STM32 从入门到精通

2012年3月版本　V2.0　　作者：jesse

STM32神舟系列开发板产品目录：

【神舟 I 号：STM32F103RBT6 + 2.8″TFT 触摸彩屏】

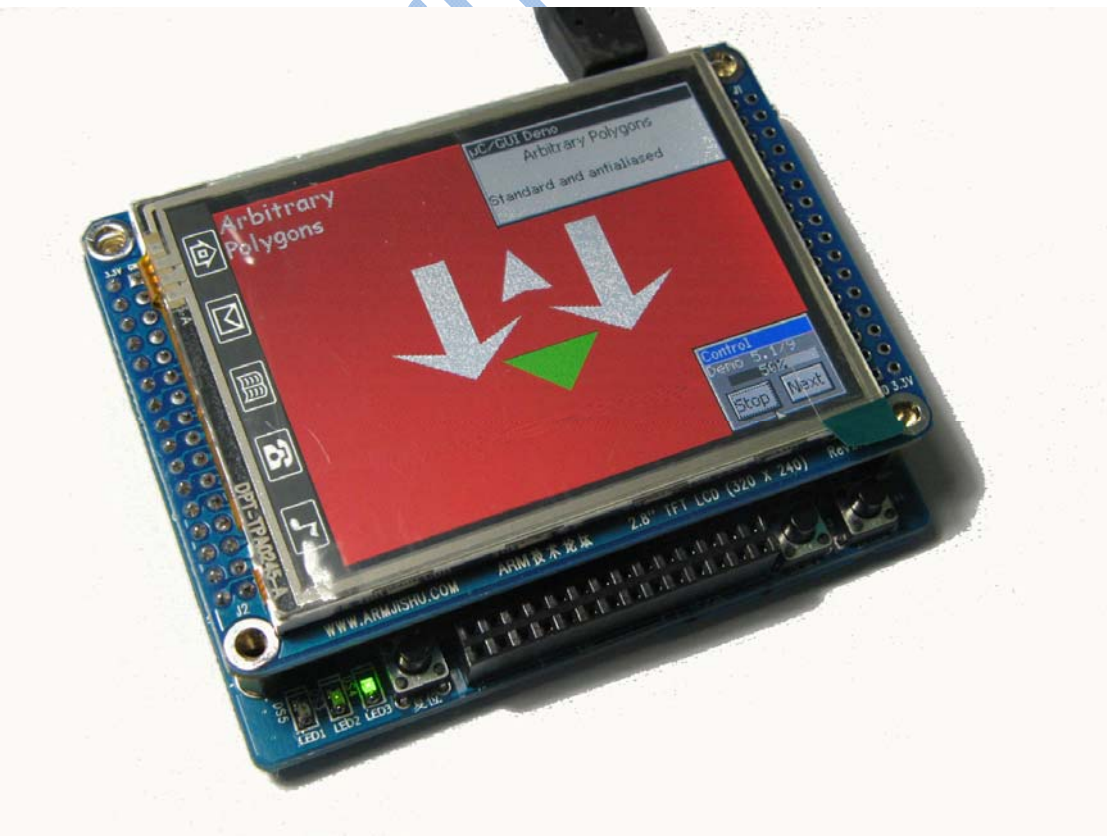
神舟 II号：STM32F103VCT6 + 3.2″TFT 触摸彩屏

神舟III号：STM32F103ZET6 + 3.2″TFT 触摸彩屏

神舟 IV号：STM32F107VCT6 + 3.2″TFT 触摸彩屏

神舟STM32核心板：四层核心板（STM32F103ZET6+STM32F107VCT6）

神舟王：STM32F103/107核心板(底板不变，主芯片可灵活搭配）+ 超全功能底板（支持MP3,以太网,收音机,无线,SRAM，Nor/Nand flash,鼠标,键盘,红外接收,CAN,温度传感,示波器,电压表,USB HOST,步进电机等）



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 I 号 STM32 开发板配套用户手册，详细介绍 STM32 的开发过程和神舟 I 号的使用指导。

本文档如有升级恕不另行通知，欢迎访问www.armjishu.com获取最新手册文档及最新固件代码。

www.armjishu.com

目 录

前言必读（文档结构与快速入门） **8**

第 0 章 入门了解篇 **8**

0.1 STM32 嵌入式入门必看之文章（学习STM32 的理由！！！！） 8
0.2 如何从零开始开发一款嵌入式产品（20 年的嵌入式经验分享学习神舟系列） 10
0.2.1 需求定义 *10*
0.2.2 处理器的选择 *12*
0.2.3 开发成本的预测和估计 *16*
0.2.4 产品开发设计文档（需要包括硬件和软件两个方面） *16*
0.2.5 嵌入式高手对技术的理解（含辛茹苦这么多年的精华体验） *18*

第 1 章 STM32F103RBT硬件体系结构 **19**

1.1 STM32RBT简介 19
1.1.1. 特性 *19*
1.1.2. 器件信息 *20*
1.1.3. 结构 *21*
1.2 引脚描述 22
1.2.1 STM32 引脚封装 *22*
1.2.2 STM32 引脚描述 *22*
1.3 系统控制模块 26
1.3.1 系统控制模块功能汇总 *26*
1.3.2 引脚描述 *26*
1.3.3 寄存器描述 *26*
1.3.4 复位 *26*
1.4 时钟系统 33
1.4.1 时钟系统分析 *34*
1.4.2 寄存器描述 *38*
1.4.3 振荡器（STM32 内部有RC做振荡器，外部有晶振做振荡器） *38*
1.6 存储器寻址 39

第 2 章 神舟I号开发套件硬件结构 **42**

2.1 电路原理图 42
2.2 功能特点 43
2.3 神舟I号开发板硬件电路分析 45
2.3.1 STM32F103RBT6 处理器 *45*
2.3.2 LED指示灯 *48*
2.3.3 普通按键与复位按键 *49*
2.3.4 USB接口 *50*
2.3.5 USB转串口接口 *51*
2.3.6 供电电源 *52*
2.3.7 RTC实时时钟 *53*
2.3.8 电位器（ADC介绍） *55*
2.3.9 EEPROM存储器（IIC接口控制） *55*
2.3.10 W25X16 16M SPI FLASH（LCD屏上安装） *56*
2.3.11 MicroSD卡接口 *57*
2.3.12 JTAG调试接口 *59*
2.3.13 液晶显示模块 *61*
2.3.14 温度传感器 *62*
2.3.15 315M无线模块 *63*
2.3.16 2.4G无线模块 *64*

www.armjishu.com

2.3.17 液晶屏 *65*
2.3.18 其他扩展接口 *65*
2.4 硬件结构说明 67
2.5 连接器说明 68
2.6 硬件资源分配 70
2.6.1 处理器连接外围器件管脚分配 *70*
2.7 STM32 系列ARM最小系统设计 72

第 3 章 其他篇 79

3.1 液晶屏显示屏入门 79
3.2 液晶屏底板设计说明 80

第 4 章 STM32 神舟I号基本操作篇 81

4.1 简介 81
4.2 MDK 4.12 集成开发环境的组成 81
4.3 安装MDK的流程步骤 81
4.4 工程的编辑 82
4.4.1 建立工程 *82*
4.4.2 建立文件 *85*
4.4.3 添加文件到工程 *86*
4.4.4 管理工程目录以及源文件 *88*
4.4.5 编译和连接工程 *91*
4.4.6 打开旧工程 *93*
4.5 RVMDK使用技巧 95
2.3.1. 快速定位函数/变量被定义的地方 *95*
2.3.2. 快速注释与快速消注释 *96*
2.3.3. 快速打开头文件 *96*
4.6 JLINK V8 仿真器的安装与应用 97
4.6.1 JLINK V8 仿真器简介 *98*
4.6.2 JLINK ARM主要特点 *98*
4.6.3 JLINK V8 仿真器安装 *98*
4.6.4 JLINK V8 仿真器配置（MDK KEIL环境） *100*
4.6.5 JLINK V8 仿真器配置（IAR 环境） *106*
4.6.6 J-FLASH如何烧写固件到芯片FLASH里 *110*
4.6.7 JLINK Commander工具查看相关信息 *114*
4.6.8 JLINK V8 仿真器如何自动升级 *115*
4.7 在MDK开发环境中JLINK V8 的调试技巧 117
4.8 如何通过串口下载一个固件到神舟I号开发板 119
4.9 从零开始新建一个STM32 的工程模板 128
4.10 何给神舟I号板供电 146
4.10.1 使用USB供电 *146*
4.10.2 使用USB转串口接口供电 *146*
4.10.3 使用JLINK V8 供电 *147*
4.11 烧录固件程序的三种方法 147
4.12 如何编译和运行光盘里的第一个程序： 148
4.13 如何用JLINK V8 仿真和调试第一个程序： 152

第 5 章 STM32 神舟I号快速入门篇 156

5.1 理解芯片控制的原理 156
5.2 芯片管脚控制LED灯原理图解释 156
5.3 芯片管脚控制原理（如何阅读芯片手册） 157
5.4 实际例程详解 161
5.4.1 原理图说明 *161*
5.4.2 超级简单的例程结构（只有一个main.c文件） *161*

www.armjishu.com

5.4.3 main.c全部代码粘贴： *162*
5.4.4 程序初始化代码定义说明（包含芯片手册阅读方法） *164*
5.4.5 程序代码的struct与芯片手册寄存器如何对应 *166*
5.4.6 C语言程序代码如何真正访问芯片内部寄存器 *166*
5.4.7 mian函数剖析 *168*
5.4.8 库函数与我们这个例程之间的关系 *169*

第 6 章 STM32 神舟I号功能部件基础篇 169

6.1 STM32 神舟I号实验例程结构 169
6.2 通用输入/输出（GPIO） 171
6.2.1 特性 *171*
6.2.2 应用领域 *171*
6.2.3 管脚描述 *172*
6.2.4 功能描述 *172*
6.2.5 寄存器描述 *178*
6.2.6 寄存器小结 *182*
6.2.7 例程01 单个LED点灯程序 *182*
6.2.8 例程02 单个LED灯闪烁 *184*
6.2.9 例程03 LED流水灯程序 *185*
6.3 KEY_LED按键与 315M无线模块实验 187
6.2.1 实验的意义与作用 *187*
6.2.2 实验原理 *187*
6.2.3 硬件设计 *188*
6.2.4 软件设计 *188*
6.2.5 下载与测试 *192*
6.3 USART-COM串口发送实验 193
6.3.1 实验的意义与作用 *193*
6.3.2 实验原理 *193*
6.3.3 硬件设计 *195*
6.3.4 软件设计 *195*
6.3.5 下载与现象 *197*
6.4 USART-COM串口发送与接收实验 200
6.4.1 验的意义与作用 *200*
6.4.2 实验原理 *200*
6.4.3 硬件设计 *200*
6.4.4 软件设计 *200*
6.4.5 下载与现象 *202*
6.5 ADC模数转换实验 203
6.5.1 实验的意义与作用 *203*
6.5.2 实验原理 *203*
6.5.3 硬件设计 *204*
6.5.4 软件设计 *204*
6.5.5 下载与现象 *206*
6.6 EEPROM读写程序实验 207
6.6.1 实验的意义与作用 *207*
6.6.2 试验原理 *207*
6.6.3 硬件设计 *208*
6.6.4 软件设计 *209*
6.6.5 下载与测试 *212*
6.7 SPI FLASH（W25X16）读写程序实验 213
6.7.1 SPI FLASH（W25X16）读写程序实验的意义与作用 *213*
6.7.2 实验原理 *213*
6.7.3 硬件设计 *216*
6.7.4 软件设计 *216*

www.armjishu.com

6.7.5 下载与测试现象 223
6.8 实时时钟与年月日实验 224
6.8.1 实验的意义与作用 224
6.8.2 实验原理 224
6.8.3 硬件设计 225
6.8.4 软件设计 226
6.8.5 下载与测试 232
6.9 独立看门狗实验 233
6.9.1 实验的意义与作用 233
6.9.2 实验原理 233
6.9.3 硬件设计 235
6.9.4 软件设计 235
6.9.5 下载与测试 237
6.10 SysTick实验 238
6.10.1 硬件设计 238
6.10.2 软件设计 239
6.10.3 下载与测试 240
6.11 TFT彩屏显示实验 241
6.11.1 实验的意义与作用 241
6.11.2 实验原理 241
6.11.3 硬件设计 243
6.11.4 软件设计 244
6.11.5 下载与现象 249
6.12 TFT触摸屏显示加触摸实验 250
6.12.1 实验的意义与作用 250
6.12.2 实验原理 250
6.12.3 硬件设计 251
6.12.4 软件设计 251
6.12.5 下载与现象 256
6.13 DS18B20 温度传感器实验 257
6.13.1 实验的意义与作用 257
6.13.2 试验原理 257
6.13.3 硬件设计 258
6.13.4 软件设计 258
6.13.5 下载与现象 260
6.14 2.4G模块通信试验 261
6.14.1 2.4G模块通信实验的意义与作用 261
6.14.2 实验原理 261
6.14.3 硬件设计 261
6.14.4 软件设计 262
6.14.5 下载与测试现象 268
6.15 USB遥控鼠标实验 269
6.15.1 实验的意义与作用 269
6.15.2 实验原理 269
6.15.3 硬件设计 270
6.15.4 软件设计 271
6.15.5 下载与测试 274
6.16 Micro SD卡实验 275
6.16.1 实验的意义与作用 275
6.16.2 实验原理 275
6.16.3 硬件设计 277
6.16.4 软件设计 278
6.16.5 下载与测试 282
6.17 SD-USB读卡器实验 283
www.armjishu.com

6.17.1 实验的意义与作用 284
6.17.2 试验原理 284
6.17.3 硬件设计 284
6.17.4 软件设计 285
6.17.5 下载与测试 288
6.18 uCOS_UCGUI_DEMO实验 289
第 7 章 实验现象 290
附件 1：JLINK V8 用户手册 291
附件 2：JLINK转接板简介 291
附件 3：《TCP/IP 协议栈LWIP的设计与实现》 291
附件 4：项目合作与技术支持联系方式 291

www.armjishu.com

前言必读（文档结构与快速入门）

本文档为 STM32 神舟系列前言必读，如果您是初学者，请您先参看搭建环境的章节，然后开始打开光盘中的例程，通过编译和下载例程，然后运行程序；然后通过从 0 开始搭建一个例程环境，本例程是搭建用 JLINK V8 仿真器与开发板进行搭配，进行下载和运行的。

第 0 章的 0.1 章节是入门必看的文章，明白为什么 STM32 以及 STM32 神舟系列在嵌入式领域所处的地位。

第 0 章的 0.2 章节是笔者 20 年嵌入式经验撰写而成的经验之贴，希望大家能慢慢的阅读并多读几遍，您每提高一个层次，再回头看看这篇文章，都会有不同的感受和领悟，他有助于您永远建立在一个整体把握的全局观上的层次，加快进步的速度。这个章节对初学来说，是一个需要慢慢品味和见识的过程；对高手来说，或许您曾经经历过的一些事情，都还在历历在目。

该文档手册主要是针对神舟系列开发板写的，如果您是设计产品，那么还需要参考一下其他的手册，这里分为几个阶段，我大概归纳了一下：

1）通常在芯片选型的初期（这里指具体选哪款芯片，例如是 STM32F103RBT，还是 STM32F103VCT 呢？），首先要看“XX 数据手册”以评估该产品是否能够满足设计上的功能需求

2）然后在设计硬件时，可以通过这个“XX 数据手册”获得电压，电流，管脚分配，驱动能力等信息；

3）在基本选定所需的芯片后，再查看“XX 技术参考手册”获知各功能模块的工作模式是否符合要求（例如 STM32F103 系列微处理器的 USB 和 CAN 不能同时共用的，所以设计产品时必须提前了解这些相关信息）；

4）在确定选型进入编程阶段时，再查阅该“XX 技术参考手册”了解各项具体功能的实现方式和寄存器的配置实用。

另外：关于 Cortex-M3 核心，SysTick 定时器和 NVIC 的详细说明，可以参考另一篇 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》

第0章 入门了解篇

0.1 STM32嵌入式入门必看之文章（学习STM32的理由！！！！）

为什么要写这篇文章呢？这是一篇关于嵌入式入门的文章，因为我在进入嵌入式这个领域之前，也是遇到过非常多非常多的困难，所以呢，希望写下这篇文章，让大家看看少走弯路。

首先，我打算先列举一下大家问得最多的几个问题，然后我们一起由问题切入进行一些讨论。

问题 1：我是学单片机好还是直接学 STM32 好？？？

问题 2：STM32 如何才能快速入门？

问题 3：为什么是 STM32 呢？为什么不是 ARM9，ARM11 呢？

下面我将逐个答复

首先问题 1：我是单片机好还是直接学 STM32 好呢？

答：首先我们谈下目的，你学习的最终目的是能够开发产品或者成为项目经理；而目前市场上的单片机都基本用 C 语言了，用汇编进行开发的项目已经非常少了

在硬件接口这个环节上都一样：学 51 单片机和学 STM32 都是一样的，主要看其市场上的资料是不是足够充分，都是调用硬件的接口，控制 I/O，完成相应的功能

在软件这个环节上，STM32 要强过 51 单片机：为什么这么说呢？学 STM32 它自带一个官方的库，而这个库的源代码是开放的；而 51 单片机却没有，也就说，当你做一个具体项目的时候，用 STM32 开发项目速度会比用 51 单片机要轻松快捷，方便，并且 BUG 也少.

ST 的这个库专门是由 ST 官方团队打造的，是经过详细认真测试过的，例如：你要做处理器支持 USB 的 U 盘功能，对 51 来说，你可能需要在网上去找这样一个驱动，然后移植到 51 单片机上，这样的缺点是：移植本身就比较累，比较麻烦，并可能有些代码不一定经过验证了的，很难保证没有 BUG；而 STM32 是官方推荐的库源代码，这些功能早就有了，所以用 STM32 开发项目做起来非常的轻松，愉快，放心，BUG 也少。

从选型方面考虑：STM32 的性价比与单片机相比，虽说单片机很便宜了，一般市场上就 5 块钱左右，而 STM32 最便宜的是 10 元多一点，但是如果比性价比，STM32 还是很要强过 51 单片机很多，为什么呢？因为首先 STM32 是 72MHZ 的主频，而 51 单片机是 10 多 M，随着未来产品功能增多了，要求越来越高，可能 51 单片机的速度和性能就满足不了需求，而 STM32 足够强劲的主频，可以延缓这个问题，使得您的产品周期生命得到延长，而且你有其他新需求也可以灵活增加上来，因为 STM32 完全有能力负担得起。

问题 2：STM32 如何才能快速入门？

答：首先就是最好具备基本的电子基础，如果您只有纯软件基础，那就应该打扎实 C 语言基础，然后弄一块开发板，边学边补充硬件电子方面的知识，因为软件背景去理论的学硬件，那是非常枯燥的事情，而且效果也不一定理想；那么此时你就需要一款比较的板子，板子大与小，功能多与少都不是那么重要，这里最重要的就是要资料丰富，资料丰富，才是最好的；我当时就从同事那里弄了一块 STM32 神舟系列的板子，大概花了 1，2 个星期就正式入门了。以下是我摘抄的关于那个板子的一些特点，大家可以看看：

关于 STM32 神舟开发板的特点有哪些呢？？？？？

特点 1：STM32 神舟系列的每款开发板都有一个非常详细的几百页的说明手册。有很多爱好者反映，买到的开发板没有手册或手册不全，手册不详细，拿到手后没有什么用，无从下手，那么您可以考虑 STM32 神舟系列的开发板，其中神舟 IV 号的手册有将近 700 多页之多，无论还是从原理还是从代码都是非常值得阅读的资料书籍。

特点 2：开发板所带的例程代码非常好，易懂和方便移植。很多开发板的代码写得很难阅读，不规范，有的甚至是用寄存器实现的代码，可读性非常差，并且不容易重用到新的实际项目中，而 STM32 神舟系列的开发板，全部用 ST 的专用库实现，库代码全部开源，库即是将底层寄存器部分代码全部封装成函数，融入了软件设计的架构理念，想跟踪到硬件实现的驱动底层，就跟进对应的函数即可看到一切原始代码，所以您可以有选择的想看寄存器版本就看寄存器版本，想看函数库版本就有函数库版本，无论对实际项目也好，针对学习也好，两全其美！！！

特点 3：神舟系列板子因为系列全，所以技术支持以及技术进一步拓展空间大。许多单独的开发板技术支持根本不行，有的就算是技术支持不错，但是你想更深入一步去学习，就很难了；而成一个完整系列的 STM32 神舟系列是值考虑和选择的，因为该系列包含了 103RBT,VCT,VET,ZET,以及 107VCT 多个系列的开发板，无论你先学难，再拖展知识面，还是怎么样都好，绝对不怕资料少！没资料！！！而且各个系列之间还可以相互借鉴，相互依存，各个设计的高手非常多，大家一起相互交流，产生更多更新碰撞和资料。

特点 4：硬件资料丰富。硬件资源以及相关资料都比同类开发板要多很多，因为该系列是一个组织在维护和发展，并不是单独的个人爱好所设计的板子，这个组织走在最前沿，不断收集行业内的知识，在神舟系列上进行验证和实现，所以资料也会越来越多。

特点 5：板子的网络接口特别加强。许多 STM32 开发板没有考虑到网口这块以及无线 2.4G(WIFI)和 315M 通信的例程，随着嵌入式设备在网络方面日益普及，以太网以及无线网都是嵌入式设备中必不可少的一个环节，就算有的嵌入式设备不需要网口，但是做为提供开发板的设计，不可不考虑加强网口这块的例程，代码，讲解，为各个开发爱好者做一个提前准备和设计，而这些接口神舟系列有已经直接运行的代码以及详细的讲解，大家获得相关资料，相互参考一下。

问题 3：为什么是 STM32 呢？为什么不是 ARM9，ARM11 呢？

答：这里有个误区，很多同学如果就仅仅希望入门嵌入式，那就尽量不要选择 ARM9 和 ARM11，为什么呢？因为诱

惑太多了，因为你一旦选择 ARM9 或 ARM11，那么这个平台就仅仅 linux 和 wince 等操作系统内核，驱动，应用，各种协议，硬件原理图，等都有够学一年半载了，没这点时间，你无法完全掌握，所以并不适合入门，周期太长，难度相对来说较大。

入门最好选择主频低一点的处理器，一切都是先掌握好原理，弄明白，弄透彻了，一切就都好办！所以单片机中，STM32 是目前最最主流的芯片，加上目前 STM32 的资料非常的多，所以，最好还是推荐 STM32 的开发板做为入门级的板子，学会之后，即可自己独立开发出各种产品，STM32 官方提供的开源代码库也是非常好用，将底层的各种汇编，管脚定义都封装成了各个功能函数，开发起来非常方便,快捷！

我的另外一篇文章：《如何从零开始开发一款嵌入式产品（20 年的嵌入式经验分享学习，来自STM32 神舟系列开发板设计师的总结）》http://www.cnblogs.com/stm32/archive/2011/04/25/2028503.html

0.2 如何从零开始开发一款嵌入式产品（20年的嵌入式经验分享学习神舟系列）

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。

0.2.1 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的主要客户来制定；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 资料的工作要做，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，这样当以后事务要求逐渐提高，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乏，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

0.2.2 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 IV 号的 107 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的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干兆字节 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 IV 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。

0.2.3 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。

0.2.4 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。

d. 软件经常是工程项目最后完成的部分，因此其文档也经常因时间不够而欠缺完整。实际上，软件文档是否详细、

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.

0.2.5 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。

第1章 STM32F103RBT硬件体系结构

1.1 STM32RBT简介

STM32F103RBT6是基于Corte-M3内核的微控制器，工作频率为72MHz，内置高速存储器**(高达128K字节的闪存和20K字节的SRAM)**，丰富的增强I/O端口和联接到两条APB总线的外设。所有型号的器件都包含2个12位的ADC、3个通用16位定时器和1个PWM定时器，还包含标准和先进的通信接口：多达2个I_2C接口和SPI接口、3个USART接口、一个USB接口和一个CAN接口。

STM32F103 RBT6处理器的供电电压为2.0V至3.6V，包含-40°C至+85°C温度范围和-40°C至+105°C的扩展温度范围。一系列的省电模式保证低功耗应用的要求。

这些丰富的外设配置，使得STM32F103RBT微控制器适合于多种应用场合：

- 电机驱动和应用控制
- 医疗和手持设备
- PC游戏外设和GPS平台
- 工业应用：可编程控制器(PLC)、变频器、打印机和扫描仪
- 警报系统、视频对讲、和暖气通风空调系统等

1.1.1. 特性

- Cortex-M3 处理器，最高 72MHz 工作频率；
- 存储器：128K 字节的程序存储器（ROM）；20K 字节的 SRAM；
- 时钟：内嵌出厂调校的 8MHz 和 40KHz 的 RC 振荡器，并且 32kHz RTC 振荡器也带校准功能
- 复位：上电/断电复位（POR/PDR）
- 电源管理：2.0—3.6 伏供电和 I/O 引脚，可编程电压检测（PVD）
- 低功耗：可设置睡眠、停机和待机等三种模式
- AD：2 个 12 位的模数转换器，1us 转换时间（多达 16 个输入通道），转换范围是 0 至 3.6V；双采样和保持功能，内部带温度传感器
- DMA：7 通道 DMA 控制器，支持的外设有定时器、ADC、SPI、I2C 和 USART
- I/O 端口：51 个 I/O 口，所有的 I/O 口都可以映像到 16 个外部中断；几乎所有 I/O 口可以容忍 5V 信号
- 定时器

-----3 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 位自减计数器

- 其他外围通信接口

-----多达 2 个 I2C 接口（支持 SMBus/PMBus）
-----多达 3 个 USART 接口（支持 ISO7816 接口，LIN，IrDA 接口和调制解调控制）
-----多达 2 个 SPI 接口（18M 位/秒）
-----CAN 接口
-----USB2.0 全速接口

- 安全：96 位的芯片唯一代码，CRC 计算单元
- 调试模式：同时至此后单线 SWD 调试和 JTAG 接口

1.1.2. 器件信息

STM32F103RB 红框里，具体的资源列表信息一览表：

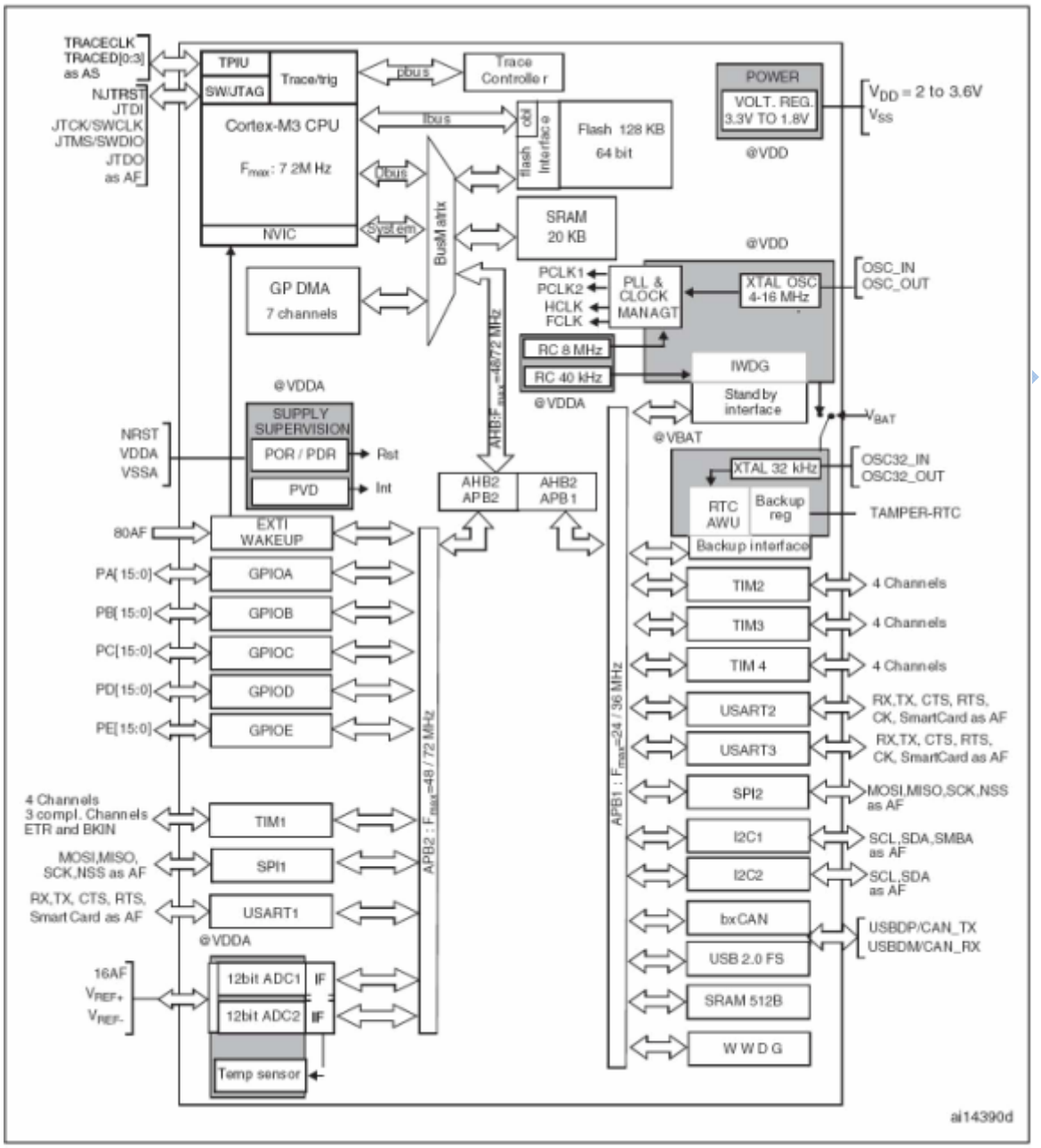
STM32(ARM Cortex-M3) 32位微控制器产品列表(截至2009年8月)

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(IC/OC/PWM)	16位高级(IC/OC/PWM)	16位基本	SPI	I²C	USART[5] UART	USB全速	CAN 2.0	以太网	I²S	SDIO	ADC(通道)	DAC(通道)		
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	LQFP64/TFBGA64
	STM32F103R6	72	32K	10K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103R8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103RB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		51	LQFP64/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	LQFP64/WLCSP64
	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	LQFP64/WLCSP64
	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	LQFP64/WLCSP64

宏	MCU型号	FLASH大小
STM32F10X_LD	STM32F101xx STM32F102xx STM32F103xx	16 ~ 32 Kbytes
STM32F10X_MD	STM32F101xx STM32F102xx STM32F103xx	64 ~ 128 Kbytes
STM32F10X_HD	STM32F101xx STM32F103xx	256 ~ 512 Kbytes
STM32F10X_CL	STM32F105xx STM32F107xx	忽略

STM32神舟系列开发板

1.1.3. 结构



STM32F103RBT 模块框图

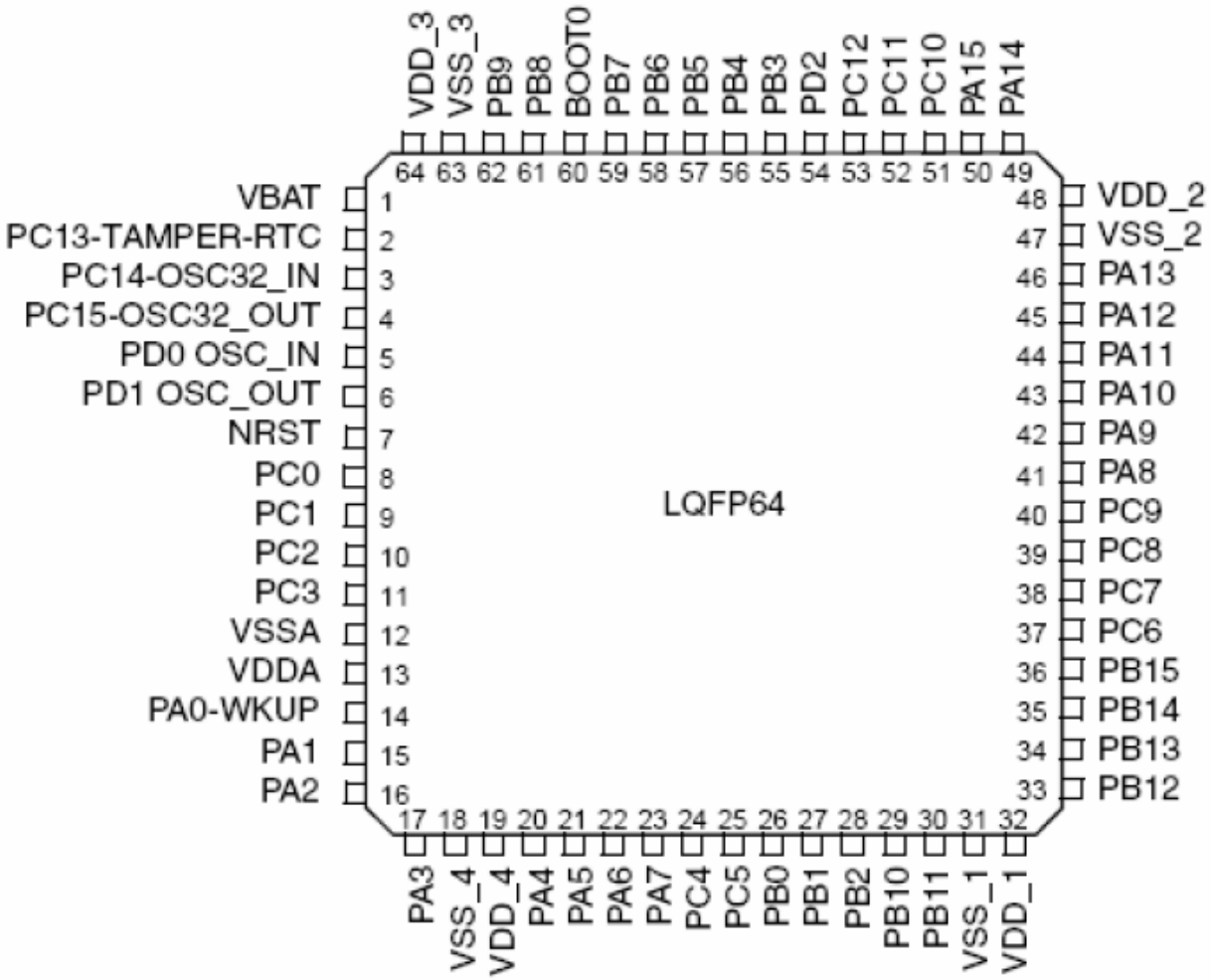
总线矩阵用来将处理器和调试接口与外部总线相连。总线矩阵与下面的外部总线相连：

- 总线矩阵：总线矩阵由四个驱动部件和四个被动部件构成
  -----四个驱动部件：CPU的DCode、系统总线、DMA1总线和DMA2总线
  -----四个被动部件：闪存存储器接口、SRAM、FSMC和AHB2APB桥
- **ICode总线：**该总线将Cortex-M3内核的指令总线与闪存指令接口相连接，即从代码空间取指令和向量，指令预取在此总线上完成，是32位的总线。
- **DCode总线：**该总线将Cortex-M3内核的DCode总线与闪存存储器的数据接口相连接，用于对代码空间进行数据加载/存储以及调试访问，是32位的总线。
- **系统总线：**该总线连接Cortex-M3内核的系统总线（又称为外设总线）到总线矩阵，用于对系统空间执行取指令和向量，数据加载/存储以及调试访问，是32位的总线
- **DMA总线：**此总线将DMA接口与总线矩阵相联，总线矩阵协调着CPU的DCode和DMA到SRAM、闪存和外设的访问。
- **AHB/APB桥：**两个AHB/APB桥在AHB喝2个APB总线间提供同步连接；APB1操作速度限于36MHz，APB2

操作于全速（最高72MHz）

1.2 引脚描述

1.2.1 STM32引脚封装



Stm32F103RBT 64 脚封装

1.2.2 STM32引脚描述

STM32F103RBT6 的引脚描述以及对应的功能简述：

符号	引脚位置	类型	描述
PA0/ WKUP/USART2_CTS/ ADC12_IN0/TIM2_CH1_ETR	14	I/O	**PA0**—串行总线接口 0 **WKUP**—唤醒请求 **USART2_CTS**—通用同步异步收发器 2 信号线 **ADC12_IN0**—模拟/数字转换信号输入 0 **TIM2_CH1_ETR**—高级控制定时器 2 信号线
PA1/USART2_RTS/ADC12_IN1/ TIM2_CH2	15	I/O	**PA1**—串行总线接口 1 **USART2_RTS**—通用同步异步收发器 2 信号线 **ADC12_IN1**—模拟/数字转换信号输入 1

			TIM2_CH2—高级控制定时器 2 信号线
PA2/USART2_TX/ADC12_IN2/ TIM2_CH3	16	I/O	**PA2**—串行总线接口 2 **USART2_TX**——通用同步异步收发器 2 信号线 **ADC12_IN2**—模拟/数字转换信号输入 2 **TIM2_CH3**—高级控制定时器 2 信号线
PA3/USART2_RX/ADC12_IN3 TIM2_CH4	17	I/O	**PA3**—串行总线接口 3 **USART2_RX**——通用同步异步收发器 2 信号线 **ADC12_IN3**—模拟/数字转换信号输入 3 **TIM2_CH4**—高级控制定时器 2 信号线
PA4/SPI1_NSS/USART2_CK /ADC12_IN4	20	I/O	**PA4**—串行总线接口 4。GPIO 管脚，一旦使能 DAC 通道，应当设置成模拟输入(AIN)。 **SPI1_NSS**—复用重映射和调试 I/O 配置寄存器 AFIO_MAPR **USART2_CK**—通用同步异步收发器 2 信号线 **ADC12_IN4**—模拟/数字转换信号输入 4
PA5/SPI1_SCK/ADC12_IN5	21	I/O	**PA5**—串行总线接口 5。GPIO 管脚，一旦使能 DAC 通道，应当设置成模拟输入(AIN)。 **SPI1_SCK**—复用重映射和调试 I/O 配置寄存器 AFIO_MAPR **ADC12_IN5**—模拟/数字转换信号输入 5
PA6/SPI1_MISO/ADC12_IN6 /TIM3_CH1	22	I/O	PA6—串行总线接口 6 **SPI1_MISO**—复用重映射和调试 I/O 配置寄存器 AFIO_MAPR **ADC12_IN6**—模拟/数字转换信号输入 6 **TIM3_CH1**—高级控制定时器 3 信号线
PA7/SPI1_MOSI/ADC12_IN7 /TIM3_CH2	23	I/O	**PA7**—串行总线接口 7 **SPI1_MOSI**—复用重映射和调试 I/O 配置寄存器 AFIO_MAPR **ADC12_IN7**—模拟/数字转换信号输入 7 **TIM3_CH2**—高级控制定时器 3 信号线
PA8/USART1_CK/TIM1_CH1 /MCO	41	I/O	**PA8**—串行总线接口 8 **USART1_CK**—通用同步异步收发器 1 信号线 **TIM1_CH1**—高级控制定时器 1 信号线 **MCO**—微控制器时钟输出
PA9/USART1_TX/TIM1_CH2	42	I/O	**PA9**—串行总线接口 9 **USART1_TX**—通用同步异步收发器 1 信号线 **TIM1_CH2**—高级控制定时器 1 信号线
PA10/USART1_RX/TIM1_CH3/ USBDM/CAN_RX	43	I/O	**PA10**—串行总线接口 10 **USART1_RX**—通用同步异步收发器 1 信号线 **TIM1_CH3**—高级控制定时器 1 信号线
PA11/USART1_CTS/ USBDM/CAN_RX /TIM1_CH4	44	I/O	**PA11**—串行总线接口 11 **USART1_CTS**—通用同步异步收发器 1 信号线 **USBDM**—USB 信号 **CAN_RX** —CAN 信号线 **TIM1_CH4**—高级控制定时器 1 信号线
PA12/ USART1_RTS/ USBDP CAN_TX/ TIM1_ETR	45	I/O	**PA12**—串行总线接口 12 **USART1_RTS**—通用同步异步收发器 1 信号线 **USBDP**—USB 信号 **CAN_TX** —CAN 信号线 **TIM1_ETR**—高级控制定时器 1 信号线

PA13	46	I/O	**PA13**—串行总线接口 13
PA14	49	I/O	**PA14**—串行总线接口 14
PA15	50	I/O	**PA15**—串行总线接口 15
PB0/ADC12_IN8/TIM3_CH3	26	I/O	**PB0**—串行接口 0 **ADC12_IN8**—模拟/数字转换信号输入 8 **TIM3_CH3**—高级控制定时器 3 信号线
PB1/ADC12_IN9/TIM3_CH4	27	I/O	**PB1**—串行接口 1 **ADC12_IN9**—模拟/数字转换信号输入 9 **TIM3_CH4**—高级控制定时器 3 信号线
PB2	28	I/O	**PB2**—串行接口 2
PB3	55	I/O	**PB3**—串行接口 3
PB4	56	I/O	**PB4**—串行接口 4
PB5/ I2C1_SMBAI	57	I/O	**PB5**—串行接口 5 **I2C1_SMBAI**— I2C 寄存器 1 信号线
PB6/ I2C1_SCL/ TIM4_CH1	58	I/O	**PB6**—串行接口 6 **I2C1_SCL**—I2C 寄存器 1 信号线 **TIM4_CH1**—高级控制定时器 4 信号线
PB7/ I2C1_SDA/ TIM4_CH2	59	I/O	**PB7**—串行接口 7 **I2C1_SDA**—I2C 寄存器 1 信号线 **TIM4_CH2**—高级控制定时器 4 信号线
PB8/ TIM4_CH3	61	I/O	**PB8**—串行接口 8 **TIM4_CH3**—高级控制定时器 4 信号线
PB9/ TIM4_CH4	62	I/O	**PB9**—串行接口 9 **TIM4_CH4**—高级控制定时器 4 信号线
PB10/I2C2_SCL/USART3_TX	29	I/O	**PB10**—串行接口 10 **I2C2_SCL**—I2C 寄存器 2 信号线 **USART3_TX**—通用同步异步收发器 3 信号线
PB11/I2C2_SDA/USART3_RX	30	I/O	**PB11**—串行接口 11 **I2C2_SDA**—I2C 寄存器 2 信号线 **USART3_RX**—通用同步异步收发器 3 信号线
PB12/SPI2_NSS/I2C2_SMBAI/ USART3_CK/TIM1_BKIN	33	I/O	**PB12**—串行接口 12 **SPI2_NSS**—SPI 控制寄存器 2，从设备选择。这是一个可选的管脚，用来选择主/从设备。 **I2C2_SMBAI**—I2C 寄存器 2 信号线 **USART3_CK**—通用同步异步收发器 3 信号线 **TIM1_BKIN**—高级控制定时器 1 信号线
PB13/SPI2_SCK/USART3_CTS /TIM1_CH1N	34	I/O	**PB13**—串行接口 13 **SPI2_SCK**—SPI 控制寄存器 2，SCK 用作时钟，主模式中的 MOSI 或从模式中的 MISO 用作数据通信 **USART3_CTS**—通用同步异步收发器 3 信号线 **TIM1_CH1N**—高级控制定时器 1 信号线
PB14/SPI2_MISO/USART3_RTS/ TIM1_CH2N	35	I/O	**PB14**—串行接口 14 **SPI2_MISO**— SPI 控制寄存器 2,主设备输出/从设备输入管脚 **USART3_RTS**—通用同步异步收发器 3 信号线 **TIM1_CH2N**—高级控制定时器 1 信号线
PB15/SPI2_MOS/TIM1_CH3N	36	I/O	**PB15**—串行接口 15 **SPI2_MOS**—SPI 控制寄存器 2,主设备输出/从设备输入管脚

www.armjishu.com

			TIM1_CH3N—高级控制定时器 1 信号线
PB15/SPI2_MOS/TIM1_CH3N	36	I/O	**PB15**—串行接口 15 **SPI2_MOS**—SPI 控制寄存器 2,主设备输出/从设备输入管脚 **TIM1_CH3N**—高级控制定时器 1 信号线
PC0/ADC12_IN10	8	I/O	**PC0**—通用数字输出\输入接口 0 ADC12_IN10— ADC2 接口时钟输入
PC1/ADC12_IN11	9	I/O	**PC1**—通用数字输出\输入接口 1 **ADC12_IN11**— ADC2 接口时钟输入
PC2/ADC12_IN12	10	I/O	**PC2**—通用数字输出\输入接口 2 **ADC12_IN12**— ADC2 接口时钟输入
PC3/ADC12_IN13	11	I/O	**PC3**—通用数字输出\输入接口 3 **ADC12_IN13**— ADC2 接口时钟输入
PC4/ADC12_IN14	24	I/O	**PC4**—通用数字输出\输入接口 4 **ADC12_IN14**—模拟/数字转换信号输入 14
PC5/ADC12_IN15	25	I/O	**PC5**—通用数字输出\输入接口 5 **ADC12_IN15**—模拟/数字转换信号输入 15
PC6	37	I/O	**PC6**—通用数字输出\输入接口 6
PC7	38	I/O	**PC7**—通用数字输出\输入接口 7
PC8	39	I/O	**PC8**—通用数字输出\输入接口 8
PC9	40	I/O	**PC9**—通用数字输出\输入接口 9
PC10	51	I/O	**PC10**—通用数字输出\输入接口 10
PC11	52	I/O	**PC11**—通用数字输出\输入接口 11
PC12	53	I/O	**PC12**—通用数字输出\输入接口 12
PC13/TAMPER/RTC	2	I/O	**PC13**—计算机接口，可以作为通用 I/O 口、TAMPER 引脚、RTC 校准时钟、RTC 闹钟或秒输出，当后备区域由 VBAT 供电时(VDD 消失后模拟开关连到 VBAT)，可以作为 TAMPER 引脚、RTC 闹钟或秒输出 **TAMPER**—侵入检测引脚 **RTC**—校准时钟
PC14/OSC32_IN	3	I/O	**PC14**—计算机接口，可以用于 GPIO 或 LSE 引脚。当后备区域由 VBAT 供电时(VDD 消失后模拟开关连到 VBAT)，只能用于 LSE 引脚 **OSC32_IN**—LSE 振荡器引脚。晶体振荡输入
PC15/OSC32_OUT	4	I/O	**PC15**—计算机接口，可以用于 GPIO 或 LSE 引脚。当后备区域由 VBAT 供电时(VDD 消失后模拟开关连到 VBAT)，只能用于 LSE 引脚 **OSC32_OUT**—LSE 振荡器引脚。晶体振荡输出
PD2/ TIM3_ETR	54	I/O	**PD2**—GPIO 端口 **TIM3_ETR**—高级控制定时器 3 信号线
BOOT0	60	I	**BOOT0**—启动配置引脚选择三种不同启动模式
OSC_IN	5	I	**OSC_IN**—外部振荡器引脚引脚输入
OSC_OUT	6	O	**OSC_OUT**—外部振荡器引脚引脚输出
NRST	7	I/O	**NRST**—复位引脚，当输入低电平时系统复位
Vss_1	31	S	**Vss_1**—接地引脚
Vdd_1	32	S	**Vdd_1**—为 I/O 引脚和内部调压器供电
Vss_2	47	S	**Vss_2**—接地引脚
Vdd_2	48	S	**Vdd_2**—为 I/O 引脚和内部调压器供电
Vss_3	63	S	**Vss_3**—接地引脚

www.armjishu.com

Vdd_3	64	S	**Vdd_3**—为 I/O 引脚和内部调压器供电
Vss_4	18	S	**Vss_4**—接地引脚
Vdd_4	19	S	**Vdd_4**—为 I/O 引脚和内部调压器供电
Vssa	12	S	**Vssa**—模仪接地引脚
Vdda	13	S	**Vdda**—为 ADC、复位模块、RC 振荡器和 PLL 的模拟部分提供供电
Vbat	1	S	**Vbat**—为 RTC 和后备寄存器供电

1）I = 输入，O = 输出，S = 电源，HiZ = 高阻
2）FT：容忍 5V

1.3系统控制模块

1.3.1 系统控制模块功能汇总

系统控制模块包括一些系统特性和控制寄存器，它们的许多功能与特定的外设无关，这些模块包括：

- 复位
- 掉电检测
- 外部中断输入
- 各种系统控制和状态
- 代码安全与调试

为了满足将来扩展的需要，每种类型的功能都有其对应寄存器，不需要的位被定义为保留位。不同的功能不共用相同的寄存器地址。

1.3.2 引脚描述

引脚名称	类型	描述
NRST 复位	输入	外部复位输入----低电平有效，其中 I/O 口和外设将恢复其默认状态
EXIT 中断	输入	外部中断输入----低电平/高电平或下降/上升沿有效的通用中断输入，该引脚可用于将处理器从睡眠、深度睡眠或掉电模式中唤醒

1.3.3 寄存器描述

寄存器名称	描述	类型	复位值	地址
外 部 中 断				
EXTI_IMR	中断屏蔽寄存器	R/W	0x00000000	0x40010400
EXTI_EMR	事件屏蔽寄存器	R/W	0x00000000	0x40010404
EXTI_RTSR	上升沿触发选择寄存器	R/W	0x00000000	0x40010408
EXTI_FTSR	下降沿触发选择寄存器	R/W	0x00000000	0x4001040C
EXTI_SWIER	软件中断事件寄存器	R/W	0x00000000	0x40010410
EXTI_PR	挂起寄存器	Rc_w1	0xXXXXXXXX	0x40010414

备注：read/clear (rc_w1)：软件可以读此位，也可以通过写'1'清除此位，写'0'对此位无影响

1.3.4 复位

1.描述

复位是 CPU 的初始化操作，其目的是使 CPU 及各个寄存器处于一个确定的初始状态，把 PC 初

始化为 0000H，使单片机从 0000H 单元开始执行程序，即让处理器从第一条指令开始执行程序等。系统正常上电即可以复位，复位系统不可或缺的，它和时钟系统有着同等重要的作用，比如一个计算机系统的复位不可靠将带来很多意想不到的麻烦。

STM32F10xxx 支持三种复位形式，NRST 引脚复位、看门狗复位、上电复位和掉电复位、掉电检测复位，软件复位。

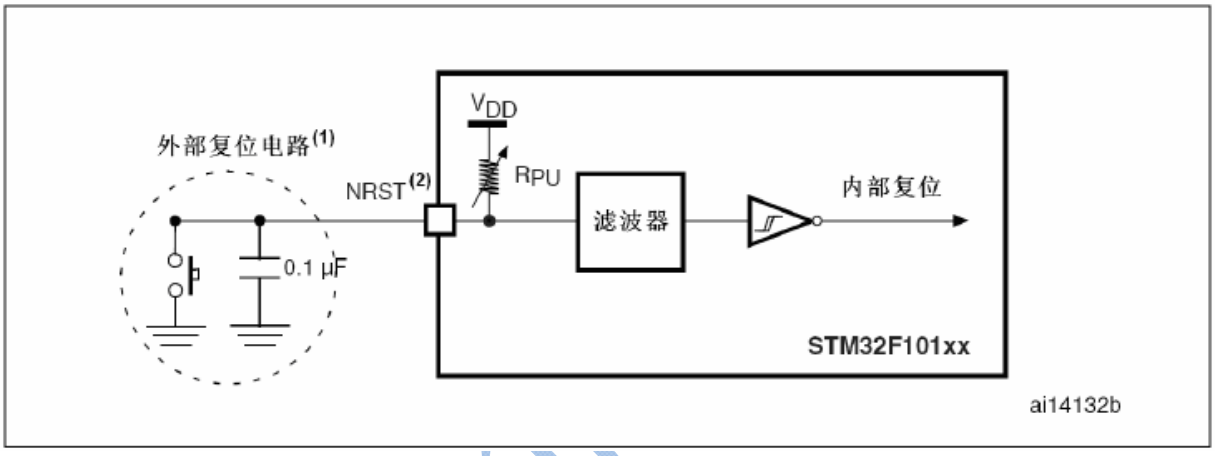
（1）NRST 引脚复位

外部复位是通过把芯片 NRST 引脚拉为低电平使芯片复位。复位信号出现以后不能立即撤除，必须在晶振运行稳定并且 STM32F013XXX 的 OSC_IN 脚上出现符合规定的时钟信号时才能撤除复位信号（表示晶振已经从加电开始正常工作）；

实际应用时，我们并不需要去检测 OSC_IN 引脚的波形来判断是否可以撤除复位信号，只需按照芯片厂家给出的官方数据手册进行操心即可，而常用的复位器件很容易满足这些参数。

- 如果使用的是外部晶振，上电后 NRST 脚上的复位信号至少要保持 10ms
- 若晶振已稳定运行且 OSC_IN 脚上已出现稳定信号，而 NRST 脚的信号只需保持 300ns

图24　建议的NRST引脚保护



STM32F103XXX 的 NRST 增加一个 0.1uf 的电容，该电容充当了一个额外的干扰滤波器，该滤波器可以滤除非常短促的脉冲信号，使处理器不会被干扰脉冲意外复位或不稳定的复位信号复位多次。

（2）看门狗复位

STM32F103XXX 内置看门狗部件，用户可以利用看门狗来复位处理器，看门狗复位时，将复位整个系统，具体操作方法参看“看门狗定时器”的具体章节。

STM32F103RBT 有 2 个看门狗，分别为独立看门狗和窗口看门狗：

---------独立看门狗：独立看门狗是基于一个 12 位的递减计数器和一个 8 位预分频器，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，所以称它为独立看门狗，它的特点是可以运行于停机和待机模式，或作为一个自由定时器（正因为它具有与主时钟不同的独立时钟）为应用程序提供超时管理。

---------窗口看门狗：窗口看门狗内有一个 7 位的递减计数器，它是由主时钟驱动，也正因为它是主时钟驱动的，并且它只有 7 位递减计数器，少于独立看门狗的 12 位递减计数器，所以它的特点是时间较独立看门狗要短，相当于具有早起的预警功能。

（3）上电复位（POR）

STM32F10XXX 具有上电复位功能，当供电电压由低向高上升越过规定的阀值（2V 左右）之前，保持芯片内部系统一直处于复位状态，当越过这个阈值后的一小段时间后（下图中的“滞后时间”），结束复位并开始执行指令，芯片开始工作，上电复位无需外部复位电路。这个阈值就是表中的倒数第 4 行（最小值=1.8，典型值=1.88，最大值=1.96）

（4）掉电复位（PDR）

当 VDD 引脚电压变化值低于设定的阈值（通常 Vpdr，STM32 芯片大约是 2V 左右）时，处理器

内部的掉电检测器将触发复位信号，使芯片内部产生复位，这个阈值就是表中倒数第 3 行（最小值=1.84，典型值=1.92，最大值=2.0）；低电压下，片内各种功能部件的操作都变得不可靠，掉电检测复位可防止 Flash 的内容发生改变。

（5）关于上电复位（POR）与掉电复位（PDR）的区别

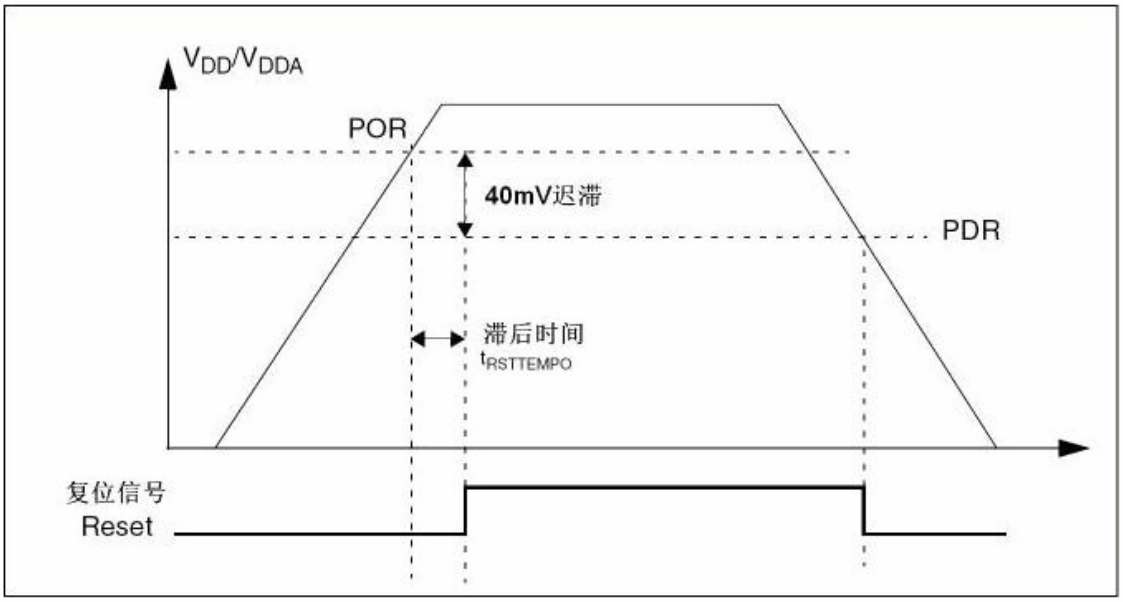
可以看到 POR 比 PDR 大了 0.04v，这就是表中倒数第 2 行，VPDRhst（PDR 迟滞）=40mV，掉电复位的值在低于 POR 这个值时还没有被进行复位操作，主要是要获得稳定可靠的复位信号，有这个迟滞范围，就不会被一些毛刺信号所影响到。

当 VDD 上升越过 POR 阈值时，内部并不马上结束复位，而是等待一小段时间，这就是表中最后一行 Trsttempo，它的典型值是 2.5ms，也是使得上电复位信号成为一个稳定的信号，消除信号的抖动，才给复位操作。

这个滞后时间是为了等待供电电压能够升高到最低可靠工作电压以上，我们看到 POR 阈值最小值有 1.8V，最大也只有 1.96V，都低于数据手册中给出最低可靠工作电压 2.0V，所以这个滞后时间是十分必要的，如果供电电压上升缓慢，尤其是从 1.8V 升到 2.0V 以上超过 2.5ms，则很可能造成上电复位后芯片不能正常工作的情况。

符号	参数	条件	最小值	典型值	最大值	单 位
$V_{POR/PDR}$	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
$V_{PDRhyst}$(2)	PDR迟滞			40		mV
$T_{RSTTEMPO}$(2)	复位持续时间		1	2.5	4.5	ms

图4　　上电复位和掉电复位的波形图



（6）掉电检测复位（PVD）

STM32F103XXX 包含一个 VDD/VDDA（3V3）引脚电压的二级检测。当 VDD/VDDA（3V3）电压变化至阈值 Vpvd（可以通过程序代码来设置这个电压 Vpvd）左右时就会产生中断，中断处理程序可以发出警告信息或将微控制器转入安全模式，该功能需要通过程序开启。

- 掉电检测阈值

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 000：2.2V　　100：2.6V 001：2.3V　　101：2.7V 010：2.4V　　110：2.8V 011：2.5V　　111：2.9V 注：详细说明参见数据手册中的电气特性部分。

这个阈值是相对的，可以通过电源控制寄存器（PWR_CR）选择，从 2.2V---2.9V 不等，正常工作时，掉电检测电路对这些阈值的反馈都有一些滞后，这一滞后使得不会因为突然的毛刺起伏来影响到芯片正常工作，从而得到相对稳定可靠的中断信号。

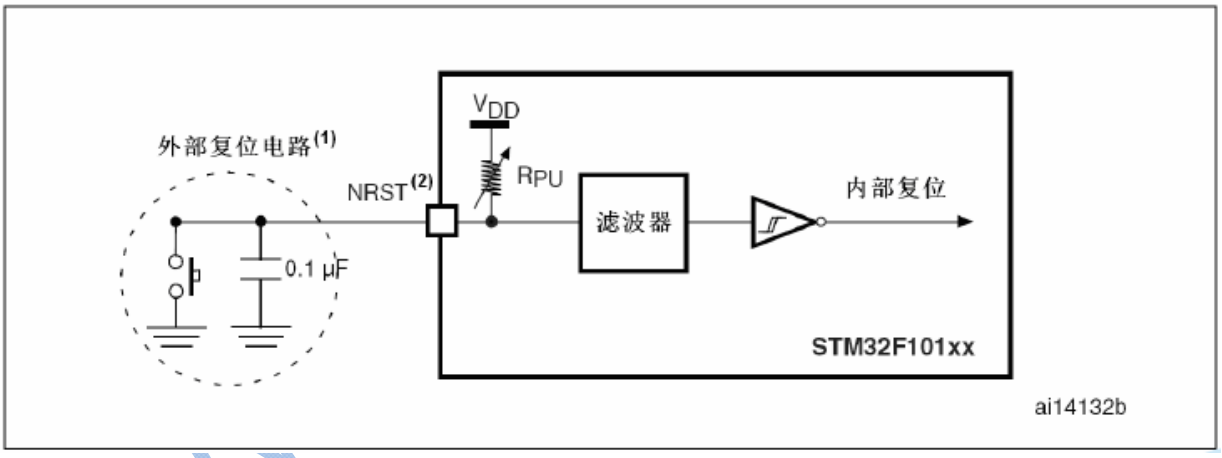
（7）软件复位

通过将 Cortex-M3 中断应用和复位控制寄存器中的 SYSRESETREQ 位置'1',可实现软件复位

2. 硬件复位流程

STM32F103XXX 的 NRST 引脚为**施密特**触发（下面有关于施密特触发的解释）输入引脚，任何复位源可使其复位有效，一旦操作电压达到规定的门限值，该引脚就会发生突变。复位信号将保持有效直至外部的复位信号被撤除，振荡器开始运行，当时钟计数超过了固定的时钟个数后，Flash 控制器和其他一些控制器就会完成其初始化。

图24　建议的NRST引脚保护



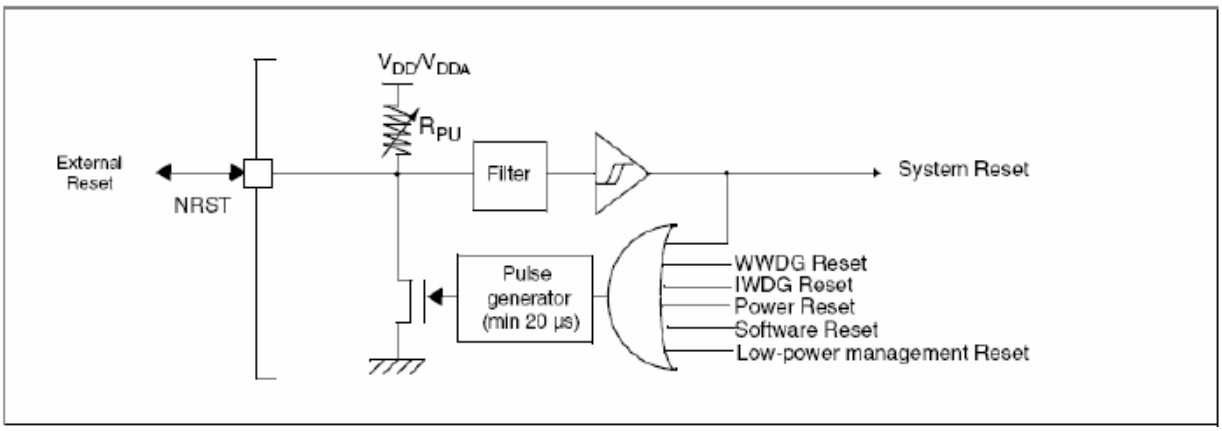


图 复位电路

当 CPU 的 VDD 电源达到 2.0V 时，片内 RC 振荡器开始起振，晶振起振以后，需经过一段时间才能稳定。此外，在上电过程中，复位信号需保持一段时间低电平，直至振荡信号稳定。

一旦晶振稳定且外部复位信号撤销，系统内的唤醒定时器即开始对振荡时钟计数，计满后，处理器和所有外设寄存器都恢复为默认状态。

关于施密特触发器：该触发器门电路有一个阈值电压，当输入电压从低电平上升到阈值电压或从高电平下降到阈值电压时电路的状态将发生变化。施密特触发器是一种特殊的门电路，与普通的门电路不同，施密特触发器有两个阈值电压，分别称为正向阈值电压和负向阈值电压。在输入信号从低电平上升到高电平的过程中使电路状态发生变化的输入电压称为正向阈值电压，在输入信号从高电平下降到低电平的过程中使电路状态发生变化的输入电压称为负向阈值电压。正向阈值电压与负向阈值电压之差称为回差电压。

它是一种阈值开关电路，具有突变输入——输出特性的门电路。这种电路被设计成阻止输入电压出现微小变化（低于某一阈值）而引起的输出电压的改变。

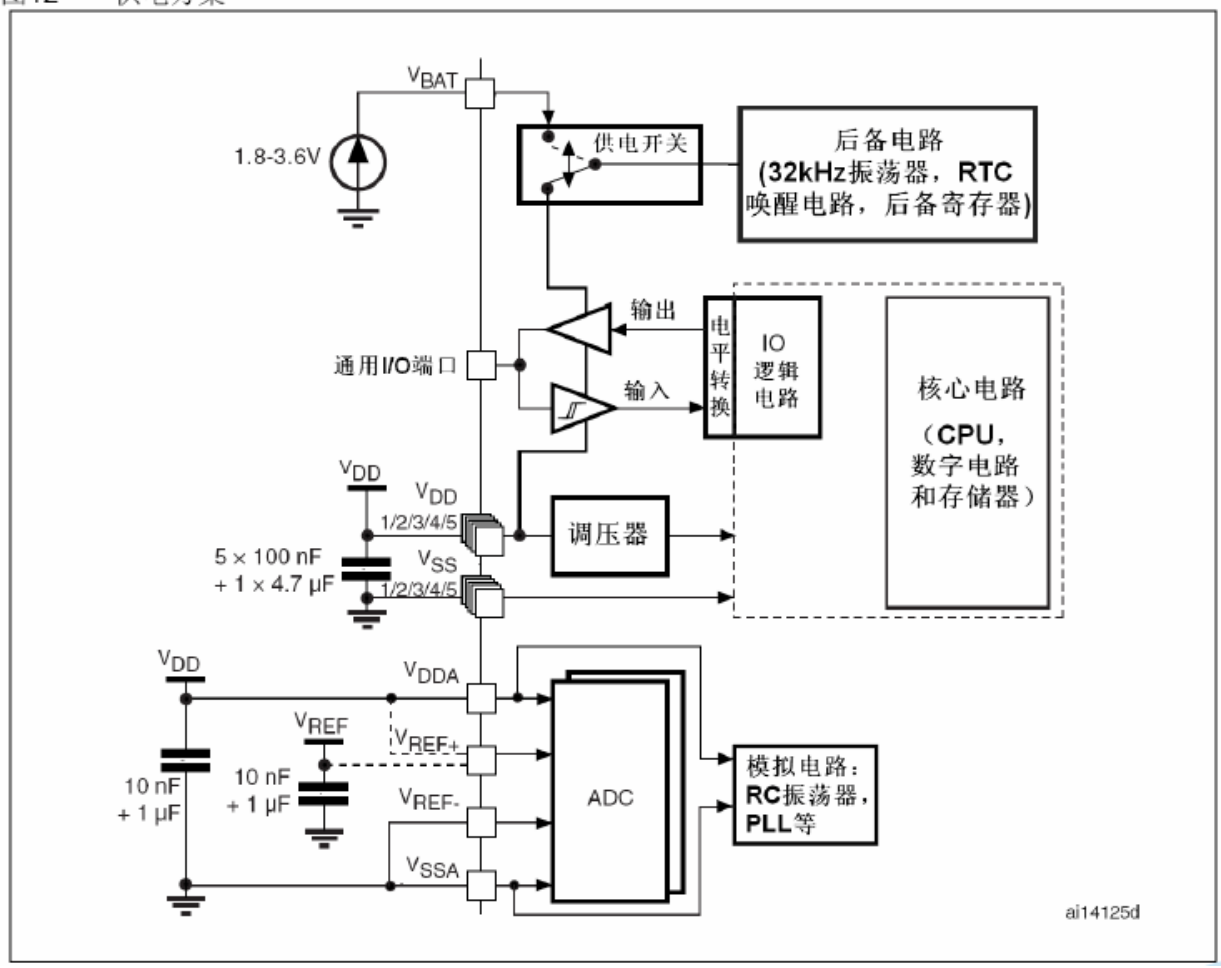
3. 关于电源

STM32F103XXX 含有 3 组电源引脚---VDD、VBAT 和 VSSA

- VDD（数字电源）：VDD=2.0~3.6V，VDD 引脚为 I/O 引脚和内部调压器供电
- VDDA（模拟电源）：VDDA=2.0~3.6V，为 ADC、复位模块、RC 振荡器和 PLL 的模拟部分提供供电。使用 ADC 时，VDDA 不得小于 2.4V。VDDA 和 VSSA 必须分别连接到 VDD 和 VSS。
- VBAT（纽扣电池电源）：VBAT=1.8~3.6V，当关闭 VDD 时，（通过内部电源切换器）为 RTC、外部 32kHz 振荡器和后备寄存器供电。

关于如何连接电源引脚的详细信息，参见下图的供电方案：

图12　供电方案



一般说来，各个电源引脚的上电是无顺序的。

4. 各复位源区别

当电源电压V_{DD}/V_{DDA}低于指定的限位电压V_{POR}/V_{PDR}（上电复位电压或掉电复位电压）时，系统将一直保持为复位状态，而无需外部复位电路，此时芯片不能正常的工作。关于上电复位和掉电复位

电压的关键阈值细节请参考数据手册的电气特性部分，CPU的电源电压Vdd/Vdda只有超过大于这个上电复位的阈值才能结束复位状态，开始正常工作；当CPU的电源电压Vdd/Vdda低于掉电复位电压的关键阈值时，CPU会由正常工作的状态变成复位状态。

下面关于一些不同复位源的区别比较，从软件方面复位和硬件方面复位，上电方面复位和掉电方面复位等，以及内部方面复位和外部方面复位来做分析，具体可以参考下面：

- 上电复位和掉电复位使特定引脚的值锁存已配置器件（例如电源引脚 VDD 或 VDDA，这个值是芯片出厂就设定好了的），其它复位无此功能（请见下图，上电复位的典型值是 1.88V）
- 上电复位和掉电复位的一个重要区别在于两者阈值不同：两者相差 0.04v

符号	参数	条件	最小值	典型值	最大值	单位
$V_{POR/PDR}$	上电/掉电复位阈值	下降沿	1.8[1]	1.88	1.96	V
		上升沿	1.84	1.92	2.0	V
$V_{PDRhyst}$[2]	PDR迟滞			40		mV
$T_{RSTTEMPO}$[2]	复位持续时间		1	2.5	4.5	ms

Vpor

Vpdr

- 外部复位和看门狗复位也有一些小的区别。外部复位之后，处理器首先会判断引脚 BOOT0，BOOT1（PB2）的状态，从而决定是否从哪里启动（SRAM，串口，FLASH 三种启动方式），而看门狗复位则无此功能。

BOOT1（J8）	BOOT0（J7）	功能	说明
X	0	User Boot(默认)	用主闪存存储器，即Flash启动
0	1	System Boot	系统存储器启动，用于串口下载
1	1	SRAM Boot	SRAM启动

5. 复位

系统时钟的选择是再启动时进行，复位内部8MHz的RC振荡器被选为默认的CPU时钟，随后可以选择外部、具有失效监控的4~16MHz时钟；在系统复位后，BOOT管脚的值将被锁存，用户可以通过设置BOOT1和BOOT0引脚的状态，来选择在复位后的启动模式。在一段时间内，BOOT管脚应保持为需要的启动配置，在启动延迟之后，因为固定的存储器映像，代码区始终从地址0开始。

Cortex-M3处理器复位后，从地址0开始运行程序；复位后的0地址处其实是Boot Block引导程序的地址，它做了一个地址映射，所以实际上，复位后首先运行的是Boot Block引导程序。

6. 引导块（Boot Block）

引导块（Boot Block）是芯片生成时由厂家固化在其中的一段代码，我们无法修改或删除，这段代码在复位后被首先运行。

该引导块通过在处理器上电时判断引脚 BOOT0，BOOT1（PB2）的状态，从而决定是否从哪里启动（SRAM，串口，FLASH 三种启动方式），该内嵌在 CPU 中的引导块位于 FLASH 中。

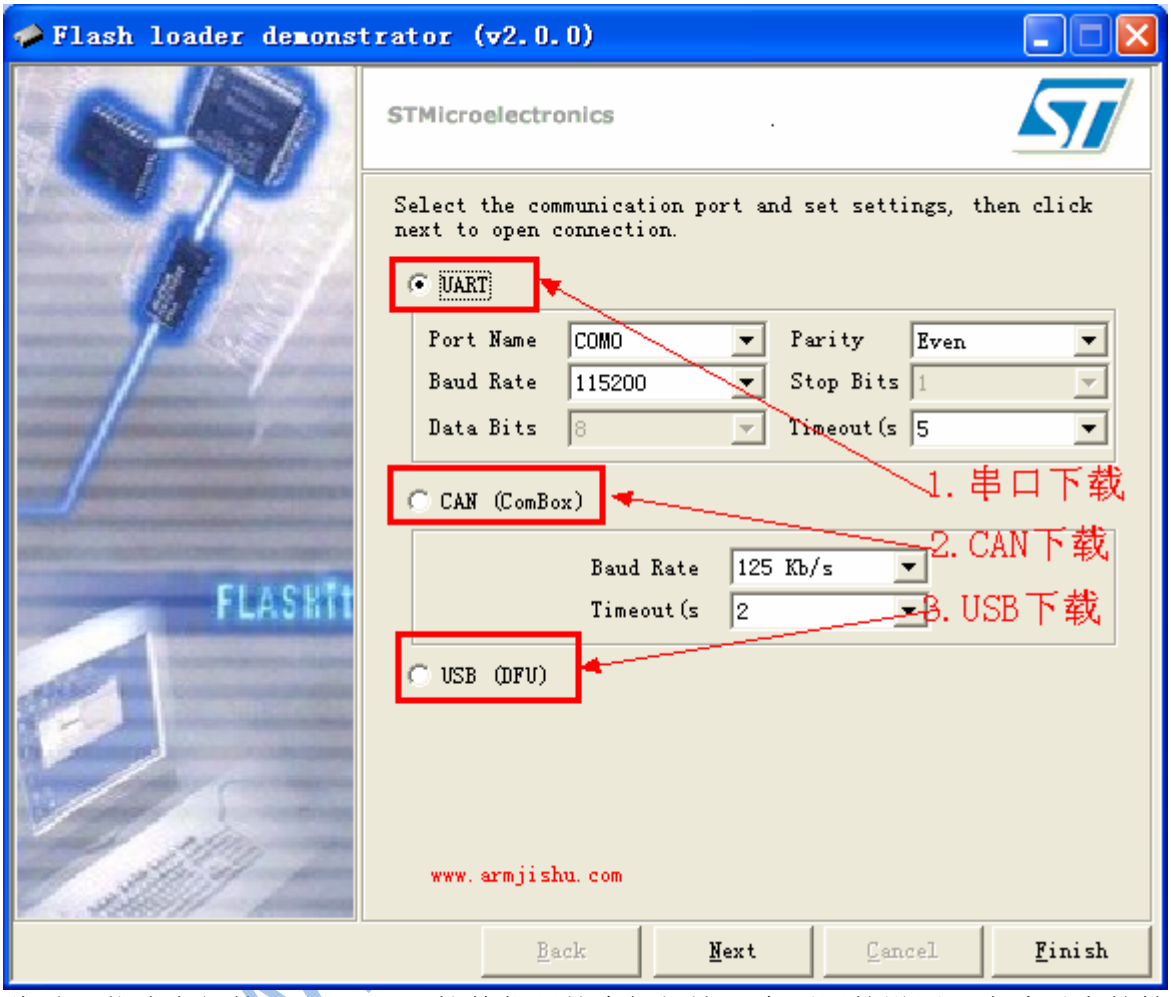
如果 STM32 处理器的 boot 引导模式如下时，：

- BOOT0 = 1
- BOOT1 = 0

在 CPU 进行 reset 复位后，这个两个引脚（BOOT0 和 BOOT1）的值会在 SYSCLK 的第个上升沿的时候会被读出来，看到底是从用户 flash 存储区启动，还是从内部 RAM 启动，还是从系统内部存储区启动（包括 USART 启动，或 CAN 启动，USB 设备几种不同的启动方式等），即当 BOOT0=1 和 BOO1=0 的时就会是这种启动方式，此时在 CPU 内部的引导块 Boot Block 就会进入 System Boot 模式：

BOOT1（J8）	BOOT0（J7）	功能	说明
X	0	User Boot(默认)	用主闪存存储器，即Flash启动
0	1	System Boot	系统启动，串口/CAN/USB下载
1	1	SRAM Boot	SRAM启动

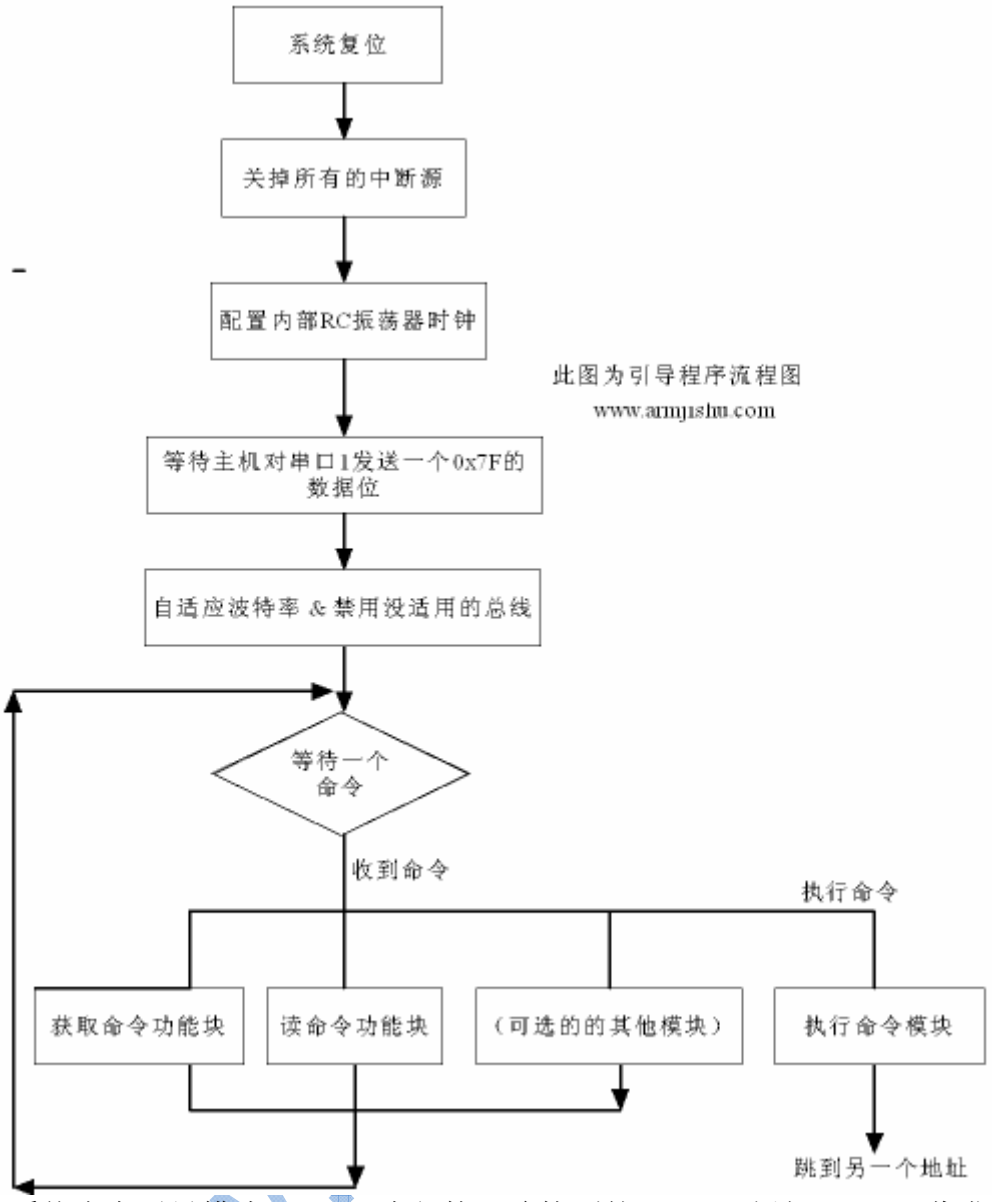
在这里，用户可以选择 3 种下载方式：



此时，芯片内部的 Boot Block 软件代码就会根据这 3 个不同的设置，去读对应的管脚，例如，请参见如何从串口下载一个程序到芯片里，如何从 USB 口下载一个程序到芯片里，请见对应的章节。

原理就是，芯片内部的引导程序根据我们在 PC 机上的对应选项，对 3 个不同的方式进行配置，例如，我选择的是串口，首先是 PC 机上的上位机软件识别到芯片的串口已经就绪了，开始进行存储容量，下载地址以及位置等细节的设置，包括芯片型号等信息；设置好了，再选择对应的二进制代码或者 HEX 文件，那么芯片内部的引导程序就会根据我们设置的配置文件，将代码下载 STM32 芯片中；

CAN 下载方式和 USB 下载方式都是类似，先让 PC 机上位机识别到芯片对应的 CAN 或 USB 接口 ，然后再在上位机软件中进行下载具体位置和目标芯片型号等配置的设置，最后选择要下载到 STM32 芯片中的程序，然后进行下载操作。



在系统内存引导模式，RS232 串行接口连接到的 STM32（例如，RS232 收发器 max3232 芯片）有直接联系的 USART1_RX（PA10）和 USART1_TX（PA9）引脚。

一旦系统内存引导模式输入和微控制器已被配置为从串口启动，如上所述，bootloader 代码开始扫描的 USART1_RX 的行针，等待收到 0x7F 的数据帧：一个起始位，0x7F 的数据位，偶校验位和一个停止位。

这个数据帧的持续时间使用 SysTick 定时器来衡量，根据收到的数据以及定时器的时间，计算出当前相应的波特率；接下来，该代码初始化串行接口。使用此计算波特率，发送确认字节（0x79）返回到主机，这标志着 STM32F10xxx 芯片已经准备接收用户命令。

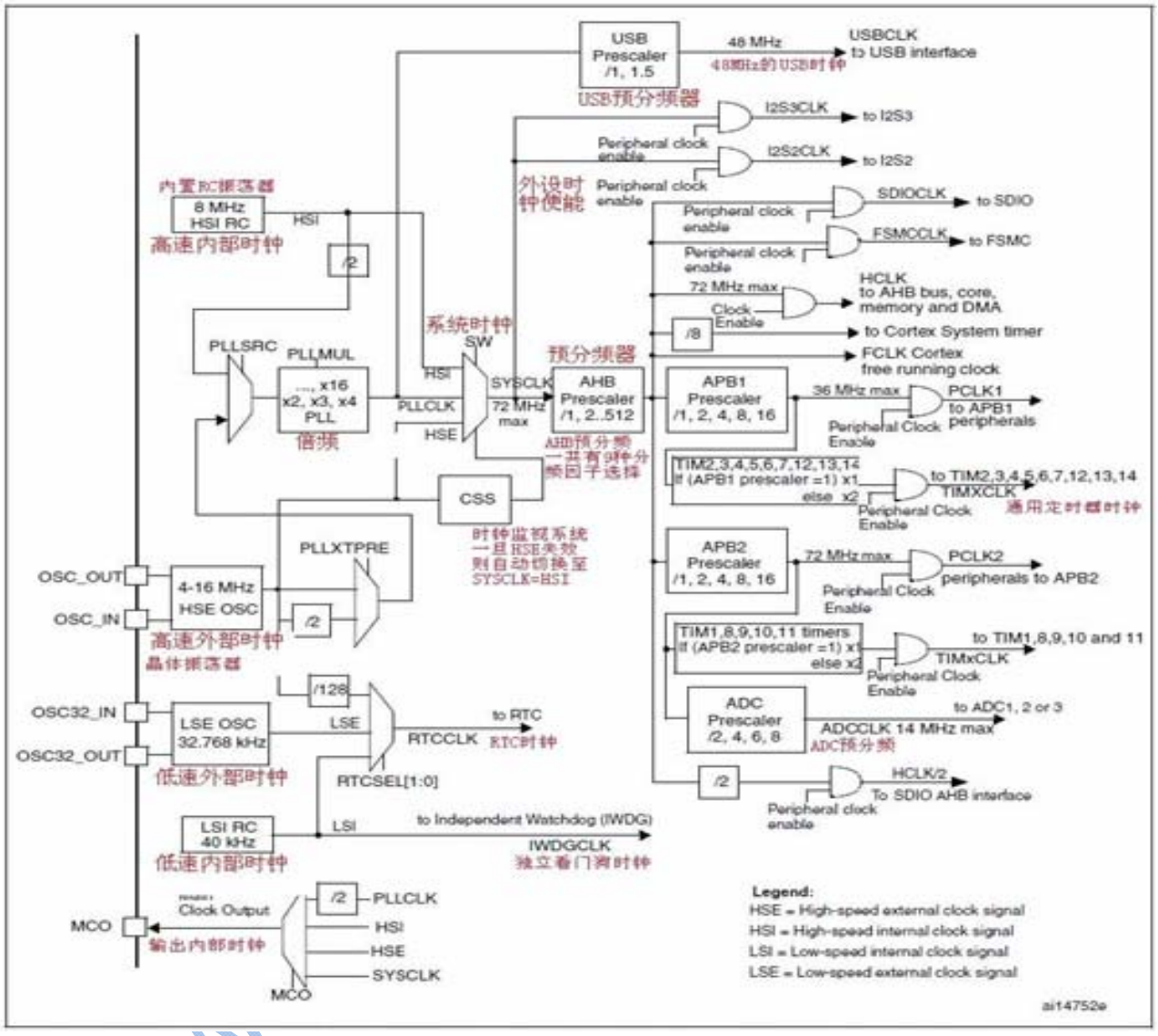
不过这里非常重要的一点，就是 STM32F103XXX 没有 USB 和 CAN 引导下载的功能。

1.4 时钟系统

1.4.1 时钟系统分析

STM32 芯片内部的时钟系统结构图如下所示，我们先对芯片内部的时钟树做一个分析，然后再来针对我们振荡器来做一个介绍，下面是各个模块的框架图：

时钟树：



1. STM32共有五个时钟来源：

在STM32 系统中，共有五个时钟源，分别为HSE、HSI、LSE、LSI、PLL。由图可以看出，HSI 和LSI 为片内RC 振荡器，HSI 为8MHz 而LSI 为40KHz；HSE 和LSE 为外部时钟源；PLL 则需要HSE 和HSI 来提供时钟。

（1）HSE：外部高速时钟信号

可以通过外部直接提供时钟，从OSC_IN 输入，或使用外部陶瓷/晶体谐振器。外部直接提供的时钟可以达到25MHz，用户可以通过设置时钟信号控制寄存器RCC_CR 中的HSEBYP 和HSEON 位来选择该模式。此时OSC_OUT 引脚为高阻状态。

（2）HSI：内部高速时钟信号

该时钟通过8MHz 的内部RC 振荡器产生，并且可被直接用做系统时钟，或者经过2 分频后作为PLL 的输入。它比HSE 有更快的启动时间，但频率精确度没有外部晶体振荡器

高。而且根据制造工艺的不同，不同芯片之间的RC 振荡器频率也是不同。出厂时，每个设备频率已被校准至1%（25 摄氏度）。出厂校验值被装载到时钟控制寄存器RCC_CR 的HSICAL [7：0] 位。在不同的电压或者温度下， 可以通过RCC_CR 中的HSITRIM[4：0] 位来调整HIS 的频率。并可以通过时钟控制寄存器RCC_CR 的HISON 位打开或者禁用。

（3） LSE：低速外部时钟信号

振荡器是一个32.768KHz 的低速外部晶体/陶瓷振荡器， 它可以向RTC 提供高精准时钟。

（4）LSI：低速内部时钟信号

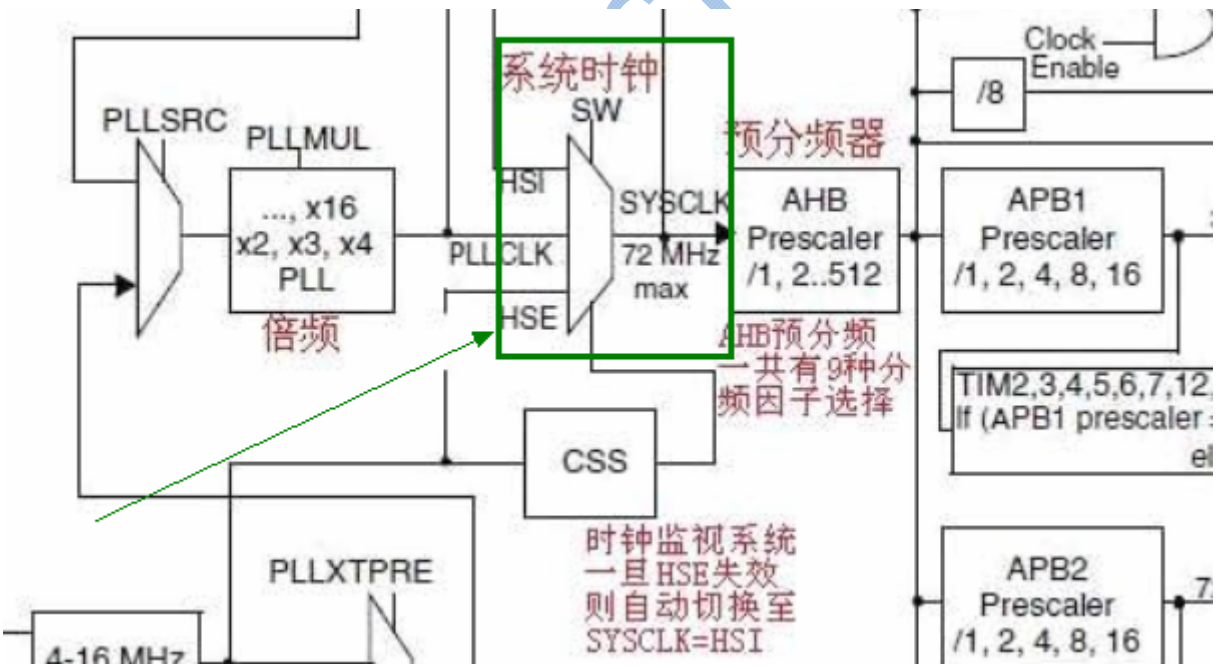
振荡频率为40KHz 左右（30-60KHz 之间）。

（5）PLL：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。

2. 系统时钟模块（SYSCLK）

系统时钟SYSCLK 是供STM32 中绝大部分部件工作的时钟源。详情请参见上面时钟树图，系统时钟可选择为HSI、PLLCLK或HSE输出。



系统时钟模块图

HSI 与HSE可以通过分频加至PLLSRC，并由PLLMUL 进行倍频后经选择直接充当SYSCLK。系统时钟最大频率为72MHz， 它通过AHB 分频器分频后送给各模块使用，AHB 分频器可选择1、2、4、8、16、64、128、256、512 分频。

AHB 分频器输出的时钟送给5 大模块使用：

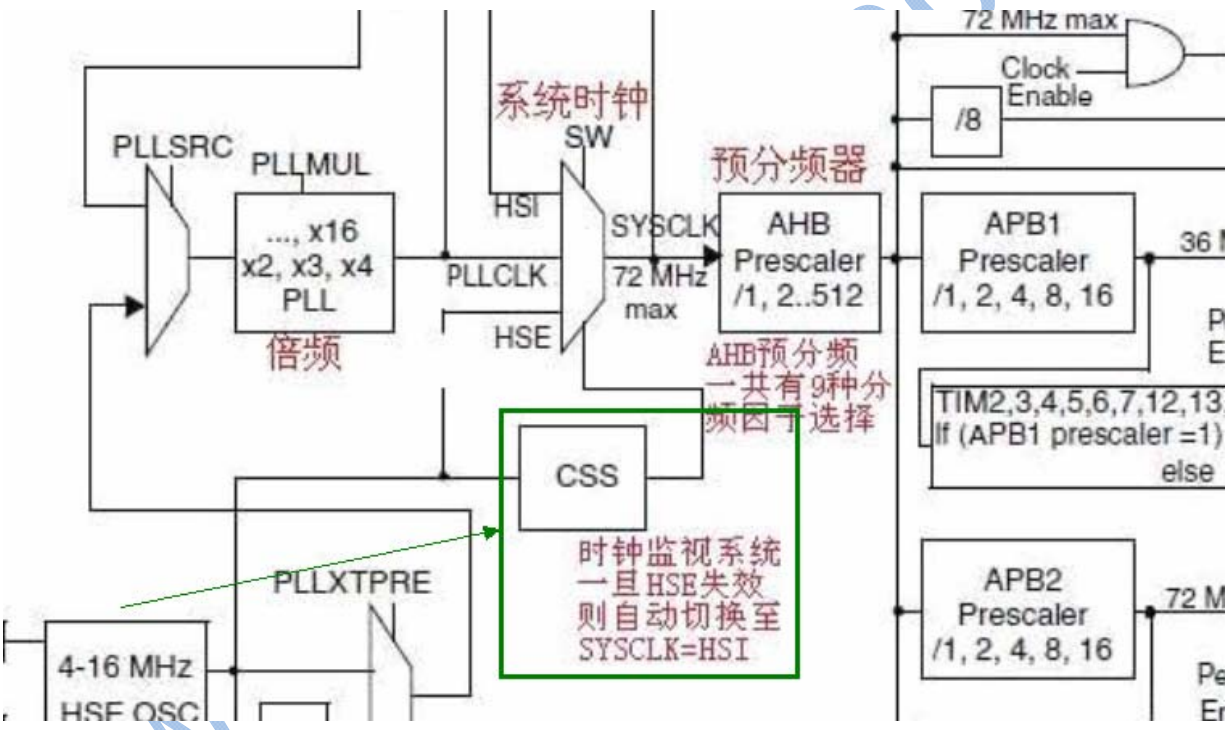
（1）送给AHB 总线、内核、内存和DMA 使用的HCLK 时钟。

（2）通过8 分频后送给Cortex 的系统定时器时钟。

（3）直接送给Cortex 的空闲运行时钟FCLK。
（4）送给APB1 分频器。APB1 分频器可选择为1、2、4、8、16 分频，其输出中一路供APB1 外设使用(PCLK1，最大频率36MHz)；另一路送给定时器(Timer)2、3、4 的倍频器使用（TIMXCLK），该倍频器可选择1或者2 倍频，输出供定时器2、3、4 使用。
（5）送给APB2 分频器。APB2 分频器可供选择为1、2、4、8、16 分频，其输出一路供APB2 外设使用(PCLK2，最大频率72MHz)；一路送给定时器(Timer)1 的倍频器使用（TIM1CLK），该倍频器可选择1 或者2 倍频，输出供定时器1 使用；另外，APB2 分频器还有一路输出供ADC 分频器使用， 分频后送给ADC 模块使用，ADC 分频器可选择为2、4、6、8 分频。

3. 时钟安全系统模块（CSS）

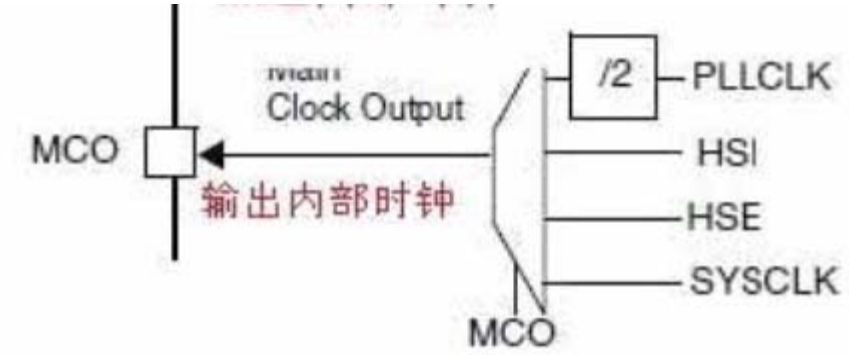
图中在SYSCLK 选择端下方有个CSS 模块，CSS 是一个时钟安全系统，用来保证系统时钟在HSE 失效时能继续工作。



时钟安全系统模块图

时钟检测器在HSE 振荡器启动延时后被使能，并当振荡器停止时禁用。如果在HSE振荡器时钟上检测到一个失效，这个振荡器将被自动禁用；一个时钟失效事件打断TIM1 高级控制定时器的输入， 并且产生一个中断来通知软件该次失效，使得MCU 能够进行补救措施。

4. 系统时钟输出模块（MCO）

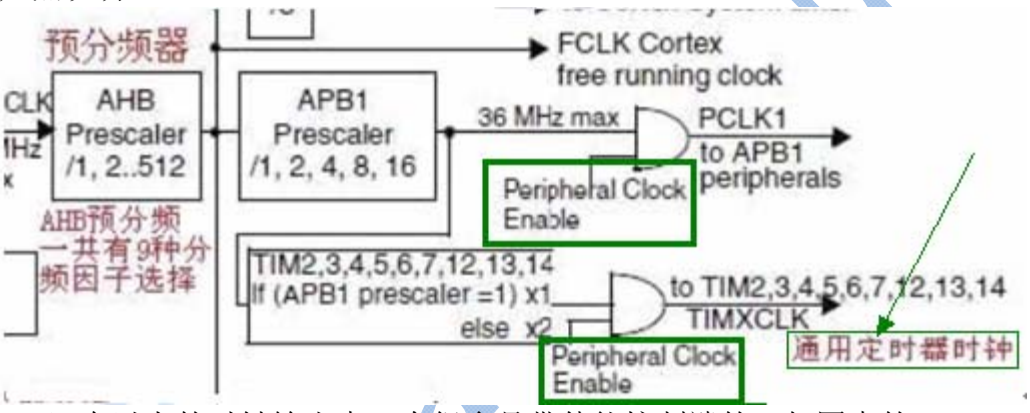


系统时钟输出（MCO）模块，使得时钟能够输出到外部MCO引脚，相关的GPIO 端口的配置寄存器必须被编程为复用功能模式。下面4个时钟的任意一个可被选作MCO时钟：

（1）SYSCLK
（2）HIS
（3）HSE
（4）2分频的PLLCLK

以上这些可通过STM32的时钟配置寄存器RCC_CFGR 的MCO[2：0]位进行选择。

几点说明:



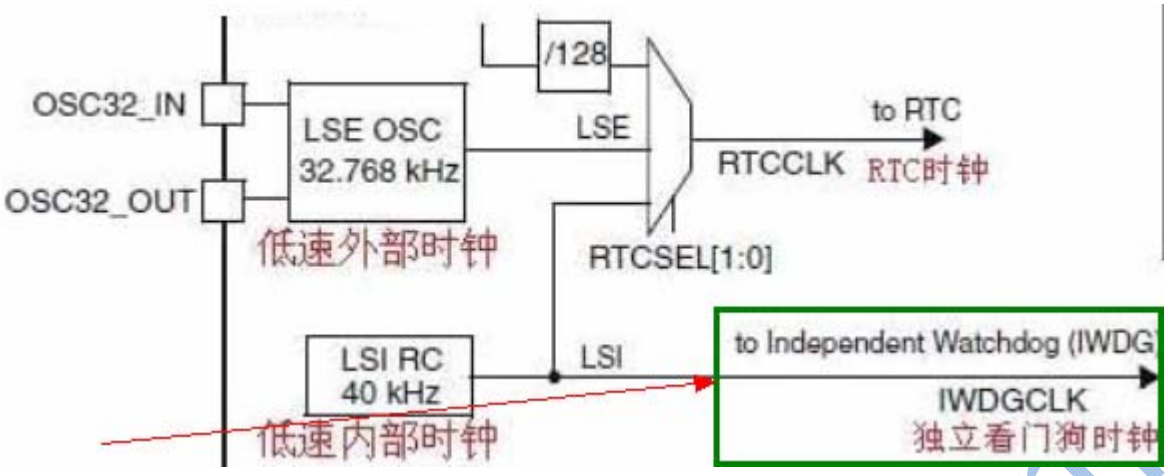
（1）在以上的时钟输出中，有很多是带使能控制端的（如图中的Peripheral Clock Enable），例如AHB 总线时钟、内核时钟、各种APB1外设、APB2 外设等等。当需要使用某模块时，记得一定要先使能对应的时钟。

（2）需要注意的是定时器的倍频器，当APB的分频为1 时，它的倍频值为1，否则它的倍频值就为2。

（3）连接在APB1(低速外设)上的设备有：电源接口、备份接口、CAN、USB、I2C1、I2C2、UART2、UART3、SPI2、窗口看门狗、Timer2、Timer3、Timer4。

（4）连接在APB2(高速外设)上的设备有：UART1、SPI1、Timer1、ADC1、ADC2、所有普通IO 口(PA~PE)、第二功能IO 口。

（5）如果独立的看门狗（IWDG）被硬件选项或者软件访问启动了，LSI 振荡器将被强制打开， 并且不能被禁用， 在LSI 振荡器开始工作后，它的时钟被提供给IWDG。



1.4.2 寄存器描述

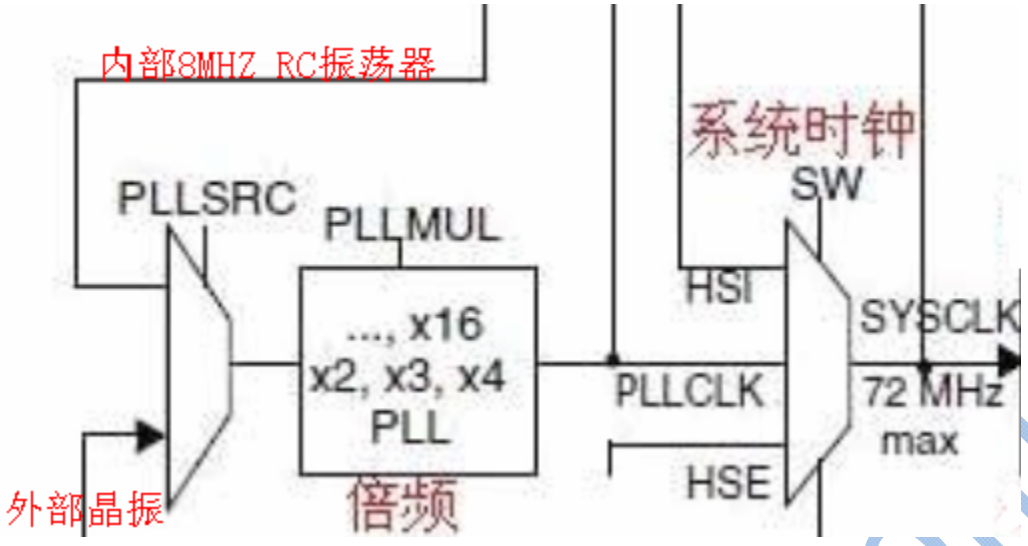
所有寄存器的功能都在下面：

名称	描述	访问	复位值	地址
时钟源选择（内部或外部高速时钟源）				
RCC_CR	时钟控制寄存器	R/W	0x0000 xx83	0x4000 2800
时钟分频和配置器（配置 PLL,USB,ADC,APB,AHB,SW 的时钟）				
RCC_CFGR	时钟配置寄存器	R/W	0x0000 0000	0x4000 2804
RCC_CIR	时钟中断寄存器	R/W	0x0000 0000	0x4000 2808
时钟复位（控制 AD/DA/CAN/I2C/TIM 等部件复位）				
RCC_APB2RSTR	APB2 外设复位寄存器	R/W	0x0000 0000	0x4000 280C
RCC_APB1RSTR	APB1 外设复位寄存器	R/W	0x0000 0000	0x4000 2810
时钟使能（控制 AD/DA/CAN/I2C/TIM/GPIO 等部件使能）				
RCC_AHBENR	AHB 外设时钟使能寄存器	R/W	0x0000 0014	0x4000 2814
RCC_APB2ENR	APB2 外设时钟使能寄存器	R/W	0x0000 0000	0x4000 2818
RCC_APB1ENR	APB1 外设时钟使能寄存器	R/W	0x0000 0000	0x4000 281C
备份区域 RTC 时钟的选择（控制 LSE,LSI,HSE 等外部内部时钟选择成 RTC）				
RCC_BDCR	备份域控制寄存器	R/W	0x0000 0000	0x4000 2820
复位使能（包括低功耗，看门狗，软件复位，上电/掉电和 NRST 复位等）				
RCC_CSR	控制/状态寄存器	R/W	0x0C00 0000	0x4000 2824

1.4.3 振荡器（STM32内部有RC做振荡器，外部有晶振做振荡器）

STM32 自动选择内部 8MHZ 的 RC 振荡器作为系统的时钟源，这使得系统能在没有外部晶振的情况下运行；BootLoader 程序也是使用内部 RC 振荡器作为时钟源。

用户可以通过软件方式修改时钟源选择寄存器，从而选择 3 种振荡器中的一种作为系统主时钟源。**注意，切换前必须保证即将使用的时钟源已经可用。**



所有振荡器（内部 8MHZ RC 振荡器或外部晶振等）在用作 CPU 时钟源时，可用通过 PLLSRC 获得较高的时钟，但注意，必须小于 72MHZ，因为 STM32F103XXX 芯片的最高主频就是 72MHZ。

1．内部 RC 振荡器

复位时内部 8MHHz 的 RC 振荡器被选为默认的 CPU 时钟，既可以直接对 CPU 提供时钟，也可以通过驱动 PLL（请见图，倍频模块）经过倍频后再为 CPU 提供时钟源；40KHz 的 RC 振荡器可用作看门狗定时器的时钟源。

2．外部晶振振荡器

外部晶振振荡器（主振荡器）可作为 CPU 的时钟源（不管是否使用 PLL 倍频）。主振荡器工作在 4MHz ~ 16MHz 下；该频率也可通过主 PLL 倍频来提高至 CPU 操作频率的最大值。

当检测外部时钟失效时，它将被隔离，系统将自动地切换到内部的 RC 振荡器。

3. RTC 振荡器

RTC 时钟可以是一个使用外部晶体的 32.768kHz 的振荡器，也可以使用内部的 40kHz 低功耗 RC 振荡器、或者是外部时钟经过 128 分频后再提供给 RTC 时钟都可。

1.6 存储器寻址

1. 片上Flash存储器系统

STM32微控制器内部Flash容量表如下表所示，片内Flash可用于代码和数据存储，对片内Flash的编程有三种方法：

- 使用JTAG仿真/JLINK V8调试器，通过芯片的JTAG接口下载程序；
- 使用在系统编程技术（即ISP），通过USART1串行接口下载程序
- 使用在应用编程技术（即IAP），通过IAP，可实现用户程序运行时对Flash进行擦除/编程，这样就为数据存储和现场固件的升级都带来了极大的灵活性。

器件	管脚数量	芯片内部FLASH（字节）	芯片SRAM（字节）
STM32F103C8T	48	64K	20K

STM32F103RBT	64	128K	20K
STM32F103VCT	100	256K	48K
STM32F103VET	100	512K	64K
STM32F103ZET	144	512K	64K

STM32F103RBT的主存储块为16Kb X 64位，每个主存储块划分为128个1K字节的页；这里要指出一点，STM32的Flash存储系统共有3种大小分别为：

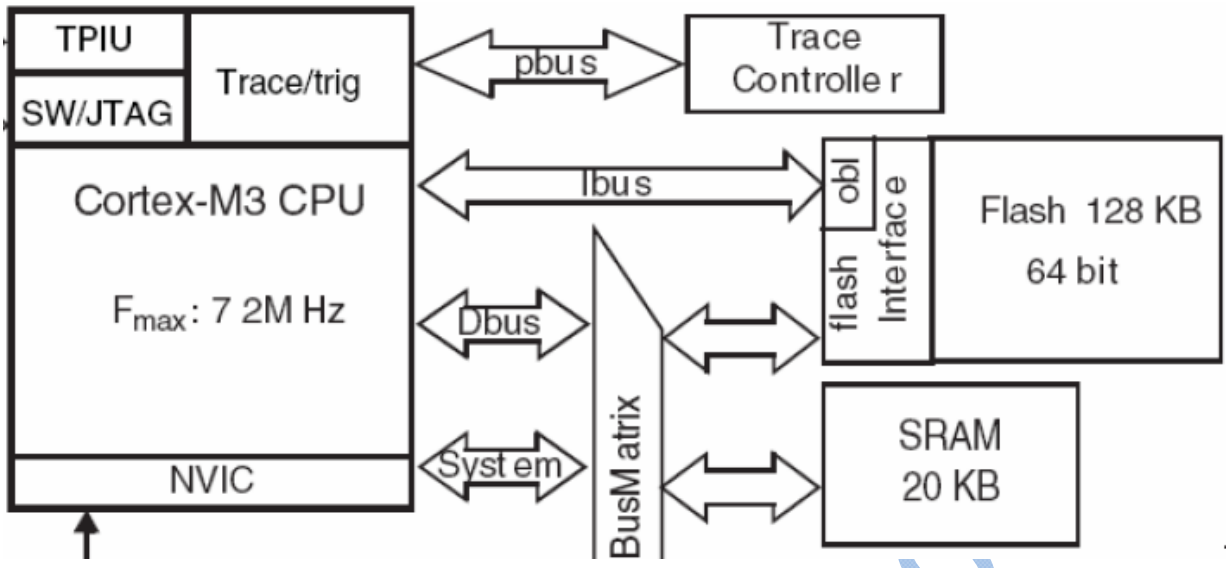
- 32K；主存储块为4Kb X 64位，每个主存储块划分为32个1K字节的页
- 128K；主存储块为16Kb X 64位，每个主存储块划分为128个1K字节的页
- 512K；主存储块为64Kb X 64位，每个主存储块划分为256个2K字节的页

下图是128K Flash的分页架构表：

模块	名称	地址	大小
主存储块	页0	0x0800 0000 – 0x0800 03FF	1K
	页1	0x0800 0400 – 0x0800 07FF	1K
	页2	0x0800 0800 – 0x0800 0BFF	1K
	页3	0x0800 0C00 – 0x0800 0FFF	1K
	页4	0x0800 1000 – 0x0800 13FF	1K
	…	…	…
	…	…	…
	页127	0x0801 FC00 – 0x0801 FFFF	1K
信息块	系统存储器	0x1FFF F000 – 0x1FFF F7FF	2K
	用户选择字节	0x1FFF F800 – 0x1FFF F80F	16
FLASH 存储器 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

2. 片内静态RAM

STM32F103RBT中集成了1块静态RAM，它可以用来运行程序代码或存储数据：



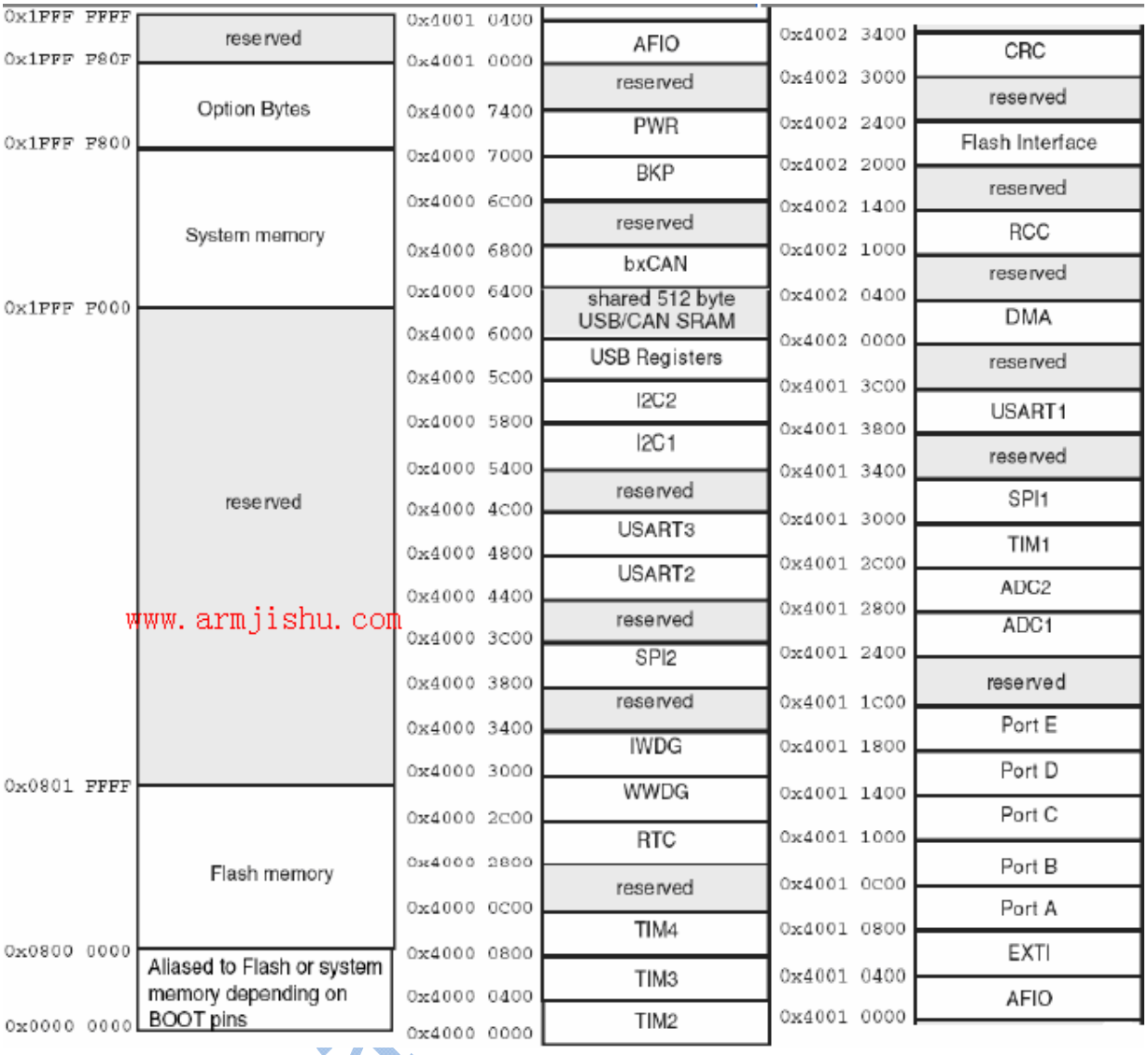
3. 存储器映射与外设寻址

程序存储器（片上Flash）、数据存储器（片内静态RAM即SRAM）、寄存器和输入输出端口被组织在同一个4GB的线性地址空间内，可访问的存储器地址空间被分成8个主要块，每个块为512MB。 数据字节以小端格式存放在存储器中（个字里的最低地址字节被认为是该字的最低有效字节，而最高地址字节是最高有效字节）。

地址范围	用途	描述
0x0000 0000-0x0800 0000	BOOT引导区域	128MB选择从Flash启动或还是系统内存启动区域
0x0800 0000-0x0801 FFFF	Flash存储	128KB片内FLASH（STM32F103RBT）
0x1FFF F000- 0x1FFF F800	系统存储	BOOTLAODER存储在这个区域必须通过BOOT1和BOOT0引脚才能访问到这个区域
0x2000 0000-0x2000 5000	SRAM区域	20K片内SRAM区域
0x4000 0000-0x4002 3400	APB外设	APB总线存储区域，包括以下外设（TIM,RTC,WDG,USART,I2C,USB,CAN,PWR,GPIOX,SPI,DMA,ADC,CRC）

4. 存储器映射

所谓的存储器映射，就是为存储器分配地址的过程，STM32存储空间由几个不同的存储区域组成，下么是复位后从用户编程角度所看到的地址空间映射：

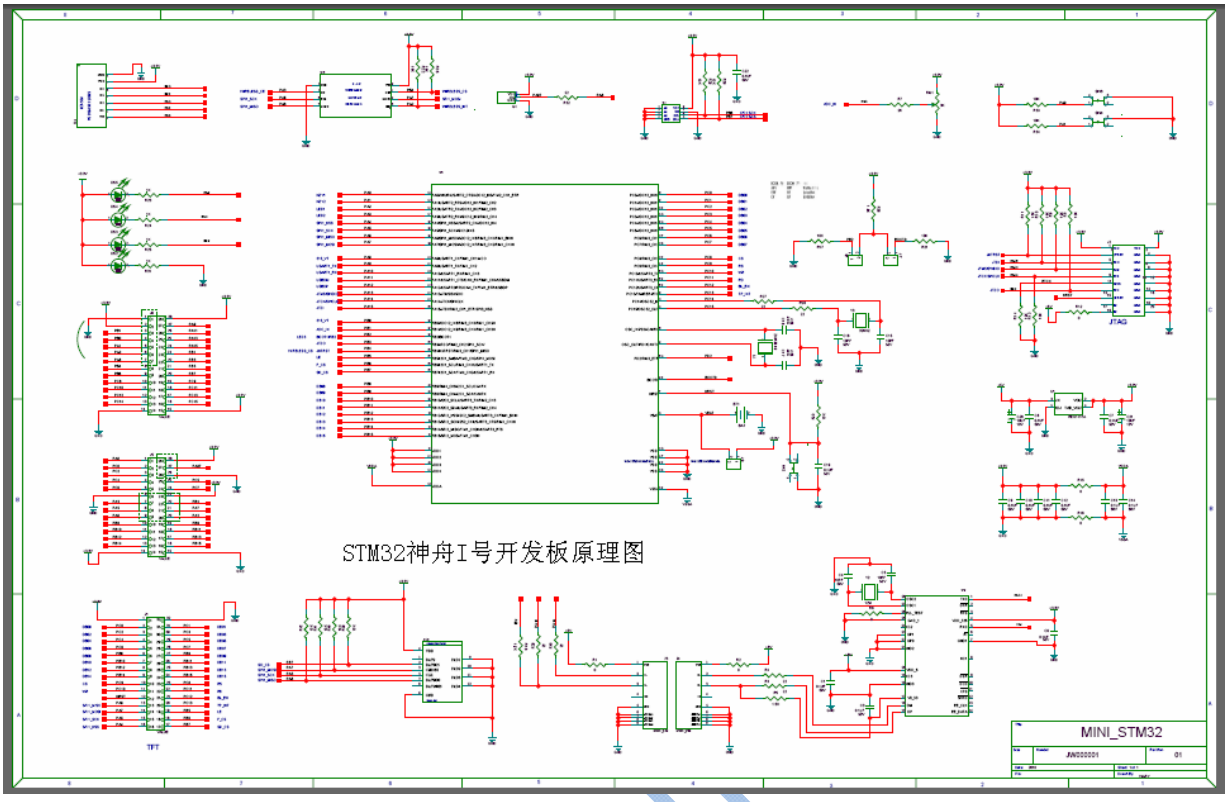


www.armjishu.com

第2章 神舟I号开发套件硬件结构

2.1 电路原理图

STM32神舟I号开发板的电路图和2.8寸触摸屏的电路图见产品光盘。



2.2 功能特点

神舟I号STM32开发板是一款基于STM32F103RBT6的开发板，面向学生以及想从事STM32开发的工作者等广大爱好者，神舟I号开发板功能强大，外围资源齐全，性价比非常高，尺寸为67mm * 82mm。

神舟I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：

www.armjishu

MCU 启动模式配置

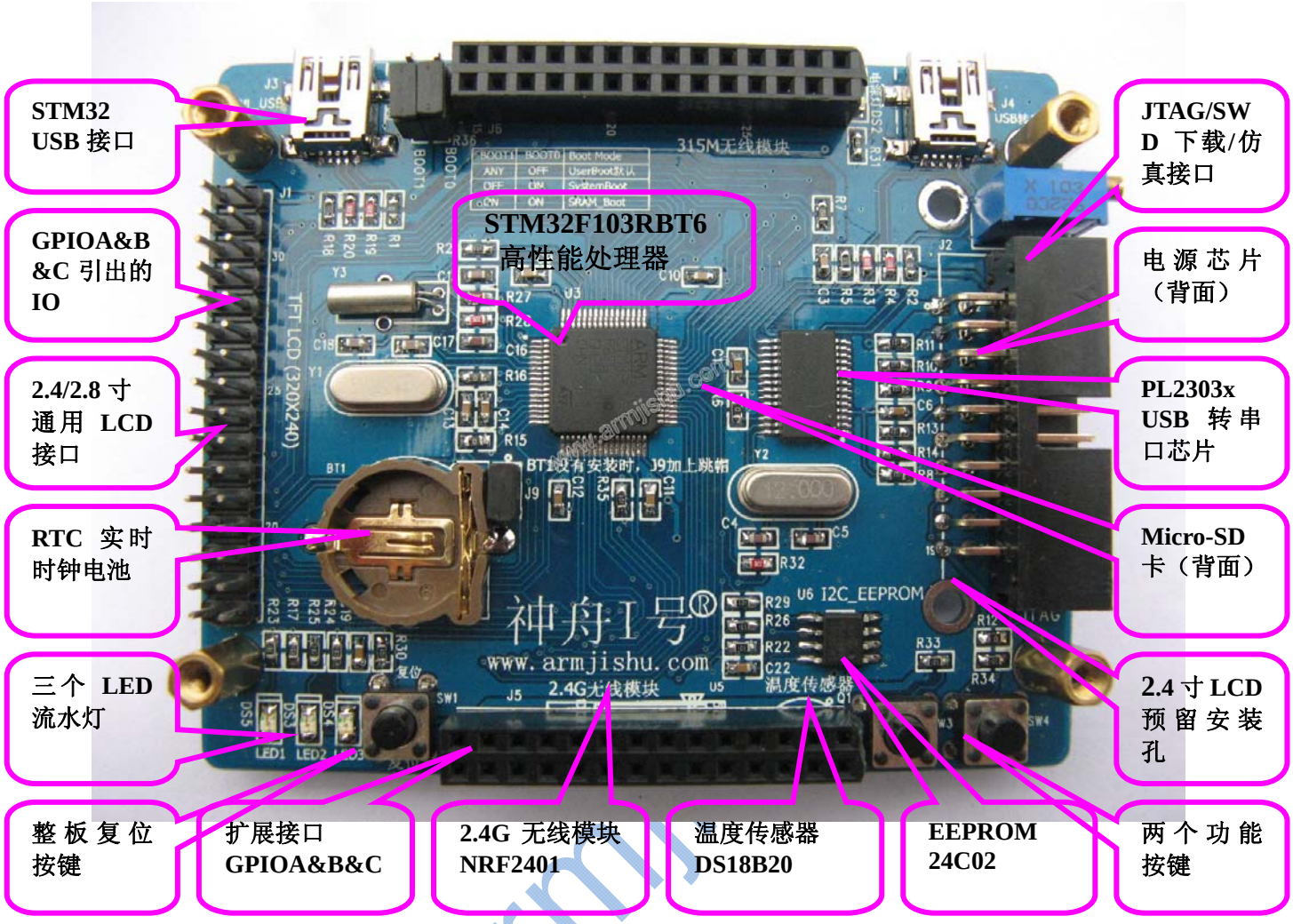
扩展接口 GPIOA&B&C

315M 无线模块

电源指示灯

STM32 USB 转串口

电位器 AD



图表 1 神舟 I 号开发板外观与功能

神舟I号开发板板载资源如下：

◆ STM32F103RBT6, ARM Cortex-M3 内核，ARM Cortex-M3 内核，主频 72Mhz，内部含有 128K 字节的 FLASH 和 20K 字节的 SRAM

◆ 1 个 USB 全速接口 （包括+5V 电源的输入）

◆ 1 个 USB 串口接口，采用 PL2303x 方案进行 USB 转串口（包括+5V 电源的输入）

◆ 1 个标准的 2.4 / 2.8 寸 TFT LCD 接口，支持触摸屏，分辨率 320X240, 26 万色（包括 1 个 SD 卡接口）

◆ 1 个 Micro SD 卡座

◆ 1 个 2.4G 无线通信模块接口

◆ 1 个 315M 无线通信模块接口

◆ 1 个温度传感器接口

◆ 1 个 IIC 接口的 EEPROM 芯片，24C02，容量 2K 比特

◆ 1 个复位按钮，控制整板硬件复位

◆ 2 个用户功能按钮

◆ 1 个电源指示灯（绿色）

◆ 3 个用户状态指示灯（LED1~LED3：绿色）

- 2 个启动模式选择配置接口
- 1 个 RTC 后备电池座，并带电池
- 1 个标准的 JTAG/SWD 仿真调试下载接口
- 1 路电位器（可调电阻）模拟输入，可以做模数转换实验
- 支持从 JLINK 取 5V 电源或 3.3V 电源
- 除晶振占用的 IO 管脚外，其余大部分 IO 口全部引出到扩展双排插针

从上面的板载资源可以看出，神舟I号开发板板载资源虽然简单，但其实也可以制造出许多非常丰富的实验例程来，这些基本包括STM32爱好者常用的硬件资源，尤其是从51入门到STM32的爱好者能得到非常多的帮助。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I号开发板的特点包括：

1） 外观小巧。整个板子尺寸为67mm * 82mm。

2） 性价比高。功能强大，但价格便宜

3） 设计灵活。板上除晶振外的所有GPIO口通过双排排母全部引出，可以通过外接一些模块，增加板子的功能；例如已经调试好的2.4G无线模块，315M无线模块，温度传感器模块等，都可以插入该排母直接与神舟I号进行调试，提供全部的代码和测试方法以及相关的详细文档资料。

4） 资源丰富。板载近十种外设及接口，小巧而精致。

5） 调试方便。与主流调试仿真工具JLINK V8完美结合，让您快速找到代码的BUG。

6） 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。

7） 教程齐全。接近200页的教程资料，共计近二十个实例，使用ST标准库，方便用户修改升级。

8） 提升空间。如果由神舟I号学习完成后，还可以考虑直接升级到神舟II号，神舟III号，神舟IV号，内部源代码都是可以相互借鉴，为您提供更多更丰富的资料，方便学习。

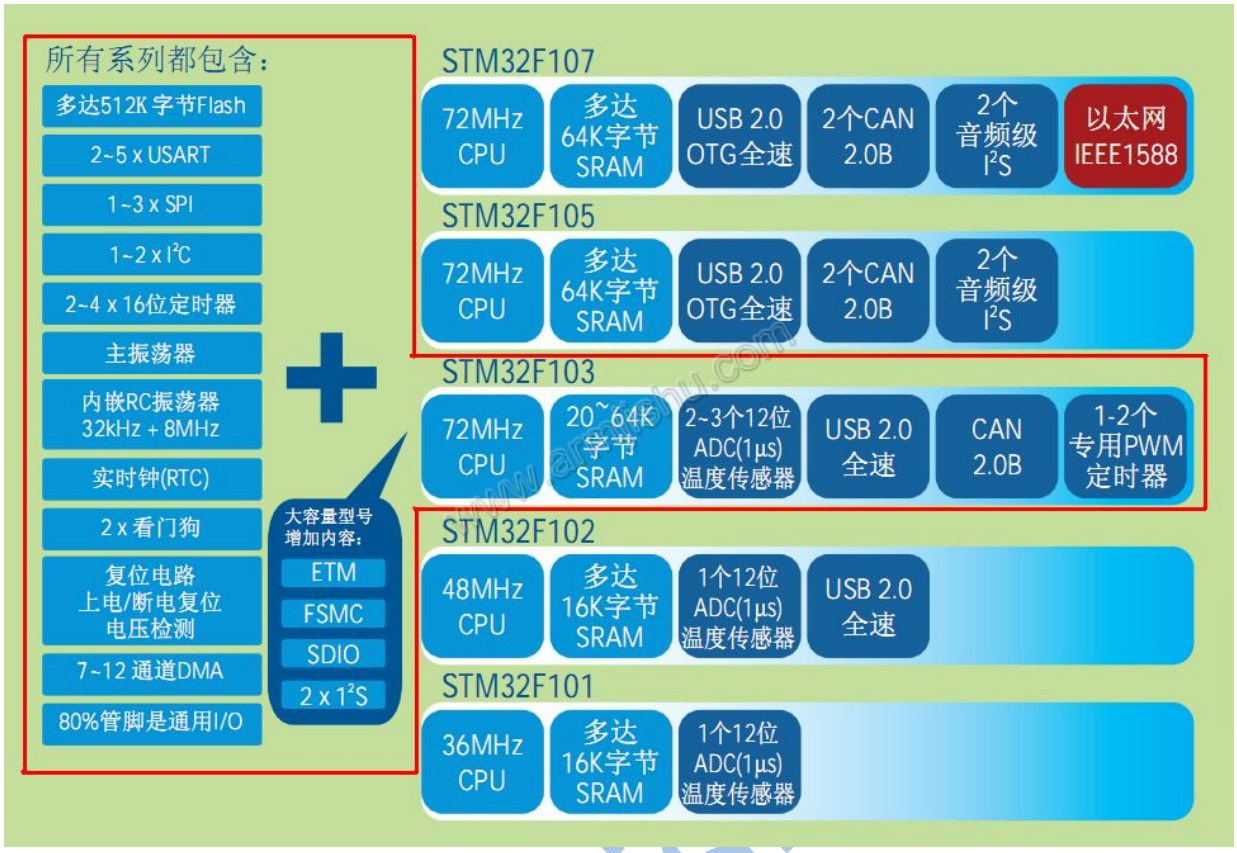
2.3 神舟I号开发板硬件电路分析

下面详细介绍神舟I号各功能模块的硬件实现与原理。

2.3.1 STM32F103RBT6处理器

开发板使用了STMF103系列中的高性能、高配置的Cortex-M3内核32位处理器STM32F103RBT6，72M主频，LQFP64封装，片内FLASH容量:128K,片内SRAM容量:20K。

STM32家族主要产品系列家谱如下图所示，如下图所示：



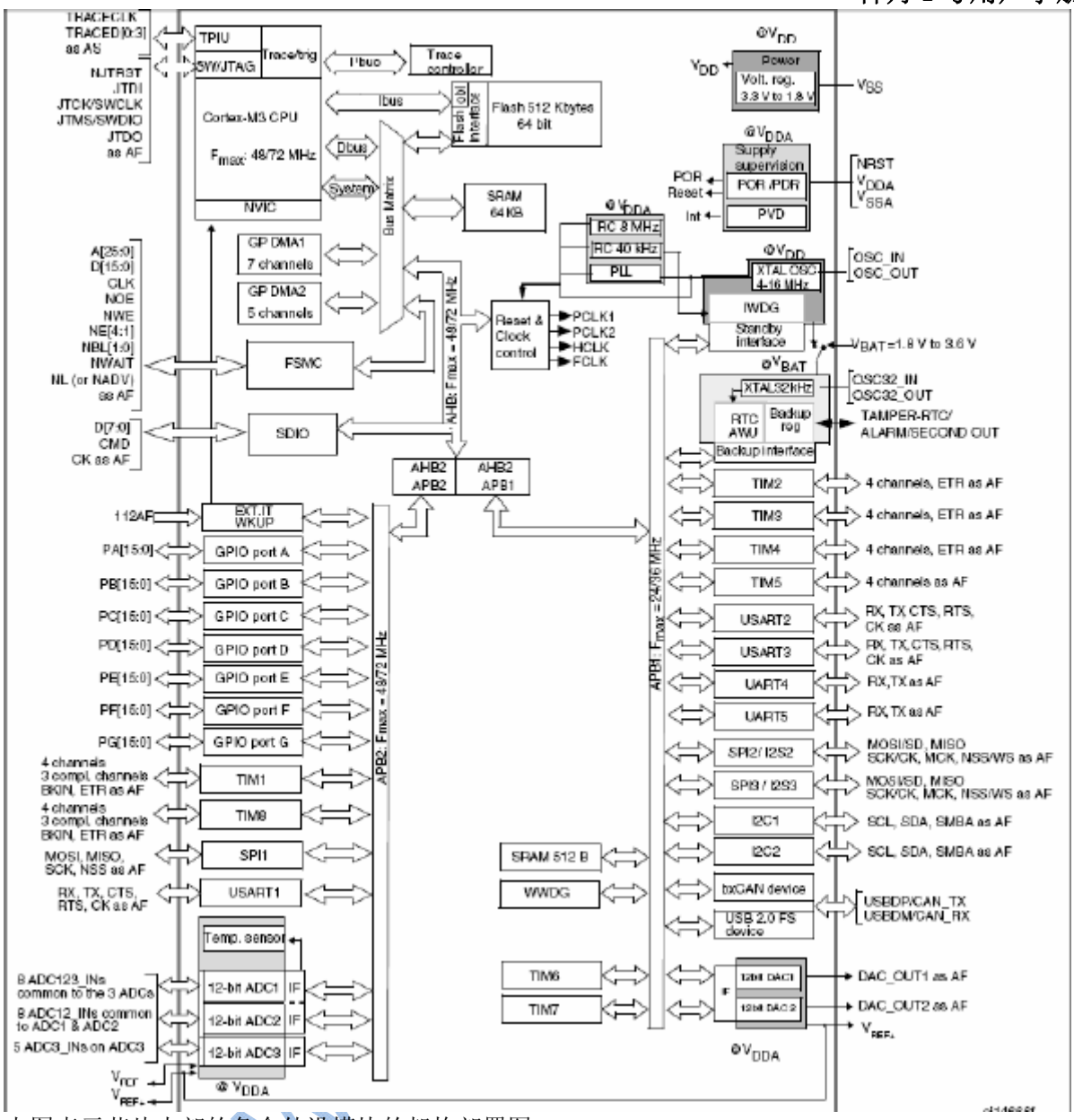
图表 2 STM32 家族主要产品系列家谱

STM32F103的产品列表出下图所示，神舟I号开发板选用的是外设资源和管脚资源较为丰富的64脚LQFP封装的STM32F103RBT6芯片，该芯片具有20K SRAM，128K FLASH，3个普通的16位定时器，1个16位的高级定时器，2个SPI，2个IIC，3个串口，1个USB，1个CAN，2个12位的ADC，51个通用IO口，性价比非常高。

STM32(ARM Cortex-M3) 32位微控制器产品列表(截至2009年8月)																					
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(IC/OC/PWM)	16位高级(IC/OC/PWM)	16位基本	SPI	I²C	USART[2] UART	USB全速	CAN 2.0	以太网	I²S	SDIO	ADC(通道)	DAC(通道)		
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	LQFP64/TFBGA64
	STM32F103R6	72	32K	10K		2(8/8/8)	1(4/4/6)		1	1	2	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103R8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		51	LQFP64/TFBGA64
	STM32F103RB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		51	LQFP64/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	LQFP64/WLCSP64
	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	LQFP64/WLCSP64
	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	LQFP64/WLCSP64
100脚	STM32F103V8	72	64K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		80	LQFP100/LFBGA100
	STM32F103VB	72	128K	20K		3(12/12/12)	1(4/4/6)		2	2	3	1	1				2/(16)		80	LQFP100/LFBGA100
	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	LQFP100/LFBGA100
	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	LQFP100/LFBGA100
	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	LQFP100/LFBGA100
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
64脚	STM32F105R8	72	64K	20K		4(16/16/16)	1(4/4/6)	2	3	2	3+2	OTG	2		2		2/(16)	1(2)	51	LQFP64
	STM32F105RB	72	128K	32K		4(16/16/16)	1(4/4/6)	2	3	2	3+2	OTG	2		2		2/(16)	1(2)	51	LQFP64
	STM32F107RB	72	128K	48K		4(16/16/16)	1(4/4/6)	2	2	1	3+2	OTG	2	●	1		2/(16)	1(2)	51	LQFP64
	STM32F105RC	72	256K	64K		4(16/16/16)	1(4/4/6)	2	3	2	3+2	OTG	2		2		2/(16)	1(2)	51	LQFP64
	STM32F107RC	72	256K	64K		4(16/16/16)	1(4/4/6)	2	2	1	3+2	OTG	2	●	1		2/(16)	1(2)	51	LQFP64
100脚	STM32F105V8	72	64K	20K		4(16/16/16)	1(4/4/6)	2	3	2	3+2	OTG	2		2		2/(16)	1(2)	80	LQFP100/BGA100[3]
	STM32F105VB	72	128K	32K		4(16/16/16)	1(4/4/6)	2	3	2	3+2	OTG	2		2		2/(16)	1(2)	80	LQFP100/BGA100[3]
	STM32F107VB	72	128K	48K		4(16/16/16)	1(4/4/6)	2	2	1	3+2	OTG	2	●	1		2/(16)	1(2)	80	LQFP100/BGA100[3]
	STM32F105VC	72	256K	64K		4(16/16/16)	1(4/4/6)	2	3	2	3+2	OTG	2		2		2/(16)	1(2)	80	LQFP100/BGA100[3]
	STM32F107VC	72	256K	64K		4(16/16/16)	1(4/4/6)	2	2	1	3+2	OTG	2	●	1		2/(16)	1(2)	80	LQFP100/BGA100[3]

通过上表可以看到，STM32F103RBT暂时不提供FSMC，以太网，I2S，SDIO，DAC通道等接口，这些接口其他型号有提供，具体的大家可以参考上面这张表。

www.armjishu

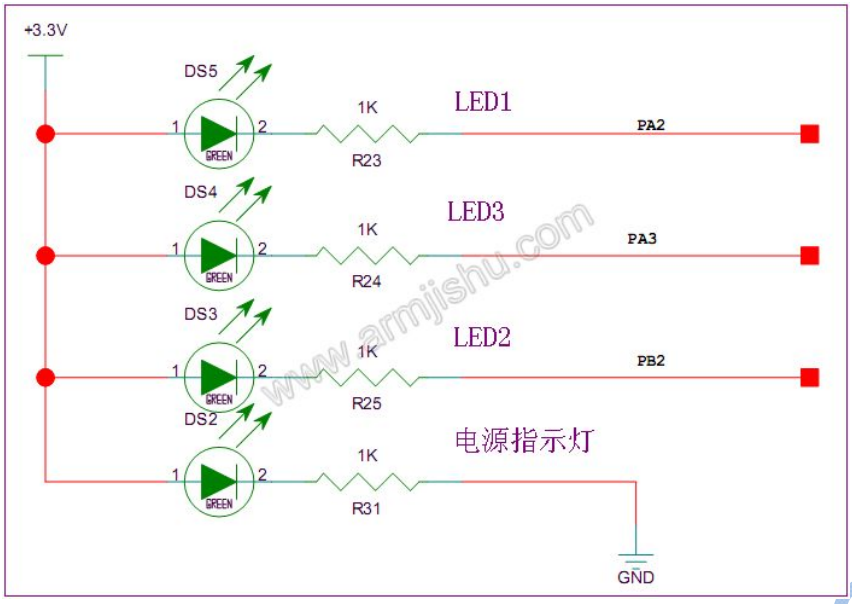


上图表示芯片内部的各个外设模块的架构部署图

2.3.2 LED指示灯

神舟I号提供了4路LED指示灯，其中一个LED作为电源的指示灯，如下图所示，对应板子上的【电源灯DS2】，上电就会亮。

另外三个LED绿灯（在板子的标号是LED1、LED2、LED3）可作为程序点灯功能使用，比如用于流水灯控制，或是其他功能的辅助点灯之用。分别与GPIOA2、GPIOB2、GPIOA3三个管脚连接，当管脚为低电平时，对应的LED指示灯亮。

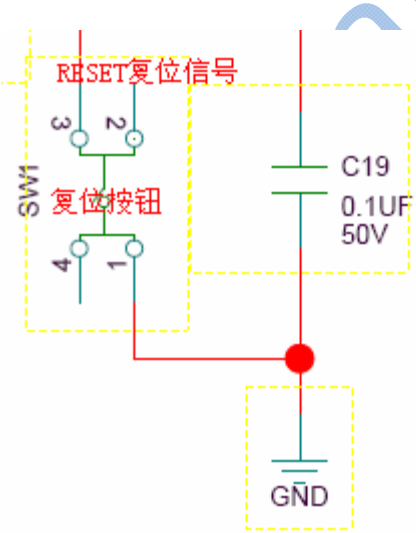


在神舟 I 号中，除了电源指示灯外，还有 3 个 LED 指示灯，由 GPIO 管脚控制 LED 灯的亮灭，当 GPIO 管脚输出低电平时，LED 指示灯亮。反之，当 GPIO 管脚输出高电平时，LED 指示灯灭。如下图所示：

2.3.3 普通按键与复位按键

板子提供一个复位按钮，STM32F10XXX 是低电平复位的，STM32 神舟系列开发板板载的复位按键，可用于复位整个开发板，还具有复位 LCD 液晶屏的功能，原理是因为 LCD 液晶屏模块的复位引脚和 STM32 的复位引脚是连接在一起的，当然也有其他各个芯片的复位引脚也是可以连在一起，这样设计能够实现一按按钮是 STM32 神舟开发板的整板硬件都会被同时进行复位。

STM32 神舟系列开发板本次设计采用的是简单的“RC＋按键”复位形式，复位电路的连接示意图如下图所示，该复位电路可以实现上电自动复位功能和手动按键复位功能，下图就是复位按键的图：

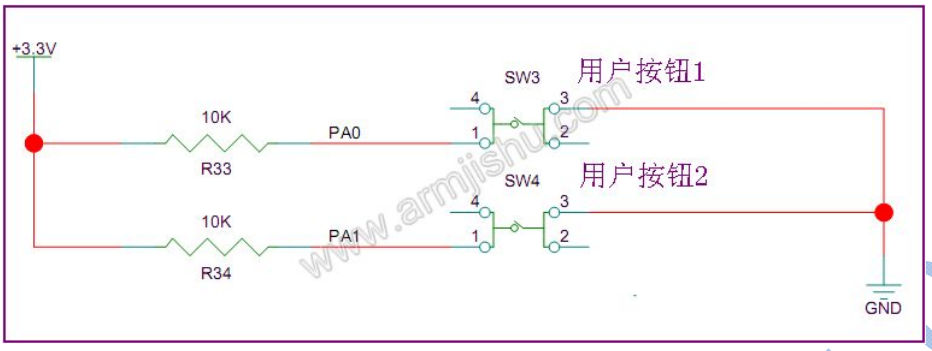


复位电路原理图

1）上电自动复位原理：上电前电容里的电荷放光，系统上电瞬间，3.3V 通过电阻向电容充电，在充电过程中（充电过程中需要一些时间，而此时芯片始终处于复位状态下），电容的电压缓慢上升直到 3.3V，只要电压没到 3.3V 这个值时芯片复位脚就近似于低电平，于是芯片检查到复位引脚低电平则导致系统复位，充电完毕后，电压接近 3.3V 时，芯片复位脚近似高电平，于是系统停止复位开始工作，复位完成。

2）手动按键复位原理：系统正常工作时，电容里已经充满了电荷，所以电容正极的电压为电源电压 3.3V，此时按下按键，则电容的正负极被短路，所以芯片复位脚为低电平，系统复位，同时由于电容正负极被短路电容里的电荷迅速释放干净，当按键释放时 3.3V 通过电阻向电容充电，原理与上电复位原理一致。

STM32 神舟 I 号开发板除了复位按键外，还板载 2 个功能按键，可由用户自定义的功能按键，这两个按键分别与 PA0，PA1 这两个 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，这两个按键的设置方便了人机交互：



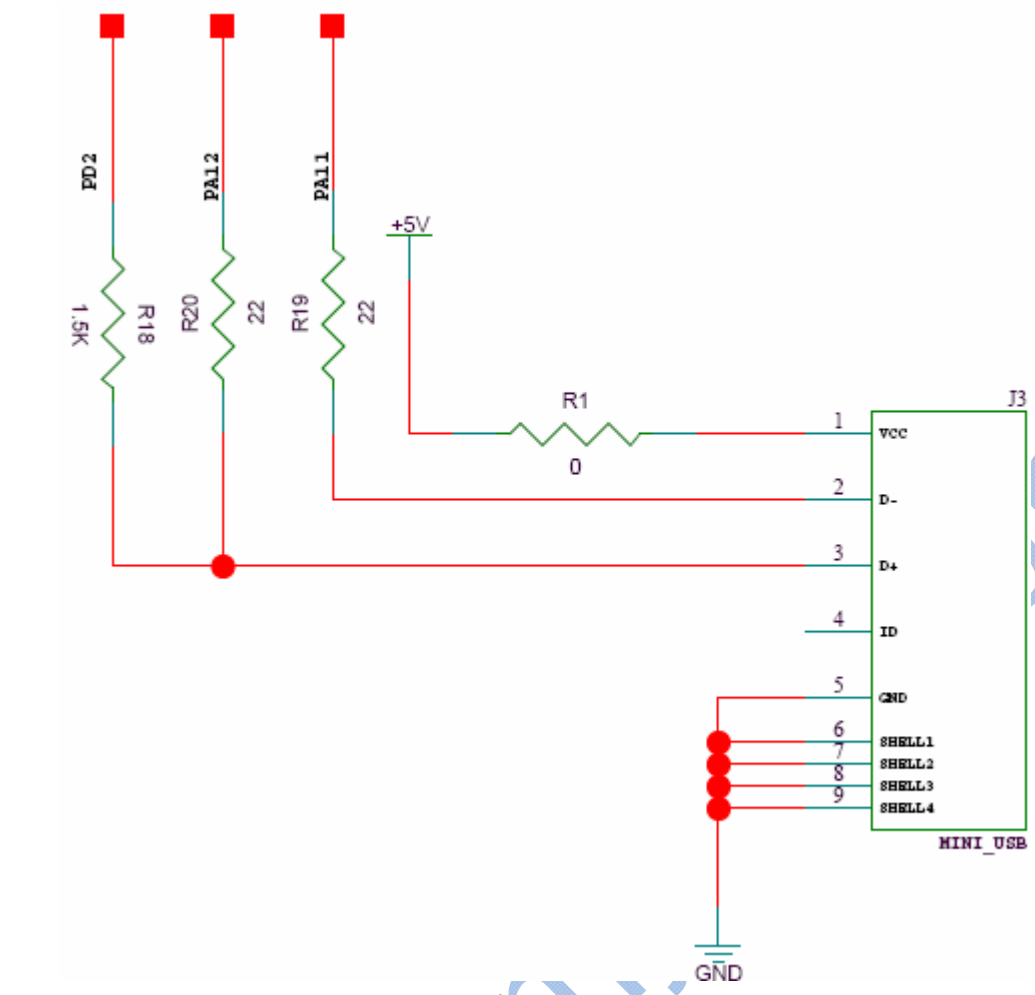
2.3.4 USB接口

STM32F103 系列带有一个 USB2.0 全速从设备接口，，可配置 1 到 8 个 USB 端点，它至少有 4 根信号线，包括 D+和 D-信号线，以及 Vcc 和 GND；当 USB 插入时（Vcc=5V）就会有下面几种情况：一种是如为充电器，则充电，这个没什么好说的；第二种是如为 USB 设备，则再分为低速和高速这两种情况，低速和高速是通过检测设备的 D+或 D-信号线是否有达到 3.5V 来区分的，因为插入前 D+和 D-都是低电平；

所谓“低速 USB 设备”是指当 USB 低速设备为（如鼠标，键盘）时，D-这条信号线为 3.5V 的电压，那么 STM32 芯片就会默认为是低速设备；反之，对于全速设备（如 U 盘，打印机，扫描仪），D+信号线就会被上拉到 3.5V。

所以，当 USB 设备端插入主机后，主机通过检测 D+和 D-来确认是高速设备还是低速设备。

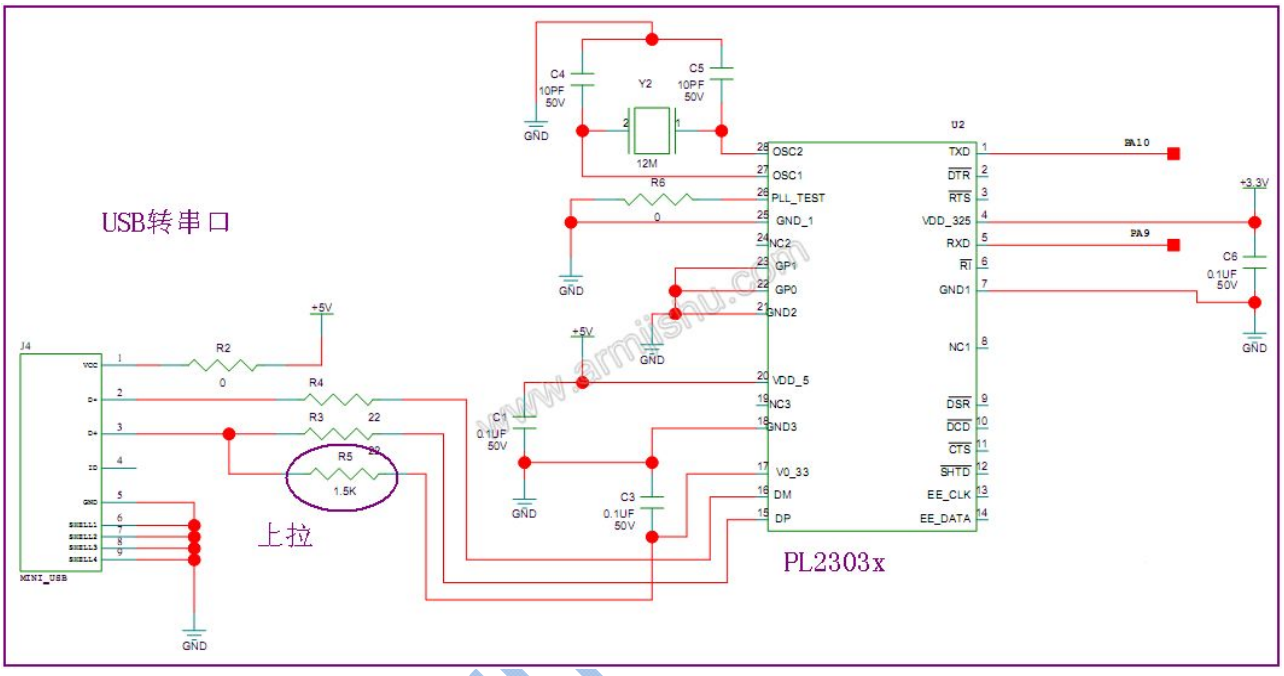
www.armjishu.com



2.3.5 USB转串口接口

此处将STM32F103RBT6的USART1通过PL2303x芯片进行电平转换，接口方式为MINI-USB，PL2303x是Prolific推出的USB转串口桥芯片，USB侧支持USB1.1规范，串口侧支持标准的RS232串行接口和握手协议：

在下图中，我们将USB口的D+信号通过PL2303的输出电压口拉高至3.3V，表示STM32神舟I号默认为高速USB从设备，其中PL2303需要一颗12MHZ的晶振提供时钟：

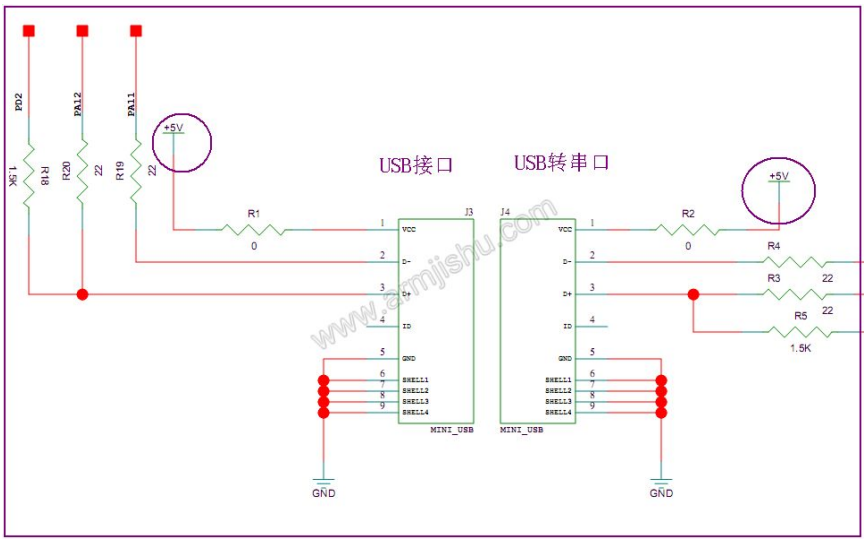


提供 USB 转串行口主要是因为现在绝大部分笔记本都没有串口了，大部分都只提供 USB 口，所以为了方便，我们设计了这个 USB 硬件接口的串口接口，只要你将电脑的 USB 线插入该口，那么电脑将多生成一个串口设备。

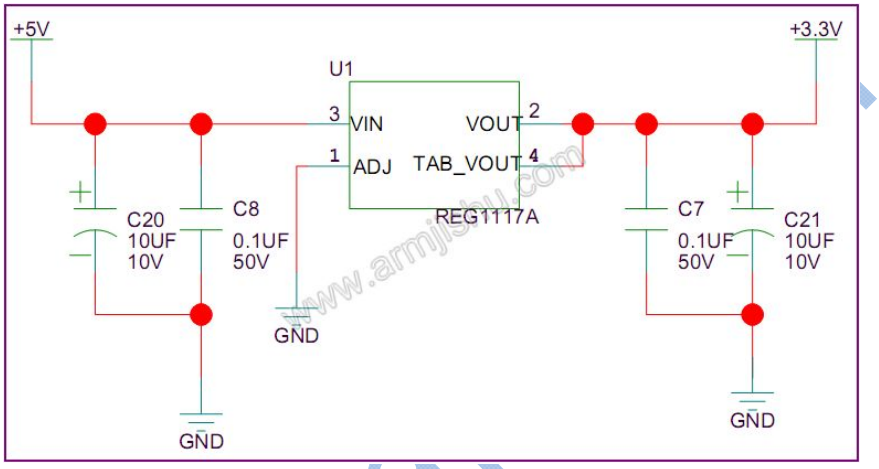
2.3.6 供电电源

由于 STM32F103RBT6 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 I 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。

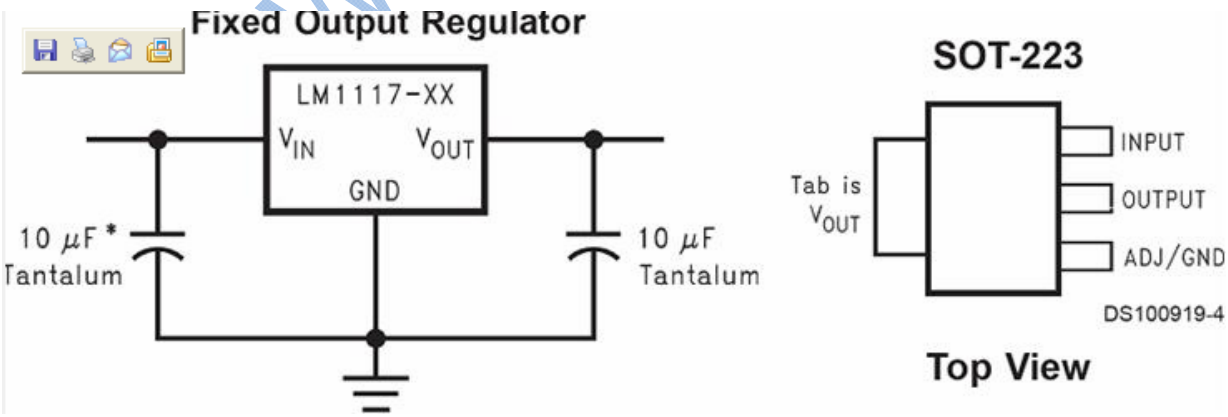
STM32 神舟 I 号需要外部提供 5V 的外接电源，我们借助 USB 接口向产品提供 5V 的电源输入（USB 可提供最大 500mA 的电流），板上的有两个 MINI-USB 口，其中一个是 USB 转串口接口，两个口都可以接收外部 5V 电源，然后经过 AMS-1117 3.3V 电源转换芯片将输入电压转换成 3.3V 供开发板使用：



5V 电压经过我们的 AMS1117-3.3V 电源处理芯片后将输出 3.3V 稳定电压供给 STM32 处理器和其他器件使用（STM32 核心处理器是需要 3.3V 的供电电压）：



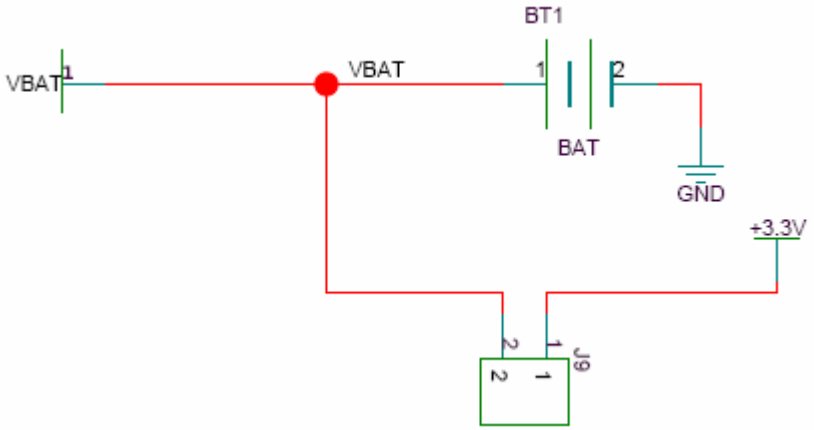
3.3V电压芯片选用3.3V的AMS1117电源转换芯片，电源电路图和其IC封装的管脚定义如下图所示，当然也有很多PIN-TO-PIN可替代型号



2.3.7 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，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J9的跳线帽断开。VBAT管脚由电池供电，如没有安装电池，将J9使用跳线帽短接，VBAT管脚由+3.3V系统电源供电。

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

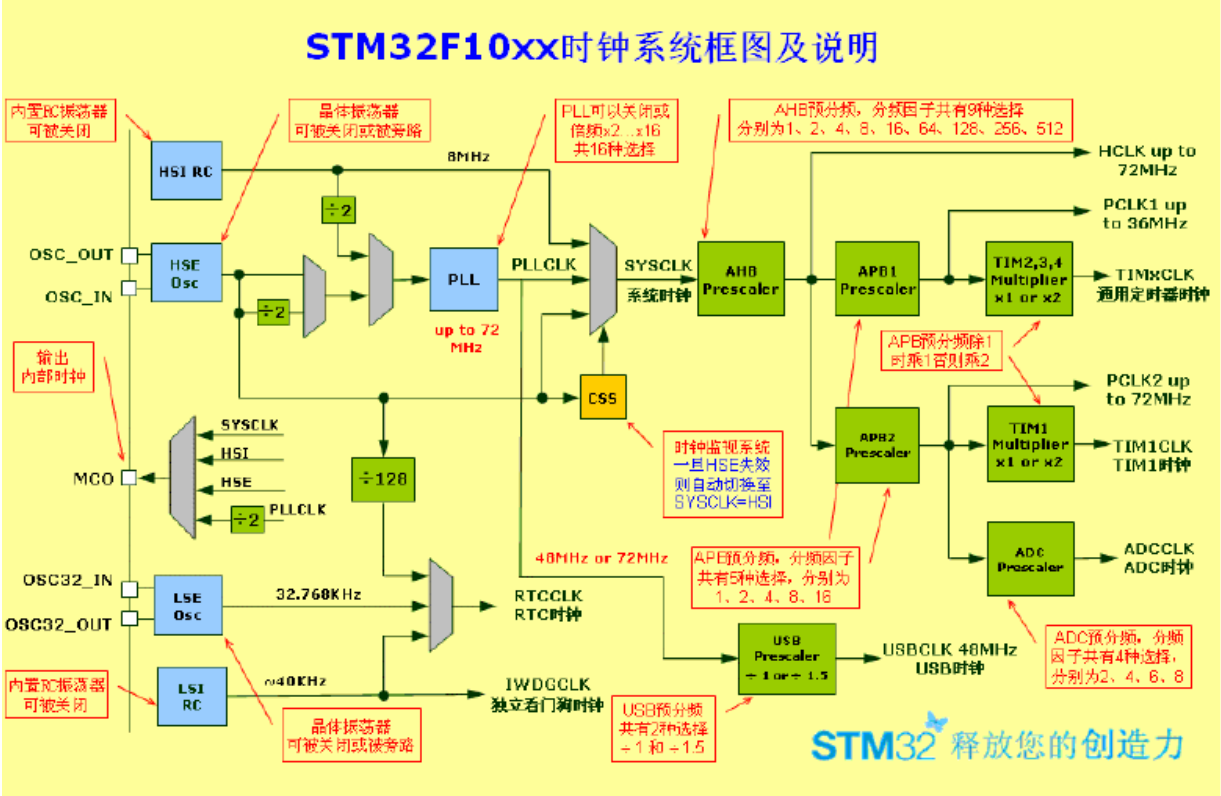
STM32内部RTC功能非常实用，经常用于实时时钟计数，计时，以保证系统时钟的同步。它的供电和时钟是独立于内核的，可以说是STM32内部独立的外设模块，RTC内部寄存器不受系统复位掉电的影响。

对于RTC来说，我们可以选择以下三种RTC的时钟源：

----HSE时钟除以128

----LSE振荡器时钟

----LSI振荡器时钟



现在参看上图，简单介绍一下以下几个时钟：

1）HSI：此时钟信号由内部8MHz的RC振荡器产生，可直接作为系统时钟或在2分频后作为PLL输入

2）HSE：一般被称为外部晶体/陶瓷谐振器（一般选用8M手册中提到范围为4-16Mz）,HSE是用户外部时钟（最高可达24Mhz从SOC_IN引脚输入，并保证OSC_OUT悬空）

在神舟I号STM32开发板上，我们采用外部电池供电和32.768K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J9跳线帽来去掉RTC时钟功能，而不影响系统正常运行

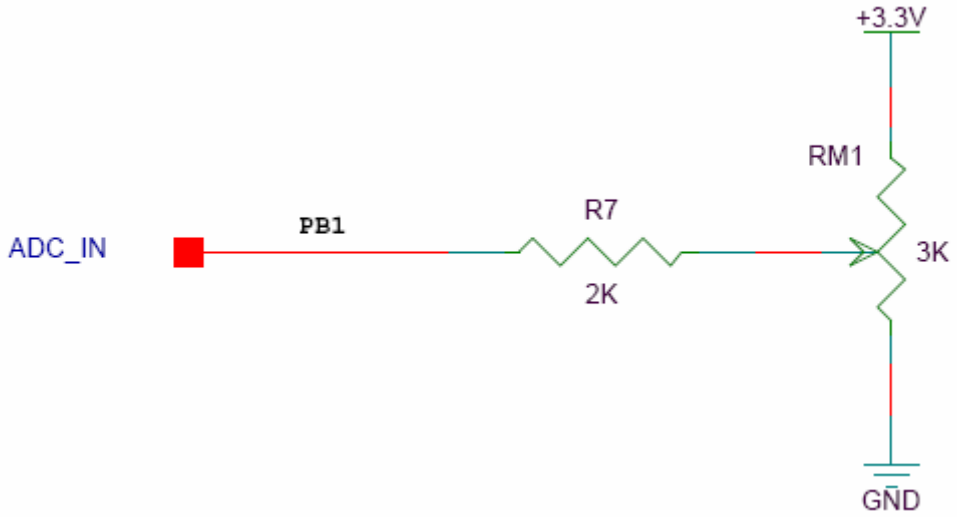
3）LSE：是一个32.768kHz的低速外部晶体或陶瓷谐振器。它为实时时钟或者其他定时功能提供一个低功耗且精确的时钟源。

4）LSI RC：担当一个低功耗时钟源的角色，它可以在停机和待机模式下保持运行，为独立看门狗和自动唤醒单元提供时钟。LSI时钟频率大约40kHz（在30kHz和60kHz之间）。

2.3.8 电位器（ADC介绍）

STM32F103RBT 拥有 2 个 16 通道的 ADC（是 12 位的逐次逼近型模拟数字转换器），这些 ADC 可以独立使用，有些可以使用双重模式（提高采样率），其最大的转换速率为 1Mhz，也就是转换时间为 1us。

神舟I号STM32开发板，提供了一路可调电阻设备RM1，该可调电阻通过我们旋转电位器的按钮，CPU的AD接口进行采样，就可以将模拟信号转换成数字信号，下面是原理图：

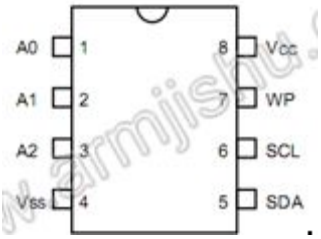


STM32 神舟 I 号上用的电位器是最最普通的通用电位器，目的是让大家通过这个电位器的了解来获得相关产品通用功能的应用知识，方便以后自行设计各种产品。

2.3.9 EEPROM存储器（IIC接口控制）

我们使用 IIC 总线来访问和控制 EEPROM 存储器，IIC 主要是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间，IC 与 IC 之间进行双向传送，高速的 IIC 速率一般可达 400kbps 以上。

EEPROM是一种电擦除可编程只读存储器，其主要特点能让CPU进行在线修改和擦除，并能在断电的情况下保持修改的结果，用于掉电数据保存，其功能相当于磁盘，因为STM32内部没有EEPROM，我们这里外扩了一片24C02的EEPROM，可用于设备的一些配置数据，或者一些不需要经常修改的数据保存。



STM32神舟I号所自带的24C02容量为2Kbit，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24C02~24C512全系列的EEPROM芯片的。其原理图如下：

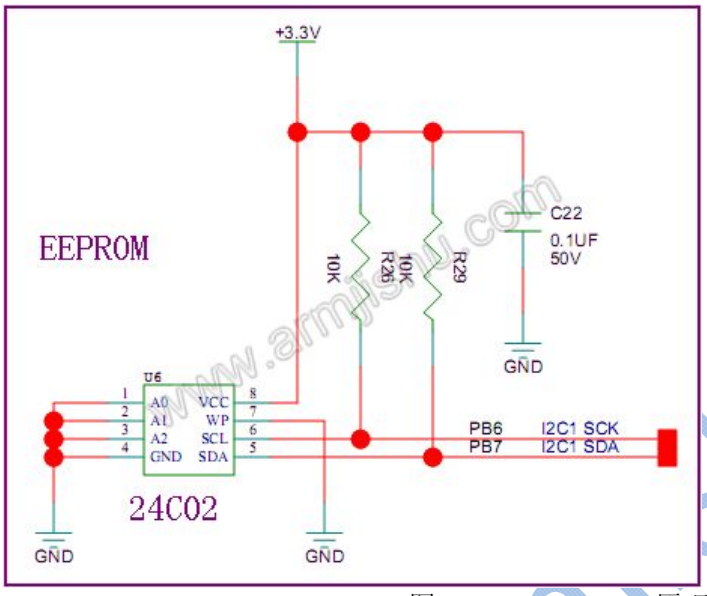
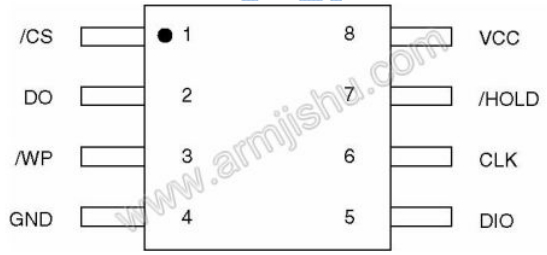


图1.2.2.1　EEPROM原理图

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，写程序的时候要注意下。

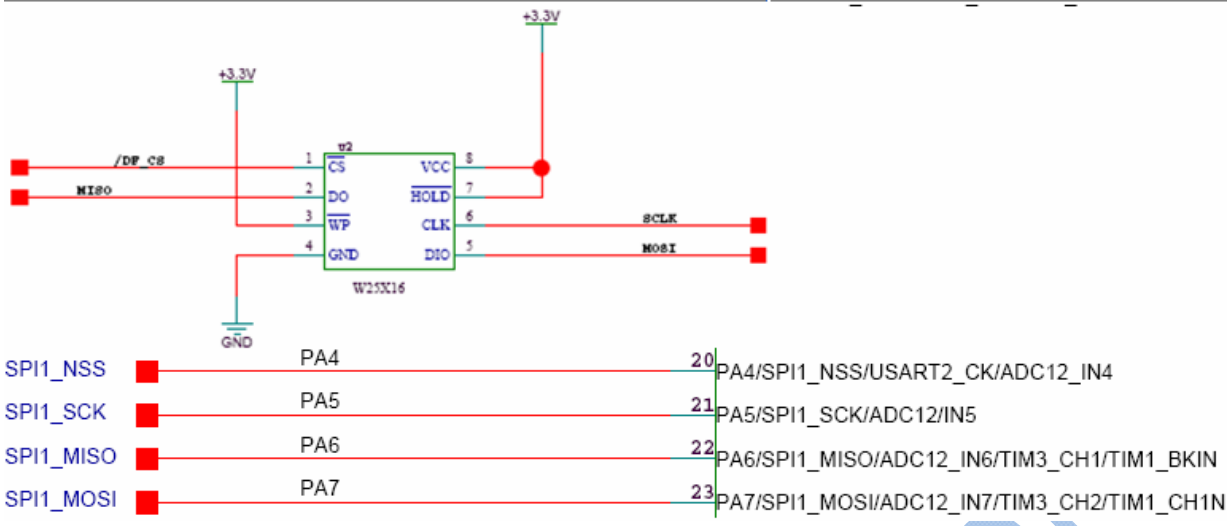
2.3.10 W25X16 16M SPI FLASH（LCD屏上安装）

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，STM32神舟I号在背屏上板载了一片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，比如字库等，以下是该芯片的信号定义：



STM32 神舟 I 号是用 SPI1 接口来控制这颗 FLASH 的，SPI1 接口被多个控制对象复用，复用信号是 MISO，MOSI，　SCLK 三个，CS 进行片选，SPI 接口一般使用 4 条线：

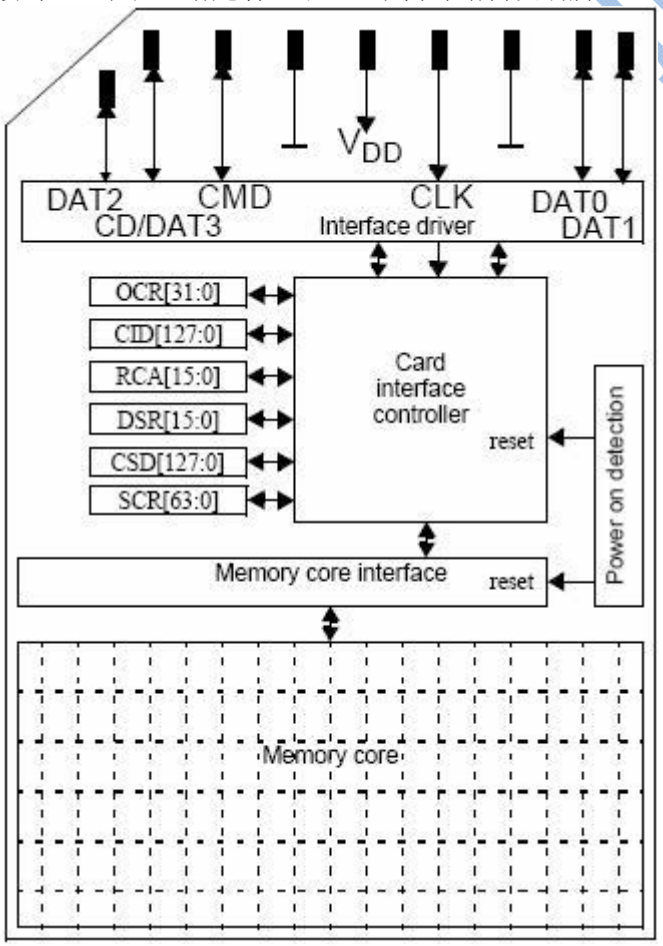
1）　MISO：主设备数据输入，从设备数据输入
2）　MOSI：主设备数据输出，从设备数据输入
3）　SCLK：时钟信号，由主设备产生
4）　CS：从设备选信号，由主设备控制

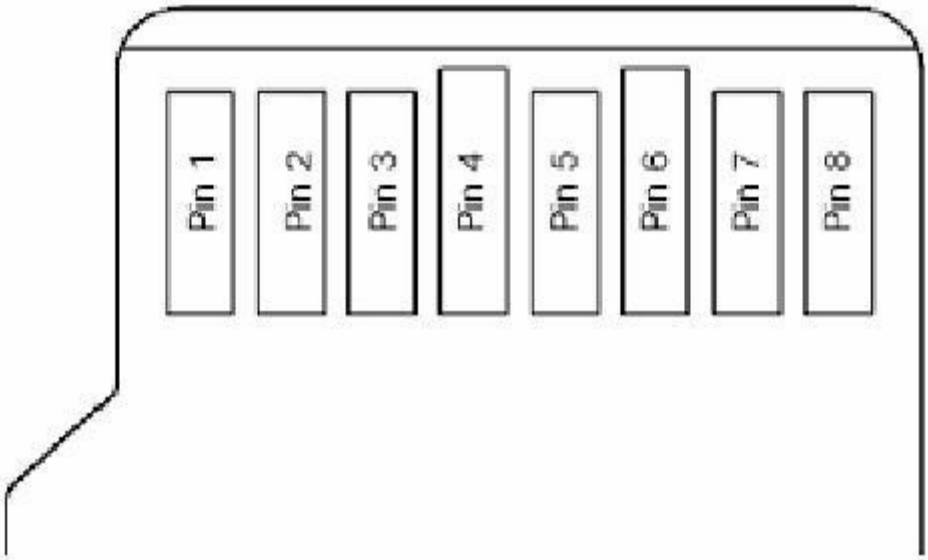


W25X16与STM32的连接，板上的W25X16是直接连在STM32F103RBT6上，连接关系如上图。

2.3.11 MicroSD卡接口

SD 卡（又称 Secure Digital Memory Card）是一种为满足安全性，容量，性能和使用环境等各方面的需求而设计的一种新型存储器件，SD 卡允许在两种模式下工作，即 SD 模式和 SPI 模式。因为 STM32F103RBT 不带 SD 模式，所以只能工作在 SPI 模式下，那么 STM32 处理器如何在 SPI 模式下读写 SD 卡呢？请先看一下 SD 卡内部结构及引脚：





关于SD卡SD模式和SPI模式两种模式各个管脚的分配表：

Pin#	SD Mode			SPI Mode		
	Name	Type[1]	Description	Name	Type	Description
1	DAT2	I/O/PP	Data Line [Bit 2]	RSV		Reserved
2	CD/DAT3[2]	I/O/PP[3]	Card Detect / Data Line [Bit 3]	CS	I[3]	Chip Select (neg true)
3	CMD	PP	Command/Response	DI	I	Data In
4	V_{DD}	S	Supply voltage	V_{DD}	S	Supply voltage
5	CLK	I	Clock	SCLK	I	Clock
6	V_{SS}	S	Supply voltage ground	V_{SS}	S	Supply voltage ground
7	DAT0	I/O/PP	Data Line [Bit 0]	DO	O/PP	Data Out
8	DAT1	I/O/PP	Data Line [Bit 1]	RSV		Reserved

SD卡和MicroSD卡仅仅是封装上不同，MicroSD卡更小,但是它们协议是相同的，一般我们用单片机操作SD卡时，都不需要对文件系统（例如FAT分区表信息）做处理，原因如下：

1） 操作FAT分区表要增加程序代码量，增加SRAM的消耗，对于便携应用来说代码大小和占用SRAM的多少至关重要；
2） 即使我们对FAT分区表不做任何了解，实际上我们一样可以向SD上写入数据，这就表明使用FAT对我们做数据存储来说意义不是很大；
3） 耗费大量经历和时间去了解FAT分区表对于我们做嵌入式软件开发的人来说有些得不偿失。
4） SD卡支持SD模式和SPI模式两种；SPI模式不需要知道FAT就可以做SD数据操作，这时候SD卡对于我们来说实际上就是个大的，快速的，方便的，容量可变的外部存储器。基于以上原因，一般情况下对SD卡的操作只需要了解SPI通讯就可以了，现在大部分单片机都有SPI接口。

神舟I号开发板上有一个MicroSD卡接口，该接口可以通过程序设计成为Micro SD卡读卡器。其原理图如下：

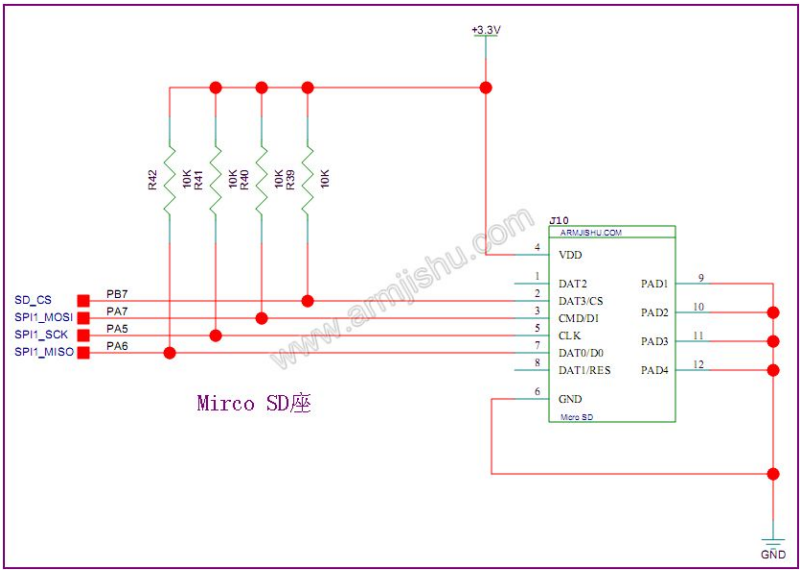


图1.2.9.1　Micro SD卡接口原理图

一般SD卡可通过SPI接口或者SDIO接口来通信，虽然SDIO模式通信速率更高，但在这里因为我们的STM32F103RBT没有SDIO接口，只有SPI接口，所以我们使用的SPI模式进行通信：

1）SD_CS：连接到单片机的片选SD管脚，只有单片机设置SD_CS/为低电平时才可以操作SD卡。

2）SPI1_MOSI：连接单片机SPI总线的MOSI管脚（SPI数据输入），单片机从这个管脚读取SD卡内的数据。

3）SPI1_MISO：连接单片机SPI总线的MISO管脚（SPI数据输出）、单片机通过这个管脚向SD卡内写入数据。

4）SPI1_SCK：连接单片机SPI总线的SCK(SPI时钟）

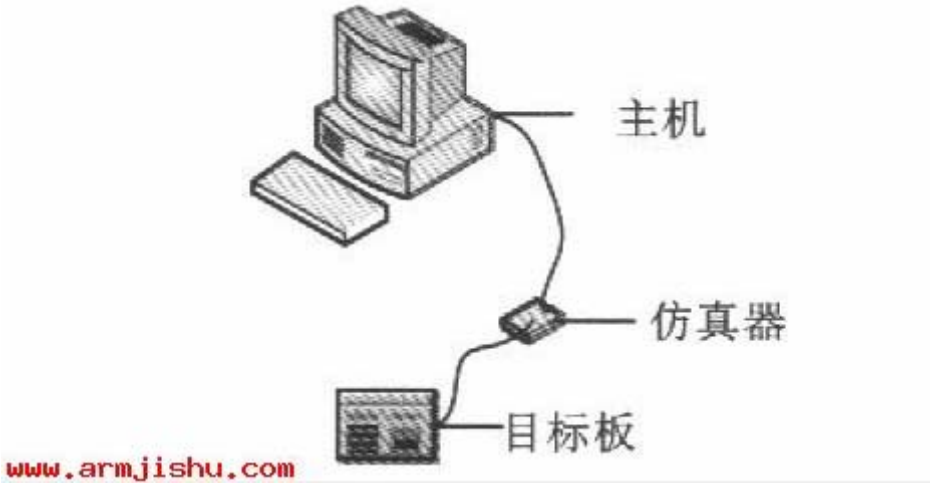
SD管脚实际上在SD卡内部连接到了GND，当SD插座上没插入SD卡时，单片机从这个管脚能读到高电平（前提是使用单片机内部上拉输入，或者外部增加一个上拉电阻），一旦插入SD卡，这个管脚就变成低电平，这个功能用来检测是否插入SD卡。

MicroSD卡的连接和SD卡大同小异，只是MicroSD卡比SD卡少一个GND管脚，所以不能使用上面做的这种插入卡的检测，实际上现在很多SD卡/MicroSD卡插座都有插入检测管脚，当然，一分钱一分货，价格上当然也要贵一些。

STM32神舟I号就自带了迷你SD卡接口，迷你SD卡作为常见的存储设备，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，不但可以用来提供数据，也可以用来存储数据，这样使得我们的板子可以完成更多的功能，**同时，我们利用ST官方网站提供的资料，已经成功的通过USB接口将SD卡接口变成为SD读卡器，感兴趣的爱好者可通过我们的产品，获取相关的源码。**

2.3.12 JTAG调试接口

标准的20针JTAG，直接可以和JLINK V8仿真器连接的，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，如下图所示，通过USB与仿真器相连接，再将仿真器的JTAG口与目标板STM32神舟I号的JTAG调试接口连接，就可以开始进行调试了：



那么什么是处理器的调试接口呢？STM3210xxx使用Cortex-M3内核，该内核含硬件JTAG调试模块，支持复杂的调试操作。硬件调试模块允许内核在取指(指令断点)或访问数据（数据断电）时停止。内核停止时，内核的内部状态和系统的外部状态都是可以查询的。完成查询后，内核和外设可以被复原，程序将继续执行。

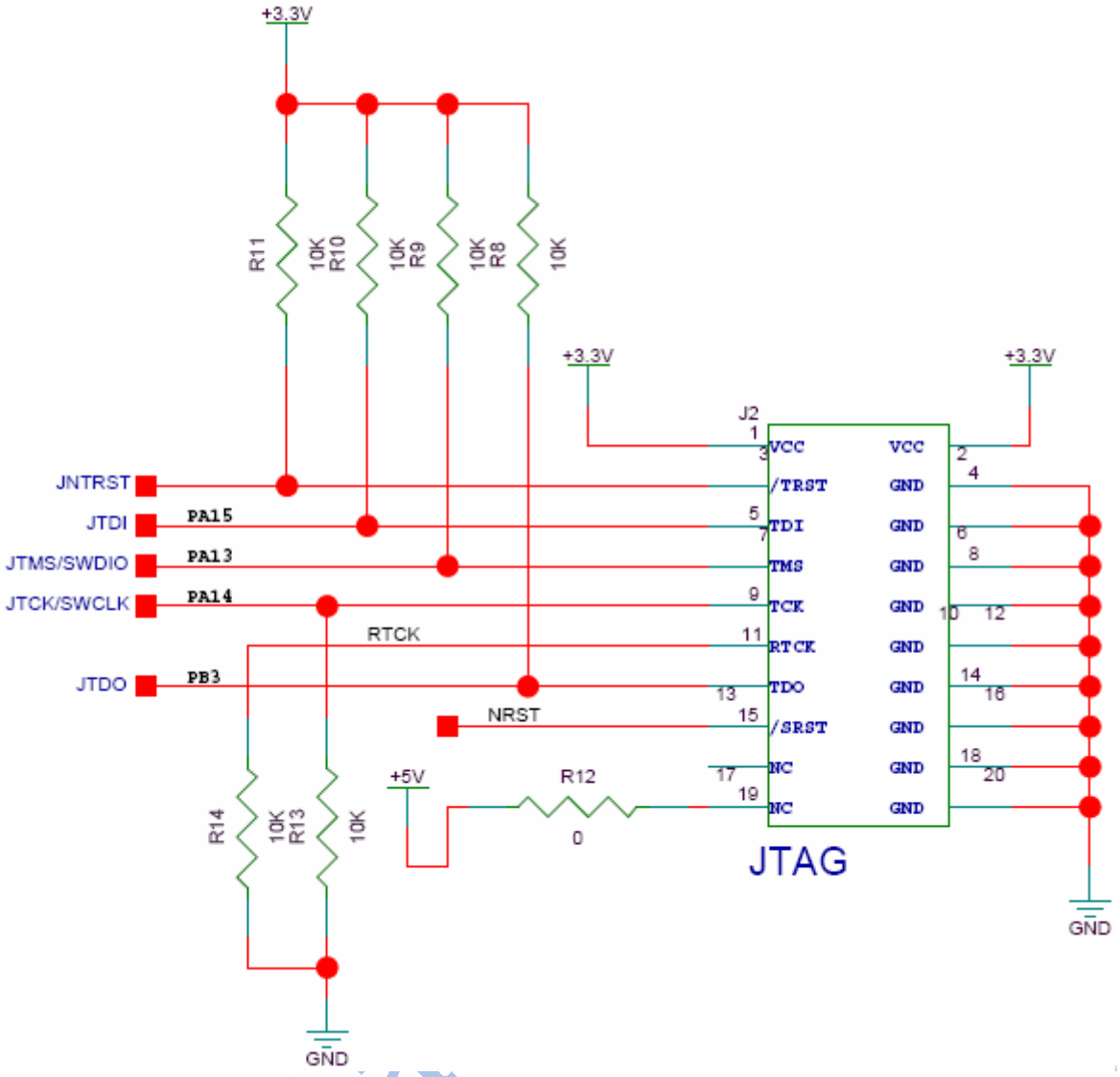
当STM32F10X微控制器连接到调试器并开始调试时，调试器将使用内核的硬件调试模块进行调试操作。

支持两种调试接口：

- SW串行接口
- JTAG调试接口

下面是我们STM32神舟I号JTAG口调试接口原理图，我们可以通过该接口下载程序和硬件调试：

www.armjishu.com



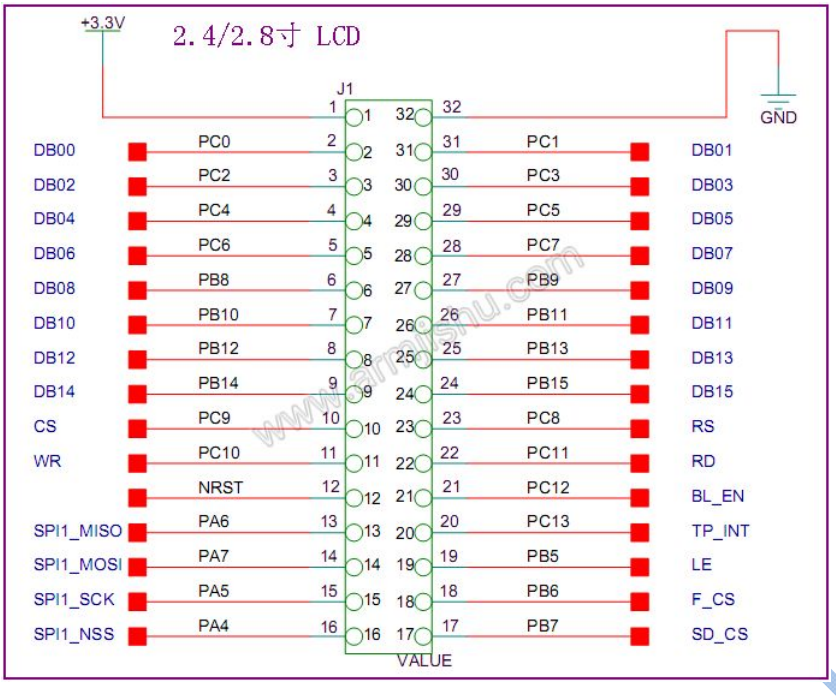
下图是调试接口的信号说明分布图：

SWJ-DP端口引脚名称	JTAG 调试接口		SW 调试接口		引脚分配
	类型	描述	类型	调试功能	
JTMS/SWDIO	输入	JTAG模式选择	输入/输出	串行数据输入/输出	PA13
JTCK/SWCLK	输入	JTAG时钟	输入	串行时钟	PA14
JTDI	输入	JTAG数据输入	——	——	PA15
JTDO/TRACESWO	输出	JTAG数据输出	——	跟踪时为TRACESWO信号	PB3
JNTRST	输入	JTAG模块复位	——	——	PB4

由上面这些接口可以知道，有了这个JTAG调试接口，就能够让我们更方便快捷的观察到处理器芯片的状态情况。

2.3.13 液晶显示模块

神舟I号开发板载有目前比较通用2.4/2.8寸液晶触摸显示模块接口。其设计原理图如下：



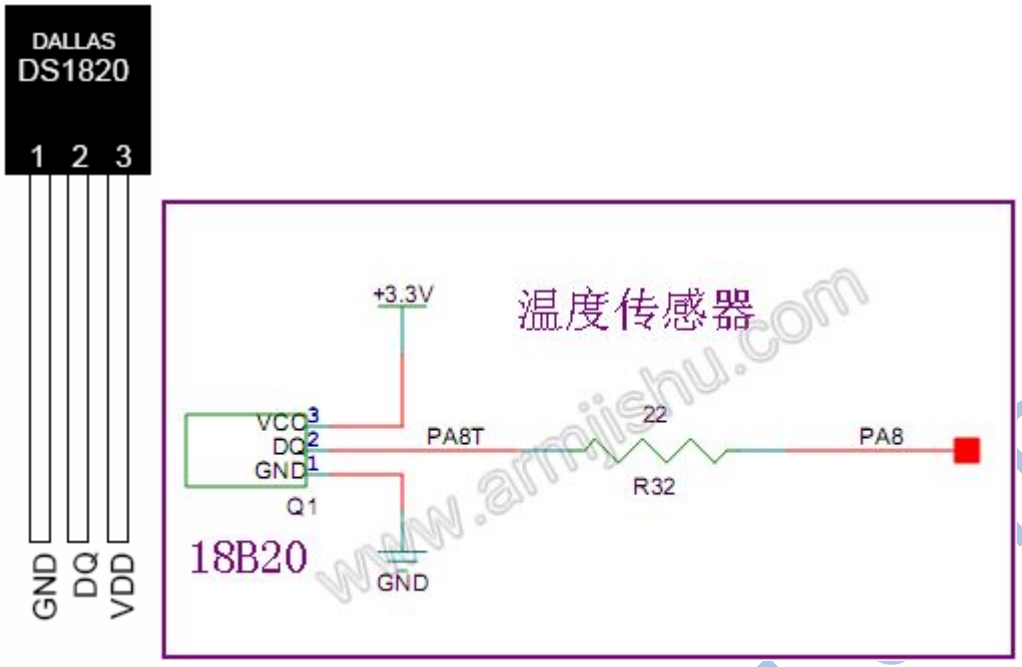
图表 3 液晶显示模块原理图

神舟I号通过FSMC总线对屏进行访问操作，此外，2.4/2.8寸LCD屏模块上还集成了SPI FLASH和SPI接口的SD卡座（神舟I号板载了一个Micro SD卡座），在这里由于资源有限，神舟I号只支持2.4/2.8寸屏，而对于LCD屏上的SD卡座还是神舟I号板载的Micro SD卡座，源码都是支持的，但注意要求不要同时使用两个SD卡。

2.3.14 温度传感器

此温度传感器为 DALLAS 公司推出的单总线数字温度计，型号为 DS18B20。利用此温度传感器，我们可以检测周围环境温度，或是运用于机箱以及温度受控的环境中，通过检测环境温度，设置告警信息，启动散热装置，达到自动降温的效果。通过源码设计，可以将检测到的温度以数字形式从串口、或是 LCD 屏上显示。

DS18B20的温度检测与数字数据输出全集成于一个芯片之上，从而抗扰力更强；其一个工作周期可分成两个部分，即温度检测和数据处理。我们使用PA8管脚连接温度传感器的数据输出端。此温度传感器的温度检测有效范围为-55 ~ 125度。

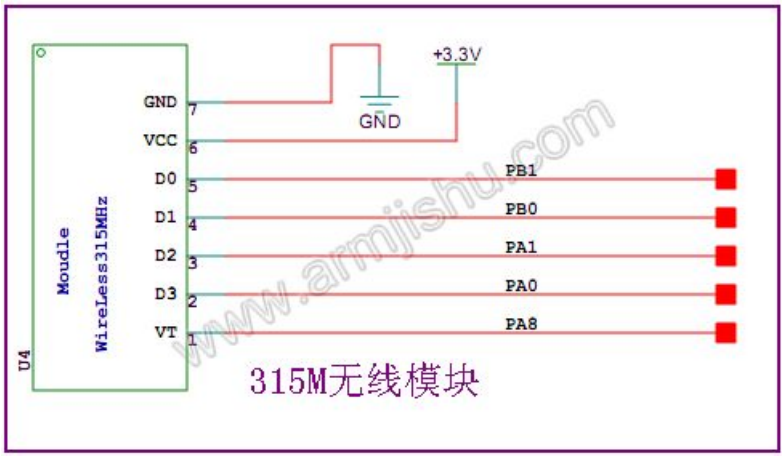


2.3.15 315M无线模块

315M无线模块，可通过遥控控制进行远程控制，用于轿车，摩托车等相关私人财产的安防控制。接触它，使用它，便可清楚安防的无线遥控原理原来就是这么简单，相关产品和模块如下所示。

神舟I号开发板预留了315M无线模块接口，315M无线模块，可以接收遥控器的信号，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，而处理器通过GPIO管脚直接将315M无线模块解码后的数据D0~3读取，并分析。

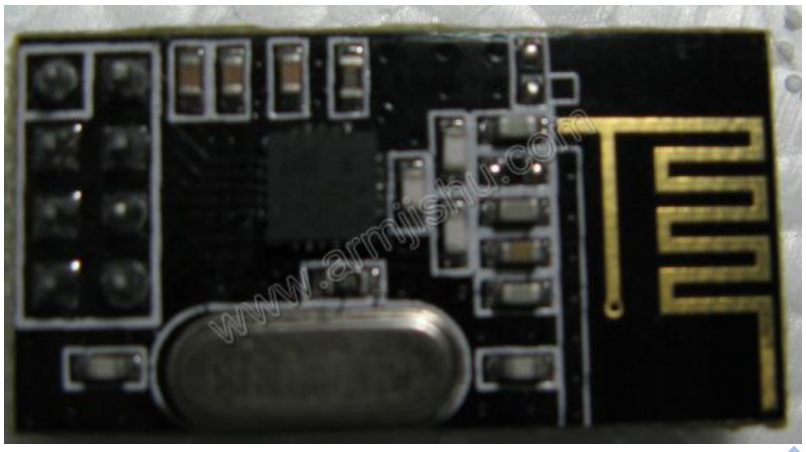
当无线模块的VT脚（PA8）有效（低电平）时，表示无线模块接收到遥控的按键信号；当VT管脚无效（高电平）时，表示无线模块没有接收到遥控的按键信号，此时不管D0~3的数据为多少，处理器不会对其进行解析。



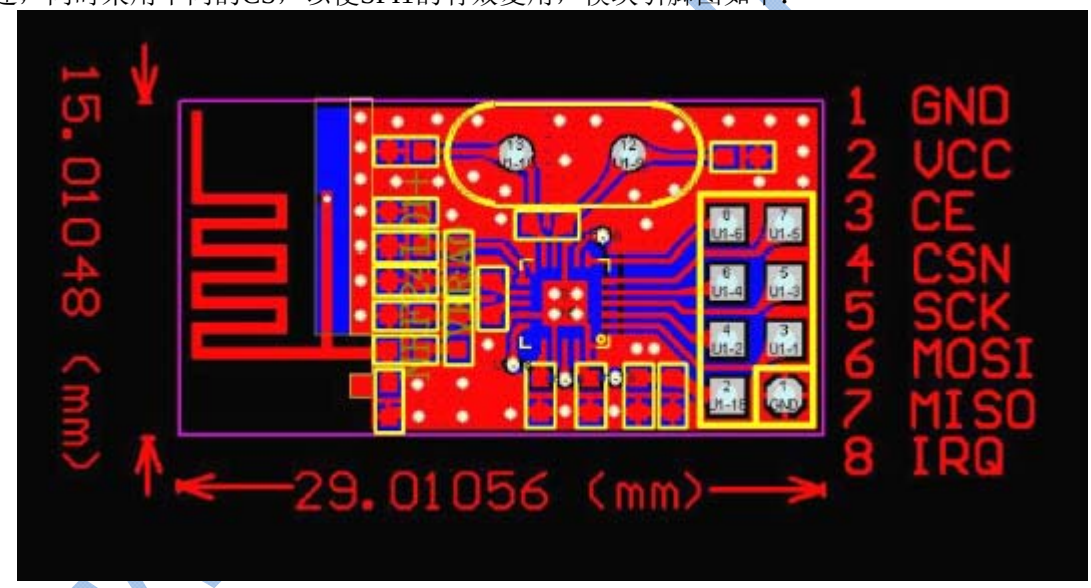
无线模块接口原理图

2.3.16 2.4G无线模块

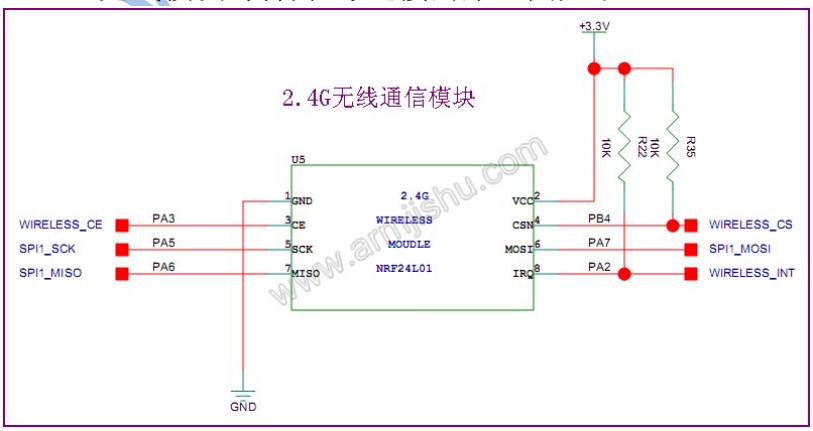
神舟I号提供了1个2.4G的无线模块接口，只要插入模块，我们就可以实现无线通讯信，从而使得我们的板子具备无线功能，由这个功能不一定是标配，所以我们将该接口进行预留，大家可以根据自己的喜好，选择不同的无线模块来使用，产品图片如下：



下图为2.4G无线模块NRF24L01的连接示意图，STM32F103RBT6通过SPI1与NRF24L01无线模块相连，同时采用不同的CS，以便SPI1的有效复用，模块引脚图如下：



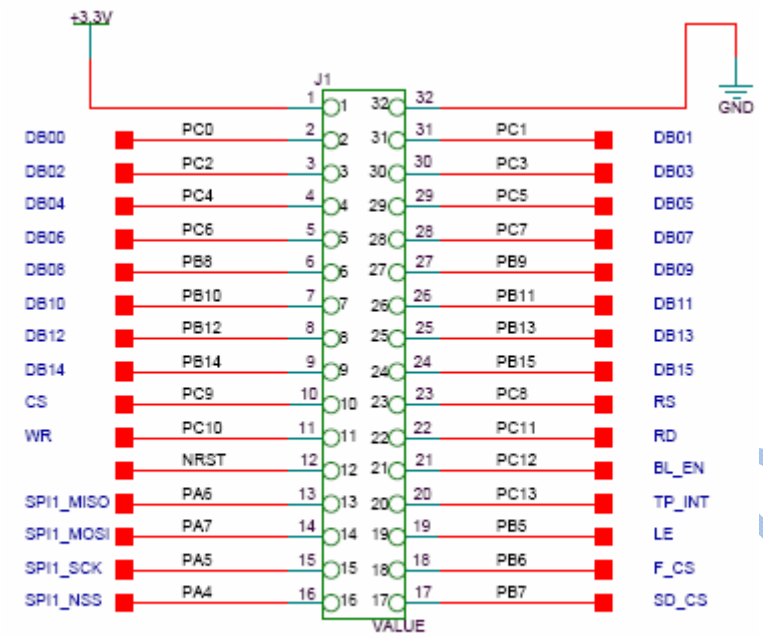
2.4无线模块与神舟I号连接的原理图如下：



由于无线模块必须是双向的，所以必须要有2套板子之间才能够互联通讯。

2.3.17 液晶屏

STM32神舟I号提供TFT液晶接口，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，不仅可支持2.8寸大小，还能支持3.2寸大小的液晶屏。



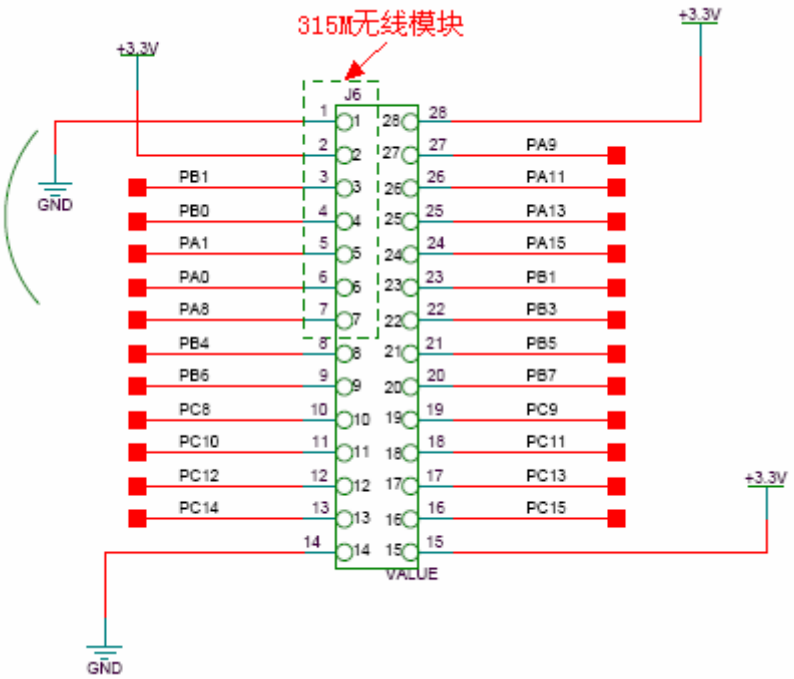
TFT

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，该接口是一个目前比较通用的 LCD 接口，可以将标配件 2.8 寸屏和底板插入 TFT 液晶接口，即可实现彩色图形显示。

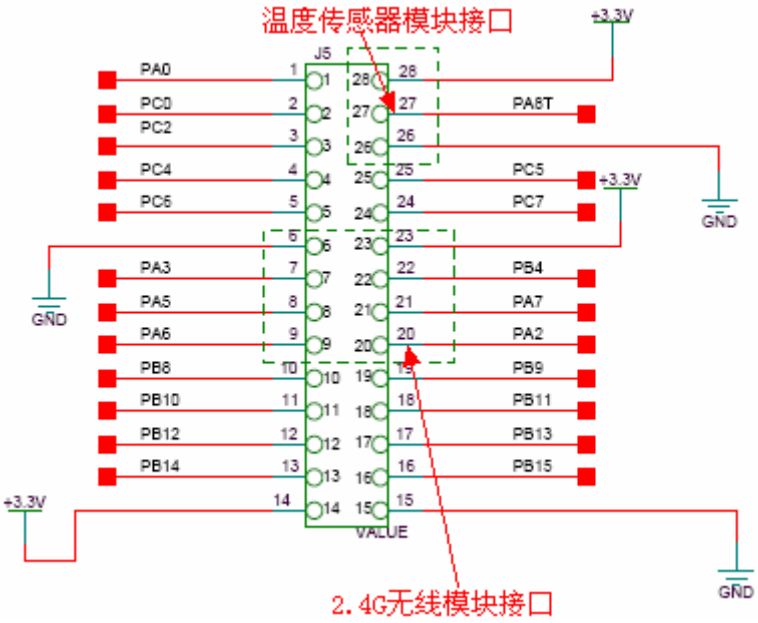
2.3.18 其他扩展接口

STM32神舟I号扩展了56个IO口出来，其中有个位置可以通过外接模块，下面可以插入我们armjishu.com提供的315M无线模块，即可进行无线模块的收发实验：

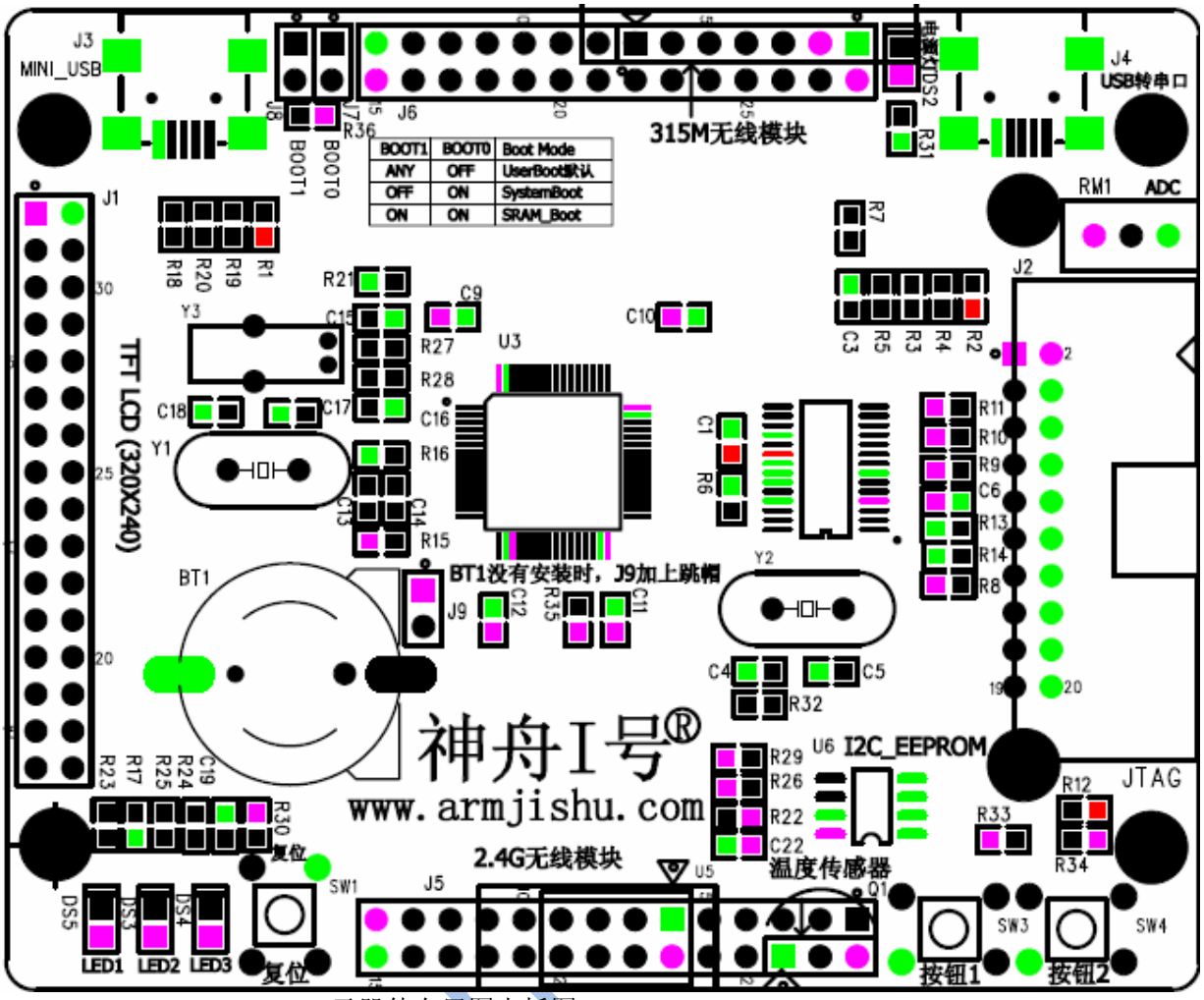
www.armjishu.com



下面这里提供了两个接口，一个是DS18B20的温度传感器，一个是2.4G无线模块，armjishu.com 均有提供，直接插入，就可以正常运行，实现相应的功能：

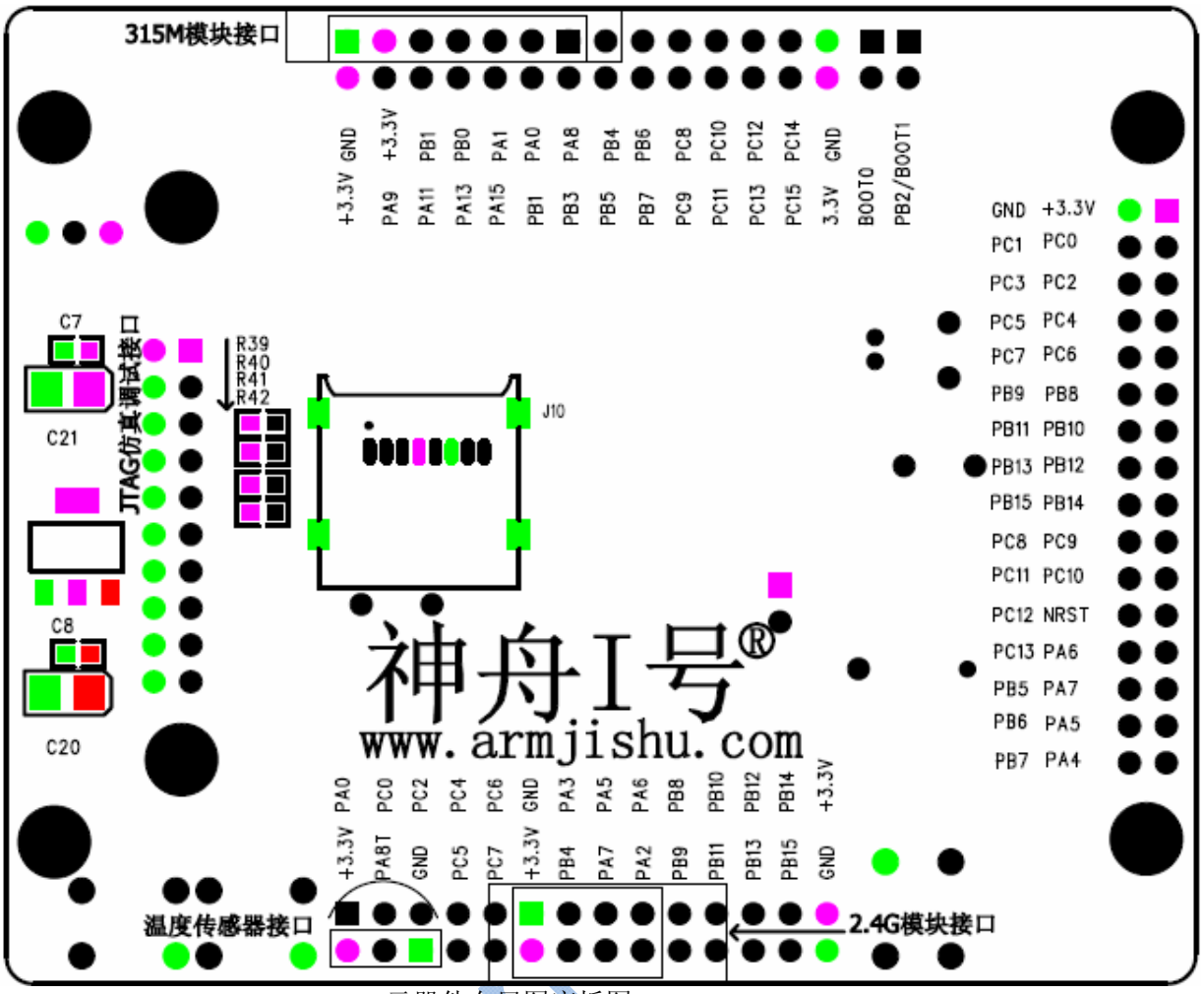


2.4 硬件结构说明



元器件布局图上板图

www.armjishu.com

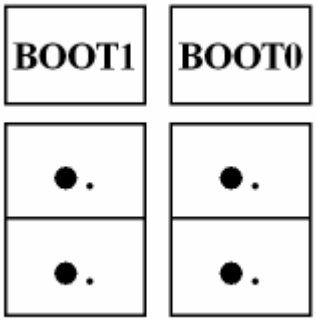


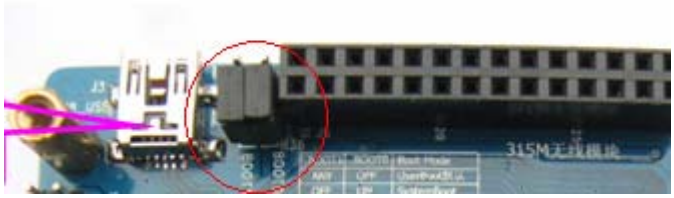
元器件布局图底板图

2.5 连接器说明

1. 处理器启动方式的设置：

下图是 STM32 神舟 I 号引导启动的跳线布局图

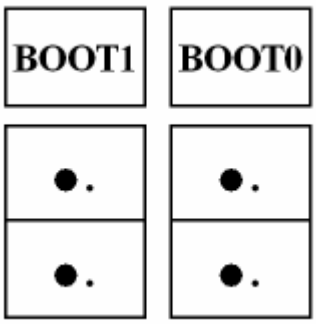




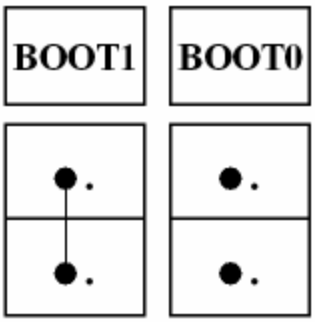
上图中右下角的BOOT0，BOOT1用于设置STM32的启动方式，其对应启动模式如下表所示：

BOOT1（J8）	BOOT0（J7）	功能	说明
ANY	OFF	User Boot(默认)	用主闪存存储器，即Flash启动
OFF	ON	System Boot	系统存储器启动，用于串口下载
ON	ON	SRAM Boot	SRAM启动，用于SRAM中调试代码

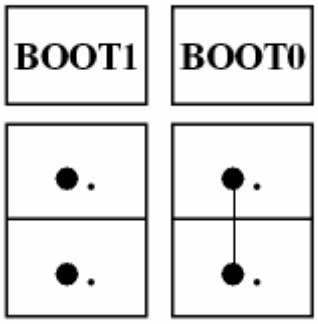
✧ 从主闪存存储器启动：主闪存存储器被映射到启动空间（0x0000 0000），但仍然能够在它原来的地址（0x0800 0000）访问它，即闪存存储器的内容可以在两个地址区域访问，0x0000 0000或0x0800 0000。



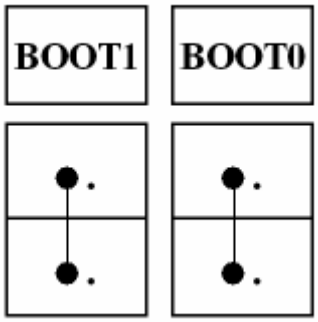
或



✧ 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。

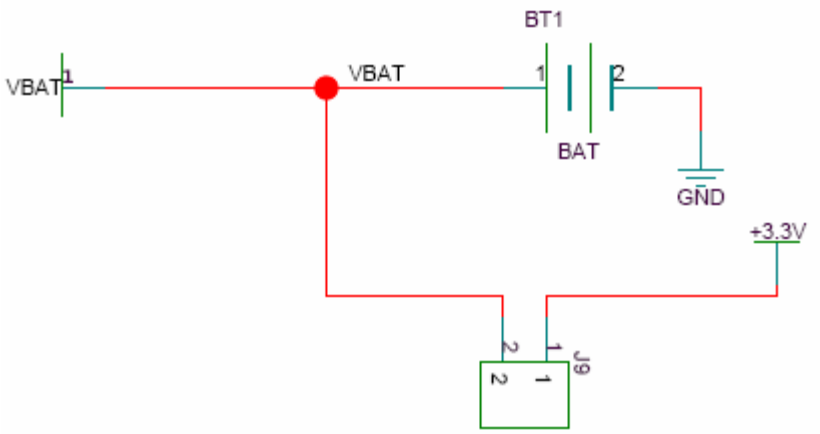


2. RTC 实时时钟设置

1） 有电池时，请断开 J9 的跳帽，使得电池 BAT 为处理器供电

2） 无电池 BAT 时，短路 J9，使 3.3V 电源供电

www.armjishu.com



2.6 硬件资源分配

2.6.1 处理器连接外围器件管脚分配

STM32F103RBT6的资源与管脚分配如下表所示。

www.armjishu.com

管脚号	信号名	功能接口		
14	PA0	315M 无线模块	用户按钮 1	
15	PA1		用户按钮 2	
26	PB0			
27	PB1		ADC	
41	PA8		温度传感器	
60	BOOT0		启动模式	
28	PB2	LED2		
16	PA2	LED1	2.4G 无线模块	
17	PA3	LED3		
56	PB4	Micro SD		2.4/2.8″ LCD 接口
21	PA5			
22	PA6			
23	PA7			
59	PB7		I2C 24C02	
58	PB6			
57	PB5			
20	PA4			
8	PC0			
9	PC1			
10	PC2			
11	PC3			
24	PC4			
25	PC5			
37	PC6			
38	PC7			
61	PB8			
62	PB9			
29	PB10			
30	PB11			
33	PB12			
34	PB13			
35	PB14			
36	PB15			
39	PC8			
40	PC9			
51	PC10			
52	PC11			
53	PC12			
2	PC13			
7	NRST	JTAG/SWD 接口		
55	PB3			

www.armjishu.com

49	PA14			
50	PA15			
42	PA9	USB 转串口		
43	PA10			
44	PA11	MINI-USB 接口		
45	PA12			
54	PD2			

说明：功能接口中同一填充颜色表示相同的接口，模块重叠部分为管脚复用。

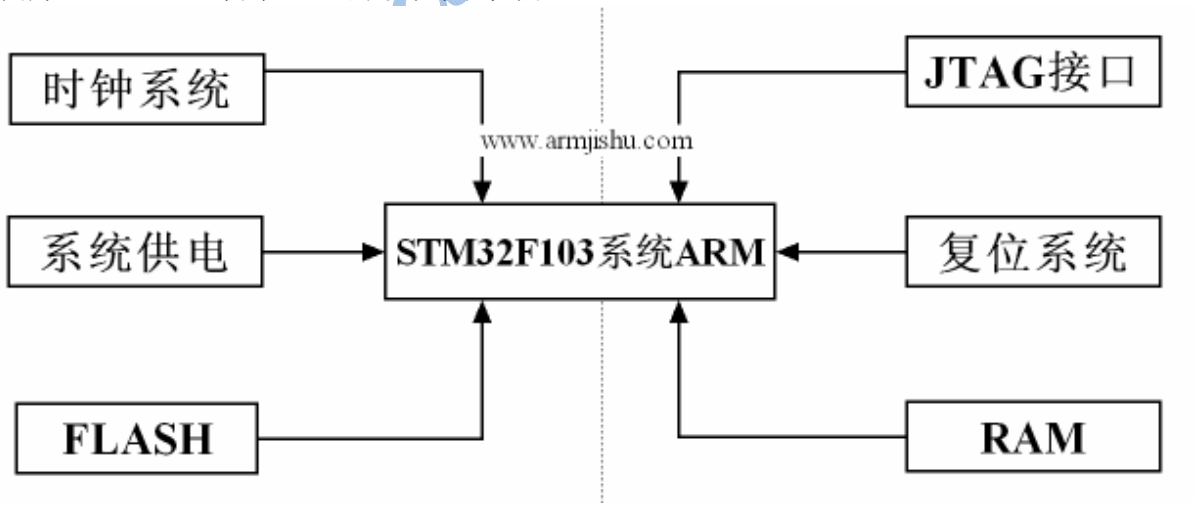
2.7 STM32系列ARM最小系统设计

单片机是一门实践性较强的技术，很多初学者在学习单片机技术开发的时候往往一头雾水，不知何从下手。为此，我们通过讲述单片机原理，通过搭建一个最小系统来加深了解一下 STM32 的最小配置的工作系统。

在了解原理之前， 首先让我们思考一个问题，什么是单片机， 单片机有什么用？ 这是一个有意思的问题，因为任何人都不能给出一个被大家都认可的概念，那到底什么是单片机呢？ 普遍来说，单片机又称单片微控制器， 是在一块芯片中集成了 CPU（ 中央处理器）、RAM（ 数据存储器）、ROM（ 程序存储器）、定时器/ 计数器和多种功能的 I/O（ 输入/ 输出） 接口等一台计算机所需要的基本功能部件，从而可以完成复杂的运算、逻辑控制、通信等功能。在这里，我们没必要去找到明确的概念来解析什么是单片机，特别在使用 C 语言编写程序的时，不用太多的去了解单片机的内部结构以及运行原理等。从应用的角度来说，通过从简单的程序入手，慢慢的熟悉然后逐步深入精通单片机。

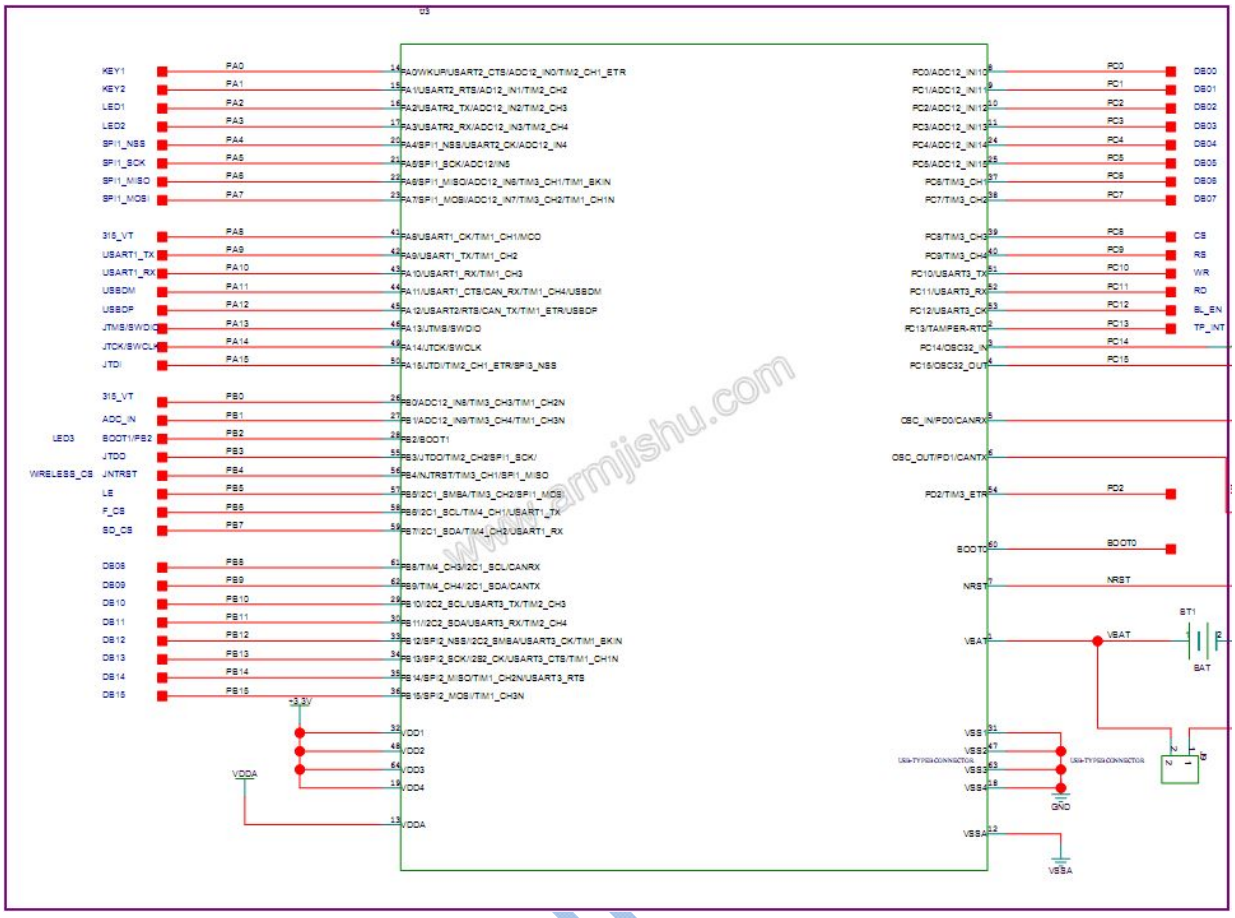
在简单了解了什么是单片机之后，然后我们来构建单片机的最小系统，单片机的最小系统就是让单片机能正常工作并发挥其功能时所必须的组成部分，也可理解为是用最少的元件组成的单片机可以工作的系统。对 STM32 系列单片机来说， 最小系统一般应该包括： 单片机、时钟电路、复位电路、输入/ 输出设备等。

下图为 STM32F103 系列 ARM 的最小系统框图：



1. 最小系统原理图：

◆ **STM32F103RBT6处理器**



◆ STM32最小系统原理图：

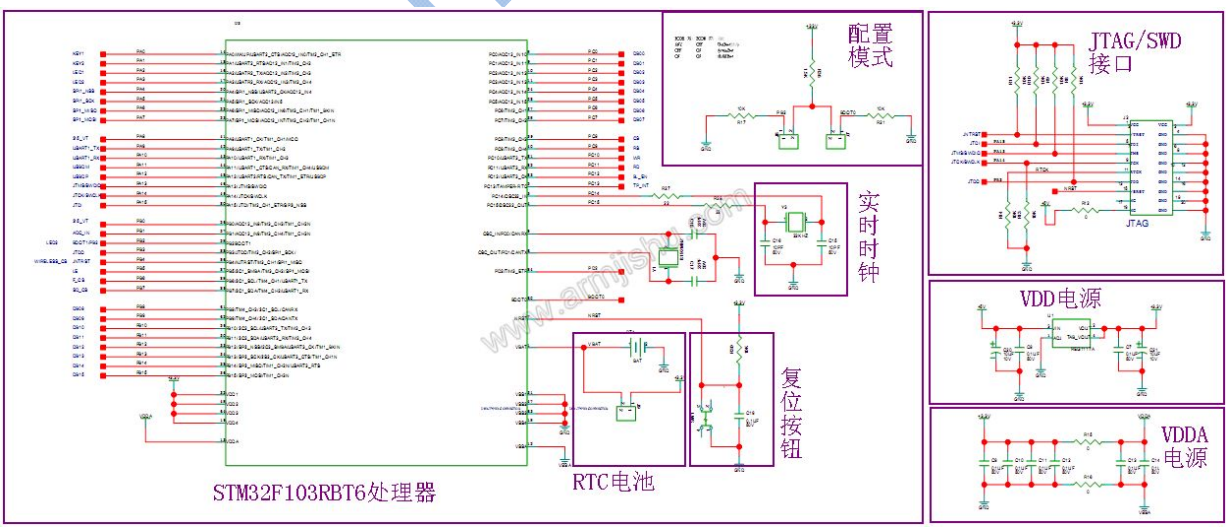


图1.2.1.1 MCU部分原理图

2. 系统存储器 FLASH 和 SRAM

芯片内置 128K 字节的 FLASH 存储器，还有多达 20K 字节的内置 SRAM。

3. 芯片启动模式设置

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。

启动模式选择引脚		启动模式	说明
BOOT1	BOOT0		
X	0	主闪存存储器	主闪存Flash存储器被为启动区域
0	1	系统存储器	串口,CAN等接口被选为启动区域
1	1	内置SRAM	内置SRAM被选为驱动区域

STM32芯片的BOOT0(60脚)和BOOT1 (28脚)是内置SRAM启动和Flash程序存储器或串口等其他接口启动的选择管脚。当BOOT0 保持低电平时，stm32芯片访问从内部flash存储器开始；当BOOT0 保持高电平时，BOOT1为低电平时，则从串口或类似其他CAN等接口启动；当BOOT0和BOOT1都为高电平时，则从内置SRAM启动

对于现今的绝大部分单片机来说，其内部的程序存储器（一般为flash）容量都很大，因此基本上不需要外接程序存储器，而是直接使用内部的存储器。

在我们这个最小系统上，默认启动模式是将BOOT0接低电平(GND),BOOT1无论接高电平VCC还是低电平GND，都只从内部Flash程序存储器启动。这一点一定要注意，如果启动模式没有处理好，会导致程序执行不正常或者系统无法正常启动。

4. 电源供电设计

电源系统为整个系统提供能量，是整个系统工作的基础，设计电源系统的过程实质是一个权衡的过程，必须考虑如下因素：

- 输出功率
- 输入的最大电压和电流
- 安全因素
- 输出纹波
- 电磁兼容和电磁干扰
- 体积限制
- 功耗限制
- 成本限制

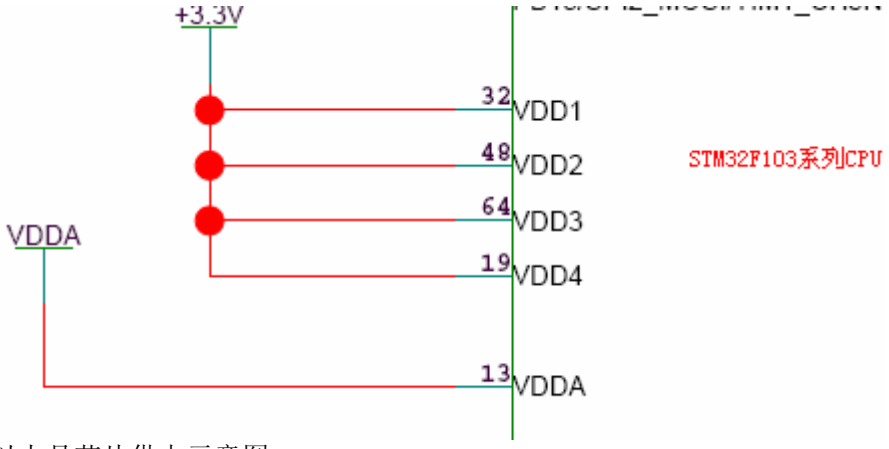
电源设计本身是一个很大的课题，不是本手册说能够容纳得下的，本手册也不打算详细介绍，大家如需要了解，请参考相关书籍。

一般来说，在纯数字电路中没有模拟电路，不存在小信号放大的需要，因此电路对噪声不敏感（即噪声容限比较大），系统对电源电路的噪声抑制性能要求不高。

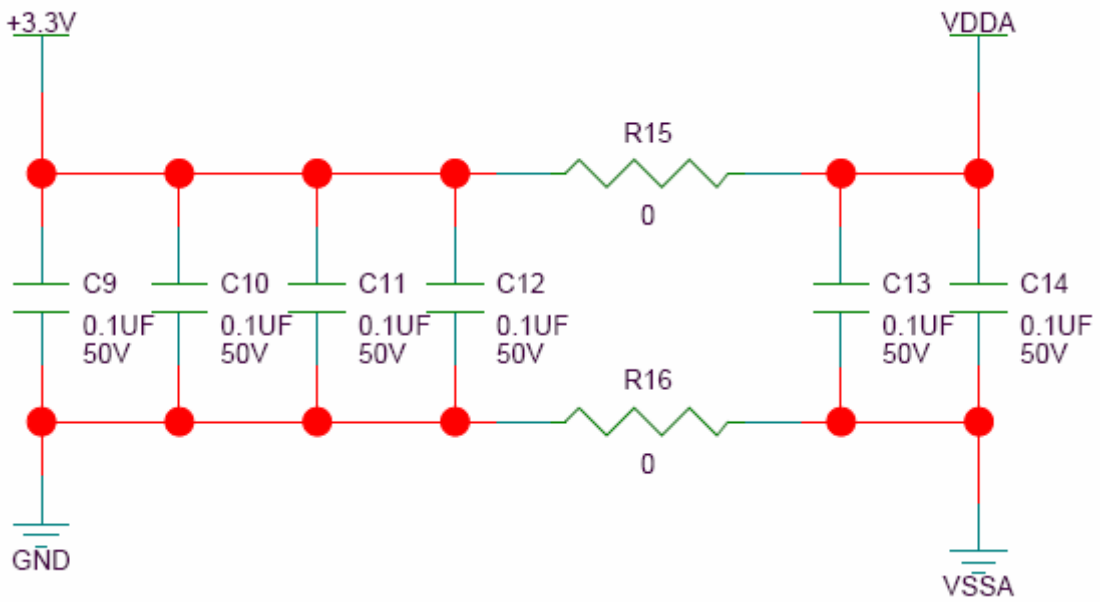
对于包含模拟电路的数字/模拟混合电路系统中，系统则对电源电路要求较高，特别是噪声抑制性能上。外部输入模拟信号经过模拟放大整形电路后进入 A/D 转换器变成数字量。模拟电路容易受到各种干扰的影响，使信号畸形导致测量不准确。模拟电路遭受干扰的途径很多，但是通过电源引入的噪声最大。

数字电路通常是噪声干扰的产生源，若电源设计不得当，电源电路便会成为数字电路产生的噪声传递给模拟电路的直接通道。为了防止电源电路成为噪声传递的通道，最好的方法是数字电路和模拟电路分开供电，在电源地线处理上，通常采用单点接地方法将数字电源地和模拟电源地在总电源处通过小磁珠或 0 欧姆电阻相连。

www.armjishu.com



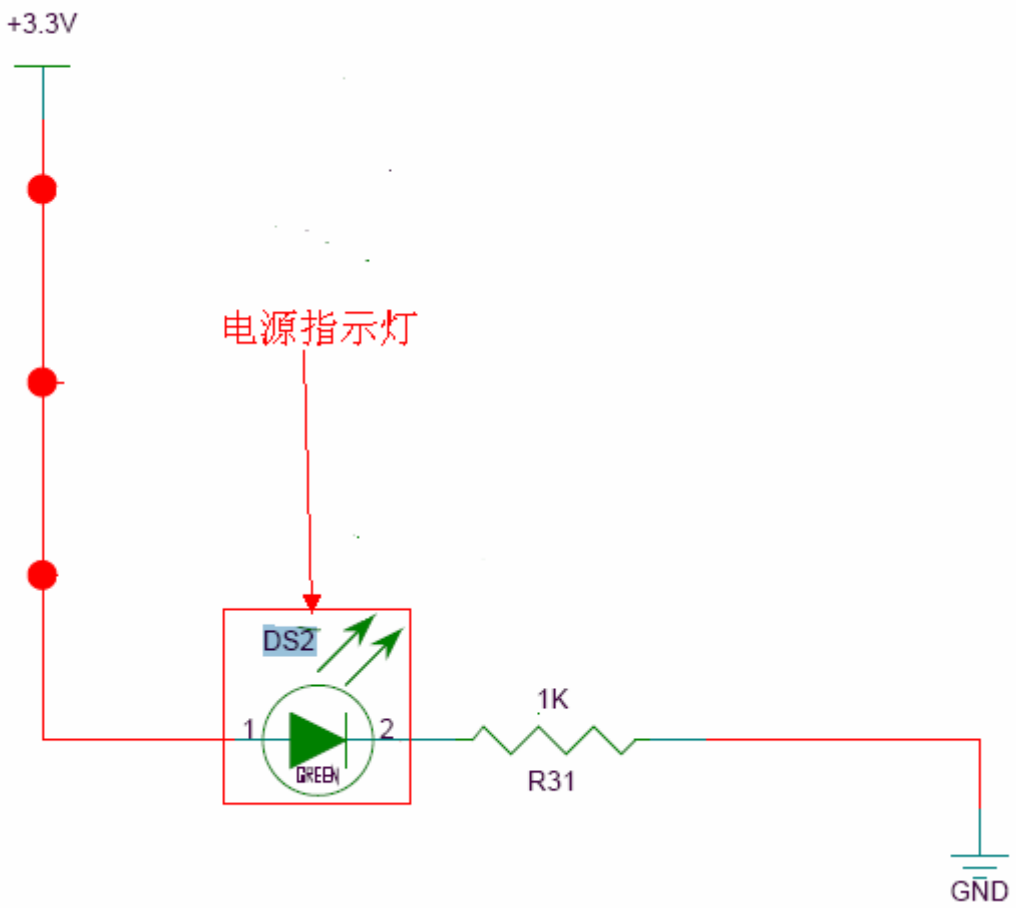
以上是芯片供电示意图；



以上是数字电源和模拟电源被 0 欧姆电阻隔离开的示意图（电阻 R15 和电阻 R16），防止数字电源与模拟电源之间有一些干扰存在。

5. LED 驱动电路（电源指示灯）

细心的爱好者可能已经发现，在最小系统中，发光二极管（LED）的接法是采取了电源接到二极管正极再经过 1K 电阻接到单片机 I/O 口上的。



为什么这么接呢？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。其次，STM32 单片机（如本实验板中所使用的 STM32F103RBT 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)。

而灌电流（要 VCC 往内输入电流）的方式确非常轻松，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。另外，图中的电阻为 1K 阻值，是为了限制电流，让发光二极管的工作电流限定在 2mA~10mA。

6. 时钟系统（当不被使用时，任一个时钟源都可被独立地启动或关闭，由此优化系统功耗）

STM32 系列的 ARM 的所有时钟从三种基本时钟源得到：

- **内部 HSI 时钟信号：**

  由 8MHz 的 RC 振荡器产生，可直接作为系统时钟或在 2 分频后作为 PLL 输入；HIS RC 振荡器能够在不需要任何外部器件的条件下提供系统时钟；它的启动时间比 HSE 晶体振荡器短；然后，即使校准之后它的时钟频率精度仍较差。
- **外部 HSE 时钟可以是以下两种中的任何一种：**

  ----- 外部时钟源（外部时钟的最高频率可以达到 25MHz）

  ----- 晶振（建议最好是 4～16MHz 的主时钟更为稳定）
- **内部 PLL 时钟：**

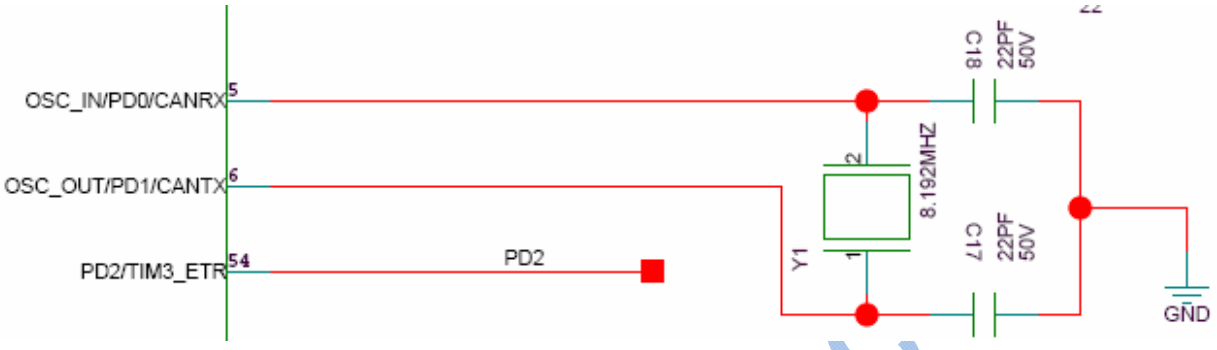
  内部 PLL 可以用来倍频 HIS RC 的输出时钟或 HSE 晶体输出时钟。PL 的设置（选择 HIS 振荡器除 2 或 HSE 振荡器为 PLL 的输入时钟，和选择倍频因子）必须在其被激活前完成，一

旦 PLL 被激活，这些参数就不能被改动。

这些设备有以下 2 种二级时钟源：

- 40kHz 低速内部 RC，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。
- 32.768kHz 低速外部晶体也可用来通过程序选择驱动 RTC

以下是电路原理图，我们分别使用了一颗 8MHz 的晶体和 32.768kHz 的晶体：



在设计时钟电路之前，让我们先了解下 STM32 单片机上的时钟管脚：

OSC_IN（5 脚）：芯片内部振荡电路输入端。

OSC_OUT（6 脚）：芯片内部振荡电路输出端。

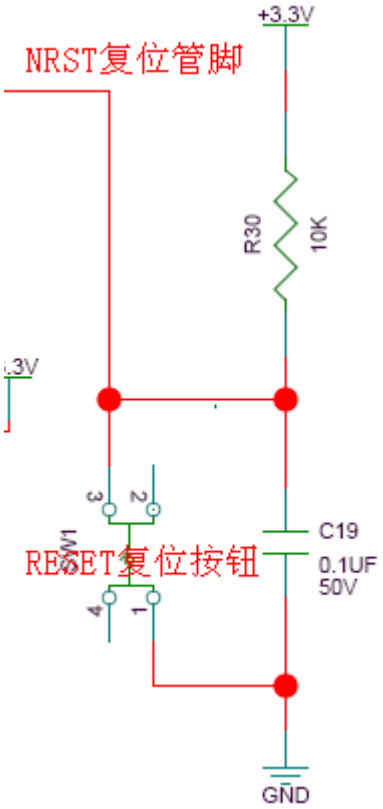
OSC_IN 和 OSC_OUT 是独立的输入和输出反相放大器，它们可以被配置为使用石英晶振的片内振荡器，或者是器件直接由外部时钟驱动。图 2 中采用的是内时钟模式，即采用利用芯片内部的振荡电路，在 OSC_IN、OSC_OUT 的引脚上外接定时元件（一个石英晶体和两个电容），内部振荡器便能产生自激振荡。一般来说晶振可以在 4 ～ 16MHz 之间任选，频率越高功耗也就越大。在本实验套件中采用的 8.192MHZ 的石英晶振。和晶振并联的两个电容的大小对振荡频率有微小影响，可以起到频率微调作用。当采用石英晶振时，电容可以在 20 ～ 40pF 之间选择（本实验套件使用 30pF）；当采用陶瓷谐振器件时，电容要适当地增大一些，在 40 ～ 60pF 之间。这个要参考硬件手册，我们这里选取 50pF 的陶瓷电容就可以了。

另外值得一提的是如果读者自己在设计单片机系统的印刷电路板（PCB） 时，晶体和电容应尽可能与单片机芯片靠近，以减少引线的寄生电容，保证振荡器可靠工作。检测晶振是否起振的方法可以用示波器可以观察到 OSC_OUT 输出的十分漂亮的正弦波，也可以使用万用表测量（ 把挡位打到直流挡，这个时候测得的是有效值）OSC_OUT 和地之间的电压时，可以看到 2V 左右一点的电压。

7. 复位系统

板子提供一个复位按钮，STM32F10XXX 是低电平复位的，STM32 神舟系列开发板板载的复位按键，可用于复位整个开发板，还具有复位 LCD 液晶屏的功能，原理是因为 LCD 液晶屏模块的复位引脚和 STM32 的复位引脚是连接在一起的，当然也其他各个芯片的复位引脚也是可以连在一起，这样设计能够实现一按按钮是 STM32 神舟开发板的整板硬件都会被同时进行复位。

神舟系列开发板本次设计采用的是简单的“RC＋按键”复位形式，复位电路的连接示意图如下图所示，该复位电路可以实现上电自动复位功能和手动按键复位功能，下图就是复位按键的图：



复位电路原理图

在 STM32 单片机系统中，复位电路是非常关键的，当程序跑飞（运行不正常）或死机（停止运行）时，就需要进行复位。

STM32 系列单片机的复位引脚 NRST(第 7 管脚) 出现几个机器周期以上的低电平时，单片机就执行复位操作。如果 NRST 持续为低电平，单片机就处于循环复位状态。

复位操作通常有两种基本形式：上电自动复位和开关复位。

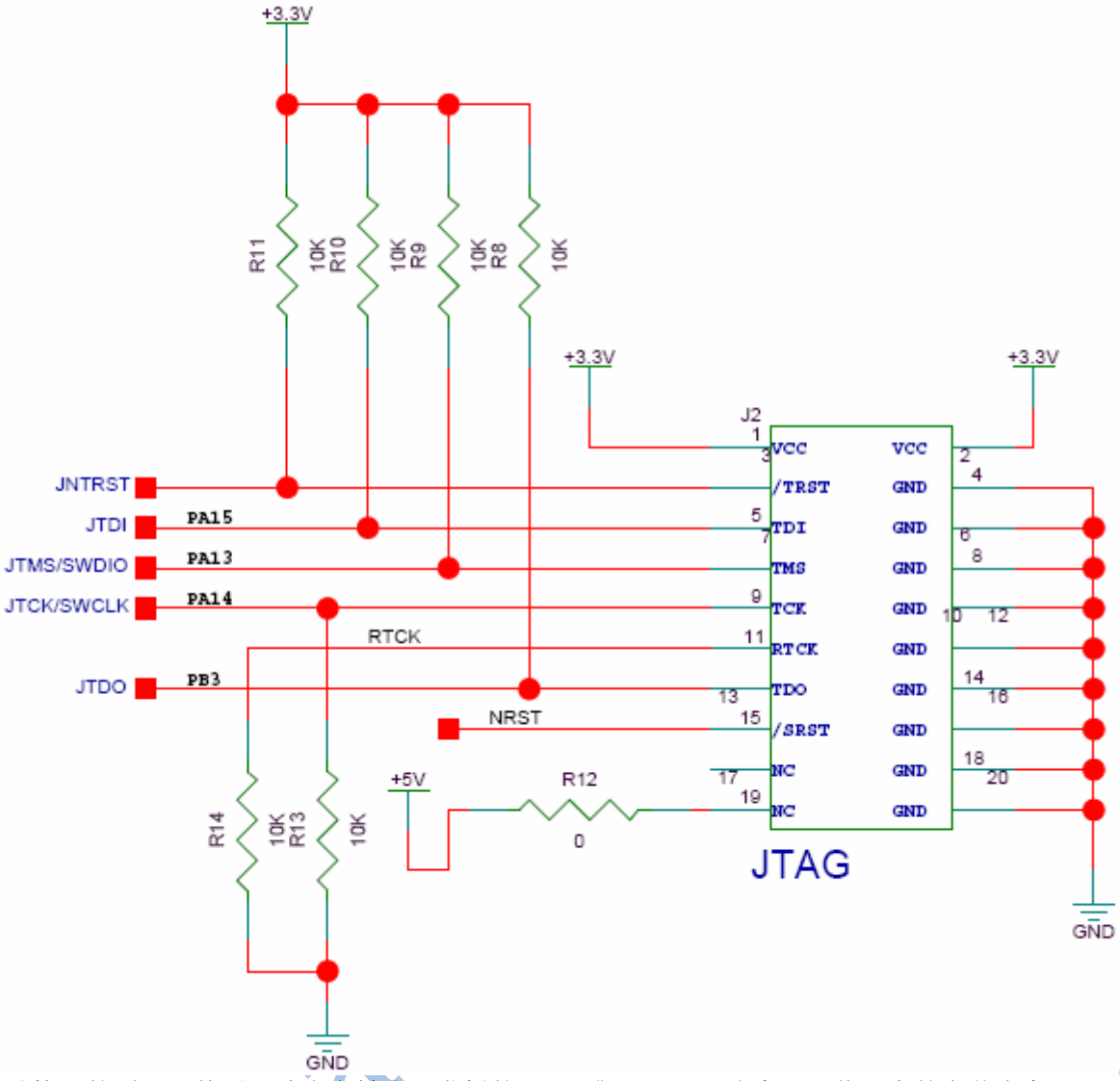
上图中所示的复位电路就包括了这两种复位方式。上电瞬间，电容两端电压不能突变，此时电容的正极和 NRST 相连，电压慢慢升高，NRST 复位管脚 的输入为低，芯片被复位。随之+3.3V 电源给电容充电，电阻上的电压逐渐变大，最后 NRST 复位管脚上的电压编程+3.3v，芯片正常工作。并联在电容的两端为复位按键。

当复位按键没有被按下的时候电路实现上电复位，在芯片正常工作后，通过按下按键使 RESET 管脚出现电容瞬间放电变成低电平达到手动复位的效果。一般来说，只要 RESET 管脚上保持 10ms 以上的高电平，就能使单片机有效的复位。图中所示的复位电阻和电容为经典值，实际制作是可以用同一数量级的电阻和电容代替，读者也可自行计算 RC 充电时间或在工作环境实际测量，以确保单片机的复位电路可靠。

8. 调试接口

调试接口采用 ARM 公司提出的标准 20 脚 JTAG 仿真调试接哦库，调试接口如下图所示：

www.armjishu.com



调试接口的原理和使用，请参考神舟开发板的JTAG或JLINK V8仿真器下载对应的章节内容。

第3章 其他篇

3.1 液晶屏显示屏入门

TFT 显示屏是目前最好的 LCD 彩色显示设备之一，是现在笔记本电脑和台式机上的主流显示设备，STM32 神舟 I 号就是通过处理器与 TFT 液晶屏进行通信，将需要显示的数据，需要控制的相关的命令，通过信号线传输给 TFT 液晶屏，并从 TFT 液晶屏或得反馈，从而达到控制和使得液晶屏显示我们想要的图像内容，液晶是一种性能介于液体和晶体之间的一种有机高分子材料，它既有液体的流动性，又有晶体结构排列的有序性。

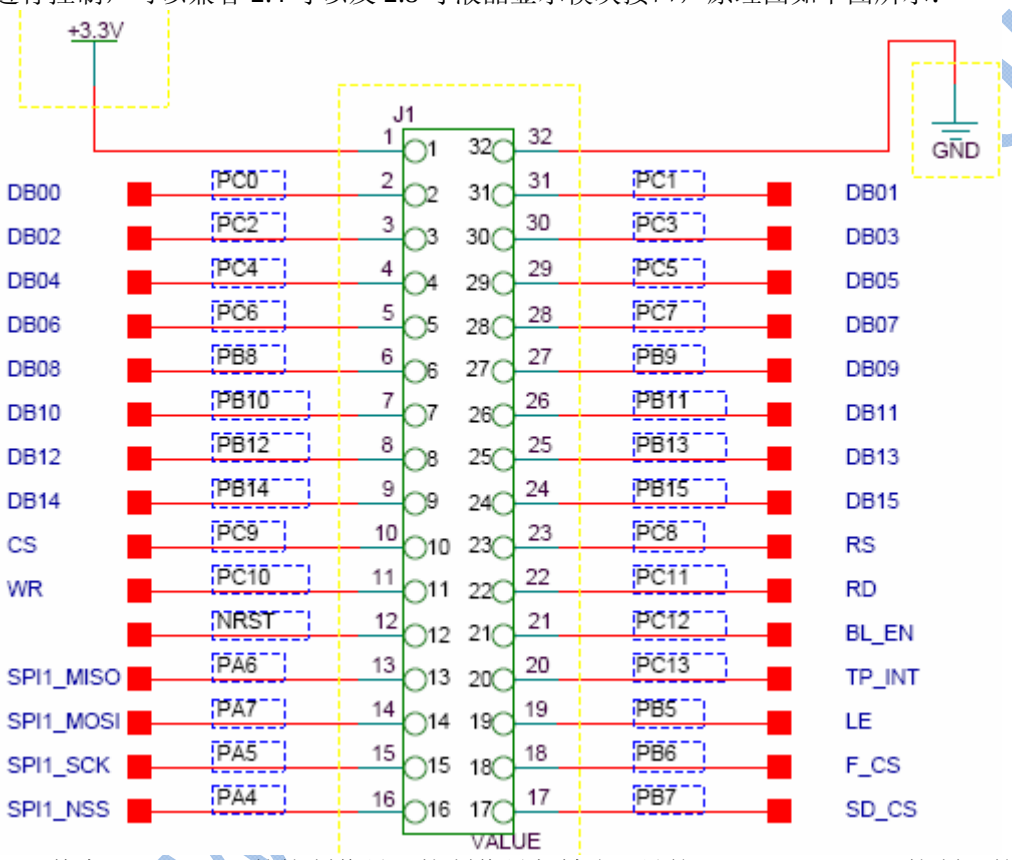
关于 TFT（Thin Film Transistor，薄膜晶体管）它是有源型的 LCD，其显示屏的每个液晶像素点都是由集成在像素点后面的薄膜晶体管来控制，使每个象素都能保持一定电压，从而可以做到高速度，高亮度，高对比度的显示。我们这里所使用 TFT 液晶屏实际上已经有了一个控制器，比如 IL9320 液晶屏控制芯片是目前最常用的液晶屏控制器，我们的 CPU 实际上就是与这个控制器进行通信，从而达到控制 TFT LCD 液晶屏的目的，这个 IL9320 液晶屏控制芯片如果换成另外一种控制芯片比如

SSD1289 液晶屏控制芯片，那么相应的液晶屏驱动也是需要做相应的修改，因为两款不同液晶屏控制芯片本身的管脚定义有可能不同，通信协议也有区别，所以 CPU 与这些液晶屏控制芯片沟通的时候，也是要根据具体的芯片型号，选择支持不同的芯片驱动。

目前 STM32 神舟 I 号开发板液晶屏驱动能支持多达几十种不同控制器芯片的液晶屏，因为液晶屏都是一样的，市场上各种各样的 2.4 寸/2.8 寸屏唯一不同的就是这颗控制器芯片，我们的驱动可以尝试与控制器进行一问一答的沟通方式，获得插入的液晶屏模块控制器的真正型号，自动配置好相对应的驱动，从而达到支持几十种不同类型的屏的目的。

3.2 液晶屏底板设计说明

STM32 神舟 I 号提供了 TFT 液晶接口，TFT 液晶接口通过 STM32F103 系列的 RBT 芯片管脚来进行控制，可以兼容 2.4 寸以及 2.8 寸液晶显示模块接口，原理图如下图所示：



其中，TFT LCD 的控制信号，控制信号与神舟 I 号的 STM32F103RBT 控制器的连接关系如下所示：

CS ←→ PC9 LCD 片选信号
RS ←→ PC8 命令/数据标志（1，读写数据；0，读写命令）
WR ←→ PC10 向 LCD 写入数据
RD ←→ PC11 从 LCD 读取数据
BL_EN ←→ PC12 背光控制

神舟系列的液晶模块支持触摸功能，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。

可以将标配的 2.4/2.8 寸 TFT 液晶屏模块板插入到 STM32 神舟 I 号开发板的液晶接口上，即实现彩色图形的显示。

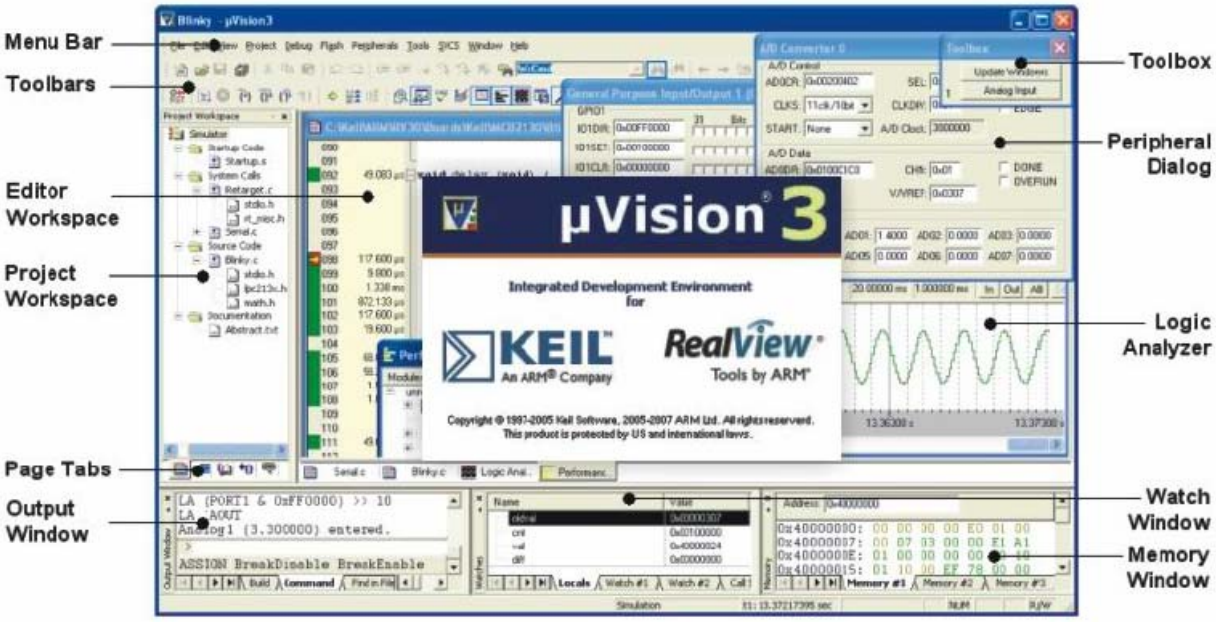
第4章 STM32神舟I号基本操作篇

4.1 简介

本章将简单介绍使用MDK 4.12建立工程，编译连接设置，调试操作等。还介绍基于STM32系列ARM微控制器的工程模板的使用，JLINK V8仿真器的安装与使用，在三种环境中调试的配置，GPIO程序在内部RAM中调试，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。

4.2 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μVision4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%。



4.3 安装MDK的流程步骤

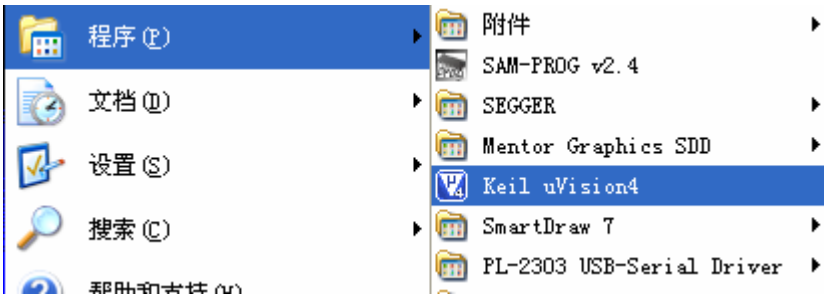
具体安装方法和流程如下：

1. 安装 Keil MDK4.12 版本，即 uV4
2. 打开 uVision4，点击 File---License Management….打开 License Management 窗口，复制右上角的 CID
3. 打开 KEIL_Lic.exe 注册机，在 CID 窗口里填上刚刚复制的 CID，其他设置不变
4. 点击 Generate 生成许可号，复制许可号
5. 将许可号复制到 License Management 窗口下部的 New License ID Code，点击右侧的 Add LIC
6. 若上方的 Product 显示的是 RealView MDK-ARM 即表示注册成功，Support Period 为有效期，一般可以到 10 年左右，若有效期较短，可多次生成许可号重新注册。

4.4 工程的编辑

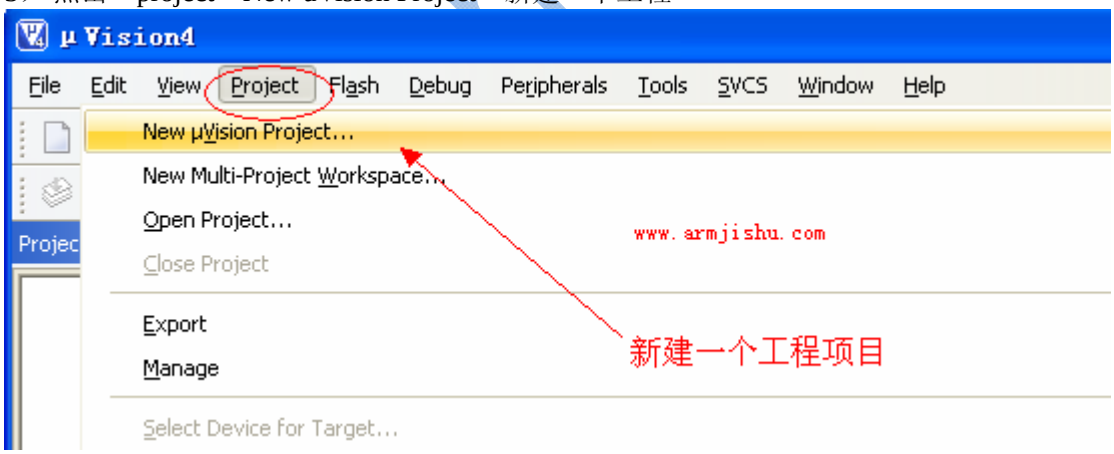
4.4.1 建立工程

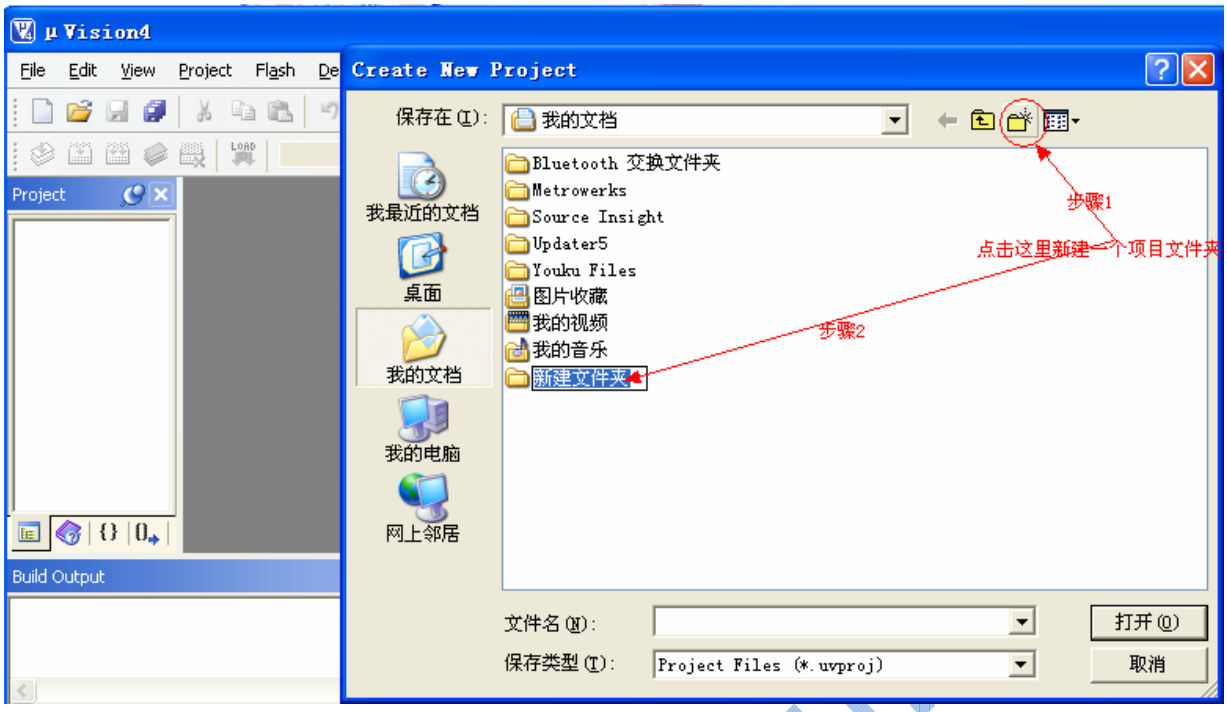
1）点击 WINDOWS 操作系统的【开始】→【程序】→【Keil uVision4】启动 Keil uVision 或在桌面双击【Keil uVision4】快捷方式启动。启动 MDK4.12 如图所示：



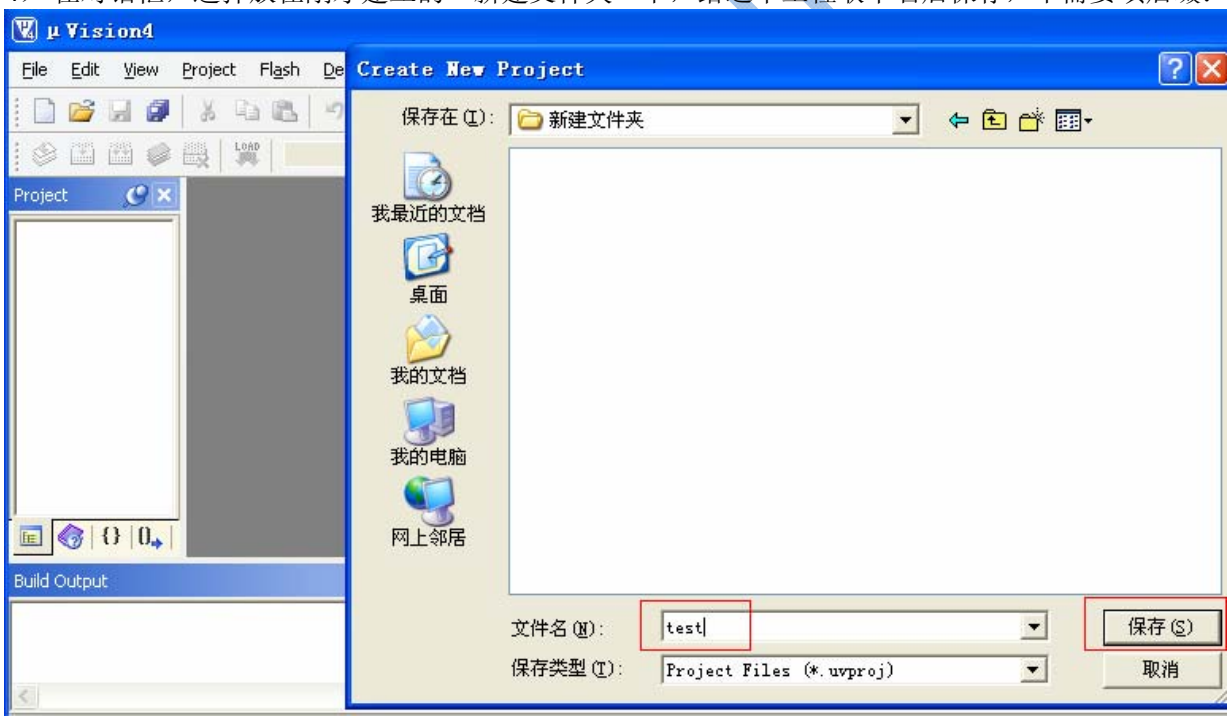
2）点击之后，出现启动画面：

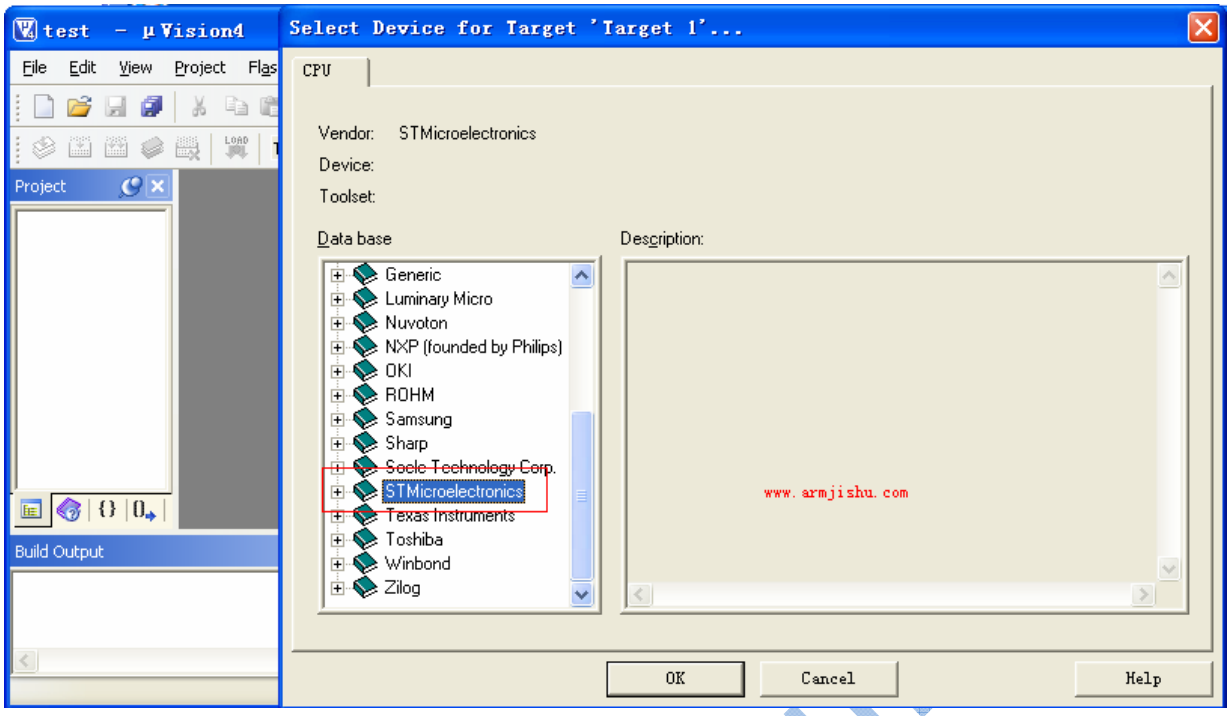
3） 点击“project---New uVision Project”新建一个工程



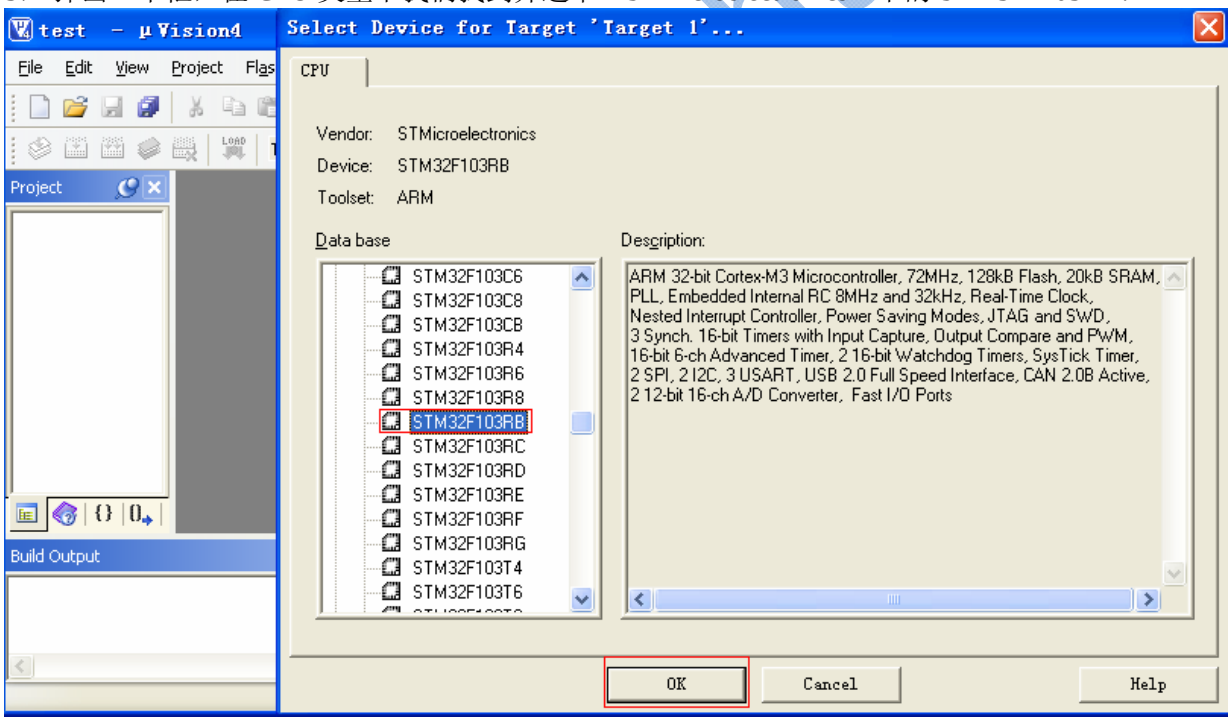


4）在对话框，选择放在刚才建立的“新建文件夹”下，给这个工程取个名后保存，不需要填后缀：

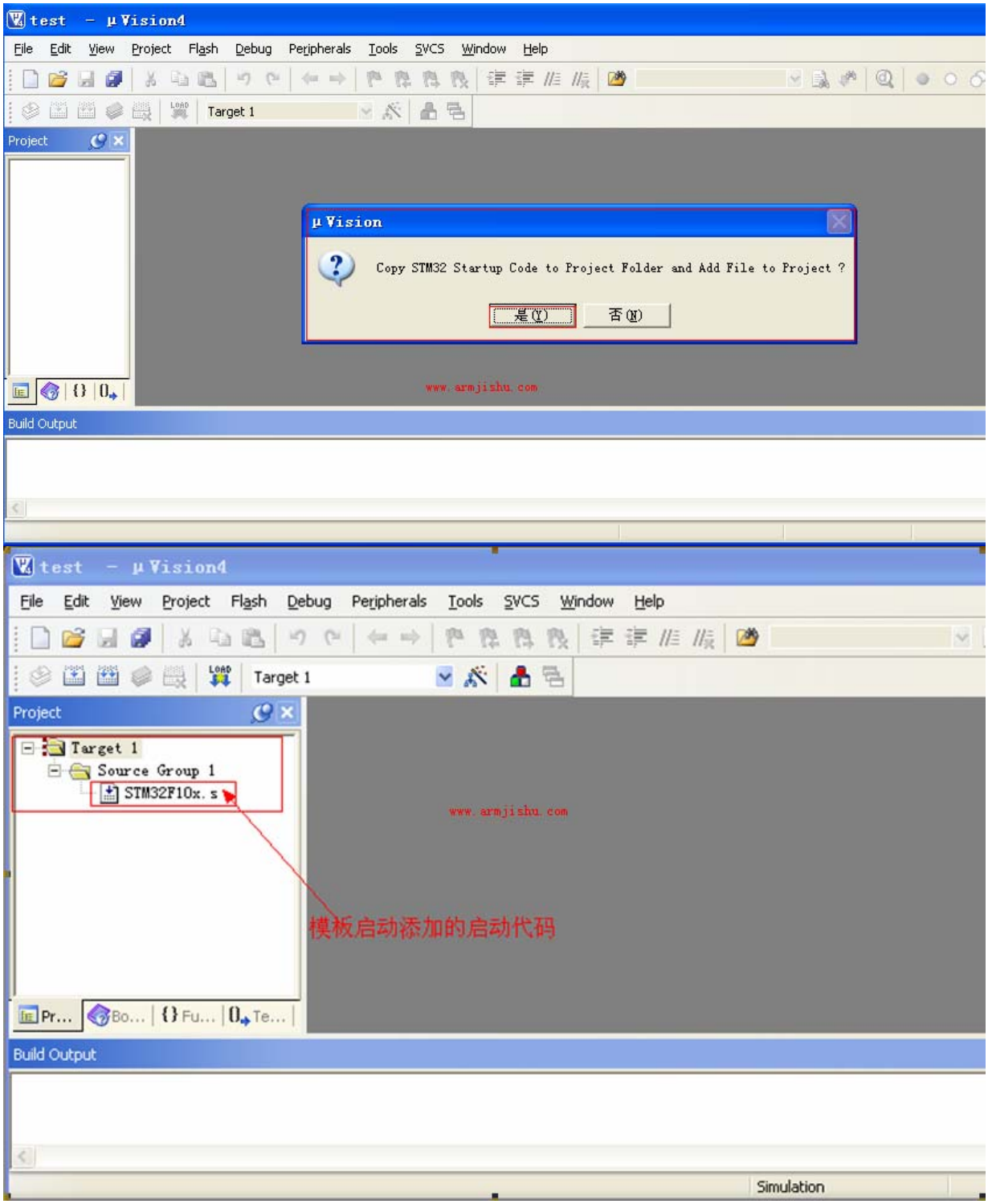




5） 弹出一个框，在 CPU 类型下我们找到并选中“STMicroelectronics”下的 STM32F103RB：



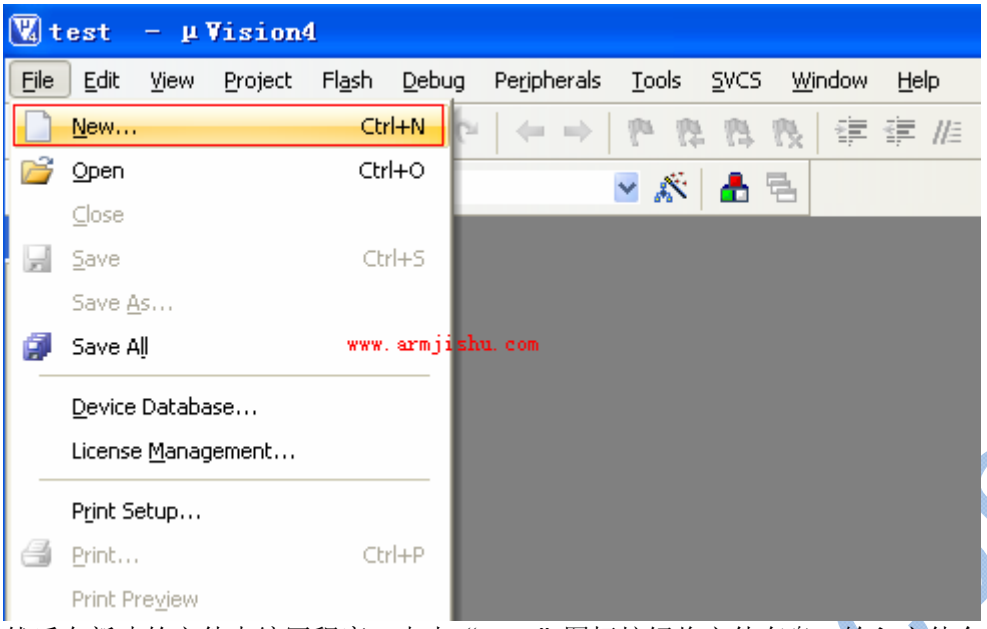
6） 出现一个提示框，是否复制 STM32 启动代码到工程文件夹，我们选择【是】，就可以看到 STM32 的启动代码自动添加进来了：



7） 到这里工程全部建立完毕

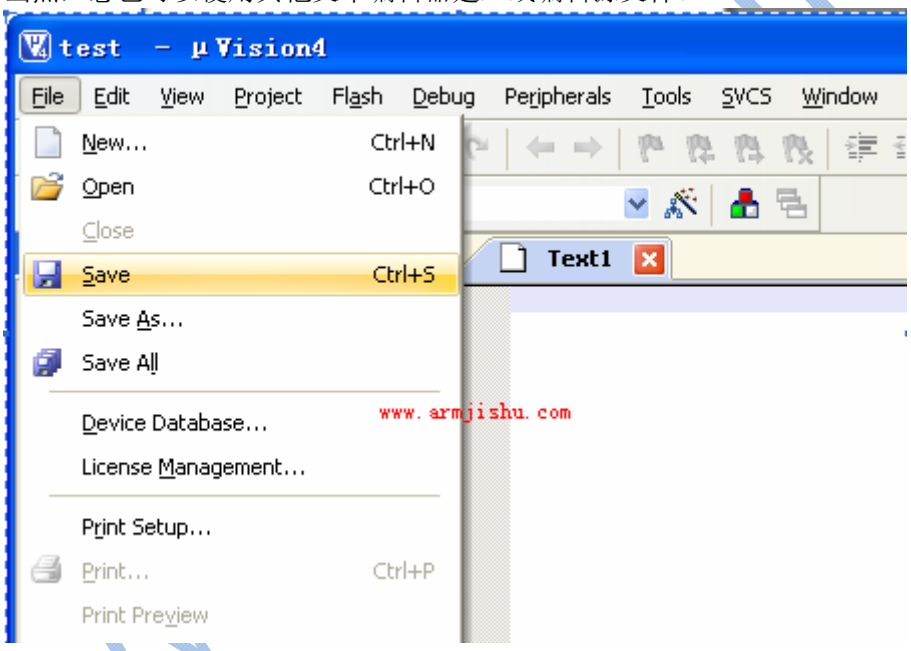
4.4.2 建立文件

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



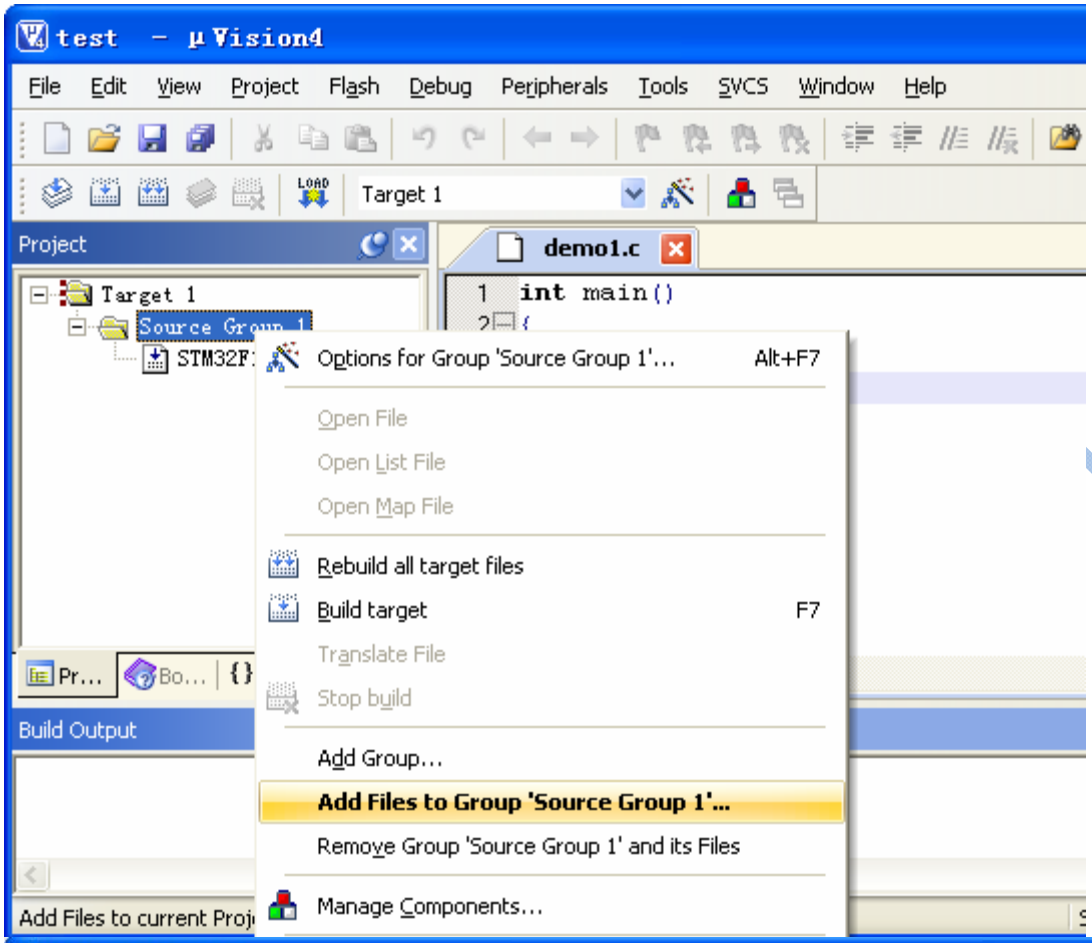
然后在新建的文件中编写程序，点击“Save”图标按钮将文件存盘，输入文件全名，如 demo.c。注意，请将文件保存到相应工程的目录下，以便于管理和查找。

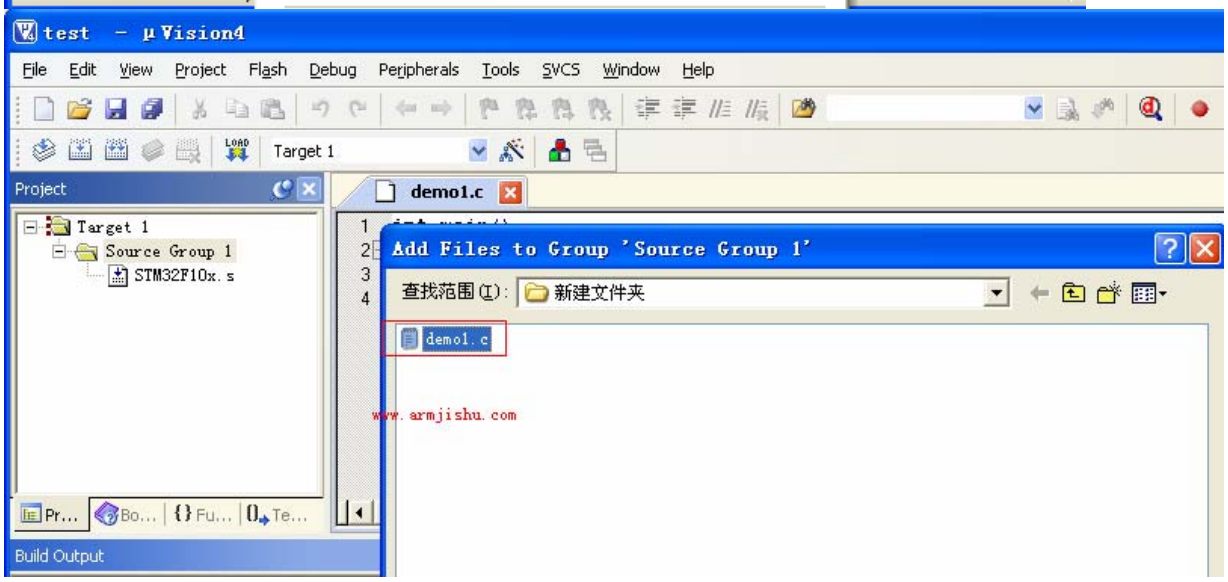
当然，您也可以使用其他文本编辑器建立或编辑源文件。

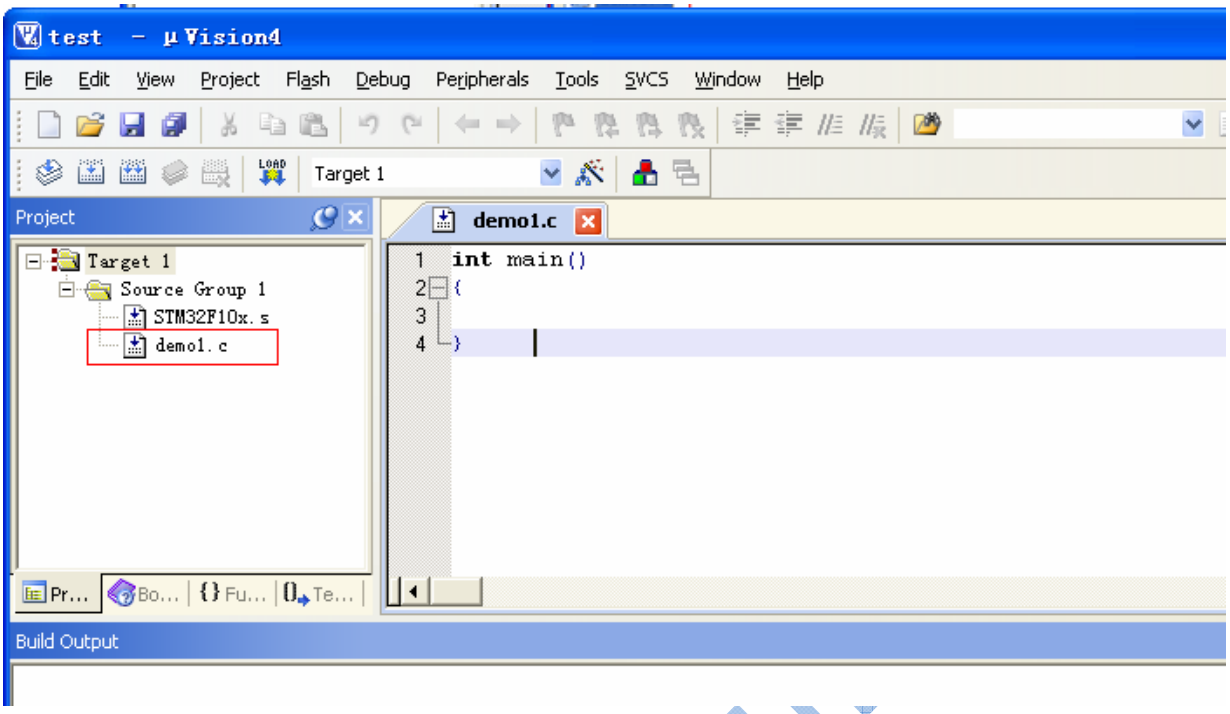


4.4.3 添加文件到工程

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。


test - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
demo1.c
Target 1
Source Group 1
STM32F
int main()
Options for Group 'Source Group 1'... Alt+F7
Open File
Open List File
Open Map File
Rebuild all target files
Build target F7
Translate File
Stop build
Add Group...
Add Files to Group 'Source Group 1'...
Remove Group 'Source Group 1' and its Files
Manage Components...
Build Output
Add Files to current Proj

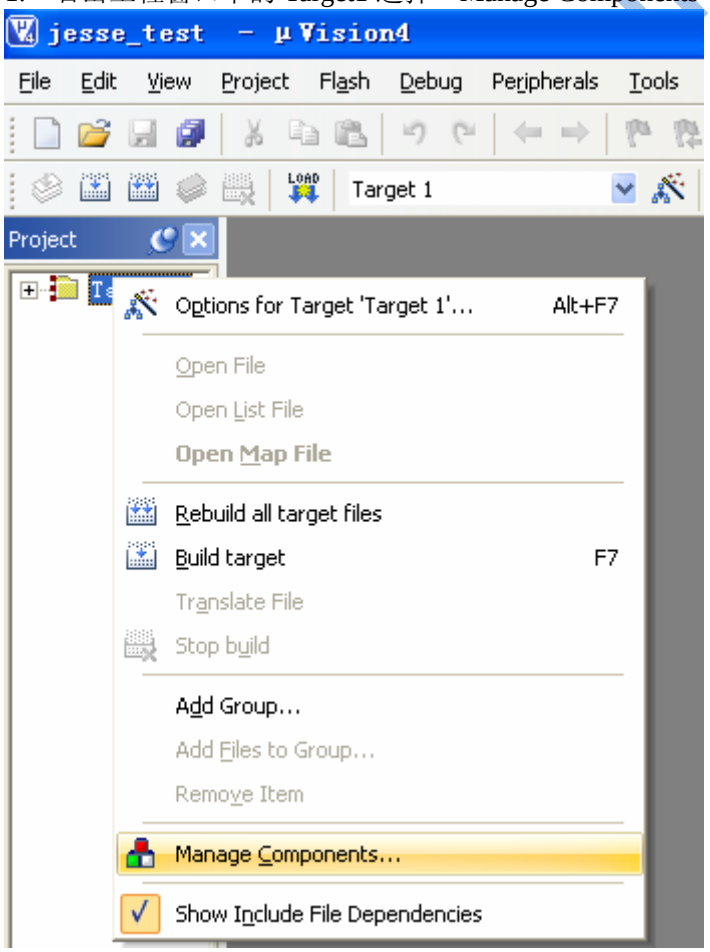

test - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
demo1.c
Target 1
Source Group 1
STM32F10x.s
Add Files to Group 'Source Group 1'
查找范围(I): 新建文件夹
demo1.c
www.armjishu.com
Build Output



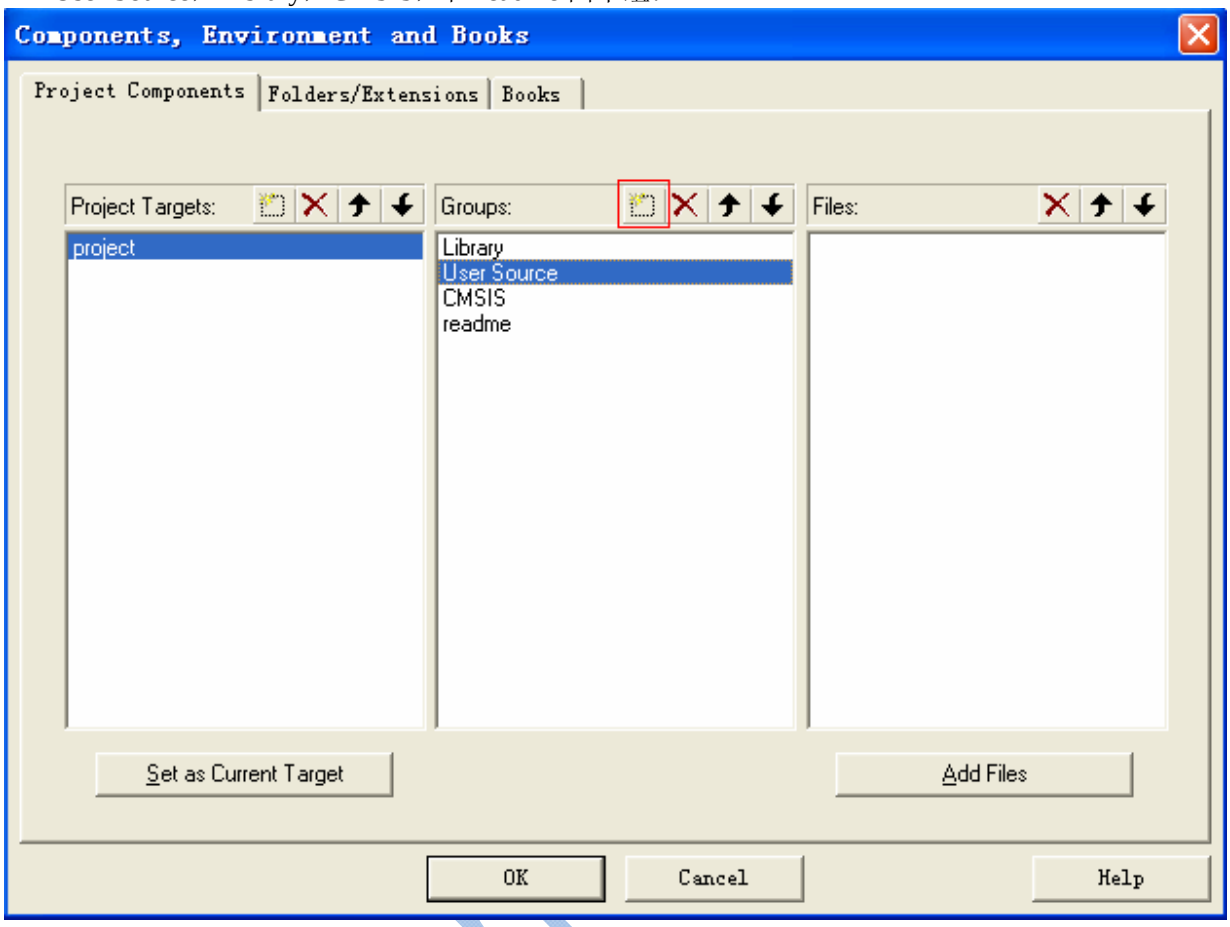
4.4.4 管理工程目录以及源文件

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：

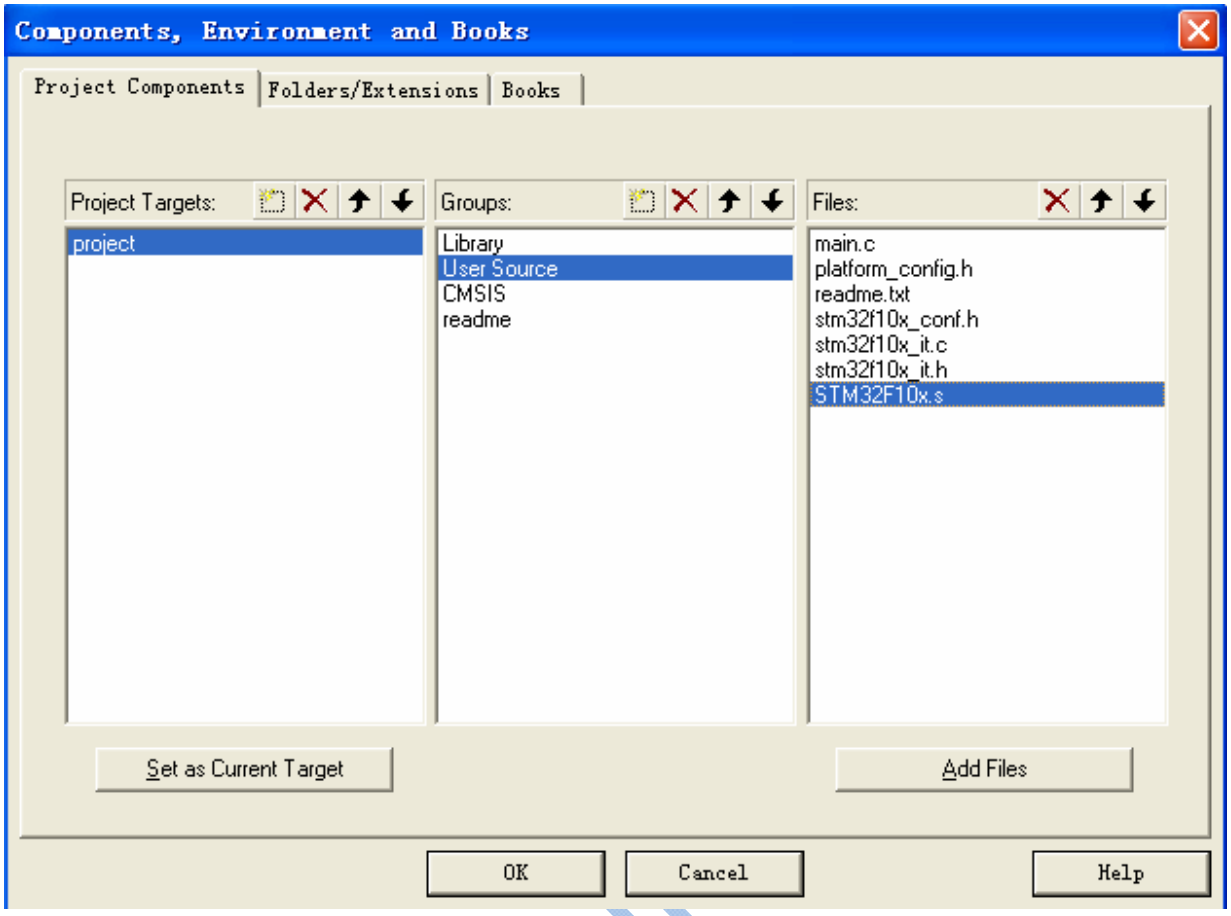
1. 右击工程窗口中的 Target1 选择“Manage Components……”

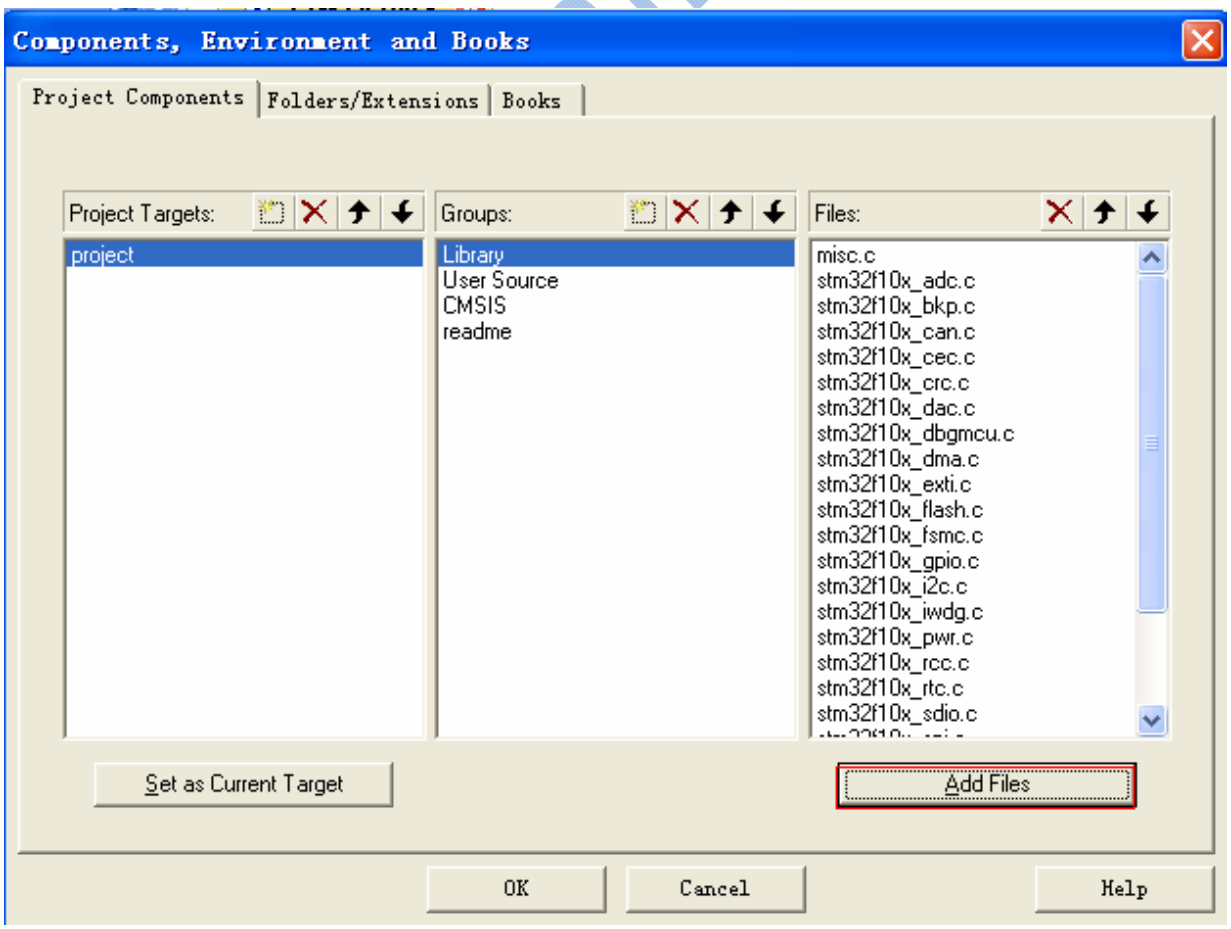


2. 在 Project Target 中把 Target1 改成你想要的名字，可以不改，这里改为 project，然后在 groups 单击新建按钮，这些组对应我们实实在在的文件夹，方便源文件的分类和管理，例如我们这里新建 User Source，Library，CMSIS，和 readme 四个组：

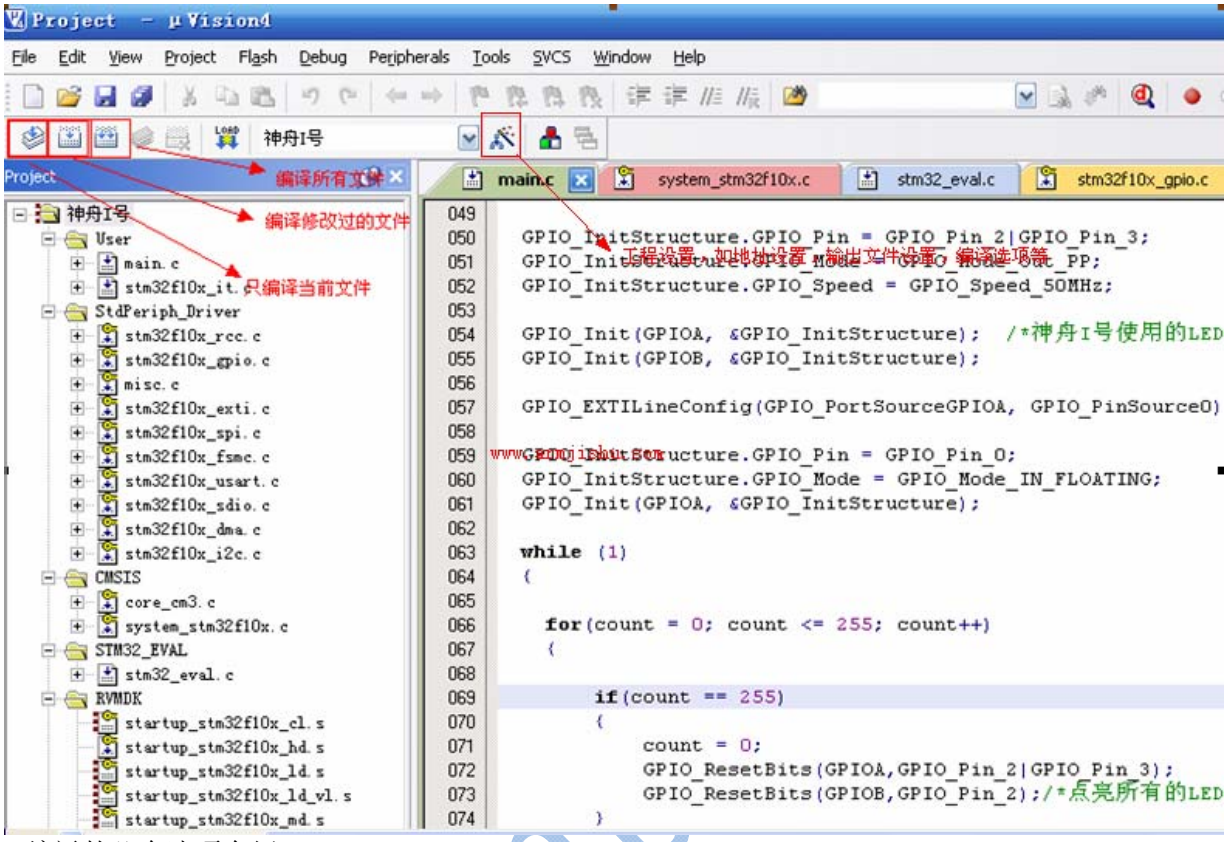


3. 先选中一个组之后，在 Files 点击 Add Files，例如在 User Source 组添加 User 文件夹中的源文件，添加完成该文件就成功添加到工程中了：


Components, Environment and Books
Project Components
Folders/Extensions
Books
Project Targets:
project
Groups:
Library
User Source
CMSIS
readme
Files:
main.c
platform_config.h
readme.txt
stm32f10x_conf.h
stm32f10x_it.c
stm32f10x_it.h
STM32F10x.s
Set as Current Target
Add Files
OK
Cancel
Help


Components, Environment and Books
Project Components
Folders/Extensions
Books
Project Targets:
project
Groups:
Library
User Source
CMSIS
readme
Files:
misc.c
stm32f10x_adc.c
stm32f10x_bkp.c
stm32f10x_can.c
stm32f10x_cec.c
stm32f10x_crc.c
stm32f10x_dac.c
stm32f10x_dbgmcu.c
stm32f10x_dma.c
stm32f10x_exti.c
stm32f10x_flash.c
stm32f10x_fsmc.c
stm32f10x_gpio.c
stm32f10x_i2c.c
stm32f10x_iwdg.c
stm32f10x_pwr.c
stm32f10x_rcc.c
stm32f10x_rtc.c
stm32f10x_sdio.c
Set as Current Target
Add Files
OK
Cancel
Help

4.4.5 编译和连接工程



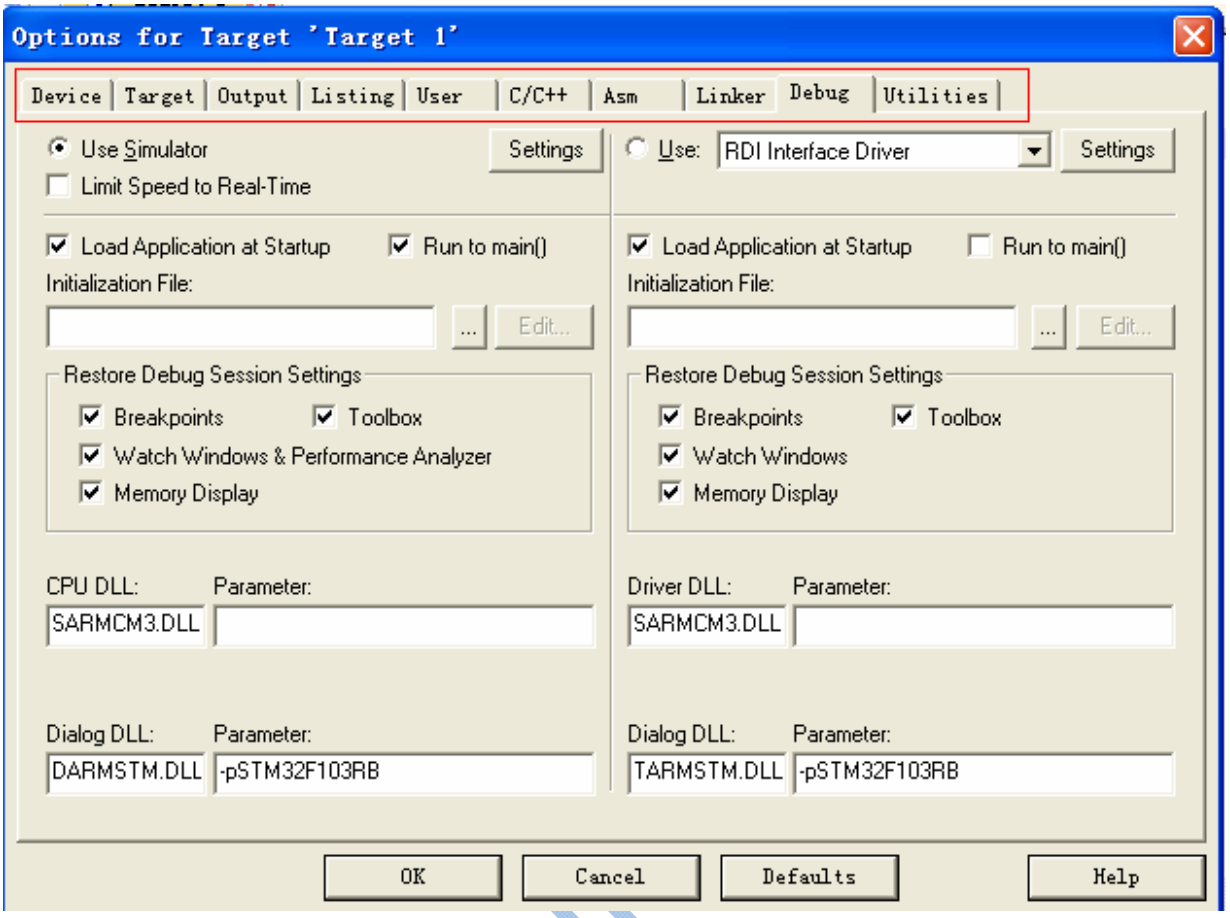
1.编译的几个选项介绍：

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：选择这个按键将将重新编译链接整个工程文件。

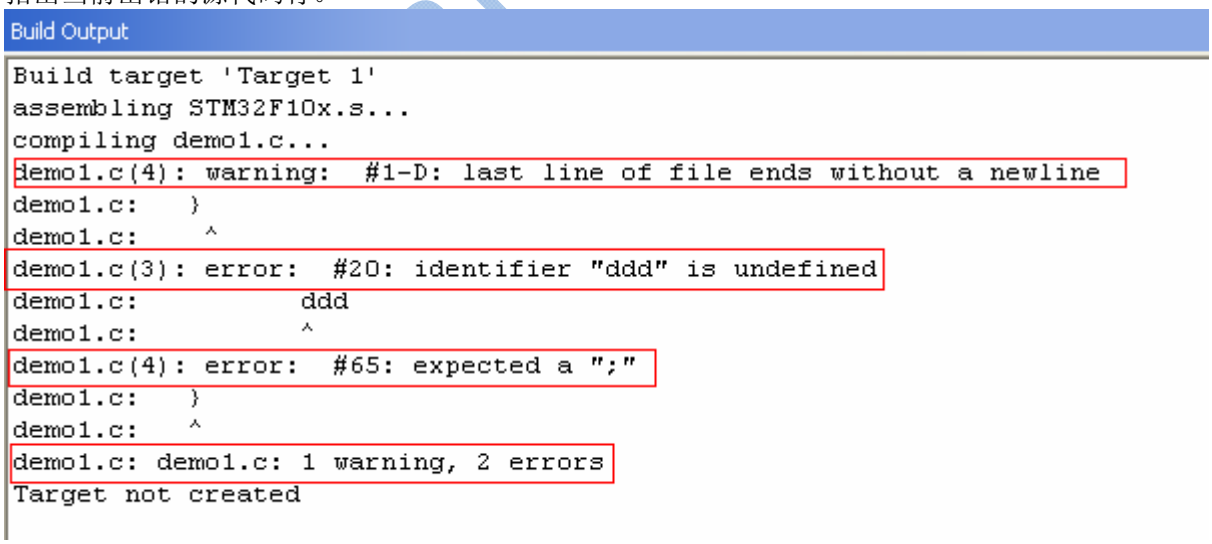
4 关于工程设置框：

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，我们可以在这里面根据需要进行一系列详细的设置。



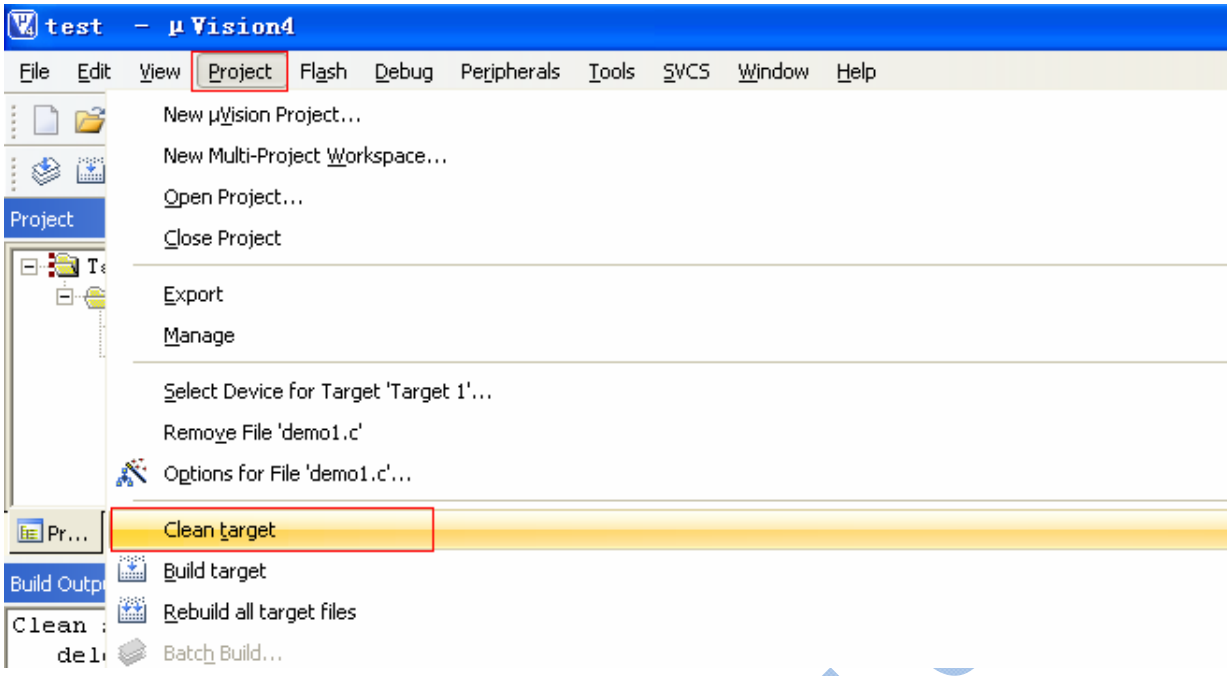
5　调试时输出的错误：

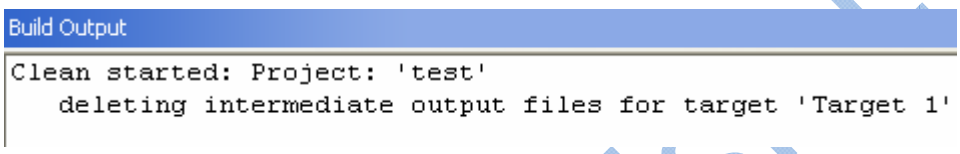
对于简单的软件调试，可以不进行连接地址的设置，直接点击工程窗口的“build”图标按钮，即可完成编译连接。若编译出错，会有相应的出错提示，双击出错提示行信息，编辑窗即会使用光标指出当前出错的源代码行。



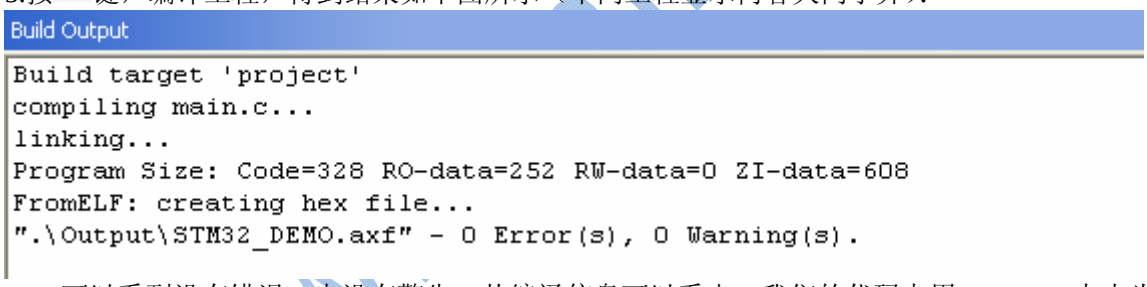
4.清除编译之后的垃圾：

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）

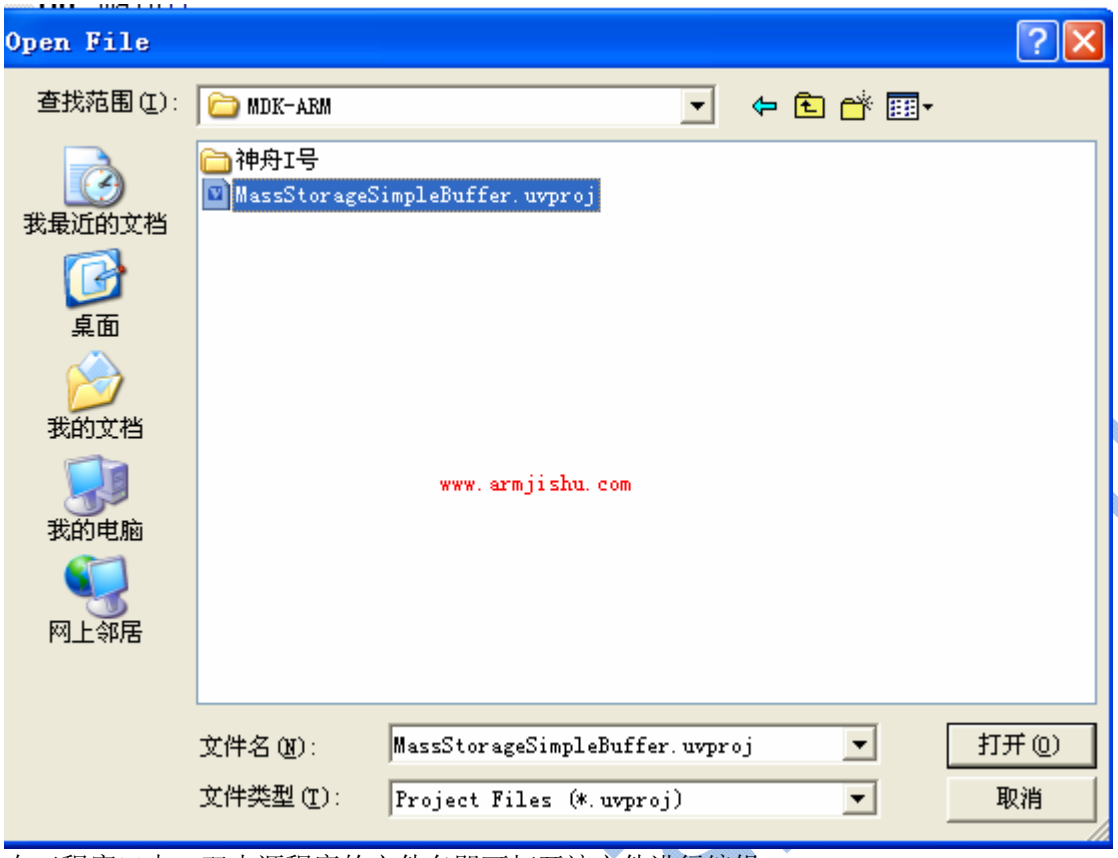
RW-data：即 Read Write-data，表示已可以读写的变量（SRAM）

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

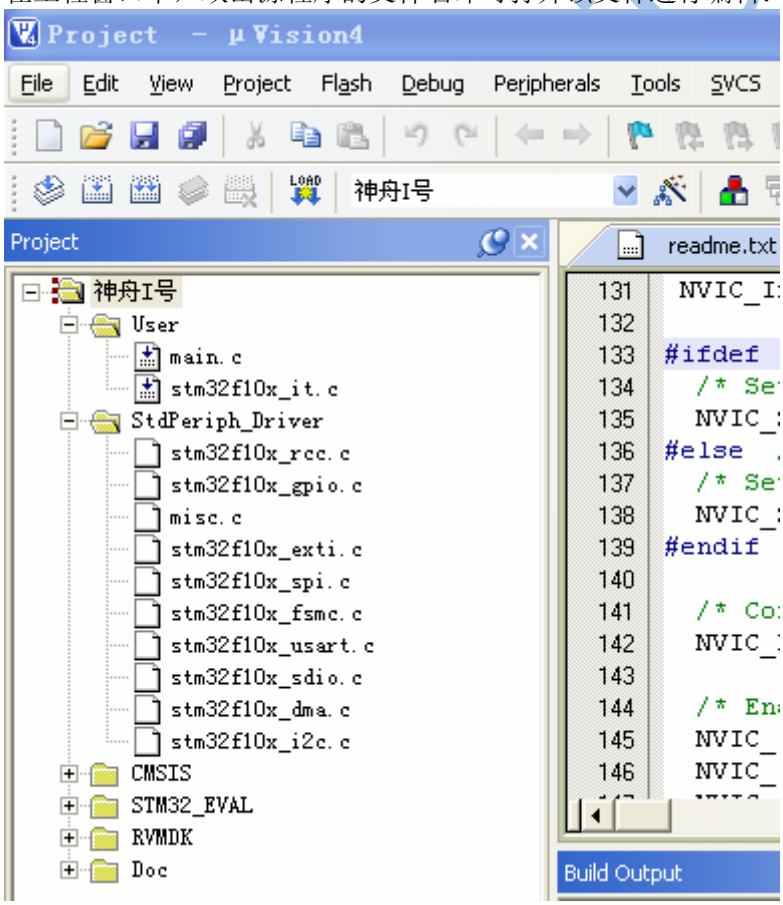
有了这个就可以知道当前使用的 flash 和 sram 大小了，所以，大家不要误解程序的大小就是.hex 文件的大小，这是不对的，程序的大小是编译后的 Code 和 RO-data 之和。

4.4.6 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：



4.5 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。

2.3.1. 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 ：

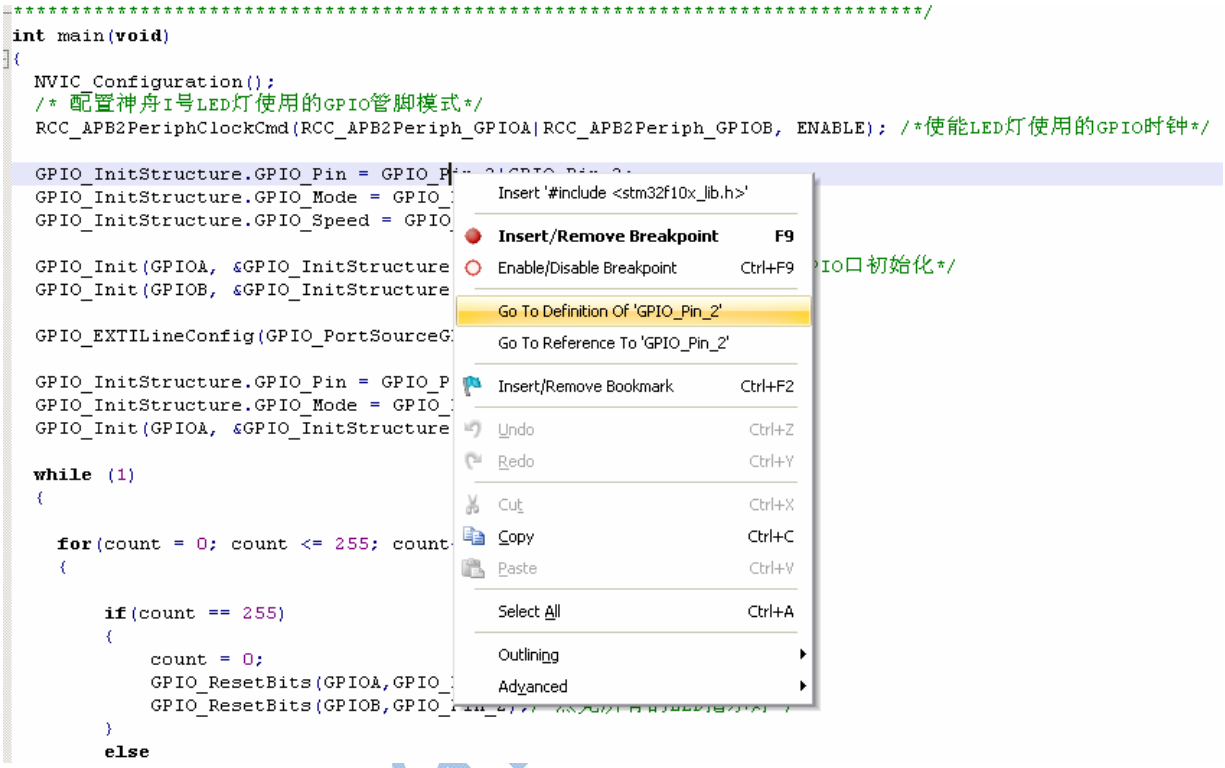
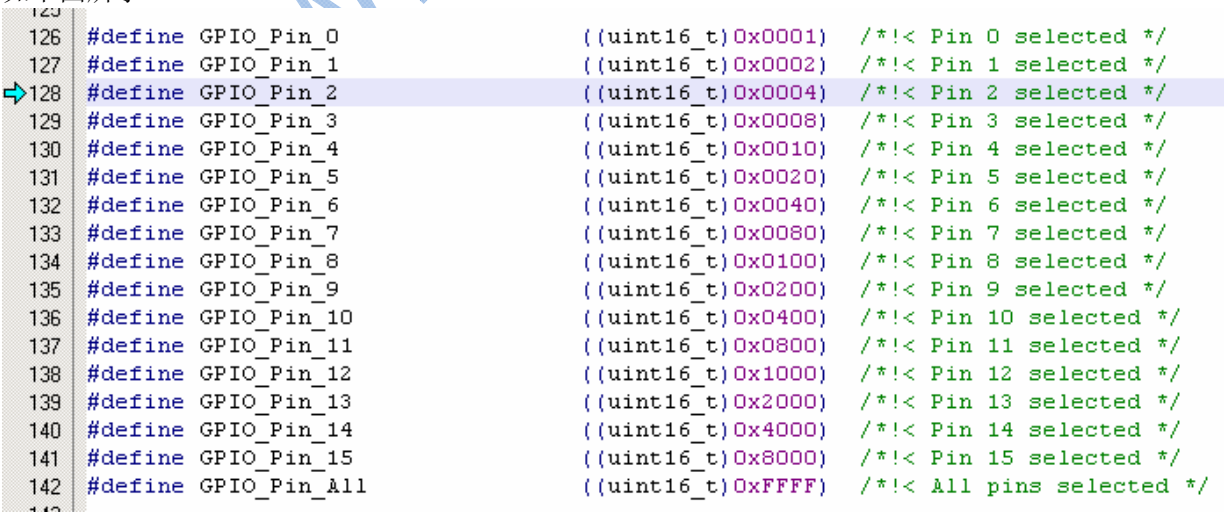


图 2.8.2.3 快速定位

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



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。

2.3.2. 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;
```

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           break;
```

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。

2.3.3. 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：

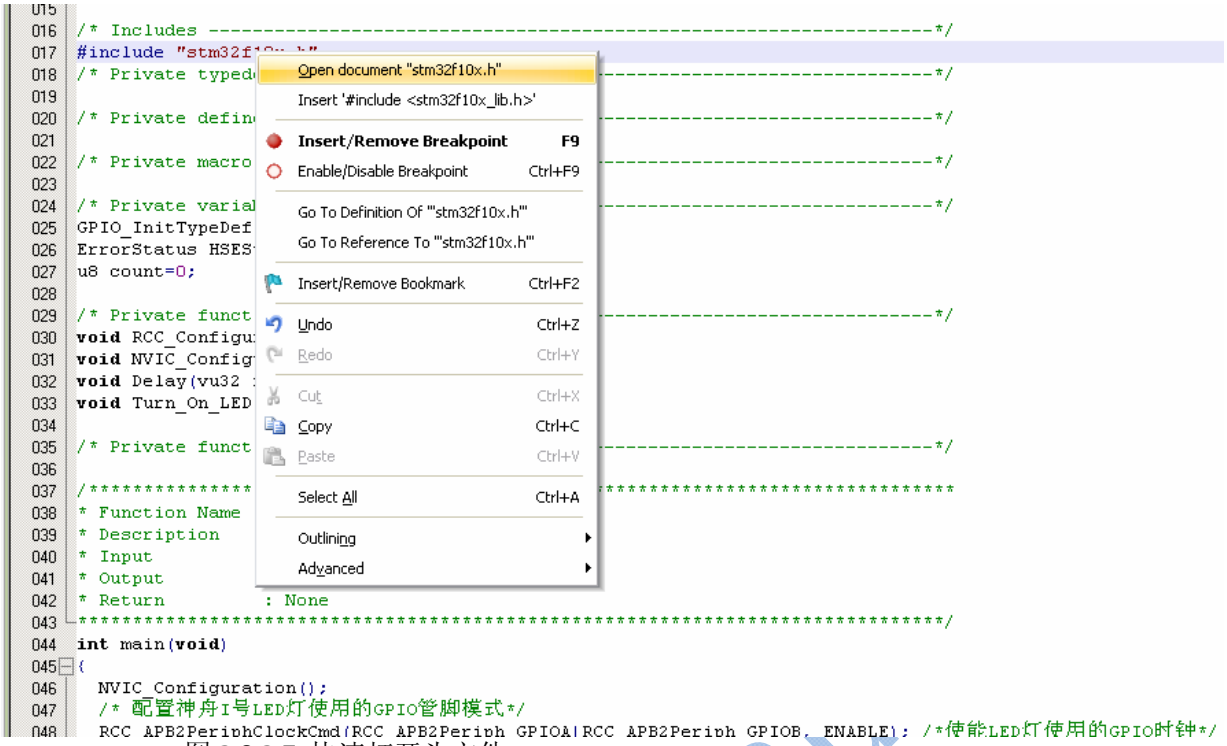


图 2.8.2.7 快速打开头文件

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。

4.6 JLINK V8仿真器的安装与应用

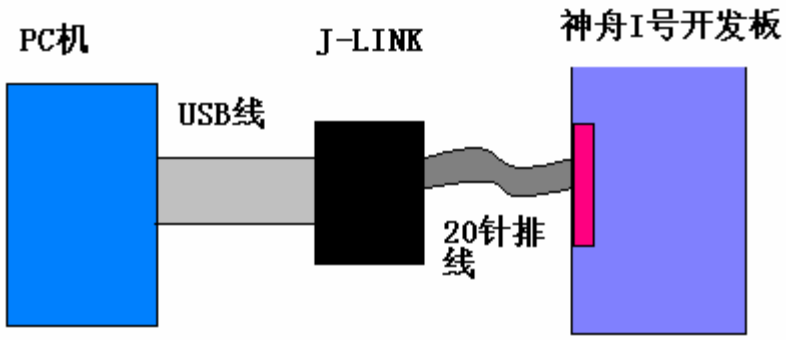
在代码编译通过后，我们需要验证程序是否与我们的设计预期相符合，一般有几种方式来验证。

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 I 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 I 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 I 号开发板。并给神舟 I 号上电。



4.6.1 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。

4.6.2 JLINK ARM主要特点

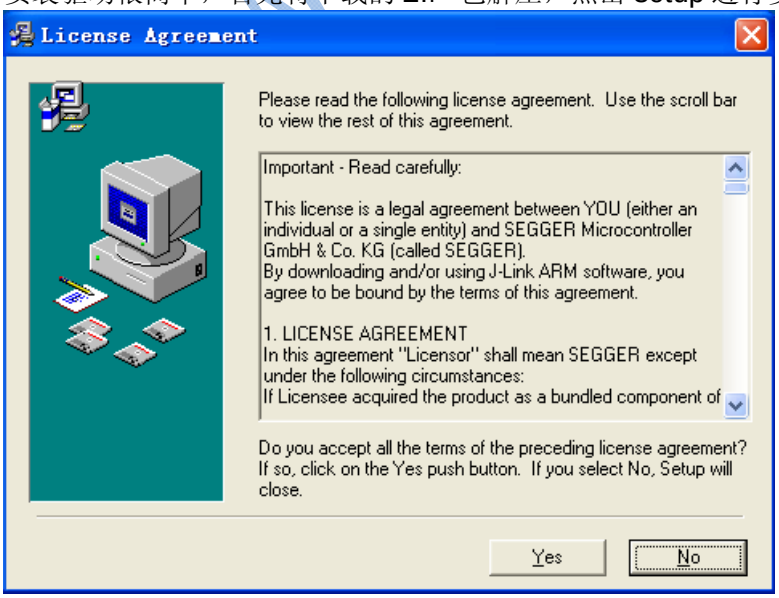
- ✧ USB 接口符合 USB2.0 规范
- ✧ 标准 20 芯 JTAG 接口
- ✧ 支持全系列 ARM 7/9/11，Cortex_M0/M1/M3 ARM 核，包括 Thumb 模式
- ✧ IAR EWARM 集成开发环境无缝连接的 JTAG 仿真器
- ✧ USB 接口供电，无需外接电源
- ✧ J-LINK 支持对目标板 5V（300mA），3.3V(400mA)供电
- ✧ 带 USB 连接线和 20 芯扁平电缆
- ✧ 支持 RDI 接口，J-LINK 可用于具有 RDI 接口的开发环境，支持主流的开发环境，包括 ADS,IAR,KEIL,WINARM,REALVIEW 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

4.6.3 JLINK V8仿真器安装

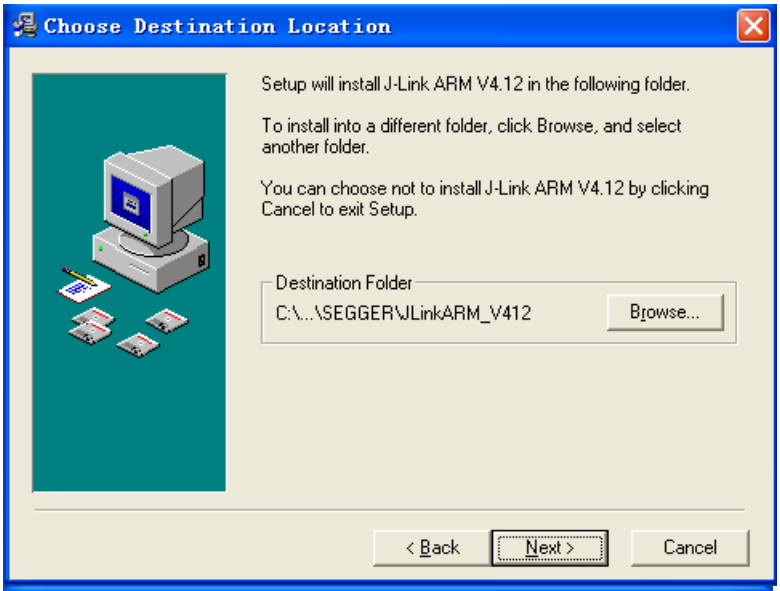
首先到http://www.segger.com/cms/jlink-software.html下载最新的J-LINK 驱动软件，J-LINK ARM software and documentation pack ，内含USB driver, J-Mem, J-LINK.exe and DLL for ARM, J-Flash and J-LINK RDI。

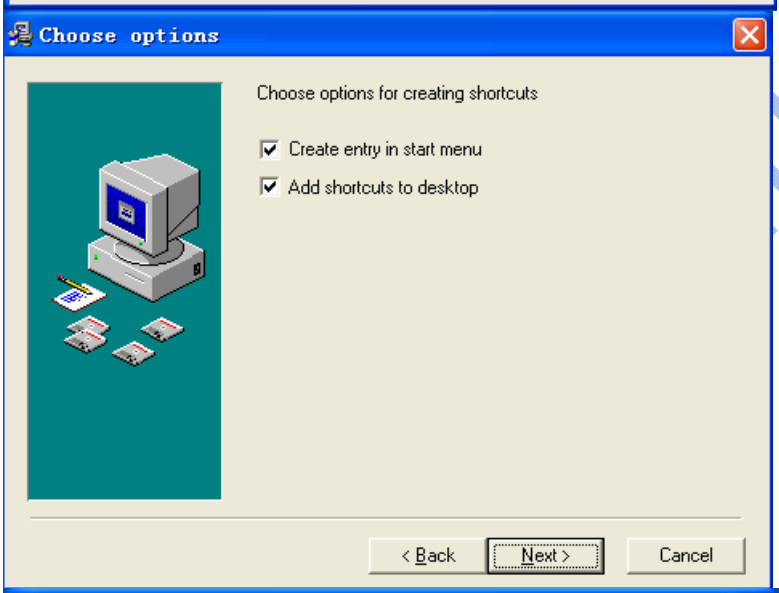
注意：SEGGER 公司升级比较频繁，请密切留意 SEGGER 公司网站，下载最新驱动，以支持更多器件！

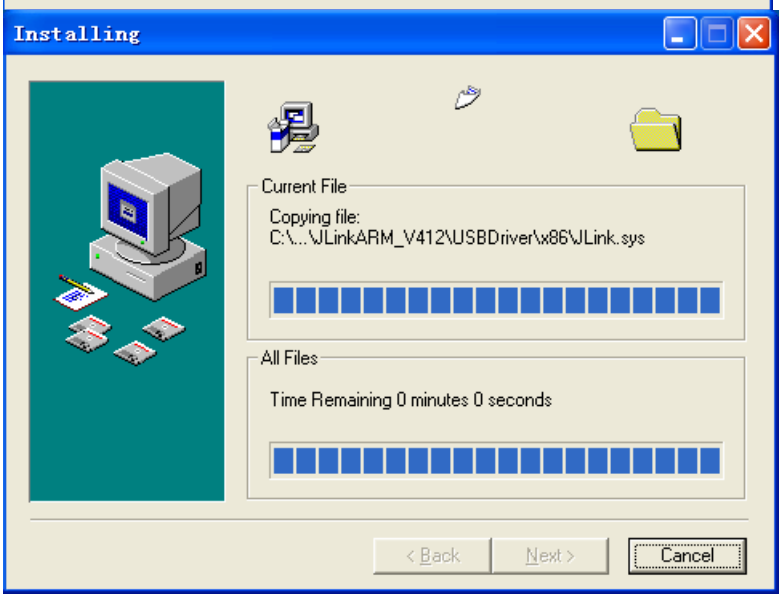
安装驱动很简单，首先将下载的 ZIP 包解压，点击 Setup 进行安装。



点击 YES 同意安装，后续选择 NEXT，缺省设置安装即可。





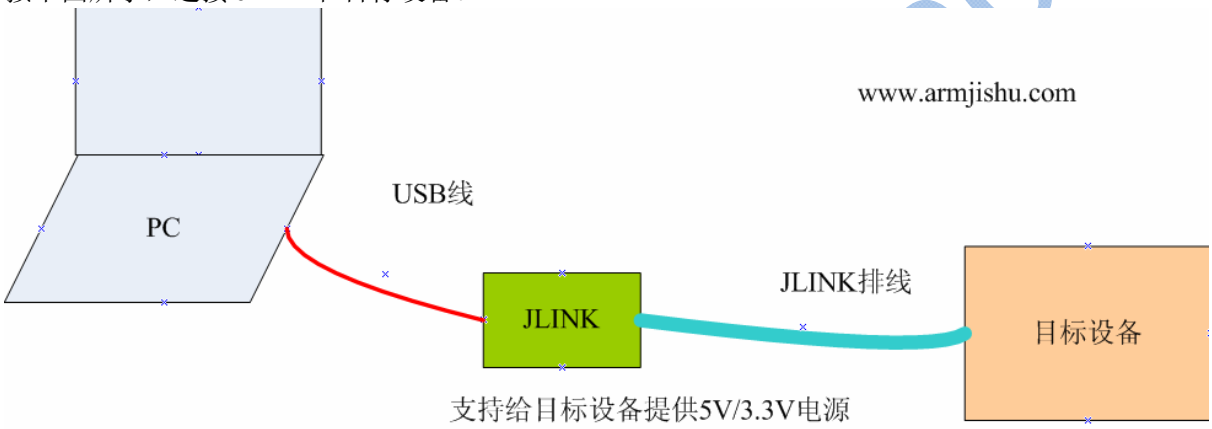


安装完成后，桌面上可以看到如下两个图标。

安装完成后，请插入 J-LINK 硬件，然后系统提示发现新硬件，一般情况下会自动安装驱动，如果没有自动安装，请选择手动指定驱动程序位置（安装目录），然后将驱动程序位置指向到 J-LINK 驱动软件的安装目录下的 Driver 文件夹，驱动程序就在改文件夹下。

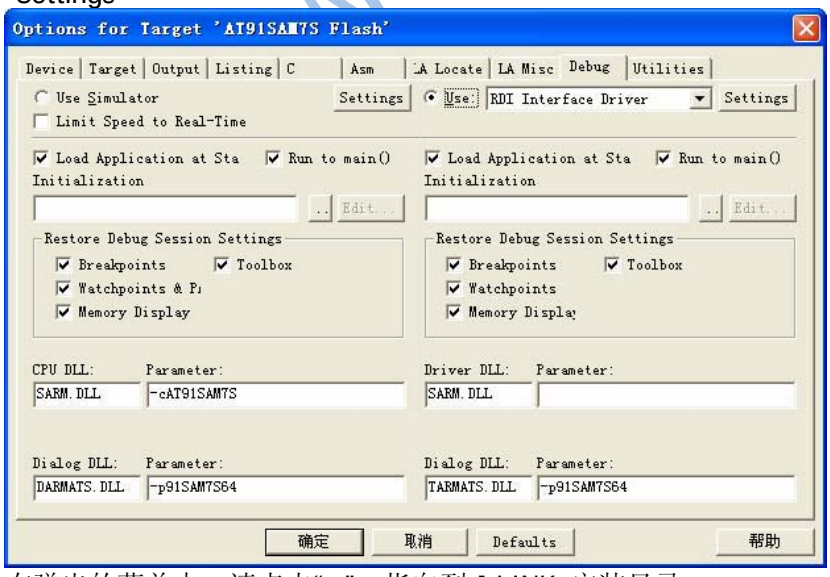
4.6.4 JLINK V8仿真器配置（MDK KEIL环境）

按下图所示，连接 JLINK 和目标设备。

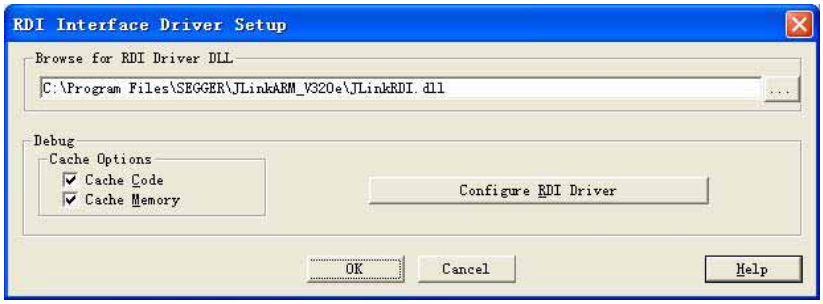


(i) 使用J-LINK进行DEBUG的设置

打开 KEIL 工程后，选择项目设置的 Debug 菜单。如下图选择“RDI Interface Driver”，点击“Settings”

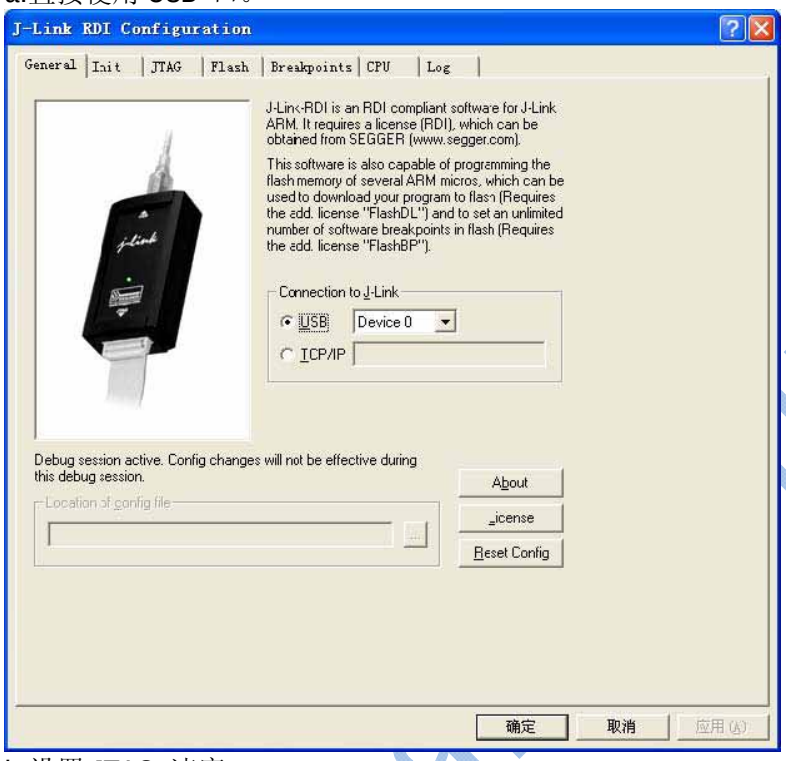


在弹出的菜单中，请点击“...”，指向到 J-LINK 安装目录。



点击“Configure RDI Driver”出现以下几个选项卡，按如下进行设置。

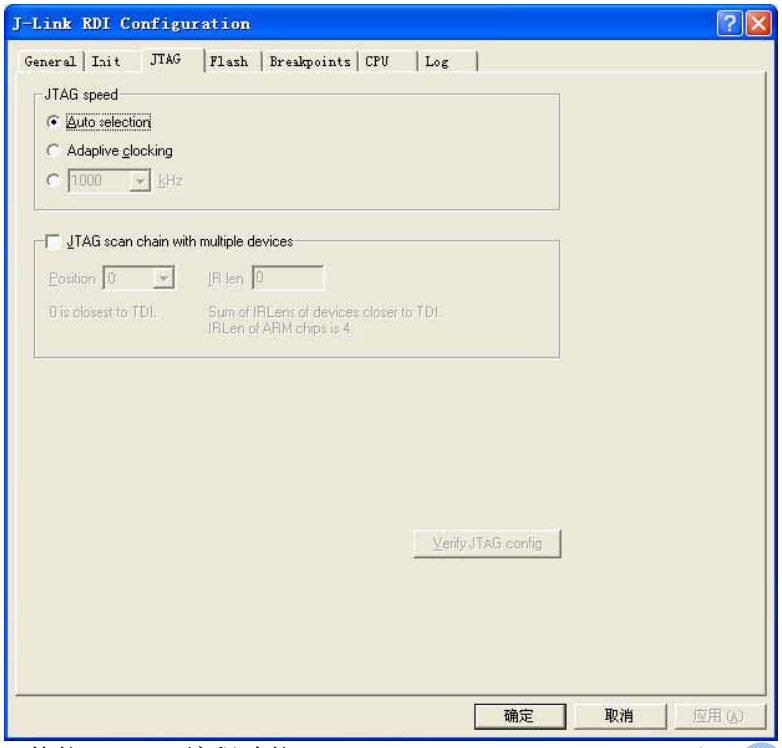
a.直接使用 USB 口。



b.设置 JTAG 速度。

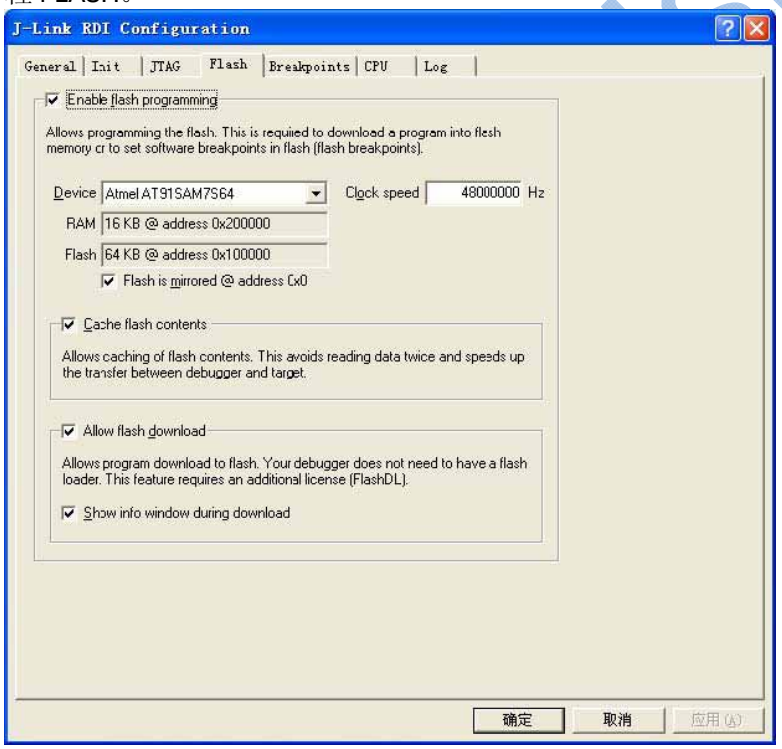
如果是-S 内核，建议使用 Auto 方式，如果是非-S 内核，可以直接使用最高速度 12M。使用过程中如果出现不稳定情况，可以将 JTAG 时钟速度适当调低。

www.armjishu.com



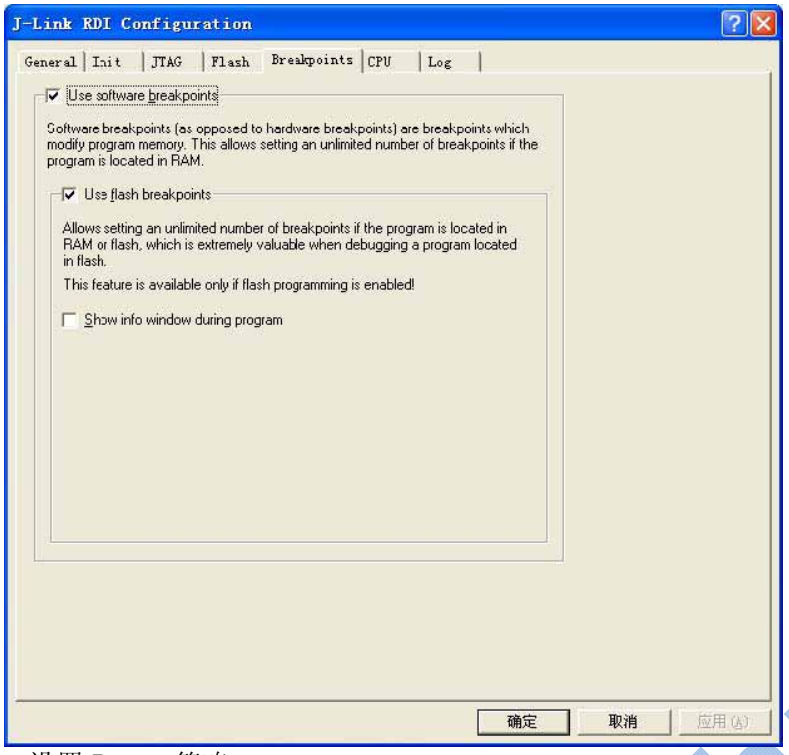
c.使能 FLASH 编程功能。

如果你的目标芯片是带片内 FLASH 的 ARM， 就可以使用该功能，这样子在调试前 J-LINK 就会先编程 FLASH。



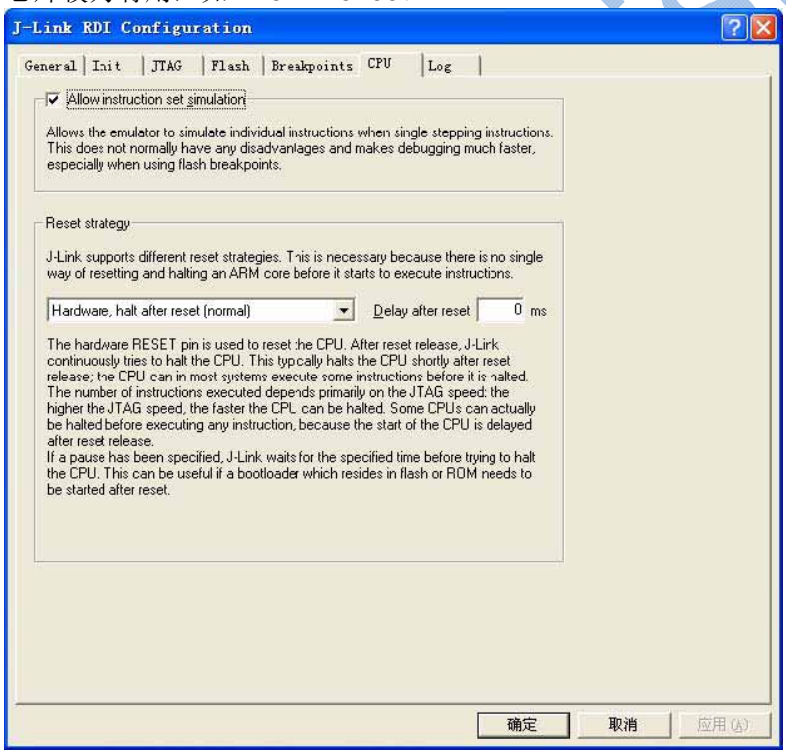
d.使用软件断点。

如果是带片内 FLASH 的 ARM，建议使用该功能，可以打上 n 多断点，方便调试。



e.设置 Reset 策略。

有好几种 Reset 策略可选，同时可以设置 Reset 后的延迟时间，这个设置对于需要较长复位时间的芯片较为有用，如 AT91RM9200。

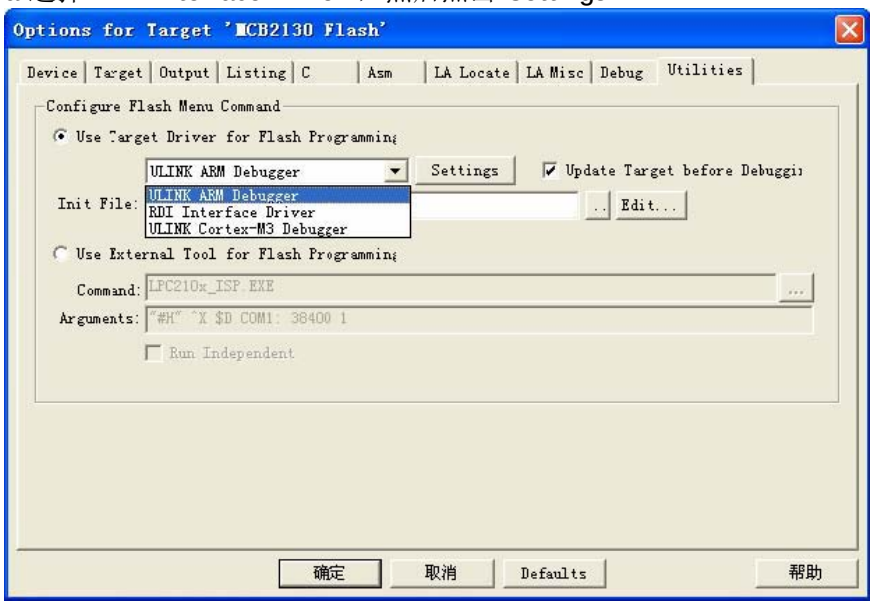


以上设置是用 J-LINK 进行 Debug 的设置。

(ii) 使用KEIL的DOWNLOAD功能

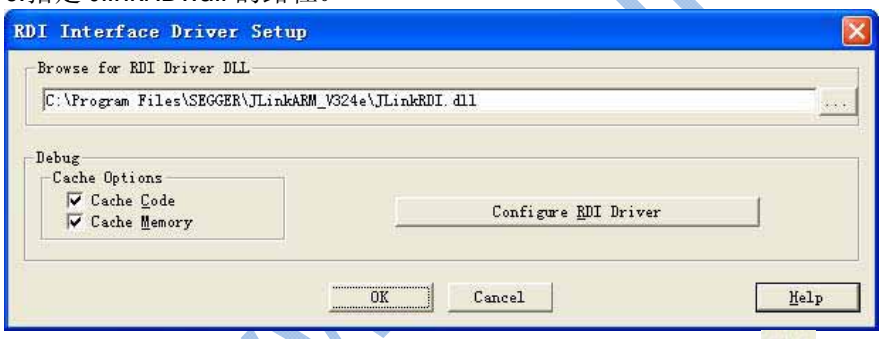
如果要使用KEIL 提供的 即“DOWNLOAD”功能则在完成前一步的设置外，还需要在“Utilities”菜单里面进行和“Debug”一样的设置：

a.选择“RDI Interface Driver”，然后点击“Settings”



b.选择“J-LINK Flash Programmer”

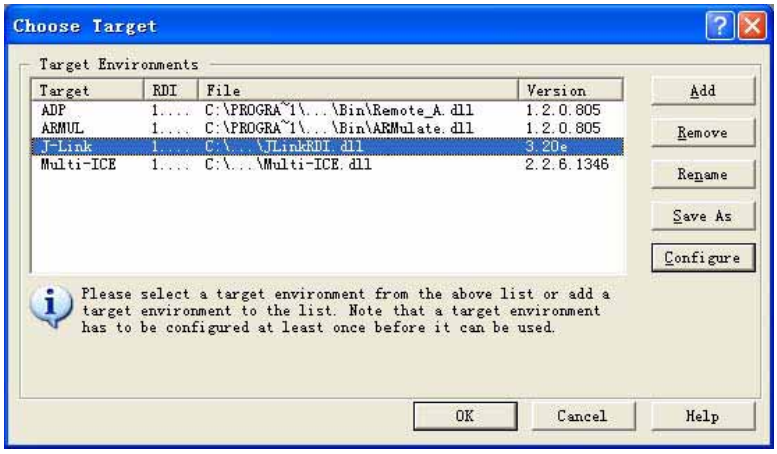
c.指定 JlinkRDI.dll 的路径。



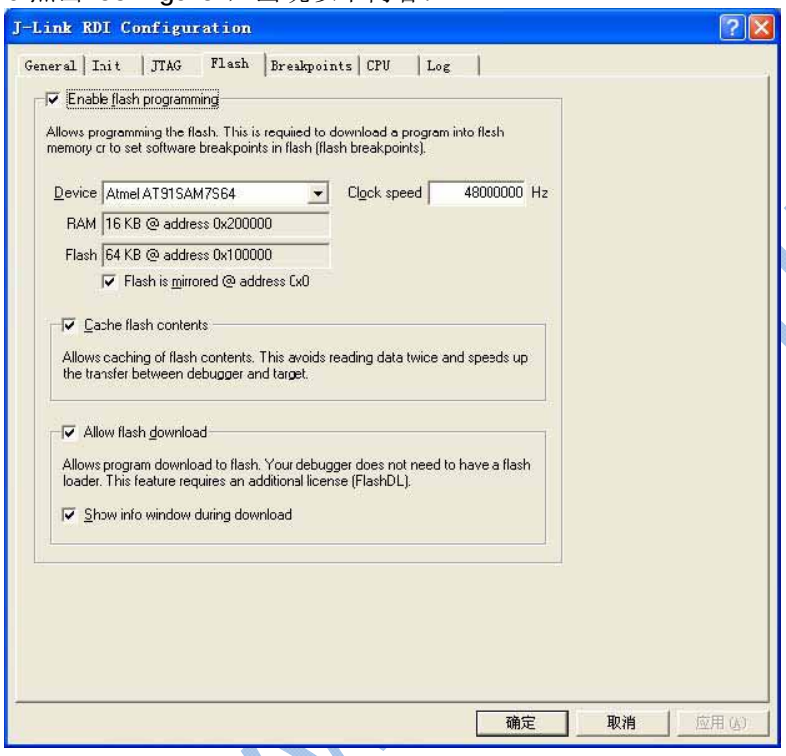
d. 按”Debug”菜单进行相同的设置。完成设置后，就可以通过 按钮进行直接下载。注意，该功能只支持具备片内 FLASH 的 ARM7/9 芯片。

(iii) 在ADS 下使用J-LINK的设置：

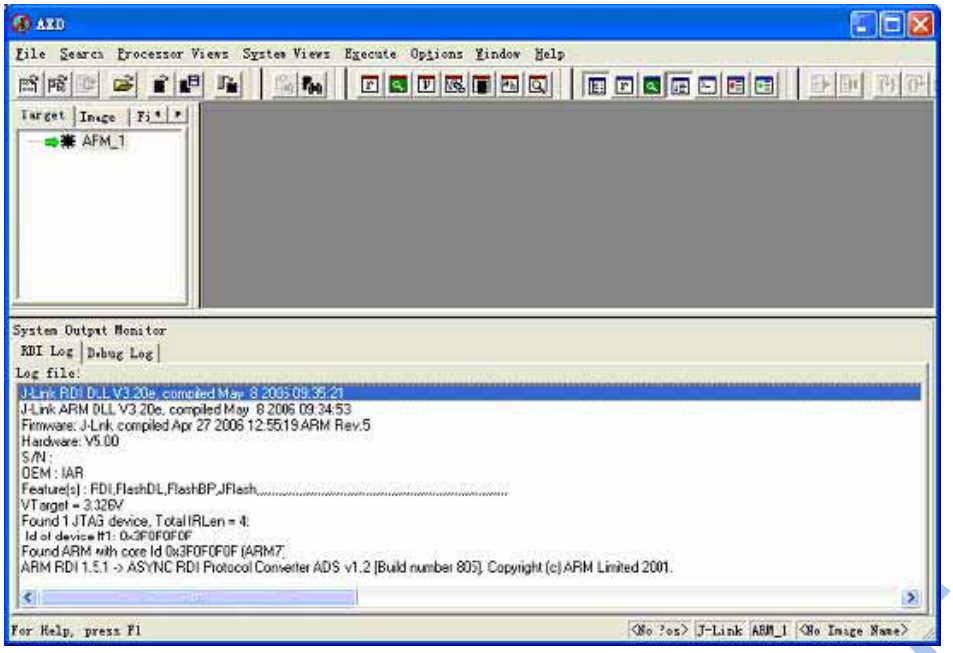
a.点击“Add”，选择 J-LINKRDI.DLL：



b.点击“Configure”，出现以下内容：



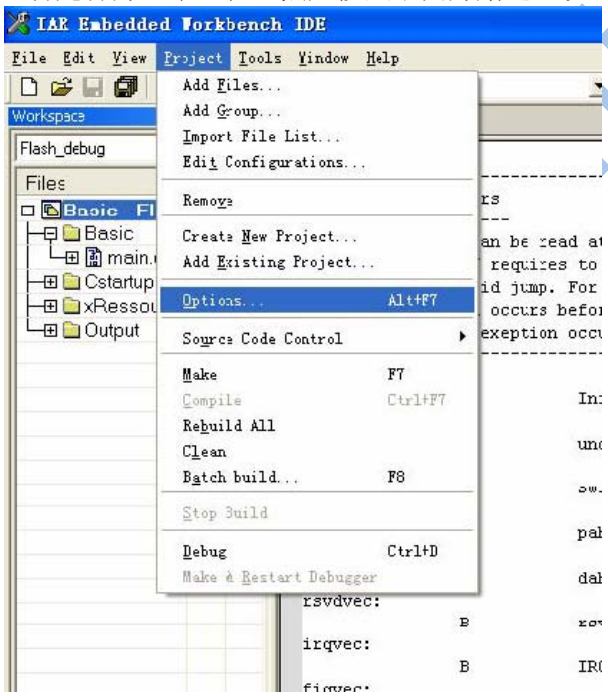
c.进入 AXD 后的信息(注意 LOG FILE 的内容)：

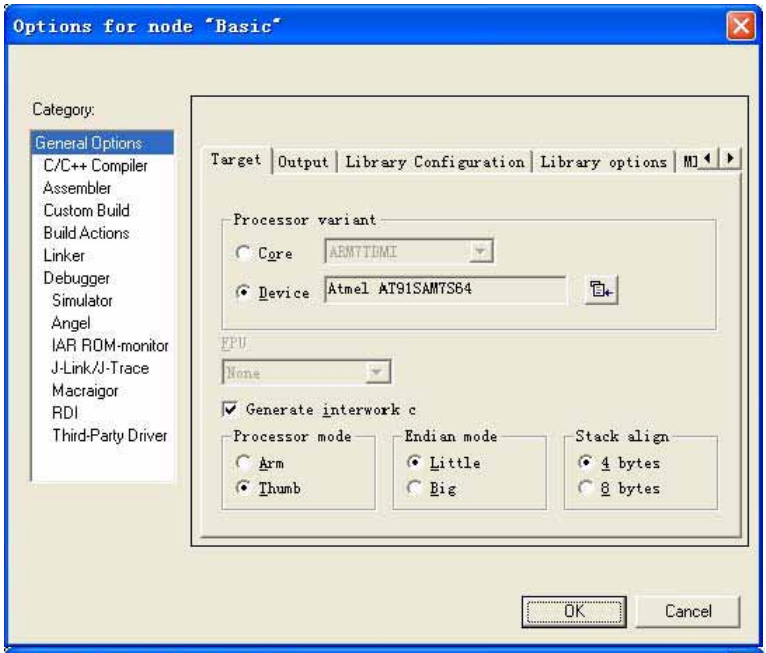


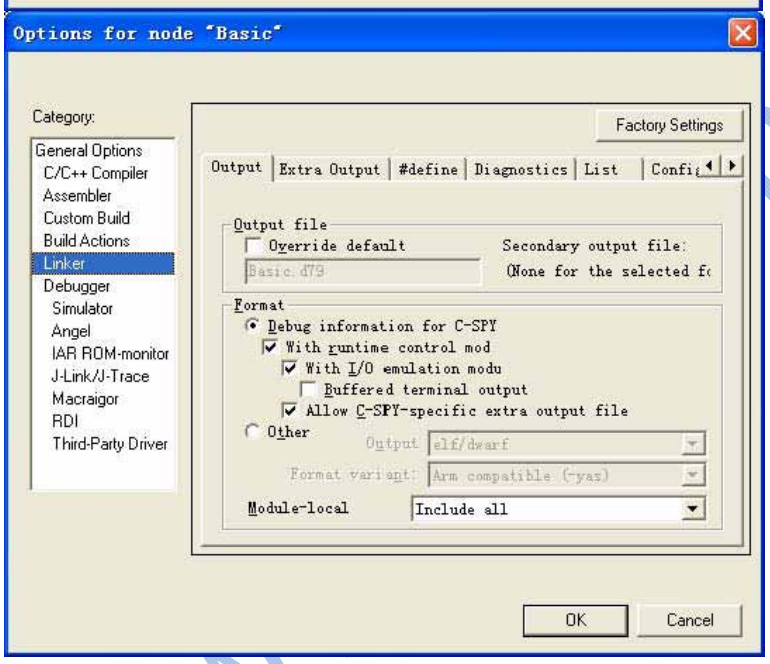
4.6.5 JLINK V8仿真器配置（IAR 环境）

在 IAR 既可以使用 IAR 提供的 J-LINK 的驱动，也可以使用 RDI 接口的驱动，推荐使用 RDI 接口的驱动，因为 IAR 版本的 J-LINK 对速度和功能做了限制。

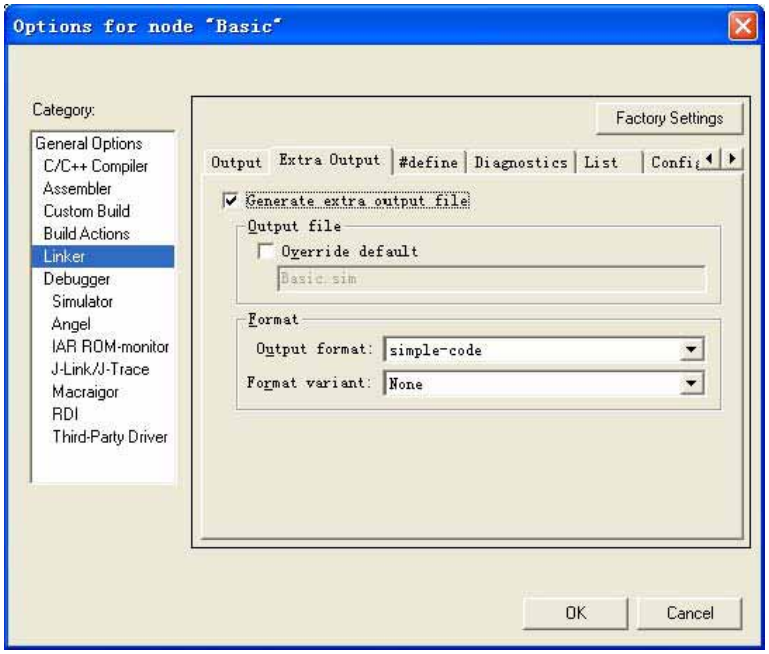
a.首先打开一个工程，然后按照下图开始进入设置页面：



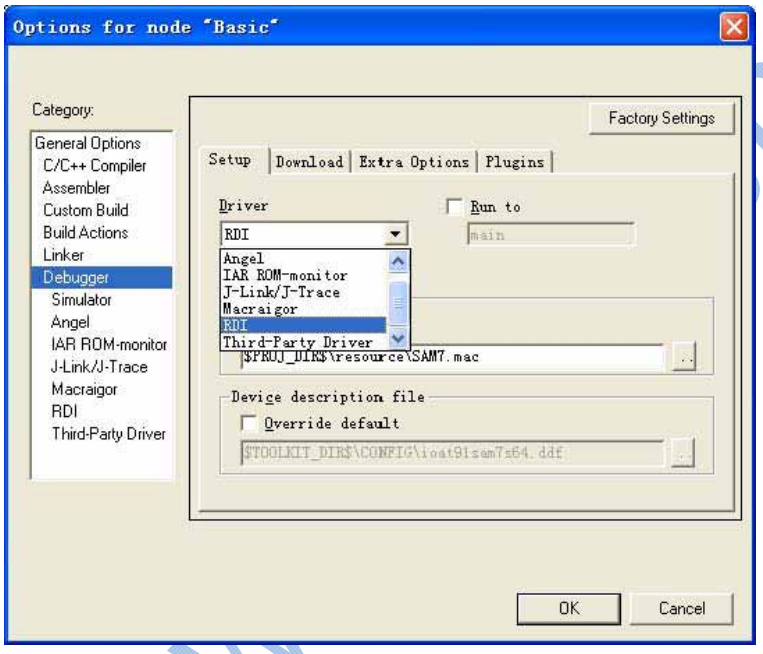




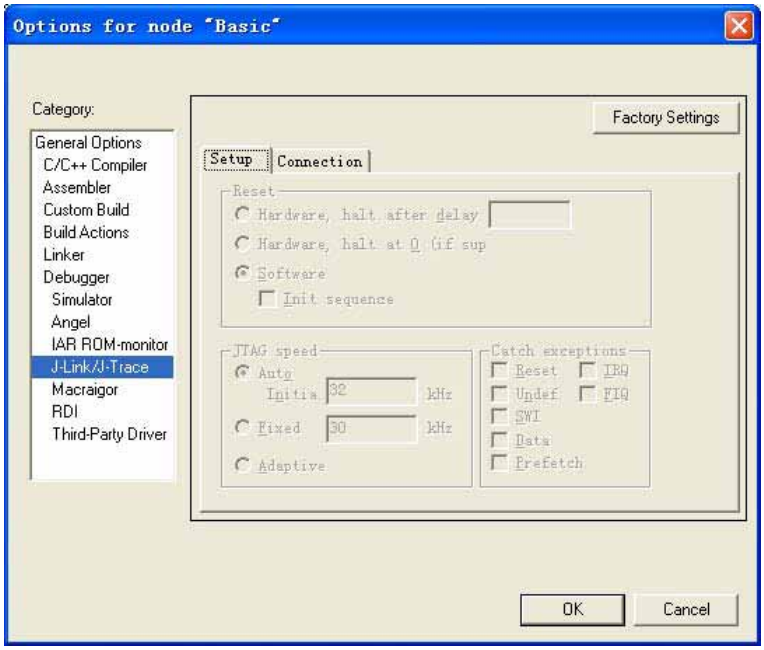
www.armjishu.com



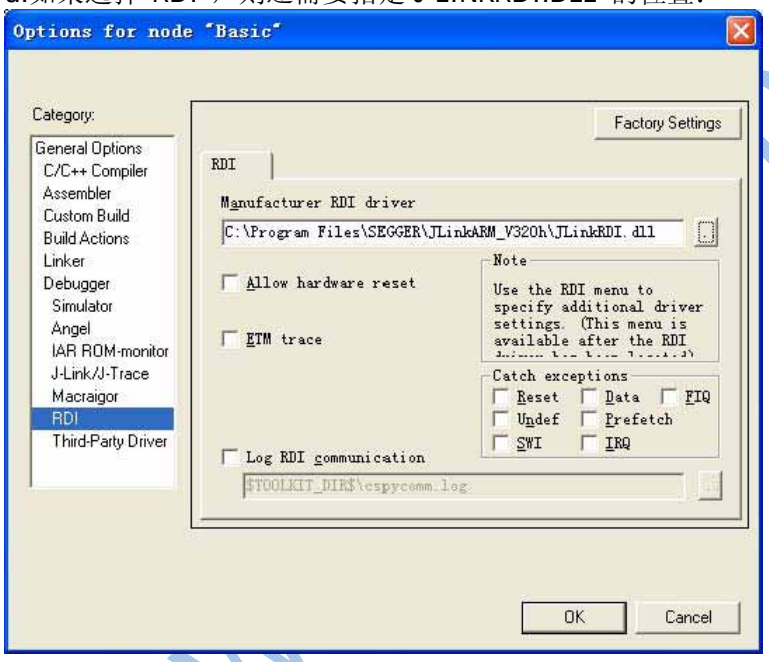
b.选择“J-Link/J-Trace”或“RDI”，建议选择“RDI”



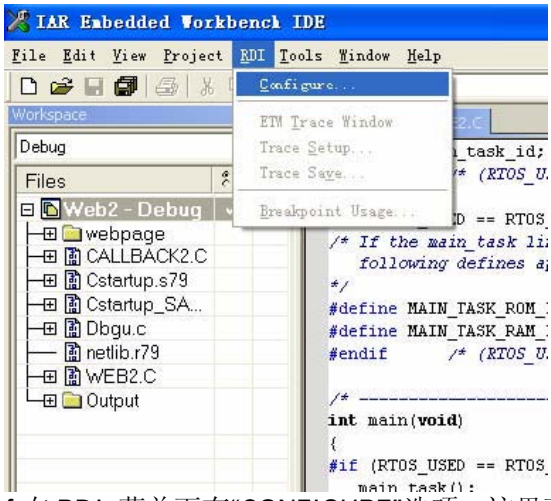
c.如果选择“J-LINK/J-TRACE”，则无需额外设置：



d.如果选择"RDI"，则还需要指定 J-LINKRDI.DLL 的位置：



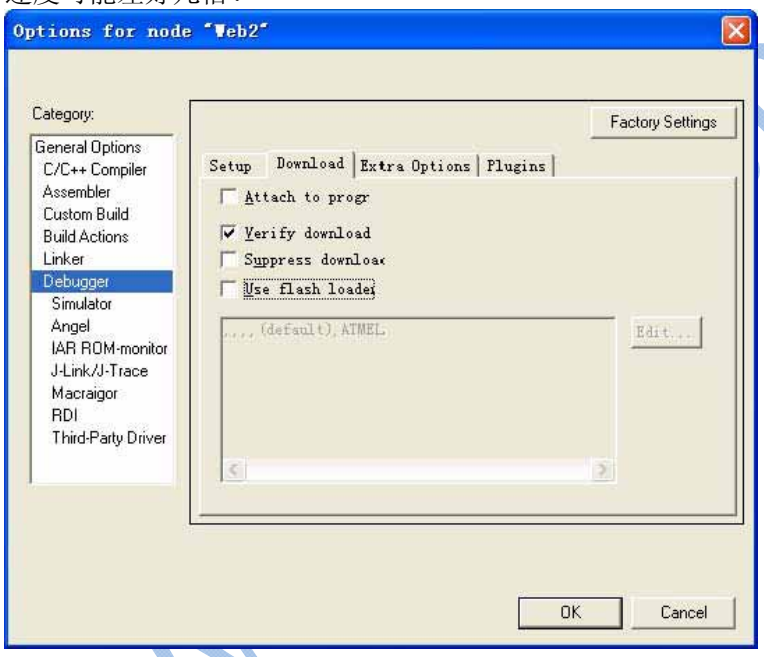
e.设置完成后将多出一个 RDI 菜单，如下图：



f.在 RDI 菜单下有“CONFIGURE”选项，这里可以对 JTAG 时钟，FLASH，断点，CPU 等进行设置，请注意里面的 FLASH 和 CPU 型号与目标板相吻合。

g.另外，IAR 下使用 J-LINK 的时候，注意不要再使用 IAR 自带的 FLASHLOADER 软件进行 FLASH 下载：

将“Use flash loader”前的勾去掉，使用 J-LINK 的 FLASH 编程算法和使用 IAR 的 FLASHLOADER ，速度可能差好几倍！



4.6.6 J-FLASH如何烧写固件到芯片FLASH里

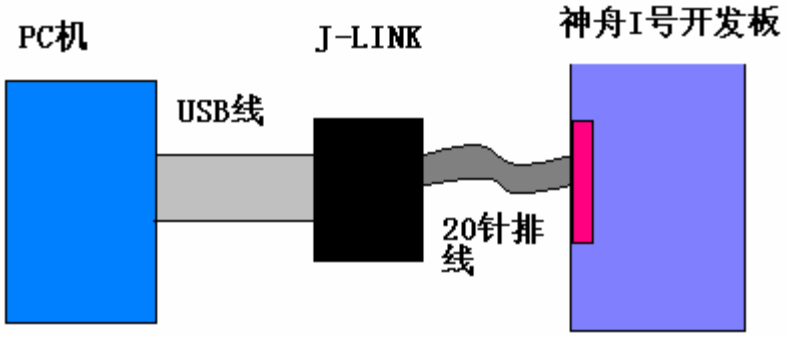
STM32的程序下载有多种方法，可以通过USB、串口、JTAG、SWD等方式下载。这几种方式都可以用来给神舟I号开发板下载程序，这里详细介绍通过JLINK 仿真器下载固件到神舟I号开发板的过程。

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I号开发板与JLINK V8搭配使用。

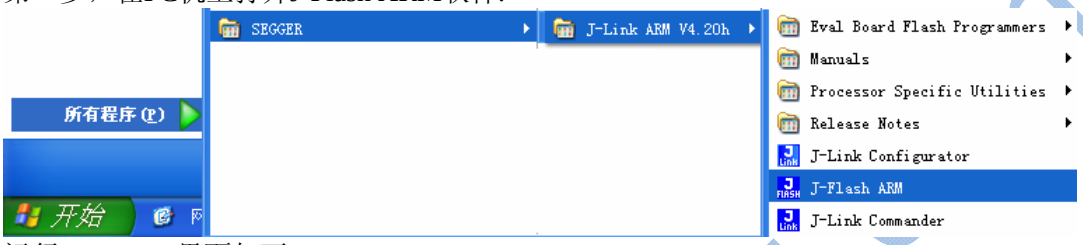
神舟I号提供的官方资源，包括源代码，使用的开发环境为MDK 4.12；使用的仿真调试工具为JLINK V8。

以下详细描述使用JLINK V8下载固件到神舟I号开发板的过程。

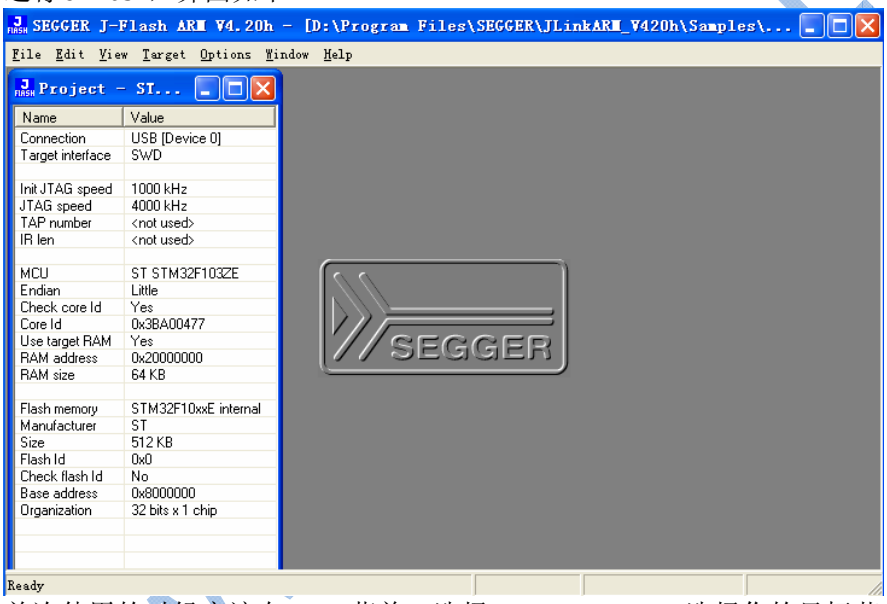
第一步，按下图连接好JLINK V8，神舟I号与PC机。



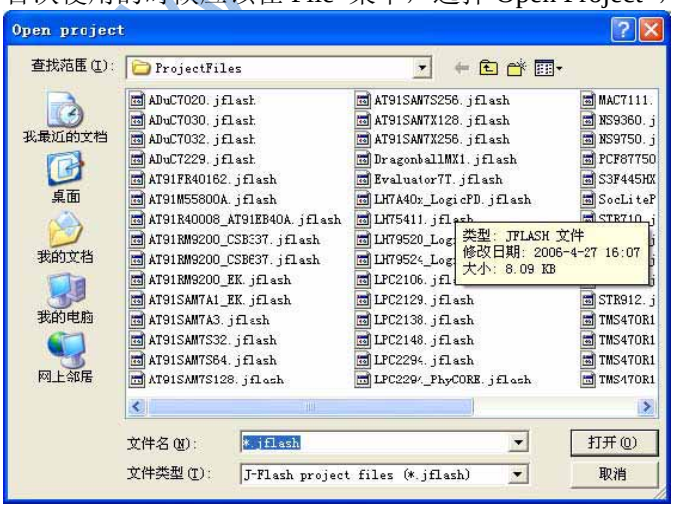
第二步，在PC机上打开J-Flash ARM软件.



运行 J-Flash，界面如下



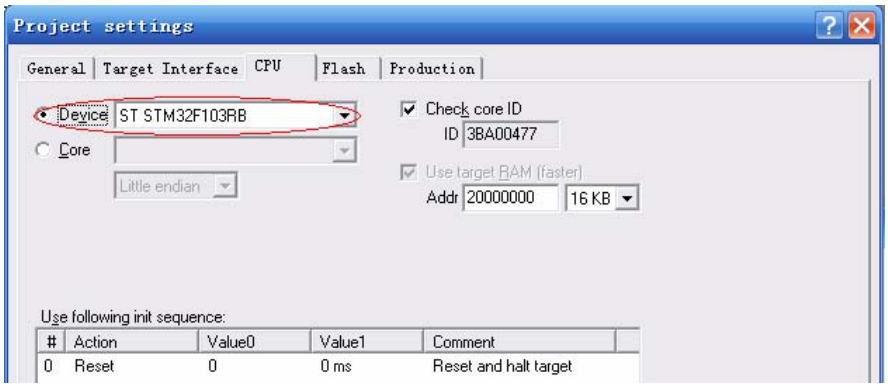
首次使用的时候应该在 File 菜单，选择 Open Project ，选择你的目标芯片：



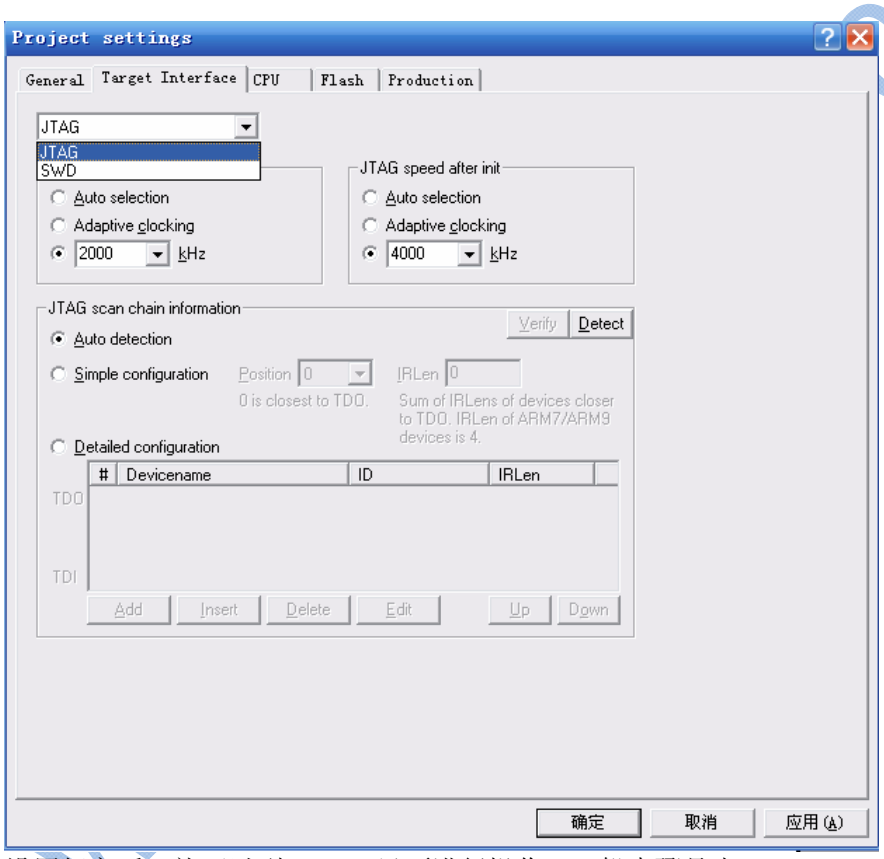
然后通过“File”菜单下的“Open…”来打开需要烧写的文件，可以是.bin 格式，也可以是.hex 格式，

甚至可以是.mot 格式。注意起始地址。

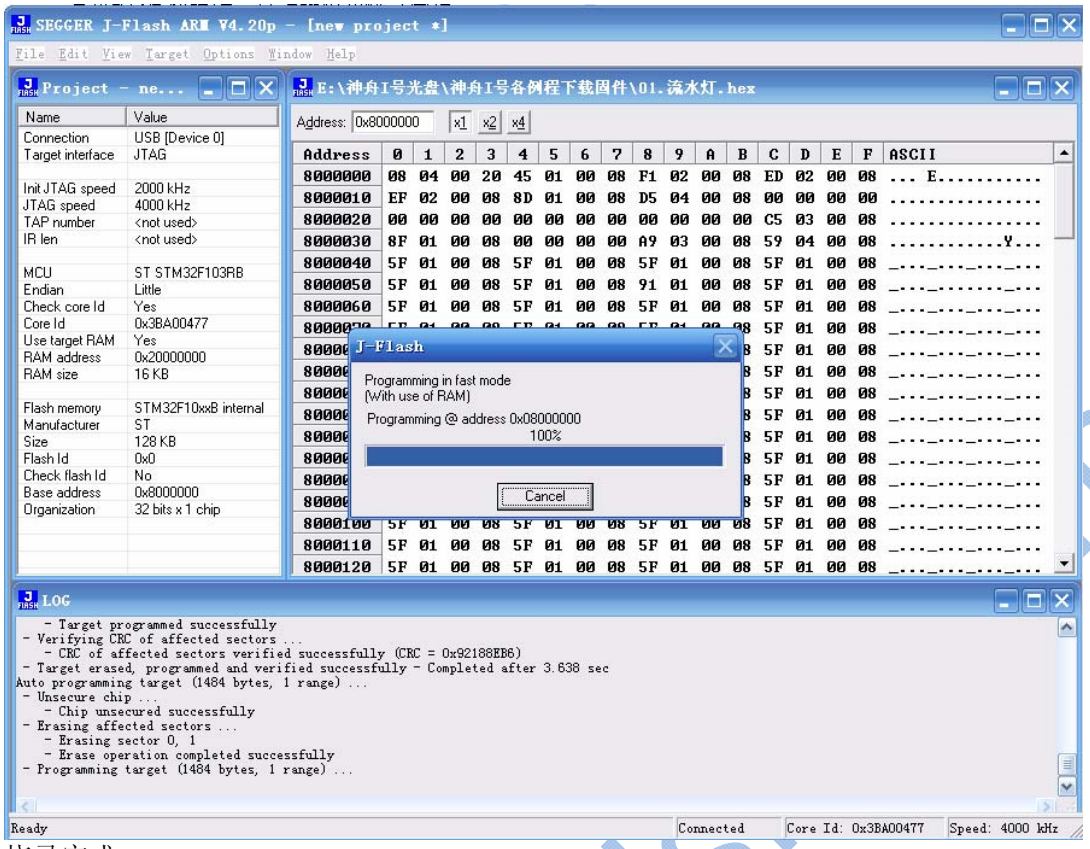
接下来在“Options”选择“Project settings”在弹出的对话框中，点击 CPU 标签，选择神舟 I 号使用的处理器 STM32F103RB。



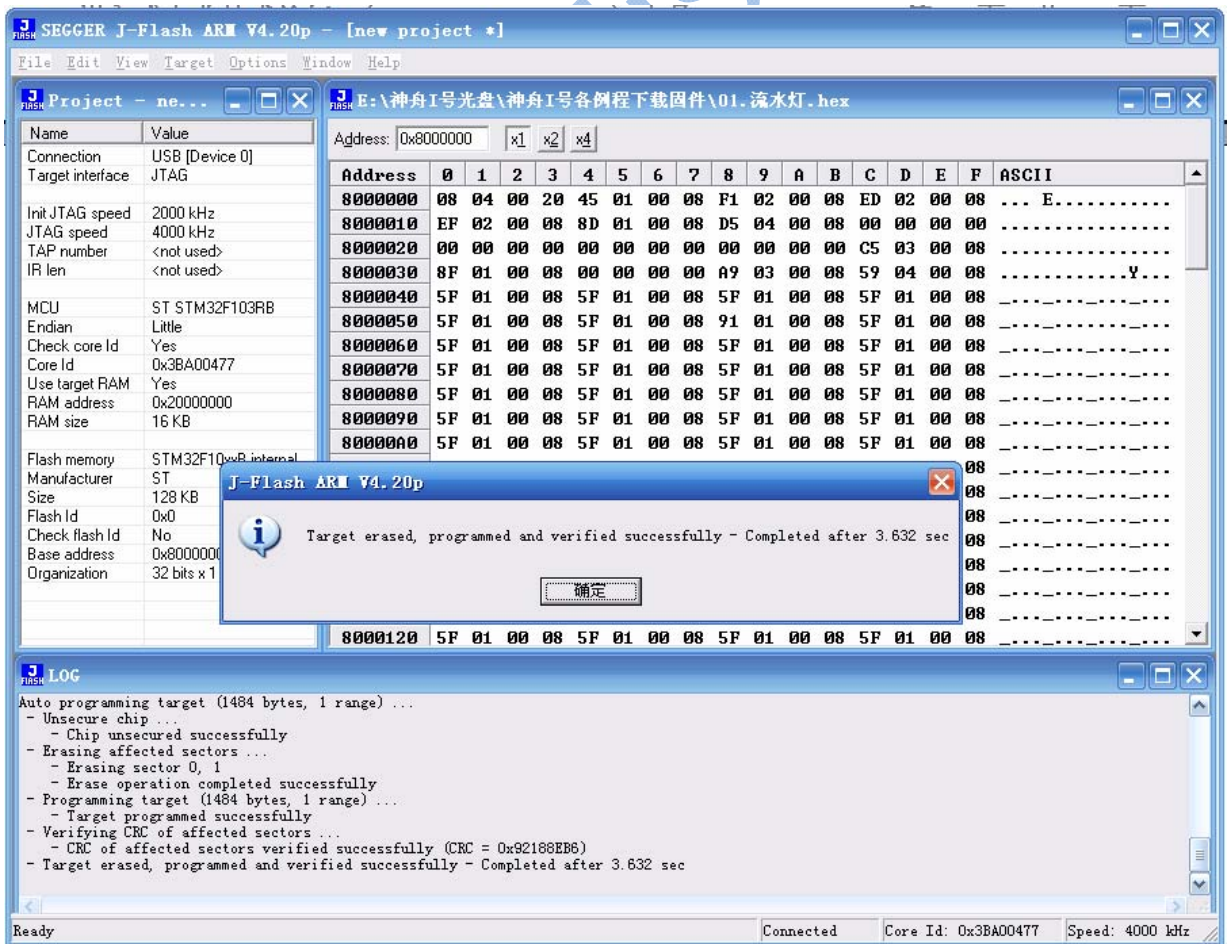
点击 Target Interface 选择使用 JTAG 还是 SWD 接口，JLINK V8 可以选择用 JTAG 或 SWD 接口来下载，在线调试，而这两种接口 STM32 处理器也都是支持的。



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。



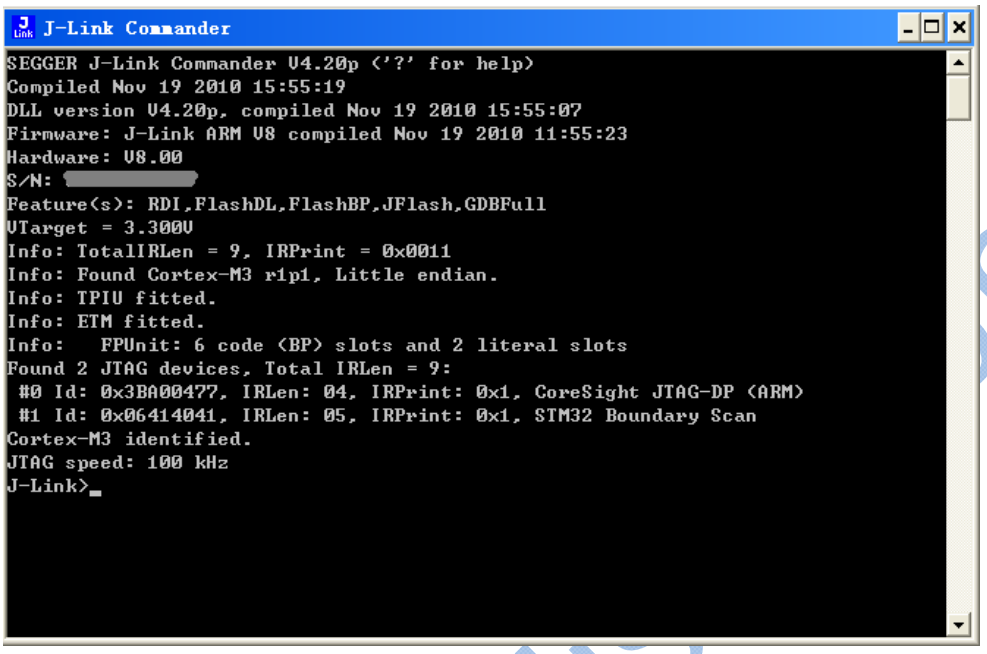
烧录完成



关于 J-FLASH ARM 更详细的操作请参阅 JLINK 的用户手册。

4.6.7 JLINK Commander工具查看相关信息

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 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。

www.armjishu.com

```
J-Link>?

Available commands are:
-----------------------
f           Firmware info
h           halt
g           go
Sleep       Waits the given time (in milliseconds). Syntax: Sleep <delay>
s           Single step the target chip
st          Show hardware status
hwinfo      Show hardware info
mem         Read memory.            Syntax: mem  <Addr>, <NumBytes> (hex)
w1          Write  8-bit items. Syntax: w1 <Addr>, <Data> (hex)
w2          Write 16-bit items. Syntax: w2 <Addr>, <Data> (hex)
w4          Write 32-bit items. Syntax: w4 <Addr>, <Data> (hex)
wm          Write test words. Syntax: wm <NumWords>
is          Identify length of scan chain select register
ms          Measure length of scan chain. Syntax: ms <Scan chain>
mr          Measure RTCK react time. Syntax: mr
q           Quit
qc          Close JLink connection and quit
r           Reset target          (RESET)
rx          Reset target          (RESET). Syntax: rx <DelayAfterReset>
RSetType    Set the current reset type. Syntax: RSetType <type>
Regs        Display contents of registers
wreg        Write register.   Syntax: wreg <RegName>, <Value>
SetBP       Set breakpoint.   Syntax: SetBP <addr> [A/T] [S/H]
SetWP       Set Watchpoint. Syntax: <Addr> [R/W] [<Data> [<D-Mask>] [A-Mask]]
ClrBP       Clear breakpoint. Syntax: ClrBP  <BP_Handle>
```

www.armjishu.com

4.6.8 JLINK V8仿真器如何自动升级

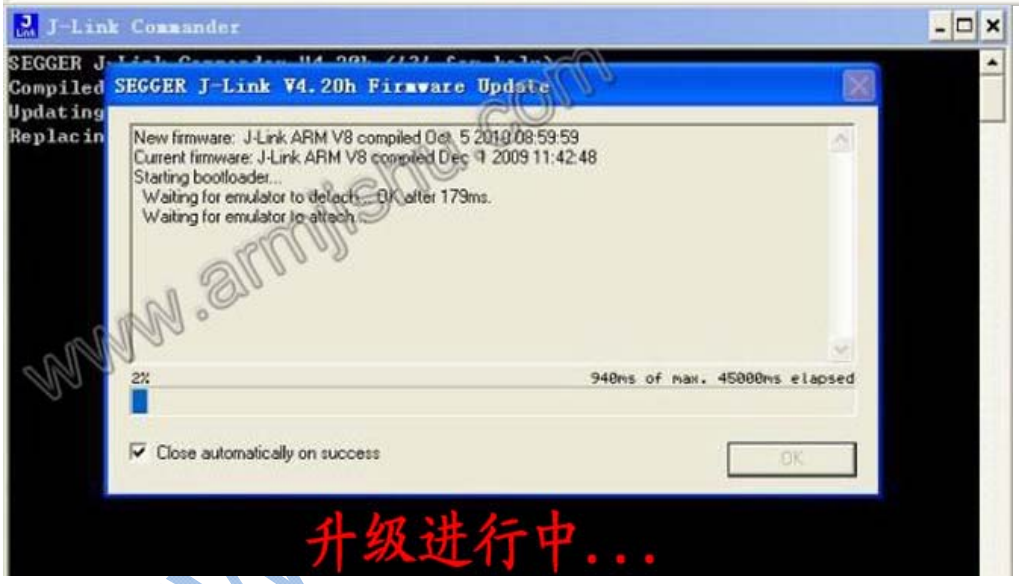
每一次连接 J-LINK, J-LINKARM.dll 都会自动检查 J-LINK 的 firmware 是否最新，如果不是，DLL 将会自动更新设备固件，一般可以在 3S 内完成，升级完成后，J-LINK 不需要重新启动。
建议总是使用最新版本的的 J-LINKARM.dll.

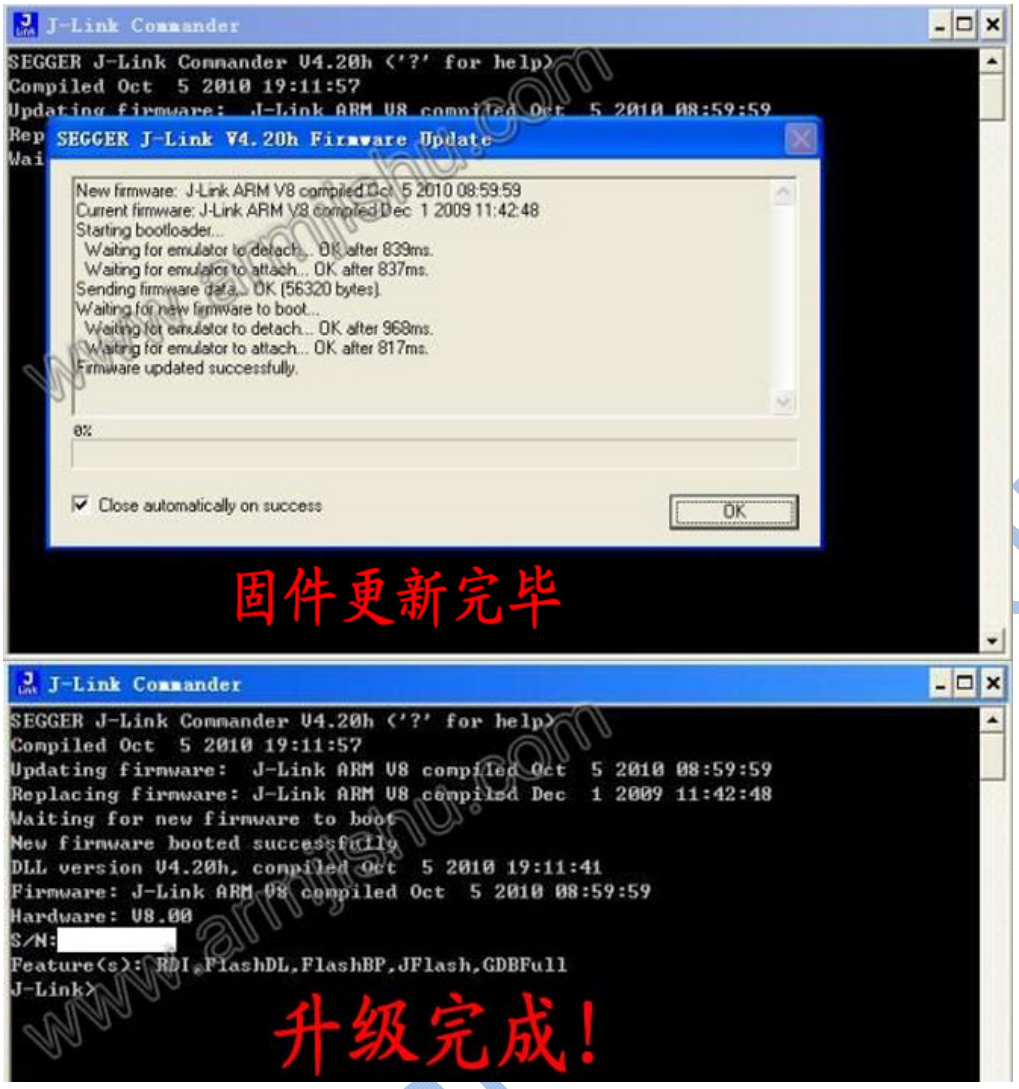
下图是自动更新固件的示意图。

```
J-Link Commander
SEGGER J-Link Commander V4.08l ('?' for help)
Compiled Sep 17 2009 09:42:18
Updating firmware:  J-Link ARM V8 compiled Aug 18 2009 09:06:07
Replacing firmware: J-Link ARM V8 compiled May 27 2009 17:31:22
  ... Firmware update successful. CRC=07DE
Waiting for new firmware to boot
New firmware booted successfully
DLL version V4.08l, compiled Sep 17 2009 09:41:55
Firmware: J-Link ARM V8 compiled Aug 18 2009 09:06:07
Hardware: V8.00
```

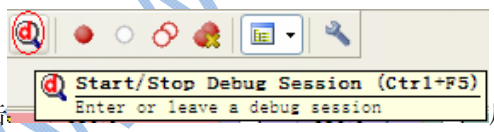
下图是自动更新固件过程示意图：

J-Link Commander
SEGGER J-Link Commander V4.20h ('?' for help)
Compiled Oct 5 2010 19:11:57
J-Link V4.20h Firmware update
A new firmware version is available for the connected emulator.
Do you want to update to the latest firmware version ?
NOTE: Updating to the latest firmware version is strongly recommended.
New features / improvements may not be available without an firmware update.
是(Y)
否(N)
检测到新的固件，提示是否升级，点“是”

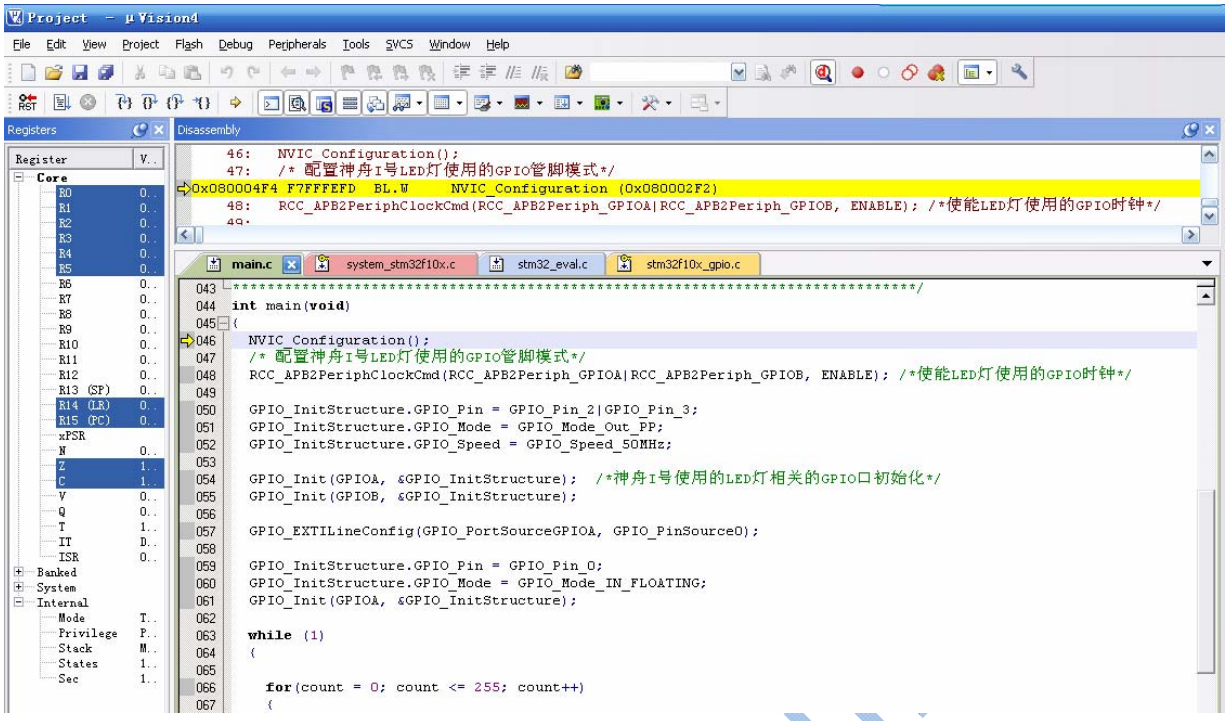

J-Link Commander
SEGGER J-Link V4.20h Firmware Update
New firmware: J-Link ARM V8 compiled Oct 5 2010 08:59:59
Current firmware: J-Link ARM V8 compiled Dec 1 2009 11:42:48
Starting bootloader...
Waiting for emulator to detach... OK after 179ms.
Waiting for emulator to attach...
2%
940ms of max. 45000ms elapsed
Close automatically on success
OK
升级进行中...



4.7 在MDK开发环境中JLINK V8的调试技巧

接下来，点击 

启动在线仿真。软件窗口如下图所示。



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，就可以使程序停止下来，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。

4.8 如何通过串口下载一个固件到神舟I号开发板

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I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I号开发板的过程。

1. 开发板硬件设置

第一步，神舟I号启动模式设置为“System Boot”模式；将跳线J7的跳帽连接（即BOOT0=ON）和J8的断开（BOOT1=OFF），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：

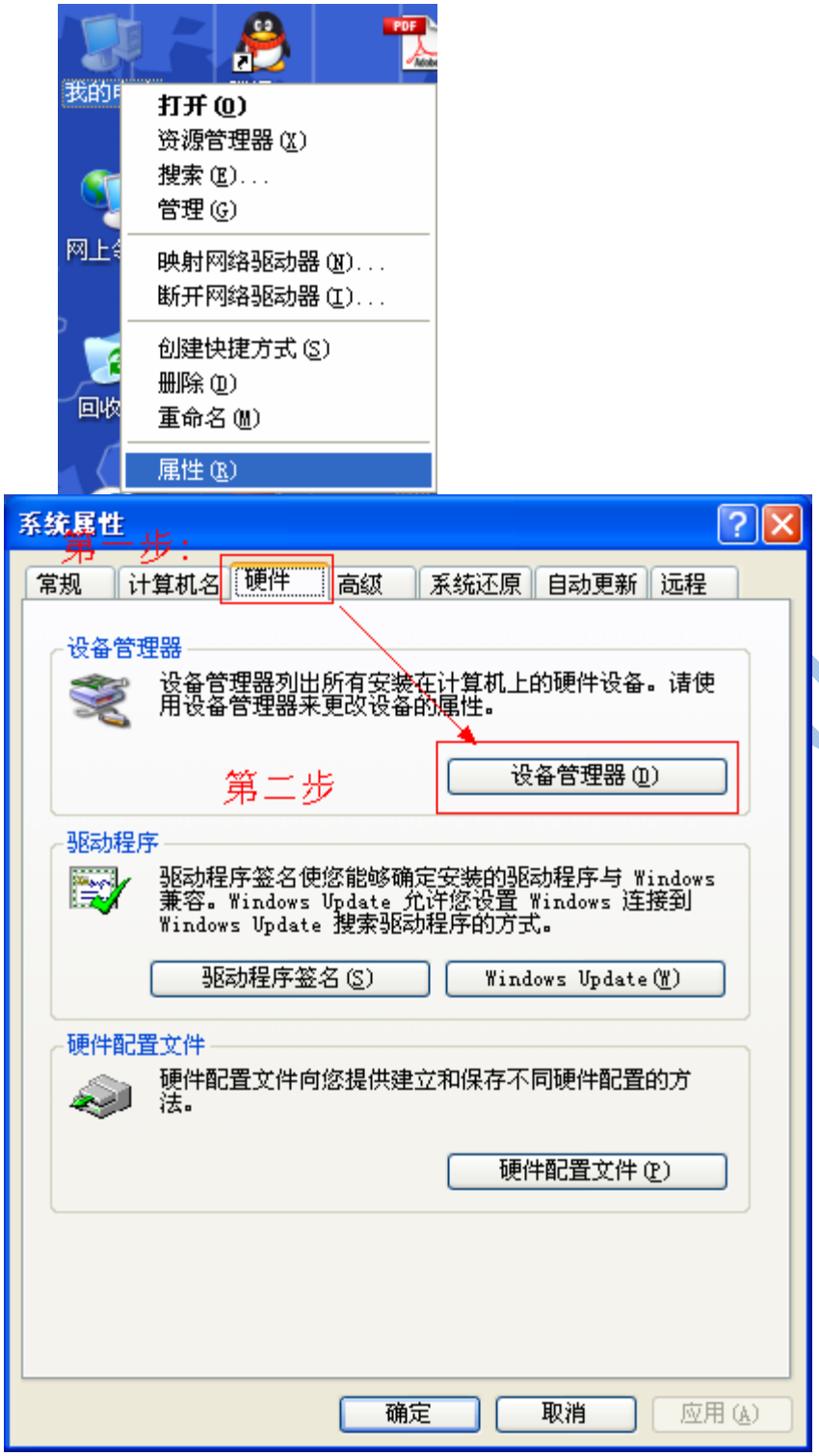
BOOT1（J8）	BOOT0（J7）	功能	说明
ANY	OFF	User Boot(默认)	用主闪存存储器，即Flash启动
OFF	ON	System Boot	系统存储器启动，用于串口下载
ON	ON	SRAM Boot	SRAM启动，用于SRAM中调试代码

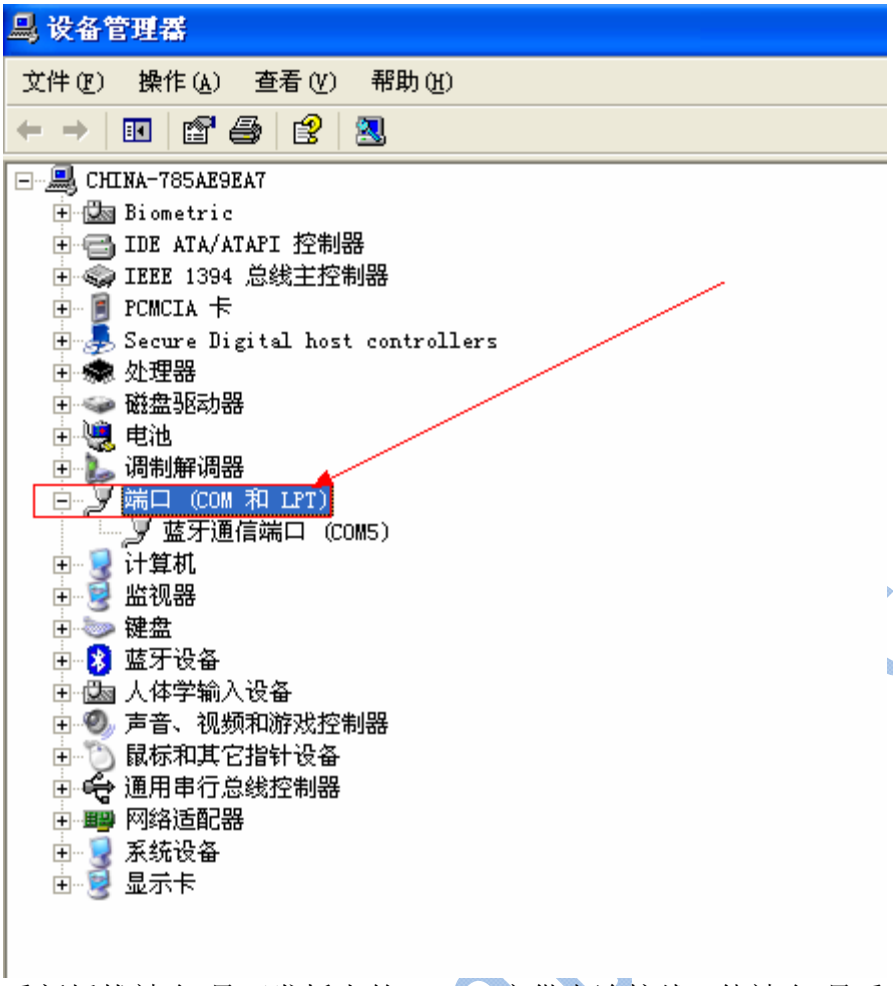
第二步，使用神舟I号配套的串口线连接PC机串口与开发板的串口1，并通电：

2. 软件操作串口下载程序

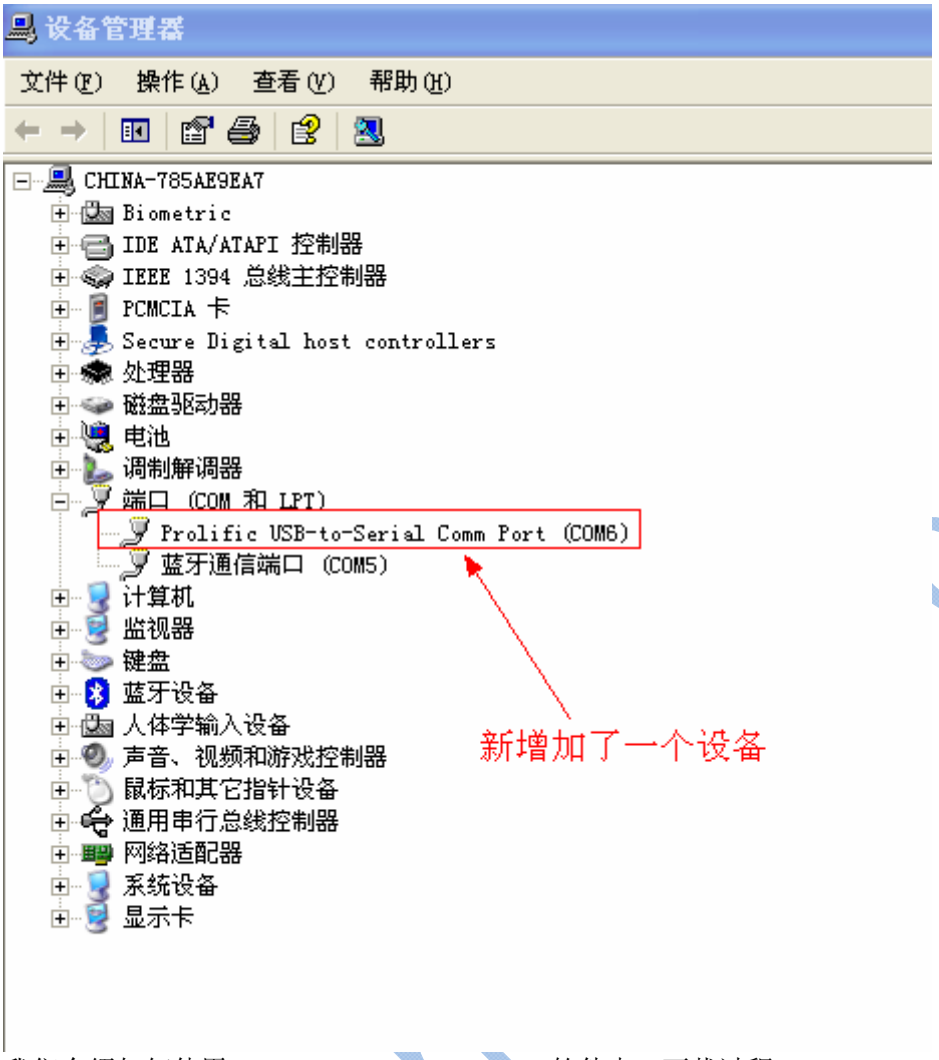
请注意，此时开发板已经通电，并且USB供电线插在J4的USB座上

第一步：查看USB转串口接口所占用的COM口，首先点到 我的电脑，单击右键，选择“属性”：


打开(O)
资源管理器(X)
搜索(E)...
管理(G)
映射网络驱动器(N)...
断开网络驱动器(I)...
创建快捷方式(S)
删除(D)
重命名(M)
属性(R)
系统属性
第一步：
常规
计算机名
硬件
高级
系统还原
自动更新
远程
设备管理器
设备管理器列出所有安装在计算机上的硬件设备。请使用设备管理器来更改设备的属性。
第二步
设备管理器(D)
驱动程序
驱动程序签名使您能够确定安装的驱动程序与 Windows 兼容。Windows Update 允许您设置 Windows 连接到 Windows Update 搜索驱动程序的方式。
驱动程序签名(S)
Windows Update(W)
硬件配置文件
硬件配置文件向您提供建立和保存不同硬件配置的方法。
硬件配置文件(P)
确定
取消
应用(A)



重新插拔神舟I号开发板上的J4 USB座供电连接线，使神舟I号重新上电，可以看到我们的PC电脑识别到了一个新的串口设备，端口占用的是COM6：

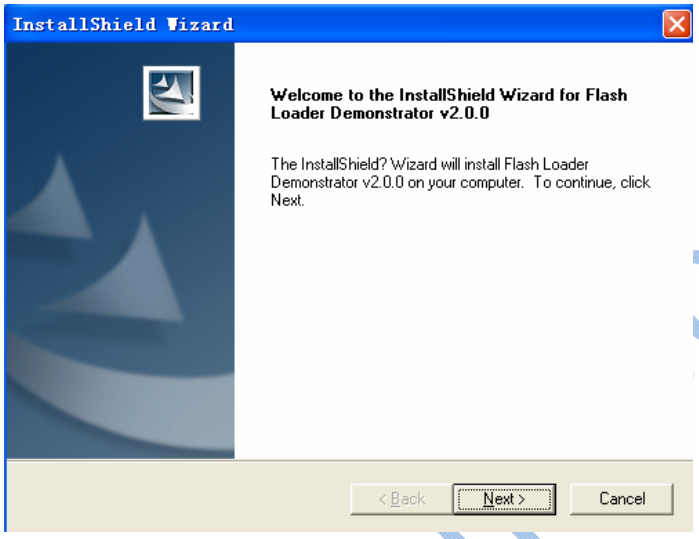


我们介绍如何使用Flash_Loader_Demonstrator软件串口下载过程。

第二步：安装um0462_Flash_Loader_Demonstrator_V2.0_Setup.zip软件（STM32串口下载软件）

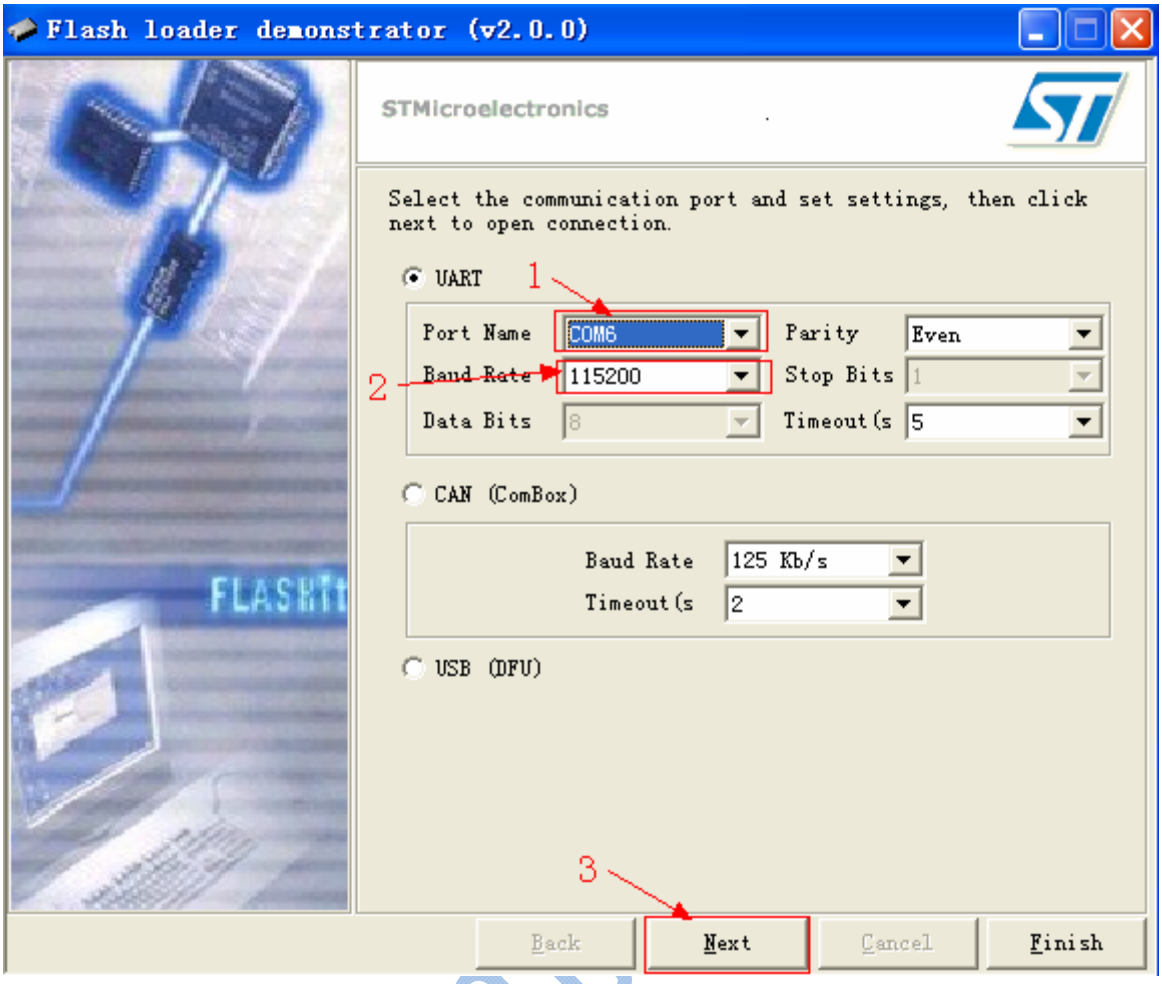
首先打开 神舟I号光盘\串口下载软件目录，解压um0462_Flash_Loader_Demonstrator_V2.0_Setup.zip文件后，双击

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）

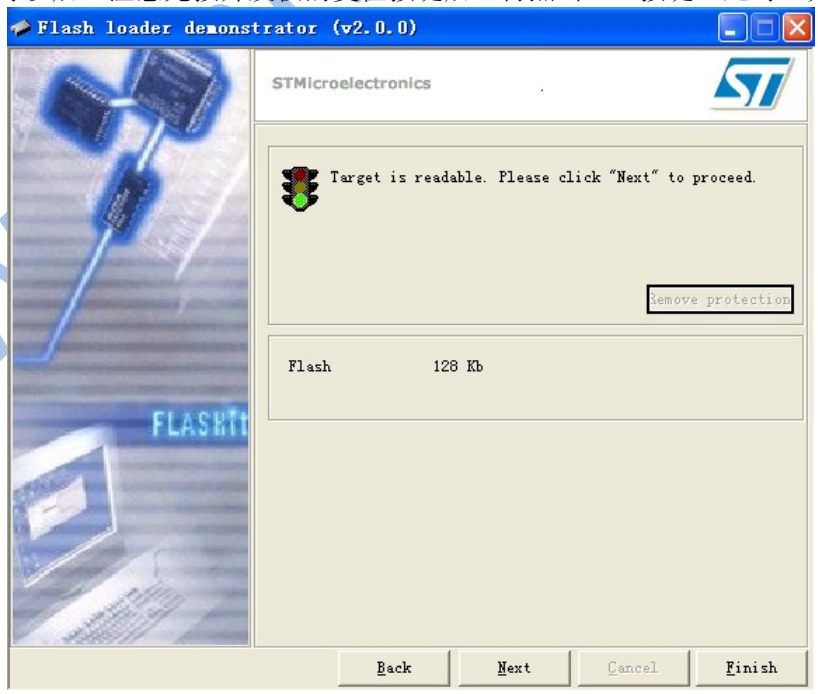


第二步，安装好软件后，运行“开始”→“程序”→“STMicroelectronics”→“Flash_Loader_Demonstrator”→“Flash Loader Demo”，在弹出界面的中，选择正确的PC机使用的串口号和串行通信波特率（这里波特率可以随意，我们这里115200是最大的波特率了，表示最极限的速度下，我们的下载也是成功的）

www.armjishu.com



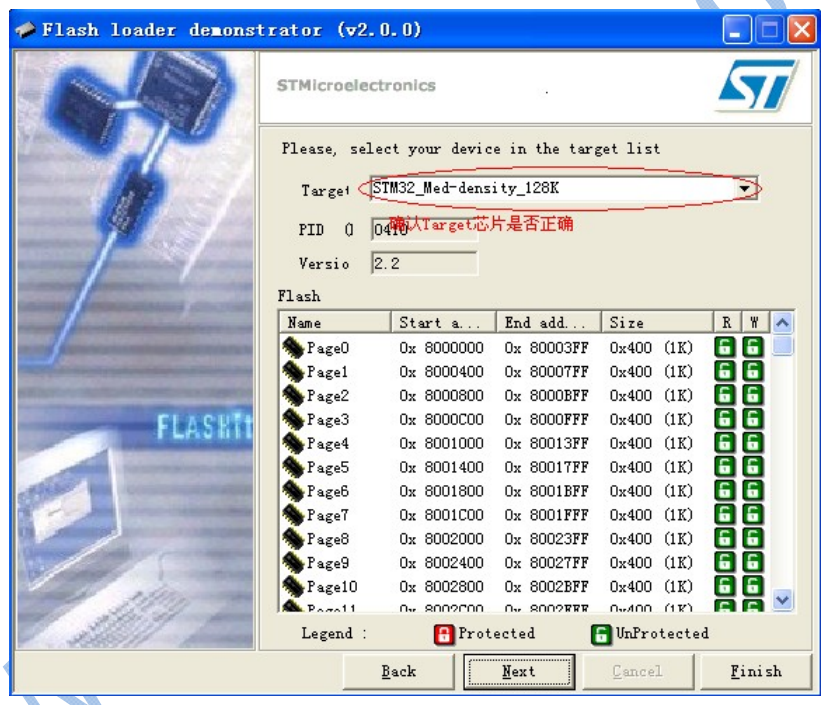
按上图设置好以后，注意先按开发板的复位按键后，再点击Next按键，此时，如下界面。



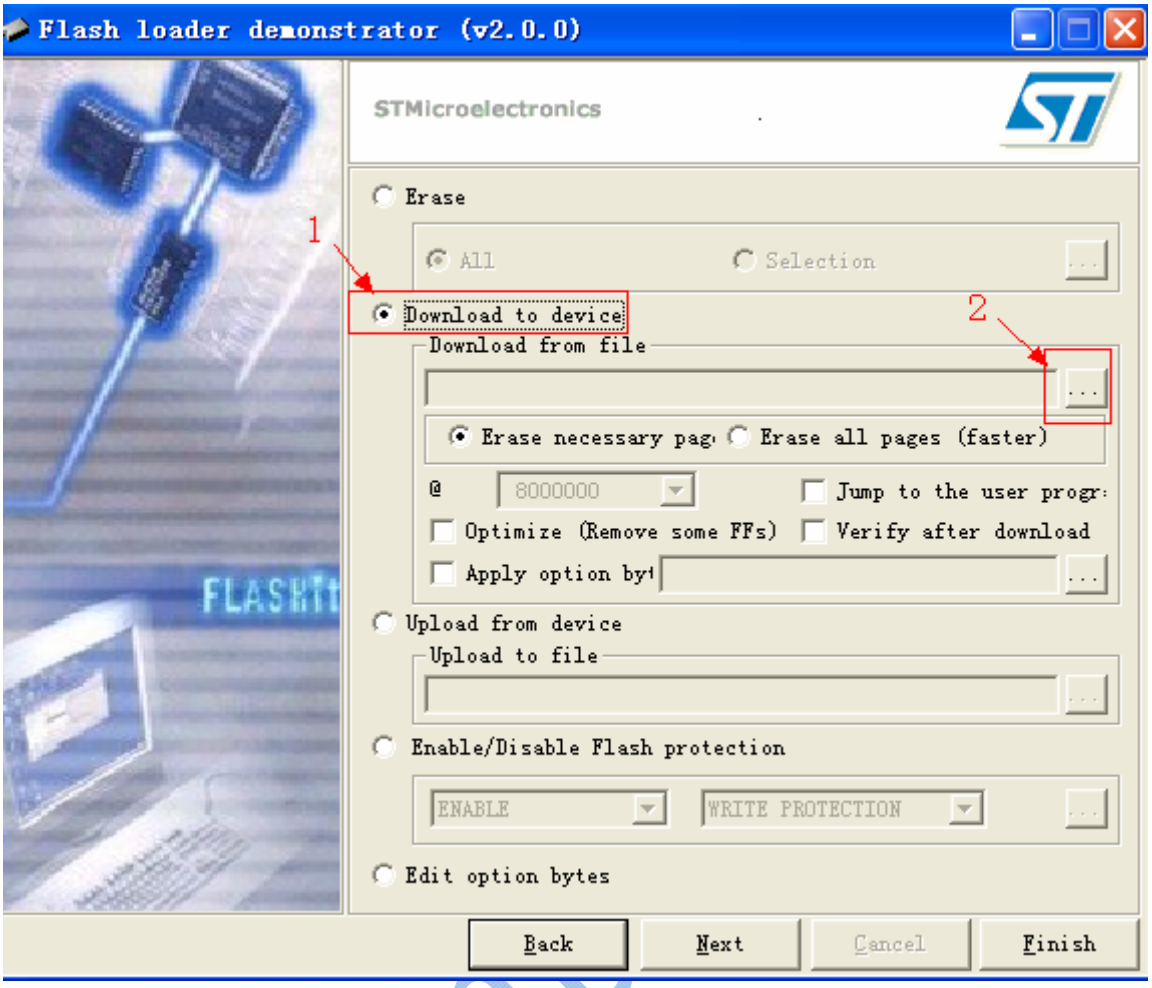
我们通过查询手册，可以知道目标芯片的ROM容量大小，在这里选择目标开发板的

STM32F103RBT6目标芯片型号是正确的：

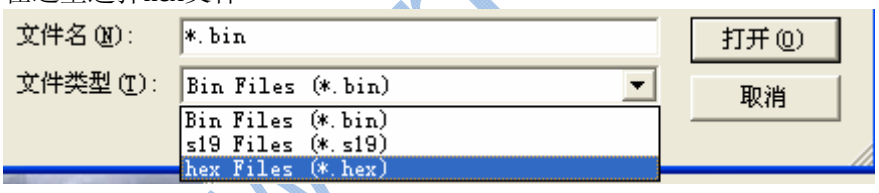
STM32(				
型号		CPU 频率 (MHz)	程序空间 (字节)	RAM (字节)
48脚	STM32F103C4	72	16K	6K
	STM32F103C6	72	32K	10K
	STM32F103C8	72	64K	20K
	STM32F103CB	72	128K	20K
64脚	STM32F103R4	72	16K	6K
	STM32F103R6	72	32K	10K
	STM32F103R8	72	64K	20K
	STM32F103RB	72	128K	20K
	STM32F103RC	72	256K	48K
	STM32F103RD	72	384K	64K
	STM32F103RE	72	512K	64K

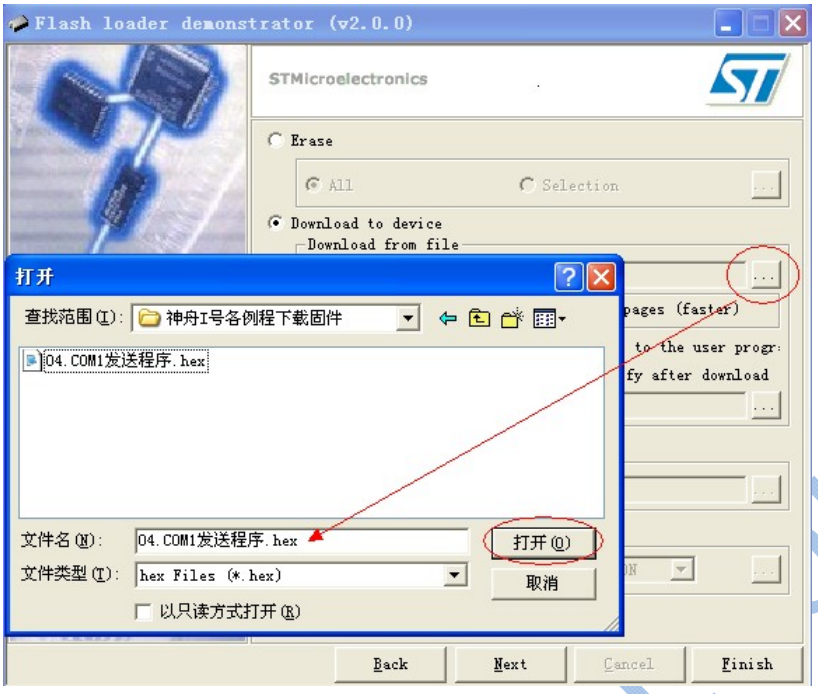


浏览加载需要下载的HEX文件：

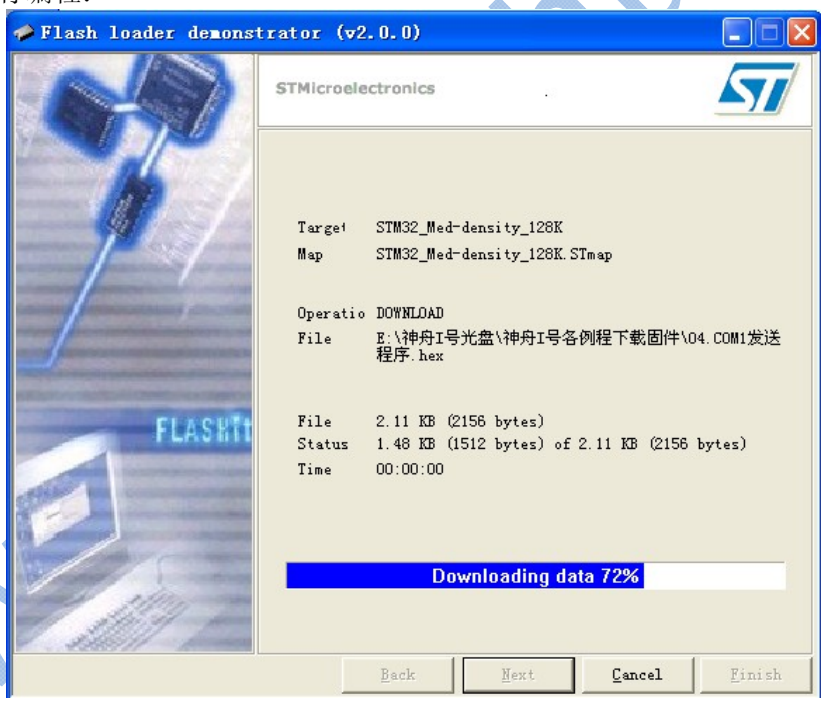

Flash loader demonstrator (v2.0.0)
STMicroelectronics
Erase
All
Selection
1
Download to device
2
Download from file
Erase necessary pag
Erase all pages (faster)
@
8000000
Jump to the user progr
Optimize (Remove some FFs)
Verify after download
Apply option by
Upload from device
Upload to file
Enable/Disable Flash protection
ENABLE
WRITE PROTECTION
Edit option bytes
Back
Next
Cancel
Finish

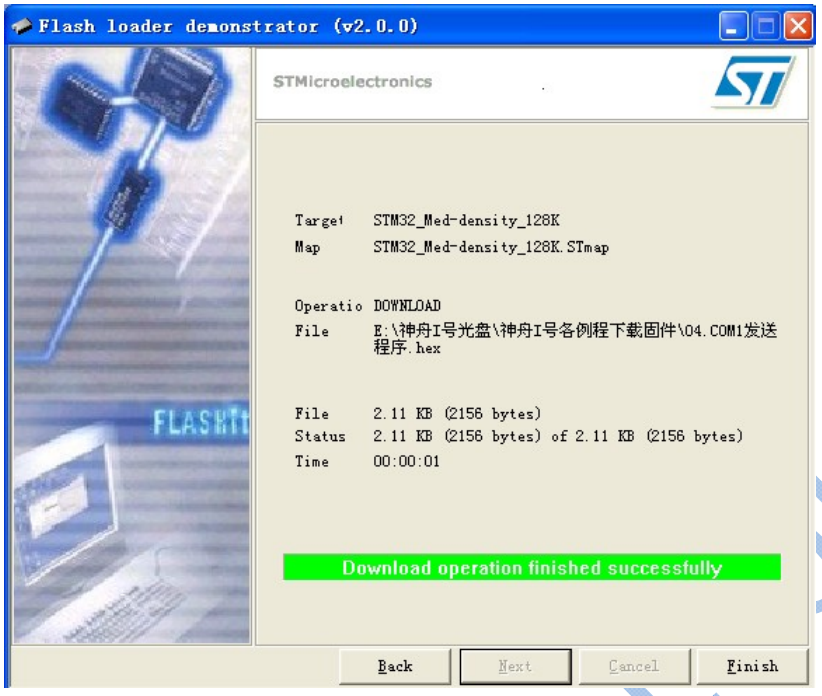
在这里选择hex文件


文件名(N):
*.bin
打开(O)
文件类型(T):
Bin Files (*.bin)
取消
Bin Files (*.bin)
s19 Files (*.s19)
hex Files (*.hex)


Flash loader demonstrator (v2.0.0)
STMicroelectronics
Erase
All
Selection
Download to device
Download from file
打开
查找范围(I):
神舟I号各例程下载固件
04.COM1发送程序.hex
文件名(N):
04.COM1发送程序.hex
打开(O)
文件类型(T):
hex Files (*.hex)
取消
以只读方式打开(R)
Back
Next
Cancel
Finish

点击Next进行编程：


Flash loader demonstrator (v2.0.0)
STMicroelectronics
Target STM32_Med-density_128K
Map STM32_Med-density_128K.STmap
Operatio DOWNLOAD
File E:\神舟I号光盘\神舟I号各例程下载固件\04.COM1发送程序.hex
File 2.11 KB (2156 bytes)
Status 1.48 KB (1512 bytes) of 2.11 KB (2156 bytes)
Time 00:00:00
Downloading data 72%
Back
Next
Cancel
Finish



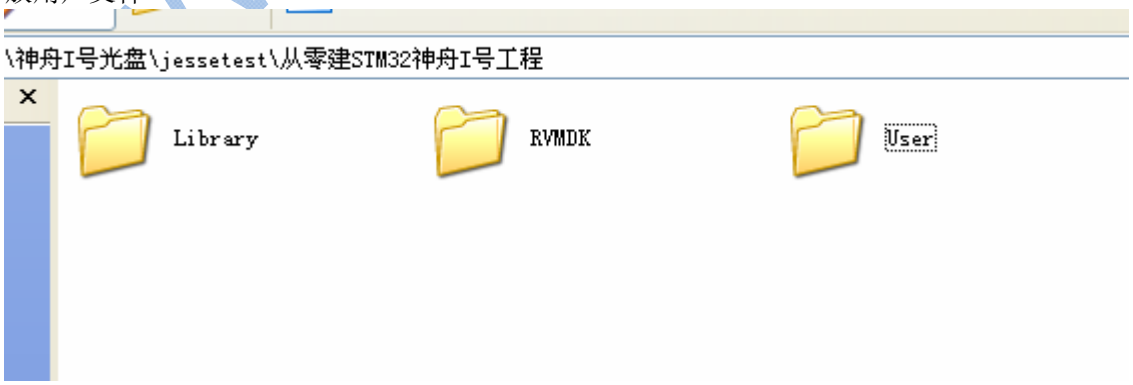
至此，HEX文件已经成功下载了，为了看到程序运行的现象，我们需要将启动模式设置为User Boot模式，即将J7断开。复位神舟I号开发板即可看到程序运行的实验现象。

4.9 从零开始新建一个STM32的工程模板

上面是说的如何运行光盘中的一个例程，下面我们打算从 0 开始，重新创建一个新的例程，包括把库文件这些也加载进来，这样大家可以更能够深入理解环境是如何搭建的：

1 新建一个文件夹（从零建 STM32 神舟 I 号工程）

2.在刚才的文件夹下新建如下三个文件夹，其中 Library 放官方的库文件，RVMDK 放工程文件，User 放用户文件



4. 我们下载目前最新 3.5 版本的 STM32 库文件（2011-06-15），进入 http://www.st.com/cn/mcu/product/164495.jsp页面，找到要下载的固件，

固件下载链接已经找到了，直接点击即可下载：
http://www.st.com/cn/com/SOFTWARE_RESOURCES/SW_COMPONENT/FIRMWARE/stm32f10x_stdperiph_lib.zip

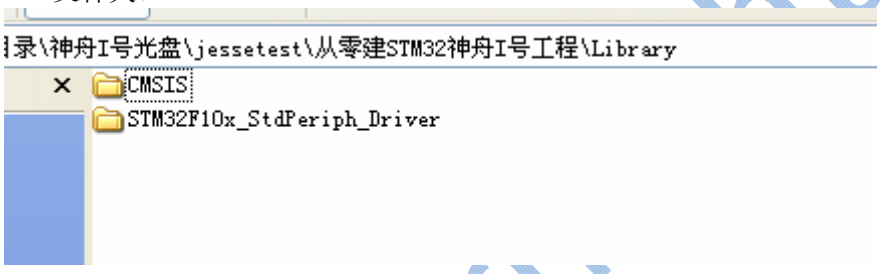
STM32F10x standard peripheral library	3.5.0	21617KB

5. 下载完后进行解压缩：

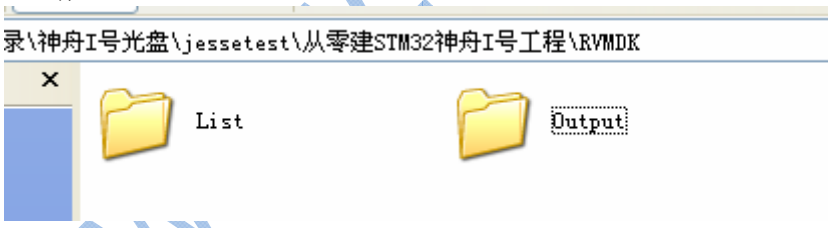
stm32f10x_stdperiph_lib	文件夹	
stm32f10x_stdperiph_lib.zip	21,617 KB	WinRAR ZIP

6. 进入目录“stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries “就可以看到我们的库文件了：

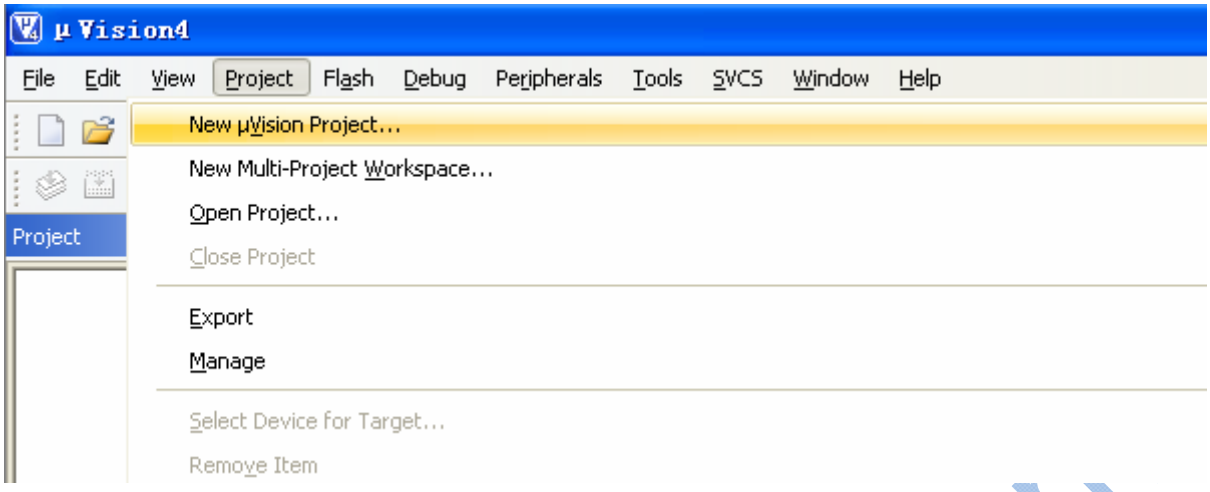
7. 把官方的库文件复制到 Library 文件下。这里只复制 CMSIS 文件夹和 STM32F10x_StdPeriph_Driver 文件夹。

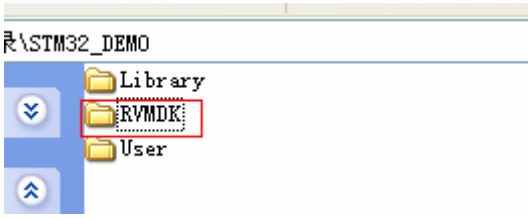


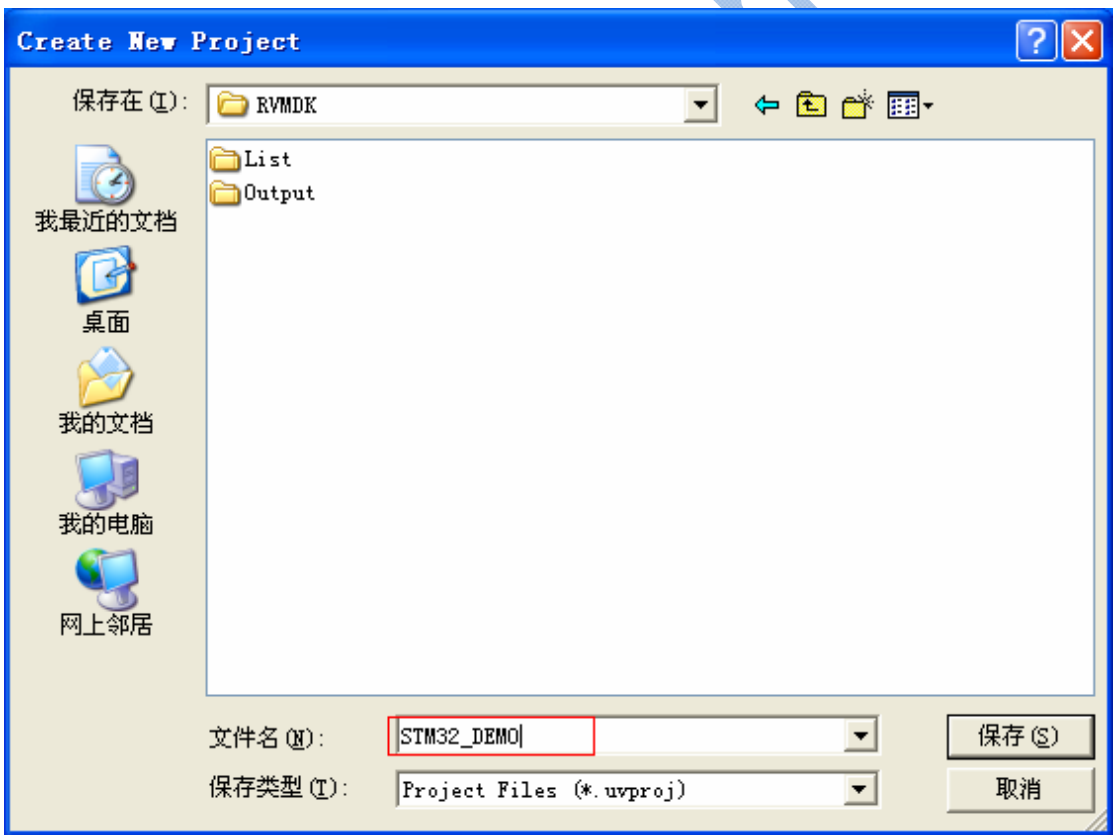
8. 在 RVMDK 文件夹中新建 List 和 Output 文件夹用来装 MDK 产生的临时文件和一些“垃圾”文件，给别人发送代码交流的时候可以把这两个文件夹里面的东西删除后发送，能减少大概 20M 的内存。

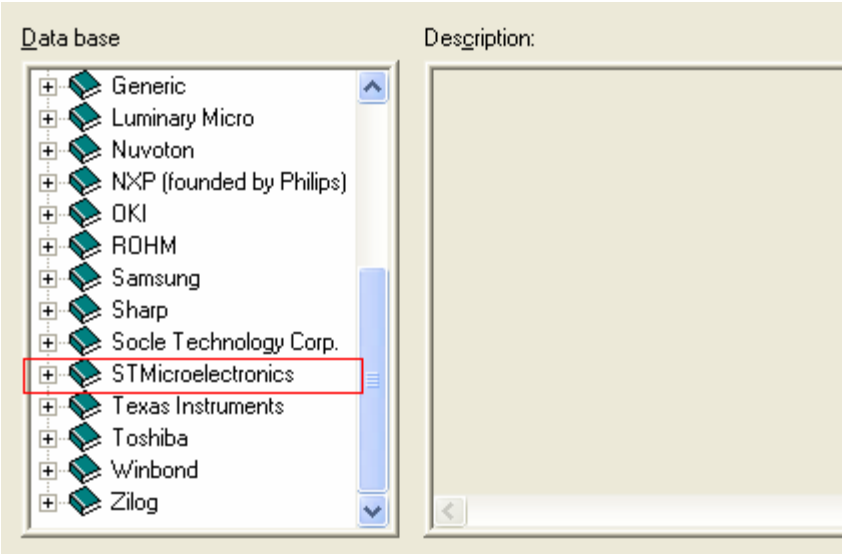


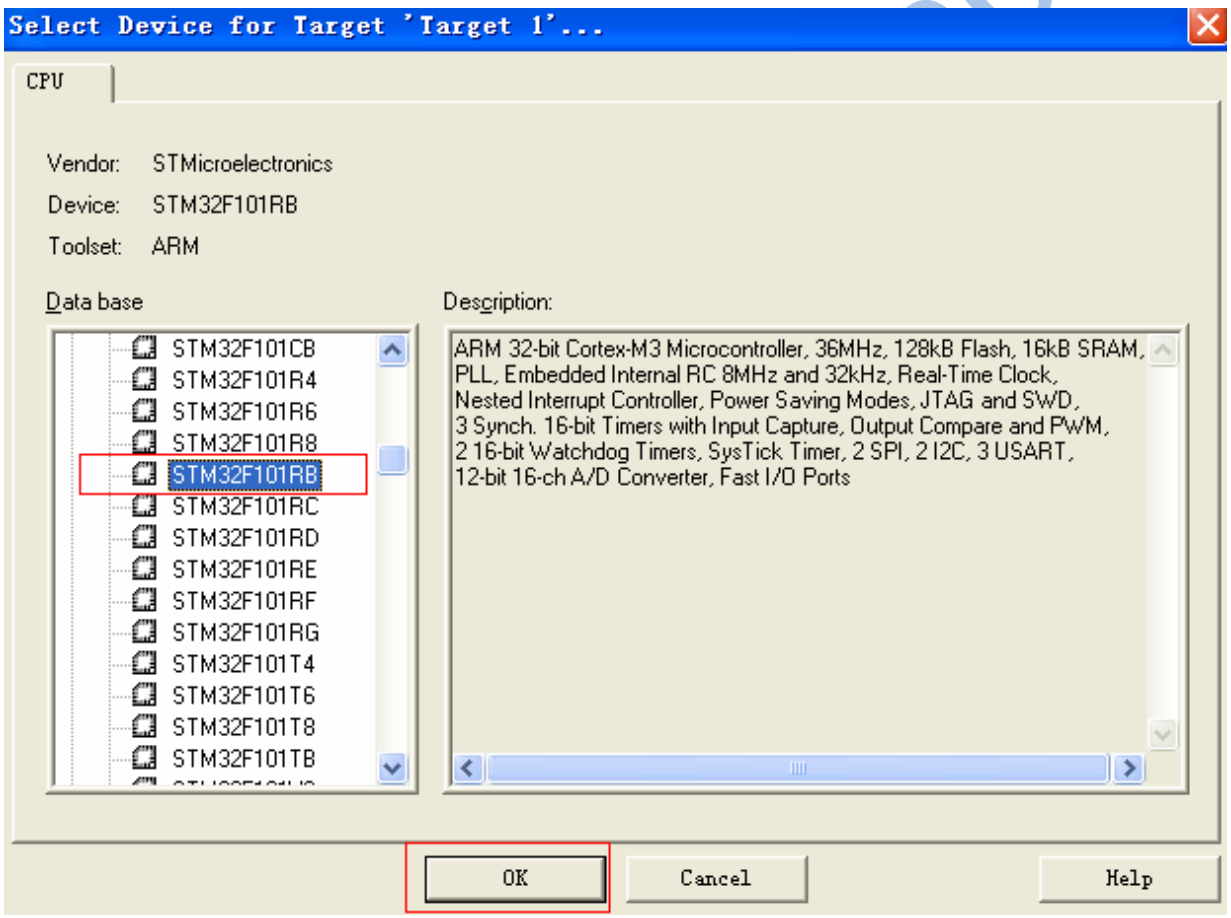
9. 下面我们先打开 Keil 软件在 RVMDK 文件夹的根目录新建工程，如下图：


µ Vision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
New µVision Project...
New Multi-Project Workspace...
Open Project...
Close Project
Export
Manage
Select Device for Target...
Remove Item
Project


录\STM32_DEMO
Library
RVMDK
User

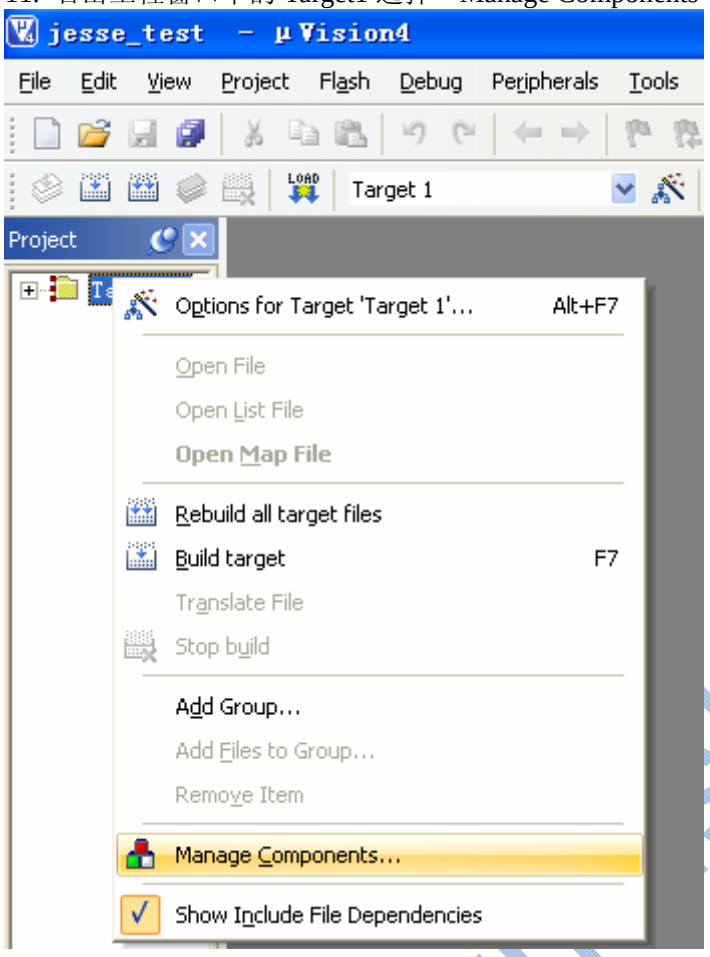

Create New Project
保存在(I):
RVMDK
List
Output
我最近的文档
桌面
我的文档
我的电脑
网上邻居
文件名(N):
STM32_DEMO
保存(S)
保存类型(T):
Project Files (*.uvproj)
取消





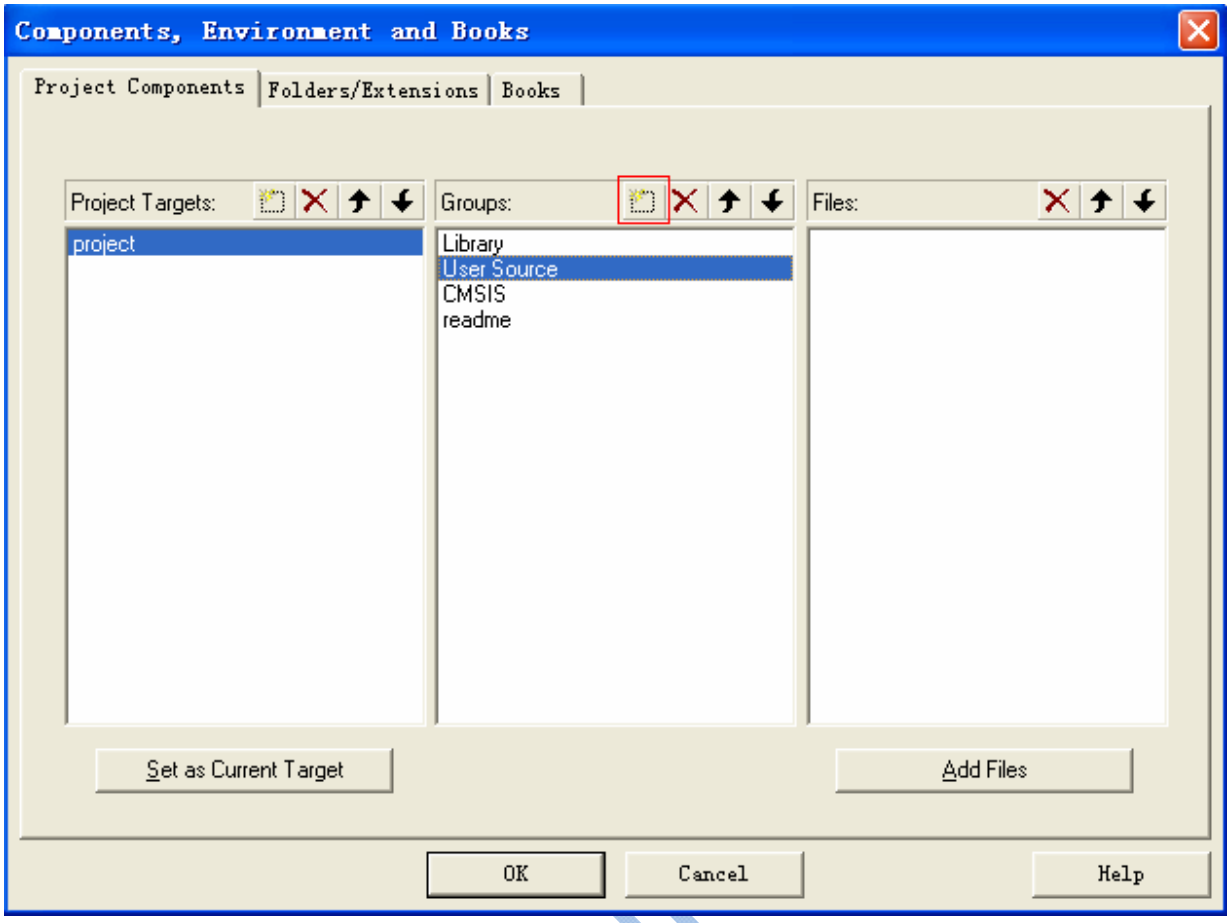
10. 弹出是否添加 STM32 的启动文件的对话框，这里选【是】!

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



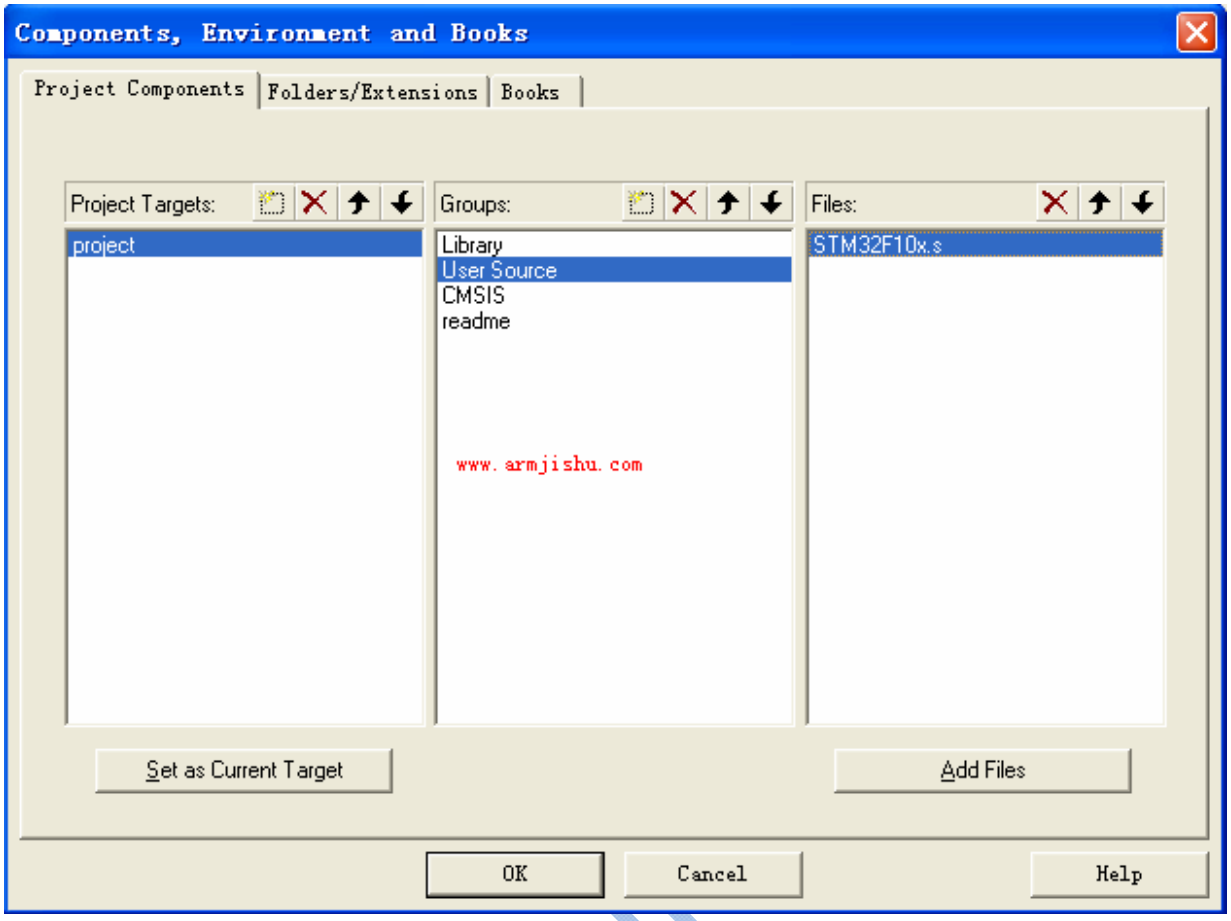
12. 在 Project Target 中把 Target1 改成你想要的名字，可以不改，这里改为 project，然后在 groups 单击新建按钮，新建 User Source，Library，CMSIS，和 readme 四个组：

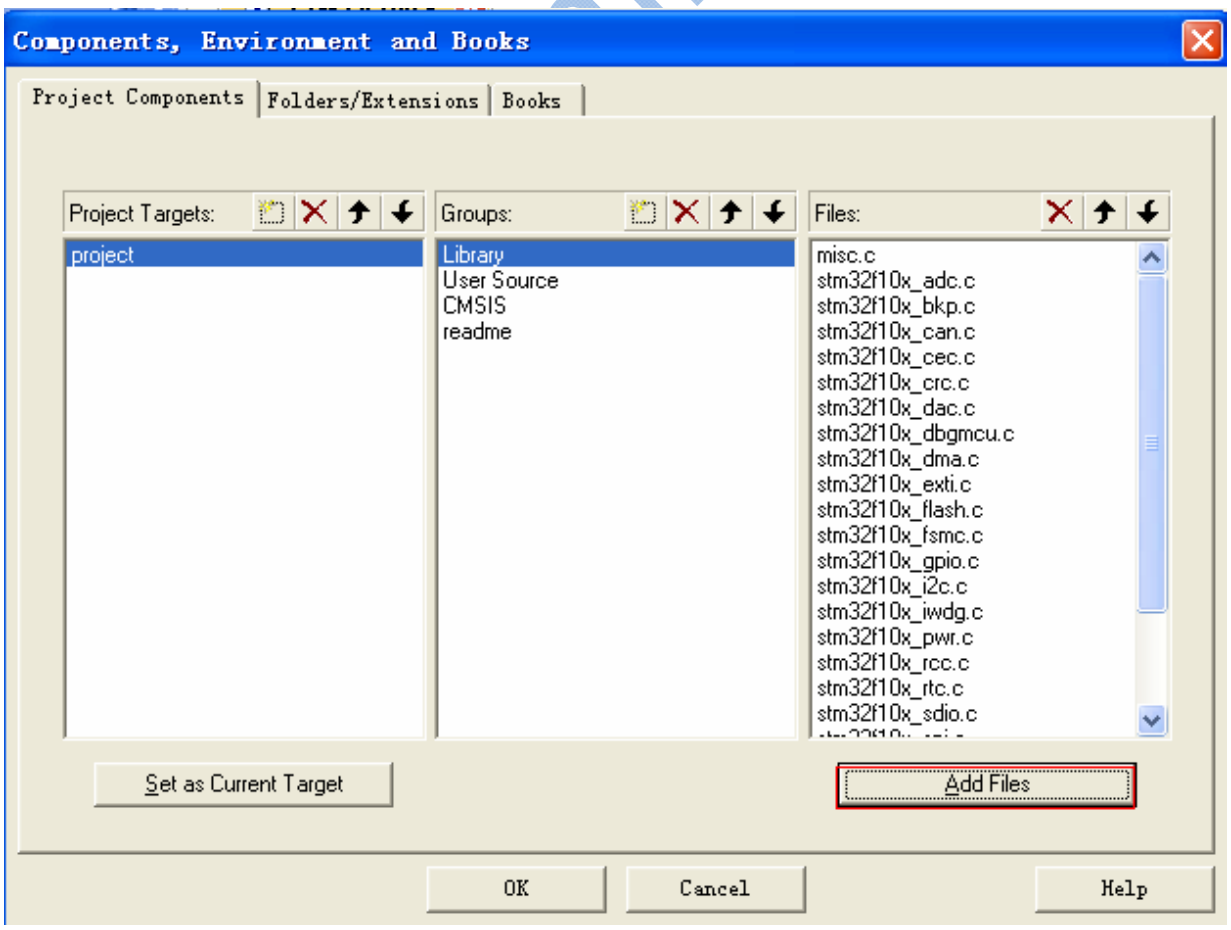
www.armjishu.com



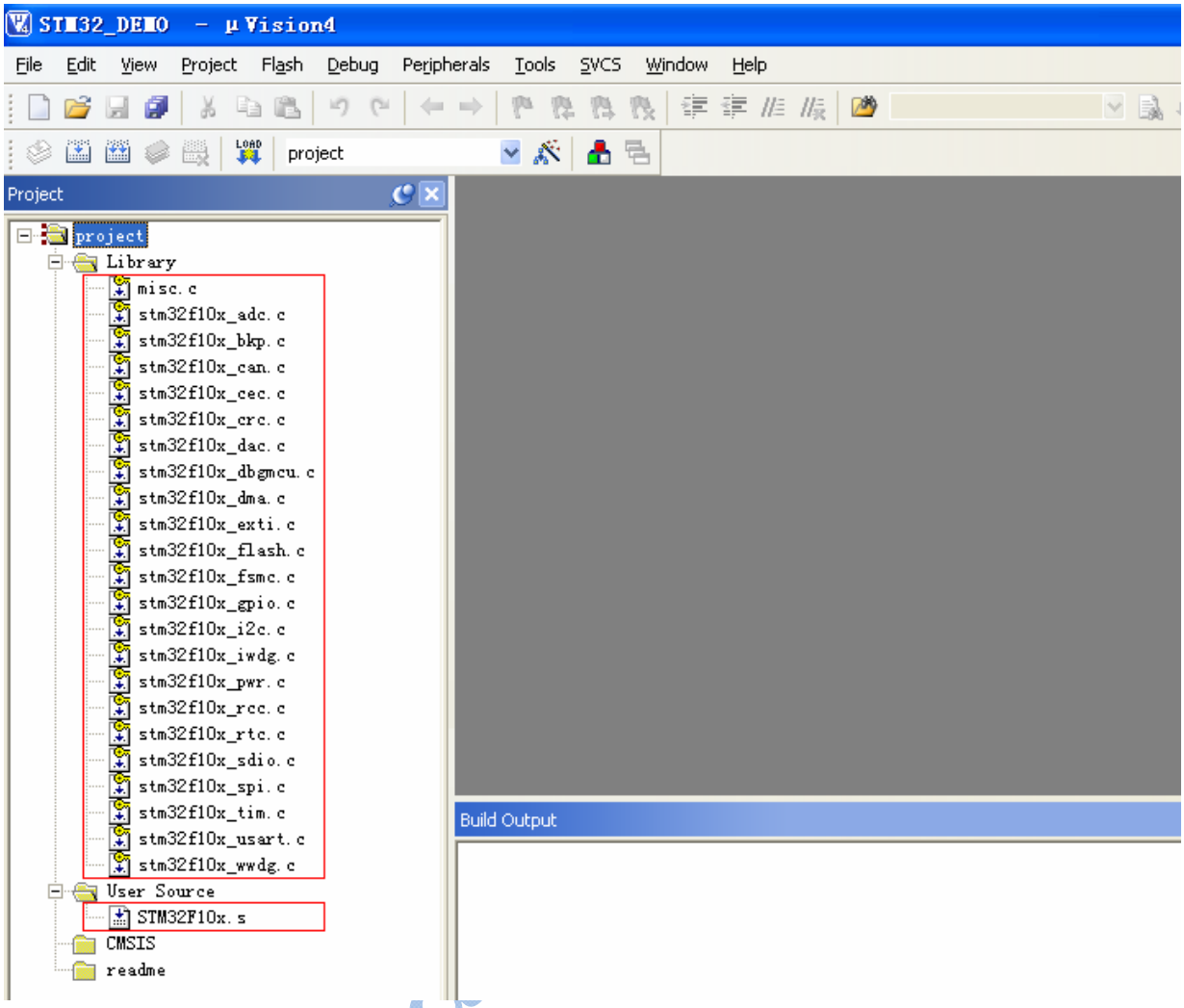
13. 先选中一个组之后，在 Files 点击 Add Files，在 User Source 组添加启动文件 STM32F10x.s，在 library 组添加 Library->STM32F10x_StdPeriph_Driver->src 中的所有.c 文件：

www.armjishu

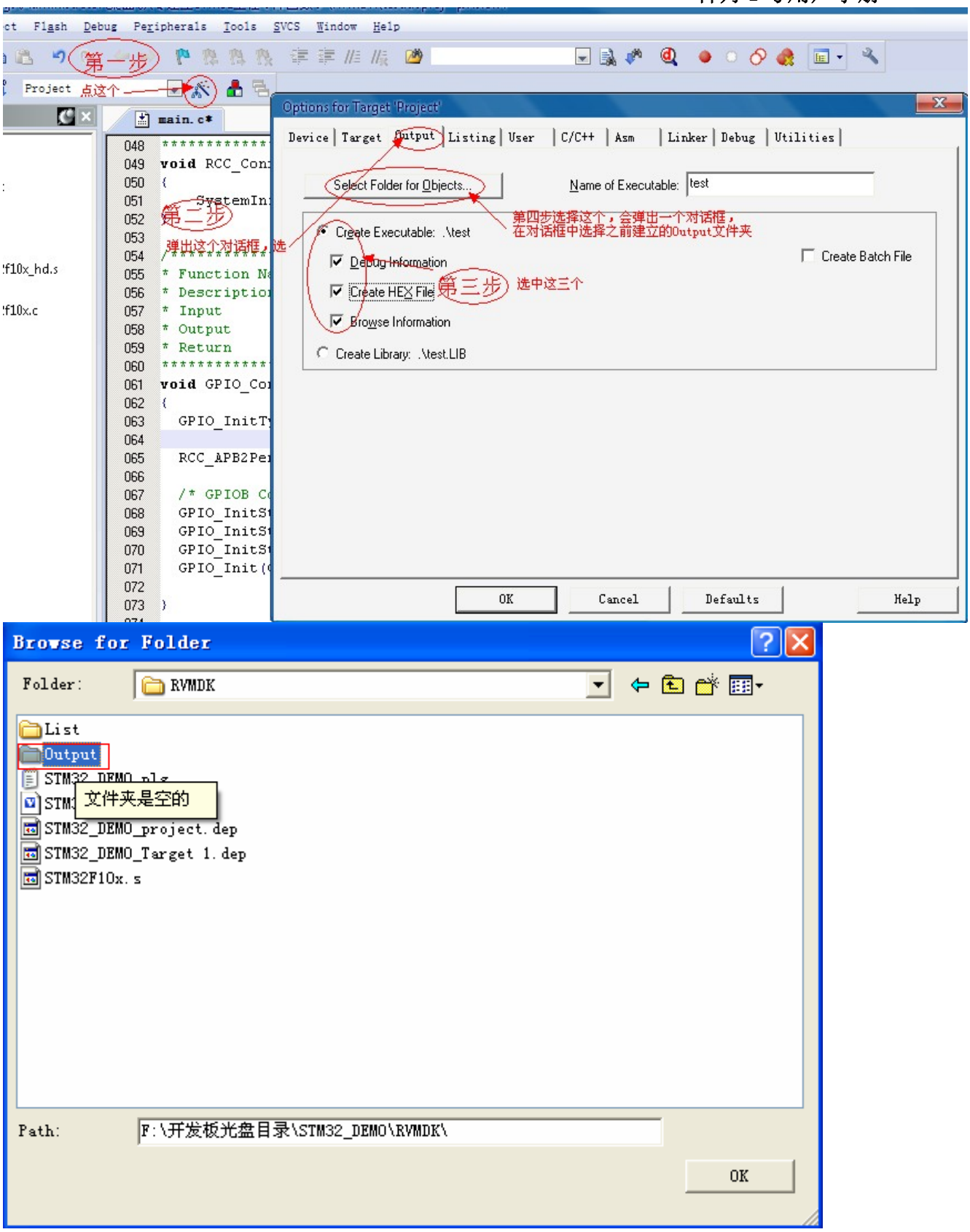

Components, Environment and Books
Project Components
Folders/Extensions
Books
Project Targets:
Groups:
Files:
project
Library
User Source
CMSIS
readme
STM32F10x.s
www.armjishu.com
Set as Current Target
Add Files
OK
Cancel
Help

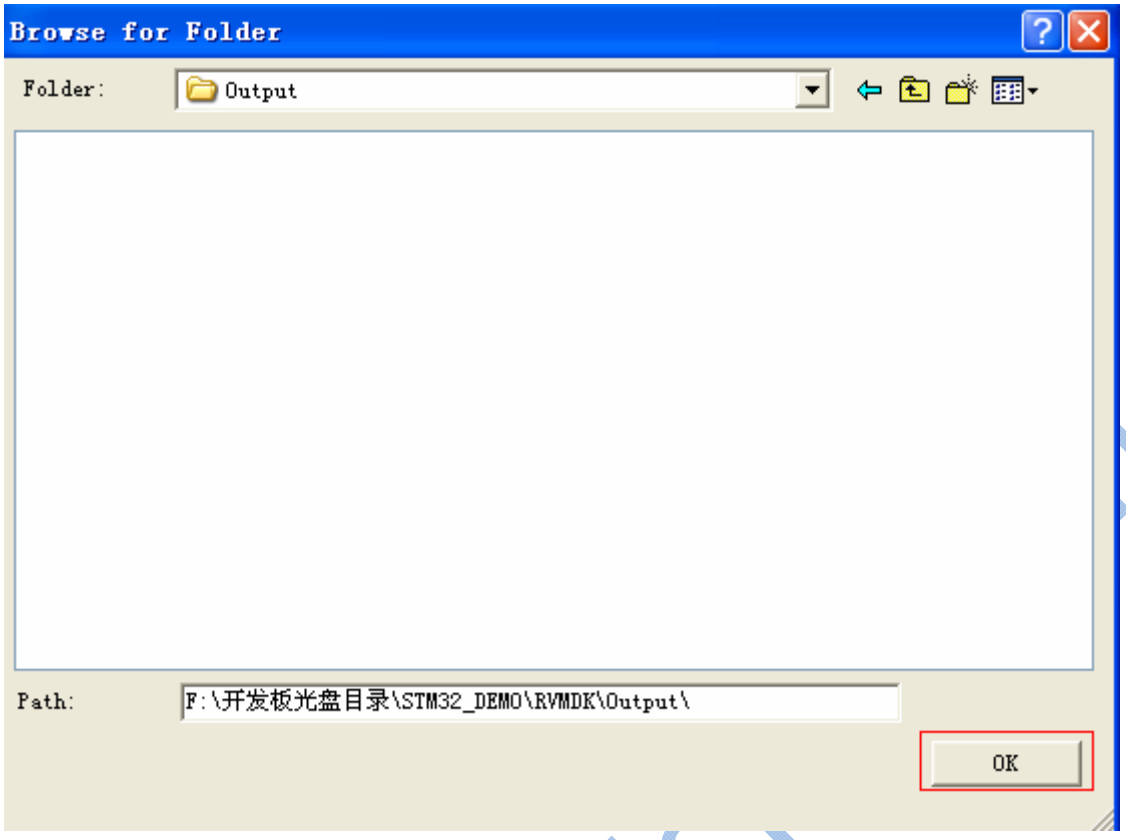

Components, Environment and Books
Project Components
Folders/Extensions
Books
Project Targets:
Groups:
Files:
project
Library
User Source
CMSIS
readme
misc.c
stm32f10x_adc.c
stm32f10x_bkp.c
stm32f10x_can.c
stm32f10x_cec.c
stm32f10x_crc.c
stm32f10x_dac.c
stm32f10x_dbgmcu.c
stm32f10x_dma.c
stm32f10x_exti.c
stm32f10x_flash.c
stm32f10x_fsmc.c
stm32f10x_gpio.c
stm32f10x_i2c.c
stm32f10x_iwdg.c
stm32f10x_pwr.c
stm32f10x_rcc.c
stm32f10x_rtc.c
stm32f10x_sdio.c
Set as Current Target
Add Files
OK
Cancel
Help

14. 点【OK】后，可以看到左边出现了我们刚才所添加的文件

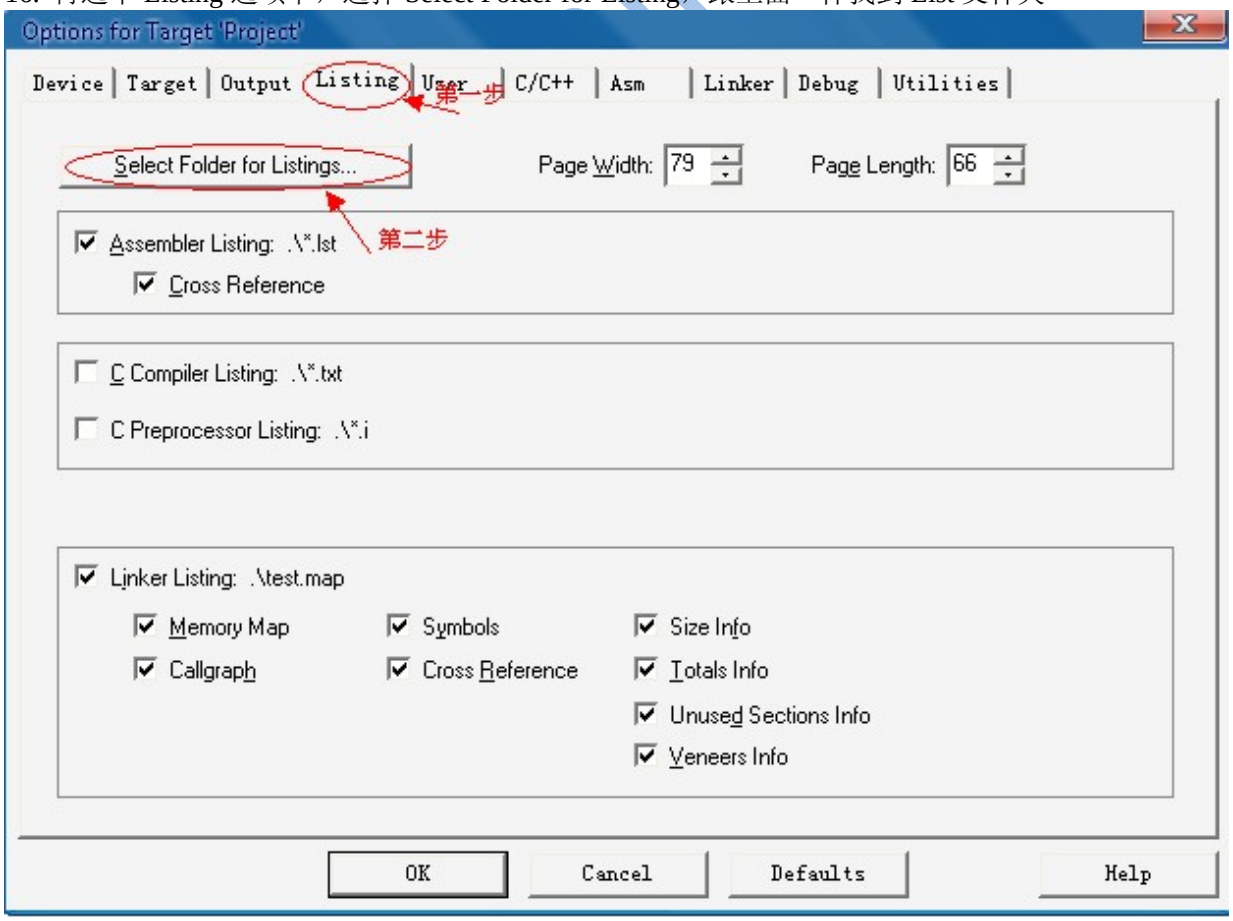


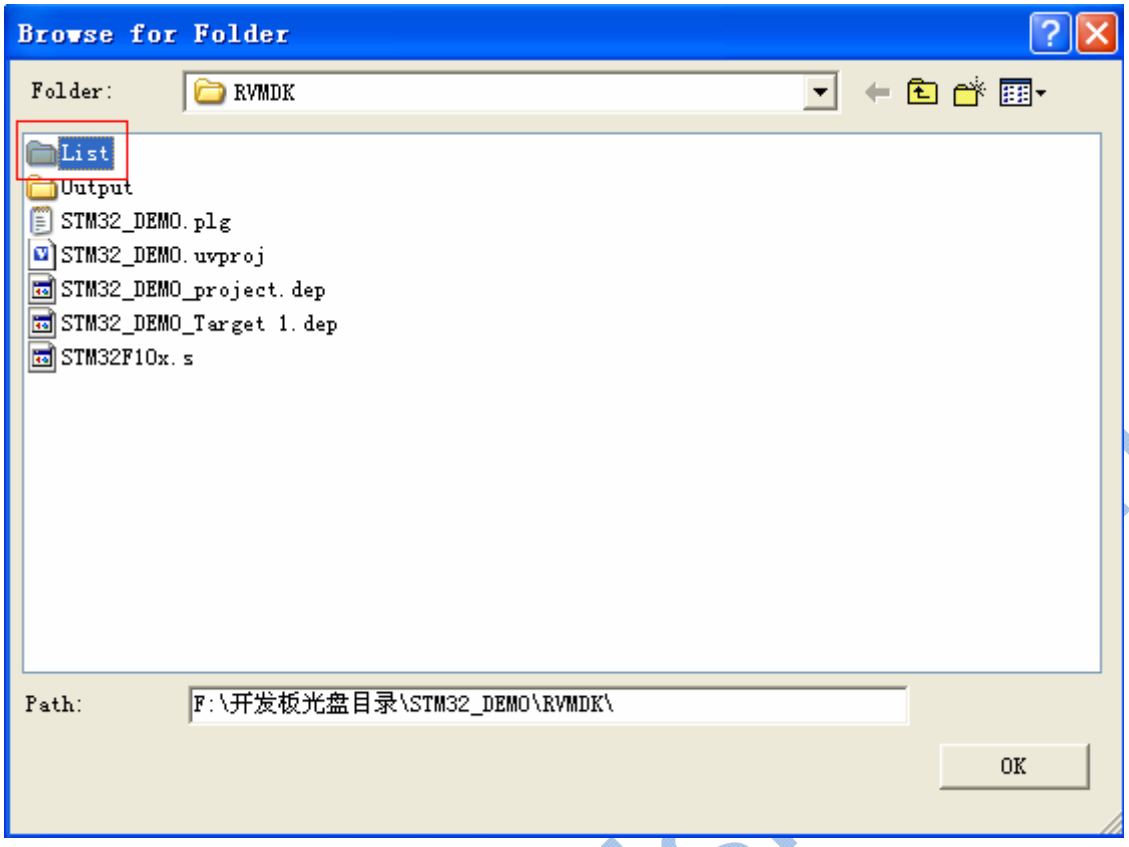
15. 然后是工程设置。第一步点工程设置选项，第二步点击 Options for Target Project 窗口的 Output 选项卡，第三步选中调试信息、产生 hex 文件、和浏览信息，第四步点击 Select Folder For Objeces 设置，在弹出的窗口中找到 RVMDK 文件夹中的 Output 文件夹，选择 Output 文件夹后确定

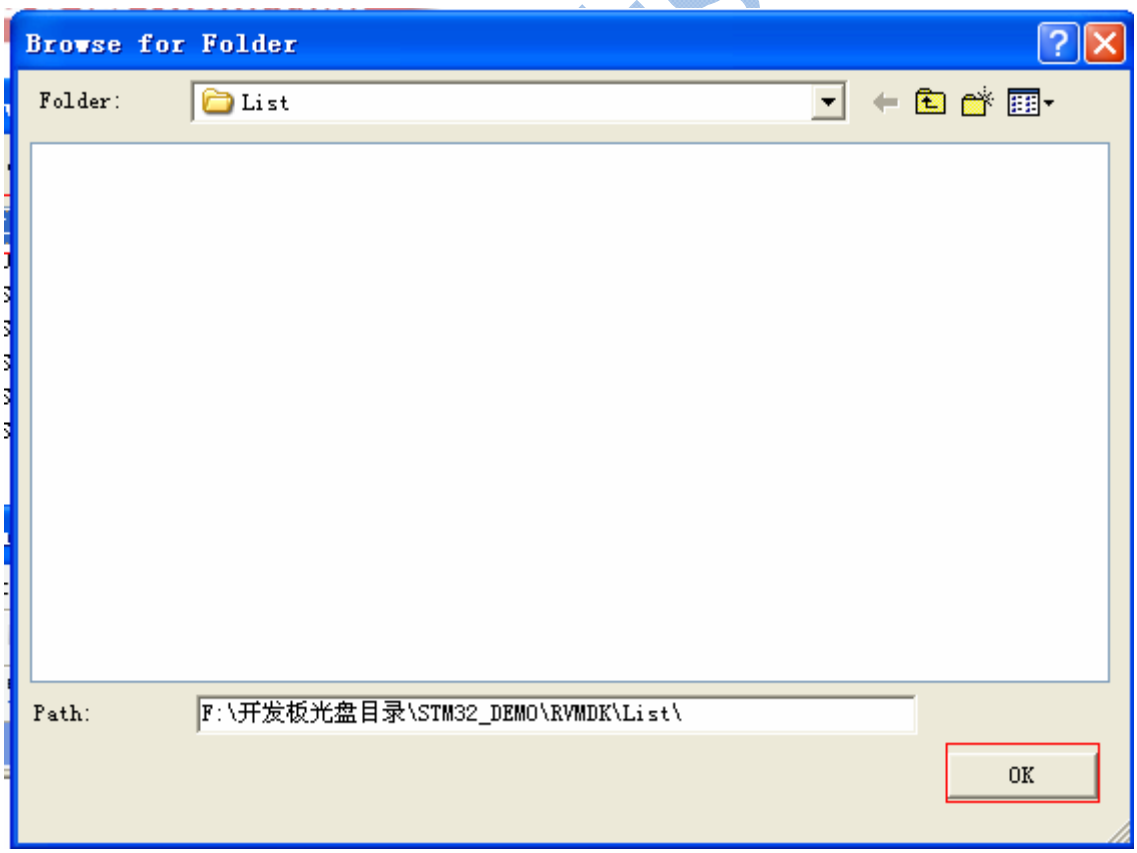

第一步
点这个
第二步
弹出这个对话框，选
Options for Target 'Project'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Select Folder for Objects...
Name of Executable: test
第四步选择这个，会弹出一个对话框，
在对话框中选择之前建立的Output文件夹
Create Executable: .\test
Debug Information
Create HEX File
第三步
选中这三个
Browse Information
Create Library: .\test.LIB
Create Batch File
OK
Cancel
Defaults
Help
Browse for Folder
Folder:
RVMDK
List
Output
文件夹是空的
STM32_DEMO_project.dep
STM32_DEMO_Target 1.dep
STM32F10x.s
Path:
F:\开发板光盘目录\STM32_DEMO\RVMDK\
OK



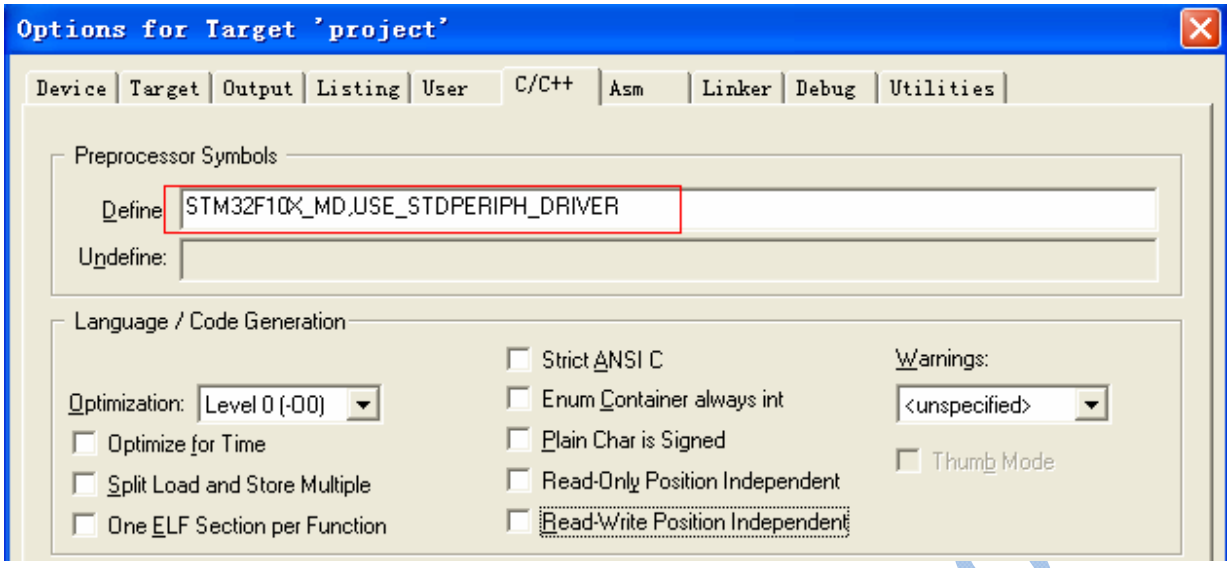
16. 再选中 Listing 选项卡，选择 Select Folder for Listing，跟上面一样找到 List 文件夹







17. 选中 C/C++选项卡，在 Define 中输入 STM32F10X_MD,USE_STDPERIPH_DRIVER



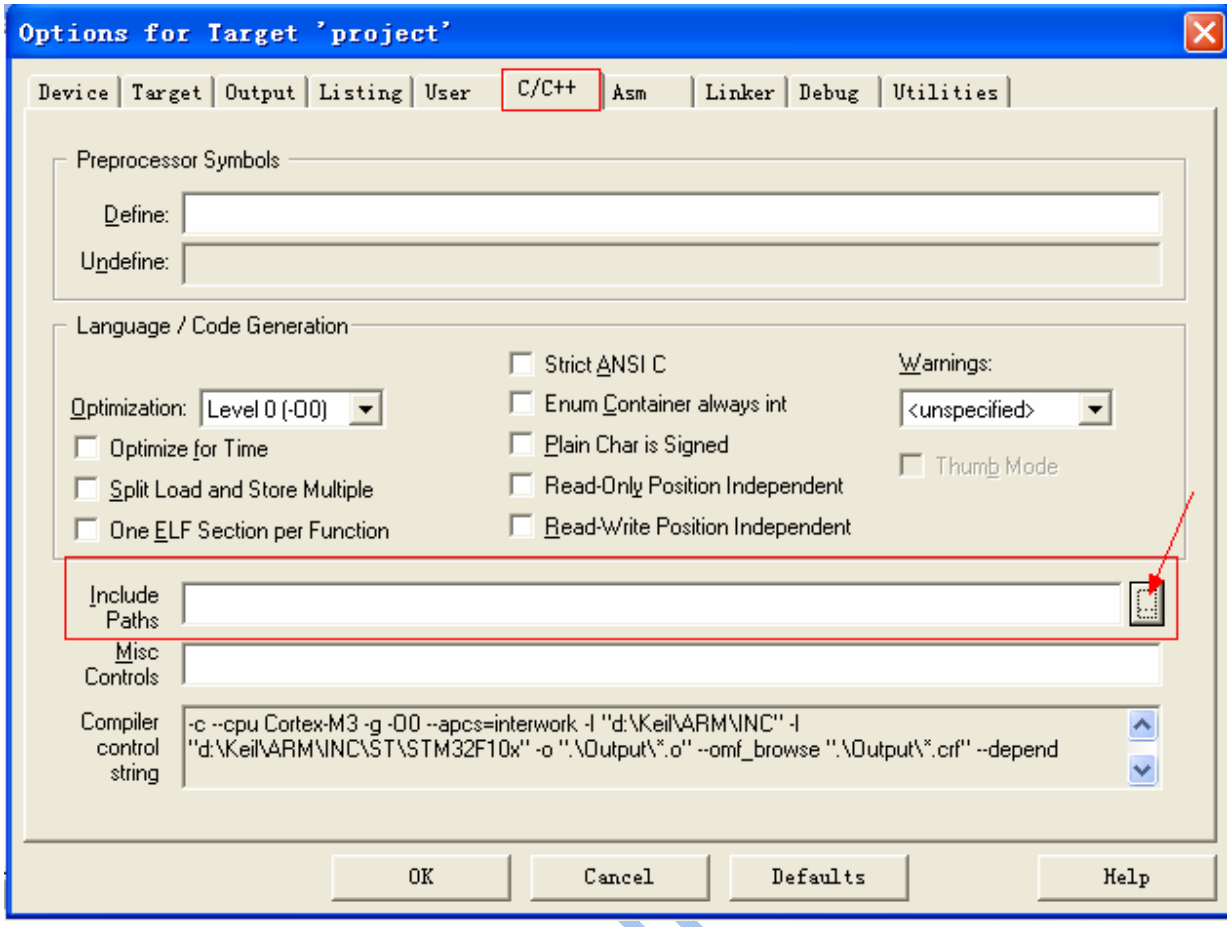
题外话：至于要使用哪个宏，则需要根据具体的芯片来进行选择。那么，这些宏又对应着哪些具体的芯片呢？其实我们可以在《Reference manual》的第九章中找到答案，大家可以自己看看；

如果是 STM32F105xx 和 STM32F107xx，是 Connectivity Line Devices，则宏定义选择的是 STM32F10X_CL；如果是别的型号，则根据 FLASH 的容量来进行选择。可能文字写的有点不太清楚，还是以表格来说明：

宏	MCU型号	FLASH大小
STM32F10X_LD	STM32F101xx STM32F102xx STM32F103xx	16～32 Kbytes
STM32F10X_MD	STM32F101xx STM32F102xx STM32F103xx	64～128 Kbytes
STM32F10X_HD	STM32F101xx STM32F103xx	256～512 Kbytes
STM32F10X_CL	STM32F105xx STM32F107xx	忽略

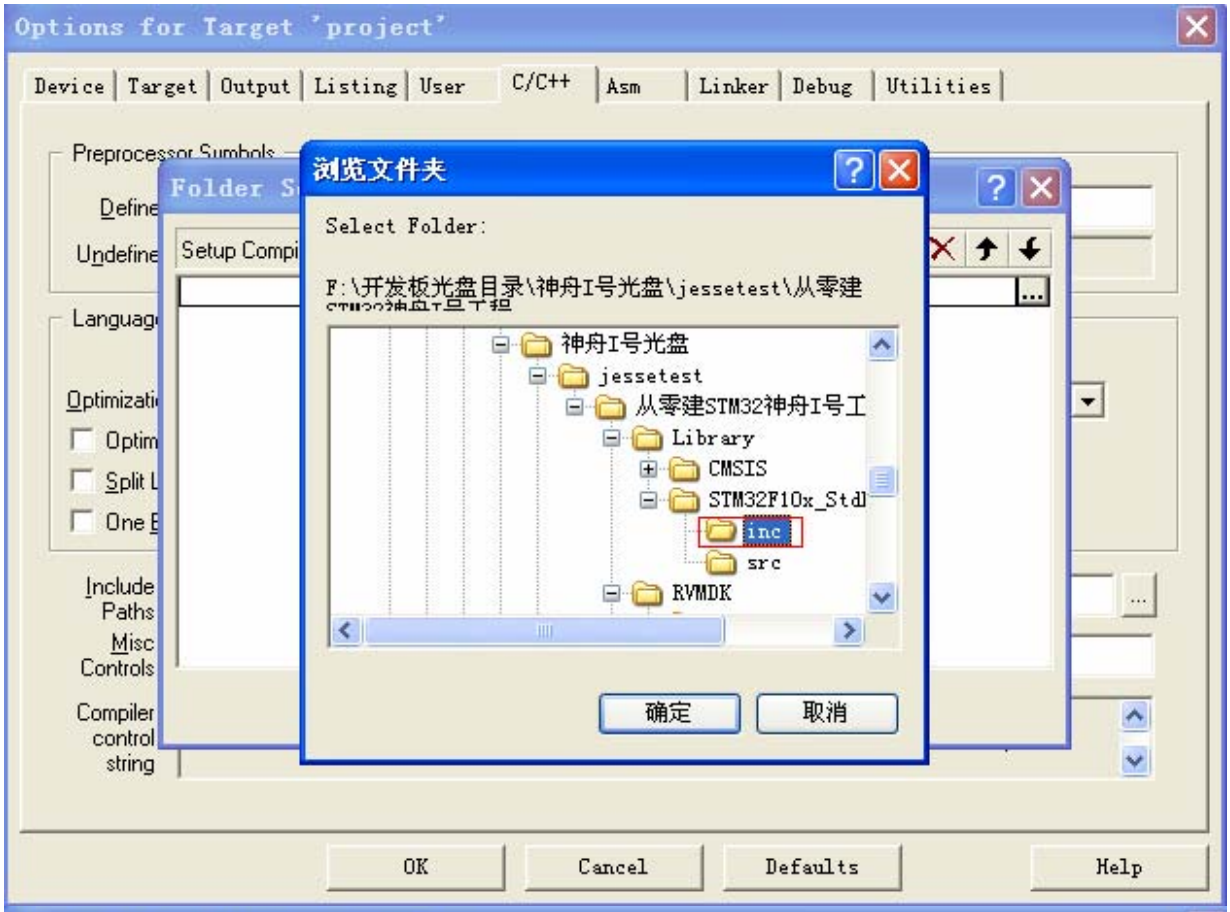
STM32神舟系列开发板

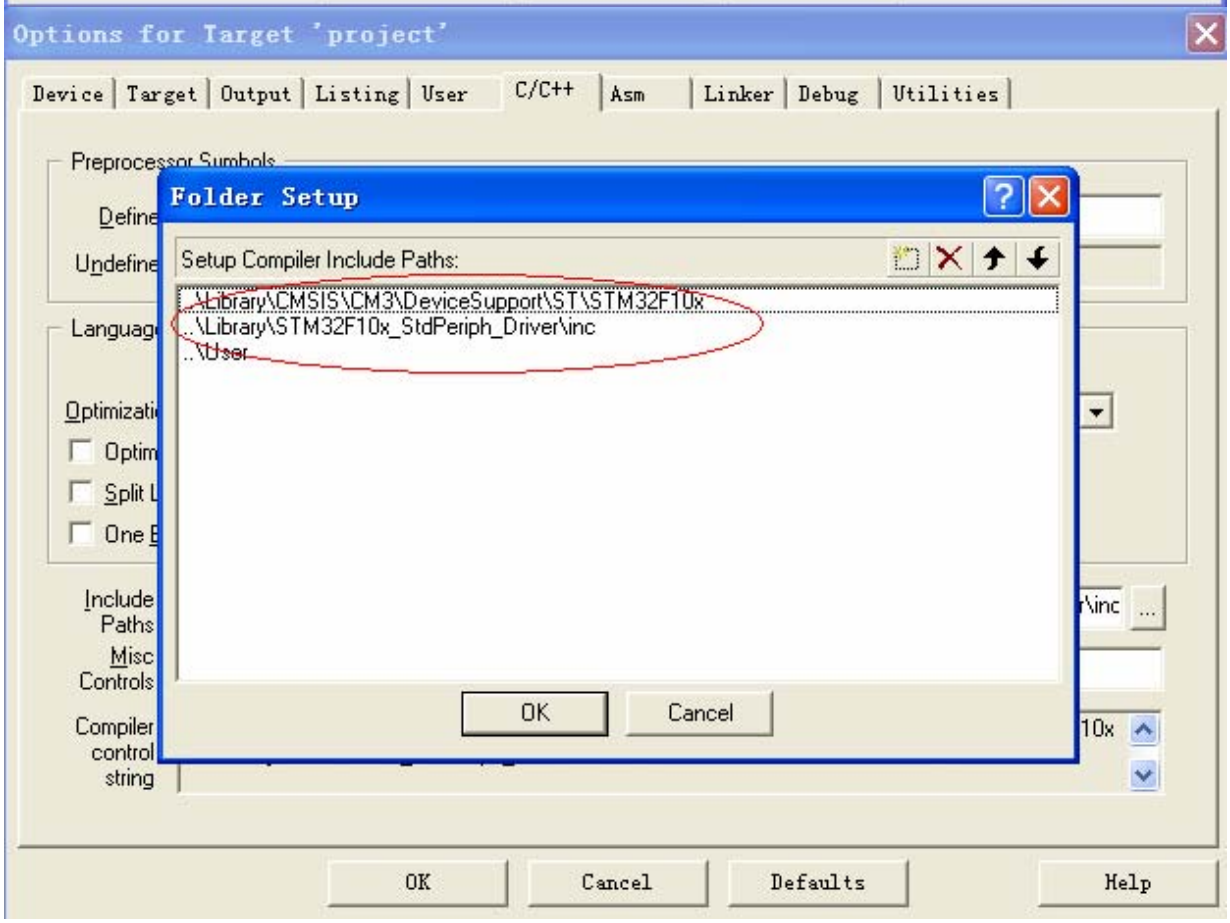
18. 然后再 Include Paths 中点浏览。

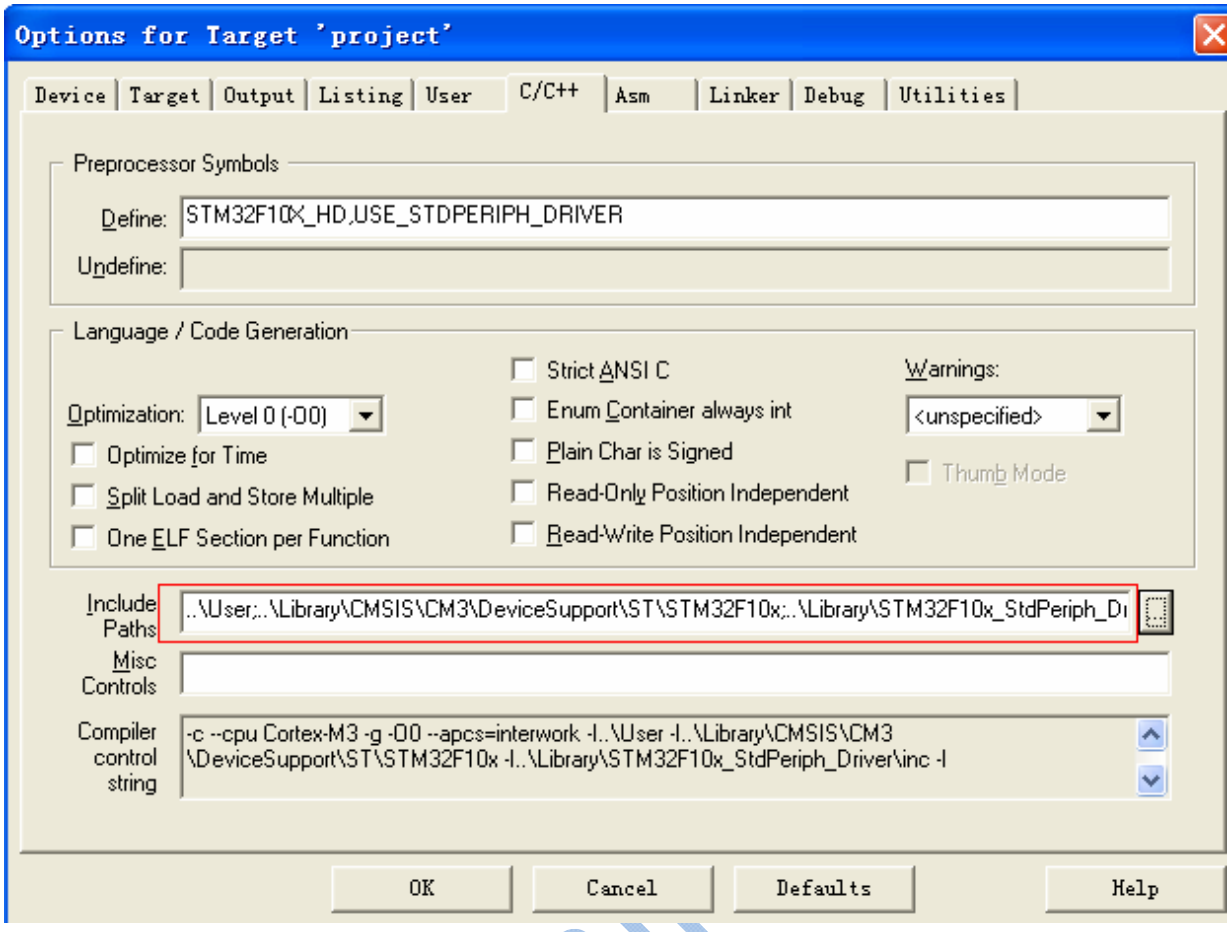


19. 新建一个，点指向目录，然后找到..\User 和 ..\Library\CMSIS\CM3\DeviceSupport\ST\STM32F10x 以及..\Library\STM32F10x_StdPeriph_Driver\inc

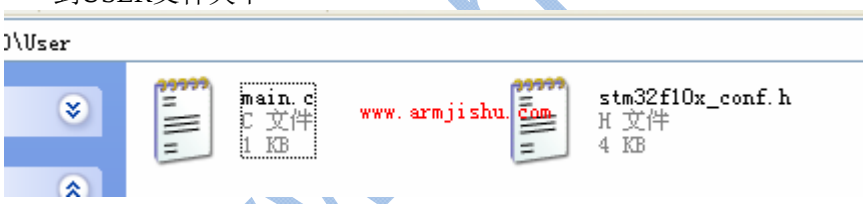
www.armjishu


Options for Target 'project'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Preprocessor Symbols
Define
Undefine
浏览文件夹
Select Folder:
F:\开发板光盘目录\神舟I号光盘\jessetest\从零建
神舟I号光盘
jessetest
从零建STM32神舟I号工
Library
CMSIS
STM32F10x_Std
inc
src
RVMDK
确定
取消
OK
Cancel
Defaults
Help

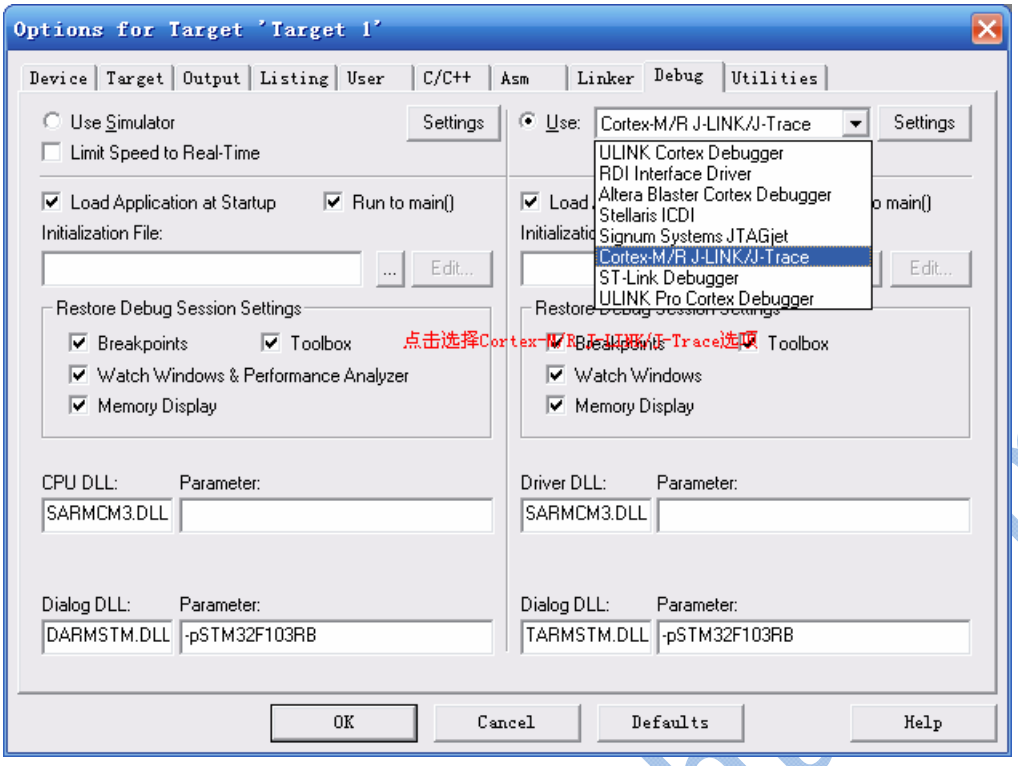

Options for Target 'project'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Preprocessor Symbols
Define
Undefine
Folder Setup
Setup Compiler Include Paths:
..\Library\CMSIS\CM3\DeviceSupport\ST\STM32F10x
..\Library\STM32F10x_StdPeriph_Driver\inc
..\User
OK
Cancel
OK
Cancel
Defaults
Help

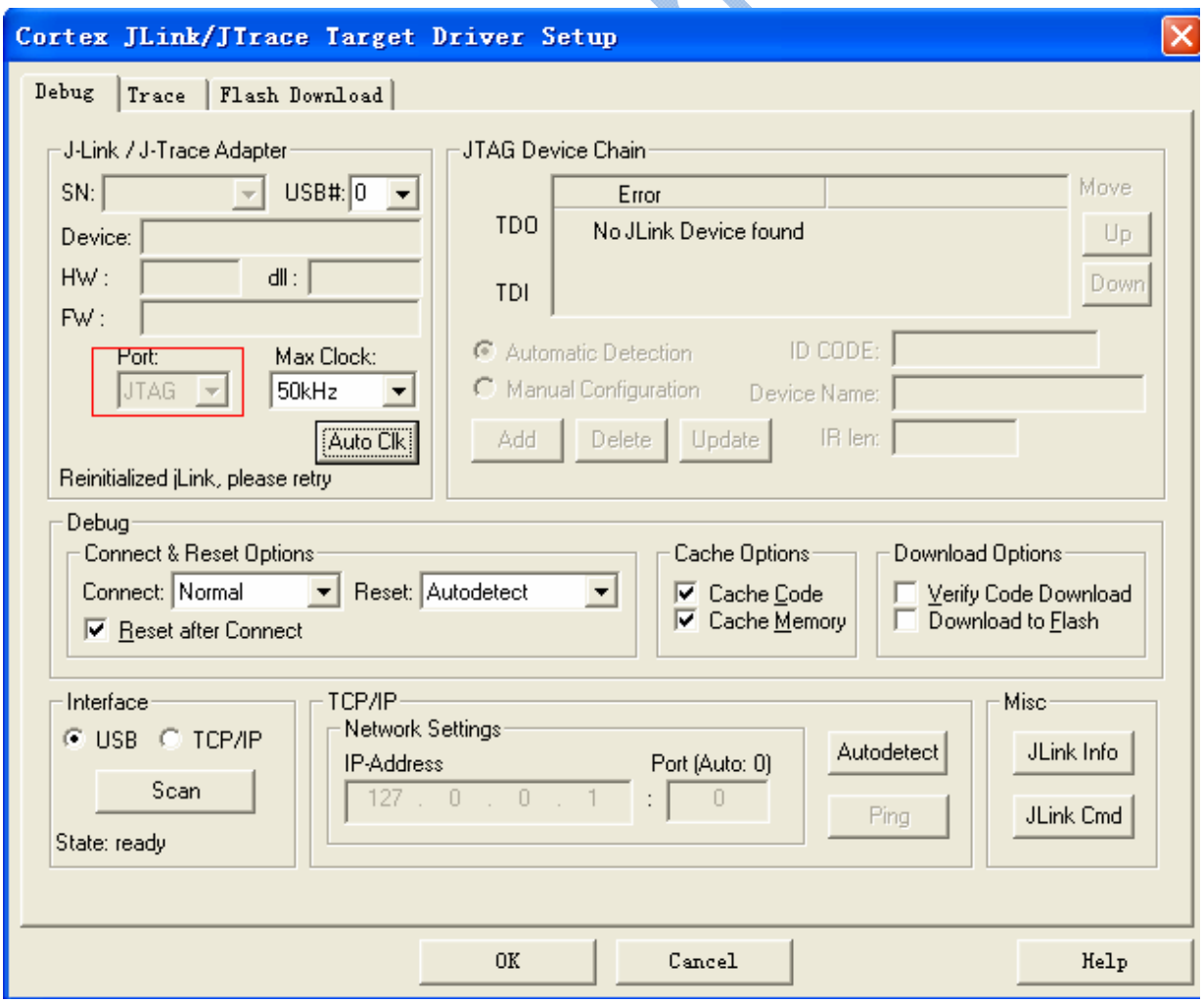


20. 下载STM32F10X_CONF.H文件 ：http://files.cnblogs.com/stm32/stm32f10x_conf.rar，将其解压缩到USER文件夹下



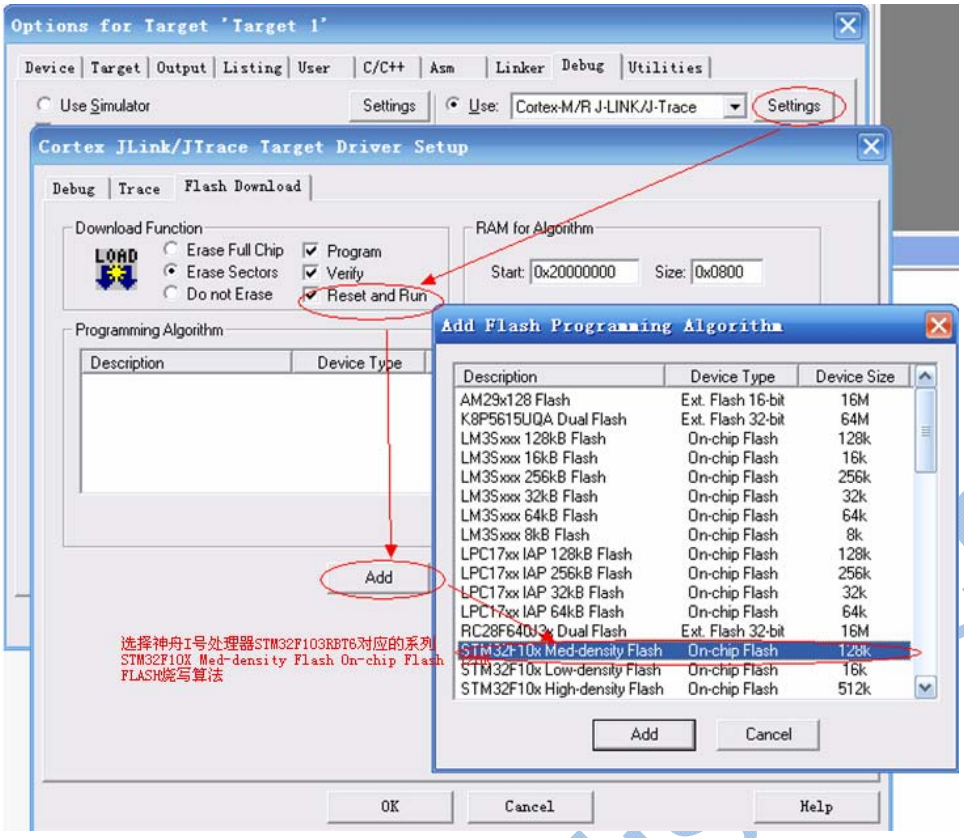
21. 选择 Debug 选项卡，这里是用 JLINK V8 仿真器，所以选择 Use->Cortex-M/R J-LINK/J-Trace,然后选择 Settings,在 port 中选择 SW（亦可用 JTAG），点击 Auto Clk，然后确定：



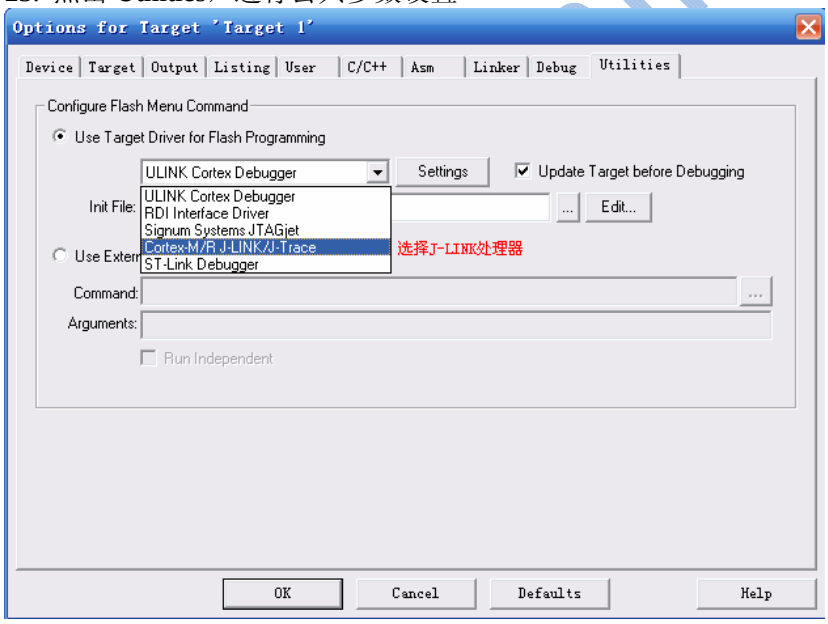


22. 点击右侧的 Setting，配置 JLINK 下载参数，主要是选择目标处理器的系列，添加目标芯片对应

的烧写算法

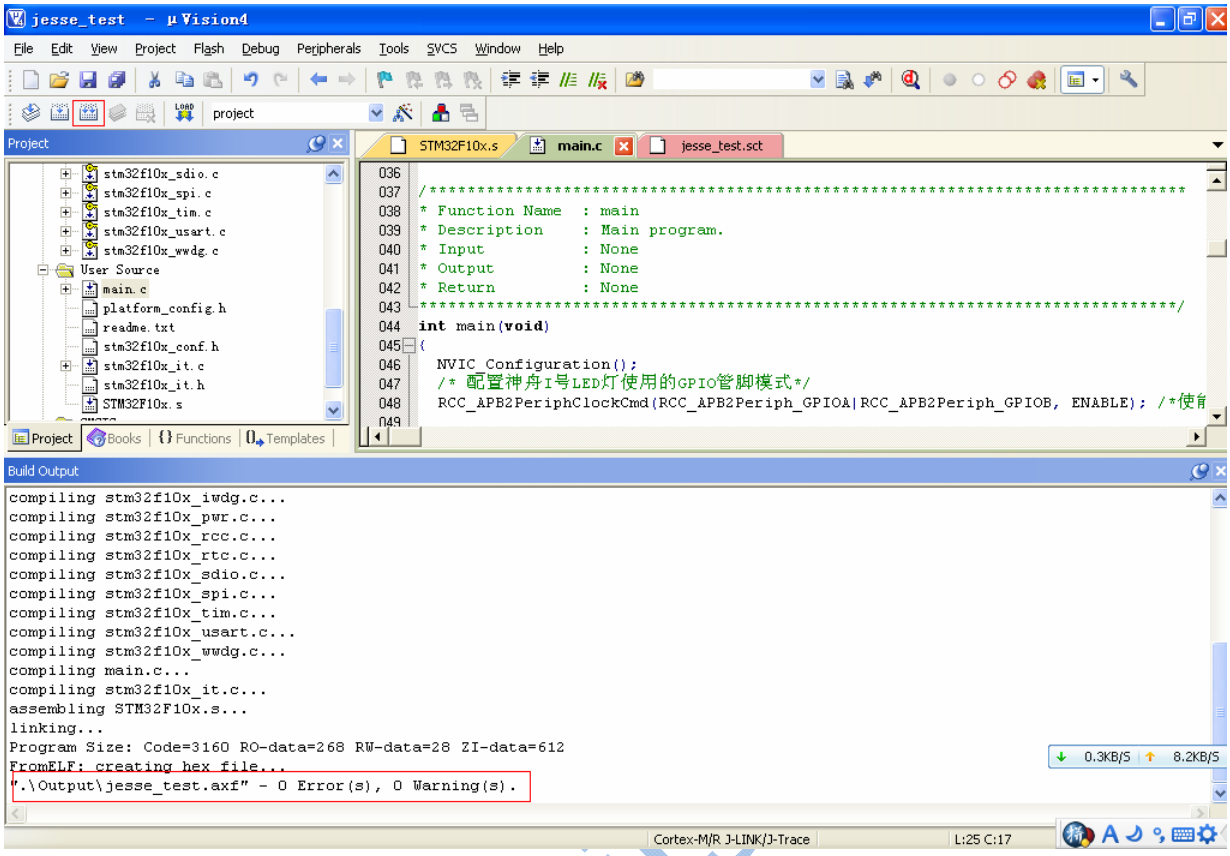


23. 点击 Utilities，进行公共参数设置

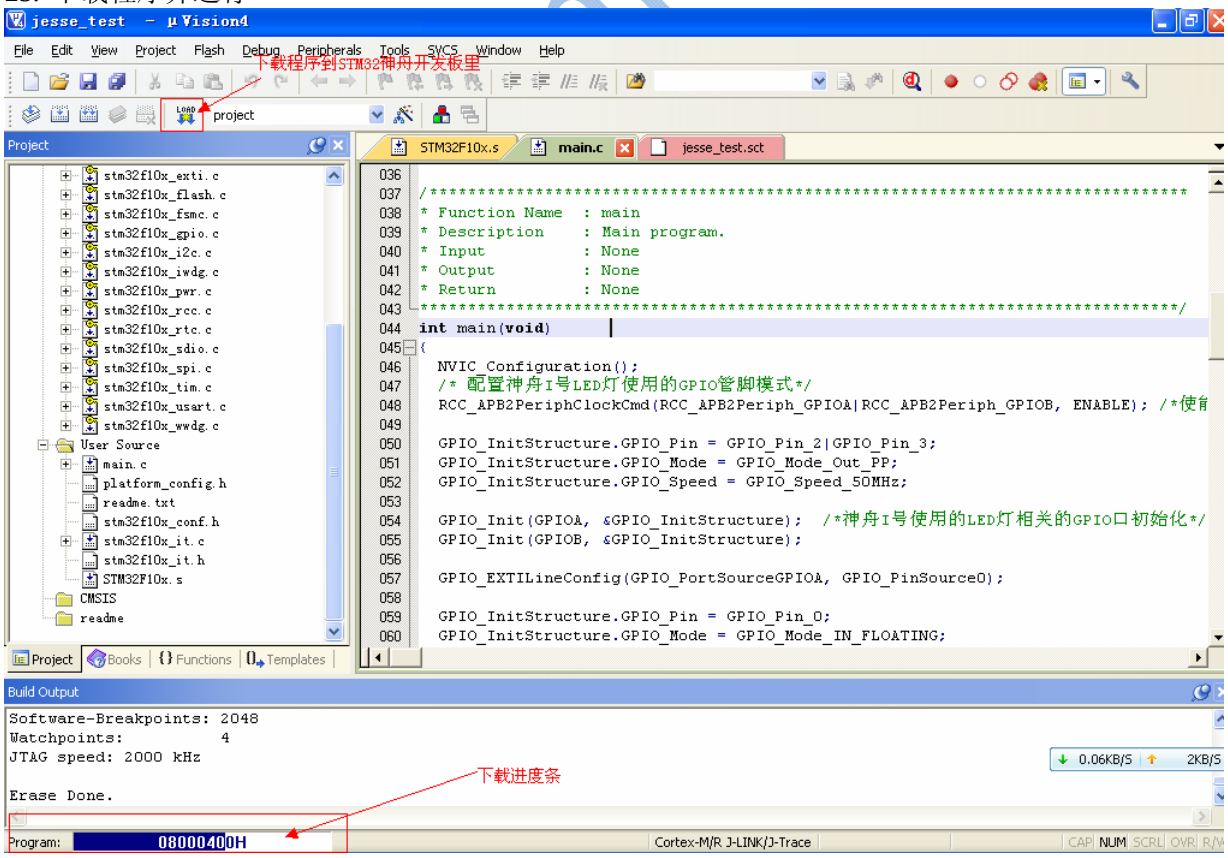


至此，我们已经完成了一个RVMDK工程的参数设置。下面我们主要介绍MDK工程的文件管理方面的功能。

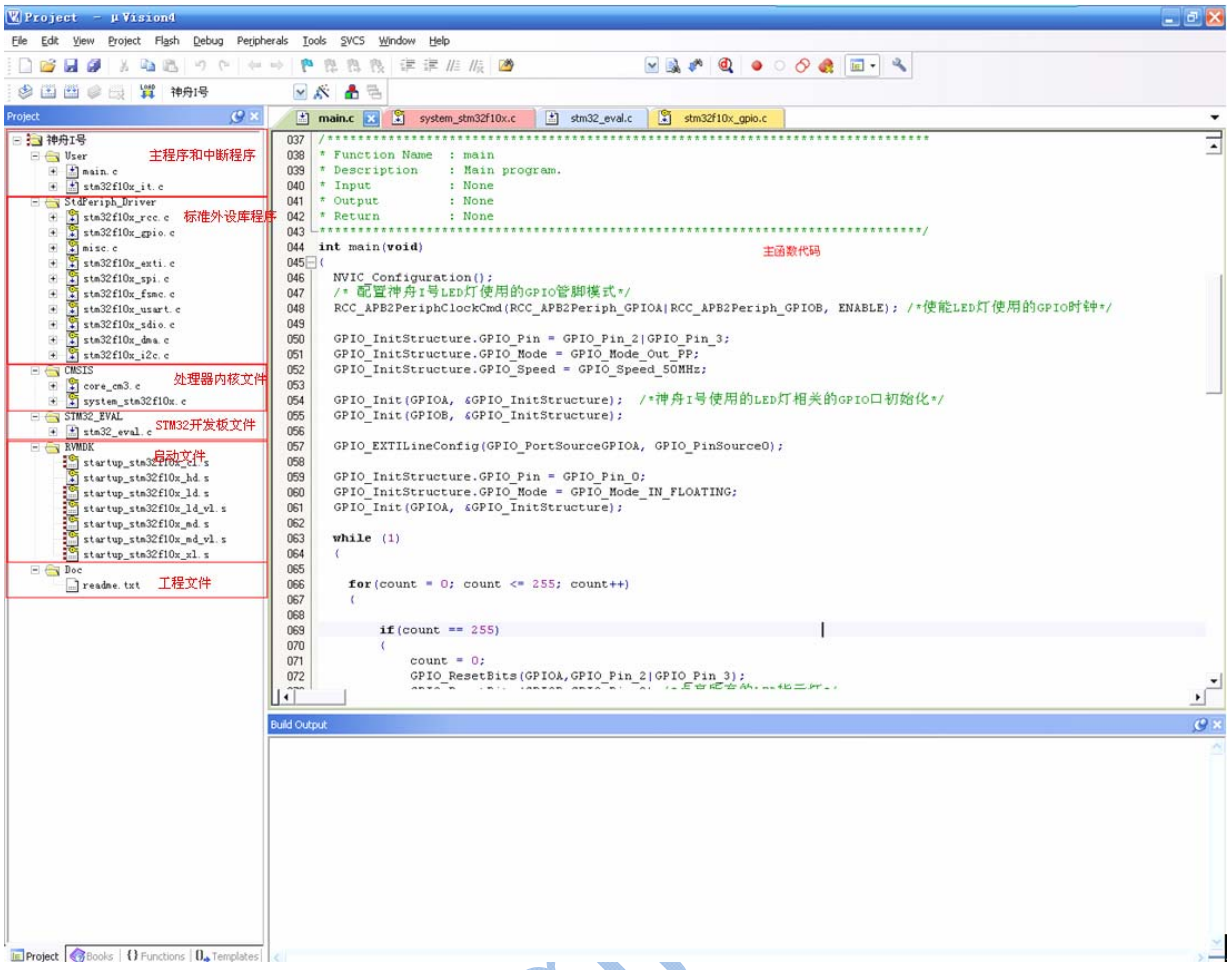
24. 编译工程，成功！



25. 下载程序并运行



26. 并成功运行,最后大家看一下我们的目录结构情况:



在完成以上操作以后，一个MDK工程就设置完成了，只需要依据实际应用要求编写相关的代码即可。下面是神舟I号提供的实验例程的文件分类管理后的效果图。大家可以参考实验例程进行文件分类管理。

4.10 何给神舟I号板供电

神舟 I 号 STM32 开发板一共支持三种供电方式，分别是：

- 使用 USB 接口供电
- 使用 USB 转串口接口供电
- 使用 JLINK V8 供电

4.10.1 使用USB供电

神舟 I 号支持 USB 供电方式，使用 USB 供电时，请使用随神舟 I 号配置的 USB 电缆连接开发板的 USB 接口（J3）和 PC 机的 USB 接口，选择 USB 供电方式。在正常情况下，USB 转串口接口附近的电源指示 LED 灯（DS2），将变亮，表示神舟 I 号已经正常供电。

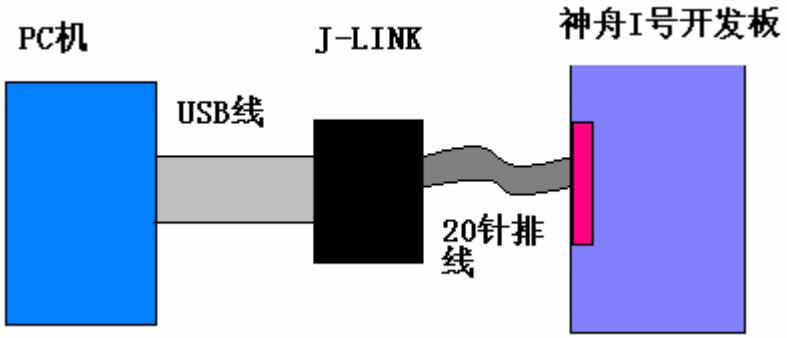
4.10.2 使用USB转串口接口供电

神舟 I 号支持 USB 转串口供电方式，使用 USB 转串口供电时，请使用随神舟 I 号配置的 USB 电缆连接开发板的 USB 转串口接口（J4）和 PC 机的 USB 接口，选择 USB 转串口供电方式。在正常情况下，USB 转串口接口附近的电源指示 LED 灯（DS2），将变亮，表示神舟 I 号已经正常供电。

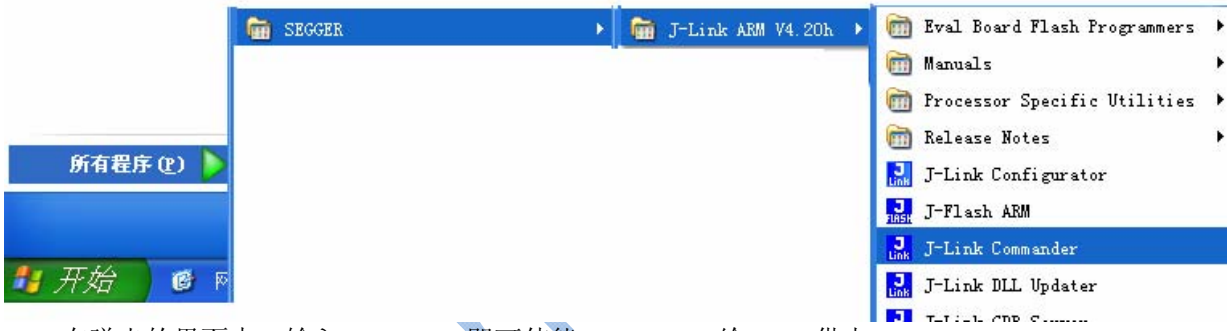
4.10.3 使用JLINK V8供电

除了前面提到的两种供电方式外，神舟 I 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 I 号供电。

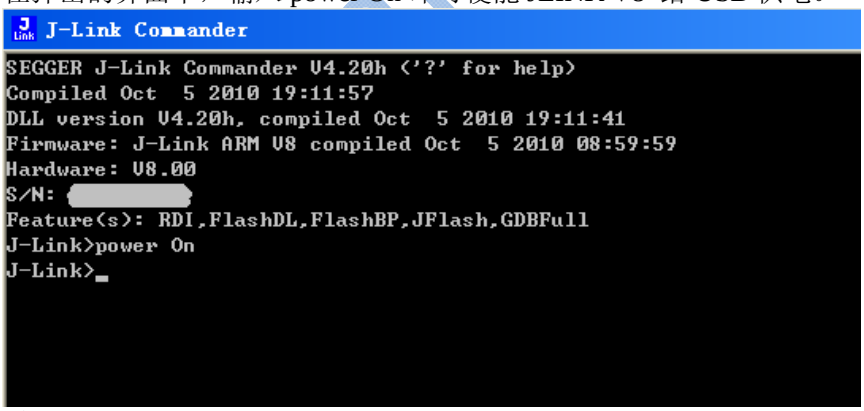
首先按下图连接神舟 I 号开发板，JLINK V8 与 PC 机。



在 PC 机上打开 J-LINK Commander，并敲上命令“power on”，此时 DS2 电源指示灯点亮，表示神舟 I 号已经正常供电（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。



说明：请尽量不要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 I 号 STM32 开发板。对于同时需要 USB 接口，又要 USB 转串口接口的操作，请尽量控制连接在同一台 PC 的 USB 接口上。

4.11 烧录固件程序的三种方法

在代码调试通过以后，我们需要将编译好的程序烧录到开发板中，让它离线运行，烧录固件一般有三种方法：

方法一：在 MDK 集成开发环境中，直接将程序烧录到开发板中。将神舟 I 号与 JLINK 连接好以后，点击 将程序烧录到开发板中，烧录成功后，会输出如下信息。

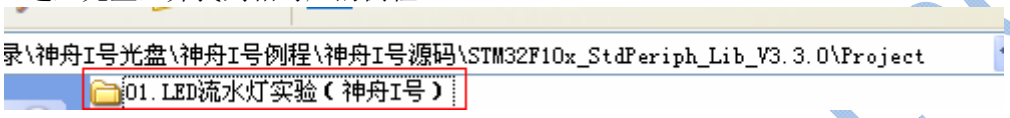
```
Build Output
* JLink Info: TotalIRLen = 9, IRPrint = 0x0011
* JLink Info: Found Cortex-M3 r1p1, Little endian.
* JLink Info: TPIU fitted.
* JLink Info: ETM fitted.
* JLink Info:   FPUnit: 6 code (BP) slots and 2 literal slots
Hardware-Breakpoints: 6
Software-Breakpoints: 2048
Watchpoints:          4
JTAG speed: 2000 kHz

Erase Done.
Programming Done.
Verify OK.
```

方法二：使用J-Flash软件将编译好的固件烧录到开发板中。具体操作见 [如何使用J-FLASH ARM烧写固件到芯片FLASH]。

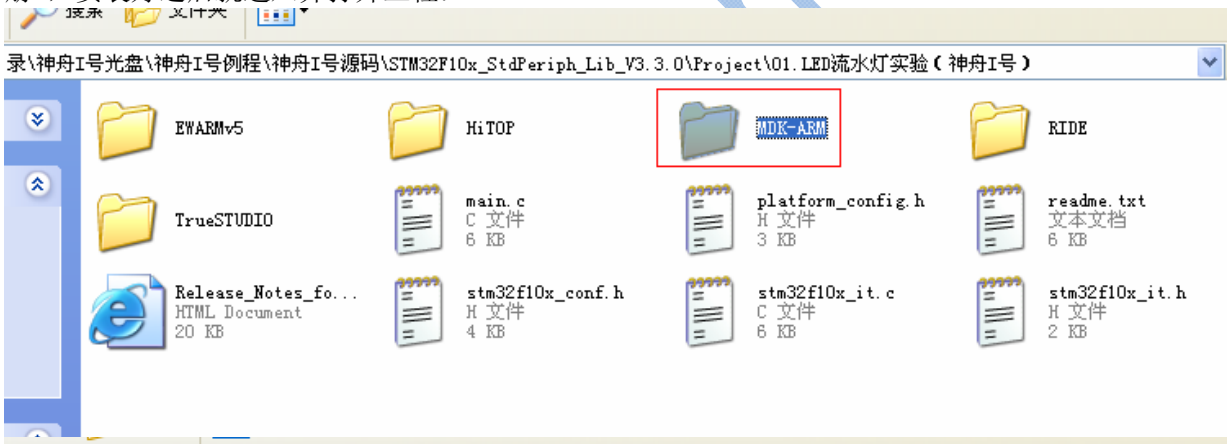
方法三：用串口烧录的方式将固件烧录到芯片里面去

4.12 如何编译和运行光盘里的第一个程序：

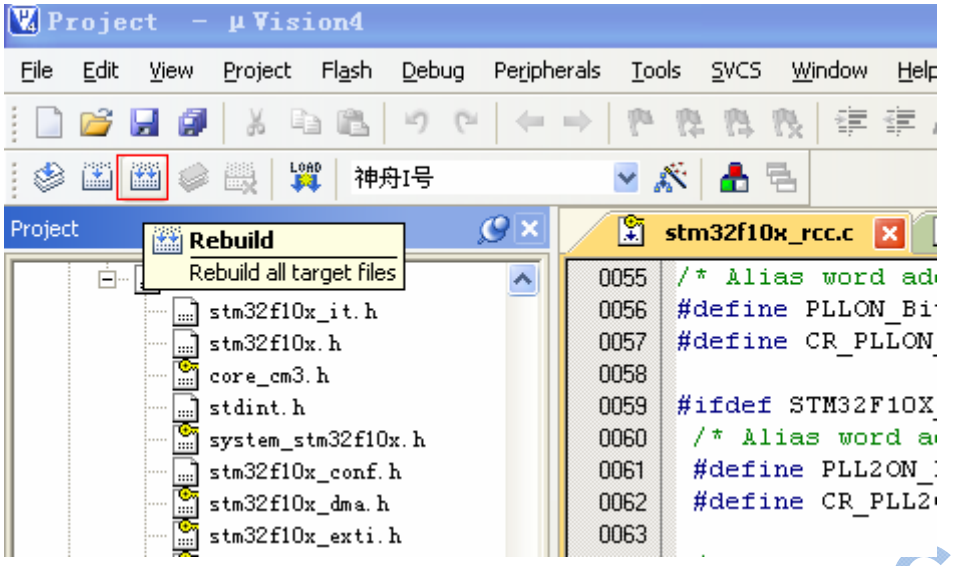
1.进入光盘，并找到相对应的例程：



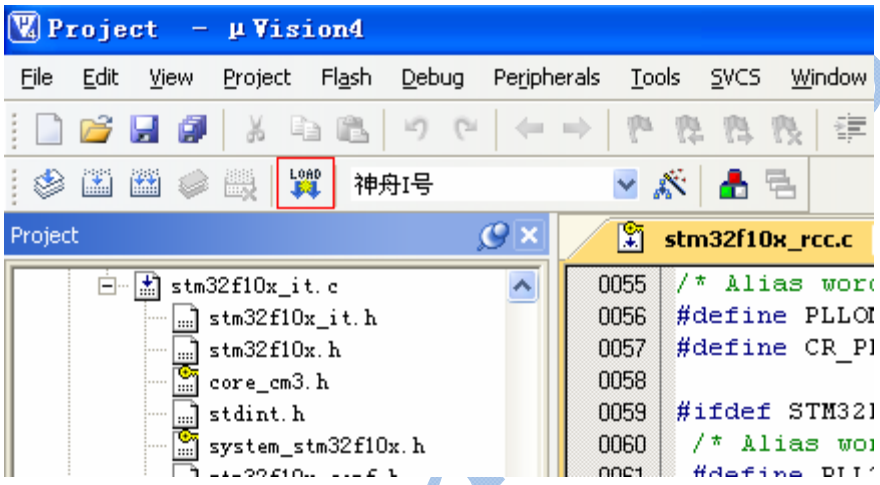
2.当然你应该先安装好 MDK 软件程序，具体安装请参照光盘里对应的目录“MDK 编译软件以及手册”，安装好之后就进入并打开工程：



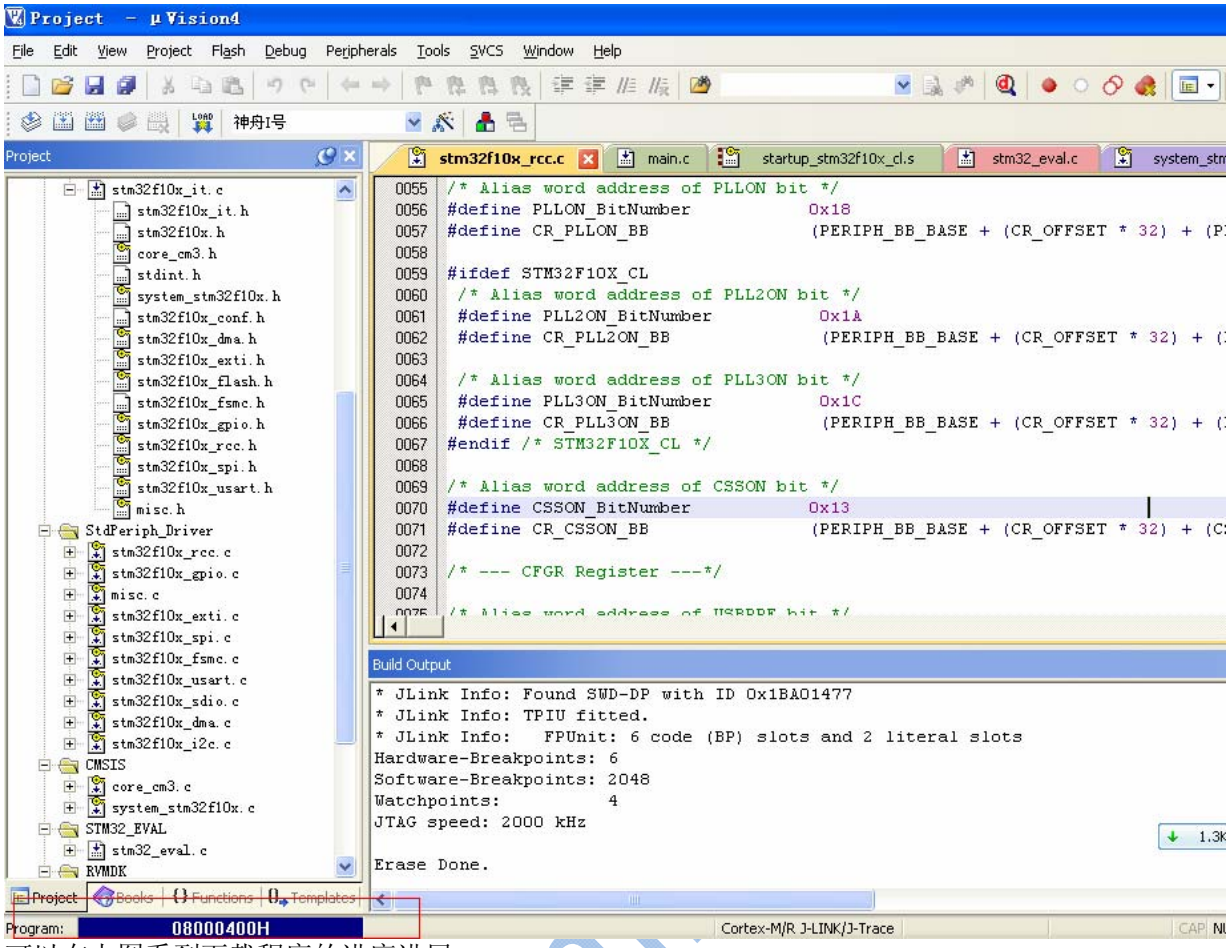
3.点一下重新编译的按钮，



4.将 JLINK V8 仿真器连到开发板上，上电完毕后，按下【LOAD】按钮：



www.armjishu.com



可以在上图看到下载程序的进度进展

另：如果没有 JLINK V8 仿真器下载，那么可以选择用串口的方式进行程序下载，详细请参考串口下载对应章节。

5.请按一下板子上的【复位】按钮，对板子进行复位后，即可看到下载后的程序运行效果：

5.可以看到流水灯的效果

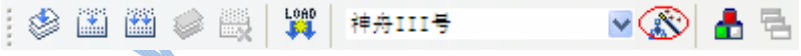
www.armj

4.13 如何用JLINK V8仿真和调试第一个程序：

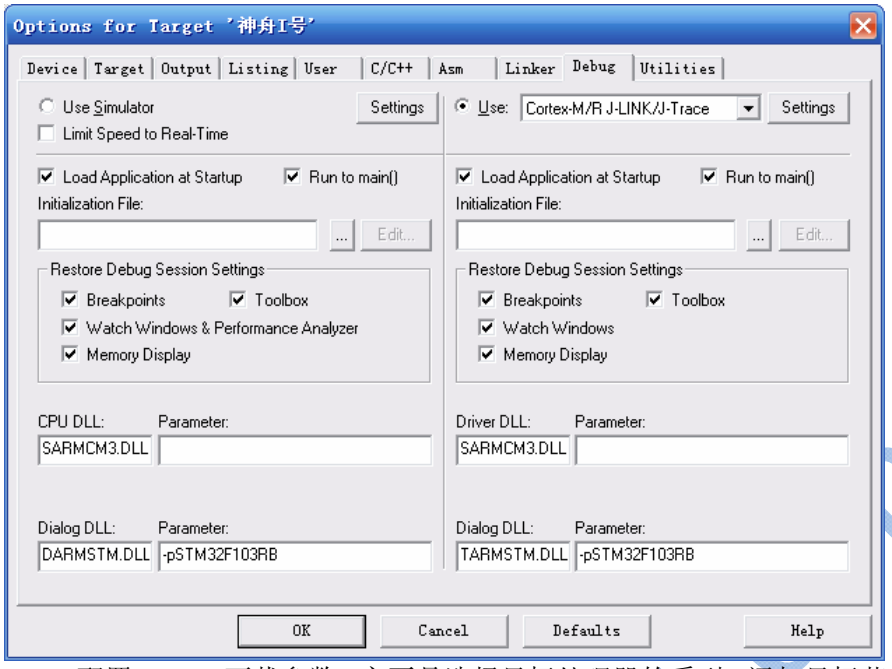
利用串口，我们只能下载程序，并不能实时跟踪，而利用调试工具，比如JLINK、ULINK等就可以实时跟踪程序，使你的开发事半功倍。这里我们以JLINK V8为例，说说如何在线调试。

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。

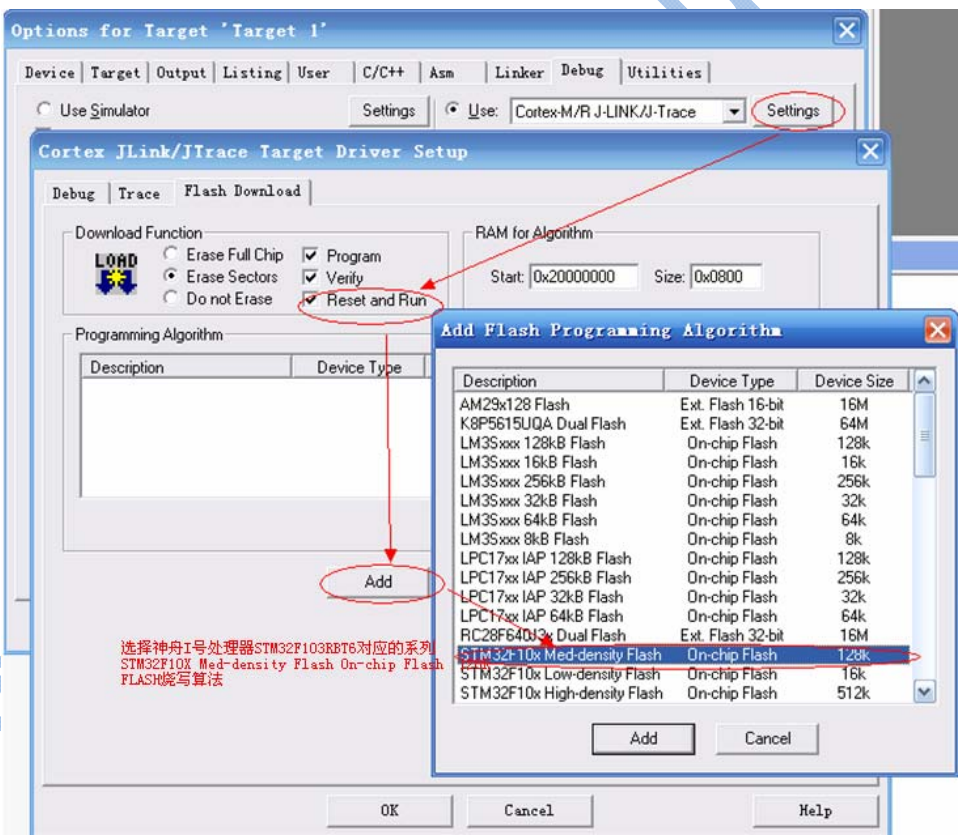
JLINKV8的安装我们这里就不说了，JLINK的光盘里面有详细的资料。在安装了JLINK V8之后，我们接上JLINK-V8，并把JTAG口插到神舟I号开发板上，打开*神舟I号光盘\神舟I号源码\STM32F10x_StdPeriph_Lib_V3.3.0\Project\01.LED流水灯实验（神舟I号）\MDK-ARM*目录下的工流水灯工程文件Project.uvproj，点击



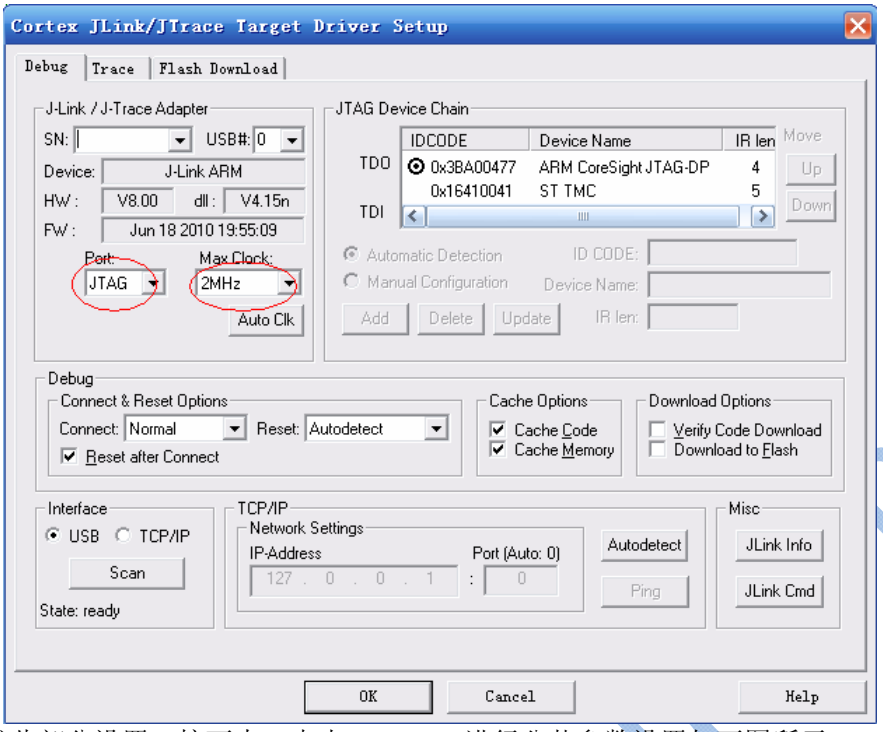
，打开Options for Target‘神舟I号’选项卡，在Debug栏选择仿真工具为Cortex-M3 J-LINK，如下图所示：



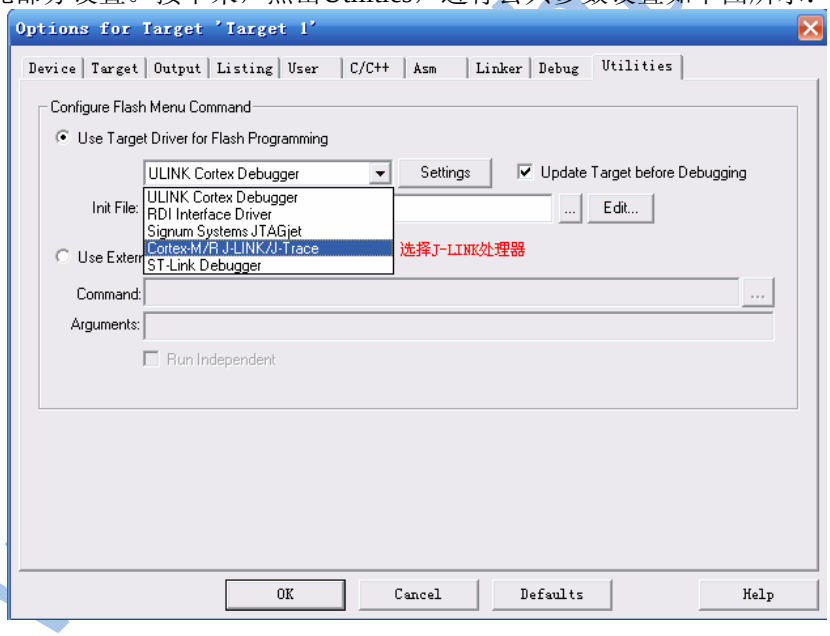
点击右侧的Setting，配置JLINK 下载参数，主要是选择目标处理器的系列，添加目标芯片对应的烧写算法。



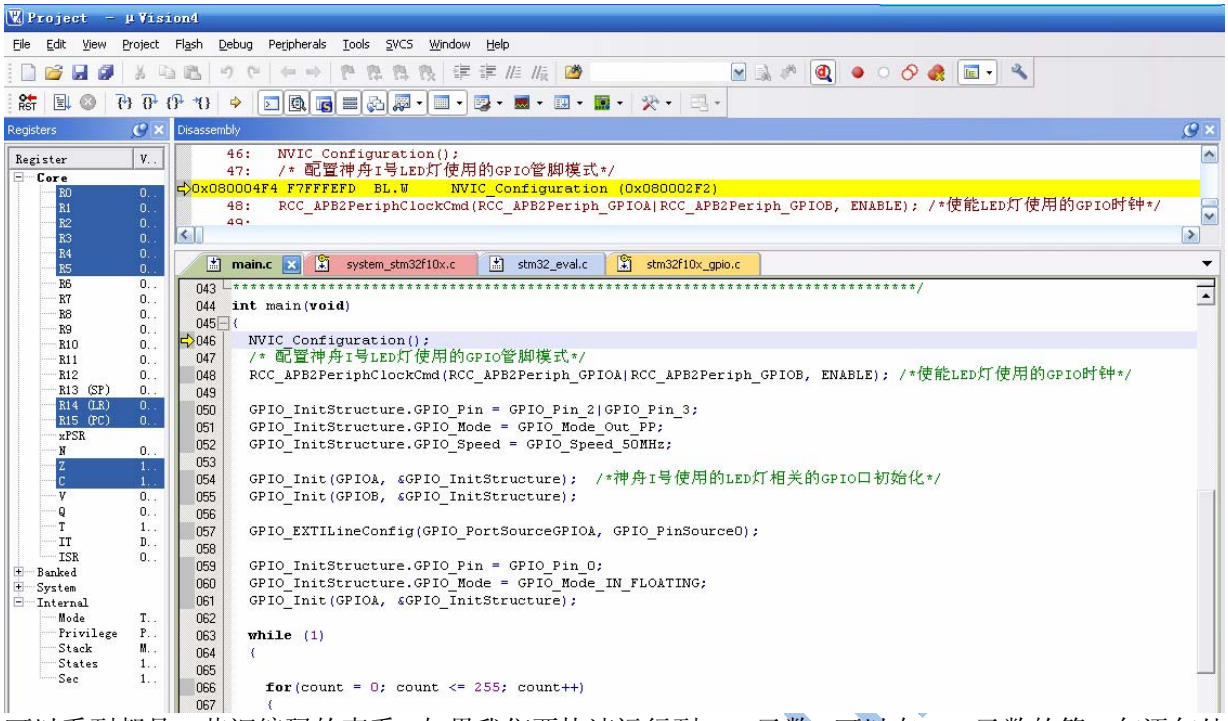
上图中，我们使用J-LINK V8的JTAG模式调试（当然也可以进行SWD模式调试，只要我们在Port处选择SW即可）。Max Clock，可以点击Auto Clk来自动设置，上图中JLINK自动设置最大时钟为2Mhz（注意这里不能设置的太大，如果太大，可能导致JTAG使用不了！但SWD模式的时候，可以设置最大10Mhz）。



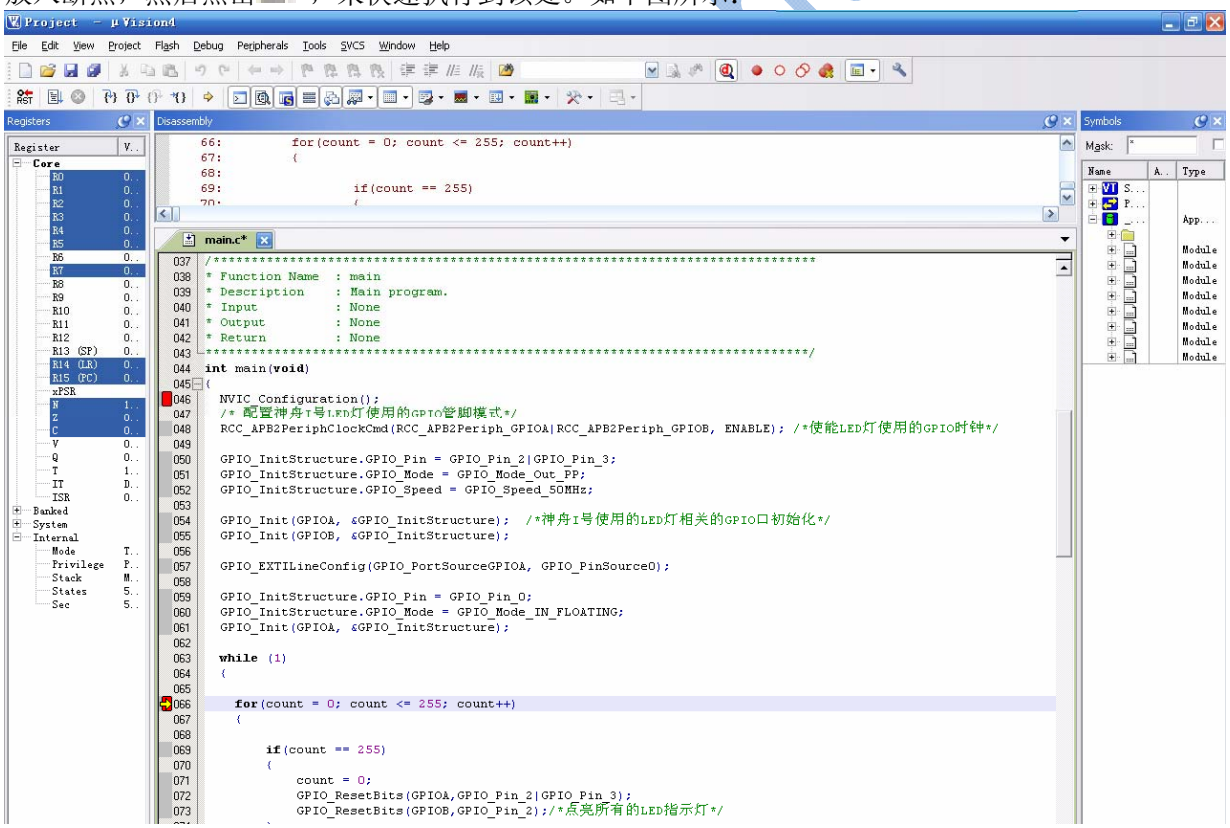
单击OK，完成此部分设置。接下来，点击Utilities，进行公共参数设置如下图所示：



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：



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。

第5章 STM32神舟I号快速入门篇

5.1 理解芯片控制的原理

如果要说做单片机很难吗？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。

无论管脚是输入还是输出，它的目的是传输数据，传输信息，所以要么是高电平，要么低电平，通过 010101 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。

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。

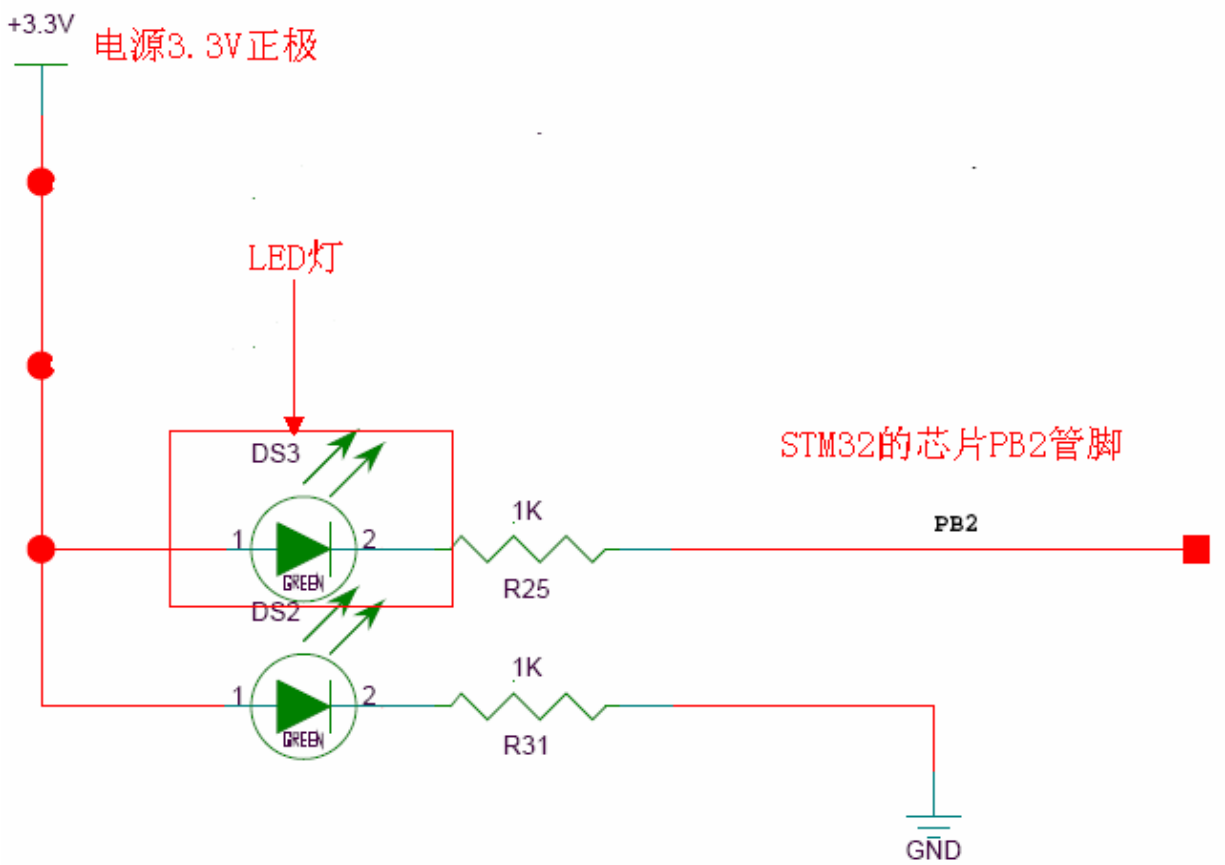
而这些协议，无非就是按照预先规定的表达方式进行通信，比如举个例子，我约定先连续发 4 个 1,，然后再发 4 个 0，就表示芯片 A 要开始发数据给芯片 B 了，即芯片 A 通过它的芯片管脚发‘11110000’给到芯片 B 的时候，那么芯片 B 就知道芯片 A 要给它真正的数据，它就要做好准备工作，准备好之后，芯片 B 就会给芯片 A 一个回应，当芯片 A 收到芯片 B 的回应，就正式开始发数据。

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到的串口协议，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。

5.2 芯片管脚控制LED灯原理图解释

芯片管脚如何控制一个 LED 灯，大家看一下下面的原理图，我们想用 STM32 的管脚 PB2 去驱动 DS3 这个 LED 灯亮，只要使得 PB2 输出低电平，LED 灯就会变亮；如果 PB2 输出高电平，LED 灯就会熄灭。



为什么这么接呢？我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。其次，STM32 单片机（如本实验板中所使用的 STM32F103RBT 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)。

而灌电流（要 VCC 往内输入电流）的方式确非常轻松，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。另外，图中的电阻为 1K 阻值，是为了限制电流，让发光二极管的工作电流限定在 2mA~10mA。

5.3 芯片管脚控制原理（如何阅读芯片手册）

实际上，点亮这个 LED 灯，只需要使得我们的 STM32 芯片的 PB2 管脚输出低电平就可以了，那么如何控制一个 PB2 管脚的状态呢？

我们来举个例子，实际上 STM32 的 PB2 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，打开这个文档 STM32F103中文参考手册.pdf ，然后看到这是一个 524 页的 PDF 手册：

1 / 524 105% 查找

STM32F10xxx参考手册

参考手册

小，中和大容量的 STM32F101xx, STM32F102xx 和 STM32F103xx
ARM 内核 32 位高性能微控制器

然后我们看一下这个文档的目录，可以看到文档的 7.2 节是专门描述 GPIO 寄存器的章节，具体内容大家自己打开文档阅读一下：

目录 STM32F10xxx参考手册

7.2 GPIO寄存器描述 这些都是控制GPIO管脚的芯片内部寄存器 75
7.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E) 75
7.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E) 75
7.2.3 端口输入数据寄存器(GPIOx_IDR) (x=A..E) 76
7.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) 76
7.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E) 77
7.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E) 77
7.2.7 端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) 77
7.3 复用功能I/O和调试配置(AFIO) 78
7.3.1 把OSC32_IN/OSC32_OUT作为GPIO 端口PC14/PC15 78
7.3.2 把OSC_IN/OSC_OUT引脚作为GPIO端口PD0/PD1 78
7.3.3 CAN复用功能重映射 79
7.3.4 JTAG/SWD复用功能重映射 79
7.3.5 ADC复用功能重映射 80
7.3.6 定时器复用功能重映射 80
7.3.7 USART复用功能重映射 81
7.3.8 I^2C 1 复用功能重映射 82
7.3.9 SPI 1复用功能重映射 82
7.4 AFIO寄存器描述 83
7.4.1 事件控制寄存器(AFIO_EVCR) 83

我们进入文档，看其中一个寄存器到底写了些什么：

7.2.1 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

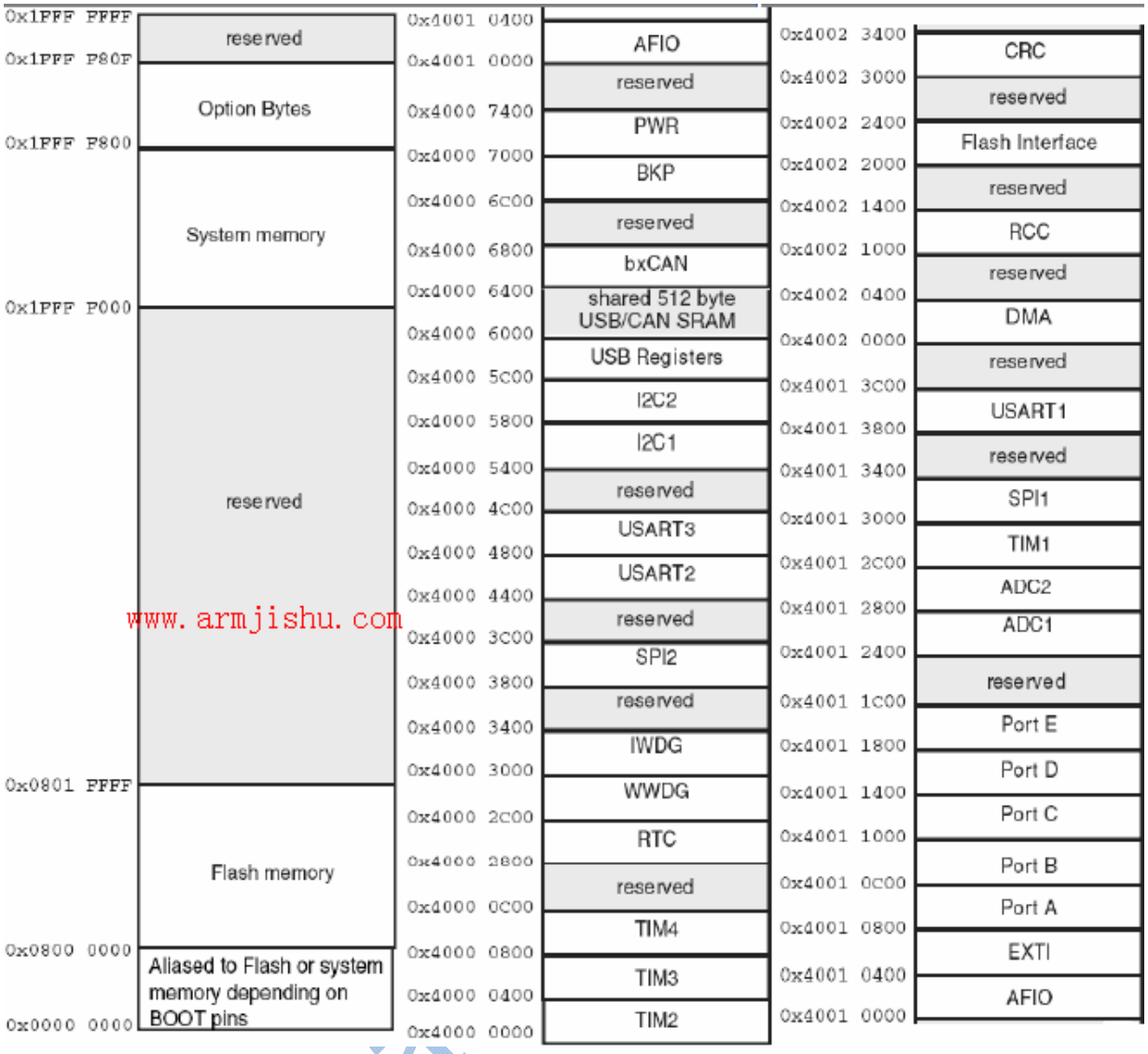
位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 0…7) GPIO端口模式的配置 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 1）输入有4种 www.armjishu.com 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式 2）输出有4种
位29:28 25:24 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) 软件通过这些位配置相应的I/O端口，请参考表15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端口速率配置

可以看到，从 0~31 总共有 32 个位的寄存器，可以去设置；其中 0、1 和 4、5 和 8、9 和 12、13 位以此类推到 28、29 每两个位为一个组，每组有 4 个状态，每个状态都对应不同的模式，比如这 2 个位都是 1 则表示‘输出模式，最大速度 50MHz’，2 个为都是 0 则表示‘输入模式（复位后的状态）’；

上面 CNFy 配置端口 8 种模式也是同样的道理，至于为什么有这八种模式，大家可以去查阅一下手册。

是不是看得有点头晕了，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），从而达到我们控制一个 LED 灯亮和灭。

那么我们怎么来改变这些寄存器呢？那寄存器的值保存在什么位置呢？答案就是在 SRAM 里，在内存映射里。从芯片一上电，正常工作开始，所有寄存器的值都会在实时保存在内存中的一个固定地址，这个固定地址也是芯片厂商一开始就定好了的，不能改变，大家看下图：



www.armjishu.com

比如我们要访问 GPIO 管脚 PB2，那么首先就要找到端口 B 的位置，即 Port B；我们可以从上图找到，Port B 的位置在内存中的 0x40001 0c00 这个地址，那么再看一下手册的寄存器描述

7.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址：0x00

复位值：0x4444 4444

看到一个偏移地址没？对，GPIOB_CRL 这个寄存器的地址在哪呢？就在 Port B 的位置+偏移地址就可以算出它的地址，即：

GPIOB_CRL 寄存器地址：0x40001 0c00 + 0x00 = 0x40001 0c00

我们再看一个地址：

7.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)

偏移地址：0x04

复位值：0x4444 4444

GPIOB_CRL 寄存器地址：0x40001 0c00 + 0x04 = 0x40001 0c04

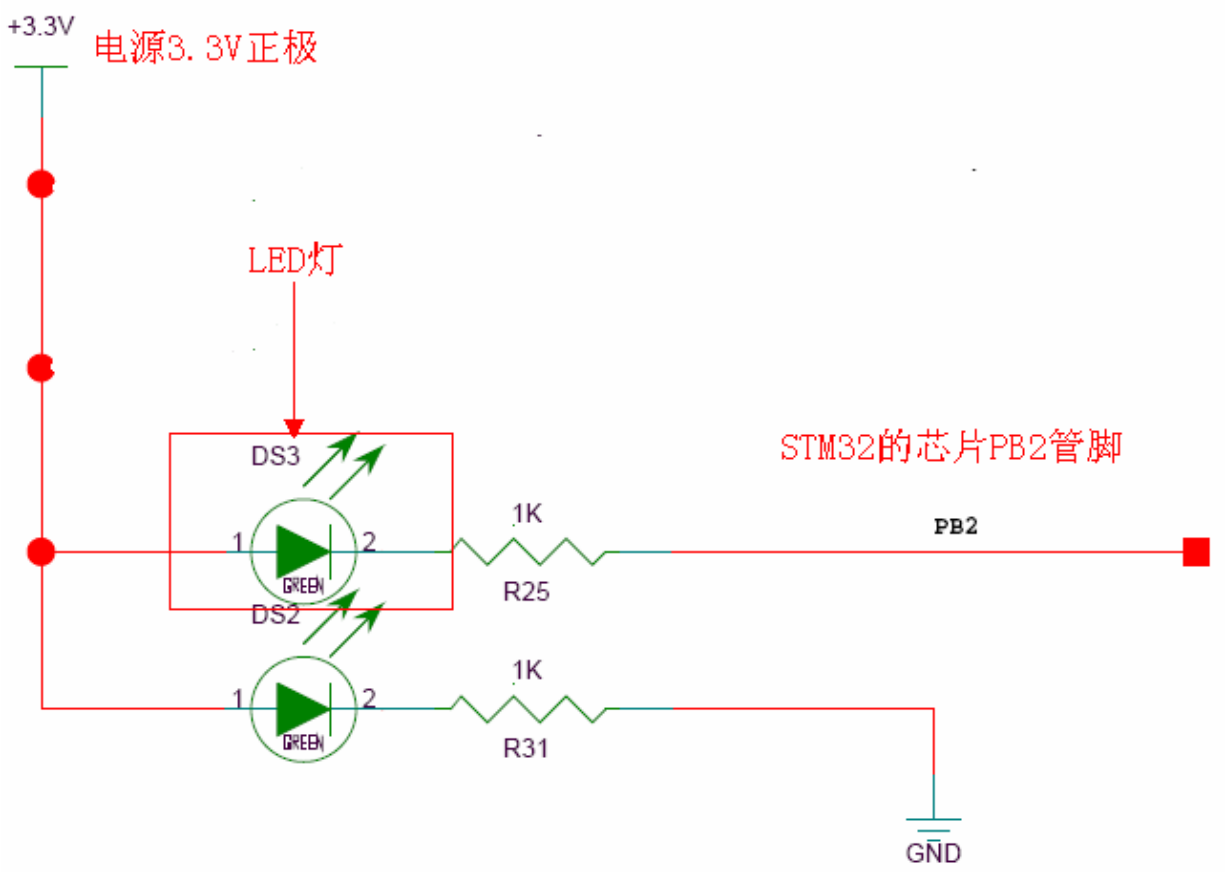
也就是说，当我们访问 0x40001 0c04 这个地址所指向的内容时，实际上就是在访问 GPIOB_CRH 这个寄存器了，就这么简单。

5.4 实际例程详解

5.4.1 原理图说明

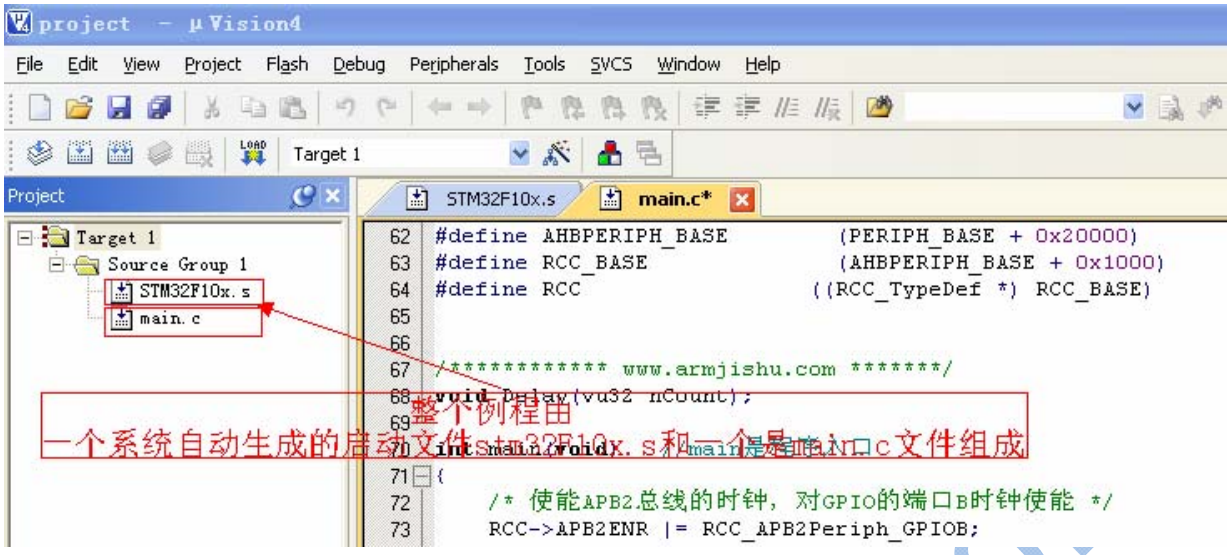
同样的原理，这样，我们就可以访问到所有的寄存器了，然后通过改写这些寄存器的值，控制芯片做不同的操作。好了，下面我们正式写个例程来感受一下.这个例程用 C 语言来修改这个内存地址的内容，从而控制寄存器，通过寄存器控制 STM32 芯片的 PB2 管脚使得一个灯亮和灭的。

原理图如下，上面已经有介绍：



5.4.2 超级简单的例程结构（只有一个main.c文件）

这个例程，我们将所有的代码都写到了一个 mian.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：



5.4.3 main.c全部代码粘贴：

以下是 main.c 的源文件，读者可以直接粘贴编译：

```
/******************* 此段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)

/************ 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 GPIOB_BASE               (APB2PERIPH_BASE + 0x0C00)
#define GPIOB                   ((GPIO_TypeDef *) GPIOB_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 B 时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- 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）---*/
    GPIOB->CRL &=   0xFFFFF0FF;
    GPIOB->CRL |=   0x00000300;

    while (1)
    {
        GPIOB->BRR = GPIO_Pin_2;
        Delay(0x2FFFFF);
        GPIOB->BSRR = GPIO_Pin_2;
        Delay(0x2FFFFF);
    }
}

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 **********/

5.4.4 程序初始化代码定义说明（包含芯片手册阅读方法）

下面开始，分别讲解初始化代码中的声明部分，这个部分读者可以根据自己的习惯做不同的声明，我们这里的声明方式是从库中摘抄过来的，具备一定的参考性，以后自己可以同样的思想去看库函数的例程：

```
#define      __IO    volatile                    /*!< defines 'read / write' permissions   */
typedef unsigned          int uint32_t;
typedef __IO uint32_t  vu32;
typedef unsigned short    int uint16_t;
```

1. 定义声明，用一个代号表示，方便程序和理解

```
#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 */
```

2. 每一个16进数都是由4个二进制组成的，比如0xF==0x1111；那么这里用2个字节，4位16进制数，共16个二进制数分别表示一个端口16个不同的管脚

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

3. 设置端口的偏移量

1. 定义声明这个不用说，一般程序都有
2. 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。

www.armjishu.com

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

这个16位的功能，可以用其他寄存器完成，比如GPIO BSR

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

7.2.6 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

位31:16	保留。
位15:0	**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

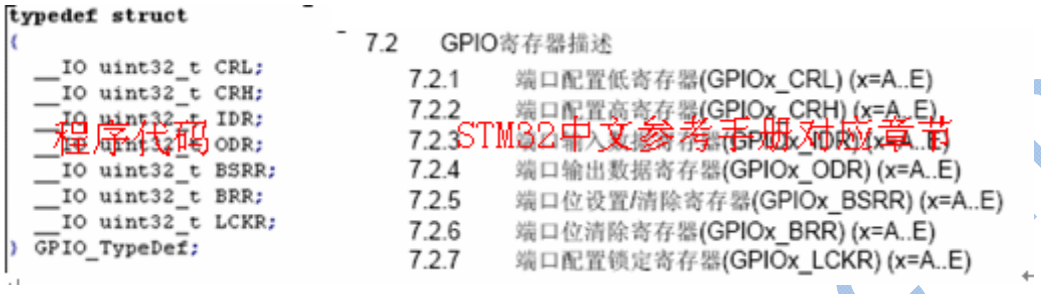
上面这 2 个寄存器都是 32 位的，实际上我们可以只对前面 16 位（0~15 位）控制就能达到我们控制 GPIO 管脚高低电平的目的了 。

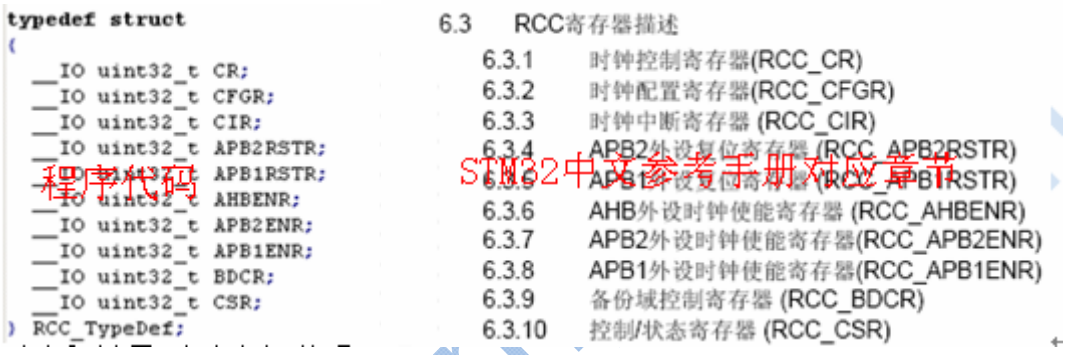
3. 设置端口偏移量我们后面再说

5.4.5 程序代码的struct与芯片手册寄存器如何对应

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：





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 神舟系列开发板的官方工程师。

5.4.6 C语言程序代码如何真正访问芯片内部寄存器

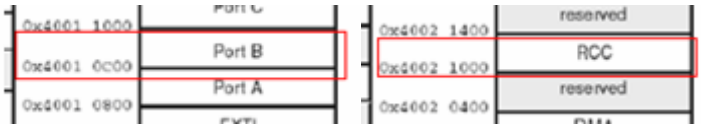
C 语言程序代码如何真正访问芯片内部寄存器的呢？大家看下面这些定义：

```
/********* GPIOB管脚的内存对应地址 *******/
#define PERIPH_BASE            ((uint32_t)0x40000000)
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x10000)
#define GPIOB_BASE             (APB2PERIPH_BASE + 0x0C00)
#define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)

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

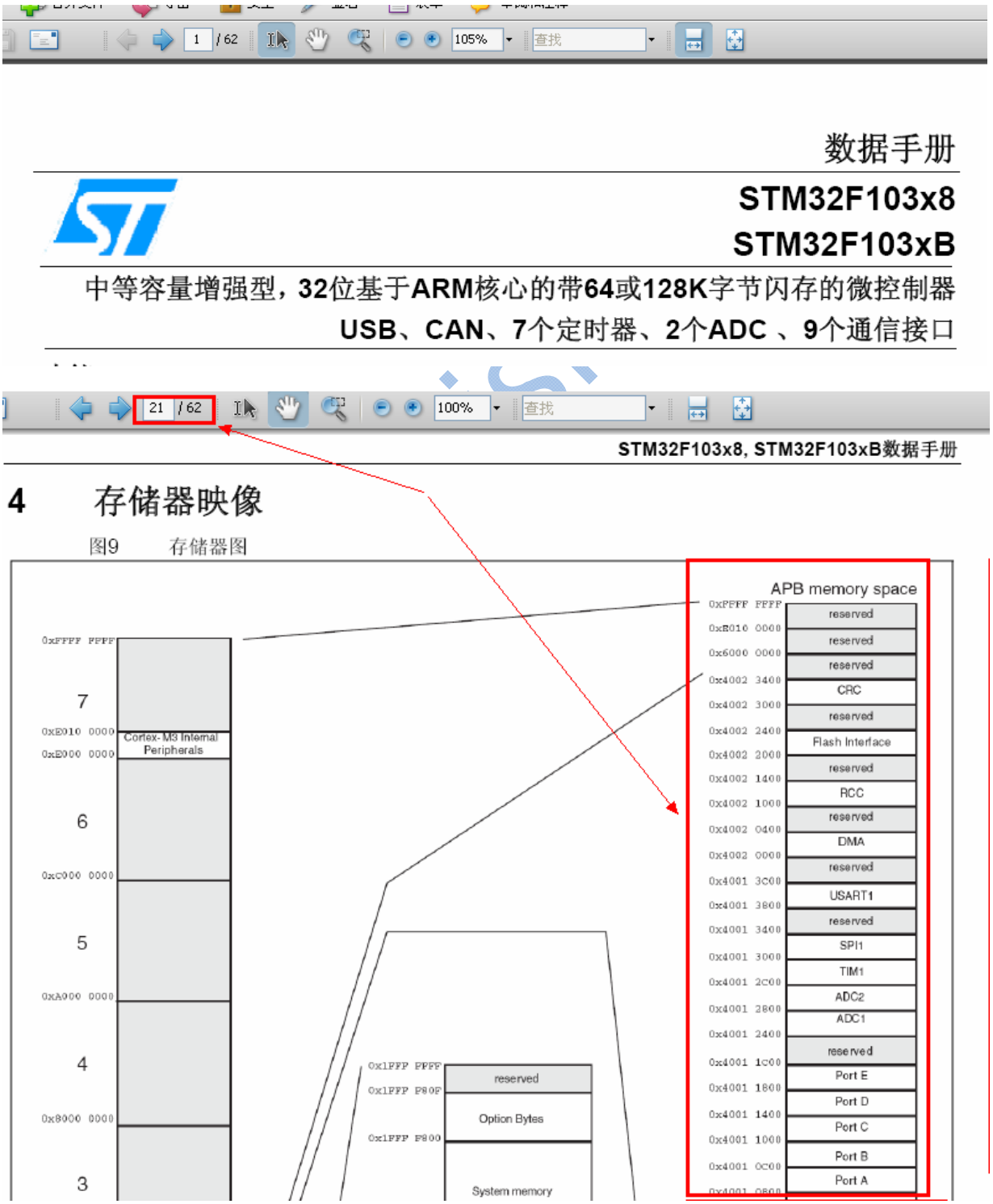
通过这几个 define 可以算出来一下地址：

GPIOB = 0x4000 0000 + 0x1 0000 + 0x0C00 = 0x4001 0C00 刚好与 PortB 在内存中的位置对应上

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



更多内存映射可以找 STM32F103XB 的数据手册资料，如下图，可以详细的进行了解了解：



5.4.7 mian函数剖析

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B时钟使能 */
    RCC->APB2ENR |= RCC_APB2Periph_GPIOB;

    /*-- 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）---*/
    GPIOB->CRL &=  0xFFFFF0FF;
    GPIOB->CRL |=  0x00000300;

    while (1)
    {
        GPIOB->BRR = GPIO_Pin_2;  /*点亮所有的LED指示灯 GPIO_ResetBits(GPIOB,GPIO_Pin_2)*/
        Delay(0x2FFFFF);
        GPIOB->BSRR = GPIO_Pin_2;/* 熄灭LED指示灯 GPIO_ResetBits(GPIOB,GPIO_Pin_2)*/
        Delay(0x2FFFFF);
    }
}

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

步骤1　步骤2　步骤3　步骤4

步骤 1：使能 APB2 总线的时钟。

对 GPIO 的端口 B 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 GPIOB 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功能块激活，激活后就可以使用。

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 GPIOB_CRL 寄存器，我们将这个寄存器设置为了 0x0000 0300。

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

GPIOB_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 GPIOB_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 GPIOB_BSRR 和寄存器 GPIOB_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 GPIOB_ODR 的代码，这个留个大家来做练习吧。

7.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E)

地址偏移：0Ch

复位值：00000000h

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。

5.4.8 库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。

第6章 STM32神舟I号功能部件基础篇

经过前两章的学习，我们对神舟I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。本章将通过二十八个实例，向大家介绍STM32的强大功能，及开发实例。

6.1 STM32神舟I号实验例程结构

前面，我们介绍了 MDK 软件的基本操作，包括工程建立，代码编译仿真和下载等，在这一章节，我们以神舟 I 号入门例程——流水灯为例，说明神舟 I 号示例例程的结构。

由于神舟 I 号所有例程都是基于 STM32F10x_StdPeriph_Lib_V3.3.0 库编写的，所有实验例程都统一结构，与官方的库结构保持一致，方便实验例程统一管理。在这里以 LED 流水灯例程为例，说明实验例程的结构，以便大家能够顺利的使用和验证例程。

首先，打开*神舟 I 号光盘\神舟 I 号源码*，可以看到源码的目录结构如下：

其中 STM32F10x_StdPeriph_Lib_V3.3.0 为 ST 官方提供的库文件，它的下级文件夹含义如下：

文件与文件夹名	内容描述

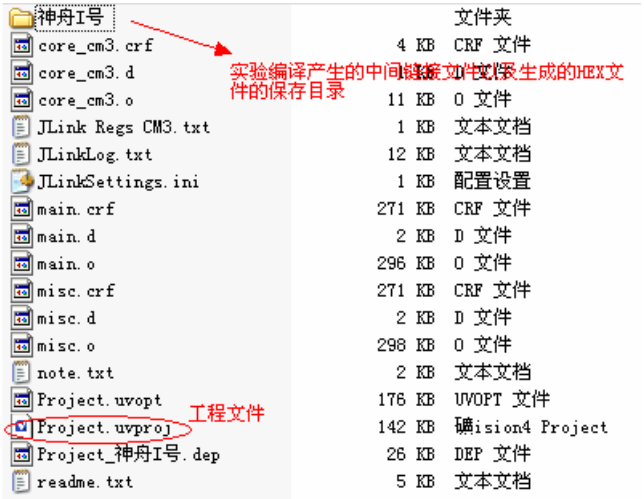
_htmresc 文件夹	Html 网页里使用的图片文件
Libreries 文件夹	ST 官方理工的库文件，包括 Cortex-M3 内核相关文件，以及 stm32f10x 启动文件等。具体结构如下：
Project 文件夹	包含 ST 官方提供的示例程序库，工程模板以及神舟 I 号实验例程。
Utilities 文件夹	公共文件夹，包含 ST 官方评估板相关头文件。
Releas_Notes.html	STM32F10x_StdPeriph_Lib_V3.3.0 说明文件
Stm32f10x_stdperiph_lib_um.chm	STM32F10x_StdPeriph_Lib_V3.3.0 帮助文件

点击 *Project/01.LED 流水灯（神舟 I 号）*目录，内部文件夹如下：

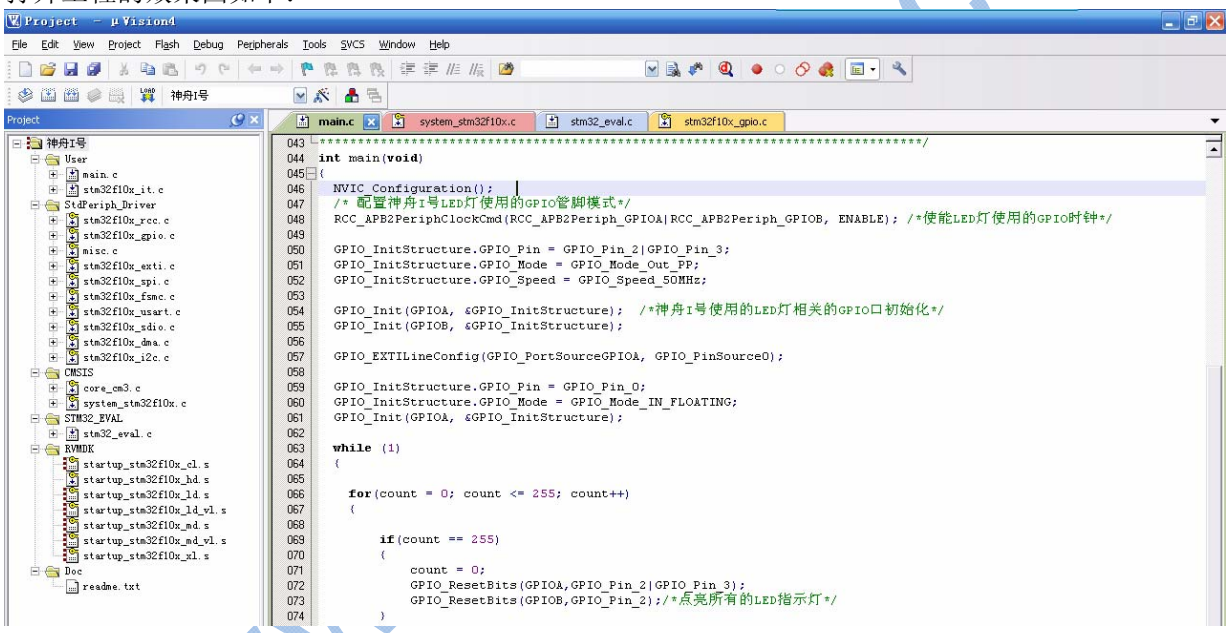
注意，每一个实验例程的结构都与上图类似，和 ST 官方提供的工程模板保持一致。上图中的各个文件夹为各个集中开发环境的工程文件，由于神舟 I 号是基于 MDK 集成开发环境的，因此，我们打开

MDK-ARM 文件夹，点击

即可打开工程文件。



打开工程的效果图如下：



工程打开后，即可按照前面几节描述的，对工程进行编译，仿真，下载等操作。

6.2 通用输入/输出（GPIO）

6.2.1 特性

- STM32F103RBT6 总共有 51 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）

6.2.2 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

6.2.3 管脚描述

表 1：端口 A GPIO 管脚描述

管脚名称	类型	描 述
PA[15:1]	I/O	通用输入/输出 PA1 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 PB1 到 PB15

表 3：端口 C GPIO 管脚描述

管脚名称	类型	描 述
PC[12:0]	I/O	通用输入/输出 PC1 到 PC12
PC[15:13]	I/O	通用输入/输出 PC13 到 PC15 的 I/O 口功能有限制（同一时间内只有一个 I/O 口可以作为输出，速度必须限制在 2MHZ 内，而且这些 I/O 口不能当作电流源（如驱动 LED））

表 4：端口 D GPIO 管脚描述

管脚名称	类型	描 述
PD[1:0]	I/O	通用输入/输出 PD0 到 PD1（该两个管脚被映射 OSC_IN 和 OSC_OUT 时钟）
PD[2]	I/O	通用输入/输出 PD2

6.2.4 功能描述

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。除了具有模拟输入功能的端口，所有的 GPIO 引脚都有大电流通过能力，具体如下 8 种模式：

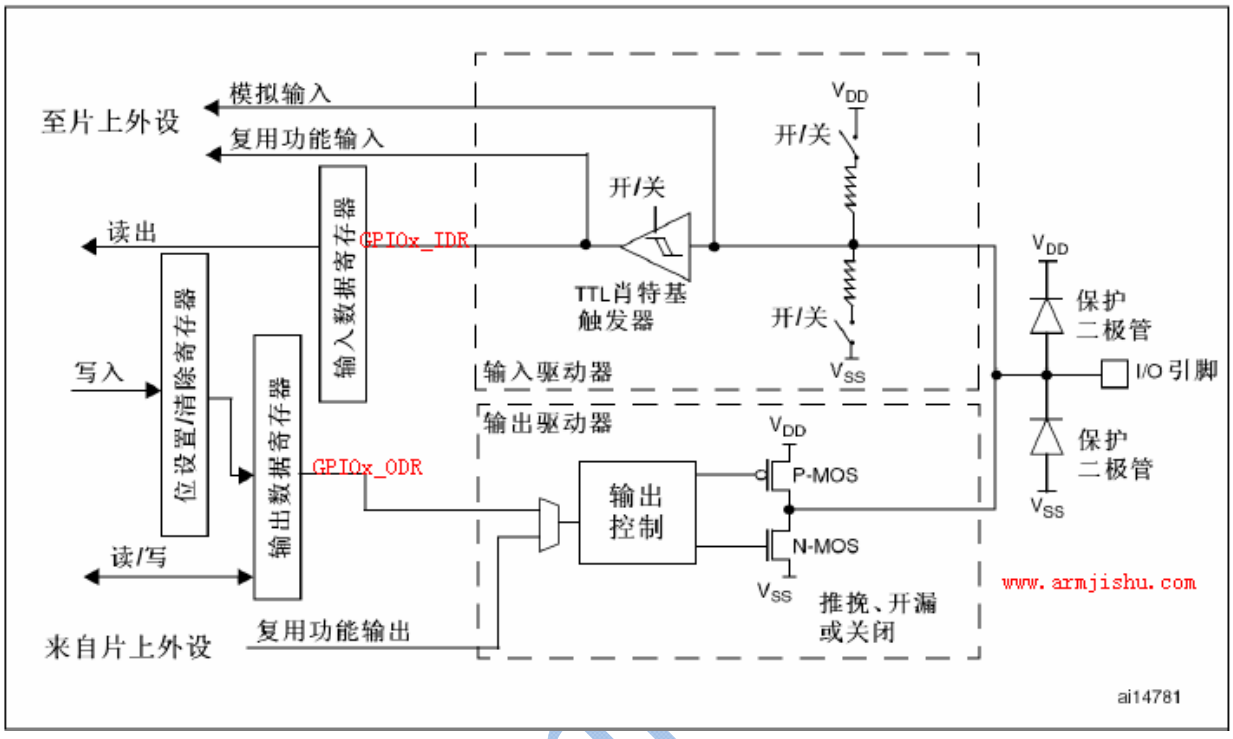
- 1）输入浮空
- 2）输入上拉
- 3）输入下拉
- 4）模拟输入
- 5）开漏输出
- 6）推挽式输出
- 7）推挽式复用功能
- 8）开漏复用功能

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，以上这些功能，在我们的 STM32 库文件中所对应的代码如下：

```
typedef enum    www.armjishu.com
{ GPIO_Mode_AIN = 0x0,            //模拟输入
  GPIO_Mode_IN_FLOATING = 0x04,   //浮空输入
  GPIO_Mode_IPD = 0x28,           //下拉输入
  GPIO_Mode_IPU = 0x48,           //上拉输入
  GPIO_Mode_Out_OD = 0x14,        //开漏输出
  GPIO_Mode_Out_PP = 0x10,        //推挽输出
  GPIO_Mode_AF_OD = 0x1C,         //开漏复用功能
  GPIO_Mode_AF_PP = 0x18          //推挽复用功能
}GPIOMode_TypeDef;
```

每个 GPIO 端口都需要与 7 个寄存器打交道：

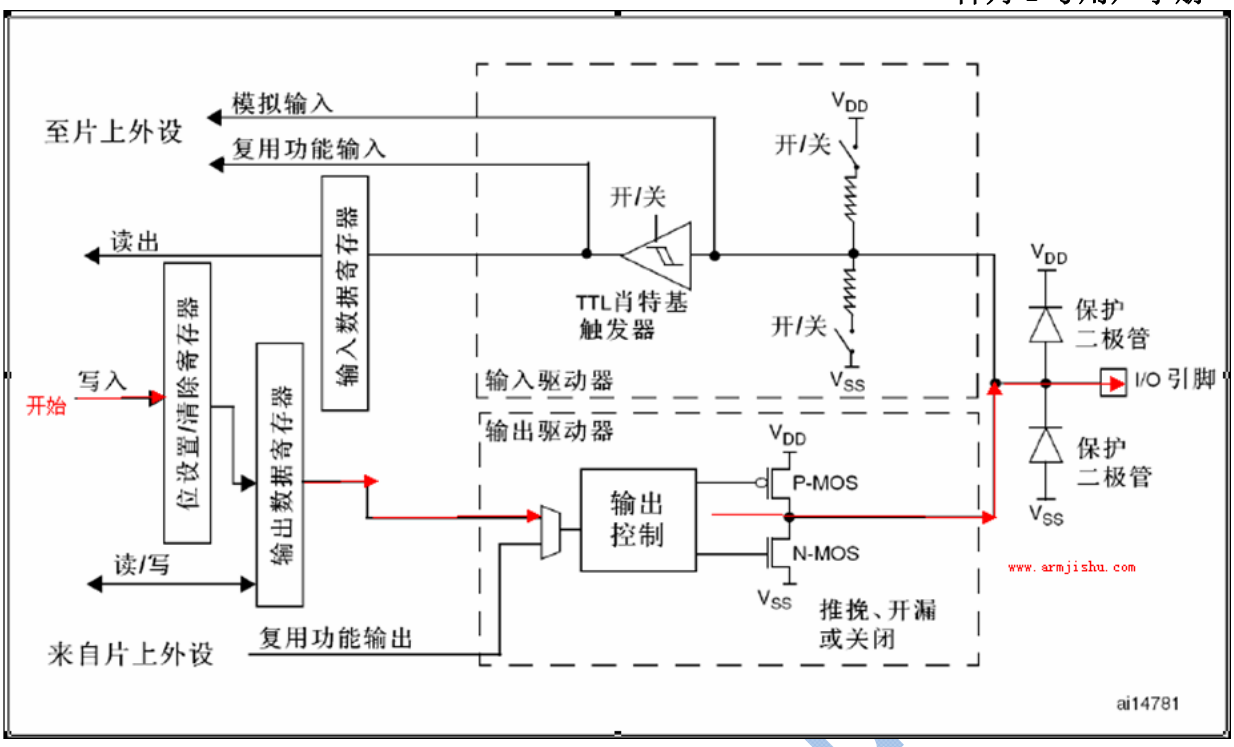
- 2 个 32 位配置寄存器 GPIOx_CRL 和 GPIOx_CRH
- 2 个 32 位数据寄存器 GPIOx_IDR 和 GPIOx_ODR
- 1 个 32 位置位/复位寄存器 GPIOx_BSRR
- 1 个 16 位复位寄存器 GPIOx_BRR
- 1 个 32 位锁定寄存器 GPIOx_LCKR



(iv) 5.1.4.1. 通用IO（GPIO）

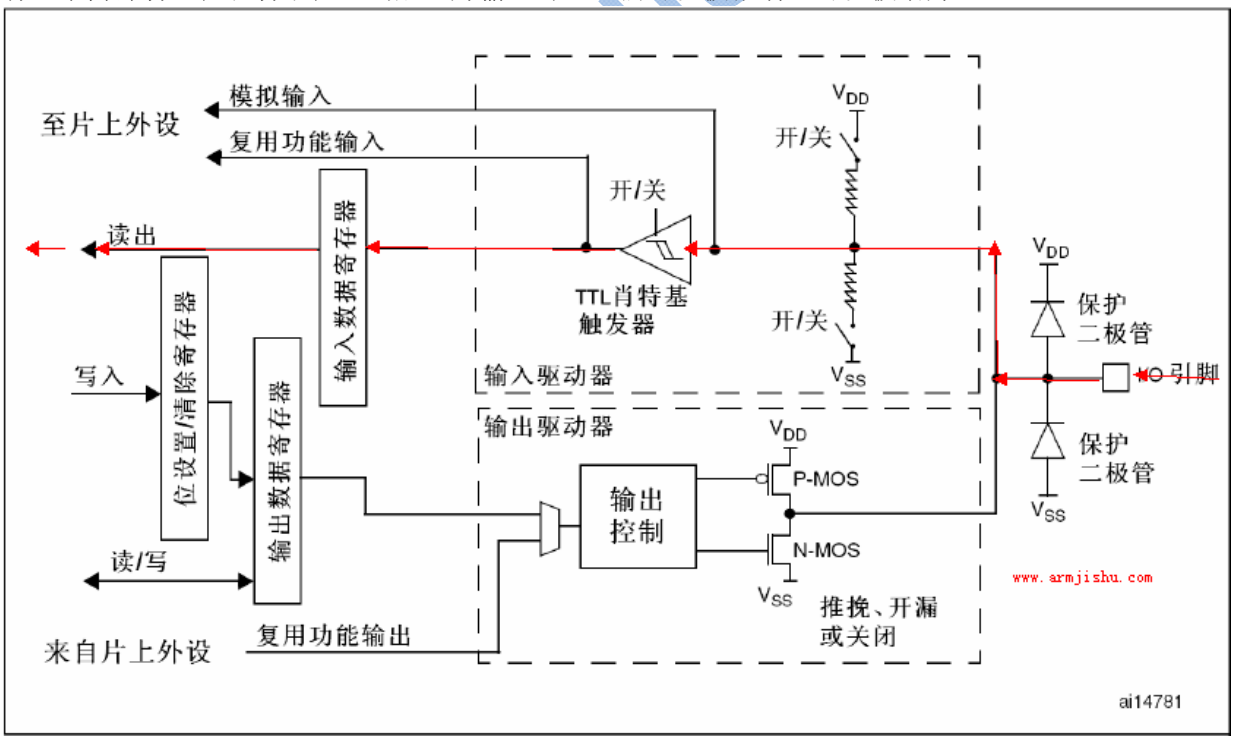
1）当作为输出配置时

写到输出数据寄存器（GPIOx_ODR）上的值输出到相应的 I/O 引脚；可以以【推挽模式】或【开漏模式】（当输出 0 时，只有 N-MOS 被打开）使用输出驱动器



2）当作为输入配置时：

输入数据寄存器（GPIOx_IDR）在每个 APB2 时钟周期捕捉 I/O 引脚上的数据；所有 GPIO 引脚有一个内部弱上拉和弱下拉，当配置为输入时，它们可以被激活也可以被断开。



(v) 5.1.4.2 复位后GPIO初始状态

复位期间和刚复位后，如果复用功能未开启，I/O 端口默认被配置成【浮空输入模】式（CNFx[1:0]=01b,MODEx[1:0]=00b）

(vi) 5.1.4.3. 单独的位设置或位清除

该功能方便软件改变单独的 GPIO 管脚高低电平设置，例如，端口配置好以后只需 GPIO_SetBits（GPIOx，GPIO_Pin_x）就可以实现对 GPIOx 的 pinx 位为高电平。

这是在单次 APB2 时钟写操作里，通过对“置位/复位寄存器”（GPIOx_BSRR）中想要更改的位写“1”来实现的，没被选中的位将不被更改。

(vii) 5.1.4.4. 外部中断/唤醒线

所有端口都有外部中断能力，但前提是端口必须配置成输入模式，这样就可以监控到外部输入进来的中断了。

(viii) 5.1.4.5 复用功能

该 GPIO 口原本是【A 功能】，如果使用默认复用【功能 B】时，可以通过对端口位的寄存器进行设置，配置成复用输出功能，则引脚和原本的【输出寄存器】（A 功能）断开，并和【片上外设的输出信号】（B 功能）连接。

另：如果把一个 GPIO 管脚配置成复用输出功能，但外设没有被激活，它的输出将不确定，以下是三种复用功能情况：

- 复用输入功能，端口必须配置成输入模式（浮空，上拉或下拉）且输入引脚必须由外部驱动
- 复用输出功能，端口必须配置成复用功能输出模式（推挽或开漏）
- 双向复用功能，端口位必须配置复用功能输出模式（推挽或开漏），这时，输入驱动器被配置成浮空输入模式。

(ix) 5.1.4.6 软件重新映射I/O复用功能

为了使不同器件封装的外设 I/O 功能的数量达到最优，可以把一些复用功能重新映射到其他一些脚上。这可以通过软件配置相应的寄存器来完成(参考 AFIO 寄存器描述)。这时，复用功能就不再映射到它们的原始引脚上了。

(x) 5.1.4.7 GPIO锁定机制

锁定机制允许冻结 IO 配置。当在一个端口位上执行了锁定(LOCK)程序，在下一次复位之前，将不能再更改端口位的配置。例如可以对复位端口进行锁定。

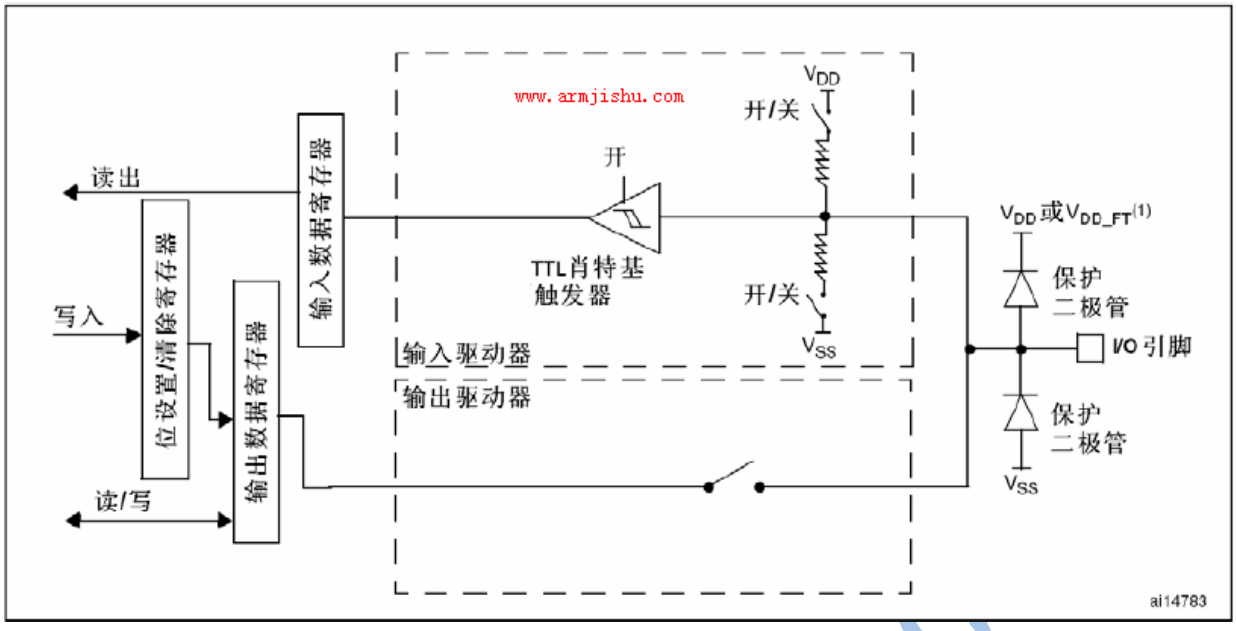
(xi) 5.1.4.8 输入配置

当 I/O 端口被配置成输入时，将出现以下的情况：

- 输出缓冲器被禁止
- 施密特触发输入被激活
- 根据输入配置（上拉，下拉或浮动）的不同，弱上拉和下拉电阻被连接
- 出现在 I/O 脚上的数据在每个 APB2 时钟被采样到输入数据寄存器
- 对输入数据寄存器的读访问可得到 I/O 状态

下图是输入浮空/上拉/下拉配置：

www.armjishu.com



(xii) 5.1.4.9 输出配置

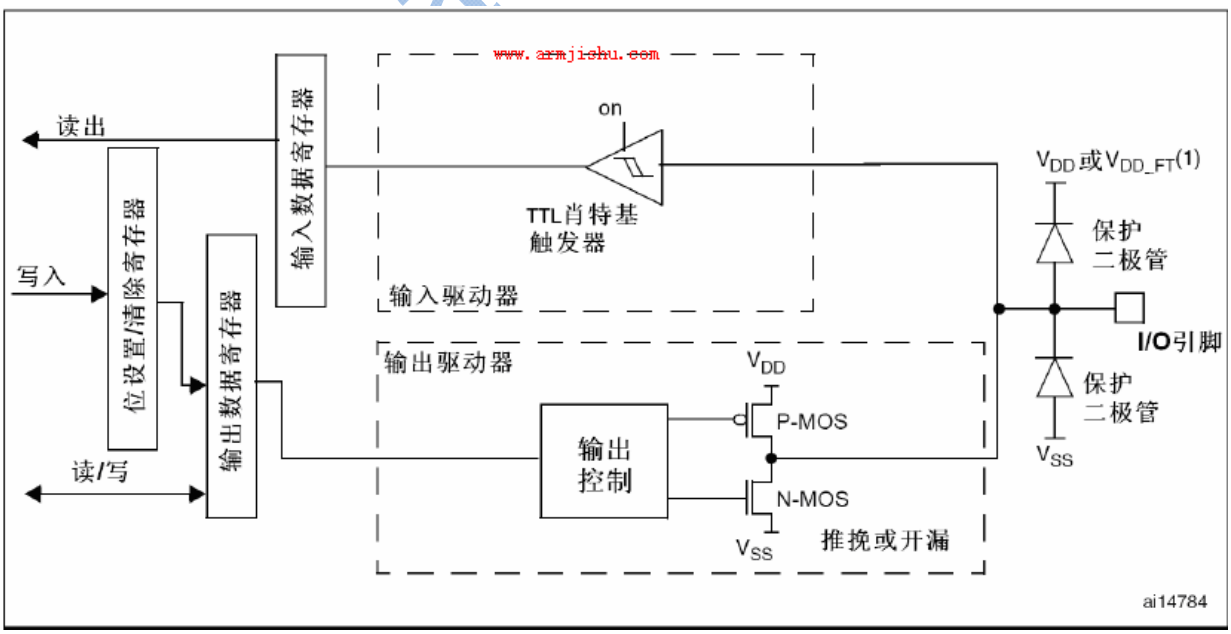
当 I/O 端口被配置为输出时：

- 输出缓冲器被激活

------开漏模式：输出寄存器上的“0”激活 N-MOS，而输出寄存器上的“1”将端口置于高阻状态（P-MOS 从不被激活）。

------推挽模式：输出寄存器上的“0”激活 N-MOS，而输出寄存器上的“1”将激活 P-MOS

- 施密特触发输入被激活
- 弱上拉和下拉电阻被禁止
- 出现在 I/O 脚上的数据在每个 APB2 时钟被采样到输入数据寄存器
- 在开漏模式时，对输入数据寄存器的读访问可得到 I/O 状态
- 在推挽模式时，对输出数据寄存器的读访问得到最后一次写的值

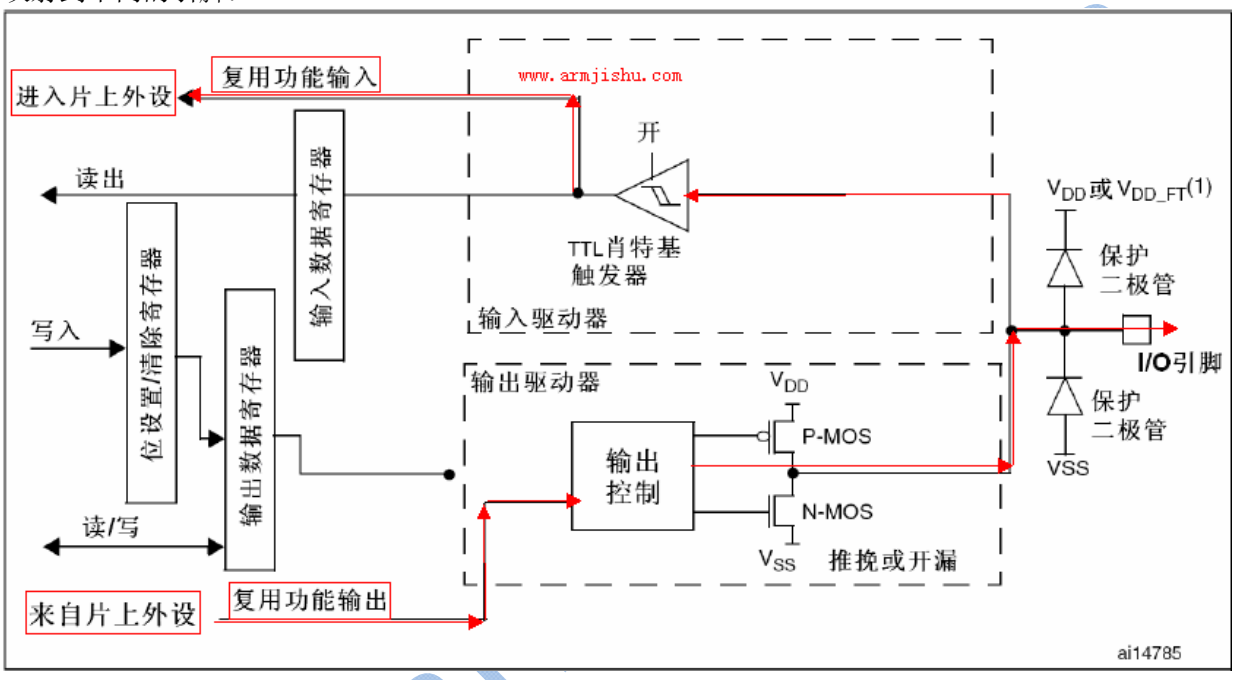


(xiii) 5.1.4.10 复用功能配置

当 I/O 端口被配置为复用功能时：

- 在开漏或推挽式配置中，输出缓冲器被打开
- 内置外设的信号驱动输出缓冲器（复用功能输出）
- 施密特触发输入被激活
- 弱上拉和下拉电阻被禁止
- 在每个 APB2 时钟周期，出现在 I/O 脚上的数据被采样到输入数据寄存器
- 开漏模式时，读输入数据寄存器时可得到最后一次写的值

下图示出了 I/O 端口位的复用功能配置，一组复用功能 I/O 寄存器允许用户把一些复用功能重新映射到不同的引脚：

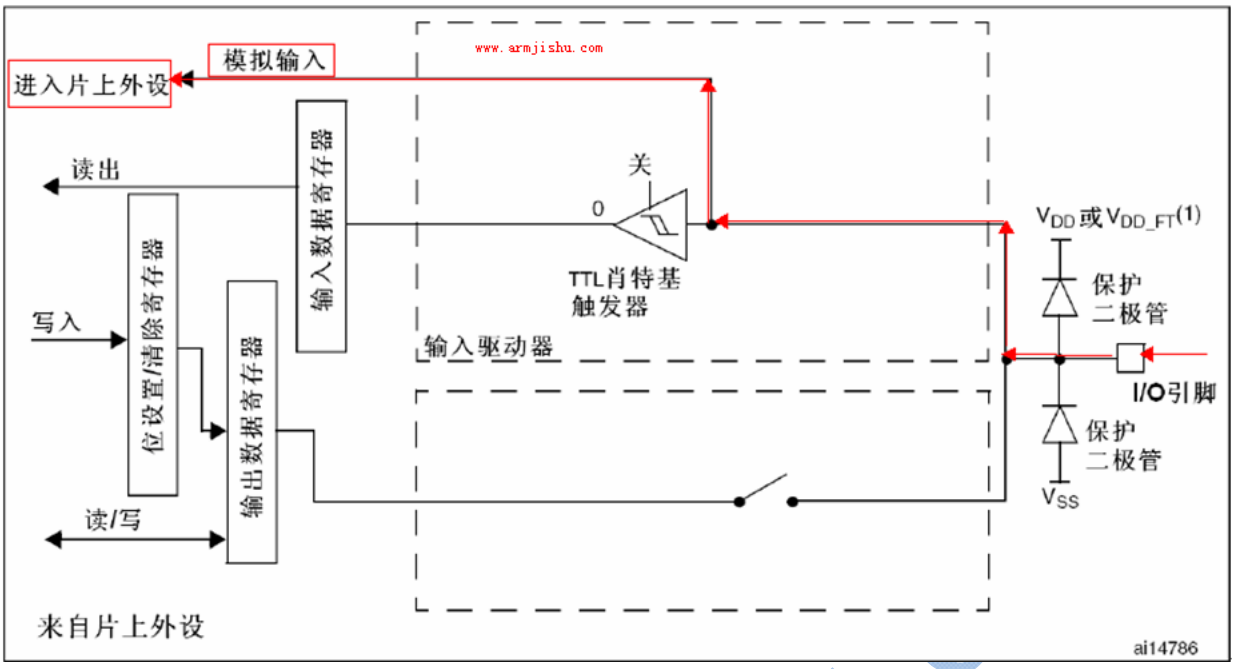


(xiv) 5.1.4.11 模拟输入配置

当 I/O 端口被配置为模拟输入配置时：

- 输出缓冲器被禁止
- 禁止施密特触发输入，实现了每个模拟 I/O 引脚上的零消耗。施密特触发输出值被强置为‘0’
- 弱上拉和下拉电阻被禁止
- 读取输入数据寄存器时数值为‘0’

下图示出了



6.2.5 寄存器描述

1. 端口配置低寄存器（GPIOx_CRL）（x=A…E）

作用：主要配置芯片的低 8 位引脚

偏移地址和：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 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) (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

可以看到，该寄存器 32 位中，每 4 位描述一个引脚，该寄存器总共可以描述 8 个引脚，主要用来配置芯片的低 8 位引脚。

2. 端口配置高寄存器（GPIOx_CRH）（x=A…E）

作用：主要配置芯片的高 8 位引脚

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 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**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 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 = 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

可以看到，该寄存器 32 位中，每 4 位描述一个引脚，该寄存器总共可以描述 8 个引脚，主要用来配置芯片的高 8 位引脚。

3. 端口输入数据寄存器（GPIOx_IDR）（x=A…E）

地址偏移：0x08

复位置：0x0000 XXXX

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	保留，始终读为0。
位15:0	**IDRy[15:0]**：端口输入数据(y = 0…15) (Port input data) 这些位为只读并只能以字(16位)的形式读出。读出的值为对应I/O口的状态。

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。

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

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

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

7. 端口配置锁定寄存器（GPIOx_LCKR）（x=A…E）

当执行正确的写序列设置了下图的寄存器位 16（LCKK）时，该寄存器用来锁定端口位的配置。位[15:0]用于锁定 GPIO 端口的配置。在规定的写入操作期间，不能改变 LCKP[15:0]。当对相应的端口执行了 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。

注：每个锁定位锁定 CRL，CRH 寄存器中的相应的 4 个位，即一个管脚。

地址偏移：0x18

复位置：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															LCKK
															rw

WWW.ARMJISHU.COM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:17	保留。
位16	LCKK：锁键 (Lock key) 该位可随时读出，它只可通过锁键写入序列修改。 0：端口配置锁键位激活 1：端口配置锁键位被激活，下次系统复位前GPIOx_LCKR寄存器被锁住。 锁键的写入序列： 写1 -> 写0 -> 写1 -> 读0 -> 读1 最后一个读可省略，但可以用来确认锁键已被激活。 注：在操作锁键的写入序列时，不能改变LCK[15:0]的值。 操作锁键写入序列中的任何错误将不能激活锁键。
位15:0	LCKy: 端口x的锁位y (y = 0…15) (Port x Lock bit y) 这些位可读可写但只能在LCKK位为0时写入。 0：不锁定端口的配置 1：锁定端口的配置

6.2.6 寄存器小结

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 口的方向**

2） STM32 的 ODR 寄存器是用来控制 **IO 口的输出电平或者上下拉电阻的**

3） STM32 的 IDR 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 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

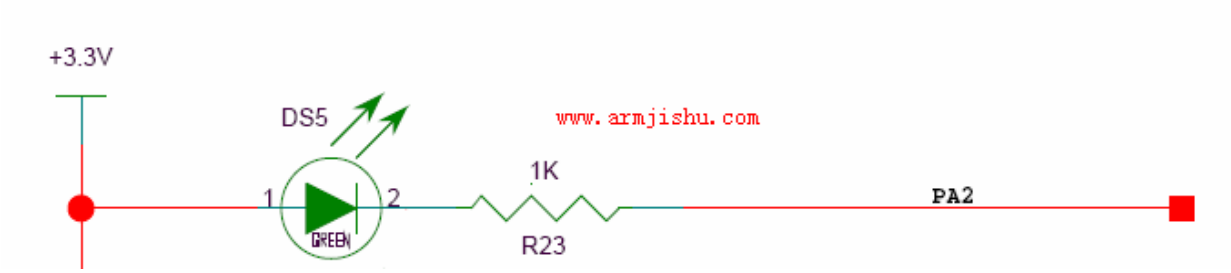
5） STM32 的 BRR 用来清除某端口的某一位位 0，如果该寄存器某位为 0，那么它所对应的那个引脚位不产生影响；如果该寄存器某位为 1，则清除对应的引脚位。

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。

6.2.7 例程01 单个LED点灯程序

4. 示例简介

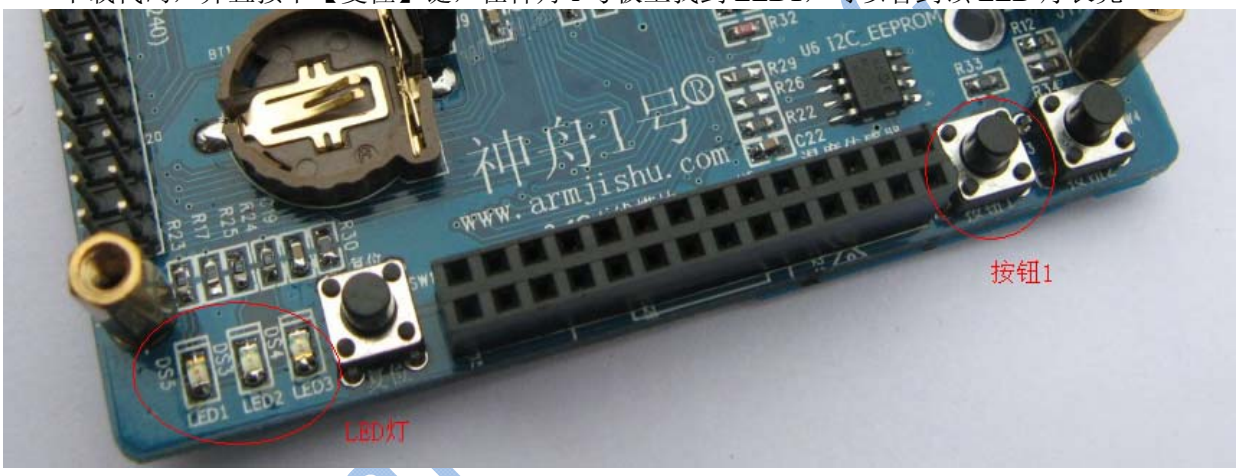
LED 灯的正极接的是 3.3V 电源，所以我们编程让 LED 负极拉低即 GPIO 引脚端口 A 的 Pin2 拉低，那么 LED 灯就会变亮，相关电路图如下图所示：



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。

5. 调试说明

下载代码，并且按下【复位】键，在神舟 I 号板上找到 LED1，可以看到该 LED 灯长亮



6. 关键代码：

相关代码如下图程序清单：

```
/******************** (C) COPYRIGHT 2011 www.armjishu.com  ********************
* Author             : jesse
* Organization      :www.armjishu.com
* Version            : V1.0
* Date               : 08/01/2011
* Description        : Main program body.
*******************************************************************************/
#include "stm32f10x.h"
GPIO_InitTypeDef GPIO_InitStructure;

int main(void)
{

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);  /**** 初始化GPIO的A端口的RCC时钟 ****/
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;             /**** 设置需要初始化的管脚2 ****/
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  /**** 配置管脚为推免输出模式  ****/
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; /**** 配置管脚输出模式最大速度为50MHZ ****/

  GPIO_Init(GPIOA, &GPIO_InitStructure);                /**** 初始化GPIO的A端口 ****/

 /**** 设置GPIO的A端口的Pin2为低电平，从而点亮LED指示灯(请参看电路原理图) ****/
  GPIO_ResetBits(GPIOA,GPIO_Pin_2);

}
```

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 PA2 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 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															

www.armjishu.com

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

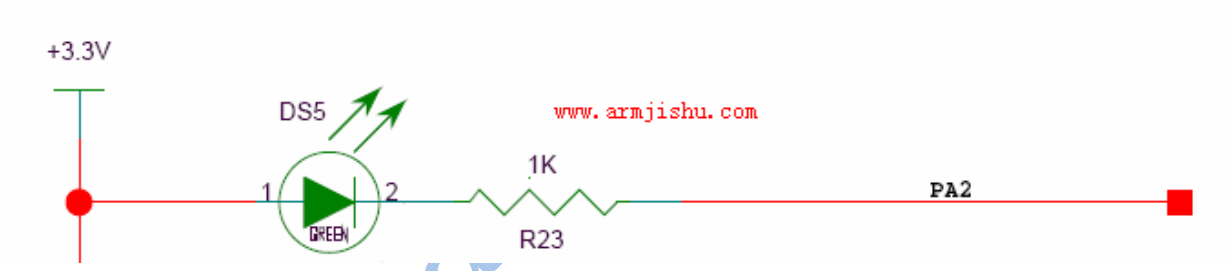
图：寄存器 APB2ENR 各位描述

我们要使能的是 PORTA 的时钟使能位，大家可以从上表看得到在 bit2 这个位，只需要将这个位置 1 就可以使能 PORTA 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 `RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);`；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。

6.2.8 例程02 单个LED灯闪烁

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 A 的 Pin2 拉低，那么 LED 灯就会变亮；同样将 Pin2 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 I 号板上找到 LED1，可以看到该 LED 灯不停的闪烁。

3. 关键代码

```
#include "stm32f10x.h"
GPIO_InitTypeDef GPIO_InitStructure;
u8 count=0;

void Delay(vu32 nCount)                                  /**** 延时程序 ****/
{
  for(; nCount != 0; nCount--);
}

int main(void)
{
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);  /**** 初始化GPIO的A端口的RCC时钟 ****/
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;               /**** 设置需要初始化的管脚2 ****/
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;        /**** 配置管脚为推免输出模式  ****/
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;      /**** 配置管脚输出模式最大速度为50MHZ ****/

  GPIO_Init(GPIOA, &GPIO_InitStructure);                  /**** 初始化GPIO的A端口 ****/

  while(1)
  {
        GPIO_SetBits(GPIOA,GPIO_Pin_2);          /* 设置Pin2管脚为1，输出高电平，熄灭LED指示灯 */
        Delay(0x2FFFFF);                         /**** 延时程序 ****/
        GPIO_ResetBits(GPIOA,GPIO_Pin_2);        /* 设置Pin2管脚为0，输出低电平，点亮LED指示灯 */
        Delay(0x2FFFFF);                         /**** 延时程序 ****/
  }
}
```

www.armjishu.com

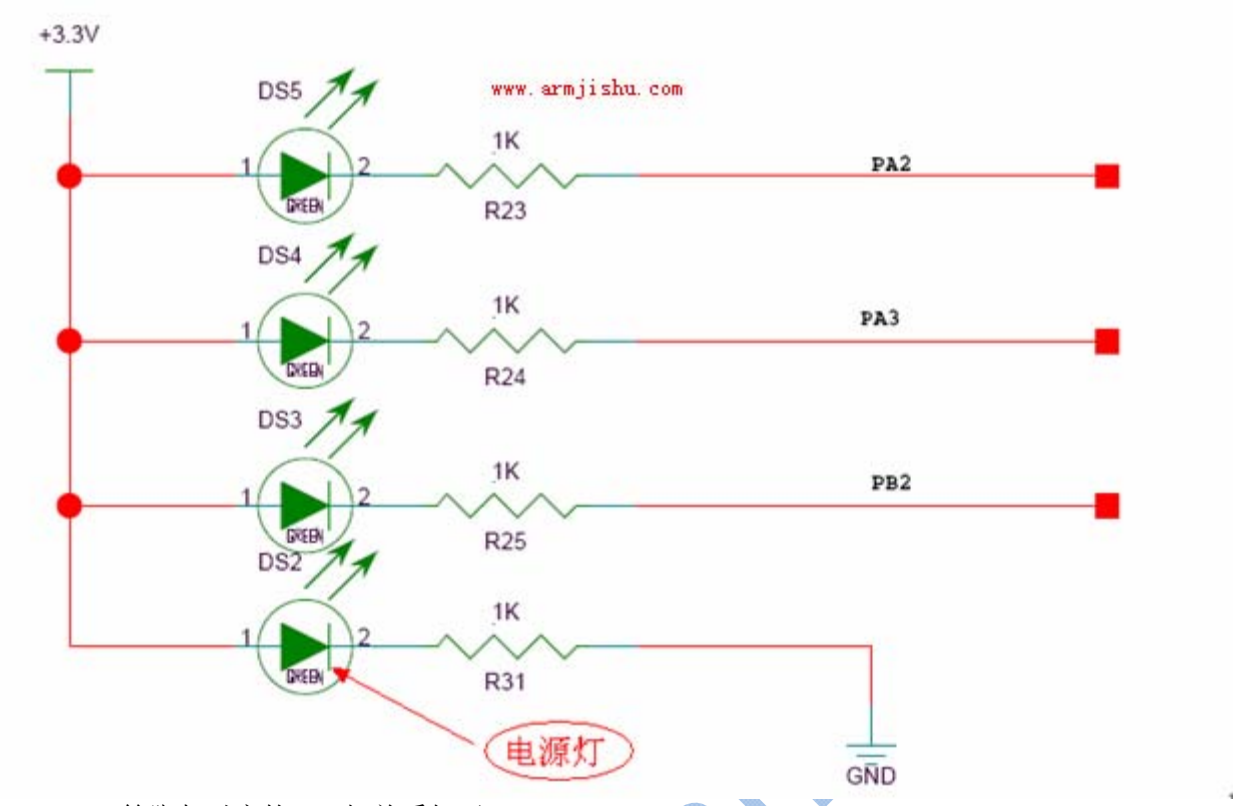
这里和上面程序不同之处是增加了延时等待代码，就是上面代码的“Delay（）”函数，然后我们将管脚拉高和拉低，在一个“whlie（1）”这个死循环里最终实现了 LED 灯不停的闪烁。

6.2.9 例程03 LED流水灯程序

1. 示例简介

在神舟I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：



GPIO管脚与对应的LED灯关系如下：

LED灯	LED灯对应的GPIO
LED1	PA2
LED2	PB2
LED3	PA3

2. 调试说明

下载代码，并且按下【复位】键，在神舟I号板上找到LED1，LED2，LED3三个灯，可以看到这三个灯轮流闪烁，流水灯。

3. 关键代码

```
int main(void)
{
  /**** 初始化GPIOA和GPIOB两个端口的RCC时钟 ****/
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB, ENABLE);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_3;      /**** 设置需要初始化的管脚2和管脚3 ****/
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;          /**** 配置管脚为推免输出模式  ****/
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;         /**** 配置管脚输出模式最大速度为50MHZ ****/

  GPIO_Init(GPIOA, &GPIO_InitStructure);                            /**** 初始化GPIO的A端口 ****/
  GPIO_Init(GPIOB, &GPIO_InitStructure);                            /**** 初始化GPIO的B端口 ****/

  while(1)
  {
    for(nCount = 0; nCount < 3;nCount++)          /**** LED流水灯一共有3个，逐个轮流来点亮 ****/
    {
     /**** 将GPIOA的Pin2和Pin3置1，拉成高电平，熄灭对应的LED灯 ****/
        GPIO_SetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);

     /**** 将GPIOB的Pin2置1，拉成高电平，熄灭对应的LED灯 ****/
        GPIO_SetBits(GPIOB,GPIO_Pin_2);

        Turn_On_LED(nCount);                  /**** 按nCount的值点亮相对应的灯 ****/
        Delay(0x8FFFF);                       /**** 延时程序，让灯点亮的状态持续一会 ****/
    }
  }
}
```

www.armjishu.com

程序主要设计思路就是先将所有 LED 灯都熄灭，然后用函数“`Turn_On_LED(nCount);`”每次只点亮一个指定的 LED 灯，并且延时持续点亮一会，然后再全部熄灭，再点亮下一个 LED 灯，如此循环，形成 LED 流水灯。

6.3 KEY_LED按键与315M无线模块实验

上一节简单的介绍了STM32的IO口作为输出功能使用，这一节，我们将向大家介绍如何使用STM32的IO口作为输入功能，其中包括按键的输入，也有315M无线模块通过解码后，由IO口输入。通过本节的学习，你将了解到STM32的IO口作为输入使用的方法，也对无线通信传输，解码有一些概念性的了解。本节分为如下几个小节：

6.2.1 实验的意义与作用

STM32的IO口在上一节的流水灯实验中已经有了详细的介绍，这里我们不再多说。STM32的IO口用作输出时，可以通过寄存器GPIO>ODR配置不同管脚的状态电平，而做输入使用的时候，则是通过读取GPIOx—>IDR寄存器的内容来识别读取IO口的状态的。了解了这一点，我们就可以开始的代码的编写。

这一节，我们将通过神舟I号板载有的2个按键：按键1和按键2，来控制板上的3个LED（LED1，LED2，LED3），其中按键1控制LED1变亮，而按键2控制LED2灯灭。

同时，借用315M无线模块进行实验，无线模块的遥控按钮，按下任一个按钮，LED1~LED3便出现一种组合亮灯。通过此实验，可以让爱好者享受无线控制的乐趣。

6.2.2 实验原理

这个例程的实验原理主要包括两个：一、通过神舟I号开发板上的两个按钮，按钮1和按钮2，控制三个流水灯的点亮和关闭的状态。其中的原理是通过GPIOX->IDR寄存器读取按钮连接的GPIO管脚的电平变化，作为点灯或关灯的触发事件，然后再由通过GPIOX->ODR寄存器的配置，输出控制灯的点亮或是关闭。按钮按下为“0”，松开为“1”（详见原理图）。

二、关于315M无线模块的实验原理，315M配套的遥控设备，每按下一个按钮，以315MHz左右的频率通过无线传输，而315M模块由28cm长度的天线接收传输的信号，并由PT2272进行解码，同时置VT端有效高电平，然后将解码后的数据由D0~D3连接的GPIO管脚输入到STM32处理器中，处理器根据接收到的信号，对应输出控制LED的组合点灯亮状态。

www.armjishu.com

6.2.3 硬件设计

该实验所需要的硬件电路在神舟I号开发板上都已经连接好了，不需要经过任何设置，直接编写代码就可。LED的连接在上一节已经介绍过了。如下图所示：

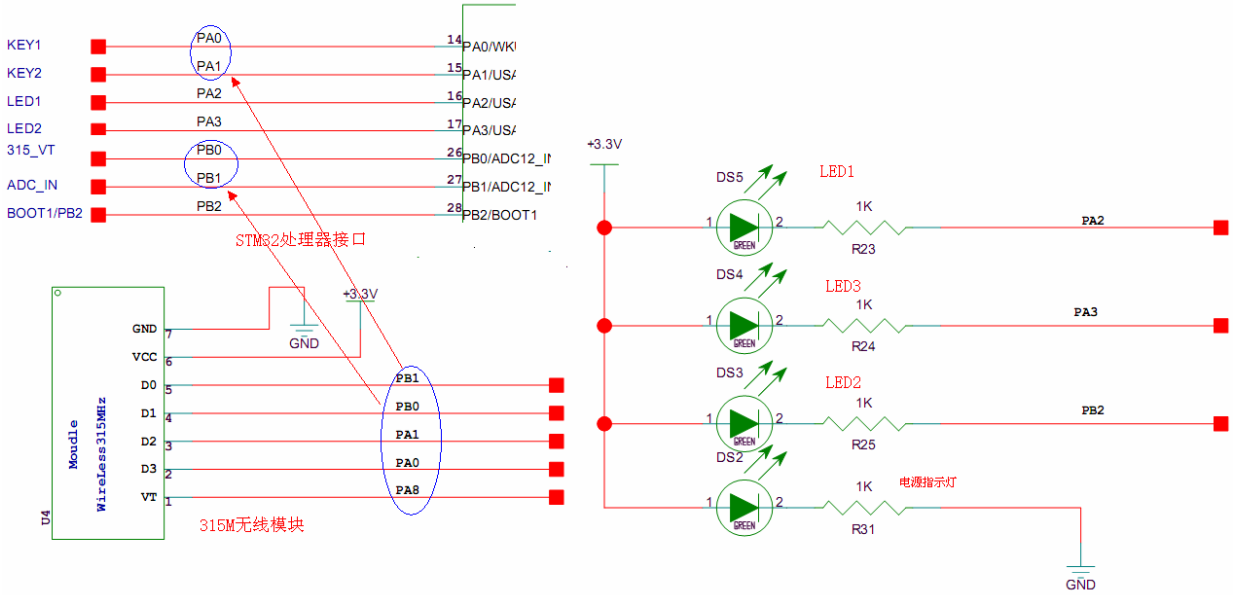


图3.2.2.1 315M无线模块与STM32连接原理图

这里再次强调的是315M的VT端是高电平有效的。而且在315M无线模块的供电电压为3.3V，目前是为了直接连接STM32处理的IO管脚（虽然有的IO管脚能够支持5V）

6.2.4 软件设计

这里的代码有些是与上一次有点相同的，特别是对于使用的GPIO的初始化等，不同的是上一节点灯时，GPIO管脚是作为输出，而在这一节中，GPIO却是作为输入使用的。如按键的输入，或是315M无线模块的数据的输入。

神舟I号的“KEY_LED按键与315M无线模块实验”位于*神舟I号光盘\神舟I号源码\STM32F10x_StdPeriph_Lib_V3.3.0.rar\\Project\02. KEY_LED按键与315M无线模块实验（神舟I号）*目录中。进入*02. KEY_LED按键与315M无线模块实验（神舟I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。

神舟I号按键的使用的GPIO的接口定义

```
/*神舟I号按键相关定义*/
#define RCC_KEY1                    RCC_APB2Periph_GPIOA      /*按键1（315MD3）使用的GPIO时钟*/
#define GPIO_KEY1_PORT              GPIOA                     /*按键1（315MD3）使用的GPIO组*/
#define GPIO_KEY1                   GPIO_Pin_0                /*按键1（315MD3）连接的GPIO管脚号*/

#define RCC_KEY2                    RCC_APB2Periph_GPIOA      /*按键2（315MD2）使用的GPIO时钟*/
#define GPIO_KEY2_PORT              GPIOA                     /*按键2（315MD2）使用的GPIO组*/
#define GPIO_KEY2                   GPIO_Pin_1                /*按键2（315MD2）连接的GPIO管脚号*/
```

神舟I号315M无线模块除了与上面按键复用外，剩下的D0和D1以及VT端的接口定义

```
/*315M无线模块与处理器连接管脚的定义，其中有两个数据管脚与上述的按键进行复用*/

#define RCC_315MD0                  RCC_APB2Periph_GPIOB      /*315MD0使用的GPIO时钟*/
#define GPIO_315MD0_PORT            GPIOB                     /*315MD0使用的GPIO组*/
#define GPIO_315MD0                 GPIO_Pin_1                /*315MD0连接的GPIO管脚号*/

#define RCC_315MD1                  RCC_APB2Periph_GPIOB      /*315MD1使用的GPIO时钟*/
#define GPIO_315MD1_PORT            GPIOB                     /*315MD1使用的GPIO组*/
#define GPIO_315MD1                 GPIO_Pin_0                /*315MD1连接的GPIO管脚号*/

#define RCC_315MVT                  RCC_APB2Periph_GPIOB      /*315MVT 使用的GPIO时钟*/
#define GPIO_315MVT _PORT           GPIOB                     /*315MVT 使用的GPIO组*/
#define GPIO_315MVT                 GPIO_Pin_0                /*315MVT 连接的GPIO管脚号*/
```

对于一些使用的值进行宏定义

```
/* Values magic to the Board keys */
#define  NOKEY  0              /*一些值的宏定义*/
#define  KEY1   1
#define  KEY2   2
#define  D0     3
#define  D1     4
#define  VT     5
```

按键以及315M模块使用的GPIO接口的初始化

```
void GPIO_KEY_Config(void)
{                                   /*初始化按键1/2，以及315M模块的D0~D3和VT管脚的使用的管脚模式，管脚号以及使能*/
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Configure KEY1(D3) Button */
  RCC_APB2PeriphClockCmd(RCC_KEY1, ENABLE);                /*KEY1（D3）*/

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Pin = GPIO_KEY1;
  GPIO_Init(GPIO_KEY1_PORT, &GPIO_InitStructure);

  /* Configure KEY2(D2) Button */
  RCC_APB2PeriphClockCmd(RCC_KEY2, ENABLE);                /*KEY2（D2）*/

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Pin = GPIO_KEY2;
  GPIO_Init(GPIO_KEY2_PORT, &GPIO_InitStructure);

   /* Configure D0 data */
  RCC_APB2PeriphClockCmd(RCC_315MD0, ENABLE);                  /*D0 data*/

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Pin = GPIO_315MD0;
  GPIO_Init(GPIO_315MD0_PORT, &GPIO_InitStructure);

  /* Configure D1 data*/
  RCC_APB2PeriphClockCmd(RCC_315MD1, ENABLE);                  /*D1 data*/

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Pin = GPIO_315MD1;
  GPIO_Init(GPIO_315MD1_PORT, &GPIO_InitStructure);

    /* Configure VT data */
  RCC_APB2PeriphClockCmd(RCC_315MVT, ENABLE);                  /*VT data*/

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Pin = GPIO_315MVT;
  GPIO_Init(GPIO_315MVT_PORT, &GPIO_InitStructure);
```

GPIO管脚扫描函数，主要是针对两个按键，以及315M模块的数据和VT控制端的检测

```
u8 ReadKeyDown(void)
{
  /* 1 key is pressed */
  if(!GPIO_ReadInputDataBit(GPIO_KEY1_PORT, GPIO_KEY1))      /*检测按键1是否按下,为0*/
  {
    return KEY1;                                             /*返回KEY1*/
  }
  /* 2 key is pressed */
  if(!GPIO_ReadInputDataBit(GPIO_KEY2_PORT, GPIO_KEY2))      /*检测按键2是否按下, 为0*/
  {
    return KEY2;                                             /*返回KEY2*/
  }
                                                             /*注意以下是高电平时有效*/
  /*D0数据有效*/
    if(GPIO_ReadInputDataBit(GPIO_315MD0_PORT, GPIO_315MD0))
  {
    return D0;                                               /*返回D0*/
  }

   /*D1数据有效*/
  if(GPIO_ReadInputDataBit(GPIO_315MD1_PORT, GPIO_315MD1))
  {
    return D1;                                               /*返回D1*/
  }

  /*VT信号有效*/
  if(GPIO_ReadInputDataBit(GPIO_315MVT_PORT, GPIO_315MVT))
  {
    return VT;                                               /*返回VT*/
  }

  /* No key is pressed */
  else                                                        /*没有按键按下*/
  {
    return NOKEY;
  }
}
```

三个LED灯LED1~LED3的初始化，在上节已经讲过，在此简单提及

```
void LED_config(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  /*使能LED灯使用的GPIO时钟*/
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB|RCC_APB2Periph_AFIO, ENABLE);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_3;            /*使能GPIO管脚2和3*/
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;              /*设置为推挽输出*/
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;              /*端口速率*/

  GPIO_Init(GPIOA, &GPIO_InitStructure);  /*神舟I号使用的LED灯相关的GPIO口初始化*/
  GPIO_Init(GPIOB, &GPIO_InitStructure);

}
```

以下为几个点灯关灯的函数，虽然可以直接使用命令控制点灯，但在这里使用函数，直观，调用简单。

```
void Led_Turn_on_all(void)

void Led_Turn_off_all(void)
...
void Led_Turn_on_1(void)/*定义LED1点亮的函数,
...
void Led_Turn_on_2(void)
...
void Led_Turn_on_3(void)
```

扫描按键以及315M模块相连接的GPIO管脚电平的变化函数

```
void Led_Turn_on(u8 led)      /*判断点灯函数，根据传递的数据进行点灯。*/
{                             /*传递的数据可以来自按键电平，也可以来自315M模块的解码数据*/
    Led_Turn_off_all();

    /* Turn Off Select LED */
    switch(led)
    {
        case 0:                          /*来自按键1或是来自315M模块解码数据*/
          Led_Turn_on_1();
          break;

        case 1:                          /*来自按键1或是来自315M模块解码数据*/
          Led_Turn_on_2();
          break;

        case 2:                         /*来自315M模块解码数据*/
          Led_Turn_on_3();
          break;

        case 3:                         /*来自315M模块解码数据*/
          Led_Turn_on_all();
          break;

        default:
          Led_Turn_on_all();
          break;
    }
}
```

主函数的设计

```
int main(void)
{
  u8 KeyNum = 0;

  LED_config();
  Led_Turn_on_all();        /*打开所有灯*/
  Delay_ARMJISHU(6000000);
  Led_Turn_off_all();         /*关闭所有灯*/
  Delay_ARMJISHU(6000000);
  Led_Turn_on_all();           /*打开所有灯*/

  GPIO_KEY_Config();

  /* Main loop */
  while (1)
  {
      KeyNum = ReadKeyDown();                /*获取与按键以及315M模块相连接的GPIO管脚值*/
      if (KeyNum != VT)                      /*判断是否是VT管脚有效*/
         Led_Turn_on(KeyNum-1);              /*不是VT管脚有效，则是根据按键值，进行点灯操作*/
      else                                   /*是VT管脚有效，315M模块正在工作*/
         KeyNum = ReadKeyDown();             /*判断是哪个数据管脚有效，对应进行点灯*/
         Led_Turn_on(KeyNum-1);
  }
}
```

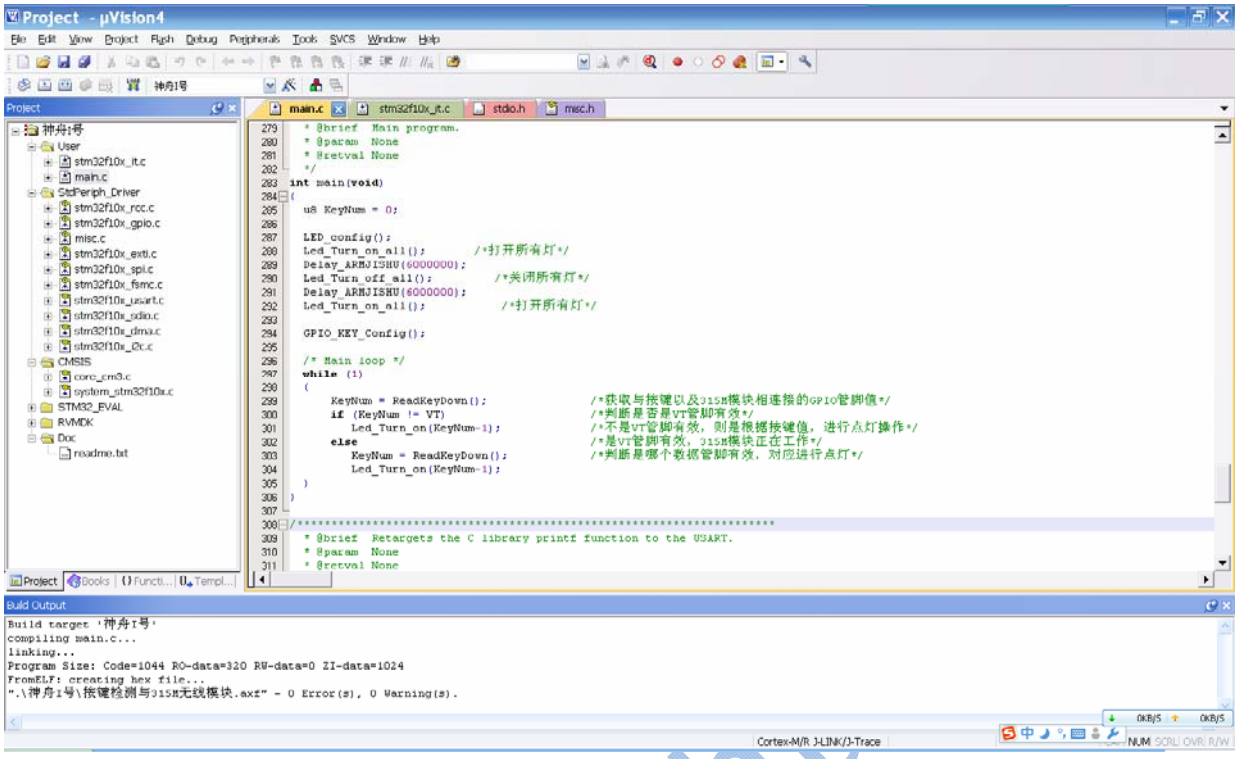


图3.2.3.3　按键检测与315M无线模块实验编译图

6.2.5　下载与测试

在 *神舟I号光盘\编译好的固件\02.KEY_LED按键与315M无线模块实验*目录下的按键检测与315M无线模块.hex文件即为前面我们分析的按键检测与315M无线模块实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I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I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I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。

www.armjishu

6.3 USART-COM串口发送实验

前面两节介绍了STM32的IO口基本操作。这一节我们将学习STM32的常用接口——串口。通过本节的学习，你将了解到STM32串口的原理、初始化以及发送功能。而对于串口的发送与接收实验，我们下节再继续学习。

6.3.1 实验的意义与作用

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，打印以及输入相关消息，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I号的各个例程的调试具有至关重要作用。在此例程中，我们先来学习串口的发送功能。

6.3.2 实验原理

学习STM32的串口，我们先了解两个相关的寄存器：

1，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，包含了发送和接收两部分。当向该寄存器写数据的时候，串口就会自动发送，当收到收据的时候，也是存在该寄存器内。该寄存器的各位描述如下：

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
保留							DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

寄存器USART_DR各位描述

位8:0	**DR[8:0]**：数据值 (Data value) 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248)。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)会被后来的校验位该取代。 当使能校验位进行接收时，读到的MSB位是接收到的校验位。

注意到，虽然是一个32位寄存器，但是[31:9]强制为“0”，只用了低9位（DR[8:0]）。

2，串口状态寄存器USART_SR。串口的状态可以通过状态寄存器USART_SR读取。USART_SR的各位描述如下：

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
保留						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc w0	rc w0	r	rc w0	rc w0	r	r	r	r	r

寄存器USART_SR各位描述

这里我们主要关注一下三位，第5、6、7位RXNE、TC和TXE。

位7	**TXE**:发送数据寄存器空 (Transmit data register empty) 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。
位6	**TC**：发送完成 (Transmission complete) 当包含有数据的一帧发送完成后，并且TXE=1时，由硬件将该位置'1'。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。
位5	**RXNE**：读数据寄存器非空 (Read data register not empty) 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。

串口最基本的设置，就是波特率的设置，然后配置数据位长度，奇偶校验位等信息。至于串口时钟、串口复位、串口的控制等等，详细请参考《【中文】STM32F系列ARM内核32位高性能微控制器参考手册V10_1.pdf》第516页开始阐述USART串行口以及540页开始关于寄存器的描述。

www.armjishu

6.3.3 硬件设计

串口接口的设计，我们常用的是采用DB9公头插针，然后通过MAX3232等电平转换芯片与处理器相连接，此设计对于神舟I号开发板来说也是可以的，但考虑到板件的小巧等原因，在神舟I号开发板上，串口是采用MINI-USB的接口方式，其中通过PL2303芯片后与STM32处理器相连接，外围电路也相对简单，如下图所示：PA9和PA10两个IO管脚

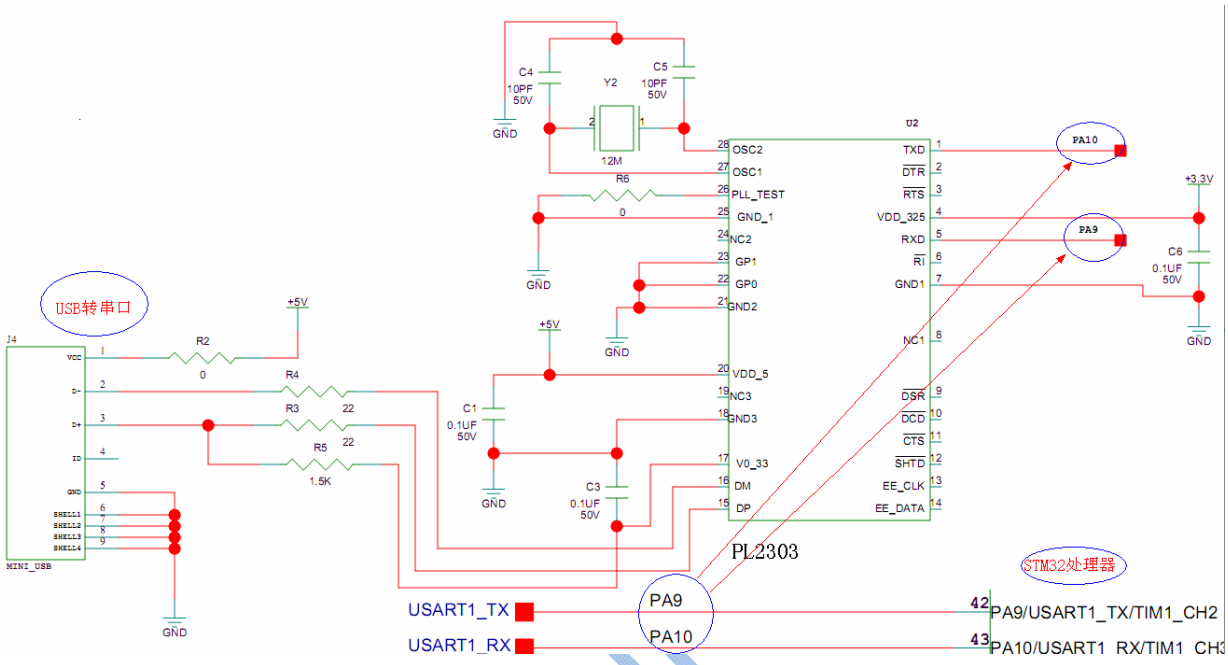


图3.3.2.1 硬件连接图

USART1_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。

USART1_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引脚处于高电平。

6.3.4 软件设计

在软件设计中，我们先来了解关于ST库函数中配置串口参数。关于STM32的USART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V3.3.0版本的库为例，这两个文件位于“STM32F10x_StdPeriph_Lib_V3.3.0\Libraries\STM32F10x_StdPeriph_Driver”目录的“src”和“inc”文件夹里：

首先是关于库函数模板中关于PUTCHAR_PROTOTYPE宏定义，这是为了兼容不同的编译平台的。

```
/* 以下的说明，主要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__ */
```

其次，我们关注串口的发送函数，通过此函数，我们发送一个字符出去。

```
/**
  * @brief  Retargets the C library printf function to the USART.
  * @param  None
  * @retval None
  */
PUTCHAR_PROTOTYPE
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the USART */
  USART_SendData(EVAL_COM1, (uint8_t) ch);       /*发送一个字符函数*/

  /* Loop until the end of transmission */
  while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TC) == RESET)     /*等待发送完成*/
  {
  }

  return ch;
}
```

其中，“USART_SendData(EVAL_COM1, (uint8_t) ch);”是调用“stm32f10x_usart.c”库文件中的字符发送函数，然后是while循环等待发送结束“(USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TC) == RESET)”。如果不等待发送结束就返回，则在字符串发送函数中PUTCHAR_PROTOTYPE函数会被连续循环调用多次，如果上一次发送的字节在寄存器中还没有发送结束，此时再次写同样的发生寄存器会将刚才的数据覆盖掉，接收侧将得不到预期的数据。

接下来，我们再看看与USART相关的结构体，主要是用来配置串口的波特率，数据位，奇偶校验等信息的。

```
typedef struct
{
  uint32_t USART_BaudRate;
  uint16_t USART_WordLength;
  uint16_t USART_StopBits;
  uint16_t USART_Parity;
  uint16_t USART_Mode;
  uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;
```

这是关于USART_InitTypeDef结构体定义。

了解了库函数中关于USART的一些相关的函数，结构体定义后，我们回到本实验中。本实验中直接调用库函数进行设计。在本实验的主程序中，需要对USART_InitTypeDef结构体进行初始化的赋值，配置串口的基本信息，如下所示：

```
  /* 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
  */
  /*串口配置，主要与串口终端软件的设置一致*/
  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;             /*没有奇偶校验码*/
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;   /*没有数据流控制*/
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;          /*发送或接收*/

  /*完成串口COM1的时钟配置、GPIPO配置，根据上述参数初始化并使能*/
  STM_EVAL_COMInit(COM1, &USART_InitStructure);
```

循环打印，以及附加点灯：

```
/*重复发送~神舟I号 串口1测试程序，并且LED1（PA2）灯闪烁~*/
 while (1)
 {
        GPIO_ResetBits(GPIOA, GPIO_Pin_2);
        Delay(0x3FFFFF);
        GPIO_SetBits(GPIOA, GPIO_Pin_2);
        Delay(0x3FFFFF);
        printf("\n\r神舟I号 串口1测试程序\n");
 }
```

6.3.5 下载与现象

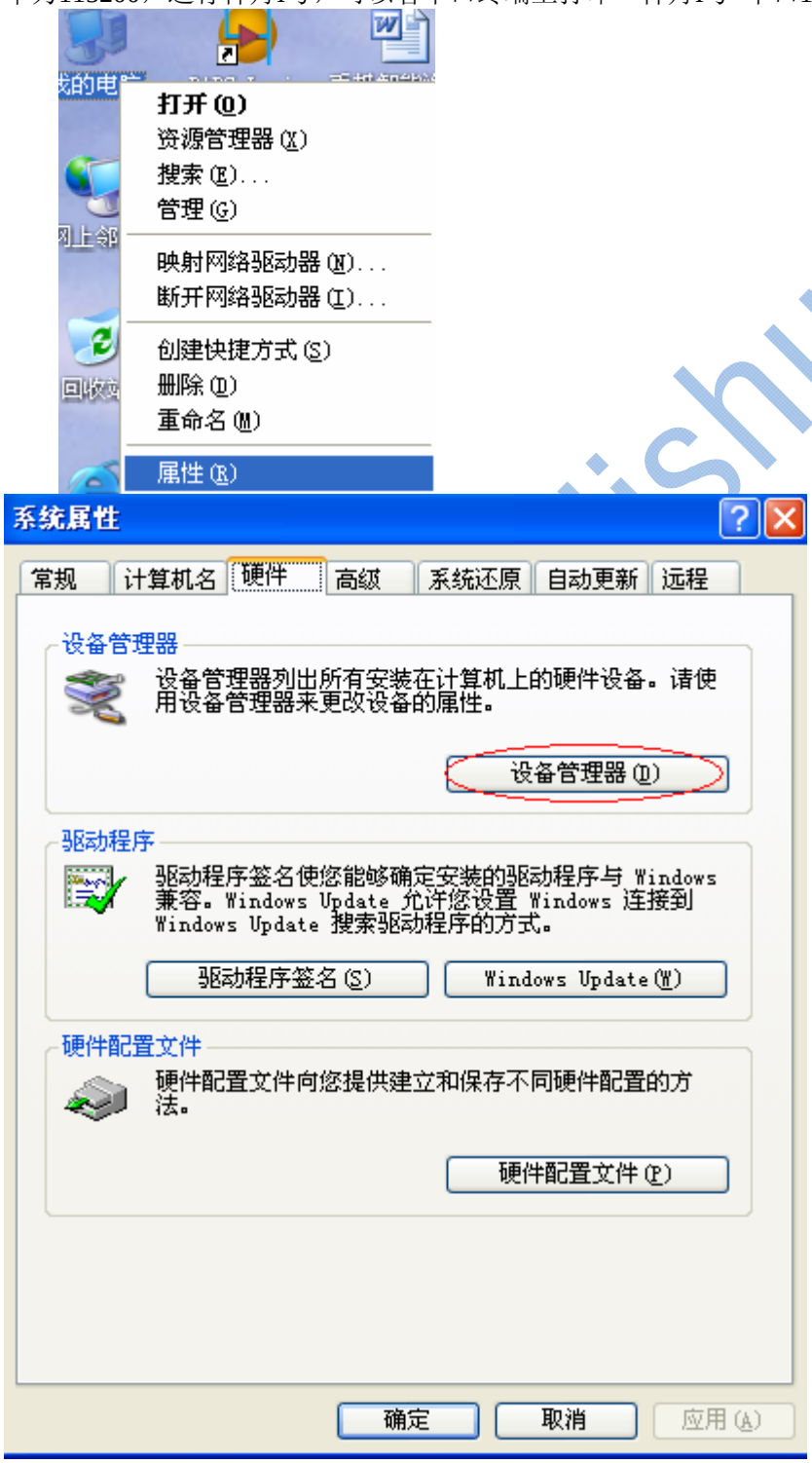
在 *神舟I号光盘\编译好的固件*目录下的*COM1发送程序.hex*文件即为前面我们分析的USART-COM串口发送实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

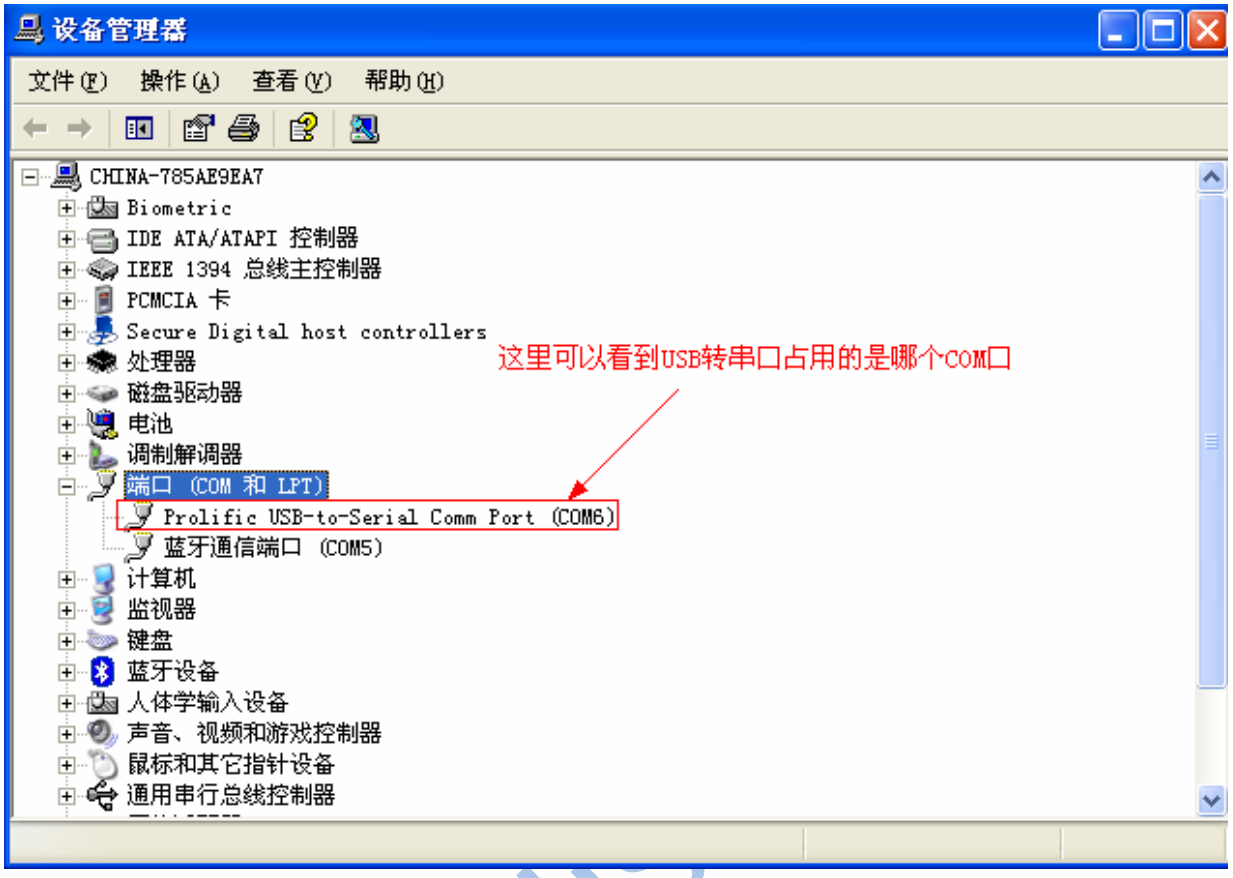
如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I号开发板* 小节进行操作。

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I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。

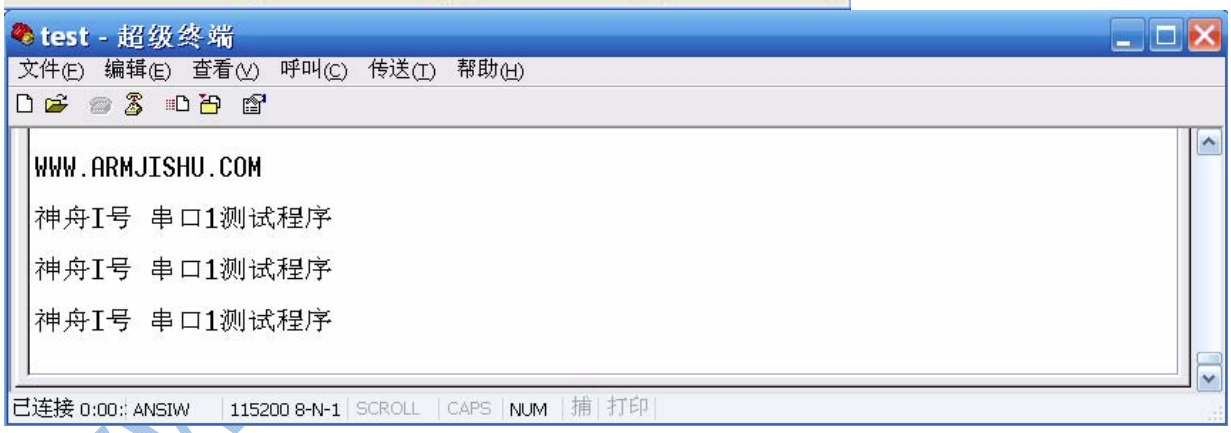
测试现象：下载固件到神舟I号开发板上，连接神舟I号的串口到PC上，选择相关的COM号，波特率为115200，运行神舟I号，可以看串口终端上打印“神舟I号 串口1测试程序”字符。





【开始】→【程序】→【附件】→【通讯】→【超级终端】打开新建一个超级终端

COM6 属性
端口设置
每秒位数(B): 115200
数据位(D): 8
奇偶校验(P): 无
停止位(S): 1
数据流控制(F): 硬件
还原为默认值(R)
确定
取消
应用(A)


test - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
WWW.ARMJISHU.COM
神舟I号 串口1测试程序
神舟I号 串口1测试程序
神舟I号 串口1测试程序
已连接 0:00:: ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印

6.4 USART-COM串口发送与接收实验

接着上一节的内容，这节我们重点来学习串口的收的功能，通过接收键盘等终端输入设备后，将接收的数据进行显示。同时在了解了上一节串口的基本原理以及硬件设计后，本节将不再重复，直接软件的设计。

6.4.1 验的意义与作用

串口的接收功能同样非常重要，发送功能可以将系统告警信息，系统运行状态或是采集数据等进行显示；而接收功能，则是可以将host端的控制命令等接收后并交与MCU处理，实现对板件、或是模块的控制，这将为我们的开发过程或是调试过程带来极大的方便。

6.4.2 实验原理

请详见上一节3.3 “USART-COM串口发送实验”的实验原理。

6.4.3 硬件设计

请详见上一节3.3 “USART-COM串口发送实验”的硬件设计。

6.4.4 软件设计

上节的软件设计中提到了几个有关于串口的发送（输出）函数，这一节我们主要了解串口的一些接收相关的函数，而至于串口的波特率等设置，在此不重复讲述。

首先我们看看main主程序中的循环设计：

```
while (1)
{
        GPIO_SetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);      /*关闭所有的LED指示灯*/
        GPIO_SetBits(GPIOB,GPIO_Pin_2);
        GetInputString(inputstr);                       /*获取键盘输入的字符*/
        printf("\r\n\n\n WWW.ARMJISHU.COM your input is:\n\r[%s]\n\r", inputstr); /*将输入字符进行显示*/
        GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);        /*打开所有的LED指示灯*/
        GPIO_ResetBits(GPIOB,GPIO_Pin_2);
        Delay(0x8FFFFF);
```

其中GPIO_SetBits和GPIO_ResetBits函数是用来关闭LED灯和打开LED灯；而用来接收键盘的输入字符的函数GetInputString(inputstr)，具体的设计如下所示：

```
void GetInputString (uint8_t * buffP)
{
  uint32_t bytes_read = 0;
  uint8_t c = 0;
  do
  {
    c = GetKey();              /*输入的字符*/
    if (c == '\r')             /*判断，输入为~对齐~*/
      break;
    if (c == '\b')             /* 输入为空格 */
    {
      if (bytes_read > 0)      /*如果有其他的字符*/
      {
        printf("\b \b");       /*在串口窗口，打印空格~ ~*/
        bytes_read --;
      }
      continue;
    }
    if (bytes_read >= (CMD_STRING_SIZE))   /*溢出*/
    {
      printf("Command string size overflow\r\n");
      bytes_read = 0;
      continue;
    }
    if (c >= 0x20 && c <= 0x7E)        /*有效字符内，进行接收*/
    {
      buffP[bytes_read++] = c;         /*将接收字符，放在buf中*/
      SerialPutChar(c);                /*打印字符c*/
    }
  }
  while (1);
  printf(("\n\r"));                    /*回车换行*/
  buffP[bytes_read] = '\0';
}
```

128个

通过上述的函数，我们可以了解到，由键盘输入时存在几种情况：一、是特殊的字符，如“回车，换行”，空格；二、输入超过128个键盘键子，视为溢出；三、当输入字符符合要求后，将字符进行存储，同时调用串口的函数，判断串口发送缓冲区是否为空，进行了解串口是否将打印完毕。如下所示：

```
/**
  * @brief  Print a character on the HyperTerminal
  * @param  c: The character to be printed
  * @retval None
  */
void SerialPutChar(uint8_t c)
{
  USART_SendData(EVAL_COM1, c);
  while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TXE) == RESET)
  {
  }
}
```

其实，按照我们上节学习的，SerialPutChar函数中的参数USART_FLAG_TXE也可以使用USART_FLAG_TC来替代的。

另外，c = GetKey()是用来接收键盘输入的字符的，我们具体了解GetKey函数的设计：

```
/**
  * @brief  Get a key from the HyperTerminal
  * @param  None
  * @retval The Key Pressed
  */
uint8_t GetKey(void)
{
  uint8_t key = 0;

  /* Waiting for user input */
  while (1)
  {
    if (SerialKeyPressed((uint8_t*)&key)) break;
  }
  return key;

}
```

而SerialKeyPressed（）则是通过函数USART_GetFlagStatus（）串口数据寄存器接收标志函数来实现的，判断标志位“USART_FLAG_RXNE” 是否为“非空”，如果“非空”，并将数据返回；至于USART_GetFlagStatus（）函数，请详见“stm32f10x_usart.c”文件中的介绍。

```
 /*串口输入函数*/   /*主要用来接收键盘输入的按键信息*/
uint32_t SerialKeyPressed(uint8_t *key)
{

  if ( USART_GetFlagStatus(EVAL_COM1, USART_FLAG_RXNE) != RESET)      /*判断是否有数据接收*/
  {
    *key = (uint8_t)EVAL_COM1->DR;              /*将数据返回*/
    return 1;                                   /*返回值1*/
  }
  else
  {
    return 0;                                   /*不返回*/
  }
}
```

6.4.5 下载与现象

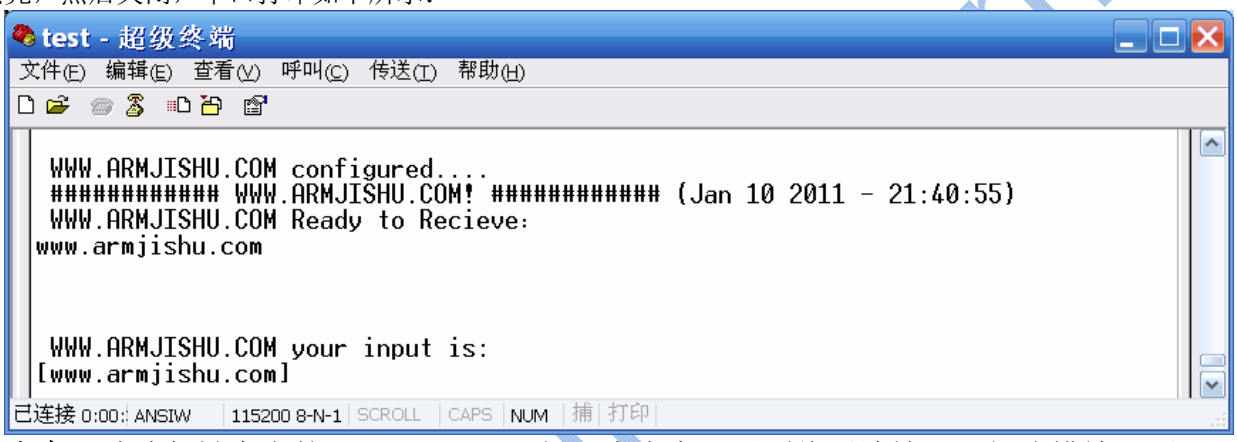
在 *神舟I号光盘\编译好的固件*目录下的*COM1发送接收程序.hex*文件即为前面我们分析的USART-COM串口接收与发送实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I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I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。

测试现象：连接好神舟I号到PC，打开串口软件，设置好波特率为115200，运行程序，看到提示后，输入“www.armjishu.com”字符后，串口打印，同时注意到在你回车的瞬间，LED1~3灯都会被点亮，然后关闭，串口打印如下所示：



注意：请确保键盘上的“Scroll Lock”灯没有点亮，否则将无法输入。解决措施：再按一下“Scroll Lock”按钮，“Scroll Lock”灯灭，此时可以正常输入。

www.armjishu.com

6.5 ADC模数转换实验

在学习了STM32的GPIO口的操作，以及USART-COM串口后，在之前的基础上，这届我们将接着简单学习STM32的ADC操作，神舟I号将在电位器上采集到的数据通过ADC转换后，通过PC的串口将数据打印出来。

6.5.1 实验的意义与作用

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，受到干扰严重，直接限制、影响我们的通信传输质量。因此，在现阶段的做法，在数据进行传输第一步都会进行模拟/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传输。

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后，从PC的串口打印的过程。

6.5.2 实验原理

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转换可以单词、连续、扫描或间断式执行。ADC的结果可以左对齐或是右对齐方式u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。

对于ADC中涉及到的几个寄存器，如ADC控制器（ADC_CR）、ADC的采样时间寄存器（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位高性能微控制器参考手册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”中有相关定义以及初始化。ADC的项目工程中，直接调用。

说到ADC时，那么我们就有必要提及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，所以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。但是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是用来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的。无须CPU 干预，而数据可以通过DMA快速的搬移，节省了CPU的资源。这样便为快速的，高性能的ADC提供通道。简单了解DMA具有多个channel即可。详细请参阅《【中文】STM32F系列ARM内核32位高性能微控制器参考手册V10_1》资料142页开始的关于DMA的描述。

www.armjishu.com

6.5.3 硬件设计

神舟I号的电位器与PB1管脚相连，在此之前PB1也用于315M无线模块的数据管脚上，注意管脚间的复用。

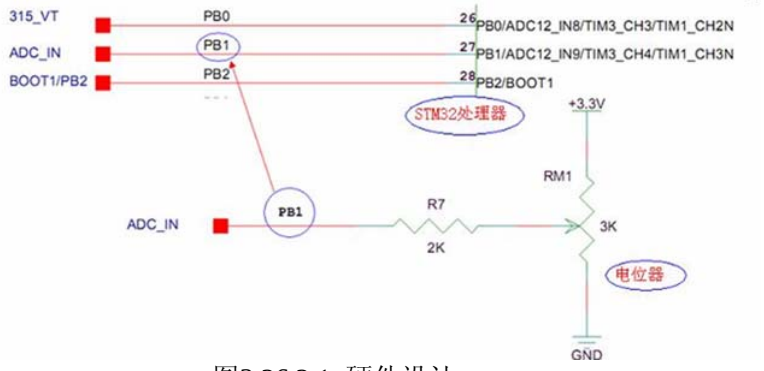


图3.26.2.1 硬件设计

6.5.4 软件设计

如上面实验原理中提到的，本节的软件设计主要是针对ADC转换器以及DMA转换器的初始化以及调用，至于串口方面在上节已经做了介绍，在此只是提及而已。

神舟I号的“05.ADC模数转换实验（神舟I号）”位于神舟I好光盘/神舟I号源码/STM32F10X_StdPeriph_Lib_V3.3.3.rar\\Project\05.ADC模数转换实验（神舟I号）目录中。进入/05.ADC转换实验（神舟I号）\MDK-ARM目录后，双击Project.uvproj可以打开工程，以下工程文件中主要代码的解释与说明。先对使用的变量以及设备进行声明

```
/* Private define ------------------------------------------------------------*/
#define ADC1_DR_Address    ((uint32_t)0x4001244C)    //ADC数据寄存器的基地址
/* Private macro -------------------------------------------------------------*/
/* Private variables ---------------------------------------------------------*/
USART_InitTypeDef USART_InitStructure;          //串口、ADC、DMA声明

ADC_InitTypeDef ADC_InitStructure;
DMA_InitTypeDef DMA_InitStructure;
// 注：ADC为12位模数转换器，只有ADCConvertedValue的低12位有效
__IO uint16_t ADCConvertedValue;
```

串口的配置

```
/* 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;                                    //串口初始化定义
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);              //使用COM1
```

串口的打印信息

```
/* Output a message on Hyperterminal using printf function */                    //串口打印
printf("\n\r\n\r\n\r\n\r");
printf("\n\rUSART Printf Example: retarget the C library printf function to the USART\n\r");
printf("\r\n\n\n WWW.ARMJISHU.COM configured....");
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==============================================================================");
printf("\n\r");
```

下面主要是DMA得结构体的使用配置，包括时钟使能、DMA通道选择、目标存储地址，转换模式、数据宽度以及优先级等等。

```
/* Enable DMA1 clock */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);          //使能DMA时钟

/* Enable ADC1 and GPIOC clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOC, ENABLE);     //使能ADC和GPIOC时钟

/* DMA1 channel1 configuration ----------------------------------------------*/
DMA_DeInit(DMA1_Channel1);          //开启DMA1的第一通道
DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;          //DMA对应的外设基地址
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADCConvertedValue;    //内存存储基地址
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);
while (1)
{
  ADCConvertedValueLocal = ADCConvertedValue;
  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\n\n ARMJISHU.COM ADCConvertedValue is 0x%X, Precent is %d%%, Voltage is %d.%d%dV",      //将计算的进行打印
  ADCConvertedValueLocal, Precent, Voltage/1000, (Voltage%1000)/100, (Voltage%100)/10);

  printf("\r\n ARMJISHU.COM 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);

  Delay_ARMJISHU(8000000);
}
```

数据发送

```
PUTCHAR_PROTOTYPE
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the USART */
  USART_SendData(EVAL_COM1, (uint8_t) ch);          //发送一字节数据

  /* Loop until the end of transmission */
  while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TC) == RESET)
  {}                  //等待发送完成

  return ch;
}
```

对于ADC连接的GPIO管脚配置进行说明

```
/**
  * @brief  Configures the different GPIO ports.
  * @param  None
  * @retval None
  */
void ADC_GPIO_Configuration(void)                          //ADC配置函数
{
  GPIO_InitTypeDef GPIO_InitStructure;
  //PB1 作为模拟通道输入引脚
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;                //管脚1
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;            //输入模式
  GPIO_Init(GPIOB, &GPIO_InitStructure);                   //GPIO组
}
```

其它涉及到的有关于ADC的寄存器在ST标准库中的外设驱动文件“stm32f10x_adc.c”中已经声明了，在此直接调用，不继续深究。

6.5.5 下载与现象

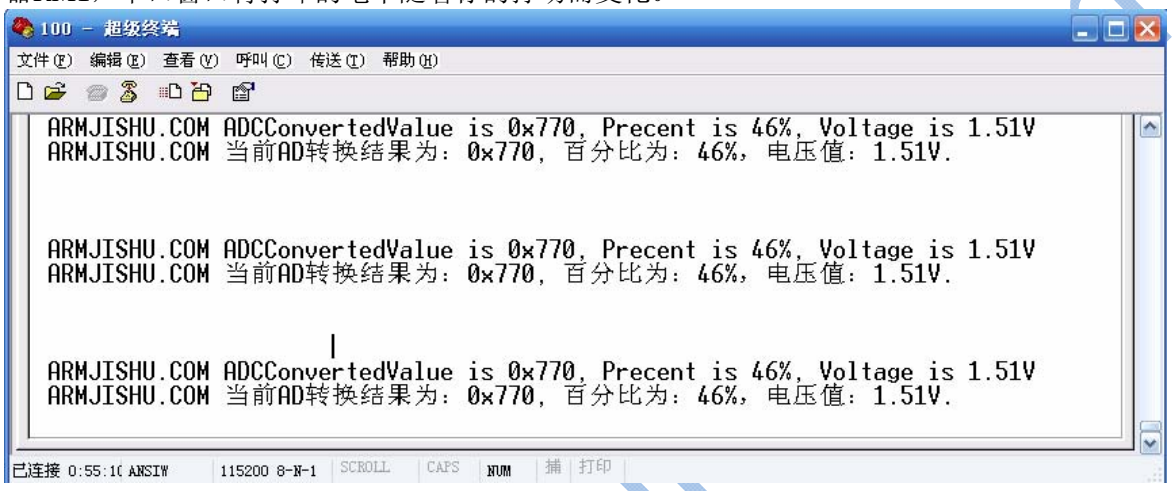
在 *神舟I号光盘\神舟I号各例程下载固件*目录下的 *"05.ADC模数转换..hex"* 文件即为前面我们分析*05.ADC模数转换实验*的编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I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I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。

测试现象：下载固件后，复位运行程序，可以从串口打印信息了解到电位器对应的电平值，调动电位器RM1，串口窗口将打印的电平随着你的拧动而变化。



www.armjishu.com

6.6 EEPROM读写程序实验

6.6.1 实验的意义与作用

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)，读写访问简单。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。

6.6.2 试验原理

24CXX系列EEPROM采用的访问接口是I2C接口。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。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总线在传送数据过程中共有三种类型信号， 它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL为高电平时，SDA由高电平向低电平跳变，开始传送数据。

结束信号：SCL为高电平时，SDA由低电平向高电平跳变，结束传送数据。

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总线时序图如下：

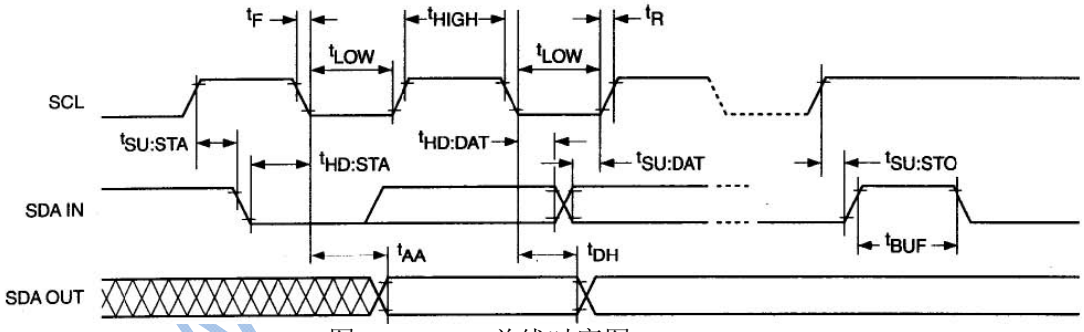


图3.16.1.1 IIC总线时序图

神舟I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。

下面我们来了解实验的基本原理：神舟I号通过STM32F103RBT6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EERPOM是否可以正常访问。

神舟I号板上的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。

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I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I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I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I^2C总线
- I^2C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

神舟I号使用的EEPROM芯片型号为24C02,该芯片的容量为2Kbit，也就是256个字节，对于我们一般的应用是足够了。但是你也可以依据实际的需要，更换更大容量的芯片，因为在原理上是兼容24C02~24C512全系列的EEPROM芯片的。

6.6.3 硬件设计

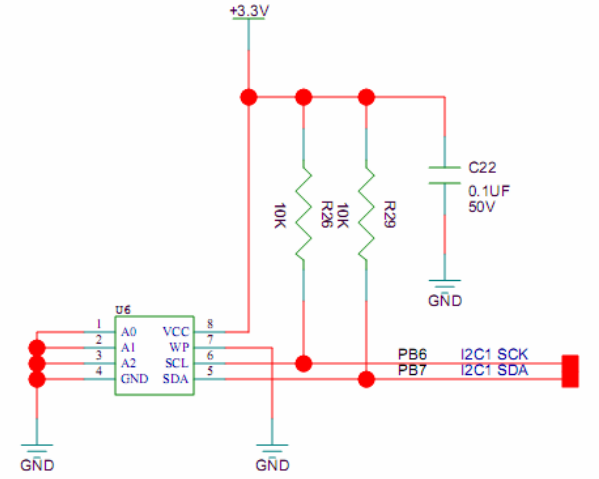
本实验需要用到的硬件资源：

- 串口 1：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 《串口的发送与接收实验》。
- I2C EEPROM 24C02

STM32F103RBT6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I号开发板上使用I2C1与24C02芯片进行相连，其原理图如下：



如上图中，A0~A2全部接地，对应I2C的硬件地址为0xA0。

6.6.4 软件设计

神舟 I 号 EEPROM 读写试验位于 *神舟I 号光盘\源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\\Project\06.EEPROM读写程序（神舟I 号）*目录。

进入*06.EEPROM读写程序（神舟I 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。

第一步我们先了解 I2C1 的初始化函数：

```
void I2C_Configuration(void)
{
    I2C_InitTypeDef  I2C_InitStructure;
    GPIO_InitTypeDef  GPIO_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1,ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);

    /* PB6,7 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;      //设置管脚为复用共呢问你个开漏输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);              //I2C接口使用的GPIO管脚初始化

    I2C_DeInit(I2C1);
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;           //设置I2C接口模式
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;   //设置I2C接口的高低电平周期
    I2C_InitStructure.I2C_OwnAddress1 = 0x30;            //设置I2C接口的主机地址
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;          //设置是否开启ACK响应
    I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = 100000;           //100K速度

    I2C_Cmd(I2C1, ENABLE);                               //使能I2C接口
    I2C_Init(I2C1, &I2C_InitStructure);                  //I2C接口初始化
    /*允许1字节1应答模式*/
    I2C_AcknowledgeConfig(I2C1, ENABLE);                 //使能I2C接口响应

    printf("I2C_Configuration----\n\r");
}
```

I2C1 的初始化函数中，首先将 I2C1 接口初始化为复用功能开漏输出，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 SMBA 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。

初始化后 I2C1 的接口后，我们就可以通过 I2C1 对 EEPROM AT24C02 芯片进行读写操作了。下面学习 EEPROM 读写访问程序：

```
void I2C_Test(void)
{
    u16 i;
    u8 I2c_Buf[256];
    printf("写入数据\n\r ");
    //填充缓冲
    for(i=0;i<=255;i++)
    {
        I2c_Buf[i]=i;
        printf("0x%x ",I2c_Buf[i]);
        if(i%16 == 15)                  //每16个换行
        {
            printf("\n\r");
        }
    }
    printf("\n\r");
    I2C_WriteS_24C(0,I2c_Buf,256);    //将I2C_Buf中顺序递增的数据写入到EEPROM中
    //清缓冲
    for(i=0;i<=255;i++)
    {
        I2c_Buf[i]=0;
    }
    //读操作
    printf("读出的数据\n\r");
    I2C_ReadS_24C(0,I2c_Buf,256);   //将EEPROM读出的数据顺序保持到I2C_Buf中

    //打印I2C_Buf中的数据
    for(i=0;i<256;i++)
    {
        if(I2c_Buf[i]!=i)
        {
            printf("错误: I2C EEPROM写入与读出的数据不一致\n\r");
            while(1);
        }
        printf("0x%X ", I2c_Buf[i]);
        if(i%16 == 15)                  //每16个换行
        {
            printf("\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。由于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。本函数也是用来作为上电检测I2C EEPROM是否在位，或是异常的检测函数。

其中I2C_WriteS_24C函数，是24CXX EEPROM的页写入函数，在神舟I号中，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：

```
void I2C_WriteS_24C(u8 addr,u8* pBuffer,  u16 no)
{
    u8 temp;

    //先把页不对齐的部分写入
    temp=addr % I2C_PAGESIZE;
    if(temp)
    {
        temp=I2C_PAGESIZE-temp;
        I2C_PageWrite_24C(addr,pBuffer,  temp);     //将页不对齐的字节写入EEPROM
        no-=temp;
        addr+=temp;
        pBuffer+=temp;
        I2C_Standby_24C();                          //判断EEPROM是否忙
    }
    //从页对齐开始写
    while(no)
    {
        if(no>=I2C_PAGESIZE)
        {
            I2C_PageWrite_24C(addr,pBuffer,  I2C_PAGESIZE);     //将页对齐的字节写入EEPROM
            no-=I2C_PAGESIZE;
            addr+=I2C_PAGESIZE;
            pBuffer+=I2C_PAGESIZE;
            I2C_Standby_24C();          //判断EEPROM是否忙
        }
        else
        {
            I2C_PageWrite_24C(addr,pBuffer,  no);
            no=0;
            I2C_Standby_24C();
        }
    }
}
```

而对于读函数 I2C_ReadS_24C（）是用来读取 24CXX EEPROM 的页数据的，其具体的函数实现如下所示：

```
void I2C_ReadS_24C(u8 addr ,u8* pBuffer,u16 no)
{
    if(no==0) return;
    while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));
    /*允许1字节1应答模式*/
    I2C_AcknowledgeConfig(I2C1, ENABLE);
    /* 发送起始位 */
    I2C_GenerateSTART(I2C1, ENABLE);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));/*EV5,主模式*/
#ifdef AT24C01A
    /*发送器件地址(写)*/
    I2C_Send7bitAddress(I2C1,  EEPROM_ADDR, I2C_Direction_Transmitter);
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
    /*发送地址*/
    I2C_SendData(I2C1, addr);
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));/*数据已发送*/
    /*起始位*/
    I2C_GenerateSTART(I2C1, ENABLE);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
    /*器件读操作*/
    I2C_Send7bitAddress(I2C1, EEPROM_ADDR, I2C_Direction_Receiver);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
#else
    /*发送器件地址(读)24C01*/
    I2C_Send7bitAddress(I2C1, addr<<1, I2C_Direction_Receiver);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
#endif
    while (no)
    {
        if(no==1)
        {
            I2C_AcknowledgeConfig(I2C1, DISABLE);   //最后一位后要关闭应答的
            I2C_GenerateSTOP(I2C1, ENABLE);         //发送停止位
        }
        while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED)); /* EV7 */
        *pBuffer = I2C_ReceiveData(I2C1);
        pBuffer++;
        no--;     /*递减读字节计数器*/
    }
    //再次允许应答模式
    I2C_AcknowledgeConfig(I2C1, ENABLE);
}
```

6.6.5 下载与测试

在 *神舟I号光盘\编译好的固件*目录下的EEPROM读写.hex文件即为前面我们分析的EEPROM读写实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I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I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I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。

测试现象：将固件下载到板件，连接串口，设置波特率为115200，运行程序，可以看到串口的打印信息，如下图所示，表示EEPROM读写正常。

```
 WWW.ARMJISHU.COM configured....
 ############ WWW.ARMJISHU.COM! ############ (Jan  4 2011 - 23:05:03)I2C_Configuration----
写入数据
0x0  0x1  0x2  0x3  0x4  0x5  0x6  0x7  0x8  0x9  0xa  0xb  0xc  0xd  0xe  0xf
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f
0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f
0x50 0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5a 0x5b 0x5c 0x5d 0x5e 0x5f
0x60 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f
0x70 0x71 0x72 0x73 0x74 0x75 0x76 0x77 0x78 0x79 0x7a 0x7b 0x7c 0x7d 0x7e 0x7f
0x80 0x81 0x82 0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8a 0x8b 0x8c 0x8d 0x8e 0x8f
0x90 0x91 0x92 0x93 0x94 0x95 0x96 0x97 0x98 0x99 0x9a 0x9b 0x9c 0x9d 0x9e 0x9f
0xa0 0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8 0xa9 0xaa 0xab 0xac 0xad 0xae 0xaf
0xb0 0xb1 0xb2 0xb3 0xb4 0xb5 0xb6 0xb7 0xb8 0xb9 0xba 0xbb 0xbc 0xbd 0xbe 0xbf
0xc0 0xc1 0xc2 0xc3 0xc4 0xc5 0xc6 0xc7 0xc8 0xc9 0xca 0xcb 0xcc 0xcd 0xce 0xcf
0xd0 0xd1 0xd2 0xd3 0xd4 0xd5 0xd6 0xd7 0xd8 0xd9 0xda 0xdb 0xdc 0xdd 0xde 0xdf
0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6 0xe7 0xe8 0xe9 0xea 0xeb 0xec 0xed 0xee 0xef
0xf0 0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8 0xf9 0xfa 0xfb 0xfc 0xfd 0xfe 0xff

读出的数据
0x0  0x1  0x2  0x3  0x4  0x5  0x6  0x7  0x8  0x9  0xA  0xB  0xC  0xD  0xE  0xF
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

 ############ WWW.ARMJISHU.COM! ############ test finish!
```

6.7 SPI FLASH（W25X16）读写程序实验

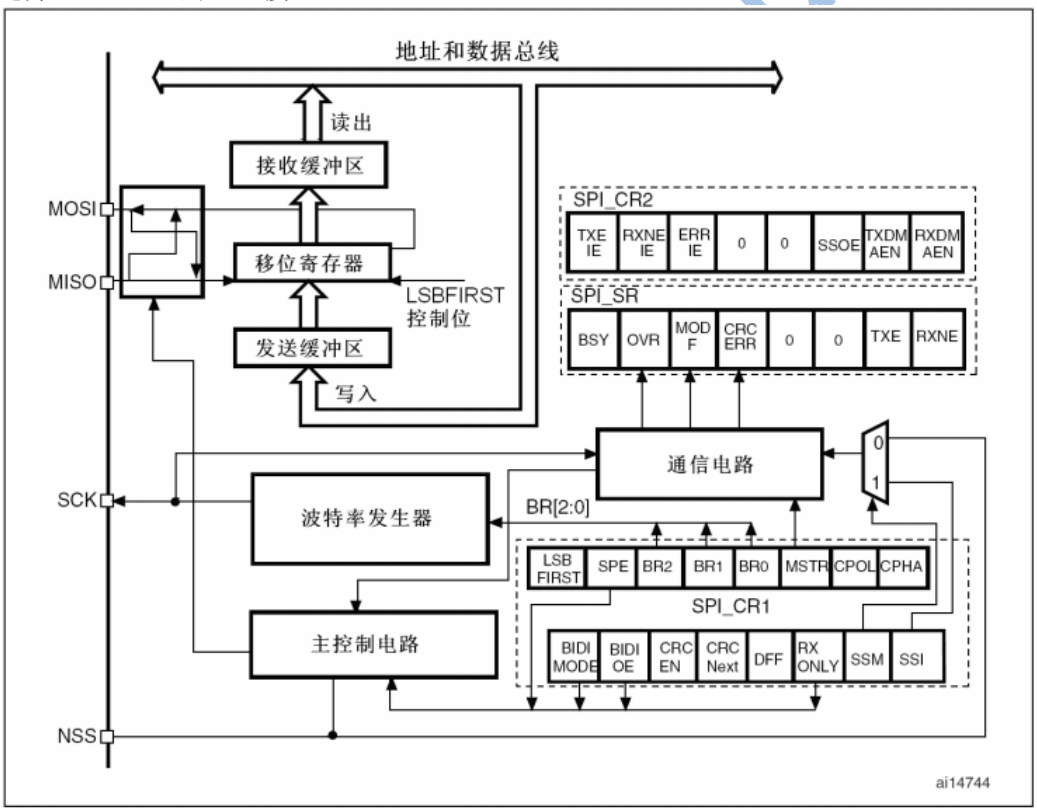
这一节我们借用串口向大家介绍SPI。本节将利用SPI来实现对LCD屏上的FLASH（W25X16）的读写，并将结果通过串口显示在PC机上。后续升级，便将显示结果从LCD屏上显示出来。本节分为如下几个部分：

6.7.1 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 I 号硬件上使用到 SPI 接口的有，W25X16，ADS7843 触摸芯片，这两个都位于 LCD 屏上，还有母板上的扩展接口上的 2.4G 无线模块接口，也是使用 SPI 接口。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。

6.7.2 实验原理

SPI（串行外设接口）是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议，STM32 也有 SPI 接口，可以配置为 SPI 协议或者 I2S 协议。。下面先介绍 STM32 的 SPI 接口：



通常 SPI 通过 4 个引脚与外部器件相连：

- MISO：主设备输入/从设备输出引脚。该引脚在从模式下发送数据，在主模式下接收数据。
- MOSI：主设备输出/从设备输入引脚。该引脚在主模式下发送数据，在从模式下接收数据。
- SCK：串口时钟，作为主设备的输出，从设备的输入
- 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

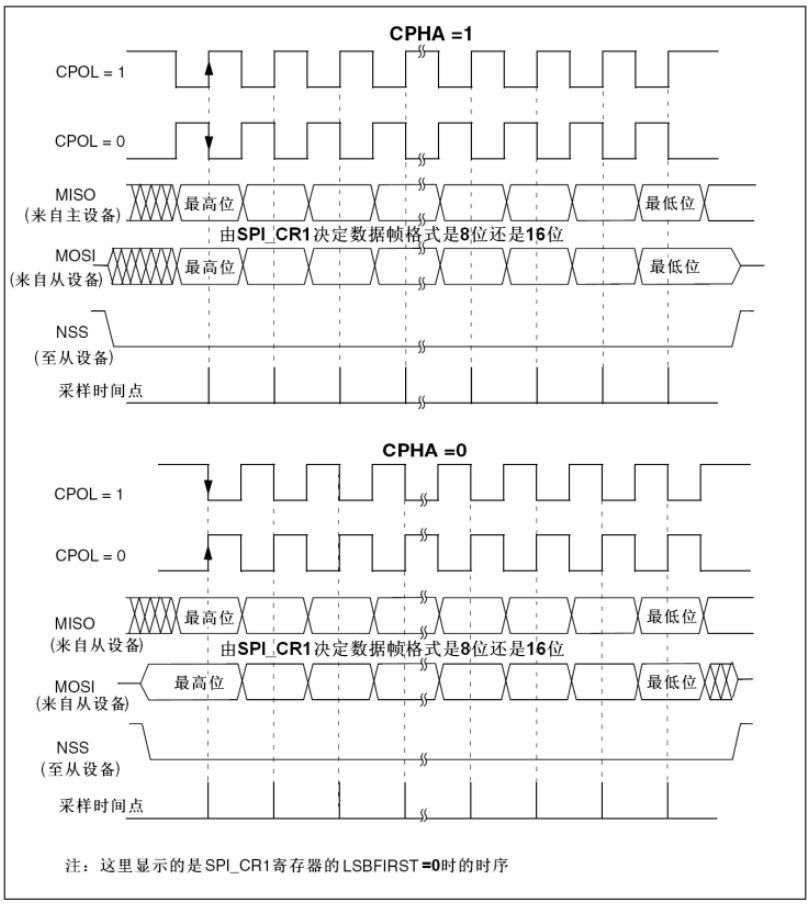
备的一个标准 I/O 引脚来驱动。一旦被使能(SSOE 位)，NSS 引脚也可以作为输出引脚，并在 SPI 处于主模式时拉低；此时，所有的 SPI 设备，如果它们的 NSS 引脚连接到主设备的 NSS 引脚，则会检测到低电平，如果它们被设置为 NSS 硬件模式，就会自动进入从设备状态。当配置为主设备、NSS 配置为输入引脚(MSTR=1，SSOE=0)时，如果 NSS 被拉低，则这个 SPI 设备进入主模式失败状态：即 MSTR 位被自动清除，此设备进入从模式。

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'时就是下降沿，CPOL 位为'1'时就是上升沿)进行数据位采样，数据在第一个时钟边沿被锁存。

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。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。



图表 4 数据时钟时序图

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 寄存器中的 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 脚缠身串行时钟。

配置步骤：

a) 通过 SPI_CR1 寄存器的 BR[2:0]位定义串行时钟波特率；
b) 选择CPOL和CPHA位，定义数据传输和串行时钟间的相位关系，如图表 5 数据时钟时序图所示。
c) 设置 DFF 位来定义 8 位或 16 位数据帧格式；
d) 配置 SPI_CR1 寄存器的 LSBFIRST 位定义帧格式；
e) 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；
f) 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。

接下来简单了解一下W25X16芯片。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。

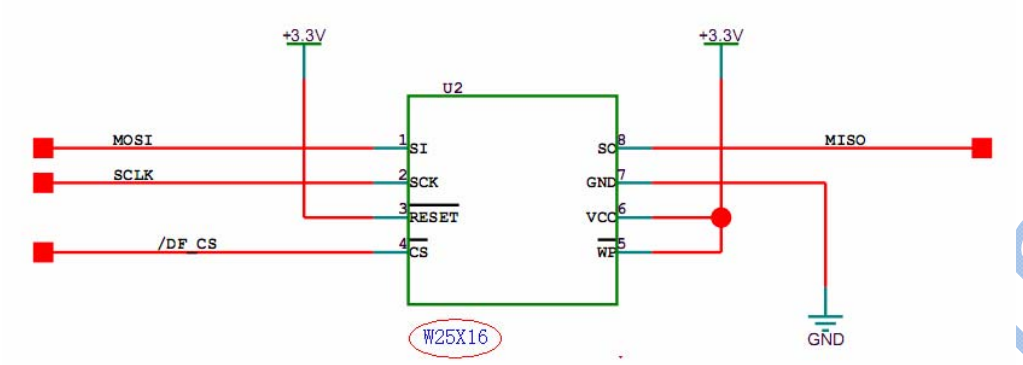
W25X16芯片将2M的容量分为32个块（Block），每个块大小为64K字节，每个块又分为16个扇区（Sector），每个扇区4K个字节。W25X16的最少擦除单位为一个扇区，也就是每次必须擦除4K个字节。这样我们需要给W25X16开辟一个至少4K的缓存区。

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I号光盘\神舟I号相关参考资料\外围器件数据手册”文件下的《W25X16 SPI Flash数据手册.pdf》。

www.armjishu.com

6.7.3 硬件设计

W25X16芯片位于LCD屏上，通过TFT LCD屏座（J1）与STM32的SPI1接口相连。在LCD屏的原理设计如下：



神舟I号母板的SPI的设计如下：

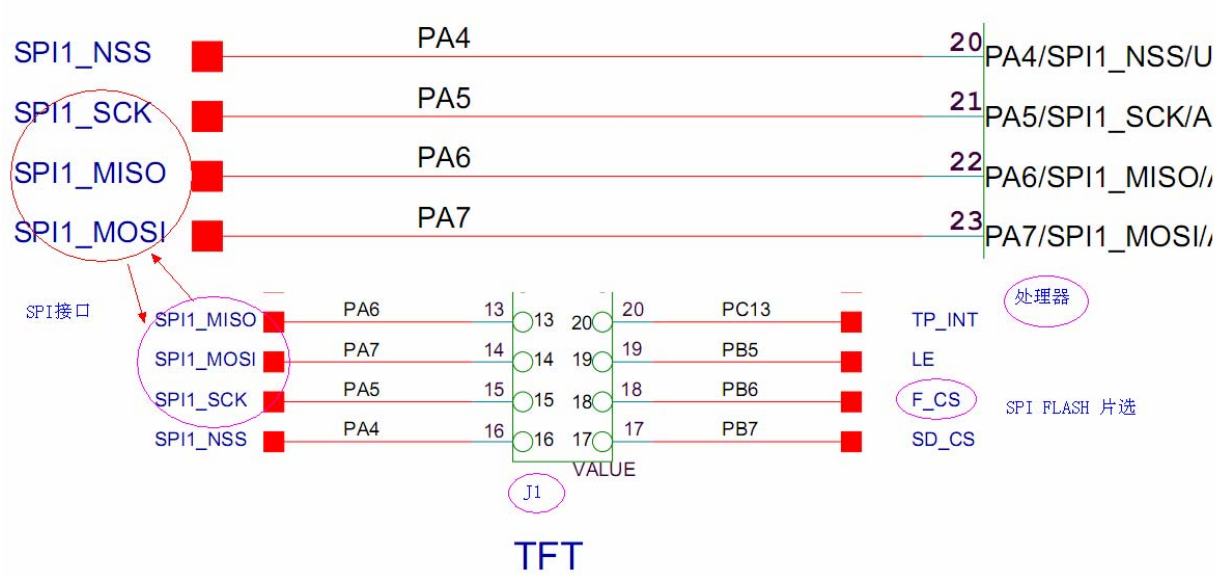


图3.17.2.1 STM32F103RBT6与W25X16连接电路图

同时，在实现SPI1读写W25X16芯片时，LED1灯也慢速的闪烁，LED灯的原理设计在前面已经详细描述，在此不再累赘。

6.7.4 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，如下所示：

```
//神舟I号 开发板

#ifndef __SPI_H                          //头文件声明
#define __SPI_H
#include "stm32f10x.h"                   //包括的文件

void SPI1_Init(void);                    //初始化SPI口
u8 SPI1_ReadWriteByte(u8 Data);        //SPI读写一个字节的函数

#endif
```

下面介绍spi.c文件

本例程中我们使用SPI1，主模式读取FLASH芯片，初始化SPI1接口

```
//神舟I号 开发板
#include "spi.h"

//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W25X16进行读写操作，先对SPI1进行初始化
void SPI1_Init(void)
    {
    SPI_InitTypeDef  SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA|RCC_APB2Periph_SPI1, ENABLE );

    //SPI1口初始化
    /* Configure SPI1 pins: SCK, MISO and MOSI */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /*配置FLASH的片选 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;          //SPI CS PB6
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  //复用推挽输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    GPIO_SetBits(GPIOB,GPIO_Pin_6);                   //先不使能SPI CS

    /* SPI1 configuration */                                  //初始化SPI1结构体
    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_256;  //定义波特率预分频的值:波特率预分频值为256
    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);                                       //根据SPI_InitStruct中指定的参数初始化外设SPI1寄存器

    /* Enable SPI1  */
    SPI_Cmd(SPI1, ENABLE);                                                    //使能SPI1外设

    SPI1_ReadWriteByte(0xff);                                                 //启动传输
    }
```

SPI接口读写FLASH的一个字节的函数，如下所示：

```
u8 SPI1_ReadWriteByte(u8 Data)        //SPI读写数据函数
    {
    u8 retry=0;
    /* Loop while DR register in not emplty */
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET)    //发送缓存标志位为空
        {
        retry++;
        if(retry>200)return 0;
        }
    /* Send byte through the SPI1 peripheral */
    SPI_I2S_SendData(SPI1, Data);                                     //通过外设SPI1发送一个数据
    retry=0;
    /* Wait to receive a byte */
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET); //接收缓存标志位不为空
        {
        retry++;
        if(retry>200)return 0;
        }
    /* Return the byte read from the SPI bus */
    return SPI_I2S_ReceiveData(SPI1);                                  //通过SPI1返回接收数据
    }
```

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。我们同样先看看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，如下所示：

```
//神舟I号 STM32开发板

#ifndef __FLASH_H
#define __FLASH_H
#include <stm32f10x.h>
//Flash 对应的CS
#define Set_SPI_FLASH_CS  {GPIO_SetBits(GPIOB,GPIO_Pin_6);}
#define Clr_SPI_FLASH_CS  {GPIO_ResetBits(GPIOB,GPIO_Pin_6);}

//W25X16 ID
#define FLASH_ID 0XEF14
// 指令设置     详见《神舟I号光盘》中的《W25X16 SPI Flash数据手册.pdf》第15页
#define W25X_WriteEnable        0x06
#define W25X_WriteDisable       0x04
#define W25X_ReadStatusReg      0x05
#define W25X_WriteStatusReg     0x01
#define W25X_ReadData           0x03
#define W25X_FastReadData       0x0B
#define W25X_FastReadDual       0x3B
#define W25X_PageProgram        0x02
#define W25X_BlockErase         0xD8
#define W25X_SectorErase        0x20
#define W25X_ChipErase          0xC7
#define W25X_PowerDown          0xB9
#define W25X_ReleasePowerDown   0xAB
#define W25X_DeviceID           0xAB
#define W25X_ManufactDeviceID   0x90
#define W25X_JedecDeviceID      0x9F

u16  SPI_Flash_ReadID(void);          //读取FLASH ID
u8   SPI_Flash_ReadSR(void);          //读取状态寄存器
void SPI_FLASH_Write_SR(u8 sr);       //写状态寄存器
void SPI_FLASH_Write_Enable(void);    //写使能
void SPI_FLASH_Write_Disable(void);   //写保护
void SPI_Flash_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead);   //读取flash
void SPI_Flash_Write(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite);//写入flash
void SPI_Flash_Erase_Chip(void);          //整片擦除
void SPI_Flash_Erase_Sector(u32 Dst_Addr);//扇区擦除
void SPI_Flash_Wait_Busy(void);           //等待空闲
void SPI_Flash_PowerDown(void);           //进入掉电模式
void SPI_Flash_WAKEUP(void);              //唤醒
#endif
```

而对于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。

```c
//神舟I号 STM32开发板
#include "flash.h"
#include "spi.h"
#include "delay.h"

//W25X16容量为2M字节,共有32个Block,512个Sector, 一个Block包括16个扇区, 一个Sector为4Kbytes大小。
//W25X16中的一些寄存器的资料, 详见《神舟I号光盘》中的《W25X16 SPI Flash数据手册.pdf》第11页开始

u8 SPI_Flash_ReadSR(void)
{
    u8 byte=0;
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_ReadStatusReg);     //发送读取状态寄存器命令
    byte=SPI1_ReadWriteByte(0Xff);              //读取一个字节
    Set_SPI_FLASH_CS;                           //取消片选
    return byte;
}
//写SPI_FLASH状态寄存器
//只有SPR,TB,BP2,BP1,BP0(bit 7,5,4,3,2)可以写!!!
void SPI_FLASH_Write_SR(u8 sr)
{
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_WriteStatusReg);    //发送写取状态寄存器命令
    SPI1_ReadWriteByte(sr);                     //写入一个字节
    Set_SPI_FLASH_CS;                           //取消片选
}
//SPI_FLASH写使能
//将WEL置位
void SPI_FLASH_Write_Enable(void)
{
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_WriteEnable);           //发送写使能
    Set_SPI_FLASH_CS;                           //取消片选
}
//SPI_FLASH写禁止
//将WEL清零
void SPI_FLASH_Write_Disable(void)
{
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_WriteDisable);      //发送写禁止指令
    Set_SPI_FLASH_CS;                           //取消片选
}

u16 SPI_Flash_ReadID(void)        //读取芯片ID W25X16的ID:0XEF14
{
    u16 Temp = 0;
    Clr_SPI_FLASH_CS;
    SPI1_ReadWriteByte(0x90);//发送读取ID命令
    SPI1_ReadWriteByte(0x00);
    SPI1_ReadWriteByte(0x00);
    SPI1_ReadWriteByte(0x00);
    Temp|=SPI1_ReadWriteByte(0xFF)<<8;
    Temp|=SPI1_ReadWriteByte(0xFF);
    Set_SPI_FLASH_CS;
    return Temp;
}
```

```
//读取SPI FLASH
//在指定地址开始读取指定长度的数据
//pBuffer:数据存储区
//ReadAddr:开始读取的地址(24bit)
//NumByteToRead:要读取的字节数(最大65535)
void SPI_Flash_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead)
{
    u16 i;
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_ReadData);         //发送读取命令
    SPI1_ReadWriteByte((u8)((ReadAddr)>>16));  //发送24bit地址
    SPI1_ReadWriteByte((u8)((ReadAddr)>>8));
    SPI1_ReadWriteByte((u8)ReadAddr);
    for(i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPI1_ReadWriteByte(0XFF);   //循环读数
    }
    Set_SPI_FLASH_CS;                           //取消片选
}


//SPI在一页(0~65535)内写入少于256个字节的数据
//在指定地址开始写入最大256字节的数据
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大256),该数不应该超过该页的剩余字节数!!!
void SPI_Flash_Write_Page(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
{
    u16 i;
    SPI_FLASH_Write_Enable();                  //SET WEL
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_PageProgram);      //发送写页命令
    SPI1_ReadWriteByte((u8)((WriteAddr)>>16)); //发送24bit地址
    SPI1_ReadWriteByte((u8)((WriteAddr)>>8));
    SPI1_ReadWriteByte((u8)WriteAddr);
    for(i=0;i<NumByteToWrite;i++)SPI1_ReadWriteByte(pBuffer[i]);//循环写数
    Set_SPI_FLASH_CS;                           //取消片选
    SPI_Flash_Wait_Busy();                     //等待写入结束
}
//无检验写SPI FLASH
//必须确保所写的地址范围内的数据全部为0XFF,否则在非0XFF处写入的数据将失败!
//具有自动换页功能
//在指定地址开始写入指定长度的数据,但是要确保地址不越界!
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大65535)
//CHECK OK
void SPI_Flash_Write_NoCheck(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
{
    u16 pageremain;
    pageremain=256-WriteAddr%256; //单页剩余的字节数
    if(NumByteToWrite<=pageremain)pageremain=NumByteToWrite;//不大于256个字节
    while(1)
    {
        SPI_Flash_Write_Page(pBuffer,WriteAddr,pageremain);
        if(NumByteToWrite==pageremain)break;//写入结束了
        else //NumByteToWrite>pageremain
        {
            pBuffer+=pageremain;
            WriteAddr+=pageremain;

            NumByteToWrite-=pageremain;            //减去已经写入了的字节数
            if(NumByteToWrite>256)pageremain=256; //一次可以写入256个字节
            else pageremain=NumByteToWrite;        //不够256个字节了
        }
    };
}
```

SPI写FLASH函数

```
u8 SPI_FLASH_BUF[4096];
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++)        //复制
            {
                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
        }
    };
}
```

www.armj

```
void SPI_Flash_Erase_Chip(void)
{
    SPI_FLASH_Write_Enable();                   //SET WEL
    SPI_Flash_Wait_Busy();
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_ChipErase);         //发送片擦除命令
    Set_SPI_FLASH_CS;                           //取消片选
    SPI_Flash_Wait_Busy();                      //等待芯片擦除结束
}

void SPI_Flash_Erase_Sector(u32 Dst_Addr)       //擦除一个扇区
{
    Dst_Addr*=4096;
    SPI_FLASH_Write_Enable();                   //SET WEL
    SPI_Flash_Wait_Busy();
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_SectorErase);       //发送扇区擦除指令
    SPI1_ReadWriteByte((u8)((Dst_Addr)>>16));   //发送24bit地址
    SPI1_ReadWriteByte((u8)((Dst_Addr)>>8));
    SPI1_ReadWriteByte((u8)Dst_Addr);
    Set_SPI_FLASH_CS;                           //取消片选
    SPI_Flash_Wait_Busy();                      //等待擦除完成
}
void SPI_Flash_Wait_Busy(void)                  //等待空闲
{
    while ((SPI_Flash_ReadSR()&0x01)==0x01);    // 等待BUSY位清空
}
void SPI_Flash_PowerDown(void)                  //掉电模式
{
    Clr_SPI_FLASH_CS;                           //使能器件
    SPI1_ReadWriteByte(W25X_PowerDown);         //发送掉电命令
    Set_SPI_FLASH_CS;                           //取消片选
    delay_us(3);                                //等待TPD
}
//唤醒
void SPI_Flash_WAKEUP(void)
{
    Clr_SPI_FLASH_CS;                            //使能器件
    SPI1_ReadWriteByte(W25X_ReleasePowerDown);   //send W25X_PowerDown command 0xAB
    Set_SPI_FLASH_CS;                            //取消片选
    delay_us(3);                                 //等待TRES1
}
```

其中涉及到的delay函数，我们在此不做为详细讲解。

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，在此不累赘。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I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。否则准备好，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();                    //如果读取W25X16的ID不正确时，从串口打印出来
        printf("\n\r ID:%d",i);
        printf("\n\rW25X16 Check Failed!");
        Delay(0xaFFFF);

        printf("\n\rplease check!");          //提示，W25X16需要确认

        Delay(0xaFFFF);
        GPIO_ResetBits(GPIOB,  GPIO_Pin_2);     //点灯提示
        Delay(0xaFFFF);

        GPIO_SetBits(GPIOB,  GPIO_Pin_2);

    }
    printf("\n\rW25X16 Ready!");               //检测到W25X16芯片后，准备读写操作
    //显示提示信息
    printf("\n\r按钮1:Write 按钮2:Read");      //提示按钮意义
```

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();                      //扫描按键是否按下
    if(key==KEY1)                           //如果按钮1按下,写入SPI FLASH
    {
        printf("\n\r Start Write W25X16....");
        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\rW25X16 Write Finished!");   //提示传送完成
    }
    if(key==KEY2)                           //如果按钮2按下,则是读取写入的字符传字符串并显示
    {
        printf("\n\r Start Read W25X16.... ");
        SPI_Flash_Read(datatemp,1000,SIZE);  //从1000地址处开始,读出SIZE个字节，并放在datatemp字符组中
        printf("\n\rThe Data Readed Is: %s  ",datatemp);   //打印字符组数据

    }
    i++;
    Delay(0xFFFF);
    if(i>0&& i<100)
    {
        GPIO_SetBits(GPIOA, GPIO_Pin_2);;   //提示系统正在运行，关灯

    }
    else if(i >= 100 && i < 200)
    {

        GPIO_ResetBits(GPIOA, GPIO_Pin_2);;//提示系统正在运行，  点灯

    }
    i = i % 200;
}
```

6.7.5 下载与测试现象

在 *神舟I号光盘\编译好的固件\07.SPI FLASH 读写.hex*，目录下的*SPI FLASH 读写.hex*文件即为前面我们分析的SPI FLASH（W25X16）读写程序实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I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I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I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。

测试现象：下载固件后，连接串口到PC机，按照要求设置波特率为115200等，按下按键1时，显示flash的写操作，写完成；按下按键2时，在串口上直接打印字符组“神舟I号 SPI 读写访问程序”，到此，我们完成了SPI FLASH（W25X16）读写程序实验。

6.8 实时时钟与年月日实验

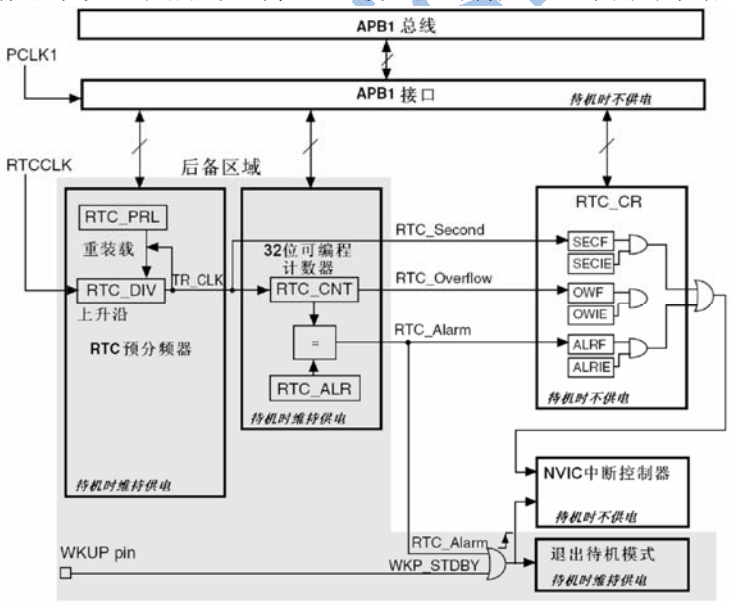
在前面学习到了几个较为基础的例程，特别是串口的输入，输出实验，本节我们在之前的基础上学习 STM32 的 RTC 原理并通过每秒显示当前实时时间的例程来将掌握 RTC 实时时钟功能及用法。

6.8.1 实验的意义与作用

RTC（Real-time clock）是实时时钟的意思。神舟 I 号开发板的处理器 STM32F103RBT6 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。

6.8.2 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。



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：

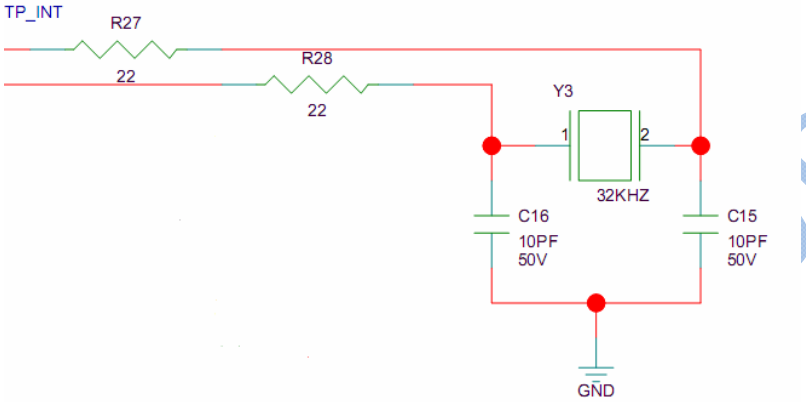
- 设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位，使能电源和后备接口时钟
- 设置寄存器PWR_CR的DBP位，使能对后备寄存器和RTC的访问。

关于RTC和BKP等详细资料，请参考《【中文】STM32F系列ARM内核32位高性能微控制器参考手册V10_1.pdf》中第47页开始的介绍

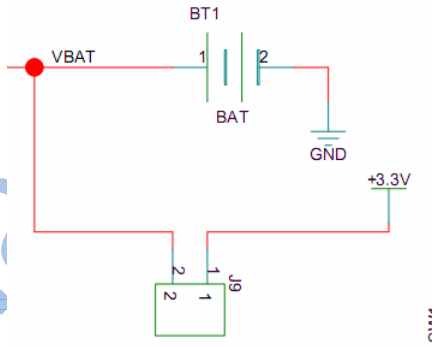
6.8.3 硬件设计

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，外围电路主需要一个32.768KH在的晶振和VBAT供电电池即可。

STM32F103RBT6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，将JP9的使用跳线帽断开。VBAT管脚由电池供电，如没有安装电池，将JP9的1，2脚使用跳线帽短接，VBAT管脚由+3.3V系统电源供电。

6.8.4 软件设计

大家先打开“实时时钟与年月日实验”的Project.uvproj工程文件，我们先从main主程序开始开始了解本实验的代码设计。其中涉及到LED灯的设置以及串口的配置请参考之前的例程，在此重点关注RTC的相关设计。

首先，我们先来看看主程序中的中断配置函数：

```
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
     */
  /* Initialize LED1 mounted on STM3210X-EVAL board */
  STM_EVAL_LEDInit(LED1);                    /*初始化评估板上的LED灯*/
  STM_EVAL_LEDInit(LED2);
  STM_EVAL_LEDInit(LED3);
  STM_EVAL_LEDOff(LED2);
  /* 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_2;
  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);
  printf("\r\n\n\n\r WWW.ARMJISHU.COM  USART2 configured....");
  STM_EVAL_LEDOn(LED3);

  InterruptConfig();        /*中断配置*/
  /* NVIC configuration */
  NVIC_Configuration();     /*RTC中断设置*/
```

其中InterruptConfig()函数主要是配置中断的起始地址为0x08000000，如下所示：

```
void InterruptConfig(void)                    /*设置中断起始地址 0x0800 0000*/
{
  /* Set the Vector Table base address at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x00000);

}
```

“InterruptConfig()”函数是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。

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 此方就是 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)       /*RTC 中断设置*/
{
  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;   RTC中断线
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

而至于RTC_IRQn中断处理函数，我们在此顺便了解一下，打开stm32f10x_it.c驱动文件，找到void RTC_IRQHandler(void)函数：

```
void RTC_IRQHandler(void)
{
  if (RTC_GetITStatus(RTC_IT_SEC) != RESET) /*判断RTC是否发生秒中断*/
  {
    /* Clear the RTC Second interrupt */
    RTC_ClearITPendingBit(RTC_IT_SEC);  /*清除中断标志*/

    /* Toggle LED1 */
    STM_EVAL_LEDToggle(LED1);   /*取反，点灯/灭灯*/

    /* Enable time update */
    TimeDisplay = 1;    /*设置全局变量为1*/

    /* Wait until last write operation on RTC registers has finished */
    RTC_WaitForLastTask();   /*等待上一次对RTC寄存器的写操作完成*/

    /* Reset RTC Counter when Time is 23:59:59 */
    if (RTC_GetCounter() == 0x00015180)
    {
      RTC_SetCounter(0x0);    /*如果时就爱你达到了23:59:59时，则下一秒时间为00:00:00*/
      /* Wait until last write operation on RTC registers has finished */
      RTC_WaitForLastTask();
    }
  }
}
```

下面我们来看看一个if…else…语句的作用：

```
if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)    /*判断RTC是否启用*/
{
  /* Backup data register value is not correct or not yet programmed (when
     the first time the program is executed) */

  printf("\r\n\n RTC not yet configured....");     /*RTC需要配置*/

  /* RTC Configuration */
  RTC_Configuration();        /*RTC配置*/

  printf("\r\n RTC configured....");

  /* Adjust time by values entred by the user on the hyperterminal */
  Time_Adjust();          /*通过超级终端显示输入调试时间值*/

  BKP_WriteBackupRegister(BKP_DR1, 0xA5A5); /*DR1写A5A5用于校准判断，BKP_DR1的值只会被侵入中断复位*/
}
else     /*RTC已经经过校准*/
{
  /* 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();      /*等待同步*/

  /* Enable the RTC Second */
  RTC_ITConfig(RTC_IT_SEC, ENABLE);

  /* Wait until last write operation on RTC registers has finished */
  RTC_WaitForLastTask();  /*等待RTC稳定*/
}
```

详解：“if…else…”语句是判断系统时间是否已经设置，判断RTC后备寄存器1的值是否为先前写入的0xA5A5，如果不是，则说明RTC是第一次上电，需要配置RTC，提示用户通过串口终端更改系统的时间，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。

而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。

下面RCC_ClearFlag（）函数是清除上次为复位的原因记录：

```
/* Clear reset flags */
RCC_ClearFlag();        /*清除标志*/

/* Display time in infinite loop */
printf("\r\n\n WWW.ARMJISHU.COM  Display time in infinite loop....\r\n");
Time_Show();    /*显示时间*/
```

而Time_Show()函数则是一个while死循环，每一秒打印一次系统时间。全局变量TimeDisplay判断是否为1，如果为1则打印时间并将请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
  */
void Time_Show(void)        /*时间显示函数*/
{
/*全局变量TimeDisplay判断是否为1，如果为1则打印时间并将请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*/
  printf("\n\r");

  /* Infinite loop */
  while (1)
  {
    /* If 1s has paased */
    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；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：

```
void Time_Display(uint32_t TimeVar)
{
  to_tm(TimeVar, &systmtime);

  printf("\r www.armjishu.com  当前时间为: %d年 %d月 %d日 (星期%s)  %0.2d:%0.2d:%0.2d", systmtime.tm_year,
  systmtime.tm_mon, systmtime.tm_mday, WEEK_STR[systmtime.tm_wday], systmtime.tm_hour, systmtime.tm_min,
   systmtime.tm_sec);

}
```

我们再回到刚才的“if…else…”语句中，大家注意到，在判断备份寄存器1值不是0xA5A5后，需要配置系统时钟，那么注意到RTC_Configuration函数的配置，如下所示：

```
void RTC_Configuration(void)
{
  /* Enable PWR and BKP clocks */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE); /*使能电源和后备接口时钟*/

  /* Allow access to BKP Domain */
  PWR_BackupAccessCmd(ENABLE);       /*开始后备区域侵入检测*/

  /* Reset Backup Domain */
  BKP_DeInit();              /*后备备份寄存器复位*/

  /* Enable LSE */
  RCC_LSEConfig(RCC_LSE_ON);    /*打开LSE晶体振荡器，以便为32.768K时钟提供精确时钟源*/
  /* Wait till LSE is ready */
  while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)  /*等待LSE晶体振荡器稳定*/
  {}

  /* Select LSE as RTC Clock Source */
  RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);   /*LSE为时钟源*/

  /* Enable RTC Clock */
  RCC_RTCCLKCmd(ENABLE);          /*使能RTC时钟*/

  /* Wait for RTC registers synchronization */
  RTC_WaitForSynchro();        /*等待RTC时钟同步*/

  /* Wait until last write operation on RTC registers has finished */
  RTC_WaitForLastTask();       /*RTC操作完成*/

  /* Enable the RTC Second */
  RTC_ITConfig(RTC_IT_SEC, ENABLE);  /*打开RTC时钟*/

  /* Wait until last write operation on RTC registers has finished */
  RTC_WaitForLastTask();       /*RTC操作完成*/

  /* Set RTC prescaler: set RTC period to 1sec */
  /*设置RTC的周期为1秒*/
  RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */

  /* Wait until last write operation on RTC registers has finished */
  RTC_WaitForLastTask();    /*RTC操作完成*/
}
```

大家注意到RTC_Configuration函数中多次出现RTC_WaitForLastTask()操作。

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。）

“if…else…”语句中显示，当RTC是第一次上电时，需要对RTC进行配置，此时将调用RTC的调整函数Time_Adjust（）：

```
void Time_Adjust(void)
{
  /* Wait until last write operation on RTC registers has finished */
  RTC_WaitForLastTask();
  /* Get time entred by the user on the hyperterminal */
  Time_Regulate(&systmtime);
  /* Get wday */
  GregorianDay(&systmtime);
  /* Change the current time */
  RTC_SetCounter(mktimev(&systmtime)); /*把时间转化为RTC计数值写入RTC寄存器中*/
  /* Wait until last write operation on RTC registers has finished */
  RTC_WaitForLastTask();
}
```

其中Time_Regulate（）函数是用来调用用户输入函数，请求用户输入时分秒信息并将其转化为计数数值。

```
void Time_Regulate(struct rtc_time *tm)
{
  uint32_t DataIn = 0xFF;
  printf("\r\n===========www.armjishu.com==============Time Settings==================");
  printf("\r\n  请输入年份(Please Set Years):  20");
  while (DataIn == 0xFF)
  {
    DataIn = USART_Scanf(99);
  }
  printf("\n\r  年份被设置为:  20%0.2d\n\r", DataIn);
  tm->tm_year = DataIn+2000;
  DataIn = 0xFF;
  printf("\r\n  请输入月份(Please Set Months):  ");
  while (DataIn == 0xFF)
  {
    DataIn = USART_Scanf(12);
  }
  printf("\n\r  月份被设置为:  %d\n\r", DataIn);
  tm->tm_mon= DataIn;
  DataIn = 0xFF;
  printf("\r\n  请输入日期(Please Set Dates):  ");
  while (DataIn == 0xFF)
  {
    DataIn = USART_Scanf(31);
  }
  printf("\n\r  日期被设置为:  %d\n\r", DataIn);
  tm->tm_mday= DataIn;
  DataIn = 0xFF;
  printf("\r\n  请输入时钟(Please Set Hours):  ");
  while (DataIn == 0xFF)
  {
    DataIn = USART_Scanf(23);
  }
  printf("\n\r  时钟被设置为:  %d\n\r", DataIn);
  tm->tm_hour= DataIn;
  DataIn = 0xFF;
  printf("\r\n  请输入分钟(Please Set Minutes):  ");
  while (DataIn == 0xFF)
  {
    DataIn = USART_Scanf(59);
  }
  printf("\n\r  分钟被设置为:  %d\n\r", DataIn);
  tm->tm_min= DataIn;
  DataIn = 0xFF;
  printf("\r\n  请输入秒钟(Please Set Seconds):  ");
  while (DataIn == 0xFF)
  {
    DataIn = USART_Scanf(59);
  }
  printf("\n\r  秒钟被设置为:  %d\n\r", DataIn);
  tm->tm_sec= DataIn;
}
```

6.8.5 下载与测试

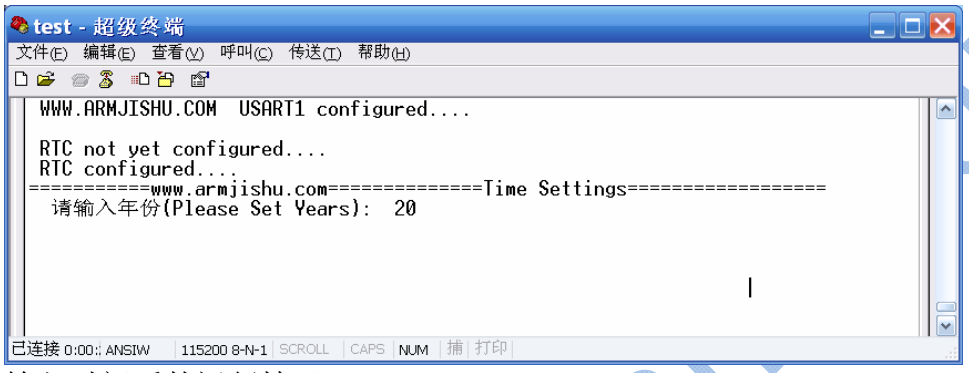
在 *神舟I号光盘\编译好的固件*目录下的“实时时钟.hex”文件即为前面我们分析的“实时时钟与年月日实验”编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I号开发板* 小节进行操作。

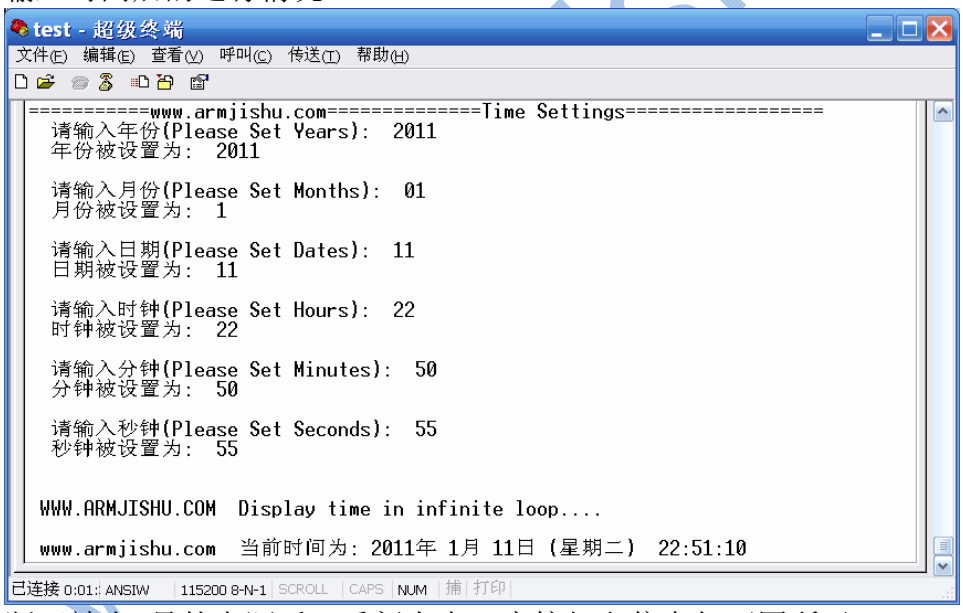
如果使用串口下载固件，请按 *如何使用串口下载一个固件到神舟I号开发板*小节进行操作。

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。

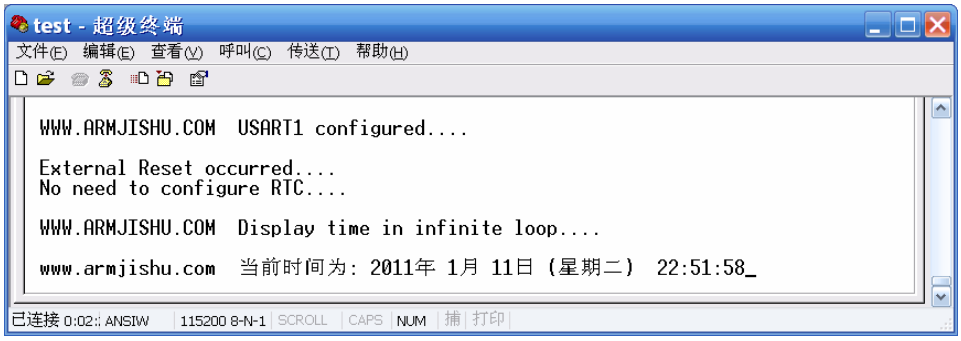
测试现象：去掉J9跳线帽，将RTC电池装上，连接神舟I号到PC上，打开串口，设置波特率为115200，运行程序，如下所示，提示设置系统时钟：



输入时间后的运行情况：



断开神舟I号的电源后，重新上电，直接打印信息如下图所示：



6.9 独立看门狗实验

本节我们将带大家学习stm32 的看门狗，STM32 内部自带两个看门狗：独立看门狗（IWDG）以及窗口看门狗（WWDG）。本节主要学习独立看门狗，而窗口看门狗只是简单的提及，如有兴趣学习的朋友，请登录www.armjishu.com留下您的宝贵意见，我们将根据您的需要上传相关资料。

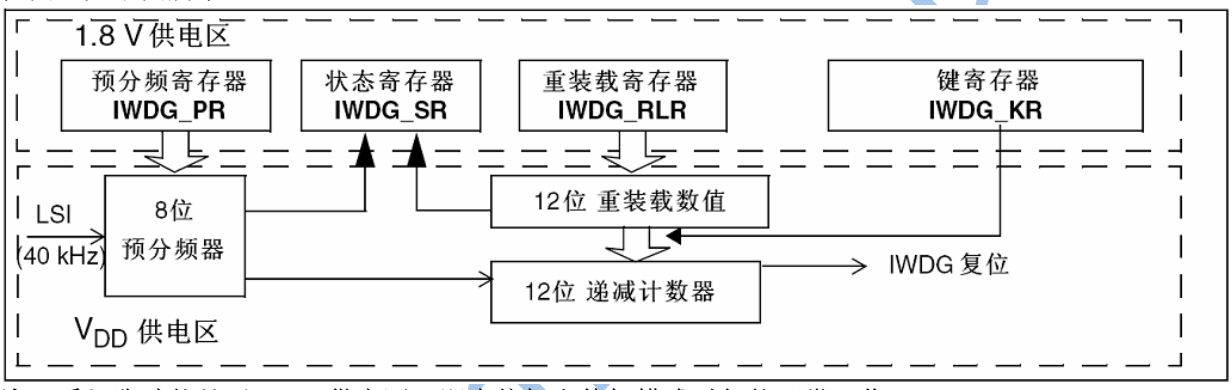
6.9.1 实验的意义与作用

随着系统功能强大，软件代码不断增多，常出现死循环或是程序跑飞，而对于运行的控制系统软件，出现死循环或是系统瘫痪时，要求能够在没有人为干预的条件下自动恢复。这种关键特性经常运用在高有效性系统或是高可靠性的系统上。而改善这类系统的可靠性的一种简单、有效的措施就是采用看门狗功能。简单来说，看门狗的运用能够让系统变得更加可靠。

6.9.2 实验原理

看门狗的工作原理：在系统运行以后，也就启动了看门狗的计数器，看门狗就开始自动计数，如果到了一定的时间还没有清看门狗（喂狗），那么看门狗计数器就会溢出，从而引起看门狗中断，造成系统复位。

下面我们了解 STM32 内部独立看门狗的相关实现原理。首先，我们先看独立看门狗模块的功能框图，如下图所示：



注：看门狗功能处于 VDD 供电区，即在停机和待机模式时仍能正常工作。

顺便了解看门狗的超时时间表，如下所示：

看门狗超时时间(40kHz的输入时钟(LSI))(1)

预分频系数	PR[2:0]位	最短时间(ms) RL[11:0] = 0x000	最长时间(ms) RL[11:0] = 0xFFF
/4	0	0.1	409.6
/8	1	0.2	819.2
/16	2	0.4	1638.4
/32	3	0.8	3276.8
/64	4	1.6	6553.6
/128	5	3.2	13107.2
/256	(6或7)	6.4	26214.4

注：这些时间是按照 40Khz时钟给出的。实际上，MCU内部的RC频率会在 30Khz到 60Khz之间变化，此外，即使RC振荡器的频率是精确的，确切的时序仍然依赖于APB接口时钟与RC振荡器时钟之间的相位差，因此总会有一个完整的RC周期是不确定的。通过对LSI进行校准可获得相对精确的看门狗超时时间。详细资料以及有关LSI校准的问题，请参考《【中文】STM32F系列ARM内核 32 位高性能微控制器参考手册V10_1.pdf》第 316 页以及 6.2.5 节。

通过独立看门狗模块的功能框图，我们来了解框图中涉及到的几个寄存器：

1、 键寄存器（IWDG_KR）

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) (Key value) 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。(见17.3.2节) 写入0xCCCC，启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。

无论什么时候，只要往键寄存器 IWDG_KR 中写入 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。

2、 预分频寄存器（IWDG_PR）

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	**PR[2:0]**: 预分频因子 (Prescaler divider) 这些位具有写保护设置，参见17.3.2节。通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子，IWDG_SR寄存器的PVU位必须为0。 000: 预分频因子=4　　100: 预分频因子=64 001: 预分频因子=8　　101: 预分频因子=128 010: 预分频因子=16　　110: 预分频因子=256 011: 预分频因子=32　　111: 预分频因子=256 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。

3、 重装载寄存器（IWDG_RLR）

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	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:12	保留，始终读为0。
位11:0	**RL[11:0]**: 看门狗计数器重装载值 (Watchdog counter reload value) 这些位具有写保护功能，参看17.3.2节。用于定义看门狗计数器的重装载值，每当向IWDG_KR寄存器写入0xAAAA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。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，参照表83。 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。 注：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RVU位为0时，读出的值才有效。

注意到，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_KP 寄存器中写入 0x5555。以不同的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。

通过以上对 stm32 内部独立看门狗的相关寄存器的了解，我们将总结操作独立看门狗的步骤：

一、在键寄存器（IWDG_KR）写入 0x5555，取消 IWDG_RP 和 IWDG_RLR 的写保护，以便对 IWDG_RP 和 IWDG_RLR 进行操作。

二、向 IWDG_KR 写入 0xAAAA，这将使得 STM32 重新装载 IWDG_RLR 的值到看门狗计数器中。

三、在键寄存器（IWDG_KR）中写入 0xCCCC，开始启动独立看门狗功能；此时计数器开始从其复位值 0xFFF 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）。

四、重复，在计数器计数到一定时间后，需要向 IWDG_KR 写入 0xAAAA 以喂狗，或是清看门狗计数。

6.9.3 硬件设计

本节的独立看门狗在硬件设计上没有具体电路，但是对于看门狗的实现功能，我们借助 LED1~3 以及按钮 1 和按钮 2 进行实验。

6.9.4 软件设计

软件的设计思路为：stm32 内部独立看门狗每隔 1s 必须喂狗一次，否则系统复位重启，此时伴随着 LED1 灯闪烁；如果在 1S 时间内通过按钮 1 或按钮 2 进行喂狗（即使按下按钮 1 或是按钮 2），系统正常运行；如果按下按钮 1 时，进行喂狗，串口打印“IWDG 重载寄存器值 1”，并点灯 LED1~3；如果按下按钮 2 时，进行喂狗，串口打印“IWDG 重载寄存器值 2”，并熄灭 LED1~3。

神舟I号独立看门狗实验位于 *神舟I 号光盘\神舟I 号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\\Project\09. 独立看门狗实验（神舟I 号）*目录。

进入 *09. 独立看门狗实验（神舟 I 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。

本例程相对较为简单，主要是针对 IWDG 独立看门狗的一些配置，而至于主程序中涉及到的 LED 以及串口的配置等说明，在先前章节已经讲解过，在此不重复。下面先来了解独立看门狗的配置：

```
//分频数为64,重载值为625,溢出时间为1s
/* IWDG timeout equal to 1s (the timeout may varies due to LSI frequency dispersion) */
/* Enable write access to IWDG_PR and IWDG_RLR registers */
//IWDG_WriteAccess_Enable:使能对寄存器IWDG_PR和IWDG_RLR的写操作
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);  //使能对寄存器IWDG_PR和IWDG_RLR的写操作

/* IWDG counter clock: 40KHz(LSI) / 64 = 625Hz */
IWDG_SetPrescaler(IWDG_Prescaler_64);  //设置IWDG预分频值:设置IWDG预分频值为64

/* Set counter reload value to 625 */
IWDG_SetReload(625);  //设置IWDG重装载值

/* Reload IWDG counter */
IWDG_ReloadCounter();  //按照IWDG重装载寄存器的值重装载IWDG计数器

/* Enable IWDG (the LSI oscillator will be enabled by hardware) */
IWDG_Enable();  //使能IWDG
```

其中，IWDG_WriteAccessCmd 函数、IWDG_SetPrescaler 函数、IWDG_SetReload 函数、IWDG_ReloadCounter 函数以及 IWDG_Enable 函数，都是引用 STM32 外部设备驱动文件 stm32f10x_iwdg.h，请查阅“stm32f10x_iwdg.h”文件即可。

接着我们往下学习，关于按钮 1 和按钮 2 的初始化声明，同时在 while 函数中对按钮的扫描，以判断是否进行喂狗。如果按钮按下，则进行喂狗。

```
GPIO_KEY_Config();      初始化配置按钮1和按钮2
Delay(7000000);         //可以看到LED1闪烁
GPIO_ResetBits(GPIOA,GPIO_Pin_2);

while(1)
{
    key=ReadKeyDown();      扫描按钮是否按下
    if(key==1)
    {
        /* Reload IWDG counter */      如果按下，重装载计数器值，也就是喂狗
        IWDG_ReloadCounter();      //按照IWDG重装载寄存器的值重装载IWDG
        printf("IWDG重载寄存器值1\n\r");
        GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);
        GPIO_ResetBits(GPIOB,GPIO_Pin_2);
    }
    else if(key==2)
    {
        /* Reload IWDG counter */
        IWDG_ReloadCounter();      //按照IWDG重装载寄存器的值重装载IWDG
        printf("IWDG重载寄存器值2\n\r");
        GPIO_SetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);
        GPIO_SetBits(GPIOB,GPIO_Pin_2);
    }
}
```

其中，IWDG_ReloadCounter 重装载函数也是引用“stm32f10x_iwdg.h”驱动的，而 GPIO_KEY_Config 函数，主要是实现对按钮的配置：

```
void GPIO_KEY_Config(void)                    //按钮配置函数
{
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Configure KEY1 Button */
  RCC_APB2PeriphClockCmd(RCC_KEY1, ENABLE);

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;     //按钮1配置
  GPIO_InitStructure.GPIO_Pin = GPIO_KEY1;
  GPIO_Init(GPIO_KEY1_PORT, &GPIO_InitStructure);

  /* Configure KEY2 Button */
  RCC_APB2PeriphClockCmd(RCC_KEY2, ENABLE);          //按钮2配置

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Pin = GPIO_KEY2;
  GPIO_Init(GPIO_KEY2_PORT, &GPIO_InitStructure);
}
```

而 ReadKeyDown（）扫描按钮是否被按下的函数，具体实现如下所示：

```
u8 ReadKeyDown(void)                              //按钮扫描
{
  /* 1 key is pressed */
  if(!GPIO_ReadInputDataBit(GPIO_KEY1_PORT, GPIO_KEY1))
  {
    while(!GPIO_ReadInputDataBit(GPIO_KEY1_PORT, GPIO_KEY1))        //按钮1按下
    {
        ;//reserved
    }
    return KEY1;
  }
  /* 2 key is pressed */
  if(!GPIO_ReadInputDataBit(GPIO_KEY2_PORT, GPIO_KEY2))         //按钮2按下
  {
    while(!GPIO_ReadInputDataBit(GPIO_KEY2_PORT, GPIO_KEY2))
    {
        ;//reserved
    }
    return KEY2;
  }
  /* No key is pressed */
  else
  {
    return NOKEY;
  }
}
```

6.9.5 下载与测试

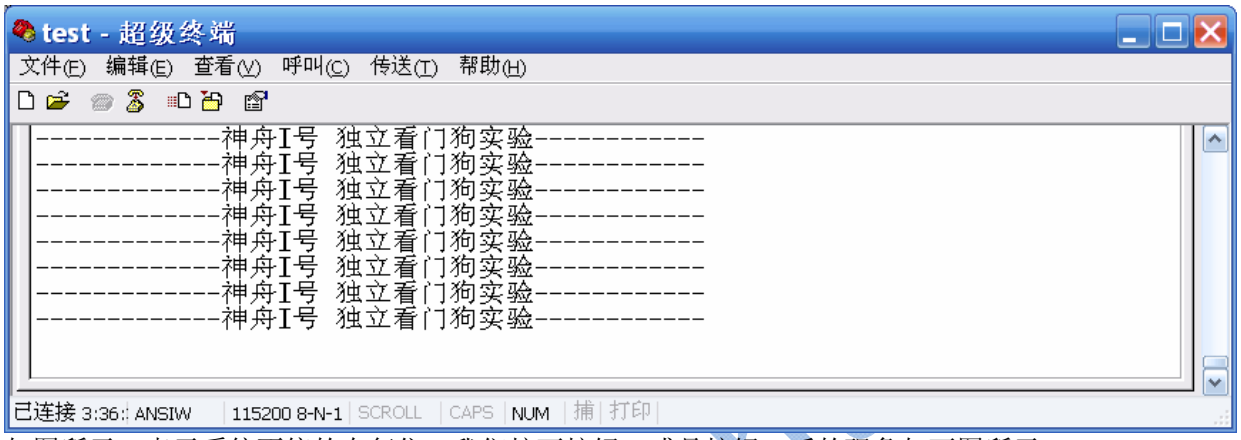
在 *神舟I号光盘\编译好的固件*目录下的*09.独立看门狗.hex*文件即为前面我们分析的“独立看门狗实验”编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I号开发板* 小节进行操作。

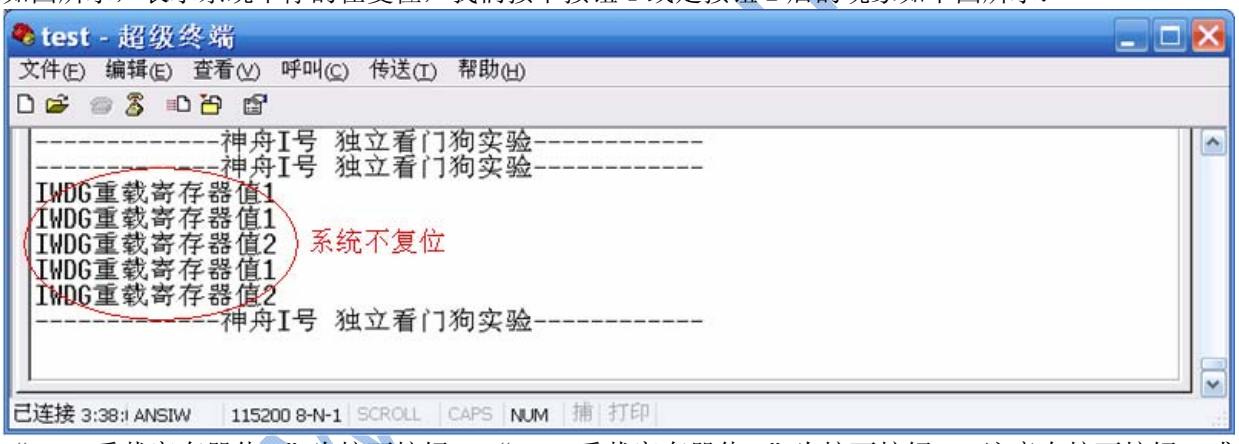
如果使用串口下载固件，请按 *如何使用串口下载一个固件到神舟I号开发板*小节进行操作。

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。

测试现象：下载固件到神舟I号开发板上，连接串口到PC上，配置波特率为115200，运行程序，串口打印如下，且LED1不停的闪烁，频率为1s：



如图所示，表示系统不停的在复位，我们按下按钮 1 或是按钮 2 后的现象如下图所示：



“IWDG 重载寄存器值 1”为按下按钮 1；“IWDG 重载寄存器值 2”为按下按钮 2，注意在按下按钮 1 或按钮 2 时，LED1~LED3 的变化。按钮 1 时，LED1~LED3 为亮；按钮 2 时，LED1~LED3 为熄灭。

6.10 SysTick实验

6.10.1 硬件设计

SysTick这部分不需要硬件接口，这里仅在中断产生时，进行点灯操作。

www.armjishu.com

6.10.2 软件设计

神舟I号SysTick实验位于*神舟I号光盘\神舟I号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\\Project\01. SysTick实验（神舟I号）*目录。

进入*01. SysTick实验（神舟I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。

本实例中还进行了点灯操作，至于使用GPIO管脚点灯的操作，在例程1和例程2中已经详细介绍，本历程中主要是针对SysTick示例中新增添的代码进行说明。

初始化配置SysTick函数以及中断优选级

```
void SysTick_Configuration(void)
{
  /* Setup SysTick Timer for 10 msec interrupts   */
  if (SysTick_Config(SystemCoreClock / 100))        /*SysTick配置函数*/
  {
    /* Capture error */
    while (1);
  }

 /* Configure the SysTick handler priority */
  NVIC_SetPriority(SysTick_IRQn, 0x0);              /*设置SysTick的中断优选级*/
}
```

SysTick配置函数具体如下所示

```
uint32_t SystemCoreClock          = SYSCLK_FREQ_72MHz;        /*!< System Clock Frequency (Core Clock) */

static __INLINE uint32_t SysTick_Config(uint32_t ticks)        初始化SysTick定时器相关的四个寄存器值，详见上面描述
{
  if (ticks > SysTick_LOAD_RELOAD_Msk)  return (1);            /* Reload value impossible */

  SysTick->LOAD  = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;      /* set reload register */
  NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);  /* set Priority for Cortex-M0 System Interrupts */
  SysTick->VAL   = 0;                                          /* Load the SysTick Counter Value */
  SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_TICKINT_Msk   |
                   SysTick_CTRL_ENABLE_Msk;                    /* Enable SysTick IRQ and SysTick Timer */
  return (0);                                                  /* Function successful */
}
```

设置中断向量起始地址

```
void InterruptConfig(void)
{
  /* Set the Vector Table base address at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x00000);          /*设置中断向量起始地址*/

}
```

每一个SysTick中断产生，都调用了LED灯闪烁函数，进行点灯，关灯操作。

```
void LED_Spark(void)                          //LED闪烁函数
{
  static __IO uint32_t TimingDelayLocal = 0;  //初始化静态变量

  if (TimingDelayLocal != 0x00)
  {
    if(TimingDelayLocal < 50)  // 后50次熄灭LED指示灯
    {
      Led_Turn_off_all();
    }
    else                  // 前50次点亮LED指示灯
    {
      Led_Turn_on_all();
    }
    TimingDelayLocal--; // 该函数每一次被调用静态本地变量TimingDelayLocal便会减一
  }
  else
  {
    TimingDelayLocal = 100;     //赋值
  }
}
```

SysTick产生中断处理

```
extern void LED_Spark(void);                                  /*声明*/

 void SysTick_Handler(void)          /*SysTick中断处理*/
 {
   LED_Spark();
 }
```

主函数中的一些初始化操作

```
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);

  InterruptConfig();                  /*设置中断向量起始地址*/
  SysTick_Configuration();            /*SysTick配置*/

  /* Main loop */
  while (1)
  {
      Delay_ARMJISHU(1000000);
  }
}
```

6.10.3 下载与测试

在 *神舟I号光盘\编译好的固件*目录下的*SysTick实验*.hex文件即为前面我们分析的SysTick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I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I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I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。

下载成功后，运行程序，从神舟I号的LED1~LED3可以看来每一秒，LED灯闪烁一次。其中LED灯亮500ms，灭500ms。（50x10ms时基得到的）

6.11 TFT彩屏显示实验

学习了前面较为基础的实验后，本节我们来了解TFT彩屏的显示实验。主要是了解如何采用各种不同的颜色显示不同的界面，同时可以将预先设置的字符用不同的颜色进行显示。业界上的2.8寸和3.2寸的彩屏手机，也都是采用相类似的TFT实现的。下面我们简单了解TFT彩屏的显示过程。

6.11.1 实验的意义与作用

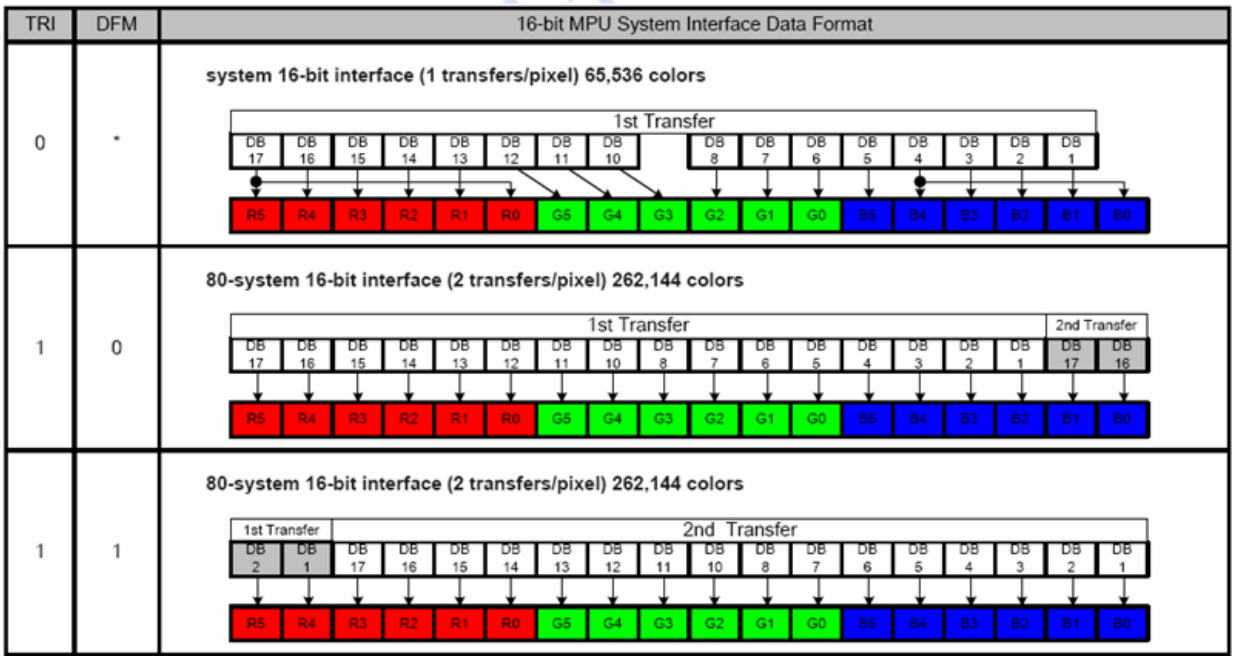
TFT屏在信息行业快速发展中得到广泛的运用，中高端的手机、可视电话、便携式的VCD、平板电脑等等；通过本节的实验，让大家了解TFT彩屏的显示原理，同时在本节的基础上，我们在下一节将学习TFT触摸屏的显示与触摸实验。

6.11.2 实验原理

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。

对于我们神舟I号开发板上配带的TFT LCD屏，采用的ILI9320控制器，LCD屏为320x240分辨率，16位真彩显示。我们下面就ILI9320控制器进行简单的介绍。

ILI9320控制器是一款带有262144中颜色的单芯片SoC驱动的晶体管显示器，320x240的分辨率，包括720路源极驱动以及320路的栅极驱动，自带有显存，容量为172800字节。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。对于数据越大时，表示的颜色越深。

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	读型号
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	EOR	0	0	0	0	0	0	0	0	
03h	Entry Mode	W	1	TRI	DFM	0	BGR	0	DACKE	HWM	0	0	0	I/D1	I/D0	AM	0	0	0	入口控制
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	显示控制
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	FMI2	FMI1	FMI0	
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	VSPL	HSPL	0	DPL	EPL	
20h	Horizontal GRAM Address Set	W	1	0	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	AD8	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	0	EXT_R	0	FR_SEL1	FR_SEL0	0	0	0	0	
50h	Horizontal Address Start Position	W	1	0	0	0	0	0	0	0	0	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0	行起始地址
51h	Horizontal Address End Position	W	1	0	0	0	0	0	0	0	0	HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0	行结束地址
52h	Vertical Address Start Position	W	1	0	0	0	0	0	0	0	VSA8	VSA7	VSA6	VSA5	VSA4	VSA3	VSA2	VSA1	VSA0	列起始地址
53h	Vertical Address End Position	W	1	0	0	0	0	0	0	0	VEA8	VEA7	VEA6	VEA5	VEA4	VEA3	VEA2	VEA1	VEA0	列结束地址

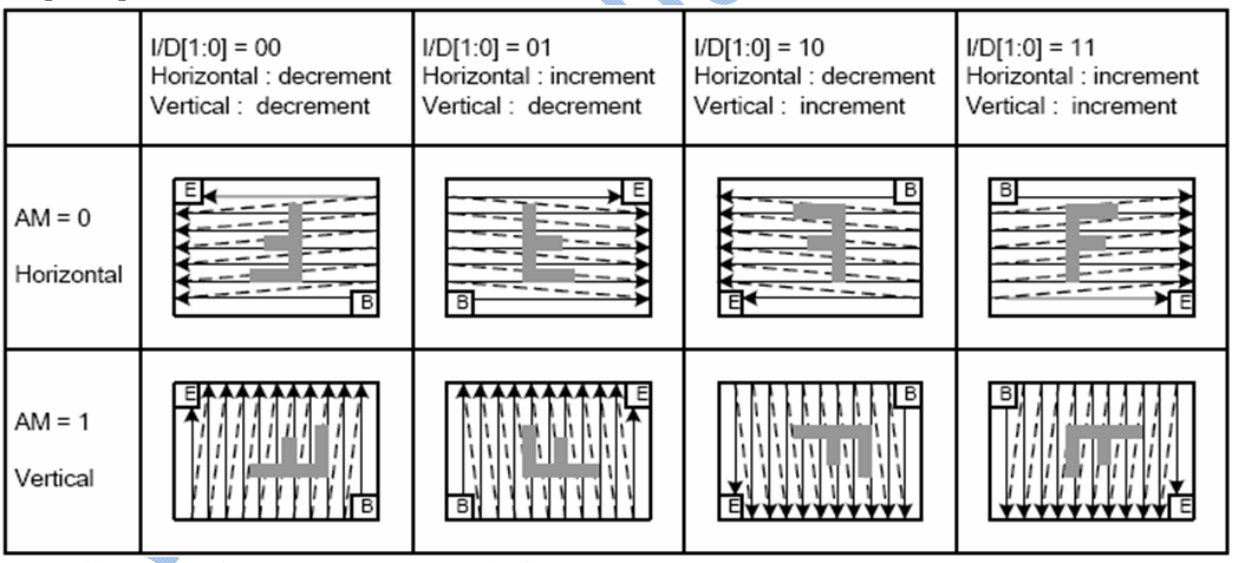
首先，我们看看指令00h，当为读操作时，读取控制器的型号；当为写操作时，打开/关闭OSC振荡器。我们在代码设计中就是通过指令00h读取控制器的型号，从而针对具体型号的控制器进行初始化操作，以便于兼容ILI93XX系列的控制器。

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。

第二个指令为03h，进入模式命令。在这个命令中，我们关注AM、I/D1、I/D0三个比特位。

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：



第三个指令为07h，显示控制命令。

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

接着我们了解第四个和第五个指令为20h和21h，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。

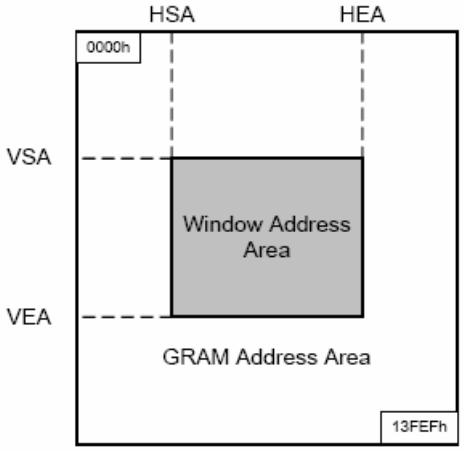
第六个指令为22h，（读/写）数据（到/从）GRAM。当这个指令执行时，地址计数器自动增加和减少。

下面再看看50h ~ 53h指令，垂直和水平RAM地址位置设置。

	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	0	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0
R51h	W	1	0	0	0	0	0	0	0	0	HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0
R52h	W	1	0	0	0	0	0	0	0	VSA8	VSA7	VSA6	VSA5	VSA4	VSA3	VSA2	VSA1	VSA0
R53h	W	1	0	0	0	0	0	0	0	VEA8	VEA7	VEA6	VEA5	VEA4	VEA3	VEA2	VEA1	VEA0

HAS[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

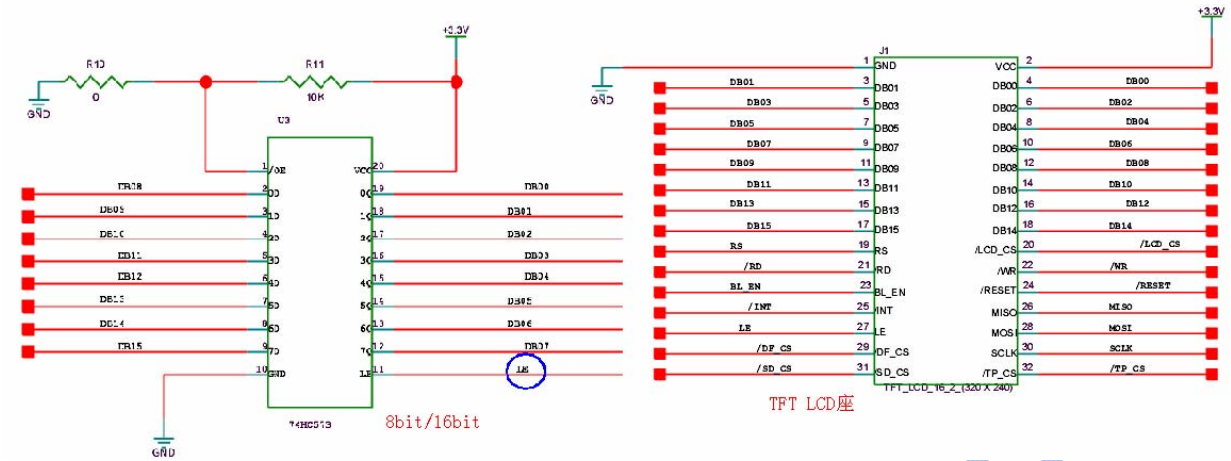
“00”h ≤VSA[7:0] ≤VEA[7:0] ≤”13F”h

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。

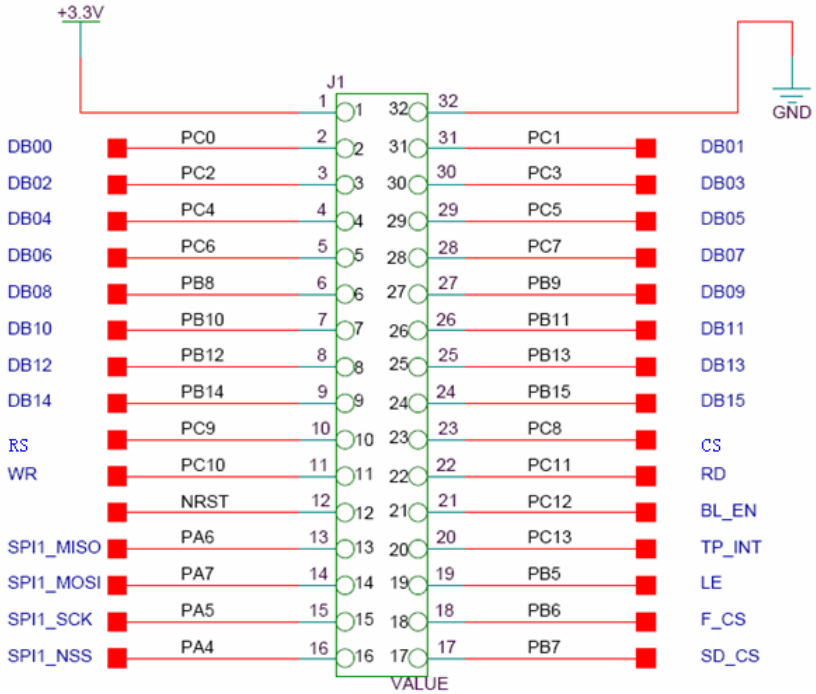
6.11.3 硬件设计

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I号开发板上的TFT座的信号连接情况。

首先是TFT的硬件设计：



其次，神舟I号开发板上的TFT座的信号连接设计：



其中，值得一提的是TFT LCD的控制信号，控制信号与神舟I号控制器的连接关系，如下所示：

CS	<——>	PC8	LCD片选信号
RS	<——>	PC9	命令/数据标志（1，读写数据；0，读写命令）
WR	<——>	PC10	向LCD写入数据
RD	<——>	PC11	从LCD读取数据
BL_EN	<——>	PC12	背光控制

6.11.4 软件设计

打开*神舟 I 号光盘\神舟 I 号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\ 11.TFT 彩屏显示实验（神舟 I 号）*目录。进入 *11.TFT 彩屏显示实验（神舟 I 号）\ MDK-ARM* 目录后，双击 Project.uvproj 打开工程，以下我们学习代码的设计。

一、我们先看看 TFT LCD 屏相关的数据总线以及控制信号线的定义：

LCD 屏控制信号的宏定义：

```
/* LCD Control pins */
#define LCD_Pin_BL      GPIO_Pin_12
#define LCD_PORT_BL     GPIOC
#define LCD_CLK_BL      RCC_APB2Periph_GPIOC

#define LCD_Pin_WR      GPIO_Pin_10
#define LCD_PORT_WR     GPIOC
#define LCD_CLK_WR      RCC_APB2Periph_GPIOC

#define LCD_Pin_CS      GPIO_Pin_9
#define LCD_PORT_CS     GPIOC
#define LCD_CLK_CS      RCC_APB2Periph_GPIOC

#define LCD_Pin_RS      GPIO_Pin_8
#define LCD_PORT_RS     GPIOC
#define LCD_CLK_RS      RCC_APB2Periph_GPIOC

#define LCD_Pin_RD      GPIO_Pin_11
#define LCD_PORT_RD     GPIOC
#define LCD_CLK_RD      RCC_APB2Periph_GPIOC
```

实际使用中的命令定义，同时包括数据总线作为输入和输出时的定义：

```
#define Lcd_Light_ON  GPIO_SetBits(LCD_PORT_BL, LCD_Pin_BL);
#define Lcd_Light_OFF GPIO_ResetBits(LCD_PORT_BL, LCD_Pin_BL);

#define SetCs  GPIO_SetBits(LCD_PORT_CS, LCD_Pin_CS);
#define ClrCs  GPIO_ResetBits(LCD_PORT_CS, LCD_Pin_CS);

#define SetWr  GPIO_SetBits(LCD_PORT_WR, LCD_Pin_WR);
#define ClrWr  GPIO_ResetBits(LCD_PORT_WR, LCD_Pin_WR);

#define SetRs  GPIO_SetBits(LCD_PORT_RS, LCD_Pin_RS);
#define ClrRs  GPIO_ResetBits(LCD_PORT_RS, LCD_Pin_RS);

#define SetRd  GPIO_SetBits(LCD_PORT_RD, LCD_Pin_RD);
#define ClrRd  GPIO_ResetBits(LCD_PORT_RD, LCD_Pin_RD);

#define LCD_Write(LCD_DATA)  {\
                              GPIO_Write(GPIOC, ((GPIOC->ODR&0XFF00)|(LCD_DATA&0x00FF)));\
                              GPIO_Write(GPIOB, ((GPIOB->ODR&0X00FF)|(LCD_DATA&0xFF00)));\
                             }     //数据输出

#define LCD_Read()   (GPIO_ReadInputData(GPIOB)&0XFF00)|(GPIO_ReadInputData(GPIOC)&0X00FF)      //数据输入
```

二、接着，我们再来关注ILI9320控制器的代码设计，对于ILI9320控制器，需要对其工作模式进行配置：

1、 屏幕的旋转定义，`#define ID_AM  110`

2、 几个常用指令寄存器的宏定义：

```
/* LCD Registers */
#define R0              0x00
#define R3              0x03
#define R7              0x07
#define R32             0x20
#define R33             0x21
#define R34             0x22
#define R50             0x32
#define R51             0x33
#define R52             0x34
#define R53             0x35
```

3、 画笔颜色的宏定义：

```
/* LCD color */ //画笔颜色
#define White           0xFFFF
#define Black           0x0000
#define Grey            0xF7DE
#define Blue            0x001F
#define Blue2           0x051F
#define Red             0xF800
#define Magenta         0xF81F
#define Green           0x07E0
#define Cyan            0x7FFF
#define Yellow          0xFFE0
```

4、 ILI9320控制器的初始化：

```
void ili9320_Initializtion(void)
{
  u16 i;
  u8 str[] = "www.armjishu.com";
  u8 len = sizeof(str)-1;
  u16 StartX;

  LCD_Init();
  Delay(5); /* delay 50 ms */
  LCD_WriteReg(0x0000,0x0001);          //start internal osc
  Delay(5); /* delay 50 ms */
  LCD_DB_AS_InPut();
  Delay(1); /* delay 50 ms */
  DeviceIdCode = LCD_ReadReg(0x0000);
```

其中，涉及到LCD_Init（）的初始化函数，以及开启内部OSC振荡器，同时使用00h寄存器读取LCD型号。如下解析：

打开OSC命令，`LCD_WriteReg(0x0000,0x0001);  //start internal osc`

读取LCD屏型号命令，`DeviceIdCode = LCD_ReadReg(0x0000);`

LCD_Init（）函数的展开设计，主要是对于LCD屏连接的数据总线以及控制信号的使能配置，

```
void LCD_Init(void)                    //初始化，请详见上面定义
{
 /* Configure the LCD Control pins ----------------------------------------*/
  LCD_Pins_Config();
}
void LCD_Pins_Config(void)        //LCD屏的数据总线以及控制管脚配置
{
  GPIO_InitTypeDef GPIO_InitStructure;
  RCC_APB2PeriphClockCmd(LCD_CLK_RS | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC|
                         LCD_CLK_WR | LCD_CLK_RD | LCD_CLK_CS | LCD_CLK_BL, ENABLE);
  // PC0~7  <----> DB0~7
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPI
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;       //默认配置为输出
  GPIO_Init(GPIOC, &GPIO_InitStructure);
  //PB8~15  <----> DB8~15
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_1
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;       //默认配置为输出
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  /*************************LCD控制信号*******************************/
  //LCD_Pin_BL
  GPIO_InitStructure.GPIO_Pin = LCD_Pin_BL;        //背光控制
  GPIO_Init(LCD_PORT_BL, &GPIO_InitStructure);
  //LCD_Pin_WR
  GPIO_InitStructure.GPIO_Pin = LCD_Pin_WR;        //写信号控制
  GPIO_Init(LCD_PORT_WR, &GPIO_InitStructure);

  //LCD_Pin_CS
  GPIO_InitStructure.GPIO_Pin = LCD_Pin_CS;        //片选控制
  GPIO_Init(LCD_PORT_CS, &GPIO_InitStructure);

  //LCD_Pin_RS
  GPIO_InitStructure.GPIO_Pin = LCD_Pin_RS;
  GPIO_Init(LCD_PORT_RS, &GPIO_InitStructure);

  //LCD_Pin_RD
  GPIO_InitStructure.GPIO_Pin = LCD_Pin_RD;       //读信号
  GPIO_Init(LCD_PORT_RD, &GPIO_InitStructure);

  SetCs      //不使能片选
}
```

设置窗口区域的函数，通过50h~53h命令，规划区域大小：

```
void ili9320_SetWindows(u16 StartX,u16 StartY,u16 EndX,u16 EndY)  //设置窗口区域
{
  ili9320_SetCursor(StartX,StartY);
  LCD_WriteReg(0x0050, StartX);
  LCD_WriteReg(0x0052, StartY);
  LCD_WriteReg(0x0051, EndX);
  LCD_WriteReg(0x0053, EndY);
}
```

设置指定点的显示ASCII码字符的函数：

```
void ili9320_PutChar(u16 x,u16 y,u8 c,u16 charColor,u16 bkColor)  //
{
  u16 i=0;
  u16 j=0;
  u8 tmp_char=0
  if(HyalineBackColor == bkColor)
  {
    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
          {
            // do nothing // 透明背景
          }
      }
    }
  }
  else
  {
        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
          }
      }
    }
  }
}
```

详细的关于ILI9320的驱动函数，请参考源码中的设计以及注释，下面我们简单看看主函数main一些设计。

主函数中显示字符组的定义，

```
u8 c[] = "www.armjishu.com";            //预先设定的显示字符c
u8 c2[] = "TFT LCD 320X240";            //预先设定的显示字符c2
u8 *str;
u16 charColor;
u16 bkColor;

len = sizeof(c)-1;                      //取字符组长度
c2len = sizeof(c2)-1;
```

主函数中，循环显示的设计源码如下：

```
while (1)
{
  ili9320_Clear(Red);                                          //红屏
  ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);  //屏的20处，显示字符组c
  ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);  //屏的200处，显示字符组c2
  Delay_ARMJISHU(4000000);

  ili9320_Clear(Green);                                        //绿屏
  ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);  //屏的20处，显示字符组c
  ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);  //屏的200处，显示字符组c2
  Delay_ARMJISHU(4000000);

  ili9320_Clear(Blue);                                         //蓝屏
  ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);
  ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);
  Delay_ARMJISHU(4000000);

  ili9320_Clear(Yellow);                                       //黄屏
  ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);
  ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);
  Delay_ARMJISHU(4000000);

  ili9320_Clear(Magenta);                                      //灰屏
  ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);
  ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);
  Delay_ARMJISHU(4000000);

  ili9320_Clear(Cyan);
  ili9320_PutStr_16x24_Center(20, c, len,charColor, bkColor);
  ili9320_PutStr_16x24_Center(200, c2, c2len,charColor, bkColor);
  Delay_ARMJISHU(4000000);


  ili9320_ColorScreen();
  ili9320_PutStr_16x24_Center(108, c, len,White, HyalineBackColor);   //屏的中间处，显示字符组c
  Delay_ARMJISHU(4000000);
  Delay_ARMJISHU(4000000);
  Delay_ARMJISHU(4000000);
  Delay_ARMJISHU(4000000);

  ili9320_GreyScreen();
  ili9320_PutStr_16x24_Center(108, c, len,White, HyalineBackColor);  //屏的中间处，显示字符组c
  Delay_ARMJISHU(4000000);
  Delay_ARMJISHU(8000000);
  Delay_ARMJISHU(4000000);
  Delay_ARMJISHU(4000000);
  Delay_ARMJISHU(4000000);
```

对于TFT LCD显示的源码简单介绍到此，有兴趣的朋友可以继续深入研究其它函数的意义以及作用，以方便下节TFT 触摸屏显示加触摸实验的学习，如有不懂之处，登录www.armjishu.com论坛提出您的宝贵意见，我们一起探讨。

www.

6.11.5 下载与现象

在 *神舟I号光盘\神舟I号各例程下载固件*目录下的*11.彩屏显示.hex*文件即为前面我们分析的TFT彩屏显示实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I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I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I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。

现象：将固件下载到开发板后，我们观看神舟I号的LCD屏显示现象。如下所示：

6.12 TFT触摸屏显示加触摸实验

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。

6.12.1 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。

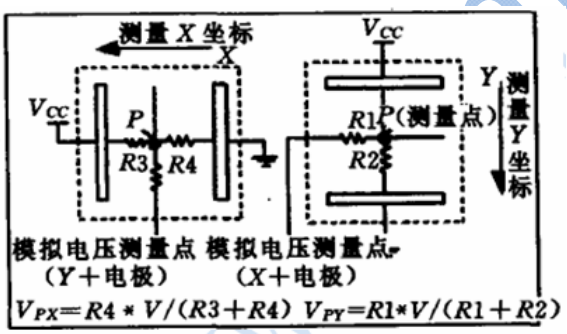
6.12.2 实验原理

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。

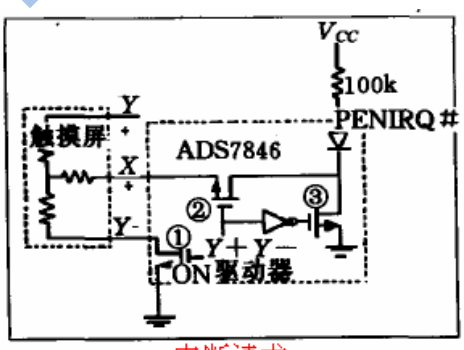
神舟I号的触摸屏附在LCD屏的表面上，与LCD屏相配合使用，主要是用的触摸芯片是ADS7843，业界上与ADS7843芯片相兼容的触摸芯片还有ADS7846、AK4182、XPT2046以及TSC2046等，驱动基本上一致。

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）



测量关系

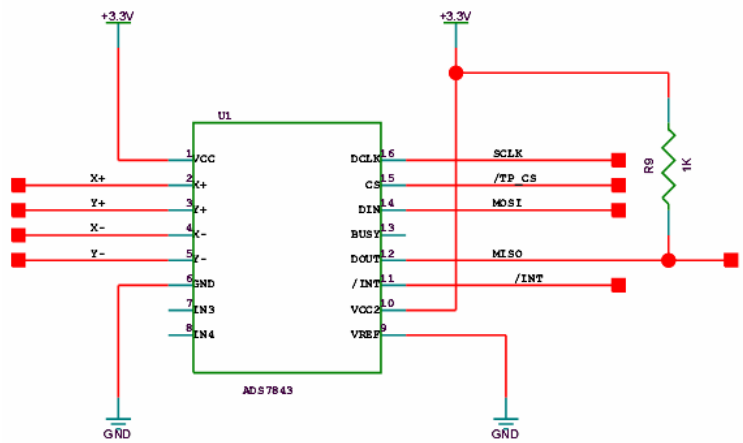


中断请求

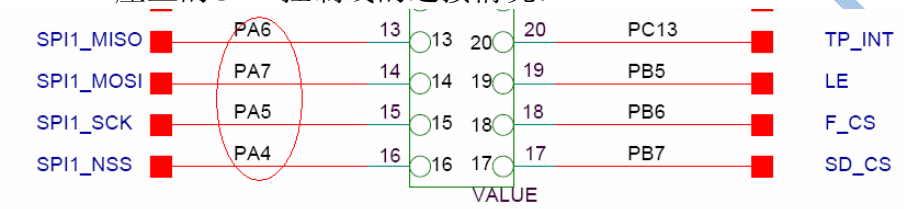
当触摸屏受到点击或是挤压的时，ADS7843通过中断请求通知处理器（STM32F103RBT6）有触摸发生。如“中断请求”图所示，当没有触摸时，MOSFET①和②打开、③关闭，则中断输出引脚通过外加的上拉电阻输出为高，当有触摸时，①和③打开，②关闭，则中断输出引脚通过③内部的连接到地，输出为低，从而向处理器发出中断请求。

6.12.3 硬件设计

ADS7843触摸芯片现位于TFT LCD屏上，而通过处理器的SPI1总线由TFT座控制触摸芯片。X+、X-、Y+和Y-则连接到LCD触摸屏上。



TFT座上的SPI1控制线的连接情况：



6.12.4 软件设计

打开神舟 I 号光盘\神舟 I 号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\ 12.TFT 触摸屏显示加触摸实验（神舟 I 号）目录。进入 12.TFT 触摸屏显示加触摸实验（神舟 I 号）\ MDK-ARM 目录后，双击 Project.uvproj 打开工程，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，使用 SPI1 对 ADS7843 芯片进行控制时，SPI1 的初始化控制，在前面也提及过。

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;     //PC13管脚
  //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_InterruptConfig(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：

```
/* ARMJISHU_TouchScreen_ADS7843 */

#define RCC_ADS7843_CS                        RCC_APB2Periph_GPIOA
#define GPIO_ADS7843_CS_PORT                  GPIOA
#define GPIO_ADS7843_CS                       GPIO_Pin_4

#define RCC_ADS7843_INT                       RCC_APB2Periph_GPIOC
#define GPIO_ADS7843_INT_PORT                 GPIOC
#define GPIO_ADS7843_INT                      GPIO_Pin_13
#define GPIO_ADS7843_EXTI_LINE                EXTI_Line13
#define GPIO_ADS7843_EXTI_PORT_SOURCE         GPIO_PortSourceGPIOC
#define GPIO_ADS7843_EXTI_PIN_SOURCE          GPIO_PinSource13

#define GPIO_ADS7843_EXTI_IRQn                EXTI15_10_IRQn

#define GPIO_ADS7843_INT_VALID    !GPIO_ReadInputDataBit(GPIO_ADS7843_INT_PORT, GPIO_ADS7843_INT)
```

接下来，我们看看中断的处理函数，打开stm32f10x_it.c文件，找到“EXTI15_10_IRQHandler”函数：

```
void EXTI15_10_IRQHandler(void) /* TouchScreen */
{
  if(EXTI_GetITStatus(EXTI_Line13) != RESET)
  {
    ARMJISHU_TouchScreen_ADS7843();
    printf("* ");
    /* Clear the EXTI Line 13 */
    EXTI_ClearITPendingBit(EXTI_Line13);
  }
}
```

处理器接收到PC13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。

www.armjishu.com

```
#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：

```
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);

    GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);
    GPIO_ResetBits(GPIOB,GPIO_Pin_2);/*点亮所有的LED指示灯*/

    ili9320_ColorScreen();
    ili9320_PutStr_16x24_Center(108, c, len,White, HyalineBackColor);
    Delay_ARMJISHU(10000000);
    GPIO_SetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);        /*关闭所有的LED指示灯*/
    GPIO_SetBits(GPIOB,GPIO_Pin_2);

    ili9320_GreyScreen();
    ili9320_PutStr_16x24_Center(108, c, len,White, HyalineBackColor);
    GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);
    GPIO_ResetBits(GPIOB,GPIO_Pin_2);/*点亮所有的LED指示灯*/
    Delay_ARMJISHU(2000000);
    GPIO_SetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3);        /*关闭所有的LED指示灯*/
    GPIO_SetBits(GPIOB,GPIO_Pin_2);
}
```

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。

www.armjishu.com

6.12.5 下载与现象

在 *神舟I号光盘\神舟I号各例程下载固件*目录下的*12.TFT触摸屏显示加触摸.hex*文件即为前面我们分析的TFT触摸屏显示加触摸实验编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I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I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I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。

现象：将固件下载到神舟I号后，复位，LCD屏就可以写字，我们写上“神舟I号”字样，如下图所示：

6.13 DS18B20温度传感器实验

本节将通过DS18B20温度传感器，测试环境温度的例程，给大家讲解温度传感器的工作原理，同时引发大家对传感器在实际运用的一些拓展认识，如仓库管理，可通过传感器测试仓库的温度，湿度；智能系统的传感器采样等等。

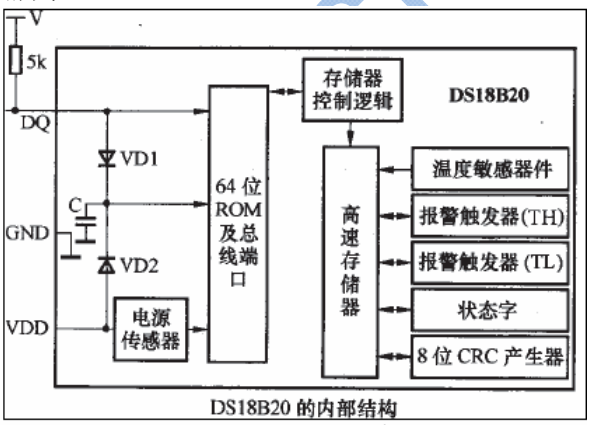
6.13.1 实验的意义与作用

现实生活中，对于一些比较恶劣的环境运用中，在人员无法深入具体环境下，通常都是采用传感器进行探测，再根据实际情况做出具体的措施。通过本实验，让大家了解温度传感器的工作原理；认识单总线的原理；以及温度传感器带实验例程中的驱动代码设计。

6.13.2 试验原理

本实验采用的温度传感器的型号为DS18B20，它是单总线数字式智能型传感器，直接将温度物理量转化为数字信号并以总线方法传送到处理器（stm32f103rbt6）进行数据处理。DS18B20数字式智能型温度传感器对于失策的温度提供了9~12位的数据和报警温度寄存器，测试温度范围为-55℃ ～ +125℃，其中在-10℃ ～ +85℃ 的范围内的测量精度为±0.5℃。

DS18B20内部结构主要由4部分组成：64为ROM、温度传感器、非挥发的温度报警触发器TH和TL、配置寄存器。每一个DS18B20包括一个唯一的64位长的序号，该序号值存放在DS18B20内部的ROM（只读存储器）中。DS18B20的总线仅由一根线组成，与总线相连的器件应具有漏极开路或三态输出，以保证有足够负载能力驱动该总线。DS18B20的DQ端是开漏输出的，单总线要求加一只5K左右的上拉电阻。其内部电路图，如下图所示：



DS18B20 的内部结构

DS18B20的单总线命令序列如下步骤所示：

第一步：初始化；

第二步：ROM操作命令（跟随需要交换的数据）；

第三步：功能命令（跟随需要交换的数据）

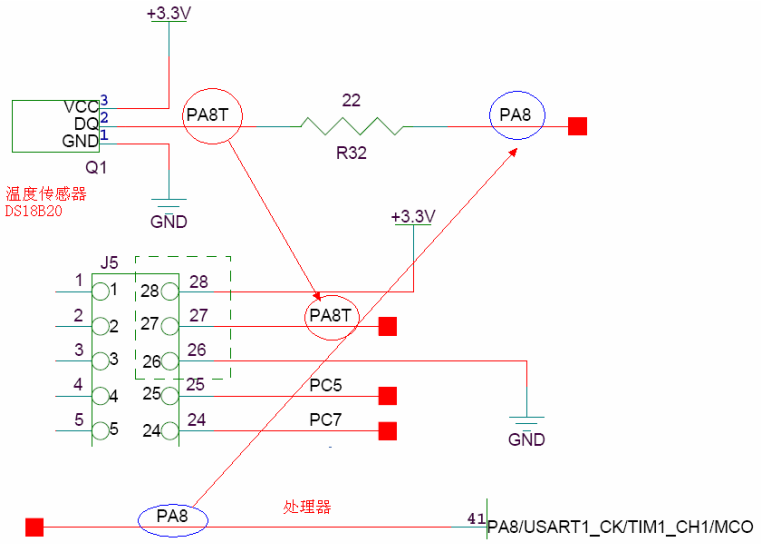
每次访问单总线期器件，必须严格遵守这个命令序列。如果出现序列混乱，则总线器件不会响应主机（除搜索ROM命令和报警搜索命令外）。

实际中对于读取DS18B20温度传感器的操作步骤应该如下所示：

复位—>skip ROM（0xCC）—>开始转换（0x44）—>延迟—>复位—>skip ROM（0xCC）—>读取存储器（0xBE）—>连续读出两个字节数据（TH和TL），温度值—>结束。

6.13.3 硬件设计

硬件上的设计，对于DS18B20温度传感器来说，相对较为简单。处理器上我们选择PA8与DS18B20的DQ端相连接，而至于DS18B20电源和GND，直接连接到神舟I号的电源和地上。在此需要提出的是，DS18B20的接口现设置为扩展座上，方便用户的插拔使用。对于DS18B20与处理器STM32F103RBT6间的连接原理如下所示：



6.13.4 软件设计

打开神舟I号光盘\神舟I号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\ 13.18B20温度传感实验（神舟I号）目录。进入13.18B20温度传感实验（神舟I号）\ MDK-ARM 目录后，双击Project.uvproj打开工程，以下我们学习代码的设计，主要是与DS18B20的相关的驱动代码。

本实验，还是借助串口打印，将DS18B20测试到的环境温度值通过串口打印出来，至于串口的声明前面已经提到，在此不涉及到。

先对DS18B20进行初始化，同时也包括检测DS18B20是否在位的功能，以此可以判断DS18B20的连接情况，如果DS18B20不在位，我们则打印“DS18B20 不在位，请检查连接！”

```
while(DS18B20_Init())//初始化DS18B20,兼检测DS18B20
{
printf("\n\r DS18B20 不在位，请检查连接\n ");
    delay_ms(500);
}
```

而对于DS18B20的初始化函数，如下所示：

```
u8 DS18B20_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;                    //GPIO
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA, ENABLE );  //使能PORTA口时钟

    /* Configure PA8 */
    GPIO_InitStructure.GPIO_Pin = DS18B20_DQ_OUT_PIN;        //温度传感器的DQ管脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;         //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(DS18B20_DQ_OUT_PORT, &GPIO_InitStructure);
    /* Deselect the PA8 Select high */
    GPIO_SetBits(DS18B20_DQ_OUT_PORT,DS18B20_DQ_OUT_PIN);

    DS18B20_Rst();
    return DS18B20_Check();
```

其中对于处理器的PA8管脚的声明在DS18B20.H中进行描述：

```
/*选择PA8作为DS18B20的数据输出输入管脚*/
#define DS18B20_DQ_OUT_PORT              GPIOA
#define DS18B20_DQ_OUT_CLK               RCC_APB2Periph_GPIOA
#define DS18B20_DQ_OUT_PIN               GPIO_Pin_8
#define Set_DS18B20_DQ_OUT      {GPIO_SetBits(DS18B20_DQ_OUT_PORT,DS18B20_DQ_OUT_PIN);}
#define Clr_DS18B20_DQ_OUT      {GPIO_ResetBits(DS18B20_DQ_OUT_PORT,DS18B20_DQ_OUT_PIN);}

#define DS18B20_DQ_IN_PORT              GPIOA
#define DS18B20_DQ_IN_CLK               RCC_APB2Periph_GPIOA
#define DS18B20_DQ_IN_PIN               GPIO_Pin_8
#define DS18B20_DQ_IN           (GPIO_ReadInputDataBit(DS18B20_DQ_IN_PORT, DS18B20_DQ_IN_PIN))
```

那么检测DS18B20是否在位的函数描述，我们下面进行设计：

```
u8 DS18B20_Check(void)      //检测DS18B20温度传感器是否在位：在，返回0；否则返回1
{
    u8 retry=0;
    DS18B20_IO_IN();        //设置 PA0作为输入
    while (DS18B20_DQ_IN&&retry<200)
    {
        retry++;
        delay_us(1);
    };
    if(retry>=200)return 1;
    else retry=0;
    while (!DS18B20_DQ_IN&&retry<240)
    {
        retry++;
        delay_us(1);
    };
    if(retry>=240)return 1;
    return 0;
}
```

DS18B20温度传感器安装可靠后，正常工作情况下，将不停的检测环境温度，并且通过转换将检测到的温度值通过串口打印出来。

```
    while(1)
    {
        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_ms(200);
    }
```

其中最主要的就是DS18B20_Get_Temp()函数，获取环境温度的函数。当然先让我们先了解一个函数DS18B20_Start()，此函数主要就是函数每一次读写操作的第一步骤：复位—>skip ROM（0xCC）—>开始转换（0x44）—>延迟

```
void DS18B20_Start(void)            // ds1820 开始转换
{
    DS18B20_Rst();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc);     // 跳过ROM 空间
    DS18B20_Write_Byte(0x44);     // 转换
}
short DS18B20_Get_Temp(void)
{
    u8 temp;
    u8 TL,TH;
    short tem;
    DS18B20_Start ();             // ds1820开始转换
    DS18B20_Rst();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc);    // 跳过ROM
    DS18B20_Write_Byte(0xbe);    // 读暂存存储器
    TL=DS18B20_Read_Byte();      // 低8位
    TH=DS18B20_Read_Byte();      // 高8位

    if(TH>7)
    {
        TH=~TH;
        TL=~TL;
        temp=0;                  //温度为负
    }else temp=1;                //温度为正
    tem=TH;                      //获得高八位
    tem<<=8;
    tem+=TL;                     //获得低八位
    tem=(float)tem*0.625;        //转换
    if(temp)return tem;          //返回温度值
    else return -tem;
}
```

执行了第一步的操作后，后面从DS18B20_Rst（）后的代码设计将是根据思路（复位—>skip ROM（0xCC）—>读取存储器（0xBE）—>连续读出两个字节数据（TH和TL），温度值—>结束）而进行的，直到取到温度值TH和TL，转换为tem，并进行显示。

6.13.5 下载与现象

在 *神舟I号光盘\神舟I号各例程下载固件*目录下的*13.温度传感器.hex*文件即为前面我们分析的18B20温度传感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I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I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I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。

现象：将编译后的固件下载到神舟I号开发板后，接上DS18B20温度传感器后，通过串口（波特率为115200）进行显示，具体的如下图所示：

6.14 2.4G模块通信试验

在前几节中，我们简单了解了315M的无线模块后，本节将为大家讲解2.4G模块的通信实验，采用的是nRF24L01模块。nRF24L01是一款新型单片射频收发器件，工作在2.4GHz ~ 2.5GHz 开放的ISM频段。同时，本节我们将使用SPI总线实现对nRF24L01模块的控制。但由于nRF24L01模块的收发双向性，我们至少需要使用两个模块同时正常工作才能演示实验。请对本节感兴趣的爱好者至少准备两块带有nRF24L01模块的开发板，以便实验之用。

6.14.1 2.4G模块通信实验的意义与作用

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是如何实现的？从信号的编码，信道传输，信号解码以及传输过程中，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，满足大家在无线通信世界中的求知欲望。

6.14.2 实验原理

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，其特点如下所示：

1、 采用GFSK方式调制，数据传输率为1Mb/s或是2Mb/s；
2、 具有自动应答和自动再发射功能；
3、 125个频道，可以满足多点通信；
4、 具有CRC校验；
5、 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。下面我们简介nRF24L01芯片的工作收发原理：

发射数据时，首先将nRF24L01配置为发射模块：接着把发送地址和发送数据按照时序由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块，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，接收方同时进入发射状态，回传应答信号。直到接收结束，便将使能关闭，进行空闲模式。

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。

6.14.3 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模块是否在位。如果在位，便提示选择nRF24L01模块的工作模式：发送或接收。硬件设计如下所示：

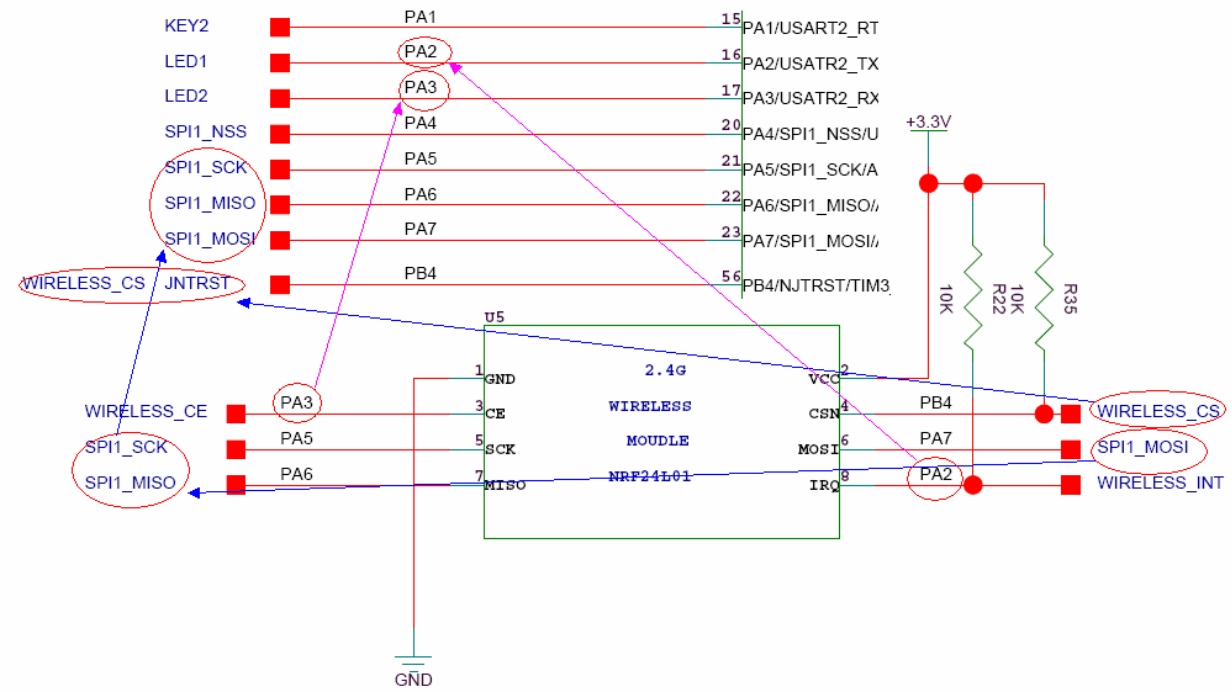


图3.18.2.1 nRF24L01模块与STM32F103RBT6连接图

6.14.4 软件设计

本例程的实验思想是借用两块带有 nRF24L01 无线模块的神舟 I 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：发送或是接收，在开发板上按钮 1 和按钮 2 分别对应着接收模式和发送模式。通过选择按钮 1 或按钮 2，进行无线的通信过程，而对于按钮的初始化和使用操作，在前面的章节已经详细讲解过，在此不累赘。

打开*神舟 I 号光盘\神舟 I 号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\14.2.4G 无线通信实验（神舟 I 号）*目录。进入 *14.2.4G 无线通信实验（神舟 I 号）\ MDK-ARM* 目录后，双击 Project.uvproj 可以打开工程，以下我们学习代码的设计。

首先我们了解 SPI 接口的设计，基本上与 SPI FLASH（W25X16）实验中使用的 SPI 设计相类同，有一点点区别，如下所示，主要是在 SPI 的工作模式和工作频率（在此设置为 9MHz）的区别，同时也在此增加了 JTAG 不使能以及 SW 的使能操作：

```
GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable , ENABLE);          //JTAG-DP 失能 + SW-DP使能

/* SPI1 configuration */                                            //初始化SPI1结构体
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_Low;                          //串行时钟在不操作时，时钟为低电平
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;                        //第一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;  //SPI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
```

接着我们再来了解 nRF24L01 无线模块的驱动设计，详细资料请参考路径“..\ 神舟 I 号光盘\神舟 I 号相关参考手册\外围器件数据手册\NRF24L01 资料”目录中的 nRF24L01 相关资料。

对于 nRF24L01.h 的代码设计，主要关注通过 SPI 访问 nRF24L01 的一些操作命令以及寄存器地址，同时对于 nRF24L01 与处理器相连接的控制管脚的宏定义，还有就是 nRF24L01.C 中使用到的一些函数的声明。

```
//神舟I号开发板
#ifndef __NRF24L01_H
#define __NRF24L01_H
#include "stm32f10x.h"
//NRF24L01 驱动函数
/*****************************************************************************************************/
//NRF24L01寄存器操作命令
#define SPI_READ_REG    0x00  //读配置寄存器,低5位为寄存器地址
#define SPI_WRITE_REG   0x20  //写配置寄存器,低5位为寄存器地址
#define RD_RX_PLOAD     0x61  //读RX有效数据,1~32字节
#define WR_TX_PLOAD     0xA0  //写TX有效数据,1~32字节
#define FLUSH_TX        0xE1  //清除TX FIFO寄存器.发射模式下用
#define FLUSH_RX        0xE2  //清除RX FIFO寄存器.接收模式下用
#define REUSE_TX_PL     0xE3  //重新使用上一包数据,CE为高,数据包被不断发送.
#define NOP             0xFF  //空操作,可以用来读状态寄存器
//SPI(NRF24L01)寄存器地址
#define CONFIG          0x00  //配置寄存器地址;bit0:1接收模式,0发射模式;bit1:电选择;bit2:CRC模式;bit3:CRC使能;
                              //bit4:中断MAX_RT(达到最大重发次数中断)使能;bit5:中断TX_DS使能;bit6:中断RX_DR使能
#define EN_AA           0x01  //使能自动应答功能  bit0~5,对应通道0~5
#define EN_RXADDR       0x02  //接收地址允许,bit0~5,对应通道0~5
#define SETUP_AW        0x03  //设置地址宽度(所有数据通道):bit1,0:00,3字节;01,4字节;02,5字节;
#define SETUP_RETR      0x04  //建立自动重发;bit3:0,自动重发计数器;bit7:4,自动重发延时 250*x+86us
#define RF_CH           0x05  //RF通道,bit6:0,工作通道频率;
#define RF_SETUP        0x06  //RF寄存器;bit3:传输速率(0:1Mbps,1:2Mbps);bit2:1,发射功率;bit0:低噪声放大器增益
#define STATUS          0x07  //状态寄存器;bit0:TX FIFO满标志;bit3:1,接收数据通道号(最大:6);bit4,达到最多次重发
                              //bit5:数据发送完成中断;bit6:接收数据中断;
#define MAX_TX          0x10  //达到最大发送次数中断
#define TX_OK           0x20  //TX发送完成中断
#define RX_OK           0x40  //接收到数据中断
#define OBSERVE_TX      0x08  //发送检测寄存器,bit7:4,数据包丢失计数器;bit3:0,重发计数器
#define CD              0x09  //载波检测寄存器,bit0,载波检测;
#define RX_ADDR_P0      0x0A  //数据通道0接收地址,最大长度5个字节,低字节在前
#define RX_ADDR_P1      0x0B  //数据通道1接收地址,最大长度5个字节,低字节在前
#define RX_ADDR_P2      0x0C  //数据通道2接收地址,最低字节可设置,高字节,必须同RX_ADDR_P1[39:8]相等;
#define RX_ADDR_P3      0x0D  //数据通道3接收地址,最低字节可设置,高字节,必须同RX_ADDR_P1[39:8]相等;
#define RX_ADDR_P4      0x0E  //数据通道4接收地址,最低字节可设置,高字节,必须同RX_ADDR_P1[39:8]相等;
#define RX_ADDR_P5      0x0F  //数据通道5接收地址,最低字节可设置,高字节,必须同RX_ADDR_P1[39:8]相等;
#define TX_ADDR         0x10  //发送地址(低字节在前),ShockBurstTM模式下,RX_ADDR_P0与此地址相等
#define RX_PW_P0        0x11  //接收数据通道0有效数据宽度(1~32字节),设置为0则非法
#define RX_PW_P1        0x12  //接收数据通道1有效数据宽度(1~32字节),设置为0则非法
#define RX_PW_P2        0x13  //接收数据通道2有效数据宽度(1~32字节),设置为0则非法
#define RX_PW_P3        0x14  //接收数据通道3有效数据宽度(1~32字节),设置为0则非法
#define RX_PW_P4        0x15  //接收数据通道4有效数据宽度(1~32字节),设置为0则非法
#define RX_PW_P5        0x16  //接收数据通道5有效数据宽度(1~32字节),设置为0则非法
#define FIFO_STATUS     0x17  //FIFO状态寄存器;bit0,RX FIFO寄存器空标志;bit1,RX FIFO满标志;bit2,3,保留
                              //bit4,TX FIFO空标志;bit5,TX FIFO满标志;bit6,1,循环发送上一数据包.0,不循环;
```

下面是有关 SPI 控制管脚的宏定义以及函数声明：

```
//NRF24L01控制操作
//NRF2401片选信号
#define Clr_NRF24L01_CE       {GPIO_ResetBits(GPIOA,GPIO_Pin_3);}
#define Set_NRF24L01_CE       {GPIO_SetBits(GPIOA,GPIO_Pin_3);}

//SPI片选信号
#define Clr_NRF24L01_CSN      {GPIO_ResetBits(GPIOB,GPIO_Pin_4);}
#define Set_NRF24L01_CSN      {GPIO_SetBits(GPIOB,GPIO_Pin_4);}

//NRF2401_IRQ数据输入
#define Clr_NRF24L01_IRQ      {GPIO_ResetBits(GPIOA,GPIO_Pin_2);}
#define Set_NRF24L01_IRQ      {GPIO_SetBits(GPIOA,GPIO_Pin_2);}

#define NRF24L01_IRQ          (GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_2))

//NRF24L01发送接收数据宽度定义
#define TX_ADR_WIDTH    5                                   //5字节的地址宽度
#define RX_ADR_WIDTH    5                                   //5字节的地址宽度
#define TX_PLOAD_WIDTH  32                                  //20字节的用户数据宽度
#define RX_PLOAD_WIDTH  32                                  //20字节的用户数据宽度


void NRF24L01_Init(void);                                    //NRF24l01初始化
void RX_Mode(void);                                          //配置为接收模式
void TX_Mode(void);                                          //配置为发送模式
u8 NRF24L01_Write_Buf(u8 regaddr, u8 *pBuf, u8 datalen);     //写数据区
u8 NRF24L01_Read_Buf(u8 regaddr, u8 *pBuf, u8 datalen);      //读数据区
u8 NRF24L01_Read_Reg(u8 regaddr);                            //读寄存器
u8 NRF24L01_Write_Reg(u8 regaddr, u8 data);                  //写寄存器
u8 NRF24L01_Check(void);                                     //检查NRF24L01是否在位
u8 NRF24L01_TxPacket(u8 *txbuf);                             //发送一个包的数据
u8 NRF24L01_RxPacket(u8 *rxbuf);                             //接收一个包的数据
#endif
```

在 nRF24L01.C 的驱动代码中，主要是 nRF24L01 的初始化，nRF24L01 是否在位的检测函数，以及通过 SPI 读写 nRF24L01 的函数，还有就是 nRF24L01 的发送模式、接收模式以及数据包的发送和接收处理。

nRF24L01 初始化函数

```
//神舟I号开发板
#include "NRF24L01.h"
#include "delay.h"
#include "spi.h"
//NRF24L01 驱动函数

const u8 TX_ADDRESS[TX_ADR_WIDTH]={0x34,0x43,0x10,0x10,0x01}; //发送地址
const u8 RX_ADDRESS[RX_ADR_WIDTH]={0x34,0x43,0x10,0x10,0x01}; //发送地址
//初始化24L01的IO口
void NRF24L01_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB, ENABLE);  //使能GPIO的时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4;       //PA3作为NRF2401的CE   PA4作为SPI_NSS的连接信号
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;           //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    Set_NRF24L01_CE;                                           //初始化时先拉高
    Set_SPI_7843_NSS;

   //NRF2401_CS（PB4）、FLASH_CS(PB6)、SD_CS(PB7)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4|GPIO_Pin_6|GPIO_Pin_7;  //NRF2401_CS、FLASH_CS、SD_CS
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP ;   //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    Set_SD_CS;                                          //SPI总线的其他设备置无效
    Set_SPI_FLASH_CS;
    Set_NRF24L01_CSN;                                   //初始化时先拉高

    //配置NRF2401的IRQ
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU  ;     //上拉输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA,GPIO_Pin_2);

    SPI1_Init();                                       //初始化SPI
    Clr_NRF24L01_CE;                                   //使能24L01
    Set_NRF24L01_CSN;                                  //SPI片选取消
}

//上电检测NRF24L01是否在位
//写5个数据然后再读回来进行比较，相同时返回值:0，表示在位;否则返回1，表示不在位
u8 NRF24L01_Check(void)
{
    u8 buf[5]={0XA5,0XA5,0XA5,0XA5,0XA5};
    u8 buf1[5];
    u8 i;
    NRF24L01_Write_Buf(SPI_WRITE_REG+TX_ADDR,buf,5);//写入5个字节的地址.
    NRF24L01_Read_Buf(TX_ADDR,buf1,5);                 //读出写入的地址
    for(i=0;i<5;i++)if(buf1[i]!=0XA5)break;
    if(i!=5)return 1;                                  //NRF24L01不在位
    return 0;                                          //NRF24L01在位
}
//通过SPI写寄存器
u8 NRF24L01_Write_Reg(u8 regaddr,u8 data)
{
    u8 status;
    Clr_NRF24L01_CSN;                      //使能SPI传输
    status =SPI1_ReadWriteByte(regaddr);   //发送寄存器号
    SPI1_ReadWriteByte(data);              //写入寄存器的值
    Set_NRF24L01_CSN;                      //禁止SPI传输
    return(status);                        //返回状态值
}
//读取SPI寄存器值 ，regaddr:要读的寄存器
u8 NRF24L01_Read_Reg(u8 regaddr)
{
    u8 reg_val;
    Clr_NRF24L01_CSN;                  //使能SPI传输
    SPI1_ReadWriteByte(regaddr);       //发送寄存器号
    reg_val=SPI1_ReadWriteByte(0XFF);//读取寄存器内容
    Set_NRF24L01_CSN;                  //禁止SPI传输
    return(reg_val);                   //返回状态值
}
```

```
//在指定位置读出指定长度的数据
//*pBuf:数据指针
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Read_Buf(u8 regaddr,u8 *pBuf,u8 datalen)
{
    u8 status,u8_ctr;
    Clr_NRF24L01_CSN;                       //使能SPI传输
    status=SPI1_ReadWriteByte(regaddr);     //发送寄存器值(位置),并读取状态值
    for(u8_ctr=0;u8_ctr<datalen;u8_ctr++)pBuf[u8_ctr]=SPI1_ReadWriteByte(0XFF);//读出数据
    Set_NRF24L01_CSN;                       //关闭SPI传输
    return status;                          //返回读到的状态值
}
//在指定位置写指定长度的数据
//*pBuf:数据指针
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Write_Buf(u8 regaddr, u8 *pBuf, u8 datalen)
{
    u8 status,u8_ctr;
    Clr_NRF24L01_CSN;                                   //使能SPI传输
    status = SPI1_ReadWriteByte(regaddr);               //发送寄存器值(位置),并读取状态值
    for(u8_ctr=0; u8_ctr<datalen; u8_ctr++)SPI1_ReadWriteByte(*pBuf++); //写入数据
    Set_NRF24L01_CSN;                                   //关闭SPI传输
    return status;                                      //返回读到的状态值
}
//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(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
}
//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
}
```

```
//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;                                 //CE为高,进入接收模式
}
//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;                                  //CE为高,10us后启动发送
}
```

到此，我们已经完成 nRF24L01 的驱动函数。接着我们继续了解 main 函数中的一些相关的设计。在初始化串口、点灯、按键以及 SPI 和 nRF24L01 模块后，我们看看主程序中对于 nRF24L01 是如何调用的。

检测 nRF24L01 是否在位：

```
while(NRF24L01_Check())//检测不到24L01
{
    printf("nRF24L01检测出错!\n\r");
    delay_ms(500);
    printf("请确认nRF24L01的连接!    ");
    delay_ms(500);
}
```

提示按键模式，以及扫描按键，确认工作在哪种模式：

```
printf("\n\r请选择接收或发送模式\n\r");
printf("按钮1:接收模式   按钮2:发送模式\n\r");

   while(1)//在该部分确定进入哪个模式!
   {
    key=ReadKeyDown();
    if(key==1)
    {
        mode=0;
        break;
    }else if(key==2)
    {
        mode=1;
        break;
        }
    delay_ms(5);
   }
```

接收模式的处理：

```
if(mode==0)//RX模式
  {
    printf("NRF24L01 接收模式\n\r");
    printf("等待接收数据\n\r");
    RX_Mode();
    while(1)
    {
        if(NRF24L01_RxPacket(tmp_buf)==0)//一旦接收到信息,则显示出来.
        {
            tmp_buf[32]=0;//加入字符串结束符
            printf("接收到数据为:");
            for (i=0;i < 32;i++)
            {
                printf("%c",tmp_buf[i]);
            }
            printf("\n\r");

            delay_us(2000);
        }
        else delay_us(100);
        t++;
        if(t==10000)//大约1s钟改变一次状态
        {
            t=0;
        }
    }
  }
```

发送模式的处理：

```
else//TX模式
{
 printf("NRF24L01 发送模式\n\r");
 TX_Mode();
 mode=' ';//从空格键开始
 while(1)
 {
     if(NRF24L01_TxPacket(tmp_buf)==TX_OK)
     {
         printf("正在发送数据: ");
         key=mode;
         for(t=0;t<32;t++)
         {
             key++;
             if(key>('~'))key=' ';
             tmp_buf[t]=key;
             printf("%c",tmp_buf[t]);
         }
         printf("\n\r");
         mode++;
         if(mode>'~')mode=' ';
         tmp_buf[32]=0;//加入结束符
         delay_ms(100);

     }else
     {
         printf("请确认接收端是否正常\n\r");
     }
     delay_ms(1500);
 }
 }
```

主函数中涉及到点灯以及按键的初始化等，在此不讨论。

6.14.5 下载与测试现象

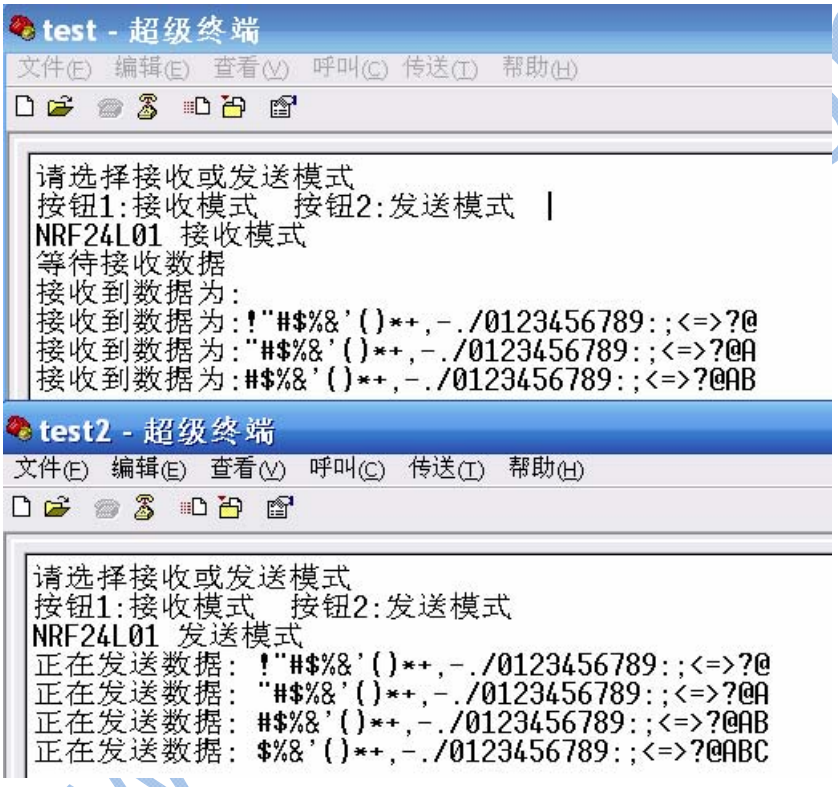
在 *神舟I号光盘\编译好的固件*\目录下的 *"24G无线通信实验.hex"* 文件即为前面我们分析*14.2.4G无线通信实验（神舟I号）*的编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I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I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。

测试现象：将编译生成的固件分别下载到两块神舟 I 号开发板上，接上 2.4G 无线模块，连接串口、打开 PC 串口设置波特率为 115200，在串口提示下分别选择发送和接收模式，串口打印如下图所示：



6.15 USB遥控鼠标实验

本节我们将学习USB设备。USB接口为STM32F103RBT6自带的接口，且只能作为从设备，而不能作为主机模式。本节将借助STM32的USB接口实现遥控PC的鼠标实验。

6.15.1 实验的意义与作用

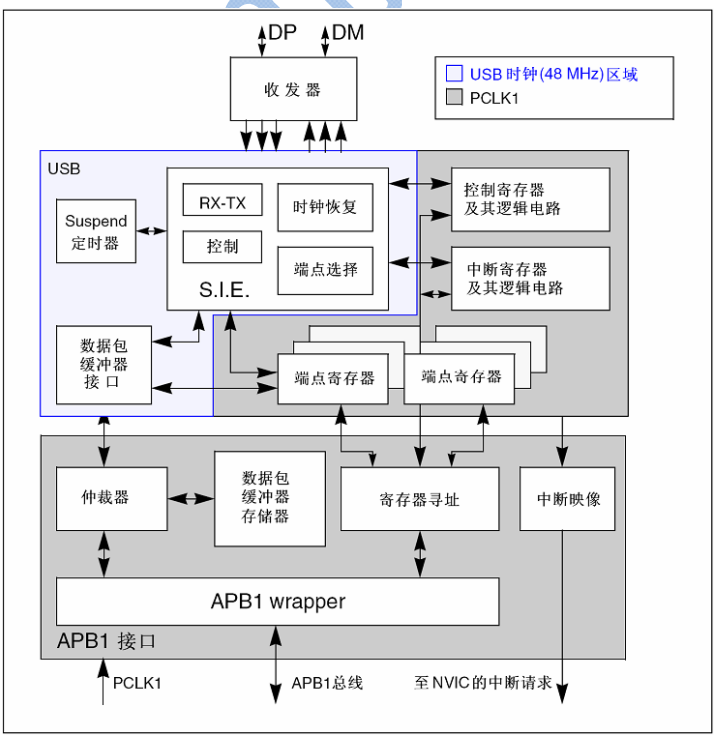
USB：Universal Serial BUS（通用串行总线），作为一种支持即插即用和热插拔功能的方便的常用接口，已经广泛运用：如鼠标、键盘、打印机、扫描仪、摄像头、闪存盘、MP3机、手机、数码相机、移动硬盘、外置光软驱、USB网卡、ADSL Modem、Cable Modem等，几乎所有的外部设备。因此，借助STM32开发板学习USB设备接口是具有重要意义；同时，在神舟I号的从USB模式下，我们将引导大家扩展视野，积极了解USB作为主设备或是OTG时，USB的使用情况，扩展大家的知识面，提高大家的学习兴趣。

6.15.2 实验原理

USB 的发展已经经过了近十年，从 USB1.0~USB2.0 甚至即将面世的 USB3.0 的发展都体现出 USB 的重要市场地位。

USB 设备作为一个完整的硬件设备时，是由硬件和驱动两部分组成的。硬件部分上，有的 phy 和 mac 是分开的，也有的 phy 集成在处理器内部（比如 STM32 的 USB 接口）。其他硬件的详见下面的硬件设计。下图为 USB 的框图。

驱动部分中包括一些系统的配置、CPU 的设备以及 USB 协议栈模块等几个部分。另外、USB 总线上传输的数据有两种，一是差模数据线上的包；另外一种是特殊定义的数据的数据，如复位信号、唤醒信号等。所以，驱动中的 USB 栈需要能够识别和处理这些不同的数据内容，因此，整个 USB 协议栈的内容非常庞大。关于 USB 协议等一些复杂的内容，由于作者知识有限，篇幅有限，在此简单了解而已。



USB的框图

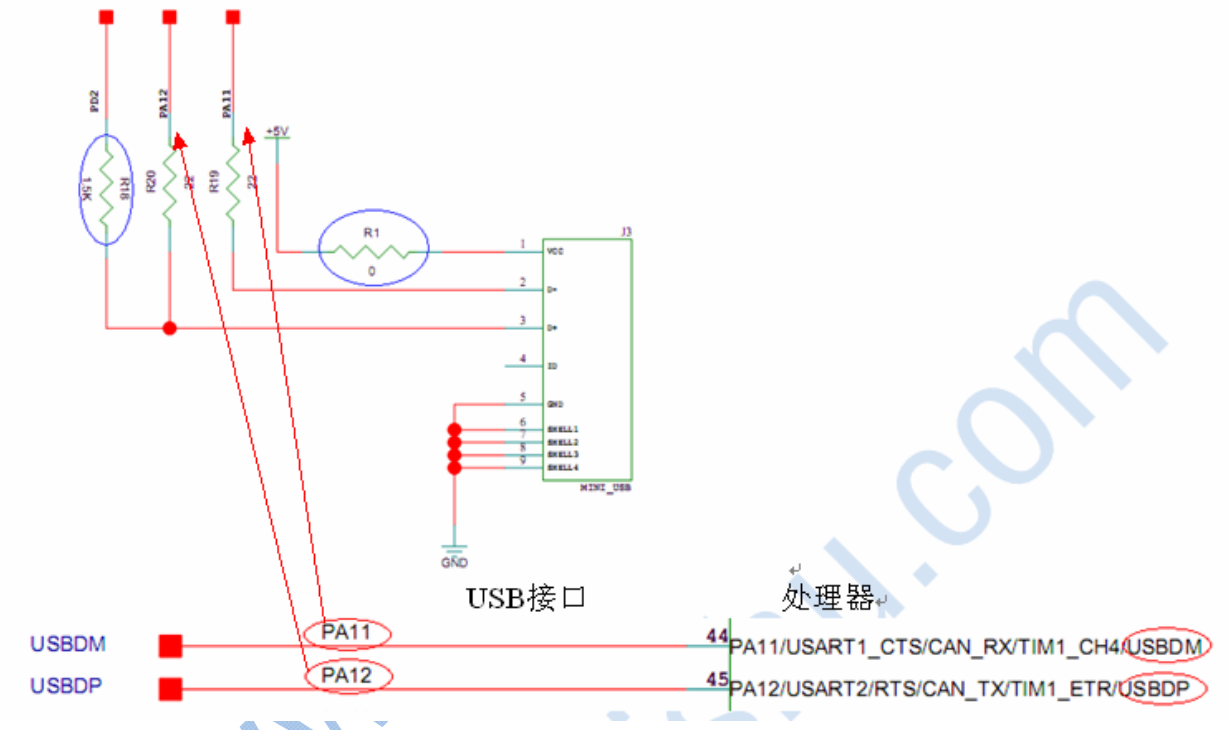
这里给大家提几个寄存器：端点寄存器、中断寄存器、控制寄存器等，详见《【中文】STM32F系列ARM内核32位高性能微控制器参考手册V10_1.pdf》第406~420页文档内容。

6.15.3 硬件设计

神舟I号的USB部分设计，主要提及两点：一是关于MINI-USB接口可以作为神舟I号整板的供电接口，接入5V电源；二是关于USBDP端的上拉，采用1.5K电阻可控上下拉。主要是用于接入USB线缆到PC时，处理器通过PD2对USBDP端进行上拉，PC通过此电平的变化，识别到USB设备。

在这里我们浅谈USB的硬件连接情况：标准的USB接口共有四根线构成，包括VCC（5V）、GND、USBDP和USBDM。其中USBDP和USBDM是以差分电压的方式进行数据的传输的。而在能够支持主机模式接口上，USBDP和USBDM都是接了15K的电阻到地，如果没有设备接入时，USBDP和USBDM都是低电平，而如果有USB设备接入时，相应的管脚通过从设备的1.5K电阻上拉而变为高电平，从而主机判断是高速设备还是低速设备。如USBDP接1.5K电阻到VCC，表示此USB设备为高速的；如果USBDM接1.5K电阻到VCC，表示此USB设备为低速。

另外采用PD2管脚可控1.5K的上下拉，**好处主要是**：可以完成程序的初始化后再打开USBDP的上拉，避免PC与USB设备相连后，PC要求USB设备马上响应USB总线上的信号，但此时USB设备可能正在初始化，导致USB设备与PC通讯失败。



6.15.4 软件设计

本例程的设计思想为：通过按钮1和按钮2，控制PC的鼠标的移动。

打开神舟 I 号光盘\神舟 I 号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\15.USB 遥控鼠标实验（神舟 I 号）目录。进入 15. USB 遥控鼠标实验（神舟 I 号）\ MDK-ARM 目录后，双击 Project.uvproj 可以打开工程，以下我们学习代码的设计。

本例程移植ST的JoyStickMouse例程的相关部分的。

下面我们了解主程序的设计，主要实现USB的相关初始化控制，包括时钟，中断等，同时扫描配置按键是否被按下，从而实现通过USB遥控鼠标方向的功能，详见下面。

```
int main(void)
{

#ifdef DEBUG
   debug();
#endif

  Printf_Init();

  printf("\n\r神舟I号 USB触控鼠标实验\n\r");
  printf("\n\r按钮1: 鼠标向上; 按钮2: 鼠标向下\n\r");

  Set_System();                /*初始化RCC,HSE等设备*/

  USB_Interrupts_Config();     /*配置USB的中断函数*/

  Set_USBClock();              /*设置USB为48MHz*/

  USB_Init();                  /*USB的相关初始化*/

  while (1)
  {
    if (JoyState() != 0)       /*判断案件是否按下*/
    {
      Joystick_Send(JoyState());     /*将相应按键按下的转换为X/Y方向上的位置变化,
                                       并通过发送缓冲区修改鼠标位置*/
      Delay(10000);
    }
  }
}
```

Printf打印函数我们就不说了，第二个函数为Set_System()，主要针对HSE、RCC一些设备的初始化，需要强调的是对于我们再实验中使用的两个按钮的声明（本实验考虑神舟I号板只有两个按钮，因此只控制鼠标进行上下移动，其他朋友可以到程序中修改返回者以修改控制鼠标的不同方向），以及1.5K电阻上拉管脚（PD2）的配置，如下所示：

```
/* PD.02 used as USB pull-up */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOD, &GPIO_InitStructure);

/* Configure the JoyStick IOs */
/* Key up + Key down  */
GPIO_InitStructure.GPIO_Pin = JOY_UP | JOY_DOWN ;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

以下是实现PD2的上拉作用的：

```
void USB_Cable_Config (FunctionalState NewState)
{
  if (NewState != DISABLE)
  {

    GPIO_SetBits(GPIOD, GPIO_Pin_2);
  }
  else
  {
    GPIO_ResetBits(GPIOD, GPIO_Pin_2);
  }
}
```

第三个函数为USB的中断函数的配置，同时我们将USB的中断处理函数也归结到一起，便于查阅，如下所示：

```
void USB_Interrupts_Config(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;

#ifdef  VECT_TAB_RAM
  /* Set the Vector Table base location at 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH */
  /* Set the Vector Table base location at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif

  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);

  NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
/*******************************************************************************
* Function Name  : USB_LP_CAN_RX0_IRQHandler
* Description    : This function handles USB Low Priority or CAN RX0 interrupts
*                  requests.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void USB_LP_CAN1_RX0_IRQHandler(void)
{
    USB_Istr();
}
```

其中USB_Istr()函数为USB的中断状态寄存器的分类判断，从而实现处理不同的中断源。

第四个函数主要是配置USB的时钟为48MHz，72MHz的1.5分频得到的，具体如下：

```
/*******************************************************************
* Function Name  : Set_USBClock
* Description    : Configures USB Clock input (48MHz).
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************
void Set_USBClock(void)
{
  /* Select USBCLK source */
  RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);

  /* Enable USB clock */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
}
```

第五个函数为USB的相关的初始化，涉及到的内容较为多，也相对复杂，在此不详述。

我们下面看看按键的扫描函数JoyState（），其实作为原创的，是可以控制鼠标的上下左右等方向，在此根据神舟I号实际情况，只配置其中的上和下两个方向，其他的由朋友们自己实现，如下所示：

```
u8 JoyState(void)
{
  /* "up" key is pressed */
  if (!GPIO_ReadInputDataBit(GPIOA, JOY_UP))
  {
    return UP;
  }
  if (!GPIO_ReadInputDataBit(GPIOA, JOY_DOWN))
  {
    return DOWN;
  }
  /* No key is pressed */
  else
  {
    return 0;
  }
}
```

当检测到按键按下后，我们便需要分析处理到底是需要控制鼠标的那个方向呢？对上述JoyState（）的返回值进行分析判断，从而实现遥控鼠标的移动方向，如下所示：

```
void Joystick_Send(u8 Keys)
{
  u8 Mouse_Buffer[4] = {0, 0, 0, 0};
  s8 X = 0, Y = 0,BUTTON=0;
  switch (Keys)
  {
    case LEFT:
      X += CURSOR_STEP;
      break;
    case RIGHT:
      X -= CURSOR_STEP;
      break;
    case UP:
      Y -= CURSOR_STEP;
      break;
    case DOWN:
      Y += CURSOR_STEP;
      break;
     case LEFT_BUTTON:
      BUTTON = BUTTON|0x01;
      break;
     case RIGHT_BUTTON:
      BUTTON = BUTTON|0x02;
      break;
    default:
      return;
  }
  /* prepare buffer to send */
  Mouse_Buffer[0] = BUTTON;
  Mouse_Buffer[1] = X;
  Mouse_Buffer[2] = Y;
  /*copy mouse position info in ENDP1 Tx Packet Memory Area*/
  UserToPMABufferCopy(Mouse_Buffer, GetEPTxAddr(ENDP1), 4);
  if(Mouse_Buffer[0] != 0)
  {
    Mouse_Buffer[0] = 0;
    UserToPMABufferCopy(Mouse_Buffer, GetEPTxAddr(ENDP1), 4);
  }
  /* enable endpoint for transmission */
  SetEPTxValid(ENDP1);
}
```

后面的工作就是将共用的数据缓冲区，以改变鼠标的当前状态。关于USB的软件方面我们就简单讲解到这里，后续的需要朋友们自己深究。

6.15.5 下载与测试

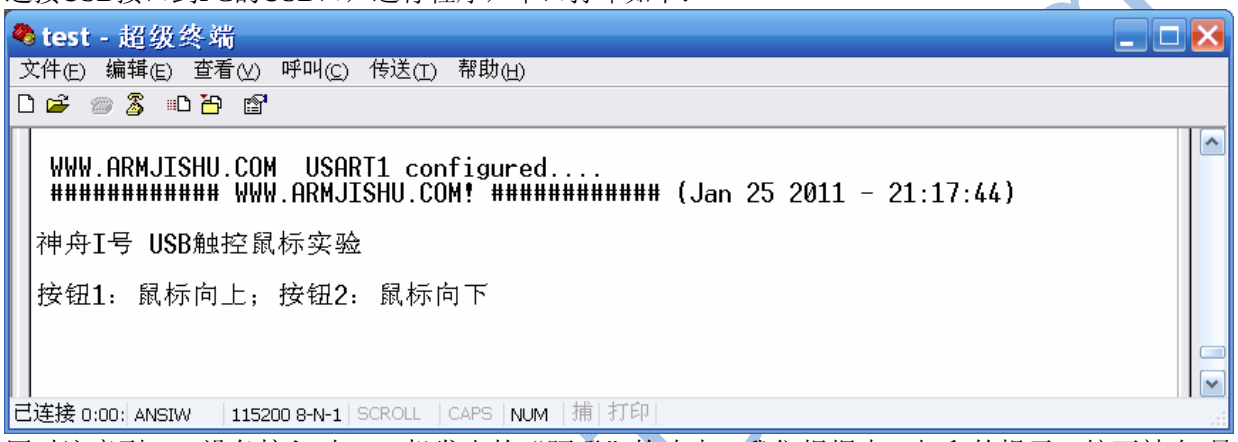
在 *神舟I号光盘\编译好的固件*目录下的 *"USB遥控鼠标实验.hex"* 文件即为前面我们分析*15. USB遥控鼠标实验（神舟I号）*的编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I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I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。

测试现象：将编译好的固件下载到神舟I号开发板上，连接串口到PC上，设置好波特率为115200，连接USB接口到PC的USB口，运行程序，串口打印如下：



同时注意到USB设备接入时，PC机发出的"叮噔"的响声。我们根据串口打印的提示，按下神舟I号的按钮1，观察鼠标的方向，按下按钮2，观察鼠标的方向。

本实验先通过串口以及按钮控制鼠标的方向，下次更新，我们将通过LCD触摸屏，触摸控制鼠标，敬请期待！

www.armjishu.com

6.16 Micro SD卡实验

上节我们带着大家学习了如何使用STM32的USB来做一个遥控PC鼠标，本节我们将继续学习Micro SD卡的实验。学习这两节后，我们将在下一节，将USB以及Micro SD卡实验综合起来，做成SD-USB读卡器的实验，敬请期待！

6.16.1 实验的意义与作用

神舟I号开发板板载了Micro SD卡座，Micro SD卡作为常见的存储设备，是很多便携设备的存储媒介，比如手机存储卡等。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，

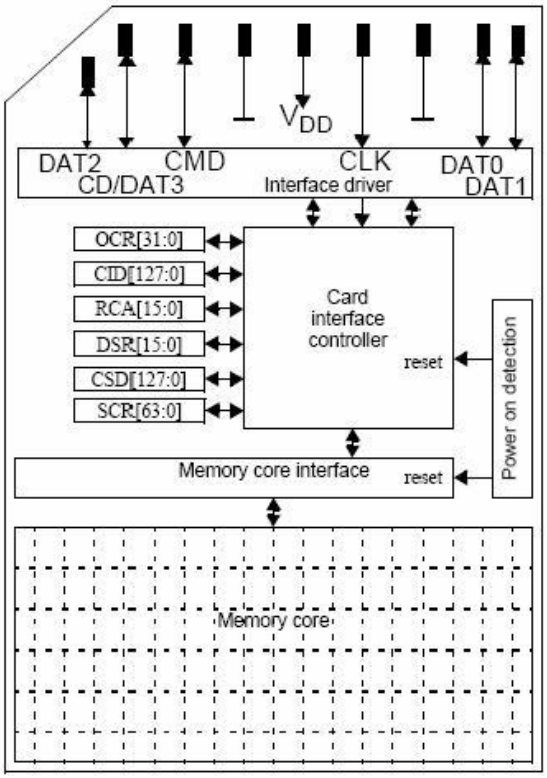
6.16.2 实验原理

SD卡可以通过SDIO方式来进行读写，也可以通过SPI方式来进行读写，神舟I号中采用SPI对SD卡进行读写操作，以简化电路的设计。

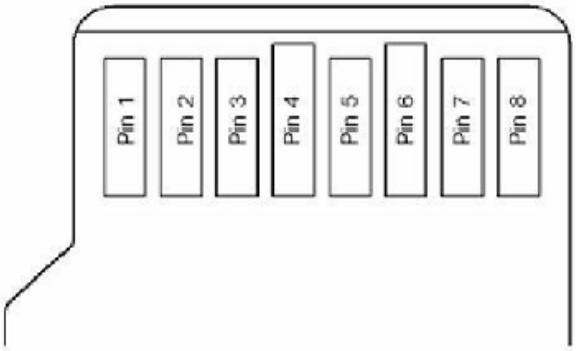
管脚	1	2	3	4	5	6	7	8	9
SDIO 模式	CD/DAT3	CMD	VSS	VCC	CLK	VSS	DAT0	DAT1	DAT2
SPI 模式	CS	MOSI	VSS	VCC	CLK	VSS	MISO	NC	NC

SPI模式做SD数据操作时根本不需要知道FAT，这时候SD卡对于我们来说实际上就是个大的、快速的、方便的、容量可变的外部存储器。

下面我们先简单了解SD卡的相关资料，如下所示为SD的内部框图以及神舟I号自带的Micro SD开管脚图

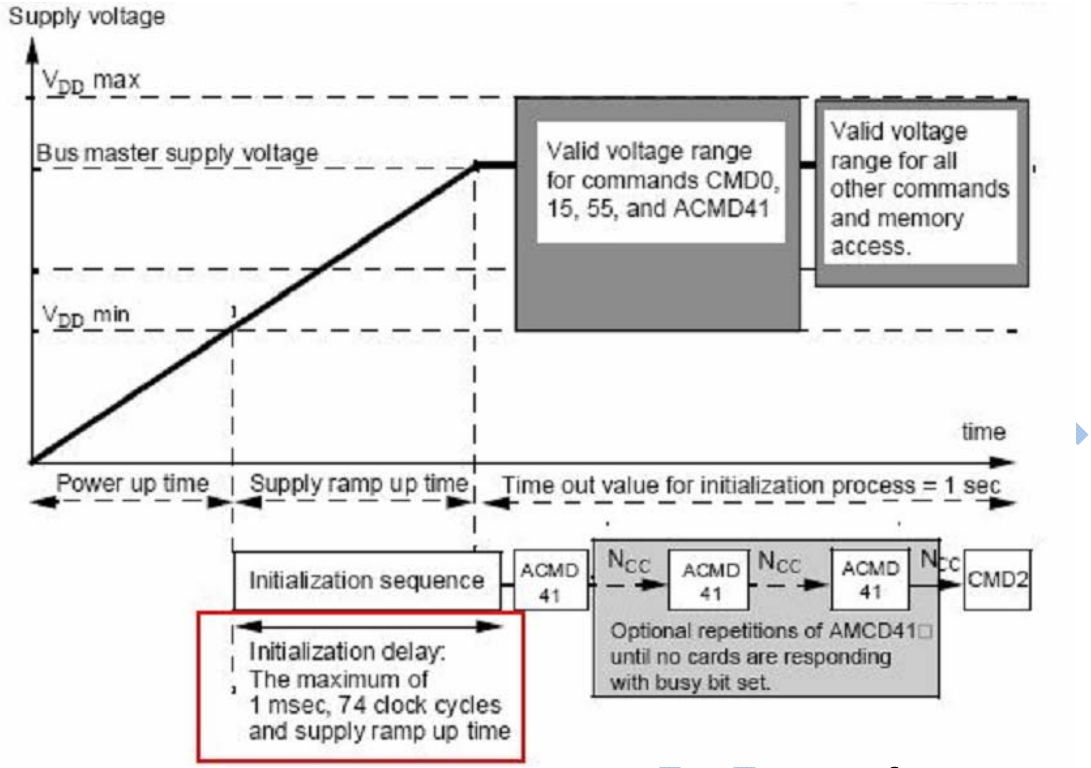


SD卡内部框图



Micro SD卡管脚图

SD卡上电后的初始化流程：



神舟I号上电后，主机host启动SCK以及在CMD线上发送74个高电平的信号①，接着发送CMD0进行SPI模式，然后发送CMD1激活初始化进程。注意，识别阶段的时钟频率必须限定在400kHz以内，而在SD卡识别后，时钟频率便可提高到25MHz。

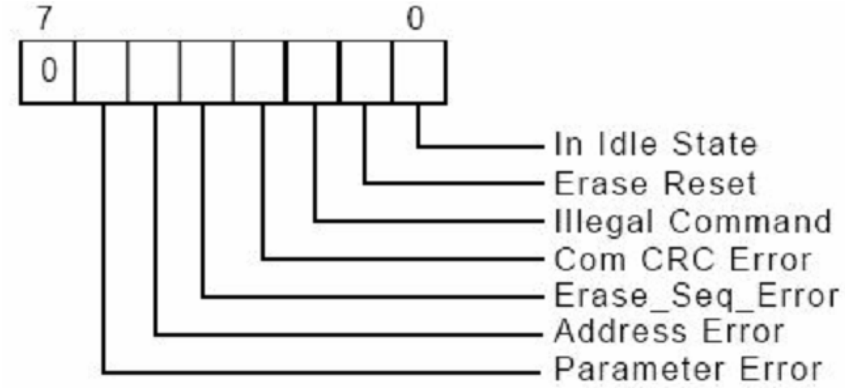
为什么要在CMD0上发送74个CLK的高电平的信号呢？因为在上电初期，电压的上升过程根据SD卡组织的计算约合为64个CLK周期才能到达SD卡的正常工作电压，也就是所说的Supply ramp up time，其它的10个CLK是为了与SD卡同步，之后开始CMD0的操作。如果发送100个CLK周期也是可以的。

在此提到了CMD0，我们就继续了解除了CMD0外的其它的SPI的控制命令：

Command	Mnemonic	Argument	Reply	Description
0 (0x00)	GO_IDLE_STATE	none	R1	Resets the SD card.
9 (0x09)	SEND_CSD	none	R1	Sends card-specific data.
10 (0x0a)	SEND_CID	none	R1	Sends card identification.
17 (0x11)	READ_SINGLE_BLOCK	address	R1	Reads a block at byte address.
24 (0x18)	WRITE_BLOCK	address	R1	Writes a block at byte address.
55 (0x37)	APP_CMD	none	R1	Prefix for application command.
59 (0x3b)	CRC_ON_OFF	Only Bit 0	R1	Argument sets CRC on (1) or off (0).
41 (0x29)	SEND_OP_COND	none	R1	Starts card initialization.

其它详细的资料，请参考《SD卡的读写规范》等文档 。

而其中有些命令发送出去后会有返回值，表示的是错误码。比如CMD0，CMD1返回值为R1格式，一个字节的长度，其中第0和第7位为0，其它位表示错误码，详细如下所示：

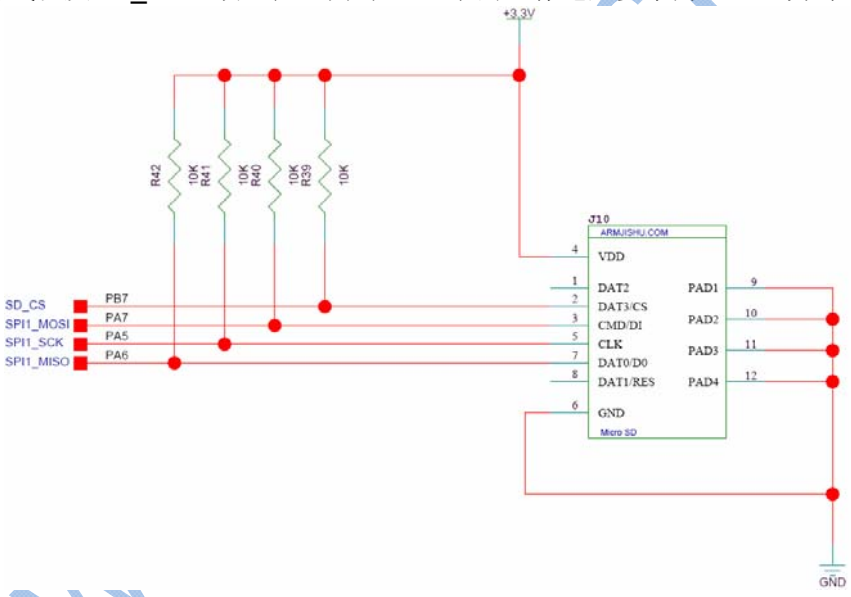


下面再简单了解向SD卡写入一个CMD或是ACMD指令的过程：

首先使CS为低电平，SD卡使能；其次在SD卡的Din写入指令；写入指令后还要附加8个填充时钟，是为了SD卡完成内部操作；之后在SD卡的Dout上接收回应；回应接收完毕，使CS为低电平，在附加8个填充时钟。其中附加的8个时钟周期是为允许SD完成任何没有完结的操作，保证SD的可靠操作。对应这些额外的时钟输入数据必须全为1。

6.16.3 硬件设计

神舟I号SD卡的硬件设计主要是采用SPI模式，使用处理器的SPI接口与SD卡相连接，同时要求SPI总线以及SD_CS必须上拉。同时，SD卡的工作电压要求为3.3V。另外SD的PAD都接地。



6.16.4 软件设计

前面我们已经提到过本例程采用SPI接口对SD卡进行操作，而SPI总线的驱动和使用介绍在前面已经讲解过了，同样的串口的初始化配置等在此也不说明，本节软件的设计主要是针对SD卡的驱动而进行的。

打开*神舟 I 号光盘\神舟 I 号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\15.Micro SD 卡实验（神舟 I 号）*目录。进入 *15.Micro SD 卡实验（神舟 I 号）\ MDK-ARM* 目录后，双击 Project.uvproj 可以打开工程，以下我们学习代码的设计。

首先我们先看看SD卡的初始化，此初始化过程也包括对SD卡的是否在位的检测，如下所示：

```
while(SD_Init()!=0)                          //检测SD卡是否在位
  {
      printf("SD Card 检测出错!");           //检测不到SD卡
      Delay(50000);
      printf("请检查!       ");
      Delay(50000);
  }
```

SD卡的初始化步骤如下所示：1、初始化读写SD卡的硬件条件（SPI接口和其它有用的管脚，如写保护）；2、上电延时过程（74个CLK以上）；3、复位SD卡 **CMD0**；4、激活SD卡，内部初始化并获取存储卡的类型**CMD1**，**CMD55**，**ACMD41**；5、查询OCR，获取卡供电情况 **CMD58**；6、是否使用CRC **CMD59** ；7、设置读、写块数据长度，512B **CMD16**；8、读取CSD，获取存储卡的其它参数信息，**CMD9**；9、8个CLK后，禁止片选。

以下具体的代码设计：

1、初始化读写SD卡的硬件条件，将SPI总线上悬挂的其它设备的片选都至高，使其无效。

```
u8 SD_Init(void)
    {
    GPIO_InitTypeDef GPIO_InitStructure;
    u8 r1;      // 存放SD卡的返回值
    u16 retry;  // 用来进行超时计数
    u8 buff[6];
    //设置硬件上与SD卡相关联的控制引脚输出
    //避免NRF24L01/W25X16等的影响

    RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB, ENABLE );

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;       // PA4作为SPI_NSS的连接信号
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;          //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    Set_SPI_7843_NSS;                                         //初始化时先拉高

   //NRF2401_CS（PB4）、FLASH_CS(PB6)、SD_CS(PB7)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4|GPIO_Pin_6|GPIO_Pin_7;  //NRF2401_CS、FLASH_CS、SD_CS
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP ;   //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    Set_SPI_FLASH_CS;                                   //SPI总线的其他设备置无效
    Set_NRF24L01_CSN;                                   //初始化时先拉高

    SPI1_Init();
```

2、延迟74个CLK以上，并且复位SD卡到idle状态，都是由SD_Idle_Sta（）函数实现。

```
Set_SD_CS;
if(SD_Idle_Sta()) return 1;//超时返回1 设置到idle 模式失败
//-----------------SD卡复位到idle结束-----------------
//获取卡片的SD版本信息
Clr_SD_CS;
```

3、激活SD卡，内部初始化并获取存储卡的类型CMD1，CMD55，ACMD41：

```
//-----------------SD卡、MMC卡初始化开始-----------------
//发卡初始化指令CMD55+ACMD41
// 如果有应答，说明是SD卡，且初始化完成
// 没有回应，说明是MMC卡，额外进行相应初始化
retry = 0;
do
    {
    //先发CMD55，应返回0x01；否则出错
    r1 = SD_SendCommand(CMD55, 0, 0);
    if(r1 == 0XFF)return r1;//只要不是0xff,就接着发送
    //得到正确响应后，发ACMD41，应得到返回值0x00，否则重试200次
    r1 = SD_SendCommand(ACMD41, 0, 0);
    retry++;
    }while((r1!=0x00) && (retry<400));
// 判断是超时还是得到正确回应
// 若有回应：是SD卡；没有回应：是MMC卡
//----------MMC卡额外初始化操作开始------------
if(retry==400)
    {
    retry = 0;
    //发送MMC卡初始化命令（没有测试）
    do
        {
        r1 = SD_SendCommand(1,0,0);
        retry++;
        }while((r1!=0x00)&& (retry<400));
    if(retry==400)return 1;   //MMC卡初始化超时
    //写入卡类型
    SD_Type = SD_TYPE_MMC;
    }
//----------MMC卡额外初始化操作结束------------
```

4、查询OCR，获取卡供电情况 **CMD58：**

在2.0版本上需要确认是SD2.0还是SD2.0HC卡，通过读取OCR进行判断：

```
//-----------鉴别SD2.0卡版本开始-----------
r1 = SD_SendCommand_NoDeassert(CMD58, 0, 0);
if(r1!=0x00)
    {
    Set_SD_CS;//释放SD片选信号
    return r1;  //如果命令没有返回正确应答，直接退出，返回应答
    }//读OCR指令发出后，紧接着是4字节的OCR信息
buff[0] = SPI1_ReadWriteByte(0xFF);
buff[1] = SPI1_ReadWriteByte(0xFF);
buff[2] = SPI1_ReadWriteByte(0xFF);
buff[3] = SPI1_ReadWriteByte(0xFF);
//OCR接收完成，片选置高
Set_SD_CS;
```

5、是否使用CRC CMD59，在此禁止CRC：

```
//禁止CRC校验
r1 = SD_SendCommand(CMD59, 0, 0x95);
if(r1 != 0x00)return r1;  //命令错误，返回r1
```

6、设置读、写块数据长度，512B **CMD16：**

```
//设置Sector Size
r1 = SD_SendCommand(CMD16, 512, 0x95);
if(r1 != 0x00)return r1;//命令错误，返回r1
//-----------------SD卡、MMC卡初始化结束-----------------
```

7、读取CSD，获取存储卡的其它参数信息，**CMD9：**

读取SD的容量，我们采用另外一个函数：

```
u8 SD_GetCSD(u8 *csd_data)
    {
    u8 r1;
    r1=SD_SendCommand(CMD9,0,0xFF);//发CMD9命令，读CSD
    if(r1)return r1;  //没返回正确应答，则退出，报错
    SD_ReceiveData(csd_data, 16, RELEASE);//接收16个字节的数据
    return 0;
    }
```

8、8个CLK后，禁止片选：

8个CLK时钟周期，刚好为一个字节宽度，因此，在此通过SPI1_ReadWriteByte函数作为8个CLK的延时。

```
//片选置高，结束本次命令
Set_SD_CS;
//多发8个CLK，让SD结束后续操作
SPI1_ReadWriteByte(0xFF);
```

以上的整个过程中都是按照初始化的步骤展开描述，将每个步骤与实际代码设计紧密结合，方便大家的学习了解。

接着我们了解一下SD卡的读单块数据的操作：1、主机发送CMD17命令；2、接收SD卡响应R1；3、接收读数据起始令牌0xFE；4、接收数据；5、接收2B CRC；6、8个CLK时钟延时，禁止片选。具体设计如下所示：

```
u8 SD_ReadSingleBlock(u32 sector, u8 *buffer)
    {
    u8 r1;

    //如果不是SDHC，给定的是sector地址，将其转换成byte地址
    if(SD_Type!=SD_TYPE_V2HC)
        {
        sector = sector<<9;
        }
    r1 = SD_SendCommand(CMD17, sector, 0);//读命令
    if(r1 != 0x00)return r1;
    r1 = SD_ReceiveData(buffer, 512, RELEASE);
    if(r1 != 0)return r1;    //读数据出错！
    else return 0;
    }
```

其中SD_ReceiveData函数的设计如下所示:

```
u8 SD_ReceiveData(u8 *data, u16 len, u8 release)
    {
    // 启动一次传输
    Clr_SD_CS;
    if(SD_GetResponse(0xFE))//等待SD卡发回数据起始令牌0xFE
        {
        Set_SD_CS;
        return 1;
        }
    while(len--)//开始接收数据
        {
        *data=SPI1_ReadWriteByte(0xFF);
        data++;
        }
    //下面是2个伪CRC（dummy CRC）
    SPI1_ReadWriteByte(0xFF);
    SPI1_ReadWriteByte(0xFF);
    if(release==RELEASE)//按需释放总线，将CS置高
        {
        Set_SD_CS;//传输结束
        SPI1_ReadWriteByte(0xFF);
        }
    return 0;
    }
```

下面看看写入单块数据的操作：1、主机发送CMD24命令；2、接收卡响应R1；3、接收读数据起始令牌0xFE；4、接收数据；5、接收2B CRC；6、8个CLK，禁止片选。代码如下：

```
u8 SD_WriteSingleBlock(u32 sector, const u8 *data)
    {
    u8 r1;
    u16 i,retry;
    //如果不是SDHC，给定的是sector地址，将其转换成byte地址
    if(SD_Type!=SD_TYPE_V2HC)
        {
        sector = sector<<9;
        }
    r1 = SD_SendCommand(CMD24, sector, 0x00);
    if(r1 != 0x00)
        {
        return r1;  //应答不正确，直接返回
        }
    //开始准备数据传输
    Clr_SD_CS;
    SPI1_ReadWriteByte(0xff);       //先放3个空数据，等待SD卡准备好
    SPI1_ReadWriteByte(0xff);
    SPI1_ReadWriteByte(0xff);
    SPI1_ReadWriteByte(0xFE);       //放起始令牌0xFE
    //放一个sector的数据
    for(i=0;i<512;i++)
        {
        SPI1_ReadWriteByte(*data++);
        }
    //发2个Byte的dummy CRC
    SPI1_ReadWriteByte(0xff);
    SPI1_ReadWriteByte(0xff);
    //等待SD卡应答
    r1 = SPI1_ReadWriteByte(0xff);
    if((r1&0x1F)!=0x05)
        {
        Set_SD_CS;
        return r1;
        }
    retry = 0;
    while(!SPI1_ReadWriteByte(0xff))  //等待操作完成
        {
        retry++;
        if(retry>0xfffe)          //如果长时间写入没有完成，报错退出
            {
            Set_SD_CS;
            return 1;             //写入超时返回1
            }
        }
    Set_SD_CS;                    //写入完成，片选置1  ,禁用
    SPI1_ReadWriteByte(0xff);     //8个CLK延时
    return 0;
    }
```

6.16.5 下载与测试

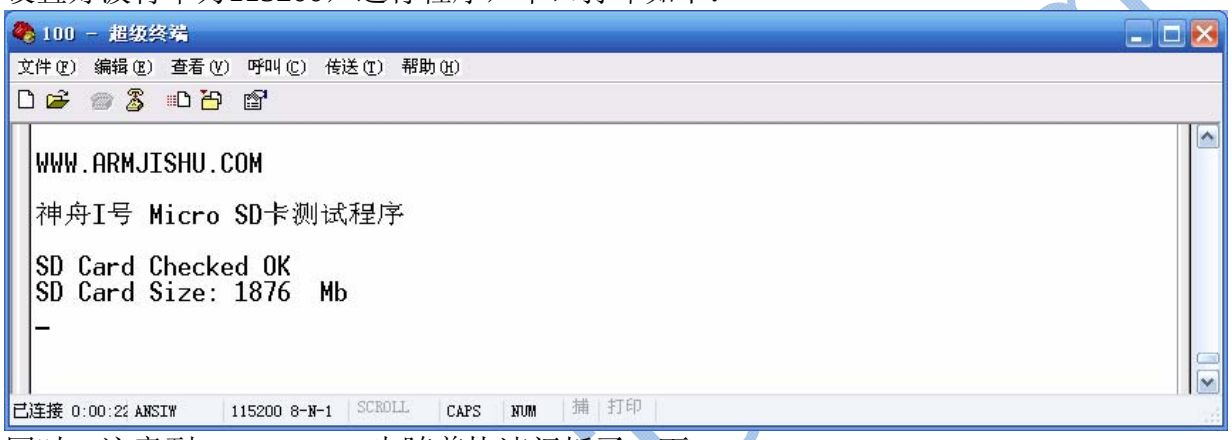
在 *神舟I号光盘\编译好的固件*目录下的 *"Micro SD卡实验.hex"* 文件即为前面我们分析*16. Micro SD卡实验（神舟I号）*的编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I号开发板中，观察运行效果。

如果使用JLINK下载固件，请按 *如何使用JLINK V8下载固件到神舟I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I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。

测试现象：将编译好的固件下载到神舟I号开发板上，插入SD卡，连接串口到PC上，设置好波特率为115200，运行程序，串口打印如下：



同时，注意到LED1~LED3也随着快速闪烁了一下。

www.armjishu.com

6.17 SD-USB读卡器实验

通过前两节关于USB以及SD卡的实验，大家对这两个设备已有了一定的认识，正如上一节所说的，这一节我们将在前两节的基础上，将它们进行整合，形成了STM32的SD-USB读卡器。

www.armjishu.com

6.17.1 实验的意义与作用

神舟I号开发板板载了SD卡座，而STM32F103又有USB，且在板上带有USB连接头，这样我们便可以通过STM32的USB来读写SD卡，从而实现一个USB读卡器。这样，通过此节的实验，大家对USB以及SD卡实验的学习加深了，同时也让自己多了一个SD卡读卡器而兴奋。

6.17.2 试验原理

USB读卡器的实现最重要的有两个部分：USB部分和SD卡部分。USB部分同上一节的差不多，只是这一节我们的STM32F103被识别成一个大容量存储设备。SD卡部分，最重要的就是2个函数，一个SD_WriteBlock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SD_ReadBlock函数，该函数用于读取SD卡上面的数据。

这里的数据并不需要经过文件系统处理，而是完全电脑控制，我们要做的就是读写 SD 卡就够了。本实验我们也是参考 Mass_Storage 例程而来的。SD 卡的读写可以采用 SDIO 方式，也可以采用 SPI，在上节我们也强调，神舟 I 号的 SD 卡采用 SPI 方式进行读写。

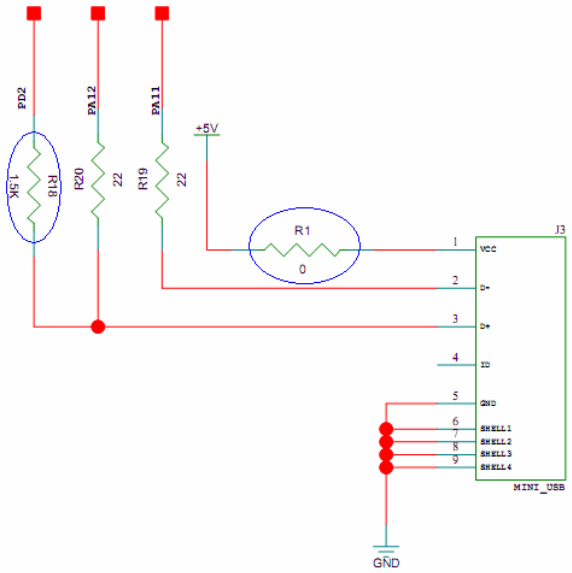
6.17.3 硬件设计

本节涉及到的硬件设计主要为两方面，一是USB；二是SD卡。前两节都单独描述过，在此再次申明硬件原理的设计。

USB部分：

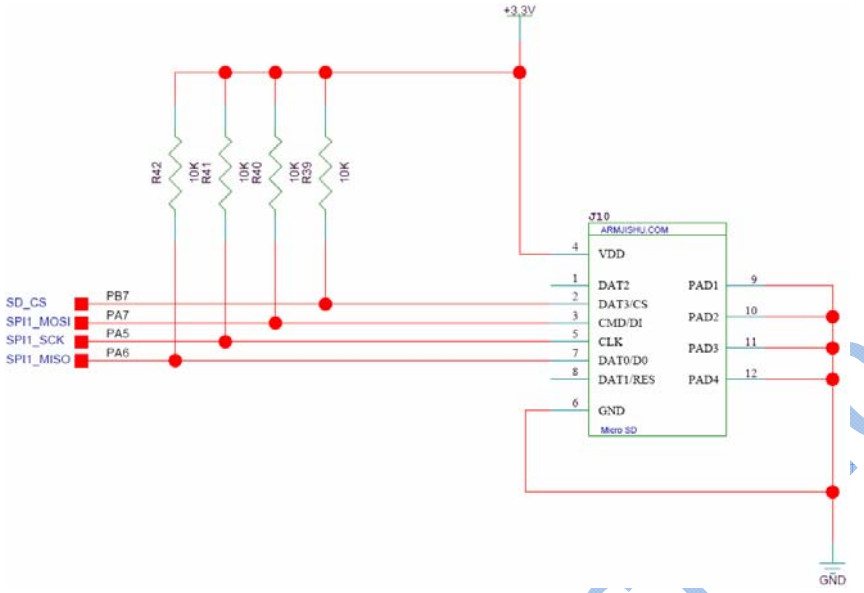
神舟I号的USB部分设计，主要提及两点：一是关于MINI-USB接口可以作为神舟I号整板的供电接口，接入5V电源；二是关于USBDP端的上拉，采用1.5K电阻可控上下拉。主要是用于接入USB线缆到PC时，处理器通过PD2对USBDP端进行上拉，PC通过此电平的变化，识别到USB设备。

另外采用PD2管脚可控1.5K的上下拉，**好处主要是：**可以完成程序的初始化后再打开USBDP的上拉，避免PC与USB设备相连后，PC要求USB设备马上响应USB总线上的信号，但此时USB设备可能正在初始化，导致USB设备与PC通讯失败。



SD卡部分：

神舟I号SD卡的硬件设计主要是采用SPI模式，使用处理器的SPI接口与SD卡相连接，要求SPI总线以及SD_CS必须上拉。同时，SD卡的工作电压要求为3.3V。另外SD的PAD都接地。



6.17.4 软件设计

本例程的设计思路为：在SD卡插入后，开始USB的配置，然后驱动PD2管脚，对USBDP进行上拉，让PC识别USB设备；在此SD-USB读卡器作为PC的一个从设备，PC对SD-USB读卡器的操作，实际也就是读写操作，主要涉及到两个函数：SD_WriteBlock函数和SD_ReadBlock函数（此两个文件位于stm32_eval_spi_sd.c文件中，并在mass_mal.c文件中进行调用，使用对SD卡的读写操作。）。

打开*神舟I号光盘\神舟I号源码\ STM32F10x_StdPeriph_Lib_V3.3.0.rar\ \Project\17.SD-USB读卡器实验（神舟I号）*目录。进入*17.SD-USB读卡器实验（神舟I号）\ MDK-ARM* 目录后，双击Project.uvproj可以打开工程，以下我们学习代码的设计（这些代码都是从ST提供的例程Mass_Storage里面移植过来的）。

关于USB的配置以及SD卡的部分驱动说明在前两节已经讲解过，在此就不重复描述。主要是关注SD的读写操作两个函数。其中此两个函数都处于ST库中的Mass_Storage实验中，我们在此不加入中文注释，大家请查阅文件中的英文注释即可。

首先是SD_WriteBlock写函数：实现PC通过USB对SD卡的写操作。

```
SD_Error SD_WriteBlock(uint8_t* pBuffer, uint32_t WriteAddr, uint16_t BlockSize)
{
  uint32_t i = 0;
  SD_Error rvalue = SD_RESPONSE_FAILURE;

  /*!< SD chip select low */
  SD_CS_LOW();
  /*!< Send CMD24 (SD_CMD_WRITE_SINGLE_BLOCK) to write multiple block */
  SD_SendCmd(SD_CMD_WRITE_SINGLE_BLOCK, WriteAddr, 0xFF);
  /*!< Check if the SD acknowledged the write block command: R1 response (0x00: no errors) */
  if (!SD_GetResponse(SD_RESPONSE_NO_ERROR))
  {
    /*!< Send a dummy byte */
    SD_WriteByte(SD_DUMMY_BYTE);
    /*!< Send the data token to signify the start of the data */
    SD_WriteByte(0xFE);
    /*!< Write the block data to SD : write count data by block */
    for (i = 0; i < BlockSize; i++)
    {
      /*!< Send the pointed byte */
      SD_WriteByte(*pBuffer);
      /*!< Point to the next location where the byte read will be saved */
      pBuffer++;
    }
    /*!< Put CRC bytes (not really needed by us, but required by SD) */
    SD_ReadByte();
    SD_ReadByte();
    /*!< Read data response */
    if (SD_GetDataResponse() == SD_DATA_OK)
    {
      rvalue = SD_RESPONSE_NO_ERROR;
    }
  }
  /*!< SD chip select high */
  SD_CS_HIGH();
  /*!< Send dummy byte: 8 Clock pulses of delay */
  SD_WriteByte(SD_DUMMY_BYTE);
  /*!< Returns the reponse */
  return rvalue;
}
```

其次是SD_ReadBlock读函数：实现PC通过USB对SD卡的读操作。

```
SD_Error SD_ReadBlock(uint8_t* pBuffer, uint32_t ReadAddr, uint16_t BlockSize)
{
  uint32_t i = 0;
  SD_Error rvalue = SD_RESPONSE_FAILURE;

  /*!< SD chip select low */
  SD_CS_LOW();
  /*!< Send CMD17 (SD_CMD_READ_SINGLE_BLOCK) to read one block */
  SD_SendCmd(SD_CMD_READ_SINGLE_BLOCK, ReadAddr, 0xFF);
  /*!< Check if the SD acknowledged the read block command: R1 response (0x00: no errors) */
  if (!SD_GetResponse(SD_RESPONSE_NO_ERROR))
  {
    /*!< Now look for the data token to signify the start of the data */
    if (!SD_GetResponse(SD_START_DATA_SINGLE_BLOCK_READ))
    {
      /*!< Read the SD block data : read NumByteToRead data */
      for (i = 0; i < BlockSize; i++)
      {
        /*!< Save the received data */
        *pBuffer = SD_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 SD) */
      SD_ReadByte();
      SD_ReadByte();
      /*!< Set response value to success */
      rvalue = SD_RESPONSE_NO_ERROR;
    }
  }
  /*!< SD chip select high */
  SD_CS_HIGH();

  /*!< Send dummy byte: 8 Clock pulses of delay */
  SD_WriteByte(SD_DUMMY_BYTE);
  /*!< Returns the reponse */
  return rvalue;
}
```

SD-USB读卡器的主函数主要是针对外设设备、特别是是USB的相关设置：如时钟、中断和初始化等。

```
int main(void)
{
  Set_System();            //初始化一些外设等
  Set_USBClock();
  Led_Config();
  USB_Interrupts_Config();
  USB_Init();
  while (bDeviceState != CONFIGURED);

  USB_Configured_LED();

  printf("\n\r###### WWW.ARMJISHU.COM! ###### SD Card Reader is Ready.....");

  while (1)
  {}
}
```

6.17.5 下载与测试

在 *神舟I号光盘\编译好的固件*目录下的 *“SD-USB读卡器.hex”*文件即为前面我们分析 *17.SD-USB读卡器实验（神舟I号）*的编译好的固件，我们可以直接通过JLINK V8将固件下载到神舟I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I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I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。

测试现象：在代码编译成功之后，我们通过下载代码到神舟I号开发板上，在USB配置成功并有SD卡接入的时候，打开我的电脑，显示USB读卡器盘符，如下所示：

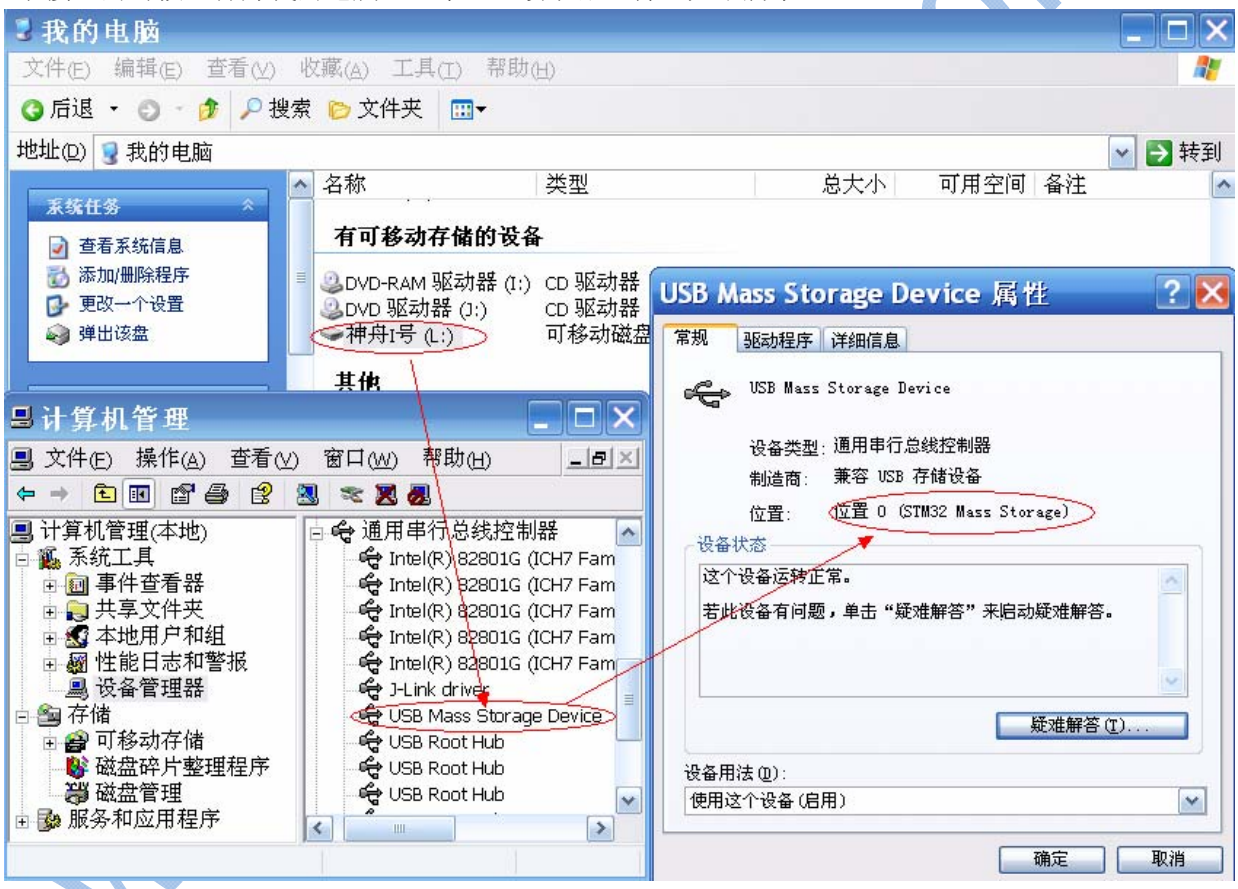


图3.27.4.3可移动存储设备

我们试着在移动磁盘里面拷贝数据出来或者写入数据进去，此过程可以观察LED1和LED3的闪烁情况。

6.18 uCOS_UCGUI_DEMO实验

www.armjishu.com

第7章 实验现象

神舟I号一共提供近20个示例程序，本章节详描述各个程序的现象和操作说明。

序号	试验名称	操作步骤与试验现象
1	LED 流水灯试验	1.当程序运行时，左下角 LED1~LED3 这三个 LED 指示灯轮流闪亮。
2	KEY_LED 按键与 315M 无线	1.当程序运行时，蜂鸣器周期性的鸣响，同时 DS1 也按同样的周期点亮和熄灭
3	USART-COM1 串口发送	1.使用配套的串口线连接电脑与神舟 I 号的串口 1 2.打开电脑的超级终端，选择对应的串口，并按如下参数设置“波特率：115200，数据位：8，奇偶校验：无，停止位：无，数据流控制：无” 3.电脑上超级终端周期性的接收到“神舟 I 号 串口 1 测试程序”字符。
4	USART-COM1 串口接收与发送	1.使用配套的串口线与连接电脑与神舟 I 号的串口 1 2.打开电脑的超级终端，选择对应的串口，并按如下参数设置“波特率：115200，数据位：8，奇偶校验：无，停止位：无，数据流控制：无” 3.电脑上超级终端显示串口已配置信息。 4.此时键盘上按任意字母，回车确认后，神舟 I 号通过超级终端打印回显之前按键按下的字符数据。
5.	ADC 模数转换	通过串口打印 ADC 可调电阻上的电压值，调整可调电阻（RM1）的值，可以看到串口打印信息上的 AD 转换结果、百分比以及电压值变化。
6	EEPROM 读写程序	首先程序将 0x00~0xFF 写入 EEPROM 对应的 0x00~0xFF 地址，再读出。 串口显示从 0x00~0xFF 读出的值。
7	SPI Flash（W25X16）读写程序	1、在提示“按钮 1：write；按钮 2：read”，按下按钮 1，显示“W25X16 Write Finished！” 2、接着按下按钮 2，串口则打印预先存储的字符组“神舟 I 号 SPI 读写访问程序”。
8	实时时钟实验	如果采用了电池供电，若没有配置过时间，则串口提示配置时间参数；反之串口显示时间。（按提示输入时间数字） 如果没有采用电池供电，请将 J9 跳帽接上，使用 3.3V 供电，但无法保存时间，每一次重启都需要配置时间
9	独立看门狗	1、正常时 LED1 亮着，隔一定时间没有喂狗，LED1 就会闪烁，表示没有喂狗复位； 2、如果在有效时间内喂狗的话，也就是按下按钮 1 时，LED1~LED3 全亮；按下按钮 2 时，LED1~LED3 全灭。但同时都喂狗了。（串口没有信息打印）
10	SysTick 系统滴答	3 个 LED 指示灯不停闪烁（500ms 亮，500ms 灭）

www.armjishu.com

11	彩屏显示实验	刷新各种颜色，并带有“www.armjishu.com”字符
12	TFT 触摸屏显示加触摸实验	1、LCD屏上不停的刷新图片，且都带有www.armjishu.com； 2、串口打印 LCD 驱动芯片型号 3、支持触摸操作。
13	18B20 温度传感实验	1、设置串口为 115200 的波特率，装上 18B20 温度传感器，串口打印“当前温度为：xxx”。 2、如果没有装上 18B20，串口则出现“ds18b20 err”。
14	2.4G 无线通信实验	1. 使用两个神舟 I 号板件，接上两个 NRF24L01 后，根据按钮 1 和按钮 2 选择发送或是接收。两个板件的串口一个打印，“正在发送数据”，另外一个显示“接收到的数据为”，当其中一个失控时，有相关提示确认。 2. 如果只有一块板件，且装上 24L01 时，按钮 1，“等待发送数据”，按钮 2 时“请确认接收端是否正常” 3. 波特率为 9600
15	USB 遥控鼠标实验	1. 目前的现象只有一个：连接USB后，可以控制鼠标匀速左移；（后续丰富其操作，请关注www.armjishu.com网站的更新）
16	SD 卡实验	1. 连接串口，插入 SD 卡，串口显示 SD 卡的容量，并显示 section0 的数据 2. 波特率为 115200；
17	SD-USB 读卡器实验	插入 SD 后，连上 USB 后，可当 U 盘使用
18	uCos+uCGUI 程序	功能齐全，详细请实际操作

附件 1：JLINK V8 用户手册

附件 2：Jlink 转接板简介

附件 3：《TCP/IP 协议栈 LwIP 的设计与实现》

附件 4：项目合作与技术支持联系方式

可以发送邮件：armjishu.com@163.com 或 QQ：7819226 联系人：Jesse

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】