读写访问程序运行结果: ");
    if(WriteReadStatus == 0)
    {
        printf("\n\r Nand Flash读写访问成功");
        GPIO_ResetBits(GPIO_LED, DS2_PIN);
    }
    else
    {
        printf("\n\r Nand Flash读写访问失败");
        printf("0x%x", WriteReadStatus);

        GPIO_ResetBits(GPIO_LED, DS3_PIN);
    }
} ? end if (NAND_ID.Maker_ID==NA... ?
else
{
    printf("\n\r 没有检测到Nand Flash的ID");
    GPIO_ResetBits(GPIO_LED, DS4_PIN);
}

while(1)
{
    GPIO_ResetBits(GPIO_LED, DS1_PIN);
    Armjishu_Delay(0x3FFFFFF);
    GPIO_SetBits(GPIO_LED, DS1_PIN);
    Armjishu_Delay(0x3FFFFFF);
}
```

7.33.5 下载与测试

在 [神舟III号光盘\编译好的固件\14.Nand Flash访问试验](#) 目录下的 Nand Flash 读写.hex 文件即为前面我们分析的 Nand Flash 访问试验编译好的固件，我们可以直接通过 JLINK V8 将固件下载到神舟 III 号开发板中，观察运行效果。

如果使用 JLINK 下载固件，请按 [如何使用 JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.33.6 实验现象

将程序下载到神舟 III 号后，重现上电运行，正常情况下，串口打印如下信息。

```
-----  
神舟III号 Nand Flash读写程序  
--DS1闪烁表示神舟III号正常运行  
--DS2--亮， 表示读写Nand Flash成功  
--DS3--亮， 表示读写Nand Flash失败  
--DS4--亮， 表示没有读到Nand Flash的ID  
-----  
Nand Flash ID:0xad      0xf1      0x80      0x1d  
Nand Flash读写访问程序运行结果：  
Nand Flash读写访问成功
```

同时神舟 III 号的 LED 指示灯也会指示运行结果，具体 LED 指示灯的状态及其含义如下：

| LED指示灯 | 含义 |
|--------|---------------------------------------|
| DS1闪烁 | 程序正在执行过程中 |
| DS2亮 | 写入神舟III号SRAM的数据与读出的数据一致，也就是说访问SRAM成功 |
| DS3亮 | 写入神舟III号SRAM的数据与读出的数据不一致，也就是说访问SRAM失败 |
| DS4亮 | 没有读到Nand Flash的ID，或者ID不正确 |

7.34 2.4G无线模块实验

7.34.1 2.4G无线模块通讯实验的意义与作用

现实生活中，无线通信到处存在，手机、电视、无线遥控以及卫星等等。很多爱好者非常期望了解无线通信时如何实现的？从信号的编码，信道传输，信号编码以及传输过程中，根据距离控制发送功率等等？以下我们将简单了解这一系列的实现过程。

7.34.2 试验原理

首先我们简单了解nRF24L01无线模块的特点以及工作原理。

nRF24L01无线模块，主要芯片是nRF24L01，其特点如下：

- ◆ 采用 GFSK 方式调制，数据传输率为 1Mb/s 或者 2Mb/s;
- ◆ 具有自动应答和自动再发射功能；
- ◆ 125 个频道，可以满足多点通信；
- ◆ 具有 CRC 校验；
- ◆ 低电压供电：1.9V~3.6V；

nRF24L01内置频率合成器、功率放大器、晶体振荡器、调制器等功能模块。其功耗低，在以-6dBm 功率发射时，工作电流只有9mA；而接收时，工作电流只有12.3mA。接下来，我们一起了解nRF24L01 的收发原理。

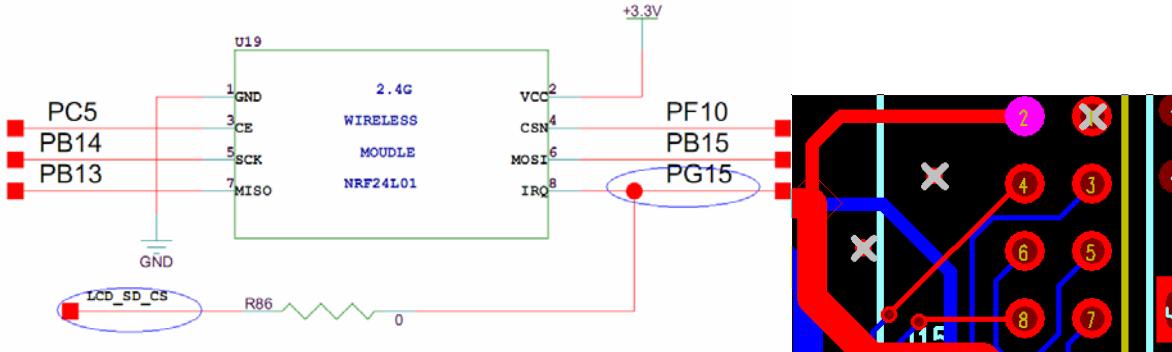
发送数据时，首先将nRF24L01配置为发送模式；接着把发送地址和发送数据按照时序要求经过SPI 总线写入nRF24L01缓存区，发送数据必须在SPI片选CSN为低时，连续写入。而发送地址在发射时写入一次即可，然后使能管脚置高并保持10uS以上，同时延迟130uS后发送数据；如果开启自动应答功能时，nRF24L01在发送数据后就进入接收模式，接收应答信号。若收到应答，则表示此次通信成功；若没有收到应答，就自动重新发射该数据（自动重启功能需开启）。当发送成功后，IRQ中断标志变低，通过SPI通知处理器。在一次发送成功后，如果发送堆栈中有数据而且使能为高时，则进入下一次发射；否则进入空闲模式。

在接收数据时，先将nRF24L01设置为接收模式，接着延迟130uS进入接收状态等待数据的到来。当接收检测到有效的地址和CRC时，就将数据包存储在接收堆栈中，同时中断标志位IRQ置低，通知处理器取数据。如果开启自动应答，接收方同时接入发射状态，回传应答信号。直到接收结束，便将使能关闭，进入空闲模式。

关于nRF24L01的配置以及模式选择等详细资料介绍，请参考nRF24L01的详细资料。

7.34.3 硬件设计

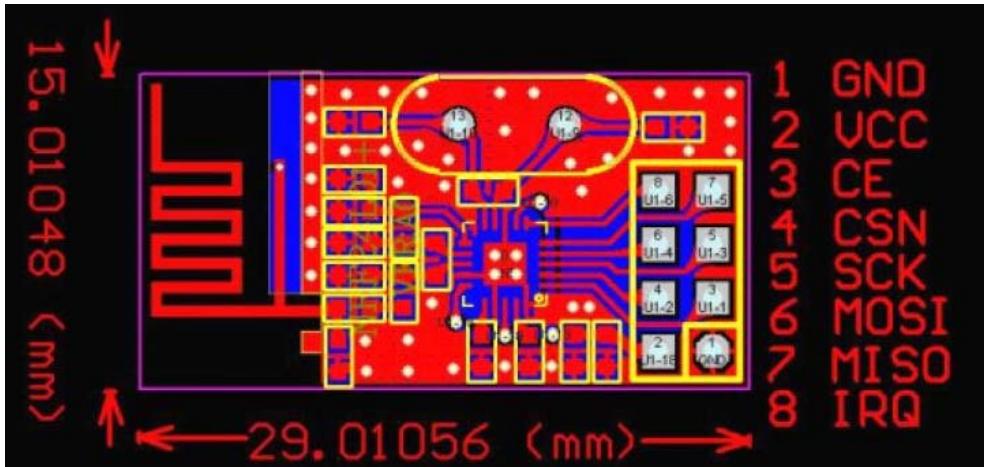
2.4G无线模块的通信，处理器通过SPI总线控制nRF24L01模块。在上电后先检测nRF24L01模式是否在位。如果在位，便提示选择nRF24L01模块的工作模式：发送或接收。硬件设计如下：



GPIO 管脚与 2.4G 无线模块 nRF24L01 管脚对应关系

| nRF24L01 管脚 | GPIO管脚 | PCB管脚 | SPI信号 | 说明 |
|-------------|--------|-------|-----------|---|
| SCK | PB14 | 5 | SPI2_SCK | SPI2接口信号 |
| MISO | PB13 | 7 | SPI2_MISO | |
| MOSI | PB15 | 6 | SPI2_MOSI | |
| CE | PC5 | 3 | — | 由于 SPI2 上除了与 2.4G 无线模块 nRF24L01X 相连以外，还与音频 DA 芯片 PCM1770 以及 TFT 的触摸芯片相连，而 SPI2 本身只有一根 CS 信号，因此在这里使用 PC5 作为 2.4G 无线模块 nRF24L01 的 SPI2 接口的 CS 信号。 |
| CSN | PF10 | 4 | — | 2.4G无线模块nRF24L01的模块选择信号 |
| IRQ | PG15 | 8 | — | 2.4G 无线模块 nRF24L01 的中断输出信号 注意： 由于硬件资源的限制，PG15 除了与 2.4G 无线模块 nRF24L01 模块的中断输出连接外，还与 LCD 模块上的 SD 卡片选信号相连，因此在使用神舟 III 号时，2.4G 无线模块 nRF24L01 与屏上的 SD 卡不能同时使用。 |
| GND | — | 1 | — | 地信号 |
| VCC | — | 2 | — | 电源输入 |

而2.4G无线模块管脚定义如下。



请用随神舟III号STM32开发板配送的杜邦线连接2.4G模块与板上的2.4G无线模块的座，具体连接对应关系如下。

| nRF24L01无线模块 | 神舟III号2.4G模块扩展座 | 信号名 |
|--------------|-----------------|------|
| 1 | 1 | GND |
| 2 | 2 | VCC |
| 3 | 3 | PC5 |
| 4 | 4 | PF10 |
| 5 | 7 | PB13 |
| 6 | 6 | PB15 |
| 7 | 5 | PB14 |
| 8 | 8 | PG15 |

7.34.4 软件设计

本例程的试验思想是借用两块带有2.4G无线模块nRF24L01的神舟III号开发板，上电初始化检测nRF24L01模块是否在位，如果没有在位，则提示检查nRF24L01无线模块的连接情况；如果在位，则提示选择nRF24L01工作模式：发送或者接收，在开发板上USER1按键和USER2按键分别对应着接收模式和发送模式。通过选择USER1或USER2按键，进行无线的通信过程。而对于按键的初始化及按键检测，在前面的章节已经详细讲解，在此不赘述。

神舟III号Nand Flash读写试验位于 [神舟III号光盘\源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\Project\16.2.4G无线模块实验（神舟III号）](#) 目录。

进入 [16.2.4G无线模块实验（神舟III号）\MDK-ARM](#) 目录后，双击Project.uvproj可以打开工程，以下为工程文件中主要代码的解释与说明。

◆ LED 与按键等初始化

主要是2.4G无线模块实验需要用到的串口，LED灯和按键的初始化。配置LED灯使用的GPIO管脚为推挽输出模式，按键使用的GPIO管脚配置为上拉输入模式，另外配置了串口1作为我们的输入输出设备，波特率为115200，相关的函数如下。

```
LED_Config();           //LED使用的GPIO接口初始化
GPIO_SetBits(GPIO_LED, DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN); //关闭所有的LED指示灯

/* USARTx configured as follow:
   - BaudRate = 115200 baud
   - Word Length = 8 Bits
   - One Stop Bit
   - No parity
   - Hardware flow control disabled (RTS and CTS signals)
   - Receive and transmit enabled
*/
USART_InitStructure.USART_BaudRate = 115200;           //配置串口
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
STM_EVAL_COMInit(COM1, &USART_InitStructure);      //串口使用声明

KEY_Config();           //按键使用的GPIO接口初始化
```

◆ NRF24L01 无线模块访问接口初始化

在实验例程中，NRF24L01_Init() 函数来完成无线模块使用的接口的初始化，主要是分配给 NRF24L01 的 CE 信号以及 SPI CS 信号的舒适化。以及 SPI2 接口的初始化。

```
void NRF24L01_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_NRF24L01_CE, ENABLE); //使能GPIO的时钟
    GPIO_InitStructure.GPIO_Pin = NRF24L01_CE;           //NRF24L01 模块片选信号
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;      //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_NRF24L01_CE, &GPIO_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_NRF24L01_CSN, ENABLE); //使能GPIO的时钟
    GPIO_InitStructure.GPIO_Pin = NRF24L01_CSN;          //推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_NRF24L01_CSN, &GPIO_InitStructure);

    Set_NRF24L01_CE; //初始化时先拉高
    Set_NRF24L01_CSN; //初始化时先拉高

    //配置NRF24L01的IRQ
    GPIO_InitStructure.GPIO_Pin = NRF24L01_IRQ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_NRF24L01_IRQ, &GPIO_InitStructure);
    GPIO_SetBits(GPIO_NRF24L01_IRQ, NRF24L01_IRQ);

    SPI2_Init(); //初始化SPI
    Clr_NRF24L01_CE; //使能NRF24L01
    Set_NRF24L01_CSN; //SPI片选取消
} ? end NRF24L01_Init ?
```

其中 SPI2_Init 函数是 SPI2 接口的初始化函数，配置 SPI2 的时钟频率以及工作模式，具体函数如下，它与前面所讲的 SPI FLASH (W25X16) 访问的 SPI 接口初始化基本相同。主要区别是在 SPI 的工作模式和工作频率区别，如下图所示：

```
//串行外设接口SPI的初始化，SPI配置成主模式
//本例程选用SPI1对NRF24L01进行读写操作，先对SPI1进行初始化
void SPI2_Init(void)
{
    SPI_InitTypeDef SPI_InitStruct;
    GPIO_InitTypeDef GPIO_InitStruct;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|RCC_APB2Periph_AFIO, ENABLE );
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE );

    /*SPI1口初始化
    /* Configure SPI1 pins: SCK, MISO and MOSI */
    GPIO_InitStruct.GPIO_Pin = SPI2_MISO| SPI2_MOSI| SPI2_SCK;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_SPI2, &GPIO_InitStruct);

    GPIO_SetBits(GPIO_SPI2, SPI2_MISO| SPI2_MOSI| SPI2_SCK); //初始化SPI1结构体

    /* SPI2 configuration */
    SPI_InitStruct.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //SPI1设置为两线全双工
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master; //设置SPI1为主模式
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b; //SPI发送接收8位帧结构
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_Low; //串行时钟在不操作时，时钟为低电平
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_1Edge; //第一个时钟沿开始采样数据
    SPI_InitStruct.SPI_NSS = SPI NSS_Soft; //NSS信号由软件（使用SSI位）管理
    SPI_InitStruct.SPI_BaudRatePrescaler =SPI_BaudRatePrescaler_8; //SPI波特率预分频值为8
    SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB; //数据传输从MSB位开始
    SPI_InitStruct.SPI_CRCPolynomial = 7; //CRC值计算的多项式

    SPI_Init(SPI2, &SPI_InitStruct); //根据SPI_InitStruct中指定的参数初始化外设SPI2寄存器

    /* Enable SPI2 */
    SPI_Cmd(SPI2, ENABLE); //使能SPI1外设

    SPI2_ReadWriteByte(0xff); //启动传输
} ? end SPI2_Init ?
```

在完成上述程序后，我们就可以通过SPI2接口正常访问无线模块了，在实际使用无线模块收发数据之前我们还需要设计模块的工作模式。正常情况下，一个带有2.4G无线模块的神舟III号设置为发送模式后，另一个带有2.4G无线模块的神舟III号开发板设置为接收模式，即可接收到前一个模块发送的数据。因此，设置2.4G无线模块的发送接收模式，主要是通过以下两个函数RX_Mode和Tx_Mode来设置无线模块的工作模式。相关代码如下：

```
//该函数初始化NRF24L01到RX模式
//设置RX地址,写RX数据宽度,选择RF频道,波特率和LNA HCURR
//当CE变高后,即进入RX模式,并可以接收数据了
void RX_Mode(void)
{
    Clr_NRF24L01_CE;
    NRF24L01_Write_Buf(SPI_WRITE_REG+RX_ADDR_P0, (u8*) RX_ADDRESS, RX_ADR_WIDTH); //写RX节点地址

    NRF24L01_Write_Reg(SPI_WRITE_REG+EN_AA, 0x01); //使能通道0的自动应答
    NRF24L01_Write_Reg(SPI_WRITE_REG+EN_RXADDR, 0x01); //使能通道0的接收地址
    NRF24L01_Write_Reg(SPI_WRITE_REG+RF_CH, 40); //设置RF通信频率
    NRF24L01_Write_Reg(SPI_WRITE_REG+RX_PW_P0, RX_PLOAD_WIDTH); //选择通道0的有效数据宽度
    NRF24L01_Write_Reg(SPI_WRITE_REG+RF_SETUP, 0x0f); //设置TX发射参数,0db增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(SPI_WRITE_REG+CONFIG, 0x0f); //配置基本工作模式的参数,PWR_UP,EN_CRC,16BIT_CRC,接收模式
    Set_NRF24L01_CE;
}

//该函数初始化NRF24L01到TX模式
//设置TX地址,写TX数据宽度,设置RX自动应答的地址,填充TX发送数据,选择RF频道,波特率和LNA HCURR
//PWR_UP,CRC使能
//当CE变高后,即进入RX模式,并可以接收数据了
//CE为高大于10us,则启动发送.
void TX_Mode(void)
{
    Clr_NRF24L01_CE;
    NRF24L01_Write_Buf(SPI_WRITE_REG+TX_ADDR, (u8*) TX_ADDRESS, TX_ADR_WIDTH); //写TX节点地址
    NRF24L01_Write_Buf(SPI_WRITE_REG+RX_ADDR_P0, (u8*) RX_ADDRESS, RX_ADR_WIDTH); //设置TX节点地址,主要为了使能ACK

    NRF24L01_Write_Reg(SPI_WRITE_REG+EN_AA, 0x01); //使能通道0的自动应答
    NRF24L01_Write_Reg(SPI_WRITE_REG+EN_RXADDR, 0x01); //使能通道0的接收地址
    NRF24L01_Write_Reg(SPI_WRITE_REG+SETUP_RETR, 0x1a); //设置自动重发间隔时间:500us + 86us,最大自动重发次数:10次
    NRF24L01_Write_Reg(SPI_WRITE_REG+RF_CH, 40); //设置RF通道为40
    NRF24L01_Write_Reg(SPI_WRITE_REG+RF_SETUP, 0x0f); //设置TX发射参数,0db增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(SPI_WRITE_REG+CONFIG, 0x0e); //配置基本工作模式的参数,PWR_UP,EN_CRC,16BIT_CRC,接收模式,开启所有中断
    Set_NRF24L01_CE;
}
```

最后，我们就可以通过NRF24L01_RxPacket和NRF24L01_TxPacket两个函数启动对应的接收和发送流程。

```
//启动NRF24L01发送一次数据
//txbuf:待发送数据首地址
//返回值:发送完成状况
u8 NRF24L01_TxPacket(u8 *txbuf)
{
    u8 state;
    Clr_NRF24L01_CE;
    NRF24L01_Write_Buf(WR_TX_PLOAD, txbuf, TX_PLOAD_WIDTH); //写数据到TX BUF 32个字节
    Set_NRF24L01_CE; //启动发送
    while(READ_NRF24L01_IRQ!=0); //等待发送完成
    state=NRF24L01_Read_Reg(STATUS); //读取状态寄存器的值
    NRF24L01_Write_Reg(SPI_WRITE_REG+STATUS, state); //清除TX_DS或MAX_RT中断标志
    if(state&MAX_TX) //达到最大重发次数
    {
        NRF24L01_Write_Reg(FLUSH_TX, 0xFF); //清除TX FIFO寄存器
        return MAX_TX;
    }
    if(state&TX_OK) //发送完成
    {
        return TX_OK;
    }
    return 0xff; //其他原因发送失败
} ? end NRF24L01_TxPacket ?
//启动NRF24L01发送一次数据
//txbuf:待发送数据首地址
//返回值:0, 接收完成; 其他, 错误代码
u8 NRF24L01_RxPacket(u8 *rxbuf)
{
    u8 state;
    state=NRF24L01_Read_Reg(STATUS); //读取状态寄存器的值
    NRF24L01_Write_Reg(SPI_WRITE_REG+STATUS, state); //清除TX_DS或MAX_RT中断标志
    if(state&RX_OK) //接收到数据
    {
        NRF24L01_Read_Buf(RD_RX_PLOAD, rxbuf, RX_PLOAD_WIDTH); //读取数据
        NRF24L01_Write_Reg(FLUSH_RX, 0xFF); //清除RX FIFO寄存器
        return 0;
    }
    return 1; //没收到任何数据
}
```

7.34.5 下载与测试

在 [神舟III号光盘编译好的固件16. 2.4G无线模块试验](#) 目录下的2.4G无线模块实验.hex文件即为本节我们分析的实验所编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟III号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.34.6 实验现象

将一块带有2.4G无线模块的神舟III号开发板烧录好固件后，上电运行，按USER1按键设置为接收模式，同时，将另一块烧录好固件的带有2.4G无线模块的神舟III号开发板，上电运行，按USER2键设置为发送模式。

正常情况下，处于发送模式的神舟III号串口1将打印如下信息。

神舟III号2.4G无线模块实验程序

```
请设置NRF24L01无线模块的工作模式
--USER1 按键:设置NRF24L01为接收模式
--USER2 按键:设置NRF24L01为发送模式
--TAMEPR按键:退出NRF24L01发送接收
```

NRF24L01 发送模式设置成功

```
正在发送数据: ! "#$%&' ()*+, -./0123456789;; <=>?@A
正在发送数据: "#$%&' ()*+, -./0123456789;; <=>?@AB
正在发送数据: ##$%&' ()*+, -./0123456789;; <=>?@ABC
正在发送数据: %$&' ()*+, -./0123456789;; <=>?@ABCD
正在发送数据: %&' ()*+, -./0123456789;; <=>?@ABCDE■
```

处于接收模式的神舟III号串口1将打印如下信息。

神舟III号2.4G无线模块实验程序

```
请设置NRF24L01无线模块的工作模式
--USER1 按键:设置NRF24L01为接收模式
--USER2 按键:设置NRF24L01为发送模式
--TAMEPR按键:退出NRF24L01发送接收
```

NRF24L01 接收模式设置成功

```
等待接收数据      接收到的数据
接收到数据为:JKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
接收到数据为:KLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
接收到数据为:LMNOPQRSTUVWXYZ[\]^_`abcdefghijkljk
接收到数据为:MNOPQRSTUVWXYZ[\]^_`abcdefghijkljk■
```

7.35 收音机实验

这一节我们将向大家介绍如何通过STM32处理器访问TEA5767收音模块，控制收音模块，使之自动搜台，并选择某一频段，将收到的音乐通过耳机播放出来。本节分为如下几个部分：

- 4.17.1 收音机实验的意思与作用 内部温度传感器简介
- 4.17.2 硬件设计
- 4.17.3 软件设计
- 4.17.4 下载与测试
- 4.17.5 实验现象

7.35.1 收音机实验的意义与作用

PHILIPS的TEA5767高灵敏度收音模块芯片，这块芯片属于低电压和低功耗的全集成单芯片FM收音产品，可完全免费调到欧洲、美国和日本的调频波段，FM频率可以支持76MHz~108MHz，收音效果非常出色，可存储50个电台频道.数码录音和高清晰度CD直录功能（LINE-IN），可以直接通过转录线把传统音响上的音乐以MP3格式录制到内置的闪存里，并可以将收音内容直接录制下来，同时它支持80MHZ以下的校园网FM广播，还有实用的高清晰度CD直录功能（LINE-IN）现场录音功能等等！

TEA5767内置了主频高达75MHZ的数字信号处理器，实现384KBPS/48KHZ的MD级高品质MP3音乐文件回放，加上拥有一般MP3播放器难以企及的高保真回放线路（信噪比高达95DB，THD总谐波失真率〈0.05%〉同时它非常省电。

通过本实验，我们将熟悉FM收音的基本实现过程，以及TEA5767模块的使用。

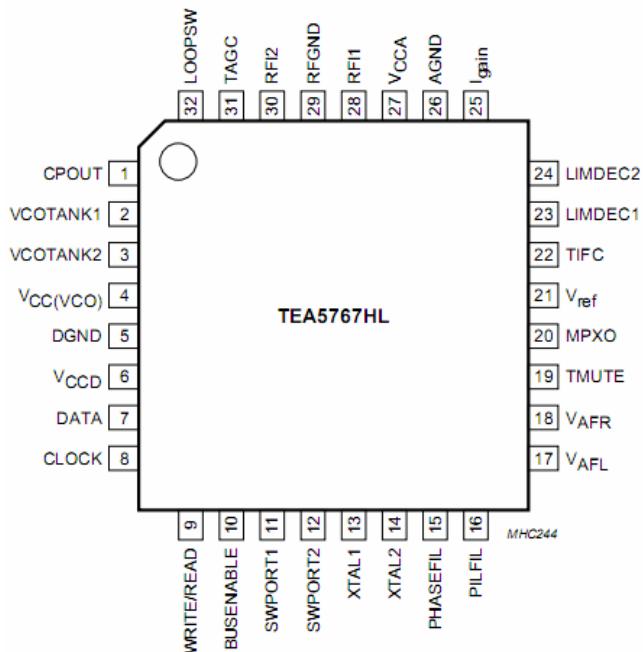
7.35.2 实验原理

在神舟III号STM32开发板中，我们使用目前主流的模块来实现收音机功能，模块的主芯片为TEA5767HL，这是一款有Philip针对手持应用场合推出的低功耗FM立体收音芯片。

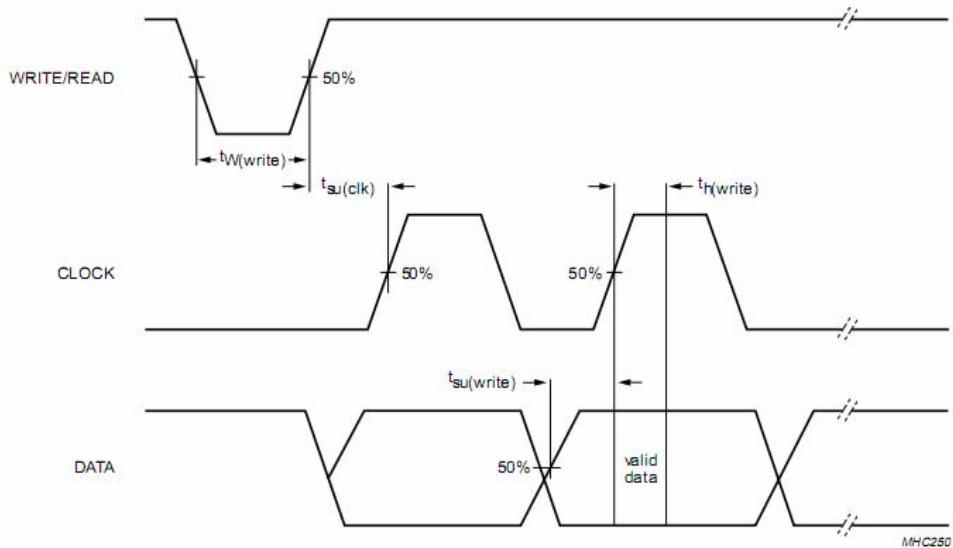
芯片具有如下特性：

- ◆ 高灵敏、低噪声高频放大器，
- ◆ 收音频率：87.6MHz~108MHz，（支持频率范围在76MHz~87.5MHz 之间的校园收音频道），
- ◆ LC 调谐振荡器使成本更低，RF AGC 电路
- ◆ 内置调频中频选择，I2C 总线控制
- ◆ 内置FM 立体声解调器，PLL 合成调谐解码器
- ◆ 两个可编程端口，软静音，SNC（立体声噪声消除）
- ◆ 自适应立体声解码，自动搜索功能
- ◆ 等待模式，需要一个7.6MHz晶体
- ◆ 40 脚LQFP 封装

芯片的管脚示意图如下：



芯片本身通过三线接口进行访问，分别是WRITE/READ，CLOCK和DATA三个管脚，WRITE/READ管脚上的一个上升沿表示使能数据写入如TEA5767，对应的时序示意图如下：



在进行写操作时，都是5个字节连续写入的，因此，我们在访问TEA5767模块时，应确保是5个字节连续写入的，各个字节的含义如下：

第一个字节

Format of 1st data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| MUTE | SM | PLL13 | PLL12 | PLL11 | PLL10 | PLL9 | PLL8 |

Description of 1st data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|-----------|--|
| 7 | MUTE | if MUTE = 1 then L and R audio are muted; if MUTE = 0 then L and R audio are not muted |
| 6 | SM | Search Mode: if SM = 1 then in search mode; if SM = 0 then not in search mode |
| 5 to 0 | PLL[13:8] | setting of synthesizer programmable counter for search or preset |

第一个字节可以实现静音控制，控制搜台模式以及PLL的[13:8]这几位。

第二个字节

Format of 2nd data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| PLL7 | PLL6 | PLL5 | PLL4 | PLL3 | PLL2 | PLL1 | PLL0 |

Description of 2nd data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|----------|--|
| 7 to 0 | PLL[7:0] | setting of synthesizer programmable counter for search or preset |

第二个字节，设置PLL的[7:0]位。

第三个字节

Format of 3rd data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| SUD | SSL1 | SSL0 | HLSI | MS | ML | MR | SWP1 |

Description of 3rd data byte bits

| BIT | SYMBOL | DESCRIPTION |
|---------|----------|---|
| 7 | SUD | Search Up/Down: if SUD = 1 then search up; if SUD = 0 then search down |
| 6 and 5 | SSL[1:0] | Search Stop Level: see Table 8 |
| 4 | HLSI | HIGH/LOW Side Injection: if HLSI = 1 then HIGH side LO injection; if HLSI = 0 then LOW side LO injection |
| 3 | MS | Mono to Stereo: if MS = 1 then forced mono; if MS = 0 then stereo ON |
| 2 | ML | Mute Left: if ML = 1 then the left audio channel is muted and forced mono; if ML = 0 then the left audio channel is not muted |
| 1 | MR | Mute Right: if MR = 1 then the right audio channel is muted and forced mono; if MR = 0 then the right audio channel is not muted |
| 0 | SWP1 | Software programmable port 1: if SWP1 = 1 then port 1 is HIGH; if SWP1 = 0 then port 1 is LOW |

Table 8 Search stop level setting

| SSL1 | SSL0 | SEARCH STOP LEVEL |
|------|------|-----------------------------|
| 0 | 0 | not allowed in search mode |
| 0 | 1 | low; level ADC output = 5 |
| 1 | 0 | mid; level ADC output = 7 |
| 1 | 1 | high; level ADC output = 10 |

第四个字节

Format of 4th data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| SWP2 | STBY | BL | XTAL | SMUTE | HCC | SNC | SI |

Description of 4th data byte bits

| BIT | SYMBOL | DESCRIPTION |
|-----|--------|--|
| 7 | SWP2 | Software programmable port 2: if SWP2 = 1 then port 2 is HIGH; if SWP2 = 0 then port 2 is LOW |
| 6 | STBY | Standby: if STBY = 1 then in standby mode; if STBY = 0 then not in standby mode |
| 5 | BL | Band Limits: if BL = 1 then Japanese FM band; if BL = 0 then US/Europe FM band |
| 4 | XTAL | if XTAL = 1 then $f_{xtal} = 32.768 \text{ kHz}$; if XTAL = 0 then $f_{xtal} = 13 \text{ MHz}$ |
| 3 | SMUTE | Soft MUTE: if SMUTE = 1 then soft mute is ON; if SMUTE = 0 then soft mute is OFF |
| 2 | HCC | High Cut Control: if HCC = 1 then high cut control is ON; if HCC = 0 then high cut control is OFF |
| 1 | SNC | Stereo Noise Cancelling: if SNC = 1 then stereo noise cancelling is ON; if SNC = 0 then stereo noise cancelling is OFF |
| 0 | SI | Search Indicator: if SI = 1 then pin SWPORT1 is output for the ready flag; if SI = 0 then pin SWPORT1 is software programmable port 1 |

第五个字节

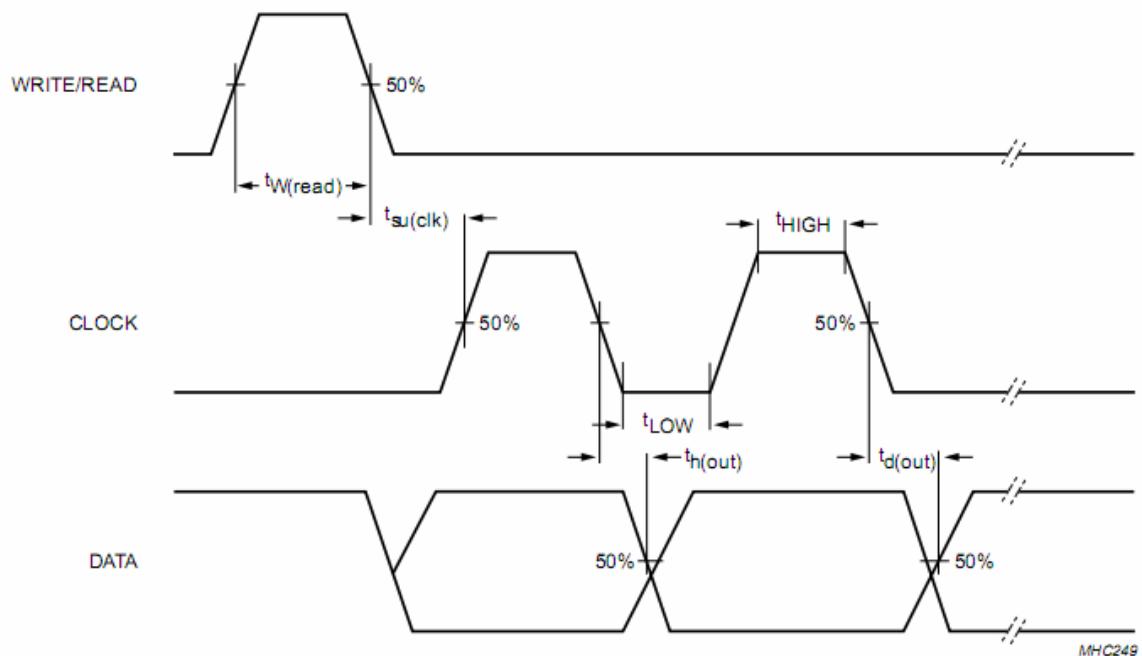
Format of 5th data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| PLLREF | DTC | - | - | - | - | - | - |

Description of 5th data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|--------|---|
| 7 | PLLREF | if PLLREF = 1 then the 6.5 MHz reference frequency for the PLL is enabled; if PLLREF = 0 then the 6.5 MHz reference frequency for the PLL is disabled |
| 6 | DTC | if DTC = 1 then the de-emphasis time constant is 75 μ s; if DTC = 0 then the de-emphasis time constant is 50 μ s |
| 5 to 0 | - | not used; position is don't care |

相反的，WRITE/READ管脚上的一个下降沿表示从TEA5767读出数据，对应的时序示意图如下：



在进行读操作时，都是5个字节连续读的，5个字节的含义如下：

第一个字节

Format of 1st data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| RF | BLF | PLL13 | PLL12 | PLL11 | PLL10 | PLL9 | PLL8 |

Description of 1st data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|-----------|---|
| 7 | RF | Ready Flag: if RF = 1 then a station has been found or the band limit has been reached; if RF = 0 then no station has been found |
| 6 | BLF | Band Limit Flag: if BLF = 1 then the band limit has been reached; if BLF = 0 then the band limit has not been reached |
| 5 to 0 | PLL[13:8] | setting of synthesizer programmable counter after search or preset |

第二个字节

Format of 2nd data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| PLL7 | PLL6 | PLL5 | PLL4 | PLL3 | PLL2 | PLL1 | PLL0 |

Description of 2nd data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|----------|--|
| 7 to 0 | PLL[7:0] | setting of synthesizer programmable counter after search or preset |

第三个字节

Format of 3rd data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| STEREO | IF6 | IF5 | IF4 | IF3 | IF2 | IF1 | IF0 |

Description of 3rd data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|-----------|--|
| 7 | STEREO | Stereo indication: if STEREO = 1 then stereo reception; if STEREO = 0 then mono reception |
| 6 to 0 | PLL[13:8] | IF counter result |

Table 8 Search stop level setting

| SSL1 | SSL0 | SEARCH STOP LEVEL |
|------|------|-----------------------------|
| 0 | 0 | not allowed in search mode |
| 0 | 1 | low; level ADC output = 5 |
| 1 | 0 | mid; level ADC output = 7 |
| 1 | 1 | high; level ADC output = 10 |

第四个字节

Format of 4th data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| LEV3 | LEV2 | LEV1 | LEV0 | CI3 | CI2 | CI1 | 0 |

Description of 4th data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|----------|--|
| 7 to 4 | LEV[3:0] | level ADC output |
| 3 to 1 | CI[3:1] | Chip Identification: these bits have to be set to logic 0 |
| 0 | - | this bit is internally set to logic 0 |

第五个字节

Format of 5th data byte

| BIT 7 (MSB) | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 (LSB) |
|-------------|-------|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

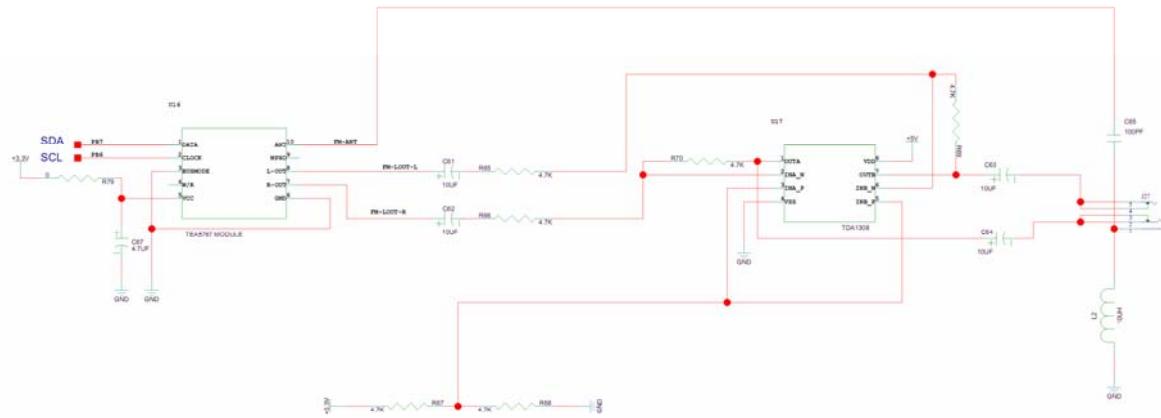
Description of 5th data byte bits

| BIT | SYMBOL | DESCRIPTION |
|--------|--------|--|
| 7 to 0 | - | reserved for future extensions; these bits are internally set to logic 0 |

神舟III号采用的是现成的TEA5767收音机模块，在访问时，我们只需要标准的I2C接口即可对他进行访问。

7.35.3 硬件设计

在神舟III号开发板中，STM32处理器通过PB6,PB7访问TEA5767模块，PB6,PB7为STM32本身的I2C总线GPIO管脚，我们可以通过I2C接口访问TEA5767模块，在本例程中，是通过PB6,PB7软件模式I2C接口对模块进行的访问。



在上图中，收音机模块的ANT管脚通过电容与音频座相连，这样当耳机插入到音频座以后，就可以作为收音的天线使用。

TEA5767模块外形图如下。



7.35.4 软件设计

在本实验中，我们需要通过串口1控制收音模块的配置，包括串口1的输入输出，在本实验中，我们采用的是串口中断模式，因此，在NVIC_Configuration()函数中完成。

```
void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Configure the NVIC Preemption Priority Bits */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);

    /* Enable the USART1 Interrupt */
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

}

```

◆ GPIO 初始化

在本实验中，使用到的硬件资源有I2C1接口有LED灯以及串口以及I2C1接口用于收音机模块的初始化，在使用之前，我们需要对相关的GPIO初始化，对应的函数主要是GPIO_Configuration()，具体代码如下：

```
void GPIO_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd( RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                           RCC_APB2Periph_GPIOC | RCC_APB2Periph_GPIOD |
                           RCC_APB2Periph_GPIOE|RCC_APB2Periph_GPIOF|RCC_APB2Periph_AFIO, ENABLE);

    /* I2C1 管脚 初始化 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /*LED灯初始化*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOF, &GPIO_InitStructure);

    /*串口管脚初始化*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;           //USART1 TX
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       //复用推挽输出
    GPIO_Init(GPIOA, &GPIO_InitStructure);             //A端口

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;           //USART1 RX
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //复用开漏输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);             //A端口
}
? end GPIO_Configuration ?
```

在上面的代码中，将I2C1接口使用的GPIO管脚PB6,PB7初始化为开漏输出模式；将LED灯使用的GPIO管脚PF6~9初始化为推挽输出模式；将串口USART1_TX初始化为复用推挽输出模式，USART1_RX初始化为复用开漏输入模式。关于GPIO使用的更多详细介绍请参见《4.1 流水灯实验》一节介绍。

串口1初始化

在本实验中，串口1用于收音机实验的提示信息输出以及输入设置，在设置串口波特率以及串口参数后，我们选择串口1使用中断模式接收键盘数据，相关代码如下。

```
/*串口初始化*/
/* USARTx configured as follow:
   - BaudRate = 115200 baud
   - Word Length = 8 Bits
   - One Stop Bit
   - No parity
   - Hardware flow control disabled (RTS and CTS signals)
   - Receive and transmit enabled
*/
USART_InitStructure.USART_BaudRate = 115200;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

/* Configure USART1 */
USART_Init(USART1, &USART_InitStructure);

/* Enable USART1 Receive and Transmit interrupts */
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
//USART_ITConfig(USART1, USART_IT_TXE, ENABLE);

/* Enable the USART1 */
USART_Cmd(USART1, ENABLE);
```

当我们按电脑键盘上按键时，程序将检测到串口接收到数据，进入对应的串口接收中断服务程序 USART1_IRQHandler (void)，判断用户输入的按键值，并设置对应的标志 rec_f，以便退出中断服务程序后，主程序依据标志执行对于那个的操作。中断服务程序具体代码如下：

```
void USART1_IRQHandler(void)
{
    unsigned int i;

    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) //如果是串口1接收中断
    {
        RxBuffer1[RxCounter1++] = USART_ReceiveData(USART1); //从接收数据寄存器读取一个字节的数据
        //如果输入的字符为's'或's'，那么设置rec_f为2，同时将串口数据保存到BUFFER中
        if(RxBuffer1[RxCounter1-1] == 'S' || RxBuffer1[RxCounter1-1] == 's')
        {
            rec_f=2;
            for(i=0; i< RxCounter1; i++) TxBuffer1[i] = RxBuffer1[i];
            RxCounter1=0;
        }
        //如果输入的字符为'D'或'd'，那么设置rec_f为3
        else if(RxBuffer1[RxCounter1-1] == 'D' || RxBuffer1[RxCounter1-1] == 'd')
        {
            rec_f=3;
        }
        //如果输入的字符为'M'或'm'，那么设置rec_f为1，同时将串口数据保存到BUFFER中
        else if(RxBuffer1[RxCounter1-1] == 'M' || RxBuffer1[RxCounter1-1] == 'm')
        {
            for(i=0; i< RxCounter1; i++) TxBuffer1[i] = RxBuffer1[i];
            len=RxCounter1;
            rec_f=1;
            RxCounter1=0;
        }
        //如果输入的字符为'P'或'p'，那么设置rec_f为4，同时将串口数据保存到BUFFER中
        else if(RxBuffer1[RxCounter1-1] == 'P' || RxBuffer1[RxCounter1-1] == 'p')
        {
            for(i=0; i< RxCounter1; i++) TxBuffer1[i] = RxBuffer1[i];
            len=RxCounter1;
            rec_f=4;
            RxCounter1=0;
        }
        //如果输入的字符为'H'或'h'，那么设置rec_f为5，显示帮助菜单
        else if(RxBuffer1[RxCounter1-1] == 'H' || RxBuffer1[RxCounter1-1] == 'h')
        {
            rec_f=5;
        }
    } // end if USART_GetITStatus(USA...
}
```

前面介绍了串口，LED灯等通用外围资源的设置，在接下来我们将进行TEA5767的访问，首先看一下TEA5767的模式设置。

◆ TEA5767 模式设置

```
uint8_t Tx_Buffer[] = { 0XF0, 0X2C, 0XD0, 0X12, 0X40 };

//设置TEA5767工作模式
if( I2C_Write(Tx_Buffer, TEA5767_WRITEADDR, 5) == FALSE )
{
    //如果设置失败，则点亮LED灯DS1，并串口答应提示信息
    GPIO_ResetBits(GPIO_LED, DS1_PIN);
    printf("\n\rTEA5767收音模块模式设置失败，请检查硬件连接");
}
```

其中Tx_Buffer的值为TEA5767的模式设置，主要是设置TEA5767静音，处于搜台模式，具体TxBuffer的参数含义请查看TEA5767数据手册，或者查看43.17.2实验原理的相关描述。

◆ TEA5767 搜台 PLL 设置

TEA5767的搜台PLL设置主要是在SetPLL函数实现的，程序语句FM_FREQ设置的频率计算PLL值，并依据串口接收中断服务程序中的设置的rec_f状态标志变量的值，控制收音模块启动搜台或者只是设置搜台的频率而不改变TEA5767的工作模式，具体代码如下：

```
void SetPLL(void)
{
    //计算 PLL 值
    FM_PLL=(unsigned long) ((4000*(FM_FREQ/1000+225))/32768);
    if(rec_f==2)
    {
        //PLL高字节值
        PLL_HIGH=(unsigned char) (((FM_PLL >> 8)&0X3f)|0xc0); //启动搜台
    }
    else
    {
        //PLL高字节值
        PLL_HIGH=(unsigned char) (((FM_PLL >> 8)&0X3f)); //不更改搜台控制及静音控制位
    }
    //首次运行设置为静音
    if(Mute_Flag == 1)
    {
        PLL_HIGH = PLL_HIGH | 0x80;
        Mute_Flag = 0;
    }
    Tx_Buffer[0]=PLL_HIGH; //写入第一字节值
    PLL_LOW=(unsigned char) FM_PLL; //PLL低字节值
    Tx_Buffer[1] = PLL_LOW; //写入第二字节值

    I2C_Write(Tx_Buffer, TEA5767_WRITEADDR, 5);
} ? end SetPLL ?
```

◆ 主程序

在主程序中，只要检测到电脑上键盘有输入按键，就将依据中断服务程序中设置的状态，执行相应的程序。

```
while (1)
{
    if(rec_f!=0)
    {
        //如果是直接输入收音频率
        if(rec_f==1)
        {
            //直接输入频率
            a=atof(TxBuffer1);
            FM_FREQ=a*1000000;
            SetPLL();
            printf("\n\r 当前FM频率是: %g\n MHz ", a);

        }
        //如果是自动搜台
        else if(rec_f==2)
        {
            printf("\n\r 搜索FM节目！");
            Tx_Buffer[0] = 0xFE; //自动搜台
            FM_FREQ=87500000; //自动搜台起始频率
            FM_FREQ=FM_FREQ+100000;
            SetPLL(); //启动自动搜台
            Delay(0xffff); //延时等待
            ch2=0;
            while(1)
            {
                fm_pub:
                FM_FREQ=FM_FREQ+100000;
                if(FM_FREQ>108000000 || rec_f != 2) //如果到达最大搜台频率,或者用户按了其他按键,退出搜台
                {
                    printf("\n\r 退出搜台程序");
                    FM_FREQ=98800000;
                    break;
                }

                SetPLL(); //设置搜台PLL
                Delay(0xffff);
                I2C_ReadByte(Rx_Buffer, 5, TEA5767_READADDR);
                a=FM_FREQ;
                a=a/1000000;

                if((Rx_Buffer[0]&0x3f) !=(Tx_Buffer[0]&0x3f) || (Rx_Buffer[1] !=Tx_Buffer[1]) || (Rx_Buffer[1]&0x80=0x80)
                   || Rx_Buffer[2]<50 || Rx_Buffer[2]>=56 || (Rx_Buffer[3]>>4)<7 || (Rx_Buffer[3]>>4)>14)
                {
                    printf("\r\n当前FM频率: %g MHz ", a);
                }
                else
                {
                    printf("\r\n当前FM频率: %g MHz 有信号! %u %u", a, Rx_Buffer[2], Rx_Buffer[3]>>4);
                    fm_ch[ch2]= FM_FREQ;
                    ch2++;
                    goto fm_pub;
                }
            } ? end while 1 ?
            if(FM_FREQ!=98800000) goto fm_pub;
            printf("\n\r 共有%u 个有效的FM频率: ", ch2);
            ch1=ch2;
            while(ch1--)
            {
                a=fm_ch[ch1];
                a=a/1000000;
                printf("\n\r -- %u FM频率: %g MHz \n", ch1, a);
            }
        } ? end if rec_f==2 ?
        //显示有效频率
        else if(rec_f==3){
            printf("\n\r 共有%u 个有效的FM频率: ", ch2);
            ch1=ch2;
            while(ch1--)
            {
                a=fm_ch[ch1];
                a=a/1000000;
                printf("\n\r -- %u FM频率: %g MHz \n", ch1, a);
            }
        }
    }
}
```

```
// 直接设置收音频率
else if(rec_f==4)
{
    ch1=atoi(TxBuffer1);
    FM_FREQ=fm_ch[ch1];
    a=fm_ch[ch1];
    a=a/1000000;
    printf ("\r\n 当前FM频率是: %g\n MHz \n", a);
}
// 显示提示信息
else if(rec_f==5)
{
    printf ("\n\r-----");
    printf ("\n\rWWW.ARMJISHU.COM ");
    printf ("\n\r神舟III号 STM32开发板TEA5767收音模块程序");
    printf ("\n\r");
    printf ("\n\r H(h)      ---帮助");
    printf ("\n\r S(s)      ---搜索节目 ");
    printf ("\n\r D(d)      ---显示有效节目");
    printf ("\n\r xxP(xxP)  ---播放选定的节目(如10P) ");
    printf ("\n\r xx.xM(xx.xm)---直接选定频率(如100.8M) ");
    printf ("\n\r");
    printf ("\n\rWWW.ARMJISHU.COM ");
    printf ("\n\r-----");
}

rec_f=0;
SetPLL();
} ? end if rec_f!=0 ?
} ? end while 1 ?
```

7.35.5 下载与测试

在 [神舟III号光盘编译好的固件17. 收音机试验](#)目录下的收音机实验.hex文件即为本节我们分析的实验所编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟III号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

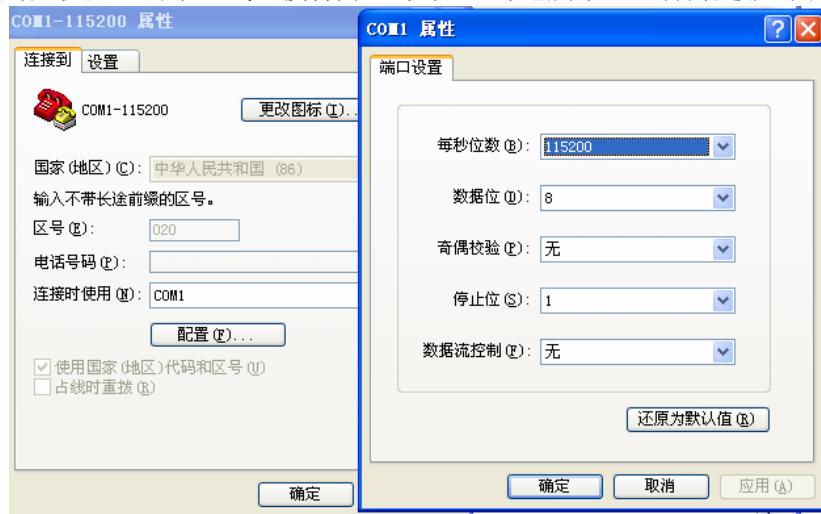
如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.35.6 实验现象

将程序下载到神舟III号后，按如下连接好硬件：

1. 将耳机插入到神舟III号的J27（收音机音频接口）中；
2. 用随板配置的串口线连接神舟III号串口1与电脑串口，打开超级终端，按如下参数配置串口。



上电运行，串口1将打印如下信息：

www.ARMJISHU.COM
神舟III号 STM32 开发板TEA5767收音模块程序
H(h) ---帮助
S(s) ---搜索节目
D(d) ---显示有效节目
xxP(xx) ---播放选定的节目(如10P)
xx.XM(xx.XM)---直接选定频率(如100.8M)
www.ARMJISHU.COM

按照串口提示的信息，即可执行对应的操作，如按键盘上的S，可以开始搜台，1P，可以选择第一个频道（注意：播放选定频道之前需要先搜台）。以下为一些操作效果图

```
当前FM频率: 107.1 MHZ
当前FM频率: 107.2 MHZ
当前FM频率: 107.3 MHZ
当前FM频率: 107.4 MHZ
当前FM频率: 107.5 MHZ
当前FM频率: 107.6 MHZ
当前FM频率: 107.7 MHZ 有信号!      55  13
当前FM频率: 107.8 MHZ
当前FM频率: 107.9 MHZ
当前FM频率: 108 MHZ
退出搜台程序
共有9个有效的FM频率:
-- 8 FM频率: 107.7 MHZ
-- 7 FM频率: 106.6 MHZ
-- 6 FM频率: 103.6 MHZ
-- 5 FM频率: 102.7 MHZ
-- 4 FM频率: 100 MHZ      按S(s)自动搜台,
-- 3 FM频率: 96.2 MHZ    左边为自动搜台结果
-- 2 FM频率: 91.4 MHZ
-- 1 FM频率: 89.3 MHZ
-- 0 FM频率: 88.5 MHZ
请输入xxP(xxP) 播放选定的节目
当前FM频率是: 89.3 MHZ  输入“1P”选择第1频道
当前FM频率是: 91.4 MHZ ■ 输入“91.4M”选择91.4MHz频率
```

7.36 TFT彩色液晶屏只显示红色

本实验向大家介绍如何使用 STM32 驱动 LCD 屏，及使用触摸屏控制器检测触点坐标

7.36.1 术语解释

1) 引言

将普通图像的信息数字化后，即得到数字图像，最后转换为合适计算机处理的数字。LCD 控制器则被用于将数字图象传输到 LCD 显示屏进行显示。所以，如果需要理解 LCD 控制器的工作原理，有必要了解一些图象基础知识

2) 帧

显示屏所显示的一幅完整画面就是一个帧。

3) 象素

“象素”是由图像和元素这两个单词的字母所组成的，是构成数字图象的最小单位。我们若把数字图象放大数倍，就会发现数字图象其实是由许多色彩相近的小方点所组成，这些小方点就是“象素”。

4) 分辨率

分辨率有很多种，例如打印分辨率、影像分辨率等。此处仅仅就 LCD 显示器的分辨率进行阐释。分辨率是指显示器所能显示点数的多少，可以把整个显示器想象成是一个棋盘，而分辨率就是棋盘经线和纬线交叉点的数目。由于屏幕上的点、线和面都是由点组成的，所以显示器可以显示的点数越多，显示画面就越精细。

对于 LCD 显示器来说，象素的数目和分辨率在数值上相等，都等于屏幕上横向点个数和纵向点个数的乘积。

比如 STM32 神舟系列开发板所配的 2.8 寸或者 3.2 寸液晶屏的分辨率是 240*320 的，有这么多个点。

7.36.2 液晶彩屏原理简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，因而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

因为STM32内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路(液晶屏和触摸屏的驱动电路是独立的)，STM32芯片通过驱动芯片来控制液晶屏和触摸屏。以神舟III号开发板3.2寸液晶屏(240*320)为例，它使用ILI9320或者SSD1289芯片控制液晶屏，通过XPT2046芯片控制触摸屏。

液晶屏的控制芯片内部结构相对比较复杂，液晶屏的彩屏驱动电路最主要的是位于中间GRAM(Graphics RAM)，可以理解为显存。GRAM就好比是一个彩屏数据缓冲buffer，我们可以把大批的显示内容以显示矩阵的形式写到buffer里，让彩屏LCD来读取buffer里的数据再由彩屏驱动芯片显示到显示屏上，随着GRAM逐渐丰富和完善，除了显示矩阵以外还放很多的命令，GRAM中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把GRAM存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。到现在，这些驱动/控制电路以及buffer都合起来放在一片芯片中，统一被称为driver IC，这个driver IC就是上一段我们所说的ILI9320或者SSD1289的driver IC芯片。

7.36.3 液晶彩屏图像像素分析

TFT就是“Thin Film Transistor”的简称，一般代指薄膜液晶显示器，而实际上指的是薄膜晶体管(矩阵)——可以“主动的”对屏幕上的各个独立的像素进行控制。对于图象产生的基本原理为：显示屏由许多可以发出任意颜色的光线的像素组成，主要控制各个像素显示相应的颜色就可以达到目的。在TFT LCD中一般会采用背光技术，为了能精确的控制每一个像素的颜色和亮度就需要在每一个想色之后安装一个类似百叶窗的开关，当“百叶窗”打开时光线就可以透射过来，而“百叶窗”关上之后，光线就无法透射。

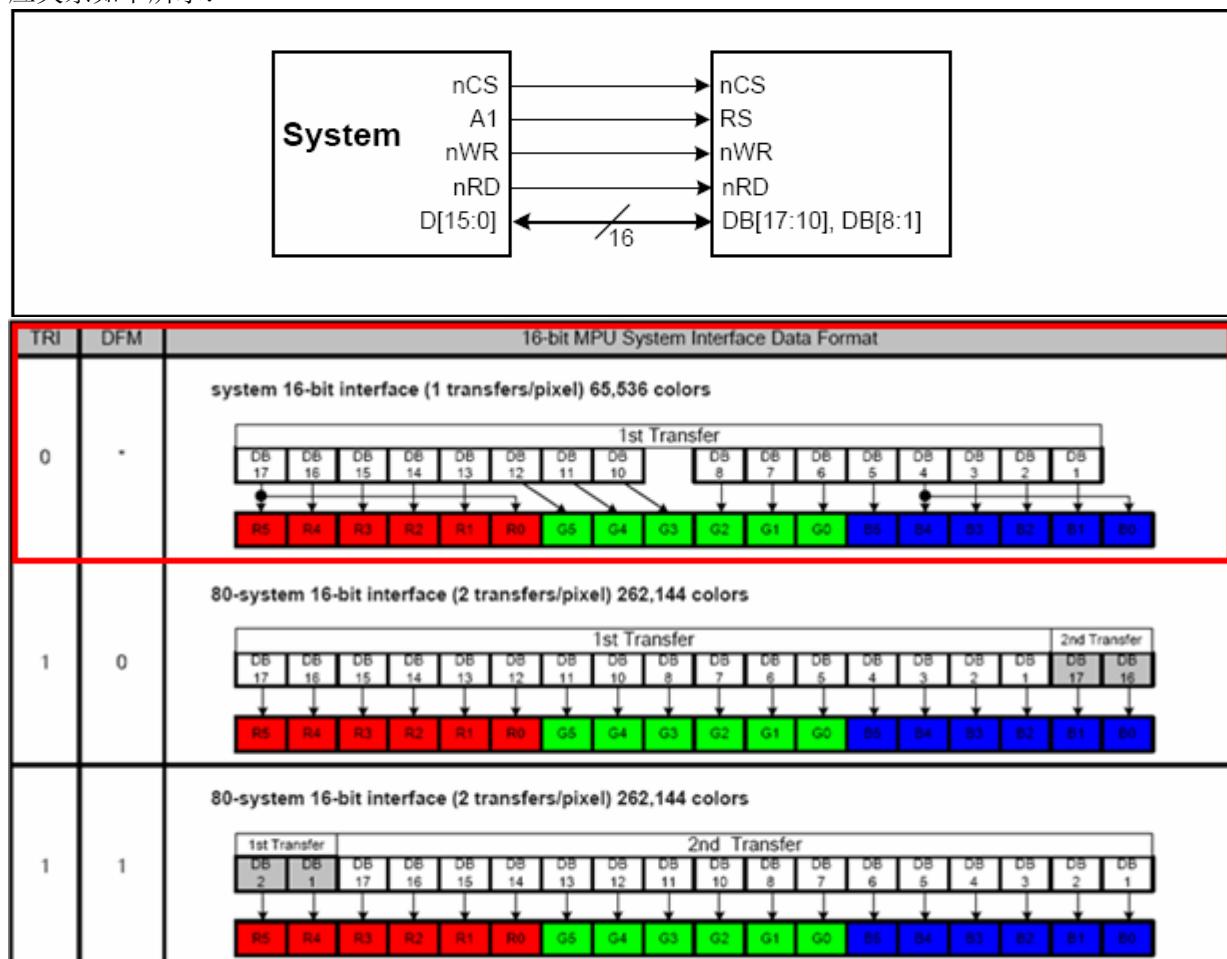
图像数据的像素点由红(R)、绿(G)、蓝(B)三原色组成，三原色根据其深浅程度被分为0~255个级别，它们按不同比例的混合可以得出各种色彩。如R: 255, G255, B255混合后为白色。根据描述像素点数据的长度，主要分为8、16、24及32位。如以8位来描述的像素点可表示 $2^8=256$ 色，16位描述的为 $2^{16}=65536$ 色，称为真彩色，也称为64K色。实际上受人眼对颜色的识别能力的限制，16位色与12位色已经难以分辨了。

神舟王III号开发板上配带的TFT LCD屏，LCD屏为320x240分辨率，ILI9320液晶控制器液晶控制器自带显存，其总大小为自带显存，其总大小为172800（240*320*18/8）字节。

STM32神舟王III号开发板支持2.8/3.2寸的ILI9320或SSD1289的TFT LCD，在本例程中，我们以ILI9320控制器进行简单的介绍。

ILI9320采用的是16位控制模式，以16位描述的像素点。按照标准格式，16位的像素点的三原色描述的位数为R: G: B = 5: 6: 5，描述绿色的位数较多是因为人眼对绿色更为敏感。

ILI9320最高能够控制18位的LCD，但为了数据传输简便，我们采用它的16位控制模式，以16位描述的像素点。按照标准格式，16位的像素点的三原色描述的位数为R: G: B = 5: 6: 5，描述绿色的位数较多是因为人眼对绿色更为敏感。ILI9320控制模块的16位数据与显存间的对应关系如下所示：



我们且看第一种配置TRI为0，DFM任意时的图形，低5位为蓝色B，中间6位为绿色G，最高5位为红色R；图中的是默认16条数据线时，像素点三原色的分配状况，DB1~DB5为5根线为蓝色，DB6~DB8以及DB10~DB12六根线为绿色，DB13~DB17这五根线为红色。这样分配有D0和D9位是无效的，使得刚好使用完整的16位。

RGB比例为5: 6: 5是一个十分通用的颜色标准， $5+6+5=16$ 位，表示5根线表示红色，6根线表示蓝色，5根线表示蓝色；如黑色的编码为0x0000，白色的编码为0xffff，红色为0xf800。比如红色0xf800，化成RGB5: 6: 5的二进制就是11111 000000 00000，刚好可以看到R是红色的5位数据值是11111，G是绿色的值是000000，B是蓝色00000，因为红色寄存器内容都是1，而其他两种颜色都为0，所以最终会显示出红色。

7.36.4 控制器命令分析

我们这里用IL9320的命令做分析，其他像SSD1289，IL9341，IL9325等都是同样的道理；下面我们了解几个重要的指令描述，如下图红框中内容。

| No. | Registers Name | R/W RS | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
|-----|-----------------------------------|--------|--|------|--------|-------|-----|-------|-------|------|-------|------|---------|---------|---------|------|------|----------|-------|
| IR | Index Register | W 0 | - | - | - | - | - | - | - | - | ID7 | ID6 | ID5 | ID4 | ID3 | ID2 | ID1 | ID0 | |
| SR | Status Read | R 0 | L7 | L6 | L5 | L4 | L3 | L2 | L1 | L0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 00h | Driver Code Read | R 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | |
| 00h | Start Oscillation | W 1 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | OSC | 打开OSC | |
| 01h | Driver Output Control 1 | W 1 | 0 | 0 | 0 | 0 | 0 | SM | 0 | SS | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 02h | LCD Driving Control | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | BC0 | EOB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 03h | Entry Mode | W 1 | TRI | DFM | 0 | BGR | 0 | DACKE | HWM | 0 | 0 | 0 | ID1 | ID0 | AM | 0 | 0 | 0 | |
| 04h | Resize Control | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | RCV1 | RCV0 | 0 | 0 | 0 | RCH1 | RCH0 | 0 | 0 | RS21 | RS20 |
| 07h | Display Control 1 | W 1 | 0 | 0 | PTDE1 | PTDE0 | 0 | 0 | BASEE | 0 | 0 | 0 | GON | DTE | CL | 0 | D1 | D0 | |
| 08h | Display Control 2 | W 1 | 0 | 0 | 0 | 0 | FP3 | FP2 | FP1 | FP0 | 0 | 0 | 0 | 0 | BP3 | BP2 | BP1 | BP0 | |
| 09h | Display Control 3 | W 1 | 0 | 0 | 0 | 0 | 0 | PTS2 | PTS1 | PTS0 | 0 | 0 | PTG1 | PTG0 | ISC3 | ISC2 | ISC1 | ISC0 | |
| 0Ah | Display Control 4 | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | FMARKOE | FM12 | FM11 | FM10 | |
| 0Ch | RGB Display Interface Control 1 | W 1 | ENC2 | ENC1 | ENC0 | 0 | 0 | 0 | 0 | RM | 0 | 0 | DM1 | DM0 | 0 | 0 | RIM1 | RIM0 | |
| 0Dh | Frame Maker Position | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | FMP8 | FMP7 | FMP6 | FMP5 | FMP4 | FMP3 | FMP2 | FMP1 | FMP0 | |
| 0Fh | RGB Display Interface Control 2 | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | VSP1 | HSPL | 0 | DPL | EPL | |
| 20h | Horizontal GRAM Address Set | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AD7 | AD6 | AD5 | AD4 | AD3 | AD2 | AD1 | AD0 | 行地址设置 | |
| 21h | Vertical GRAM Address Set | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AD16 | AD15 | AD14 | AD13 | AD12 | AD11 | AD10 | AD9 | 列地址设置 | |
| 22h | Write Data to GRAM | W 1 | RAM write data (WD17-0) / read data (RD17-0) bits are transferred via different data bus lines according to the selected interfaces. | | | | | | | | | | | | | | | 写数据到GRAM | |
| 29h | Power Control 7 | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | VCM4 | VCM3 | VCM2 | VCM1 | VCM0 | |
| 2Bh | Frame Rate and Color Control | W 1 | 16M | EN | Dither | 0 | 0 | 0 | 0 | 0 | EXT_R | 0 | FR_SEL1 | FR_SEL0 | 0 | 0 | 0 | 0 | 行起始地址 |
| 50h | Horizontal Address Start Position | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | HSA7 | HSA6 | HSA5 | HSA4 | HSA3 | HSA2 | HSA1 | HSA0 | |
| 51h | Horizontal Address End Position | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | HEA7 | HEA6 | HEA5 | HEA4 | HEA3 | HEA2 | HEA1 | HEA0 | |
| 52h | Vertical Address Start Position | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | VSA8 | VSA7 | VSA6 | VSA5 | VSA4 | VSA3 | VSA2 | VSA1 | VSA0 | |
| 53h | Vertical Address End Position | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | VEA8 | VEA7 | VEA6 | VEA5 | VEA4 | VEA3 | VEA2 | VEA1 | VEA0 | |

8.2. Instruction Descriptions

| No. | Registers Name | R/W RS | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-----|-------------------------|--------|-----|-----|-----|-----|-----|-----|----|----|-----|-----|-----|-----|-----|-----|-----|-------|
| IR | Index Register | W 0 | - | - | - | - | - | - | - | - | ID7 | ID6 | ID5 | ID4 | ID3 | ID2 | ID1 | ID0 |
| SR | Status Read | R 0 | L7 | L6 | L5 | L4 | L3 | L2 | L1 | L0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 00h | Driver Code Read | R 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 00h | Start Oscillation | W 1 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | OSC | 打开OSC |
| 01h | Driver Output Control 1 | W 1 | 0 | 0 | 0 | 0 | 0 | SM | 0 | SS | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

1) 00h指令，当为读操作时，读取控制器的型号；当为写操作时，打开/关闭OSC振荡器，当写操作设置OSC比特位为1时，开启内部振荡器；为0时，停止振荡器。然后至少等待10ms时钟稳定后，再继续其它功能的设置。我们在代码设计中就是通过指令00h读取LCD控制器的型号，从而针对具体型号的控制器进行初始化操作，以便于兼容各种不同系列的LCD控制器。

2) 03h指令，入口模式命令。

| | | | | | | | | | | | | | | | | | | |
|-----|-------------------|-----|-----|-----|-------|-------|---|-------|-------|------|---|---|------|------|----|---|------|------|
| 03h | Entry Mode | W 1 | TRI | DFM | 0 | BGR | 0 | DACKE | HWM | 0 | 0 | 0 | I/D1 | I/D0 | AM | 0 | 0 | 0 |
| 04h | Resize Control | W 1 | 0 | 0 | 0 | 0 | 0 | 0 | RCV1 | RCV0 | 0 | 0 | RCH1 | RCH0 | 0 | 0 | RSZ1 | RSZ0 |
| 07h | Display Control 1 | W 1 | 0 | 0 | PTDE1 | PTDE0 | 0 | 0 | BASEE | 0 | 0 | 0 | GON | DTE | CL | 0 | D1 | D0 |

我们重点关注的是I/D0、I/D1、AM这3个位，因为这3个位控制了屏幕的显示方向。AM控制GRAM的更新方向：当AM=“0”，表示地址更新方向为垂直方向；当AM=“1”，表示地址更新方向为水平方向；I/D[1:0]：当更新一个显示数据时，控制地址计数器自动增1和减1。详细内容如下所示：

| | I/D[1:0] = 00 Horizontal : decrement Vertical : decrement | I/D[1:0] = 01 Horizontal : increment Vertical : decrement | I/D[1:0] = 10 Horizontal : decrement Vertical : increment | I/D[1:0] = 11 Horizontal : increment Vertical : increment |
|----------------------|---|---|---|---|
| AM = 0 Horizontal | | | | |
| | | | | |

图：GRAM显示方向设置图

通过这几个位的设置，我们就可以控制屏幕的显示方向了，这种方法虽然简单，但是不是很通用，比如不同的液晶，可能这里差别就比较大，不能完全通用，比如9341和9320就完全不通用，具体要参考液晶屏手册。

3) 07h，显示控制命令。该命令CL位用来控制是8位彩色，还是26万色。为0时26万色，为1时8位色。D1、D0、BASEE这三个位用来控制显示开关与否的。当全部设置为1的时候开启显示，全0是关

闭。我们一般通过该命令的设置来开启或关闭显示器，以降低功耗。

| R/W | RS | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-----|----|-----|-----|-------|-------|-----|-----|-------|----|----|----|-----|-----|----|----|----|----|
| W | 1 | 0 | 0 | PTDE1 | PTDE0 | 0 | 0 | BASEE | 0 | 0 | 0 | GON | DTE | CL | 0 | D1 | D0 |

D[1:0]: 设置为“11”时，打开显示；设置为“00”时，关闭显示；而配合BASEE比特位，可以用来设置开启或是关闭显示器时，系统是挂起还是运行状态，以此设置在挂起状态，达到降低功耗的目的。

| D1 | D0 | BASEE | Source, VCOM Output | ILI9320 internal operation |
|----|----|-------|---------------------|----------------------------|
| 0 | 0 | 0 | GND | Halt |
| 0 | 1 | 1 | GND | Operate |
| 1 | 0 | 0 | Non-lit display | Operate |
| 1 | 1 | 0 | Non-lit display | Operate |
| 1 | 1 | 1 | Base image display | Operate |

而CL比特位，当CL为1时，选择8位彩色；当CL为“0”时，选择为262144彩色：

| CL | Colors |
|----|---------|
| 0 | 262,144 |
| 1 | 8 |

4) 20h和21h指令

| | | | | | | | | | | | | | | | | | | | |
|-----|-----------------------------|---|---|---|---|---|---|---|---|---|------|------|------|------|------|------|------|-----|-----|
| 20h | Horizontal GRAM Address Set | W | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AD7 | AD6 | AD5 | AD4 | AD3 | AD2 | AD1 | AD0 | |
| 21h | Vertical GRAM Address Set | W | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | AD16 | AD15 | AD14 | AD13 | AD12 | AD11 | AD10 | AD9 | AD8 |

分别为设置GRAM的行地址（X坐标）和列地址（Y坐标）。我们通过此两个指令的设置，指定需要写入的点，然后再设置颜色，便实现在指定点写入一个颜色的；20h用于设置列地址（X坐标，0~239），21h用于设置行地址（Y坐标，0~319）。当我们要在某个指定点写入一个颜色的时候，先通过这两个命令设置到改点，然后写入颜色值就可以了。

5) 22h指令

| | | | | | | | | | | | | | | | | | |
|-----|--------------------|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 22h | Write Data to GRAM | W | 1 | RAM write data (WD17-0) / read data (RD17-0) bits are transferred via different data bus lines according to the selected interfaces. | | | | | | | | | | | | | |
|-----|--------------------|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

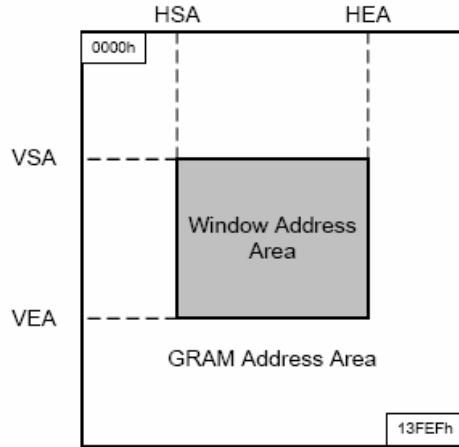
写数据到GRAM命令，当写入了这个命令之后，地址计数器才会自动的增加和减少。该命令是我们要介绍的这一组命令里面唯一的单个操作的命令，只需要写入该值就可以了，其他的都是要先写入命令编号，然后写入操作数。

6) R50~R53，行列GRAM地址位置设置。这几个命令用于设定你显示区域的大小，我们整个屏的大小为240*320，但是有时候我们只需要在其中的一部分区域写入数据，如果用先写坐标，后写数据这样的方式来实现，则速度大打折扣。此时我们就可以通过这几个命令，在其中开辟一个区域，然后不停的丢数据，这样就不需要频繁的写地址了，大大提高了刷新的速度。

| R/W | RS | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|------|----|-----|-----|-----|-----|-----|-----|----|------|------|------|------|------|------|------|------|------|
| R50h | W | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | HSA7 | HSA6 | HSA5 | HSA4 | HSA3 | HSA2 | HSA1 | HSA0 |
| R51h | W | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | HEA7 | HEA6 | HEA5 | HEA4 | HEA3 | HEA2 | HEA1 | HEA0 |
| R52h | W | 1 | 0 | 0 | 0 | 0 | 0 | 0 | VSA8 | VSA7 | VSA6 | VSA5 | VSA4 | VSA3 | VSA2 | VSA1 | VSA0 |
| R53h | W | 1 | 0 | 0 | 0 | 0 | 0 | 0 | VEA8 | VEA7 | VEA6 | VEA5 | VEA4 | VEA3 | VEA2 | VEA1 | VEA0 |

HSA[7:0]/HEA[7:0]: 指定区域的垂直方向上的起点和终点。通过设置HAS和HEA比特，以限制GRAM区域的垂直方向上的大小。

VSA[8:0]/VEA[8:0]: 指定区域的水平方向上的起点和终点。通过设置VSA和VEA比特，以显示GRAM区域的水平方向上的大小。



其中：

“00”h ≤ HAS[7:0] ≤ HEA[7:0] ≤ “EF”h
“00”h ≤ VSA[7:0] ≤ VEA[7:0] ≤ “13F”h

可以看到 IL9320 中设置这个显示区域的代码如下，这里是设置 240-*320 大小：

```
LCD_WriteReg(0x50, 0); // Set X Start.  
LCD_WriteReg(0x51, 239); // Set X End.  
LCD_WriteReg(0x52, 0); // Set Y Start.  
LCD_WriteReg(0x53, 319); // Set Y End.
```

命令部分到此，我们先简单了解到这里。有兴趣的朋友可以参阅《ILI9320控制器资料》一文，我们接下来看看要如何才能驱动STM32神舟系列的TFTLCD模块，TFTLCD显示需要的相关设置步骤如下：

1) 初始化设置STM32与彩屏模块相连接的IO管脚

这一步，先将我们与TFTLCD模块相连的IO口进行初始化，以便驱动LCD。

2) 初始化彩屏模块。

通过向彩屏模块写入一系列的设置，来启动TFTLCD的显示。为后续显示字符和数字做准备。比如：

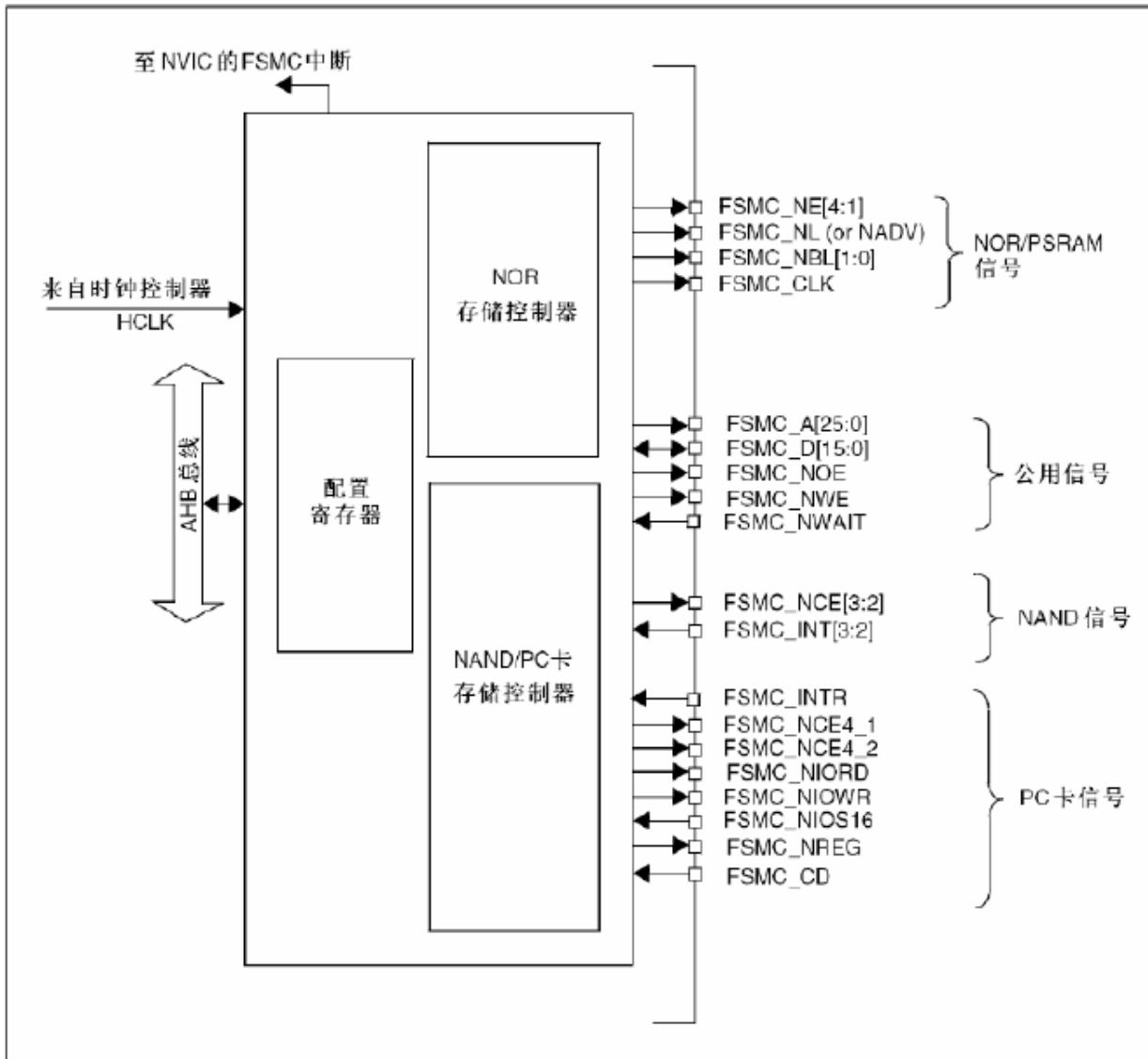
【复位TFT】→【驱动IC初始化代码】→【复位所有的寄存器】→【开启显示】→【显存清0】→【开始显示】→【显示各种图画】

3) 开始使用彩屏，通过彩屏模块的一些控制信号线来控制彩屏。

这里就是通过我们设计的程序，将要显示的字符送到彩屏模块就可以了，这些函数将在软件设计的时候介绍。通过以上三步，我们就可以使用STM32神舟系列彩屏模块来显示字符和数字了，并且可以显示各种颜色的背景。

7.36.5 FSMC介绍

STM32系列中内部集成256 KB以上FlaSh，后缀为xC、xD和xE的高存储密度微控制器特有的存储控制机制，一般芯片引脚数在100脚以上的STM32F103芯片都带有FSMC接口，FSMC能够根据不同的外部存储器类型，发出相应的数据/地址/控制信号类型以匹配信号的速度，从而使得STM32系列微控制器不仅能够应用各种不同类型、不同速度的外部静态存储器，而且能够在不增加外部器件的情况下同时扩展多种不同类型的静态存储器，满足系统设计对存储容量、产品体积以及成本的综合要求。



图：FSMC框图

从上图我们可以看出，STM32的FSMC将外部设备分为3类：NOR/PSRAM设备、NAND设备、PC卡设备。他们共用地址数据总线等信号，他们具有不同的CS以区分不同的设备。

FSMC主要的优势有以下几点：

- 1) 支持多种静态存储器类型。STM32通过FSMC可以与SRAM、ROM、PSRAM、NOR Flash和NANDFlash存储器的引脚直接相连。
- 2) 支持丰富的存储操作方法。FSMC不仅支持多种数据宽度的异步读/写操作，而且支持对NOR/PSRAM/NAND存储器的同步突发访问方式。
- 3) 支持同时扩展多种存储器。FSMC的映射地址空间中，不同的BANK是独立的，可用于扩展不同类型的存储器。当系统中扩展和使用多个外部存储器时，FSMC会通过总线悬空延迟时间参数的设置，防止各存储器对总线的访问冲突。
- 4) 支持更为广泛的存储器型号。通过对FSMC的时间参数设置，扩大了系统中可用存储器的速度范围，为用户提供了灵活的存储芯片选择空间。
- 5) 支持代码从FSMC扩展的外部存储器中直接运行，而不需要首先调入内部SRAM。

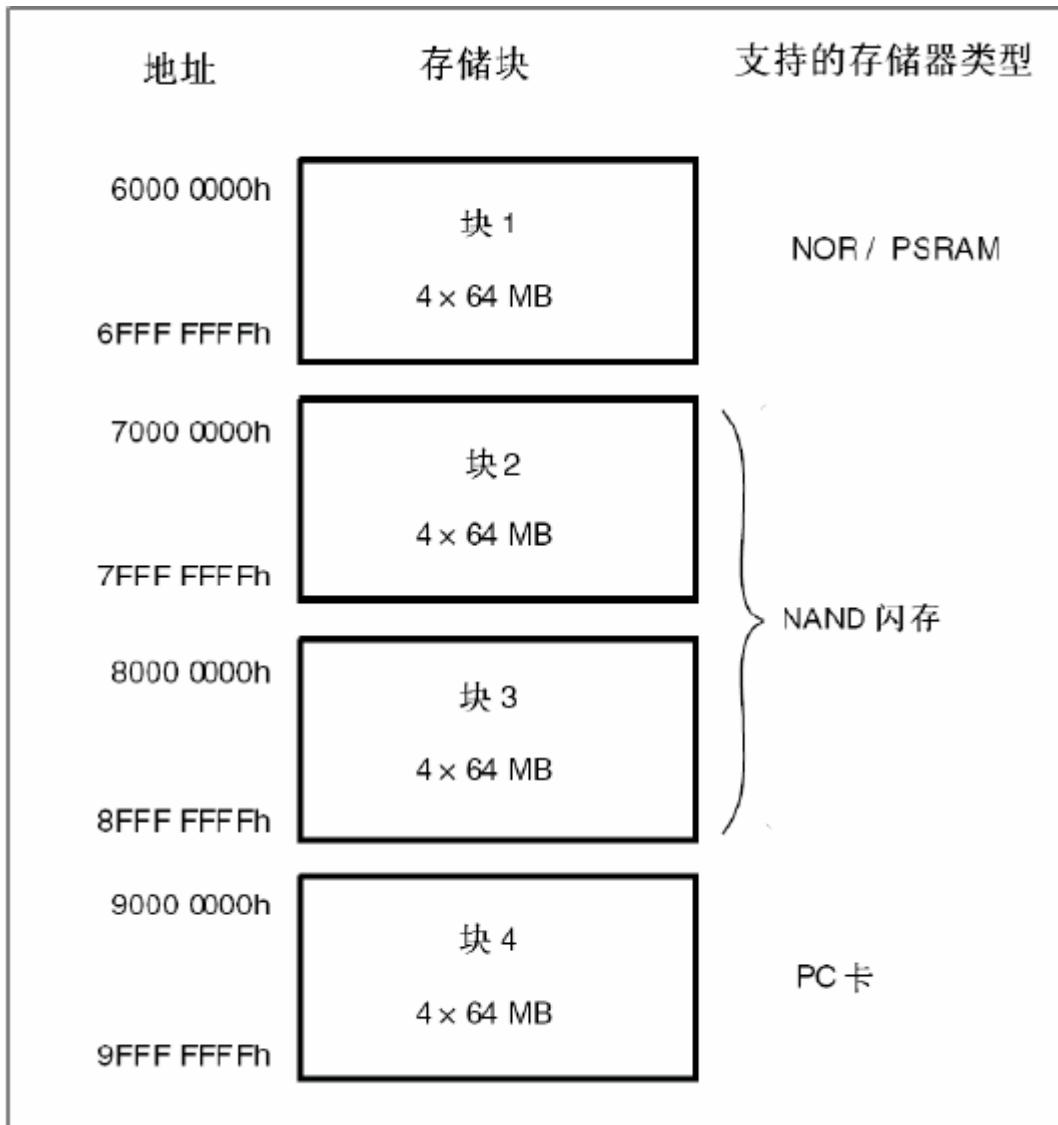
这里我们介绍下为什么可以把TFTLCD当成SRAM设备用：首先我们了解下外部SRAM的连接，外部SRAM的控制一般有：地址线（如A0~A18）、数据线（如D0~D15）、写信号（WR）、读信号（OE）、片选信号（CS），如果SRAM支持字节控制，那么还有UB/LB信号。而TFTLCD的信号包括：RS、D0~D15、
嵌入式专业技术论坛 (www.armjishu.com) 出品 第 607 页, 共 900 页

WR、RD、CS、RST和BL等，其中真正在操作LCD的时候需要用到的就只有：RS、D0~D15、WR、RD和CS；这里比较关键点就是TFT LCD并不需要用到FSMC的地址线（如A0~A18），它只需要用到16根数据线以及一些其他的控制信号线就可以了，并且操作时序和SRAM的控制完全类似，下面是对比表：

| 资源 | SRAM | 神舟TFT LCD |
|--------------------|------|-----------|
| 写信号 (WE) | 有 | 有 |
| 读信号 (OE) | 有 | 有 |
| 片选信号 (CS) | 有 | 有 |
| 数据线 (D0~D15) | 有 | 有 |
| 地址线 (如A0~A18) | 有 | 无 |
| RS (TFT中决定是数据还是命令) | 无 | 有 |
| RST | 无 | 有 |
| BL | 无 | 有 |

TFT LCD通过RS信号来决定传送的数据是数据还是命令，本质上可以理解为一个地址信号，比如我们把RS接在A0上面，那么当FSMC控制器写的地址刚好让A0这根信号线为0（A1~A18因为并没有连接到TFT上，所以可以忽略不计）的时候，会使得连在A0上的TFT信号RS也为0，对TFT LCD来说，就是写命令；同样，当FSMC写地址的地址让A0这个地址变成1的时候，A0上的TFT信号RS也会变成1，对TFTLCD来说，就是写数据了。这样，就把数据和命令区分开了，当然RS也可以接在其他地址线上，STM32神舟开发板是把RS连接在FSMC_A0上面的。

STM32的FSMC支持8/16/32位数据宽度，我们这里用到的LCD是16位宽度的，所以在配置FSMC控制器属性的时候，选择16位宽就OK了。我们再来看看FSMC的外部设备地址映像，STM32的FSMC将外部存储器划分为固定大小为256M字节的四个存储块，如下图：



图：FSMC存储块地址映像

从上图可以看出，FSMC总共管理1GB空间，拥有4个存储块（Bank），本章，我们用到的是块1，所以在本章我们仅讨论块1的相关配置，其他块的配置，请参考《STM32参考手册》第19章（324页）的相关介绍。

STM32的FSMC存储块1（Bank1）被分为4个区，每个区管理64M字节空间，每个区都有独立的寄存器对所连接的存储器进行配置。Bank1的256M字节空间由28根地址线（2的28次方是256M，其中26根地址线来自外部存储器地址FSMC_A[25:0]），另外2根地址线对4个区进行寻址（AHB总线是26和27）：

| Bank1所选区 | 片选信号 | 地址范围 | AHB总线 | |
|----------|----------|-------------------------|---------|--------------|
| | | | [27:26] | [25:0] |
| 第1区 | FSMC_NE1 | 0X6000,0000~0X63FF,FFFF | 00 | FSMC_A[25:0] |
| 第2区 | FSMC_NE2 | 0X6400,0000~0X67FF,FFFF | 01 | |
| 第3区 | FSMC_NE3 | 0X6800,0000~0X6BFF,FFFF | 10 | |
| 第4区 | FSMC_NE4 | 0X6C00,0000~0X6FFF,FFFF | 11 | |

当我们选择使用Bank1的第2个区，即使用FSMC_NE2来外接外部设备的时候，即对应了FSMC[27:26]=01，我们要做的就是配置对应第2区的寄存器组，来适应外部设备即可；同样，如果当我们选择使用Bank1的第4个区，即使用FSMC_NE4来外接外部设备的时候，即对应了FSMC[27:26]=11，我们要做的就是配置对应第4区的寄存器组，STM32的FSMC总线各Bank配置寄存器如表：

| 内部控制器 | 存储块 | 管理的地址范围 | 支持的设备类型 | 配置寄存器 |
|-------------------------------|-------|-------------------------------|----------------------------------|---|
| NOR FLASH 控制器 | Bank1 | 0X6000, 0000~ 0X6FFF, FFFF | SRAM / ROM NOR FLASH PSRAM | FSMC_BCR1/2/3/4 FSMC_BTR1/2/2/3 FSMC_BWTR1/2/3/4 |
| NAND FLASH /PC CARD 控制器 | Bank2 | 0X7000, 0000~ 0X7FFF, FFFF | NAND FLASH | FSMC_PCR2/3/4 FSMC_SR2/3/4 FSMC_PMEM2/3/4 FSMC_PATT2/3/4 |
| | Bank3 | 0X8000, 0000~ 0X8FFF, FFFF | | FSMC_PI04 |
| | Bank4 | 0X9000, 0000~ 0X9FFF, FFFF | PC Card | |

可以看到，我们需要操作的TFT彩屏与操作SRAM是相同的控制器，都是负责存储块的Bank1的内部控制器NOR FLASH控制器，对于NOR FLASH控制器，主要是通过FSMC_BCRx (x=1、2、3、4)、FSMC_BTRx (x=1、2、3、4) 和FSMC_BWTRx (x=1、2、3、4) 寄存器设置（其中x=1~4，对应Bank中的4个区），下面介绍一下这几个配置寄存器各负责的功能：

FSMC_BCRx (x=1, 2, 3, 4)：SRAM/NOR闪存片选控制寄存器 (FSMC_BCR1, FSMC_BCR2, FSMC_BCR3, FSMC_BCR4) 每个Bank对应1个寄存器，这个寄存器包含了每个存储器块的控制信息，可以用于SRAM、ROM、异步或成组传输的NOR闪存存储器 (TFT LCD也可以)，主要包括设置数据总线宽度，比如8位、16位；存储类型的设置，比如SRAM或者PSRAM或者NOR闪存；地址/数据是否复用；存储块使能设置。

FSMC_BTRx (x=1, 2, 3, 4)：闪存片写时序寄存器 (FSMC_BTR1, FSMC_BTR2, FSMC_BTR3, FSMC_BTR4)；可以通过这个寄存器设置模式，总线恢复时间，时钟分频，数据保持时间等时序相关的设置。

FSMC_BWTRx (x=1, 2, 3, 4)：闪存写时序寄存器 (FSMC_BWTR1, FSMC_BWTR2, FSMC_BWTR3, FSMC_BWTR4)；采用哪种访问模式，数据保持时间，地址保持时间，底子好建立时间（单位是HCLK时钟周期）

通过这3个寄存器，可以设置FSMC访问外部存储器的时序参数，可以使得支持更多的外部存储器，拓宽了可选用的外部存储器的速度范围。例如：FSMC的NOR FLASH控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC将HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号FSMC_CLK。此时需要的设置的时间参数有2个（这些都不用太关注，有需要时再去研究，我们已经实现了具体的代码，可以直接从代码中去看一下如何进行设置的）：

1, HCLK与FSMC_CLK的分频系数(CLKDIV)，可以为2~16分频；

2, 同步突发访问中获得第1个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC主要设置3个时间参数（这些都不用太关注，有需要时再去研究，我们已经实现了具体的代码，可以直接从代码中去看一下如何进行设置的）：地址建立时间(ADDSET)、数据建立时间(DATAST) 和地址保持时间(ADDHLD)。FSMC综合了SRAM / ROM、PSRAM和NOR Flash产品的信号特点，定义了4种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数，如下表所列：

| 时序模型 | 简单描述 | 时间参数 |
|------|--------------------------|--------------------|
| 异步 | Mode1 | SRAM/CRAM 时序 |
| | ModeA | SRAM/CRAM OE 选通型时序 |
| | Mode2/B | NOR FLASH 时序 |
| | ModeC | NOR FLASH OE 选通型时序 |
| | ModeD | 延长地址保持时间的异步时序 |
| 同步突发 | 根据同步时钟FSMC_CK读取多个顺序单元的数据 | CLKDIV、DATLAT |

这里只是简单的介绍一些关于FSMC的参数设置，FSMC最终被设置成能够按照FSMC所控制的目

标器件的时序，规范来与之沟通，至于详细的设置，在这个例程中不需要花费太多精力关注细节，大体知道是这么一回事就可以了，如果有必要用到的时候，就具体去进行研究就可以了。

7.36.6 FSMC函数初始化

1. FSMC初始化函数：

根据前面的讲解，初始化**FSMC**主要是初始化三个寄存器**FSMC_BCRx**，**FSMC_BTRx**，**FSMC_BWTRx**，那么在固件库中是怎么初始化这三个参数的呢？

固件库提供了3个**FSMC**初始化函数分别为

FSMC_NORSRAMInit(); (可以用来初始化NOR和SRAM存储器)

FSMC_NANDInit(); (可以用来初始化NAND存储器)

FSMC_PCCARDInit(); (可以用来初始化PC CARD等存储器)

这三个函数分别用来初始化4种类型存储器。这里根据名字就很好判断对应关系，下面我们看看函数定义：

```
void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct);
```

这个函数只有一个入口参数，也就是**FSMC_NORSRAMInitTypeDef**类型指针变量，这个结构体的成员变量非常多，因为**FSMC**相关的配置项非常多。

```
typedef struct
{
    uint32_t FSMC_Bank;
    uint32_t FSMC_DataAddressMux;
    uint32_t FSMC_MemoryType;
    uint32_t FSMC_MemoryDataWidth;
    uint32_t FSMC_BurstAccessMode;
    uint32_t FSMC_AsynchronousWait;
    uint32_t FSMC_WaitSignalPolarity;
    uint32_t FSMC_WrapMode;
    uint32_t FSMC_WaitSignalActive;
    uint32_t FSMC_WriteOperation;
    uint32_t FSMC_WaitSignal;
    uint32_t FSMC_ExtendedMode;
    uint32_t FSMC_WriteBurst;
    FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;
    FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;
}FSMC_NORSRAMInitTypeDef;
```

从这个结构体我们可以看出，前面有13个基本类型(**unit32_t**)的成员变量，这13个参数是用来配置片选控制寄存器**FSMC_BCRx**。最后面还有两个**FSMC_NORSRAMTimingInitTypeDef**指针类型的成员变量。**FSMC**有读时序和写时序之分，这里就是用来设置读时序和写时序的参数了，也就是说，这两个参数是用来配置寄存器**FSMC_BTRx**(闪存片读时序寄存器)和**FSMC_BWTRx**(闪存片写时序寄存器)，后面我们会讲解到。下面我们主要来看看模式A下的相关配置参数：

参数-存储块标号：参数**FSMC_Bank**用来设置使用到的存储块标号和区号，前面讲过，我们是使

用的存储块1区号4，所以选择值为FSMC_Bank1_NORSRAM4。

参数 `FSMC_MemoryType` 用来设置存储器类型，我们这里是 SRAM，所以选择值为 `FSMC_MemoryType_SRAM`。

参数-数据位宽度：参数 `FSMC_MemoryDataWidth` 用来设置数据宽度，可选8位还是16位，这里我们是16位数据宽度，所以选择值为 `FSMC_MemoryDataWidth_16b`。

参数-写使能：参数 `FSMC_WriteOperation` 用来设置写使能，毫无疑问，我们前面讲解过我们要向 TFT 写数据，所以要写使能，这里我们选择 `FSMC_WriteOperation_Enable`。

参数 `FSMC_ExtendedMode` 是设置扩展模式使能位，也就是是否允许读写不同的时序，这里我们采取的读写不同时序，所以设置值为 `FSMC_ExtendedMode_Enable`。

下面再介绍一下其他几个相关参数的意义吧：参数 `FSMC_DataAddressMux` 用来设置地址/数据复用使能，若设置为使能，那么地址的低16位和数据将共用数据总线，仅对 NOR 和 PSRAM 有效，所以我们设置为默认值不复用，值 `FSMC_DataAddressMux_Disable`。

参数 `FSMC_BurstAccessMode`, `FSMC_AsynchronousWait`, `FSMC_WaitSignalPolarity`, `FSMC_WaitSignalActive`, `FSMC_WrapMode`, `FSMC_WaitSignal` `FSMC_WriteBurst` 和 `FSMC_WaitSignal` 这些参数在成组模式同步模式才需要设置，大家可以参考中文参考手册了解相关参数的意思。

接下来我们看看设置读写时序参数的两个变量 `FSMC_ReadWriteTimingStruct` 和 `FSMC_WriteTimingStruct`，他们都是 `FSMC_NORSRAMTimingInitTypeDef` 结构体指针类型，这两个参数在初始化的时候分别用来初始化片选控制寄存器 `FSMC_BTRx` 和写操作时序控制寄存器 `FSMC_BWTRx`。下面我们看看 `FSMC_NORSRAMTimingInitTypeDef` 类型的定义：

```
typedef struct
{
    uint32_t FSMC_AddressSetupTime;
    uint32_t FSMC_AddressHoldTime;
    uint32_t FSMC_DataSetupTime;
    uint32_t FSMC_BusTurnAroundDuration;
    uint32_t FSMC_CLKDivision;
    uint32_t FSMC_DataLatency;
    uint32_t FSMC_AccessMode;
}FSMC_NORSRAMTimingInitTypeDef;
```

这个结构体有7个参数用来设置FSMC读写时序。其实这些参数的意思我们前面在讲解FSMC的时序的时候有提到，主要是设计地址建立保持时间，数据建立时间等等配置，对于我们的实验中，读写时序不一样，读写速度要求不一样，所以对于参数 `FSMC_DataSetupTime` 设置了不同的值，大家可以对照理解一下。记住，这些参数的意义在前面讲解 `FSMC_BTRx` 和 `FSMC_BWTRx` 寄存器的时候都有提到，大家可以翻过去看看。

2.FSMC使能函数

FSMC对不同的存储器类型同样提供了不同的使能函数：

```
void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);
void FSMC_NANDCmd(uint32_t FSMC_Bank, FunctionalState NewState);
void FSMC_PCCARDCmd(FunctionalState NewState);
```

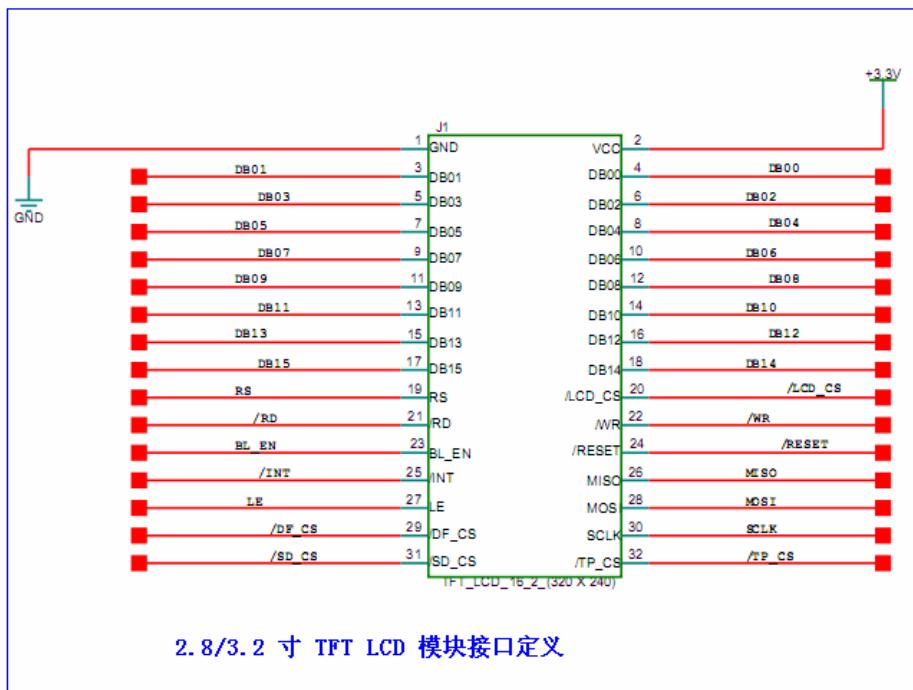
这个就比较好理解，我们这里不讲解，我们是 SRAM，所以使用的第一个函数；这些如果有必要再去详细了解，因为这个都是 ST 官方设计 STM32 芯片的时候就已经预先规定好的了，我们只需要按照他们的规则和要求去设置和使用它就足够了，因为其他芯片厂商不一定会使用 FSMC 这个协议的。

7.36.7 硬件设计

本实验的硬件设计包括两个方面：

- 一、TFT LCD屏的原理设计，主要体现TFT LCD屏上的信号连接情况；
- 二、神舟III号开发板上的TFT座的信号连接情况。

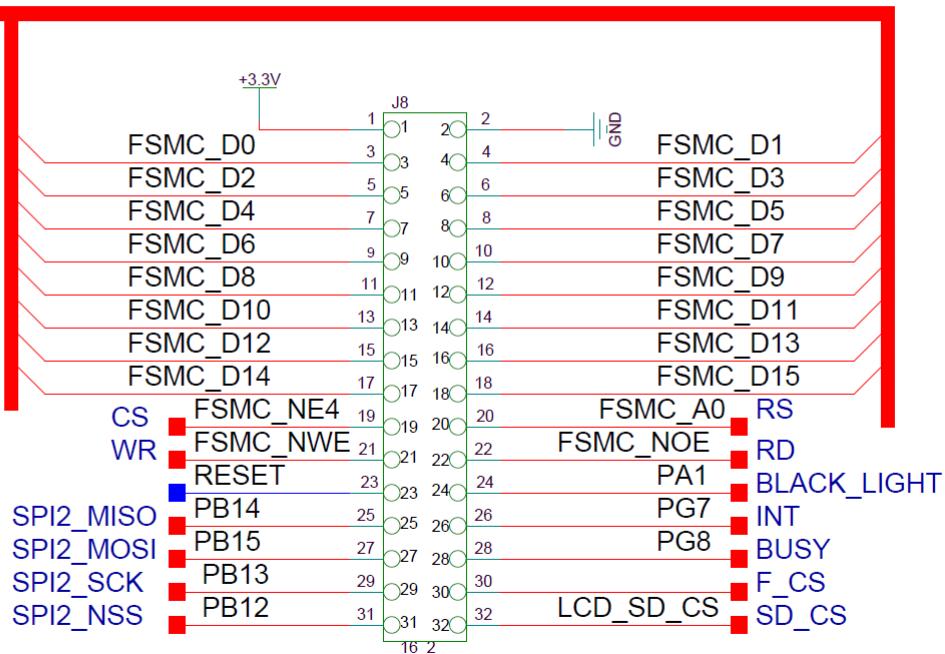
首先是TFT的硬件设计：



下面这些信号是真正与LCD液晶屏相连接的：

2.8/3.2 LCD

FSMC_D[15:0]



可以看到FSMC的数据线FSMC_D0~FSMC_D15都被使用了，而FSMC的地址线只使用了FSMC_A0这根管脚跟RS（TFT LCD选择数据线是发的是命令还是发的是数据这根关键的信号线）连接。

7.36.8 软件分析

进入例程的文件夹，然后打开\ MDK-ARM \ Project.uvproj 文件

```

C:\Documents and Settings\Administrator\Desktop\神舟III号彩色液晶屏\39.TFT彩色液晶屏只显示红色(神舟王103-库函数版)\MDK-ARM\Project.uvproj
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
File Explorer Task List Properties Project Navigator Search
Project main.c
main.c
01 **** (C) COPYRIGHT 2013 STMicroelectronics ****
02 * File Name      : main.c
03 * Author         : WWW.ARMJISHU.COM之STM32核心团队 JESSE
04 * Version        :
05 * Date          :
06 * Description    : STM32神舟III号开发板 广州
07 ****
08
09 #include "stm32f10x.h"
10 #include "ili9320.h"
11 #include "ili9320_api.h"
12 #include <stdio.h>
13
14 int main(void)
15 {
16     ili9320_Initialization(); //TFT LCD彩色液晶屏初始化
17     while (1)
18     {
19         ili9320_Clear(Red); //TFT LCD彩色液晶屏显示红色
20     }
21 }

```

可以看到工程已经被打开，下面开始具体分析程序代码：

```

int main(void)
{
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化
    while (1)
    {
        ili9320_Clear(Red); //TFT LCD彩色液晶屏显示红色
    }
}

```

代码分析 1：首先掉用 ili9320_Initialization() 函数对 TFT LCD 彩色液晶屏进行初始化，然后在 while() 循环中调用 ili9320_Clear(Red) 函数来循环显示红色。

代码分析 2：LCD_WriteReg() 这个是写 LCD 的寄存器函数，可以看到这个函数实现代码如下：

```

void LCD_WriteReg(u16 LCD_Reg,u16 LCD_RegValue)
{
    LCD->LCD_REG = LCD_Reg;
    LCD->LCD_RAM = LCD_RegValue;
}

```

可以看到 LCD 这个结构体的定义如下：

```
#define LCD_BASE ((u32)(0x60000000 | 0x0C000000))
```

这里的 LCD_BASE，是根据外部电路的连接来确定的，这里使用的 Bank1 的第 4 个区，就是从地址 0x6C00,0000 开始，看下表：

| Bank1所选区 | 片选信号 | 地址范围 | AHB总线 | |
|----------|----------|-------------------------|---------|--------------|
| | | | [27:26] | [25:0] |
| 第1区 | FSMC_NE1 | 0X6000,0000~0X63FF,FFFF | 00 | FSMC_A[25:0] |
| 第2区 | FSMC_NE2 | 0X6400,0000~0X67FF,FFFF | 01 | |
| 第3区 | FSMC_NE3 | 0X6800,0000~0X6BFF,FFFF | 10 | |
| 第4区 | FSMC_NE4 | 0X6C00,0000~0X6FFF,FFFF | 11 | |

在使用 FSMC 驱动液晶屏的时，通过前面的硬件原理图可以看到，TFT LCD 的 RS 是接在 FSMC_A0 上面，CS 接在 FSMC_NE4 上，并且是 16 位数据总线。

#define LCD ((LCD_TypeDef *) LCD_BASE) 关于 LCD_TypeDef 结构体地址，被强制指定到地址 LCD_BASE（地址为：0x6C00,0000）那么可以得到 LCD->LCD_REG 的地址就是 0x6C00,0000，对应的 FSMC_A0 的状态为 0（即 RS=0，RS 是前面提到过的 TFT 液晶屏命令和数据的控制信号线，写 0 表示命令），那么 LCD->LCD_RAM 的地址就是 0x6C00,0001（结构体地址自增，因为都是 vu16 型号的），那么此时对应的 FSMC_A0 的状态为 1（即 RS=1，RS 是前面提到过的 TFT 液晶屏命令和数据的控制信号线，写 1 表示数据）。

```

typedef struct
{
    vu16 LCD_REG;
    vu16 LCD_RAM;
} LCD_TypeDef;

```

所以，有了这个定义，LCD->LCD_REG 就是写命令，而 LCD->LCD_RAM 就是默认对某个寄存器写数据；所以如果是写 LCD，可以先确定写哪个寄存器，然后再往这个寄存器写什么数据内容：

LCD->LCD_REG = 要写的命令,写哪个寄存器

LCD->LCD_RAM = 对哪个寄存器要写什么样的数据

反过来，读 LCD 的内部的数据就是相反：

LCD->LCD_REG = 要读的那个寄存器

读出来的数据是什么 = LCD->LCD_RAM

代码分析 3: LCD_X_Init() 函数完成 LCD 与 CPU 之间管脚的初始化, 包括初始化 FSMC 总线

void LCD_X_Init(void)

{

 LCD_CtrlLinesConfig(); //配置 LCD 控制管脚

 LCD_FSMCConfig(); //配置 FSMC 总线接口

}

1) LCD_CtrlLinesConfig()配置 LCD 控制管脚

```
void LCD_CtrlLinesConfig(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable FSMC, GPIOD, GPIOE, GPIOF, GPIOG and AFIO clocks */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOE |
                           RCC_APB2Periph_GPIOF | RCC_APB2Periph_GPIOG | RCC_APB2Periph_GPIOC |
                           RCC_APB2Periph_AFIO, ENABLE);
    /* Set PD.00(D2), PD.01(D3), PD.04(NOE), PD.05(NWE), PD.08(D13), PD.09(D14),
       PD.10(D15), PD.14(D0), PD.15(D1) as alternate
       function push pull */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_4 | GPIO_Pin_5 |
                                 GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_14 |
                                 GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
```

配置 LCD 控制管脚, 连接 TFT 彩色液晶屏的这些管脚都要被初始化一下时钟和 GPIO 管脚功能, 具体哪些管脚可以看下 TFT 液晶屏的原理图。

2) LCD_FSMCConfig()配置 FSMC 总线接口

配置 FSMC 总线接口, 以及 FSMC 总线控制器的初始化

```
FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM4;
FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;
FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_SRAM;
FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;
FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait = FSMC_AsynchronousWait_Disable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_BeforeWaitState;
FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &Timing_read;
FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &Timing_write;

FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
```

这里设置了比如: 数据/地址复用=禁用; 数据宽度 = 16 位; 写操作 = 使能; 扩展模式 = 使能; 异步等待 = 禁用等, 主要是配置了 FSMC 控制器, 具体细节如果有兴趣的话可以看下 STM32 的使用手册的 FSMC 章节进行研究, 这里代码中都已经设置配好了。

代码分析 4: 读取液晶屏的 ID 型号, 0x0000h 指令, 当为读操作时, 读取控制器的型号;

```
u16 LCD_ReadReg(u8 LCD_Reg)
{
    u16 data; /* Write 16-bit Index (then Read Reg) */
    LCD->LCD_REG = LCD_Reg;
    data = LCD->LCD_RAM;
    return data;
}
```

- 1) LCD->LCD_REG = 0x0000 表示这是 0x0000 指令，是读取液晶屏的 ID 型号，实际上是将 LCD 的 LCD_TypeDef 这个 STRUCT 结构体的地址设定到 0x0000 然后接下来用 LCD->LCD_RAM 来读取数据。
- 2) data = LCD->LCD_RAM 是用来读取数据，这个 0x0000 指令地址的数据就是液晶屏的 ID 号，比如 IL9320 的液晶屏读出来的 ID 号是 0x9320，比如 SSD1289 型号的液晶屏读出来的 ID 号就是 0x8989

代码分析 5：开始判断液晶屏是哪个型号，比如是 9320 还是 SSD1289 型号，判断出型号之后，就会进入该型号具体的配置显示过程

```
if (DeviceCode==0x9320||DeviceCode==0x9300)
{
    else if (DeviceCode==0x9331)
    {
        LCD_WriteReg(0x00E7, 0x1014);
        LCD_WriteReg(0x0001, 0x0100); //
        LCD_WriteReg(0x0002, 0x0200); //
        LCD_WriteReg(0x0003, 0x1030); //
        LCD_WriteReg(0x0008, 0x0202); //
        LCD_WriteReg(0x0009, 0x0000); //
        LCD_WriteReg(0x000A, 0x0000); //
        LCD_WriteReg(0x000C, 0x0000); //
        LCD_WriteReg(0x000D, 0x0000); //
        LCD_WriteReg(0x000F, 0x0000); //
        //*****Power On sequence
        LCD_WriteReg(0x0010, 0x0000); //
        LCD_WriteReg(0x0011, 0x0007); //
        LCD_WriteReg(0x0012, 0x0000); //
        LCD_WriteReg(0x0013, 0x0000); //
        ili9320_Delay(200); // Dis-charge
        LCD_WriteReg(0x0010, 0x1690); //
        LCD_WriteReg(0x0011, 0x0227); //
        ili9320_Delay(50); // Delay 50ms
        LCD_WriteReg(0x0012, 0x000C); //
        ili9320_Delay(50); // Delay 50ms
        LCD_WriteReg(0x0013, 0x0800); //
        LCD_WriteReg(0x0029, 0x0011); //
        LCD_WriteReg(0x002B, 0x000B); //
    }
}
```

```
else if (DeviceCode==0x8989)
{
    LCD_WriteReg(0x0000,0x0001);      //打开晶振
    LCD_WriteReg(0x0010,0x0000);
    Delay(5); // Wait 30ms
    LCD_WriteReg(0x0007,0x0233);
    LCD_WriteReg(0x0011,0x6078); //定义数据格式 16位色
    LCD_WriteReg(0x0002,0x0600);
    LCD_WriteReg(0x0003,0xA8A4); //0x0804
    LCD_WriteReg(0x000C,0x0000);
    LCD_WriteReg(0x000D,0x080C);
    LCD_WriteReg(0x000E,0x2900);
    LCD_WriteReg(0x001E,0x00B8);
    LCD_WriteReg(0x0001,0x293F);
    LCD_WriteReg(0x0010,0x0000);
    LCD_WriteReg(0x0005,0x0000);
    LCD_WriteReg(0x0006,0x0000);
    LCD_WriteReg(0x0016,0xEF1C);
    LCD_WriteReg(0x0017,0x0003);
    LCD_WriteReg(0x0007,0x0233);      //0x0233
    LCD_WriteReg(0x000B,0x0000|(3<<6));
    LCD_WriteReg(0x000F,0x0000);      //扫描开始地址
    LCD_WriteReg(0x0041,0x0000);
    LCD_WriteReg(0x0042,0x0000);
    LCD_WriteReg(0x0048,0x0000);
    LCD_WriteReg(0x0049,0x013F);
    LCD_WriteReg(0x004A,0x0000);
```

比如设置晶振打开，液晶屏的显示大小（比如是 230*320），显示的扫描方式是从左边往右边扫描还是从下往上扫描等参数，具体可以详细看一下液晶屏的寄存器说明表。

代码分析 6: ili9320_Clear(Red) 将 TFT LCD 彩色液晶屏显示红色，将屏幕填充成指定的颜色，如清屏，则填充 0xffff；如显示红色，则填充 0xF800。

```
void ili9320_Clear(u16 Color)
{
    u32 index=0;
    ili9320_SetCursor(0,0);
    LCD_WriteRAM_Prep(); /* Prepare to write GRAM */
    for(index=0;index<76800;index++)
    {
        LCD->LCD_RAM=Color;
    }
}
```

- i. ili9320_SetCursor(0,0)先设置坐标，然后往坐标写颜色
- ii. LCD_WriteRAM_Prep()准备填充 GRAM，填充 GRAM 就是写数据到液晶屏里
- iii. 写颜色，一个点一个点的写，这里的液晶屏是 320*240 点阵的，所以 $320 \times 240 = 76800$

```
for(index=0;index<76800;index++)
{
    LCD->LCD_RAM=Color;
}
```

代码分析 7: ili9320_SetCursor(0,0) 通过 ili9320_SetCursor() 函数可以实现光标的位置，可以看到 SSD1289 芯片手册中，关于寄存器 4E 和 4F 寄存器分别是设置光标的 X 位置和 Y 位置：

```

void ili9320_SetCursor(u16 x,u16 y)
{
    if(DeviceCode==0x8989)
    {
        LCD_WriteReg(0x004e,y);      //行
        LCD_WriteReg(0x004f,x);    //列
    }
}

```

| | |
|-------------|---|
| R4Eh | Set GDDRAM X address counter (0000h) |
| R4Fh | Set GDDRAM Y address counter (0000h) |

表示刷屏就是从这个点开始刷。

代码分析 8: LCD_WriteRAM_Prepae () 函数

```

void LCD_WriteRAM_Prepae(void)
{
    ClrCs
    LCD->LCD_REG = R34;
    SetCs
}

```

R34, 写数据到 GRAM 命令, 当写入了这个命令之后, 地址计数器才会自动的增加和减少。该命令是我们要介绍的这一组命令里面唯一的单个操作的命令, 只需要写入该值就可以了, 其他的都是要先写入命令编号, 然后写入操作数。

代码分析 8: 开始一个点一个点的写入颜色到 GRAM 里, 因为之前的 R34 已经定位好了 LCD 的 STRUCT 结构体的位置, 那么可以计算一下 GRAM 的大小, 总共是 $320*240 = 76800$ 个点, 每个点的颜色是 5:6:5 的模式是 16 位来表示, 就是 2 个字节, 所以 GRAM 的大小是 $76800*2$ 字节 = 150KB 大小, 也就是说, 液晶屏的 GRAM 大小最少是这么大才能存到一屏数据。

```

for(index=0;index<76800;index++)
{
    LCD->LCD_RAM=Color;
}

```

7.36.9 下载与测试

如果使用JLINK下载固件, 请按 [如何使用JLINK V8下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件, 请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中, 下载编译好的固件或者在线调试, 请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.36.10 实验现象

将固件下载到神州 III 号开发板后, 复位, 神州 III 号开发板正常情况下将显示红色



7.37 TFT彩色液晶屏显示蓝色

7.37.1 硬件设计

硬件设计同上

7.37.2 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件

A screenshot of the MDK-ARM integrated development environment. The title bar shows the path: C:\Documents and Settings\Administrator\桌面\神舟III号彩色液晶屏\40.TFT彩色液晶屏只显示蓝色(神舟III号-库函数版)\MDK-ARM\Project.uvproj. The main window displays the 'main.c' source code. The code initializes the LCD and then repeatedly clears it with blue color. A project tree on the left shows files like 'stm32f10x_it.c', 'main.c', 'ili9320_api.c', and 'ili9320.c'.

```
01 //***** (C) COPYRIGHT 2013 STMicroelectronics *****
02 * File Name      : main.c
03 * Author         : WWW.ARMJISHU.COM之STM32核心团队 JESSE
04 * Version        :
05 * Date          :
06 * Description    : STM32神舟III号开发板 广州
07 ****
08
09 #include "stm32f10x.h"
10 #include "ili9320.h"
11 #include "ili9320_api.h"
12 #include <stdio.h>
13
14 int main(void)
15 {
16     ili9320_Initialization(); //TFT LCD彩色液晶屏初始化
17     while (1)
18     {
19         ili9320_Clear(Blue); //TFT LCD彩色液晶屏显示红色
20     }
21 }
```

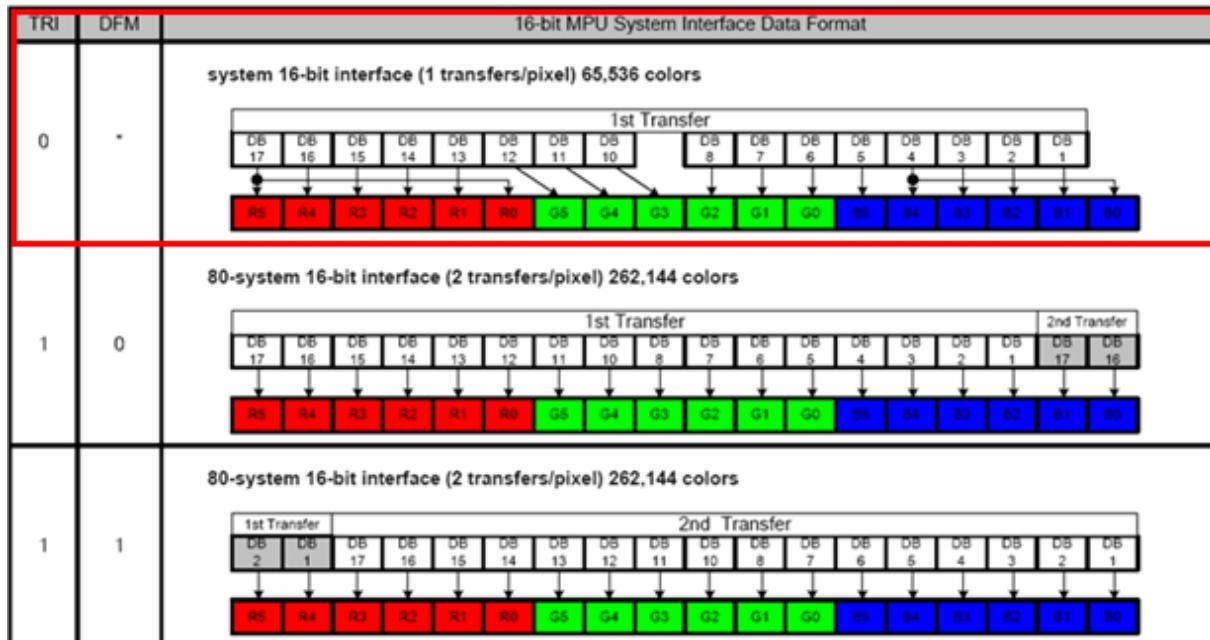
可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化
    while (1)
    {
        ili9320_Clear(Blue); //TFT LCD彩色液晶屏显示蓝色
    }
}
```

代码分析 1: ili9320_Clear(Blue);可以看到还有很多其他的颜色，其中 blue 是蓝色

```
#define White          0xFFFF //白色
#define Black           0x0000 //黑色
#define Grey            0xF7DE //灰色
#define Blue             0x001F //蓝色
#define Red              0xF800 //红色
#define Magenta          0xF81F //蓝色
#define Green             0x07E0 //绿色
#define Yellow            0xFFE0 //黄色
```

其中#define Blue 0x001F 我们把 0x001F 化成二进制就是 0000 0000 0001 1111，对于 5:6:5 的显示模式来说，可以看下图，DB1~DB5 这个 5 位是都是 1，它是设定蓝色颜色的；而 DB6~DB12 都是 0 表示不显示绿色；DB12~DB17 也是为 0，它是设定红色颜色，表示不显示红色，这样一来就只会显示出蓝色的颜色。



7.37.3 下载与测试

如果使用JLINK下载固件，请按 [如何使用JLINK V8下载固件到神舟III号开发板](#) 小节进行操作。
如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.37.4 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将显示蓝色



7.38 TFT彩色液晶屏如何显示一个点

7.38.1 扫描的简要分析

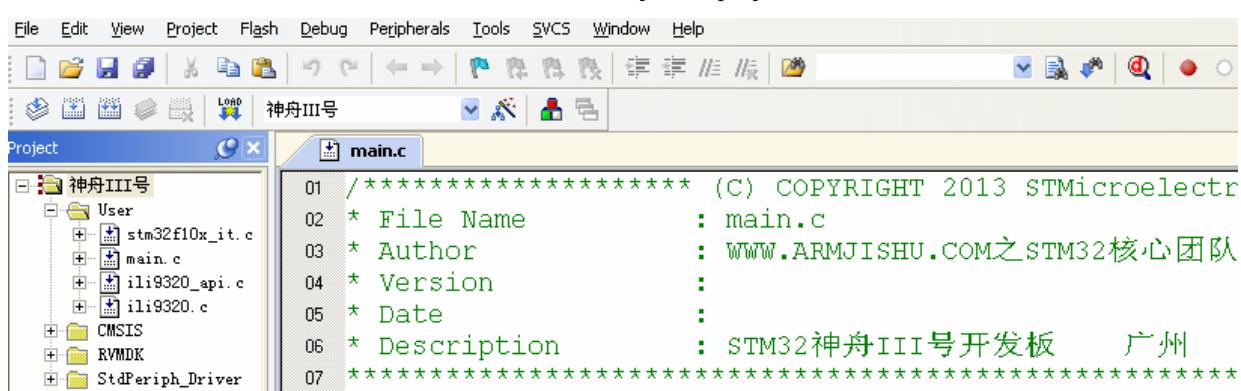
本实验在 LCD 屏上指定的位置显示一个点。

7.38.2 硬件设计

硬件设计同上

7.38.3 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化

    while (1)
    {
        ili9320_SetPoint(120, 160, Black);
    }
}
```

代码分析 1: ili9320_Initialization(), 彩屏初始化函数。前面我们已经讲了, 我们就不再重复, 这里它
嵌入式专业技术论坛 (www.armjishu.com) 出品 第 622 页, 共 900 页

主要是将彩屏初始化，并将彩屏刷成绿色。

代码分析 2: ili9320_SetPoint(120, 160, Black), 画点函数。

```
void ili9320_SetPoint(u16 x,u16 y,u16 point)
{
    if ( (x>320) || (y>240) ) return;
    ili9320_SetCursor(x,y);

    LCD_WriteRAM_Prepares();
    LCD_WriteRAM(point);
}
```

画点的时候，先确定位置，然后再画点。值得注意的是，这个点有点小，仔细看才能看出来。

代码分析 3: 函数 ili9320_SetCursor()设置光标的位置，即指定画点的位置。

```
void ili9320_SetCursor(u16 x,u16 y)
{
    if(DeviceCode==0x8989)
    {
        LCD_WriteReg(0x004e,x); //行
        LCD_WriteReg(0x004f,y); //列
    }
    else if((DeviceCode==0x9919))
    {
        LCD_WriteReg(0x004e,x); // 行
        LCD_WriteReg(0x004f,y); // 列
    }
    else
    {
        LCD_WriteReg(0x0020,x); // 行
        LCD_WriteReg(0x0021,0x13f-y); // 列
    }
}
```

不同的液晶控制器，设置光标位置使用的寄存器是不同的。比如说，控制器 8989 设置光标的时候用到的是寄存器 0x004e 和 0x004f。而控制器 9320，用到的是 0x0020 和 0x0021。这个在控制器的芯片手册中有具体的说明。

设置光标函数，有两个参数，这两个参数是有范围限定的。比如我们这里的液晶屏是 240x320。那么函数的第一个参数的范围是：0<=Xpos<=240。第二个参数的范围是：0<=Ypos<=320。

代码分析 4: LCD_WriteRAM_Prepares()函数，准备向 RAM 中写数据。

```
void LCD_WriteRAM_Prepares(void)
{
    ClrCs
    LCD->LCD_REG = R34;
    SetCs
}
```

这里向控制器送入 LCD_REG_34 命令。而 LCD_REG_34 等于 0x22。

```
#define LCD_REG_33          0x21
#define LCD_REG_34          0x22
#define LCD_REG_36          0x24
```

0x22 命令是写数据到 GRAM 命令，写入这个命令后，地址计数器才会自动的增加和减少。这里我们写入这个命令，为下面填充数据做准备。

代码分析 5：调用 LCD_WriteRAM() 函数，向 LCD RAM 写数据。

```
void LCD_WriteRAM(u16 RGB_Code)
{
    ClrCs
    /* Write 16-bit GRAM Reg */
    LCD->LCD_RAM = RGB_Code;
    SetCs
}
```

通过函数 LCD->LCD_RAM = RGB_Code 向 GRAM 写数据。即向对应的彩屏的 FSMC 端口发送数据，我们这里发送的是画点的数据。

为什么是彩屏的 FSMC 呢？这个可以从地址上看得出来。LCD 是类型为 LCD_TypeDef，指向 LCD_BASE 的指针。LCD_BASE 的地址定义如下：

```
/* Private typedef -----
typedef struct
{
    vu16 LCD_REG;
    vu16 LCD_RAM;
} LCD_TypeDef;

/* LCD is connected to the FSMC Bank1 NOR/SRAM4 and
#define LCD_BASE ((u32)(0x60000000 | 0x0C000000))
#define LCD ((LCD_TypeDef *) LCD_BASE)
```

画点函数，就是通过彩屏的 FSMC 端口向 GRAM 存储单元传输数据。而 GRAM 中每个存储单元都对应着液晶控制器面板的一个像素点。LCD 控制器（9320、8989 等等）通过其它的组件的共同作用把我们写入 GRAM 存储单元中的数据转化为液晶面板控制信号，使对应的像素点呈现特定的颜色。

传输完数据之后，拉高片选信号线（CS），表示一次传输数据完成。到此画点函数，我们分析完成。

7.38.4 下载与测试

如果使用 JLINK 下载固件，请按 [3.3 如何使用 JLINK 软件下载固件到神舟 III 号开发板小节](#) 进行操作。
如果使用串口下载固件，请按 [3.4 如何通过串口下载一个固件到神舟 III 号开发板小节](#) 进行操作。

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 [3.8 如何在 MDK 开发环境中使用 JLINK 在线调试小节](#) 进行操作。

7.38.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将显示绿色，在屏的中间显示“ARMjishu.com”字样。

示一个黑点。注意，我们只是显示一个点，看的时候要仔细，可以把保护膜撕下来观察黑点。



7.39 TFT彩色液晶屏显示一个数字1

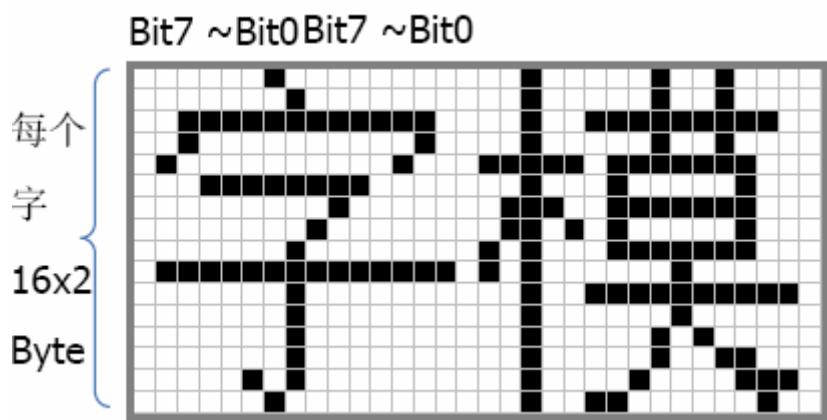
7.39.1 扫描的简要分析

本实验我们在 LCD 屏上指定的位置显示一个数字 1。我们这个 1 是通过字模显示的。

7.39.2 什么是字模

液晶屏其实就是一个由许多像素点组成的点阵，若要在上面显示一个字符，则需要很多像素点共同构成，比如 8*16 的 ASCII 码字符，或者 16*16 的点阵显示的汉字。

如果每个 8*16 或者 16*16 的点阵区域来显示一个字符，把黑色的像素点以 1 来表示，空白以 0 表示，每个像素点的状态以一个二进制位来记录，用 8*16/8=16 个字节或者 16*16/8=32 个字节就可以把这个字记录下来。这个 16 字节或者 32 字节就称之为该字符的字模。当然还有其他常用的字模是 24*24，32*32 的。

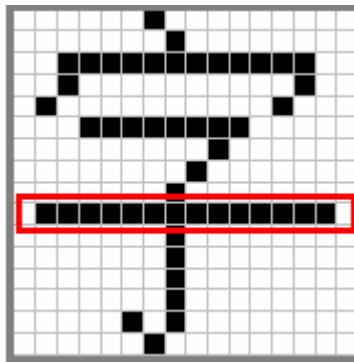


可以把这个‘字模’两个字转化成以下的 32 个字节的字模：

```
0x02, 0x00, 0x01, 0x00, 0x3F, 0xFC, 0x20, 0x04, 0x40, 0x08, 0x1F, 0xE0, 0x00, 0x40,  
0x00, 0x80,  
0xFF, 0xFF, 0x7F, 0xFE, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x05, 0x00,  
0x02, 0x00,
```

比如看下，开头的 0x02 和 0x00，就是显示‘字’的最上面那一点，显示第一行；比如下面这一行黑体，

可以数一下，从上面往下数是第 10 行，那么可以看到第 10 行是 0x7F 和 0xFE，确实是一个正确的字模显示；在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模：



7.39.3 ASCII码的字符解释

下图是 ASCII 码字符表

ASCII 字符代码表 一

| | | ASCII非打印控制字符 | | | | | | | | | | ASCII 打印字符 | | | | | | | | | | | |
|------|-----|--------------|---------------|------|-----|-------|------|----|------|-----|--------|------------|----|------|----|------|----|------|----|------|-----|------------|----|
| | | 0000 | | | | | 0001 | | | | | 0010 | | 0011 | | 0100 | | 0101 | | 0110 | | 0111 | |
| 高四位 | 低四位 | +逆制 | 字符 | ctrl | 代码 | 字符解释 | +逆制 | 字符 | ctrl | 代码 | 字符解释 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 |
| | | 0 | BLANK NULL | ^@ | NUL | 空 | 16 | ▶ | ^ P | DLE | 数据链路转意 | 32 | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p | |
| 0000 | 0 | 0 | ☺ | ^A | SOH | 头标开始 | 17 | ◀ | ^ Q | DC1 | 设备控制 1 | 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 0001 | 1 | 1 | ☻ | ^B | STX | 正文开始 | 18 | ↑ | ^ R | DC2 | 设备控制 2 | 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 0010 | 2 | 2 | ♥ | ^C | ETX | 正文结束 | 19 | !! | ^ S | DC3 | 设备控制 3 | 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 0011 | 3 | 3 | ◆ | ^D | EOT | 传输结束 | 20 | ¶ | ^ T | DC4 | 设备控制 4 | 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 0100 | 4 | 4 | ♣ | ^E | ENQ | 查询 | 21 | ƒ | ^ U | NAK | 反确认 | 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 0101 | 5 | 5 | ♦ | ^F | ACK | 确认 | 22 | ■ | ^ V | SYN | 同步空间 | 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 0110 | 6 | 6 | ♠ | ^G | BEL | 震铃 | 23 | ↑ | ^ W | ETB | 传输块结束 | 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 1000 | 8 | 8 | █ | ^H | BS | 退格 | 24 | ↑ | ^ X | CAN | 取消 | 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 1001 | 9 | 9 | ○ | ^I | TAB | 水平制表符 | 25 | ↓ | ^ Y | EM | 媒体结束 | 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 1010 | A | 10 | ▣ | ^J | LF | 换行/新行 | 26 | → | ^ Z | SUB | 替换 | 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 1011 | B | 11 | ♂ | ^K | VT | 竖直制表符 | 27 | ← | ^ [| ESC | 转意 | 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 1100 | C | 12 | ♀ | ^L | FF | 换页/新页 | 28 | ↙ | ^ \ | FS | 文件分隔符 | 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 1101 | D | 13 | ♪ | ^M | CR | 回车 | 29 | ↔ | ^] | GS | 组分隔符 | 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } |
| 1110 | E | 14 | ♫ | ^N | SO | 移出 | 30 | ▲ | ^ 6 | RS | 记录分隔符 | 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 1111 | F | 15 | ⌚ | ^O | SI | 移入 | 31 | ▼ | ^- | US | 单元分隔符 | 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | Δ |
| | | | | | | | | | | | | | | | | | | | | | | Back space | |

现在将这些 ASCII 字符表转化成 8x16 的数组，也就是说一个字符的高度是 8，宽度是 16，16 也就是两个字节，那么一个字符就是 8 行，每行都是 16 个 bit，等于是 $8*16/8 = 16$ 个字节大小。可以看到下面的 ASCII 字符，红框框住的是 16 个字节，显示一个字符。

可以看到整个数组 `ascii_8x16[1536]` 总共有 1536 个成员，每个成员就是一个单独的字节，而 16 个字节就是一个字符，那么这个数组总共能描述多少个字符呢？总共是 $1536/16 = 96$ 个字符，从下面的这张表来看，这个 96 个字符就是描述的是从 ASCII 值 32 到 127 这 96 个字符。

| ASCII 打印字符 | | | | | | | | | | | |
|------------|----|------|----|------|----|------|----|------|----|------|--------------|
| 0010 | | 0011 | | 0100 | | 0101 | | 0110 | | 0111 | |
| 2 | | 3 | | 4 | | 5 | | 6 | | 7 | |
| +进制 | 字符 | +进制 | 字符 | +进制 | 字符 | +进制 | 字符 | +进制 | 字符 | +进制 | 字符 ctrl |
| 32 | | 48 | 0 | 64 | @ | 80 | P | 96 | ' | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | △ Back space |

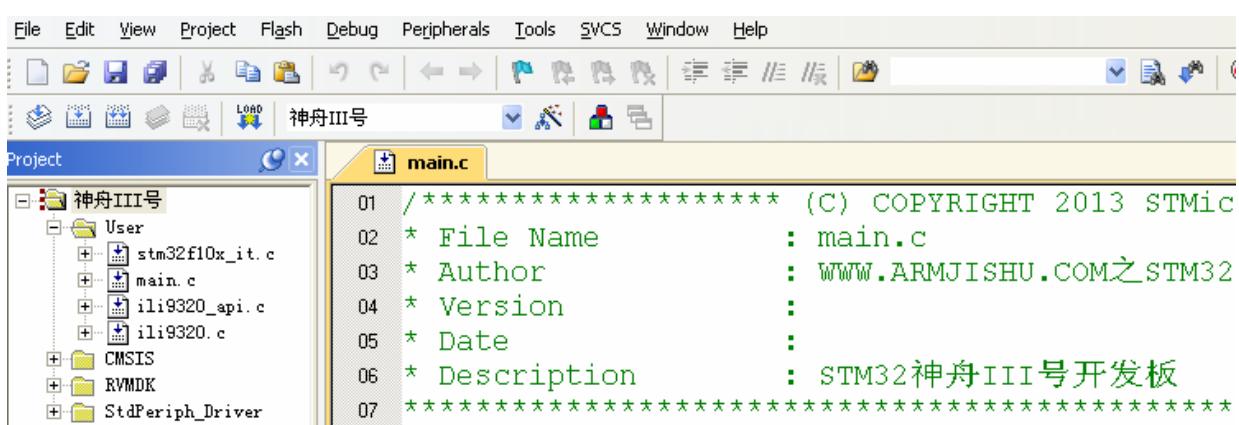
比如我们希望显示www.armjishu.com 到液晶屏上，那么这个LOGO所有的字符都能在ASCII码表中通过查表获得，接下来的软件分析会进一步分析如何用具体的软件代码来显示字符到LCD液晶屏上。

7.39.4 硬件设计

硬件设计同上

7.39.5 软件分析

进入例程的文件夹，然后打开\ MDK-ARM \ Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化

    while (1)
    {
        LCD_DrawChar(10, 100, &ASCII_Table[0]);
    }
}
```

代码分析 1: ili9320_Initialization(), 彩屏初始化函数。前面我们已经讲了，我们就不再重复，这里它主要是将彩屏初始化，并将彩屏刷成绿色。

代码分析 2: LCD_DrawChar(10, 100, &ASCII_Table[0]), 显示数字 1 函数。

```
void LCD_DrawChar(uint16_t Xpos, uint16_t Ypos, const uint8_t *cpFontArray)
{
    uint32_t index = 0, i = 0, j = 0, k = 0, y;
    uint8_t Xaddress = 0;
/******Armjishu.com Add { *****/
    uint32_t Width = 24;
    uint32_t Height = 48;
    uint32_t BytesPreChar = 144;
/******Armjishu.com Add } *****/
    Xaddress = Xpos;

    LCD_SetCursor(Xaddress, Ypos);

    for(y = 0; y < Height; y++)
    {
        LCD_WriteRAM_Prepares();
        for(i = Width; i > 0;)
        {
            if(i>=8)
            {
                k = 8;
            }
```

函数 LCD_DrawChar (), 一共 3 个参数，第一、第二个参数设定字符显示的坐标。第三个参数设定要显示的字符。

代码分析 3: 显示字符函数中。本实验使用的字符大小是 24x48。即显示一个字模需要 $24 \times 48 = 1152$ 个 bit, $1152/8 = 144$ 个字节。

```
uint32_t Width = 24;
uint32_t Height = 48;
uint32_t BytesPreChar = 144;
```

代码分析 3: 设定字符显示的位置调用函数 LCD_SetCursor (), 这个函数我们前面已经说明，这里就不再重复。下面的 for 循环用到我们前面 24、48，即 Width 和 Height 数据。For 循环通过这两个数据限定显示字符的大小。

```
for(y = 0; y < Height; y++)
{
    LCD_WriteRAM_Prepares();
    for(i = Width; i > 0;)
    {
        if(i>=8)
        {
```

我们显示一个字符使用到 $24 \times 48 = 1152$ 个 bit，我们编写代码的时候，设定高度（Height）是 48，宽度（width）是 24。每画完一次宽度（width），高度（Height）加 1。依次次循环，最终将整个字符显示出来。

代码分析 4：在 `for(i = Width; i > 0;)` 循环里面，对数据进行相应的判断及显示。

```
for(y = 0; y < Height; y++)
{
    LCD_WriteRAM_Prepares();
    for(i = Width; i > 0;)
    {
        if(i>=8)
        {
            k = 8;
        }
        else
        {
            k = i;
        }

        for(j = 0; j < k; j++)
        {
            if((cpFontArray[index] & (0x80 >> j)) == 0x00)
            {
                if(HyalineBack == HyalineBackDis)
                {
                    LCD_WriteRAM(BackColor); //putchar(' ');
                }
                else
                {
                    LCD_SetCursor(Xaddress, Ypos+(Width - i)+1);
                    LCD_WriteRAM_Prepares();
                }
            }
            else
            {
                LCD_WriteRAM(TextColor); //putchar('*');
            }
        }
    }
}
```

我们这里 `i = Width`，前面我们定义了 `Width` 等于 24，所以调用 `if` 语句进行判断赋值的时候 `K=8`。`cpFontArray[index]` 是我们传进来的第三个参数。这个参数是一个指针，指向我们要显示的数据的首地址。最终通过判断决定对应的点是显示黑色(`BackColor`)还是白色(`TextColor`)。

代码分析 5：我们知道显示一个点，是设定显示点的坐标，然后写数据。我们数字 1 也是一样的道理，不同的是我们写入了很多点的数据。我们看一下显示的结果。

下面我们将对数字 1 的数据进行分析，在此我们数字 1 设定的字模是 24×48 。

```
unsigned char ASCII_Table[144] =  
{  
    0x00, 0x00, 0x00,  
    0x00, 0x08, 0x00,  
    0x00, 0x18, 0x00,  
    0x00, 0x38, 0x00,  
    0x07, 0xF8, 0x00,  
    0x00, 0x38, 0x00,  
}, 0x38, 0x00,
```

上面我们说了，字模的宽度是 24，高度是 48。我们先显示完一个宽度，高度加 1，再显示一个宽度，高度再加 1，如此循环，最终显示一个字符。数字 1 的字模数据中，为了方便大家观察，我们将数据分成 3 个字节一行。刚好是 $3 \times 8 = 24$ (bit)，和宽度吻合。其实这样划分之后，大家数一下数字 1 数据的高度，刚好是 48 行的。

我们显示一个数字 1，从形状上，我们可以看到它的头是斜的，尾部是一横，中间部分是 1 坚。那么我们的数据中间部分应该是一样的。我们对一下数据发现中间的数据都是 0x00, 0x38, 0x00。我们将它化为二进制等于 0000 0000 0011 1000 0000 0000，就是说一个宽度中，中间有 3 个点是黑色的，两边都是白色的。连续这样的数据就组成了 1 坚（数字 1 的中间部分）。

数字 1 的头部是上窄下宽，那么头部的数据从上往下，应该是上面的数据是 1 的位数比下面的少。

```
UXUU, UXUU, UXUU,  
0x00, 0x00, 0x00,  
0x00, 0x08, 0x00,  
0x00, 0x18, 0x00,  
0x00, 0x38, 0x00,  
0x07, 0xF8, 0x00,  
0x00, 0x38, 0x00,  
0x00, 0x38, 0x00,  
0x00, 0x38, 0x00,  
0x00, 0x38, 0x00,
```

大家观察一下，它头部的数据。从上往下，数据中 1 的位数增大。

大家看一下数字 1 尾部的数据：

```
0x00,0x38,0x00,  
0x00,0x38,0x00,  
0x00,0x7C,0x00,  
0x07,0xFF,0xC0,  
0x00,0x00,0x00,  
0x00,0x00,0x00,
```

0x07,0xFF,0xC0, 化为二进制为 0000 0111 1111 1111 1100 0000, 这样的一个宽度数据组成一横(数字 1 的底部)。

代码分析 6: 我们看一下实验的效果



通过显示的结果我们可以推断出, 液晶屏刷的方向。我们可以看到显示的 1 是竖着的, 我们可以判断出它的刷屏是在水平方向上的。如果刷屏方向是在竖直方向上的, 那么数字 1 的显示应该是卧倒的。

那么竖屏是从左到右刷, 还是从右到左刷呢? 刷屏的起始点在哪里呢? 我们看一下数字 1 的头部, 它的突出部分是在左边。那么它刷屏的方向, 是从左到右刷屏。从右向左的刷的话, 突出部分是指向右边的。刷屏的起始点是在左上角的第一个点。如果是在右下脚的话, 显示的数字 1 就倒立过来了。

那么通过数字 1 的显示, 可以看出屏在显示数字 1 的时候显示的方向是: 从左到右刷屏, 而且是从左上角开始刷屏。

7.39.6 下载与测试

如果使用JLINK下载固件, 请按 [如何使用JLINK V8下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件, 请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中, 下载编译好的固件或者在线调试, 请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.39.7 实验现象

将固件下载到神舟 III 号开发板后, 复位, 神舟 III 号开发板正常情况下将显示数字 1。



7.40 TFT彩色液晶屏显示变异的数字1

7.40.1 简要分析

本实验我们对数字 1 的字模数据，进行改造显示特殊的数字 1，改变显示 1 的数组的值，看看这个数字 1 会不会变形。

7.40.2 软件分析

代码分析 1：显示数字 1 的实验的时候，我们已经对相关的函数进行了解释。我们这里主要针对两个实验的数字 1 的数据进行对比分析。

(显示数字 1 的数据)

(显示变异数字 1 的数据)

我们观察它两的数据发现，变异的数字1的数据和普通数字1的数据不同的地方是在数据的中间

部分。普通的数字 1 的数据的中间部分是一长串的宽度数据，`0x00,0x38,0x00`。变异数字 1 的数据中间部分是一段宽度数据 `0x00,0x38,0x00`，加上一段宽度数据 `0x00,0xFF,0x00`，再加上一段宽度数据 `0x00,0x38,0x00` 构成。

`0x00,0x38,0x00` 化为二进制的数据是 `0000 0000 0011 1000 0000 0000`。`0x00,0xFF,0x00` 化为二进制的数据是 `0000 0000 1111 1111 0000 0000`。很明显，`0x38` 只有 3 个 1，而 `0xFF` 是 8 个 1。那么最终在液晶屏上显示的时候，特殊的数字 1 中间部分不是规则的一竖，应该是两头小一点，中间胖一点的一竖。

7.40.3 软件分析

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将显示特殊数字 1。



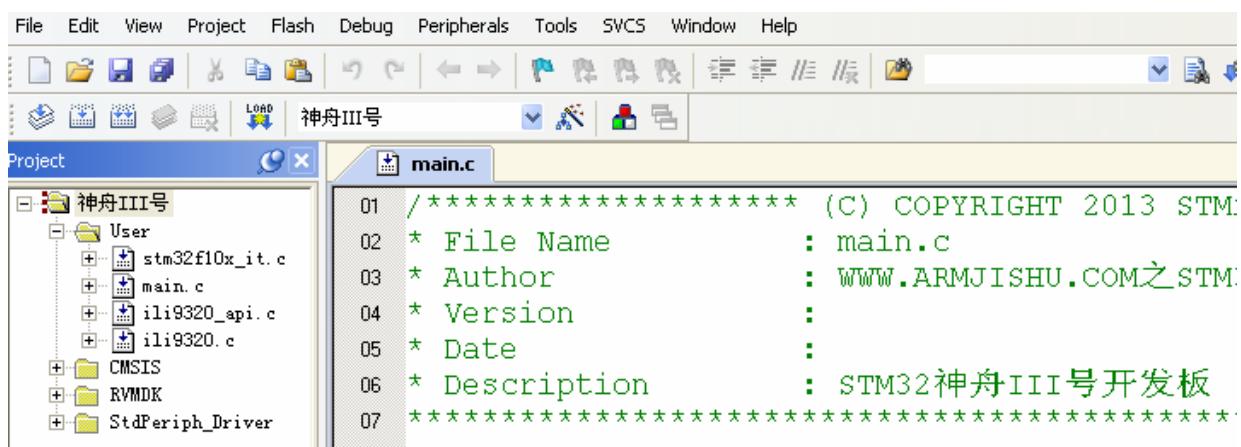
7.41 TFT彩色液晶屏显示数字9

7.41.1 简要分析

上个例程显示了一个数字 1 之后，这个例程我们打算显示一个数字 9，再加深一下这个字模转成数字的过程。

7.41.2 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化

    while (1)
    {
        LCD_DrawChar(10, 100, &ASCII_Table[0]);
    }
}
```

代码分析 1：显示数字 1 的实验的时候，我们已经对相关的函数进行了解释。我们这里主要针对数字 9 的数据进行对比分析。

(显示数字1的数据)

```
unsigned char ASCII_Table[144] = {  
    0x00, 0x00, 0x00,  
    0x01, 0xF8, 0x00,  
    0x07, 0x06, 0x00,  
    0x0C, 0x01, 0x00,  
    0x1C, 0x01, 0x80,  
    0x38, 0x00, 0xC0,  
    0x38, 0x00, 0xC0,  
    0x30, 0x00, 0x60,  
    0x70, 0x00, 0x60,  
    0x70, 0x00, 0x60,  
    0x70, 0x00, 0x70,  
    0x70, 0x00, 0x70,  
    0x70, 0x00, 0x70,  
    0x70, 0x00, 0x70,  
    0x70, 0x00, 0xF0,  
    0x38, 0x00, 0xF0,  
    0x38, 0x01, 0xF0,  
    0x3C, 0x03, 0x70,  
    0x1F, 0x0E, 0x70,  
    0x0F, 0xFC, 0x70,  
    0x03, 0xF0, 0x70,  
    0x00, 0x00, 0x60,
```

(显示数字9的数据)

我们观察它两的数据发现，变异的数字 1 的数据和数字 9 的数据有很多的不同点。首先，数字 1 的中间部分是规律性的 0x00.0x38.0x00。而数字 9 并没有这样子的规律。

我们看一下，数字 9 的数据。开头的都是 0x00，那些都是空白的，我们不管它。第一个非 0 的数据宽度是 0x01,0xF8,0x00。化为二进制是 0000 0001 1111 1000 0000 0000，大家发现这个二进制中间部分有连续的 6 个 1，那么我们的这个 9 的头部是封顶的。

往下的一长串数据中，中间部分都都有 0。比如 `0x70,0x00,0x60`。我们这个数字 9，中间是空的。空完之后，下面封一次底，完成数字 9 的上面的圆圈部分。

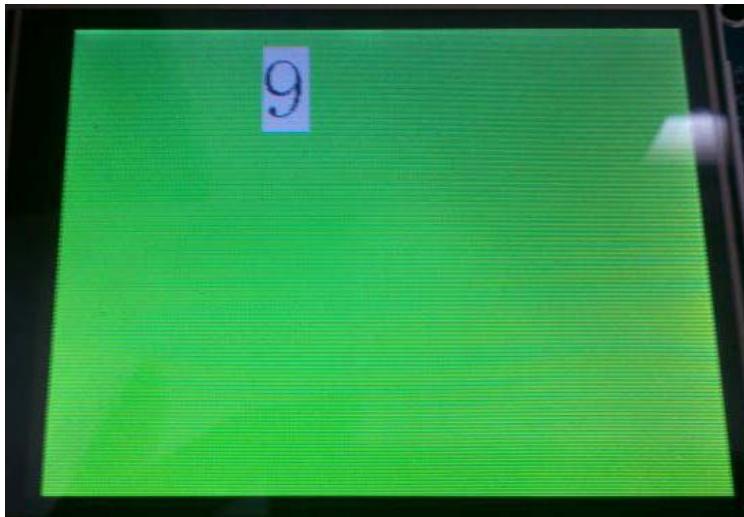
0x0F, 0xFC, 0x70,
0x03, 0xF0, 0x70,

7.4.1.3 下载测试

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.41.4 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将显示数字 9。



7.42 TFT彩色液晶屏显示一个英文

7.42.1 简要分析

本实验我们显示一个大写的英文字母 A。

7.42.2 硬件设计

硬件设计同上

7.42.3 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件

A screenshot of the MDK-ARM integrated development environment. The top menu bar includes File, Edit, View, Project, Flash, Debug, Peripherals, Tools, SVCS, Window, and Help. Below the menu is a toolbar with various icons. The title bar shows "神舟III号". The left side features a "Project" view with a tree structure containing "User", "CMSIS", "RVMKD", and "StdPeriph_Driver" folders. The main workspace shows the file "main.c" with the following content:

```
01 /***** (C) COPYRIGHT 2013 STM
02 * File Name      : main.c
03 * Author         : WWW.ARMJISHU.COM之STM
04 * Version        :
05 * Date          :
06 * Description    : STM32神舟III号开发板
07 *****/
```

可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化

    while (1)
    {
        LCD_DrawChar(10, 100, &ASCII_Table[0]);
    }
}
```

代码分析 1：显示英文字母的实验原理，和显示数字的实验原理是一样的。我们主要对比一下显示数字 9 和显示英文 A 的数据。

```
unsigned char ASCII_Table[144] =
{
    0x00,0x00,0x00,
    0x00,0x00,0x00,
    0x00,0x00,0x00,
    0x00,0x00,0x00,
    0x00,0x00,0x00,
    0x00,0x00,0x00,
    0x00,0x00,0x00, 显示数字9的数据
    0x00,0x00,0x00, 据
    0x00,0x00,0x00,
    0x01,0xF8,0x00,
    0x07,0x06,0x00,
    0x0C,0x01,0x00,
    0x1C,0x01,0x80,
    0x38,0x00,0xC0,
    0x38,0x00,0xC0,
    0x30,0x00,0x60,
    0x70,0x00,0x60,
    0x70,0x00,0x60,
    0x70,0x00,0x70,
    0x70,0x00,0x70,
    0x70,0x00,0x70,
    0x70,0x00,0x70,
    0x70,0x00,0x70,
    0x70,0x00,0xF0,
    0x38,0x00,0xF0,
    0x38,0x01,0xF0,
    0x3C,0x03,0x70,
    0x1F,0x0E,0x70.
},                                     unsigned char ASCII_Table[144] =
{
    0x00,0x00,0x00,
    0x00,0x00,0x00, 显示英文A的数据
    0x00,0x00,0x00,
    0x00,0x00,0x00,
```

这个和显示数字，基本是一样的，最大的区别是显示的内容不一样。这个可以从显示数据中看出来。我们显示引文的时候，字模的大写也是 24x48。

分析一下字母 A 的数据。我们发现字母 A 的数据，只有一行是连续不断的十几个位都是 1。从 A 的形状上不难猜出它描述的是 A 中间的横杠。

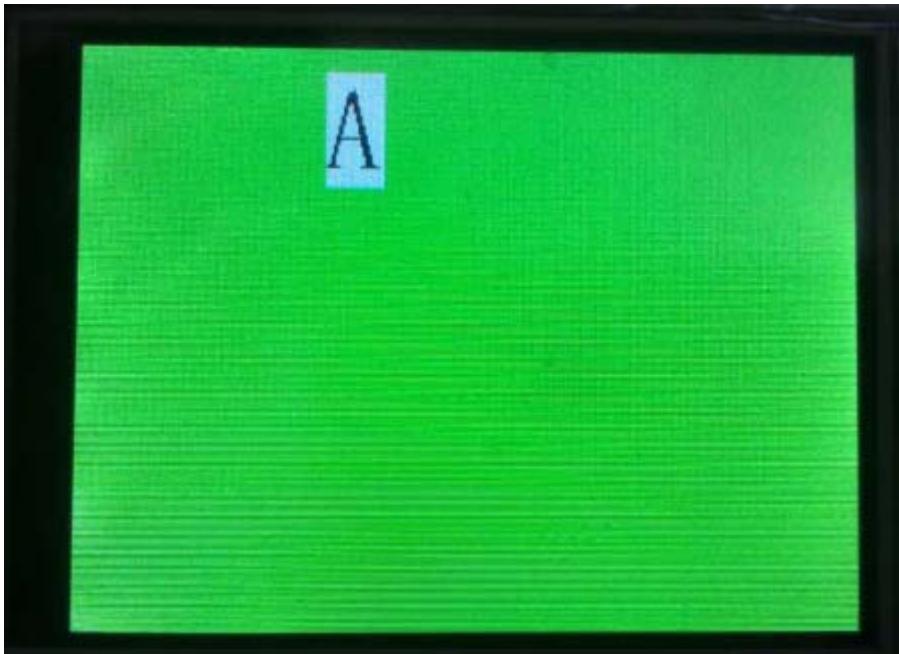
```
0x0C,0x03,0x80,
0x0F,0xFF,0x80,
0x0C,0x03,0x80,
```

7.42.4 下载与测试

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.42.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将显示英文字母 A。



7.43 TFT彩色液晶屏显示26个英文字母

7.4.3.1 简要分析

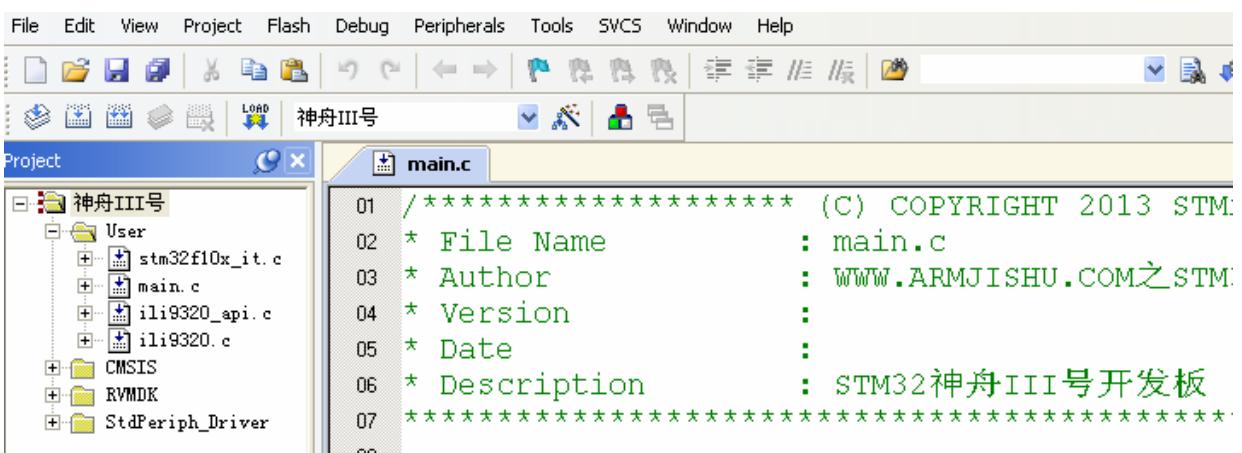
本实验显示 26 个英文字母

7.4.3.2 硬件设计

硬件设计同上

7.43.3 软件分析

进入例程的文件夹，然后打开\MDK-ARM \ Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    uint8_t i=0,j=0;
    ili9320_Initialization(); //TFT LCD彩色液晶屏初始化

    /*TFT-LCD彩屏显示数字1*/
    LCD_DrawChar(10, 100, &ASCII_Table[0]);
    for(j=0;j<2;j++)
    {
        for(i=0;i<13;i++)
        {
            LCD_DrawChar(j*48, i*24, &ASCII_Table[(i+j*13)*144]);
        }
    }
}
```

代码分析 1: ili9320_Initialization(), 彩屏初始化函数。前面我们已经讲了, 我们就不再重复, 这里它的主要是将彩屏初始化, 并将彩屏刷成绿色。

代码分析 2：显示 26 个英文字母。

```
for(j=0;j<2;j++)
{
    for(i=0;i<13;i++)
    {
        LCD_DrawChar(j*48, i*24, &ASCII_Table[(i+j*13)*144]);
    }
}
```

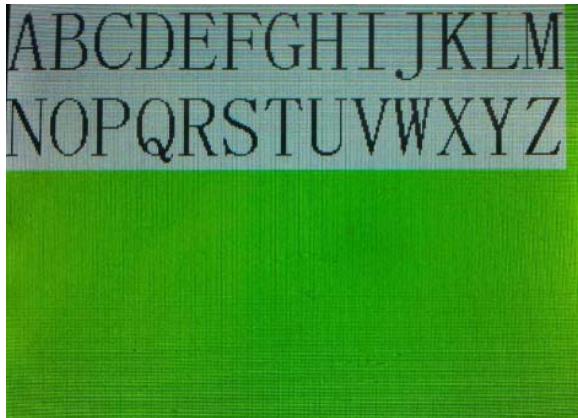
本实验和显示单个数字、单个英文基本一致。不同的是，通过 for 循环增加了显示的英文的数量。一个英文字母的宽度是 24 位，13 个占的位数是 $24 \times 13 = 312$ (bit)。我们屏幕宽度 320，所以我们分两行显示，每行显示 13 个字母。具体的字符数据如下：

7.43.4 下载与测试

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.4.3.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将显示 26 个英文字母。



7.44 TFT彩色液晶屏显示图片

7.44.1 图片格式介绍

图片格式有非常多的种类，比如 JPEG、BMP 和 GIF 等多种不同的类型，我们实验中主要是使用 LCD 屏来显示 BMP 图片。

BMP 文件格式，又称为位图（Bitmap）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 文件保存了一幅图像中所有的像素。

BMP 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，每种模式中单一像素点的大小分别为 1/8 字节，1/2 字节，1 字节和 32 字节。目前最常见的是 256 位色 BMP 和 24 位色 BMP。

BMP 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，只有一大小 || 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，但在 24 位色 BMP 文件则没有，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。

7.44.2 深入了解BMP图片

BMP 文件的格式大体上分为 4 部分。

第一部分为位图文件头。

第二部分为位图信息头。

第三部分为调色板。

第四部分就是实际的图像数据了。

BMP 的内容，网上的资料很多，都说烂了。从操作上讲，如果想在液晶屏上显示图片的话，还是比较简单的。大体上说，使用我们提供的图像转换软件，将要显示的图像转换成数据，替代我们提供的代码中的图像数据就可以了。

针对本实验，我们直接看一下转换出来的 BMP 图像数据。

转换的出的数据，并非都是用于绘制的图像实际数据。在图像数据之前还有一串数据。比如：
0X00,0X10,0X40,0X01,0XF0,0X00,0X01,0X1B,。

这里的 0x40,0x01, 表示图像的宽度。0xF0, 0x00 表示图像的高度。0x0140 化为 10 进制等于 320; 0x00F0 化为 10 进制是 240。其它位代表的意义，我们在代码中进行说明，在这里不一列举。

7.44.3 使用工具将图片转换成二进制码



在本实例中，在main函数中有以下函数调用：

```
LCD_Image2LcdDrawBmp565Pic(0, 0, gImage_C);
```

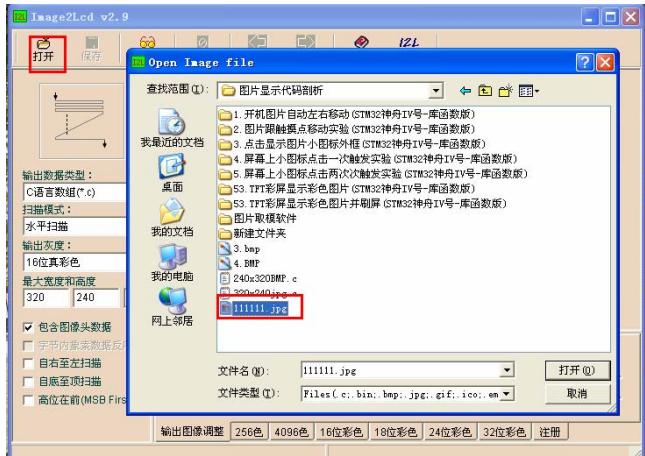
`LCD_Image2LcdDrawBmp565Pic()` 函数可以在指定的位置显示图片。当然显示的大小是有限制的。比如我们本实验显示的图片是 240×320 ，屏幕也是 240×320 。那么我们显示图片的时候，指定的显示坐标应该是 $(0,0)$ ，刚好整屏显示。

本实验使用的图片转换工具是：Img2Lcd.exe。

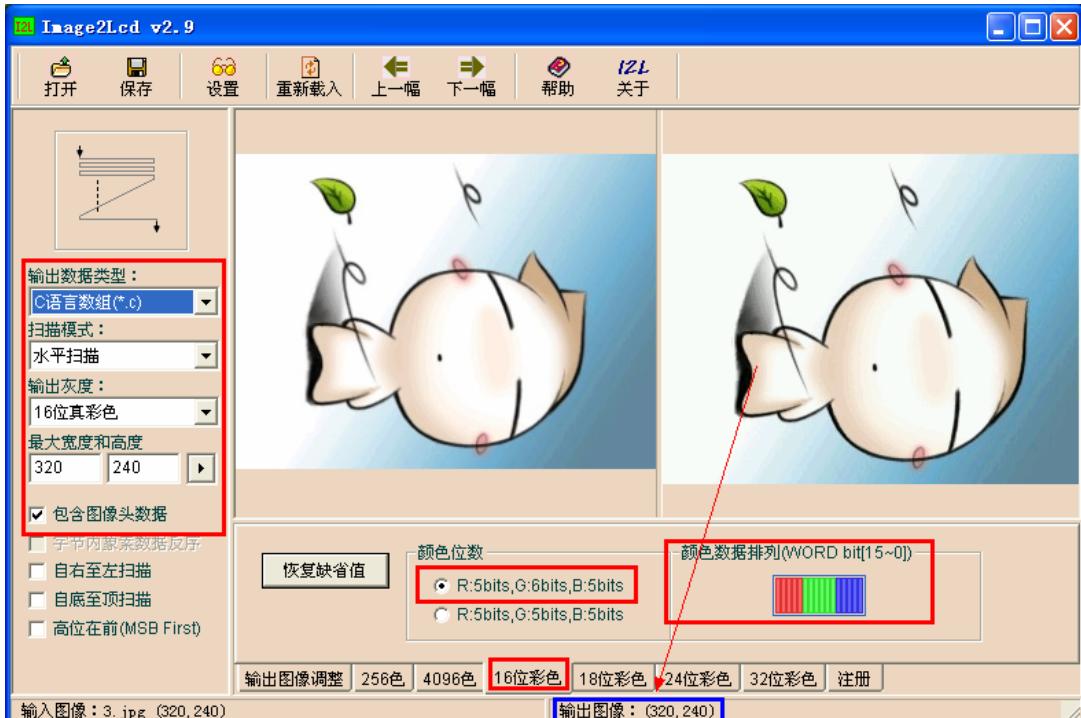


该软件的使用方法如下：

- 1) 双击“Image2Lcd 2.9.exe”，打开软件。
 - 2) 单击“打开”选项，选择要转换的图片。



3) 对转换的图片进行设置。具体的设置入下图：



我们这里是C语言，输出的数据类型选择“C语言数组 (*.C)”。这里大家要留意的是“输出图像”。我们屏幕最大的宽度和高度是320和240。假如输出的图像刚好是(320,240)，在代码中设置图像显示位置的时候应该是在开始的位置，即坐标(0, 0)。如果选择其他位置，图片的显示错乱。

扫描模式，我们选择水平扫描。

“输出灰度”这里，我们选择的是“16位真彩色”。本实验中针对的是16位彩色，颜色数据的排序是红、绿、蓝。颜色的位数分别是红色5位，绿色6位，蓝色5位。

勾选“包含图像头数据”。我们转换出来的数据，并非都是用于绘制的图像实际数据，它还包含了图像头数据。

将数组拷贝出来，替换 QQ_LCON_80X80.C 文件中的“gImage_QQ_ICON_80X80_16[153608]”数组内容和 QQ_LCON_80X80.h 文件中对应的数组定义，重新编译后即可显示用户自己的图片。

7.44.4 硬件设计

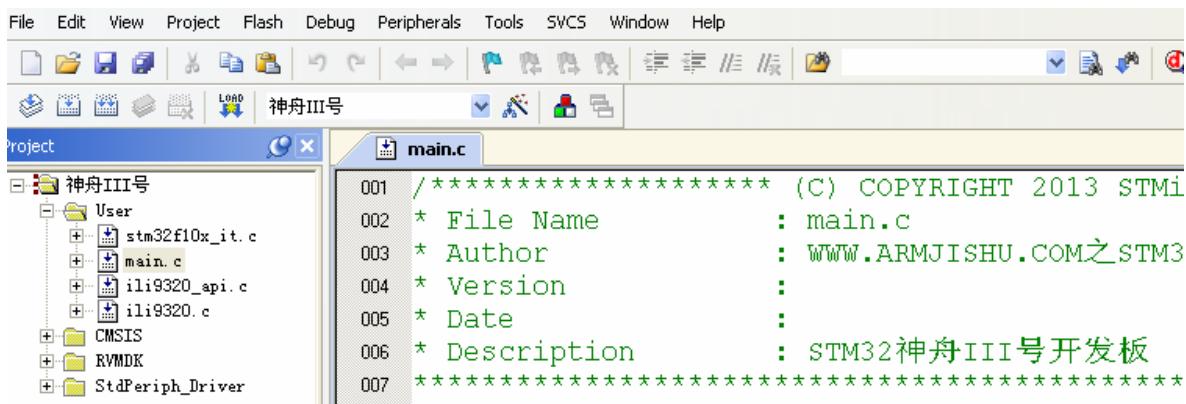
硬件设计同上

7.44.5 软件分析

进入例程的文件夹，然后打开\Project\Project.uvproj文件

嵌入式专业技术论坛 (www.armjishu.com) 出品

第 642 页，共 900 页



进入到 main () 函数中：

```

int main(void)
{
    Printf_Init();
    printf("\n\r\n ");
    printf("\n\r---- 神舟III号 ----- \n\r ");

    ili9320_Initialization();

    while (1)
    {
        LCD_Image2LcdDrawBmp565Pic(0, 0, gImage_C );
    }
}

```

代码分析 1：液晶屏的初始化等细节前面的例程都有详细介绍，这个例程中主要是 LCD_Image2LcdDrawBmp565Pic() 函数显示了图片。

代码分析 2：LCD_Image2LcdDrawBmp565Pic(0, 0, gImage_C)

```

void LCD_Image2LcdDrawBmp565Pic(uint16_t startX, uint16_t startY, const uint8_t *BmpAddress)
{
    HEADCOLOR * BmpHeadr;

    uint16_t Width, Height;
    uint8_t *BmpData;

    BmpHeadr = (HEADCOLOR *) BmpAddress;
    Width = BmpHeadr->w;
    Height = BmpHeadr->h;
    BmpData = (uint8_t *) (BmpAddress + sizeof(HEADCOLOR));
    if ((BmpHeadr->is565 == 1) && (BmpHeadr->gray == 16))
    {
        LCD_DrawBmp565Pic(StartX, StartY, Width, Height, (const uint16_t *) BmpData);
    }
    else
    {
        LCD_DEBUG_PRINTF("\n\r LCD_Image2LcdDrawBmp565Pic: Not Image2Lcd or Bmp565 format.");
    }
}

```

本函数有 3 个参数，第一、第二个参数设置图片显示的位置。第三个参数是指针，指向我们转换出来的数组数据的首地址。

代码分析 3：画图函数中首先定义了 HEADCOLOR 类型的“* BmpHeadr”。我们看一下 HEADCOLOR 是什么。

```
typedef struct _HEADCOLOR
{
    unsigned char scan;
    unsigned char gray;
    unsigned short w;
    unsigned short h;
    unsigned char is565;
    unsigned char rgb;
}HEADCOLOR;
```

大家可以发现，HEADCOLOR 里面分别包含了多个数据。有 4 个 unsigned char 类型，两个 unsigned short 类型。我们算一下它们总共占用了多少字节。unsigned char 占用一个字节，unsigned short 占用两个字节。那么它们一共就是占用了 8 个字节。

这 8 个字节就是，我们转换的图片的头数据。它们的意义分别为：scan，扫描模式。gray，灰度值。w，h 分别是图像的宽度和高度。is565，16 位彩色。Rgb，描述 R G B 颜色分量的排列顺序。

代码分析 4：定义完 “* BmpHeadr” 后，将指针 BmpHeadr 指向我们转换出来的图片数组数据，读取图片的宽度（Width）和高度（Height）。并跳过图像的头数据，将指针 BmpData 指向显示图片的实际数据。

```
BmpHeadr = (HEADCOLOR *) BmpAddress;
Width = BmpHeadr->w;
Height = BmpHeadr->h;
BmpData = (uint8_t *) (BmpAddress + sizeof(HEADCOLOR));
```

代码分析 5：通过对图像头数据的判断，调用 LCD_DrawBmp565Pic() 函数显示图片。

```
if ((BmpHeadr->is565 == 1) && (BmpHeadr->gray == 16))
{
    LCD_DrawBmp565Pic(StartX, StartY, Width, Height, (const uint16_t *) BmpData);
}
else
{
    LCD_DEBUG_PRINTF("\n\r LCD_Image2LcdDrawBmp565Pic: Not Image2Lcd or Bmp565 format.");
}
```

代码分析 6：图片显示函数 LCD_DrawBmp565Pic()。

```
void LCD_DrawBmp565Pic(uint16_t StartX, uint16_t StartY,
{
    uint32_t total;
    uint32_t i, j;
    uint32_t pointor;
    uint16_t line;

    //LCD_DEBUG_PRINTF("LCD_DrawBmp565Picture StartX %d \
    //                                , StartX, StartY, Width, Height);

    line=StartX;
#if 0 // 这种方法绘制速度快，但是寄存器因为LCD驱动芯片的限制
    total = Width * Height;
    LCD_WriteReg(0x0044, 0xEF00 + StartX); //Specify the start address
    LCD_WriteReg(0x0045, StartY); //Specify the start address
    LCD_WriteReg(0x0046, StartY + Width - 1 ); //Specify the end address
    LCD_SetCursor(line, StartY);
    LCD_WriteRAM_Prep
```

LCD_DrawBmp565Pic(uint16_t StartX, uint16_t StartY, uint16_t Width, uint16_t Height, const uint16_t *BmpAddress) 函数一共包含 5 个参数。第一、第二个参数是显示图像的起始点，本实验我们给的点是 (0,0)。第三、第四个参数分别表示图像的宽度和高度。第 5 个参数就是指向跳过图像头数据后的第一个数据，即我们用于绘制图画的第一个数据。

代码分析 7：绘制图片，也是一个点一个点的画。只是速度太快我们人眼无法看出。

```
LCD_WriteReg(0x0044,0xEF00);           //Specify the start/e
LCD_WriteReg(0x0045,0x0000);           //Specify the start p
LCD_WriteReg(0x0046,0x013F);           //Specify the end pos.
se
pointor = 0;
for (i=0;i<Height;i++)
{
    LCD_SetCursor(line, StartY);
    LCD_WriteRAM_Prep(); /* Prepare to write GRAM */
    for (j=0;j<Width;j++)
    {
        LCD_WriteRAM(BmpAddress[pointor]);
        pointor++;
    }
    line++;
}
```

显示图片的时候，首先设定显示的范围，这里用到了 3 个寄存器 0x0044、0x0045、0x0046。我们算一下给 0x0044 和 0x0046 的数据，0xEF00 和 0x013F。我们将十六进制 0xEF 化为十进制得 239。0x13F 化为十进制是 319。相信大家对这两个数据非常的熟悉。对 0x0044、0x0045、0x0046 的介绍大家参考控制芯片的手册。

显示的时候，设置显示时的位置。写入命令 0x22，准备写入 GRAM。然后调用 LCD_WriteRAM() 函数写数据。本实验中，可以通过两个 for 循环，先显示完一个宽度（Width），高度（Height）加 1，依此类推，绘制完整幅图片。

7.44.6 下载与测试.

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.44.7 实验现象

将固件下载到神舟 III 号开发板后，复位，神舟 III 号开发板正常情况下将显示一张图片。



7.45 TFT 彩色液晶屏显示图片并刷屏【未更新】

7.45.1 硬件设计

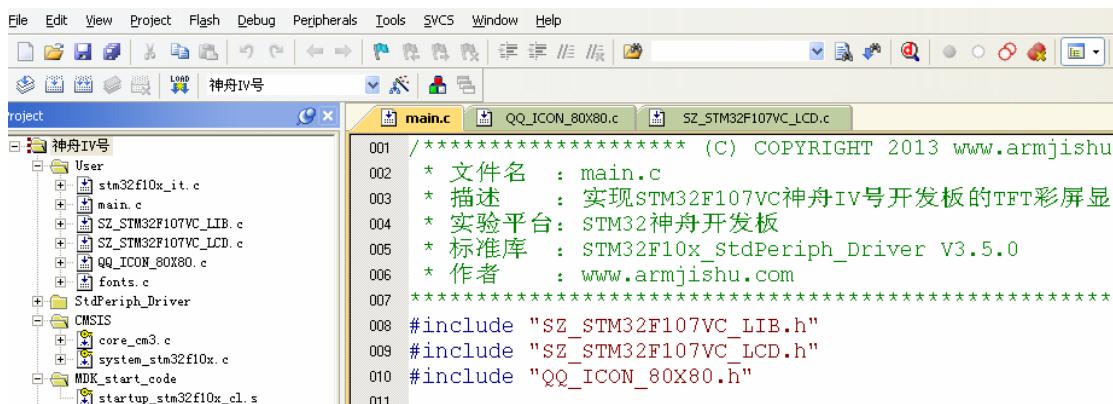
硬件设计同上。

7.45.2 软件设计

进入例程的文件夹，然后打开 Project\Project.uvproj 文件

嵌入式专业技术论坛 (www.armjishu.com) 出品

第 645 页，共 900 页



可以看到工程已经被打开，下面开始具体分析程序代码：

```

int main(void)
{
    /*!< 在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()
       所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为
       SystemInit()函数的实现位于system_stm32f10x.c文件中。
    */
    uint32_t x = 0, y = 0;
    HEADCOLOR * BmpHeadr;

    /* 初始化板载LED指示灯 */
    SZ_STM32_LEDInit(LED1);
    SZ_STM32_LEDInit(LED2);
    SZ_STM32_LEDInit(LED3);
    SZ_STM32_LEDInit(LED4);

    /* 注意串口2使用Printf时"SZ_STM32F107VC_LIB.c"文件中fputc定义中设
     * 串口2初始化 */
    SZ_STM32_COMInit(COM2, 115200);

    /* 初始化系统定时器SysTick,每秒中断1000次 */
    SZ_STM32_SysTickInit(1000);

    /* TFT-LCD初始化 */
    SZ_STM32_LCDInit();
    /* 延迟,间隔 */

    while(1)
    {
        LCD_Clear(LCD_COLOR_GREEN);
        delay(16000000);
        LCD_Image2LcdDrawBmp565Pic(80, 120, gImage_QQ_ICON_80X80_16);
    }
}

```

本实验和显示图片实验原理是一样的，不同的是添加的刷屏的功能。在 while (1) 循环中，先将屏刷成绿色，然后 LCD_Image2LcdDrawBmp565Pic () 函数将 QQ 图片显示屏的中央。再进入一个 while () 循环，整并显示 QQ 图标。

7.45.3 下载与测试

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 如何通过 MDK 编译和在线调试 小节进行操作

7.45.4 实验现象

将固件下载到神舟III号后，复位，神舟III号正常情况下将显示图片，不同的图片不停的切换。



7.46 开机TFT液晶屏图片自动左右移动实验

7.46.1 简要分析

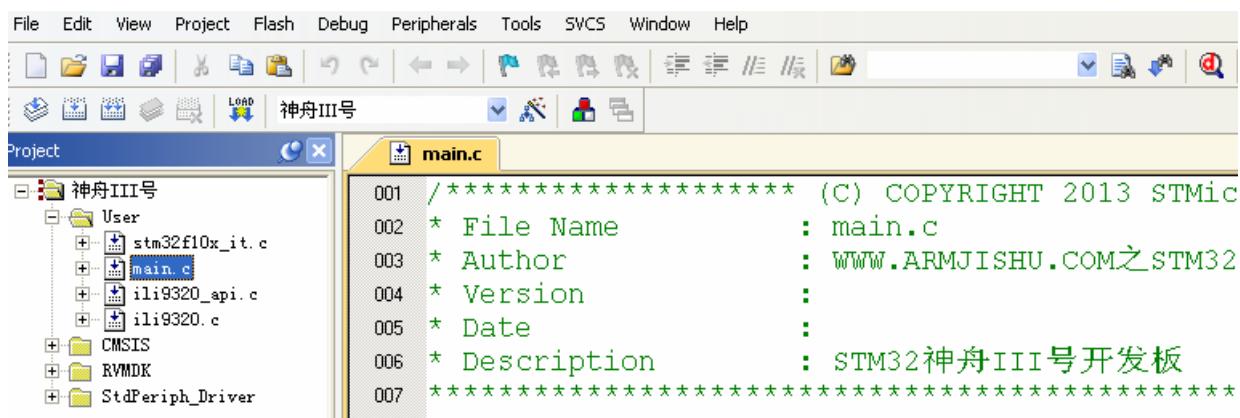
我们已经知道如何显示图片，本实验开机显示图片的时候，我们让图片左右移动。

7.46.2 硬件设计

硬件设计同上

7.46.3 软件分析

进入例程的文件夹，然后打开\Project\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

代码分析 1：本实验通过 改变画图函数 LCD_Image2LcdDrawBmp565Pic () 的第二个参数，使图片左右移动。显示图片的时候，行坐标不变，列坐标左右移动，从而实现图片的左右移动。

```

while (1)
{
    for (i=0;i<320;i++)
    {
        LCD_Image2LcdDrawBmp565Pic(0, i, gImage_C);
        delay(1);
    }
    for (i=320;i>0;i--)
    {
        LCD_Image2LcdDrawBmp565Pic(0, i, gImage_C);
        delay(1);
    }
}

```

代码分析 2：改变一个值，就使得图片先往一边移动，然后又往另一边移动。

7.46.4 下载与测试

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.46.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将看到下面的图片左右移动。



7.47 TFT触摸屏实验

在上一节的TFT彩屏显示实验中，我们对于TFT LCD屏显示已经有了一定程度的掌握。这节我们将在TFT彩屏显示实验的基础上，加上TFT LCD屏的触摸功能，将触摸采样到的数据在LCD屏上进行显示，主要是借助SPI1总线实现对触摸芯片ADS7843的控制。

7.47.1 实验的意义与作用

触摸屏逐渐取代键盘成为通信常用的人机交互工具，手机支持触摸功能、PDA手持设备等等的运用。本节实验，我们将针对LCD的触摸功能进行详解，剖析触摸采样到LCD屏显示间的处理过程。

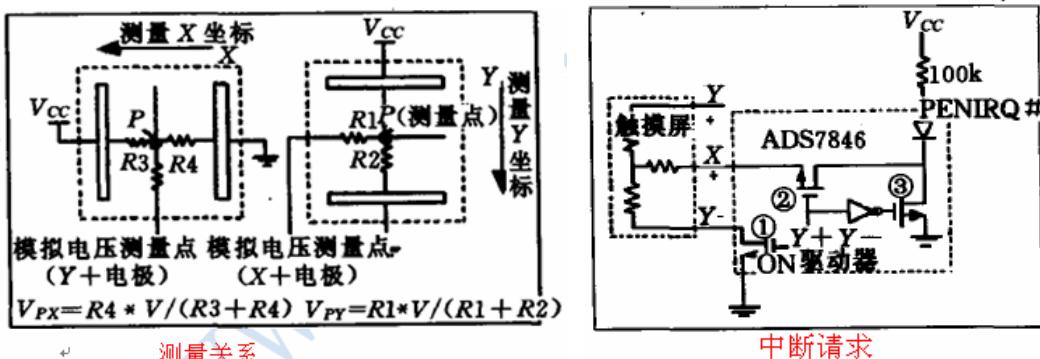
7.47.2 实验原理

触摸屏一般分为电阻、电容、表面声波、红外线扫描和矢量压力传感等，其中使用最多的是四线或无线电阻触摸屏。四线电阻触摸屏是由两个透明电阻膜构成的，在它的水平和垂直电阻网上施加电压，就可以通过A/D转换面板在触摸点测量出电压，从而对应出坐标值。

神舟I号的触摸屏附在LCD屏的表面上，与LCD屏相配合使用，主要是用的触摸芯片是ADS7843，业界上与ADS7843芯片相兼容的触摸芯片还有ADS7846、AK4182、XPT2046以及TSC2046等，驱动基本上一致。

ADS7843是一款4线式触摸屏控制器，内含12位分辨率，125KHz转换速率，逐步逼近型的A/D转换器。

下面我们将通过ADS7843芯片讲解触摸原理。ADS7843内部有一个由多个模拟开关组成的供电测量电路网络和12位的A/D转换器。其可以根据处理器（stm32f103RTB6通过SPI总线）发来的不同测试命令导通不同的模拟开关，以便向工作面电极对提供电压，并把相应测量电极上的触点坐标位置所对应的电压模拟量引入到A/D转换器。在触摸点X、Y坐标的测试过程中，测试电压与测量点的等效电路如下图所示：（P为测量点）

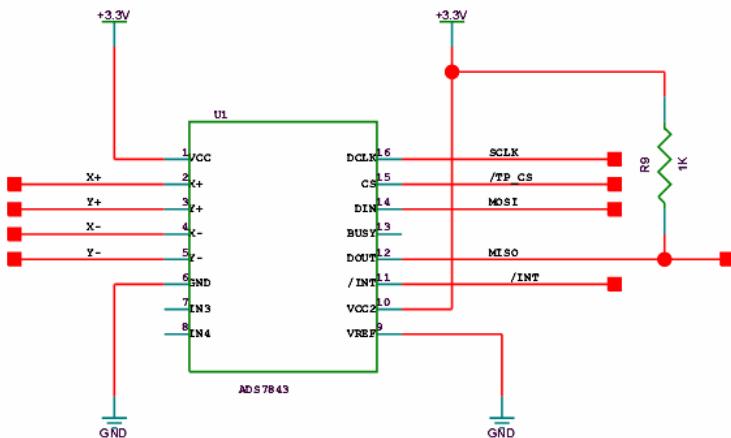


当触摸屏受到点击或是挤压的时，ADS7843通过中断请求通知处理器（STM32F103RBT6）有触摸发生。如“中断请求”图所示，当没有触摸时，MOSFET①和②打开、③关闭，则中断输出引脚通过外加的上拉电阻输出为高，当有触摸时，①和③打开，②关闭，则中断输出引脚通过③内部的连接到地，输出为低，从而向处理器发出中断请求。

7.47.3 硬件设计

TFTLCD 模块自带的触摸屏控制芯片为 XPT2046。XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置，除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。XPT2046 采用微小的封装形式：TSSOP-16,QFN-16(0.75mm 厚度)和VFBGA-48。工作温度范围为-40°C~+85°C。

触摸芯片现位于 TFT LCD 屏上，处理器通过 SPI2 总线控制触摸芯片。X+、X-、Y+和 Y-则连接到 LCD 触摸屏上。



7.47.4 软件设计

打开神舟 III 号触摸屏例程。以下我们学习代码的设计，主要是与 ADS7843 相关的代码。至于 TFT LCD 屏相关的显示代码在上节已经讲解过，在此不重复说明。同时，STM32F103 对 ADS7843 通过 GPIO 模拟 SPI 接口的方式进行访问。

首先，ADS7843芯片的初始化函数：

```
void ADS7843_Init(void)           //触摸芯片初始化
{
    ADS7843_CS_config();          //使能LCD
    ADS7843_CS_HIGH();            //关闭LCD
    SPI1_Config();                //配置SPI1
    SPI1_Init_For_Byte();
    SPI1_MOSI_HIGH();
    SPI1_SCK_LOW();
    ADS7843_INT_config();        //中断的配置
    ADS7843_INT_EXIT_Init();
    ADS7843_InterruptConfig();
}
```

其中ADS7843的片选CS以及中断管脚的配置如下所示：

```
void ADS7843_CS_config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    /* Enable GPIOB, GPIOC and AFIO clock */
    RCC_APB2PeriphClockCmd(RCC_ADS7843_CS , ENABLE); //RCC_APB2Periph_AFIO

    /* pins configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_ADS7843_CS;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIO_ADS7843_CS_PORT, &GPIO_InitStructure);
}
```

ADS7843芯片的中断管脚的声明

```

static void ADS7843_INT_config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    /* Enable GPIOB, GPIOC and AFIO clock */
    RCC_APB2PeriphClockCmd(RCC_ADS7843_INT , ENABLE); //RCC_APB2Periph_AFIO

    /* LEDs pins configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_ADS7843_INT;
    //GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIO_ADS7843_INT_PORT, &GPIO_InitStructure);
}

```

EXTI中断线上的中断映射关系

```

static void ADS7843_IRQHandler(void) //EXTI中断线上的中断映射
{
    NVIC_InitTypeDef NVIC_InitStructure;
    /* Set the Vector Table base address at 0x08000000 */
    NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0000);
    /* Configure the Priority Group to 2 bits */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

    /* Enable the EXTI15_10_IRQn Interrupt */
    NVIC_InitStructure.NVIC IRQChannel = GPIO_ADS7843_EXTI_IRQn; //处理器的中断配置, EXTI15_10_IRQn
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

详细的中断说明，可参考文档[《【中文】STM32F系列ARM内核32位高性能微控制器参考手册V10.1.pdf》](#)中第137页的资料。

芯片以及中断管脚的宏定义：

| | |
|---------------------------------------|----------------------|
| #define RCC_ADS7843_CS | RCC_APB2Periph_GPIOB |
| #define GPIO_ADS7843_CS_PORT | GPIOB |
| #define GPIO_ADS7843_CS | GPIO_Pin_12 |
| | |
| #define RCC_ADS7843_INT | RCC_APB2Periph_GPIOG |
| #define GPIO_ADS7843_INT_PORT | GPIOG |
| #define GPIO_ADS7843_INT | GPIO_Pin_7 |
| #define GPIO_ADS7843_EXTI_LINE | EXTI_Line7 |
| #define GPIO_ADS7843_EXTI_PORT_SOURCE | GPIO_PortSourceGPIOG |
| #define GPIO_ADS7843_EXTI_PIN_SOURCE | GPIO_PinSource7 |
| #define GPIO_ADS7843_EXTI_IRQn | EXTI9_5_IRQn |

接下来，我们看看中断的处理函数，打开stm32f10x_it.c文件，找到“[EXTI9_5_IRQHandler](#)”函数：

```
void EXTI9_5_IRQHandler(void) /* TouchScreen */
{
    if(EXTI_GetITStatus(EXTI_Line7) != RESET)
    {
        //printf("\n\r tp");
        ARMJISHU_TouchScreen_ADS7843();

        /* Clear the EXTI Line 5 */
        EXTI_ClearITPendingBit(EXTI_Line7);
    }
}
```

处理器接收到PG7管脚的中断请求后，响应中断，处理ARMJISHU_TouchScreen_ADS7843（）函数，此函数的处理程序如下所示：

```
void ARMJISHU_TouchScreen_ADS7843(void)
{
    u16 xdata, ydata;
    u32 xScreen, yScreen;
    static u16 sDataX, sDataY;

    ADS7843_Rd_Addata(&xdata, &ydata); //读取采样数据
    xScreen = _AD2X(ydata);
    yScreen = _AD2Y(xdata);

    if((xScreen>1) && (yScreen>1) && (xScreen<320-1) && (yScreen<240-1))
    {
        printf("\n\r%d,%d", xScreen, yScreen);
        if((GPIO_ADS7843_INT_VALID) && distence(sDataX, xScreen) && distence(sDataY, yScreen))
        {
            LCD_BIG_POINT(320-xScreen, yScreen); //采样数据显示
        }
        sDataX = xScreen;
        sDataY = yScreen;
    }
}
```

处理器响应中断后，读取采样数据，并将采样数据进行转换，得到显示的坐标，并将结果进行显示。那么，读取采样数据过程的实现，则是通过4次采样，确定采样结果，进行A/D转换。下面了解读取采样数据的函数。

```
#define times 4
static void ADS7843_Rd_Addata(u16 *X_Addata,u16 *Y_Addata) //触摸读取数据
{
    u16 i,j,k,x_addata[times],y_addata[times];
    for(i=0;i<times;i++) //采样4次.
    {
        ADS7843_SPI_Start();
        ADS7843_WrCmd( CHX );
        y_addata[i] = ADS7843_Read();
        ADS7843_CS_HIGH();

        ADS7843_SPI_Start();
        ADS7843_WrCmd( CHY );
        x_addata[i] = ADS7843_Read();
        ADS7843_CS_HIGH();
    }
    for(i=0;i<times;i++)
    {
        for(j=times;j<times-1;j++)
        {
            if(x_addata[j] > x_addata[i])
            {
                k = x_addata[j];
                x_addata[i] = x_addata[j];
                x_addata[j] = k;
            }
        }
    }
    for(i=0;i<times;i++)
    {
        for(j=times;j<times-1;j++)
        {
            if(y_addata[j] > y_addata[i])
            {
                k = y_addata[j];
                y_addata[i] = y_addata[j];
                y_addata[j] = k;
            }
        }
    }
    *X_Addata=(x_addata[1] + x_addata[2]) >> 1;
    *Y_Addata=(y_addata[1] + y_addata[2]) >> 1;
}
```

下面我们看看主程序的循环处理是如何实现的：

```
/* Main loop */
while (1)
{
    DrawPicture_Center((u16 *)picture);
    ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);
    ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);
    Delay_ARMJISHU(10000000);

    ili9320_ColorScreen();
    ili9320_PutStr_16x24_Center(108, c, len,White, HyalineBackColor);
    Delay_ARMJISHU(10000000);

    ili9320_GreyScreen();
    ili9320_PutStr_16x24_Center(108, c, len,White, HyalineBackColor);
    Delay_ARMJISHU(4000000);
}
```

其中值得一提的有两点：一、延时函数Delay_ARMJISHU(); 二、则是DrawPicture_Center();

1、延时函数Delay_ARMJISHU():

```
static void Delay_ARMJISHU(__IO uint32_t nCount)
{
    for (; nCount != 0; nCount--)
    {
        if(GPIO_ADS7843_INT_VALID)
        {
            ARMJISHU_TouchScreen_ADS7843();
        }
    }
}
```

在延时函数中，对中断管脚的判断，如果有效时，则在延时的过程对触摸进行响应。当然，在延迟时，如果触摸了，也可以采用中断进行响应，但是，这样在图片的刷新过程中可能会导致图片的混乱。

2、绘制图片的函数DrawPicture_Center():

```
void DrawPicture_Center(u16 *PictureAddr)           //绘制图形函数
{
    PictureWidth = (picture[0x13] << 8) | picture[0x12];      //图形的宽度
    PictureHeight = (picture[0x17] << 8) | picture[0x16];     //图形的高度          控制图形在LCD屏上的大小

    printf("\n\r PictureWidth is %d  0x%X ", PictureWidth, PictureWidth);
    printf("\n\r PictureHeight is %d  0x%X ", PictureHeight, PictureHeight);

    ili9320_Clear(Blue);
    ili9320_DrawPicture(0, (240-PictureWidth+1)/2, 320-1, ((240+PictureWidth+1)/2)-1, (u16 *) (picture + BmpHeadSize));
}
```

通过此函数可以将自己喜欢的图片，通过转换放在picture.h文件后，进行显示。

7.47.5 下载与现象

现象：将固件下载到神舟III号后，复位，LCD屏就可以写字，我们写上“神舟III号”字样，如下图所示：



7.48 图片跟触摸点移动实验

7.48.1 简要分析

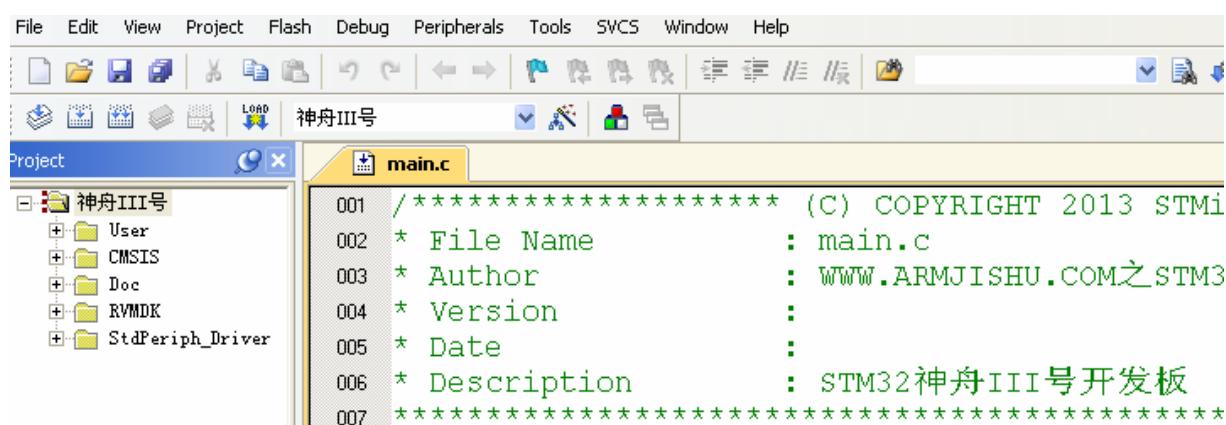
我们已经知道如何显示图片，本实验我们先显示图片，然后让图片跟随触摸点移动。

7.48.2 硬件设计

硬件设计同上

7.48.3 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

代码分析 1：本实验我们这里添加了触摸的功能。通过函数 ARMJISHU_TouchScreen_ADS7843()，读取触摸点的信息。然后通过触摸信息，调用 LCD_Image2LcdDrawBmp565Pic(0, TSC_Value_X, gImage_C) 函数显示图片，从而实现图片的移动。

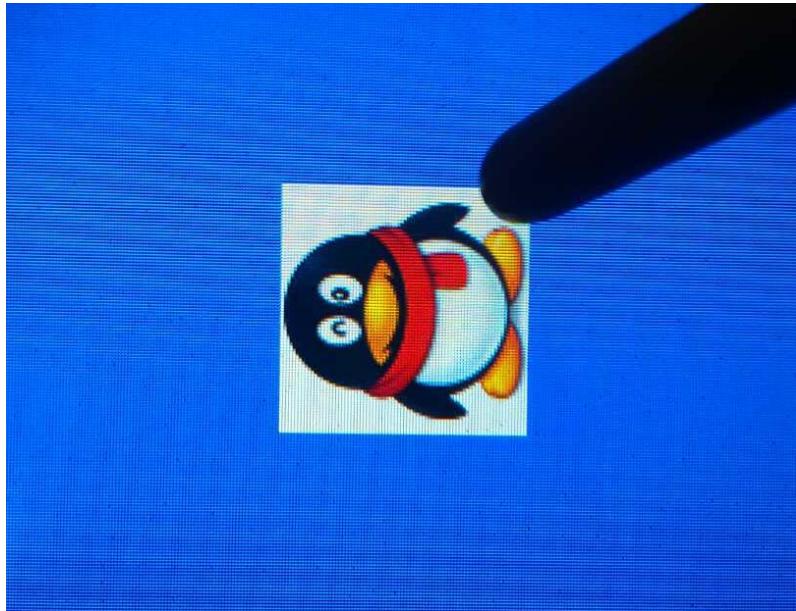
```
/* Main loop */
while (1)
{
    ARMJISHU_TouchScreen_ADS7843();
    LCD_Image2LcdDrawBmp565Pic(TSC_Value_Y, TSC_Value_X, gImage_QQ);
}
```

7.48.4 下载与测试

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 如何通过 MDK 编译和在线调试 小节进行操作

7.48.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将看到图片跟随触摸点移动。



7.49 点击图片小图标显示小图标外框实验

7.49.1 简要分析

本实验中，我们点击图片中的小图标的时候，小图标变颜色

7.49.2 硬件设计

硬件设计同上

7.49.3 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件

The screenshot shows the MDK-ARM IDE interface. The Project Explorer on the left lists a project named "神舟III号" containing folders for User, CMSIS, Doc, RVMDK, and StdPeriph_Driver. The main window displays the content of the main.c file:

```
001 / ***** (C) COPYRIGHT 2013 STMic
002 * File Name      : main.c
003 * Author         : WWW.ARMJISHU.COM之STM32
004 * Version        :
005 * Date           :
006 * Description    : STM32神舟III号开发板
007 *****
```

下面开始具体分析程序代码：

代码分析 1：ili9320_Initialization()，彩屏初始化函数。前面我们已经讲了，我们就不再重复，这里它的主要是将彩屏初始化，并将彩屏刷成绿色。

代码分析 2：ADS7843_Init()函数，初始化触摸功能。大家可以参考触摸屏的章节。这里就不重复。

代码分析 3：点击小图标后，显示小图标。

```
/* Main loop */
while (1)
{
    ARMJISHU_TouchScreen_ADS7843();
    if(TSC_Value_X<(125+69)&&TSC_Value_X>125&&TSC_Value_Y<(85+60)&&TSC_Value_Y>85)
    {
        LCD_Image2LcdDrawBmp565Pic(93, 125, gImage_SmallIcon);
        ili9320_Delay(100);
    }
}
```

我们首先，显示一个全屏的程序。通过函数 ARMJISHU_TouchScreen_ADS7843()读取触摸点的信息。通过 for 语句判断触摸点是否在所限定的范围内，如果在的话，在原来图片上显示一个预先做好的小图片，将原来图片中的小图标覆盖，从而实现显示小图标外框功能。

7.49.4 下载与测试

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 如何通过 MDK 编译和在线调试 小节进行操作。

7.49.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将正常显示如下图片，我们点击图片中央的小图标，小图标变色。



7.50 点击图片小图标触发事件实验

7.50.1 简要分析

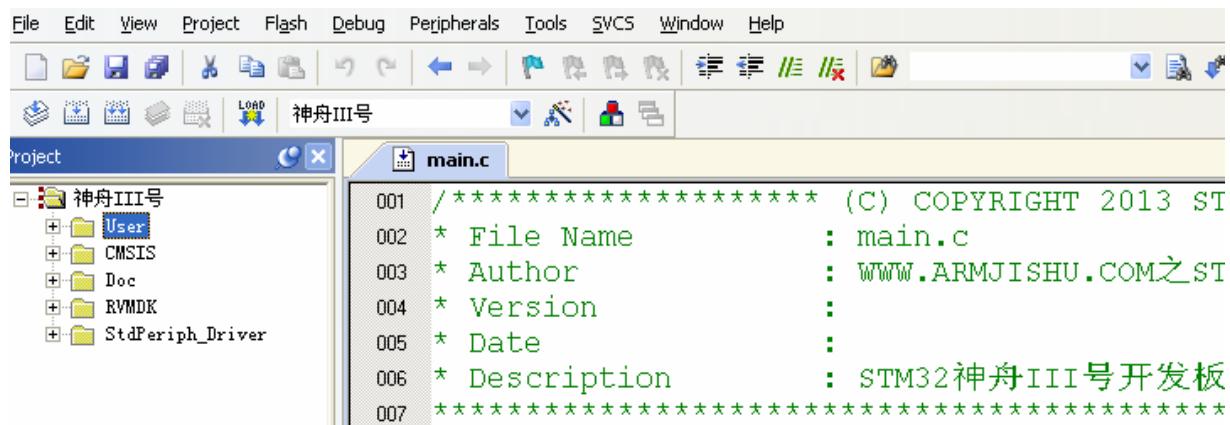
本实验中，我们点击图片中间的小图标的时候，刷屏程序和显示字符程序得予运行。

7.50.2 硬件设计

硬件设计同上

7.50.3 软件分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



下面开始具体分析程序代码：

代码分析 1：本实验，初始化触摸屏，调用函数 LCD_Image2LcdDrawBmp565Pic(0, 0, gImage_image)，显示图片。

代码分析 2：点击小图标后，触发显示事件。

```
/* Main loop */
while (1)
{
    ARMJISHU_TouchScreen_ADS7843();
    if(TSC_Value_X<(125+69)&&TSC_Value_X>125&&TSC_Value_Y<(85+60)&&TSC_Value_Y>85)
    {
        LCD_Image2LcdDrawBmp565Pic(93, 125, gImage_SmallIcon);
        ili9320_Delay(100);
        ili9320_Clear(Black);
        while(1);
    }
}
```

我们首先，显示一个有 9 个图标的图片。通过函数 ARMJISHU_TouchScreen_ADS7843() 读取触摸点的信息。通过 if 语句判断触摸点是否在所限定的范围内，如果在的话，执行触发事件将屏刷成黑色。大家可以在此添加自己的触发事件。

7.50.4 下载与测试

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 [如何通过 MDK 编译和在线调试](#) 小节进行操作

7.50.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将正常显示如下图片，我们点击图片中央的小图标，小图标变色。随后屏幕变成黑色。



7.51 双击小图标触发实验

7.51.1 简要分析

本实验中，我们点击两次图片中间的小图标的时候，刷屏程序和显示字符程序得予运行

7.51.2 硬件设计

硬件设计同上

7.51.3 软件分析

进入例程的文件夹，然后打开Project\Project.uvproj 文件

The screenshot shows a software development environment with a toolbar at the top and a project tree on the left. The project tree is titled '神舟IV号' and contains several subfolders and files under 'User'. The 'main.c' file is open in the editor, displaying the following code:

```
01 /***** (C) COPYRIGHT 2013 www.armjishu.com *****
02 * 文件名 : main.c
03 * 描述 : 实现STM32F107VC神舟IV号开发板的TFT彩屏屏幕上小图标点击两次触发实验
04 * 实验平台: STM32神舟开发板
05 * 标准库 : STM32F10x_StdPeriph_Driver V3.5.0
06 * 作者 : www.armjishu.com
07 *****
08 #include "SZ_STM32F107VC_LIB.h"
09 #include "SZ_STM32F107VC_LCD.h"
10 #include "SZ_cursor.h"
11 #include "SZ_TouchScreen.h"
12 #include "QQ_ICON_80X80.h"
```

下面开始具体分析程序代码：

代码分析 1：本实验，初始化触摸屏，调用函数 LCD_Image2LcdDrawBmp565Pic(0, 0, gImage_image)，显示图片。

代码分析 2：点击两次小图标后，触发显示事件。

```
while (1)
{
    while(1)
    {
        ARMJISHU_TouchScreen_ADS7843();
        //在此添加用户的触发事件
        if(TSC_Value_X<(125+69)&&TSC_Value_X>125&&TSC_Value_Y<(85+60)&&TSC_Value_Y>85)
        {
            LCD_Image2LcdDrawBmp565Pic(93, 125, gImage_SmallIcon);
            Delay(10);
            i=TSC_Value_X;
            break;
        }
    }
    Delay(100);
    while(1)
    {
        ARMJISHU_TouchScreen_ADS7843();
        if(TSC_Value_X != i)
        {
            //在此添加用户的触发事件
            if(TSC_Value_X<(125+69)&&TSC_Value_X>125&&TSC_Value_Y<(85+60)&&TSC_Value_Y>85)
            {
                LCD_Image2LcdDrawBmp565Pic(93, 125, gImage_SmallIcon);
                Delay(10);
                ili9320_Clear(Black);
                while(1);
            }
        }
    }
}
```

我们首先，显示一个有 9 个图标的图片。再通过两次读取触摸信息，实现点击两次图标，触发刷屏程序和显示字符串程序。

while (1) 大循环里面有两个 while 循环。大循环中程序进入第一个 while 循环。判断触摸点的位置，如果是在中间的小图标的范围内的话，使小图标底色变色，并跳出本循环。而后，进入第二个 while 循环，等待下一个触摸的到来。如果又发生触摸，并且范围还在中间的小图标的范围内，则触发刷屏程序。

7.51.4 实验现象

将固件下载到神舟 III 号开发板后，复位，神州 III 号开发板正常情况下将正常显示如下图片，我们点击图片中央的小图标两次。随后屏幕变成黑色。

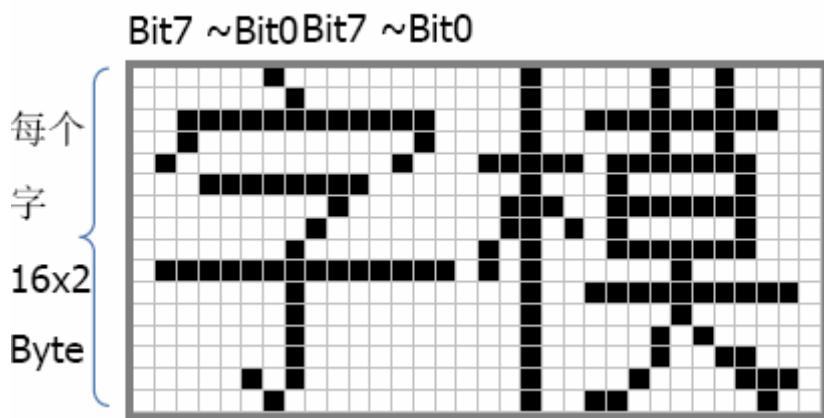


7.52 TFT彩色液晶屏显示英文字

7.52.1 什么是字模

液晶屏其实就是一个由许多像素点组成的点阵，若要在上面显示一个字符，则需要很多像素点共同构成，比如 8*16 的 ASCII 码字符，或者 16*16 的点阵显示的汉字。

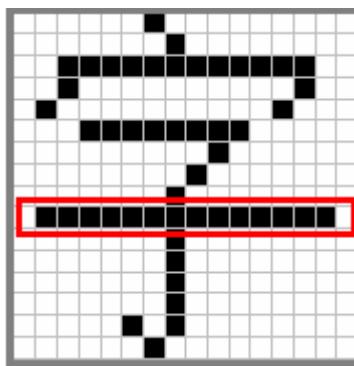
如果每个 8*16 或者 16*16 的点阵区域来显示一个字符，把黑色的像素点以 1 来表示，空白以 0 表示，每个像素点的状态以一个二进制位来记录，用 $8*16/8=16$ 个字节或者 $16*16/8=32$ 个字节就可以把这个字记录下来。这个 16 字节或者 32 字节就被称为该字符的字模，当然还有其他常用的字模是 24*24, 32*32 的。



可以把这个‘字模’两个字转化成以下的 32 个字节的字模：

```
0x02, 0x00, 0x01, 0x00, 0x3F, 0xFC, 0x20, 0x04, 0x40, 0x08, 0x1F, 0xE0, 0x00, 0x40,  
0x00, 0x80,  
0xFF, 0xFF, 0x7F, 0xFE, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x05, 0x00,  
0x02, 0x00,
```

比如看下，开头的 0x02 和 0x00，就是显示‘字’的最上面那一点，显示第一行；比如下面这一行黑体，可以数一下，从上面往下数是第 10 行，那么可以看到第 10 行是 0x7F 和 0xFE，确实是一个正确的字模显示；在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模：



7.52.2 ASCII 码的字符解释

下图是 ASCII 码字符表

ASCII 字符代码表 一

| 高四位 | | ASCII非打印控制字符 | | | | | | | | | | ASCII 打印字符 | | | | | | | | | | | |
|------|-----|--------------|---------------|----|------|-------|------|------|----|------|--------|------------|-----|------|-----|------|-----|------|-----|------|-----|----------------|---|
| | | 0000 | | | | | 0001 | | | | | 0010 | | 0011 | | 0100 | | 0101 | | 0110 | | 0111 | |
| | | 0 | | 1 | | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | | | | | |
| 低四位 | +逆制 | 字符 | ctrl | 代码 | 字符解释 | +逆制 | 字符 | ctrl | 代码 | 字符解释 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | |
| 0000 | 0 | 0 | BLANK NULL | ^@ | NUL | 空 | 16 | ▶ | ^P | DLE | 数据链路转意 | 32 | | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 0001 | 1 | 1 | ☺ | ^A | SOH | 头标开始 | 17 | ◀ | ^Q | DC1 | 设备控制 1 | 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 0010 | 2 | 2 | ☻ | ^B | STX | 正文开始 | 18 | ↑ | ^R | DC2 | 设备控制 2 | 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 0011 | 3 | 3 | ♥ | ^C | ETX | 正文结束 | 19 | !! | ^S | DC3 | 设备控制 3 | 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 0100 | 4 | 4 | ◆ | ^D | EOT | 传输结束 | 20 | ¶ | ^T | DC4 | 设备控制 4 | 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 0101 | 5 | 5 | ♣ | ^E | ENQ | 查询 | 21 | ƒ | ^U | NAK | 反确认 | 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 0110 | 6 | 6 | ♠ | ^F | ACK | 确认 | 22 | ■ | ^V | SYN | 同步空闲 | 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 0111 | 7 | 7 | ● | ^G | BEL | 震铃 | 23 | ↓ | ^W | ETB | 传输块结束 | 39 | ' | 55 | 7 | 71 | G | 87 | w | 103 | g | 119 | w |
| 1000 | 8 | 8 | ▣ | ^H | BS | 退格 | 24 | ↑ | ^X | CAN | 取消 | 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 1001 | 9 | 9 | ○ | ^I | TAB | 水平制表符 | 25 | ↓ | ^Y | EM | 媒体结束 | 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 1010 | A | 10 | ▣ | ^J | LF | 换行/新行 | 26 | → | ^Z | SUB | 替换 | 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 1011 | B | 11 | ♂ | ^K | VT | 竖直制表符 | 27 | ← | ^[| ESC | 转意 | 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 1100 | C | 12 | ♀ | ^L | FF | 换页/新页 | 28 | ﹍ | ^` | FS | 文件分隔符 | 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 1101 | D | 13 | ♪ | ^M | CR | 回车 | 29 | ↔ | ^] | GS | 组分隔符 | 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } |
| 1110 | E | 14 | ♫ | ^N | SO | 移出 | 30 | ▲ | ^6 | RS | 记录分隔符 | 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 1111 | F | 15 | ⌚ | ^o | SI | 移入 | 31 | ▼ | ^- | US | 单元分隔符 | 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | △ |
| | | | | | | | | | | | | | | | | | | | | | | ^Back space | |

现在将这些 ASCII 字符表转化成 8x16 的数组，也就是说一个字符的高度是 8，宽度是 16，16 也就是两个字节，那么一个字符就是 8 行，每行都是 16 个 bit，等于是 $8 \times 16 / 8 = 16$ 个字节大小。可以看到下面的 ASCII 字符，红框框住的是 16 个字节，显示一个字符。

```
unsigned char const ascii_8x16[1536] = {
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x18,0x3C,0x3C,0x18,0x18,0x18,0x18,0x00,0x00,0x00,0x00,0x00,
0x00,0x66,0x66,0x66,0x24,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x6C,0x6C,0xFE,0x6C,0x6C,0x6C,0x6C,0x6C,0x00,0x00,0x00,
0x18,0x18,0x7C,0xC6,0xC2,0xC0,0x7C,0x06,0x86,0xC6,0x7C,0x18,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0xC2,0xC6,0x0C,0x18,0x30,0x60,0xC6,0x86,0x00,0x00,0x00,
0x00,0x00,0x38,0x6C,0x38,0x76,0xDC,0xCC,0xCC,0x76,0x00,0x00,0x00,0x00,
0x00,0x30,0x30,0x30,0x60,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0C,0x18,0x30,0x30,0x30,0x30,0x30,0x18,0x0C,0x00,0x00,0x00,
0x00,0x00,0x30,0x18,0x0C,0x0C,0x0C,0x0C,0x0C,0x18,0x30,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x66,0x3C,0xFF,0x3C,0x66,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x18,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x06,0x0C,0x18,0x30,0x60,0xC0,0x80,0x00,0x00,0x00,
0x00,0x00,0x7C,0xC6,0xCE,0xD6,0xE6,0xC6,0x7C,0x00,0x00,0x00,0x00,
0x00,0x00,0x18,0x38,0x78,0x18,0x18,0x18,0x18,0x18,0x7E,0x00,0x00,0x00,0x00,
0x00,0x00,0x7C,0xC6,0x06,0x0C,0x18,0x30,0x60,0xC0,0xC6,0xFE,0x00,0x00,0x00,0x00,
0x00,0x00,0x7C,0xC6,0x06,0x06,0x3C,0x06,0x06,0x06,0x06,0x06,0x7C,0x00,0x00,0x00,0x00,
0x00,0x00,0x0C,0x1C,0x3C,0x6C,0xCC,0xFE,0x0C,0x0C,0x0C,0x1E,0x00,0x00,0x00,0x00,
0x00,0x00,0xFE,0xC0,0xC0,0x0C,0x0C,0x0E,0x06,0x06,0xC6,0x7C,0x00,0x00,0x00,0x00,
0x00,0x00,0x38,0x60,0xC0,0xC0,0xFC,0xC6,0xC6,0xC6,0x7C,0x00,0x00,0x00,0x00,
0x00,0x00,0xFE,0xC6,0x06,0x06,0x0C,0x18,0x30,0x30,0x30,0x00,0x00,0x00,0x00,
0x00,0x00,0x7C,0xC6,0xC6,0x7C,0xC6,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x7C,0xC6,0xC6,0x7E,0x06,0x06,0x06,0x0C,0x78,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x18,0x18,0x00,0x00,0x00,0x18,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x00,0x00,0x00,0x18,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x06,0x0C,0x18,0x30,0x60,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x7C,0xC6,0x0C,0x18,0x18,0x00,0x18,0x18,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x7C,0xC6,0xC6,0xDE,0xDE,0xDC,0xC0,0x7C,0x00,0x00,0x00,0x00,
0x00,0x00,0x10,0x38,0x6C,0xC6,0xC6,0xFE,0xC6,0xC6,0xC6,0x00,0x00,0x00,0x00,
0x00,0x00,0xFC,0x66,0x66,0x66,0x7C,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00,
0x00,0x00,0x3C,0x66,0xC2,0xC0,0xC0,0xC0,0xC2,0x66,0x3C,0x00,0x00,0x00,0x00,
0x00,0x00,0xF8,0x6C,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0xF8,0x00,0x00,0x00,0x00,
0x00,0x00,0xFE,0x66,0x62,0x68,0x68,0x68,0x60,0x62,0x66,0x6E,0x00,0x00,0x00,0x00,
0x00,0x00,0xFE,0x66,0x62,0x68,0x68,0x68,0x60,0x60,0x60,0x00,0x00,0x00,0x00}
```

可以看到整个数组 ascii_8x16[1536]总共有 1536 个成员，每个成员就是一个单独的字节，而 16 个字节就是一个字符，那么这个数组总共能描述多少个字符呢？总共是 $1536/16 = 96$ 个字符，从下面的这张表来看，这个 96 个字符就是描述的是从 ASCII 值 32 到 127 这 96 个字符。

| ASCII 打印字符 | | | | | | | | | | | |
|------------|----|------|----|------|----|------|----|------|----|------|-----------------|
| 0010 | | 0011 | | 0100 | | 0101 | | 0110 | | 0111 | |
| 2 | | 3 | | 4 | | 5 | | 6 | | 7 | |
| +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 | +逆制 | 字符 |
| 32 | | 48 | 0 | 64 | @ | 80 | P | 96 | ' | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | △ Back space |

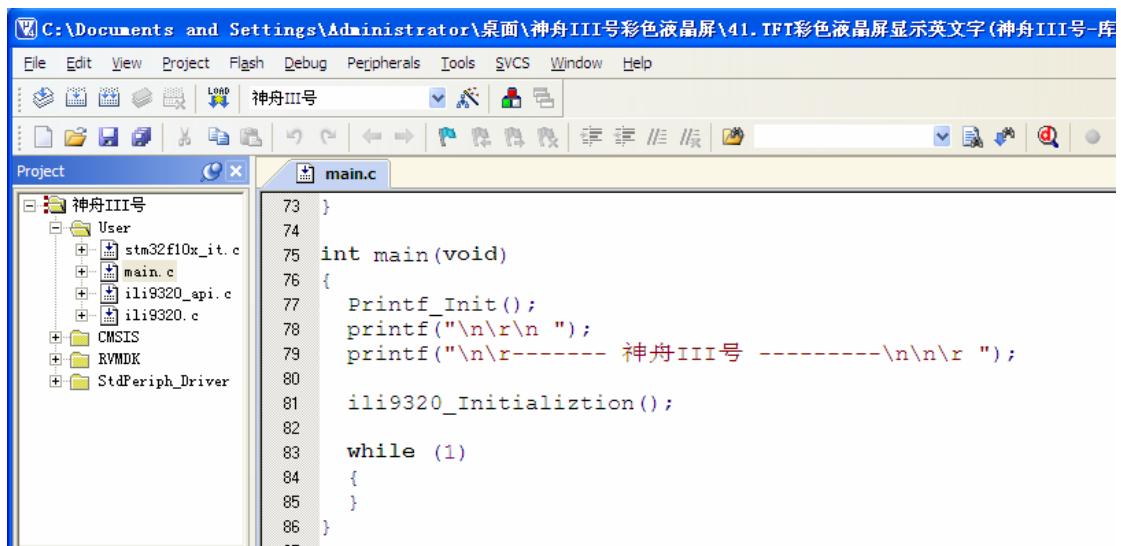
比如我们希望显示www.armjishu.com 到液晶屏上，那么这个LOGO所有的字符都能在ASCII码表中通过查表获得，接下来的软件分析会进一步分析如何用具体的软件代码来显示字符到LCD液晶屏上。

7.52.3 硬件设计

硬件设计同上

7.52.4 软件分析

进入例程的文件夹，然后打开\Project\MDK-ARM\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```

int main(void)
{
    Printf_Init();
    printf("\n\r\n ");
    printf("\n\r----- 神舟III号 ----- \n\n\r ");

    ili9320_Initialization();

    while (1)
    {
    }
}

```

我们主要讲解：屏方面的代码，串口打印部分参考前面的相关串口章节内容。

代码分析 1： ili9320_Initialization() 函数中的 ili9320_PutChar() 函数就是显示英文字符的，该函数完成的是在指定座标显示一个 8x16 点阵的 Ascii 字符， ili9320_PutChar(u16 x,u16 y,u8 c,u16 charColor,u16 bkColor) 函数的 x 是行座标； y 是列座标； charColor 是字符的颜色； bkColor 是字符背景颜色。

代码分析 2： ili9320_PutChar((StartX+8*i),60,str[i],Yellow, Red) 可以看到显示的字符是黄色的字符颜色，背景色是红色的。

str[] = " Welcome jesse to www.armjishu.com !" 这个是希望显示的字符，这里需要注意的是，字符是一个一个来显示的， str[i] 表示需要显示的那个字符，这个字符显示的位置在 X 坐标是 (StartX.+8*i) ,y 坐标是 60 。

代码分析 3：接下来详细分析一下 ili9320_PutChar() 函数，看看到底如何显示一个字符的

```
void ili9320_PutChar(u16 x,u16 y,u8 c,u16 charColor,u16 bkColor)
{
    u16 i=0;
    u16 j=0;
    u8 tmp_char=0;

    for (i=0;i<16;i++)
    {
        tmp_char=ascii_8x16[((c-0x20)*16)+i];
        for (j=0;j<8;j++)
        {
            if ((tmp_char >> 7-j) & 0x01 == 0x01)
            {
                ili9320_SetPoint(x+j,y+i,charColor); // 字符颜色
            }
            else
            {
                ili9320_SetPoint(x+j,y+i,bkColor); // 背景颜色
            }
        }
    }
}
```

a) str[] = " Welcome jesse to www.armjishu.com ! ", str[i]是一个 ASCII 字符，可以参考上面进行查表得知

b) tmp_char = Ascii_8x16[((c-0x20)*16)+i]

首先数组 Ascii_8x16[1536]每 16 个成员是一个字符，因为 ASCII 码是从 32 开始的，而数组是从 0 开始的，所以首先要减掉一个 32 (16 进制 0x20 化成二进制就是 32); 然后(c-0x20)*16 是因为 Ascii_8x16[] 数组中每经过 16 个字节就会真正指向下一个字符的首地址。

| ASCII 打印字符 | | | | | | | | | | | |
|------------|------|------|------|------|------|-----|----|-----|-----|-----|--------------|
| 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | | | | | | |
| 2 | 3 | 4 | 5 | 6 | 7 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 |
| 32 | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p | |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | △ Back space |

- c) for (i=0;i<16;i++)一个 for 循环，把显示一个字符的 16 个字节都循环到位
- d) 接下来，把显示一个字符的 16 个字节里，每个字节都是 8 个 bit 都循环到位，并且判断这个 bit 是 1 还是 0，如果是 1，就显示字符的颜色，表示这个字符这里是黑体的；如果是 0，就显示预先设置好的背景的颜色。

```

for (j=0;j<8;j++)
{
    if((tmp_char >> 7-j) & 0x01 == 0x01)
    {
        ili9320_SetPoint(x+j,y+i,charColor); // 字符颜色
    }
    else
    {
        ili9320_SetPoint(x+j,y+i,bkColor); // 背景颜色
    }
}

```

7.52.5 下载与测试

如果使用JLINK下载固件，请按 [如何使用JLINK V8下载固件到神舟III号开发板](#) 小节进行操作。
如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小

节进行操作。

7.52.6 实验现象

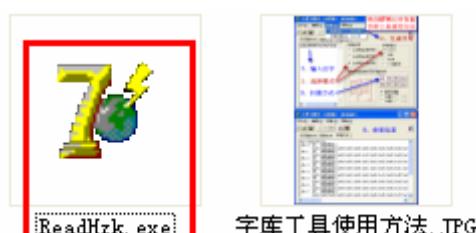
将固件下载到神舟 III 开发板后，复位，神舟 III 号开发板正常情况下将显示一串英文字符。



7.53 TFT彩色液晶屏显示中文字

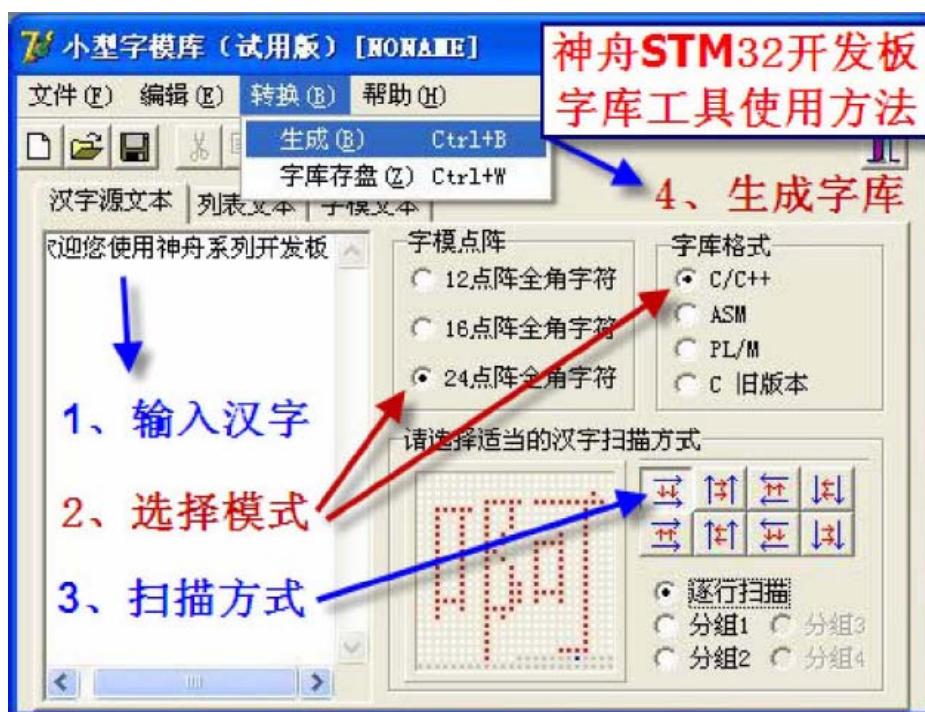
7.53.1 使用工具将中文字转换成二进制码

进入光盘打开神舟 I 号的软件工具中，找到‘神舟开发板汉字生成软件工具’文件夹：



第二步：打开这个软件

第三步：



第四步：把结果复制进去

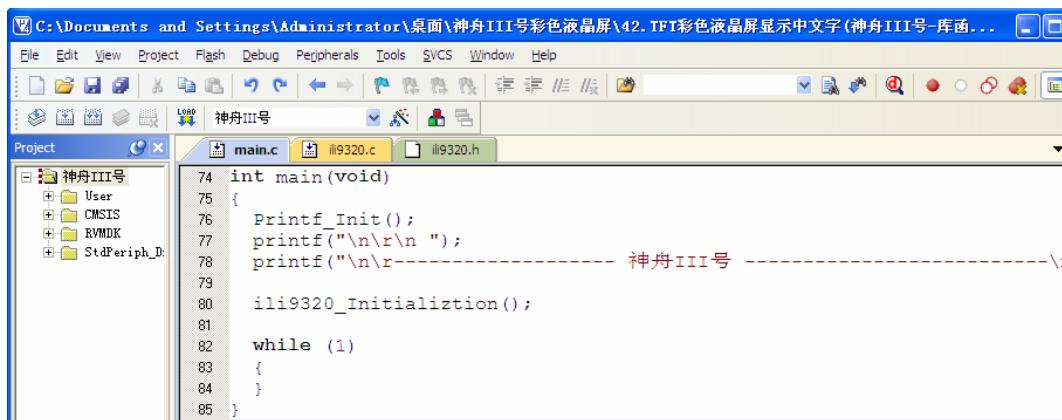


7.53.2 硬件设计

硬件设计同上

7.53.3 软件分析

进入例程的文件夹，打开例程



进入到 main() 函数中：

```

int main(void)
{
    Printf_Init();
    printf("\n\r\n");
    printf("\n\r----- 神舟III号 ----- \n\n\r");
    ili9320_Initialization();

    while (1)
    {
    }
}

```

代码分析 1：液晶屏的初始化等细节前面的例程都有详细介绍，这个例程中主要是这个函数用来显示中文字符的 LCD_DisplayWelcomeStr(0x60); 它显示了这个数组中所有的字符

```

void LCD_DisplayWelcomeStr(u8 Line)
{
    u16 num = 0;

    /* Send the string character by character on LCD */
    for(num=0; num<13; num++)
    {
        /* Display one China character on LCD */
        LCD_DrawChinaChar(Line, num*24+4, (u8 *)WelcomeStr[num]);
    }
}

```

用一个 for 循环来显示 WelcomeStr[13][] 中的 13 个字符：



具体的每个字符显示是通过 LCD_DrawChinaChar() 这个函数进行显示的

```

const u8 WelcomeStr[13][72]={
    //★欢迎您使用神舟系列开发板
    //No:0 ★ 使用频度=1
    ( 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x18,0x00,0x00,0x18,0x00,
    0x18,0x00,0x00,0x3C,0x00,0x00,0x3C,0x00,0x00,0x7C,0x00,0x3F,0xFF,0xFC,0x1F,0xFF,0xF8,0x07,0xFF,
    0xE0,0x03,0xFF,0xC0,0x01,0xFF,0x80,0x01,0xFF,0x80,0x01,0xFF,0x80,0x01,0xFF,0x80,0x03,0xE7,0xC0,
    0x03,0x81,0xC0,0x03,0x00,0xC0,0x04,0x00,0x60,0x00,0x00,0x00,0x00,0x00,0x00,0x00),
    //No:1 欢 使用频度=1
    ( 0x00,0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x06,0x00,0x00,0x46,0x00,0x3F,0xC4,0x00,0x00,
    0xC4,0x04,0x00,0x8F,0xFE,0x20,0x88,0x08,0x11,0x89,0x90,0x09,0x11,0x80,0x05,0x21,0x80,0x02,0x21,
    0x80,0x03,0x03,0x80,0x05,0x82,0x80,0x05,0x82,0x40,0x08,0xC2,0x40,0x08,0x46,0x60,0x10,0x4C,0x20,
    0x20,0x08,0x30,0x40,0x10,0x18,0x00,0x60,0x0E,0x00,0x80,0x00,0x00,0x00,0x00),
    //No:2 迎 使用频度=1
    ( 0x00,0x00,0x00,0x00,0x00,0x18,0x02,0x00,0x0C,0x0E,0x00,0x06,0x72,0x08,0x04,0x41,0xFC,0x00,
    0x41,0x08,0x00,0x41,0x08,0x04,0x41,0x08,0x7E,0x41,0x08,0x04,0x41,0x08,0x41,0x08,0x04,0x41,
    0x08,0x04,0x45,0x08,0x04,0x59,0x08,0x04,0x61,0x78,0x04,0x41,0x18,0x04,0x01,0x00,0x1A,0x01,0x00,
    0x71,0x00,0x00,0x60,0x80,0x02,0x00,0x3F,0xFC,0x00,0x00,0x00,0x00,0x00,0x00),
    //No:3 您 使用频度=1
    ( 0x00,0x00,0x00,0x00,0x00,0x03,0x18,0x00,0x06,0x18,0x00,0x04,0x30,0x00,0x0C,0x3F,0xFC,0x0C,
    0x44,0x10,0x14,0x43,0x20,0x24,0x82,0x00,0x44,0x32,0x40,0x04,0x22,0x20,0x04,0x42,0x18,0x04,0x82,
    0x18,0x05,0x1E,0x08,0x04,0x06,0x00,0x20,0x00,0x01,0x98,0x20,0x09,0x8C,0x10,0x09,0x88,0x4C,
    0x19,0x80,0x4C,0x31,0x80,0x44,0x01,0x80,0xE0,0x00,0xFF,0xC0,0x00,0x00,0x00),
    //No:4 使 使用频度=1
    ( 0x00,0x00,0x00,0x00,0x00,0x01,0x81,0x00,0x03,0x01,0x00,0x03,0x01,0x04,0x02,0xFF,0xFE,0x06,
    0x01,0x00,0x04,0x01,0x00,0x0E,0x21,0x18,0x0E,0x3F,0xE8,0x16,0x21,0x08,0x26,0x21,0x08,0x26,0x21,
    0x08,0x46,0x3F,0xF8,0x06,0x21,0x00,0x06,0x13,0x00,0x06,0x12,0x00,0x06,0x0A,0x00,0x06,0x06,0x00,
    0x06,0x06,0x00,0x06,0x0F,0x00,0x06,0x10,0xE0,0x06,0x60,0x3C,0x01,0x80,0x00),
    //No:5 用 使用频度=1
}

```

代码分析 2：LCD_DrawChinaChar(X, Y, const u8 *c) 函数是显示一个字符到 LCD 上，X 和 Y 分别是
嵌入式专业技术论坛（www.armjishu.com）出品 第 670 页，共 900 页

LCD 横坐标和纵坐标，`const u8 *c` 是需要显示的字符的字模，就是一些二进制数，这个函数的具体实现如下：

```
void LCD_DrawChinaChar(u8 Xpos, u16 Ypos, const u8 *c)
{
    u32 index = 0, i = 0, j = 0;
    u8 Xaddress = 0;
    Xaddress = Xpos;
    ili9320_SetCursor(Ypos, Xaddress);

    for(index = 0; index < 24; index++)
    {
        LCD_WriteRAM_Prepares(); /* Prepare to write GRAM */
        for(j = 0; j < 3; j++)
        {
            for(i = 0; i < 8; i++)
            {
                if((c[3*index + j] & (0x80 >> i)) == 0x00)
                {
                    LCD_WriteRAM(0);
                }
                else
                {
                    LCD_WriteRAM(0x00);
                }
            }
        }
        Xaddress++;
        ili9320_SetCursor(Ypos, Xaddress);
    }
}
```

- a) 首先先设置好光标，就是字符显示的起始点，这个在代码中也是由用户确定的位置，你想到 LCD 哪个位置显示都可以自己灵活的去通过修改参数来达到目的
- b) 因为这里转的字符的字模是 72 个字节，每个字节可以描述 8 个点，每个点显示到 LCD 上需要一个一个来描述，所以可以计算一下 72 个字节分成 24 组，每组 3 个字节，每个字节可以描述 LCD 显示 8 个点，每个点调用如果是 1 就显示字体的颜色，如果是 0 就显示字体背景的颜色，这里颜色的显示是通过调用 `LCD_WriteRAM()` 函数来完成的
- c) 在写颜色之前，每次都调用一下 `LCD_WriteRAM_Prepares()` 这个函数，告诉液晶屏准备写 GRAM，写 GRAM 就是写 LCD，LCD 的颜色实际上是通过写 GRAM，再由 LCD 内部的控制器去从 GRAM 中抓取显示内容，显示到 LCD 上的。

7.53.4 下载与测试

如果使用 JLINK 下载固件，请按 [如何使用 JLINK V8 下载固件到神舟 III 号开发板](#) 小节进行操作。
如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟 III 号开发板](#) 小节进行操作。

如果在 MDK 开发环境中，下载编译好的固件或者在线调试，请按 [如何通过 MDK 编译和在线调试](#) 小节进行操作。

7.53.5 实验现象

将固件下载到神舟 III 号开发板后，复位，神舟 III 号开发板正常情况下将显示一串中文字符。



7.54 TFT彩色液晶屏显示中英文字

7.54.1 硬件设计

硬件设计同上。

7.54.2 软件分析

软件设计参考 TFT 液晶屏显示英文章节和 TFT 液晶屏显示英文章节。

7.54.3 下载与测试

如果使用JLINK下载固件，请按 [如何使用JLINK V8下载固件到神舟III号开发板](#) 小节进行操作。
如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.54.4 实验现象

将固件下载到神舟III号后，复位，神舟III号正常情况下将显示一串英文字符和中文字符



7.55 SD卡访问实验

7.55.1 SD卡的发展历程

SD卡(Secure Digital Memory Card)是一种基于半导体闪存工艺的存储卡，于1999年由日本松下主导概念，参与者东芝和美国SanDisk公司(闪迪公司)进行实质研发而完成。2000年这几家公司发起成立了SD协会(Secure Digital Association简称SDA)，阵容强大，吸引了大量厂商参加。其中包括IBM, Microsoft, Motorola, NEC、Samsung等。在这些领导厂商的推动下，SD卡已成为目前消费数码设备中应用最广泛的一种存储卡，例如数码相机、个人数码助理(PDA)和多媒体播放器等。

这里还要提到手册常用的TF卡。TF卡由Motorola与SanDisk(闪迪)共同推出的最新一代的记忆卡规格。TF卡加上转接器转换成SD卡，如下图：



7.55.2 SD卡的分类

SD 可分为 3 类：SD、SDHC、SDXC。

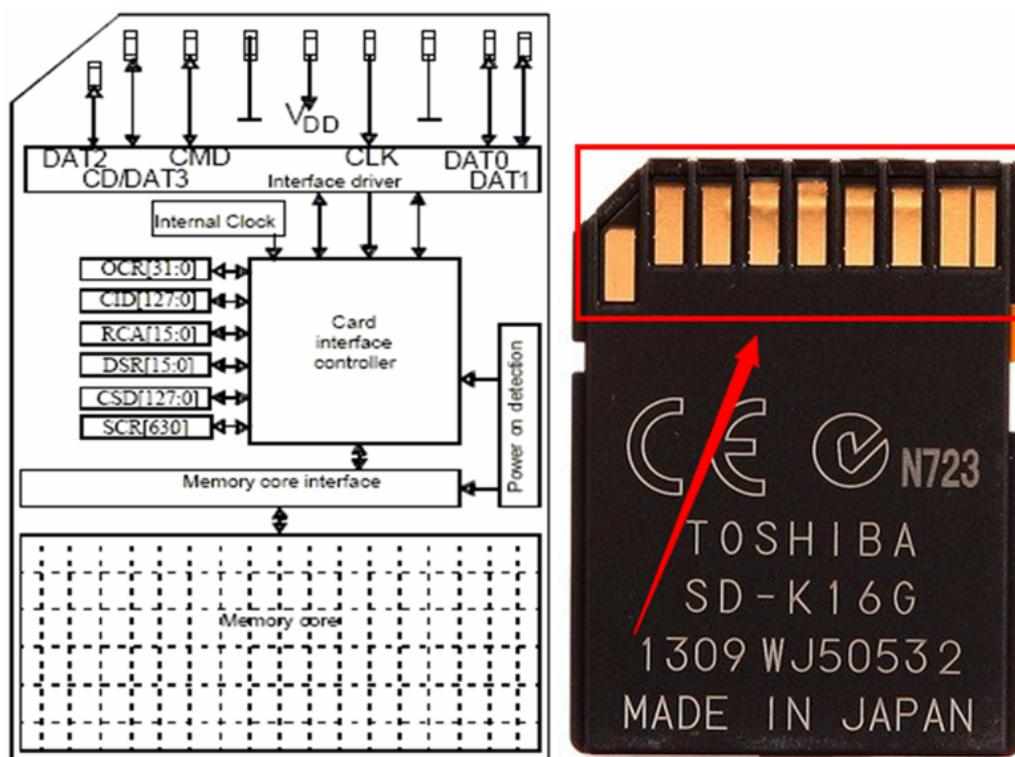
SD 是早先的版本的，据说是由 MMC 演变而来的。最大支持 2GB 大小容量

SDHC 是大容量 SD 卡，也就是 SD High Capacity，支持最大 32GB 大小容量 SDHC，Secure Digital High Capacity，大容量 SD，也就是说，超过 2G 的 SD 都叫 SDHC，因为早期的 SD 使用的是 FAT16 文件系统，并不支持大容量，而 SDHC 升级为 FAT32，才支持 2G 以上的大容量。

SDXC (SD eXtended Capacity) 是去年 09 年才发布的新标准，支持最大 2TB 的大小容量。SDXC 是 SD eXtended Capacity 的缩写，是新提出的标准，除了容量可以升级为最大 2T 以外，主要是可以支持 300M/s 的传输速度，也就是说是“高速 SD”卡。不过支持 SDXC 卡的数码相机并不多，主要都是今年推出的新品，而 SDXC 又是不可向下兼容的，不支持普通的 SD 和 SDHC 卡槽和读卡器。对于大部分数码相机和单反来说，高速 SDHC 存储卡的速度已经足以实现高速连拍和高清视频拍摄。

SD 卡 SDHC 卡协议基本兼容。但是 SDXC 卡，同这两者区别就比较大了，本章我们讨论是 SD/SDHC 卡（简称 SD 卡）。

7.55.3 SD卡的内部结构



可以看到，SD卡有9个端子，我们通过这9个端子对SD卡进行操作。CPU向SD卡的Memory code存入数据时，SD卡接口控制器（Card interface controller）起到承上启下的作用。对上连接着接口驱动（Interface driver），对下连接着存储核心接口（Memory core interface）。

1: SD卡上所有的单元由内部时钟发生器（Internal Clock）提供时钟。接口驱动单元（Interface driver）同步外部时钟的DAT和CMD信号到内部所用的时钟。

2: 本卡由6线SD卡接口控制，包括：CMD, CLK, DAT0-DAT3。

3: SD卡堆叠中为了标识SD卡，一个卡标识寄存器(CID)和一个相应地址寄存器(RCA)预先准备好。

4: 一个附加的寄存器包括不同类型操作参数。这个寄存器叫做CSD。

5: 如果接到复位命令 (CMD0) 时，CS信号有效 (低电平)，SPI模式启用。命令CMD0就是0, CMD16

就是16，其它以此类推

- 6: 使用SD卡线访问存储器还是寄存器的通信由SD卡标准定义。
- 7: 卡有自己的电源开通检测单元，无需附加的主复位信号来在电源开启后安装卡。它防短路，在带电插入或移出卡时，无需外部编程电压，编程电压卡内生成。

SD卡的内存组织形式：

数据读写的基本单元是一个字节，可以按要求组织成不同的块。

Block: 块大小可以固定，也可以改变，允许的块大小是实际大小等信息存储在CSD 寄存器。

Sector: 和擦除命令相关，由几个块组成。Sector 的大小对每个设备是固定的，大小信息存储在CSD 寄存器。

7.55.4 SD卡模式选择

SD卡共支持三种传输模式：

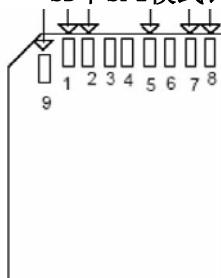
- 1) SPI模式（独立序列输入和序列输出）
- 2) 1位SD模式（独立指令和数据通道，独有的传输格式）
- 3) 4位SD模式（使用额外的针脚以及某些重新设置的针脚。支持四位宽的并行传输）

本实验我们采用的是SPI模式，我们主要对这个模式进行分析。

SD 卡可以通过单数据线 (DAT0) 或四根数据线 (DAT0-DAT3) 进行数据传输。单根数据线传输最大传输速率为25 Mbit/s，四根数据线最大传输速率为100 Mbit/s。

SPI模式相对于SD模式的不足之处是丧失了速度性能。SPI模式使用字节传输，所有的数据被融合到一些字节中。SPI模式的优点就是简化主机的设计。

SD卡SPI模式针脚定义如下：



| 针脚 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------|---------|------|-----|-----|-----|-----|------|------|------|
| SD卡模式 | CD/DAT3 | CMD | VSS | VCC | CLK | VSS | DAT0 | DAT1 | DAT2 |
| SPI模式 | CS | MOSI | VSS | VCC | CLK | VSS | MISO | NC | NC |

如何进入SPI模式：

SD卡默认为SD模式，要进入SPI模式时，要遵守如下操作：当SD卡接收RESTE命令 (CMD0) 时，拉低CS即可。命令CMD0就是0，CMD16就是16，其它以此类推。

7.55.5 SD卡SPI模式命令分析

CPU 通过 SPI 的方式向 SD 卡发送命令，SD 卡 SPI 模式命令格式：

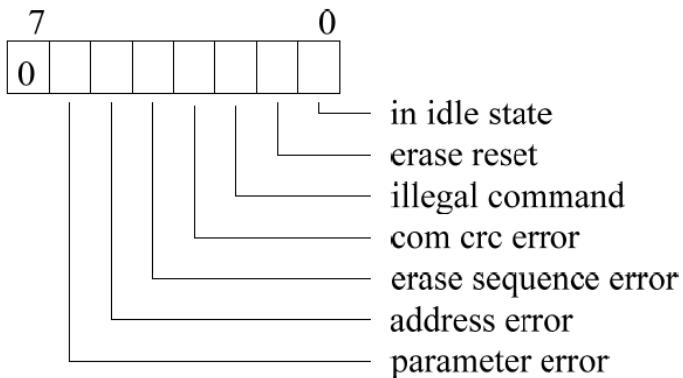
| 字节 1 | | | | 字节 2--5 | | | 字节 6 | | |
|------|---|---------|---|---------|---|-----|------|---|--|
| 7 | 6 | 5 | 0 | 31 | 0 | 7 | 1 | 0 | |
| 0 | 1 | command | | 命令参数 | | CRC | | 1 | |

由6个字节构成，高位在前。SPI模式下Command从CMD0到CMD63，这64个命令并非每一个命令在SPI模式下都可以使用。SPI命令分为11个组，各个组是多个命令的集合，每个组中的命令有相似的功能。这里介绍三个常用命令。CMD0，CMD1。

| CMD INDEX | SPI Mode | Argument | Resp | Abbreviation | Command Description |
|-----------|------------------|--|------|---------------|---|
| CMD0 | Yes | [31:0] stuff bits | R1 | GO_IDLE_STATE | Resets the SD Memory Card |
| CMD1 | Yes ¹ | [31]Reserved bit [30]HCS [29:0]Reserved bits | R1 | SEND_OP_COND | Sends host capacity support information and activates the card's initialization process. HCS is effective when card receives SEND_IF_COND command. Reserved bits shall be set to '0'. |
| CMD16 | Yes | [31:0] block length | R1 | SET_BLOCKLEN | Sets a block length (in bytes) for all following block commands (read and write) ² of a Standard Capacity Card. Block length of the read and write commands are fixed to 512 bytes in a High Capacity Card. The length of LOCK_UNLOCK command is set by this command in both capacity cards. |

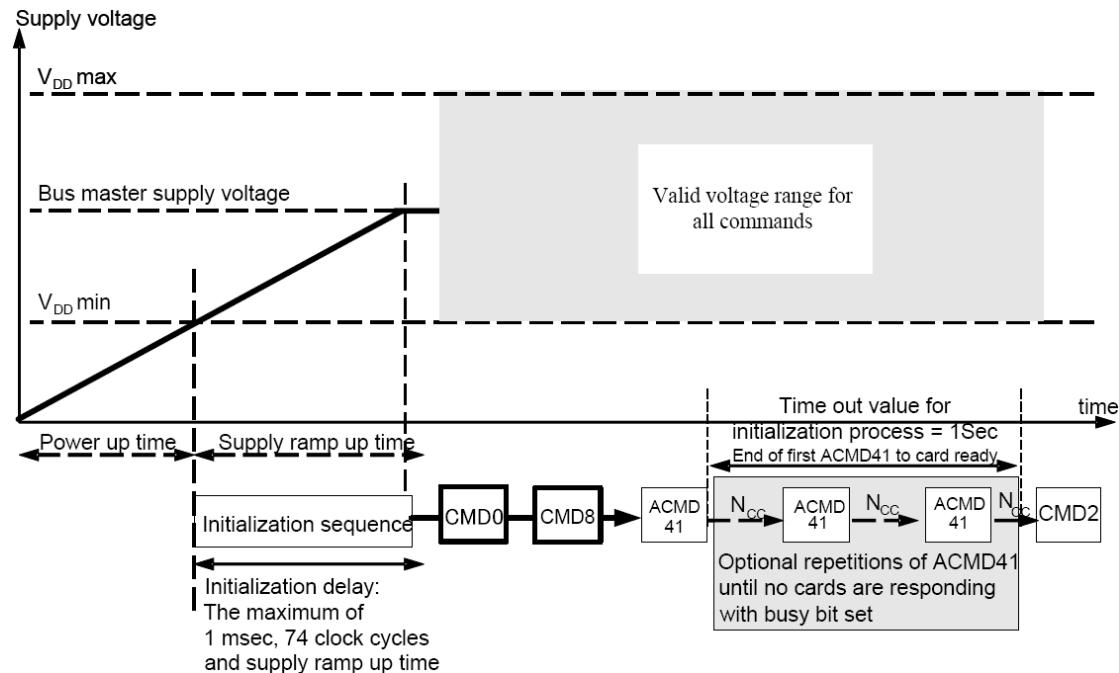
CMD0 为复位，CMD1 为激活初始化，CMD16 设置一个读写块的长度。

每发送一个命令，SD 卡都会给出一个应答，以告知主机该命令的执行情况，或者返回主机需要获取的数据。SPI 模式下，SD 卡针对不同的命令，有不同的应答，R1 的应答，如下图所示：



- In idle state:** The card is in idle state and running the initializing process.
- Erase reset:** An erase sequence was cleared before executing because an out of erase sequence command was received.
- Illegal command:** An illegal command code was detected.
- Communication CRC error:** The CRC check of the last command failed.
- Erase sequence error:** An error in the sequence of erase commands occurred.
- Address error:** A misaligned address that did not match the block length was used in the command.
- Parameter error:** The command's argument (e.g. address, block length) was outside the allowed range for this card.

7.55.6 SD卡SPI模式的读写操作:



“启动时间”被定义为从0V上升到 V_{dd_min} 的时间。在上电后，主机启动SCK及在CMD线上发送74个高电平的信号，接着发送CMD0进入SPI模式，然后发送CMD8激活初始化进程。

上电后，包括热插入，卡进入idle状态。在该状态SD卡忽略所有总线操作直到接收到ACMD41命令。ACMD41命令是一个特殊的同步命令，用来协商操作电压范围，并轮询所有的卡。除了操作电压信息，ACMD41的响应还包括一个忙标志，表明卡还在power-up过程工作，还没有准备好识别操作，即告诉主机卡还没有就绪。主机等待(继续轮询)直到忙标志清除。单个卡的最大上电时间不能操作1秒。

上电后，主机开始时钟并在CMD线上发送初始化序列，初始化序列由连续的逻辑“1”组成。序列长度为最大1毫秒，74个时钟或supply-ramp-up时间。额外的10个时钟(64个时钟后卡已准备就绪)用来实现同步。

每个总线控制器必须能执行ACMD41和CMD1。

读写操作:

读一个块的操作:

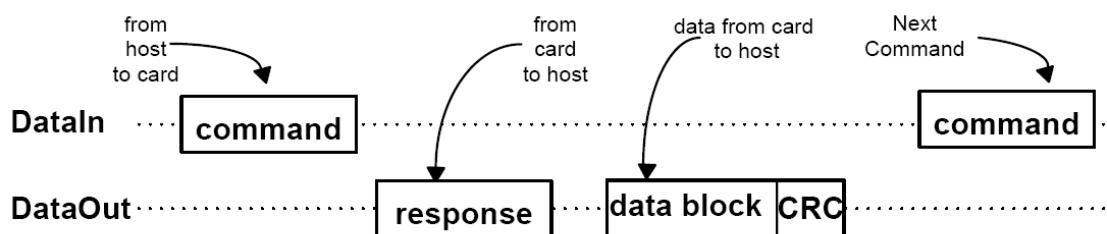


Figure 7-3: Single Block Read Operation

写一个块的操作:

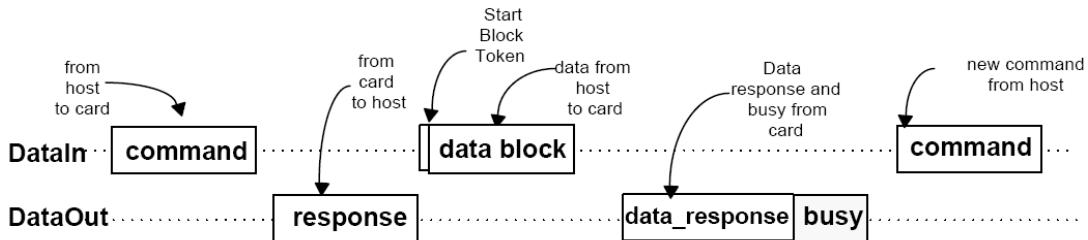


Figure 7-6: Single Block Write Operation

主机根据事先定义的长度读写一个数据块。由发送模块产生一个16位的CRC校验码，接受端根据校验码进行检验。读操作的块长度受设备sector大小(512 bytes)的限制，但是可以最小为一个字节。不对齐的访问是不允许的，每个数据块必须位于单个物理sector内。写操作的大小必须为sector大小，起始地址必须与sector边界对齐。

Multiple Block Mode: 主机可以读写多个数据块(相同长度)，根据命令中的地址读取或写入连续的内存地址。操作通过一个停止传输命令结束。写操作必须地址对齐。

7.55.7 SD卡SDIO模式命令分析

SDIO的所有命令及命令响应，都是通过SDIO-CMD引脚来传输的。

命令只能由host即STM32F103ZET的SDIO控制器发出。SDIO协议把命令分成了11种，包括基本命令，读写命令还有ACMD系列命令等。其中，在发送ACMD命令前，要先向卡发送编号为CMD55的命令。

命令格式图：

| Bit position | 47 | 46 | [45:40] | [39:8] | [7:1] | 0 |
|--------------|-----------|------------------|---------------|----------|-------|---------|
| Width (bits) | 1 | 1 | 6 | 32 | 7 | 1 |
| Value | '0' | '1' | x | x | x | '1' |
| Description | start bit | transmission bit | command index | argument | CRC7 | end bit |

Table 4-16: Command Format

其中的start bit, transmission bit, crc7, endbit, 都是由STM32中的SDIO硬件完成，我们在软件上配置的时候只需要设置command index和命令参数argument。Command index就是命令索引(编号)，如CMD0, CMD1…被编号成0, 1…。有的命令会包含参数，读命令的地址参数等，这个参数被存放在argument段。

7.55.8 SDIO模式下SD卡对host的各种命令的回复称为响应

SD卡对host的各种命令的回复称为响应，除了CMD0命令外，SD卡在接收到命令都会返回一个响应。对于不同的命令，会有不同的响应格式，共7种，分为长响应型(136bit)和短响应型(48bit)。以下图，响应6(R6)为例：

| | | | | | | | |
|---------------------|-----------|------------------|--------------------------|---------------------------------------|---|-------|---------|
| Bit position | 47 | 46 | [45:40] | [39:8] Argument field | | [7:1] | 0 |
| Width (bits) | 1 | 1 | 6 | 16 | 16 | 7 | 1 |
| Value | '0' | '0' | x | x | x | x | '1' |
| Description | start bit | transmission bit | command index ('000011') | New published RCA [31:16] of the card | [15:0] card status bits: 23,22,19,12:0 (see Table 4-35) | CRC7 | end bit |

Table 4-32: Response R6

SDIO 通过 CMD 接收到响应后，硬件去除头尾的信息，把 command index 保存到 SDIO_RESPCMD 寄存器，把 argument field 内容保存存储到 SDIO_RESPx 寄存器中。

数据写入，读取。请看下面的写数据时序图，在软件上，我们要处理的只是读忙。另外，我们的实验中用的是 Micro SD 卡，有 4 条数据线，默认的时候 SDIO 采用 1 条数据线的传输方式，更改为 4 条数据线模式要通过向卡发送命令来更改。

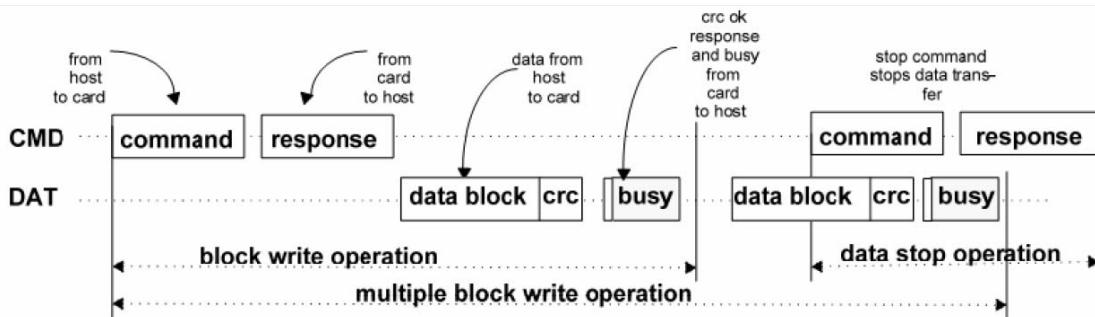


Figure 3-4: (Multiple) Block Write Operation

7.55.9 SD卡有无文件系统的区别：

CPU 通过控制器向 SD 卡的存储数据区写数据。我们要发送数据，需要确定存储什么数据，即存储数据；存储到什么地方，即存储地址。假设我们的 SD 卡容量是 2G，要存储两个数据第一个数据是 2KB 的大小，第二个数据是 4KB 的大小。当我们向 SD 卡写入 2K 的数据后，再向里边写 4KB 的数据。那么后面写入的 4KB 的数据，有没有可能将前面的 2KB 的数据覆盖，从而引起数据的丢失？回想一下，我们在电脑上往 SD 卡里存储东西的时候，直接就是往里面拖东西。没有看到地址什么的。这个是怎么回事？这个就涉及到文件系统，这里我们先引入这个概念，下一章节，再给大家分析文件系统。

7.55.10 实验原理

SD卡支持两种接口访问模式，SDIO模式和SPI模式。本次实验我们采用SDIO模式，首先程序运行后，初始化实验需要使用的串口，然后初始化访问SD卡要使用的SPI接口，按照协议要求初始化SD卡，读取SD卡的信息包括规范标准的版本、卡的容量、卡生产日期等信息，最后串口打印SD卡的起始扇区（MBR）数据。

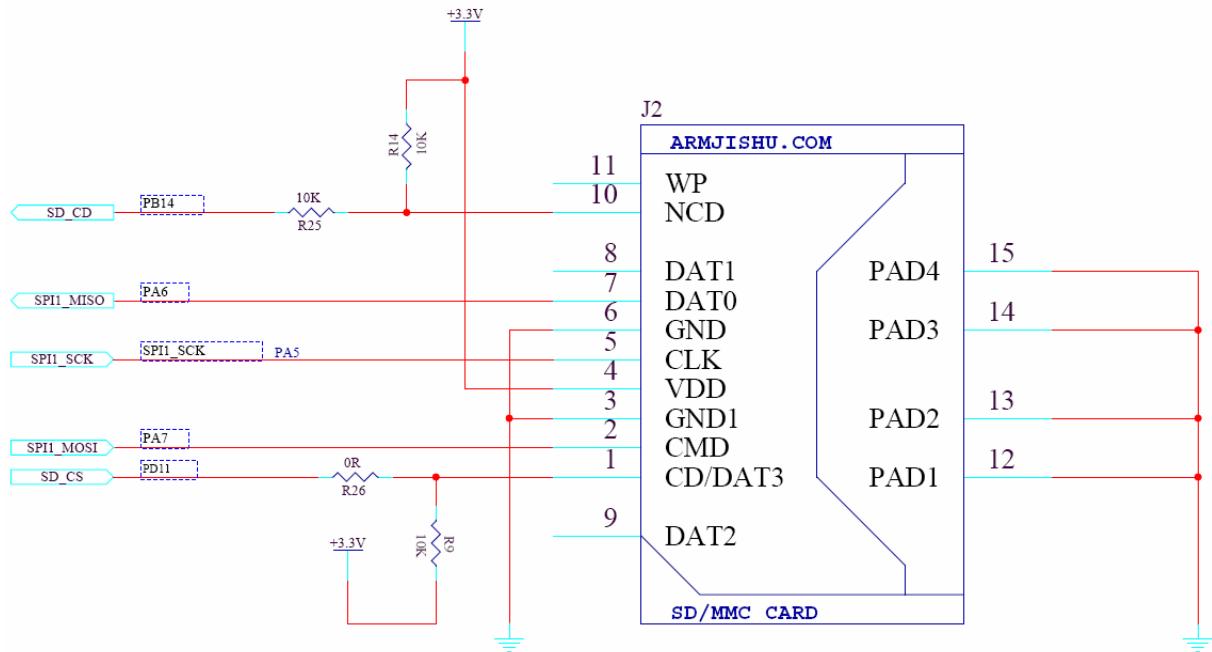
7.55.11 硬件设计

SD卡读卡器实验要用到的硬件资源有：

- 串口 1：串口 1 在本实验中打印程序运行过程中的提示信息。
- SD 卡座：神舟 III 号最大支持 2G 的 SD 卡

SD 卡座

神舟III号开发板载有标准的SD卡接口，有了这个接口，我们就可以外扩容量存储设备，可以用来记录数据。其原理图如下：



图表 17 SD 卡接口原理图

7.55.12 软件设计

我们从主函数开始分析：

```
017 int main(void)
018 {
019     /*SysTick初始化*/
020     SysTick_Configuration();
021
022     /*TFT彩屏初始化*/
023     STM3210E_LCD_Init();
024
025     /*串口初始化*/
026     USART_Configuration();
027
028     /*将彩屏刷成黑色*/
029     LCD_Clear(Black);
030
031     /*初始化并设置SDIO接口*/
032     Status = SD_InitAndConfig();
033
034     while (1)
035     {
036     }
037 }
```

代码分析 1：进入 main() 主函数后，初始化 SysTick、串口、TFT 彩屏后，就开始初始化 SD 卡，然后读取 SD 卡的参数，比如多大，容量这些；这里主要是两个大问题需要解决：

- 一. SD 卡是如何初始化
- 二. SD 卡的参数是怎样保存的，是什么格式
- 三. SD 卡的数据是怎样被读出来，一次读多少数据

代码分析 2：这里的 SD 卡采用的 SDIO 模式，STM32 通过 SDIO 的方式与 SD 卡进行沟通，那么跟 SD 卡沟通的协议和命令是什么呢？部分命令如下：

```

074 /* SDIO Commands Index */
075 #define SDIO_GO_IDLE_STATE ((u8)0)
076 #define SDIO_SEND_OP_COND ((u8)1)
077 #define SDIO_ALL_SEND_CID ((u8)2)
078 #define SDIO_SET_REL_ADDR ((u8)3)
079 #define SDIO_SET_DSR ((u8)4)
080 #define SDIO_SDIO SEN_OP_COND ((u8)5)
081 #define SDIO_HS_SWITCH ((u8)6)
082 #define SDIO_SEL_DESEL_CARD ((u8)7)
083 #define SDIO_HS_SEND_EXT_CSD ((u8)8)
084 #define SDIO_SEND_CSD ((u8)9)
085 #define SDIO_SEND_CID ((u8)10)
086 #define SDIO_READ_DAT_UNTIL_STOP ((u8)11)
087 #define SDIO_STOP_TRANSMISSION ((u8)12)
088 #define SDIO_SEND_STATUS ((u8)13)
089 #define SDIO_HS_BUSTEST_READ ((u8)14)
090 #define SDIO_GO_INACTIVE_STATE ((u8)15)
091 #define SDIO_SET_BLOCKLEN ((u8)16)
092 #define SDIO_READ_SINGLE_BLOCK ((u8)17)
093 #define SDIO_READ_MULT_BLOCK ((u8)18)
094 #define SDIO_HS_BUSTEST_WRITE ((u8)19)
095 #define SDIO_WRITE_DAT_UNTIL_STOP ((u8)20)
096 #define SDIO_SET_BLOCK_COUNT ((u8)23)
097 #define SDIO_WRITE_SINGLE_BLOCK ((u8)24)
098 #define SDIO_WRITE_MULT_BLOCK ((u8)25)
099 #define SDIO_PROG_CID ((u8)26)
100 #define SDIO_PROG_CSD ((u8)27)

```

首先介绍SDIO模式下几个重要的操作命令，如下表所示：

| 命令 | 参数 | 回应 | 描述 |
|-------------|---------|----|------------------|
| CMD0(0X00) | NONE | R1 | 复位SD卡 |
| CMD8(0X08) | | | 识别卡的协议版本 |
| CMD9(0X09) | NONE | R1 | 读取卡特定数据寄存器 |
| CMD10(0X0A) | NONE | R1 | 读取卡标志数据寄存器 |
| CMD16(0X10) | 块大小 | R1 | 设置块大小(字节数) |
| CMD17(0X11) | 地址 | R1 | 读取一个块的数据 |
| CMD24(0X18) | 地址 | R1 | 写入一个块的数据 |
| CMD41(0X29) | NONE | R1 | 引用命令的前命令 |
| CMD55(0X37) | NONE | R1 | 开始卡的初始化 |
| CMD59(0X3B) | 仅最后一位有效 | R1 | 设置CRC开启(1)或关闭(0) |

SD卡应答数据r1格式如下所示：

代码分析 3：详细分析 SD_InitAndConfig() 函数，SD 卡的初始化过程如下：

```
040 SD_Error SD_InitAndConfig(void)
041 {
042     Status = SD_Init();
043
044     if (Status == SD_OK)
045     {
046         LCD_DisplayStringLine(Line0,"SD_Init success",White,Black);
047     }
048     else
049     {
050         LCD_DisplayStringLine(Line0,"SD_Init fail",White,Black);
051     }
052     Delay(200);
053     if (Status == SD_OK)
054     {
055         /*----- Read CSD/CID MSD registers/获取SD的信息
056         Status = SD_GetCardInfo(&SDCardInfo);
057     }
058
059     if (Status == SD_OK)
060     {
061         /*----- Select Card -----
062         Status = SD_SelectDeselect((u32) (SDCardInfo.RCA << 16));
063     }
064
065     if (Status == SD_OK)
```

该函数中调用 SD_Init() 函数，对 SD 卡进行初始化。调用函数 SD_GetCardInfo() 获取 SD 卡的信息。

代码分析 4：函数 SD_Init() 分析

```
0133 SD_Error SD_Init(void)
0134 {
0135     SD_Error errorstatus = SD_OK;
0136
0137     {
0138         GPIO_InitTypeDef GPIO_InitStructure;
0139
0140         RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
0141
0142         GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
0143         GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
0144         GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
0145         GPIO_Init(GPIOB, &GPIO_InitStructure);
0146
0147         GPIO_ResetBits(GPIOB, GPIO_Pin_5) ;
0148     }
0149
0150     /* Configure SDIO interface GPIO */
0151     GPIO_Configuration();
0152
0153     /* Enable the SDIO AHB Clock */
0154     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_SDIO, ENABLE);
0155
0156     /* Enable the DMA2 Clock */
0157     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA2, ENABLE);
0158
0159     SDIO_DeInit();
```

图确定。

- 2) 分别调用了 SD_PowerON()和 SD_InitializeCards()函数，这两个函数共同实现了上面提到的卡检测、识别流程。
- 3) 调用 SDIO_Init(&SDIO_InitStructure)库函数配置 SDIO 的时钟，数据线宽度，硬件流(在读写数据的时候，开启硬件流是和很必要的，可以减少出错)

代码分析 4: SD_PowerON() 函数，SD 卡的上电。截取的部分代码如下：

```
0237
0238 /* CMD8: SEND_IF_COND -----*/
0239 /* Send CMD8 to verify SD card interface operating condition */
0240 /* Argument: - [31:12]: Reserved (shall be set to '0')
0241     - [11:8]: Supply Voltage (VHS) 0x1 (Range: 2.7-3.6 V)
0242     - [7:0]: Check Pattern (recommended 0xAA) */
0243 /* CMD Response: R7 */
0244 SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN;
0245 SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND;
0246 SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
0247 SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
0248 SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
0249 SDIO_SendCommand(&SDIO_CmdInitstructure);
0250
0251 errorstatus = CmdResp7Error();
0252
0253 if (errorstatus == SD_OK)
0254 {
0255     CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0; /* SD Card 2.0 */
0256     SDType = SD_HIGH_CAPACITY;
0257 }
0258 else
0259 {
0260     /* CMD55 */
0261     SDIO_CmdInitStructure.SDIO_Argument = 0x00;
0262 }
```

SD 卡的上电有它的一套流程，如下图：

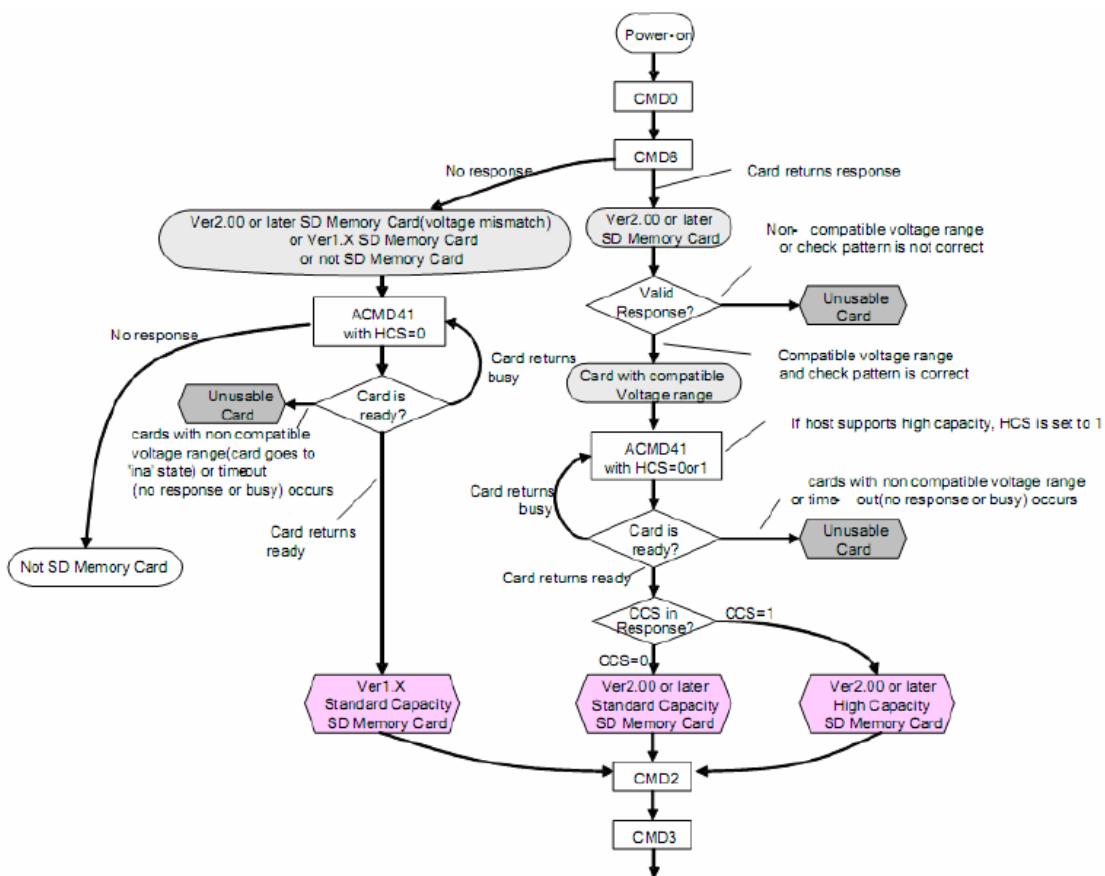


Figure 4-2: Card Initialization and Identification Flow (SD mode)

- 1) 初始化的时钟。SDIO_CK 的时钟分为两个阶段，在初始化阶段 SDIO_CK 的频率要小于 400KHz，初始化完成后可把 SDIO_CK 调整成高速模式，高速模式时超过 24M 要开启 bypass 模式，对于 SD 存储卡即使开启 bypass，最高频率不能超过 25MHz。

2.CMD8命令。

CMD8命令格式：

| Bit position | 47 | 46 | [45:40] | [39:20] | [19:16] | [15:8] | [7:1] | 0 |
|--------------|-----------|------------------|---------------|---------------|------------------------|---------------|-------|---------|
| Width (bits) | 1 | 1 | 6 | 20 | 4 | 8 | 7 | 1 |
| Value | '0' | '1' | '001000' | '00000h' | x | x | x | '1' |
| Description | start bit | transmission bit | command index | reserved bits | voltage supplied (VHS) | check pattern | CRC7 | end bit |

CMD8命令中的VHS是用来确认主机SDIO是否支持卡的工作电压的。Check pattern部分可以是任何数值，若SDIO支持卡的工作电压，卡会把接收到的check pattern数值原样返回给主机。

CMD8命令的响应格式R7：

| | | | | | | | | |
|---------------------|-----------|------------------|---------------|---------------|------------------|----------------------------|-------|---------|
| Bit position | 47 | 46 | [45:40] | [39:20] | [19:16] | [15:8] | [7:1] | 0 |
| Width (bits) | 1 | 1 | 6 | 20 | 4 | 8 | 7 | 1 |
| Value | '0' | '0' | '001000' | '00000h' | x | x | x | '1' |
| Description | start bit | transmission bit | command index | reserved bits | voltage accepted | echo-back of check pattern | CRC7 | end bit |

Table 4-33: Response R7

3) ACMD41命令。

这个命令也是用来进一步检查SDIO是否支持卡的工作电压的，协议要它在调用它之前必须先调用CMD8，另外还可以通过它命令参数中的HCS位来区分卡是SDHC卡还是SDSC卡。

确认工作电压时循环地发送ACMD41，发送后检查在SD卡上的OCR寄存器中的pwr_up位，若pwr_up位置为1，表明SDIO支持卡的工作电压，卡开始正常工作。

同时把ACMD41中的命令参数HCS位置1，卡正常工作的时候检测OCR寄存器中的CCS位，若CCS位为1则说明该卡为SDHC卡，为零则为SDSC卡。

因为ACMD41命令属于ACMD命令，在发送ACMD命令前都要先发送CMD55.

4) ACMD41命令格式

| ACMD INDEX | type | argument | resp | abbreviation | command description |
|------------|------|--|------|-----------------|--|
| ACMD41 | bcr | [31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] V _{DD} Voltage Window(OCR[23:0]) | R3 | SD_SEND_OP_COND | Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30]. |

ACMD41命令的响应 (R3)，返回的是OCR寄存器的值

代码分析5：函数SD_InitializeCards()

```

0360 SD_Error SD_InitializeCards(void)
0361 {
0362     SD_Error errorstatus = SD_OK;
0363     u16 rca = 0x01;
0364
0365     if (SDIO_GetPowerState() == SDIO_PowerState_OFF)
0366     {
0367         errorstatus = SD_REQUEST_NOT_APPLICABLE;
0368         return(errorstatus);
0369     }
0370
0371     if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
0372     {
0373         /* Send CMD2 ALL_SEND_CID */
0374         SDIO_CmdInitStructure.SDIO_Argument = 0x0;
0375         SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_ALL_SEND_CID;
0376         SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
0377         SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
0378         SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
0379         SDIO_SendCommand(&SDIO_CmdInitStructure);
0380
0381         errorstatus = CmdResp2Error();
0382
0383         if (SD_OK != errorstatus)
0384         {
0385             return(errorstatus);
0386         }
    
```

SD卡上电OK后，会进入该函数。该函数有2个重要的命令：CMD2和CMD3。

1) CMD2

CMD2命令是要求卡返回它的CID寄存器的内容。命令的响应格式 (R2)：

| Bit position | 135 | 134 | [133:128] | [127:1] | 0 |
|--------------|-----------|------------------|-----------|--|---------|
| Width (bits) | 1 | 1 | 6 | 127 | 1 |
| Value | '0' | '0' | '111111' | x | '1' |
| Description | start bit | transmission bit | reserved | CID or CSD register incl. internal CRC7 | end bit |

Table 4-30: Response R2

因为命令格式是136位的，属于长响应。软件接收的信息有128位。在长响应的时候通过SDIO.GetResponse(SDIO_RESP4)；中的不同参数来获取CID中的不同数据段的数据。

2) CMD3

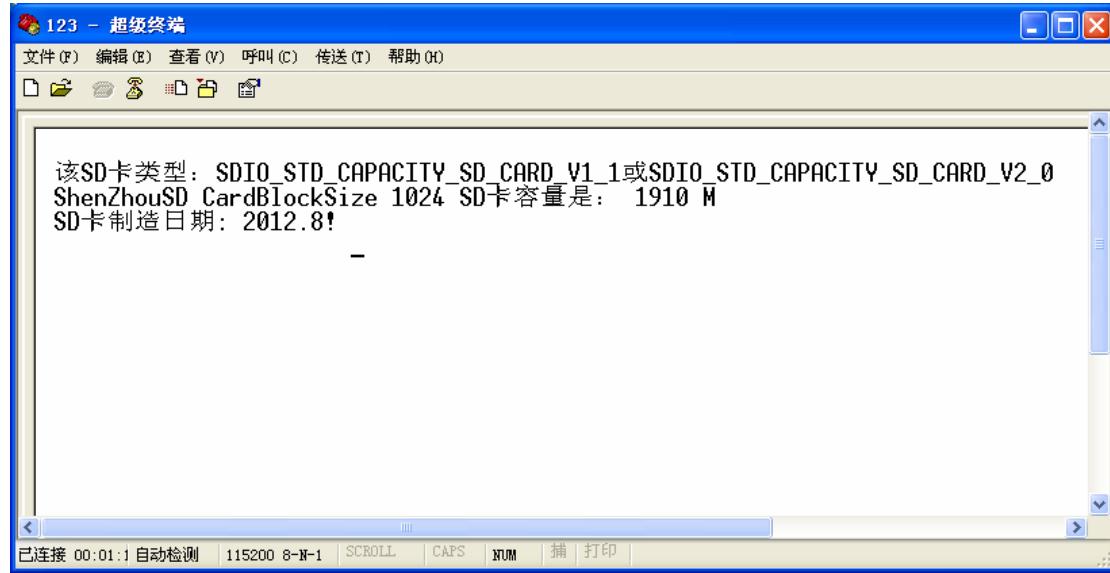
CMD3命令是要求卡向主机发送卡的相对地址。在接有多个卡的时候，主机要求接口上的卡重新发一个相对地址，这个地址跟卡的实际ID不一样。比如接口上接了5个卡，这5个卡的相对地址就分别为1，2，3，4，5.以后主机SDIO对这几个卡寻址就直接使用相对地址。这个地址的作用就是为了寻址更加简单。

7.55.13 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.55.14 实验现象

正常情况下，神舟III号串口1将打印如下信息：



可以看出测试时使用的卡容量为2GB。

7.56 SD卡 FAT文件系统访问实验

7.56.1 意义与作用

神舟III号开发板资源丰富，上一中介我们介绍了如何通过访问神舟III号上插入的SD卡。本章节我们将通过FatFS文件系统访问SD上的文件，通过本实验，我们还可以对FatFS文件系统有一个深入的了解。

7.56.2 文件系统浅析

簇(CLUST)的本意就是“一群”、“一组”，即一组扇区(一个磁道可以分割成若干个大小相等的圆弧，叫扇区)的意思。因为扇区的单位太小，因此把它捆在一起，组成一个更大的单位更方便进行灵活管理。簇的大小通常是可以变化的，是由操作系统在所谓“(高级)格式化”时规定的，因此管理也更加灵活。

通俗地讲文件就好比是一个家庭，数据就是人，即家庭成员；所谓簇就是一些单元套房；扇区是组成这些单元套房的一个个大小相等的房间。一个家庭可能住在一套或多套单元房子里，但一套房子不能同时住进两个家庭的成员。文件系统是操作系统与驱动器之间的接口，当操作系统请求从硬盘里读取一个文件时，会请求相应的文件系统(FAT 16)打开文件。扇区是磁盘最小的物理存储单元，但由于操作系统无法对数目众多的扇区进行寻址，所以操作系统就将相邻的扇区组合在一起，形成一个簇，然后再对簇进行管理。每个簇可以包括2、4、8、16、32或64个扇区。显然，簇是操作系统所使用的逻辑概念，而非磁盘的物理特性。

为了更好地管理磁盘空间和更高效地从硬盘读取数据，操作系统规定一个簇中只能放置一个文件的内容，因此文件所占用的空间，只能是簇的整数倍；如果文件实际大小小于一簇，它也要占一簇的空间。如果文件实际大小大于一簇，根据逻辑推算，那么该文件就要占两个簇的空间。所以，一般情况下文件所占空间要略大于文件的实际大小，只有在少数情况下，即文件的实际大小恰好是簇的整数倍时，文件的实际大小才会与所占空间完全一致。

文件共用一个簇，不然会造成数据混乱。

FAT16使用了16位的空间来表示每个扇区(Sector)配置文件的情形，故称之为FAT16。

FAT16由于受到先天的限制，因此每超过一定容量的分区之后，它所使用的簇(Cluster)大小就必须扩增，以适应更大的磁盘空间。所谓簇就是磁盘空间的配置单位，就象图书馆内一格一格的书架一样。每个要存到磁盘的文件都必须配置足够数量的簇，才能存放到磁盘中。FAT16各分区与簇大小的关系如下表：

| 分区大小 | FAT16簇大小 | 扇区 | 扇区数 |
|---------------|----------|------|-----|
| 16MB-127MB | 2KB | 512B | 4 |
| 128MB-255MB | 4KB | 512B | 8 |
| 256MB-511MB | 8KB | 512B | 16 |
| 512MB-1023MB | 16KB | 512B | 32 |
| 1024MB-2047MB | 32KB | 512B | 64 |

大家可以计算出来，2的16次方是64K，也就是65536，如果按照最大是32KB的簇大小来计算，用65536乘以32KB，也就是说，有65536个簇，也就是2GB空间，所以如果是FAT16格式的文件系统，最大的限制是2G限制。

7.56.3 实验原理

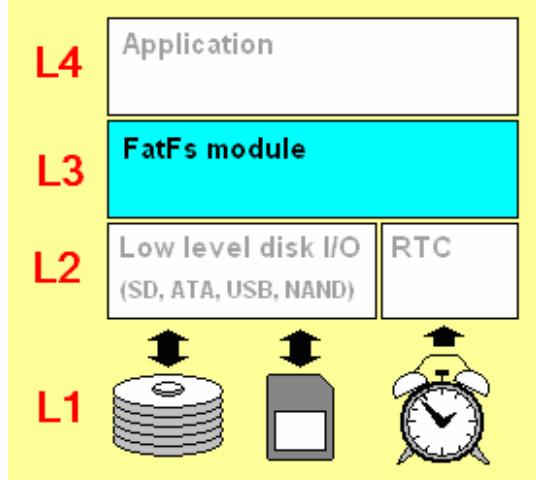
SD卡支持两种接口访问模式，SDIO模式和SPI模式。本次实验我们采用SDIO模式，首先程序运行后，初始化实验需要使用的LCD显示接口，然后初始化访问SD卡要使用的SDIO接口，按照协议要求初始化SD卡，读取SD的信息包括规范标准的版本、卡的容量、卡生产日期等信息。

然后我们将调用FatFs文件系统，访问SD卡，在TFT彩屏输出SD卡里的文件列表。

FatFs简介： FatFs是一个通用的文件系统模块，用于在小型嵌入式系统中实现FAT文件系统。FatFs 的编写遵循ANSI C，因此不依赖于硬件平台。它可以嵌入到便宜的微控制器中，如 8051, PIC, AVR, SH, Z80, H8, ARM 等等，不需要做任何修改。（对应的英文：FatFs is a generic FAT file system module for small embedded systems. The FatFs is written in compliance with ANSI C and completely separated from the disk I/O layer. Therefore it is independent of hardware architecture. It can be incorporated into low cost microcontrollers, such as AVR, 8051, PIC, ARM, Z80, 68k and etc..., without any change.）

FatFs最新版本可以在http://elm-chan.org/fsw/ff/00index_e.html下载。

FatFs的结构图如下所示：其中L1为物理的存储设备本实验具体为SD卡，RTC可以在创建和修建文件时加入时间信息；L2层位硬件接口驱动层，上一章节的实验就是L2层的实验，L3层为本实验新增的内容，本实验L4层调用L3层的进口函数实现在串口输出SD卡里的文件列表的功能。



7.56.4 硬件设计

注：本实验的硬件设计与上一章节“SD卡访问实验”完全一样，只是增加了软件的文件系统，所以本小节内容请参考“SD卡访问实验”对应部分。

7.56.5 软件设计

注：本实验的软件设计有部分与上一章节“SD卡访问实验”完全重复，所以本小节只介绍软件增加的内容，其余内容请参考“SD卡访问实验”对应部分。

```
123 int main(void)
124 {
125 #ifdef DEBUG
126     debug();
127 #endif
128
129     RCC_Configuration();
130     InterruptConfig();
131     /* Configure the systick */
132     SysTick_Configuration();
133
134     /* Initialize the LCD */
135     STM3210E_LCD_Init();
136
137     /* Clear the LCD */
138     LCD_Clear(Black);
139
140     /*初始化并设置SDIO接口*/
141     Status = SD_InitAndConfig();
142
143     /* Clear the LCD */
144     LCD_Clear(Black);
145
146     LCD_DisplayStringLine(Line3, "      Waiting....      ", White, Black);
147     LCD_DisplayStringLine(Line4, "      Read SD Card...      ", White, Black);
148
149     Delay(1000);
150     ReadSDFile();
151     Delay(100);
```

代码分析 1：分析函数 SD_InitAndConfig()

函数中调用 SD_Init() 函数初始化 SD 卡，通过返回值判断初始化是否成功。TFT 彩屏显示信息。

```
080 SD_Error SD_InitAndConfig(void)
081 {
082     Status = SD_Init();
083     if (Status == SD_OK)
084     {
085         LCD_DisplayStringLine(Line0, "SD_Init success", White, Black);
086     }
087     else
088     {
089         LCD_DisplayStringLine(Line0, "SD_Init fail", White, Black);
090     }
091     Delay(200);
092     if (Status == SD_OK)
093     {
094         /*----- Read CSD/CID MSD registers -----*/
095         Status = SD_GetCardInfo(&SDCardInfo);
096     }
097
098     if (Status == SD_OK)
099     {
100         /*----- Select Card -----*/
101         Status = SD_SelectDeselect((u32) (SDCardInfo.RCA << 16));
102     }

```

代码分析 2：函数 ReadSDFile(void)

```
039 void ReadSDFile(void)
040 {
041     // FIL file;
042     FILINFO finfo;
043     DIR dirs;
044     int i_name=0;
045     // char *fn;
046     char path[50]={"\""};
047     disk_initialize(0);
048     f_mount(0, &fs);
049     res = f_opendir(&dirs, path);
050     if (res == FR_OK)
051     {
052         while (f_readdir(&dirs, &finfo) == FR_OK)
053         {
054             if (finfo.fattrib & AM_ARC)
055             {
056                 if (!finfo.fname[0]) //文件名不为空，如果为空，则表明该目录-
057                     break;
058                 res = f_open(&fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ);
059                 stringcopy(buff_filename[i_name], (BYTE*)finfo.fname);
060                 i_name++;
061             //     res = f_read(&fsrc, &buffer, 50, &br);
062             f_close(&fsrc);
063         }
064     }
065 }
066 }
```

- 1) 接下来开始 f_opendir(&dirs, path)就开始用到了真正的 FAT 文件系统的内容了，激动人心的时刻即将到来，PATH 为路径，比如打开一个磁盘，打开一个文件夹，把这个文件夹的相关信息给到 dir 这个结构体进行保存，接下来看下面的代码：

```
FRESULT f_opendir (
    DIR *dj,                  /* Pointer to directory object to create */
    const XCHAR *path         /* Pointer to the directory path */
)
{
    FRESULT res;
    NAMEBUF(sfn, lfn);
    BYTE *dir;

    res = auto_mount(&path, &dj->fs, 0);
    if (res == FR_OK) {
        INITBUF((*dj), sfn, lfn);
        res = follow_path(dj, path);           /* Follow the path
                                                    /* Follow completed
        if (res == FR_OK) {
            dir = dj->dir;
            if (dir) {                         /* It is not the ro

```

可以看到这里的 auto_mount () 函数是将我们要打开的文件夹目录路径 path 挂在到 DIR 这个文件系统的结构体里，这里因为文件系统是一个比较庞大的系统，这里我们后续出其他神州开发板版本的时候再升级这个文件系统的章节部分。

Auto_mount 就是挂在，把磁盘需要访问的目录挂在到软件文件系统的 struct 中，然后方便在代码里对该文件夹里的文件进行访问，挂住就相当于把这个 path 指定的文件夹的内容信息，按照文件的系统的协议进行匹配。

- 2) 这几行的代码的意思分别是，读取这个目录的值，因为 f_opendir () 函数执行完毕之后，已经证明这个文件夹目录已经存在了，现在来读取这个路径下的内容，看看这个路径下有些什么东西；
- 3) 下面这几个函数也是关键的

```
while (f_readdir(&dirs, &finfo) == FR_OK)
{
    if (finfo.fattrib & AM_ARC)
    {
        if (!finfo.fname[0]) //文件名不为空，如果为空，则表明该目录下面的文件已经读完了
            break;
        res = f_open(&fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ);
        stringcopy(buff_filename[i_name], (BYTE*)finfo.fname);
        i_name++;
        res = f_read(&fsrc, &buffer, 50, &br);
        f_close(&fsrc);
    }
}
```

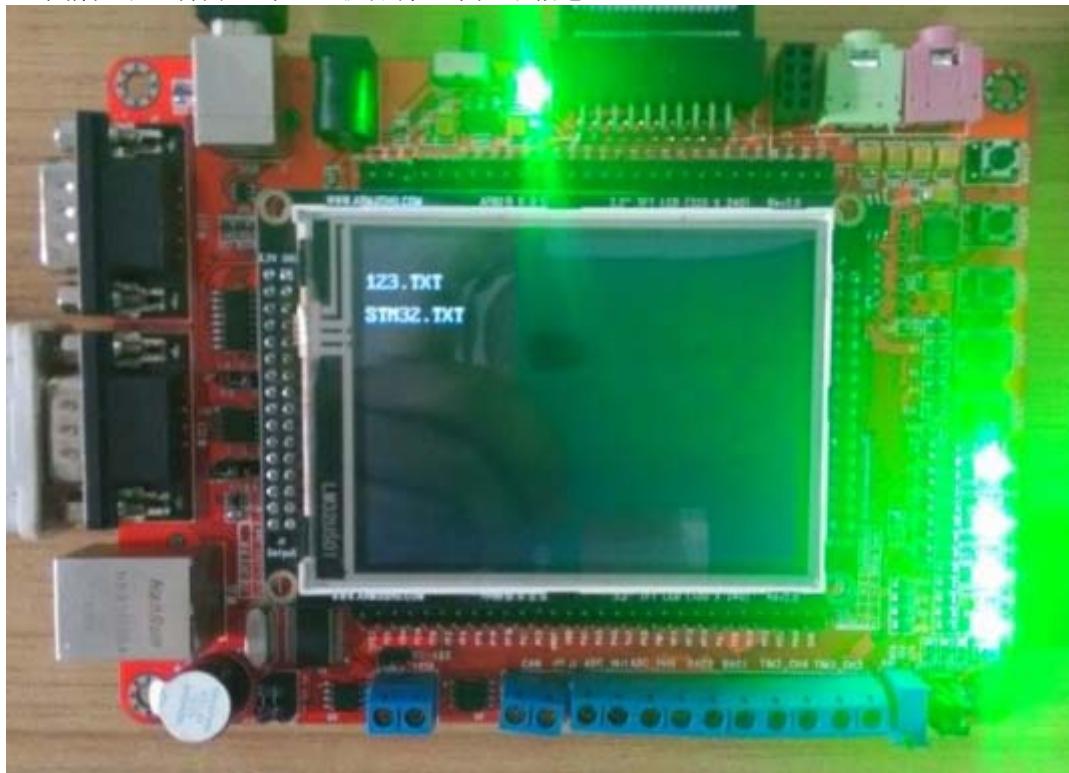
f_open () 函数是打开一个文件；FA_OPEN_EXISTING | FA_READ 这里描述权限，表示打开一个已经存在的文件，并且这个文件可以有被读的权限；

7.56.6 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按[3.5如何在MDK开发环境中使用JLINK在线调试](#)小节进行操作。

7.56.7 实验现象

正常情况下，神舟III号TFT彩屏将显示如下信息：



7.57 SD卡读卡器实验

7.57.1 SD卡读卡器实验的意义与作用

神舟III号开发板资源丰富，关于USB部分，除了前一个例程介绍的，如何通过USB接口访问神舟III号板载的NAND Flash以外，本节将介绍，如何通过USB接口访问神舟III号上插入的SD卡。即用STM32处理器实现我们常说的SD卡读卡器。

通过本实验，我们将对前一例程中使用的USB接口更进一步的认识，同时，我们还可以对SD卡有一个更深入的了解。本实验参考ST的Mass_Storage例程，针对神舟III号的硬件修改代码设计，使用SDIO方式来读写SD卡，实现SD读卡器功能。

7.57.2 试验原理

SD卡读卡器实验原理简介：首先程序运行后，初始化实验需要使用的到串口和GPIO管脚，然后初始化访问SD卡要使用的SDIO接口与USB接口。检测SD卡在，待SD卡插入之后，就开始USB的配置，在配置成功之后既可以在电脑上发现可移动磁盘了。

其中DS2和DS3 LED用于指示初始化过程，DS1用来指示USB正在读写SD卡。

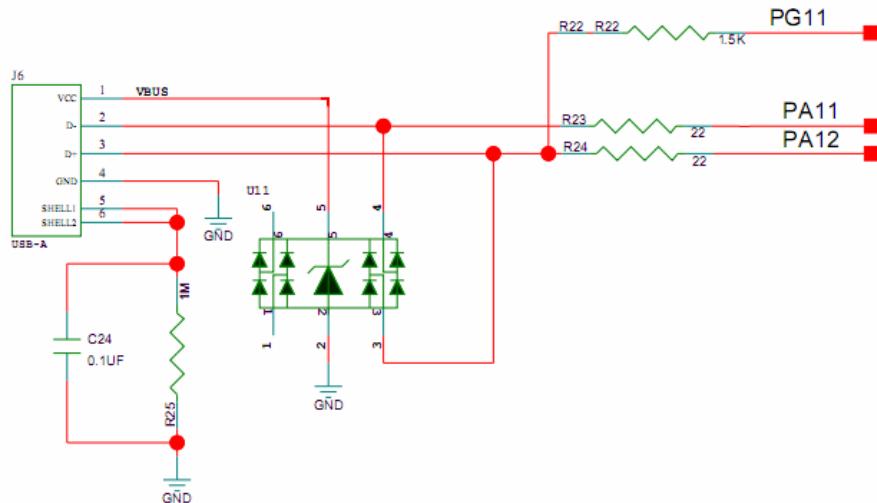
7.57.3 硬件设计

SD卡读卡器实验要用到的硬件资源有：

- 串口 1：串口 1 在本实验中打印程序运行过程中的提示信息。关于串口，在前面的实验已经进行了详细的讲解，在这里就不在重复。
具体见 4.5 串口 1 的发送与接收实验。
- LED 指示灯：LED 指示灯主要用于指示程序运行状态。
- SD 卡座：神舟 III 号最大支持 2G 的 SD 卡
- USB 接口：使用 USB 接口与电脑连接，将 SD 卡插入神舟 III 号 SD 卡座后，运行程序，可以在电脑上检测到一个 U 盘设备。并可以进行读写操作。

SD 卡读卡器实验使用的资源主要是 USB 接口和 SD 卡座两部分，分别如下图所示：

4.20.3.1.USB 接口硬件原理



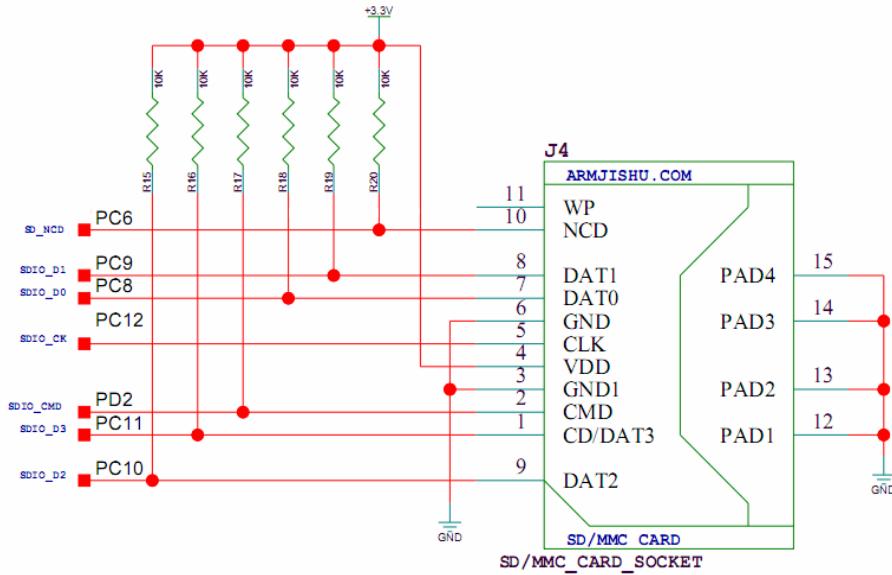
上图中的 J6 为神舟 III 号上的 USB 接口 B 型座。U11 为 USB 接口 ESD 防护器件。, 满足 ESD 防护标准 IEC61000-4-2(ESD 15kV air, 8kV Contact) 。

STM32F103ZET 处理器管脚功能具体如下

| 管脚 | 功能描述 |
|------|--|
| PA11 | USB 接口差分信号的负端 |
| PA12 | USB 接口差分信号的正端 |
| PG11 | USB 接口上拉控制管脚, 用于判断是否有设备插入, 当 PG11 输出高电平时, 电脑上检测到一个全速 USB 设备。 |

4.20.3.2.SD 卡座硬件原理

神舟III号开发板载标准的SD卡座，采用SDIO接口通信，通过这个接口，我们就可以外扩容量存储设备，可以用来记录数据，存放音乐文件等。其原理图如下：

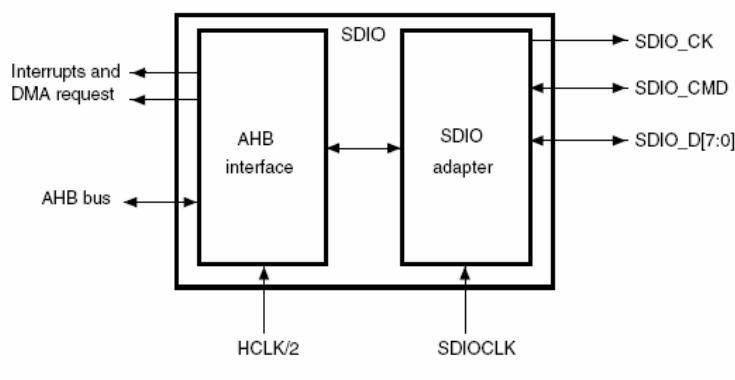


STM32F103ZET6 处理器的 SD/SDIO MMC 卡主机模块(SDIO)在 AHB 外设总线和多媒体卡(MMC)、SD 存储卡、SDIO 卡和 CE-ATA 设备间提供了操作接口。

SDIO 的主要功能如下：

- ◆ 与多媒体卡系统规格书版本 4.2 兼容。支持三种不同数据总线模式：1 位(默认)、4 位和 8 位。
- ◆ 与较早的多媒体卡系统规格版本全兼容(向前兼容)。
- ◆ 与 SD 存储卡规格版本 2.0 全兼容。
- ◆ 与 SD I/O 卡规格版本 2.0 全兼容：支持良种不同的数据总线模式：1 位(默认)和 4 位。
- ◆ 完全支持 CE-ATA 功能(与 CE-ATA 数字协议版本 1.1 全兼容)。
- ◆ 8 位总线模式下数据传输速率可达 48MHz。
- ◆ 数据和命令输出使能信号，用于控制外部双向驱动器。

SDIO 的逻辑框图如下：



ai14740

在神舟 III 号中，使用的是 SDIO 接口来读写访问 SD 卡，其中 SDIO 工作在 4 位数据模式，具体的管脚功能如下表。

| 管脚 | 功能 | 描述 |
|------|----------|------------------------------|
| PC6 | SC_NCD | SD 卡在位检测 |
| PC8 | SDIO_D0 | 多媒体卡/SD/SDIO 卡数据。双向数据总线 |
| PC9 | SDIO_D1 | |
| PC10 | SDIO_D2 | |
| PC11 | SDIO_D3 | |
| PC12 | SDIO_CK | 多媒体卡/SD/SDIO 卡时钟。从主机至卡的时钟线 |
| PD2 | SDIO_CMD | 多媒体卡/SD/SDIO 卡命令。双向的命令/响应信号线 |

4.20.3.3.SD 卡介绍

SD卡 (Secure Digital Memory Card) 中文翻译为安全数码卡，是一种基于半导体快闪记忆器的新一代记忆设备，它被广泛地于便携式装置上使用，例如数码相机、个人数码助理(PDA)和多媒体播放器等。SD卡由日本松下、东芝及美国SanDisk公司于1999年8月共同开发研制。大小犹如一张邮票的SD记忆卡，重量只有2克，但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。

SD卡一般支持2种操作模式：

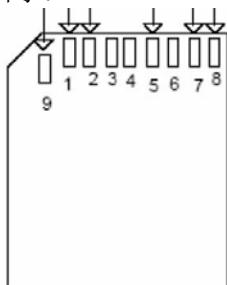
1， SD卡模式；

2， SPI模式；

主机可以选择以上任意一种模式同SD卡通信，SD卡模式允许4线的高速数据传输。SPI模式允许简单的通过SPI接口来和SD卡通信，这种模式同SD卡模式相比就是丧失了速度。

在神舟III号中，是使用SDIO接口来访问SD卡，即所说的SD卡模式。

SD卡的引脚排序如下图所示：



SD卡引脚功能描述如下表所示：

| 针脚 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------|---------|------|-----|-----|-----|-----|------|------|------|
| SD卡模式 | CD/DAT3 | CMD | VSS | VCC | CLK | VSS | DAT0 | DAT1 | DAT2 |
| SPI模式 | CS | MOSI | VSS | VCC | CLK | VSS | MISO | NC | NC |

7.57.4 软件设计

4.20.4.1.GPIO 与串口初始化

在SD卡读卡器实验中,使用了神舟III号的LED灯指示程序运行状态,而串口则输出提供信息,这一些硬件资源,再前面已经详细讲解过,因此在这里,只是简单的介绍他们的初始化程序。

串口1 初始化

```
/* USARTx configured as follow:  
   - BaudRate = 115200 baud  
   - Word Length = 8 Bits  
   - One Stop Bit  
   - No parity  
   - Hardware flow control disabled (RTS and CTS signals)  
   - Receive and transmit enabled  
  
 */  
USART_InitStructure.USART_BaudRate = 115200;  
USART_InitStructure.USART_WordLength = USART_WordLength_8b;  
USART_InitStructure.USART_StopBits = USART_StopBits_1;  
USART_InitStructure.USART_Parity = USART_Parity_No;  
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;  
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;  
  
STM_EVAL_COMInit(COM1, &USART_InitStructure);
```

◆ GPIO 初始化程序

```
GPIO_InitTypeDef GPIO_InitStructure;  
  
/* 配置神舟III号LED灯使用的GPIO管脚模式*/  
RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE); /*使能LED灯使用的GPIO时钟*/  
  
GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
  
GPIO_Init(GPIO_LED_PORT, &GPIO_InitStructure); /*神州III号使用的LED灯相关的GPIO口初始化*/  
  
GPIO_SetBits(GPIO_LED_PORT, DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN); /*关闭所有的LED指示灯*/  
  
/* 配置神舟III号USB上拉电阻使用的GPIO管脚模式*/  
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_DISCONNECT, ENABLE); /*使能USB连接控制的GPIO时钟*/  
GPIO_InitStructure.GPIO_Pin = USB_DISCONNECT_PIN;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  
  
GPIO_Init(USB_DISCONNECT, &GPIO_InitStructure);
```

4.20.4.2.MAL (媒体接入层) 初始化

MAL初始化函数MAL_Init (uint8_t lun) 主要实现SD接口的初始化，获取SD卡的信息等。

```
status = SD_Init(); //SD卡接口初始化
status = SD_GetCardInfo(&SDCardInfo); //获取SD卡信息
status = SD_SelectDeselect((uint32_t)(SDCardInfo.RCA << 16));
status = SD_EnableWideBusOperation(SDIO_BusWide_4b); //设置SDIO接口数据宽度
status = SD_SetDeviceMode(SD_DMA_MODE); //设置工作模式
```

SD卡的典型初始化过程如下：

- 初始化与SD卡连接的硬件条件；
- 上电延时 (>74个CLK)；
- 复位卡；
- 激活卡，内部初始化并获取卡类型；
- 查询OCR，获取供电状况 ()；
- 是否使用CRC；
- 设置读写块数据长度；
- 读取CSD，获取存储卡的其他信息)；
- 发送8CLK后，禁止片选；

这样我们就完成了对SD卡的初始化，这里面我们一般设置读写块数据长度为256个字节，并禁止使用CRC。在完成了初始化之后，就可以开始读写数据了。

4.20.4.3. 中断配置

在SD读卡器实验中，使用到了USB接口和SDIO接口，程序运行时，需要对他们的中断组和中断优先级进行配置，一般来说，越需要及时响应的中断设置的中断组级别越高（0级最高），同一组内的中断优先级也越高（0是同一组中断内的最高优先级）。如下为SD卡读卡器的中断优先级配置。

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn; //USB 低优先级配置
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

NVIC_InitStructure.NVIC_IRQChannel = USB_HP_CAN1_TX_IRQn; //USB高优先级配置
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

#ifndef USE_STM3210E_EVAL
NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn; //SDIO 优先级配置
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
#endif /* USE_STM3210E EVAL */
```

如上设计中，SDIO_IRQn的优先级最高，当SDIO_IRQn产生中断时，可以抢占其他中断，优先执行，主要是它的NVIC_IRQChannelPreemptionPriority比其他的中断要高。而另两个USB的中断，他们的优先级在相同的组里，USB的两个中断同时产生时，处理器自动优先执行USB_HP_CAN1_TX_IRQn中断。

4.20.4.4. USB 接口初始化

USB接口初始化在前面已经进行了详细的实验，在本实验中，就不再详细描述，相关的初始化代码如下：

```
/*USB接口初始化*/  
Set_USBClock(); //设置USB接口时钟  
USB_Init(); //USB接口初始化  
while (bDeviceState != CONFIGURED)  
{  
    ; //reserved  
}  
USB_Configured_LED(); //设置USB接口的LED指示灯状态
```

当USB接口初始化完成以后，LED灯DS3将常亮。

4.20.4.4.SD 卡数据读写

SD卡部分，最重要的就是2个函数，一个MSD_WriteBuffer函数，用于向SD卡写入数据，当你要COPY文件到SD卡的时候，就是由这个函数完成的。另外一个是MSD_ReadBuffer函数，该函数用于读取SD卡上面的数据。

SD卡读取数据，具体过程如下：

- 发送MSD_READ_SINGLE_BLOCK(CMD17);
- 接收卡响应R1;
- 接收数据起始令牌0XFE;
- 接收数据;
- 接收2个字节的CRC，如果没有开启CRC，这两个字节在读取后可以丢掉。
- 8CLK之后禁止片选；

```
uint8_t MSD_ReadBlock(uint8_t* pBuffer, uint32_t ReadAddr, uint16_t NumByteToRead)
{
    uint32_t i = 0;
    uint8_t rvalue = MSD_RESPONSE_FAILURE;

    /* MSD chip select low */
    MSD_CS_LOW();
    /* Send CMD17 (MSD_READ_SINGLE_BLOCK) to read one block */
    MSD_SendCmd(MSD_READ_SINGLE_BLOCK, ReadAddr, 0xFF);

    /* Check if the MSD acknowledged the read block command: R1 response (0x00: no errors) */
    if (!MSD.GetResponse(MSD_RESPONSE_NO_ERROR))
    {
        /* Now look for the data token to signify the start of the data */
        if (!MSD.GetResponse(MSD_START_DATA_SINGLE_BLOCK_READ))
        {
            /* Read the MSD block data : read NumByteToRead data */
            for (i = 0; i < NumByteToRead; i++)
            {
                /* Save the received data */
                *pBuffer = MSD_ReadByte();
                /* Point to the next location where the byte read will be saved */
                pBuffer++;
            }
            /* Get CRC bytes (not really needed by us, but required by MSD) */
            MSD_ReadByte();
            MSD_ReadByte();
            /* Set response value to success */
            rvalue = MSD_RESPONSE_NO_ERROR;
        }
    }

    /* MSD chip select high */
    MSD_CS_HIGH();
    /* Send dummy byte: 8 Clock pulses of delay */
    MSD_WriteByte(DUMMY);
    /* Returns the reponse */
    return rvalue;
} ? end MSD_ReadBlock ?
```

SD卡的写入读数据差不多，写数据通过CMD24来实现，具体过程如下：

- 发送MSD_WRITE_BLOCK(CMD24);
- 接收卡响应R1;
- 发送写数据起始令牌0XFE;
- 发送数据;
- 发送2字节的伪CRC;
- 8CLK之后禁止片选;

```
uint8_t MSD_WriteBuffer(uint8_t* pBuffer, uint32_t WriteAddr, uint32_t NumByteToWrite)
{
    uint32_t i = 0, NbrOfBlock = 0, Offset = 0;
    uint8_t rvalue = MSD_RESPONSE_FAILURE;
    NbrOfBlock = NumByteToWrite / BLOCK_SIZE; /* Calculate number of blocks to write */
    MSD_CS_LOW(); /* MSD chip select low */
    /* Data transfer */
    while (NbrOfBlock --)
    {
        /* Send CMD24 (MSD_WRITE_BLOCK) to write blocks */
        MSD_SendCmd(MSD_WRITE_BLOCK, WriteAddr + Offset, 0xFF);
        /* Check if the MSD acknowledged the write block command: R1 response (0x00: no errors) */
        if (MSD.GetResponse(MSD_RESPONSE_NO_ERROR))
        {
            return MSD_RESPONSE_FAILURE;
        }
        MSD_WriteByte(DUMMY); /* Send dummy byte */
        MSD_WriteByte(MSD_START_DATA_SINGLE_BLOCK_WRITE); /* Send the data token to signify the start of data */
        /* Write the block data to MSD : write count data by block */
        for (i = 0; i < BLOCK_SIZE; i++)
        {
            MSD_WriteByte(*pBuffer); /* Send the pointed byte */
            pBuffer++; /* Point to the next location where the byte read will be saved */
        }
        Offset += 512; /* Set next write address */
        /* Put CRC bytes (not really needed by us, but required by MSD) */
        MSD_ReadByte();
        MSD_ReadByte();
        /* Read data response */
        if (MSD.GetDataResponse() == MSD_DATA_OK)
        {
            rvalue = MSD_RESPONSE_NO_ERROR; /* Set response value to success */
        }
        else
        {
            rvalue = MSD_RESPONSE_FAILURE; /* Set response value to failure */
        }
    } ? end while NbrOfBlock-- ?
    MSD_CS_HIGH(); /* MSD chip select high */
    MSD_WriteByte(DUMMY); /* Send dummy byte: 8 Clock pulses of delay */
}
```

以上是一个典型的写SD卡过程。关于SD卡的介绍，我们就介绍到这里，更详细的介绍请参考SD卡的参考资料。

7.57.5 下载与测试

在 [神舟III号光盘\编译好的固件\20.USB读卡器](#)实验目录下的USB读卡器.hex文件即为前面我们分析的USB读卡器实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟III号开发板中，观察运行效果。

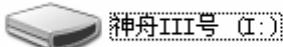
如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.57.6 实验现象

将程序下载到神舟III号后，插入SD卡（最大支持2G），将神舟III号的USB接口与电脑的电脑连接后，重新上电运行，电脑上将提示发现USB设备。



进入电脑的设备管理器，可以看到USB SD读卡器的详细信息。



与此同时，神舟III号的串口1（波特率115200）将打印如下信息。

神舟III号

```
USART Printf Example: retarget the C library printf function to the USART
#####
www.ARMJISHU.COM! #####
(Dec 25 2010 - 15:04:35)
```



```
www.ARMJISHU.COM use __STM32F10X_STDPERIPH_VERSION 3.3.0
```

```
产品内部Flash大小为：512K字节！ www.armjishu.com
```

```
--DS1闪烁：正在进行读写操作
--DS2亮：表示MAL配置完成
--DS3亮：USB接口初始化完成
```

神舟III号的LED灯也将指示USB读卡器的状态。

| LED指示灯 | 含义 |
|--------|---|
| DS1闪烁 | USB读卡器正在进行读写操作，当我们往SD卡内拷贝数据或者从SD卡读取数据，将看到DS1闪烁。 |
| DS2亮 | MAL配置完成指示灯，正常情况下，DS2亮。 |
| DS3亮 | USB接口初始化完成指示灯，正常情况下，DS3亮。 |

7.58 音频播放试验

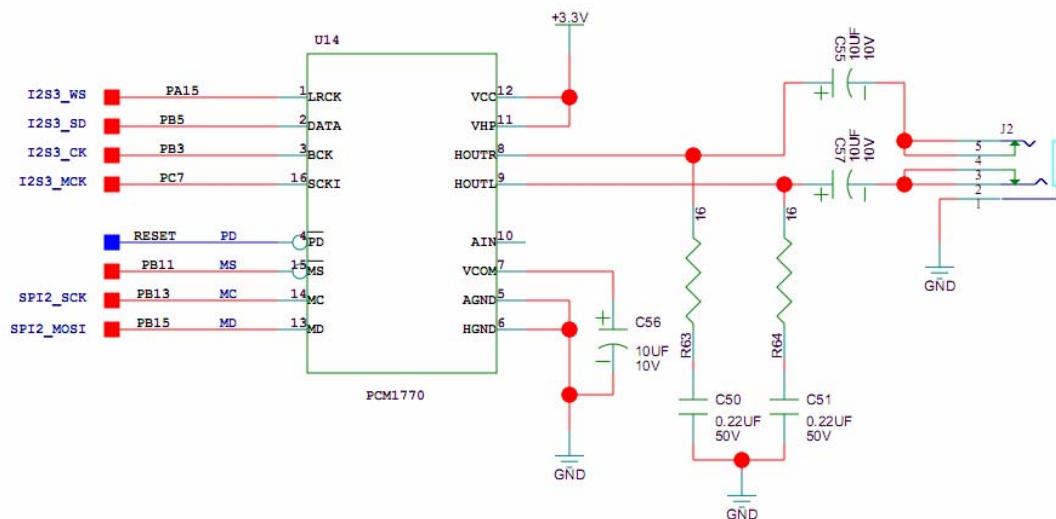
7.58.1 实验原理

本实验主要是初始化DA芯片PCM1770后，处理器读取预先放置在处理器内部的音频文件，判断音频文件是否合法，并依据音频文件的格式设置I2S3接口的参数，然后处理器采用中断方式通过I2S3接口重复播放该音频文件。

7.58.2 硬件设计

SD卡读卡器实验要用到的硬件资源有：

- 串口 1：串口 1 在本实验中打印程序运行过程中的提示信息。关于串口，在前面的实验已经进行了详细的讲解，在这里就不在重复。
具体见 4.5 串口 1 的发送与接收实验。
- PCM1770：将从 I2S3 接口传过来的音频音频信号输出到耳麦接口。



D/A 芯片 PCM1770 通过 I2S3 接口与 STM32F103ZET6 处理器连接。音频信号通过 I2S3 接口传到 D/A 芯片，转换成音频信号播出。相关管脚定义如下：

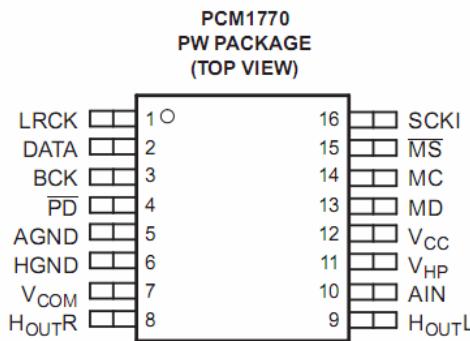
| GPIO 管脚 | 功能描述 | 说明 |
|---------|----------|--|
| PA15 | I2S3_WS | 左右通道时钟，频率和音频采样速率一致。该信号与 JTAG 接口的 JTDO 信号复用同一个管脚。 |
| PB3 | I2S3_CK | 串行比特时钟。该信号与 JTAG 接口的 JTDO 信号复用同一个管脚。 |
| PB5 | I2S_SD | 串行音频信号。 |
| PC7 | I2S3_MCK | 系统时钟输入 |

STM32F103ZET6 处理器通过 SPI2 接口访问 DA 芯片 PCM1770，对它内部数据进行读取和配置。相关管脚定义如下：

| GPIO 管脚 | 功能描述 | 说明 |
|---------|-----------|------------------|
| PB11 | SPI2_NSS | SPI2 接口片选信号 |
| PB13 | SPI2_SCK | SPI2 接口 SCK 时钟信号 |
| PB15 | SPI2_MOSI | SPI2 接口的 MOSI 信号 |

4.21.2.1.DAC 芯片说明

在神舟 III 号中，使用了 TI 公司推出的低电压，低功耗带耳机放大的 DAC 芯片 PCM1770，来实现音频的 DA 转换。



| TERMINAL NAME NO. | I/O | DESCRIPTION |
|-------------------|-----|---|
| AGND 5 | - | Analog ground. This is a return for V_{CC} . |
| AIN 10 | I | Monaural analog signal mixer input. The signal can be mixed with the outputs of the L- and R-channel DACs. |
| BCK 3 | I/O | Serial bit clock. Clocks the individual bits of the audio data input, DATA. In the slave interface mode, this clock is input from an external device. In the master interface mode, the PCM1770 device generates the BCK output to an external device. |
| DATA 2 | I | Serial audio data input |
| HGND 6 | - | Analog ground. This is a return for V_{HP} . |
| HOUTL 9 | O | L-channel analog signal output of the headphone amplifiers |
| HOUTR 8 | O | R-channel analog signal output of the headphone amplifiers |
| LRCK 1 | I/O | Left and right clock. Determines which channel is being input on the audio data input, DATA. The frequency of LRCK must be the same as the audio sampling rate. In the slave interface mode, this clock is input from an external device. In the master interface mode, the PCM1770 device generates the LRCK output to an external device. |
| MC 14 | I | Mode control port serial bit clock input. Clocks the individual bits of the control data input, MD. |
| MD 13 | I | Mode control port serial data input. Controls the operation mode on the PCM1770 device. |
| MS 15 | I | Mode control port select. The control port is active when this terminal is low. |
| PD 4 | I | Reset input. When low, the PCM1770 device is powered down, and all mode control registers are reset to default settings. |
| SCKI 16 | I | System clock input |
| V_{CC} 12 | - | Power supply for all analog circuits except the headphone amplifier. |
| V_{COM} 7 | - | Decoupling capacitor connection. An external 10- μ F capacitor connected from this terminal to analog ground is required for noise filtering. Voltage level of this terminal is 0.5 V_{HP} nominal. |
| V_{HP} 11 | - | Analog power supply for the headphone amplifier circuits. The voltage level must be the same as V_{CC} . |

7.58.3 软件设计

在本实验中，采用的是 I2S 中断方式来播放音乐，主程序实现如下

```
/*LED灯初始化*/
LED_Config();
/*串口初始化*/
Serial_Init();

printf("\n\r-----");
printf("\n\r神舟III号音频播放实验");
printf("\n\r-----");
printf("\n\r串口初始化完成");
/* I2S GPIO接口配置 */
I2S_GPIO_Config();

/*SPI2接口初始化*/
SPI2_Config();
PCM1770_CS_config();
SPI2_Init_For_PCM1770();
printf("\n\rPCM1770 SPI接口初始化完成");

/*中断向量与中断优先级配置*/
InterruptConfig();
/*SysTick 配置*/
SysTick_Configuration();

I2S_CODEC_Init(OutputDevice_SPEAKER, AUDIO_FILE_ADDRESS); //I2C CODEC初始化
I2S_CODEC_Play(1);

while(1)
{
    ;
}
```

在上面程序中，I2S_Codec_init() 函数主要是实现音源文件是否正确，并设置合适的频率播放音源文件。在此就不进行详细讲解，有兴趣的朋友可以查阅工程文件，了解函数的具体实现，接下来我们对一些基本的函数进行分析。

➤ LED GPIO 与串口初始化

在音频播放实验中，LED 主要用于指示程序运行状态，因此，我们首先运行 LED_Config 函数，将 LED 使用的 GPIO 接口配置成推挽输出。

```
void LED_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    /* Enable GPIOB, GPIOC and AFIO clock */
    RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE);

    /* LEDs pins configuration */
    GPIO_InitStruct.GPIO_Pin = DS1_PIN | DS2_PIN | DS3_PIN | DS4_PIN ;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIO_LED_PORT, &GPIO_InitStruct);

    GPIO_SetBits(GPIO_LED_PORT, GPIO_LED_ALL);
}
```

为了从串口输出程序运行提示信息，我们需要初始化串口 1，关于串口 1 的配置在前面的例程中已经进行了详细的讲解，在这里，我们就不在进行分析。

➤ I2S 接口初始化

查看神舟 III 号原理图可知, D/A 芯片 PCM1770 通过 I2S3 接口与 STM32F103ZET6 处理器连接。音频信号通过 I2S3 接口传到 D/A 芯片, 转换成音频信号播出。相关管脚定义如下:

| GPIO 管脚 | 功能描述 | 说明 |
|---------|----------|---|
| PA15 | I2S3_WS | 左右通道时钟, 频率和音频采样速率一致。 该信号与 JTAG 接口的 JTDO 信号复用同一个管脚。 |
| PB3 | I2S3_CK | 串行比特时钟。该信号与 JTAG 接口的 JTDI 信号复用同一个管脚。 |
| PB5 | I2S_SD | 串行音频信号。 |
| PC7 | I2S3_MCK | 系统时钟输入 |

由于 SPI3/I2S3 的部分管脚与 JTAG 管脚共享 (SPI3_NSS/I2S3_WS 与 JTDO, SPI3_SCK/I2S3_CK 与 JTDI), 因此这些管脚不受 IO 控制器控制, 他们(在每次复位后)被默认保留为 JTAG 用途。如果用户想把管脚配置给 SPI3/I2S3, 必须(在 DEBUG 时)关闭 JTAG 并切换至 SWD 接口, 或者(在标准应用时)同时关闭 JTAG 和 SWD 接口。

在初始化 I2S 接口的时候, 使用 Remap 功能, 关闭了 JTAG 功能。在使用本程序时, JTAG 接口就不能使用, 但 SWD 调试接口可以正常使用。因此, 在本实验中, 在 MDK 中调试或者使用 J-FLASH 烧录固件的时候, 必须配置成 SWD 模式。

```
void I2S_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    /* Enable GPIOB, GPIOC and AFIO clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC |
                           RCC_APB2Periph_AFIO, ENABLE);

    /* I2S3 SD, CK and WS pins configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* I2S2 MCK pin configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

? end I2S_GPIO_Config ?

➤ SPI 接口初始化

查看神舟 III 号原理图可知, STM32F103ZET6 处理器通过 SPI2 接口访问 DA 芯片 PCM1770, 对它内部数据进行读取和配置。相关管脚定义如下:

| GPIO 管脚 | 功能描述 | 说明 |
|---------|-----------|------------------|
| PB11 | SPI2_NSS | SPI2 接口片选信号 |
| PB13 | SPI2_SCK | SPI2 接口 SCK 时钟信号 |
| PB15 | SPI2_MOSI | SPI2 接口的 MOSI 信号 |

相关的初始化程序如下,SPI2_Config 函数主要是初始化 SPI2 接口使用的 GPIO 管脚。

```
void SPI2_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // Enable SPI1 and GPIOA & GPIOB clocks
    RCC_APB2PeriphClockCmd(RCC_SPI2, ENABLE);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);

    // Set PB13,14,15 as Output push-pull - SCK, MISO and MOSI
    GPIO_InitStructure.GPIO_Pin = SPI2_SCK | SPI2_MOSI | SPI2_MISO;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(SPI2_PORT, &GPIO_InitStructure);
}
```

初始化 PCM1770 使用的 SPI2 接口 CS 信号, 输出高电平。

```
void PCM1770_CS_config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // Enable SPI1 and GPIOA & GPIOB clocks
    RCC_APB2PeriphClockCmd(RCC_PCM1770_CS, ENABLE);

    // Configure PB11 as Output push-pull - PCM1770 Chip select
    GPIO_InitStructure.GPIO_Pin = GPIO_PCM1770_CS;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIO_PCM1770_CS_PORT, &GPIO_InitStructure);

    SPI2_PCM1770_CS_HIGH();
}
```

初始化 SPI 接口, 设置 SPI2 接口的工作模式。

```
void SPI2_Init_For_PCM1770(void)
{
    SPI_InitTypeDef SPI_InitStructure;

    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_16b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_64;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 7;
    SPI_Init(SPI2, &SPI_InitStructure);

    SPI_Cmd(SPI2, ENABLE);
}
```

➤ SysTick 初始化

其中SysTick主要是初始化硬件SysTick定时器，定时产生中断，在本程序中，初始化SysTick为10mS，相关程序如下。

```
void SysTick_Config(void)
{
    /* Setup SysTick Timer for 10 msec interrupts */
    if (SysTick_Config(SystemCoreClock / 100))
    {
        /* Capture error */
        while (1);
    }

    /* Configure the SysTick handler priority */
    NVIC_SetPriority(SysTick_IRQn, 0x0);
}
```

在本实验中，SysTick 定时器的时钟源是 72M，即 SystemCoreClock 为 72000000Hz，所以 SysTick_Config(SystemCoreClock/ 100) 就是10ms时基。

在音频播放实验中，我们采用的是 I2S3 中断方式的方式来实现音频播放。当 I2S3 产生中断时，就进入 I2S3 的中断服务程序，调用音频播放程序 I2s_CODEC_DataTransfer()，因此，我们可以重复听到相同的音频。

InterruptConfig函数按照中断的重要性，进行中断组设置和中断优先级设计，相关代码如下：

```
void InterruptConfig(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Set the Vector Table base address */
    NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0000);

    /* SPI2 IRQ Channel configuration */
    NVIC_InitStructure.NVIC_IRQChannel = SPI3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

在本实验中，相关的中断之后SPI3中断，因此设置中断组为0，中断优先级也为0即可。

I2S3 的中断服务程序如下。当 I2S3 进入中断服务程序时，就调用 I2S_CODEC_DataTransfer()程序播放音乐。

```
void SPI3_IRQHandler(void)
{
    //static int IRQcounter = 0;

    if ((SPI_I2S_GetITStatus(SPI3, SPI_I2S_IT_TXE) == SET))
    {
        /* Send data on the SPI3 and Check the current commands */
        I2S_CODEC_DataTransfer();
    }
}
```

7.58.4 下载与测试

在 [神舟III号光盘\编译好的固件\21音频播放实验](#) 目录下的音频播放实验.hex文件即为前面我们分析的音频播放实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟III号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

注意：由于本实验中,I2S3与JTAG功能接口有复用，为了使用I2S3接口，我们需要禁止JTAG接口，在调试和下载时，请使用SWD接口。

7.58.5 试验现象

将程序下载到神舟III号后，将耳机插入音频口（J2）以后，重新上电运行，正常情况下，从耳机里可以听到悦耳的歌声。同时串口1（波特率115200）输出如下打印信息，提示程序运行情况。

7.59 硬件CRC循环冗余检验实验

CRC 是(Cyclic Redundancy Check)的缩写，意思是循环冗余校验。CRC 循环冗余校验技术主要应用于核实数据远程传输或者数据存储的正确性和完整性。神舟系列开发板使用的 STM32 芯片都内置了一个硬件的 CRC 计算模块，本章节我们讲述该硬件模块的使用。

7.59.1 意义与作用

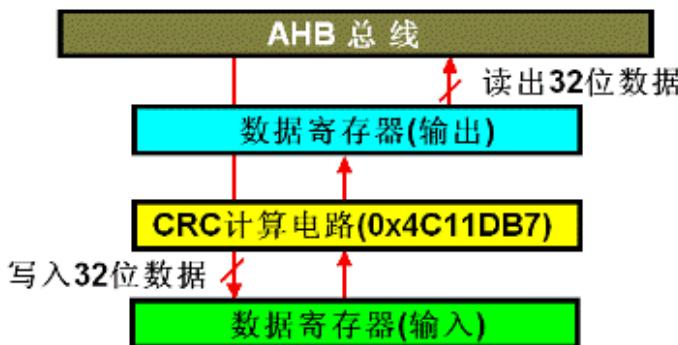
CRC 校验可以硬件完成，可以软件实现。本多处理器内部没有硬件 CRC 校验电路，只能使用软件实现，但是软件实现需要占用 CPU 和 RAM 资源，而且速度稍慢。神舟系列开发板使用的 STM32 芯片内置了一个硬件的 CRC 计算模块，可以在通信的检测错误和数据完整性方面发挥优异的性能，本章节我们讲述该硬件模块的使用。

7.59.2 实验原理

CRC 循环冗余校验计算单元是根据固定的生成多项式得到任一 32 位全字的 CRC 计算结果。在其他的应用中，CRC 技术主要应用于核实数据传输或者数据存储的正确性和完整性。标准 EN/IEC 60335-1 即提供了一种核实闪存存储器完整性的方法。CRC 计算单元可以在程序运行时计算出软件的标识，之后与在连接时生成的参考标识比较，然后存放在指定的存储器空间。

所有的 STM32 芯片都内置了一个硬件的 CRC 计算模块，可以很方便地应用到需要进行通信的程序中，这个 CRC 计算模块使用常见的、在以太网中使用的计算多项式：

$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$ 写成十六进制就是：0x4C11DB7。STM32 芯片的硬件 CRC 计算单元框图如下：



图表 18 CRC 计算单元框图

使用这个内置 CRC 模块的方法非常简单，既首先复位 CRC 模块(设置 CRC_CR=0x01)，这个操作把 CRC 计算的余数初始化为 0xFFFFFFFF；然后把要计算的数据按每 32 位分割为一组数据字，并逐个地把这组数据字写入 CRC_DR 寄存器(既上图中的绿色框)，写完所有的数据字后，就可以从 CRC_DR 寄存器(既下图中的兰色框)读出计算的结果。

注意：虽然读写操作都是针对 CRC_DR 寄存器，但实际上访问的是不同的物理寄存器。

下面是用 C 语言描述的这个计算模块的算法，大家可以把它放在通信的另一端，对通信的正确性进行验证：

```
DWORD dwPolynomial = 0x04c11db7;
DWORD cal_crc (DWORD *ptr, int len)
{
    DWORD    xbit;
    DWORD    data;
    DWORD    CRC = 0xFFFFFFFF;      // init
    while (len--)
    {
        xbit = 1 << 31;
        data = *ptr++;
        for (int bits = 0; bits < 32; bits++)
        {
            if (CRC & 0x80000000)
            {
                CRC <<= 1;
                CRC ^= dwPolynomial;
            }
            else
                CRC <<= 1;

            if (data & xbit)
                CRC ^= dwPolynomial;

            xbit >>= 1;
        }
    }
    return CRC;
} ? end cal_crc ?
```

有几点需要说明：

- 1) 上述算法中变量 CRC，在每次循环结束包含了计算的余数，它始终是向左移位(既从最低位向最高位移动)，溢出的数据位被丢弃。
- 2) 输入的数据始终是以 32 位为单位，如果原始数据少于 32 位，需要在低位补 0，当然也可以高位补 0。
- 3) 假定输入的 DWORD 数组中每个分量是按小端存储。
- 4) 输入数据是按照最高位最先计算，最低位最后计算的顺序进行。

例如：

如果输入 0x44434241，内存中按字节存放的顺序是：0x41, 0x42, 0x43, 0x44。计算的结果是：
0xCF534AE1

如果输入 0x41424344，内存中按字节存放的顺序是：0x44, 0x43, 0x42, 0x41。计算的结果是：
0xABCF9A63

7.59.3 硬件设计

CRC 计算单元是 STM32 处理器内部硬件组件，这部分不需要硬件电路，这里仅将计算结果在串口打印出来。

7.59.4 软件设计

本实验循环重复计算一段已知数据的 CRC 校验结果并打印，这段已知数据如下：

```
static const uint32_t DataBuffer[BUFFER_SIZE] =
{
    0x00001021, 0x20423063, 0x408450a5, 0x60c670e7, 0x9129a14a, 0xb16bc18c,
    0xd1ade1ce, 0xf1ef1231, 0x32732252, 0x52b54294, 0x72f762d6, 0x93398318,
    0xa35ad3bd, 0xc39cf3ff, 0xe3de2462, 0x34430420, 0x64e674c7, 0x44a45485,
    0xa56ab54b, 0x85289509, 0xf5cfc5ac, 0xd58d3653, 0x26721611, 0x063076d7,
    0x569546b4, 0xb75ba77a, 0x97198738, 0xf7dfe7fe, 0xc7bc48c4, 0x58e56886,
    0x78a70840, 0x18612802, 0xc9ccd9ed, 0xe98ef9af, 0x89489969, 0xa90ab92b,
    0x4ad47ab7, 0x6a961a71, 0x0a503a33, 0x2a12dbfd, 0xfbfbfeb9e, 0x9b798b58,
    0xbb3bab1a, 0x6ca67c87, 0x5cc52c22, 0x3c030c60, 0x1c41edae, 0xfd8fcdec,
    0xad2abd0b, 0x8d689d49, 0x7e976eb6, 0x5ed54ef4, 0x2e321e51, 0x0e70ff9f,
    0xefbedfdd, 0xcffcbf1b, 0x9f598f78, 0x918881a9, 0xb1caa1eb, 0xd10cc12d,
    0xe16f1080, 0x0a130c2, 0x20e35004, 0x40257046, 0x83b99398, 0xa3fb3da,
    0xc33dd31c, 0xe37ff35e, 0x129022f3, 0x32d24235, 0x52146277, 0x7256b5ea,
    0x95a88589, 0xf56ee54f, 0xd52cc50d, 0x34e224c3, 0x04817466, 0x64475424,
    0x4405a7db, 0xb7fa8799, 0xe75ff77e, 0xc71dd73c, 0x26d336f2, 0x069116b0,
    0x76764615, 0x5634d94c, 0xc96df90e, 0xe92f99c8, 0xb98aa9ab, 0x58444865,
    0x78066827, 0x18c008e1, 0x28a3cb7d, 0xdb5ceb3f, 0xfb1e8bf9, 0x9bd8abbb,
    0x4a755a54, 0x6a377a16, 0x0af11ad0, 0x2ab33a92, 0xed0fdd6c, 0xcd4dbdaa,
    0xad8b9de8, 0x8dc97c26, 0x5c644c45, 0x3ca22c83, 0x1ce00cc1, 0xef1fff3e,
    0xdf7caf9b, 0xbfb8fd9, 0x9ff86e17, 0x7e364e55, 0x2e933eb2, 0x0ed11ef0
};
```

关于串口打印相关知识请查看“串口 2 的 Printf 输出实验”章节。

本实验比较简单，下面我们之间看 MAIN 主函数，MAIN 主函数开始首先是初始化串口 2，如下：

```
int main(void)
{
    // 配置串口参数
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    STM32_Shenzhou_COMInit(&USART_InitStructure);

    /* 初始化系统定时器SysTick,每秒中断1000次 */
    SZ_STM32_SysTickInit(1000);

    /* Enable CRC clock */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_CRC, ENABLE);
```

首先调用 SZ_STM32_SysTickInit(1000) 函数初始化系统定时器，调用 RCC_AHBPeriphClockCmd()使能 CRC 时钟。

最后是 while 循环，循环内首先给 CRC 寄存器置初值，CRC_ResetDR 置初值很关键，否则循环打印出的 CRC 结果一次和一次不同，最后是计算已知数据的 CRC 校验结果并打印，如下：

```
while (1)
{
    CRC_ResetDR();

    /* Compute the CRC of "DataBuffer" */
    CRCValue = CRC_CalcBlockCRC((uint32_t *)DataBuffer, BUFFER_SIZE);
    Delay(6000000);
    printf("\n\r The CRC Value of DataBuffer[] is 0x%X!\t www.armjishu.com\n\r", CRCValue);

    /* 将>DataBuffer<的校验结果再做一次校验应该为0 */
    CRCValue = CRC_CalcCRC(CRCValue);

    printf(" The CRC Value of DataBuffer[] and CRCValue is 0x%X(should be 0)!\t \n\r", CRCValue);

    /* 延迟, 间隔 */
    Delay(60000000);
}
```

7.59.5 下载与验证

如果使用JLINK下载固件,请按[3.2如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作。

如果使用串口下载固件,请按[3.3如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

如果在MDK开发环境中,下载编译好的固件或者在线调试,请按[3.5如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

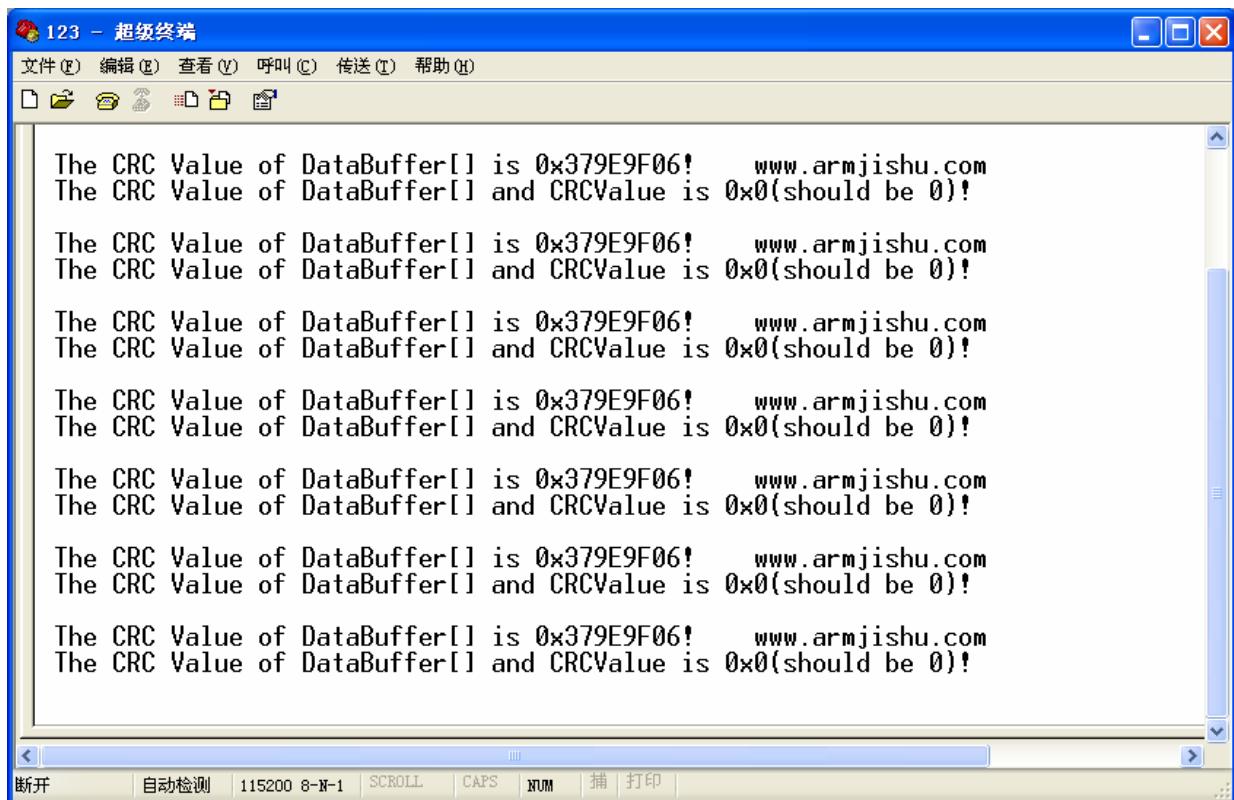
7.59.6 实验现象

将固件下载在神舟III号STM32开发板后,用随板配置的串口线连接神舟III号串口1与电脑的串口,打开超级终端,并按如下参数配置串口。



图表 19 超级终端串口参数配置

以下是下载固件后上电运行神舟 III 号时串口的打印信息,每次校验得到相同结果:



7.60 PVD电源电压监测实验

7.60.1 意义与作用

相信大家都有这样的经历，使用 PC 电脑做文档编辑时突然停电而没有保存文档，重新开机后可能刚才的辛苦付之东流。工业控制领域的主控系统突然断电可能会造成无法挽回的经济损失甚至造成严重的安全事故，这样的意外时一个可靠的工控系统无法容忍的。如果在主控掉电的瞬间能及时执行一些操作，比如及时存盘或者及时关闭正在高速运转的机械设备等，神舟系列 STM32 开发板的处理器内部集成了 PVD，可以帮您实现这个应用。

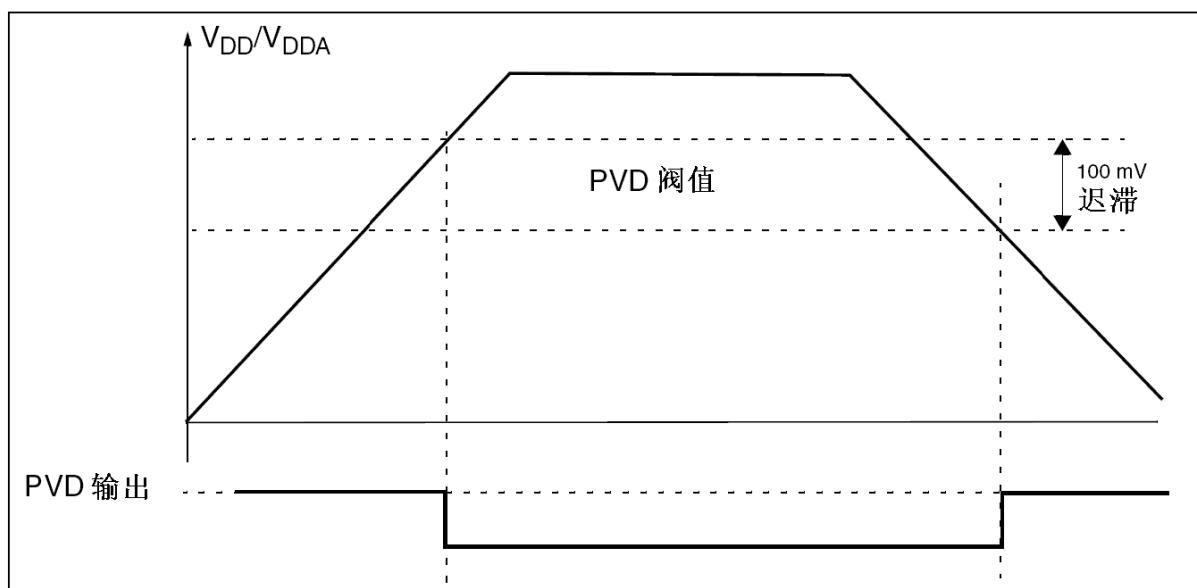
PVD 是 Power supply supervisor 的缩写，意思是可编程电压监测器。

用户可以利用 PVD 对 VDD 电压与电源控制寄存器(PWR_CR)中的 PLS[2:0]位进行比较来监控电源，这几位选择监控电压的阀值。通过设置 PVDE 位来使能 PVD。电源控制/状态寄存器(PWR_CSR)中的 PVDO 标志用来表明 VDD 是高于还是低于 PVD 的电压阀值。该事件在内部连接到外部中断的第 16 线，如果该中断在外部中断寄存器中是使能的，该事件就会产生中断。当 VDD 下降到 PVD 阀值以下和(或)当 VDD 上升到 PVD 阀值之上时，根据外部中断第 16 线的上升/下降边沿触发设置，就会产生 PVD 中断。例如，这一特性可用于用于执行紧急关闭任务。

7.60.2 实验原理

本实验演示神舟系列 STM32 开发板处理器内部的 PVD 电压监控功能，当检测到电压低于 2.9V（可以灵活设置这个阈值）时，将产生 PVD 中断，我们在中断中以串口打印“Bye!”来演示“执行紧急处理任务”。

电源管理器（PVD）



图表 20 STM32 的 PVD 的门限示意图

1) PVD = Programmable Voltage Detector 可编程电压监测器

它的作用是监视供电电压，在供电电压下降到给定的阀值以下时，产生一个中断，通知软件做紧急处理。在给出表格的上半部分就是可编程的监视阀值数据。当供电电压又恢复到给定的阀值以上时，也会产生一个中断，通知软件供电恢复。供电下降的阀值与供电上升的 PVD 阀值有一个固定的差值，这就是表中的 VPVDhyst(PVD 迟滞)这个参数，通过列出的 PVD 阀值数据可以看到这个差别。

引入这个差值的目的是为了防止电压在阀值上下小幅抖动，而频繁地产生中断。

电源管理器 (PVD) 寄存器的值与电压阈值之间的关系：

| 符号 | 参数 | 条件 | 最小值 | 典型值 | 最大值 | 单位 |
|----------------------|----------------|--------------------|--------------------|------|------|----|
| V_{PVD} | 可编程的电压检测器的电平选择 | PLS[2:0]=000 (上升沿) | 2.1 | 2.18 | 2.26 | V |
| | | PLS[2:0]=000 (下降沿) | 2 | 2.08 | 2.16 | V |
| | | PLS[2:0]=001 (上升沿) | 2.19 | 2.28 | 2.37 | V |
| | | PLS[2:0]=001 (下降沿) | 2.09 | 2.18 | 2.27 | V |
| | | PLS[2:0]=010 (上升沿) | 2.28 | 2.38 | 2.48 | V |
| | | PLS[2:0]=010 (下降沿) | 2.18 | 2.28 | 2.38 | V |
| | | PLS[2:0]=011 (上升沿) | 2.38 | 2.48 | 2.58 | V |
| | | PLS[2:0]=011 (下降沿) | 2.28 | 2.38 | 2.48 | V |
| | | PLS[2:0]=100 (上升沿) | 2.47 | 2.58 | 2.69 | V |
| | | PLS[2:0]=100 (下降沿) | 2.37 | 2.48 | 2.59 | V |
| | | PLS[2:0]=101 (上升沿) | 2.57 | 2.68 | 2.79 | V |
| | | PLS[2:0]=101 (下降沿) | 2.47 | 2.58 | 2.69 | V |
| | | PLS[2:0]=110 (上升沿) | 2.66 | 2.78 | 2.9 | V |
| | | PLS[2:0]=110 (下降沿) | 2.56 | 2.68 | 2.8 | V |
| | | PLS[2:0]=111 (上升沿) | 2.76 | 2.88 | 3 | V |
| | | PLS[2:0]=111 (下降沿) | 2.66 | 2.78 | 2.9 | V |
| $V_{PVDrhyst}^{(2)}$ | PVD迟滞 | | | 100 | | mV |
| $V_{POR/PDR}$ | 上电/掉电复位阀值 | 下降沿 | 1.8 ⁽¹⁾ | 1.88 | 1.96 | V |
| | | 上升沿 | 1.84 | 1.92 | 2.0 | V |
| $V_{PDRrhyst}^{(2)}$ | PDR迟滞 | | | 40 | | mV |
| $T_{RSTTEMPO}^{(2)}$ | 复位迟滞时间 | | 1 | 2.5 | 4.5 | ms |

1. 产品的特性由设计保证至最小的数值 $V_{POR/PDR}$ 。

2. 由设计保证，不在生产中测试。

用户可以利用PVD对VDD电压与电源控制寄存器中的PLS[2:0]位进行比较来监控电源，这几位选择监控电压的阀值。通过设置PVDE位来使能PVD。

电源控制/状态寄存器中的PVDO标志用来表明VDD是高于还是低于PVD的电压阀值。该事件在内部连接到外部中断的第16线。当VDD下降到PVD阀值以下和（或）当VDD上升到PVD阀值之上时，根据外部中断第16线的上升/下降边沿触发设置，就会产生PVD中断。这一特性可用于用于执行紧急关闭任务。

7.60.3 硬件设计

PVD 可编程电压监测器单元是 STM32 处理器内部硬件组件，这部分不需要硬件电路，这里仅将计算结果在串口打印出来。

7.60.4 软件设计

我们先从 MAIN 函数看起，如下所示，第一部分是初始化串口等，这部分知识在前面章节已经讲解过，请回顾相关章节的知识。

```
int main(void)
{
    /*串口参数配置*/
    USART_InitStructure.USART_BaudRate = 115200;           /*设置波特率为115200*/
    USART_InitStructure.USART_WordLength = USART_WordLength_8b; /*设置数据位为8位*/
    USART_InitStructure.USART_StopBits = USART_StopBits_1;     /*设置停止位为1位*/
    USART_InitStructure.USART_Parity = USART_Parity_No;        /*无奇偶校验*/
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; /*发送与接收*/
    /*完成串口COM1的时钟配置、GPIO配置，根据上述参数初始化并使能*/
    STM32_Shenzhou_COMInit(&USART_InitStructure);
    printf("\n\r-----\n");
    printf("\n\rWWW.ARmjishu.COM\n");

    SZ_STM32_SysTickInit(1000);
    /* 可编程电压监测器(programmable voltage detector)初始化 */
    SZ_STM32_PVDConfiguration();
    /* 配置PVD电压为2.9V */
    PWR_PVDLevelConfig(PWR_PVDLevel_2V9);

    while (1)
    {
        printf("\n\r PVD电源电压监测实验，断电PVD中断中会有串口打印。\\r\\n");
        Delay(0xFFFF);
    }
}
```

接下来是使能 PVD 相关的寄存器和相关参数，其中监控电压的阈值就是下面这个函数决定的“`PWR_PVDLevelConfig(PWR_PVDLevel_2V9)`”。

```
void PWR_PVDLevelConfig(uint32_t PWR_PVDLevel)
{
    uint32_t tmpreg = 0;
    /* Check the parameters */
    assert_param(IS_PWR_PVD_LEVEL(PWR_PVDLevel));
    tmpreg = PWR->CR;
    /* Clear PLS[7:5] bits */
    tmpreg &= CR_PLS_Mask;
    /* Set PLS[7:5] bits according to PWR_PVDLevel value */
    tmpreg |= PWR_PVDLevel;
    /* Store the new value */
    PWR->CR = tmpreg;
}
```

你可以根据你的实际需要更改参数来设置不同的监控电压的阈值，可选的参数有：

```
#define PWR_PVDLevel_2V2      ((uint32_t)0x00000000)
#define PWR_PVDLevel_2V3      ((uint32_t)0x00000020)
#define PWR_PVDLevel_2V4      ((uint32_t)0x00000040)
#define PWR_PVDLevel_2V5      ((uint32_t)0x00000060)
#define PWR_PVDLevel_2V6      ((uint32_t)0x00000080)
#define PWR_PVDLevel_2V7      ((uint32_t)0x000000A0)
#define PWR_PVDLevel_2V8      ((uint32_t)0x000000C0)
#define PWR_PVDLevel_2V9      ((uint32_t)0x000000E0)
```

最后是循环打印提示信息等待 PVD 中断，用户可以使用关闭神舟开发板电源的方式来产生 PVD 中断：

```
while (1)
{
    printf("\n\r PVD电源电压监测实验，断电PVD中断中会有串口打印。 \r\n");
    Delay(0x3FFFF);
}
```

MAIN 中的 SZ_STM32_PVDConfiguration()函数配置使能外部中断第 16 线的上升/下降边沿触发：

```
/* Configure EXTI Line16(PVD Output) to generate an
   interrupt on rising and falling edges */
EXTI_ClearITPendingBit(EXTI_Line16);
EXTI_InitStructure.EXTI_Line = EXTI_Line16;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

设置外部中断第 16 线的中断优先级：

```
/* Enable the PVD Interrupt */
NVIC_InitStructure.NVIC IRQChannel = PVD_IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

中断处理函数如下：

```
void PVD_IRQHandler(void)
{
    if (EXTI_GetITStatus(EXTI_Line16) != RESET)
    {
        printf("Bye!");
        EXTI_ClearITPendingBit(EXTI_Line16);
    }
}
```

7.60.5 下载与验证

如果使用JLINK下载固件,请按[3.3如何使用JLINK软件下载固件到神舟III号开发板小节](#)进行操作。

如果使用串口下载固件,请按[3.4如何通过串口下载一个固件到神舟III号开发板小节](#)进行操作。

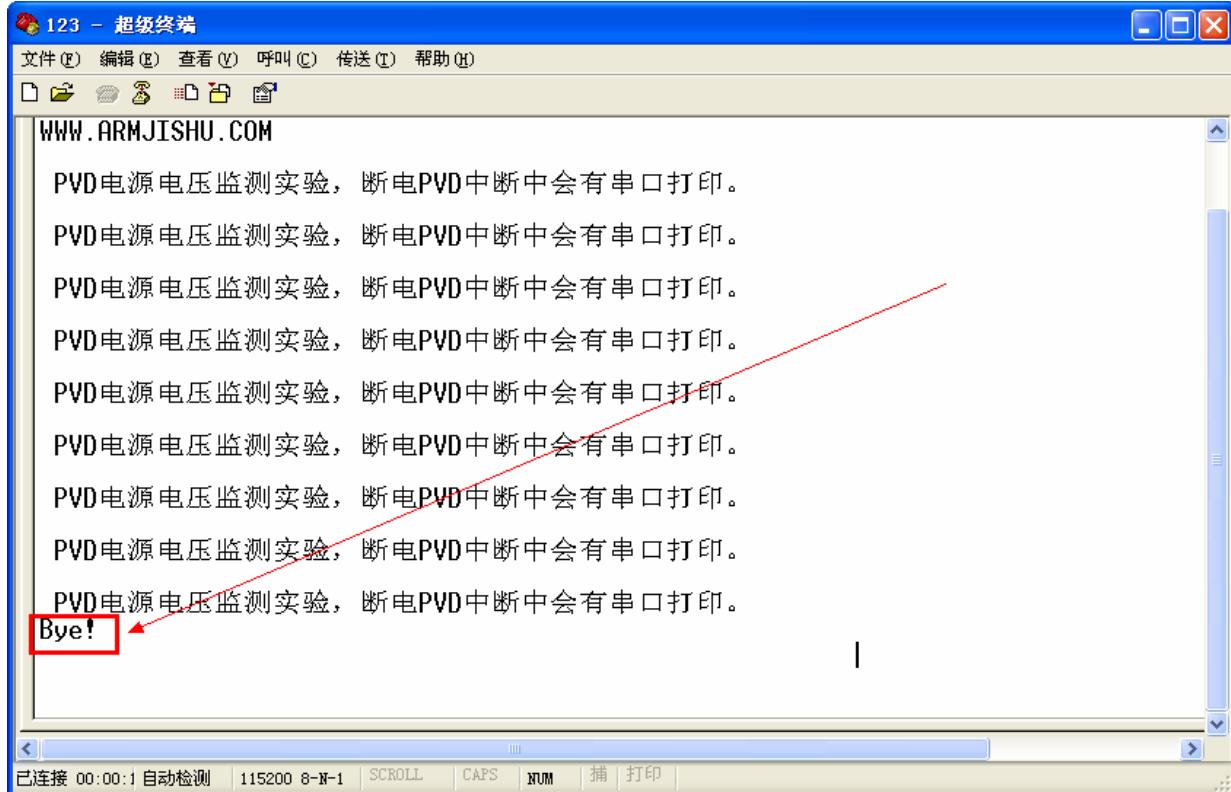
如果在MDK开发环境中,下载编译好的固件或者在线调试,请按[3.8如何在MDK开发环境中使用JLINK在线调试小节](#)进行操作。

7.60.6 实验现象

本实验演示神舟系列 STM32 开发板处理器内部的 PVD 电压监控功能,当检测到电压低于 2.9V (可以灵活设置这个阈值) 时,将产生 PVD 中断,我们在中断中以串口打印“Bye!”来演示“执行紧急处理任务”。

将固件下载在神舟 III 号 STM32 开发板后,用随板配置的串口线连接神舟 III 号串口 1 与电脑的串口,打开超级终端,串口参数配置为 115200 8-N-1。用户可以使用关闭神舟开发板电源的方式来产生 PVD 中断。

连接好串口,按下复位键之后,开发板一直向串口打印数据。断电,串口打印“Bye!”。



7.61 STM32低功耗之--STANDBY待机模式

7.61.1 意义与作用

在系统或电源复位以后，微控制器处于运行状态。当 CPU 不需继续运行时，可以利用多种低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗、最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。

7.61.2 STM32低功耗模式介绍

STM32 有一个低功耗模式，在系统或电源复位以后，微控制器处于运行状态。运行状态下的 HCLK 为 CPU 提供时钟，内核执行程序代码。当 CPU 不需继续运行时，可以利用多个低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗，最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。STM32 的 3 种低功耗模式：STM32 的低功耗模式有 3 种：

- 1)睡眠模式（CM3 内核停止，外设仍然运行）
- 2)停止模式（所有时钟都停止）
- 3)待机模式（1.8V 内核电源关闭）

| 模式 | 进入操作 | 唤醒 | 对1.8V区域时钟的影响 | 对VDD区域时钟的影响 | 电压调节器 |
|--|---------------------------------------|--|---------------------------------------|---------------------------------------|-------|
| 睡眠 (SLEEP-NOW或SLEEP-ON-EXIT) | WFI | 任一中断 | CPU 时钟关，对其他时钟和 ADC 时钟无影响 | 无 | 开 |
| | WFE | 唤醒事件 | | | |
| 停机 | PDDS 和 LPDS 位 +SLEEPDEEP 位 +WFI 或 WFE | 任一外部中断(在外部中断寄存器中设置) | 所有使用 1.8V 的区域的时钟都已关闭，HSI 和 HSE 的振荡器关闭 | 在低功耗模式下可进行开/关设置(依据电源控制寄存器(PWR_CR)的设定) | 关 |
| 待机 | PDDS 位 +SLEEPDEEP 位 +WFI 或 WFE | WKUP 引脚的上升沿、RTC 警告事件、NRST 引脚上的外部复位、IWDG 复位 | | | |

7.61.3 待机模式

待机模式可实现系统的最低功耗。该模式是在 Cortex-M3 深睡眠模式时关闭电压调节器。整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失，只有备份的寄存器和待机电路维持供电。

在这三种低功耗模式中，最低功耗的是待机模式，在此模式下，最低只需要 2uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 20uA 左右。最后就是睡眠模式了。用户可以根据自己的需求来决定使用哪种低功耗模式。

我们仅对 STM32 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32 的最低功耗。该模式是在 CM3 深睡眠模式时关闭电压调节器。整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失。仅备份的寄存器和待机电路维持供电。

待机模式下的输入/输出端口状态在待机模式下，所有的 I/O 引脚处于高阻态，除了以下的引脚：

- 复位引脚(始终有效)

- 当被设置为防侵入或校准输出时的 TAMPER 引脚
- 被使能的唤醒引脚

7.61.4 进入待机模式

关于如何进入待机模式，详见表 12。

可以通过设置独立的控制位，选择以下待机模式的功能：

- 独立看门狗(IWDG)：可通过写入看门狗的键寄存器或硬件选择来启动 IWDG。一旦启动了独立看门狗，除了系统复位，它不能再被停止。
- 实时时钟(RTC)：通过备用区域控制寄存器(RCC_BDCR)的 RTCEN 位来设置。
- 内部 RC 振荡器(LSI RC)：通过控制/状态寄存器(RCC_CSR)的 LSION 位来设置。
- 外部 32.768kHz 振荡器(LSE)：通过备用区域控制寄存器(RCC_BDCR)的 LSEON 位设置。

表12 待机模式

| 待机模式 | 说明 |
|------|--|
| 进入 | 在以下条件下执行WFI(等待中断)或WFE(等待事件)指令： - 设置Cortex™-M3系统控制寄存器中的SLEEPDEEP位 - 设置电源控制寄存器(PWR_CR)中的PDDS位 - 清除电源控制/状态寄存器(PWR_CSR)中的WUF位 |
| 退出 | WKUP引脚的上升沿、RTC闹钟事件的上升沿、NRST引脚上外部复位、IWDG复位。 |
| 唤醒延时 | 复位阶段时电压调节器的启动。 |

WFI = wait for interrupt 等待中断，即下一次中断发生前都在此 hold 住不干活

WFE = wait for event 等待事件，即下一次事件发生前都在此 hold 住不干活

上表还列出了退出待机模式的操作，从上表可知，我们有 4 种方式可以退出待机模式，即当一个外部复位(NRST 引脚)、IWDG 复位、WKUP 引脚上的上升沿或 RTC 闹钟事件发生时，微控制器从待机模式退出。从待机唤醒后，除了电源控制/状态寄存器(PWR_CSR)，所有寄存器被复位。从待机模式唤醒后的代码执行等同于复位后的执行(采样启动模式引脚，读取复位向量等)。电源控制/状态寄存器(PWR_CSR)将会指示内核由待机状态退出。

说明：进入待机模式，JTAG 是无法仿真和下载程序的，如果你写这个程序的时候不小心使系统一直进待机模式，也不要怕。可以通过切换 BOOT 引脚，使用串口 ISP 下载程序，清除 FLASH 就可以啦。

7.61.5 待机模式寄存器介绍

上面我们已经说明了进入待机模式的通用步骤，其中涉及到 2 个寄存器，电源控制寄存器(PWR_CR) 和电源控制/状态寄存器(PWR_CSR)。下面我们介绍一下这两个寄存器：

电源控制寄存器 (PWR_CR)，该寄存器的各位描述如下

4.4 电源控制寄存器

可以用半字(16位)或字(32位)的方式操作这些外设寄存器。

4.4.1 电源控制寄存器(PWR_CR)

地址偏移: 0x00

复位值: 0x0000 0000 (从待机模式唤醒时清除)



| | | | | | | | | | |
|-----------|--|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 位 31:9 | 保留。始终读为0。 | | | | | | | | |
| 位 8 | DBP: 取消后备区域的写保护 在复位后, RTC和后备寄存器处于被保护状态以防意外写入。设置这位允许写入这些寄存器。 0: 禁止写入RTC和后备寄存器 1: 允许写入RTC和后备寄存器 注: 如果RTC的时钟是HSE/128, 该位必须保持为'1'。 | | | | | | | | |
| 位 7:5 | PLS[2:0]: PVD电平选择 这些位用于选择电源电压监测器的电压阀值 <table style="margin-left: 20px;"> <tr><td>000: 2.2V</td><td>100: 2.6V</td></tr> <tr><td>001: 2.3V</td><td>101: 2.7V</td></tr> <tr><td>010: 2.4V</td><td>110: 2.8V</td></tr> <tr><td>011: 2.5V</td><td>111: 2.9V</td></tr> </table> 注: 详细说明参见数据手册中的电气特性部分。 | 000: 2.2V | 100: 2.6V | 001: 2.3V | 101: 2.7V | 010: 2.4V | 110: 2.8V | 011: 2.5V | 111: 2.9V |
| 000: 2.2V | 100: 2.6V | | | | | | | | |
| 001: 2.3V | 101: 2.7V | | | | | | | | |
| 010: 2.4V | 110: 2.8V | | | | | | | | |
| 011: 2.5V | 111: 2.9V | | | | | | | | |
| 位 4 | PVDE: 电源电压监测器(PVD)使能 0: 禁止PVD 1: 开启PVD | | | | | | | | |
| 位 3 | CSBF: 清除待机位 始终读出为0 0: 无功效 1: 清除SBF待机位(写) | | | | | | | | |
| 位 2 | CWUF: 清除唤醒位 始终读出为0 0: 无功效 1: 2个系统时钟周期后清除WUF唤醒位(写) | | | | | | | | |
| 位 1 | PDDS: 掉电深睡眠 与LPDS位协同操作 0: 当CPU进入深睡眠时进入停机模式, 调压器的状态由LPDS位控制。 1: CPU进入深睡眠时进入待机模式。 | | | | | | | | |
| 位 0 | LPDS: 深睡眠下的低功耗 PDDS=0时, 与PDDS位协同操作 0: 在停机模式下电压调压器开启 1: 在停机模式下电压调压器处于低功耗模式 | | | | | | | | |

这里我们通过设置 PWR_CR 的 PDDS 位, 使 CPU 进入深度睡眠时进入待机模式, 同时我们通
嵌入式专业技术论坛 (www.armjishu.com) 出品
页

过 CWUF 位，清除之前的唤醒位。

电源控制/状态寄存器（PWR_CSR）的各位描述如下

4.4.2 电源控制/状态寄存器(PWR_CSR)

地址偏移: 0x04

复位值: 0x0000 0000 (从待机模式唤醒时不被清除)

与标准的APB读相比，读此寄存器需要额外的APB周期



| | |
|-------|---|
| 位31:9 | 保留。始终读为0。 |
| 位 8 | EWUP: 使能WKUP引脚 0: WKUP引脚为通用I/O。WKUP引脚上的事件不能将CPU从待机模式唤醒 1: WKUP引脚用于将CPU从待机模式唤醒，WKUP引脚被强置为输入下拉的配置(WKUP引脚上的上升沿将系统从待机模式唤醒) 注：在系统复位时清除这一位。 |
| 位 7:3 | 保留。始终读为0。 |
| 位 2 | PVDO: PVD输出 当PVD被PVDE位使能后该位才有效 0: V_{DD}/V_{DDA} 高于由PLS[2:0]选定的PVD阀值 1: V_{DD}/V_{DDA} 低于由PLS[2:0]选定的PVD阀值 注：在待机模式下PVD被停止。因此，待机模式后或复位后，直到设置PVDE位之前，该位为0。 |
| 位 1 | SBF: 待机标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CSBF位清除。 0: 系统不在待机模式 1: 系统进入待机模式 |
| 位 0 | WUF: 唤醒标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CWUF位清除。 0: 没有发生唤醒事件 1: 在WKUP引脚上发生唤醒事件或出现RTC闹钟事件。 注：当WKUP引脚已经是高电平时，在(通过设置EWUP位)使能WKUP引脚时，会检测到一个额外的事件。 |

这里，我们通过设置 PWR_CSR 的 EWUP 位，来使能 WKUP 引脚用于待机模式唤醒。WUF 位也可以用来检查是否发生了唤醒事件。

7.61.6 待机唤醒步骤

1) 使能电源时钟。

因为要配置电源控制寄存器，所以必须先使能电源时钟。

2) 设置 WK_UP 引脚作为唤醒源。

因为要配置电源控制寄存器，所以必须先使能电源时钟。然后再设置 PWR_CSR 的 EWUP 位，使能 WK_UP 用于将 CPU 从待机模式唤醒。

3) 设置 PDDS 位，执行 WFI 指令，进入待机模式。

接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。

4) 最后编写 WK_UP 中断函数。

因为我们通过 WK_UP 中断来唤醒 CPU，所以我们有必要设置一下该中断函数，同时我们也通过该函数里面进入待机模式。

通过以上几个步骤的设置，我们就可以使用 STM32 的待机模式了，并且可以通过 WK_UP 来唤醒 CPU。

7.61.7 硬件设计

本实验中，我们使用了 LED 指示灯与两个按键完成本次的实验，配置 USER2 按键作为进入待机模式按键，WAKEUP 按键作为唤醒按键即可，程序实现的功能是：按下 USER2 进入待机模式，LED 都灭了。再按下 WAKEUP 唤醒进入工作模式，LED 灯全部闪烁。使用的按键、指示灯请参考前面的相关章节。

7.61.8 软件设计

我们先从 MAIN 函数看起，如下所示，第一部分是初始化按键、LED 指示灯等等，这部分知识在前面章节已经讲解过，请回顾相关章节的知识

```
int main(void)
{
    LED_config();
    Led_Turn_on_all();                                /*上电初始化时，打开所有灯*/
    Delay_ARMJISHU(2000000);
    Led_Turn_off_all();                               /*关闭LED灯*/
    Delay_ARMJISHU(2000000);

    SZ_STM32_KEYInit();
}
```

接下来是对 WakeUp 管脚唤醒处理器函数，系统定时器与 RTC 时钟的初始化，相关函数在后面讲解

```
/* 使能WakeUp管脚唤醒处理器 Enable WKUP pin */
PWR_WakeUpPinCmd(ENABLE);

/* RTC待机模式初始化 */
RTC_STANDBY_Configuration();

InterruptConfig();                                    /*设置中断向量起始地址*/
SysTick_Configuration();                           /*SysTick配置*/

```

循环工作等待按键中断进入待机模式或者重新运行程序。中间并对 LED 进行一个延时闪烁的操作。

```
/* Main loop */
while (1)
{
    Led_Turn_on_all();                                /*上电初始化时，打开所有灯*/
    Delay_ARMJISHU(2000000);
    Led_Turn_off_all();                               /*关闭LED灯*/
    Delay_ARMJISHU(2000000);
}
```

深入分析程序代码

RTC 待机模式初始化函数，该函数是对一些时钟使能与寄存器配置的函数，通过函数“PWR_GetFlagStatus”的返回值判断是否是等于“RESET”。无论返回值是否等于“RESET”，都各种进行了一个对 RTC 配置操作。

```
/* RTC待机模式初始化 */
RTC_STANDBY_Configuration();

void RTC_STANDBY_Configuration(void)
{
    /* Enable PWR and BKP clocks */
    /* PWR时钟（电源控制）与BKP时钟（RTC后备寄存器）使能 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP,
                           ENABLE);

    /* Allow access to BKP Domain */
    /*使能RTC和后备寄存器访问 */
    PWR_BackupAccessCmd(ENABLE);

    /* Enable WKUP pin */
    PWR_WakeUpPinCmd(ENABLE);

    /* Check if the StandBy flag is set */
    if(PWR_GetFlagStatus(PWR_FLAG_SB) != RESET)
    {/* System resumed from STANDBY mode */

        /* Clear StandBy flag */
        PWR_ClearFlag(PWR_FLAG_SB);

        /* Wait for RTC APB registers synchronisation */
        RTC_WaitForSynchro();
    }
}
```

```
}

else
{
    /* Reset Backup Domain */
    /* 将外设BKP的全部寄存器重设为缺省值 */
    //BKP_DeInit();

    /* Enable LSE */
    /* 使能LSE（外部32.768KHz低速晶振）*/
    RCC_LSEConfig(RCC_LSE_ON);

    /* Wait till LSE is ready */
    /* 等待外部晶振震荡稳定输出 */
    while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
    {
    }

    /* Select LSE as RTC Clock Source */
    /* 使用外部32.768KHz晶振作为RTC时钟 */
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

    /* Enable RTC Clock */
    /* 使能 RTC 的时钟供给 */
    RCC_RTCCLKCmd(ENABLE);
```

当函数“PWR_GetFlagStatus”的返回值不等于“RESET”的话，进行一个PWR_ClearFlag(PWR_FLAG_SB)的操作，这个函数的目的是使电源控制寄存器(PWR_CR)的第四位置1，清除待机位，从而退出待机模式。而PWR_FLAG是等于0x00000002的；

```
void PWR_ClearFlag(uint32_t PWR_FLAG)
{
    /* Check the parameters */
    assert_param(IS_PWR_CLEAR_FLAG(PWR_FLAG));

    PWR->CR |= PWR_FLAG << 2;
}
```

| | |
|-----|--|
| 位 3 | CSBF: 清除待机位 始终读出为0 0: 无功效 1: 清除SBF待机位(写) |
|-----|--|

当函数“PWR_GetFlagStatus”的返回值等于“RESET”的话则是对我们的RTC进行配置操作了，从运行模式复位。

根据实验现象得到，我们按下 USER2 按键的时候，STM32 进入待机模式，那么我们看下它是如何实现的，首先我们看下对 KEY 进行初始化的函数

```
SZ_STM32_KEYInit();
```

```
void SZ_STM32_KEYInit( )
{
    GPIO_InitTypeDef GPIO_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable the BUTTON Clock */
    /* 使能KEY按键对应GPIO的Clock时钟 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO,
                           ENABLE);

    /* Configure Button pin as input floating */
    /* 初始化KEY按键的GPIO管脚，配置为带上拉的输入 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    /* Connect Button EXTI Line to Button GPIO Pin */
    /* 将KEY按键对应的管脚连接到内部中断线 */
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource3);

    /* Configure Button EXTI line */
    /* 将KEY按键配置为中断模式，下降沿触发中断 */
    EXTI_InitStructure EXTI_Line = EXTI_Line3;
    EXTI_InitStructure EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure EXTI_Trigger = EXTI_Trigger_Falling;
    EXTI_InitStructure EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    /* Enable and set Button EXTI Interrupt to the lowest priority */
    /* 将KEY按键的中断优先级配置为最低 */
    NVIC_InitStructure.NVIC_IRQChannel = EXTI3_IRQn;
}
```

根据原理图可得到我们的按键 USER2 为上拉输入的一个按键，我们对这个按键进行了一个中断的配置，按键按下的时候产生中断信号，而由于它是上拉输入的，按键按下的时候，由高电平变为低电平，所以我们要把它配置为下降沿触发中断。

KEY1 的中断函数为：EXTI3_IRQHandler，该函数中，判断“EXTI_GetITStatus ()；”函数返回值是否是等于“RESET”，如果不等于的话，执行该函数“EXTI3_IRQHandler ()”否则退出。

```
void EXTI3_IRQHandler(void)
{
    if (EXTI_GetITStatus(EXTI_Line3) != RESET)
    {
        /* Clear the EXTI Line 4 */
        EXTI_ClearITPendingBit(EXTI_Line3);

        /* Request to enter STANDBY mode (Wake Up flag is cleared in PWR_EnterSTANDBYMode())
    }
}
```

通过判断按键产生的中断信号，使得 EXTI_Line3 中断线产生触发请求，进入中断服务函数后，执行“PWR_EnterSTANDBYMode ()”函数。

9.3.6 挂起寄存器(EXTI_PR)

偏移地址: 0x14

复位值: 0xFFFF XXXX

| | | | | | | | | | | | | | | | |
|--------|--|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | PR19 | PR18 |
| | | | | | | | | | | | | | | PR17 | PR16 |
| | | | | | | | | | | | | | | rc | wl |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PR15 | PR14 | PR13 | PR12 | PR11 | PR10 | PR9 | PR8 | PR7 | PR6 | PR5 | PR4 | PR3 | PR2 | PR1 | PR0 |
| rc | wl | rc | wl | rc | wl | rc | wl | rc | wl | rc | wl | rc | wl | rc | wl |
| 位31:19 | 保留, 必须始终保持为复位状态(0)。 | | | | | | | | | | | | | | |
| 位18:0 | PRx: 挂起位 (Pending bit) 0: 没有发生触发请求 1: 发生了选择的触发请求 当在外部中断线上发生了选择的边沿事件, 该位被置'1'。在该位中写入'1'可以清除它, 也可以通过改变边沿检测的极性清除。 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | | |

“PWR_EnterSTANDBYMode ()” 函数的目的是配置我们的 STM32 进入我们的待机模式

```
/*
 * @brief Enters STANDBY mode.
 * @param None
 * @retval None
 */
void PWR_EnterSTANDBYMode (void)
{
    /* Clear Wake-up flag */
    PWR->CR |= PWR_CR_CWUF;
    /* Select STANDBY mode */
    PWR->CR |= PWR_CR_PDDS;
    /* Set SLEEPDEEP bit of Cortex System Control Register */
    SCB->SCR |= SCB_SCR_SLEEPDEEP;
    /* This option is used to ensure that store operations are completed */
    #if defined ( __CC_ARM )
        __force_stores();
    #endif
    /* Request Wait For Interrupt */
    __WFI();
}
```

先是通过对我们的 STM32 进行一个清除唤醒位的操作

```
#define PWR_CR_CWUF ((uint16_t) 0x0004)
```

| | |
|-----|---|
| 位 2 | CWUF: 清除唤醒位 始终读出为0 0: 无功效 1: 2个系统时钟周期后清除WUF唤醒位(写) |
|-----|---|

然后是进入待机模式

```
#define PWR_CR_PDDS ((uint16_t) 0x0002)
```

| | |
|-----|---|
| 位 1 | PDDS: 掉电深睡眠 与LPDS位协同操作 0: 当CPU进入深睡眠时进入停机模式, 调压器的状态由LPDS位控制。 1: CPU进入深睡眠时进入待机模式。 |
|-----|---|

这样, 我们的 STM32 即可进入了我们的待机模式

那么 STM32 是如何退出待机模式的呢? 我们通过前面的描述中可知, 退出待机模式的方法有四种, 分别是:

退出

WKUP引脚的上升沿、RTC闹钟事件的上升沿、NRST引脚上外部复位、IWDG复位。

我们这里用到的是 WKUP 引脚控制, 通过这个引脚的地址, 使得按键按下去的时候, 产生一个上升沿唤醒我们的 STM32, 在这里我们使用的是 PWR_WakeUpPinCmd(FunctionalState NewState); 函数实现。

```
/* 也许WakeUp管脚(KEY4)唤醒处理器 Enable WKUP pin */
PWR_WakeUpPinCmd(ENABLE);
```

到这里, 我们已经能实现通过按键使得 STM32 进入待机模式与退出待机模式的实验。

7.61.9 下载与验证

如果在MDK开发环境中, 下载编译好的固件或者在线调试, 请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.61.10 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后，按下复位键。神舟 III 号开发板上的 LED 灯不断的闪烁。按下按键 USER2，进入待机模式，LED 灯停止闪烁。按下按键 WAKEUP，唤醒，LED 灯恢复闪烁状态。

7.62 STM32低功耗之--STOP停止模式实验

7.62.1 意义与作用

在系统或电源复位以后，微控制器处于运行状态。当 CPU 不需继续运行时，可以利用多种低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗、最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。

本实现通过神舟V号STM32开发板学习STM32的低功耗模式中的STOP停止模式。

7.62.2 实验原理

停止模式是在Cortex™-M3的深睡眠模式基础上结合了外设的时钟控制机制，在停止模式下电压调节器可运行在正常或低功耗模式。此时在1.8V供电区域的所有时钟都被停止，PLL、HSI和HSE RC振荡器的功能被禁止，SRAM和寄存器内容被保留下来。在停止模式下，所有的I/O引脚都保持它们在运行模式时的状态。

进入停止模式：

关于如何进入停止模式，详见表11。在停止模式下，通过设置电源控制寄存器(PWR_CR)的LPDS位使内部调节器进入低功耗模式，能够降低更多的功耗。如果正在进行闪存编程，直到对内存访问完成，系统才进入停止模式。如果正在进行对APB的访问，直到对APB访问完成，系统才进入停止模式。

可以通过对独立的控制位进行编程，可选择以下功能：

- 独立看门狗(IWDG)：可通过写入看门狗的键寄存器或硬件选择来启动IWDG。一旦启动了独立看门狗，除了系统复位，它不能再被停止。
- 实时时钟(RTC)：通过备份域控制寄存器 (RCC_BDCR)的RTCEN位来设置。
- 内部RC振荡器(LSI RC)：通过控制/状态寄存器 (RCC_CSR)的LSION位来设置。
- 外部32.768kHz振荡器(LSE)：通过备份域控制寄存器((RCC_BDCR)的LSEON位设置。

在停止模式下，如果在进入该模式前ADC和DAC没有被关闭，那么这些外设仍然消耗电流。通过设置寄存器ADC_CR2的ADON位和寄存器DAC_CR的ENx位为0可关闭这2个外设。

退出停止模式

关于如何退出停止模式，详见下表。当一个中断或唤醒事件导致退出停止模式时，HSI RC振荡器被选为系统时钟。当电压调节器处于低功耗模式下，当系统从停止模式退出时，将会有一段额外的启动延时。如果在停止模式期间保持内部调节器开启，则退出启动时间会缩短，但相应的功耗会增加。

下面的表格说明如何进入和退出停止模式：

表11 停止模式

| 停止模式 | 说明 |
|------|--|
| 进入 | <p>在以下条件下执行WFI(等待中断)或WFE(等待事件)指令：</p> <ul style="list-style-type: none"> - 设置Cortex-M3系统控制寄存器中的SLEEPDEEP位 - 清除电源控制寄存器(PWR_CR)中的PDDS位 - 通过设置PWR_CR中LPDS位选择电压调节器的模式 <p>注：为了进入停止模式，所有的外部中断的请求位(挂起寄存器(EXTI_PR))和RTC的闹钟标志都必须被清除，否则停止模式的进入流程将会被跳过，程序继续运行。</p> |
| 退出 | <p>如果执行WFI进入停止模式：</p> <p>设置任一外部中断线为中断模式(在NVIC中必须使能相应的外部中断向量)。参见中断向量表(表54)。</p> <p>如果执行WFE进入停止模式：</p> <p>设置任一外部中断线为事件模式。参见唤醒事件管理(第9.2.3节)。</p> |
| 唤醒延时 | HSI RC唤醒时间 + 电压调节器从低功耗唤醒的时间。 |

7.62.3 硬件设计

STOP 停止模式是 STM32 处理器的一种低功耗模式，这部分不需要特殊硬件电路，只要最小系统运行正常即可，本实验使用的按键、指示灯和串口电路请参考前面的相关章节。

7.62.4 软件设计

我们先从 MAIN 函数看起，如下所示，第一部分是初始化按键、LED 指示灯等等，这部分知识在前面章节已经讲解过，请回顾相关章节的知识。

```
int main(void)
{
    LED_config();
    Led_Turn_on_all();
    Delay_ARMJISHU(2000000);
    Led_Turn_off_all();
    Delay_ARMJISHU(2000000);

    SZ_STM32_KEYInit();
```

接下来是 RTC 中断、系统定时器初始化，相关函数在后面讲解

```
/* RTC闹钟初始化 */
RTCAlarm_Configuration();

/* 配置RTC闹钟中断向量参数 */
RTCAlarm_NVIC_Configuration();

InterruptConfig();                                /*设置中断向量起始地址*/
SysTick_Config();                                /*SysTick配置*/
```

最后是循环进入 STOP 停止模式（PWR_EnterSTOPMode）等待按键中断或中 RTC 闹钟中断来唤醒处理器回到正常的工作模式：

```
while (1)
{
    /* RTC中断处理函数和按键中断处理函数位于"stm32f10x_it.c"文件的 */
    /* Wait till RTC Second event occurs */
    RTC_ClearFlag(RTC_FLAG_SEC);
    while (RTC_GetFlagStatus(RTC_FLAG_SEC) == RESET);

    /* Alarm in 5 second */
    RTC_SetAlarm(RTC_GetCounter() + 5);
    /* Wait until last write operation on RTC registers has finished */
    RTC_WaitForLastTask();

    /* Request to enter STOP mode with regulator in low power mode*/
    PWR_EnterSTOPMode(PWR_Regulator_LowPower, PWR_STOPEntry_WFI);

    /* Configures system clock after wake-up from STOP: enable HSE, PLL
     * PLL as system clock source (HSE and PLL are disabled in STOP mode) */
    SYSCCLKConfig_STOP();
}
```

深入分析实验代码

MAIN 函数中的 EXTI_Configuration 函数是配置使能外部中断第 17 线的上升边沿触发，这样才能产生 RTC 闹钟报警中断：

```
/* RTC闹钟初始化 */
RTCAlarm_Configuration();
```

```
/*
 * @函数名 RTCAlarm_Configuration
 * @功能 RTC闹钟初始化
 * @参数 无
 * @返回值 无
 */
void RTCAlarm_Configuration(void)
{
    /* Enable PWR and BKP clocks */
    /* PWR时钟（电源控制）与BKP时钟（RTC后备寄存器）使能 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

    /* Allow access to BKP Domain */
    /* 使能RTC和后备寄存器访问 */
    PWR_BackupAccessCmd(ENABLE);

    /* Reset Backup Domain */
    /* 将外设BKP的全部寄存器重设为缺省值 */
    //BKP_DeInit();

    /* Enable LSE */
    /* 使能LSE（外部32.768KHz低速晶振） */
    RCC_LSEConfig(RCC_LSE_ON);

    /* Wait till LSE is ready */
    /* 等待外部晶振震荡稳定输出 */
    while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
        {}

    /* Select LSE as RTC Clock Source */
    /* 使用外部32.768KHz晶振作为RTC时钟 */
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

    /* Enable RTC Clock */
    /* 使能 RTC 的时钟供给 */
    RCC_RTCCLKCmd(ENABLE);

    /* Wait for RTC registers synchronization */
    /* 等待RTC寄存器同步 */
    RTC_WaitForSynchro();

    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();

    /* Enable the RTC Alarm interrupt */
    /* 使能RTC的闹钟中断 */
    RTC_ITConfig(RTC_IT_ALR, ENABLE);

    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();

    /* Set RTC prescaler: set RTC period to 1sec */
    /* 32.768KHz晶振预分频值是32767,如果对精度要求很高可以修改此分频值来校准晶振 */
    RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */

    /* Wait until last write operation on RTC registers has finished */
    /* 等待上一次对RTC寄存器的写操作完成 */
    RTC_WaitForLastTask();
}
```

配置嵌套中断向量表，外部中断第 17 线中断函数，使能并设置外部中断第 17 线的 RTC 闹钟报警中断优先级：

```
/*
 * @函数名 RTCAlarm_NVIC_Configuration
 * @功能 配置RTC闹钟中断向量参数函数
 * @参数 无
 * @返回值 无
*/
void RTCAlarm_NVIC_Configuration(void)
{
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Configure EXTI Line17 (RTC Alarm) to generate
       an interrupt on rising edge */
    EXTI_ClearITPendingBit(EXTI_Line17);
    EXTI_InitStructure.EXTI_Line = EXTI_Line17;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    /* 2 bits for Preemption Priority and 2 bits for Sub Priority */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

    /* Enable the RTC Interrupt */
    NVIC_InitStructure.NVIC IRQChannel = RTCAlarm_IRQn;
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

MAIN 函数中的 SysTick_Configuration 函数是系统滴答中断及其中断优先级：这样在运行模式时使 4 个 LED 闪烁。系统滴答中断时使 4 个 LED 闪烁

```
void SysTick_Handler(void)
{
    LED_Spark();
}
```

```
void LED_Spark(void)
{
    static __IO uint32_t TimingDelayLocal = 0;

    if (TimingDelayLocal != 0x00)
    {
        if(TimingDelayLocal < 50) // 后50次熄灭
        {
            Led_Turn_off_all();
        }
        else // 前50次点亮LED指示
        {
            Led_Turn_on_all();
        }
        TimingDelayLocal--; // 该函数每一次被调用
    }
    else
    {
        TimingDelayLocal = 100; //赋值
    }
}
```

根据 USER2 初始化后的按键中断函数，我们看下对 KEY 进行初始化的函数

```
SZ_STM32_KEYInit();
```

```
void SZ_STM32_KEYInit()
{
    GPIO_InitTypeDef GPIO_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable the BUTTON Clock */
    /* 使能KEY按键对应GPIO的Clock时钟 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO, ENABLE);

    /* Configure Button pin as input floating */
    /* 初始化KEY按键的GPIO管脚，配置为带上拉的输入 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    /* Connect Button EXTI Line to Button GPIO Pin */
    /* 将KEY按键对应的管脚连接到内部中断线 */
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource3);
```

```
/* Configure Button EXTI line */
/* 将KEY按键配置为中断模式，下降沿触发中断 */
EXTI_InitStructure.EXTI_Line = EXTI_Line3;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);

/* Enable and set Button EXTI Interrupt to the lowest priority */
/* 将KEY按键的中断优先级配置为最低 */
NVIC_InitStructure.NVIC IRQChannel = EXTI3_IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x0F;
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x0F;
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
}
```

根据原理图可得到我们的按键 USER2 为上拉输入的一个按键，我们对这个按键进行了一个中断的配置，按键按下的时候产生中断信号，而由于它是上拉输入的，按键按下的时候，由高电平变为低电平，所以我们要把它配置为下降沿触发中断。

按键 USER2 的中断线为：EXTI3_IRQn，处理函数 EXTI3_IRQHandler（）；中，判断“EXTI_GetITStatus（）”函数返回值是否是等于“RESET”，如果不等于的话，执行该函数“EXTI3_IRQHandler（）”否则退出。

```
void EXTI3_IRQHandler(void)
{
    if (EXTI_GetITStatus(EXTI_Line3) != RESET)
    {
        EXTI_ClearITPendingBit(EXTI_Line3);
    }
}
```

进入中断函数后，调用函数 EXTI_GetITStatus(EXTI_Line3)判断它的返回值是否等于 RESET，不等于则执行“EXTI_ClearITPendingBit（）”函数。

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line)
{
    /* Check the parameters */
    assert_param(IS_EXTI_LINE(EXTI_Line));

    EXTI->PR = EXTI_Line;
}
```

9.3.6 挂起寄存器(EXTI_PR)

偏移地址: 0x14

复位值: 0xXXXX XXXX

根据寄存器 EXTI_PR 寄存器的定义，为该中断信号产生触发请求

外部中断第 17 线产生中断后进入该函数，当函数“RTC_GetITStatus”的返回值不等于“RESET”的话，执行该中断函数，**RTC 闹钟中断唤醒处理器**。

```
void RTCAlarm_IRQHandler(void)
{
    if(RTC_GetITStatus(RTC_IT_ALR) != RESET)
    {
        /* Clear EXTI line17 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line17);

        /* Check if the Wake-Up flag is set */
        if(PWR_GetFlagStatus(PWR_FLAG_WU) != RESET)
        {
            /* Clear Wake Up flag */
            PWR_ClearFlag(PWR_FLAG_WU);
        }

        /* Wait until last write operation on RTC reg */
        RTC_WaitForLastTask();
        /* Clear RTC Alarm interrupt pending bit */
        RTC_ClearITPendingBit(RTC_IT_ALR);
        /* Wait until last write operation on RTC reg */
        RTC_WaitForLastTask();
    }
}
```

并且进行一个 EXTI_ClearITPendingBit (EXTI_Line17);的操作产生中断

```
#define EXTI_Line17 ((uint32_t)0x20000)
```

9.3.5 软件中断事件寄存器(EXTI_SWIER)

偏移地址: 0x10

复位值: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | | |
|---|-------------|-------------|-------------|-------------|-------------|------------|------------|------------|------------|------------|------------|------------|------------|-------------|-------------|-------------|-------------|
| 保留 | | | | | | | | | | | | | | SWIER 19 | SWIER 18 | SWIER 17 | SWIER 16 |
| | | | | | | | | | | | | | | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| SWIER 15 | SWIER 14 | SWIER 13 | SWIER 12 | SWIER 11 | SWIER 10 | SWIER 9 | SWIER 8 | SWIER 7 | SWIER 6 | SWIER 5 | SWIER 4 | SWIER 3 | SWIER 2 | SWIER 1 | SWIER 0 | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | |
| 位31:19 保留, 必须始终保持为复位状态(0)。 | | | | | | | | | | | | | | | | | |
| 位18:0 SWIERx: 线x上的软件中断 (Software interrupt on line x) 当该位为'0'时, 写'1'将设置EXTI_PR中相应的挂起位。如果在EXTI_IMR和EXTI_EMR中允许产生该中断, 则此时将产生一个中断。 注: 通过清除EXTI_PR的对应位(写入'1'), 可以清除该位为'0'。 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | | | | | |

当函数“PWR_GetFlagStatus”的返回值不等于“RESET”的话, 执行“PWR_ClearFlag ()”; 这个函数的目的是使电源控制寄存器(PWR_CR)的第 3 位置 1, 清除 WUF 唤醒位, 而 PWR_FLAG_WU= PWR_FLAG 是等于 0x00000001 的;

```
#define PWR_FLAG_WU ((uint32_t)0x00000001)
```

```
void PWR_ClearFlag(uint32_t PWR_FLAG)
{
    /* Check the parameters */
    assert_param(IS_PWR_CLEAR_FLAG(PWR_FLAG));

    PWR->CR |= PWR_FLAG << 2;
}
```

| | |
|-----|---|
| 位 2 | CWUF: 清除唤醒位 始终读出为0 0: 无功效 1: 2个系统时钟周期后清除WUF唤醒位(写) |
|-----|---|

7.62.5 下载与验证

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.62.6 实验现象

本实验使 STM32 进入停止模式，并利用 KEY2 按键产生的外部中断线或者 RTC 闹钟报警来唤醒 STM32。通过指示灯来指示处理器的当前模式。

将固件下载到程序后，按下复位按键。4 个 LED 闪烁不停的闪烁，进入停止模式后，LED 灯停止闪烁。如果不按下按键，一段时间后 RTC 闹钟报警来唤醒 STM32，板载的 4 个 LED 灯恢复闪烁状态，如此循环。当 STM32 进入停止模式后，可以通过按下按键 USER2 来唤醒 STM32，使得 4 个 LED 灯恢复闪烁状态。

7.63 DMA 传输实验

本章我们将向大家介绍神舟III号STM32的DMA。在本章中，我们将利用STM32的DMA来实现串口2数据传送，并在TFTLCD彩屏上显示当前的传送进度。本章分为如下几个部分：

1 STM32 DMA简介

2 实验原理

3 硬件设计

4 软件设计

5 下载验证

7.63.1 STM32 DMA简介

DMA，全称为：Direct Memory Access，即直接存储器访问。

根据 ST 公司提供的相关信息，DMA 是 STM32 中一个独立与 Cortex-M3 内核的模块，有点类似与 ADC、PWM、TIMER 等模块；主要功能是通信“桥梁”的作用，可以将所有外设映射的寄存器“连接”起来，这样就可以高速访问各寄存器，其传输不受 CPU 的支配，传输还是双向的。

DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器来实行和完成。典型的例子就是移动一个外部内存的区块到芯片内部更快的内存区。像是这样的操作并没有让处理器工作拖延，反而可以被重新排程去处理其他的工作。DMA 传输对于高效能嵌入式系统算法和网络是很重要的。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，能使 CPU 的效率大为提高。

STM32最多有2个DMA控制器（DMA2仅存在大容量产品中），DMA1有7个通道。DMA2有5个通道。每个通道专门用来管理来自于一个或多个外设对存储器访问的请求。还有一个仲裁起来协调各个DMA请求的优先权。

STM32的DMA有以下一些特性：

- 每个通道都直接连接专用的硬件DMA请求，每个通道都同样支持软件触发。这些功能通过软件来配置。
- 在七个请求间的优先权可以通过软件编程设置(共有四级：很高、高、中等和低)，假如在

- 相等优先权时由硬件决定(请求0优先于请求1，依此类推)。
- 独立的源和目标数据区的传输宽度(字节、半字、全字)，模拟打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。
- 支持循环的缓冲器管理
- 每个通道都有3个事件标志(DMA 半传输，DMA传输完成和DMA传输出错)，这3个事件标志逻辑或成为一个单独的中断请求。
- 存储器和存储器间的传输
- 外设和存储器，存储器和外设的传输
- 闪存、SRAM、外设的SRAM、APB1 APB2和AHB外设均可作为访问的源和目标。
- 可编程的数据传输数目：最大为65536

神舟 III 号的主芯片 STM32F103ZET6 有两个 DMA 控制器，DMA1 和 DMA2，本章，我们仅针对 DMA1 进行介绍。

7.63.2 实验原理

本实验的原理：串口 1 以 DMA 方式发送数据。我们先定义 5200 个字节的数组 SendBuff[]，即开辟了一个空间，空间用于存放要发送的数据。将这个空间和串口用 DMA 方式绑定。DMA 传输是将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器来实行和完成。如本实验中，我们先将要发送的数据送入开辟的 SendBuff[]地址空间，初始化从这个地址空间向串口 1 发送数据这个动作。初始化完成后，传输动作本身是由 DMA 控制器来实行和完成，不用 CPU 直接控制传输。我们只需要开启 DMA 传输即可。

接下来我们就介绍本实验的基本实现步骤：

- 1) 对串口进行初始化。STM32_Shenzhou_COMInit()，我们这里用到的是串口1，我们将波特率配置成2400。
 - 2) 调用函数MYDMA_Config() 配置DMA。MYDMA_Config() 函数初始化 串口1以DMA方式发送数据这个动作动作。具体配置的内容有：使能DMA传输、传入串口1使用的通道4等，详细的配置我们在代码部分进行分析。
 - 3) 将待发送的数据传入我们开辟的空间SendBuff[]，即定义的数组。
 - 4) 调用USART_DMACmd() 函数，使能串口1的DMA发送。
 - 5) 经过上面的配置，我们调用MYDMA_Enable()函数，开始一次DMA传输。
 - 6) 开始一次传输后，我们通过函数DMA_ClearFlag(DMA1_FLAG_TC4)，清除通道7传输完成标志。为下一次DMA传输做准备。
- 到此我们实验的基本流程就完成了。

下面我们来看一下 DMA 的 6 个寄存器：

寄存器 1：DMA 中断状态寄存器（DMA_ISR）

如果开启了 DMA_ISR 中这些中断，在达到条件后就会跳到中断服务函数里面去，即使没开启，我们也可以通过查询这些位来获得当前 DMA 传输的状态。这里我们常用的是 TCIFx，即通道 DMA 传输完成与否的标志。注意此寄存器为只读寄存器，所以在这些位被置位之后，只能通过其他的操作来清除

寄存器 2：DMA 中断标志清除寄存器（DMA_IFCR）

DMA_IFCR 的各位就是用来清除 DMA_ISR 的对应位的，通过写 0 清除。在 DMA_ISR 被置位后，我们必须通过向该位寄存器对应的位写入 0 来清除。

寄存器 3：DMA 通道 x 配置寄存器（DMA_CCRx）(x=1~7, 下同)

该寄存器的我们在这里就不贴出来了，见《【中文】STM32F 系列 ARM 内核 32 位高性能微控制器参考手册 V10_1.pdf》第 150 页 10.4.3 一节。该寄存器控制着 DMA 的很多相关信息，包括数据宽度、外设及存储器的宽度、通道优先级、增量模式、传输方向、中断允许、使能等都是通过该寄存器来设置的。所以 DMA_CCRx 是 DMA 传输的核心控制寄存器。

寄存器：DMA 通道 x 传输数据量寄存器（DMA_CNDTRx）

这个寄存器控制 DMA 通道 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值会随着传输的进行而减少，当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成了。所以可以通过这个寄存器的值来知道当前 DMA 传输的进度。

寄存器 5：DMA 通道 x 的外设地址寄存器（DMA_CPARx）

该寄存器用来存储 STM32 外设的地址，比如我们使用串口 1，那么该寄存器必须写入串口 1 的地址（其实就是&USART2_DR）。如果使用其他外设，就修改成相应外设的地址就行了。

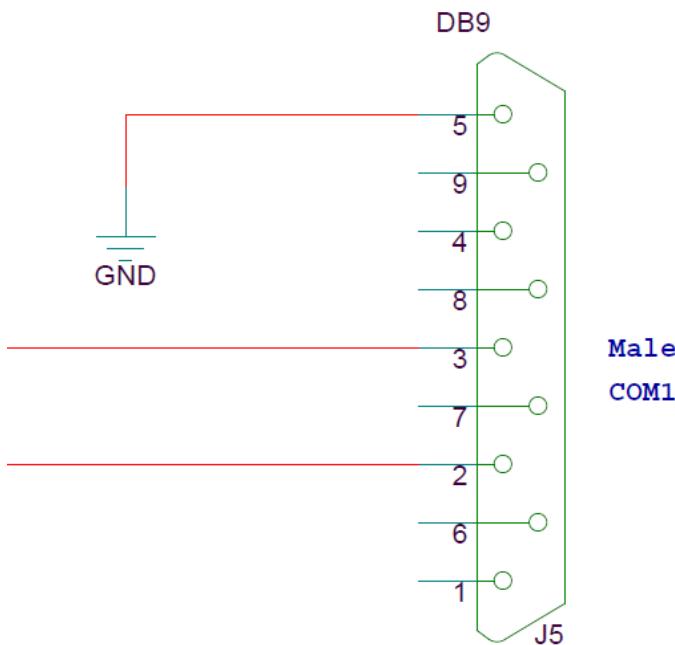
寄存器 6：DMA 通道 x 的存储器地址寄存器（DMA_CMARx）

该寄存器和 DMA_CPARx 差不多，但是是用来放存储器的地址的。比如我们使用 SendBuf[] 数组来做存储器，那么我们在 DMA_CMARx 中写入&SendBuff 就可以了。

寄存器的详细分析大家可以参考《【中文】STM32F 系列 ARM 内核 32 位高性能微控制器参考手册 V10_1.pdf》DMA 的章节，我们这里就分析到这里。

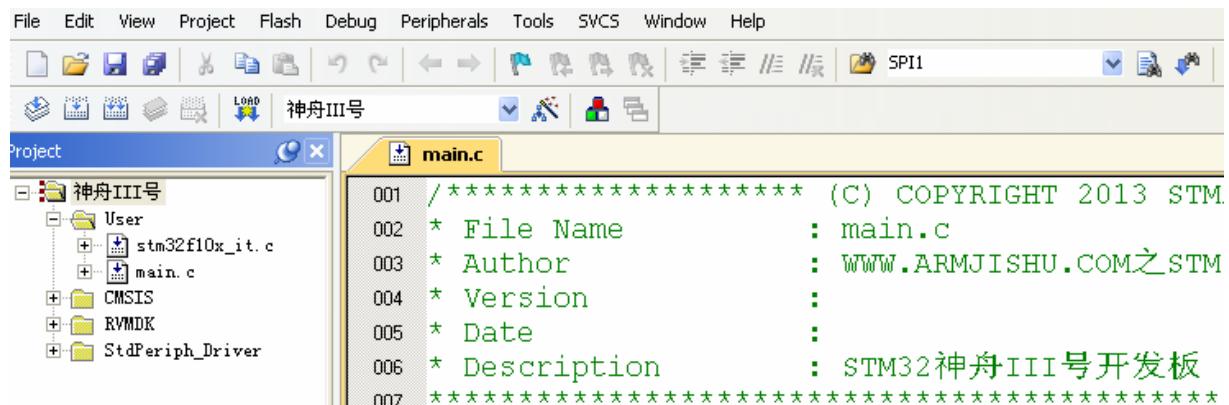
7.63.3 硬件设计

DMA 属于 STM32 内部来连接好串口 1



7.63.4 软件设计

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    u16 i;
    u8 t=0;
    u8 j,mask=0;
    /* USARTx configured as follow:
     - BaudRate = 115200 baud
     - Word Length = 8 Bits
     - One Stop Bit
     - No parity
     - Hardware flow control disabled (RTS and CTS signals)
     - Receive and transmit enabled */
    USART_InitStructure.USART_BaudRate = 2400;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    STM32_Shenzhou_COMInit(&USART_InitStructure);

    /* 初始化系统定时器SysTick,每秒中断1000次 */
    SZ_STM32_SysTickInit(1000);
```

代码分析1：调用STM32_Shenzhou_COMInit()函数初始化串口，配置串口的波特率为2400，前面已经有介绍。

代码分析2：调用SZ_STM32_SysTickInit() 函数初始化定时器。

代码分析3：调用MYDMA_Config() 函数初始化内部温度传感器。

```
MYDMA_Config(DMA1_Channel14,(u32)&USART1->DR,(u32)SendBuff,5168);
```

```

void MYDMA_Config(DMA_Channel_TypeDef* DMA_CHx, u32 cpar, u32 cmar, u16 cndtr)
{
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //使能DMA传输

    DMA_DeInit(DMA_CHx); //将DMA的通道7寄存器重设为缺省值
    DMA1_MEM_LEN=cndtr;
    DMA_InitStructure.DMA_PeripheralBaseAddr = cpar; //DMA外设ADC地址
    DMA_InitStructure.DMA_MemoryBaseAddr = cmar; //DMA内存基地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST; //数据传输方向，从内存读取发
    DMA_InitStructure.DMA_BufferSize = cndtr; //DMA通道的DMA缓存的大小
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //外设地址寄存
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; //内存地址寄存器递增
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte; //数据字节
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte; //数据宽度为8位
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal; //工作在正常缓存模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_Medium; //DMA通道 x拥有中优先级
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; //DMA通道x没有设置为内存到内存传输
    DMA_Init(DMA_CHx, &DMA_InitStructure); //根据DMA_InitStruct中指定的参数初始化DM
}

```

函数 MYDMA_Config () 传入 4 个参数。第一个参数是 DMA1_Channel4。我们这里用到的是串口 1 的传送 (USART1_TX)，它对应的是通道 4。通道的选择要参考手册《【中文】STM32F 系列 ARM 内核 32 位高性能微控制器参考手册 V10_1.pdf》。如下图：

| 外设 | 通道1 | 通道2 | 通道3 | 通道4 | 通道5 | 通道6 | 通道7 |
|------------------|----------|-----------|---------------------|-----------------------------------|-----------|-----------------------|----------------------|
| ADC | ADC1 | | | | | | |
| SPI | | SPI1_RX | SPI1_TX | SPI2_RX | SPI2_TX | | |
| USART | | USART3_TX | USART3_RX | USART1_TX | USART1_RX | USART2_RX | USART2_TX |
| I ² C | | | | I2C2_TX | I2C2_RX | I2C1_TX | I2C1_RX |
| TIM1 | | TIM1_CH1 | TIM1_CH2 | TIM1_TX4 TIM1_TRIG TIM1_COM | TIM1_UP | TIM1_CH3 | |
| TIM2 | TIM2_CH3 | TIM2_UP | | | TIM2_CH1 | | TIM2_CH2 TIM2_CH4 |
| TIM3 | | TIM3_CH3 | TIM3_CH4 TIM3_UP | | | TIM3_CH1 TIM3_TRIG | |
| TIM4 | TIM4_CH1 | | | TIM4_CH2 | TIM4_CH3 | | TIM4_UP |

第二、第三个参数分别是外设串口 1 的地址&USART1_DR、我们开辟的存储器 SendBuff 【】。我们预先将要发送的数据送入 SendBuff 【】，设定串口以 DMA 方式传输数据后，数据从 SendBuff 【】发送到串口。第四个参数是要传输的数据量。

代码分析 4：将要传输的数据送入存储空间 SendBuff 【】。

```
j = sizeof(TEXT_TO_SEND);
for (i=0; i<5168; i++) //填充ASCII字符集数据
{
    if (t>=j) //加入换行符
    {
        if (mask)
        {
            SendBuff[i]=0x0a;
            t=0;
        } else
        {
            SendBuff[i]=0x0d;
            mask++;
        }
    }
    else //复制TEXT_TO_SEND语句
    {
        mask=0;
        SendBuff[i] = TEXT_TO_SEND[t];
        t++;
    }
}
```

代码中 5168 是我们要传输的数据量。TEXT_TO_SEND[]是我们定义的要传输的数据，TEXT_TO_SEND[]={"www.armjishu.comAa STM32 DMA 串口实验"}。上面提过 SendBuff 【】是我们开辟的存储器。这段代码的意思是将 EXT_TO_SEND 定义的语句复制到我们开辟的空间中。

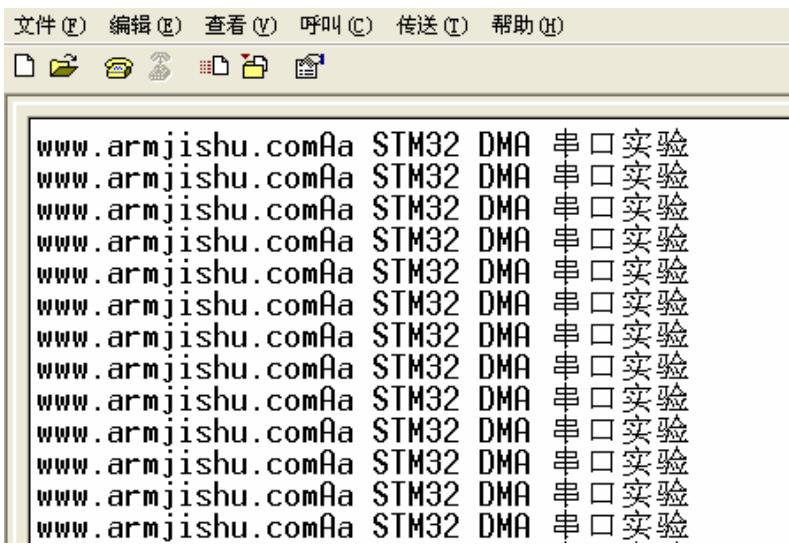
代码分析 5：进入 while (1) 循环

```
while(1)
{
    USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE); //使能串口1的DMA发送
    MYDMA_Enable(DMA1_Channel4); //开始一次DMA传输！
    //等待DMA传输完成，此时我们来做另外一些事，点灯
    //实际应用中，传输数据期间，可以执行另外的任务
    while(1)
    {
        if (DMA_GetFlagStatus(DMA1_FLAG_TC4) != RESET) //判断通道4传输完成
        {
            DMA_ClearFlag(DMA1_FLAG_TC4); //清除通道4传输完成标志
            printf("\n\rDMA传输完成\n ");
            break;
        }
    }
}
```

在 while 循环中，调用函数 USART_DMACmd () 使能串口 1 的 DMA 发送，它配置的是串口 1 的 USART_CR3 寄存器。使能完后，调用 MYDMA_Enable () 函数， DMA 开始传输数据。这时候串口 1 就开始有数据了。传输完成后，调用 DMA_ClearFlag(DMA1_FLAG_TC4)，清除通道 4 传输完成标志，为下一次传输做准备。

7.63.5 下载验证

在代码编译成功之后，我们通过下载代码到 STM32 神舟 III 号开发板上，按下复位键，STM32 通过 DMA 方式不断的往串口发送数据，串口显示如下：



7.64 STM32 内部温度传感器实验

本章我们将向大家介绍STM32的内部温度传感器。在本章中，我们将利用STM32的内部温度传感器来读取温度值，并在彩屏上显示出来。本章分为如下几个部分：

- 1 STM32 内部温度传感器简介
- 2 实验原理
- 3 硬件设计
- 4 软件设计
- 5 下载验证

7.64.1 STM32 内部温度传感器简介

STM32 有一个内部的温度传感器，可以用来测量 CPU 及周围的温度(TA)。该温度传感器在内部和 ADCx_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值。温度传感器模拟输入推荐采样时间是 $17.1 \mu s$ 。STM32 的内部温度传感器支持的温度范围为：-40~125 度。精度比较差，为 $\pm 1.5^{\circ}C$ 左右。

STM32 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部通道就差不多了。关于 ADC 的设置，我们在前面已经进行了详细的介绍，这里就不再多说。

接下来我们介绍一下和温度传感器设置相关的 2 个地方。

第一个地方，我们要使用 STM32 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CR2 的 AWDEN 位 (bit23) 设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32 的内部温度传感器固定的连接在 ADC 的通道 16 上，所以，我们在设置好 ADC 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以计算出当前温度。

7.64.2 实验原理

本实验的实验原理：通过内部温度 ADC，将内部温度传感器的电压值取出。通过电压值转换为对应的温度。

怎么通过 ADC 读取温传感器的值？

温度传感器在内部和 `ADC1_IN16` 输入通道相连接，此通道把传感器输出的电压值转换为数字值。所以通过获取通道 `ADC1_IN16` 的转换值，就可得到我们想要的温度传感器的电压值。

现在，我们就可以总结一下 STM32 内部温度传感器使用的步骤了，如下：

1) 设置 ADC，开启内部温度传感器。

关于如何设置ADC，在ADC实验已经介绍了，我们采用与上一节相似的设置。不同的是上一节温度传感器是读取外部通道的值，而内部温度传感器相当与把通道端口连接在内部温度传感器上。所以这里，我们要开启内部温度传感器功能：`ADC_TempSensorVrefintCmd(ENABLE);`

2) 读取通道 16 的 AD 值，计算结果。

在设置完之后，我们就可以读取温度传感器的电压值了，得到该值就可以用上面的公式计算温度值。从 ADC 实验的 ADC 通道与 GPIO 对应表可以知道，内部温度传感器是通过对应的是 ADC 的通道 16。其它的跟上一节的讲解是一样的。

怎么转换电压值为温度值？

在官方提供的手册《【中文】STM32F 系列 ARM 内核 32 位高性能微控制器参考手册 V10_1.pdf》169 页 11.10 章节中，给出了转换的公式及说明，根据这个公式可以实现电压值和温度值间的转换。

6. 利用下列公式得出温度

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

这里：

V_{25} = V_{SENSE} 在 25°C 时的数值

Avg_Slope = 温度与 V_{SENSE} 曲线的平均斜率(单位为 mV/ °C 或 μ V/ °C)

参考数据手册的电气特性章节中 V_{25} 和 Avg_Slope 的实际值。

我们对，温度的计算公式进行说明：

V_{25} =Vsense 在 25 度时的数值（典型值为： 1.43）。

V_{SENSE} 温度传感器输出当前的电压值的计算方法：

$V_{\text{SENSE}}=\text{ADC_ConveredValue}*\text{Vdd}/\text{Vdd_convert_value}$ (0xFFFF)

`ADC_ConveredValue` 是 `ADC_DR` 寄存器中的结果。`Vdd` 是参考电压。`Vdd_convert_value` 是 4096，因为我们这个 STM32 它的转换的精度是 12 位的。

Avg_Slope =温度与 Vsense 曲线的平均斜率（单位为 mv/°C 或 uv/°C）（典型值为 4.3Mv/°C）。

有了这些数据，就可以利用以上公式，计算出当前温度传感器的温度了。

7.64.3 硬件设计

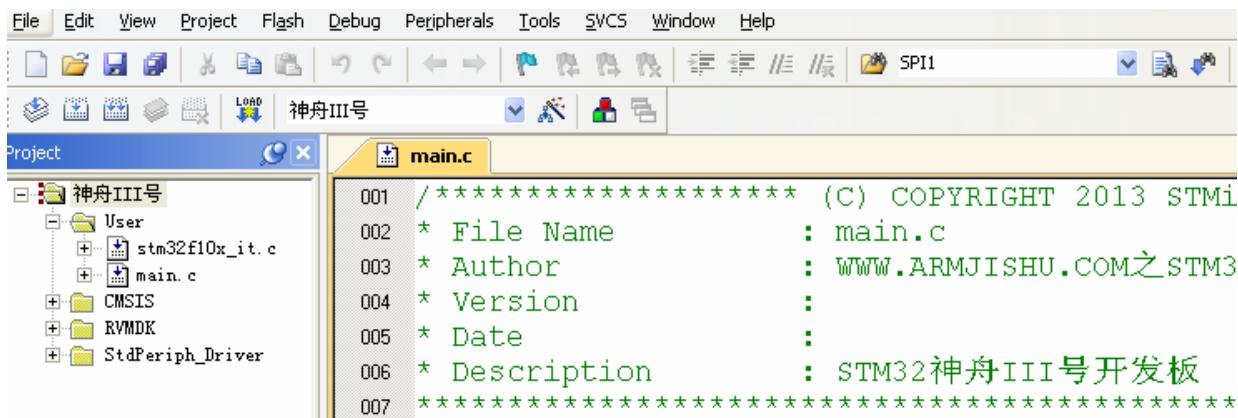
本实验用到的硬件资源有：

- 2) ADC
- 3) 内部温度传感器

前两个之前均有介绍，而内部温度传感器也是在STM32内部，不需要外部设置，我们只需要软件设置就OK了

7.64.4 软件设计

入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



代码分析1：对串口的参数进行配置。

```
USART_Initstructure USART_BaudRate = 115200;
USART_Initstructure USART_WordLength = USART_WordLength_8b;
USART_Initstructure USART_StopBits = USART_StopBits_1;
USART_Initstructure USART_Parity = USART_Parity_No;
USART_Initstructure USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_Initstructure USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
```

代码分析2：调用函数STM32_Shenzhou_COMInit(&USART_InitStructure)初始化串口。

代码分析3：调用T_Adc_Init()函数初始化内部温度传感器。

```
void T_Adc_Init(void) //ADC通道初始化
{
    ADC_InitTypeDef ADC_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_ADC1
    RCC_ADCCLKConfig(RCC_PCLK2_Div6); //分频因子6时钟为72M/6=12MHz
    ADC_DeInit(ADC1); //将外设 ADC1 的全部寄存器重设为缺省值
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; //ADC工作模式:AI
    ADC_InitStructure.ADC_ScanConvMode = DISABLE; //模数转换工作在单通
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; //模数转换工作在
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //ADC数据右对齐
    ADC_InitStructure.ADC_NbOfChannel = 1; //顺序进行规则转换的ADC通道数
    ADC_Init(ADC1, &ADC_InitStructure); //根据ADC_Initstruct中指定的参数
}
```

ADC_TempSensorVrefintCmd(ENABLE); //开启内部温度传感器

ADC_Cmd(ADC1, ENABLE); //使能指定的ADC1

ADC_ResetCalibration(ADC1); //重置指定的ADC1的复位寄存器

在初始化函数T_Adc_Init()中，对内部温度传感器进行了一系列的初始化后，调用函数ADC_TempSensorVrefintCmd()开启内部温度传感器。还进行使能指定的ADC等其它设置，这些设置主要用到了寄存器ADC_CR2。比如函数ADC_TempSensorVrefintCmd()

```
void ADC_TempSensorVrefintCmd(FunctionalState Newstate)
{
    /* Check the parameters */
    assert_param(IS_FUNCTIONAL_STATE(Newstate));
    if (Newstate != DISABLE)
    {
        /* Enable the temperature sensor and Vrefint channel*/
        ADC1->CR2 |= CR2_TSVREFE_Set;
    }
    else
    {
        /* Disable the temperature sensor and Vrefint channel*/
        ADC1->CR2 &= CR2_TSVREFE_Reset;
    }
}
```

而，CR2_TSVREFE_Set=((uint32_t)0x00800000)刚好对寄存器ADC_CR2进行置位，启动温度传感器。

11.12.3 ADC控制寄存器 2(ADC_CR2)

地址偏移: 0x08

复位值: 0x0000 0000

| | | | | | | | | | | | | | | | |
|--------------|--------------|----|-------|-------------|-------------|--------------|-------------|-------------|----|----|------------|-----|------|------|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | TS VREFE | SW START | JSW START | EXT TRIG | EXTSEL[2:0] | | 保留 | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| JEXT TRIG | JEXTSEL[2:0] | | ALIGN | 保留 | | DMA | 保留 | | | | RST CAL | CAL | CONT | ADON | |
| TW | TW | TW | TW | TW | TW | TW | TW | | | | TW | TW | TW | TW | |

位23 **TSVREFE**: 温度传感器和V_{REFINT}使能 (Temperature sensor and V_{REFINT} enable)
 该位由软件设置和清除，用于开启或禁止温度传感器和V_{REFINT}通道。在多于1个ADC的器件中，该位仅出现在ADC1中。
 0: 禁止温度传感器和V_{REFINT};
 1: 启用温度传感器和V_{REFINT}。

代码分析4: 通过函数T_Adc_Init()进行初始化ADC，启动温度传感器等设置后，我们就可以对内部温度传感器进行操作了。

```
while(1)
{
    adcx=T_Get_Adc_Average(ADC_CH_TEMP,10); //获取ADC的值
    printf("\r\n 神舟III号开发板ADC的值: %d\r\n", adcx);

    temperate=(float)adcx*(3.3/4096); //计算温度传感器的电压值
    temperate=(1.43-temperate)/0.0043+25; //通过电压值计算出当前温度值
    adcx=temperate; //显示当前的温度
    printf("\r\n 神舟III号开发板温度值: %d°C\r\n", adcx);
    Delay(1000000);
}
```

代码中函数T_Get_Adc_Average(ADC_CH_TEMP,10)获取ADC的值。温度传感器在内部和ADC1_IN16输入通道相连接，我们通过通道ADC1_IN16，多次取数求平均值得到我们所想要的温度传感器的ADC数据。

```
//获取通道ch的转换值
//取times次,然后平均
u16 T_Get_Adc_Average(u8 ch,u8 times)
{
    u32 temp_val=0;
    u8 t;
    for(t=0;t<times;t++)
    {
        temp_val+=T_Get_Adc(ch);
        delay(5);
    }
    return temp_val/times;
}
```

代码分析 5: 获取 ADC 的值后，我们对它进行串口打印对应的数据。

```
printf("\r\n 神舟III号开发板ADC的值: %d\r\n", adcx);
```

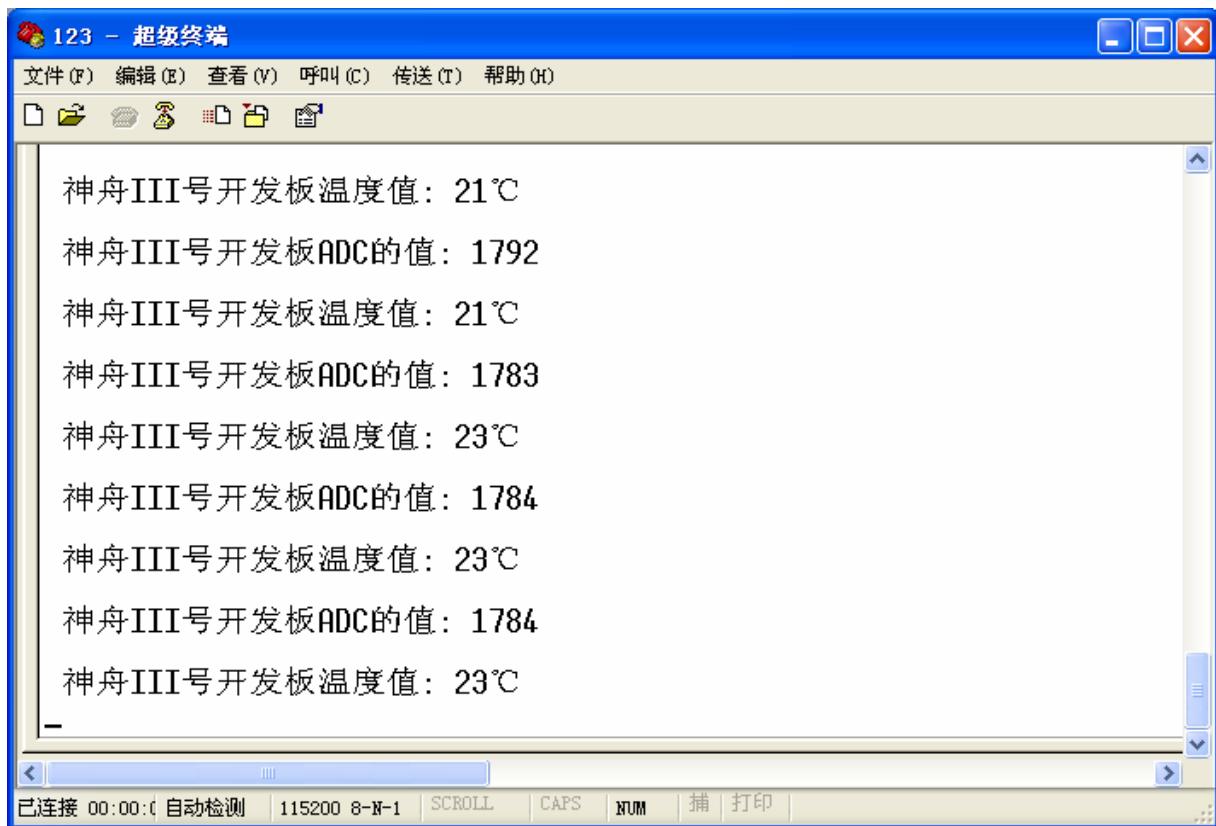
代码分析 6: 通过公式“温度={(V25-VSENSE)/Avg_Slope}+25”，计算出我们想要的温度值。并通过嵌入式专业技术论坛（www.armjishu.com）出品

串口显示出来。

```
temperate=(float)adcx*(3.3/4096); //计算温度传感器的电压值  
temperate=(1.43-temperate)/0.0043+25;//通过电压值计算出当前温度值  
adcx=temperate;//显示当前的温度  
printf("\r\n 神舟III号开发板温度值: %d°C\r\n", adcx);  
Delay(1000000);
```

7.64.5 下载验证

在代码编译成功之后，我们通过下载代码到 STM32 神舟 III 号开发板上，可以看到串口打印如图所示：



7.65 温度传感器 (DS18B20) 实验

7.65.1 意义与作用

温度传感在现代检测技术中，传感器占据着不可动摇的重要位置。主机对数据的处理能力已经相当的强，但是对现实世界中的模拟量却无能为力。如果没有各种精确可靠的传感器对非电量和模拟信号进行检测并提供可靠的数据，那计算机也无法发挥他应有的作用。传感器把非电量转换为电量，经过放大处理后，转换为数字量输入计算机，由计算机对信号进行分析处理。从而传感器技术与计算机技术结合起来，对自动化和信息化起重要作用。

在本章节，我们以 DS18B20 温度传感器做为我们的一个测温器件，检测我们的外界温度，得到较为准确的环境温度。在本章中，我们将学习使用单总线技术，通过它来实现 STM32 和外部温度传感器 (DS18B20) 的通信，并把从温度传感器得到的温度显示出来。

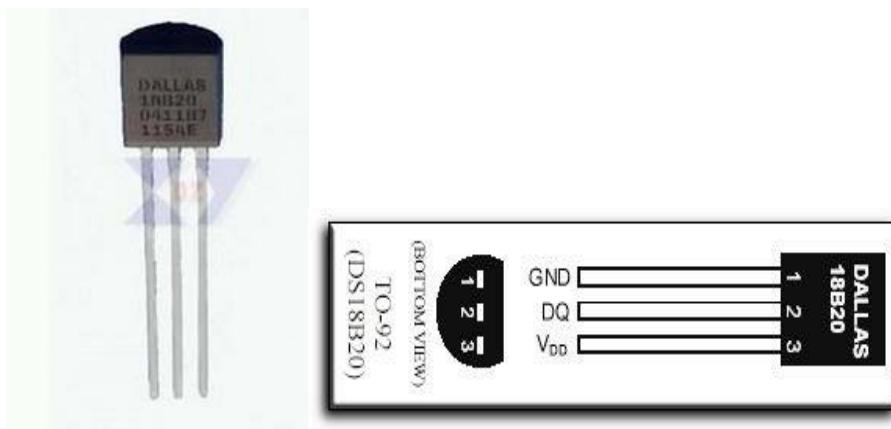
7.65.2 温度传感器的介绍

温度传感器市场的需求

能快速测出温度的工具我们可以称为温度计，温度计的核心部件是温度传感器。为什么会出现温度传感器呢？因为它可以代替人体去感受外界的温度，并且能够把一个温度数字化，具体化，避免了人体去直接接触感觉温度，过低或者过高的温度可能会冻伤或者是烧伤人体，方便人类去管理和控制调整温度的变化。日常生活中各个领域中都使用到温度传感器，像家庭中的电子温度计、工厂检测机器温度的仪器、交通工具上的等等。它为人类带来了很大的便利。

什么是温度传感器

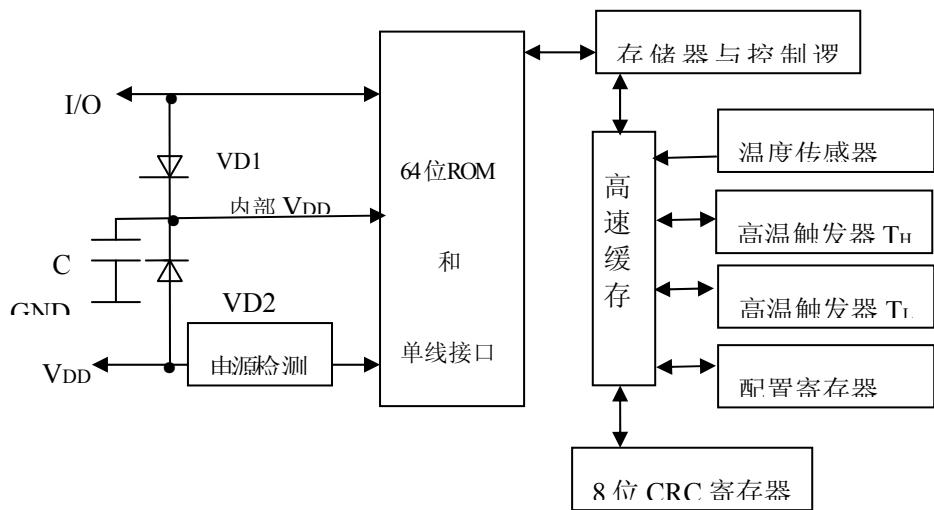
温度传感器是指能感受温度并转换成可用输出信号的传感器，我们都知道温度理论上是无上限，也无下限的；例如火山或者太阳的温度可以非常高，南北极的冰雪温度也可以非常低；所以温度范围是非常广泛的，小到零下几十摄氏度，大到上千摄氏度；根据温度变化的灵敏度和精度，又有很对不同温度传感器的种类。像家庭用的大多是 0 到 100 摄氏度的温度计，有些工厂用到的是几千度摄氏度的温度计，像那些钢筋铁厂的，单片机中我们一般使用的是 18B20 的一种温度传感器，在这个章节我们就以它为例。下图为它的实物图和封装图



单线数字温度传感器实物图（左）与外部封装图（右）

DS18B20 内部结构

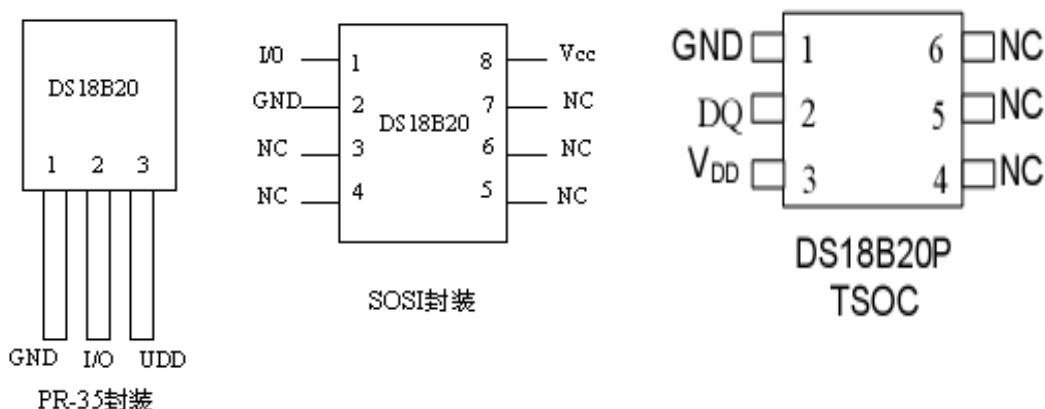
DS18B20 采用 3 脚 PR35 封装或 8 脚 SOIC 封装，其内部结构主要由四部分组成，如下图所示：



DS18B20 内部结构

- (1) 64 位光刻 ROM
- (2) 温度传感器
- (3) 非挥发的温度报警触发器 TH 和 TL
- (4) 高速暂存器。

18B20 的管脚排列如下图所示：



DS18B20 引脚分布图

| 序号 | 名称 | 引脚功能描述 |
|----|----------|--|
| 1 | GND | 地信号 |
| 2 | DQ (I/O) | 数据输入/输出引脚。开漏单总线接口引脚。当被用着在寄生电源下，也可以向器件提供电源。 |

| | | |
|---|-----------|--------------------------------|
| 3 | VDD (Vcc) | 可选择的 VDD 引脚。当工作于寄生电源时，此引脚必须接地。 |
|---|-----------|--------------------------------|

灵敏度与精度

前面我们说了灵敏度和精度，那什么是灵敏度呢？灵敏度就是温度变化了，可能人体需要 1 秒钟才能感知，但有的毕竟灵敏的温度传感器只需要零点几秒就可以感知；那什么是精度呢？有的传感器精度是 0.5 摄氏度，有的传感器精度是 0.1 摄氏度，意思就是温度每升降这个单位值，就会被感知的最小刻度。

18B20 温度传感器的通信方式

我们在这章节中讨论的 18B20 温度传感器是其中的一款普通的传感器，它能把读到的温度转成二进制的格式发送给单片机，单片机收到它发过来的数据后再转换成比如十进制的方式让我们能知道当前的温度是多少。这个我们会在下一节详细描述。它的测温范围是 -55~125 °C。固有测温分辨率为 0.5 °C。在这个范围内的温度它都能测试出来。

18B20 温度传感器只需要单线就能与单片机进行通信了，数据就是由这根信号线与单片机进行连接的，加上电源与地，也就是说 18B20 温度传感器只有 3 根引脚就可以了。在这个单线上并不是只能接一个 18B20 温度传感器的，能接多个 18B20 温度传感器，每个 18B20 温度传感器都有一个地址，单片机访问哪个地址对应的那个 18B20 温度传感器就能与单片机进行通信。

那么温度传感器怎么能够感知温度的呢？这就要涉及温度传感器究竟是什么材料制作，以及制作的原理了；比如有的材料可以感知常温下的温度，但是如果温度超过 300 摄氏度，这个材料有可能会被融化，所以每个传感器的功能都是有所限制的，也就是说，温度可测范围是有限制的，但是这个世界的传感器技术每天都在进步和发展，对于外界温度的感知技术也是在不断进步的，或许有一天，我们能有一个传感器能进入到太阳的内部去检测它的温度也说不定。

DS18B20 性能特点

- (1) 独特的单线接口方式：DS18B20 与微处理器连接时仅需要一条口线即可实现微处理器与 DS18B20 的双向通讯。
- (2) 在使用中不需要任何外围元件。
- (3) 可用数据线供电，电压范围：3.0~5.5 V。
- (4) 测温范围：-55~125 °C。固有测温分辨率为 0.5 °C。
- (5) 通过编程可实现 9~12 位的数字读数方式。
- (6) 用户可自设定非易失性的报警上下限值。
- (7) 支持多点组网功能，多个 DS18B20 可以并联在唯一的三线上，实现多点测温。
- (8) 负压特性，电源极性接反时，温度计不会因发热而烧毁，但不能正常工作。

它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~12 位的数字值读数方式。可以分别在 93.75ms 和 750ms 内完成 9 位到 12 位的数字量，并且从 18B20 读出的信息或写入 18B20 的信息仅需要一根口线（单线接口）读写，温度变换功率来源于数据总线，总线本身也可以向所挂接的 18B20 供电，无需额外为电源。

7.65.3 DS18B20温度传感器的使用

DS18B20 的连接方式

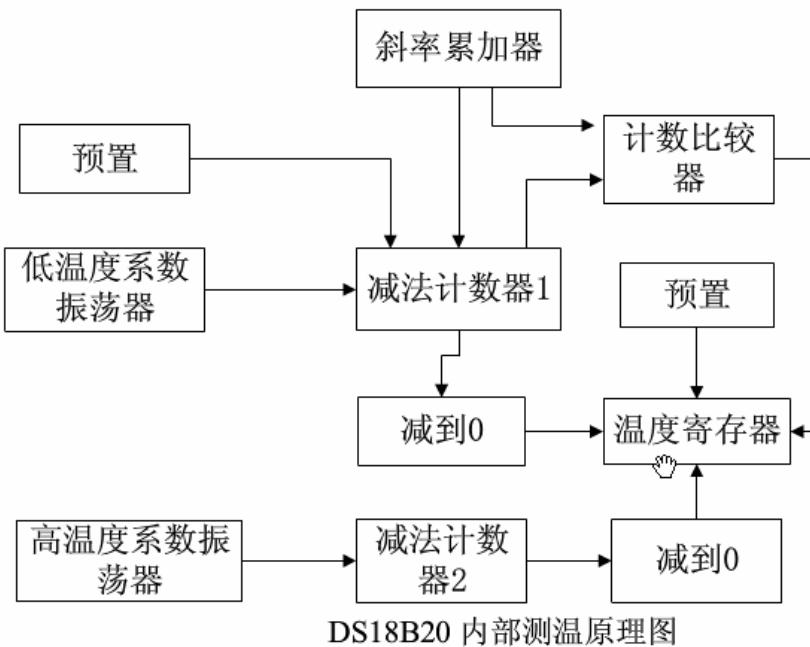
在硬件上，DS18B20 与单片机的连接有两种方法，一种是 Vcc 接外部电源，GND 接地，I/O 与单片机的 I/O 线相连；另一种是用寄生电源供电，此时 VDD、GND 接地，I/O 接单片机 I/O。无论是内部寄生电源还是外部供电，I/O 口线接一个 4.7KΩ 左右的上拉电阻即可。

DS18B20 的延时问题

虽然 DS18B20 有诸多优点，但使用起来并非易事，由于采用单总线数据传输方式，DS18B20 的数据 I/O 均由同一条线完成。因此，对读写的操作时序要求严格。为保证 DS18B20 的严格 I/O 时序，需要做较精确的延时。在 DS18B20 操作中，有了比较精确的延时保证，就可以对 DS18B20 进行读写操作、温度转换及显示等操作。

18B20 温度传感器的测温原理

我们先来看下 18B20 温度传感器内部的测温原理图，通过该图我们认识 18B20 温度传感器是如何实现测温原理的。



DS18B20 内部测温原理图

上图中，低温度系数晶振的振荡频率受温度的影响很小，用于产生固定频率的脉冲信号送给减法计数器 1，高温度系数晶振的振荡频率随温度变化而明显改变，所产生的信号作为减法计数器 2 的脉冲输入。图中还隐含着计数门，当计数门打开时，DS18B20 就对低温系数振荡器产生的时钟脉冲进行计数，进而完成温度测量。计数门的开启时间由高温度系数振荡器决定，每次测量前，首先将 -55°C 所对应的基数分别置入减法计数器 1 和温度寄存器中，减法计数器 1 和温度寄存器被预置在 -55°C 所对应的一个基数值。减法计数器 1 对低温度系数晶振产生的脉冲信号进行减法计数，当减法计数器 1 的预置值减到 0 时温度寄存器的值将加 1，减法计数器 1 的预置将重新被装入，减法计数器 1 重新开始对低温度系数晶振产生的脉冲信号进行计数，如此循环直到减法计数器 2 计数到 0 时，停止温度寄存器值的累加，此时温度寄存器中的数值即为所测温度。

图中的斜率累加器用于补偿和修正测温过程中的非线性，其输出用于修正减法计数器的预置值，只要计数们仍未关闭就重复上述过程，直到温度寄存器值达到被测温度值，这个就是 18B20 温度传感器的测温原理。

另外，由于 DS18B20 单线通信功能是分时完成的，它有严格的时隙概念，因此读写时序很重要。系统对 DS18B20 的各种操作必须按协议进行。操作协议为：初始化 DS18B20（发复位脉冲）→发 ROM 功能命令→发存储器操作命令→处理数据。

DS18B20 控制方法

单片机是如何控制温度传感器的呢？我们来看下 CPU 对温度传感器的控制命令

DS18B20 有六条控制命令，如下表所示：

DS18B20 控制命令

| 指 令 | 约定代码 | 操 作 说 明 |
|-----------|------|-----------------------------------|
| 温度转换 | 44H | 启动 DS18B20 进行温度转换 |
| 读暂存器 | BEH | 读暂存器 9 个字节内容 |
| 写暂存器 | 4EH | 将数据写入暂存器的 TH、TL 字节 |
| 复制暂存器 | 48H | 把暂存器的 TH、TL 字节写到 E2RAM 中 |
| 重新调 E2RAM | B8H | 把 E2RAM 中的 TH、TL 字节写到暂存器 TH、TL 字节 |
| 读电源供电方式 | B4H | 启动 DS18B20 发送电源供电方式的信号给主 CPU |

DS18B20 的寄存器与操作指令

CPU 对 DS18B20 的访问流程是：先对 DS18B20 初始化，再进行 ROM 操作命令，最后才能对存储器操作，数据操作。DS18B20 每一步操作都要遵循严格的工作时序和通信协议。如主机控制 DS18B20 完成温度转换这一过程，根据 DS18B20 的通讯协议，须经三个步骤：每一次读写之前都要对 DS18B20 进行复位，复位成功后发送一条 ROM 指令，最后发送 RAM 指令，这样才能对 DS18B20 进行预定的操作。

那什么是 ROM 指令，什么又是 RAM 指令呢？

(1) DS18B20 温度传感器 64 b 闪速 ROM 的结构如表 1-1，64 位光刻 ROM 是出厂前被光刻好的，它可以看作是该 DS18B20 的地址序列号，相当于我们的身份证一样，不同的器件地址序列号不同：

表 1-1 闪速 ROM 的结构

| | | |
|-----------|---------|---------------|
| 8b 检验 CRC | 48b 序列号 | 8b 工厂代码 (10H) |
|-----------|---------|---------------|

开始 8 位是产品类型的编号，接着是每个器件的惟一的序号，共有 48 位，最后 8 位是前 56 位的 CRC 校验码，这也是多个 DS18B20 可以采用一线进行通信的原因。

(2) 非易失性温度报警触发器 TH 和 TL，可通过软件写入用户报警上下限。

(3) 高速暂存存储器

DS18B20 温度传感器的内部存储器包括一个高速暂存 RAM 和一个非易失性的可电擦除的 E2RAM。后者用于存储 TH, TL 值。数据先写入 RAM, 经校验后再传给 E2RAM。而配置寄存器为高速暂存器中的第 5 个字节, 他的内容用于确定温度值的数字转换分辨率, DS18B20 工作时按此寄存器中的分辨率将温度转换为相应精度的数值。该字节各位的定义如表 1-2:

表 1-2 DS18B20 内部存储器

| | | | | | | | |
|----|----|----|---|---|---|---|---|
| TM | R1 | R0 | 1 | 1 | 1 | 1 | 1 |
|----|----|----|---|---|---|---|---|

高速暂存存储器的低 5 位一直都是 1, TM 是测试模式位, 用于设置 DS18B20 在工作模式还是在测试模式。在 DS18B20 出厂时该位被设置为 0, 用户不要去改动, R1 和 R0 决定温度转换的精度位数, 即是来设置分辨率, 如表 1-3 所示 (DS18B20 出厂时被设置为 12 位)。

表 1-3 R1 和 R0 模式表

| R1 | R0 | 分辨率 | 温度最大转换时间/mm |
|----|----|------|-------------|
| 0 | 0 | 9 位 | 93.75 |
| 0 | 1 | 10 位 | 187.5 |
| 1 | 0 | 11 位 | 275.00 |
| 1 | 1 | 12 位 | 750.00 |

由表 1-3 可见, 设定的分辨率越高, 所需要的温度数据转换时间就越长。因此, 在实际应用中要在分辨率和转换时间权衡考虑。

高速暂存存储器除了配置寄存器外, 还有其他 8 个字节组成, 其分配如下所示。其中温度信息 (第 1, 2 字节)、TH 和 TL 值第 3, 4 字节、第 6~8 字节未用, 表现为全逻辑 1; 第 9 字节读出的是前面所有 8 个字节的 CRC 码, 可用来保证通信正确。

表 1-4 R1 和 R0 模式表

| 温度低位 | 温度高位 | TH | TL | 配置 | 保留 | 保留 | 保留 | 8 位 CRC |
|------|------|----|----|----|----|----|----|---------|
|------|------|----|----|----|----|----|----|---------|

当 DS18B20 接收到温度转换命令后, 开始启动转换。转换完成后的温度值就以 16 位带符号扩展的二进制补码形式存储在高速暂存存储器的第 1, 2 字节。单片机可通过单线接口读到该数据, 读取时低位在前, 高位在后, 数据格式以 0.625 °C/LSB 形式表示。温度值格式如表 1-5:

表 1-5 温度值格式

| | | | | | | | |
|----|----|----|----|-----|-----|-----|-----|
| 23 | 22 | 21 | 20 | 2-1 | 2-2 | 2-3 | 2-4 |
| S | S | S | S | S | 26 | 25 | 24 |

对应的温度计算: 当符号位 S=0 时, 直接将二进制位转换为十进制; 当 S=1 时, 先将补码变换为原码, 再计算十进制值。表 1-6 是对应的一部分温度值。

表 1-6 部分温度值

| 温度/°C | 二进制表示 | | 十六进制表示 |
|----------|----------|----------|--------|
| +125 | 00000111 | 11010000 | 07D0H |
| +25.0625 | 00000001 | 10010001 | 0191H |

| | | | |
|----------|----------|----------|-------|
| +0.5 | 00000000 | 00001000 | 0008H |
| 0 | 00000000 | 00000000 | 0000H |
| -0.5 | 11111111 | 11111000 | FFF8H |
| -25.0625 | 11111110 | 01101111 | FE6FH |
| -55 | 11111100 | 10010000 | FC90H |

DS18B20 完成温度转换后，就把测得的温度值与 TH, TL 作比较，若 T>TH 或 T<TL，则将该器件内的告警标志置位，并对主机发出的告警搜索命令作出响应。因此，可用多只 DS18B20 同时测量温度并进行告警搜索。

(4) CRC 的产生在 64 b ROM 的最高有效字节中存储有循环冗余校验码 (CRC)。主机根据 ROM 的前 56 位来计算 CRC 值，并和存入 DS18B20 中的 CRC 值做比较，以判断主机收到的 ROM 数据是否正确。

7.65.4 软件设计

根据实验现象初步分析主函数

STM32 神舟 I 号通过与 DS18B20 的连接，烧录对应的例程，能把温度传感器检测到的温度经过各种显示器件显示出来，如串口打印、数码管、显示屏或者是彩屏等，用户能通过这些显示设备查看当前的温度，有的例程还能检测多点的温度。温度传感器只需要一根引脚即可完成单片机与它之间的通信。

初始化

```

/*
 * 所以main函数不需要再次重复初始化时钟。默认初始化系统主时钟为72MHz。
 * SystemInit() 函数的实现位于system_stm32f10x.c文件中。
 */
uint32_t Times = 0;

/* 初始化板载LED指示灯 */
SZ_STM32_LEDInit(LED1);
SZ_STM32_LEDInit(LED2);
SZ_STM32_LEDInit(LED3);

SZ_STM32_SysTickInit(1000);
/* 注意串口1使用Printf时"SZ_STM32F103RB_LIB.c"文件中fputc定义中设备改为sz_STM32_COM1 */
/* 串口1初始化 */
SZ_STM32_COMInit(COM1, 115200);

/* Infinite loop 主循环 */
printf("\n\r ======\n\r =欢迎您访问WWW.ARMJISHU.COM技术论坛获取更多关于神舟系列开发板的资料=\n\r");
printf("\n\r 神舟I号开发板 DS18B20温度传感器实验 \n\r");
delay_init(72); //延时初始化
while(DS18B20_Init()); //初始化DS18B20, 兼检测18B20
{
    ...
}

```

初始化函数中，我们对 LED、系统定时器、延时函数与串口进行了初始化，这样我们才能正常使用这些功能。

串口打印——“WWW.ARMJISHU.COM”

WWW.ARmjishu.COM

初始化温度传感器，并检测，如果初始化通过，检测没问题的话，跳过该循环，如果失败，则串口打印：神舟 III 号开发板提示：18b20 err.

```
while(DS18B20_Init())//初始化DS18B20,  
{  
    printf("\n\rds18b20 err");  
    delay_ms(500);  
}
```



假设检测到 18B20 则进入 while 循环。从温度传感器得到温度，并打印出来

```
temp=DS18B20_Get_Temp();  
if(temp<0)  
{  
    temp=-temp;  
    printf("-");  
}  
printf("当前温度为:%d.%dc\n\r",temp/10,temp%10);  
Delay_ARMJISHU(6600000);  
}
```

当前温度为:20.0C
当前温度为:20.0C
当前温度为:20.0C
当前温度为:20.0C
当前温度为:20.0C
当前温度为:20.0C
当前温度为:20.0C
当前温度为:20.0C

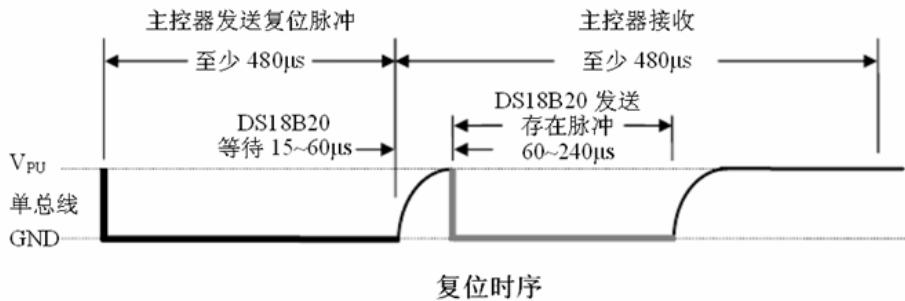
[进一步分析关联代码函数](#)

下面我们主要看下与温度传感器关联的函数

a) 温度传感器复位初始化函数

```
// 复位DS18B20 复位脉冲(最短为480us的低电平信号)
void DS18B20_Reset(void)
{
    DS18B20_IO_OUT(); // SET AS OUTPUT
    DS18B20_OUT_LOW; // 拉低DQ
    delay_us(750); // 拉低750us
    DS18B20_OUT_HIGH;
    delay_us(15); // 15US
}
```

我们看下温度传感器的复位时序图



我们能通过上面温度传感器工作的时序图看出，数据输入/输出引脚电平由高变低，延时 480us 到 960us 后（**我们的程序中是 750us**）把数据输入/输出引脚拉高，等待 15 到 60us 后（**我们的程序中是 15us**）接收到 60—240us 的存在脉冲，再判断 x 是否等于 0，不等于 0 表示初始化失败，反之则初始化成功，我们可以看到等待 60—240us 过后，应该是拉高的了。

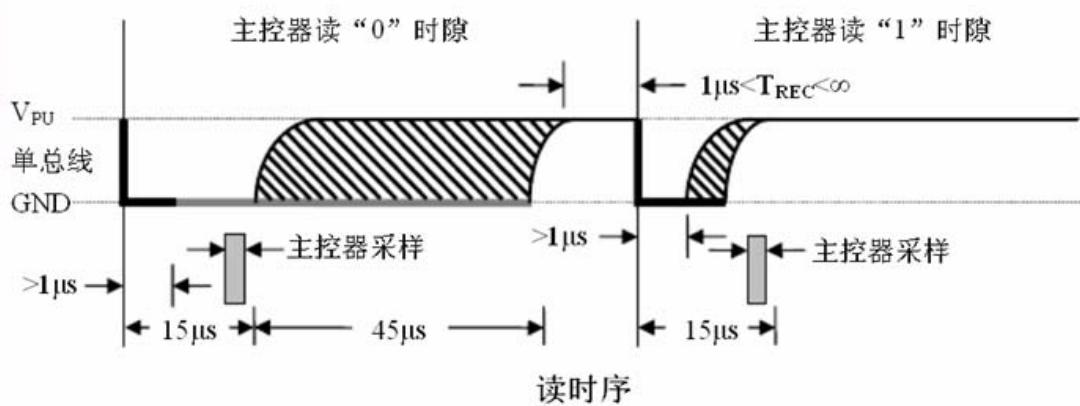
读取温度传感器数据函数

```
//从DS18B20读取一个位
//返回值: 1/0
u8 DS18B20_Read_Bit(void)           // read one bit
{
    u8 data;

    DS18B20_IO_OUT();      //SET AS OUTPUT
    DS18B20_OUT_LOW;       //拉低
    delay_us(3);
    DS18B20_OUT_HIGH;     //拉高
    DS18B20_IO_IN();       //SET AS INPUT
    delay_us(6);
    if(DS18B20_DATA_IN)
    {
        data=1;
    }
    else
    {
        data=0;
    }
    delay_us(50);

    return data;
}
```

我们看下温度传感器读时序图



在发出读暂存器命令后，主机必须立即产生读时隙以便 DS18B20 提供所需数据。DS18B20 只有在检测到主设备启动读时序后才向主设备传输数据。所以一般在主设备发送了读数据命令后，必须马上产生读时序，以便 DS18B20 能够传输数据。所有的读时序都至少需要 $60 \mu\text{s}$ ，且在两次独立的读时序之间至少需要 $1 \mu\text{s}$ 的恢复时间。每个读时序都由主设备发起，先使数据线为高电平，然后拉低数据线至少 $1 \mu\text{s}$ ，再释放数据线。在主设备发出读时序之后，DS18B20 开始在数据线上发送数据 0 或 1。若其发送 1，则保持数据线为高电平。若发送 0，则 DS18B20 拉低数据线，在该时序结束后释放数据线。DS18B20 发出的数据在起始时序之后 $15 \mu\text{s}$ 内保证可靠有效。因而主设备在读时序期间必须释放数据线，并且要在时序开始后的 $15 \mu\text{s}$ 之内读取数据线状态。

读取一个字节数据

```
//从DS18B20读取一个字节
//返回值：读到的数据
u8 DS18B20_Read_Byte(void)      // read one byte
{
    u8 i,j,dat;
    dat=0;
    for (i=1;i<=8;i++)
    {
        j=DS18B20_Read_Bit();
        dat=dat>>1;
        dat+= (j<<7);
    }

    return dat;
}
```

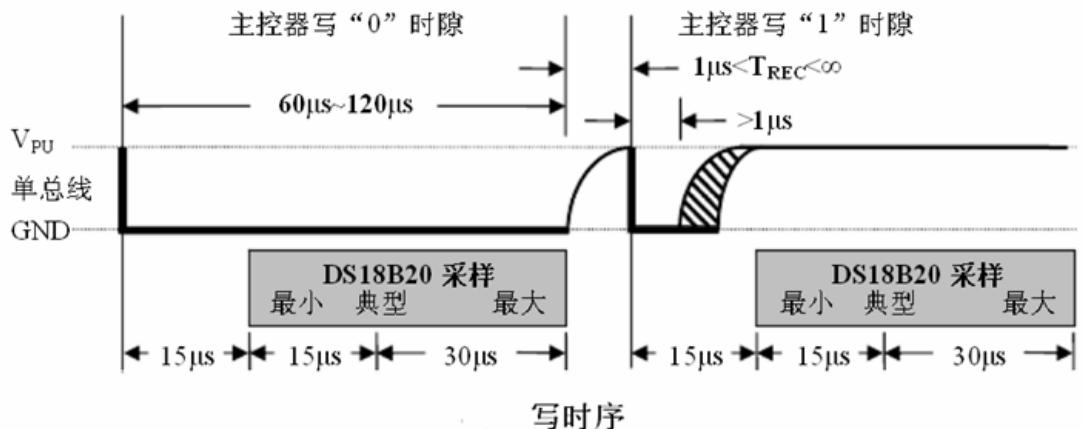
累积读取 8 位为一个字节

写函数——写数据到温度传感器

```
//写一个字节到DS18B20
//dat: 要写入的字节
/***************************************** 写DS18B20函数 *****/
void DS18B20_Write_Byte(u8 dat)
{
    u8 j;
    u8 testb;
    DS18B20_IO_OUT(); // SET AS OUTPUT;
    for (j=1;j<=8;j++)
    {
        testb=dat&0x01;
        dat=dat>>1;

        DS18B20_OUT_LOW;// Write 1
        delay_us(2);
        if (testb)
        {
            DS18B20_OUT_HIGH;
        }
        delay_us(55);
        DS18B20_OUT_HIGH;
        delay_us(5);
    }
}
```

我们看下写数据的时序图：



当单片机将总线从高拉至低电平时，就产生写时间隙，15us 之内应将所需写的位送到总线上 DS18B20 会在其后的 15 到 60us 的一个时间窗口内采样单总线。在采样的时间窗口内，如果总线为高电平，主机会向 DS18B20 写入 1；如果总线为低电平，主机会向 DS18B20 写入 0。所有的写时隙必须至少有 60us 的持续时间。相邻两个写时隙必须要有最少 1us 的恢复时间。所有的写时隙（写 0 和写 1）都由拉低总线产生

温度转换与读取温度函数

```
//从ds18b20得到温度值
//精度: 0.1C
//返回值: 温度值 (-550~1250)
short DS18B20_Get_Temp(void)
{
    u8 i, temp;
    u8 TL, TH;
    short tem;
    u8 str[9];

    //开始温度转换
    DS18B20_Reset();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc); // skip rom
    DS18B20_Write_Byte(0x44); // convert

    //开始读取温度
    DS18B20_Reset();
    DS18B20_Check();

    DS18B20_Write_Byte(0xcc); // skip rom
    DS18B20_Write_Byte(0xbe); // convert
    //TL=DS18B20_Read_Byte(); // LSB
    //TH=DS18B20_Read_Byte(); // MSB

    for (i=0;i<9;i++)
    {
        str[i] = DS18B20_Read_Byte();
    }
    if (GetCRC(str, 9) == 0)
        printf(" CRC OK ");
    else
        printf(" CRC ERR ");
}
```

```
for (i=0;i<9;i++)
{
    printf(" %02X", str[i]);
}

TL = str[0]; // LSB
TH = str[1]; // MSB
if(TH>7)
{
    TH=~TH;
    TL=~TL;
    temp=0;//温度为负
}
else
{
    temp=1;//温度为正
}
tem=TH<<8 | TL; //获得不带符号位的11位温度值

//转换 *0.625
tem = tem*10;
tem = tem>>4;

if(temp) return tem; //返回温度值
else return -tem;
}
```

温度转换时，我们需要经过 3 个步骤，分别是：

- 初始化
- ROM 操作指令 (Skip ROM)

在单点总线系统中此命令通过允许总线主机不提供 64 位 ROM 编码而访问存储器操作节省时间。
(在多于一个的从属器件责不可以使用该命令。

- 存储器操作指令 (Convert T[44h]启动温度变换)

而读取温度时需要经过 5 个步骤：

- 复位 DS18B20

- 发出 Skip ROM 命令(CCH)

- 发出 Read 命令(BEH)

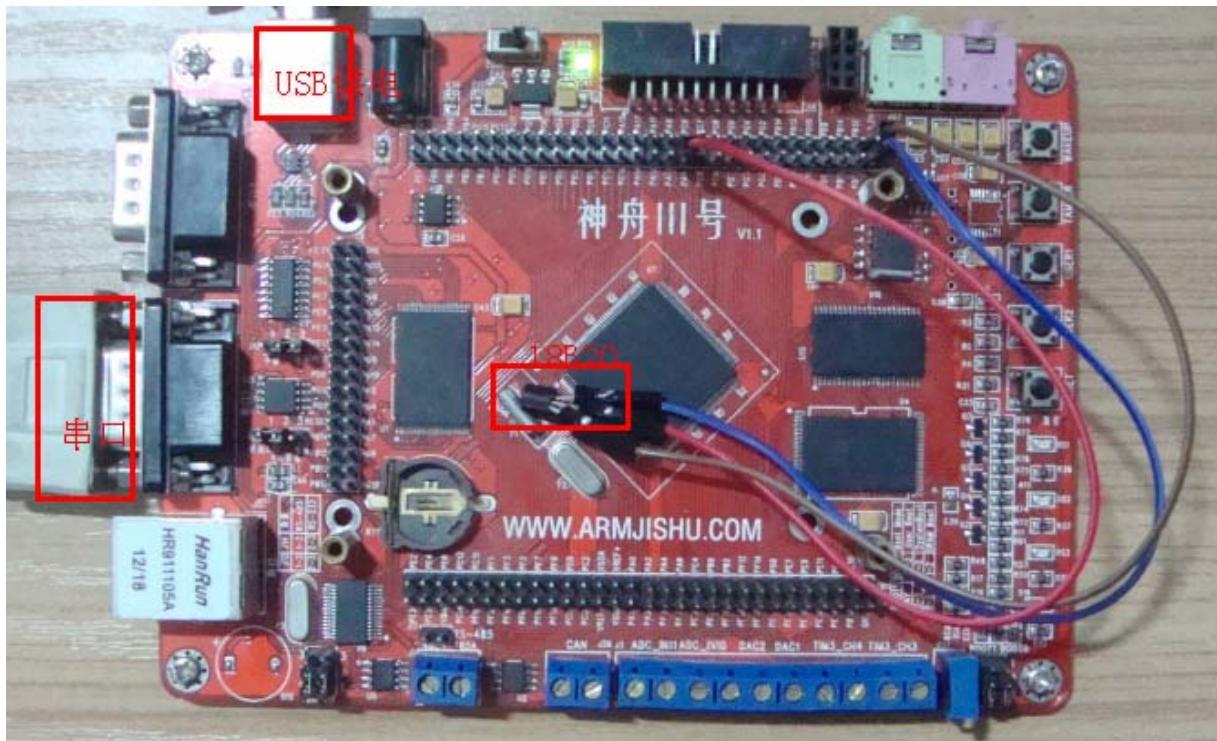
- 读两字节的温度

- 温度格式转换

7.65.5 硬件设计与实验现象

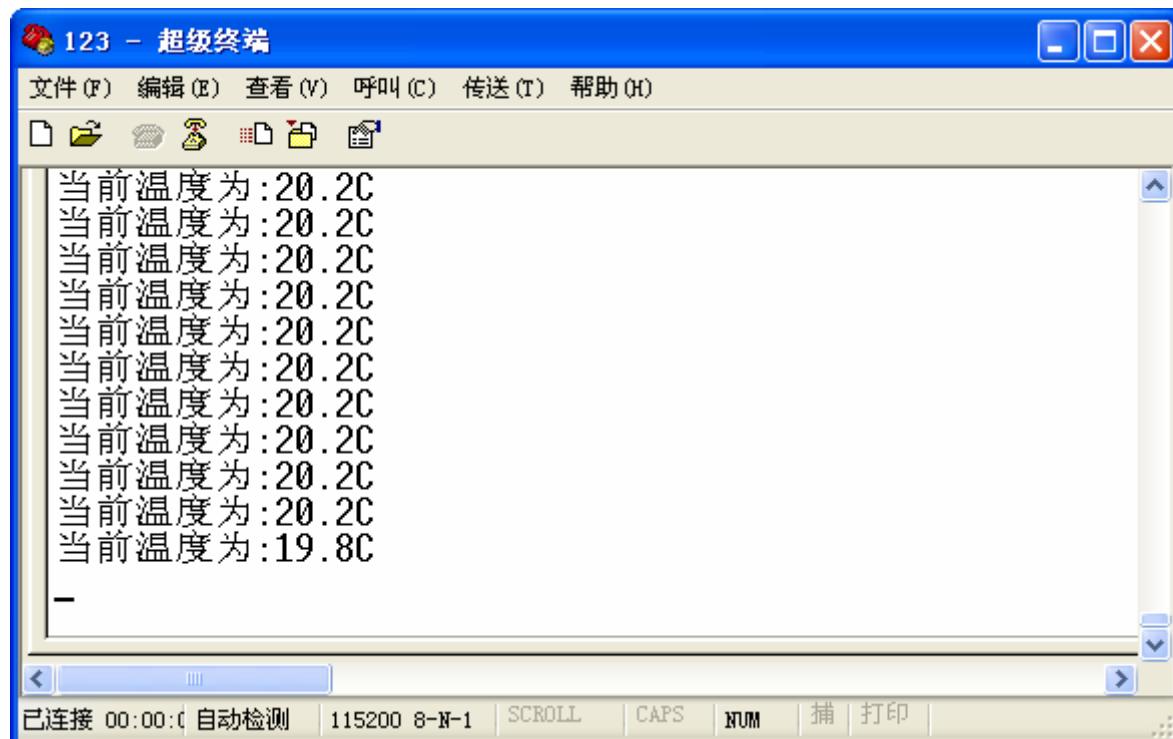
硬件连接

本实验只要接 18B20 温度传感器的数据线接到处理器的 I/O 口上即可，另外电源与地引脚分别接到开发板的电源与地，在这个实验中，我们把温度传感器的数据引脚接到了 PA0 上，所以相对应的代码我们也会根据控制 PA0 进行编写，有兴趣的用户可自己尝试修改管脚控制温度传感器。



实验现象

烧录程序到开发板上，接上串口线到电脑终端或者串口助手打印数据，串口 1 打印数据如下：



判断是否连接到温度传感器，识别产品编码与校验CRC没问题后读出温度传感器读到的温度值。

7.66 DHT11数字温湿度传感器实验

7.66.1 简介

DHT11 是一款湿温度一体化的数字传感器。

该传感器包括一个电阻式测湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。通过单片机等微处理器简单的电路连接就能够实时的采集本地湿度和温度。DHT11 与单片机之间能采用简单的单总线进行通信，仅仅需要一个 I/O 口。传感器内部湿度和温度数据 40Bit 的数据一次性传给单片机，数据采用校验和方式进行校验，有效的保证数据传输的准确性。DHT11 功耗很低，5V 电源电压下，工作平均最大电流 0.5mA。

7.66.2 DHT11的工作原理

DHT11 的湿度检测运用电容式结构，并采用具有不同保护的“微型结构”检测电极系统与聚合物覆盖层来组成传感器芯片的电容，除保护电容湿敏元件的原有特性外，还可以抵御来自外来界的影响。由于它将温度传感器与湿度传感器结合在一起而构成了一个单一的个体，因为测量精度较高且可精确的得出露点，同时不会产生由于温度与湿度之间随温度变化引起的误差。CMOSensTM 技术不仅将湿度传感器结合在一起，而且还将信号放大器、模数转换器、校准数据存储器、标准 IIC 总线等电路全部集成在一个芯片内。

DHT11 的每一个传感器都是在极为精确的湿度室中校准的，DHT11 传感器的校准系数预先存储在 OTP 内存中。经校准湿度和温度传感器与一个 14 位的 A/D 转换器相连，可将转换后的数字温湿度值送给二线 IIC 总线器件，从而将数字信号转换为符合 IIC 总线协议的串行数字信号。

工作电压范围：3.3V-5.5V

工作电流：平均 0.5mA

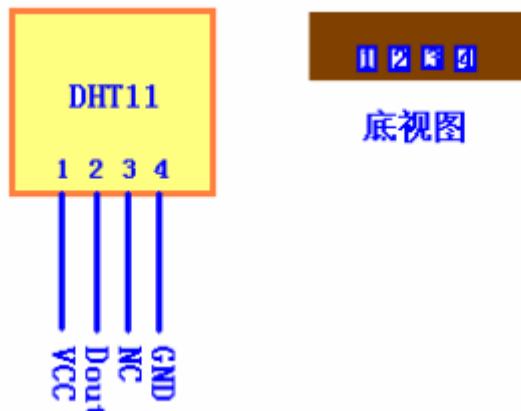
输出：单总线数字信号

测量范围：湿度 20~90%RH，温度 0~50°C

精度：湿度±5%，温度±2°C

分辨率：湿度 1%，温度 1°C

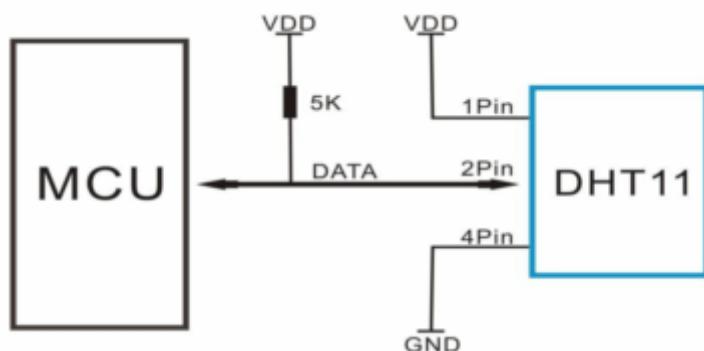
DHT11 的管脚排列如图所示：



DHT11 的引脚功能：

| Pin | 名称 | 注释 |
|-----|------|-------------|
| 1 | VDD | 供电 3—5.5VDC |
| 2 | DATA | 串行数据，单总线 |
| 3 | NC | 空脚，请悬空 |
| 4 | GND | 接地，电源负极 |

7.66.3 硬件连接与温度协议分析



典型电路

虽然 DHT11 与 DS18B20 类似，都是单总线访问，但是 DHT11 的访问，相对 DS18B20 来说要简单很多。下面我们来看看 DHT11 的数据结构。

DHT11 数字湿温度传感器采用单总线数据格式。即单个数据引脚端口完成输入输出双向传输。其数据包由 5Byte (40Bit) 组成。数据分小数部分和整数部分，一次完整的数据传输为 40bit，高位先出。DHT11 的数据格式为：8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数数据+8bit 校验和。其中校验和数据为前四个字节相加。

示例一：接收到的 40 位数据为：

| | | | | |
|-----------|-----------|-----------|-----------|-----------|
| 0011 0101 | 0000 0000 | 0001 1000 | 0000 0000 | 0100 1101 |
| 湿度高 8 位 | 湿度低 8 位 | 温度高 8 位 | 温度低 8 位 | 校验位 |

计算：

$0011\ 0101 + 0000\ 0000 + 0001\ 1000 + 0000\ 0000 = 0100\ 1101$

接收数据正确：

湿度：0011 0101=35H=53%RH

温度：0001 1000=18H=24°C

示例二：接收到的 40 位数据为：

| | | | | |
|-----------|-----------|-----------|-----------|-----------|
| 0011 0101 | 0000 0000 | 0001 1000 | 0000 0000 | 0100 1001 |
| 湿度高 8 位 | 湿度低 8 位 | 温度高 8 位 | 温度低 8 位 | 校验位 |

计算：

$0011\ 0101 + 0000\ 0000 + 0001\ 1000 + 0000\ 0000 = 0100\ 1101$

01001101 不等于 0100 1001

本次接收的数据不正确，放弃，重新接收数据。

DHT11 的值的转换也非常简单直接将值的高 8 位的二进制转换为十进制表示其整数部分，第八位表示小数部分，如上图示例一中的数据；

湿度=湿度的高八位.湿度的低八位=53.0%

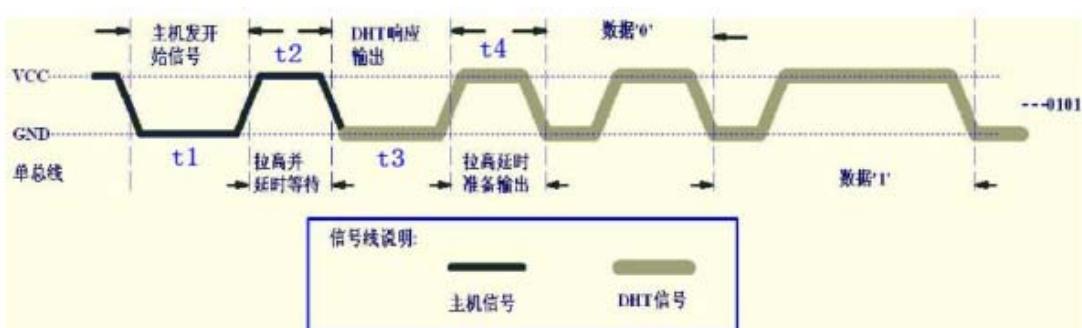
温度=温度的高八位.温度的低八位=24.0 (°C)

校验=湿度的高八位+湿度的低八位+温度的高八位+温度的低八位=77(=湿度+温度)(校验正确)

可以看出，DHT11 的数据格式是十分简单的，DHT11 和 MCU 的一次通信最大为 3ms 左右，建议主机连续读取时间间隔不要小于 100ms。

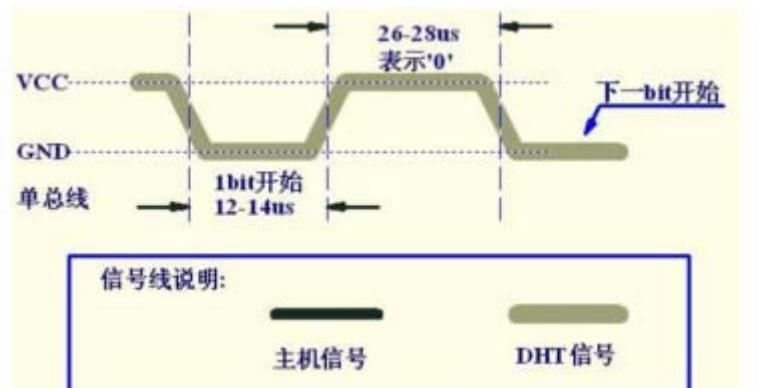
DHT11 时序

下面，我们介绍一下 DHT11 的传输时序。DHT11 的数据发送流程如下图所示；

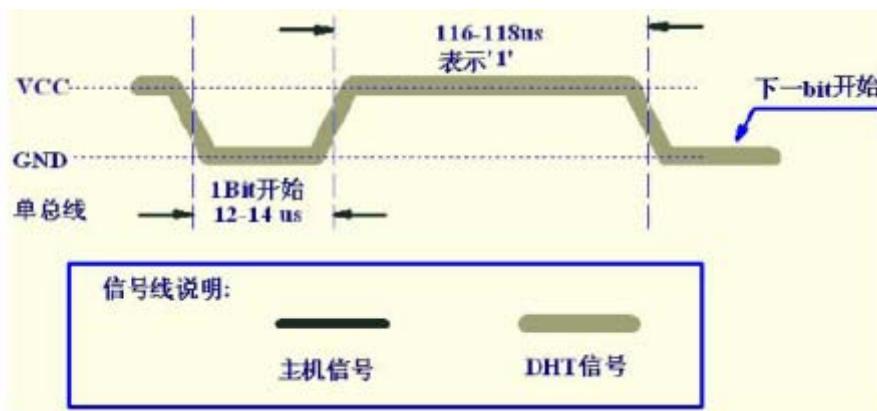


首先主机发送开始信号，即：拉低数据线，保持 t_1 (至少 18ms) 时间，然后拉高数据线 t_2 (20~40us) 时间，然后读取 DHT11 的相应，正常的话，DHT11 会拉低数据线，保持 t_3 (40~50us) 时间，作为响应信号，然后 DHT11 拉高数据线，保持 t_4 (40~50us) 时间后，开始输出数据。

DHT11 输出数字 ‘0’ 的时序如图所示；



DHT11 输出数字 ‘1’ 的时序如图所示；



7.66.4 实验现象

我们将 DHT11 连接到 STM32 神舟 3 号开发板上

| 神州 3 号开发板 | 连接线 | DHT11 | 线色 | 功能 |
|-----------|-----|-------|----|----------|
| 3.3V | 杜邦线 | VDD | 红色 | 供电 |
| PA11 | 杜邦线 | DATA | 蓝色 | 串行数据 单总线 |
| GND | 杜邦线 | NC | 白色 | 悬空 |
| GND | 杜邦线 | GND | 绿色 | 接地 |

同时用串口连接电脑，在电脑上的超级终端上显示当前温度和湿度的值如下图所示：



7.67 三轴加速度模块实验

7.67.1 三轴加速传感器有什么用

我们现在的智能手机能够自动切换横竖屏、玩游戏和切歌等，人们感觉非常的方便和智能，其实这个是由于手机里面我们用到了一个传感器，那就是加速传感器，就是通过这个传感器实现我们的这些功能的，下面我们这章就对这个传感器做一个详细的介绍与使用。而在这章中，我们使用到的是三轴加速度传感器中的一种——**ADXL345 传感器**。

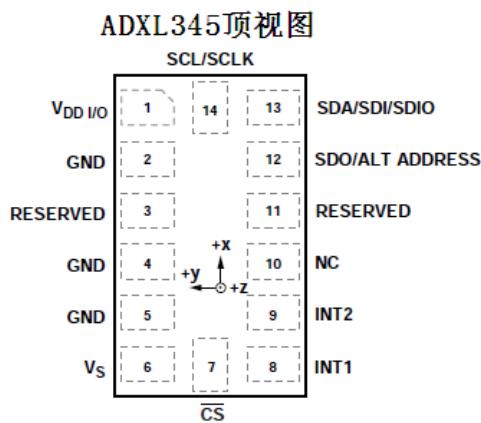
7.67.2 ADXL345简介

ADXL345 是 ADI 公司生产的一款超低功耗 3 轴加速度计，广泛应用于手机、医疗仪器、游戏和定点设备、工业仪器仪表及个人导航设备领域，它的分辨率高达 13 位，测量范围达 $\pm 16\text{g}$ 。数字输出数据为 16 位二进制补码格式，可通过 SPI(3 线或 4 线)或 I2C 数字接口访问。ADXL345 非常适合移动设备应用。它可以在倾斜检测应用中测量静态重力加速度，还可以测量运动或冲击导致的动态加速度。其高分辨率(3.9mg/LSB)，能够测量不到 1.0° 的倾斜角度变化。

ADXL345 实物图与引脚图：



ADXL345 实物图



ADXL345 引脚图

该加速度传感器的特点有：

- 分辨率高。最高 13 位分辨率。
- 量程可变。具有 $+/‐2\text{g}$, $+/‐4\text{g}$, $+/‐8\text{g}$, $+/‐16\text{g}$ 可变的测量范围。
- 灵敏度高。最高达 3.9mg/LSB , 能测量不到 1.0° 的倾斜角度变化。
- 功耗低。 $40\text{~}145\mu\text{A}$ 的超低功耗, 待机模式只有 $0.1\mu\text{A}$ 。
- 尺寸小。整个 IC 尺寸只有 $3\text{mm}\times 5\text{mm}\times 1\text{mm}$, LGA 封装。

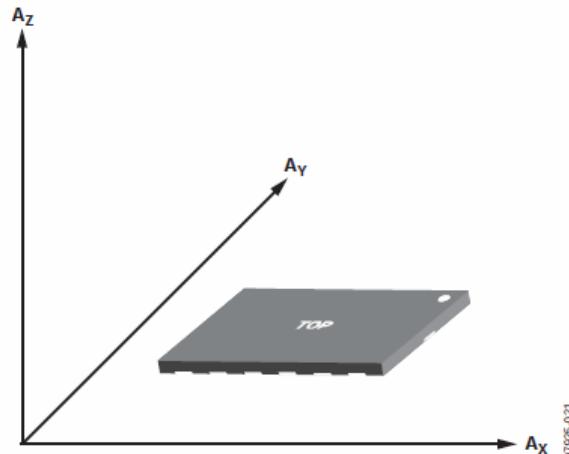
ADXL 支持标准的 I2C 或 SPI 数字接口, 自带 32 级 FIFO 存储, 并且内部有多种运动状态检测和灵活的中断方式等特性。

7.67.3 ADXL345 工作原理

ADXL345 是一款完整的 3 轴加速度测量系统, 可选择的测量范围有 $\pm 2\text{ g}$, $\pm 4\text{ g}$, $\pm 8\text{ g}$ 或 $\pm 16\text{ g}$ 。既能测量运动或冲击导致的动态加速度, 也能测量静止加速度, 例如重力加速度, 使得器件可作为倾斜传感器使用。

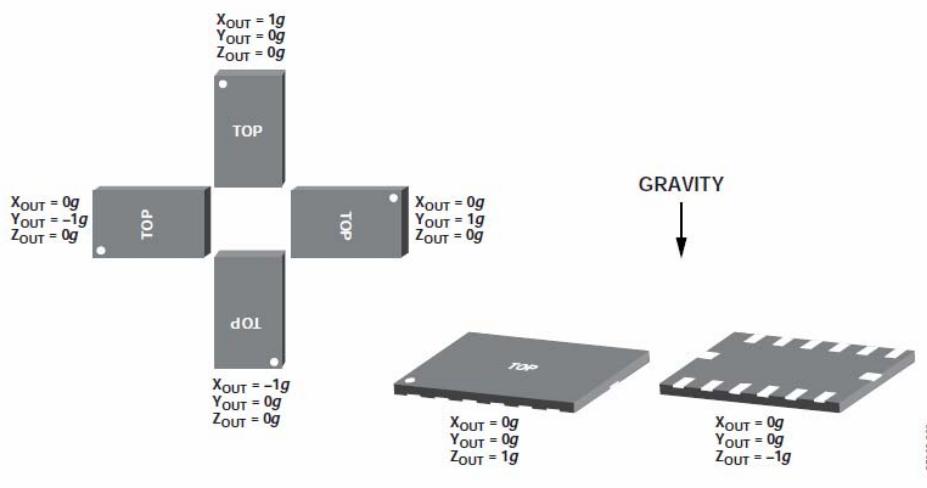
该传感器为多晶硅表面微加工结构, 置于晶圆顶部。由于应用加速度, 多晶硅弹簧悬挂于晶圆表面的结构之上, 提供力量阻力。

差分电容由独立固定板和活动质量连接板组成, 能对结构偏转进行测量。加速度使惯性质量偏转、差分电容失衡, 从而传感器输出的幅度与加速度成正比。相敏解调用于确定加速度的幅度和极性。



加速度灵敏度轴(沿敏感轴加速时相应输出电压增加)

当 ADXL345 沿轴正向加速时，它对正加速度进行检测。在检测重力时用户需要注意，当检测轴的方向与重力的方向相反时检测到的是正加速度。下图所示为输出对重力的响应。

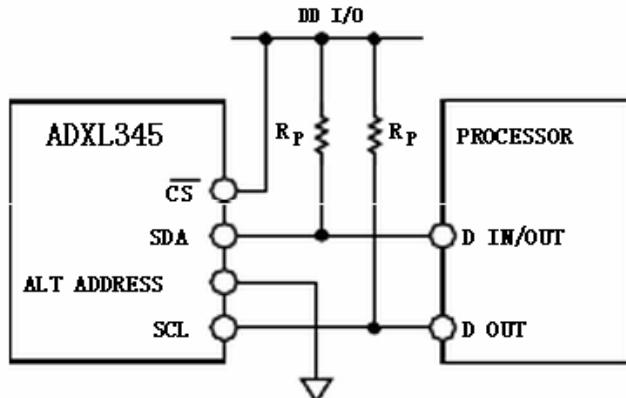


输出响应与相对于重力的方向的关系

7.67.4 ADXL345的通信方式实验原理

加速度传感器会接受外界传递的物理性输入，通过感测器转换为电子信号，在最终转换为可用的信息。主要感应方式是对微小物理量的变化进行测量，在通过电压信号来表示这些变化量。

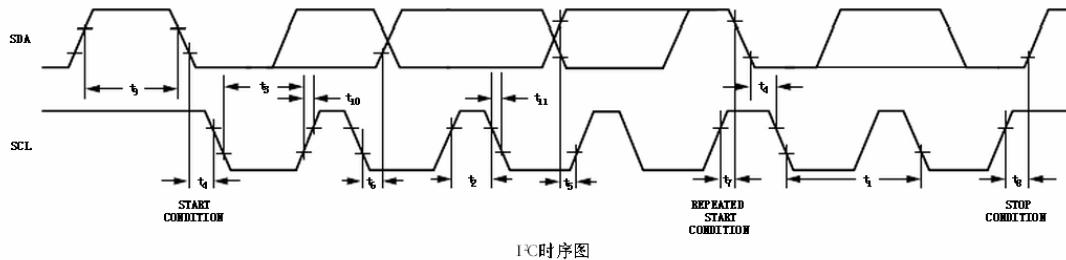
ADXL345 可通过 SPI(3 线或 4 线)或 I2C 数字接口访问，因为我们这里主要用到的是 I2C 通信方式，所以在这里我主要是介绍 I2C 的方式，SPI 我们就暂且略过。如下图所示，CS 引脚拉高至 VDD I/O，ADXL345 处于 I2C 模式，需要简单 2 线式连接。ADXL345 符合《UM10204 I2C 总线规范和用户手册》03 版(2007 年 6 月 19 日，NXP Semiconductors 提供)。如果满足了表 11 和表 12 列出的总线参数，便能支持标准(100 kHz)和快速(400 kHz)数据传输模式。如图 40 所示，支持单个或多个字节的读取/写入。ALT ADDRESS 引脚处于高电平，器件的 7 位 I2C 地址是 0x1D，随后为 R/W 位。这转化为 0x3A 写入，0x3B 读取。通过 ALT ADDRESS 引脚(引脚 12)接地，可以选择备用 I2C 地址 0x53(随后为 R/W 位)。这转化为 0xA6 写入，0xA7 读取。

I²C 连接图 (地址 0x53)

从上图可看出，ADXL345 的连接十分简单，外围需要的器件也极少（就 2 个电容），如上连接（SDO/ALT ADDRESS 接地），则 ADXL345 的地址为 0X53（不含最低位），如果 SDO/ALT ADDRESS 接高，那么 ADXL345 的地址将变为 0X1D（不含最低位）。IIC 通信的时序我们在之前已经介绍过，这里就不再细说了。

对于任何不使用的引脚，没有内部上拉或下拉电阻，因此，CS 引脚或 ALT ADDRESS 引脚悬空或不连接时，任何已知状态或默认状态不存在。使用 I²C 时，CS 引脚必须连接至 VDD I/O，ALT ADDRESS 引脚必须连接至任一 VDD I/O 或接地。

I²C 通信时序图如下：



7.67.5 ADXL345的寄存器说明

1) DATA_FORMAT 寄存器：

DATA_FORMAT 寄存器功能作用：自测力应用至传感器，造成输出数据转换；器件模式选择；中断高低电平选择；器件分辨率模式选择；左右对齐位选择；范围位选择；各轴数据的输出控制等操作功能；

2) BW_RATE 寄存器：

BW_RATE 寄存器功能作用：正常与低功率操作选择；器件带宽和输出数据速率等操作功能；

3) POWER_CTL 寄存器：

POWER_CTL 寄存器功能作用：活动和静止功能的连接位设置；自动休眠功能；待机模式与测量模式选择；工作与休眠模式选择；唤醒功能等控制操作；

4) INT_ENABLE 寄存器:

INT_ENABLE 寄存器功能作用：使能与阻止相应功能中断选择操作；

5) OFSX、OFSY、OFSZ 寄存器:

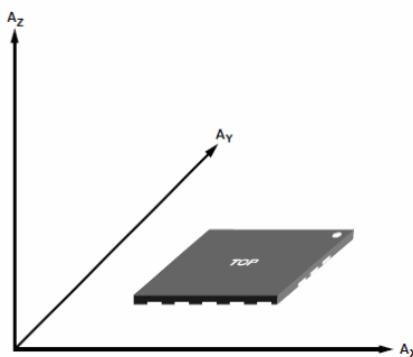
OFSX、OFSY、OFSZ 寄存器功能作用：这三个寄存器都为 8 位寄存器，在二进制补码格式中提供用户设置偏移调整。

ADXL345 有关寄存器就介绍到这里，详细的介绍，请参考 ADXL345 的数据手册。

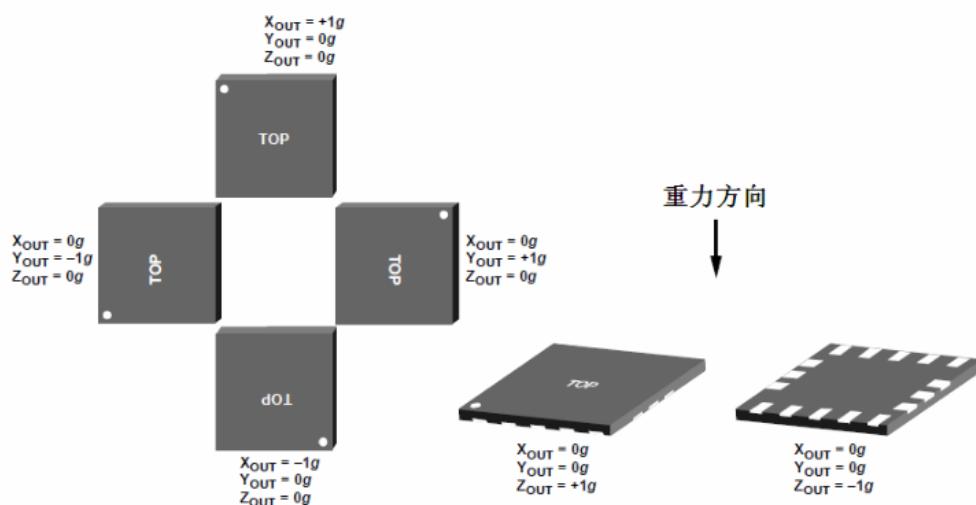
以上几个寄存器做了简单功能介绍，而要对 ADXL345 这几个寄存器要进行数据读写操作都是与 IIC 通信操作有关完成的，只要 ADXL34 配置好管脚和有关的寄存器定义就可以使用。

7.67.6 ADXL345 内部结构与管脚连接

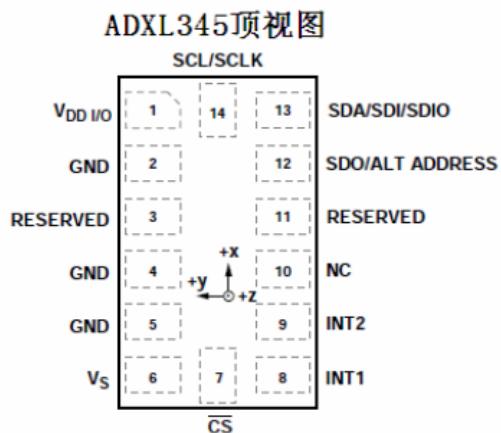
ADXL345 传感器的检测轴如图：



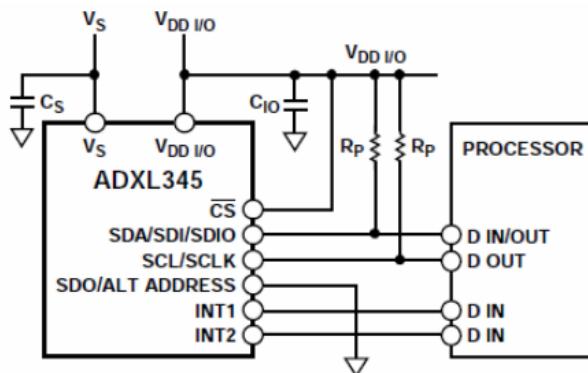
当 ADXL345 沿检测轴正向加速时，它对正加速度进行检测。在检测重力时用户需要注意，当检测轴的方向与重力的方向相反时检测到的是正加速度。输出对重力的响应，下图所示：



以上图所示列出了 ADXL345 在不同摆放方式时的输出，以便后续分析。接下来看看 ADXL345 的引脚图，如图：



ADXL345 支持 SPI 和 IIC 两种通信方式，为了节省 IO 口，采用的是 IIC 方式连接，官方推荐的 IIC 连接电路如图：



从上图可看出，ADXL345 的连接十分简单，外围需要的器件也极少（就 2 个电容），如上连接（SDO/ALT ADDRESS 接地），则 ADXL345 的地址为 0X53（不含最低位），如果 SDO/ALTADDRESS 接高，那么 ADXL345 的地址将变为 0X1D（不含最低位）。IIC 通信的时序在这里就不再细说了（可以参考 IIC 通信章节）。

ADXL345 的初始化：

介绍一下 ADXL345 的初始化步骤。ADXL345 的初始化步骤如下：

- 1) 上电
- 2) 等待 1.1ms
- 3) 初始化命令序列
- 4) 结束

其中上电这个动作发生在开发板第一次上电的时候，在上电之后，等待 1.1ms 左右，就可以开始发送初始化序列了，初始化序列一结束，ADXL345 就开始正常工作了。这里的初始化序列，最简单的只需要配置 3 个寄存器，如表所示：

| 步骤 | 寄存器地址 | 寄存器名字 | 寄存器值 | 功能描述 |
|----|-------|-------------|------|------------------|
| 1 | 0X31 | DATA_FORMAT | 0X0B | ±16g, 13 位模式 |
| 2 | 0X2D | POWER_CTL | 0X08 | 测量模式 |
| 3 | 0X2E | INT_ENABLE | 0X80 | 使能 DATA_READY 中断 |

发送以上序列给 ADXL345 以后，ADXL345 即开始正常工作，ADXL345 就介绍到这里，详细的介绍，请参考 ADXL345 的数据手册。

7.67.7 实验原理

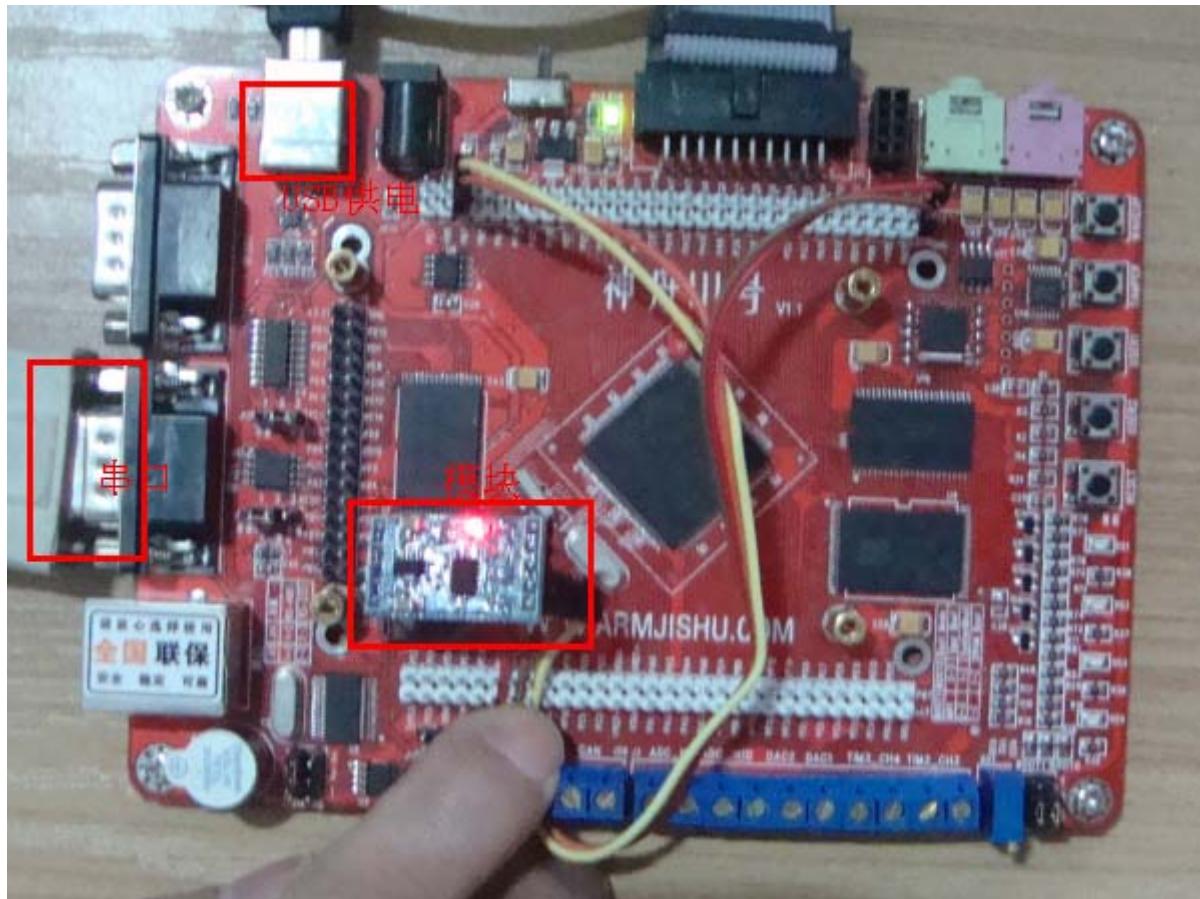
本章将使用 STM32 来驱动 ADXL345，读取 3 个方向的重力加速度值，并转换为角度，显示在串口上。

7.67.8 硬件设计

本实验采用 STM32 的 2 个普通 IO 连接 ADXL345 模块，模块详细资料可参考模块用户手册。本章实验功能简介：主函数不停的查询 ADXL345 的转换结果，得到 x、y 和 z 三个方向的加速度值（读数值），然后将其换为与自然系坐标的角度，并将结果通过串口打印出来。

模块接线图：

我们使用 2 个普通的 I/O 口 PB6 和 PB7 作为我们的 I2C 通信接口，利用板子上的电源与地给模块供电



7.67.9 软件设计

我们来看下 main 主函数部分的代码，首先是初始化部分，在这里我们对串口进行了一个初始化

```
int main(void)
{
    uint8_t t=0;
    short x,y,z;
    short angx,angy,angz;
    /* USARTX configured as follow:
       - BaudRate = 115200 baud
       - Word Length = 8 Bits
       - One Stop Bit
       - No parity
       - Hardware flow control disabled (RTS and CTS signals)
       - Receive and transmit enabled */
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
```

初始化完后对我们的 ADXL345 模块进行一个初始化，通过“ADXL345_Init ()”函数的返回值判断我们的模块 ID 是否正确，初始化成功后返回一个 0，否则返回一个 1，初始化失败后串口打印出我们读到的 ID，并一直循环打印。初始化成功后同样打印我们的读到的 ID，初始化成功的 ID 为“229”十六进制数为“0xE5”。

初始化失败后执行的函数：

```
while (ADXL345_Init ())
{
    printf ("\r\n ID %d\r\n", ADXL345_RD_Reg (DEVICE_ID));
}
```

初始化成功后串口输出我们的模块 ID 并继续往下执行：

```
printf ("\r\n ID %d\r\n", ADXL345_RD_Reg (DEVICE_ID));
while (1)
{
```

成功后打印的 ID 为：

WWW.ARMJISHU.COM

IID 229th
加速度原始值 X=VAL: 171 Y=VAL: -207 Z=VAL: -29

其中红色方框为读到的 ID，蓝色方框为串口初始化时串口输出的数据。可以看到，我们现在读到的 ID 为“229”十六进制数为“0xE5”是该模块正确的 ID。

寄存器定义

寄存器0x00—DEVID (只读)

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

DEVID寄存器保存0xE5的固定器件ID代码(345八进制)。

读完 ID 后读取我们模块上的速度原始值与角度值，并通过串口打印出来：

```
while(1)
{
    if(t%10==0)//读取频率
    {
        //得到x,y,z轴的加速度值(原始值)
        ADXL345_Read_Average(&x,&y,&z,10); //读取x,y,z三个方向的加速度值
        printf("加速度原始值 X=VAL: %d\n",x); //显示加速度原始值
        printf(" Y=VAL: %d\n",y);
        printf(" Z=VAL: %d\n",z);
        printf("\r\n");
        Delay(3000000);
        //得到角度值,并显示
        angx=ADXL345_Get_Angle(x,y,z,1);
        angy=ADXL345_Get_Angle(x,y,z,2);
        angz=ADXL345_Get_Angle(x,y,z,0);
        printf("角度值 X=ANG: %d\n",angx); //显示角度值
        printf(" Y=ANG: %d\n",angy);
        printf(" Z=ANG: %d\n",angz);
        printf("\r\n");
        Delay(3000000);
    }
}
```

加速度原始值 X=VAL: 160

Y=VAL: -216

Z=VAL: -30

角度值

X=ANG: 362

Y=ANG: -530

Z=ANG: -836

[深入分析程序代码](#)

ADXL345 初始化函数

```
//初始化ADXL345。  
//返回值:0,初始化成功;1,初始化失败。  
u8 ADXL345_Init(void)  
{  
    IIC_Init(); //初始化IIC总线  
    if(ADXL345_RD_Reg(DEVICE_ID)==0xE5) //读取器件ID  
    {  
        ADXL345_WR_Reg(DATA_FORMAT,0x2B); //低电平中断输出,13位全分辨率,输出数据右对齐,16g量程  
        ADXL345_WR_Reg(BW_RATE,0x0A); //数据输出速度为100Hz  
        ADXL345_WR_Reg(POWER_CTL,0x28); //链接使能,测量模式  
        ADXL345_WR_Reg(INT_ENABLE,0x00); //不使用中断  
        ADXL345_WR_Reg(OFSX,0x00);  
        ADXL345_WR_Reg(OFSY,0x00);  
        ADXL345_WR_Reg(OFSZ,0x00);  
        return 0;  
    }  
    return 1;  
}
```

该函数是用来初始化 ADXL345 的，采用查询的方式来读取数据的，所以在这里并没有开启中断。另外 3 个偏移寄存器，都默认设置为 0。初始化成功后返回 0 值，识别时返回 1 值，通过读取模块 ID 判断是否正确。

ADXL345_RD_XYZ 函数

```
//读取3个轴的数据  
//x,y,z:读取到的数据  
void ADXL345_RD_XYZ(short *x, short *y, short *z)  
{  
    u8 buf[6];  
    u8 i;  
    IIC_Start(); //重新启动  
    IIC_Send_Byte(ADXL_WRITE); //发送写器件指令  
    IIC_Wait_Ack();  
    IIC_Send_Byte(0x32); //发送寄存器地址(数据缓存的起始地址为0x32)  
    IIC_Wait_Ack();  
  
    IIC_Start(); //重新启动  
    IIC_Send_Byte(ADXL_READ); //发送读器件指令  
    IIC_Wait_Ack();  
    for(i=0;i<6;i++)  
    {  
        if(i==5)buf[i]=IIC_Read_Byte(0); //读取一个字节,不继续再读,发送NACK  
        else buf[i]=IIC_Read_Byte(1); //读取一个字节,继续读,发送ACK  
    }  
    IIC_Stop(); //产生一个停止条件  
    *x=(short)((u16)buf[1]<<8)+buf[0];  
    *y=(short)((u16)buf[3]<<8)+buf[2];  
    *z=(short)((u16)buf[5]<<8)+buf[4];  
}
```

该函数用于从 ADXL345 读取数据，通过该函数可以读取 ADXL345 的转换结果，得到三个轴的加速度值（仅是数值，并没有转换单位）。

ADXL345_AUTO_Adjust 函数

```

//自动校准
//xval,yval,zval:x,y,z轴的校准值
void ADXL345_AUTO_Adjust(char *xval,char *yval,char *zval)
{
    short tx,ty,tz;
    u8 i;
    short offx=0,offy=0,offz=0;
    ADXL345_WR_Reg(POWER_CTL, 0x00);           //先进入休眠模式.
    SysTickDelay(100); //delay_ms(100);
    ADXL345_WR_Reg(DATA_FORMAT, 0x2B);          //低电平中断输出,13位全分辨率,输出数据右对齐,16g量程
    ADXL345_WR_Reg(BW_RATE, 0x0A);              //数据输出速度为100Hz
    ADXL345_WR_Reg(POWER_CTL, 0x28);            //链接使能,测量模式
    ADXL345_WR_Reg(INT_ENABLE, 0x00);           //不使用中断

    ADXL345_WR_Reg(OFSX, 0x00);
    ADXL345_WR_Reg(OFSY, 0x00);
    ADXL345_WR_Reg(OFSZ, 0x00);
    SysTickDelay(12); //delay_ms(12);
    for(i=0;i<10;i++)
    {
        ADXL345_RD_Avval(&tx,&ty,&tz);
        offx+=tx;
        offy+=ty;
        offz+=tz;
    }
    offx/=10;
    offy/=10;
    offz/=10;
    *xval=-offx/4;
    *yval=-offy/4;
    *zval=-(offz-256)/4;
    ADXL345_WR_Reg(OFSX,*xval);
    ADXL345_WR_Reg(OFSY,*yval);
    ADXL345_WR_Reg(OFSZ,*zval);
}

```

该函数用于 ADXL345 的校准，ADXL345 有偏移校准的功能，该功能的详细介绍请参考 ADXL345 数据手册的第 29 页，偏移校准部分。这里我们就不细说了，如果不进行校准的话，ADXL345 的读数可能会有些偏差，通过校准，我们可以将这个偏差减少甚至消除。

ADXL345_Get_Angle 函数

```

//得到角度
//x,y,z:x,y,z方向的重力加速度分量(不需要单位,直接数值即可)
//dir:要获得的角度.0,与z轴的角度;1,与x轴的角度;2,与y轴的角度.
//返回值:角度值.单位0.1°.
short ADXL345_Get_Angle(float x,float y,float z,u8 dir)
{
    float temp;
    float res=0;
    switch(dir)
    {
        case 0://与自然z轴的角度
            temp=sqrt((x*x+y*y))/z;
            res=atan(temp);
            break;
        case 1://与自然x轴的角度
            temp=x/sqrt((y*y+z*z));
            res=atan(temp);
            break;
        case 2://与自然y轴的角度
            temp=y/sqrt((x*x+z*z));
            res=atan(temp);
            break;
    }
    return res*1800/3.14;
}

```

该函数根据 ADXL345 的读值，转换为与自然坐标系的角度。计算公式如下：

加速度传感器 Z 轴与自然坐标系 Z 轴夹角: $\angle 1 = \tan^{-1}(\frac{\sqrt{A_x^2 + A_y^2}}{A_z})$;

加速度传感器 X 轴与自然坐标系 X 轴夹角: $\angle 2 = \tan^{-1}(\frac{A_x}{\sqrt{A_y^2 + A_z^2}})$;

加速度传感器 Y 轴与自然坐标系 Y 轴夹角: $\angle 3 = \tan^{-1}(\frac{A_y}{\sqrt{A_x^2 + A_z^2}})$;

其中 Ax, Ay, Az 分别代表从 ADXL345 读到的 X, Y, Z 方向的加速度值。通过该函数, 我们只需要知道三个方向的加速度值, 就可以将其转换为对应的弧度值, 再通过弧度角度转换, 就可以得到角度值了。

7.67.10 实验现象

将模块连接到开发板上, 接线方式如下表:

| 模块上的引脚 | 开发板上的引脚 | 功能 |
|--------|---------|-----|
| 3.3V | 3.3V | 电源 |
| GND | GND | 地 |
| SCL | PB6 | 时钟线 |
| SDA | PB7 | 数据线 |

程序代码编译下载到我们的神舟 III 号板子上, 连接 USB 转串口线。根据前面介绍过的串口设置方法, 可以看到在板子复位后, 串口把我们模块的数据打印出来了, 这些数据就是我们模块的角度值和速度初始值, 移动模块的方向位置时, 可看到对应的数值相对应的发生变化。串口打印的数据如下:

```

WWW.ARmjishu.COM
IID_229th
加速度原始值 X=VAL: 162   Y=VAL: -217   Z=VAL: -14
加速度原始值 X=VAL: 155   Y=VAL: -220   Z=VAL: -20
角度值     X=ANG: 350    Y=ANG: -546    Z=ANG: -857
加速度原始值 X=VAL: 156   Y=VAL: -217   Z=VAL: -27
角度值     X=ANG: 355    Y=ANG: -539    Z=ANG: -842

```

7.68 串口IAP在线升级实验

7.68.1 意义与作用

生活中，我们往往要对一些已经在使用中的产品程序进行升级以提升性能或消除缺陷，如何对已经投入使用的产品进行方便可靠的程序在线升级，是产品设计初期必须考虑的问题，尽管目前可以通过 ISP 的功能去实现，但是这需要特定的烧写软件支持和专业人员操作。烧写软件由芯片厂商提供，不便于集成到产品的主机端软件中。在产品软件功能中添加简单易用的程序升级功能十分必要。而 IAP 的出现即可完美的实现了这个功能，在这章，我们通过对 IAP 的介绍，深入了解它、使用它，对 IAP 做一个全面的接触。

7.68.2 关于什么是IAP

什么是 IAP？

系统运行的过程中动态编程，这种编程是对程序执行代码的动态修改，而且毋须借助于任何外部力量，也毋须进行任何机械操作。这个就是 IAP(In-Application Programming)。

在应用编程 IAP 是应用在 Flash 程序存储器的一种编程模式。它可以在应用程序正常运行的情况下，通过调用特定的 IAP 程序对另外一段程序 Flash 空间进行读 / 写操作，甚至可以控制对某段、某页甚至某个字节的读 / 写操作，这为数据存储和固件的现场升级带来了更大的灵活性。

比如一款支持 Iap 的单片机，内分 3 个程序区，1 作引导程序区，2 作运行程序区，3 作下载区，芯片通过串口接收到下载命令，进入引导区运行引导程序，在引导程序下将 new code 内容下载到下载区，下载完毕并校验通过后再将下载区内容复制到 2 区，运行复位程序，则 Iap 完成；

例如在程序运行工程中产生 4k 字节数据表，为了避免占用 SRAM 空间，用户可以使用 IAP 技术将此表写入片内 Flash。又如用户在开发完一个系统后要增加新的软件功能，可以使用 IAP 技术在线升级程序，避免重新拆装设备。

Boot 简介

Boot 装载程序控制芯片复位后的初始化操作，并提供对 Flash 编程的方法。Boot 程序可以对芯片进行擦除、编程。

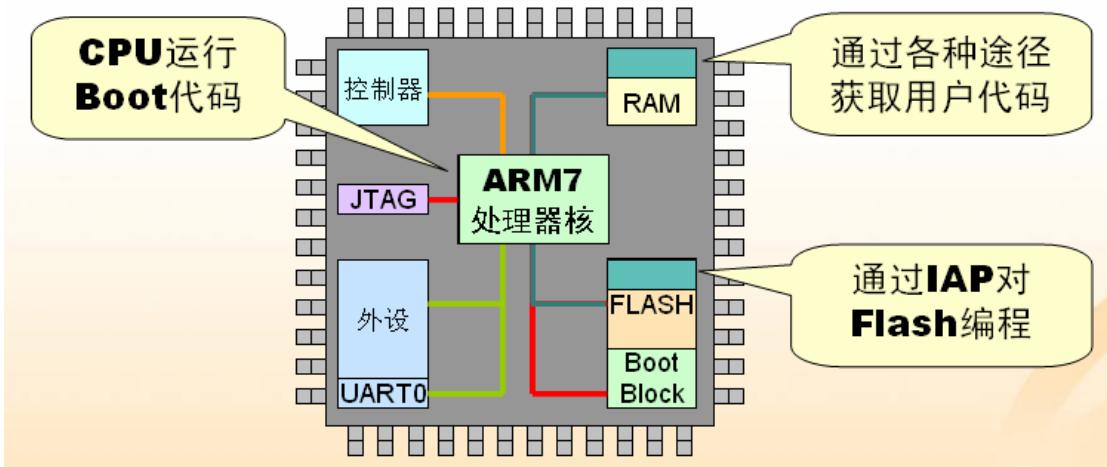
7.68.3 各种烧录方式的比较

传统的烧录方式：对单片机的程序烧录传统上是采用并行烧录器。这种方法要求先把芯片放在烧录器的夹座上进行烧录，然后取下芯片再安装到电路板上。如果要进行反复烧录，则往往要在电路板上为芯片配一个 DIP 或 PLCC 的底座。显然，这要占用比较大的面积，而且插拔也不是特别方便。对于贴片封装，比如 QFP，焊到电路板上之后，如果发现程序有误则应当重新烧录，但是想取下来时却发现并不是件容易的事情，往往只得报废。

ISP 烧录方式：ISP 烧录方式较好地解决了**传统的烧录方式**的问题。它提供了一个串行的烧录接口，不必取下芯片就可以进行烧录，非常方便，尤其是对于贴片封装的芯片。而 ISP 烧录方式也有其不足之处，每次烧录都必须要和计算机（或专用烧录设备）连机才行。有些用户希望产品在实际应用期间能够通过某种远程通信方式自动地更新程序内容，显然 ISP 已经无法满足这样的需求了，必须找到新的办法。

IAP 烧录方式：IAP 烧录方式为程序的自我更新提供了有效手段。单片机内部的 Flash 存储器保存有用户的程序代码，这些代码在正常运行期间是不能被修改的。但是有了 IAP，用户程序就能够根据需要（满足某种条件）自行修改部分甚至全部程序代码。新的程序代码可能是程序在运行过程

中自动生成的，也可能来自于远程设备，而后者更普遍。



应用场景

ISP 程序升级需要到现场解决,不过好一点的是不必拆机器了;

IAP 如果有网管系统的话,用网管下载一切搞定,人不用跑来跑去,

IAP 的触发方式:

IAP 的触发比较简单, 没有外部触发。通过一些指示位(SST 为 SC0/SC1、SFcf[1,0];Philips 为一段 IAP 子程序, 保存在 FF00H~FFFFH 地址空间中), 达到引导到 BootROM 的目的。IAP 所进入的 BootROM 里面驻留的 Boot 代码, 才是最终目标。

IAP 的优点

IAP 技术是从结构上将 Flash 存储器映射为两个存储体, 当运行一个存储体上的用户程序时, 可对另一个存储体重新编程, 之后将程序从一个存储体转向另一个。实现更加灵活, 通常可利用单片机的串行口接到计算机的 RS232 口, 通过专门设计的固件程序来编程内部存储器, 可以通过现有的 INTERNET 或其它通讯方式很方便地实现远程升级和维护。

IAP 的工作原理

在实现 IAP 功能时, 单片机内部一定要有两块存储区, 一般一块被称为 BOOT 区, 另外一块被称为存储区。单片机上电运行在 BOOT 区, 如果有外部改写程序的条件满足, 则对存储区的程序进行改写操作。如果外部改写程序的条件不满足, 程序指针跳到存储区, 开始执行放在存储区的程序, 这样便实现了 IAP 功能。

7.68.4 串口IAP的内部实现流程

IAP操作流程

IAP是用户自己的程序在运行过程中对Flash的部分区域进行烧写, 目的是为了在产品发布后可以方便地通过预留的通信口对产品中的固件程序进行更新升级。通常实现IAP功能时, 即用户程序运行中作自身的更新操作, 需要在设计固件程序时编写两个项目代码, 第一个项目程序不执行正常的功能操作, 而只是通过某种通信方式(如USB、USART)接收程序或数据, 执行对第二部分代码的更新; 第二个项目代码才是真正的功能代码。这两部分项目代码都同时烧录在Flash中, 当芯片上电后, 首先是第一个项目代码开始运行, 它作如下操作:

- 1) 检查是否需要对第二部分代码进行更新

2) 如果不需要更新则转到4)

3) 执行更新操作

4) 跳转到第二部分代码执行

第一部分代码必须通过其它手段，如JTAG或ISP烧入；第二部分代码可以使用第一部分代码IAP功能烧入，也可以和第一部分代码一起烧入，以后需要程序更新是再通过第一部分IAP代码更新。

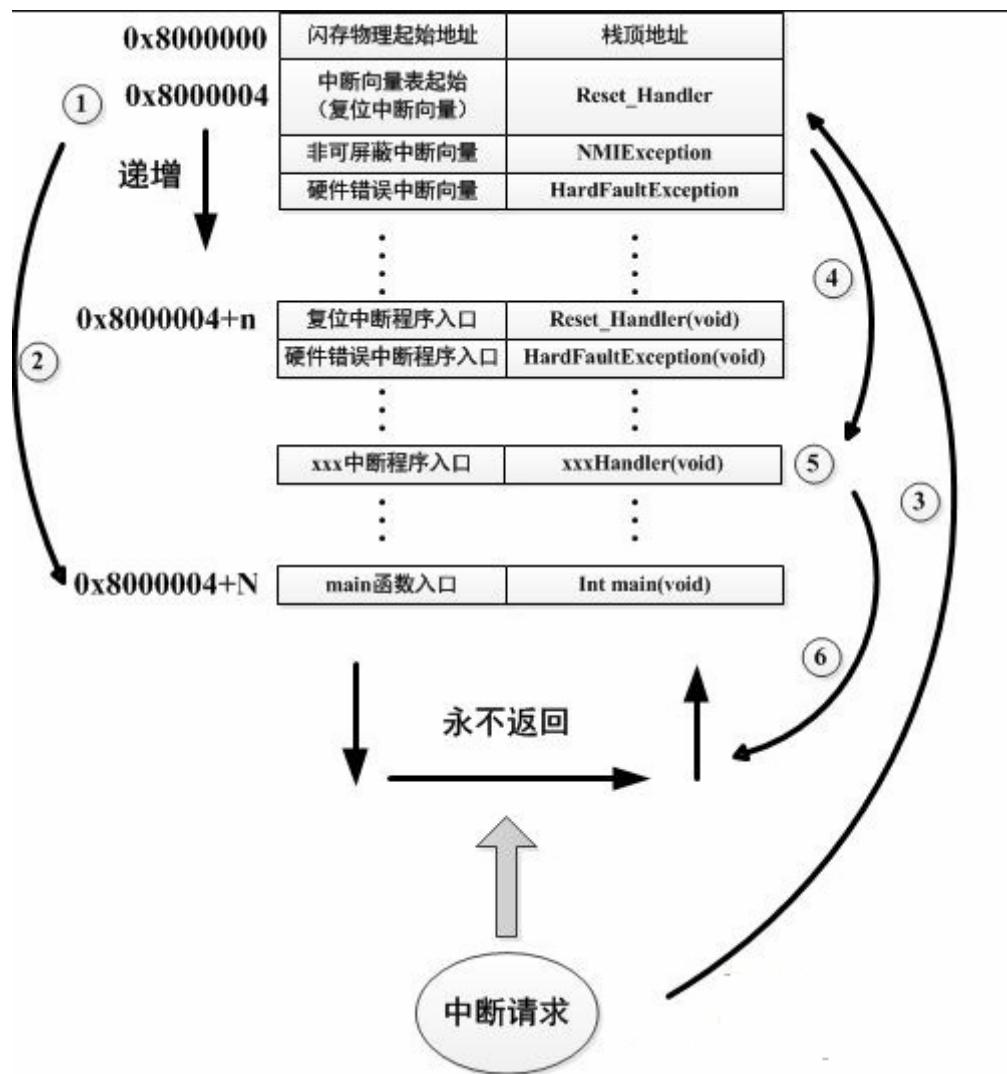
对于STM32来说，因为它的中断向量表位于程序存储器的最低地址区，为了使第一部分代码能够正确地响应中断，通常会安排第一部分代码处于Flash的开始区域，而第二部分代码紧随其后。

我们将第一个项目代码称之为Bootloader程序，第二个项目代码称之为APP程序，他们存放在STM32 FLASH的不同地址范围，一般从最低地址区开始存放Bootloader，紧跟其后的就是APP程序（注意，如果FLASH容量足够，是可以设计很多APP程序的，本章我们只讨论一个APP程序的情况）。这样我们就是要实现2个程序：Bootloader和APP。STM32的APP程序不仅可以放到FLASH里面运行，也可以放到SRAM里面运行

如果IAP程序被破坏，产品必须返厂才能重新烧写程序，这是很麻烦并且非常耗费时间和金钱的。针对这样的需求，STM32在对Flash区域实行读保护的同时，自动地对用户Flash区的开始4页设置为写保护，这样可以有效地保证IAP程序(第一部分代码)区域不会被意外地破坏。

STM32程序运行流程

我们先来看看 STM32 正常的程序运行流程，如图

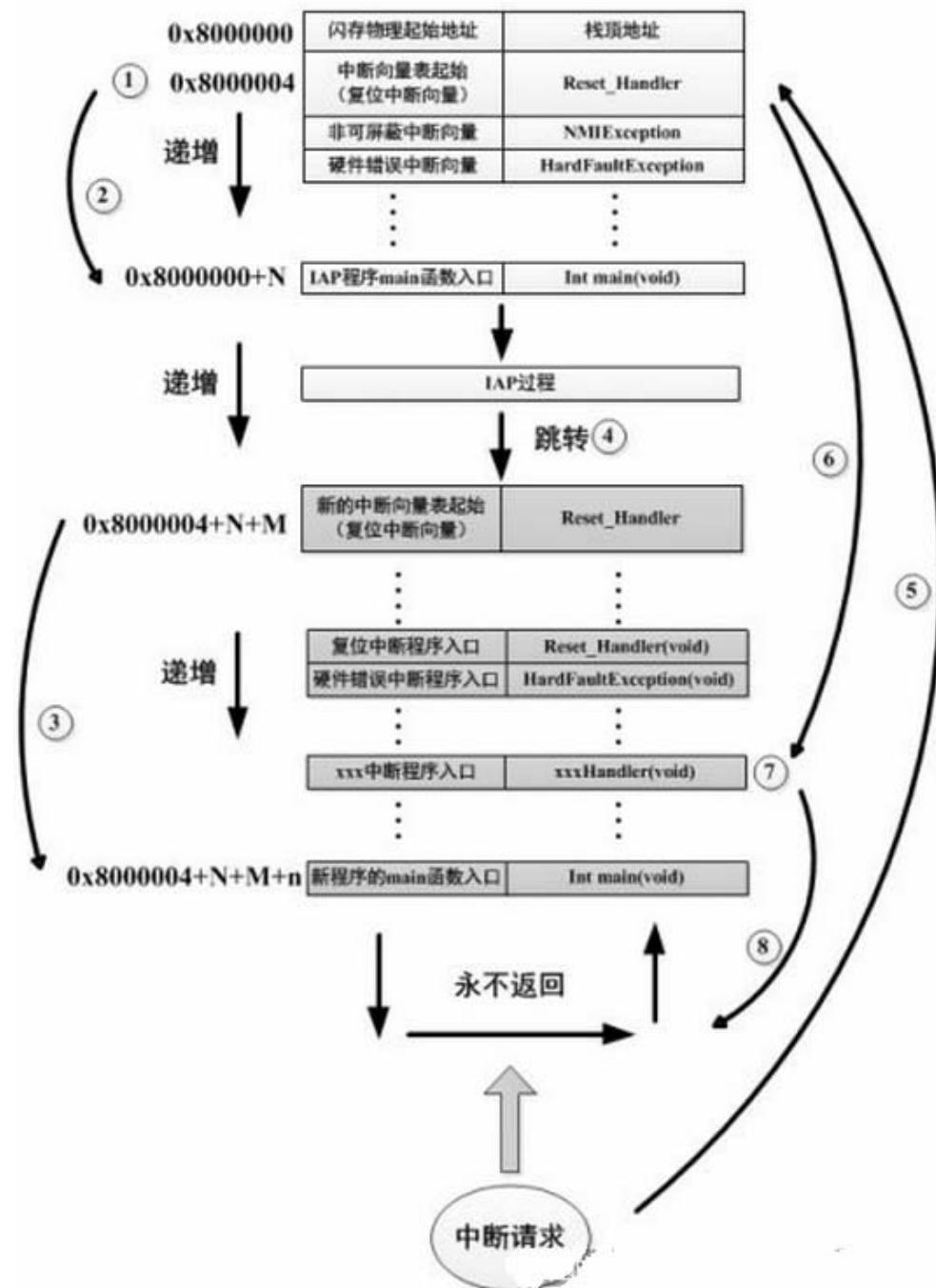


STM32 的内部闪存 (FLASH) 地址起始于 0x08000000，一般情况下，程序文件就从此地址开始写入。其内部通过一张“中断向量表”来响应中断，程序启动后，将首先从“中断向量表”取出复位中断向量执行复位中断程序完成启动，STM32 复位后，会从地址为 0x80000004 处取出复位中断向量的地址，并跳转执行复位中断服务程序，根据中断源取出对应的中断向量执行中断服务程序。

在图中，STM32 在复位后，先从 0X08000004 地址取出复位中断向量的地址，并跳转到复位中断服务程序，如图标号①所示；在复位中断服务程序执行完之后，会跳转到我们的 main 函数，如图标号②所示；而我们的 main 函数一般都是一个死循环，在 main 函数执行过程中，如果收到中断请求（发生重中断），此时 STM32 强制将 PC 指针指回中断向量表处，如图标号③所示；然后，根据中断源进入相应的中断服务程序，如图标号④所示到达⑤；在执行完中断服务程序以后，程序再次返回 main 函数执行，如图标号⑥所示。

STM32 添加 IAP 后流程

若在 STM32 中加入了 IAP 程序，则情况会如下图所示。



在图所示流程中，STM32复位后，还是从0X08000004地址取出复位中断向量的地址，并跳转到复位中断服务程序，在运行完复位中断服务程序之后跳转到IAP的main函数，如图标号①所示，此部分同前面未加IAP程序前是一样的；

在执行完IAP以后（即将新的APP代码写入STM32的FLASH，灰底部分。地址始于0x8000004+N+M）跳转至新写入程序的复位向量表，取出新程序的复位中断向量的地址，并跳转执行新程序的复位中断服务程序，随后跳转至新程序的main函数，其过程如图2的图标号③所示。新程序的main函数应该也具有永不返回的特性。同时应该注意在STM32的内部存储空间在不同的位置上出现了2个中断向量表。

在main函数执行过程中，如果CPU得到一个中断请求，PC指针仍强制跳转到地址0X08000004中断向量表处，而不是新程序的中断向量表，如图标号⑤所示；

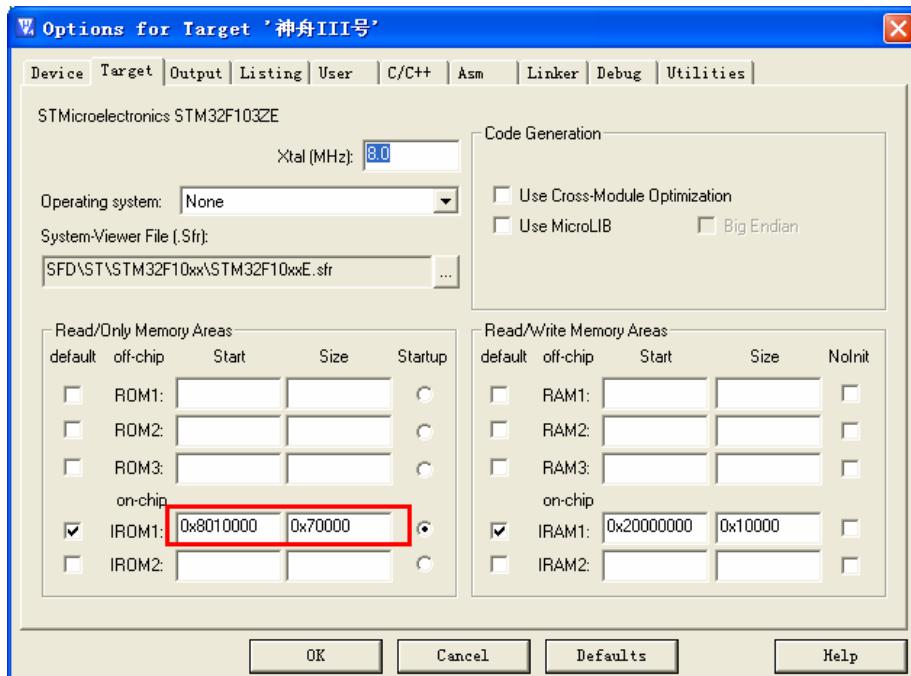
程序再根据我们设置的中断向量表偏移量，跳转到对应中断源新的中断服务程序中，如图标号⑥所示；在执行完中断服务程序后，程序返回main函数继续运行，如图标号⑧所示。

通过以上两个过程的分析，我们知道IAP程序必须满足两个要求：

- 1) 新程序必须在IAP程序之后的某个偏移量为x的地址开始；
- 2) 必须将新程序的中断向量表相应的移动，移动的偏移量为x；

APP 程序起始地址设置方法

设置程序起始位置的方法是（keil uvision4 集成开发环境）在工程的“Option for Target...”界面中的“Target”页里将“IROM”的“Start”列改为欲使程序起始的地方，如下图中我们将程序起始位置设为0x8010000。



默认的条件下，图中 IROM1 的起始地址 (Start) 一般为 0X08000000，大小 (Size) 为 0X80000，即从 0X08000000 开始的 512K 空间为我们的程序存储（因为我们的 STM32F103ZET6 的 FLASH 大小是 512K）。

图中，我们设置起始地址 (Start) 为 0X08010000，即偏移量为 0X10000 (64K 字节)，因而，留给 APP 用的 FLASH 空间 (Size) 只有 0X80000-0X10000=512K-64K=448K 字节大小了。设置好 Start 和 Size，就完成 APP 程序的起始地址设置。

这里的 64K 字节，需要大家根据 Bootloader 程序大小进行选择，确保 APP 起始地址在 Bootloader 之后，并且偏移量为 0X200 的倍数即可，）。这里我们选择 64K (0X10000) 字节，留了一些余量，方便 Bootloader 以后的升级修改。

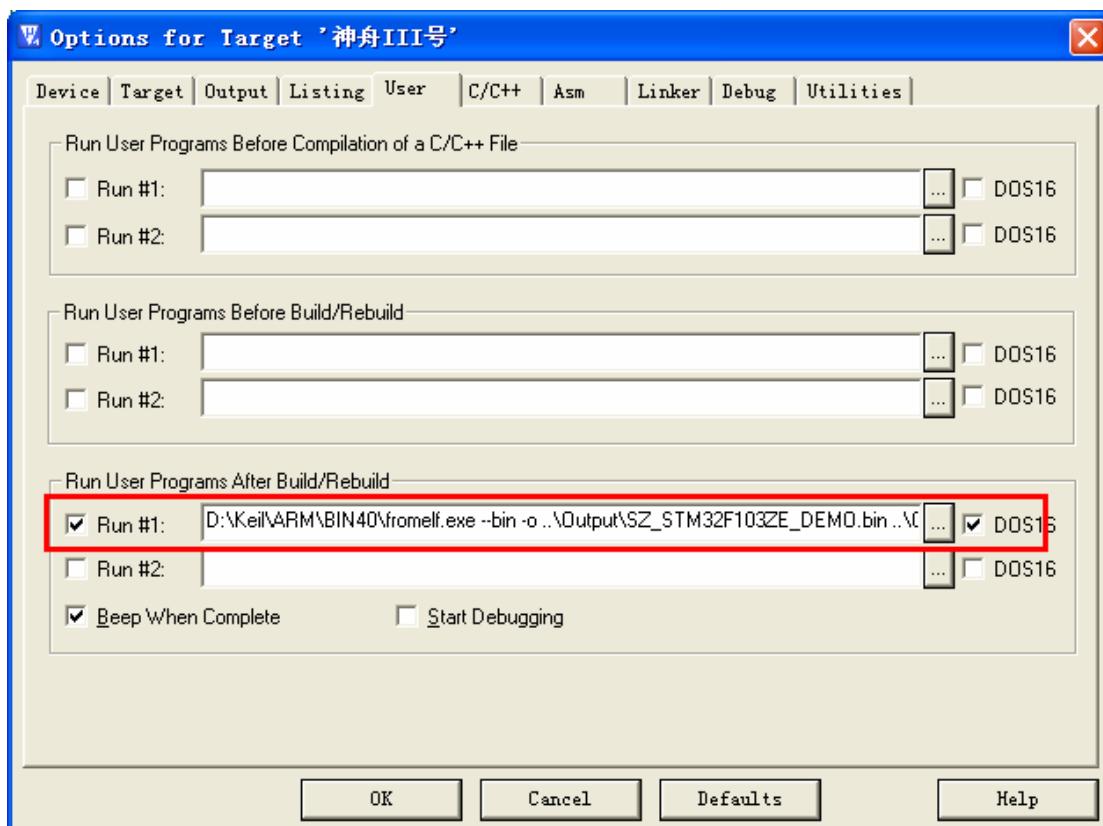
中断向量表的偏移量设置方法

生成 APP 程序

通过以上两个步骤的设置，我们就可以生成APP程序了，只要APP程序的FLASH大小不超过我们的设置即可。不过MDK默认生成的文件是.hex文件，并不方便我们用作IAP更新，我们希望生成的文件是.bin文件，这样可以方便进行IAP升级(至于为什么，请大家自行百度HEX和BIN文件的区别!)。这里我们通过MDK自带的格式转换工具fromelf.exe，来实现.axf文件到.bin文件的转换。该工具在MDK的安装目录\ARM\BIN40文件夹里面。

fromelf.exe转换工具的语法格式为：fromelf [options] input_file。其中options有很多选项可以设置，详细使用请参考《mdk如何生成bin文件.pdf》。

在MDK点击Options for Target→User选项卡，在Run User Programs After Build/Rebuild 栏，勾选Run#1 和 DOS16，并写入：D:\Keil\ARM\BIN40\fromelf.exe --bin -o ..\Output\SZ_STM32F103ZE_DEMO.bin ..\Output\SZ_STM32F103ZE_DEMO.axf，如下图所示：



设置完后，我们就可以在MDK编译成功之后，调用fromelf.exe（注意，我的MDK是安装在C:\Keil文件夹下，如果你是安装在其他目录，请根据你自己的目录修改fromelf.exe的路径），根据当前工程生成一个SZ_STM32F103ZE_DEMO.bin的文件。并存放在axf文件相同的目录下，即工程的Output文件夹里面。在得到.bin文件之后，我们只需要将这个bin文件传送给单片机，即可执行IAP升级。

最后再来APP程序的生成步骤：

1) 设置APP程序的起始地址和存储空间大小

对于在FLASH里面运行的APP程序，我们可以前面介绍的设置。

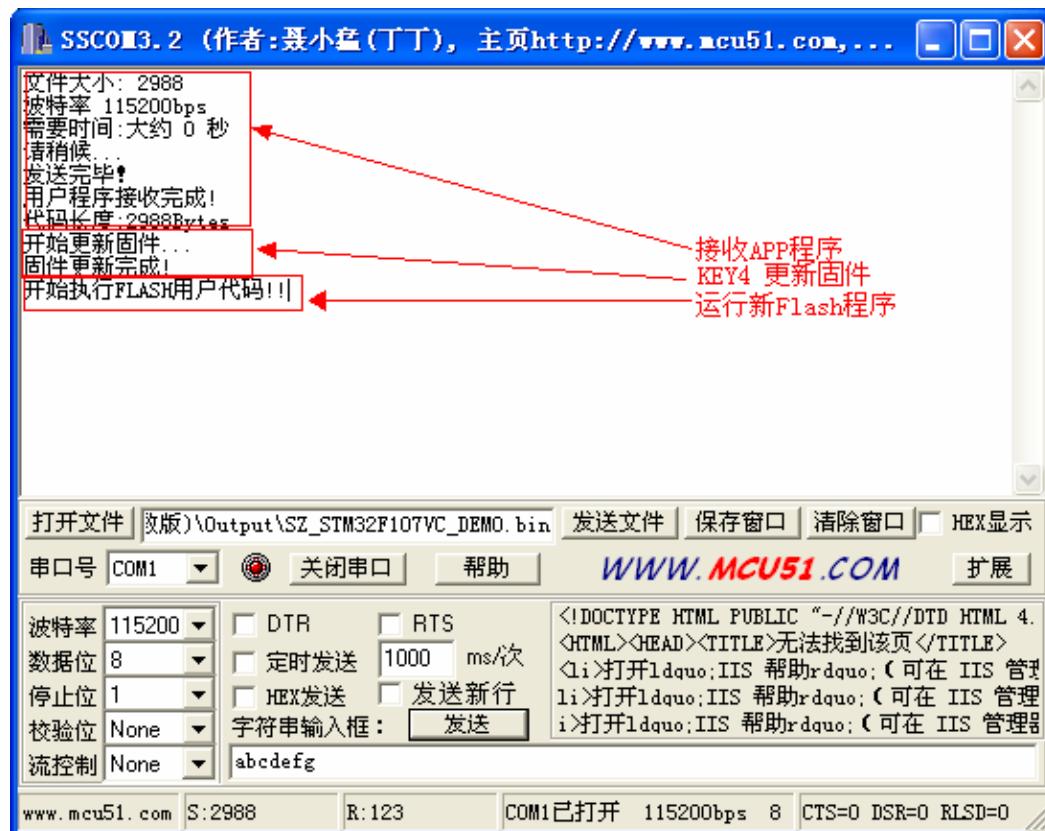
- 2) 设置中断向量表偏移量
- 3) 设置编译后运行fromelf.exe，生成.bin文件。

通过在User选项卡，设置编译后调用fromelf.exe，根据.axf文件生成.bin文件，用于IAP更新。

以上3个步骤，我们就可以得到一个.bin的APP程序，通过Bootlader程序即可实现更新。

7.68.5 实验现象

实验中，开机的时候串口先打印提示信息，等待串口输入接收APP程序（无校验，一次性接收），在串口接收到APP程序之后，即可执行IAP。按下WAMPER，将串口接收到的APP程序存放到STM32的FLASH，之后再按TAMPER既可以执行这个FLASH APP程序。通过USER2按键，可以手动清除串口接收到的APP程序。



7.68.6 代码分析

程序开始时，先是定义按键，为下面判断按键值做准备

```
#define KEY_UP      4
#define KEY_LEFT    3
#define KEY_DOWN    2
#define KEY_RIGHT   1
```

APP 代码长度标识符的定义

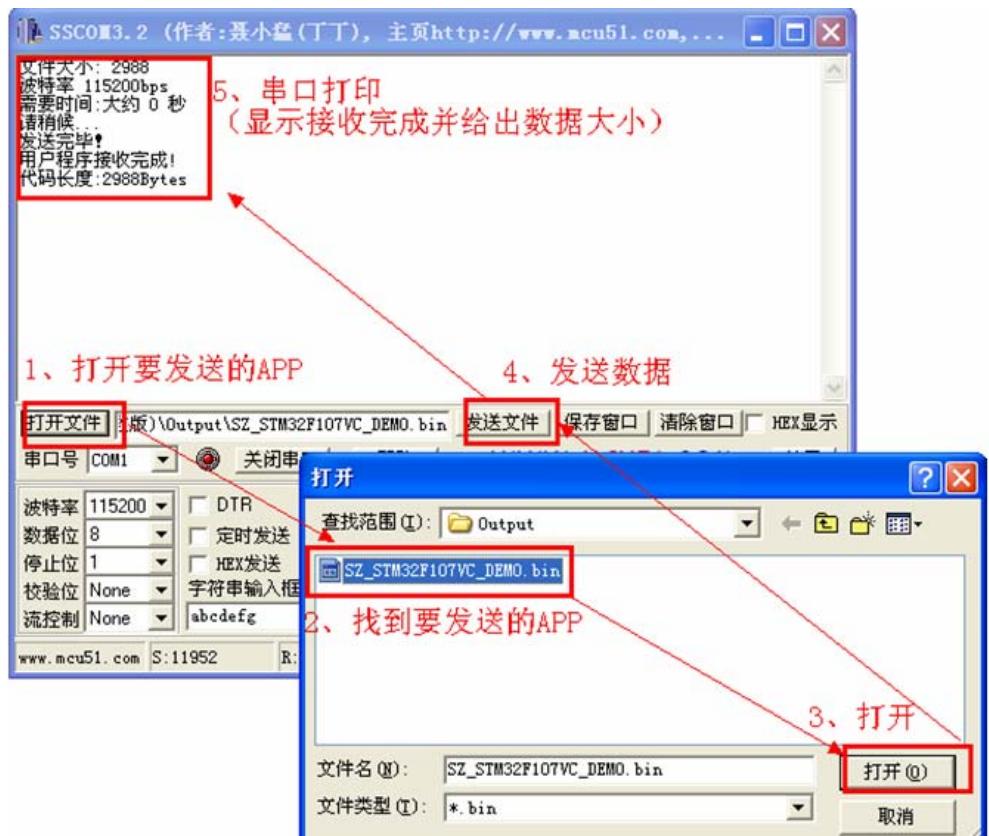
```
int main(void)
{
    u8 key;
    u16 oldcount=0; // 老的串口接收数据值
    u16 applenth=0; // 接收到的app代码长度
```

串口与按键的初始化，串口初始化完后打印“Bootlader!”现在运行的是 Bootlader!程序。

```
/* 串口1初始化 */
SZ_STM32_COMInit(115200);
/* 初始化板载按键为GPIO模式(非中断) */
SZ_STM32_KEYInit(KEY1, BUTTON_MODE_GPIO);
SZ_STM32_KEYInit(KEY2, BUTTON_MODE_GPIO);
SZ_STM32_KEYInit(KEY3, BUTTON_MODE_GPIO);
printf("Bootlader!\r\n");
delay(5000000);
```

进入死循环，开始等待串口中断接收到的数据，接收完后串口打印提示符与代码的大小，无数据接收后，还是等于原来的数据

```
while(1)
{
    if(USART_RX_CNT)
    {
        if(oldcount==USART_RX_CNT)//新周期内,没有收到任何数据,认为本次数据接收完成.
        {
            applenth=USART_RX_CNT;
            oldcount=0;
            USART_RX_CNT=0;
            printf("\r\n应用程序接收完成!\r\n");
            printf("代码长度:%dBytes\r\n", applenth);
        }else oldcount=USART_RX_CNT;
    }
    delay(4000000);
```



KEY 等于按下的按键，按下每个按键都返回一个值，具体可到 SZ_STM32_KEYScan(); 函数查看

```
key=SZ_STM32_KEYScan();
```

如果按下的是“KEY_UP”按键时，通过判断地址是否正确，更新 Flash 的固件，地址错误的话，打印错误提示信息。

```

if(key==KEY_UP)
{
    if(applenth)
    {
        printf("开始更新固件...\r\n");
        if(((vu32*)(0x20001000+4))&0xFF000000)==0x08000000)//判断是否为0x08XXXXXX.
        {
            iap_write_appbin(FLASH_APP1_ADDR, USART_RX_BUF, applenth);//更新FLASH代码
            delay(100);
            printf("固件更新完成!\r\n");
        }
        else
        {
            printf("非FLASH应用程序!\r\n");
        }
    }
    else
    {
        printf("没有可以更新的固件!\r\n");
    }
}

```

文件大小: 2988
波特率 115200bps
需要时间: 大约 0 秒
请稍候...
发送完毕!
用户程序接收完成!
代码长度: 2988Bytes
开始更新固件...
固件更新完成!

没有可以更新的固件!

如果按下的按键为“KEY_DOWN”时，清楚 Flash 已有的固件，如果没有的话，提示“没有可以清除的固件”

```
if (key==KEY_DOWN)
{
    if (applenth)
    {
        printf("固件清除完成!\r\n");
        applenth=0;
    }else
    {
        printf("没有可以清除的固件!\r\n");
    }
    delay(4000000);
}
```

文件大小: 2988
波特率 115200bps
需要时间: 大约 0 秒
请稍候...
发送完毕!
用户程序接收完成!
代码长度: 2988Bytes
开始更新固件...
固件更新完成!
固件清除完成!

固件清除完成!
BootLader!
没有可以清除的固件!

如果按下的按键为“KEY_LEFT”时，运行 Flash 里面的 APP 程序代码，如果没有程序代码运行
嵌入式专业技术论坛 (www.armjishu.com) 出品
第 796 页，共 900 页

行或者是不是 Flash 程序的话，打印“非 FLASH 应用程序,无法执行”提示信息

```
if(key==KEY_LEFT)
{
    printf("开始执行FLASH用户代码!!\r\n");
    if(((*(vu32*)(FLASH_APP1_ADDR+4))&0xFF000000)==0x08000000)//判断是否为0x08XXXXXX
    {
        iap_load_app(FLASH_APP1_ADDR); //执行FLASH APP代码
    }else
    {
        printf("非FLASH应用程序,无法执行!\r\n");
    }
    delay(4000000);
```

```
文件大小: 2988
波特率 115200bps
需要时间: 大约 0 秒
请稍候...
发送完毕!
用户程序接收完成!
代码长度: 2988Bytes
开始更新固件...
固件更新完成!
开始执行FLASH用户代码!!
```

深入分析程序代码

串口的初始化，我们使用的是串口 1，根据原理图可得出，我们串口 1 与处理器相连的是 PA9 与 PA10，要使用我们的串口 1，就要对这两个管脚进行配置，配置如下

```
void SZ_STM32_COMInit(uint32_t BaudRate)
{
    //GPIO端口设置
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1|RCC_APB2Periph_GPIOA, ENABLE); //使能USART1
    //USART1_TX PA.9
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //PA.9
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    //USART1_RX PA.10
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;//浮空输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    //USART 初始化设置

    USART_InitStructure.USART_BaudRate = BaudRate;//一般设置为115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;//字长为8位数据格式
    USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No;//无奇偶校验位
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无硬件
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式

    USART_Init(USART1, &USART_InitStructure); //初始化串口
```

串口中断函数，配置串口中断的模式，如果使能了串口为接收模式的话，进入中断配置准备进入中断

```
#if EN_USART1_RX      //如果使能了接收

NVIC_InitStructure.NVIC IRQChannel = USART1 IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority=3 ;//抢占优先级3
NVIC_InitStructure.NVIC IRQChannelSubPriority = 3;      //子优先级3
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;          //IRQ通道使能
NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化VIC寄存器

USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //开启中断
#endif
USART_Cmd(USART1, ENABLE);                      //使能串口
```

iap_write_appbin 函数与 iap_load_app 函数用于将存放在串口接收 buf 里面的 APP 程序写入到 FLASH，其参数 appxaddr 为 APP 程序的起始地址，程序先判断栈顶地址是否合法，在得到合法的栈顶地址后，通过设置栈顶地址，最后通过一个虚拟的函数跳转到 APP 程序执行代码，实现 IAP→APP 的跳转。

```
//appxaddr: 应用程序的起始地址
//appbuf: 应用程序CODE.
//appsize: 应用程序大小(字节).
void iap_write_appbin(uint32_t appxaddr, uint8_t *appbuf, uint32_t appsize)
{
    uint16_t t;
    uint16_t i=0;
    uint16_t temp;
    uint32_t fwaddr=appxaddr;//当前写入的地址
    uint8_t *dfu=appbuf;
    for (t=0;t<appsize;t+=2)
    {
        temp=(uint16_t)dfu[1]<<8;
        temp+=(uint16_t)dfu[0];
        dfu+=2;//偏移2个字节
        iapbuf[i++]=temp;
        if (i==1024)
        {
            i=0;
            STMFLASH_Write(fwaddr, iapbuf, 1024);
            fwaddr+=2048;//偏移2048 16=2*8. 所以要乘以2.
        }
    }
    if(i) STMFLASH_Write(fwaddr, iapbuf, i); //将最后的一些内容字节写进去.
}

//跳转到应用程序段
//appxaddr: 用户代码起始地址.
void iap_load_app(uint32_t appxaddr)
{
    if (((*(vu32*)appxaddr)&0x2FFE0000)==0x20000000) //检查栈顶地址是否合法.
    {
        jump2app=(iapfun)*(vu32*)(appxaddr+4);      //用户代码区第二个字为程序开始地址(复位地
        MSR_MSP(*(vu32*)appxaddr);                  //初始化APP堆栈指针(用户代码区的第一个字,
        jump2app();                                //跳转到APP.
    }
}
```

Flash 程序起始地址与接收字节参数配置

```
#ifndef __IAP_H__
#define __IAP_H__
#include "stm32f10x.h"

typedef void (*iapfun)(void); // 定义一个函数类型的参数.

#define FLASH_APP1_ADDR 0x08010000 // 第一个应用程序起始地址(存放在FLASH)
#define USART_REC_LEN 55*1024 // 定义最大接收字节数 55K
#define EN_USART1_RX 1 // 使能(1)/禁止(0)串口1接收
// 保留0X08000000~0X0800FFFF的空间为IAP使用

extern uint8_t USART_RX_BUF[USART_REC_LEN]; // 接收缓冲,最大USART_REC_LEN个字节.末字节为换行.
void iap_load_app(u32 appxaddr); // 执行flash里面的app程序
void iap_write_appbin(u32 appxaddr, u8 *appbuf, u32 applen); // 在指定地址开始,写入bin
#endif
```

我们一般是要求 bootloader 程序越小越好的，可以为 APP 节省空间，不能大于 0x10000，而我们现在编译出来的大概 6K 左右，完全满足了要求。

```
Build Output
Build target '神舟III号'
linking...
Program Size: Code=5756 RO-data=416 RW-data=92 ZI-data=62372
FromELF: creating hex file...
"..\Output\SZ_STM32F103ZE_DEMO.axf" - 0 Error(s), 0 Warning(s).
```

7.69 MP3(VS1003)音频模块实验

7.69.1 MP3介绍

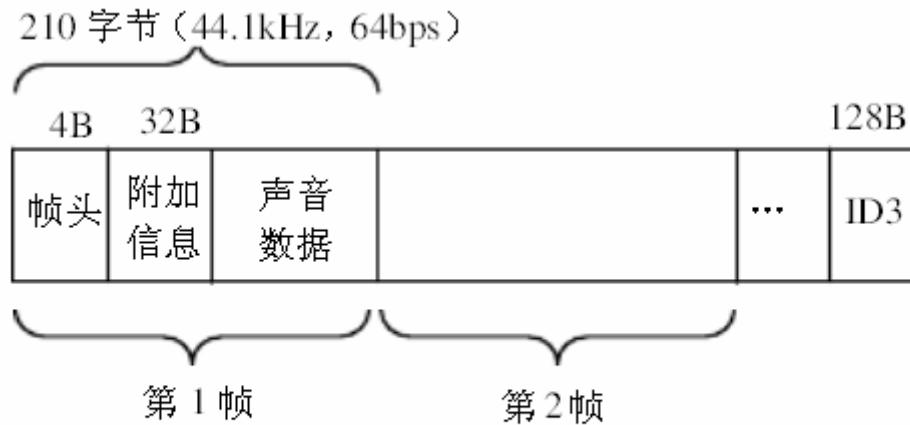
MP3 简介

动态影像专家压缩标准音频层面 3 (Moving Picture Experts Group Audio Layer III)，简称为 MP3。根据压缩质量和编码复杂程度划分为三层，即 Layer-1、Layer2、Layer3，且分别对应 MP1、MP2、MP3 这三种声音文件。MP3 是一种音频压缩技术，它被设计用来大幅度地降低音频数据量。MP3 文件是由帧(frame)构成的，帧是 MP3 文件最小的组成单位。利用 MPEG Audio Layer 3 的技术，将音乐以 1:10 甚至 1:12 的压缩率，也就是说，一分钟 CD 音质的音乐，未经压缩需要 10MB 的存储空间，而经过 MP3 压缩编码后只有 1MB 左右。压缩成容量较小的文件，而对于大多数用户来说重放的音质与最初的不压缩音频相比没有明显的下降。

MP3 文件结构

MP3 文件数据由多个帧组成，帧是 MP3 文件最小组成单位。每个帧又由帧头、附加信息和声音数据组成。关于各部分的介绍我们这里就不说了，有兴趣的可以通过网络或者其他途径进行查阅。每个帧播放时间是 0.026 秒，其长度随位率的不同而不等。有些 MP3 文件末尾有些额外字节存放非

声音数据的说明信息。MP3 文件结构如下图所示：



MP3 文件结构

MP3 工作原理

那么 MP3 的工作原理是怎么样的呢？下面我们就简单的给大家介绍一下：

首先将MP3歌曲文件从贮体上（闪存介质、硬盘介质、光碟介质）取出并读取存储器上的信号——→到解码芯片对信号进行解码——→通过数模转换器将解出来的数字信号转换成模拟信号——→再把转换后的模拟音频放大——→低通滤波后到耳机输出口，输出后就是我们所听到的音乐了。

我们的这次实验中用到的是 VS1003 的解码芯片，VS1003 通过 SPI 接口来接受输入的音频数据流，它可以是一个系统的从机，也可以作为独立的主机。这里我们只把它当成从机使用。我们通过 SPI 口向 VS1003 不停的输入音频数据，它就会自动帮我解码了，然后从输出通道输出音乐，这时我们接上耳机就能听到所播放的歌曲了

7.69.2 VS1003解码芯片介绍

VS1003 简介

前面我们介绍 MP3 的工作原理的时候，提到的一个解码芯片，那么这个 MP3 的解码芯片是一个东西呢？其实解码芯片就是能将存储在介质（Flash 或者硬盘）上的 MP3 文件进行解码，很大程度上影响产品最终的音质表现。MP3 是一种有损压缩的格式，如果 MP3 随身听拥有优秀的解码芯片就能够更好地还原音频信号的质量，很大程度上弥补音频信号的损失。对于音质的还原，芯片越好，音质就会越好。在本次实验中，我们使用到的是一款 VS1003 的音频解码芯片，通过该芯片实现我们的音频解码过程。

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_DSP4, 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。

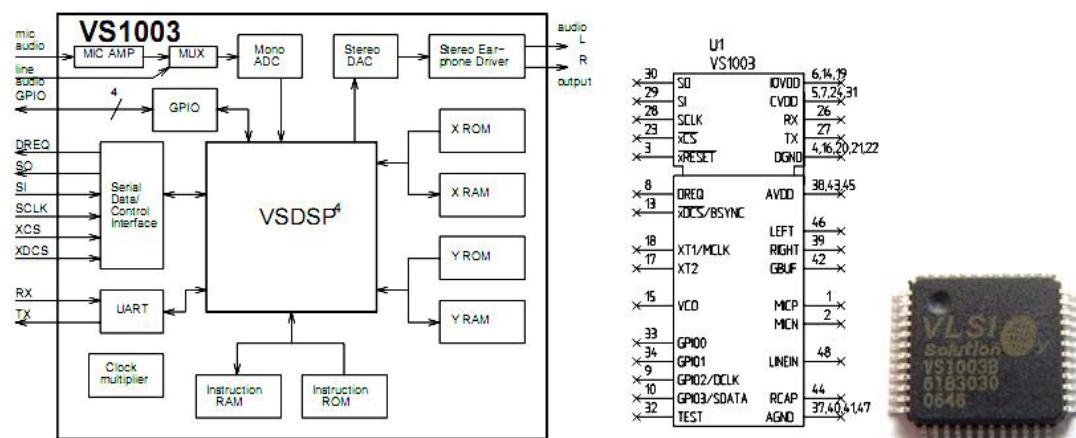
VS1003 特性

VS1003 的特性如下：

- 能解码 MPEG 1 和MPEG2 音频层 III (CBR+VBR+ABR)；WMA 4.0/4.1/7/8/9 5-384kbps 所有流文件；WAV(PCM+IMAAD-PCM);产生MIDI/SP-MIDI 文件。
- 对话筒输入或线路输入的音频信号进行IMAADPCM编码
- 支持 MP3 和WAV 流
- 高低音控制
- 单时钟操作12..13MHz
- 内部PLL锁相环时钟倍频器
- 低功耗
- 内含高性能片上立体声数模转换器，两声道间无相位差
- 内含能驱动30 欧负载的耳机驱动器
- 模拟，数字，I/O 单独供电
- 为用户代码和数据准备的5.5KB片上RAM
- 串行的控制，数据接口
- 可被用作微处理器的从机
- 特殊应用的SPI Flash引导
- 供调试用途的UART接口
- 新功能可以通过软件和 4 GPIO 添加

VS1003 内部结构与实物图

VS1003 内部结构图和实物图如下所示，它是一个 LQFP-48 封装，也有是 BGA 封装的



其中芯片的管脚分配可以通过 VS1003 的芯片手册查阅

VS1003 的通信方式

VS1003 通过7 根线同主控芯片连接，他们是：VS_RST、VS_DREQ、VS_MISO、VS_MOSI、VS_SCK、VS_XDCS、VS_XCS。（可查阅管脚分配图与介绍）

其中VS_RST 是VS1003 的复位信号线，低电平有效。VS_DREQ 是一个数据 请求信号，用来通知主机，VS1003 可以接收数据与否。VS_MISO、VS_MOSI 和VS_SCK 则是VS1003 的SPI接口他们在VS_XCS 和VS_XDCS 下面来执行不同的操作。

SPI总线，最初被用在一些Motorola 器件上-也被应用于VS1003的串行数据接口SDI和串行控制接口SCI。VS1003 的SPI 支持两种模式：1，VS1002 有效模式（即新模式）。2，VS1001 兼容模式。这里我们仅介绍VS1002 有效模式（此模式也是VS1003 的默认模式）。下表是在新模式下VS1003 的SPI 信号线功能描述：

| SDI 管脚 | SCI 管脚 | 描述 |
|--------|--------|---|
| XDCS | XCS | 低电平有效片选输入，高电平强制使串行接口进入 standby 模式，结束当前操作。高电平也强制使串行输出 SO 变成高阻态。如果 SM_SDISHARE 为 1，不使用 XDCS，但是此信号在 XCS 中产生。 |
| SCK | | 串行时钟输入。串行时钟也使用内部的寄存器接口主时钟。SCK 可以被门控或是连续的。对任一情况，在 XCS 变为低电平后，SCK 上的第一个上升沿标志着第一位数据被写入。 |
| SI | | 串行输入，如果片选有效，SI 就在 SCK 的上升沿处采样。 |
| - | SO | 串行输出，在读操作时，数据在 SCK 的下降沿处从此脚移出，在写操作时为高阻态。 |

前面我们介绍到 VS1003 的串行数据接口 SDI 和串行控制接口 SCI，下面我们简单分析下。

SDI：该串行接口作为从机模式操作，所以DCLK 信号必须由外部电路产生。数据（SDATA信号）被DCLK 的上沿或下沿时钟化。

假设 VS1003 输入的字节数据是同步的。SDI 传送可由 SCI_MODE 的内容决定是高位在前或低位在前。

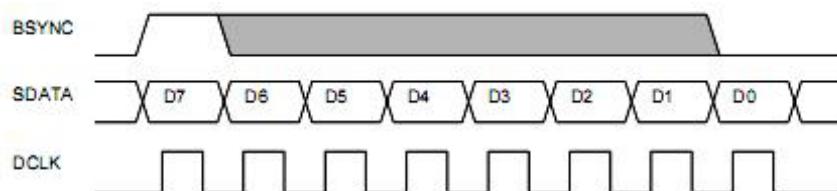


Figure 4: BSYNC Signal - one byte transfer.

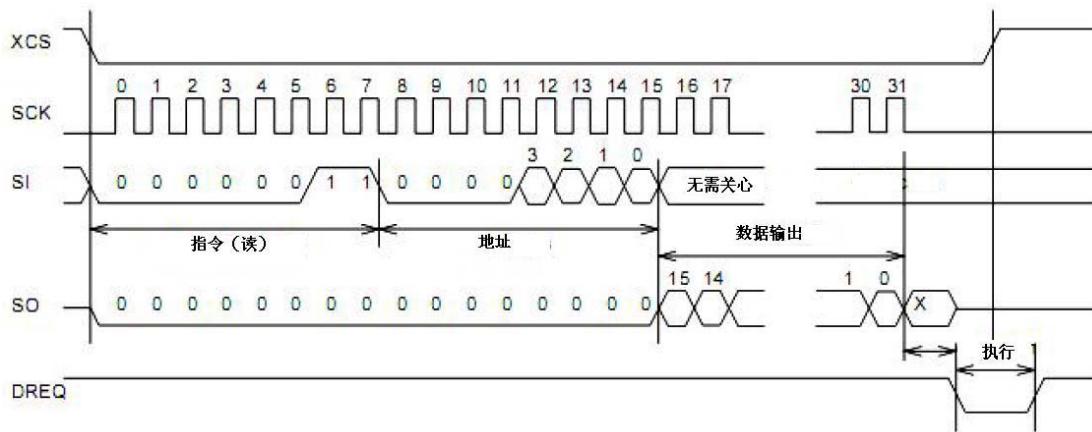
VS1001 兼容模式下 SDI 传送

SCI：SCI 串行总线命令接口包含了一个指令字节、一个地址字节和一个16 位的数据字。读写操作可以读写单个寄存器，在SCK 的上升沿读出数据位，所以主机必须在下降沿刷新数据。SCI 的字节数据总是高位在前低位在后的。第一个字节指令字节，只有2 个指令，也就是读和写，读为0X03，写为0X02。

| Instruction | | |
|-------------|-------------|------------|
| Name | Opcode | Operation |
| READ | 0b0000 0011 | Read data |
| WRITE | 0b0000 0010 | Write data |

注意：在每次SCI 操作后，DREQ 线被置0。VS1003 靠此期间操作。不允许在DREQ 变为1 之前开始新的SCI/SDI操作。

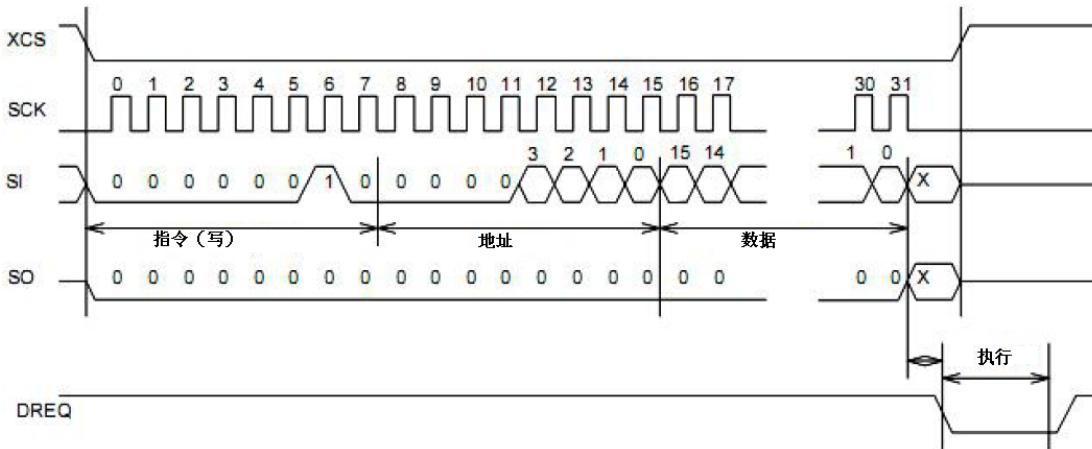
SCI 读时序如下图所示：



SCI 读时序

从图中可以看出，向VS1003 读取数据，通过先拉低XCS (VS_XCS) ，然后发送读指令 (0X03) ，再发送一个地址，最后，我们在SO 线 (VS_MISO) 上就可以读到输出的数据了。而同时SI (VS_MOSI) 上的数据将被忽略。

下面我们再来看看SCI 的写时序：



SCI 写时序

读和写基本类似，都是先发指令，再发地址。不过写时序中，我们的指令是写指令 (0X02) ，并且数据是通过SI 写入VS1003 的， SO 则一直维持低电平。细心的读者可能发现了，在图3 和图4 嵌入式专业技术论坛 (www.armjishu.com) 出品

中，DREQ 信号上都产生了一个短暂的低脉冲，也就是执行时间。这个不难理解，我们在写入和读出 VS1003 的数据之后，它需要一些时间来处理内部的事情，这段时间，是不允许外部打断的，所以，我们在SCI 操作之前，最好判断一下DREQ是否为高电平，如果不是，则等待DREQ 变为高。

了解完SCI，我们看下它的寄存器

| SCI 寄存器 | | | | |
|---------|----|--------|-------------|------------|
| 寄存器 | 类型 | 复位值 | 缩写 | 描述 |
| 0X00 | RW | 0X0800 | MODE | 模式控制 |
| 0X01 | RW | 0X003C | STATUS | VS1003状态 |
| 0X02 | RW | 0X0000 | BASS | 内置低音/高音增强器 |
| 0X03 | RW | 0X0000 | CLOCKF | 时钟频率+倍频数 |
| 0X04 | RW | 0X0000 | DECODE_TIME | 解码时间 |
| 0X05 | RW | 0X0000 | AUDATA | Misc. 音频数据 |
| 0X06 | RW | 0X0000 | WRAM | RAM 写/读 |
| 0X07 | RW | 0X0000 | WRAMADDR | RAM 写/读基址 |
| 0X08 | R | 0X0000 | HDATA0 | 流头数据0 |
| 0X09 | R | 0X0000 | HDATA1 | 流头数据1 |
| 0X0A | RW | 0X0000 | AIADDR | 用户代码起始地址 |
| 0X0B | RW | 0X0000 | VOL | 音量控制 |
| 0X0C | RW | 0X0000 | AICTRL0 | 应用控制寄存器0 |
| 0X0D | RW | 0X0000 | AICTRL1 | 应用控制寄存器1 |
| 0X0E | RW | 0X0000 | AICTRL2 | 应用控制寄存器2 |
| 0X0F | RW | 0X0000 | AICTRL3 | 应用控制寄存器3 |

VS1003 总共有16 个SCI 寄存器，在这里我们就不一一介绍了，有兴趣的可以查看下VS1003的用户手册进行查阅。我们这里只是介绍几个主要的寄存器。

首先是MODE 寄存器，SCI_MODE 用于控制VS1003 的操作，是最关键的寄存器之一，该寄存器的复位值为0x0800，其实就是默认设置为新模式。

下表为MODE 寄存器的各位描述：

| 位 | 名称 | 功能 | 值 | 描述 |
|---|--------------------------|-----------|--------|-----------------|
| 0 | SM_DIFF | 微分 | 0 1 | 正常同相音频 左声道反相 |
| 1 | SM_SETTOZERO | 设置为 0 | 0 1 | 对 错 |
| 2 | SM_RESET | 软件复位 | 0 1 | 不复位 复位 |
| 3 | SM_OUTOFWAV | 跳出 WAV 解码 | 0 1 | 不 是 |
| 4 | SMPDOWN | 掉电 | 0 1 | 电源开 掉电模式 |
| 5 | SM_TESTS | 允许 SDI 测试 | 0 1 | 不允许 允许 |
| 6 | SM_STREAM | 流模式 | 0 1 | 不 是 |
| 7 | SM_SETTOZERO 2 | 设置为 0 | 0 1 | 对 错 |

| | | | | |
|----|--------------------|------------------|--------|--------------|
| 8 | SM_DACT | DCLK 有效沿 | 0 1 | 上升沿 下降沿 |
| 9 | SM_SDIORD | SDI 位顺序 | 0 1 | 高位在前 低位在前 |
| 10 | SM_SDISHARE | 共享 SPI 片选 | 0 1 | 不 是 |
| 11 | SM_SDINEW | VS1002 自身 SPI 模式 | 0 1 | 不 是 |
| 12 | SM_ADPCM | ADPCM 录音允许 | 0 1 | 不 是 |
| 13 | SM_ADPCM_HP | ADPCM 高通滤波允许 | 0 1 | 不 是 |
| 14 | SM_LINE_IN | ADPCM 录音源选择 | 0 1 | 麦克风 线路输入 |

接着是我们的BASS 寄存器，该寄存器可以用于设置VS1003 的高低音效。该寄存器的各位描述如下图所示：

| 名称 | 位 | 描述 |
|---------------------|--------|----------------------------------|
| ST_AMPLITUDE | 15: 12 | 高音控制, 1.5dB 步进 (-8..7 ,为 0 表示关闭) |
| ST_FREQLIMIT | 11: 8 | 最低频限 1000Hz 步进 (0..15) |
| SB_AMPLITUDE | 7: 4 | 低音加重, 1dB 步进 (0..15 ,为 0 表示关闭) |
| SB_FREQLIMIT | 3: 0 | 最低频限 10Hz 步进 (2..15) |

通过这个寄存器以上位的一些设置，我们可以随意配置自己喜欢的音效（其实就是高低音的调节）。

当 SB_AMPLITUDE 不为零时，VSBE 低音提升有效。用户优先设置 SB_AMPLITUDE，大约在音频系统还原的最低频率 1.5 倍时间设置 SB_FREQLIMIT。例如，设置 SCI_BASS 为 0x00f6，将在 60Hz 以下获得 15dB 提升。因为 VSBE 尽量避免音频削波，它为动态音乐素材提供了一个最合适的低音提升，或者说，当回放音量不是开到最大，也就不能产生低音：所以源素材中一开始就必须含有一些低频成分。高音控制 VSTC 在 ST_AMPLITUDE 非零时有效。例如设置 SCI_BASS 为 0x7A00 将在 10KHz 以上获得 10.5dB 的高音提升。低音提升使用大约 3.0MIPS 和高音控制 1.2MIPS 在 44100Hz 采样率的时候。两者可同时实现。

接下来是我们的CLOCKF 寄存器，这个寄存器用来设置时钟频率、倍频等相关信息，寄存器的各位描述如下：

| SCI_CLOCKF 位 | | |
|----------------|--------|-------|
| 名称 | 位 | 描述 |
| SC_MULT | 15: 13 | 时钟倍频数 |
| SC_ADD | 12: 11 | 允许倍频 |
| SC_FREQ | 10: 0 | 时钟频率 |

SC_MULT 使内部倍频器有效，通过XTALI的倍乘，得到一个较高的频率的CLKI，对应的值如下：

| SC_MULT 域的值 | 掩码 | CLKI |
|----------------|--------|-------------|
| 0 | 0x0000 | XTALI |
| 1 | 0x2000 | XTALI × 1.5 |
| 2 | 0x4000 | XTALI × 2.0 |
| 3 | 0x6000 | XTALI × 2.5 |
| 4 | 0x8000 | XTALI × 3.0 |
| 5 | 0xA000 | XTALI × 3.5 |
| 6 | 0xC000 | XTALI × 4.0 |
| 7 | 0xE000 | XTALI × 4.5 |

SC_ADD 告诉解码器硬件允许SC_MULT 以什么数值增加倍频数。如果较多的周期是临时的需要解码WMA流。对应值如下：

| SC_ADD 域值 | 掩码 | 倍频增量 |
|--------------|--------|------|
| 0 | 0x0000 | 禁止修改 |
| 1 | 0x0800 | 0.5x |
| 2 | 0x1000 | 1.0x |
| 3 | 0x1800 | 1.5x |

SC_FREQ 用于当输入时钟是比12.288MHz 高的其它频率时。XTALI 被设置为4KHz 步进。寄存器中这个值正确的计算公式是 (XTALI-8000000) /4000 ,XTALI的单位是Hz。

注意：默认的值为0，是假设XTALI的频率是12.288MHz。

最后我们再来看下音量寄存器 (VOL) ， SCI_VOL可以控制播放器硬件音量。对每个声道，一个0 到254间的数被定义为从最大音量级别以0.5dB衰减。左声道值乘256。因而，最大的音量是0，而静音为0xFEFE。

例如：若左声道为-2.0dB，右声道为-3.5dB： $(4 \times 256) + 7 = 0x407$ 。注意，在启动的时候被设置为满音量。软件复位不会改变音量设定。

注意：设置SCI_VOL为0xFFFF将使芯片进入模拟掉电模式。

数据请求脚 DREQ

VS1003 的数据传输是通过DREQ 控制的，DREQ 脚在VS1003 的FIFO 在能够接受数据的时候输出。嵌入式专业技术论坛（www.armjishu.com）出品

出高电平。此时，VS1003可获取至少32Byte的SDI数据或一个SCI命令。遵循这个标准，当DREQ 变低时，发送器必须停止发送新的数据。

因为有32Byte 的保险区域(数据缓冲区),当检测到DREQ 信号时，发送器(MCU)须发送32Byte 的SDI数据。易于和慢速的微控制器接口。

注意：VS10xx 系列产品直到VS1002, DREQ 信号仅在SDI传送中使用。在VS1003中, DREQ 信号也被使用于告知SCI的状态。

VS1003 的工作流程

VS1003 的介绍，我们就先介绍到这里，下面我们看下如何去使用这个 VS1003，前面我们都是对它进行了一个介绍，让我们去了解了它，那么又是如何去运用它的呢？怎么样才能通过这个模块去播放出我们想要的音乐歌曲呢？下面我们就来给大家介绍下。

播放音乐所要操作的步骤流程为以下几步：

➤ 1) 时钟的选择

VS1003 操作于单时钟，12.288MHz 作为主时钟。此时钟可以由外部电路产生（连接至XTAL1）或使用内部晶体振荡器接口（XTAL1和XTAL0 脚）。

➤ 2) 复位 VS1003

这里包括了硬复位和软复位，是为了让VS1003 的状态回到原始状态，准备解码下一首歌曲。这里建议大家在每首歌曲播放之前都执行一次硬件复位和软件复位，以便更好的播放音乐。

硬件复位：当 XRESET 线被拉低，VS1003 被复位，所有的控制寄存器和内部状态都被设置为初始值。XRESET 由任何外部时钟异步产生。复位模式同时也是全掉电模式，VS1003 的数字和模拟部分仅消耗很小的功率，而且时钟停止，XTAL0 被接地。

软件复位：在一些情况下解码器软件被复位，就是 SCI_MODE 的 bit2 引起。然后等待至少 2us，DREQ 线仍然保持低电平至少 16600 个时钟周期，意味着在 12.288MHz 工作的 VS1003 有约 1.35ms 的延时。在 DREQ 变高之后，你可以照常进行回放。如果你不想 VS1003 截掉低比特率数据流的尾部，而你又想进行软件复位。建议在文件之后，复位之前遵照 DREQ 的协定，向 SDI 送入 2048 个零。这对 MIDI 文件尤其重要，尽管你可以通过 SCI_HDAT1 选取。如果你打算中断 WAV,WMA,MIDI 文件的播放，置位模式寄存器中的 SM_OUTOFWAV，并等待直到 SCI_HDAT1 被清空（2 秒超时）在继续操作之前需要软件复位。MP3 通常不允许 SM_OUTOFWAV 因为它是一种流格式，所以需要超时处理。

➤ 3) 配置 VS1003 的相关寄存器

这里我们配置的寄存器包括VS1003 的模式寄存器（MODE）、时钟寄存器（CLOCKF）、音调寄存器（BASS）、音量寄存器（VOL）等。

➤ 4) 发送音频数据

当经过以上两步配置以后，我们剩下来要做的事情，就是往VS1003 里面扔音频数据了，只要是VS1003 支持的音频格式，直接往里面丢就可以了，VS1003 会自动识别，并进行播放。不过发送数据要在DREQ 信号的控制下有序的进行，不能乱发。这个规则很简单：只要DREQ变高，就向VS1003 发送32 个字节。然后继续等待DREQ 变高，直到音频数据发送完。

➤ 5) 播放/解码

这是 VS1003 的一个常规操作模式。SDI 数据被解码，解码的采样率变换到内部模拟 DAC 允许的范围。如果找不到能被解码的数据，SCI_HDAT0 和 SCI_HDAT1 被设置为 0 并且模拟输出静音。所有的不同格式的文件可以接着播放，且在两文件之间不需要软件复位。在每个流末尾发送至少 4 个零。尽管如此，在两个流之间使用软件复位不失为一个好主意，同样要警戒紧挨着的损坏的文件。在这种情况下你可以在发送软件复位之前，等待解码完成（SCI_HDAT0 和 SCI_HDAT1 变为零）。

经过以上几步，我们就可以播放音乐了。

7.69.3 硬件设计

模块引脚说明



| 标号 | 管脚定义 | 对应开发板 I/O 接口 | 功能 |
|----|--------|--------------|--------------------|
| 1 | XDCS | PE8 | 数据片选端 |
| 2 | XCS | PE7 | 片选输入 |
| 3 | DREQ | PC13 | 数据请求脚 |
| 4 | SCLK | PA5 | 串行时钟输入端 |
| 5 | MOSI | PA7 | 数据串行输入端 |
| 6 | MISO | PA6 | 数据串行输出端 |
| 7 | XRESET | PE6 | 复位端 |
| 8 | GND | GND | 地 |
| 9 | 5V | VCC | 电源（可接开发板的 3.3V 电压） |

硬件环境搭建

根据程序代码与前面的模块管脚对应开发板的 I/O 接口，把相对应的管脚用杜邦线的模式对应连接，这个是和模块上的标明的引脚是一一对应的。

连接的结果如下图所示：



7.69.4 软件设计

进入例程的文件夹，然后打开\Project\Project.uvproj 文件

The screenshot shows the Keil uVision IDE interface. The top menu bar includes File, Edit, View, Project, Flash, Debug, Peripherals, Tools, SVCS, Window, and Help. Below the menu is a toolbar with icons for file operations like Open, Save, and Build. The title bar says "神舟III号". The left pane shows the project structure under "Project":

- 神舟III号
 - User
 - main.c
 - stm32f10x_it.c
 - vs1003.c
 - spi.c
 - StdPeriph_Driver
 - stm32f10x_rcc.c
 - stm32f10x_gpic.c
 - stm32f10x_spi.c
 - CMSIS
 - core_cm3.c
 - system_stm32f1.c
 - RVMDK
 - startup_stm32f1.s
 - Doc
 - readme.txt

```
01 /***** (C) COPYRIGHT 2013 www.armjishi.com *****
02 * 文件名 : main.c
03 * 描述 : 神舟III号开发板MP3模块播放功能
04 * 实验平台: STM32神舟开发板
05 * 标准库 : STM32F10x_StdPeriph_Driver V3.5.0
06 * 作者 : www.armjishu.com
07 *****
08 #include "stm32f10x.h"
09 #include "MP3Sample.h"
10 #include "vs1003.h"
11 #include "spi.h"
12
13 void SZ_STM32_SysTickInit(uint32_t HzPreSecond);
14 /*
```

可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    unsigned int i;

    /* 初始化系统定时器SysTick, 每秒中断1000000次 */
    SZ_STM32_SysTickInit(1000000);

    /* 初始化MP3引脚，及配置SPI*/
    InitPortVS1003();
    VS_Reset();
    /* Infinite loop 主循环 */
    while (1)
    {
        for(i=0;i<sizeof(music);i++)
        {
            while ( VS_DQ==0 ); /* 等待空闲 */
            VS_WR_Data(music[i]);
        }
    }
}
```

代码分析1：在系统启动文件(startup_stm32f10x_xx.s)中已经调用SystemInit()初始化了72MHZ时钟，最开始的例程已经对此分析过了，还有不明白的可以看下前面的例程。

代码分析 2：函数 SZ_STM32_SysTickInit(1000000)，初始化系统定时器 Systick。前面我们已经介绍，大家可以参考系统时钟章节。

代码分析 3：函数 InitPortVS1003()，初始化 MP3 模块使用的引脚。

```
void InitPortVS1003(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC|RCC_APB2Periph_GPIOE, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;           //PC13
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;         //输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;      //推挽输出
    GPIO_Init(GPIOE, &GPIO_InitStructure);

    SPI1_Init();
}
```

我们的 MP3 模块中引出了 VS1003 芯片的 9 个引脚，除去电源、地。实际上用到的引脚是 7 个。这里就是对这 7 个引脚的初始化。模块的每个引脚对应的 STM32F103ZET 芯片具体 GPIO 管脚在实验原理部分给大家列出。这里 MP3 模块和开发板之间的数据传输是通过 SPI 方式来进行的。我们使用函数 SPI1_Init() 初始化 SPI。

代码分析 4：对 MP3 模块进行复位。

```

void VS_Reset()
{
    while(VS_HD_Reset());
    VS_Soft_Reset();
}

```

这里的复位分为硬件复位、软件复位。硬件复位通过函数 VS_HD_Reset() 实现，当复位成功时函数返回 0，否则返回 1。我们这里通过语句 “while(VS_HD_Reset());” 判断硬件复位是否成功，成功时返回 0，则 while(0) 为假，跳出当前的循环，程序得以往下运行。复位不成功，程序一直没有跳出当前的 while 循环。

在函数 VS_Soft_Reset() 中，对除了软件复位之外，还对 VS1003 的时钟进行了设置。

```

void VS_Soft_Reset(void)
{
    u8 retry=0;
    while(VS_DQ==0); //等待软件复位结束
    VS_SPI_ReadWriteByte(0xFF); //启动传输
    retry=0;
    while(VS_RD_Reg(SPI_MODE)!=0x0800)// 软件复位,新模式
    {
        VS_WR_Cmd(SPI_MODE,0x0804); // 软件复位,新模式
        SysTickDelay(2000); //delay_ms(2); //等待至少1.35ms
        if(retry++>100)break;
    }
    while(VS_DQ==0); //等待软件复位结束

    retry=0;
    while(VS_RD_Reg(SPI_CLOCKF)!=0x9800)//设置VS10XX的时钟,3倍频,1.5xADD
    {
        VS_WR_Cmd(SPI_CLOCKF,0x9800); //设置VS10XX的时钟,3倍频,1.5xADD
        if(retry++>100)break;
    }
}

```

代码分析 5：进行了上面的设置，我们通过 while 主循环往 MP3 模块写数据。

```

/* Infinite loop 主循环 */
while (1)
{
    for(i=0;i<sizeof(music);i++)
    {
        while( VS_DQ==0 ); /* 等待空闲 */
        VS_WR_Data(music[i]);
    }
}

```

我们的 MP3 数据存放在 music 数组中。这个数组我们定义在 “MP3Sample.h” 文件中。我们通过函数 szieof() 得出，数组的大小，通过 for 循环往模块发送音频数据。

```

unsigned char const music[]=
0x49,0x44,0x33,0x03,0x00,0x00,0x00,0x4A,0x03,0x54,0x41,0x4C,0x42,0x00,0x00,
0x00,0x0D,0x00,0x00,0x00,0xD2,0xBB,0xD6,0xBB,0xC3,0xA8,0xB5,0xC4,0xC2,0xC3,0xD0,
0xD0,0x54,0x50,0x45,0x31,0x00,0x00,0x07,0x00,0x00,0x00,0x00,0x00,0xF8,0xE9,0xF3,
0xBF,0xC9,0x54,0x49,0x54,0x32,0x00,0x00,0x0B,0x00,0x00,0x00,0x46,0x6F,0x72,
0x65,0x76,0x65,0x72,0x20,0x32,0x31,0x41,0x50,0x49,0x43,0x00,0x00,0x24,0xB2,0x00,
0x00,0x00,0x69,0x6D,0x61,0x67,0x65,0x2F,0x6A,0x70,0x65,0x67,0x00,0x03,0x00,0xFF,
0xD8,0xFF,0xE0,0x00,0x10,0x4A,0x46,0x49,0x46,0x00,0x01,0x01,0x00,0x64,0x00,

```

发送音频数据之前，我们要判断 VS_DQ 是否等于 0，即 DREQ 信号线是否等于 0。当 DREQ 信号线为高时发送数据。这里为什么要 DREQ 信号线为高的时候，我们才发送数据呢？

VS1003 为用户准备了数据缓冲区做为音频数据的缓冲。你只要保证这个缓冲区里始终有数据作为待解码的对象，就可以得到流畅的音乐了。VS1003 设置了一个中断脚 DREQ，当 DREQ 变为高时，外部可以至少为 vs1003 发送 32 个字节。我们设计代码的时候，根据这个原理，当 DREQ 变为高时，向 VS1003 的缓冲区发送数据，从而保证缓冲区不为空，音乐流畅的播放。

代码分析 6：代码的基本思路我们已经讲完。下面我们看一下写数据函数 VS_WR_Data ()。

```
//向vs10xx写数据
//data:要写入的数据
void VS_WR_Data(u8 data)
{
    // VS_SPI_SpeedHigh(); //高速,对vs1003B,最大值不能超过36.864/4Mhz, 这里设置为9M
    VS_XDCS=0;
    VS_SPI_ReadWriteByte(data);
    VS_XDCS=1;
}
```

函数 VS_WR_Data ()，其实是调用函数 VS_SPI_ReadWriteByte ()，向 VS1003 的缓冲区写数据的。可以看到在写入的过程中 XDCS 信号线拉低、拉高这个根据时序图设定。而函数 VS_SPI_ReadWriteByte () 是 SPI 方面的内容，这些 ST 官方提供了库函数。我们的用户手册也有对应 SPI 章节的讲解，感兴趣的朋友可以参考一下。

7.69.5 下载与测试

如果使用JLINK下载固件，请按 [如何使用JLINK V8下载固件到神舟III号开发板](#) 小节进行操作。

如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。

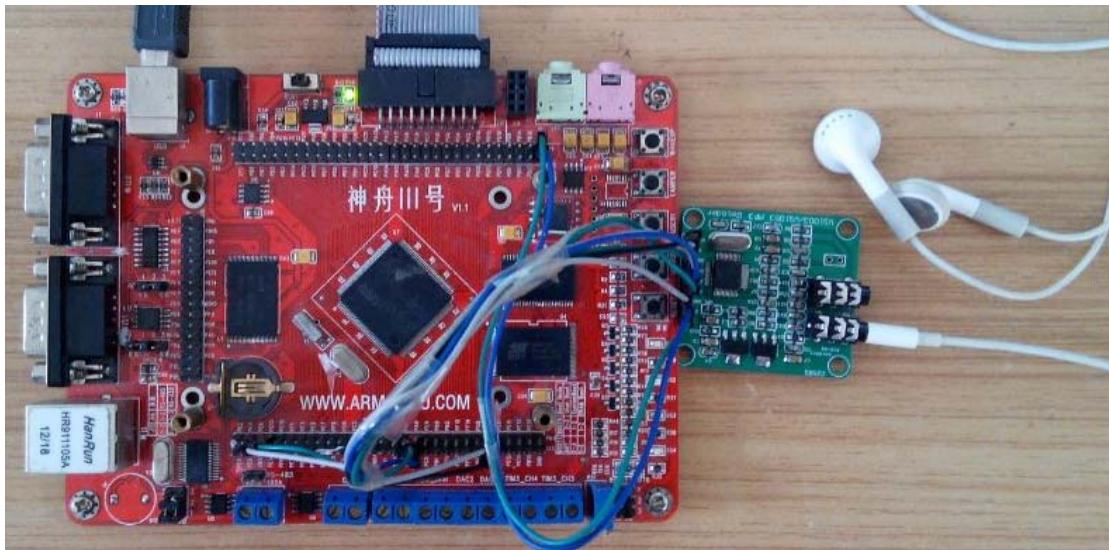
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.69.6 实验现象

按照下面的引脚表格说明将 MP3 模块连接到神舟 III 开发板。

| 标号 | 管脚定义 | 对应开发板 I/O 接口 | 功能 |
|----|--------|--------------|--------------------|
| 1 | XDCS | PE8 | 数据片选端 |
| 2 | XCS | PE7 | 片选输入 |
| 3 | DREQ | PC13 | 数据请求脚 |
| 4 | SCLK | PA5 | 串行时钟输入端 |
| 5 | MOSI | PA7 | 数据串行输入端 |
| 6 | MISO | PA6 | 数据串行输出端 |
| 7 | XRESET | PE6 | 复位端 |
| 8 | GND | GND | 地 |
| 9 | 5V | VCC | 电源(可接开发板的 3.3V 电压) |

最终的连接图如下：



我们将 MP3 实验代码下载到到神州 III 号开发板上，带上耳机就可以听到美妙的音乐了

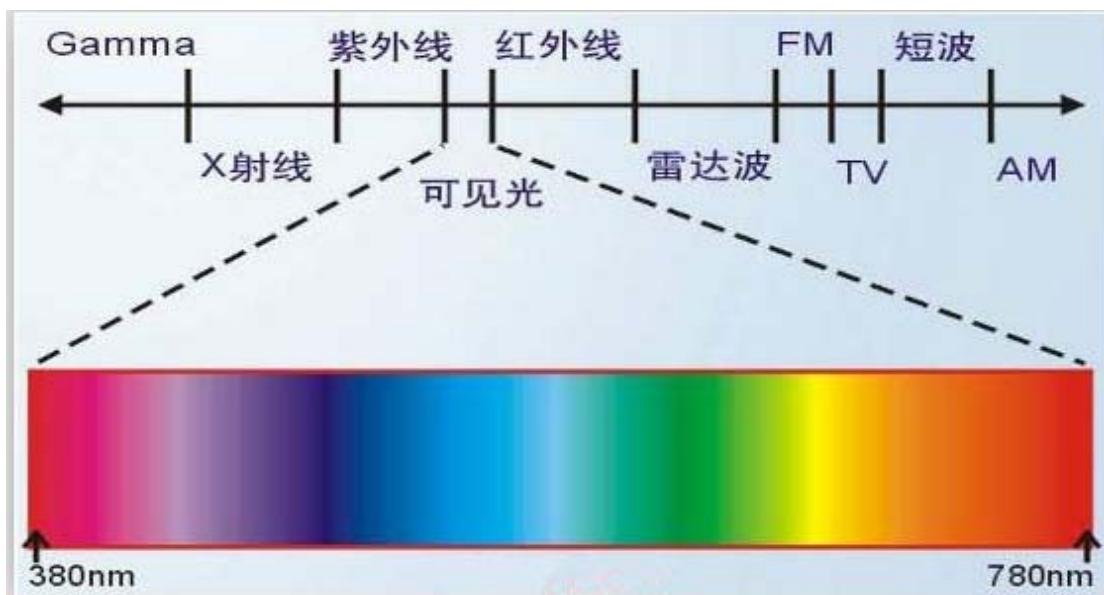
7.70 红外遥控器实验

7.70.1 什么是红外？红外的历史

在了解红外收发前，我们先来了解一下什么是红外，在日常生活中，我们通过眼睛能看到各种颜色的光。那我们是否了解各种颜色的光之间有什么区别的呢？难道就只是颜色不一样？那为什么光又分为那么多种颜色呢？

其实，光的颜色是由它的波长来决定的，那么波长又是什么呢？为了能让用户更能理解它的定义，我们不是科学家，不深入研究，只是了解一下它的定义就行了，我们前面知道了知道光的颜色是由波长决定的。光的波长就是光在单位时间内通过的位移（位移是从空间的一个位置运动到另一个位置，它的位置变化叫做在这一运动过程中的位移）。光的本质是电磁波，所以就会有频率的产生。光的频率在传播中保持不变，意思是在光通过不同介质的时候，频率不变而波长发生改变。因此，光的波长由光的频率(颜色)，以及传播的介质决定。

波长的单位我们可以用 nm 或者 um 来表示，各种颜色有各自的波长，人的眼睛能看到的可见光按波长从长到短排列，依次为红、橙、黄、绿、青、蓝、紫。其中红光的波长范围为 620~760nm；紫光的波长范围为 380~440nm。比紫光波长还短的光叫紫外线，比红光波长还长的光叫红外线。在 1800 年 4 月 24 日英国伦敦皇家学会 (ROYAL SOCIETY) 的威廉·赫歇尔发表太阳光在可见光谱的红光之外还有一种不可见的延伸光谱，具有热效应。他所使用的方法很简单，用一支温度计测量经过棱镜分光后的各色光线温度，由紫到红，发现温度逐渐增加，可是当温度计放到红光以外的部分，温度仍持续上升，因而断定有红外线的存在。红外线遥控就是利用波长为 760nm~400um 之间的近红外线来传送控制信号的。下图为各种光的波长示意图，红光外的为红外光，紫色光以外的为紫外光。



7.70.2 红外的特点和用途

红外的特点具有很强热效应、易于被物体吸收、穿透能力比可见光强、小角度（30度锥角以内），短距离、点对点直线数据传输，保密性强、传输速率较高。

红外的用途有非常多，例如在通讯领域的技术，常被应用在计算机及其外围设备、移动电话、数码相机、工业设备、网络接入设备，如调制解调器等，像我们这章所用到的红外收发，还有深海探测等无线通讯方面的技术；还可以被用于医疗用途，红外线对人体皮肤、皮下组织具有强烈的穿透力。外界红外线辐射人体产生的一次效应可以使皮肤和皮下组织的温度相应增高，促进血液的循环和新陈代谢，促进人的健康，红外线还有杀菌的能力。

因为我们这章用到的是红外通讯技术，所以它的其他用途我们就不作说明了，主要说下用作红外收发通讯的用途。红外通讯就是通过红外线传输数据。在电脑技术发展早期，数据都是通过线缆传输的，线缆传输连线麻烦，需要特制接口，颇为不便。无线通信技术逐渐发展起来

7.70.3 红外的功能参数

工作电压：2.7~5.5V 左右

接收距离：15~30M（米）左右

载波频率：37.9KHz

接收角度：+/-35°

BMP 宽度：3.5~8.5KHz

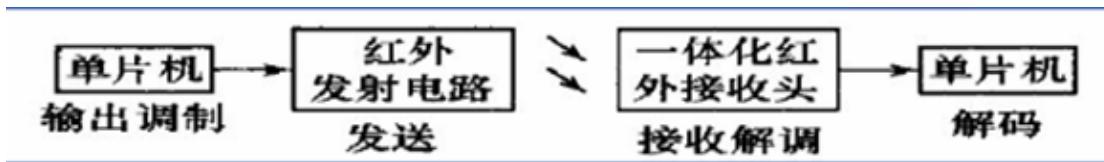
静态电流：0.9~1.5mA

抗干扰能力：综合能力较强，灵敏度较高

应用范围：通讯、医疗方面广泛应用

7.70.4 红外的工作原理

红外遥控有发送和接收两个组成部分。发送端将待发送的二进制信号编码调制为一系列的脉冲串信号，通过红外发射管发射红外信号。红外接收完成对红外信号的接收、放大、检波、整形，并解调出遥控编码脉冲，流程图如下：



红外发送原理：

把需要发送的信号编码调制为一系列的脉冲串信号由红外 LED 发送出去。

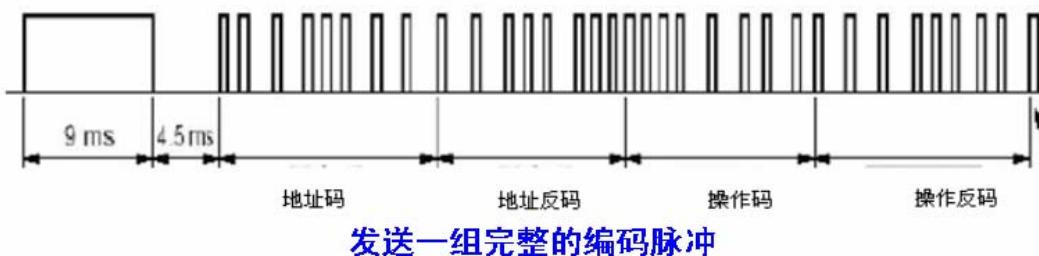
二进制信号的调制由单片机来完成，它把编码后的二进制信号调制成一定频率的间断脉冲串，相当于用二进制信号的编码乘以这个间断脉冲信号得到的间断脉冲串，即是调制后用于红外发射二极管发送的信号，如下图所示，要使红外发光二极管产生调制光，只需在它的驱动管上加上一定频率的脉冲电压。一般由键盘电路、红外编码芯片、电源和红外发射电路组成



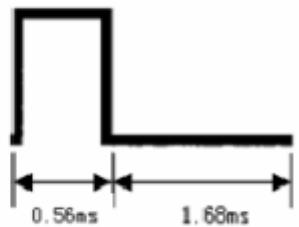
二进制码的调制

在同一个遥控电路中通常要使用实现不同的遥控功能或区分不同的机器类型，这样就要求信号按一定的编码传送，编码则会由编码芯片或电路完成。通过对用户码的检验，每个遥控器只能控制一个设备动作，这样可以有效地防止多个设备之间的干扰。就像是电视机、空调有专门的遥控器，遥控器上的按键各有各自的功能。

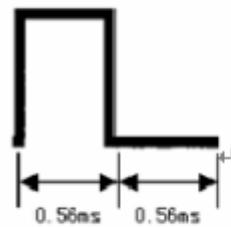
红外遥控发射芯片采用 PPM 编码方式（脉冲位置调制,又称脉位调制，由头码+脉冲数组成,称为 PPM 编码方式），当发射器按键按下后，将发射一组 108ms 的编码脉冲。遥控编码脉冲由前导码、16 位地址码（8 位地址码、8 位地址码的反码）和 16 位操作码（8 位操作码、8 位操作码的反码）组成，反码表示正数的反码与其原码相同；负数的反码是对其原码逐位取反，但符号位除外。由一个 9ms 的高电平（起始码）和一个 4.5ms 的低电平（结果码）组成作为接受数据的准备脉冲。解码的关键是如何识别“0”和“1”，从位的定义我们可以发现“0”、“1”均以以脉宽为 0.56ms 的电平开始、周期为 1.12ms 的组合表示二进制的“0”；以脉宽为 1.68ms、周期为 2.24ms 的组合表示二进制的“1”，根据这样的定义，我们就能读出发送的数据是什么了。



发送一组完整的编码脉冲



二进制码'1'



二进制码'0'

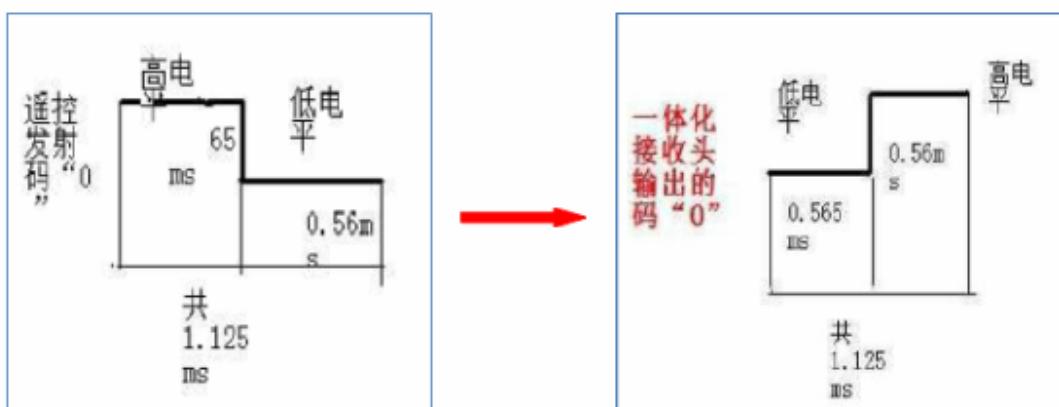
红外接收原理：

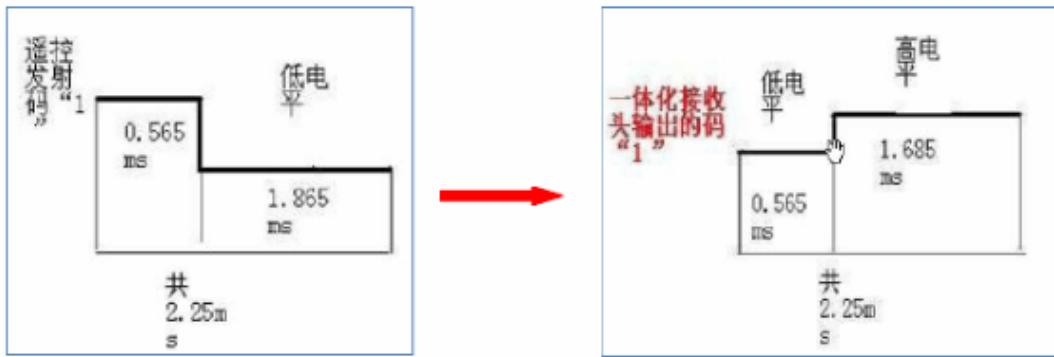
了解了红外接收头后，我们再来看下它是如何工作的，它的原理是什么。红外接收头内置接收管将红外发射管发射出来的光信号转换为微弱的电信号，此信号经由 IC 内部放大器进行放大，然后通过接收管的处理将波形整形后还原为遥控器发射出的原始编码，经由接收头的信号输出脚输入到电器上的编码识别电路，红外解码的关键就是识别 0 和 1。

我们前面介绍过红外接收是完成对红外信号的接收、放大、检波、整形，并解调出遥控编码脉冲的，这个是由接收头内部完成的。经过它的接收放大和解调会在输出端直接输出原始的信号。

红外接收头一般是接收、放大、解调一体头，一般红外信号经接收头解调后，数据“0”和“1”的区别通常体现在高低电平的时间长短或信号周期上，这个我们上面有介绍。单片机解码时，通常将接收头输出脚连接到单片机的外部中断，结合定时器判断外部中断间隔的时间从而获取数据。重点是找到数据“0”与“1”间的波形差别。

在这里特别强调：编码与解码是一对逆过程，不仅在原理上是一对逆过程，在码的发收过程也是互反的，即以前发射端原始信号是高电平，那接收头输出的就是低电平，反之亦然。因此为了保证解码过程简单方便，在编码时应该直接换算成其反码。如下图为红外发送与红外接收头接收的电平对比情况。





红外接收过程:

接收电路的红外接收管是一种光敏二极管，使用时要给红外接收二极管加反向偏压，它才能正常工作而获得高的灵敏度。红外接收二极管一般有圆形和方形两种。由于红外发光二极管的发射功率较小，红外接收二极管收到的信号较弱，所以接收端就要增加高增益放大电路。然而现在不论是业余制作或正式的产品，大都采用成品的一体化接收头，红外线一体化接收头是集红外接收、放大、滤波和比较器输出等的模块，性能稳定、可靠。所以，有了一体化接收头，人们不再制作接收放大电路，这样红外接收电路不仅简单而且可靠性大大提高。

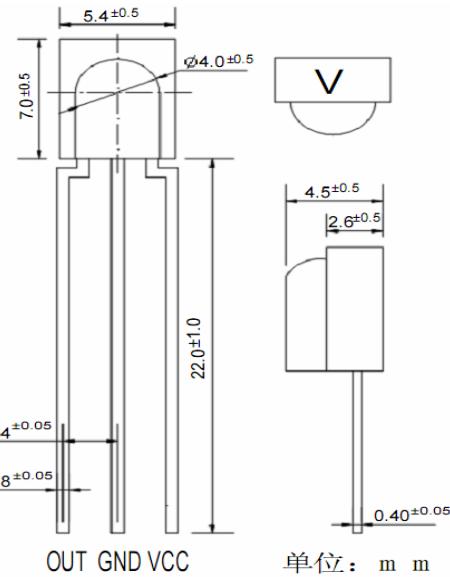
红外接收头包含两个芯片，一个是PD（即红外接收管），一个是IC。其中PD接收来自发射管的光信号（该信号已被调制），将光信号转换为电信号，即光电转换，常用于光接收器中。PD芯片属于典型的PIN结构光电二极管。由PD接收转换而来的电信号通过IC进行放大，自动增益控制，滤波，解调，波形整形，比较器输出交由后面的电路进行识别还原。以上就是红外接收头的接收过程。

7.70.5 实验原理

从上面了解了红外的收发原理后知道红外发送是以脉宽为0.56ms的电平开始、周期为1.12ms的组合表示二进制的“0”；以脉宽为1.68ms、周期为2.24ms的组合表示二进制的“1”，而编码与解码是一对逆过程因此我们就可以用定时器的输入捕获来进行脉宽的检测从而得出二进制信号。

7.70.6 硬件设计

本实验串口显示红外的信息。用到红外遥控器红外接头。我们主要看一下，红外接收头的实物图和管脚图如下：



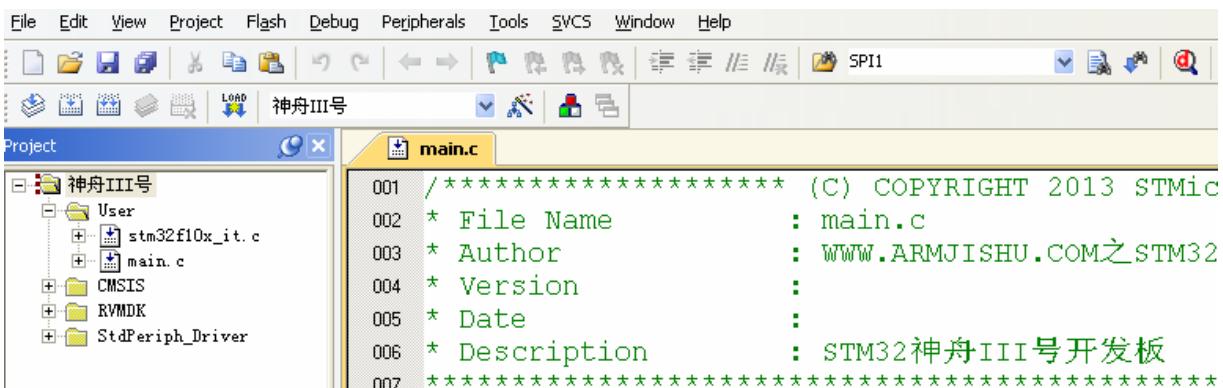
红外接收头有 3 个管脚，VCC、GND、OUT。分别是电源、地、数据引脚。

这 3 个管脚和神州 3 号开发板上的引脚的连接关系如下表。这里需要焊接将开发板上将对应的引脚引出。

| 神州 3 号开发板 | 连接线 | DJT11 | 功能 |
|-----------|---------|----------|----------|
| 3.3V | 杜邦线（绿色） | VDD | 供电 |
| PB9 | 杜邦线（白色） | DATA/OUT | 串行数据 单总线 |
| GND | 杜邦线（红色） | GND | 接地 |

7.70.7 代码分析

进入例程的文件夹，然后打开\MDK-ARM\Project.uvproj 文件



可以看到工程已经被打开，下面开始具体分析程序代码：

```
int main(void)
{
    u8 key;
    u8 t=0;
    u8 *str=0;
    /* USARTx configured as follow:
       - BaudRate = 115200 baud
       - Word Length = 8 Bits
       - One Stop Bit
       - No parity
       - Hardware flow control disabled (RTS and CTS signals)
       - Receive and transmit enabled */
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_Disable;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    STM32_Shenzhou_COMInit(&USART_InitStructure);
}
```

代码分析 1：STM32_Shenzhou_COMInit()函数初始化串口。

代码分析 2：Remote_Init()函数初始化红外遥控。

```
//红外遥控初始化
//设置IO以及定时器4的输入捕获
void Remote_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_ICInitTypeDef TIM_ICInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //使能
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE); //TIM4 |

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //1
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD; //上拉
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_SetBits(GPIOB, GPIO_Pin_9); //初始化GPIOB.9

    TIM_TimeBaseStructure.TIM_Period = 10000; //设定计数器自动重
    TIM_TimeBaseStructure.TIM_Prescaler = (72-1); //预分频器
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; //步
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
```

初始化函数用于初始化 IO 口，并配置 TIM4_CH4 为输入捕获。输入捕获，比如捕获一个高电平的时候，首先检查到一个上升沿，启动定时器计时，一段时间后来一个下降沿，定时器停止计时。这样我们就可以知道高电平持续的时间了。捕获一个低电平的时候，也是一样的道理。

本初始化设置 TIM4_CH4 相关参数，这里的配置跟输入捕获实验的配置基本接近，大家可以参考输入捕获实验的讲解。

代码分析 4：定时器 TIM4 的中断服务函数，TIM4_IRQHandler()。

```
//定时器中断服务程序
void TIM4_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM4,TIM_IT_Update)!=RESET)
    {
        if(Rmtsta&0x80)//上次有数据被接收到了
        {
            RmtSta&=~0X10; //.
            if((RmtSta&0X0F)==0X00) RmtSta|=1<<6;//.
            if((RmtSta&0X0F)<14) RmtSta++;
            else
            {
                RmtSta&=~(1<<7); //清空引导标识
                RmtSta&=0XF0; //清空计数器
            }
        }
        if(TIM_GetITStatus(TIM4,TIM_IT_CC4)!=RESET)
        {
            if(RDATA)//上升沿捕获
            {

```

在这个函数里实现对红外接收头接收到的红外信号的高电平脉冲的捕获，同时根据我们之前简介的协议内容来解码。这里主要是如何识别传输进来的数据是“1”还是“0”。

这两个数据的判别可以通过高电平脉冲持续的时间长度来判断。“0”、“1”均以脉宽为 0.56ms 的电平开始、但第二个电平的时候数据“0”的脉宽大约是 0.56ms，数据“1”的是 1.685ms。代码中判别数据是根据的是这样的思路。

```
if(RmtSta&0X10) //完成一次高电平捕获
{
    if(RmtSta&0X80)//接收到引导码
    {

        if(Dval>300&&Dval<800) //560为标准值,560us
        {
            RmtRec<<=1; //左移一位.
            RmtRec|=0; //接收到0
        }else if(Dval>1400&&Dval<1800) //1680为标准值,1680us
        {
            RmtRec<<=1; //左移一位.
            RmtRec|=1; //接收到1
        }else if(Dval>2200&&Dval<2600) //得到按键键值增加的信息 :
        {
            RmtRec<<=1; //左移一位.
            RmtRec|=1; //接收到1
        }
    }
}
```

在此中断函数里面用全局变量存储解码结果，以便其它函数调用。

代码分析 5：函数 Remote_Scan() 扫描解码结果。

```
//处理红外键值
//返回值：
//    0,没有任何按键按下
//其他,按下的按键键值。
u8 Remote_Scan(void)
{
    u8 sta=0;
    u8 t1,t2;
    if(RmtSta&(1<<6))//得到一个按键的所有信息了
    {
        t1=RmtRec>>24;          //得到地址码
        t2=(RmtRec>>16)&0xff;   //得到地址反码
        if(t1==(u8)~t2)//检验遥控识别码地址
        {
            t1=RmtRec>>8;
            t2=RmtRec;
            if(t1==(u8)~t2)sta=t1;//键值正确
        }
        if((sta==0)||((RmtSta&0X80)==0))//按键数据错
        {
            RmtSta=~(1<<6); //清除接收到有效按键标识
            RmtCnt=0;           //清除按键次数计数器
        }
    }
    return sta;
}
```

输入捕获解码的红外数据，通过该函数传送给其它函数。扫描的时候判断扫描的地址码，和地址反码中的数据是否一致。一致的话，再判断键值是否一致，一致的话将键值取出做为返回值（sta=t1）。最后，我们通过返回的键值判断我们按下遥控器上的那个按键，并通过串口输出按下的键值。

7.70.8 下载与测试

如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

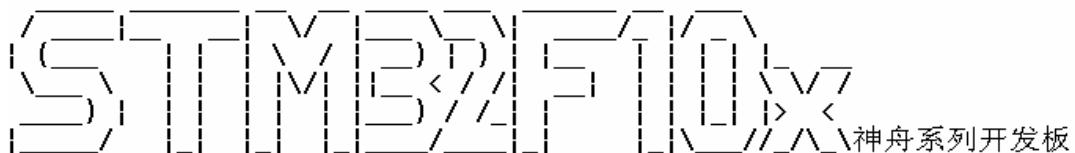
7.70.9 实验现象

我们将红外遥控连接到神州 3 号开发板上，连线方法如下：

| 神州 3 号开发板 | 连接线 | DJT11 | 功能 |
|-----------|---------|-------|----------|
| 3.3V | 杜邦线（绿色） | VDD | 供电 |
| PB9 | 杜邦线（白色） | DATA | 串行数据 单总线 |
| GND | 杜邦线（红色） | GND | 接地 |

将代码烧录到神州 3 号开发板上后，用串口连接电脑，复位，在电脑上的超级终端上也会显示下图所示：

WWW.ARMJISHU.COM USART2 printf configured...
WWW.ARMJISHU.COM! ##### (Nov 20 2013 - 10:39:12)



WWW.ARMJISHU.COM use STM32F10X_STDPERIPH_VERSION 3.5.0
产品内部Flash大小为: 256K字节! www.armjishu.com
系统内核时钟频率(SystemCoreClock)为: 72000000Hz.

按下红外遥控器的按键，超级终端上打印按键的对应信息，比如按下按键“1”，串口打印信息如下：

红外遥控器上的键值： 1

按键的值： 48

红外遥控器上的键值： 1

7.71 口程序实验

7.71.1 ENC28J60网口程序实验的意义与作用

以太网是应用最为广泛的局域网，包括标准的以太网(10Mbit/s)、快速以太网(100Mbit/s)、千兆(1000Mbit/s)以太网和10G(10Gbit/s)以太网，采用的是CSMA/CD访问控制法，它们都符合IEEE802.3规范。

在本实验中使用的ENC28J60是一款SPI接口的以太网控制芯片来实现10M以太网，对那些没有集成以太网控制器的处理器或者微控制器来说，只需要一个SPI接口或者用GPIO模拟一个SPI接口，就可以通过外扩ENC28J60实现标准的10M以太网。通过本实验，我们将了解以太网的一些基本知识，掌握如何使用ENC28J60设计以10M以太网接口，使我们以后设计的产品功能更丰富，使用更方便。

7.71.2 实验原理

以太网最早由Xerox（施乐）公司创建，在1980年，DEC、Intel和Xerox三家公司联合开发成为一个标准。以太网是应用最为广泛的局域网，包括标准的以太网(10Mbit/s)、快速以太网(100Mbit/s)和10G(10Gbit/s)以太网，采用的是CSMA/CD访问控制法，它们都符合IEEE802.3。

IEEE802.3规定了包括物理层的连线、电信号和介质访问层协议的内容。以太网是当前应用最普遍的局域网技术。它很大程度上取代了其他局域网标准，如令牌环、FDDI和ARCNET。历经100M以太网在上世纪末的飞速发展后，目前千兆以太网甚至10G以太网正在国际组织和领导企业的推动

下不断拓展应用范围。

常见的 802.3 应用为：

- ◆ 10M: 10base-T(铜线 UTP 模式)
- ◆ 100M: 100base-TX(铜线 UTP 模式)
- ◆ 100base-FX(光纤线)
- ◆ 1000M: 1000base-T(铜线 UTP 模式)

在神舟 III 号中，使用 SPI 接口的 ENC28J60 以太网控制器实现 10M 以太网功能，ENC28J60 是 MicrochipTechnology(美国微芯科技公司)近期推出的 28 引脚独立以太网控制器。符合 IEEE802.3 协议，而且只有 28 引脚，既能提供相应的功能，又可以大大简化相关设计，减小空间。

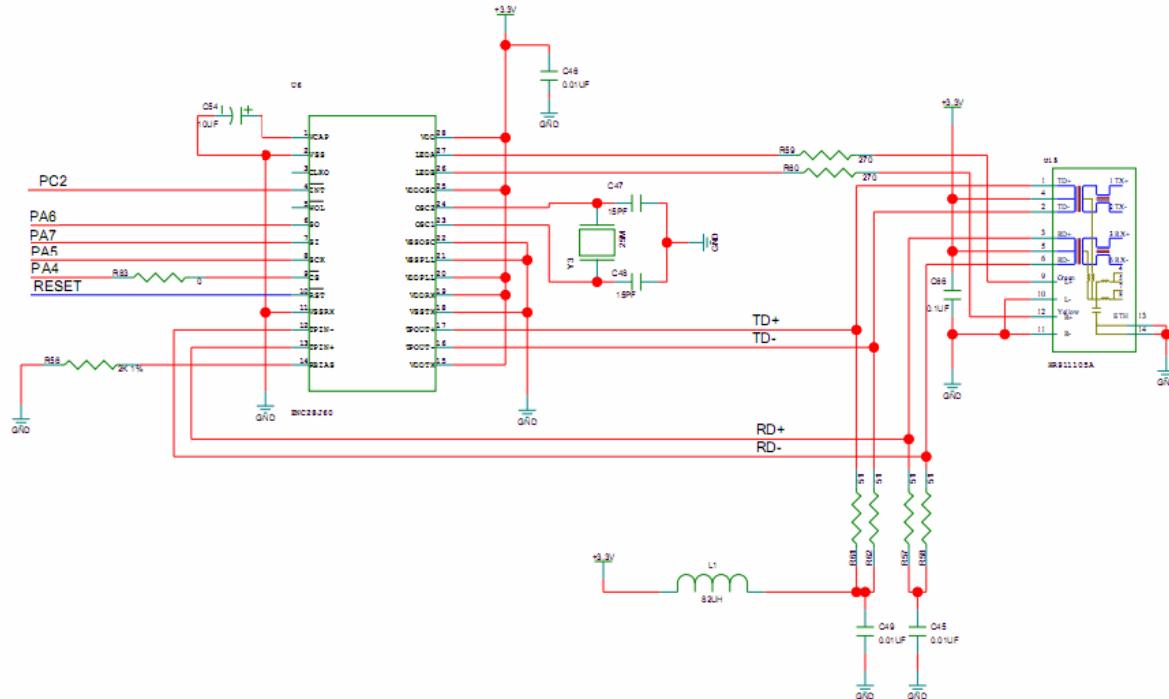
ENC28J60 与 STM32F103ZET6 处理器通过 SPI1 连接访问，支持 10Mbps。STM32F103ZET6 通过 SPI1 接口发送命令，访问 ENC28J60 的寄存器或读写接收/发送缓冲区，完成相关操作。复位也可通过 SPI1 接口由软件实现，软件复位不影响 RESET 引脚的状态。

ENC28J60 是极具特色的独立以太网控制器：SPI 接口使得小型单片机也能具有网络连接功能；集成 MAC 和 PHY 无需其他外设；具有可编程过滤功能，可自动评价、接收或拒收多种信息包，减轻了主控单片机的处理负荷；内部继承可编程的 8KB 双端口 SRAM 缓冲器，操作灵活方便。不足之处为仅支持 10BASET。

在本实验中，我们实现了 HTTP 网页访问，支持平行交叉网线自适应，一根网线就可以很方便的与路由器或 PC 连接，连接后可以通过网络访问和控制板上的资源，从而实现以太网远程控制功能。

7.71.3 硬件设计

在神舟 III 号开发板上，使用了 ENC28J60 SPI 以太网变压器以及 HR911105A 集成变压器的 RJ45 接头来实现 10M 以太网口。



ENC28J60 是一款在嵌入式系统中应用的极为普遍的 SPI 接口以太网控制器。它具有如下特性：

- ◆ IEEE802.3 兼容的以太网控制器
- ◆ 接收器和冲突抑制电路
- ◆ 支持一个带自动极性检测和校正的 10BASE-T 端口
- ◆ 支持全双工和半双工模式
- ◆ 可编程在发生冲突时自动重发
- ◆ 可编程填充和 CRC 生成
- ◆ 可编程自动拒绝错误数据包
- ◆ 最高速度可达 10Mb/s 的 SPI 接口。

神舟 III 号开发板上，处理器通过 SPI1 接口访问 ENC28J60 以太网控制器，GPIO 和接口对应关系如下表所示：

| ENC28J60管脚 | GPIO管脚 | SPI信号 | 说明 |
|------------|--------|-----------|---------------|
| /CS | PA4 | SPI1 NSS | SPI1接口信号 |
| SCK | PA5 | SPI1 SCK | |
| SO | PA6 | SPI1 MISO | |
| SI | PA7 | SPI1 MOSI | |
| /INT | PC2 | —— | ENC28J60的中断输入 |

ENC28J60 的硬件设计需要注意复位电路时钟振荡器，振荡器启动定时器，时钟输出引脚，变压器、终端和其他外部器件，输入/输出电平等几个方面。

时钟振荡器

ENC28J60 需要一个 25MHz 的晶振，接在 OSC1 和 OSC2 脚上；也可由外部时钟信号来驱动。此时 3.3V 的外部时钟接在 OSC1 脚上，OSC2 断开或者通过一个电阻接地来降低系统噪声。

振荡器启动定时器

ENC28J60 内部有一个振荡器启动时钟 OST(OscillatorStart upTimer)，上电 7500 个时钟周期(300 μs)，OST 期满后内部的 PHY 方能正常工作。这时不能发送或者接收报文。上位机可通过检测 ENC28J60 内部 ESTAT 寄存器中的 CLKRDY 位的状态来决定是否可设置发送或接收报文。

需要注意的是，当 ENC28J60 上电复位或者从 PowerDown 模式下唤醒时，必须检测 ESTAT 寄存器中的 CLKRDY 是否置位。只有 CLKRDY 置位后才能发送、接收报文，访问相关寄存器。

时钟输出引脚

CLKOUT 引脚可为系统中的其他设备提供时钟源。上电后 CLKOUT 引脚保持低电平，复位结束后 OST 计数。OST 期满后，CLKOUT 输出频率为 6.25MHz 的时钟。

时钟输出功能通过 ECOCON 寄存器禁止、调整和使能。时钟输出可设置为 1、2、3、4、8 分频，上电后默认为 4 分频。ECOCON 寄存器配置改变以后，CLKOUT 引脚有 80~320ns 的延迟(保持低电平)，然后按照设定输出固定频率的时钟信号。

软件或者 RESET 引脚上的复位信号不会影响 ECOCON 寄存器的状态。PowerDown 模式也不会影响时钟的输出。当禁止时钟输出时，CLKOUT 引脚保持低电平。

变压器、终端和其他外部器件

为了实现以太网接口 ENC28J60，需要几个标准的外部器件：脉冲变压器、偏置电阻、储能电容和去耦电容。

差分输入引脚(TPIN+/TPIN-)，需要一个 1：1 变比的脉冲变压器来实现 10BASET。差分输出引脚(TPOUT+/TPOUT-)，需要一个变比为 1：1、带中心抽头的脉冲变压器。变压器需要有 2kV 或更高的隔离能力，防静电。对变压器的详细要求请参考芯片手册第 16 章“电气特性”。每个部分都需要通过 2 个 50Ω、精度为 1% 的电阻和 1 个 0.01 μF 的电容串联后接地。

笔者采用的是中山汉仁公司的集成以太网隔离变压器 RJ45 插座 HR911105A。

所有的供电引脚(VDD、VDDOSC、VDDPLL、VDDRX、VDDTX)必须接在外部的同一个 3.3V 电源上；同理，所有的地(VSS、VSSOSC、VSSPLL、VSSTX)必须接在同一个外部地上。每个供电引脚和地之间应当接 1 个 0.1μF 的陶瓷电容去耦（电容要尽可能接近供电引脚）。

驱动双绞线接口需要较大的电流，所以电源线应尽可能宽，与引脚的连接尽可能短，以降低电源线内阻的消耗。

输入输出电平

ENC28J60 是一个 3.3V 的 CMOS 器件，但它设计得非常容易统一到 5V 系统中去：SPI、CS、SCK、SI 输入和 RESET 引脚一样，都可承受 5V 电压。当 SPI 和中断输入与 3.3V 驱动的 CMOS 输出不兼容时，可能需要一个单向的电平转换器。74HCT08 (四与门)，74ACT125(四三态缓冲器)和许多具有 TTL 电平输入的 5V CMOS 缓冲器芯片都可以提供所需的电平转换。

◆ LED 配置

LEDA 和 LEDB 引脚在复位时支持极性自动检测。既可直接驱动 LED，又可灌电流驱动。复位时 ENC28J60 检测 LED 的连接，并按照 PHLCON 寄存器的默认设置来驱动。运行过程中的 LED 极性转换直到下一次系统复位后才能被检测到。LEDB 的连接比较特殊，在复位过程中检测它的连接，决定如何初始化 PHCON1 寄存器的 PDPXMD 位。如果 LEDB 直接驱动 LED，则 PHCON1.PDPXMD 位被清零，PHY 工作在半双工模式；如果 LEDB 吸收反向电流点亮 LED，则 PHCON1.PDPXMD 被置位，PHY 工作在全双工模式；如果 LEDB 没有连接，则 PHCON1.PDPXMD 复位后的值不确定。这时主控制器必须适当设置该位，以使 PHY 工作在所需的状态(半双工或全双工)。

7.71.4 软件设计

在本实验中，我们主要是学习 ENC28J60 的使用以及简单的 TCP，UDP 包的处理，使用到资源主要有 ENC28J60,LED 灯和串口 1，在程序运行以前，我们首先需要对这些接口进行初始化。

◆ GPIO 接口初始化

在本节我们使用的 GPIO 接口资源主要由 LED 灯，串口以及 ENC28J60 芯片所占用的 SPI1 接口。因此，我们在正式运行程序之前，需要对这些 GPIO 接口进行初始化，包括设置他的输入输出属性等，相关代码如下。

```
void GPIO_Configuration(void)
{
    // 使用到的资源时钟使能
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1|RCC_APB2Periph_GPIOA
                           |RCC_APB2Periph_GPIOC|RCC_APB2Periph_GPIOF, ENABLE);

    /* LED 灯初始化 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9;      // DS1--4
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOF, &GPIO_InitStructure);

    /* 串口 1 使用的 GPIO 管脚初始化 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;          // USART1 TX
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;     // 复用推挽输出
    GPIO_Init(GPIOA, &GPIO_InitStructure);            // A 端口

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;          // USART1 RX
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; // 复用开漏输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);             // A 端口

    /* SPI FLASH 的 CS 信号初始化 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;           // SPI FLASH CS
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    /* 由于 SPI FLASH 与 ENC28J60 使用了相同的 SPI 接口，所有置高 SPI FLASH 的 CS 信号，不使能 SPI FLASH */
    GPIO_SetBits(GPIOC, GPIO_Pin_4);                    // SPI CSI

    /* ENC28J60 的 INT 中断输入初始化 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
} ? end GPIO_Configuration ?
```

串口 1 初始化

在本例程中，串口 1 用于输入输出提示信号，波特率设置为 115200，无硬件流控模式，详细代码如下：

```
void USART1_Config(void) {
    /* USART1 configuration -----
     * USART configured as follow:
     *   - BaudRate = 115200 baud
     *   - Word Length = 8 Bits
     *   - One Stop Bit
     *   - No parity
     *   - Hardware flow control disabled (RTS and CTS signals)
     *   - Receive and transmit enabled
     */
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    /* Configure USART1 */
    USART_Init(USART1, &USART_InitStructure);
    /* Enable the USART1 */
    USART_Cmd(USART1, ENABLE);
} ? end USART1 Config ?
```

◆ SPI 接口初始化

在神舟 III 号开发板上，STM32F103ZET6 处理器通过 SPI1 接口与 ENC28J60 相连，同时 SPI1 还和板载的 SPI FLASH 连接，在前面的 GPIO 接口初始化部分，我们已经初始化分配给 SPI FLASH(W25X16)的 CS 信号为高，使之在访问 ENC28J60 网口时，不会和 W25X16 产生冲突。这 SPI1_Init()函数中，主要是对 SPI1 使用的 GPIO 口进行初始化，以及 SPI1 接口的模式，速度等参数配置，具体的程序如下：

```
void SPI1_Init(void)
{
    SPI_InitTypeDef SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    /*使能SPI1和GPIOA的时钟*/
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1|RCC_APB2Periph_GPIOA, ENABLE);

    /*SPI1 接口信号配置*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /*分配给ENC28J60芯片的SPI1_NSS信号配置*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA, GPIO_Pin_4);      //置高分配给ENC28J60的 SPI1_NSS 信号

    /* SPI1接口模式参数配置 */
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStructure.SPI NSS = SPI NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 7;
    SPI_Init(SPI1, &SPI_InitStructure);

    /*使能SPI1接口*/
    SPI_Cmd(SPI1, ENABLE);
} ? end SPI1_Init ?
```

◆ ENC28J60 初始化

前面我们描述了 GPIO 接口，串口和 SPI1 接口的初始化。通过 SPI1 可以正常访问 ENC28J60。Enc28j60Init 函数实现对 ENC28J60 芯片的初始化。包括 ENC28J60 的工作模式，MAC 地址设置等，主要是通过芯片的寄存器操作，通过 enc28j60write 函数来实现，在这里就不展开分析，有兴趣的朋友可以查看 enc28j60Init 函数与 enc28j60write 函数的具体实现。

◆ WebServer 实现

在完成了上述程序以后，神舟 III 号处理器网口实验相关的硬件接口已经初始化完成，为了支持网页访问，我们需要对一些 TCP,UTP 包进行解析，这主要是 simple_server() 函数实现的。相关代码如下：

Simple_server() 函数支持 ARP 和 ICMP 包的响应，这样我们在执行 ping 神舟 III 号 IP 地址时，神舟 III 号就能做出正确的响应。相关代码如下：

```
//判断是否有接收到有效的包
plen = enc28j60PacketReceive(BUFFER_SIZE, buf);
//如果收到有效的包，plen将为非0值。
if(plen==0)
{
    continue; //没有收到有效的包就退出重新检测
}
//当收到目的地址为本机IP的ARP包时，发出ARP相应包
if(eth_type_is_arp_and_my_ip(buf,plen))
{
    make_arp_answer_from_request(buf);
    continue;
}

//判断是否接收到目的地址为本机IP的合法的IP包
if(eth_type_is_ip_and_my_ip(buf,plen)==0)
{
    continue;
}
//如果收到ICMP包
if(buf[IP_PROTO_P]==IP_PROTO_ICMP_V && buf[ICMP_TYPE_P]==ICMP_TYPE_ECHOREQUEST_V)
{
    printf("\n\r收到主机[%d.%d.%d]发送的ICMP包",buf[ETH_ARP_SRC_IP_P],buf[ETH_ARP_SRC_IP_P+1],
          buf[ETH_ARP_SRC_IP_P+2],buf[ETH_ARP_SRC_IP_P+3]);
    make_echo_reply_from_request(buf, plen);
    continue;
}
```

TCP 包处理程序，如果收到 TCP 包，且端口号为 80，则按如下程序处理。

```
//如果收到TCP包，且端口为80
if (buf[IP_PROTO_P]==IP_PROTO_TCP_V&&buf[TCP_DST_PORT_H_P]==0&&buf[TCP_DST_PORT_L_P]==mywwwport)
{
    printf("\n\r神舟III号接收到TCP包，端口为80。");
    if (buf[TCP_FLAGS_P] & TCP_FLAGS_SYN_V)
    {
        printf("包类型为SYN");
        make_tcp_synack_from_syn(buf);
        continue;
    }
    if (buf[TCP_FLAGS_P] & TCP_FLAGS_ACK_V)
    {
        printf("包类型为ACK");
        init_len_info(buf); // init some data structures
        dat_p=get_tcp_data_pointer();
        if (dat_p==0)
        {
            if (buf[TCP_FLAGS_P] & TCP_FLAGS_FIN_V)
            {
                make_tcp_ack_from_any(buf); /*发送响应*/
            }
            // 发送一个没有数据的ACK响应，等待下一个包
            continue;
        }
        if (strncmp("GET ", (char *) &(buf[dat_p]), 4)!=0)
        {
            // 如果是Telnet方式登录，返回如下提示信息
            plen=fill_tcp_data_p(buf,0,PSTR("神舟III号\r\n\nHTTP/1.0 200 OK\r\nContent-Type: text/html\r\n"));
            goto ↓SENDTCP;
        }
        if (strncmp("/ ", (char *) &(buf[dat_p+4]), 2)==0)
        {
            //如果是通过网页方式登录，输出如下提示信息
            plen=fill_tcp_data_p(buf,plen,PSTR("<p>Usage: </p>"));
            plen=fill_tcp_data_p(buf,plen,baseurl);
            plen=fill_tcp_data_p(buf,plen,PSTR("password</p>"));
            goto ↓SENDTCP;
        }
    }
}
```

```
//分析网页控制的命令类型
cmd=analyse_get_url((char *) & (buf[dat_p+5]));
if (cmd== -1)
{
    plen=fill_tcp_data_p(buf, 0, PSTR("HTTP/1.0 401 Unauthorized\r\nContent-Type: text/html\r\n\r\n"));
    goto ↓SENDTCP;
}
//网页控制点亮LED灯DS1
if (cmd==1)
{
    LED1_ON();
    i=1;
}
//网页控制熄灭LED灯DS1
if (cmd==0)
{
    LED1_OFF();
    i=0;
}
//更新网页信息
plen=print_webpage(buf, (i));
SENDTCP:
make_tcp_ack_from_any(buf); // send ack for http get
make_tcp_ack_with_data(buf, plen); // send data
continue;
} ? end if buf[TCP_FLAGS_P]&TCP_... ?
} ? end if buf[IP_PROTO_P]==IP_P... ?
```

UDP 包处理程序，当接收到 UDP 包，且端口号为 1200，则按如下进行处理。

```
//UDP包, 监听1200端口的UDP包
if (buf[IP_PROTO_P]==IP_PROTO_UDP_V&&buf[UDP_DST_PORT_H_P]==4&&buf[UDP_DST_PORT_L_P]==0xb0)
{
    payloadlen= buf[UDP_LEN_H_P];
    payloadlen=payloadlen<<8;
    payloadlen=(payloadlen+buf[UDP_LEN_L_P])-UDP_HEADER_LEN;

    //ANSWER
    for (i1=0; i1<payloadlen; i1++)
    {
        buf1[i1]=buf[UDP_DATA_P+i1];
    }

    //make_udp_reply_from_request(buf,str,strlen(str),myudpport);
    make_udp_reply_from_request(buf,buf1,payloadlen,myudpport);
}
```

7.7.1.5 下载与测试

在 [神舟III号光盘\编译好的固件\18.网口程序实验](#)目录下的ENC28J60网口程序实验.hex文件即为本节我们分析的实验所编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟III号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 [如何使用JLINK V8 下载固件到神舟III号开发板](#) 小节进行操作。
如果使用串口下载固件，请按 [如何使用串口下载一个固件到神舟III号开发板](#) 小节进行操作。
如果在MDK开发环境中，下载编译好的固件或者在线调试，请按 [如何通过MDK编译和在线调试](#) 小节进行操作。

7.71.6 实验现象

将程序下载到神舟III号后，使用网线将神舟III号连到和同一个局域网中，并确认电脑的IP与神舟III号IP在同一个网段（注意：神舟III号确认的IP是192.168.1.15，上电时，神舟III号将通过串口1打印其IP和MAC地址信息，以串口打印的信息为准）。上电运行神舟III号。

假设设置电脑的IP为192.168.1.15，打开电脑的命令行，执行“ping 192.168.1.15”，其中192.168.1.15是神舟III号的IP，正常情况下，可以ping通，打印信息如下：

```
C:\> 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

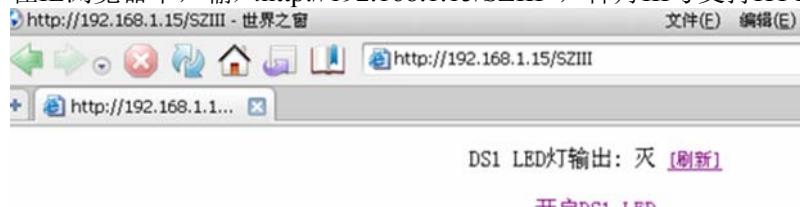
Pinging 192.168.1.15 with 32 bytes of data:

Reply from 192.168.1.15: bytes=32 time=26ms TTL=64
Reply from 192.168.1.15: bytes=32 time=17ms TTL=64
Reply from 192.168.1.15: bytes=32 time=17ms TTL=64
Reply from 192.168.1.15: bytes=32 time=17ms 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 = 17ms, Maximum = 26ms, Average = 19ms

C:\Documents and Settings\Administrator>
```

在IE浏览器中，输入http://192.168.1.15/SZIII，神舟III号支持HTTP访问，可以看到如下界面。



神舟III号STM32开发板 WEB 网页试验

点击上图中的开启DS1_LED可以控制神舟III号的LED灯DS1的亮灭。另外，在神舟III号的串口1会打印相关的提示信息，包括本机IP, MAC，以及提示接收和发送的包。

神舟III号网口测试程序
神舟III号MAC地址:0x32,0x12,0x35,0x11,0x1,0x51
IP地址:192.168.1.15
端口号:80

收到主机[192.168.1.103]发送的ARP包
神舟III号[192.168.1.15]发送ARP相应
收到主机[1.103.192.168]发送的ICMP包
神舟III号[192.168.1.15]发送ICMP包响应
收到主机[1.103.192.168]发送的ICMP包
神舟III号[192.168.1.15]发送ICMP包响应
收到主机[1.103.192.168]发送的ICMP包
神舟III号[192.168.1.15]发送ICMP包响应
收到主机[1.103.192.168]发送的ICMP包
神舟III号[192.168.1.15]发送ICMP包响应
收到主机[192.168.1.103]发送的ARP包
神舟III号[192.168.1.15]发送ARP相应
神舟III号接收到TCP包，端口为80。包类型为SYN
神舟III号[192.168.1.15]发送SYN包响应
神舟III号接收到TCP包、端口为80。包类型为ACK

7.72 UCOSII操作系统之单任务运行

7.72.3 UCOSII简单介绍

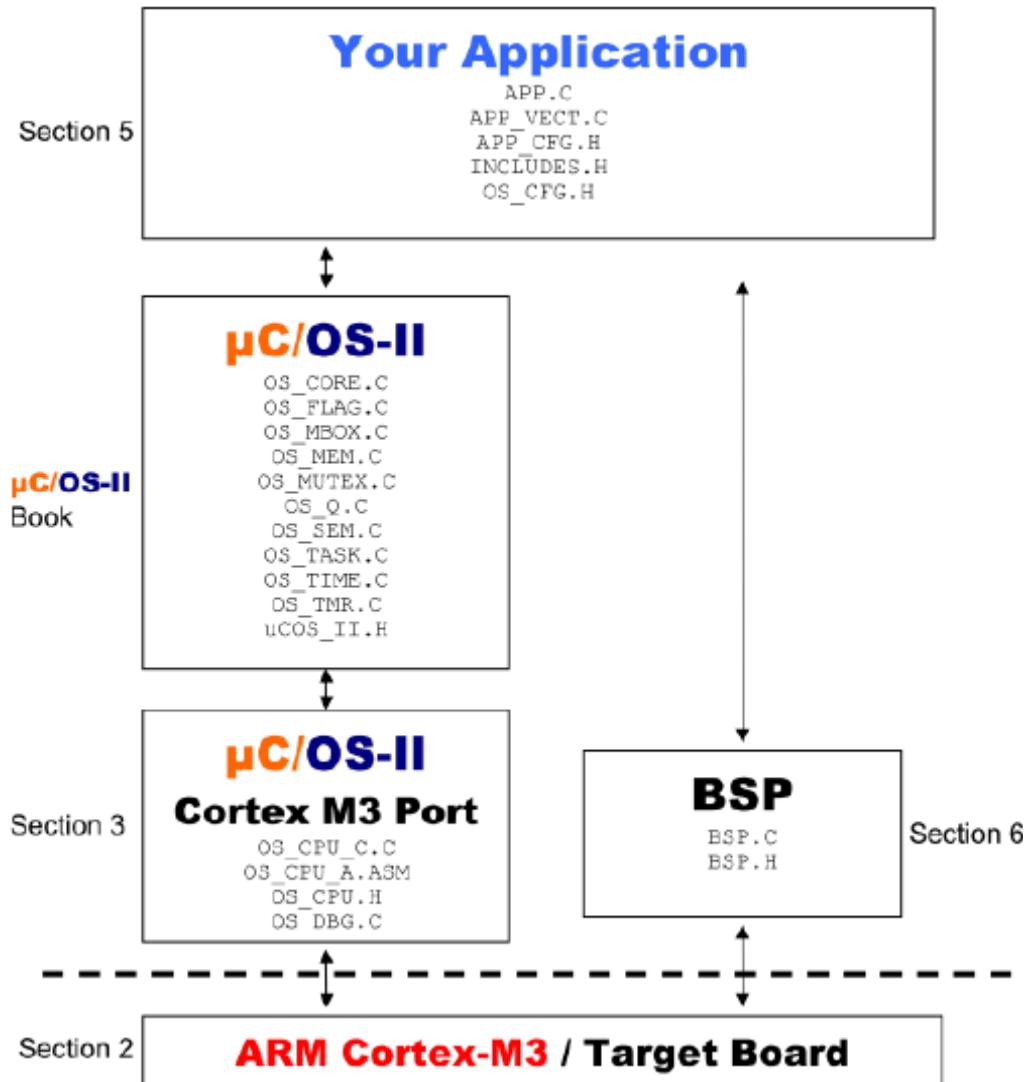
UCOSII的前身是UCOS，最早出自于1992 年美国嵌入式系统专家Jean J. Labrosse 在《嵌入式系统编程》杂志的5月和6月刊上刊登的文章连载，并把UCOS 的源码发布在该杂志的BBS 上。目前最新的版本：UCOSIII已经出来，但是现在使用最为广泛的还是UCOSII，本章我们主要针对UCOSII进行介绍。在学习本章之前，UOCS相关的知识比较多，我们实验也知识指点一下大家入门，详细了解建议大家先看看任哲老师的《嵌入式实时操作系统ucosII原理及应用》。

UCOSII是一个可以基于ROM运行的、可裁减的、抢占式、实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器，是和很多商业操作系统性能相当的实时操作系统(RTOS)。为了提供最好的移植性能，UCOSII最大程度上使用ANSI C语言进行开发，并且已经移植到近40多种处理器体系上，涵盖了从8位到64位各种CPU(包括DSP)。

UCOSII是专门为计算机的嵌入式应用设计的，绝大部分代码是用C语言编写的。CPU硬件相关部分是用汇编语言编写的、总量约200行的汇编语言部分被压缩到最低限度，为的是便于移植到任何一种其它的CPU 上。用户只要有标准的ANSI 的C交叉编译器，有汇编器、连接器等软件工具，就可以将UCOSII嵌入到开发的产品中。UCOSII具有执行效率高、占用空间小、实时性能优良和可扩展性强等特点， 最小内核可编译至 2KB 。UCOSII已经移植到了几乎所有知名的CPU 上。

UCOSII构思巧妙。结构简洁精练，可读性强，同时又具备了实时操作系统的全部功能，虽然它只是一个内核，但非常适合初次接触嵌入式实时操作系统的朋友，可以说是

麻雀虽小，五脏俱全。UCOSII（V2.91版本）体系结构如下图所示：



注意本章我们使用的是UCOSII的最新版本：V2.91版本，该版本UCOSII比早期的UCOSII（如V2.52）多了很多功能（比如多了软件定时器，支持任务数最大达到255个等），而且修正了很多已知BUG。不过，有两个文件：os_dbg_r.c和os_dbg.c，我们没有在上图列出。

7.72.4 UCOSII的学习方法

打算学习一个嵌入式操作系统，需要先研究一下UCOSII的学习方法，一方面权当学习C语言，另一方面UCOSII的代码比较小，比如2.52版本代码只有5500行左右，还是一个能接受的范围。

对于新手，入门选书是最重要的了。用了几天研究了一下参考书，和大家分享一下。

1. 嵌入式实时操作系统 μ C/OS-II(第2版) 邵贝贝 等译 北京航空航天大学出版社

应该说每一个学习UCOSII的人都应该知道这本书，也都应该看一下这本书，但是不建议作为入门书籍。这本书是UCOSII的作者原著的翻译本，必然是很详细，必然是权威，然而书中分析UCOSII内核原理是核心内容，应用则不多。想要快速上手的应当选用其他书籍，这本书应该当做手册。

2. 嵌入式实时操作系统 μ C/OS-II原理及应用(第2版) 任哲 北京航空航天大学出版社

这本书和第一本书比在讲述UCOSII原理的同时，配备了一些很简单的例子，可以在PC上调试代码。嵌入式专业技术论坛 (www.armjishu.com) 出品 第 833 页，共 900 页

码。这是一本可以反复阅读的书本，感觉不错，适合入门。当然书中有一些原理讲的不清楚，这个时候翻一下邵贝贝那本书就懂了。

3. UCOSII标准教程 杨宗德 人民邮电出版社

这本书我只翻了一下，看了看目录和前面一点内容。用的模拟环境是VC++6.0对于那些搞软件的朋友看这本书应该合适，书中的内核是2.8版本。

7.72.5 神舟官方团队点破操作系统的学习精妙之处

STM32神舟团队的工程师也对此给出了学习建议，例如51单片机我们要运行一个点LED灯亮灭的任务，还有一个要显示液晶屏两行字的任务；通常情况，可以用一个while循环来轮询这2个任务，但是如果任务多了怎么办呢？比如变成了200个任务，那么while循环将怎么管理呢？怎么样合理配置CPU资源呢？难道平均分配吗？

其实平均对每个任务分配CPU资源的方法是可行的，但是并不是最优的，比如比较重要的事情需要抢先办理，那么就逐渐有了优先级的概念；比如同时开启200个任务，有的任务执行了一下之后，需要过比较长的时间后再执行，这样的话，就可以使得下次while循环不用理睬这个任务，把这个空闲时间转让给其他需要执行的任务，这个需要等待的任务可以挂起，可以记录一下，什么时候开始执行这样的信息，这样就有了任务的其他状态，比如挂起的任务，比如预备好的任务进入等待队列等。

比如CPU同时运行200个任务，那么其中有个看门狗任务，主要负责监督CPU是否是正常的，如果出现异常，看门狗任务就会对CPU进行复位，像这样的任务重要性一定是最高的，因为CPU工作都已经不正常了，你还要它继续运行干什么，已经没有意义了，直接复位时最合适的解决办法。所以这样就形成了优先级的概念，有的任务的优先级比较高，如果这样的优先级高的任务出现了之后，就要先执行，我们可以人为来配置这些任务的优先级。

大家看到上面我们神舟工程师所描述的，可以慢慢的了解到，一个操作系统实际上是自然衍生出来的结果，一切都是必要的，而不是枯燥无味或者故意加上去的。如果从这样的切入点思路进去学习操作系统，我想是事半功倍的。多一些思考，比多看一些书，会更加的务实，功力提高也就越快。

7.72.6 实验原理

通过 UCOSII 操作系统创建一个简单任务，这个任务的工作就是实现 LED 灯闪烁操作。

7.72.7 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 1 个任务，使得 LED 灯进行闪烁，LED 灯的原理图同 LED 流水灯一样。

7.72.8 软件设计

```

int main(void)
{
    /* 配置神舟III号LED灯使用的GPIO管脚模式 */
    RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE);           /*使能LED灯使用的GPIO时钟*/
    GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_LED, &GPIO_InitStructure);             /*神州III号使用的LED灯相关的GPIO口初始化*/
    GPIO_SetBits(GPIO_LED,DS1_PIN|DS2_PIN|DS3_PIN|DS4_PIN); /*关闭所有的LED指示灯*/

    /***** UCOSII 操作系统初始化开始 { *****/
    OSInit();      /* UCOSII 操作系统初始化 */

    OSTaskCreate(Task_Shenzhou_LED, (void *)0,&start_task_stk[START_TASK_STK_SIZE-1], START_TASK_PRIO);

    OSStart();     /* 启动UCOSII操作系统 */

    /***** UCOSII 操作系统初始化结束 } *****/
}

```

代码分析：

1. 首先开始初始化 LED 灯，这段代码之前有分析过，不记得的请看一下前面的 LED 流水灯的文档

```

/* 配置神舟I号LED灯使用的GPIO管脚模式 */
RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE);           /*使能LED灯使用的GPIO时钟*/
GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIO_LED, &GPIO_InitStructure);             /*神州I号使用的LED灯相关的GPIO口初始化*/
GPIO_SetBits(GPIO_LED,DS1_PIN|DS2_PIN); /* 熄灭所有的LED指示灯 */

```

2. 开始初始化操作系统，用 OSInit，初始化 UCOSII 的所有变量和数据结构的所有变量和数据结构的所有变量和数据结构，然后通过调用 OSTaskCreate 函数创建我们的任务。

```
OSInit(); /* UCOSII 操作系统初始化 */
```

3. 初始化操作系统之后，开始创建一个任务

```

/* UCOSII创建一个任务 */
OSTaskCreate(Task_Shenzhou_LED, (void *)0,&start_task_stk[START_TASK_STK_SIZE-1], START_TASK_PRIO);
1) 用 OSTaskCreate() 函数创建一个任务

```

如果想让 UCOSII 管理用户的任务，必须先建立任务。UCOSII 提供了建立任务的函数：OSTaskCreat，我们一般用 OSTaskCreat 函数来创建任务，该函数原型为：

```
OSTaskCreate(void(*task)(void*pd),void*pdata,OS_STK*ptos,INTU prio)
```

该函数包括 4 个参数：

参数 1：task---是指向任务代码的指针，创建任务之后，就会运行 task 所指向的一个函数，我们可以在这个函数里填写这个任务要运行的内容。

参数 2：pdata---是任务开始执行时，传递给任务的参数的指针；

参数 3：ptos---是分配给任务的堆栈的栈顶指针；每个任务都有自己的堆栈，堆栈必须申明为 OS_STK 类型，并且由连续的内存空间组成。可以静态分配堆栈空间，也可以动态分配堆栈空间。

参数 4：prio---是分配给任务的优先级。

所谓的任务，其实就是一个死循环函数，该函数实现一定的功能，一个工程可以有很多这样的任务，UCOSII 对这些任务进行调度管理，让这些任务可以并发工作（注意不是同时工作！！，并发只是各任务轮流占用 CPU，而不是同时占用，任何时候还是只有 1 个任务能够占用 CPU），这就是 UCOSII 最基本的功能。

UCOS 是怎样实现多任务并发工作的呢？当一个任务 A 正在执行的时候，如果他释放了 cpu 控制权，先对任务 A 进行现场保护，然后从任务就绪表中查找其他就绪任务去执行，等到任务 A 的等待时间到了，它可能重新获得 cpu 控制权，这个时候恢复任务 A 的现场，从而继续执行任务 A，这样看起来就好像两个任务同时执行了。实际上，任何时候，只有一个任务可以获得 cpu 控制权。这个过程很负责，场景也多样，这里只是举个简单的例子说明。

2) Task_Shenzhou_LED () 函数里面实现 LED 灯的闪烁，操作系统启动的时候，就会运行一个任务，这个任务主要完成的工作就是 Task_Shenzhou_LED () 函数里面的内容，具体代码实现如下：

```
void Task_Shenzhou_LED(void *arg)
{
    (void)arg;                                // 'arg' 并没有用到，防止编译器提示警告
    while (1)
    {
        GPIO_ResetBits(GPIO_LED,DS1_PIN); //点亮LED1
        Delay(0x2FFFFF);                //延时
        GPIO_SetBits(GPIO_LED,DS1_PIN); //熄灭LED1
        Delay(0x2FFFFF);                //延时
    }
}
```

- 2) 分配一个任务的参数，这里我们设置的是 “(void *)0”，表示这个任务没有参数需要被传入
- 3) 分配给任务一个堆栈，每个任务都有自己的堆栈，堆栈必须申明为 OS_STK 类型，因为 STM32 是 32 位位宽的，这里 OS_STK 应该为 32 位数据类型，并且由连续的内存空间组成；在代码中我们使用 `typedef unsigned int OS_STK;` 无符号整型就是表示 32 位数据类型，使得 OS_STK 就成为了 32 位的；堆栈空间可以静态分配，也可以动态分配。

这里的堆栈大小确定主要是看你任务的临时变量(局部变量大小)，另外还需要考虑调用其他函数导致的堆栈增加。一般情况下在资源足够时，可以先设置一个较大的值，然后慢慢缩减和优化，如果不能正常运行了，说明你的堆栈就分配少了，这个需要靠经验值或者靠其他方式来进行检测确定。

我们这里暂时分配 100 个 OS_STK 单元的堆栈空间，应该足够使用了。

```
*****设置栈大小（单位为 os_stk ） *****
#define START_TASK_STK_SIZE 100

static OS_STK start_task_stk[START_TASK_STK_SIZE];           //定义栈
```

那么就为`&start_task_stk[START_TASK_STK_SIZE-1]`这个堆栈的栈顶的地址了。

- 4) 就是确定这个任务的优先级了，任务优先级，这个概念比较好理解，ucos 中，每个任务都有唯一的一个优先级。优先级是任务的唯一标识。在 UCOSII 中，使用 CPU 的时候，优先级高（数值小）的任务比优先级低的任务具有优先使用权，即任务就绪表中总是优先级最高的任务获得 CPU 使用权，只有高优先级的任务让出 CPU 使用权（比如延时）时，低优先级的任务才能获得 CPU 使用权。UCOSII 不支持多个任务优先级相同，也就是每个任务的优先级必须不一样。

比如：彩屏触摸实验，在显示彩屏的时候，我们还希望实现触摸，如果是 1 个死循环（一个任务），那么很可能在显示彩屏的时候，实现触摸出现停顿（感觉不灵敏），这主要是刷彩屏的时候占用太长时间，导致实现触摸的时候不能及时采集液晶屏上的触摸数据。而如果用 UCOSII 来处理，那么我们可以分 2 个任务，刷彩屏是一个任务（优先级高），采集触摸数据是另一个任务（优先级低），这两个任务都分时间片来执行，可能 1 秒钟时间各执行了几十次，这样的速度，人是感觉不到的；由于刷彩屏任务的优先级高于采集触摸数据任务，刷彩屏任务可以采集触摸数据任务，从而及时给保证彩屏一直显示很完美，而且触摸也能恰到好处的采集到，这就是 UCOSII 带来的好处。

UCOSII 早期版本只支持 64 个任务，但是从 2.80 版本开始，支持任务数提高到 255 个，不过对我们来说一般 64 个任务都是足够多了，一般很难用到这么多个任务。UCOSII 一般默认只占用了最低 2 个优先级，分别用于空闲任务（倒数第一）和统计任务（倒数第二），这个是它一启动就默认存

在的 2 个任务，不需要我们的干预就存在的，所以剩下给我们使用的任务最多可达 $255-2=253$ 个。

在这个例程中，我们设置的任务优先级为 5，具体代码如下：

```
#define START_TASK_PRIO      5
```

4. 开始正式启动 UCOSII 操作系统，可以看到，我们在创建任务 Task_Shenzhou_LED()之前首先调用 ucossi 初始化函数 OSInit()，该函数的作用是初始化 ucos 的所有变量和数据结构，该函数必须在调用其他任何 ucos 函数之前调用。在 Task_Shenzhou_LED()创建之后，我们调用 ucos 多任务启动函数 OSStart()，调用这个函数之后，任务才真正开始运行。

7.72.9 下载与测试

在代码编译成功之后，我们通过下载代码到 STM32 神舟 III 号开发板上，可以看到 LED1 以固定的频率闪烁，说明任务 Task_Shenzhou_LED()已经正常运行了！第一个操作系统任务大功告成

7.73 UCOSII操作系统多任务运行

7.73.3 SYSTICK时钟介绍

运行多个任务的时候，需要用到时间片切换，也就是说，一个时间片用完之后，就切换到另外一个任务执行一个时间片的时间，那么这里时间片该如何确定呢？在UCOSII中，我们使用了systick定时器和UCOSII共用。

首先我们简单介绍下ucos的时钟，ucos运行需要一个系统时钟节拍（类似“心跳”），而这个节拍是固定的（由OS_TICKS_PER_SEC设置），比如10ms（设置：OS_TICKS_PER_SEC = 100即可），在STM32下面，一般是由systick来提供这个节拍，也就是systick要设置为10ms中断一次，为UCOSII提供时钟节拍，而且这个时钟一般是不能被打断的（否则就不准了）。

Cortex-M3内核的处理器，内部包含了一个SysTick定时器，SysTick 是一个24 位的倒计数定时器，当计到0 时，将从RELOAD 寄存器中自动重装载定时初值。只要不把它在SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick在《STM32的参考手册》（V10.0版本）里面介绍的很简单，其详细介绍，请参阅《Cortex-M3权威指南》第133页。在以下的代码中我们就利用STM32的内部SysTick来实现延时的，这样既不占用中断，也不占用系统定时器。

因为在UCOSII下systick一旦设定之后就不能再被随意更改，对延时的设计来说，如果利用systick来做延时微秒或者毫秒的延时时，就必须想办法进行整合和计算了，要凑足我们想延时的单位数值；以延时us为例，比如希望延时100个us，先计算好这段延时需要等待的systick计数次数，这里为100/9（假设系统时钟为72Mhz，那么systick是24位的，用24位无法表示72M的主频，计算了一下，每systick增加1相当于主频次数运行了8次，这样就相当于每systick增加1经历的时间是 $=8*(1/72MHZ) = 1/9us$ ），然后我们就一直统计systick的计数变化，直到这个值变化了100/9，一旦检测到变化达到或者超过这个值，就说明延时100us时间到了。

7.73.4 实验原理

通过 UCOSII 操作系统创建 3 个简单任务，每个任务分别让不同的 LED 灯以不同的频率闪烁。

7.73.5 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 3 个任务，使得 3 个不同的 LED 灯进行闪烁，LED 灯的原理图同 LED 流水灯一样。

7.73.6 软件设计

先看下主函数：

```
int main(void)
{
    SystemInit(); /* 配置系统时钟为72M */

    SysTick_Config(SystemFrequency/OS_TICKS_PER_SEC); /* 初始化并使能SysTick定时器 */

    /* 配置神舟I号LED灯使用的GPIO管脚模式 */
    RCC_APB2PeriphClockCmd(RCC_GPIO_LED, ENABLE); /*使能LED灯使用的GPIO时钟*/
    GPIO_InitStructure.GPIO_Pin = DS1_PIN|DS2_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_LED, &GPIO_InitStructure); /*神州I号使用的LED灯相关的GPIO口初始化*/
    GPIO_SetBits(GPIO_LED, DS1_PIN|DS2_PIN); /*关闭所有的LED指示灯*/

    /***** UCOSII 操作系统初始化开始 { *****
    OSInit(); /* UCOSII 操作系统初始化 */

    /* UCOSII创建一个任务 */
    OSTaskCreate(Task_Shenzhou, (void *)0, &start_task_stk[START_TASK_STK_SIZE-1], START_TASK_PRIO);

    OSStart(); /* 启动UCOSII操作系统 */
    /***** UCOSII 操作系统初始化结束 } *****/
}
```

代码分析：

1. 首先配置系统时钟到 72M

```
int main(void)
{
    SystemInit(); /* 配置系统时钟为72M */
```

2. 初始化并使能 SysTick 定时器

```
SysTick_Config(SystemFrequency/OS_TICKS_PER_SEC); /* 初始化并使能SysTick定时器 */
```

3. 配置神舟 III 号 LED 灯使用的 GPIO 管脚模式

4. OSInit(); /* UCOSII 操作系统初始化 */.

5. UCOSII 创建一个任务

```
OSTaskCreate(Task_Shenzhou,(void*)0,&start_task_stk[START_TASK_STK_SIZE-1],
START_TASK_PRIO);
```

```
void Task_Shenzhou(void *p_arg)
{
    (void)p_arg; // 'p_arg' 并没有用到，防止编译器提示警告

    OSTaskCreate(Task_1_Shenzhou, (void *)0, &task_1_stk[TASK_1_STK_SIZE-1], TASK_1_PRIO); //创建任务1

    while (1)
    {
        GPIO_ResetBits(GPIO_LED, DS1_PIN); //点亮LED1
        OSTimeDlyHMSM(0, 0, 0, 100);
        GPIO_SetBits(GPIO_LED, DS1_PIN); //熄灭LED1
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}
```

1) 在 Task_Shenzhou () 函数中又创建了 1 任务 Task_1_Shenzhou ()

2) Task_Shenzhou () 创建任务之后，一直在 while 死循环中执行 LED1 的点灯闪烁操作

3) 任务 Task_1_Shenzhou (), 一直在 while 死循环中执行 LED3 的点灯闪烁操作

```
void Task_1_Shenzhou(void *arg)
{
    (void)arg;                                // 'arg' 并没有用到，防止编译器提示警告
    while (1)
    {
        GPIO_ResetBits(GPIO_LED, DS3_PIN); //点亮LED3
        OSTimeDlyHMSM(0, 0, 0, 300);          //延时
        GPIO_SetBits(GPIO_LED, DS3_PIN); //熄灭LED3
        OSTimeDlyHMSM(0, 0, 0, 300);          //延时
    }
}
```

6. 启动 UCOSII 操作系统

7.73.7 下载与测试

在代码编译成功之后，我们通过下载代码到 STM32 神舟 III 号开发板上，可以看到 LED1 和 LED3 都是以不同的频率在闪烁，说明 3 个任务在分别控制各自的 LED 灯，到这里多任务操作系统任务大功告成。

7.74 uCOS_UCGUI_DEMO实验

综合实验，UCOSII 操作系统+GUI 界面

7.75 1602液晶屏模块

7.75.3 7.80.1 1602字符型液晶屏简介

1) 1602 液晶屏功能、显示方式等

液晶显示屏的英文名是LiquidCrystalDisplay，简称LCD。在日常生活中，我们对液晶显示器并不陌生。液晶显示模块已作为很多电子产品的通用器件，如在计算器、万用表、电子表及很多家用电子产品中都可以看到。液晶显示屏具有体积小、重量轻、功耗低等优点，所以LCD日渐成为各种便携式电子产品的理想显示器。下图为1602字符型液晶实物图：

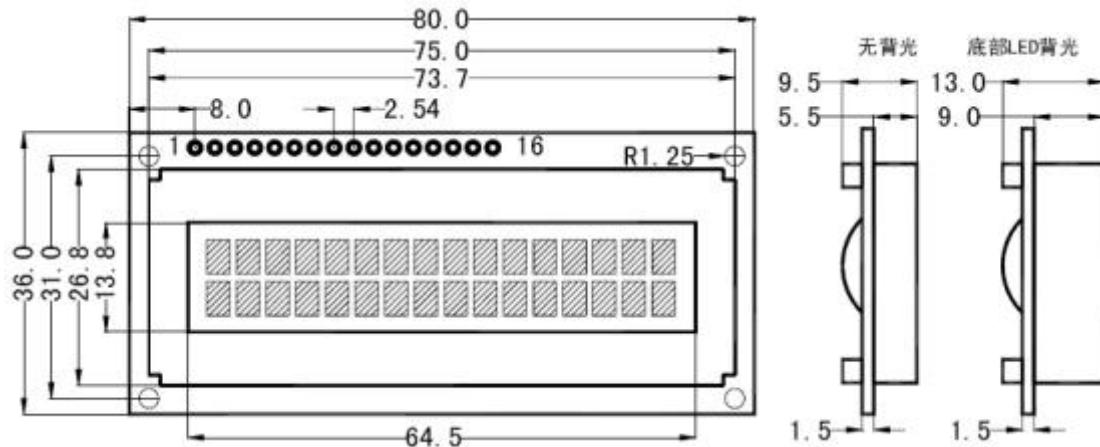


根据 LCD 的显示内容划分，可以分为段式 LCD、字符式 LCD 和点阵式 LCD 3 种。其中，字符式 LCD 以其廉价、显示内容丰富、美观、使用方便等特点，成为 LED 数码管的理想替代品。1602 液晶屏就是字符式 LCD 的一种。字符型液晶显示是一种专门用于显示字母、数字、符号等字符式嵌入式专业技术论坛（www.armjishu.com）出品

LCD

2) 1602LCD 的分类

1602LCD 分为带背光和不带背光两种，基控制器大部分为 HD44780，带背光的比不带背光的厚，是否带背光在应用中并无差别，两者尺寸差别如下图所示：



3) 1602 液晶屏的特性与参数

1602 液晶屏的特性：

- +5V 电压，对比度可调
- 内含复位电路
- 提供各种控制命令，如：清屏、字符闪烁、光标闪烁、移位等多种功能
- 有 80 字节显示数据存储器 DDRAM
- 内建有 160 个 5x7 的字符发生器 CGROM
- 8 个可由用户自定义的 5x7 的字符发生器 CGRAM

1602 型 LCD 的主要技术参数如下：

- 显示容量：16X2 个字符
- 芯片工作电压：4.5V~5.5V
- 工作电流：2.0mA(5.0V)
- 模块最佳工作电压：5.0V
- 字符尺寸：2.95X4.35 (WXH) mm

4) 液晶常用的三种连接方式

常用液晶模块连接方式及其适用范围：

(1) 金属插脚

金属引脚可直接焊接在 PCB (印刷线路板) 上，连接可靠，抗震动强，脚间距受限制 适用于音响产品，电表等。

(2) 斑马纸(热封)

柔软性连接，组装不方便。用于薄型产品的连接。适用于计数器、寻呼机、电子记事簿等。

(3) 导电橡胶

成本较低，组装方便，是较多采用的连接方式成本较低。适用于手表，游戏机，时钟，电话等。

我们的 1602LCD 属于第一种连接方式，直接通过金属引脚连接

7.75.4 1602液晶屏显示的基本原理

液晶显示的原理是利用液晶的物理特性，通过电压对显示区域进行控制，只要输入所需的控制电压，就可以显示出字符。线段的显示：点阵图形式液晶由 $M \times N$ 个显示单元组成，假设 LCD 显示屏有 64 行，每行有 128 列，每 8 列对应 1 字节的 8 位，即每行由 16 字节，共 $16 \times 8 = 128$ 个点组成，屏上 64×16 个显示单元与显示 RAM 区 1024 字节相对应，每一字节的内容和显示屏上相应位置的亮暗对应。例如屏的第一行的亮暗由 RAM 区的 000H——00FH 的 16 字节的内容决定，当 (000H)=FFH 时，则屏幕的左上角显示一条短亮线，长度为 8 个点；当 (3FFH)=FFH 时，则屏幕的右下角显示一条短亮线；当 (000H)=FFH, (001H)=00H, (002H)=00H, …… (00EH)=00H, (00FH)=00H 时，则在屏幕的顶部显示一条由 8 段亮线和 8 条暗线组成的虚线。要 LCD 显示字符“A”，则只需将 A 的 ASCII 码 41H 存入 DDRAM，控制线路就会通过 HD44780 的另一个部件字符产生器 (CGROM) 将 A 的字型点阵数据找出来显示在 LCD 上。这就是 LCD 显示的基本原理。

字符的显示：

用 LCD 显示一个字符时比较复杂，因为一个字符由 6×8 或 8×8 点阵组成，既要找到和显示屏上某几个位置对应的显示 RAM 区的 8 字节，还要使每字节的不同位为“1”，其它的为“0”，为“1”的点亮，为“0”的不亮。这样一来就组成某个字符。但由于内带字符发生器的控制器来说，显示字符就比较简单了，可以让控制器工作在文本方式，根据在 LCD 上开始显示的行列号及每行的列数找出显示 RAM 对应的地址，设立光标，在此送上该字符对应的代码即可。

汉字的显示（点阵 LCD 实现）：

汉字的显示一般采用图形的方式，事先从微机中提取要显示的汉字的点阵码（一般用字模提取软件），每个汉字占 32B，分左右两半，各占 16B，左边为 1、3、5……右边为 2、4、6……根据在 LCD 上开始显示的行列号及每行的列数可找出显示 RAM 对应的地址，设立光标，送上要显示的汉字的第一字节，光标位置加 1，送第二个字节，换行按列对齐，送第三个字节……直到 32B 显示完就可以 LCD 上得到一个完整汉字。

7.75.5 如何控制1602液晶屏（寄存器的介绍）

如何才能让液晶屏显示字符呢？用三个过程即可完成：

- 告诉液晶屏你应该双行或者单行显示，在哪里显示，显示的是 5*7 的字符还是 5*10 的字符等等
- 让液晶屏干什么的也就是液晶屏的命令。我们称这一过程为写命令。
- 告诉液晶屏显示什么字符，这一过程称为写数据。

要完成上面的操作，我们首先要先了解 LCD1602 的各个寄存器，并灵活的使用它们。

1) 1602 LCD 各寄存器介绍

控制器主要由指令寄存器 I R、数据寄存器 D R、忙识别位 B F、地址计数器 A C、
D R A M、C G R O M、C G R A M及时序发生电路组成。下面我们详细的来了解它们。

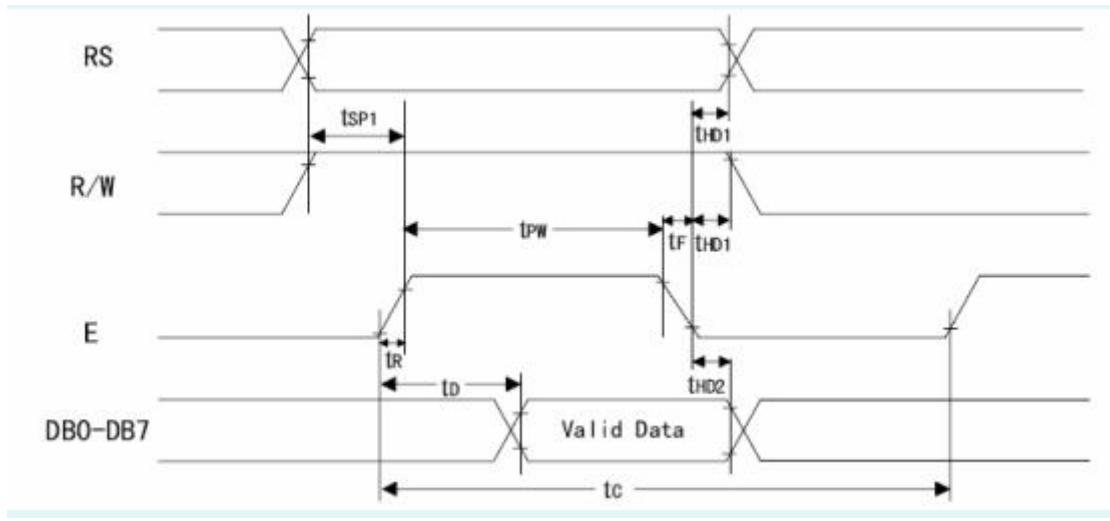
- 指令寄存器 (I R) 和数据寄存器 (D R)

1602 液晶屏内部具有两个 8 位寄存器：指令寄存器 (I R) 和数据寄存器 (DR)，用户可以通过 R S 和 R / W 输入信号组合选择指定的寄存器，进行相应操作，它们组合选择方式如下：

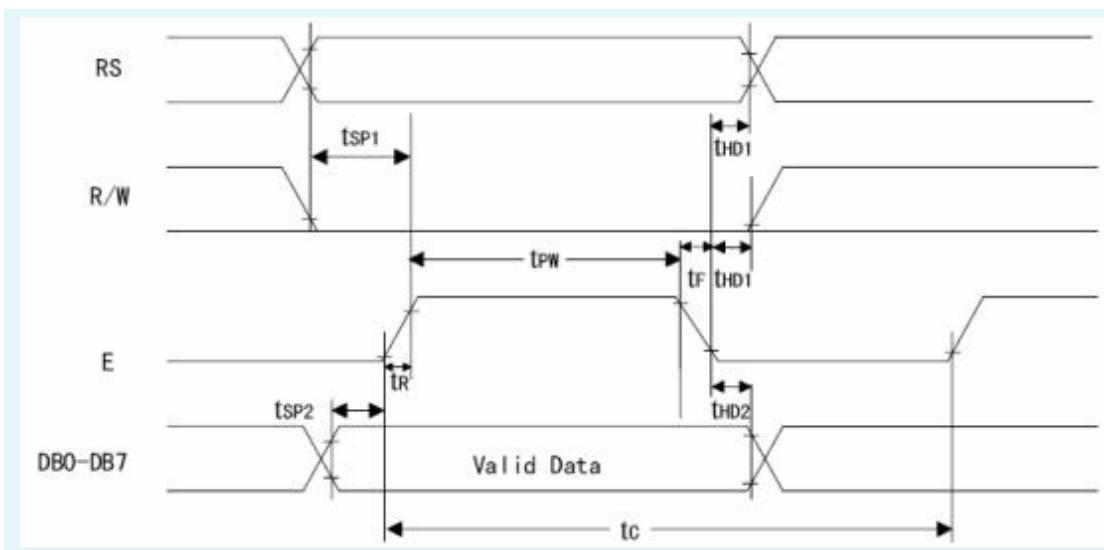
R S 和 R / W 输入信号组合

| E | R S | R / W | 说明 |
|-------|-----|-------|---|
| 1 | 0 | 0 | 将 D B 0 —— D B 7 的指令代码写入到指令寄存器中 |
| 1 → 0 | 0 | 1 | 分别将状态标志 B F 和地址计数器 (A C) 内容独到 D B 7 和 D B 6 — |
| 1 | 1 | 0 | 将 D B 0 —— D B 7 的数据写入到数据寄存器中，模块内部操作自动将数据写到 D D R A M 或 C G R A M 中 |
| 1 → 0 | 1 | 1 | 将数据寄存器内的数据读到 D B 0 —— D B 7，模块的内部操作自动将 D D R A M 或 C G R A M 中的数据送入数据寄存器中 |

R S 、 R / W 工作时序图如下图所示：



写操作时序



读操作时序

● 忙标识位 B F

忙标识为 $B\ F = 1$ 时，表明模块正在进行内部操作，此时不接收任何外部指令和数据，当 $R\ S = 0$ 、 $R\ /W = 1$ 且为高电平时， $B\ F$ 输出到 $D\ B\ 7$ 。每次最好先进行状态字检测，只有在确认 $B\ F = 0$ 之后，M P U 才能访问模块。

● 地址计数器（A C）

A C 地址计数器是 D D R A M 或 C G R A M 的地址指针（D D R A M 或 C G R A M 寄存器我们下面会有介绍）。随着 I R 中指令码的写入，指令码中携带的地址信息自动送入 A C 中，并做出 A C 作为 D D R A M 的地址指针还是 C G R A M 的地址指针的选择。

A C 具有自动加 1 或自动减 1 的功能。当 D R 与 D D R A M 或 C G R A M 之间完成一次数据传送后，A C 自动会加 1 或减 1。在 $R\ S = 0$ 、 $R\ /W = 1$ 且 E 为高电平时，A C 的内容送到 D B 6 —— D B 0。地址计数器 A C 如下图所示。

地址计数器 A C

| A C 高 3 位 | | | A C 低 4 位 | | | |
|-----------|-------|-------|-----------|-------|-------|-------|
| A C 6 | A C 5 | A C 4 | A C 3 | A C 2 | A C 1 | A C 0 |

● 显示数据寄存器（D D R A M）

D D R A M 存储显示字符的字符码，其容量的大小决定模块最多可显示的字符数目。控制器内部有 80 字节的 D D R A M 缓冲区，D D R A M 地址与 L C D 显示位置的对应关系如下图所示。

| | 显示位置 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | 40 |
|-------|------|-----|-----|-----|-----|-----|-----|-----|-------|-----|
| DDRAM | 第一行 | 00H | 01H | 02H | 03H | 04H | 05H | 06H | | 27H |
| 地 址 | 第二行 | 40H | 41H | 42H | 43H | 44H | 45H | 46H | | 67H |

也就是说想要在 LCD1602 屏幕的第一行第一列显示一个“A”字，就要向 DDRAM 的 00H 地址写入“A”字的代码就行了。但具体的写入是要按 LCD 模块的指令格式来进行的，后面我会说到的。那么一行可有 40 个地址呀？是的，在 1602 中我们就用前 16 个就行了。第二行也一样用前 16 个地址。对应如下：

| | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 00H | 01H | 02H | 03H | 04H | 05H | 06H | 07H | 08H | 09H | 0AH | 0BH | 0CH | 0DH | 0EH | 0FH |
| 40H | 41H | 42H | 43H | 44H | 45H | 46H | 47H | 48H | 49H | 4AH | 4BH | 4CH | 4DH | 4EH | 4FH |

字的字模：

01110 ○■■■○

10001 ■○○○■

10001 ■○○○■

10001 ■○○○■

11111 ■■■■■

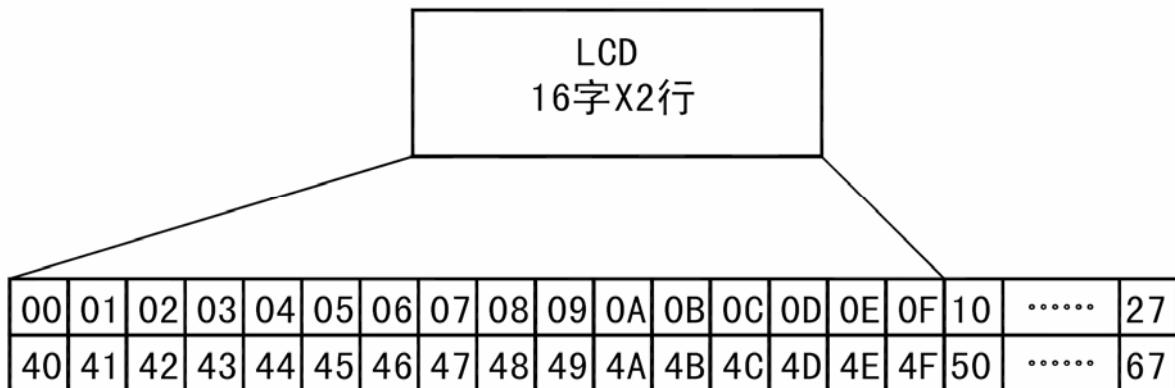
10001 ■○○○■

10001 ■○○○■

上图左边的数据就是字模数据，右边就是将左边数据用“○”代表 0，用“■”代表 1。看出是个“A”字了吗？在文本文件中“A”字的代码是 41H，PC 收到 41H 的代码后就去字模文件中将代表 A 字的这一组数据送到显卡去点亮屏幕上相应的点，你就看到“A”这个字了

RAM 地址映射图：

液晶显示模块是一个慢显示器件，所以在执行每条指令之前一定要确认模块的忙标志为低电平，表示不忙，否则此指令失效。要显示字符时要先输入显示字符地址，也就是告诉模块在，哪里显示字符，下图是 1602 的内部显示地址。



我们知道文本文件中每一个字符都是用一个字节的代码记录的。一个汉字是用两个字节的代码记录。在 PC 上我们只要打开文本文件就能在屏幕上看到对应的字符是因为在操作系统里和 BIOS 里都固化有字符字模。什么是字模？就代表了是在点阵屏幕上点亮和熄灭的信息数据。例如“A”

例如第二行第一个字符的地址是 40H，那么是否直接写入 40H 就可以将光标定位在第二行第一个字符的位置呢？事实上我们往 DDRAM 里的 00H 地址处送一个数据，譬如 0x31(数字 A 的代码)并不能显示 A 出来。这是一个令初学者很容易出错的地方，原因就是如果你要想在 DDRAM 的 00H 地址处显示数据，则必须将 00H 加上 80H，即 80H，若要在 DDRAM 的 01H 处显示数据，则必须将 01H 加上 80H 即 81H。依次类推。大家看一下控制指令的的 8 条：DDRAM 地址的设定，即可以明白是怎么样的一回事了。

● CGROM 和 CGRAM

要在 LCD1602 屏幕的第一行第一列显示一个"A"字，就要向 DDRAM 的 00H（实际地址为 80H，可参考上一段内容）地址写入“A”字的代码 41H 就行了，可 41H 这一个字节的代码如何才能让 LCD 模块在屏幕的阵点上显示“A”字呢？同样，在 LCD 模块上也固化了字模存储器，这就是 CGROM 和 CGRAM。1602 液晶模块内部已经存储了 160 个不同的点阵字符图形，存于字符产生器 CGROM(Character Generator ROM)中，另外还有 8 个允许用户自定义的字符产生 RAM，称为 CGRAM(Character Generator RAM)。

在 C G R O M 中，模块已经为 8 位二进制数的形式，生成了 5×8 点阵的字符字模组字符字模（一个字符对应一组字模）。字符字模是与显示字符点阵对应的 8×8 矩阵位图数据（与点阵行相对应的矩阵的高三位为“0”），同时每一组字符字模都有一个由其在 C G R O M 中存放地址的高 8 位数据组成的字符码对应。

在 C G R A M 中，用户可以生成自定义图形字符的字模组，可以生成 5×8 点阵的字符字模 8 组，相对应的字符码从 C G R O M 的 0 0 H—0 F F H 范围内选择，下图为 CGROM 和 CGRAM 与字符的对应关系。

| CGROM 和 CGRAM 中字符代码与字符图形对应关系 | | | | | | | | | | | | | | | |
|------------------------------|--------------|------|------|------|------|------|------|------|------|------|------|------|------|--|--|
| 高 位 低 位 | 0000 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 | | |
| xxxx0000 | CGRAM (1) | 0 | a | P | \ | p | | - | 夕 | 三 | a | P | | | |
| xxxx0001 | (2) | ! | 1 | A | Q | a | q | 口 | ア | チ | ム | ä | q | | |
| xxxx0010 | (3) | " | 2 | B | R | b | r | フ | イ | 川 | メ | ß | ø | | |
| xxxx0011 | (4) | # | 3 | C | S | c | s | 』 | ウ | ラ | モ | € | ∞ | | |
| xxxx0100 | (5) | \$ | 4 | D | T | d | t | \ | エ | ト | セ | µ | ¤ | | |
| xxxx0101 | (6) | % | 5 | E | U | e | u | ロ | オ | ナ | ユ | B | 0 | | |
| xxxx0110 | (7) | & | 6 | F | V | f | v | テ | カ | ニ | ヨ | P | Σ | | |
| xxxx0111 | (8) | > | 7 | G | W | g | w | ア | キ | ヌ | ラ | g | π | | |
| xxxx1000 | (1) | (| 8 | H | X | h | x | イ | ク | ネ | リ | ƒ | X | | |
| xxxx1001 | (2) |) | 9 | I | Y | i | y | ウ | ケ | 』 | ル | -1 | y | | |
| xxxx1010 | (3) | * | : | J | Z | j | z | エ | コ | リ | レ | j | 千 | | |
| xxxx1011 | (4) | + | : | K | [| k | { | オ | サ | ヒ | ロ | x | 万 | | |
| xxxx1100 | (5) | フ | < | L | ¥ | l | | セ | シ | フ | ワ | € | >All | | |
| xxxx1101 | (6) | - | = | M |] | m |) | ユ | ス | ヘ | ソ | æ | + | | |
| xxxx1110 | (7) | . | > | N | - | n | - | ヨ | セ | ホ | ハ | ñ | | | |
| xxxx1111 | (8) | / | ? | O | - | o | - | ツ | ソ | マ | ロ | ö | | | |

2) 1 6 0 2 L C D 指令说明

模块的内部操作有来自 M P U 的 R S 、 R / W 、 E 及数据信号 D B 0 — D B 7 决定。要对 DDRAM 的内容和地址进行具体操作，我们还需要根据 HD44780 的指令集浏览该指令集，并找出对 DDRAM 的内容和地址进行操作的指令。共 11 条指令，大致可以分为 4 大类：

- 模块功能设置，如显示格式。数据长度等。
- 设置内部 R A M 地址。
- 完成内部 R A M 数据传送。
- 完成其他功能。

一般情况下，内部 R A M 的数据传送得功能使用最为频繁，因此， R A M 中的地址指针所具备的自动加一或减一功能，在一定程度上减轻了 M P U 的变成。负担，此外，由于数据移位指令与写显示数据可同时进行，这样用户就能以最少的系统开发时间，达到最高的编程效率。

有一点需要特别注意：在每次访问模块之前， M P U 应首先检测忙标识 B F ，确认 B F = 0 后，访问过程才能进行。

1. 清屏指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /ms |
|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-------------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 清屏 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1.64 |

运行时间 (2 5 0 K H z)： 1 . 6 4 m s 功能：清 D D R A M 和 A C 值

功能：
 <1> 清除液晶显示器，即将 DDRAM 的内容全部填入"空白"的 ASCII 码 20H;
 <2> 光标归位，即将光标撤回液晶显示屏的左上方；
 <3> 将地址计数器(AC)的值设为 0 。

2. 光标归位指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /ms |
|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-------------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 光标归位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | X | 1.64 |

运行时间 (2 5 0 K H z)： 1 . 6 4 m s

功能： A C = 0 ，光标、画面回 H O M E 位

<1> 把光标撤回到显示器的左上方；
 <2> 把地址计数器(AC)的值设置为 0 ；
 <3> 保持 DDRAM 的内容不变

3. 进入模式设置指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|--------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-------------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 进入模式设置 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | S | 40 |

运行时间 (2 5 0 K H z)： 4 0 μ s 功能：设置光标、画面移动方式

设定每次定入 1 位数据后光标的移位方向，并且设定每次写入的一个字符是否移动。参数设定的

情况如下所示：

| | |
|-----|--|
| 位名 | 设置 |
| I/D | 0=写入新数据后光标左移 1=写入新数据后光标右移 |
| S | 0=写入新数据后显示屏不移动 1=写入新数据后显示屏整体右移 1 个字 |

4. 显示开关控制指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|--------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|----------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 显示开关控制 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | C | B | 40 |

运行时间 (250 KHz): 40 μ s

功能：设置显示、光标及闪烁开、关

参数设定的情况如下：

| | |
|----|--------------------|
| 位名 | 设置 |
| D | 0=显示功能关 1=显示功能开 |
| C | 0=无光标 1=有光标 |
| B | 0=光标闪烁 1=光标不闪烁 |

5. 设定显示屏或光标移动方向指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|--------------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|----------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 设定显示屏或光标移动方向 | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | X | X | 40 |

运行时间 (250 KHz): 40 μ s

功能：光标、画面移动，不影响 DDRAM

使光标移位或使整个显示屏移位。参数设定的情况如下：

| | | |
|-----|-----|--------------------|
| S/C | R/L | 设定情况 |
| 0 | 0 | 光标左移 1 格，且 AC 值减 1 |
| 0 | 1 | 光标右移 1 格，且 AC 值加 1 |
| 1 | 0 | 显示器上字符全部左移一格，但光标不动 |
| 1 | 1 | 显示器上字符全部右移一格，但光标不动 |

6. 功能设定指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|----------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 功能设定 | 0 | 0 | 0 | 0 | 1 | DL | N | F | X | X | 40 |

运行时间 (250 KHz): 40 μ s

功能: 工作方式设置 (初始化指令)

设定数据总线位数、显示的行数及字型。参数设定的情况如下:

| 位名 | 设置 |
|----|-------------------------------|
| DL | 0=数据总线为 4 位 1=数据总线为 8 位 |
| N | 0=显示 1 行 1=显示 2 行 |
| F | 0=5×7 点阵/每字符 1=5×10 点阵/每字符 |

7. 设定 CGRAM 地址指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|-------------|------|-----|-----|-----|---------------|-----|-----|-----|-----|-----|----------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 设定 CGRAM 地址 | 0 | 0 | 0 | 1 | CGRAM 的地址(6位) | | | | | | 40 |

运行时间 (250 KHz): 40 μ s

功能: 设置 C G R A M 地址。A5—A0 = 0—3 F H

设定下一个要存入数据的 CGRAM 的地址。

8. 设定 DDRAM 地址指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|-------------|------|-----|-----|---------------|-----|-----|-----|-----|-----|-----|----------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 设定 DDRAM 地址 | 0 | 0 | 1 | CDRAM 的地址(7位) | | | | | | 40 | |

运行时间 (250 KHz): 40 μ s

功能: 设置 D D R A M 地址

说明:

■ N = 0 , 一行显示 A6—A0 = 0—4 F H 。

■ N = 1 , 两行显示, 首行 A6—A0 = 0 0 H—2 F H , 次行 A6—A0。

(注意这里我们送地址的时候应该是 0x80+Address, 这也是前面说到写地址命令的时候要加上 0x80 的原因)

9. 读取忙信号或 AC 地址指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|-------------|------|-----|-----|----------|-----|-----|-----|-----|-----|-----|-------------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 读取忙碌信号或AC地址 | 0 | 1 | FB | AC内容(7位) | | | | | | 40 | |

功能：读忙 B F 值和地址计数器 A C 值

说明：B F = 1 : 忙； B F = 0 : 准备好。读取忙碌信号 BF 的内容，BF=1 表示液晶显示器忙，暂时无法接收单片机送来的数据或指令；当 BF=0 时，液晶显示器可以接收单片机送来的数据或指令。此时，A C 值即为最近一次地址设置（C G R A M 或 D D R A M ）定义。

10. 数据写入 DDRAM 或 CGRAM 指令一览

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|---------------------|------|-----|--------------|-----|-----|-----|-----|-----|-----|-----|-------------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 数据写入到 DDRAM 或 CGRAM | 1 | 0 | 要写入的数据 D7~D0 | | | | | | | | 40 |

运行时间：(2 5 0 K H z)： 4 0 μ s

功能：
 <1> 将字符码写入 DDRAM，以使液晶显示屏显示出相对应的字符；
 <2> 将使用者自己设计的图形存入 CGRAM。

11. 从 CGRAM 或 DDRAM 读出数据的指令一览

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|----------------------|------|-----|--------------|-----|-----|-----|-----|-----|-----|-----|-------------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 从 CGRAM 或 DDRAM 读出数据 | 1 | 1 | 要读出的数据 D7~D0 | | | | | | | | 40 |

运行时间：(2 5 0 K H z)： 4 0 μ s

功能：读取 DDRAM 或 CGRAM 中的内容。

基本操作时序：

- | | |
|-----|---|
| 读状态 | 输入： RS=L, RW=H, E=H 输出： DB0~DB7=状态字 |
| 写指令 | 输入： RS=L, RW=L, E=下降沿脉冲, DB0~DB7=指令码 输出： 无 |
| 读数据 | 输入： RS=H, RW=H, E=H 输出： DB0~DB7=数据 |
| 写数据 | 输入： RS=H, RW=L, E=下降沿脉冲, DB0~DB7=数据 输出： 无 |

3) 如何显示一个自定义的字符

步骤如下：

1. 先将自定义字符写入 CGRAM
2. 再将 CGRAM 中的自定义字符送到 DDRAM 中显示

我们从 CGROM 表上可以看到，在表的最左边是一列可以允许用户自定义的 CGRAM，从上往下看着是 16 个，实际只有 8 个字节可用。它的字符码是 00000000—00000111 这 8 个地址，表的下面还有 8 个字节，但因为这个 CGRAM 的字符码规定 0—2 位为地址，3 位无效，4—7 全为 0。因此 CGRAM 的字符码只有最后三位能用也就是 8 个字节了。等效为 0000X111，X 为无效位，最后三位为 000—111 共 8 个。如果我们想显示这 8 个用户自定义的字符，操作方法和显示 CGROM 的一样，先设置 DDRAM 位置，再向 DDRAM 写入字符码，例如“A”就是 41H。现在我们要显示 CGRAM 的第一个自定义字符，就向 DDRAM 写入 00000000B (00H)，如果要显示第 8 个就写入 00000111 (08H)。

我们来看下怎么向这 8 个自定义字符写入字模。

设置 CGRAM 地址的指令

| 指令功能 | 指令编码 | | | | | | | | | | 执行时间 /us |
|-------------|------|-----|-----|-----|---------------|-----|-----|-----|-----|-----|----------|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| 设定 CGRAM 地址 | 0 | 0 | 0 | 1 | CGRAM 的地址(6位) | | | | | | 40 |

从这个指令可以看出指令数据的高 2 位已固定是 01，只有后面的 6 位是地址数据，而这 6 位中的高 3 位就表示这 8 个自定义字符，最后的 3 位就是字模数据的八个地址了。例如第一个自定义字符的字模地址为 01000000—01000111 八个地址。我们向这 8 个字节写入字模数据，让它能显示出“C”

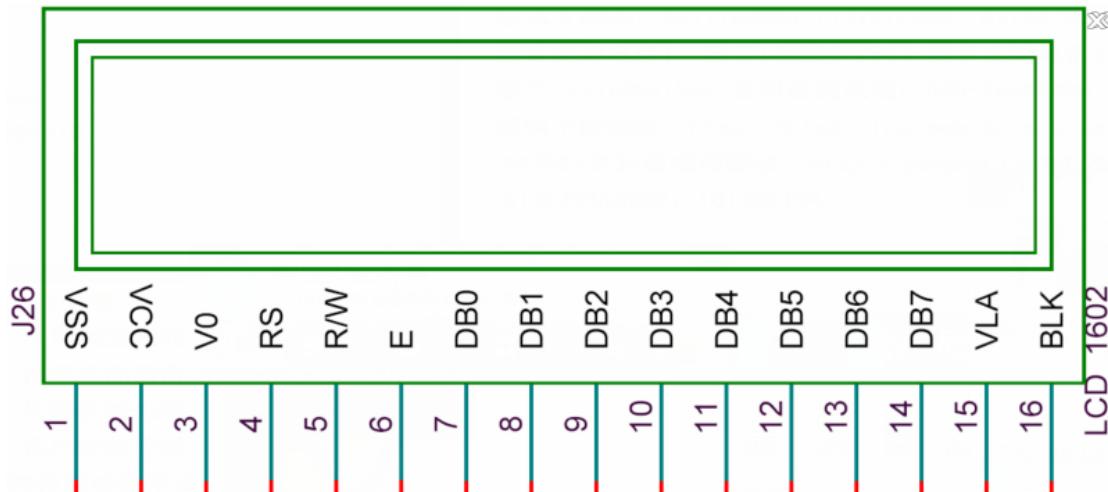
| | | | | | |
|-----|----------|-----|----------|-----|--------------|
| 地址: | 01000000 | 数据: | 00010000 | 图示: | ○○○ ■ ○○○○ |
| | 01000001 | | 00000110 | | ○○○○○ ■ □ ○○ |
| | 01000010 | | 00001001 | | ○○○○ ■ ○○○□ |
| | 01000011 | | 00001000 | | ○○○○ ■ ○○○○ |
| | 01000100 | | 00001000 | | ○○○○ ■ ○○○○ |
| | 01000101 | | 00001001 | | ○○○○ ■ ○○○□ |
| | 01000110 | | 00000110 | | ○○○○○ ■ □ ○○ |
| | 01000111 | | 00000000 | | ○○○○○ ○○○○ |

```
user[] = {0x10,0x06,0x09,0x08,0x08,0x09,0x06,0x00}; //字符'C' */
```

写入时先设置 CGRAM 地址 0x40；显示是直接取 CGRAM 的数据

7.75.6 硬件连接原理

神舟III开发板没有1602屏的接口，连接的时候使用杜邦线将开发板与1602屏连接。1602的引脚图如下：



功能管脚的介绍

| 1602 屏的管脚 | 对应神舟 III 的管脚 | 管脚的功能 |
|-----------|--------------|---------|
| VSS | GND | 地 |
| VCC | NC | |
| VO | GND | 地 |
| RS | PC10 | 数据/命令选择 |
| R/W | PC9 | 读/写选择 |
| E | PC8 | 使能信号 |
| D0-D7 | PE0-PE7 | 8 根数据线 |
| VLA | VCC | 电源 |
| BLK | GND | 地 |

第 1 脚：VSS 为地电源。

第 2 脚：悬空

第 3 脚：VL 为液晶显示器对比度调整端，接正电源时对比度最弱，接地时对比度最高，对比度过高时会产生“鬼影”，通常使用时可以通过一个 10K 的电位器调整对比度。这里为了方便我们直接接地。

第 4 脚：RS 为寄存器选择，高电平时选择数据寄存器、低电平时选择指令寄存器。

第 5 脚：R/W 为读写信号线，高电平时进行读操作，低电平时进行写操作。当 RS 和 R/W 共同为低电平时可以写入指令或者显示地址，当 RS 为低电平 R/W 为高电平时可以读忙信号，当 RS 为高电平 R/W 为低电平时可以写入数据。

第 6 脚：E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。

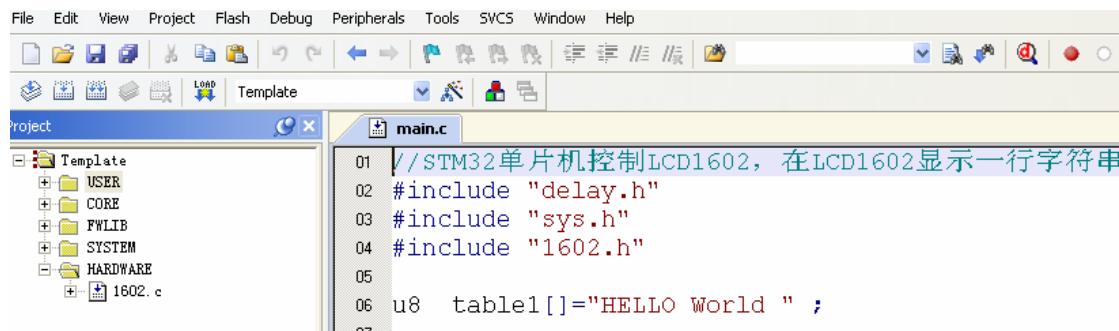
第 7~14 脚：D0~D7 为 8 位双向数据线。

第 15 脚：背光源正极。

第 16 脚：背光源负极。

7.75.7 7.80.5 代码分析

进入工程，打开 Template.uvproj 文件。



我们从主程序开始分析。

```
int main(void)
{
    u8 aa;
    delay_init(); // 延时函数初始化
    Init_LCD1602(); // LCD1602 初始化函数
    while(1)
    {
        write_com(0x80); // 第二行的首地址
        for(aa=0;aa<12;aa++)
        {
            write_dat(table1[aa]);
            delay_ms(3);
        }
    }
}
```

代码分析 1：主函数中首先定义了 aa，然后初始化延时函数。这些都不是主要的。这里不详细分析。

代码分析 2：函数 Init_LCD1602() 对 1602 屏进行初始化。

```
void Init_LCD1602()
{
    GPIO_InitTypeDef GPIO_Initstructure; // 定义了一个变量，变量的名字为GPIO_InitStruct
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE|RCC_APB2Periph_GPIOC, ENABLE);
    GPIO_Initstructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10;
    GPIO_Initstructure.GPIO_Mode = GPIO_Mode_Out_PP; // 推挽输出
    GPIO_Initstructure.GPIO_Speed = GPIO_Speed_50MHz; // IO口速度为50MHz
    GPIO_Init(GPIOC, &GPIO_Initstructure); // 根据设定参数初始化GPIOC

    GPIO_Initstructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4;
    GPIO_Initstructure.GPIO_Mode = GPIO_Mode_Out_PP; // 推挽输出
    GPIO_Initstructure.GPIO_Speed = GPIO_Speed_50MHz; // IO口速度为50MHz
    GPIO_Init(GPIOE, &GPIO_Initstructure); // 根据设定参数初始化GPIOE

    write_com(0x38); // 设置LCD两行显示，一个数据由5*7点阵表示，数据由8跟线传输
    delay_ms(2);
    write_com(0x01); // 清除屏幕显示
    delay_ms(2);
    write_com(0x06); // 设定输入方式，增量不移位
    delay_ms(2);
    write_com(0x0c); // 开整体显示，光标，不闪烁
    delay_ms(2);
}
}
```

初始化函数中，用了 PE0 到 PE7 做为 STM32 和液晶屏的数据交流线。PC8、PC9、PC10 分别是 LCD1602 的三根控制线 en, rw, rs。对它们进行初始化后，对 1602 屏进行设置。设置通过函数 write_com() 对 1602 屏发送命令进行设置。对应的命令在 1602 屏的基本原理部分有对应的说明。比如 0x01 是清除屏幕显示。

代码分析 4：初始化完成后，就可以对 LCD1602 屏发送命令和数据了。发送命令函数如下：

```
/******************写指令函数********************
void write_com(u8 com)
{
    // 写指令      输入： RS=L, RW=L, E=下降沿脉冲
    rs=0;
    rw=0;
    en=1;

    // 实际上在操作ODR寄存器，ODR寄存器是一端口输出
    // 从该寄存器读出来的数据可以用于判断当前IO口的
    GPIO_Write(GPIOE, 0X00FF&com); // 该函数一般用
    delay_ms(3);
    en=0;
}
```

Rs，数据/命令选择。rw 读/写选择。en 为使能端。rs 为 0 表示操作的是命令。rs、rw、en 管脚的定义如下：

```
#define rs PCout(10)
#define rw PCout(9)
#define en PCout(8)
```

读写函数的拉高拉低是按照读、写时序图来编写的。现在市场上这种程序很多，如果只想实现功能的话，可以直接参考他人编写的代码，这样可以大大提高效率。

代码分析 5: whlie () 循环, 向 LCD1602 屏写入数据。

```
while(1)
{
    write_com(0x80); //第一行的首地址
    for(aa=0;aa<12;aa++)
    {
        write_dat(table1[aa]);
        delay_ms(3);
    }
}
```

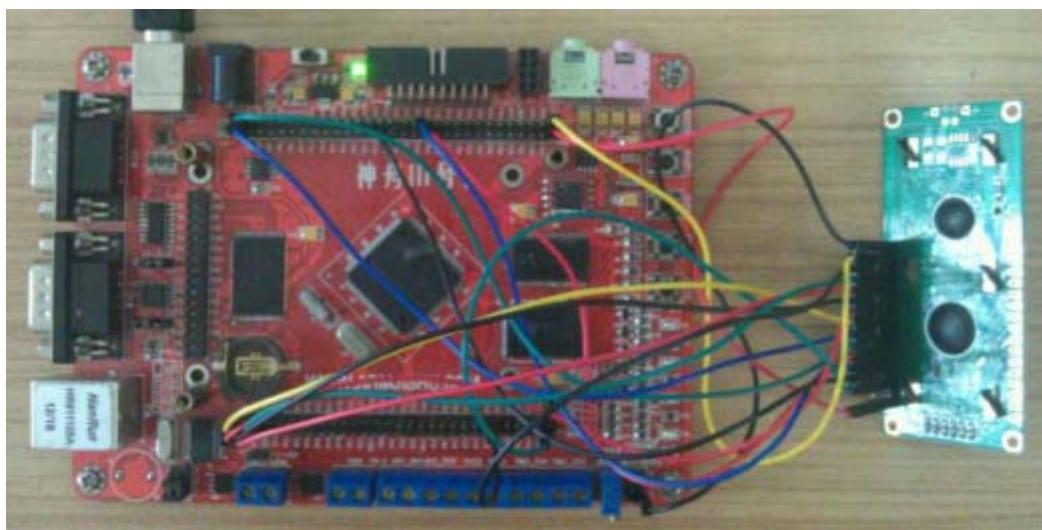
首先通过写命令函数 write_com(), 写入 0x80。确定 LCD1602 屏中显示的位置。在 for 循环中通过写数据函数 write_dat () 写入要显示的数据。显示的数据我们定义在一个数组 table1[] 中。通过本次 for 循环就可以显示我们要显示的字符数据了。

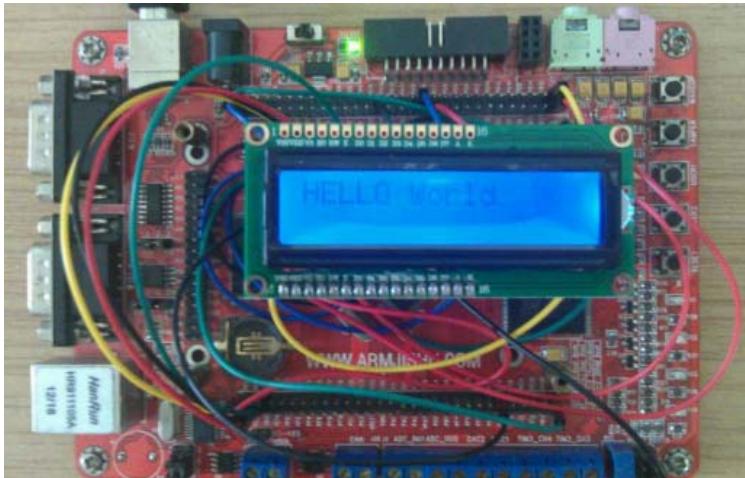
7.75.8 7.80.6 下载与验证

如果在MDK开发环境中, 下载编译好的固件或者在线调试, 请按3.5如何在MDK开发环境中使用JLINK在线调试小节进行操作。

7.75.9 7.80.7 实验现象

将固件程序下载到神舟 III 号 STM32 开发板后, 关闭电源。按照实验原理的管脚介绍, 用杜胖线将神舟 III 号与 LCD1602 连接。连接好后打开电源, 可以看到 LCD 屏上显示“HELLO World”字样。实验现象如下:

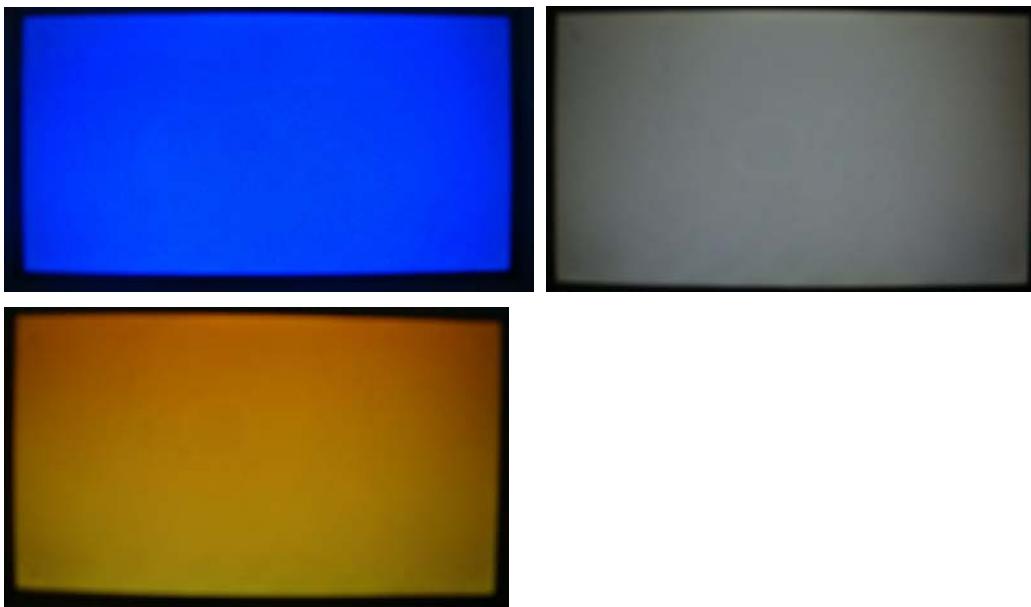




7.76 4.3寸液晶屏模块

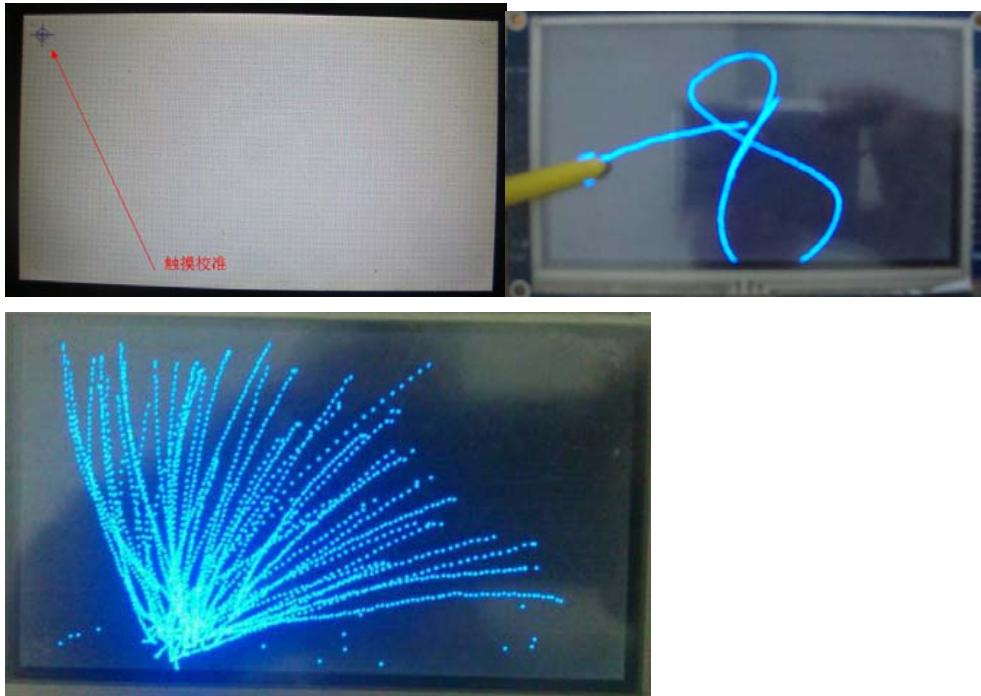
7.76.3 神舟III号_刷屏测试(4.3') 实验现象

将固件下载到神舟III号后，复位，正常情况下神舟III号4.3寸屏不停的显示不同颜色。以下是其中的三种颜色。



7.76.4 神舟III号_触摸测试(4.3') 实验现象

将固件下载到神舟III号后，复位，我们先对神舟III号4.3寸屏先进行触摸校准，然后触摸屏为底色是黑色的面板，我们可以在上面进行触摸绘画。



7.66.1 神舟III号_TFT绘图API(4.3')实验现象

将固件下载到神舟III号后，复位，正常情况下神舟III号4.3寸屏上显示“神舟科技”字样，显示矩形、圆形，显示小图片。



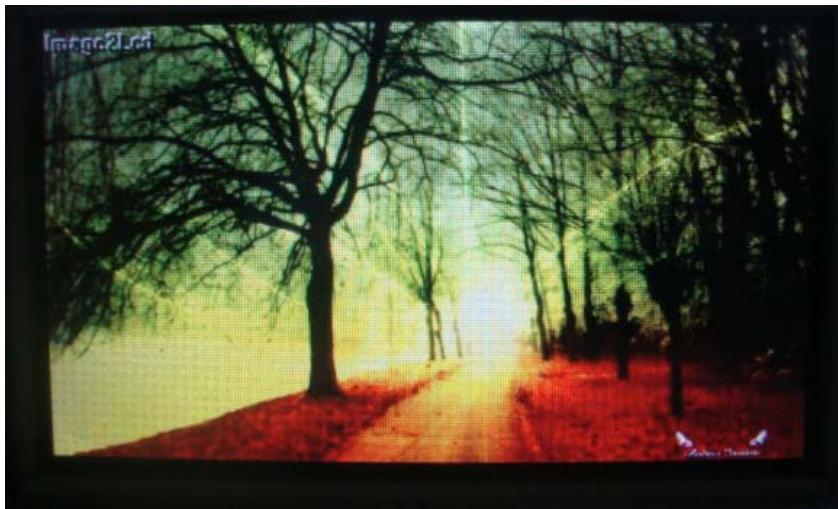
7.66.5 神舟III号_HZK16(4.3'卡在屏上) 实验现象

将工程下提供的“HZK16.bin”放置到SD卡根目录下，并将SD卡插入到4.3寸屏上。将固件下载到神舟III号后，复位，彩屏将显示英文字母及汉字。我们可以在代码中直接添加自己要显示的汉字不用进行字模转换。



7.66.4 神舟III号_电子相框实验现象

将工程下提供的图片放置到SD卡中，并将SD卡插入到开发板上。将固件下载到神舟III号后，复位，屏幕将SD卡中的图片进行切换。我们提供图片的类型是BMP格式的图片。

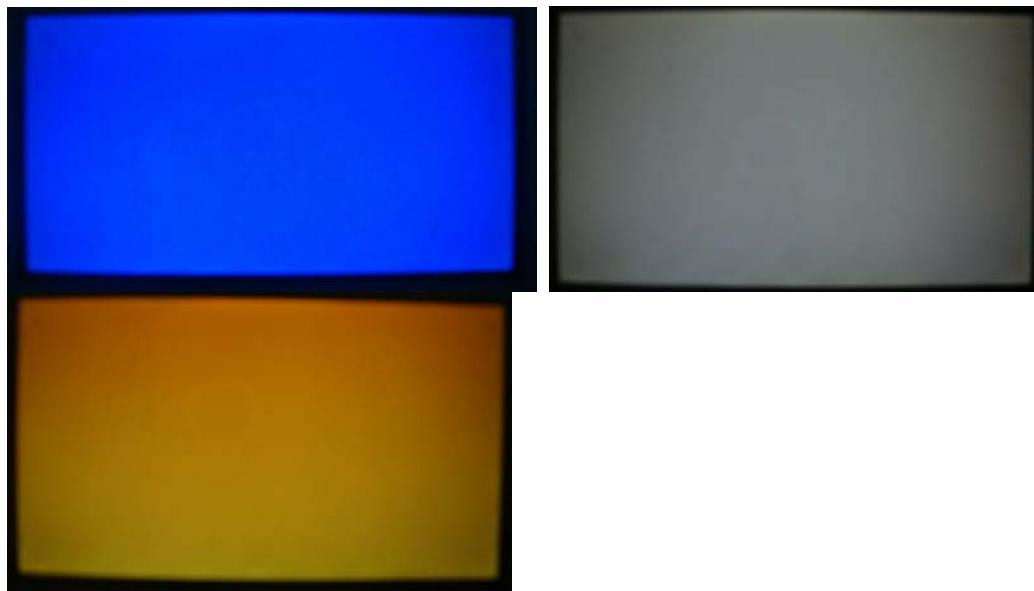


7.77 7寸液晶屏模块

7.77.1 神舟III号_TFT刷屏测试(7寸) 实验现象

神舟III底板的彩屏针脚一共引出32根，而7寸彩屏的引出的针脚是34跟。多出来的两根线分别是5V电源和地（GND）。我们将TTL模块的5V针脚和彩屏的5V针脚相连，地针脚和地针脚相连。

将固件下载到神舟III号后，复位，正常情况下神舟III号7寸屏不停的显示不同颜色。以下是其中的三种颜色。

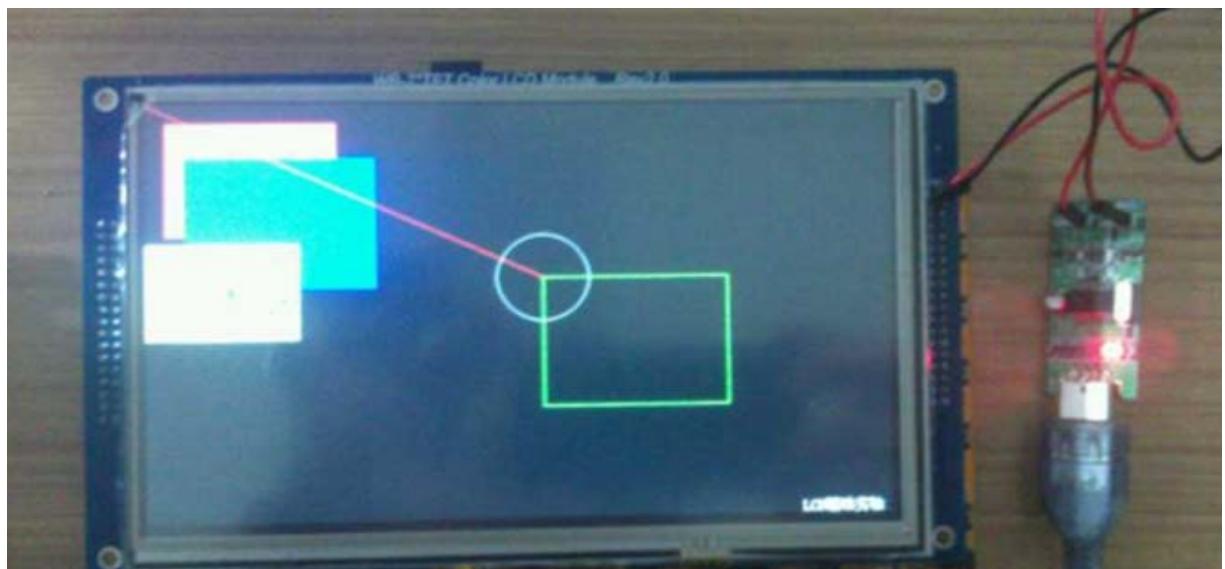


7.77.2 神舟III号_TFT绘图API(7寸) 实验现象

我们神舟III号开发板标配的是3.2寸屏。出厂的时候，在屏的四个角下面分别装了一个铜柱，使用7寸屏的时候需要把铜柱扭下来。否则开发板无法扣上7寸的彩屏。

神舟III底板的彩屏针脚一共引出32根，而7寸彩屏的引出的针脚是34跟。多出来的两根线分别是5V电源和地（GND）。我们将TTL模块的5V针脚和彩屏的5V针脚相连，地针脚和地针脚相连。（下图中红色杜邦线为电源线、黑色杜邦线为地连接线）

给开发板供电，将固件下载到神舟III号后，复位，正常情况下神舟III号7寸屏上显示矩形、圆形及显示小图片。如下图：



7.77.3 神舟III号_触摸屏测试(7寸) 实验现象

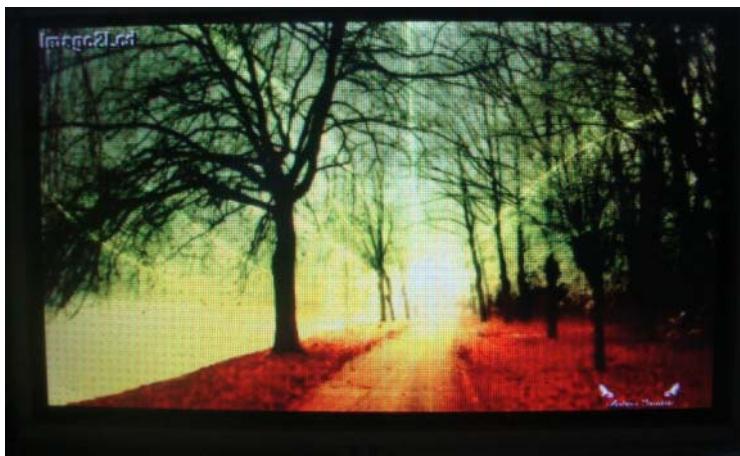
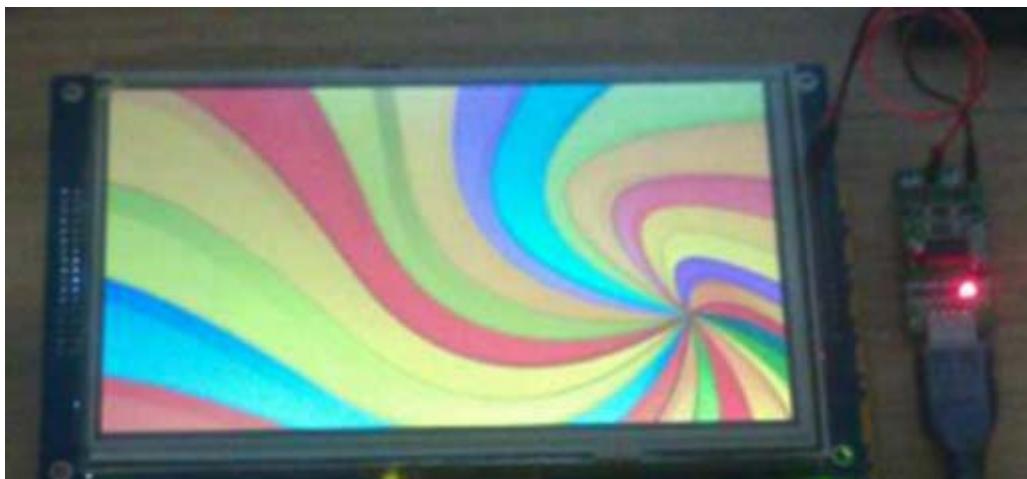
将固件下载到神舟III号，复位，我们先对神舟III号7寸屏先进行触摸校准，然后触摸屏为黑色是黑色的面板，我们可以在上面进行触摸。





7.77.4 神舟III号_电子相框_BIN(7'卡在屏上) 实验现象

sd卡格式成fat文件系统，然后放入将bmp图片文件拷贝一份到SD卡根目录下。将SD卡插入到7寸屏上。将固件下载到神舟III号后，复位，屏幕将SD卡中的图片进行切换。



附件1：如何从单片机零基础到嵌入式高手

A. 如果什么基础都没有，建议从 51 单片机开始学习，俗话说学懂 51 单片机，走遍天下都不怕。推荐学习 51 的原因如下：

51 单片机内部资源相对比较少但五脏俱全

学懂 51 单片机后，就像九阳神功，原理都一样

成就感，51 容易懂，成就感比较强

大部分各级 ARM 芯片都没有统一，51 单片机是鼻祖，新出来的模块，都会首先用 51 单片机调通

51 单片机资料非常丰富，容易学习和查找资料

B. 推荐神舟 51+ARM 单片机开发板，原因如下：

1) 15CM*20CM 的板，每大 1 平方厘米，可以增加 1 个功能

2) 手册教程 1000 页以上

3) 丰富的视频教程

4) 51+ARM 的开发板有助于学完 51，马上就可以进入 ARM 入门

5) 51+ARM 的开发板成体系，有助于一气呵成，技术支持也方便

6) 神舟 51+ARM 的 ARM 就是 STM32，神舟团队的优势非常明显

7) 视频非常精：视频教程多没用，主要要精，关于视频精，可看以下详细分析：

很多人都问 XX 祥单片机视频已经很好了，您的视频能跟他比不？这个问题一问，我们为了追求品质第一，还真的花了 2 天时间把 XX 祥的视频从头到尾的听了一遍，然后根据这个情况，进行重新录制，目前神舟 51+ARM 单片机视频第二版本具有三个特点：

特点 1：在深度上绝对超过 XX 祥视频，每个知识点我们都力求完美，包括代码的仔细程度，细节把握，初学者入门等

特点 2：在广度上，就是知识面涵盖上远远超越了 XX 祥，这个大家看开发板的资源就可以知道

特点 3：XX 祥的产品可能是 N 年前的产品，里面有一些可能淘汰的知识理念，神舟工程师都是一线的高手，综合去掉了这些陈旧淘汰的理念，加入了目前最强大的理念，相当于做了一个知识内容的升级，能让学习者避开掉 N 年前的很多坏习惯，直接导入到正轨上来。

当然光说是没有意义的，下面我们就部分视频详细说下具体差距，因为我们 51 资料足够巨大和丰富，所以这里篇幅有限，只点到为止，具体细节大家可以去看我们的光盘资料来进行学习：

神舟 51 单片机视频教程和 XX 祥的对比，整体来讲，神舟 51 单片机单片机视频第二版录制的有 17 讲，后续还会增加，包括 ARM 部分的视频教程；而 XX 祥的有 9 讲是单片机教程的，后面的几讲是做一些实践的，神舟 51 单片机视频教程比它更加丰富的有除了 ARM 的视频之外，就 51 单片机本身还有包括双色点阵，18B20 温度传感器，红外遥控器，实时时钟，步进电机直流电机的视频教程，后续还会增加 2.4G 无线，315M 无线，语音音频，485，CAN，超声波模块，重力感应等更多的丰富单片机例程和视频教程。

从细节上来看，神舟 51 单片机视频教程开始的时候，先是介绍如何去学习单片机，有什么方法去学习，再引入到单片机的内容，XX 祥的单片机视频教程差不多是直接引入到了单片机的内容当中，对

一些硬件的分析不是很详细，有些知识过早的介绍到了，用户的学习量就加大了，下面我们将对整个视频教程的每一讲来分析下 2 个视频教程的区别。

第一讲：点亮你的 LED——神舟 51 单片机介绍了单片机学习方法、数据类型与运算符的使用和 LED 的介绍与例程代码实验的分析，KEIL 的使用工程建立与仿真。XX 祥的单片机视频教程没有很好的介绍到如何是学习这个单片机，对 LED 的介绍不是很详细，硬件涉及广，信息量大，难以记全，神舟 51 单片机大多是用到什么的时候再介绍，更容易接受、记住。

第二讲：流水灯和数码管——神舟 51 单片机在点亮 LED 的基础上，通过控制一位 LED 到现在的控制 8 个 LED，从操作一位的 I/O 口到学会了操作 8 个 I/O 口的转变，通过多种的代码控制我们的 I/O 口达到我们想要的效果。并延伸到了数码管的学习与控制，对这个数码管做了一个全面的分析介绍，在这里我们还学会了函数的调用与一些 C 语言的语句与指令的使用，像子函数、数组等等。而 XX 祥单片机注重于硬件仿真这块，对硬件的介绍不多，只通过一个代码达到实验的结果，不像神舟 51 单片机用到了多种代码的介绍达到同样的结果。

第三讲：数码管的显示方式与按键——神舟 51 单片机视频教程通过前面第二讲的独立数码管的介绍延伸到了我们的 8 位数码管的学习，对数码管的 2 种显示方式做了一次详细的介绍，使用户能更好的接收学习，通过控制 8 位数码管需要用到的器件三八译码器，对这个三八译码器、锁存器做了一个全面的分析，用到什么器件就介绍什么器件，加深理解。当接收完这些知识后，通过我们的代码详细分析，让我们的单片机实现我们想要的效果，在最后还对独立按键做了一个全面的分析，包括代码的分析。这些做得都比 XX 祥单片机视频教程的好点，它对数码管的介绍不是很详细。

第四讲：硬件基础与 KEIL 仿真——神舟 51 单片机视频教程通过这一讲，加深学习 51 对硬件的理解，明白这些硬件的主要作用与使用的方法，这个是 XX 祥视频教程是没有的，学习完硬件基础后，再来单独的讲下如何去使用 KEIL 的这个软件进行一个仿真功能的作用与使用介绍，加深对 KEIL 软件的应用

第五讲：定时器的学习——神舟 51 单片机视频教程先由各个周期介绍，明白它们的不同之处，为下面的定时器学习做一个铺垫，能更好的理解定时器的作用与应用。虽然 XX 祥的也对这些周期做了一个介绍，但是缺少了他们的不同之处介绍。在定时器方面，神舟 51 单片机视频教程中，让人明白什么是定时器，它的作用与工作原理是什么，再延伸出它的工作方式，一个很好的思路，XX 祥单片机视频教程对定时器介绍的这部分不是很足，定时器的定时范围与初值计算介绍较少。理解完定时器的工作方式与初值计算后，最后通过代码去分析与实现定时器的应用。而 XX 祥单片机视频教程的代码介绍不多。

第六讲：中断系统的学习——神舟 51 单片机视频教程在教学中断的时候，先介绍了什么是中断→什么是单片机的中断→中断的优点好处→产生中断的来源。当单片机接收到这些来源后它的处理响应是怎么样的，如何配置这些中断来学习我们的单片机中断，一步扣一步的学习，加深对中断的理解，最后再通过代码的详细分析对中断应用。XX 祥单片机视频教程对中断的讲解和神舟 51 单片机的差不多，只是对中断的应用与实际例程讲解不足。

第七讲：键盘与点阵——神舟 51 单片机视频教程因为键盘的接法和点阵的接法类似，所以把它们放到一起介绍，先从矩阵键盘的介绍到它的连接方法开始，讲解了单片机是如何去识别这个按键的数值的，它的实现过程是怎么样的，最后通过我们的实际例程代码去讲解。点阵也是一样，先是对应矩阵做了一个详细的介绍，让我们知道了什么是点阵，它的作用分类都有什么，再从它的内部结构讲解，从里到外的了解了这个点阵，最后怎么样去使用这个点阵，它的原理是什么，神舟 51 单片机视频教程都做了一个详细的讲解，后面也一样，使用了我们的例程代码去分析。而我们的 XX 祥单片机视频教程，只是对键盘做了一个介绍，在扫描方法上没做多详细的讲解，对我们的点阵也是没有介绍到。

第八、九讲：串行通信——神舟 51 单片机视频教程使用了 2 讲来讲解这个串行通信，对这个串行通信做了一个非常详细的讲解，先是介绍了什么是通信，通信的分类、特点与原理等，在对通信有了初步的了解之后再来学习怎么样去使用这个通信的功能，像波特率产生的设置，怎么样去配置我们

的波特率、怎么样去操作我们的通信的寄存器等等，清楚了这些后我们还通过了例程源码详细的讲解分析我们前面所学到的各个内容，加深对通信的应用和了解。而 XX 祥单片机视频教程对这个通信也做了详细的介绍，对 USB 通信或者是 DB9 的串口通信、收发波特率与一些小细节没做到详细的说明。

第十、十一讲：I2C 通信学习——神舟 51 单片机视频教程使用了 2 讲来讲解这个 I2C 通信学习，对这个 I2C 通信做了一个非常详细的讲解，开始对 I2C 总线进行了一个详细的分析，明白它的定义、原理与特点是什么，然后从 I2C 的时序讲解它数据的发送接收是如何进行的，在运行的时候会发生什么情况都详细的描述了，还用了我们常用的 I2C 芯片 EEPROM—24C02 来做为我们学习 I2C 通信的器件，对这个芯片的操作与硬件连接都有了一定的框架，通过时序的方式发送单字节、多字节是如何操作的，和前面一样，最后都用我们的例程详细的描述了它是如何进行这个 I2C 方式传输数据与配置方式的。XX 祥单片机视频教程对于 I2C 通信的教程对这个 I2C 通信的介绍不是那么详细，也没有通过介绍一些 I2C 通信的芯片器件去讲解 I2C，对于时序的说明与例程程序的讲解也是一样的。

第十二讲：18B20 温度传感器学习——神舟 51 单片机视频教程的第十二讲是温度传感器的视频教程，主要是讲解了这个 18B20 器件是怎么样测得当前的温度，通过把这个温度的数据传送到单片机上，在显示设备上显示出来的，先是讲解了什么是温度传感器，通过了其中最常用的 18B20 来做为我们的参考器件描述温度传感器的原理、指令与时序，得出我们的一个当前温度的数据到单片机上，需要注意的是必须按照时序才能正常测温，这个可以由我们通过程序来实现。而 XX 祥单片机没有这方面的视频教程。

第十三讲：1602 液晶屏学习——神舟 51 单片机视频教程主要是通过我们的 1602 液晶屏的用户手册来讲解我们的内容，从了解什么是 1602 液晶屏到我们的操作时序接收发送数据，明白如何让这个 1602 液晶屏显示出我们想要的字符出来，最后通过了我们的例程代码讲解显示出来，修改了代码会出现什么样的状态等等，使我们对 1602 更加的熟悉。XX 祥单片机视频教程也是通过了我们的用户手册来讲解的，都是比较详细的一个教程，程序的讲解可能不是那么的详细，1602 显示的字符数据不是那么丰富。

第十四讲：红外通信的学习——神舟 51 单片机视频教程中，对红外做了一个非常详细的分析，从什么是红外通信到我们的红外通信应用，知道了红外的特点是什么。对这个红外有了一定的了解后，分析它的发送、接收的条件，红外接收、发送的电路是怎么样的。通过满足这些条件达到我们发射红外信号和接收红外信号的目的。而我们的单片机对接收到的红外信号或者是接收到红外信号的一个处理。这样就达到了我们红外通信的一个效果。最后我们通过实践代码的分析操作，使用逻辑分析仪把红外通信的波形抓出来给大家做参考，明白发射、接收波形之间的关系。XX 祥单片机视频教程没有红外通信这样一个教程。

第十五讲：单片机实时时钟的学习——神舟 51 单片机视频教程通过我们的 DS1302 实时时钟芯片来作为我们这一讲的教材器件，明白什么是实时时钟，它的作用与操作方法是怎么样的，通过分析这个芯片的特性和原理来讲解这个实时时钟，分析它的寄存器，最后我们通过程序去控制这些寄存器达到我们学习这个实时时钟的目的。XX 祥单片机视频教程上没有这个实时时钟的学习教材。

第十六讲：AD、DA 的学习——神舟 51 单片机视频教程在教学这个 AD、DA 的时候，先对这 2 个名词做了一个详细的讲解，明白了什么是 AD，什么是 DA，然后通过 2 个典型的电路对 AD、DA 的转换进行了一个详细的讲解，明白了模拟信号是如何转换成数字信号，或者是数字信号转换成模拟信号的，它们的指标又是什么，还通过了我们的单片机中的 AD、DA 转换芯片 PCF8591 做了一个介绍，最后通过了我们的程序代码分析实现我们的信号转换目的。XX 祥单片机视频教程对这个 AD、DA 也做了一个详细的讲解，不同之处是在于程序代码的分析与芯片使用的介绍。

第十七讲：电机的学习——神舟 51 单片机视频教程在这一讲中，讲解了 2 个方面的内容，一个是步进电机的学习，一个是直流电机的学习，分别对步进电机与直流电机做了一个全面的分析，从它们的作用、分类与驱动器件到我们的工作原理做了详细的分析，通过介绍我们的单片机驱动电机的驱动芯片来分别驱动我们的电机，最后我们通过例程代码的分析、控制与下载，实现了我们让电机转动起来的目的。XX 祥单片机视频教程没有对这个电机进行一个讲解。

以上为神舟 51 单片机视频与 XX 祥单片机视频的一个对比情况，此详细分析是由一名爱好者建议而提供的，不做为攻击 XX 祥产品使用，同样做为单片机领域的贡献者，我们是很尊重前辈的。

附件2：STM32神舟系列其他开发板介绍

1. 神舟 51+ARM (STM32F103C8T) 单片机开发板



2. STM32神舟I号（STM32F103RBT）开发板

STM32 从入门到精通

2013年8月版本 V3.0 作者: jesse

STM32神舟ARM系列技术开发板产品目录:

- 神舟51开发板（51+ARM）开发板
- 【**神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏**】
- 神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- 神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏
- 神舟IV号: STM32F107VCT6 + 3.2" TFT 触摸彩屏
- STM32核心板: 四层核心板(STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列(STM32F103ZET核心板)
- 神舟王207系列(STM32F207ZGT核心板)
- 神舟王407系列(STM32F407ZGT/407IGT核心板)
- 神舟王全系列(STM32F103ZET/207ZGT/407ZGT核心板)：全功能底板（支持MP3，以太网，收音机，无线，SRAM，Nor/Nand Flash，鼠标，键盘，红外接收，CAN，示波器，电压表，USB HOST，步进电机，RFID物联网等）
- 神舟51开发板(STC 51单片机+STM32F103C8T6核心板)：全功能底板（支持音频播放，无线，鼠标，键盘，红外收发，CAN，温度传感器，直流电机，步进电机，实时时钟，两路485，两路继电器，小喇叭，热敏光敏电阻，RFID物联网等）



| | |
|-------------------------|-----------|
| 5.14.8 软件设计 | 315 |
| 5.14.9 下载与现象 | 317 |
| 5.15 2.4G模块通信试验 | 318 |
| 5.15.1 2.4G模块无线模块的工作原理 | 318 |
| 5.15.2 2.4G模块通信实验的意义与作用 | 319 |
| 5.15.3 实验原理 | 319 |
| 5.15.4 硬件设计 | 320 |
| 5.15.5 软件设计 | 320 |
| 5.15.6 下载与测试现象 | 326 |
| 5.16 USB遥控鼠标实验 | 327 |
| 5.16.1 实验的意义与作用 | 327 |
| 5.16.2 实验原理 | 327 |
| 5.16.3 硬件设计 | 328 |
| 5.16.4 软件设计 | 329 |
| 5.16.5 下载与测试 | 332 |
| 5.17 MICRO SD卡实验 | 333 |
| 5.17.1 实验的意义与作用 | 333 |
| 5.17.2 实验原理 | 333 |
| 5.17.3 硬件设计 | 335 |
| 5.17.4 软件设计 | 336 |
| 5.17.5 下载与测试 | 340 |
| 5.18 SD-USB读卡器实验 | 341 |
| 5.18.1 实验的意义与作用 | 342 |
| 5.18.2 试验原理 | 342 |
| 5.18.3 硬件设计 | 342 |
| 5.18.4 软件设计 | 343 |
| 5.18.5 下载与测试 | 346 |
| 6.3 UCOSII操作系统之单任务运行 | 347 |
| 6.22.1 UCOSII介绍 | 错误！未定义书签。 |
| 6.22.2 实验原理 | 348 |
| 6.22.3 硬件设计 | 348 |
| 6.22.4 软件设计 | 349 |
| 6.22.5 下载与测试 | 351 |
| 6.4 UCOSII操作系统之单任务运行 | 351 |
| 6.23.1 SYSTICK时钟介绍 | 351 |
| 6.23.2 实验原理 | 352 |
| 6.23.3 硬件设计 | 352 |
| 6.23.4 软件设计 | 352 |
| 6.23.5 下载与测试 | 353 |
| 5.19 UCOS_UCGUI_DEMO实验 | 354 |
| 第 7 章 实验现象 | 355 |
| 附件 1：JLINK V8 用户手册 | 356 |

- 01. LED点灯实验 (STM32神舟I号-寄存器版)
- 02. LED双灯闪烁实验 (STM32神舟I号-寄存器版)
- 03. LED三个灯同时亮同时灭 (STM32神舟I号-寄存器版)
- 04. LED流水灯 (STM32神舟I号-寄存器版)
- 05. STM32芯片32MHZ频率下跑点灯程序 (STM32神舟I号-寄存器版)
- 06. STM32芯片40MHZ频率下跑点灯程序 (STM32神舟I号-寄存器版)
- 07. STM32芯片72MHZ频率下全速跑LED流水灯 (STM32神舟I号-寄存器版)
- 08. LED点灯实验-增加.h头文件 (STM32神舟I号-寄存器版)
- 09. STM32芯片72MHZ频率下全速跑LED流水灯-增加.h头文件管理 (STM32神舟I号-寄存器版)
- 10. STM32芯片72MHZ频率下全速跑LED流水灯-函数版初级 (STM32神舟I号-寄存器版)
- 11. STM32芯片72MHZ频率下全速跑LED流水灯-函数版中级 (STM32神舟I号-寄存器版)
- 12. STM32芯片72MHZ频率下全速跑LED流水灯-函数版高级 (STM32神舟I号-寄存器版)
- 13. STM32芯片按键点灯初级 (STM32神舟I号-寄存器版)
- 14. STM32芯片按键点灯中级-增加了防抖的代码 (STM32神舟I号-寄存器版)
- 15. 最简单串口打印\$字符-初级 (STM32神舟I号-寄存器版)
- 16. 单串口打印www.armjishu.com字符-中级 (STM32神舟I号-寄存器版)
- 17. 单串口打印www.armjishu.com字符-中级函数 (STM32神舟I号-寄存器版)
- 18. 单串口打印www.armjishu.com字符-高级 (STM32神舟I号-寄存器版)
- 19. 单个LED点灯程序 (STM32神舟I号-库函数版)
- 20. 单个LED灯闪烁 (STM32神舟I号-库函数版)
- 21. LED流水灯程序 (STM32神舟I号-库函数版)
- 22. KEY_LED按键与315M无线实验 (STM32神舟I号-库函数版)
- 23. USART-COM1串口接收与发送实验-初级版 (STM32神舟I号-寄存器版)
- 24. USART-COM1串口接收与发送实验-高级版 (STM32神舟I号-寄存器版)
- 25. USART-COM1串口发送实验 (STM32神舟I号-库函数版)
- 26. USART-COM1串口接收与发送实验 (STM32神舟I号-库函数版)
- 27. ADC模数转换实验 (STM32神舟I号-库函数版)
- 28. EEPROM读写程序实验 (STM32神舟I号-库函数版)
- 29. SPI FLASH(W25X16)读写程序实验 (STM32神舟I号-库函数版)
- 30. Calendar实时时钟与年月日实验 (STM32神舟I号-库函数版)
- 31. 独立看门狗-STM32芯片不断复位-设有喂狗 (STM32神舟I号-寄存器版)
- 32. 独立看门狗-STM32芯片不断复位-按键喂狗 (STM32神舟I号-寄存器版)
- 33. 独立看门狗按钮喂狗实验 (STM32神舟I号-库函数版)
- 34. 窗口看门狗-STM32芯片不断复位-设有设定窗口 (STM32神舟I号-寄存器版)
- 35. 窗口看门狗-STM32芯片不断复位-设定窗口并喂狗 (STM32神舟I号-寄存器版)
- 36. SysTick系统滴答实验 (STM32神舟I号-库函数版)
- 37. TFT彩屏显示实验 (STM32神舟I号-库函数版)
- 38. TFT触摸屏显示加触摸实验 (STM32神舟I号-库函数版)
- 39. 18B20温度传感实验 (STM32神舟I号-库函数版)
- 40. 2.4G无线通信实验 (神舟I号)
- 41. USB遥控鼠标实验 (神舟I号)
- 42. SD卡实验 (神舟I号)
- 43. SD-USB读卡器实验 (神舟I号)
- 44. uCOSII操作系统之单任务运行 (神舟I号-库函数版)
- 45. uCOSII操作系统之多任务运行 (神舟I号-库函数版)
- 46. uCOS_UCGUI_DEMO实验 (神舟I号-库函数版)

STM32 神舟 I 号最新手册和例程 8 月火热天气热情发布：<http://pan.baidu.com/share/link?shareid=340663&uk=3945314662#> ?

3. STM32神舟III号 (STM32F103ZET) 开发板



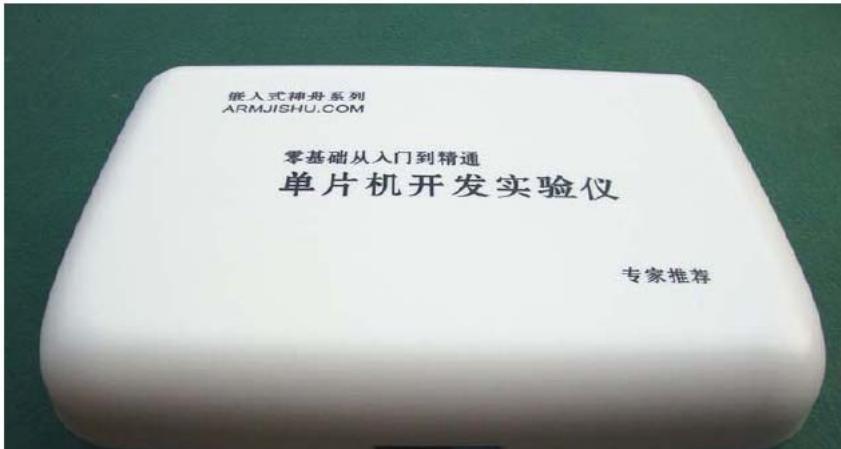
1 / 474 | 90% | | | | | 搜索

STM32 神舟 III 号开发板用户手册

2013 年 8 月版本 V2.0 作者: WWW.ARMJISHU.COM

STM32神舟ARM系列技术开发板产品目录:

- 【神舟51单片机开发板（51+ARM）开发板】
- 神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏
- 神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- 【神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏】
- 神舟IV号: STM32F107VCT6 + 3.2" TFT 触摸彩屏
- STM32核心板: 四层核心板(STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列(STM32F103ZET核心板)
- 神舟王207系列(STM32F207ZGT核心板)
- 神舟王407系列(STM32F407ZGT/407IGT核心板)
- 神舟王全系列(STM32F103ZET/207ZGT/407ZGT核心板): 全功能底板(支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
- 神舟 51 开发板 (STC 51 单片机+STM32F103C8T6 核心板): 全功能底板(支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)



- [01. LED点灯实验 (STM32神舟III号-寄存器版)]
- [02. LED双灯闪烁实验 (STM32神舟III号-寄存器版)]
- [03. LED三个灯同时亮同时灭 (STM32神舟III号-寄存器版)]
- [04. LED流水灯 (STM32神舟III号-寄存器版)]
- [05. STM32芯片32MHZ频率下跑点灯程序 (STM32神舟III号-寄存器版)]
- [06. STM32芯片40MHZ频率下跑点灯程序 (STM32神舟III号-寄存器版)]
- [07. STM32芯片72MHZ频率下全速跑LED流水灯 (STM32神舟III号-寄存器版)]
- [08. STM32芯片按键点灯-无防抖 (STM32神舟III号-寄存器版)]
- [09. STM32芯片按键点灯中级-增加了防抖 (STM32神舟III号-寄存器版)]
- [10. 最简单串口打印\$字符-初级 (STM32神舟III号-寄存器版)]
- [11. 单串口打印www.armjishu.com字符-初级 (STM32神舟III号-寄存器版)]
- [12. 单串口打印www.armjishu.com字符-中级 (STM32神舟III号-寄存器版)]
- [13. 单串口打印www.armjishu.com字符-高级 (STM32神舟III号-寄存器版)]
- [14. USART-COM1串口接收与发送实验-初级版 (STM32神舟III号-寄存器版)]
- [15. USART-COM1串口接收与发送实验-中级版 (STM32神舟III号-寄存器版)]
- [16. USART-COM1串口接收与发送实验-高级版 (STM32神舟III号-寄存器版)]
- [17. LED流水灯 (神舟III号-库函数版)]
- [18. BEEPER蜂鸣器 (神舟III号-库函数版)]
- [19. KEY_LED按键与315M无线实验 (神舟III号-库函数版)]
- [20. USART-COM1串口发送 (神舟III号-库函数版)]
- [21. 串口收发一中断方式 (神舟III号-库函数版)]
- [22. RS485总线实验 (神舟III号-库函数版)]
- [23. CAN (LoopBack) 程序 (神舟III号-库函数版)]
- [24. ADC模数转换 (神舟III号-库函数版)]
- [25. EEPROM读写程序 (神舟III号-库函数版)]
- [26. SPI FLASH (W25X16)读写程序 (神舟III号-库函数版)]
- [27. SysTick系统滴答 (神舟III号-库函数版)]
- [28. SRAM访问程序 (神舟III号-库函数版)]
- [29. NOR FLASH 测试程序 (神舟III号-库函数版)]
- [30. NAND FLASH访问程序 (神舟III号-库函数版)]
- [31. Calendar实时时钟与年月日 (神舟III号-库函数版)]
- [32. 2.4G无线通信实验 (神舟III号-库函数版)]
- [33. FM收音机 (神舟III号-库函数版)]
- [34. ENC28J60以太网卡 (神舟III号-库函数版)]
- [35. TFT彩色液晶屏只显示红色 (神舟III号-库函数版)]
- [36. TFT彩色液晶屏只显示蓝色 (神舟III号-库函数版)]
- [37. TFT彩色液晶屏显示英文字 (神舟III号-库函数版)]
- [38. TFT彩色液晶屏显示中文字 (神舟III号-库函数版)]
- [39. TFT彩色液晶屏混合显示中英文字 (神舟III号-库函数版)]
- [40. TFT彩屏显示图片 (神舟III号-库函数版)]
- [41. TFT彩屏显示图片刷屏 (神舟III号-库函数版)]
- [42. 2.8寸TFT触摸屏 (神舟III号-库函数版)]
- [43. 3.2寸TFT触摸屏 (神舟III号-库函数版)]
- [44. SD读卡器 (神舟III号-库函数版)]
- [45. MP3播放器实验 (神舟III号-库函数版)]
- [46. uCOSII操作系统之单任务运行 (神舟III号-库函数版)]
- [47. uCOSII操作系统之多任务运行 (神舟III号-库函数版)]
- [48. uCOS+uCGUI (神舟III号, FSMC 支持2.8寸和3.2寸)]

 STM32神舟III号用户手册2013.9.5.pdf

 STM32神舟III号例程源码20130905.rar

STM32 神 舟 III 号 开 发 板 光 盘 资 料 下 载 :
<http://pan.baidu.com/share/link?shareid=2752114865&uk=2687884684>

4. STM32神舟III号（STM32F103ZET）开发板

STM32 神舟 IV 号开发板用户手册

2013年10月版本 v2.0 作者: www.armjishu.com

STM32神舟ARM系列技术开发板产品目录:

- 神舟51+ARM单片机开发板开发板
- STM32神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏
- STM32神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- STM32神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏
- 【STM32神舟IV号: STM32F107VCT6 + 3.2" TFT 触摸彩屏】
- STM32核心板: 四层核心板 (STM32F103ZET+207ZGT+407ZGT+407IGT)
- STM32神舟王103系列 (STM32F103ZET核心板)
- STM32神舟王207系列 (STM32F207ZGT核心板)
- STM32神舟王407系列 (STM32F407ZGT/407IGT核心板)
- 神舟王全系列 (STM32F103ZET/207ZGT/407ZGT核心板): 全功能底板 (支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
- 神舟 51 开发板 (STC 51 单片机+STM32F103C8T6 核心板): 全功能底板 (支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)



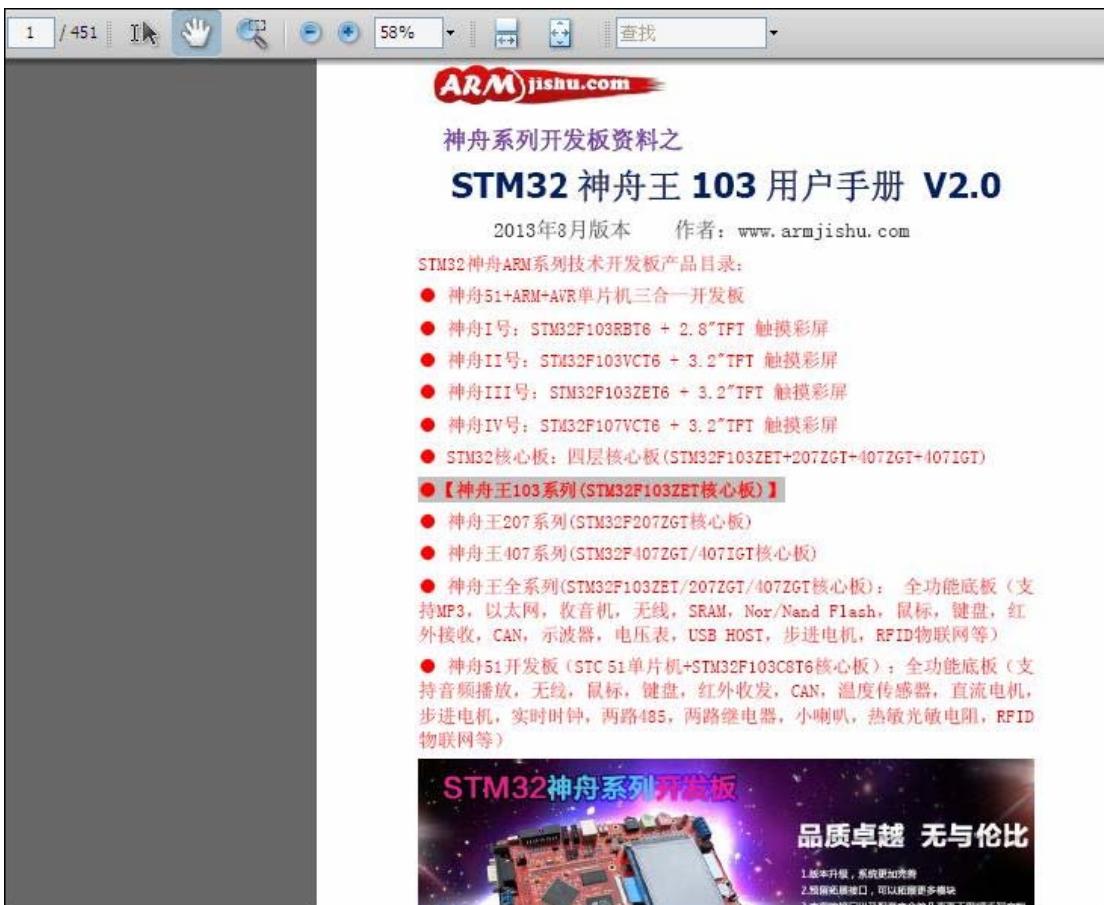
- 01. LED点灯实验 (STM32神舟IV号-寄存器版)
- 02. LED双灯闪烁实验 (STM32神舟IV号-寄存器版)
- 03. LED三个灯同时亮同时灭 (STM32神舟IV号-寄存器版)
- 04. LED流水灯 (STM32神舟IV号-寄存器版)
- 05. STM32芯片32MHZ频率下跑点灯程序 (STM32神舟IV号-寄存器版)
- 06. STM32芯片40MHZ频率下跑点灯程序 (STM32神舟IV号-寄存器版)
- 07. STM32芯片72MHZ频率下全速跑LED流水灯 (STM32神舟IV号-寄存器版)
- 08. STM32芯片按键点灯初级 (STM32神舟IV号-寄存器版)
- 09. STM32芯片按键点灯中级-增加了防抖的代码 (STM32神舟IV号-寄存器版)
- 10. 最简单串口打印\$字符-初级 (STM32神舟IV号-寄存器版)
- 11. 单串口打印www.armjishu.com字符-中级 (STM32神舟IV号-寄存器版)
- 12. 单串口打印www.armjishu.com字符-中级函数 (STM32神舟IV号-寄存器版)
- 13. 单串口打印www.armjishu.com字符-高级 (STM32神舟IV号-寄存器版)
- 14. USART-COM1串口接收与发送实验-初级版 (STM32神舟IV号-寄存器版)
- 15. USART-COM1串口接收与发送实验-中级版 (STM32神舟IV号-寄存器版)
- 16. USART-COM1串口接收与发送实验-高级版 (STM32神舟IV号-寄存器版)
- 17. LED流水灯 (STM32神舟IV号-库函数版)
- 18. BEEP蜂鸣器 (STM32神舟IV号-库函数版)
- 19. 按键扫描检测 (STM32神舟IV号-库函数版)
- 20. 按键扫描检测(子函数) (STM32神舟IV号-库函数版)
- 21. BitBand按键扫描检测 (STM32神舟IV号-库函数版)
- 22. BitBand按键扫描检测(子函数) (STM32神舟IV号-库函数版)
- 23. 按键EXTI外部中断 (STM32神舟IV号-库函数版)
- 24. SysTick系统滴答 (STM32神舟IV号-库函数版)
- 25. UART串口2Printf (STM32神舟IV号-库函数版)
- 26. UART串口2Printf输出和scanf输入 (STM32神舟IV号-库函数版)
- 27. UART串口1Printf (STM32神舟IV号-库函数版)
- 28. UART串口1Printf输出和scanf输入 (STM32神舟IV号-库函数版)
- 29. UART串口1和串口2同时格式化输出输入 (STM32神舟IV号-库函数版)
- 30. UART串口中断 (STM32神舟IV号-库函数版)
- 31. RS-485总线收发实验 (STM32神舟IV号-库函数版)
- 32. 产品唯一身份标识 (Unique Device ID) (STM32神舟IV号-库函数版)
- 33. ADC模数转换 (STM32神舟IV号-库函数版)
- 34. 简单实时时钟RTC (STM32神舟IV号-库函数版)
- 35. 实时时钟RTC与农历年月日 (STM32神舟IV号-库函数版)
- 36. EEPROM读写测试 (STM32神舟IV号-库函数版)
- 37. TIM定时器 (STM32神舟IV号-库函数版)
- 38. TIM定时器PWM控制指示灯亮度 (STM32神舟IV号-库函数版)
- 39. IWDG独立看门狗 (STM32神舟IV号-库函数版)
- 40. 315M无线模块扫描 (STM32神舟IV号-库函数版)

- 41. 315M无线模块中断 (STM32神舟IV号-库函数版)
- 42. 硬件CRC循环冗余检验 (STM32神舟IV号-库函数版)
- 43. PVD电源电压监测 (STM32神舟IV号-库函数版)
- 44. STOP停止模式 (STM32神舟IV号-库函数版)
- 45. STANDBY待机模式 (STM32神舟IV号-库函数版)
- 46. SPI存储器W25Q16_W2XQ16 (STM32神舟IV号-库函数版)
- 47. TFT彩屏彩色刷屏 (STM32神舟IV号-库函数版)
- 48. TFT彩屏显示基本图形 (STM32神舟IV号-库函数版)
- 49. TFT彩屏显示单色图片 (STM32神舟IV号-库函数版)
- 50. TFT彩屏显示中文英文字符 (STM32神舟IV号-库函数版)
- 51. TFT彩屏显示秒表 (STM32神舟IV号-库函数版)
- 52. TFT彩屏显示RTC时钟 (STM32神舟IV号-库函数版)
- 53. TFT彩屏显示彩色图片 (STM32神舟IV号-库函数版)
- 54. TFT彩屏按键控制Cursor光标图标移动 (STM32神舟IV号-库函数版)
- 55. TFT彩屏触摸控制Cursor光标箭头移动 (STM32神舟IV号-库函数版)
- 56. TFT彩屏触摸控制画板 (STM32神舟IV号-库函数版)
- 57. CAN总线回环 (STM32神舟IV号-库函数版)
- 58. 双CAN收发测试 (STM32神舟IV号-库函数版)
- 59. 神舟IV号2.4G无线通信实验 (STM32神舟IV号-库函数版)
- 60. SD卡访问实验 (STM32神舟IV号-库函数版)
- 61. SD卡的FTA文件系统访问 (STM32神舟IV号-库函数版)
- 62. 神舟IV号STM32_USB_HID人机交互设备HID实验 (STM32神舟IV号-库函数版)
- 63. 神舟IV号STM32_USB_DFU固件升级 (STM32神舟IV号-库函数版)
- 64. DFU LED流水灯 [验证-STM32_USB_DFU固件升级] (STM32神舟IV号-库函数版)
- 65. 神舟IV号STM32_USB_SD读卡器 (支持SDHC) (STM32神舟IV号-库函数版)
大小: 2.79 MB
文件夹: Libraries, Output, Project
文件: 【MDK说明】.txt, 剔除MDK产生的过程文件.bat
- 66. 神舟IV号U盘访问 (显示U盘中图片) (STM32神舟IV号-库函数版)
舟IV号-库函数版)
- 67. 音乐播放器 (STM32神舟IV号-库函数版)
串口 (STM32神舟IV号-库函数版)
- 68. 音乐播放器 +模拟输入混音+音量控制+静音控制 (STM32神舟IV号-库函数版)
文件 (STM32神舟IV号-库函数版)
- 69. 以太网触摸屏STM32F107_ETH_LCD (神舟IV号) Telnet远程登录V1.1
- 70. 以太网触摸屏STM32F107_ETH_LCD (神舟IV号) HTTP网页V1.1
- 71. 以太网STM32F107_ETH_LCD (神舟IV号) TFTP文件传输V1.1
- 72. 以太网HTTP和TFTP协议实现IAP固件升级实验 (神舟IV号)V1.1
- 73. 以太网触摸屏STM32F107_ETH_LCD (神舟IV号) 全功能发布V1.1
- 74. UCOSII操作系统之单任务运行 (STM32神舟IV号-库函数版)
- 75. UCOSII操作系统之多任务运行 (STM32神舟IV号-库函数版)
- 76. uCOS_UCGUI_DEMO3.2寸LCD (STM32神舟IV号-库函数版)
- 77. uCOS_UCGUI_DEMO2.8寸LCD (STM32神舟IV号-库函数版)

STM32 神舟王 III 号最新开发板光盘资料下载:

<http://yunpan.cn/Q9GjDFJG38Nnx>

5. STM32神舟王103开发板 (STM32F103ZET)



- 01. LED点灯实验 (STM32神舟王103-寄存器版)
- 02. LED双灯闪烁实验 (STM32神舟王103-寄存器版)
- 03. LED三个灯同时亮同时灭 (STM32神舟王103-寄存器版)
- 04. LED流水灯 (STM32神舟王103-寄存器版)
- 05. STM32芯片32MHz频率下跑点灯程序 (STM32神舟王103-寄存器版)
- 06. STM32芯片40MHz频率下跑点灯程序 (STM32神舟王103-寄存器版)
- 07. STM32芯片72MHz频率下全速跑LED流水灯 (STM32神舟王103-寄存器版)
- 08. STM32芯片按键点灯初级 (STM32神舟王103-寄存器版)
- 09. STM32芯片按键点灯中级-增加了防抖的代码 (STM32神舟王103-寄存器版)
- 10. 最简单串口打印\$字符-初级STM32神舟王103-寄存器版)
- 11. 单串口打印www.armjishu.com字符-初级 (STM32神舟王103-寄存器版)
- 12. 单串口打印www.armjishu.com字符-中级 (STM32神舟王103-寄存器版)
- 13. 单串口打印www.armjishu.com字符-高级 (STM32神舟王103-寄存器版)
- 14. USART-COM1串口接收与发送实验-初级版 (STM32神舟王103-寄存器版)
- 15. USART-COM1串口接收与发送实验-中级版STM32神舟王103-寄存器版)
- 16. USART-COM1串口接收与发送实验-高级版STM32神舟王103-寄存器版)
- 17. LED流水灯 (神舟王103-库函数版)
- 18. BEEPER蜂鸣器 (神舟王103-库函数版)
- 19. KEY_LED按键与315M无线实验 (神舟王103-库函数版)
- 20. USART-COM1串口发送 (神舟王103-库函数版)
- 21. 串口收发一中断方式 (神舟王103-库函数版)
- 22. RS485总线实验 (神舟王103-库函数版)
- 23. CAN (LoopBack) 程序 (神舟王103-库函数版)
- 24. ADC模数转换 (神舟王103-库函数版)
- 25. EEPROM读写程序 (神舟王103-库函数版)
- 26. SPI FLASH(W25X16)读写程序 (神舟王103-库函数版)
- 27. SysTick系统滴答 (神舟王103-库函数版)
- 28. SRAM访问程序 (神舟王103-库函数版)
- 29. NOR FLASH 测试程序 (神舟王103-库函数版)
- 30. NAND FLASH访问程序 (神舟王103-库函数版)
- 31. Calendar实时时钟与年月日 (神舟王103-库函数版)
- 32. 2.4G无线通信实验 (神舟王103-库函数版)
- 33. FM收音机 (神舟王103-库函数版)
- 34. 以太网卡 (神舟王103-库函数版)
- 35. SD读卡器 (神舟王103-库函数版)
- 36. MP3播放器实验 (神舟王103-库函数版)
- 37. SD读卡器 (神舟王103-库函数版)
- 38. MP3播放器实验 (神舟王103-库函数版)
- 39. TFT彩色液晶屏只显示红色 (神舟王103-库函数版)
- 40. TFT彩色液晶屏只显示蓝色-带串口打印 (神舟王103-库函数版)
- 41. TFT彩色液晶屏显示英文字 (神舟王103-库函数版)
- 42. TFT彩色液晶屏显示中文字 (神舟王103-库函数版)
- 43. TFT彩色液晶屏混合显示中英文字 (神舟王103-库函数版)
- 44. TFT彩屏显示图片 (神舟王103-库函数版)
- 45. TFT彩屏显示图片刷屏 (神舟王103-库函数版)
- 46. TFT触摸屏 (神舟王103-库函数版)
- 47. UCOSII操作系统之单任务运行 (神舟王103-库函数版)
- 48. UCOSII操作系统之多任务运行 (神舟王103-库函数版)
- 49. UCOSII+UCGUI彩色界面 (神舟王103-库函数版)

STM32 神舟王 103ZET 最新开发板光盘资料下载:

<http://pan.baidu.com/share/link?shareid=2492524018&uk=875911564>

6. STM32神舟王207开发板（STM32F207ZGT）

The screenshot shows a web browser displaying the user manual for the STM32 God舟王 207ZGT Development Board. The page title is "STM32 神舟王 207ZGT 开发板用户手册". Below the title, it says "2013 年 9 月版本 V2.0 作者: WWW.ARMJISHU.COM". A section titled "STM32神舟ARM系列技术开发板产品目录:" lists various STM32 development boards. The "STM32神舟王207系列(STM32F207ZGT核心板)" section is highlighted. Below this, there is a promotional image for the STM32 God舟王 series development board.

STM32 神舟王 207ZGT 开发板用户手册

2013 年 9 月版本 V2.0 作者: WWW.ARMJISHU.COM

STM32神舟ARM系列技术开发板产品目录:

- 【神舟51单片机开发板（51+ARM）开发板】
- 神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏
- 神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- 神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏
- 神舟IV号: STM32F107VCT6 + 3.2" TFT 触摸彩屏
- STM32核心板: 四层核心板 (STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列(STM32F103ZET核心板)
- 【神舟王207系列(STM32F207ZGT核心板)】
- 神舟王407系列(STM32F407ZGT/407IGT核心板)
- 神舟王全系列(STM32F103ZET/207ZGT/407ZGT核心板): 全功能底板 (支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
- 神舟 51 开发板 (STC 51 单片机+STM32F103C8T6 核心板): 全功能底板 (支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)

STM32神舟系列开发板

品质卓越 无与伦比

1. 版本升级，系统更加完善
2. 预留拓展接口，可以拓展更多模块
3. 丰富的接口以及配套齐全的几百页工程师手写文档
4. 原创STM32从初学者从入门到精通教程，与开发板
配套，让您学习起来更顺手

-
- 01. LED点灯实验 (STM32神舟王207-寄存器版).rar
 - 02. LED双灯闪烁实验 (STM32神舟王207-寄存器版).rar
 - 03. LED三个灯同时亮同时灭 (STM32神舟王207-寄存器版).rar
 - 04. LED流水灯 (STM32神舟王207-寄存器版).rar
 - 05. STM32芯片32MHZ频率下跑点灯程序 (STM32神舟王207-寄存器版).rar
 - 06. STM32芯片40MHZ频率下跑点灯程序 (STM32神舟王207-寄存器版).rar
 - 07. STM32芯片72MHZ频率下全速跑LED流水灯 (STM32神舟王207-寄存器版).rar
 - 08. STM32芯片按键点灯初级 (STM32神舟王207-寄存器版).rar
 - 09. STM32芯片按键点灯中级-增加了防抖的代码 (STM32神舟王207-寄存器版).rar
 - 10. LED流水灯 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 11. BEEPER蜂鸣器 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 12. RS232串口print 打印输出 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 13. 按键与315M无线实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 14. 串口数据收发实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 15. 神舟王LCD彩屏显示红色 (神舟王STM32F207-库函数版).rar
 - 16. 神舟王LCD彩屏显示蓝色 (神舟王STM32F207-库函数版).rar
 - 17. TFT彩色液晶屏显示英文字 (神舟王STM32F207-库函数版).rar
 - 18. TFT彩色液晶屏显示中文字 (神舟王STM32F207-库函数版).rar
 - 19. TFT彩色液晶屏混合显示中英文字 (神舟王STM32F207-库函数版).rar
 - 20. TFT显示图片 (神舟王STM32F207-库函数版).rar
 - 21. TFT显示图片并刷屏 (神舟王STM32F207-库函数版).rar
 - 22. ADC模数转换 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 23. EEPROM读写程序 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 24. SPI串行FLASH_W25X16_W25Q16 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 25. SysTick系统滴答 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 26. SRAM访问程序 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 27. NOR_FLASH访问 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 28. NAND_FLASH访问 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 29. DS18B20温度传感实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 30. 神舟王_LCD触摸屏 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 31. 2.4G无线通信实验两个模块间通信 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 32. Calendar实时时钟与年月日与闹钟 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 33. Calendar实时时钟与农历与闹钟 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 34. PS2键盘实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 35. IR红外线接收发送实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 36. VS1003示例音乐播放 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 37. 神舟王读取U盘实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 38. 神舟王SD卡USB读卡器实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 39. 神舟王DFU固件升级实验 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 40. 神舟王USB鼠标_按键控制 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 41. 神舟王USB鼠标_触摸屏控制 (神舟王STM32F207ZGT-库函数版本) Lib_V1.1.0.rar
 - 42. 神舟王以太网httpserver网页实验 (神舟王STM32F207ZGT-库函数版本).rar
 - 43. 神舟王以太网tftserver文件传输 (神舟王STM32F207ZGT-库函数版本).rar
 - 44. 神舟王FreeRTOS操作系统httpserver_netconn网页 (神舟王STM32F207ZGT-库函数版本).rar
 - 45. 神舟王FreeRTOS操作系统httpserver_socket网页 (神舟王STM32F207ZGT-库函数版本).rar
 - 46. 神舟王FreeRTOS操作系统udptcp_echo_server_netconn (神舟王STM32F207ZGT-库函数版本).rar

STM32 神舟王 207VGT 最新开发板光盘资料下载:

<http://pan.baidu.com/share/link?shareid=3691145012&uk=1036597165>

7. STM32神舟王407开发板（STM32F407ZGT）

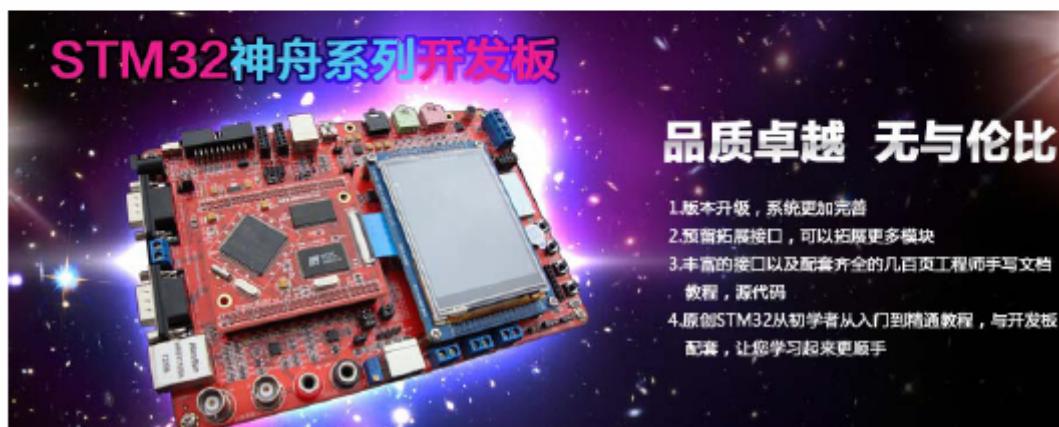


STM32 神舟王 407ZGT 开发板用户手册

2013 年 8 月版本 V2.0 作者: WWW.ARMJISHU.COM

STM32神舟ARM系列技术开发板产品目录:

- 【神舟51单片机开发板（51+ARM）开发板】
- 神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏
- 神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- 神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏
- 神舟IV号: STM32F107VCT6 + 3.2" TFT 触摸彩屏
- STM32核心板: 四层核心板(STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列(STM32F103ZET核心板)
- 神舟王207系列(STM32F207ZGT核心板)
- 【神舟王407系列(STM32F407ZGT/407IGT核心板)】
 - 神舟王全系列(STM32F103ZET/207ZGT/407ZGT核心板): 全功能底板(支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
 - 神舟 51 开发板 (STC 51 单片机+STM32F103C8T6 核心板): 全功能底板(支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)



- 01. LED点灯实验 (神舟王STM32F407-寄存器版)
- 02. LED双灯闪烁实验 (神舟王STM32F407-寄存器版)
- 03. LED三个灯同时亮同时灭 (神舟王STM32F407-寄存器版)
- 04. LED流水灯 (神舟王STM32F407-寄存器版)
- 05. STM32芯片32MHZ频率下跑点灯程序 (神舟王STM32F407-寄存器版)
- 06. STM32芯片40MHZ频率下跑点灯程序 (神舟王STM32F407-寄存器版)
- 07. STM32芯片全速72MHZ频率下跑点灯程序 (神舟王STM32F407-寄存器版)
- 08. STM32芯片按键点灯初级 (神舟王STM32F407-寄存器版)
- 09. STM32芯片按键点灯中级-增加了防抖的代码 (神舟王STM32F407-寄存器版)
- 10. LED流水灯 (神舟王STM32F407-库函数版)
- 11. 蜂鸣器 (神舟王STM32F407-库函数版)
- 12. 串口printf打印输出 (神舟王STM32F407-库函数版)
- 13. 315M无线模块 (神舟王STM32F407-库函数版)
- 14. 串口数据收发实验 (神舟王STM32F407-库函数版)
- 15. 神舟王LCD彩屏显示 (神舟王STM32F407-库函数版)
- 16. ADC模数转换彩屏显示 (神舟王STM32F407-库函数版)
- 17. EEPROM读写程序彩屏显示 (神舟王STM32F407-库函数版)
- 18. SPI_FLASH_W25X16_W25Q16 (神舟王STM32F407-库函数版)
- 19. SysTick系统滴答 (神舟王STM32F407-库函数版)
- 20. SRAM访问程序 (神舟王STM32F407-库函数版)
- 21. NOR_FLASH访问 (神舟王STM32F407-库函数版)
- 22. NAND_FLASH访问 (神舟王STM32F407-库函数版)
- 23. DS18B20温度传感实验 (神舟王STM32F407-库函数版)
- 24. 神舟王_LCD触摸屏 (神舟王STM32F407-库函数版)
- 25. 2.4G无线通信实验两个模块间通信 (神舟王STM32F407-库函数版)
- 26. Calendar实时时钟与年月日与闹钟 (神舟王STM32F407-库函数版)
- 27. Calendar实时时钟与农历与与闹钟 (神舟王STM32F407-库函数版)
- 28. PS2键盘实验 (神舟王STM32F407-库函数版)
- 29. IR红外线接收发送实验 (神舟王STM32F407-库函数版)
- 30. VS1003示例音乐播放 (神舟王STM32F407-库函数版)
- 31. 神舟王407读取U盘实验 (神舟王STM32F407-库函数版)
- 32. 神舟王407使用SD卡USB读卡器实验 (神舟王STM32F407-库函数版)
- 33. 神舟王407使用DFU固件升级 (神舟王STM32F407-库函数版)
- 34. 神舟王407使用USB鼠标_按键控制 (神舟王STM32F407-库函数版)
- 35. 神舟王407使用USB鼠标_触摸屏控制 (神舟王STM32F407-库函数版)
- 36. 神舟王407以太网tftpserver文件传输 (神舟王STM32F407-库函数版)
- 37. 神舟王407之FreeRTOS操作系统httpserver_netconn网页 (神舟王STM32F407-库函数版)
- 38. 神舟王FreeRTOS操作系统httpserver_socket网页 (神舟王STM32F407-库函数版)
- 39. 神舟王FreeRTOS操作系统udptcp_echo_server (神舟王STM32F407-库函数版)

STM32 神舟王 407 最新开发板光盘资料下载:

<http://pan.baidu.com/share/link?shareid=2333589004&uk=2285178252>

8. STM32神舟王407IGT开发板+130W摄像头 (STM32F407IGT)

9. STM32神舟王407IGT开发板+130W摄像头（STM32F407IGT）

1 / 571 | 80% | 搜索 |

STM32 神舟王 407IGT 核心板发板用户手册

2013 年 10 月版本 V2.0 作者: www.armjishu.com

STM32神舟ARM系列技术开发板产品目录:

- 【神舟51单片机开发板（51+ARM）开发板】
- 神舟I号: STM32F103RBT6 + 2.8" TFT 触摸彩屏
- 神舟II号: STM32F103VCT6 + 3.2" TFT 触摸彩屏
- 神舟III号: STM32F103ZET6 + 3.2" TFT 触摸彩屏
- 神舟IV号: STM32F107VCT6 + 3.2" TFT 触摸彩屏
- STM32核心板: 四层核心板(STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列(STM32F103ZET核心板)
- 神舟王207系列(STM32F207ZGT核心板)
- 【神舟王407系列(STM32F407ZGT/407IGT核心板)】
- 神舟王全系列(STM32F103ZET/207ZGT/407IGT核心板): 全功能底板(支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
- 神舟 51 开发板 (STC 51 单片机+STM32F103CST6 核心板): 全功能底板(支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)

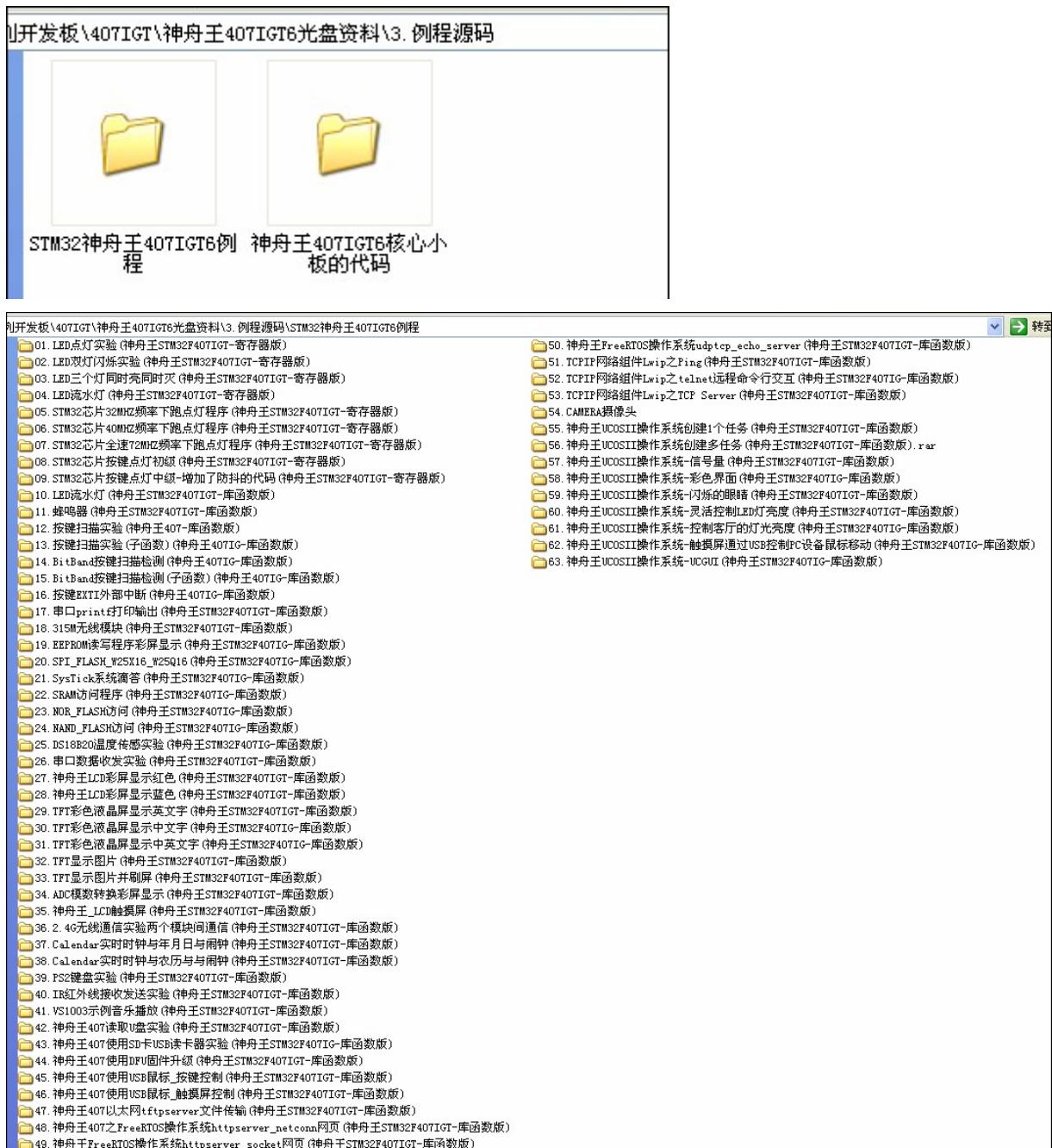


STM32神舟系列开发板
品质卓越 无与伦比

1.板卡升级，系统更加完善
2.增加拓展接口，可以拓展更多模块
3.丰富的接口以及配套齐全的几百页工程级手写文档
教程，源代码
4.帮助STM32从初学者从入门到精通教程，与开发板
配套，让您学习起来更顺手

系列开发板\407IGT\神舟王407IGT6光盘资料

| | | | | |
|----------------------|--------------|--|--|---|
| 1. 入门必读 | 2. 硬件原理图 | 3. 例程源码 | 4. 软件工具 | 5. 软件参考资料 |
| 6. 硬件参考资料 | 7. STM32视频教程 | 1. 神舟王 STM32F407IGT用户... Adobe Acrobat Do... WinRAR 压缩文件 1,250,220 KB | 3. 例程源码 WinRAR 压缩文件 1,250,220 KB | 3. 神舟王407IGT6光盘 资料(不含例程和... WinRAR 压缩文件 |
| 资料下载 文本文件 1 KB | | | | |



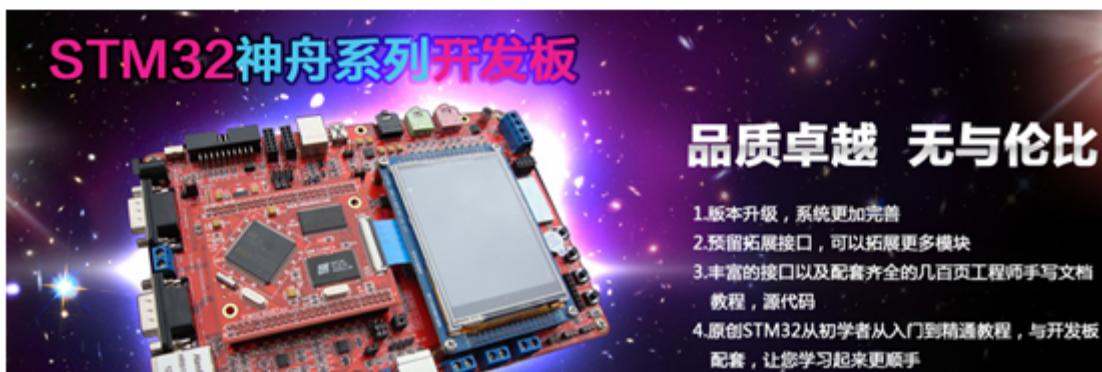
10. STM32神舟王439IGT开发板+130W摄像头（STM32F439IGT）

STM32 神舟王 439IGT 核心板发板用户手册

2013 年 11 月版本 V1.0 作者: www.armjishu.com

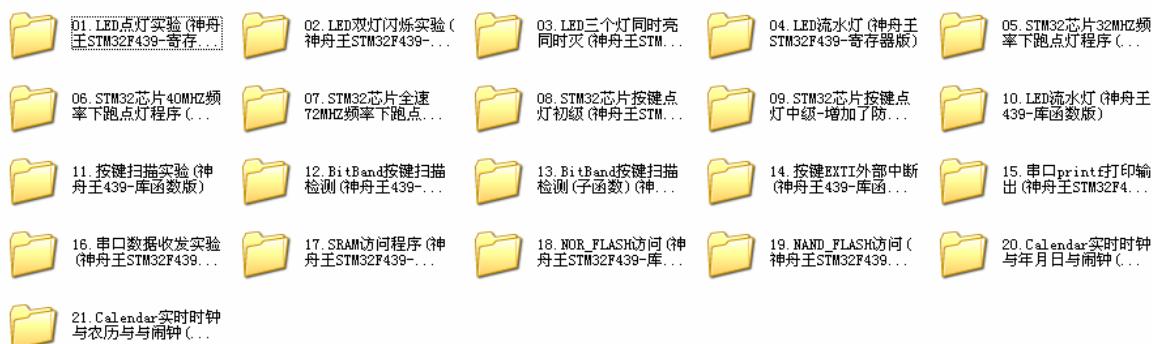
STM32神舟ARM系列技术开发板产品目录:

- 【神舟51单片机开发板（51+ARM）开发板】
- 神舟I号: STM32F103RBT6 + 2.8”TFT 触摸彩屏
- 神舟II号: STM32F103VCT6 + 3.2”TFT 触摸彩屏
- 神舟III号: STM32F103ZET6 + 3.2”TFT 触摸彩屏
- 神舟IV号: STM32F107VCT6 + 3.2”TFT 触摸彩屏
- STM32核心板: 四层核心板 (STM32F103ZET+207ZGT+407ZGT+407IGT)
- 神舟王103系列 (STM32F103ZET核心板)
- 神舟王207系列 (STM32F207ZGT核心板)
- 神舟王407系列 (**STM32F407ZGT/407IGT核心板**)
- 【**神舟王439系列 (STM32F439ZGT/439IGT核心板)**】
- 神舟王全系列 (STM32F103ZET/207ZGT/407IGT核心板): 全功能底板 (支持MP3, 以太网, 收音机, 无线, SRAM, Nor/Nand Flash, 鼠标, 键盘, 红外接收, CAN, 示波器, 电压表, USB HOST, 步进电机, RFID物联网等)
- 神舟 51 开发板 (STC 51 单片机+STM32F103C8T6 核心板): 全功能底板 (支持音频播放, 无线, 鼠标, 键盘, 红外收发, CAN, 温度传感器, 直流电机, 步进电机, 实时时钟, 两路 485, 两路继电器, 小喇叭, 热敏光敏电阻, RFID物联网等)

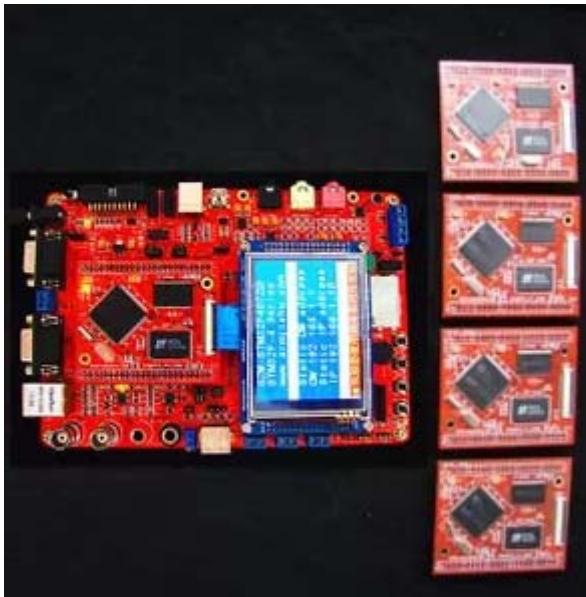




开发板\439igt\2. 神舟王439IGT6核心小板的代码



11. STM32F103/207/407/439多个核心板共同神舟王底板---真正一板变多板



- 1) 神舟王 STM32F103ZET 核心板 + 神舟王底板 + 液晶屏
- 2) 神舟王 STM32F207ZGT 核心板 + 神舟王底板 + 液晶屏
- 3) 神舟王 STM32F207IGT 核心板 + 神舟王底板 + 液晶屏
- 4) 神舟王 STM32F407ZGT 核心板 + 神舟王底板 + 液晶屏
- 5) 神舟王 STM32F407IGT 核心板 + 神舟王底板 + 液晶屏
- 6) 神舟王 STM32F417IGT 核心板 + 神舟王底板 + 液晶屏
- 7) 神舟王 STM32F439IGT 核心板 + 神舟王底板 + 液晶屏

附件3：STM32神舟团队20年工作经验心得总结

首先，如果你有幸看到这篇文章，千万不要试图在 2 个小时内阅读完，就算你 2 个小时阅读完，我相信你也不会理解里面讲解的精华之处，我相信，你应该将此文章，慢慢品尝，这绝对是一篇需要品尝 2~3 天，再结合自己过往的经验，加上自己的思考，我相信会对你不仅仅是技术能力，甚至包括整体的思维方式都会有一个非常大的提高。此篇文章摘抄于 www.armjishu.com 的坛主 jesse，如有需要转载，请注明作者，谢谢大家。

结合这篇文章，再结合 STM32 神舟系列开发板一些学习，可能会更加加深对嵌入式概念的理解。

我写这篇文章的目的，是用本人 20 年的嵌入式经验呈现给大家一副完整的产品，项目开发蓝图，用本人多年经的历总结了一些教训无私的分享给各位，希望各位今后能站在本人的肩膀之上，

少走弯路，多为公司，为个人多做贡献，那我的愿望就达到了，也同时希望能看到大家反馈和回复，留个脚印，留下你的见解和智慧，为后人乘凉打点基础，先在这谢谢各位了。

那么由此开始我们充满知识的旅程吧，最重要的一点，就是在一个产品或项目的开发过程中，如果没有明确的目标，那么成功将无从谈起，做任何事的第一步必须明确目标。

1) 需求定义

需求定义用来描述产品的基本功能，对于公司来说，需求一般由该公司的市场销售部门或该公司主要客户来制定；而对小公司或爱好者（就像 armjishu.com 里的爱好者一样），技术人员可以自己负责定义需求，并撰写成文档；对于 STM32 神舟系列开发板来说，主要就是提供各种接口，为大家开发产品时提供借鉴！

通常需求定义是围绕以下几个因素而来：

- 1) 系统的用途（定义需要系统实现的各种功能）
- 2) 实际输入输出是何种方式实现的（为元器件的选型做参考）
- 3) 系统是否需要操作界面（涉及软件层操作系统的选型）

其实对小型的嵌入式产品来说，定义需求是非常关键的，因为需求清楚了，就可以避免后续开发过程中出现的诸如随机存储器（RAM）容量不足或所选的 CPU 速度不能满足处理的需要等一系列问题。

下面举个简单的实际例子，供大家来参考：

系统描述：用于从化温泉的水泵换水系统（用 STM32 神舟 III 号开发板模拟实现）

电源输入：使用来自于变压器的 9V~12V 直流电

水泵功率：375W

- 1) 使用单相交流电机，由机械电气进行控制
- 2) 如果温泉池处于低水位，则输入开关闭合信号，以禁止水泵继续运行
- 3) 用户可以自由设置水泵运行或关闭的时间长度
- 4) 除了自动设置控制外，还需要提供一种人工装置来允许维护人员灵活控制水泵进行维修
- 5) 水泵开启/关闭/人工干预的时间可以 30 分钟为单位，在 30 分钟到 23 小时的范围内进行调节
- 6) 显示设备可以指示水泵的开关状态，剩余时间，以及水泵是否处于人工干预模式
- 7) 具备监视低水位的功能，并显示在屏幕上

如果需要商用，那么除了上面给出的功能要求外，其设计文档中还要包括电磁干扰（EMI）和电磁兼容性（EMC）认证、安全认证以及使用环境（包括环境温度、湿度、盐雾腐蚀等）等方面的需求。

实际上，以上的需求确定之后，接下来就是要考虑选择一款合适的 CPU 来满足和实现系统的功能，那么我们就要将上述 7 点用户能够理解的需求转化成我们专业领域的需求，转化如下，大家可以参

考一下：

a. 处理或更新输入输出信号的速率究竟需要多快？

解释：目前嵌入式处理器的主频一般都在几十兆到几百兆不等，单片机的主频一般是几十兆，STM32 神舟系列开发板的 CPU 都是 72MHZ，有的 ARM9，ARM11 处理器可以到几百兆；我们主要看这个产品是否需要对大量数据进行处理，或是否需要对缓冲区进行频繁操作，是否有类似的占用 CPU 资源的工作要做，这就决定我们要选择一款合适的处理器来让该产品得到最佳的性能。

b. 是否可使用单片集成电路（专用 IC）或 FPGA 来完成数据处理？

解释：如果可以的话，就不一定要选择处理器来做，用这些专业芯片就能替代

c. 系统是否有大量的用户输入输出操作（如对开关和显示设备进行频繁操作）？

解释：如果有的话，要在处理器选型的时候考虑这些因素，选择一款能够满足以上要求的 CPU.

d. 系统与其他外部设备之间需要使用何种接口？

解释：这也是需要评估处理器的一个关键问题，选择具备这些接口功能的处理器会方便于我们的电路设计以及软件编程

e. 设计完成后是否有可能需要进行改动，或在设计过程中系统需求是否可能出现变化？我们的设计是否能适应系统需求的变化？

解释：要避免选择的处理器刚好满足当前要求，这样当以后任务要求逐渐提高，处理器性能如果还有一定空间的话，那么就可以重用目前的产品；第二个就是要选择不会即将停产的芯片，很多处理器用得很广泛，可以借鉴的资料也很多，但是很可能这款芯片已经在市场上流行很长时间了，芯片厂商已经推出更新换代的替代品了，如果你选择了这款芯片，很可能 1, 2 年后就买不到这款处理器芯片了，导致不得不重新选择新的处理器，重新设计产品，这样的既耗费时间，金钱，更消耗人力，延误市场的战机。

与日常生活中的大多数事务一样，设计一个嵌入式产品的过程也必须从确定目标开始，对生产的产品进行明确定义。对产品进行定义主要是对产品是什么和能有什么功能进行描述，其次是在我们的整个开发过程中，应该要撰写一些开发文档，大概的框架的如下：

- 1) 产品需求文档：描述产品的特性
- 2) 功能需求文档：描述产品必须具备的功能
- 3) 工程说明文档：描述系统实现的方法和满足需求的手段
- 4) 硬件说明文档：对有关硬件进行描述
- 5) 软件或固件说明文档：描述特定处理器下设计微程序以及固件的方法
- 6) 测试说明文档：描述必须测试的项目和验证系统正常运行的方法

2) 处理器的选择

1) 需要使用的 I/O 管脚数量

多数处理器都是使用内存和外部管脚来控制输入输出设备的，通常处理器都会有内置 ROM 和 RAM 的，如果内置的内存就已经满足需要，那么处理器就可以节省产生引用外部存储器信号的引脚，这样处理器可为输入输出提供较多的设备管脚（某些处理器支持外部 RAM 或 ROM 的使用，但对外部存储器进行访问时，处理器一般需要占用 8 条到 10 条 I/O 管脚）。

还有，有些处理器带有专用的内部定时时钟，这类时钟也需要使用一个端口管脚来实现某些定时功能；某些处理器中还具有漏极输出和高电流输出能力，可以方便的直接驱动继电器或电磁铁线圈，而不再需要额外驱动硬件的支持。

当对处理器 I/O 管脚进行计数时，我们一定要把使用处理器内部功能（如串行接口和定时器等）时限制使用的某些管脚考虑在内。

2) 需要使用的接口数量 www.armjishu.com

嵌入式处理器的主要功能是与应用环境中的硬件进行交互操作，这不仅需要外部硬件对接口具有实时处理能力，而且还要求处理器必须以足够快的速度对接口数据进行有效处理。

举例来说，STM32 神舟系列开发板的 CPU 是 ST 公司出品的一款工业级微处理器，它基于 CORTEX M3 的核心，处理主频可达 72MHZ，同时处理器内部配置了 USB、SPI、IIC 等接口，像 STM32 神舟 III 号的 103 处理器还支持 Ethernet 等输出接口，其目的是更方便的利用这些接口开发出嵌入式产品。

需要注意的是，由于许多处理器具有的局限性没有在处理器技术资料中给予足够的说明，因此一定要仔细阅读处理器的指标说明。例如，在阅读资料的过程中发现，该资料可能会说明其串行接口可以在最高波特率下工作，但仔细研究该处理器的指标数据时，可能会发现并非该串口接口的所有操作模式都可以在最大波特率下运行。

深入了解并明确接口要求的方法：可以自己动手编写一些程序来对接口进行实际测试，以确认某种处理器是否可以满足应用的要求；因为，确认某个处理器是否可以满足接口要求并非是一件简单的任务。

3) 需要使用的内存容量

决定内存容量的大小是嵌入式产品设计过程中的一个基本步骤，如果对所需内存容量估计过高，那么我们就有可能会选择成本较高的解决方案；反之，如果低估了所需内存容量，就有可能因系统需要重新设计而导致项目不能按时完工。

a. RAM 和 ROM 的区别：存储器分为随机存储器（RAM）和只读存储器（ROM）两种。其中 ROM 通常用来固化存储一些生产厂家写入的程序或数据，用于启动电脑和控制电脑的工作方式。而 RAM 则用来存取各种动态的输入输出数据、中间计算结果以及与外部存储器交换的数据和暂存数据。设备断电后，RAM 中存储的数据就会丢失。

b. 随即存储器（RAM）的选择：RAM 容量的预测是比较直观的，我们只需把所有变量数目与所有内部缓冲区的容量以及先入先出（FIFO）队列长度和堆栈长度直接相加，就能得到所需 RAM 容量的总数。如果所需内存容量超出这类处理器的寻址范围，那么只能通过增加外部 RAM 来满足需求；然而，增加外部 RAM 的同时将会占用一定数量的 I/O 管脚来对扩展内存进行寻址，这种扩展往往会影响到处理器来实现应用的初衷。

需要注意的一个问题是，某些微处理器限制 RAM 的使用，这种限制的目的是为了借用部分内存存储器作为内部寄存器组使用。除了以上因素外，所使用的开发语言也对所需 RAM 容量有一定的影响，某些效率较低的编译程序可能会占用大量宝贵的 RAM 空间。

c. 只读存储器（ROM）的选择：系统所需 ROM 的大小应该是系统程序代码与所有基于 ROM 的数据表容量之和。预测所需 ROM 空间容量比较困难的部分是预测程序代码的长度，解决这类问题的方法只能是随着经验的逐步积累来提高预测精度。

然而，最重要的并不是精确计算程序的代码长度，而是要清楚地估算代码长度的上限。根据经验，如果 80% 的 ROM 空间被代码占用的话，那么就太拥挤了，除非能确保系统需求不会有任何变化，否则至少要为可能发生变化保留足够的备用 ROM 空间。

在多数情况下，我们可以试着在 ROM 中写入一部分程序代码，以便观察代码占用空间的情况，对于带有内部 ROM 的微处理器系统来说，系统程序都只能占用有限的程序存储器空间。

d. 经验之谈：ROM 与 RAM 使用情况相类似，程序代码长度与所选用的开发语言有关。举例来说，使用汇编语言编制的程序要比使用 C 语言编制的程序占用少得多的空间。

对于追求低成本的小型系统来说，一般不提倡使用高级程序设计语言；这是因为虽然高级语言在使用、调试以及维护方面来的比较容易，但同时这类语言需要占用更多的内存空间和大量的处理器时钟周期。

如果开发语言选择不当，其后果可能是把一个简单、低成本的单片机系统变为一个需要使用配置若
嵌入式专业技术论坛 (www.armjishu.com) 出品
页

千兆字节 RAM 空间的 64 位嵌入式处理器系统。

4) 需要使用的中断数量

中断的主要用途是向中央处理器通报当前发生的某类特殊事件，这类事件包括诸如定时器超时事件、硬件引发的事件等。

需要强调的是，多数系统设计师经常过多地使用中断功能，实际上，中断的主要作用只是中断现行程序的执行，中断最适用于必须要求中央处理器立即提供服务的事件。

在需要设计和使用中断的情况下，一定要首先确认实际需要的中断数量，然后必须考虑到系统内部占用的中断资源，如果需要使用的中断资源超出了处理器可以接收的中断数量，我们就应借助于某些特殊手段来减少所需中断信号的数量。

5) 实时处理方面的考虑 www.armjishu.com

实时处理是一个涉及范围很广的题目，其主要内容与系统的处理速度有密切联系，实时事件是嵌入式微处理器需要关注的主要任务。

例如：处理器跟串口进行通信时，通常通过上层软件（为了保证实时性，进行任务切换的时间足够短），然后再占用处理器去执行从串口拿数据的任务，并且要保证处理器的速率比串口速率快，那么处理器可以以最快的速度反应并处理串口的相关的任务，这样就可以达到最大的实时性；

另一方面，如果处理器本身就内置了串口控制器、或 DMA、或 LCD 的控制器等，那么它就可以保证直接使用这些处理器内置的接口去控制串口、液晶屏等对象，以达到最大的实时性能。

6) 该厂商是否提供好的开发工具和环境

选择一款新的处理器，很可能就要使用一个新的开发工具和开发环境，包括软件的编译环境等；对于开发日程安排比较紧张的项目来说，开发人员往往无法抽出专门的时间来研究，熟悉新的开发工具，从而也无法全面掌握开发工具的使用技巧。

并且，有的开发工具价格也比较昂贵，而且很可能只能从制造商那里购买，还有仿真工具也是需要付费的，这些对我们在选择一款处理器的时候，是都应该考虑进去的成本因素。

7) 处理器速度方面的考虑

主要考虑几个细节问题：

1) 处理器速度与处理器时钟之间的关系

例：单片机 8031 为例，由该处理器可以适应 12MHz 频率的输入时钟，因此就可以认为它是一个速度为 12MHz 的处理器了吗？不是，实际上，由于该处理器内部逻辑电路执行每条指令需要多种不同频率的时钟脉冲，因此该处理器内部时钟电路要对输入的 12MHz 时钟 12 分频处理；最终为处理器提供的只是 1MHz 主频。

有的时候，80MHz 主频的处理器（80MHz 输入时钟，80MHz 执行速度）要比 200MHz 主频的处理器（200MHz 输入时钟，50MHz 执行速度）执行速度要快得多。

2) 处理器指令系统

如果不需要执行复杂数学运算的应用，那么 RISC 指令集的处理器要快；如果执行比较复杂的操作，则 CISC 指令集的处理器速度要更快。

3) 芯片结构体系

现在有的芯片是将多个不同功能的核封装到一个芯片 IC 中，定制某种特定的功能，比如 DSP，其中包括用于实现数字解码、乘法运算的硬件乘法器和移相器等；然而，这类处理器也由其自身局限，往往在执行某些普通操作之前必须要使用额外的指令来把 RAM 中的数据放入内部寄存器，相比之下，一般处理器只允许对 RAM 中的数据进行直接访问。

8) 只读存储器（ROM）的选择

多数工程项目在其开发阶段一般使用可擦写可编程只读存储器（EPROM）或快速存储器（Flash Memory）；这类可擦写可重复写入存储器的主要优点是可多次使用。一旦产品研制完毕，就可以用一次写入设备（OTP）来取代 EPROM 存储器，一次性写入器件的外观与封装几乎与 EPROM 完全一样，惟一不同之处就是其表面没有擦出窗口，并且价格要比 EPROM 低很多。

但是，另外一种情况，如果该产品今后需要升级固件，或在线编程，那么我们还是应该选择可擦写可编程的存储器。

还有一种是非易失的存储器，例如制造一台电视机，就有可能需要该设备具有记忆上次观看最后一个频道的功能，即使在切断电源后，该频道信息也不会丢失。

总结：所以，根据不同的产品选择不同的存储器也是一门很讲究的学问。

9) 电源的要求

在某些设计中方案中，电源根本不存在问题，对电源唯一的要求就是可以为电路正常供电；实际上，选择电源主要要考虑三个方面的问题：

1) 要注意设计方案中是否对电源的供电方式有所限制，例如，是否像大多数家用电器那样需要使用
嵌入式专业技术论坛（www.armjishu.com）出品
第 890 页，共 900 页

屋内墙上的电源插座供电，或是是使用 USB 接口供电

2) 看系统是否需要使用电池供电方式，如果这样，我们就要考虑选择那种对驱动电流要求不高的处理器，然后再为其选择合适的电池。

3) 休眠电流：许多微处理器都支持低功率运行模式，在这种模式下，系统的 CPU 处理器将处于休眠状态，同时所有外部设备的电源供电都被暂时切断，以便减少系统的电能消耗；某些微处理器在这种方式下需要的维持电流极小，但也有一些微处理器在这种方式下并不能节省多少功率；不管怎样，我们都要对系统在节点模式下的工作时间有一个估测，以便对具体情况选择使用的电池。

总之，无论哪种情况，我们都要对系统需要的供电总功率做到心中有数。

10) 设备工作环境的要求

环境要求主要内容是考虑温度，湿度等；如果系统必须在温度范围较大的环境下运行，诸如用于军事设备或汽车的控制系统，那么处理器可选择的范围就要小得多；

并且由于大范围温度变化的设备通常比较昂贵，因此在设计过程中就不能再根据一般工业级器件的价格来制定预算。

11) 使用周期成本

如果我们的产品是 stm32 神舟开发板，在一般情况下，可以不必考虑在用户现场对 stm32 神舟开发板程序进行修改的问题，也不用为是否可以得到设备备件而着急，这是因为 stm32 神舟开发板是一种学习型的消费产品，仅仅只是一款开发板而已。

换句话说，如果我们的产品是价值几万块的工业设备并且需要常年不断地运行，那么我们在产品设计过程中就必须从长计议了：

- a. 首先，我们需要选择一种处理器或存储体系结构都可以升级的器件
- b. 考虑到程序升级的可能，我们还要选择较大容量的内存
- c. 最后要注意的则是所选处理器是否可以长期供货，这一点的重要性远远大于处理器的价格

除了上面的考虑之外，使用周期成本也是在设计之初要考虑的因素。总的来说，生产的部件越多，则可以接受的前期开发成本也就越大。如果产品是 mp3，我们可能会选择一个低价微处理器，同时投入一大笔钱来开发控制 mp3 的软件。

但如果我们的产品是价格昂贵的工业用设备，那么在产品的使用期内，该设备的销售量将只有几百台，毫无疑问，开发这种产品最重要的就是降低开发成本（降低开发成本而不是硬件成本！！！）；除此之外，工业产品的成本也不像家用电器或消费电子产品那么敏感。综上所述，开发工业产品当然要选择一种便于进行开发并且有助于缩短开发过程的处理器。

12) 处理器相关资料是否丰富

如果该款处理器在市场上已经用得很广了，那么我们可以获取更多的相关资料，观察人家的产品是如何使用处理器的，也能在网络上找到不少的相关的设计资料以及相关技术主题，这样就进一步降低了技术门槛，确保了使用该处理器做产品可行性，减低了风险；例如 STM32 神舟 III 号开发板就有针对该板子有个 700 多页的手册文档，如果我们选择 STM32 芯片来开发产品的话，借助详细资料开发起来就轻松了，达到事半功倍的效果。

反之，如果是厂商全新推出的处理器，因为市场上还没有可以借鉴的产品，我们就只能从全英文的芯片手册开始阅读，了解这款芯片，这样开发周期不仅变长，而且不可预知的风险也很大。

3) 开发成本的预测和估计

大多数项目或产品都有专人负责预测整个过程的开发成本，对于任何项目来说，其开发成本主要包括人力和材料开销。

预测开发成本在很大程度上需要根据经验，这也是为什么大型公司一般指定有经验的高级工程师来完成这一任务的原因，除了人力和材料的开销之外，总结下来，还有以下的开销：

- 1) 人力成本（开发人员、管理人员、销售人员、其他行政等辅助人员）的开销
- 2) 材料（硬件物料和损耗，有时候需要投几次 PCB 版才把产品稳定下来）的开销
- 3) 开发系统和开发工具软件的开销
- 4) 硬件工具的开销（例如示波器、仿真器等）

对于整个项目来说，上述的开销将直接可能导致产品成本增加，其中人力成本最为关键，尤其是在中国，呵呵。

4) 产品开发设计文档（需要包括硬件和软件两个方面）

1. 硬件文档撰写思路

- 1) 首先是需求定义或产品规格：

如果这些是产品最终目标的话，那么产品对硬件和软件的要求就是技术方案的最终目标；对硬件和软件的要求是从定义用户界面和系统功能开始的。www.armjishu.com

- 2) 其次，根据需求，系统整体定义文档中给出硬件接口的具体定义：

定义硬件最有效的方法是从需求开始描述，由于硬件必须支持系统定义的所有功能，因此硬件定义是与系统说明不可分割的；

例如，我们设计一个定时器（事先需求说明定时器不能与个人电脑连接，故无法使用 CRT 显示时间），我们只有两种选择：一种是使用发光二极管（LED），另一种是使用液晶显示器件（LCD）；尽管 LCD 的显示效果比较好，但考虑到定时器要常年位于户外，并且早期 LCD 显示器不能在低温下工作，最终还是选择 LED 设备（**这整个过程描述了我们硬件选型时的一个思路，这个是密切跟需求挂钩的**）

- 3) 一旦完成了系统整体说明文档，就开始进行系统设计：

首先要对硬件说明的内容进行细化，包括添加能让工程师理解的设计意图，以及软件工程师围绕硬件进行程序设计时需要使用的硬件信息等。

完成硬件电路板说明文档后，我们还要在该文档中增加一个用来描述系统的原始要求的前言部分，包括说明方案的设计思路和方法，除此之外，还要附上软件工程师用来对硬件进行控制所需的各类信息，这类信息主要包括如下内容（软件工程所需信息）：

-----内存和 I/O 端口地址（如果需要，还可以提供内存映射图）

- 可用内存容量
- 状态寄存器每一位的定义
- 每个端口管脚的用途
- 外部设备的驱动方法（例如，说明输入定时器电路的时钟频率等）
- 其他有管软件人员设计程序需要了解的信息

对于比较复杂的系统来说，硬件文档中经常使用两个独立的部分来进行说明；其第一部分用来描述硬件指标和工作原理，第二部分则主要为软件人员提供程序设计需要的信息。

2. 软件文档撰写思路

1) 软件文档与硬件文档的组织方法类似，软件要求文档的主要内容则是定义软件要实现的功能；一种是在简单项目设计过程中，软件定义也可以只对一种电路板使用的软件给予描述；对较复杂的项目来说，由于参与这种项目的软件人员分别负责设计驱动不同硬件部分的代码（同一电路板），因此每个软件人员可能会为自己的设计代码指定不同的定义，这类软件说明需要提供下列的内容：

- 论述包括需求定义、工程指标、硬件参数等实施项目需要的内容
- 说明软件之间、处理器之间或处理器与其内部器件之间使用的通信协议：其内容应包括对缓冲区接口机制、命令/应答协议、信号控制等协议的具体说明。
- 借助流程图、伪代码或者其他可能的方法来描述软件的实现方法和过程

2) 软件与硬件所考虑的不同之处（**此经验方便技术总监或其他相关管理者参考，因为无论是多高深的技术管理者，要么是硬件出身，要么是软件出身，要么就是非技术出身，armjishu.com 里面有少数软硬件都精通的高手**）

- a. 软件的灵活性远远大于硬件，要让软件人员搞清楚某个软件的内部格式是非常困难的任务，解决的办法：详细定义其他程序员需要了解的编程接口具体内容，以及其他工程人员在实施开发项目过程中需要使用的技术细节信息。
- b. 软件工程师只有在收到硬件说明文档后，才有可能知道如何对系统硬件进行操作；而硬件人员一般不需要了解软件程序的技术细节。
- c. 由于软件易于更改，因此程序内容经常会按销售人员提供的要求发生变更，在某些情况下，软件文档的内容无法及时反映程序的最新变化。

文档是否详细、完整，在某种程度上是与公司或客户的要求有关的。例如，军事或国家工程一般要求开发商就其所有软件实现的功能提供全面详细的文档。

e. 有个潜规则，对软件的要求越复杂，则需求的正确可能性就越小，这个是经验之谈了，我们需要把准需求这个准绳来做文章，而不是陷入个人主义以及对软件要求而凭空发挥自己不切实际的想象。

f. 我们可以先硬件设计，接着围绕该硬件编制软件。虽然实际系统的实现过程可能是软硬件并行开发，但软件人员基本上也是围绕着已经实现的硬件来进行程序设计的；对于更为复杂的系统来说，开发过程可能会出现重复。

例如，某个项目的硬件工程师和软件工程师可能会坐下来开会，共同决定使用哪种硬件来实现某种功能；软件人员可能提出需要为数据缓冲区口冲内存容量，也可能要求提供某种外部设备接口，以便充分利用现成接口程序提供的各种驱动代码。

总的来说，必须在提高软件开发效率与硬件系统的复杂性与成本之间进行权衡。

5) 嵌入式高手对技术的理解（含辛茹苦这么多年的精华体验）

有很多人认为：嵌入式系统性能的核心因素是软件功能，其实，如果按照这种逻辑，系统设计中存在的问题就应由软件人员来负责；其实这个观点实际上反映了设计嵌入式产品时如何考虑划分硬件和软件各自应实现的功能，也就是这个功能是软件实现，还是考虑用硬件来实现（硬件实现：需要购买处理该功能的硬件芯片，从而增加成本；软件实现：无需增加硬件成本，但会占用处理器以及内存的资源，这是 armjishu.com 的专家们体会到的）。

例如：我们在这里设计的基于 STM32 的神舟 II 号开发板产品，我们可以使用专业的解码芯片来负责 mp3 音乐文件的解码和播放功能，也可以使用另一种方法来解码 mp3 语音文件，让 ARM 处理器利用软件控制寄存器来驱动耳机或音响，处理器通过对 mp3 语音文件解码之后再将解码后的数据流按照一定协议格式送给音频输出的硬件接口进行播放。

优点：这种方案在硬件方面节省了一个器件，降低了成本，并且该功能还方便调试（因为是软件实现的）。

缺点：从另一个角度来看，虽然节省了一块语音解码芯片，但同时要在三个方面增加成本。

首先，要在程序中增加语音协议解码的代码；

其次，可能要把增加 ROM 来存放语音解码的协议，这样可以增加速度；

最后，运行该程序将占用处理器的时间和资源。

其实，话又说回来，对于本案例来说，上述成本的节约并不会引发任何问题，包括驱动程序增加也只需少量的，我们讨论这个 mp3 产品的案例的目的在于说明如何对软件硬件的功能进行合理划分。

总的来说，交给软件实现的功能越多，则产品的成本就越低，当然这就要处理器必须有足够的处理速度和内存空间来实现设计指定的功能；常言说得好，天下没有免费的午餐；把功能分配给软件来实现，会增加软件的复杂性、开发时间、以及程序的调试时间；然而，随着处理器的处理能力的不断提高，可以预见，越来越多的功能将会由软件来实现。

虽然在软件中实现各种功能会增加开发成本，但如果把功能移植到硬件中实现，则会增加产品的成本，这类开销是在构造每个系统组件时不可避免的。在低成本设计方案中，增加任何额外的硬件都会对产品成本产生显著的影响，因此软硬件功能划分就是一个决定产品成本的大问题。在诸如大众消费产品这一类对成本非常敏感的设计方案中，一般都会把无法通过软件实现的功能排除在外的。

Armjishu.com 团队 写于 2009 年 1 月

附件4：PCB设计建议

1. PCB设计干扰的相关基础知识

EMI：电磁干扰（Electromagnetic Interference 简称 EMI），直译是电磁干扰。这是合成词，我们应该分别考虑“电磁”和“干扰”。是指电磁波与电子元件作用后而产生的干扰现象，有传导干扰和辐射干扰两种。传导干扰是指通过导电介质把一个电网络上的信号耦合(干扰)到另一个电网络。辐射干扰是指干扰源通过空间把其信号耦合(干扰)到另一个电网络，在高速 PCB 及系统设计中，高频信号线、集成电路的引脚、各类接插件等都可能成为具有天线特性的辐射干扰源，能发射电磁波并影响其他系统或本系统内其他子系统的正常工作。

所谓“干扰”，指设备受到干扰后性能降低以及对设备产生干扰的干扰源这二层意思。第一层意思如雷电使收音机产生杂音，摩托车在附近行驶后电视画面出现雪花，拿起电话后听到无线电声音等，这些可以简称其为与“BC I”“TV I”“Tel I”，这些缩写中都有相同的“ I ”（干扰）。

那么 EMI 标准和 EMI 检测是 EMI 的哪部分呢？理所当然是第二层含义，即干扰源，也包括受到干扰之前的电磁能量。

2. 电磁干扰三要素

理论和实践的研究表明，不管复杂系统还是简单装置，任何一个电磁干扰的发生必须具备三个基本条件：首先应该具有骚扰源；其次有传播干扰能量的途径和通道；第三还必须有被干扰对象的响应。在电磁兼容性理论中把被干扰对象统称为敏感设备（或敏感器）。

因此电磁骚扰源、骚扰传播途径（或传输通道）和敏感设备称为电磁干扰三要素。

1. 电磁骚扰源分类

1.1. 一般说来电磁骚扰源分为两大类：自然骚扰源与人为骚扰源。

自然干扰源主要来源于大气层的天电噪声、地球外层空间的宇宙噪声。他们既是地球电磁环境的基本要素组成部分，同时又是对无线电通讯和空间技术造成干扰的干扰源。自然噪声会对人造卫星和宇宙飞船的运行产生干扰，也会对弹道导弹运载火箭的发射产生干扰。

人为干扰源是有机电或其他人工装置产生电磁能量干扰，其中一部分是专门用来发射电磁能量的装置，如广播、电视、通信、雷达和导航等无线电设备，称为有意发射干扰源。另一部分是在完成自身功能的同时附带产生电磁能量的发射，如交通车辆、架空输电线、照明器具、电动机械、家用电器以及工业、医用射频设备等等。因此这部分又成为无意发射干扰源。

1.2. 从电磁干扰属性来分，可以分为功能型干扰源和非功能性干扰源。

功能性干扰源系指设备实现功能过程中造成对其他设备的直接干扰；非功能性干扰源是指用电装置在实现自身功能的同时伴随产生或附加产生的副作用，如开关闭合或切断产生的电弧放电干扰。

1.3. 从电磁干扰信号频谱宽度，可以分为宽带干扰源和窄带干扰源。

他们是相对于指定感受器的带宽大或小来加以区别的。干扰信号的带宽大于指定感受器带宽的成为当代干扰，反之称为窄带干扰源。

1.4. 从干扰信号的频率范围来分

可以把干扰源分为工频与音频干扰源（50Hz 及其谐波）、甚低频干扰源（30Hz 以下）、载频干扰源（10kHz~300kHz）、射频及视频干扰源（300kHz）、微波干扰源（300MHz~100GHz）。

2. 电磁骚扰传播途径

电磁干扰传播途径一般也分为两种：即传导耦合方式和辐射耦合方式。

任何电磁干扰的发生都必然存在干扰能量的传输和传输途径（或传输通道）。通常认为电磁干扰传输有两种方式：一种是传导传输方式；另一种是辐射传输方式。因此从被干扰的敏感器来看，干扰耦合可分为传导耦合和辐射耦合两大类。

传导传输必须在干扰源和敏感器之间有完整的电路连接，干扰信号沿着这个连接电路传递到敏感器，发生干扰现象。这个传输电路可包括导线，设备的导电构件、供电电源、公共阻抗、接地平面、电阻、电感、电容和互感元件等。

辐射传输是通过介质以电磁波的形式传播，干扰能量按电磁场的规律向周围空间发射。常见的辐射耦合由三种：1. 甲天线发射的电磁波被乙天线意外接受，称为天线对天线耦合；2. 空间电磁场经导线感应而耦合，称为场对线的耦合；3. 两根平行导线之间的高频信号感应，称为线对线的感应耦合。

在实际工程中，两个设备之间发生干扰通常包含着许多种途径的耦合。正因为多种途径的耦合同时存在，反复交叉耦合，共同产生干扰，才使电磁干扰变得难以控制。

3. 敏感设备

敏感设备是对干扰对象总称，它可以是一个很小的元件或一个电路板组件，也可以是一个单独的用电设备甚至可以是一个大型系统。

3.1 印制电路板

出于技术的考虑，最好使用独立的接地层（VSS）和专门独立的供电层（VDD）的多层印制电路板，这样能提供好的耦合性能和屏蔽效果。一般 4 层板以上的结构就可以满足了，中间的两层一层是地，一层是走供电层，最上面和最下面的两层走信号线；很多应用中，受经济条件限制不能使用这样的印制电路板，一般单层和两层的 PCB 电路板就需要更多细节去考虑一下地和供电走线的配置了，下面会进行进一步的详细分析。

3. 器件位置

为了减少 PCB 上的交叉耦合（注：耦合是指两个或两个以上的电路元件或电网络的输入与输出之间存在紧密配合与相互影响，并通过相互作用从一侧向另一侧传输能量的现象），设计 PCB 时需要根据各自对 EMI 影响的不同，而把不同的电路分开。比如，大电流电路、低电压电路以及数字器件等；比如某 CPU 的晶振必须靠近该 CPU，电源芯片的滤波电容尽可能的靠近电源芯片管脚附近，这样效果都会好很多。

4. 接地和供电（VSS, VDD）

每个模块（噪声电路、敏感度低的电路、数字电路）都应该单独接地，所有的地最终都应在一个点上连到一起。尽量避免或者减小回路的区域。为了减少供电回路的区域，电源应该尽量靠近地线，这是因为，供电回路就像个天线，成为 EMI 的发射器和接收器。PCB 上没有器件的区域，需要填充为地，以提供好的屏蔽效果（特别是对单层 PCB，尤其如此）。

即使在整个 PCB 板中的布线完成得都很好，但由于电源、地线的考虑不周到而引起的干扰，会使产品的性能下降，有时甚至影响到产品的成功率。所以对电、地线的布线要认真对待，把电、地线所产生的噪音干扰降到最低限度，以保证产品的质量。

对每个从事电子产品设计的工程人员来说都明白地线与电源线之间噪音产生的原因，现只对降低式抑制噪音作以表述：众所周知的是在电源、地线之间加上去耦电容。

尽量加宽电源、地线宽度，最好是地线比电源线宽，它们的关系是：地线>电源线>信号线。

5. 数字电路与模拟电路的供电处理

现在有许多 PCB 不再是单一功能电路（数字或模拟电路），而是由数字电路和模拟电路混合构成的。因此在布线时就需要考虑它们之间互相干扰问题，特别是地线上的噪音干扰。数字电路的频率高，模拟电路的敏感度强。

对信号线来说，高频的信号线尽可能远离敏感的模拟电路器件。

对地线来说，整个 PCB 对外界只有一个结点，所以必须在 PCB 内部进行处理数、模共地的问题，而在板内部数字地和模拟地实际上是分开的它们之间互不相连，只是在 PCB 与外界连接的接口处（如插头等）。

数字地与模拟地有一点短接，请注意，只有一个连接点。也有在 PCB 上不共地的，这由系统设计来决定。在神舟开发板中，我们使用的 0 欧电阻隔离了一下，实际产品中，建议使用磁珠进行隔离会比较好。

6. 信号线布在电或地层上

像神舟王的核心板 STM32103,207,407,439 都是 4 层板，这个问题一般都会在多层设计的时候出现，在多层印制板布线时，由于在信号线层没有布完的线剩下已经不多，再多加层数就会造成浪费也会给生产增加一定的工作量，成本也相应增加了，为解决这个矛盾，可以考虑在电（地）层上进行布线。首先应考虑用电源层，其次才是地层。因为最好是保留地层的完整性。

两层板不会出现这个问题。

7. 焊盘与产品良品率质量的关系

设计的焊盘大小和形状也与产品的质量有关系，举 2 个例来说明：

在大面积的接地（电）中，常用元器件的管脚与其连接，对连接管脚的处理需要进行综合的考虑，就电气性能而言，元件管脚的焊盘与铜面满接为好，但对元件的焊接装配就存在一些不良隐患，比如有时候焊盘过大，造成加工的时候加热时间要久一些，如果加热时间不够，就有可能造成虚焊。

如果焊盘过短也会有问题，很容易造成接触不良，所以说，焊盘的长短大小，都有一个适度的大小，如果说小了短了，就会造成管脚接触不到位，焊盘面积与芯片管脚接触面过小，造成通信质量下降；并且焊盘过短和过小，将会造成贴片加工难度增加，有个调查很有意思，一批产品一次性加工出来良品率是 85%，经过摸索，这个产品被发现有个芯片总出现虚焊的情况，后来把这个芯片的焊盘加长一点点，再次批量加工，这颗芯片几乎都不会出现虚焊的情况，良品率一度上升至 98%。

8. 其他信号的注意事项

实际应用中，关注以下几点可以提高 EMC 性能：

- 那些受暂时的干扰会影响运行结果的信号(比如中断或者握手抖动信号，而不是 LED 命令之类的信号)。对于这些信号，信号线周围铺地，缩短走线距离，消除邻近的噪声和敏感的连线都可以提高 EMC 性能。对于数字信号，为有效地区别 2 种逻辑状态，必须能够达到最佳可能的信号特性余量(译注：尽可能抬高逻辑'1'的高电平，拉低逻辑'0'的低电平)。推荐使用慢速施密特触发器来消除寄生状态。
- 噪声信号(时钟等)。
- 敏感信号(高阻等)。

9. 未用到的I/O管脚

所有微控制器都为各种应用而设计，而通常的应用都不会用到所有的微控制器资源。为了提高 EMC 性能，不用的时钟、计数器或者 I/O 管脚，需要做相应处理，比如，I/O 端口应该被设置为'0' 或'1' (对不用到的 I/O 引脚上拉或者下拉)；没有用到的模块应该禁止或者“冻结”。

因为如果不冻结，有可能会出现异常情况，并且出现这些情况之后，还不方便查找问题，所以干脆将这些管脚规定好，这样方便使得产品稳定，并且万一出现问题就会比较稳定。

附件5：项目合作与技术支持联系方式

可以发送邮件：armjishu.com@163.com

由于篇幅原因，附件相关内容可在www.armjishu.com网站下载或者产品光盘的PDF电子版手册。

【全文完】



神舟 III 号用户手册

www.armjishu.com