



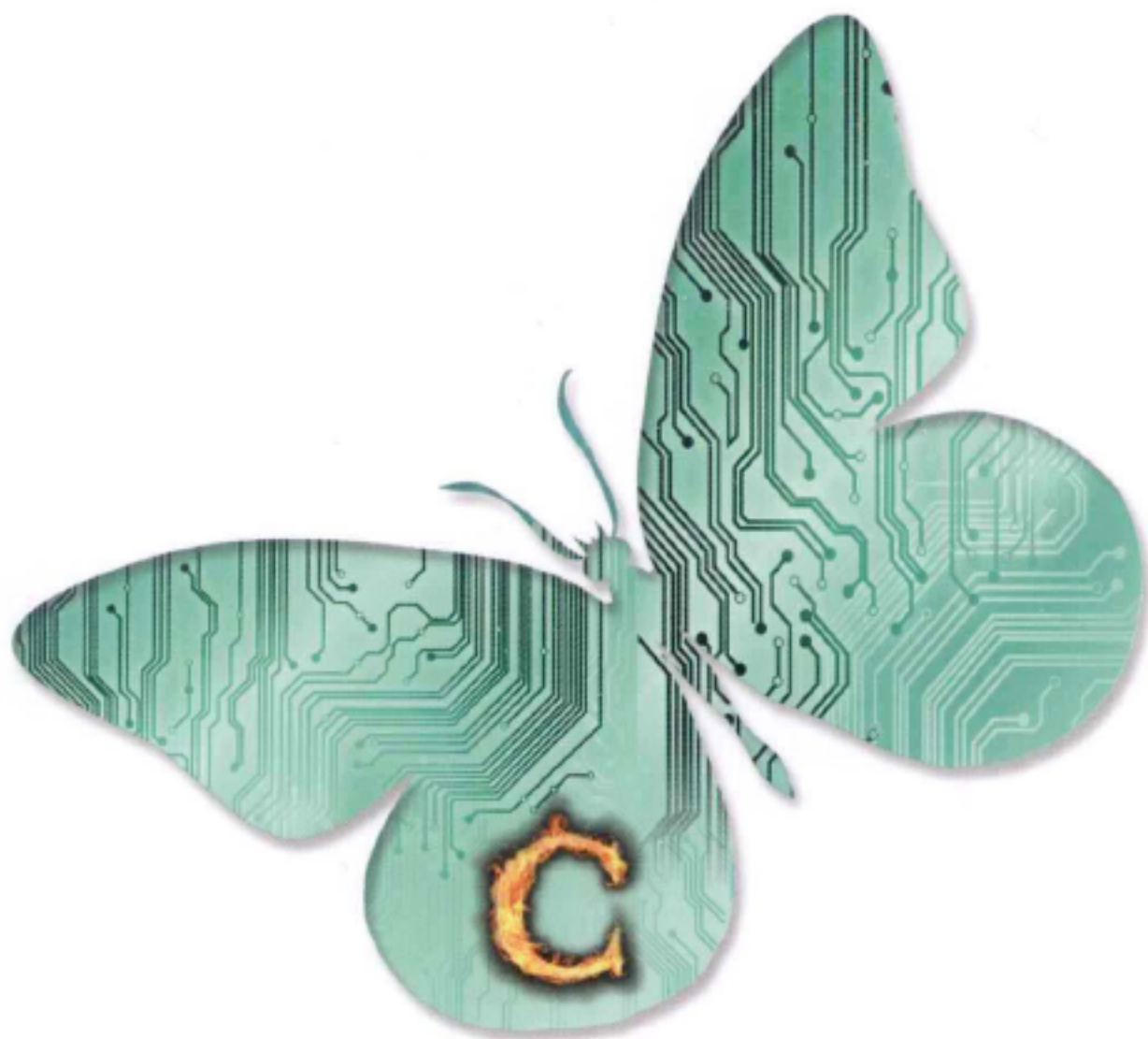
第一本深入剖析STM32官方库及其使用的权威指南

从流水灯剖析到 μ C/OS-III移植，零死角深入STM32库开发

配套业内最流行的野火STM32开发板，提供完整的工程文件和源代码，极具可操作性



单片机与嵌入式



库

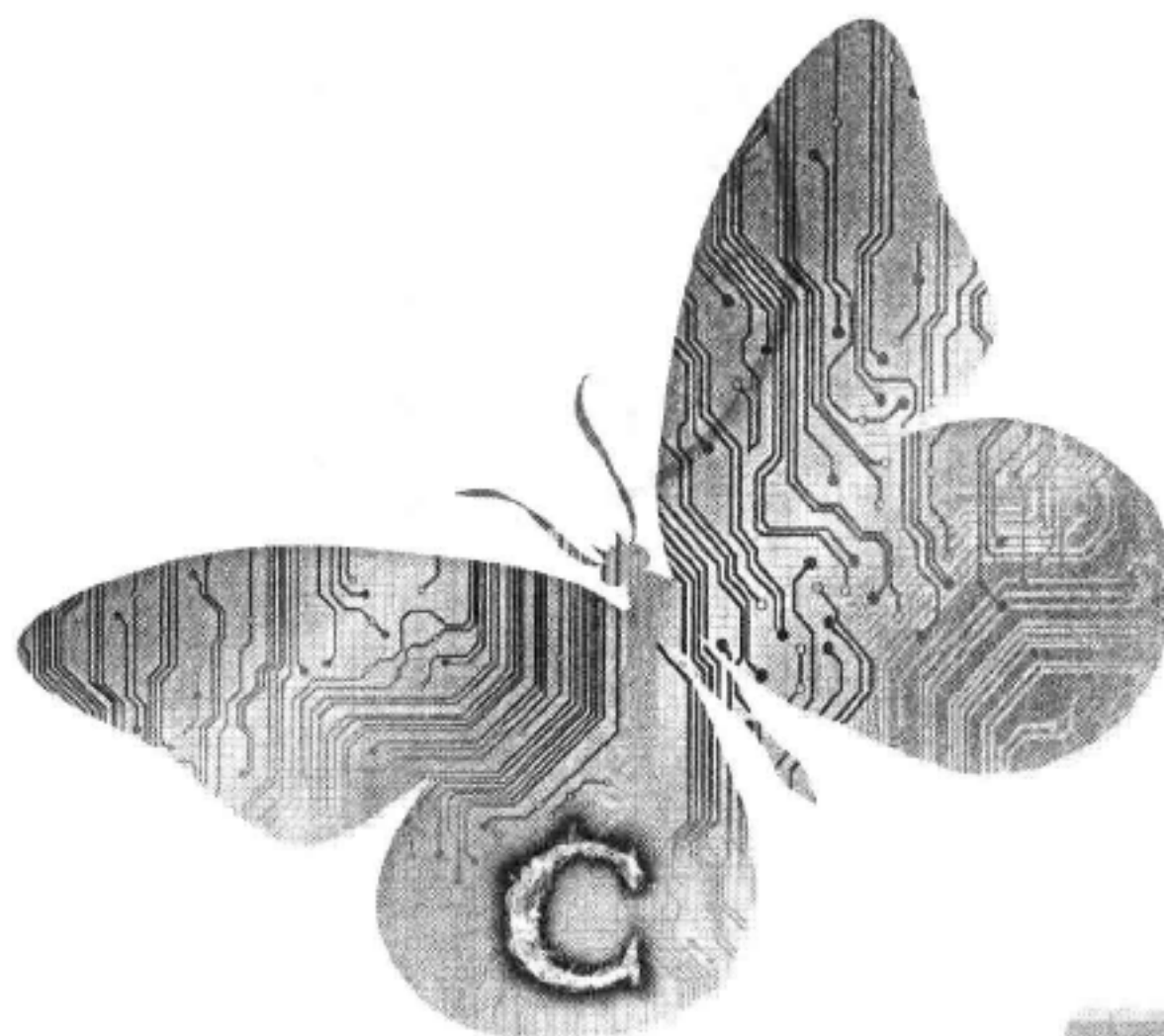
STM32库 开发实战指南

刘火良 杨森 编著



机械工业出版社
China Machine Press

单片机与嵌入式



库

STM32库 开发实战指南

刘火良 杨森 编著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

STM32库开发实战指南 / 刘火良, 杨森编著. —北京: 机械工业出版社, 2013.5

ISBN 978-7-111-42637-0

I. S… II. ①刘… ②杨… III. 微控制器 - 系统开发 - 指南 IV. TP332.3-62

中国版本图书馆CIP数据核字 (2013) 第109248号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书基于STM32F103芯片, 紧紧围绕“库”的分析和使用展开。在大量实例的基础上, 本书对于如何综合运用固件库开发项目给出了具体的范例; 在固件库的使用和学习的基础上, 又进一步讲解了结合嵌入式实时操作系统、TCP/IP协议栈进行嵌入式系统开发的方法, 让读者循序渐进、系统地掌握基于STM32官方库进行开发的方法。

本书内容翔实, 案例丰富, 操作性极强, 可作为高校电子信息、通信工程、信息工程等相关专业的教材, 也适合作为从事嵌入式领域科技工作者的参考书。

机械工业出版社 (北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 余洁

北京市荣盛彩色印刷有限公司印刷

2013年6月第1版第1次印刷

186mm × 240mm · 31印张

标准书号: ISBN 978-7-111-42637-0

ISBN 978-7-89433-959-1 (光盘)

定 价: 69.00元 (附光盘)

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzsj@hzbook.com

前 言

单片机是对 8/16 位 MCU（微控制器）的另外一种叫法。传统的 8/16 位单片机，久经岁月的洗礼，仍然在工业控制应用中大放光芒。然而，现在的工程师面对的更多的工业控制产品需求是多功能、易用界面、低功耗以及多任务等。基于这样的需求，以往的 8/16 位单片机已不能满足要求，工程师必须寻找新的符合要求的 MCU。工程师虽然可以选择诸如 ARM7、ARM9 这类速度更快的 32 位 MCU，但是鉴于对成本和开发门槛等种种考虑，它们还是不能满足需求。正是看准了这个市场，ARM 公司推出了其全新的基于 ARMv7 架构的 32 位 Cortex-M3 微控制器内核。紧随其后，ST（意法半导体）公司就推出了基于 Cortex-M3 内核的 MCU——STM32。STM32 凭借其产品线的多样化、极高的性价比、简单易用的开发方式，迅速在众多 Cortex-M3 MCU 中脱颖而出，成为最闪亮的一颗新星。STM32 一上市就迅速占领了中低端 MCU 市场，颇有星火燎原之势，这与它倡导的基于固件库的开发方式密不可分。采用库开发的方式可以快速上手，仅通过调用库里面的 API（应用程序接口）就可以迅速搭建一个大型的程序，写出各种用户所需的应用，这就大大降低了学习的门槛和开发周期。然而，又因为在开发中只是调用 API，而忽略了底层寄存器的操作，库开发被习惯了寄存器开发方式的工程师指为“于浮沙筑高台”，没有学习的价值。这种看法具有一定的片面性，他显然没有意识到这是一种全新的学习方法。试问，对于初学者，面对一个 32 位且有如此多寄存器的单片机，如果还像我们以往操作 8 位机，通过配置寄存器的方式来实现，那会是多么繁杂的一项工作？除此之外，库的开发方式自顶向下，它是迈向更高端嵌入式 Linux 开发的一个垫脚石。

库开发已成主流，这是不争的事实。STM32 固件库之所以流行并被大家所喜爱，可以归结为以下两个原因：

（1）技术潮流

1) 于个人：库开发大大地降低了学习的门槛，提高了学习的效率，使个人初步了解了大的程序设计，是一种自顶向下的学习方法，可以从上层的 API 层层跟踪到底层，可以彻彻底底地了解寄存器，了解 CPU 的内存分布，再到启动代码、开发环境的配置等。如果再深究，还会涉及编译器甚至工具链。库的学习，可不仅仅是简单地调用 API，我们需要去分析这个库是如何构建的，是如何从内存到寄存器，寄存器到结构体，结构体到更各层的 API，再到层层外设的文件关

联。这里面涉及了太多的 C 语言的知识,如关键字、宏、结构体、指针、类型转换、条件编译、断言、内联函数等。这些知识的学习,又岂能说是“于浮沙筑高台”。如果你的 C 语言还停留在基本语法的阶段,那么通过对库的使用和学习,你的 C 语言将会得到脱胎换骨的提升。学会了库开发还可以快速地迁移到 ST 其他系列单片机的学习,如 STM32F207、STM32F407,这些 MCU 的固件库基本上都是兼容的。反观如此庞大的固件库,还要相互兼容,细心的人一定可以从中获益良多!

2) 于公司:在公司产品开发中,产品上市速度是非常重要的成功因素,库开发可以极大地缩短产品研发周期,以便快速抢占市场。而且库让程序的维护成本更低,程序的升级更快捷。用库来开发,真可谓事半功倍,一箭双雕。

(2) 市场趋势

有人曾经质疑 STM32 的固件库降低了 MCU 的性能,然而,他却没有考虑到 STM32 的性能和资源已经不是传统的 8/16 单片机可比的了。强大的硬件应该与消耗这些资源的软件相匹配,否则资源就被浪费了。硬件和软件是相辅相成、共同促进的。所以硬件改善后,工程师对于 MCU 的关注应该从全局着眼。

本书采用 MDK 开发环境,全部例程基于 3.5.0 版本的固件库讲解,不是简单地调用库,而是试图通过对固件库的使用详细讲解什么是库、为什么使用库、怎样使用库等一系列问题,进而引导读者使用高效率的库开发方法。

本书采用原理分析、代码讲解、实验运用这三点连线的讲解方式,循序渐进,适合在校大学生和科研机构开发人员学习使用。全书分为四个部分,第一部分(第 1~5 章)是库开发初级篇,涉及入门的两个主题。一个是嵌入式工程师成长之路,属于方法论的问题,涉及了一个工程师从学生时代开始,在每一个不同的阶段应该学习什么、该如何进阶等。另一个是通过对库的了解和 GPIO 的学习,让读者快速掌握 STM32 的开发方法,这是入门的第一步。第二部分(第 6~16 章)是库开发中级篇,讲解了 STM32 各个外设的使用,是学习的一个进阶阶段,也是 STM32 学习的重中之重。第三部分(第 17~25 章)是库开发高级篇,是 STM32 各个外设的实战演练,如 MP3、液晶、摄像头、Wi-Fi 等,是属于项目实战的例子,一般可直接用于工程项目的开发。第四部分(第 26~28 章)是库开发系统篇,这是嵌入式系统开发的必经之路,是区别裸与不裸的分水岭;这部分讲解了 μ C/OS 最新版本 μ C/OS-III 在 STM32 中的移植,通过该移植应用实践,相信可以为以后进阶到 WinCE、Linux 操作系统的学习打下坚实的基础。

致 谢

首先要感谢本书的策划编辑张国强先生,是他对 STM32 的关注促成了这本书的出版,同时在我们撰写书稿时对本书提出了宝贵的写作建议,并对书稿进行了仔细审阅。其次要感谢野火

工作室的成员廖锦松、曾云清等人提供的部分例程及资料；感谢好友何卓波对书稿内容的校正工作。

由于本书涉及的知识面广，时间又仓促，限于笔者的水平和经验，疏漏之处在所难免，恳请专家和读者批评指正，可以发送邮件到 wildfireteam@163.com 与作者进行交流，或者到阿莫论坛野火 M3 专区 <http://www.amobbs.com> 进行讨论。

刘火良 杨森

目 录

前 言

第一部分 库开发初级篇

第 1 章 为什么学习 STM32 2

- 1.1 嵌入式技术知识结构..... 2
- 1.2 嵌入式工程师成长之路..... 3
- 1.3 为什么学习 STM32..... 4
- 1.4 如何学习 STM32..... 4

第 2 章 初识 STM32 固件库..... 5

- 2.1 STM32 神器之库开发..... 5
 - 2.1.1 什么是 STM32 库..... 5
 - 2.1.2 为什么采用库开发..... 6
- 2.2 STM32 结构及库层次关系 7
 - 2.2.1 CMSIS 标准 7
 - 2.2.2 库目录、文件简介..... 8
 - 2.2.3 STM32 固件库文件间的关系 14
 - 2.2.4 使用库帮助文档..... 15

第 3 章 GPIO 入门之流水灯..... 18

- 3.1 安装 MDK..... 18
- 3.2 建立工程模板 19
 - 3.2.1 新建工程 19
 - 3.2.2 配置 J-LINK 硬件调试 25

3.3 如何编译和下载程序..... 27

- 3.3.1 如何编译程序..... 27
- 3.3.2 如何下载程序..... 27

第 4 章 深入分析流水灯例程 30

- 4.1 STM32 的 GPIO..... 30
- 4.2 STM32 的地址映射..... 33
 - 4.2.1 温故而知新
——stm32f10x.h 文件..... 33
 - 4.2.2 外设基地址 35
 - 4.2.3 总线外设基地址..... 36
 - 4.2.4 寄存器组基地址..... 37
- 4.3 STM32 固件库对寄存器的封装 ... 38
- 4.4 STM32 的时钟系统..... 39
 - 4.4.1 时钟树 & 时钟源..... 39
 - 4.4.2 高速外部时钟..... 41
 - 4.4.3 HCLK、FCLK、PCLK1、
PCLK2..... 42
- 4.5 LED 具体代码分析 42
 - 4.5.1 实验描述及工程文件清单 42
 - 4.5.2 配置工程环境..... 43
 - 4.5.3 编写用户文件..... 44
 - 4.5.4 初始化结构体
——GPIO_InitTypeDef 类型 ... 46
 - 4.5.5 初始化库函数——GPIO_Init()..... 47

4.5.6 开启外设时钟	48
4.5.7 控制 I/O 输出高、低电平	52
4.5.8 led.h 文件	52
4.5.9 main 文件	53
4.6 GPIO_Init() 函数的实现	55
4.6.1 规范的位操作方法	55
4.6.2 GPIO_Init() 实现代码分析	55
4.6.3 再论开发方式	60
4.7 开发步骤总结	61

第 5 章 调试程序 62

5.1 MDK 软件仿真调试	62
5.2 使用 J-LINK 进行硬件调试	64
5.2.1 硬件调试	64
5.2.2 软件编译过程	65
5.3 MDK 使用小技巧	66

第二部分 库开发中级篇

第 6 章 GPIO 再举例之

按键实验 70

6.1 GPIO 的 8 种工作模式	70
6.1.1 4 种输入模式	71
6.1.2 4 种输出模式	71
6.2 按键实验分析	72
6.3 按键代码分析	72
6.3.1 实验描述及工程文件清单	72
6.3.2 配置工程环境	73
6.3.3 main 文件	73
6.3.4 GPIO 初始化配置	74
6.3.5 利用固件库的数据类型	75
6.3.6 实现 LED 反转	77
6.3.7 实验现象	77

第 7 章 EXTI 之按键中断实验 78

7.1 STM32 的中断和异常	78
7.2 NVIC 中断控制器	81
7.2.1 NVIC 结构体成员	81
7.2.2 抢占优先级和响应优先级	82
7.2.3 NVIC 的优先级组	83
7.3 EXTI 外部中断	83
7.4 中断检测按键实验分析	84
7.4.1 实验描述及工程文件清单	84
7.4.2 配置工程环境	85
7.4.3 main 文件	86
7.4.4 配置外部中断	86
7.4.5 AFIO 时钟	87
7.4.6 NVIC 初始化配置	88
7.4.7 EXTI 初始化配置	89
7.4.8 编写中断服务函数	89
7.4.9 实验现象	91

第 8 章 串口通信 (USART) 92

8.1 异步串口通信协议	92
8.2 直通线和交叉线	93
8.3 串口工作过程分析	94
8.3.1 波特率控制	94
8.3.2 收发控制	96
8.3.3 数据存储转移	96
8.4 串口通信实验分析	96
8.4.1 实验描述及工程文件清单	96
8.4.2 配置工程环境	97
8.4.3 main 文件	97
8.4.4 USART 初始化配置	98
8.4.5 printf() 函数重定向	101
8.4.6 USART1_printf() 函数	103
8.4.7 实验现象	106

第 9 章 库函数开发小结 107

- 9.1 初始化 107
- 9.2 数据输入输出 108
- 9.3 状态位、标志位 108
 - 9.3.1 事件 109
 - 9.3.2 标志位的检查与清除 109
- 9.4 外设函数分类 110

第 10 章 DMA——为 CPU 减负... 112

- 10.1 DMA 功能简介 112
- 10.2 DMA 工作分析 112
- 10.3 DMA 实例之串口通信 113
 - 10.3.1 实验描述及工程文件清单 113
 - 10.3.2 配置工程环境 114
 - 10.3.3 main 文件 114
 - 10.3.4 DMA 初始化 115
 - 10.3.5 使用 DMA 中断 121
 - 10.3.6 实验现象 123

**第 11 章 ADC 实验
(DMA 方式) 124**

- 11.1 ADC 简介 124
- 11.2 STM32 的 ADC 主要技术指标 124
- 11.3 ADC 工作过程分析 125
- 11.4 ADC 采集数据实例
(采用 DMA 模式) 126
 - 11.4.1 实验描述及工程文件清单 127
 - 11.4.2 配置工程环境 128
 - 11.4.3 main 文件 128
 - 11.4.4 ADC 初始化 129
 - 11.4.5 计算电压值 138
 - 11.4.6 实验现象 138

**第 12 章 SysTick (系统滴
答定时器) 139**

- 12.1 SysTick——操作系统的心跳 139
- 12.2 SysTick 工作分析 140
- 12.3 使用 SysTick 精确延时
实验分析 141
 - 12.3.1 实验描述及工程文件清单 142
 - 12.3.2 配置工程环境 142
 - 12.3.3 main 文件 143
 - 12.3.4 配置并启动 SysTick 143
 - 12.3.5 定时时间的计算 147
 - 12.3.6 编写中断服务函数 147
 - 12.3.7 使用 SysTick 测量时间的
功能 149
 - 12.3.8 实验现象 149

第 13 章 STM32 定时器 150

- 13.1 定时器功能简介 150
- 13.2 定时器工作分析 150
 - 13.2.1 基本定时器 150
 - 13.2.2 通用定时器 150
 - 13.2.3 高级定时器 155
- 13.3 PWM 输出实例分析 157
 - 13.3.1 实验描述及工程文件清单 157
 - 13.3.2 配置工程环境 157
 - 13.3.3 main 文件 158
 - 13.3.4 定时器初始化 159
 - 13.3.5 实验现象 164

第 14 章 I²C 接口 168

- 14.1 I²C 协议简介 168
 - 14.1.1 物理层 168

14.1.2 协议层	169
14.2 STM32 的 I ² C 特性及架构	170
14.2.1 I ² C 接口特性	170
14.2.2 I ² C 架构	170
14.3 I ² C 接口读写 EEPROM 实验	171
14.3.1 实验描述及工程文件清单	171
14.3.2 配置工程环境	171
14.3.3 main 文件	172
14.3.4 I ² C 接口初始化	173
14.3.5 对 EEPROM 的读写操作	177
14.3.6 使用 I ² C 读写 EEPROM	
流程总结	186
14.3.7 实验现象	186

第 15 章 SPI 模块

15.1 SPI 协议简介	188
15.1.1 SPI 信号线	188
15.1.2 SPI 模式	189
15.2 STM32 的 SPI 特性及架构	190
15.2.1 STM32 的 SPI 特性	190
15.2.2 STM32 的 SPI 架构分析	190
15.3 SPI 接口读取 Flash 实例分析	191
15.3.1 实验描述及工程文件清单	192
15.3.2 配置工程环境	193
15.3.3 main 文件	193
15.3.4 SPI 初始化	195
15.3.5 控制 Flash 的命令	199
15.3.6 读取厂商 ID	202
15.3.7 擦除 Flash 内容	203
15.3.8 向 Flash 写入数据	207
15.3.9 从 Flash 读取数据	210
15.3.10 小结	211
15.3.11 实验现象	211

第 16 章 CAN 控制器

16.1 CAN 协议简介	212
16.1.1 物理层	212
16.1.2 CAN 的报文种类及结构	213
16.1.3 同步	215
16.2 STM32 的 CAN 特性及架构	217
16.2.1 CAN 特性	217
16.2.2 CAN 架构	218
16.3 双 CAN 通信实验分析	219
16.3.1 实验描述及工程文件清单	219
16.3.2 配置工程环境	220
16.3.3 main 文件	221
16.3.4 配置 CAN 接口	223
16.3.5 打包报文	232
16.3.6 发送报文	234
16.3.7 接收报文、编写中断	
服务函数	234
16.3.8 实验小结	236
16.3.9 实验现象	237

第三部分 库开发高级篇

第 17 章 SDIO 之 SD 卡驱动

17.1 SD 协议简介	240
17.1.1 卡的种类	240
17.1.2 SDIO 基本架构	241
17.2 STM32 的 SDIO 接口	241
17.2.1 从 SDIO 的时钟说起	242
17.2.2 SDIO 的命令格式	242
17.2.3 数据传输格式	243
17.3 SD 卡读写实验分析	243
17.3.1 实验描述及工程文件清单	243
17.3.2 配置工程环境	244

17.3.3	main 文件	246
17.3.4	SDIO 初始化	247
17.3.5	卡的上电识别流程	249
17.3.6	卡的初始化流程	256
17.3.7	对 SD 卡进行读写	259
17.3.8	原版官方驱动例程的 bug	263
17.3.9	实验现象	264

第 18 章 文件系统之 FATFS_R0.09 265

18.1	什么是文件系统	265
18.2	FATFS 文件系统简介	266
18.2.1	FATFS 的目录结构	266
18.2.2	FATFS 帮助文档	266
18.2.3	FATFS 源码	267
18.3	移植 FATFS 文件系统实验	267
18.3.1	实验描述及工程文件清单	267
18.3.2	配置工程环境	269
18.3.3	为文件系统添加底层驱动	270
18.3.4	添加简体中文和 长文件名支持	274
18.3.5	main 文件	274
18.3.6	实验现象	277

第 19 章 MP3 播放器 278

19.1	MP3 文件探秘	278
19.1.1	文件格式	278
19.1.2	MP3 文件的原始数据	278
19.1.3	MP3 文件格式	279
19.2	VS1003 硬件解码芯片	279
19.2.1	VS1003 芯片简介	280
19.2.2	TDA1308 芯片	280
19.3	MP3 播放器实验	280
19.3.1	实验描述及工程文件清单	280

19.3.2	配置工程环境	282
19.3.3	main 文件	283
19.3.4	控制 VS1003 进入准备状态	284
19.3.5	播放 MP3 文件	286
19.3.6	STM32 的堆栈	291
19.3.7	实验现象	294

第 20 章 USB 大容量 存储器实例 295

20.1	USB 协议分析	295
20.1.1	协议版本	295
20.1.2	USB 电气特性	295
20.1.3	USB 通信模型	296
20.1.4	USB 枚举	298
20.2	STM32 的 USB 控制器	299
20.3	USB 读取 SD 卡 ——模拟 U 盘实验	301
20.3.1	实验描述及工程文件清单	301
20.3.2	配置工程环境	302
20.3.3	USB 固件库说明	303
20.3.4	main 文件	305
20.3.5	基本配置	306
20.3.6	USB 初始化	308
20.3.7	中断服务函数	310
20.3.8	BOT 和 SCSI 协议	313
20.3.9	实验现象	316

第 21 章 LCD 触摸屏画板 317

21.1	LCD 控制器简介	317
21.1.1	ILI9341 控制器结构	317
21.1.2	像素点的数据格式	317
21.1.3	ILI9341 的通信时序	319
21.2	用 STM32 驱动 LCD	320
21.2.1	FSMC 简介	320

21.2.2	用 FSMC 模拟 8080 时序	322
21.3	触摸屏感应原理	322
21.4	TSC2046 触摸屏控制器	323
21.5	LCD 触摸屏画板实验	323
21.5.1	实验描述及工程文件清单	323
21.5.2	配置工程环境	325
21.5.3	main 文件	326
21.5.4	初始化 FSMC 模式	327
21.5.5	FSMC 模拟 8080 读写 参数、命令	332
21.5.6	液晶屏画点函数	334
21.5.7	触摸屏校正	338
21.5.8	检测触点、画点	341
21.5.9	实验现象	342

第 22 章 字库及 BMP 图片 显示 343

22.1	什么是字模	343
22.2	制作字模	344
22.3	BMP 图片格式	347
22.4	显示中英文及 BMP 图片实验	351
22.4.1	实验描述及工程文件清单	351
22.4.2	配置工程环境	352
22.4.3	main 文件	352
22.4.4	显示汉字	353
22.4.5	在 SD 卡上读取与保存 BMP 图像	358
22.4.6	实验现象	364

第 23 章 OV7670 摄像头驱动 365

23.1	摄像头的分类	365
23.1.1	数字摄像头与模拟摄像头的 区别	365
23.1.2	CCD 与 CMOS 的区别	365

23.2	OV7670 介绍	366
23.2.1	OV7670 功能框架	366
23.2.2	OV7670 管脚封装	367
23.3	SCCB 总线	368
23.3.1	SCCB 接口定义	368
23.3.2	SCCB 时序描述	370
23.4	摄像头模块	372
23.4.1	摄像头模块硬件介绍	372
23.4.2	OV7670 输出时序	372
23.4.3	FIFO 时序	375
23.4.4	摄像头的驱动原理	376
23.5	摄像头驱动实验	377
23.5.1	实验描述及工程文件清单	377
23.5.2	配置工程环境	379
23.5.3	main 文件	379
23.5.4	SCCB 总线的软件实现	380
23.5.5	初始化 OV7670	386
23.5.6	采集并显示图像	388
23.5.7	实验现象	393

第 24 章 以太网及 LwIP 协议栈 移植 394

24.1	互联网模型	394
24.2	以太网	395
24.2.1	PHY 层	395
24.2.2	MAC 子层	396
24.2.3	以太网控制器	397
24.3	MAC 之上的网络层	398
24.3.1	为什么在 MAC 之上 还有分层	398
24.3.2	TCP/IP 协议中各层次的功能	398
24.3.3	LwIP 协议栈	400
24.4	ENC28J60+LwIP 以太网实验	401

24.4.1	实验描述及工程文件清单	401
24.4.2	配置工程环境	402
24.4.3	main 文件	403
24.4.4	LwIP 对底层数据结构的封装	404
24.4.5	初始化协议栈	408
24.4.6	LwIP 对底层操作的封装	410
24.4.7	轮询和计时	415
24.4.8	opt.h 文件和 debug	416
24.4.9	LwIP 应用	420
24.4.10	网页服务器	421
24.4.11	实验现象	426

第 25 章 Wi-Fi 模块 EMW3180 驱动 430

25.1	资料与工具下载	430
25.2	EMW3180 简介	430
25.3	EMW3180 驱动实验	434
25.3.1	实验描述及工程文件清单	434
25.3.2	配置工程环境	435
25.3.3	EMSP_API 函数	435
25.3.4	API 函数一览	436
25.3.5	main 文件	439
25.3.6	em380c_hal.c 文件	441
25.3.7	实验现象	445

第四部分 库开发系统篇

第 26 章 μ C/OS-III 及其源代码介绍 448

26.1	μ C/OS 简介	448
------	---------------	-----

26.1.1	操作系统与裸机的区别	448
26.1.2	μ C/OS 实时操作系统	448
26.2	μ C/OS-III 与 μ C/OS-II 的主要区别	450
26.3	μ C/OS-III 源码	450
26.4	μ C/OS-III 工程架构	452

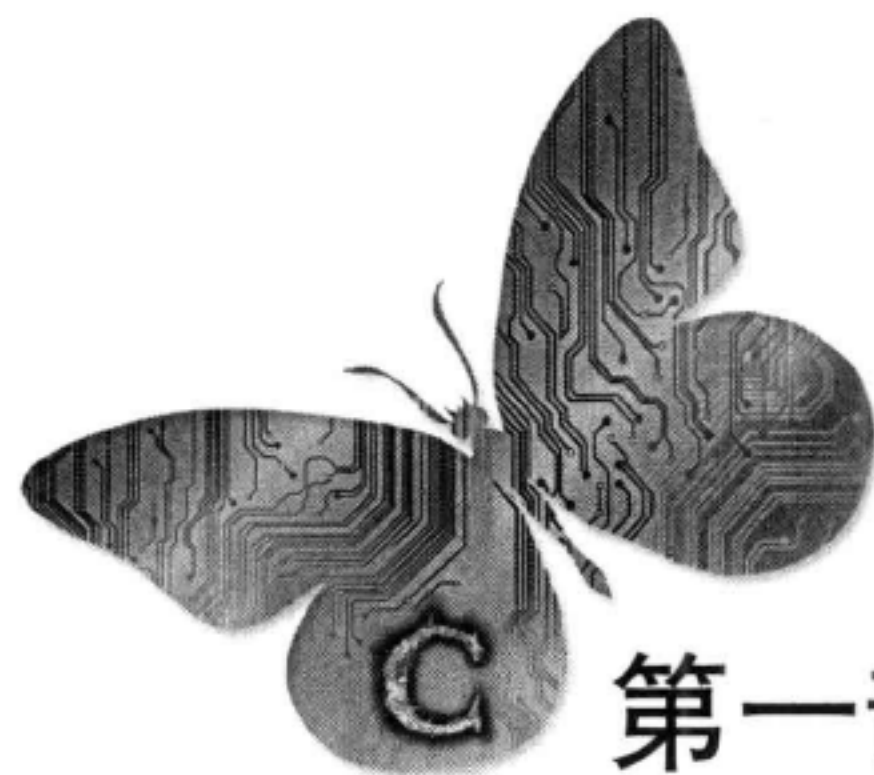
第 27 章 移植 μ C/OS-III 到 STM32 454

27.1	搭建 μ C/OS 工程文件结构	454
27.2	修改 μ C/OS 代码	459
27.2.1	修改 os_cpu.h 文件	459
27.2.2	修改 os_cpu_c.c	459
27.2.3	修改 os_cpu_a.asm 文件	460
27.2.4	修改 cpu_a.asm 文件	461
27.2.5	修改 startup_stm32f10x_hd.s 文件	462
27.2.6	修改 stm32f10x_it.c 文件	463
27.3	编写用户文件	464
27.3.1	编写 includes.h 文件	464
27.3.2	编写 BSP 相关文件	465
27.3.3	创建任务	466
27.4	配置 μ C/OS-III	468

第 28 章 运行多任务 473

28.1	创建用户任务	473
28.2	编写用户代码	476
28.3	任务执行流程	479

参考文献 482



第一部分 库开发初级篇

大多数技术书籍会告诉读者如何掌握某一门技术，但少有讨论与职业规划相关的问题，例如应该学什么以及为什么学。在本书的第一部分我们将与大家共同探讨这些问题，引导读者思考为什么学习 STM32。

初级篇可以帮助初学者快速上手 STM32，以点亮 LED 灯的实例，从软件工程的角度深入剖析什么是固件库、为什么使用固件库和怎样使用固件库；从 STM32 固件库、新建工程、编译和下载程序出发，了解如何操作 GPIO，让新手步步为营，尽享 STM32 的学习乐趣。

我们对初学者的要求是具有基本的单片机基础，如 51、AVR 等，曾使用 C 语言写过单片机程序，但不需精通。读者在学习 STM32 的时候，无需太担心自己的基础，学习这门技术的决心和持之以恒的耐心才是掌握这门技术的法宝。

- 第 1 章 为什么学习 STM32
- 第 2 章 初识 STM32 固件库
- 第 3 章 GPIO 入门之流水灯
- 第 4 章 深入分析流水灯例程
- 第 5 章 调试程序



第 1 章

为什么学习 STM32

本章系统介绍了嵌入式工程师的技术成长路线，并详细介绍技术路线中的岗位设置和知识结构，让读者对于嵌入式工程师有一个全面系统地了解，并在此基础上引导工程师规划自己职业生涯。在本章的最后，作为承上启下的内容，从为什么学习 STM32 和如何学习 STM32 这两个话题入手，引导读者开始对于 STM32 的库开发方式有个初步地理解。

1.1 嵌入式技术知识结构

嵌入式技术是专用计算机系统技术，它以应用为核心，以计算机技术为基础，软硬件均可裁剪，适用在对功能、稳定性、功耗有严格要求的系统之中。嵌入式技术的开发人员需要对整个计算机体系（从底层硬件到软件操作系统）都有了解，而在这个体系之中，每个部分都可以分出一些小领域，因而技术要求很高，见图 1-1。

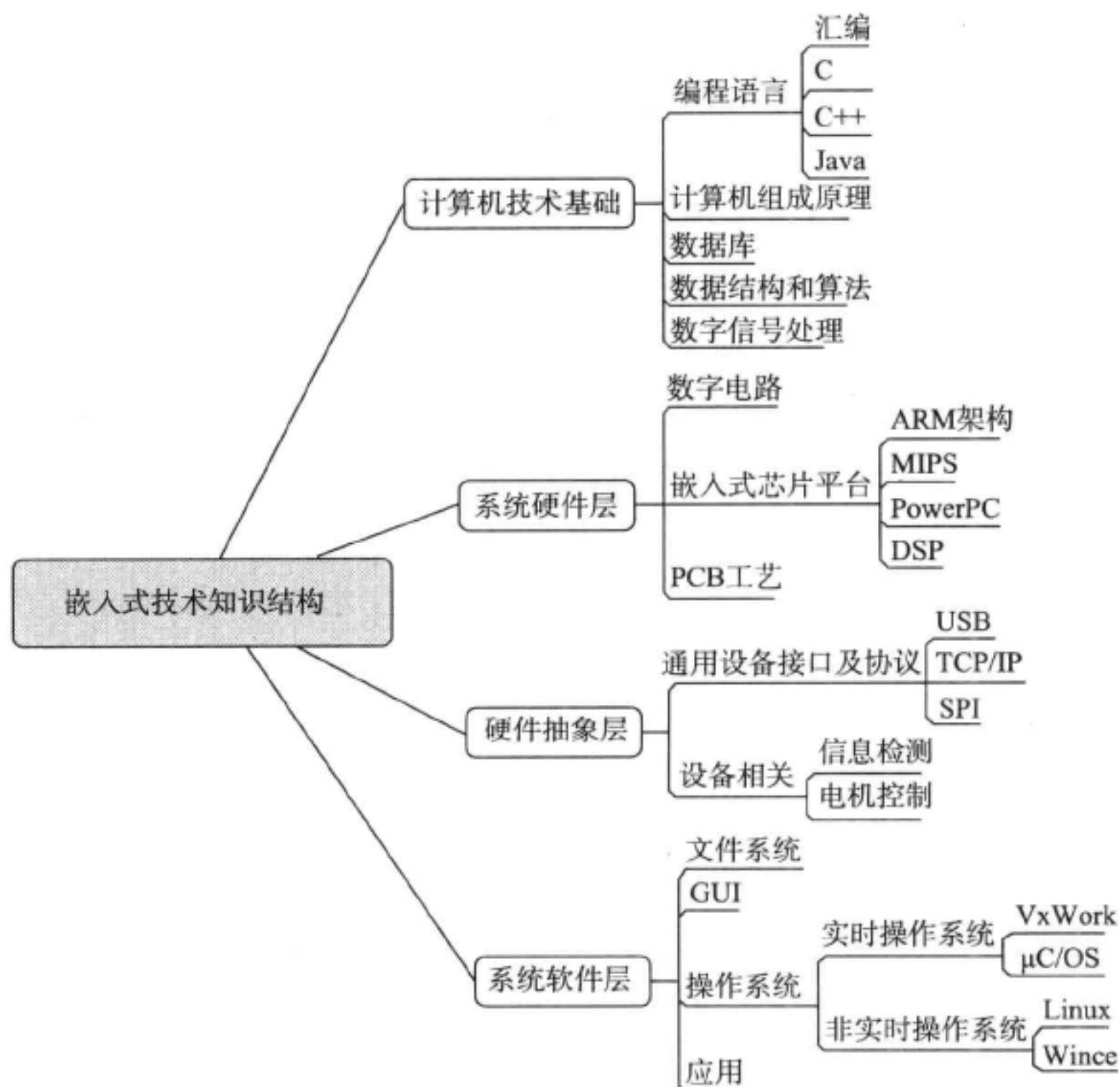


图 1-1 嵌入式技术知识结构

这个图只是粗略地概括了嵌入式技术的知识结构，但从中已经可以看出它涉及的知识面非常广，难怪众多学生甚至技术人员总是“迷茫”。不少电子专业出身的嵌入式技术人员主要从事硬件抽象层（中间层）的开发，这一层是沟通嵌入式系统的硬件层和软件操作系统的桥梁，因而主要的工作是开发驱动程序、板级应用支持、协调软硬件的开发，因而对软硬件都要有深入的了解。

1.2 嵌入式工程师成长之路

1. 从学生成为工程师

若希望从事硬件抽象层的开发，应该如何学习这些知识，才能从学生过渡到工程师呢？见图 1-2，对于希望成为其他方向的嵌入式技术人员也可以参考。

从图 1-2 可以看出，越往上层深入，就越接近于纯软件开发，但这并不代表嵌入式技术人员就不需要了解硬件，相反，上层的知识都是以底层为基础的，很多人说的“做嵌入式软件开发至少要读懂原理图”就是这个道理。

2. 职业规划

在嵌入式技术领域的公司，除了工程师还分很多职业岗位。一般公司的研发部门职位见图 1-3。

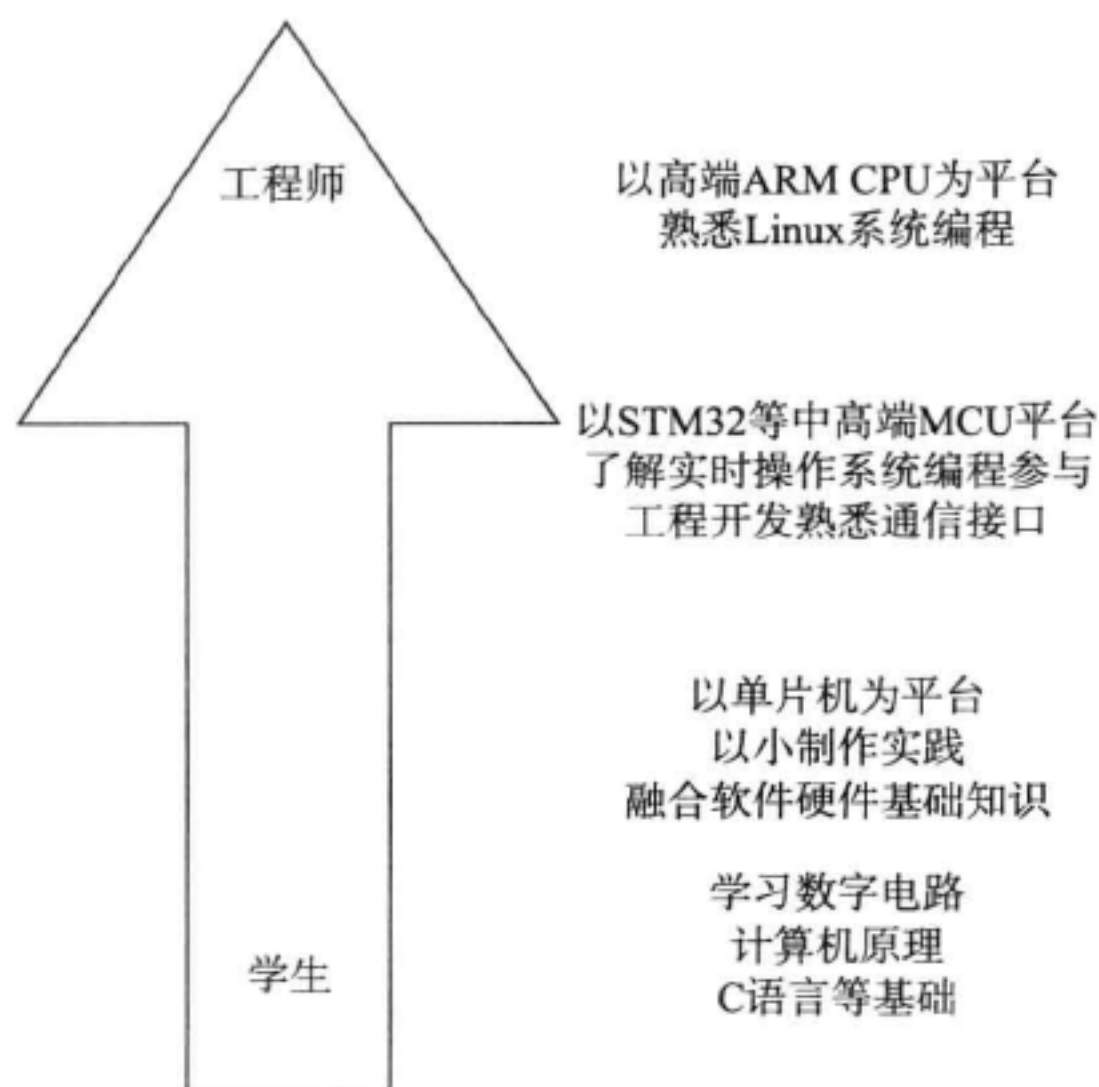


图 1-2 从事硬件抽象层开发的工程师成长之路

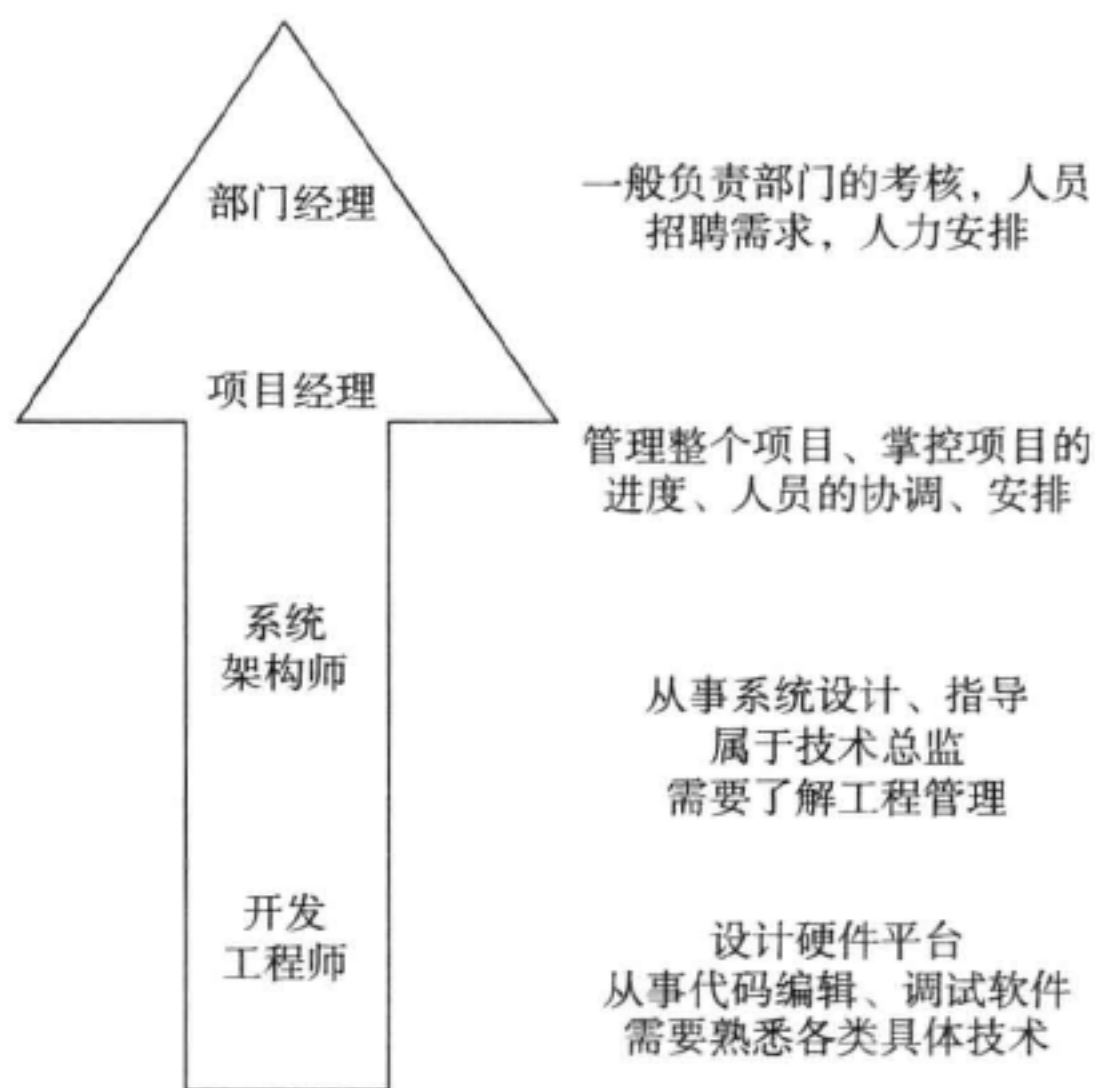


图 1-3 嵌入式工程师的职业成长路线

一般需要 3 ~ 5 年过渡到下一级的岗位，在小公司里项目经理一般也兼任部门经理。部门经理不一定要懂技术，并不是非由项目经理升职而成。直接与技术相关的是开发工程师和系统架构师，开发工程师会针对嵌入式技术的不同领域有不同的区分。在小公司里，熟悉软硬件的跨领域工程师很受欢迎，而大公司则分工明确，更看重在某领域研究得深入的开发工程师。作为系统架构师，则需要熟悉整个嵌入式领域，能够协调不同领域的开发工程师进行项目开发。

对于职业规划，不同的人有不同的见解，情况千差万别，以上所述仅供读者参考。

1.3 为什么学习 STM32

可以发现，在嵌入式领域 STM32 芯片介于低端和高端之间，它相对于普通的 8/16 位机有更多的片上外设，更先进的内核架构，可以运行 $\mu\text{C}/\text{OS}$ 等实时操作系统；相对于可运行 Linux 操作系统的高端 CPU，其成本低，实时性强。这个定位使得 STM32 不仅占领了大部分中端控制器的市场，更是成为提升开发者技术的优良过渡平台，为后续的学习打下坚实的基础。

1.4 如何学习 STM32

因为 STM32 的开发方式较普通的单片机开发还是有很大的不同，所以学习时要注意以下几点：

- 1) 转变思维，适应使用固件库的开发方式，加强运用 C 语言的能力，建立工程意识。
- 2) 熟悉 Cortex-M 系列芯片架构，了解 CMSIS 标准，熟悉 STM32 的总线架构。
- 3) 掌握 I²C、SPI、SDIO、CAN、TCP/IP 等各种通信协议，掌握了这些协议，开发软件驱动就变得相对容易了。

上面有关的内容本书都会详细讲解，但“纸上得来终觉浅，绝知此事要躬行”，读者亲自编程实践是不能少的。

初级篇可以帮助初学者快速上手 STM32，写出自己的应用程序。以点亮 LED 灯的实例，从软件工程的角度深入剖析什么是固件库、为什么使用固件库和怎样使用固件库；从 STM32 固件库、新建工程、编译和下载程序出发，了解如何操作 GPIO，让新手步步为营，尽享 STM32 的学习乐趣。

我们对初学者的要求是具有基本的单片机基础，如 51、AVR 等，曾使用 C 语言写过单片机程序，但不需精通。读者在学习 STM32 的时候，无需太担心自己的基础，我们更需要的是学习的勇气，需要的是拿下 STM32 的决心。试问，我们刚开始学习最简单的单片机的时候，是不是也没基础呢，是不是因此就停止了自己学习的脚步了呢？不是的。我们需要做的是认定一个目标，行动起来，坚持朝向目标的苦行，其中艰辛芳华，唯你自知。



第 2 章

初识 STM32 固件库

本章通过简单介绍 STM32 固件库的各个文件以及文件之间的关系，让读者建立 STM32 固件库的概念，看完后对固件库有个总体印象即可，在后期实际开发时接触了具体固件库时，再回头看看本章，相信你对 STM32 固件库又会有一个更深刻的认识。

2.1 STM32 神器之库开发

2.1.1 什么是 STM32 库

在 51 单片机的程序开发中，我们直接配置 51 单片机的寄存器，控制芯片的工作方式，如中断、定时器等。配置的时候，我们常常要查阅寄存器表，看用到哪些配置位，为了配置某功能该置 1 还是置 0。这些都是很琐碎的、机械的工作，因为 51 单片机的软件相对来说较简单，而且资源很有限，所以可以通过直接配置寄存器的方式来开发。

STM32 库是由 ST 公司针对 STM32 提供的函数接口，即 API (Application Program Interface)，开发者可调用这些函数接口来配置 STM32 的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速、易于阅读、维护成本低等优点。

当我们调用库的 API 时可以不用挖空心思去了解库底层的寄存器操作，就像当年我们学习 C 语言，用 `printf()` 函数时只是学习它的使用格式，并没有去研究它的源码实现，如非必要，可以说是老死不相往来。

实际上，库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口。库开发方式与直接配置寄存器方式的区别见图 2-1。

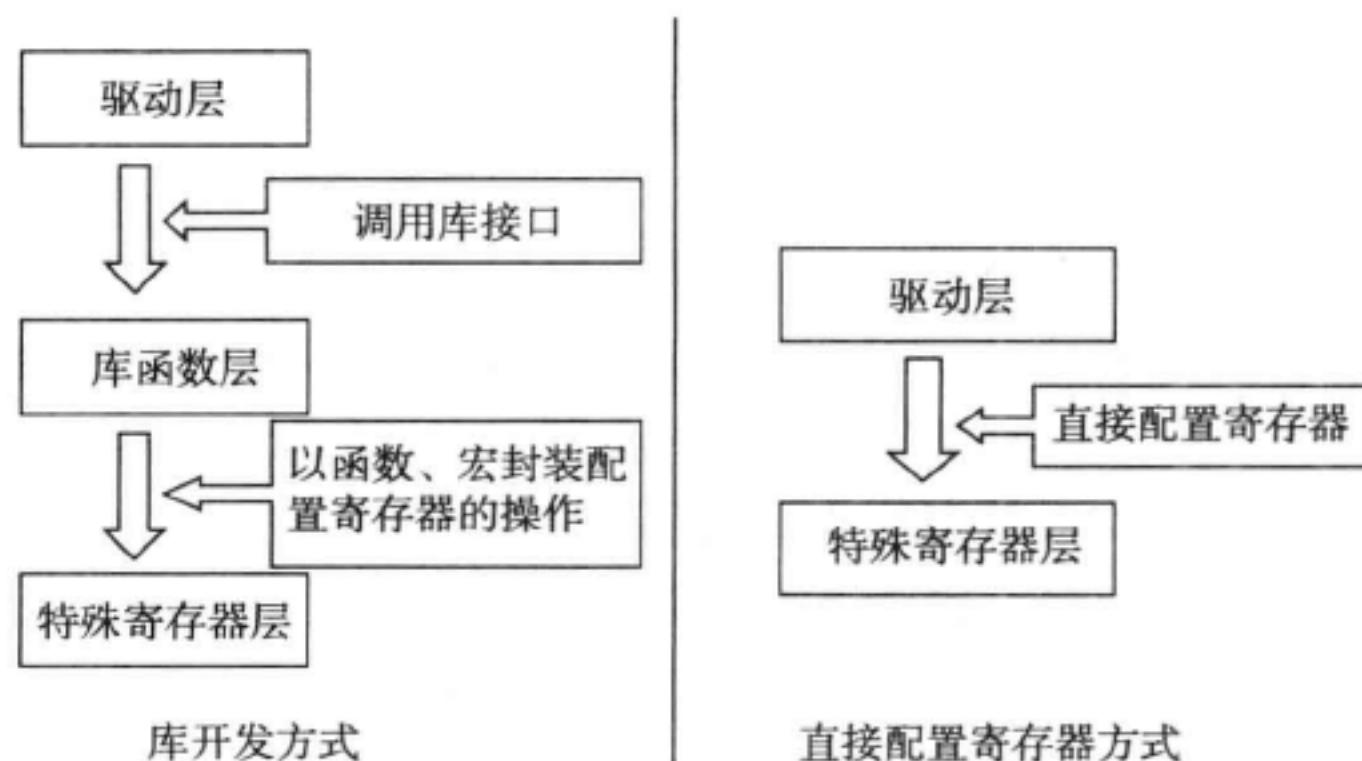


图 2-1 开发方式对比图

2.1.2 为什么采用库开发

对于 STM32，因为外设资源丰富，带来的必然是寄存器的数量和复杂度的增加，这时直接配置寄存器方式的缺陷就突显出来了：

- 1) 开发速度慢。
- 2) 程序可读性差。

这两个缺陷直接影响了开发效率、程序维护成本和交流成本。库开发方式则正好弥补了这两个缺陷。

而坚持采用直接配置寄存器方式开发的程序员，会列举以下原因：

- 1) 更直观。
- 2) 程序运行占用资源少。

初学 STM32 的读者，普遍因为第一个原因而选择以直接配置寄存器的方法来学习。认为这种方法直观，能够了解到是配置了哪些寄存器以及怎样配置寄存器。事实上，库函数的底层实现恰恰是直接配置寄存器方式的最佳例子，想深入了解芯片是如何工作的话，只要追踪到库的最底层实现就能理解，相信你会为它严谨、优美的实现方式而陶醉。要想修炼 C 语言，就从 STM32 官方库开始吧。我们将在第 4 章对 STM32 官方库进行详细分析。

相对于库开发的方式，直接配置寄存器方式生成的代码量的确会少一点，但因为 STM32 有充足的资源，权衡库的优势与不足，绝大部分时候我们愿意牺牲一点资源，选择库开发更划算。一般只有在对代码运行时间要求极其苛刻的地方，才用直接配置寄存器的方式代替，如频繁调用的中断服务函数。

对于库开发与直接配置寄存器的方式，在 STM32 刚推出时就引起程序员的激烈争论，但是，随着 STM32 官方库的完善与大家对库的了解，更多的程序员选择了库开发的方式。

本书采用 STM32 固件库进行讲解，既介绍如何使用库接口，也讲解库接口的实现方式。使读者既能利用库进行快速开发，也能深入了解 STM32 的工作原理。

为进一步解答读者为什么使用库开发，请读者先思考一下为什么采用 C 语言开发软件而不是采用汇编。相比之下，可以发现调用库接口开发与直接配置寄存器开发的关系，犹如 C 语言与汇编的关系。见表 2-1 和表 2-2。

据某 IT 大师说过，虽然无从考证，但他（她）说得很有道理：“一切计算机科学的问题都可以用分层来解决。”从汇编到 C，从直接配置寄存器到使用库，从裸机到系统，从操作系统到应用层软件，无不体现着这样的分层思想。开发的软件多了，跨越的软件层次多了，会深刻地认同他这句话，分层思想在软件开发上体现得淋漓尽致，分层使得问题变得更简单，使得能够屏蔽底层实现方式的差异，使得软件开发变成简单的调用函数接口，而不用管它的实现，大大提高效率。

库的本质就是建立了一个新的软件抽象层，库的优点，其实就是分层的优点，库的缺点，也是软件分层带来的，而对于 STM32 这样高性能的芯片，承受分层带来的痛苦相比获得的优势是值得的。

表 2-1 C 语言与库开发方式类比

特 点	C 语 言	库开发方式
更接近人的思维 (易读)	程序控制语句结构化。以函数作为 程序单位便于模块化	用结构体封装寄存器参数；用宏表示参数，意义明确； 用函数封装对寄存器的操作
移植性好	程序基本上不做修改就可应用于各 种计算机上	代码的易读性使驱动代码的修改变得非常方便

表 2-2 汇编语言与直接配置寄存器方式类比

特 点	汇 编 语 言	直接配置寄存器方式
更接近机器思维 (直观)	汇编指令为机器码的助记符，能直 接了解 CPU 的操作	直接针对寄存器的某些位进行置 1 或清 0 操作，能清 晰地看到驱动代码使用了什么寄存器
运行效率高	代码为 CPU 直接执行的指令，与编 译器优化无关	没有库函数层，省去代码为分层而消耗的资源

2.2 STM32 结构及库层次关系

2.2.1 CMSIS 标准

我们知道由 ST 公司生产的 STM32 采用的是 Cortex-M3 内核，内核是整个微控制器的 CPU。该内核是 ARM 公司设计的一个处理器体系架构，ARM 公司并不生产芯片，而是出售其芯片技术授权。ST 公司或其他芯片生产厂商如 TI，负责设计的是在内核之外的部件，被称为核外外设或片上外设、设备外设。如芯片内部的模数转换外设 ADC、串口 UART、定时器 TIM 等。

内核与外设，类似 PC 上的 CPU 与主板、内存、显卡、硬盘的关系，见图 2-2。

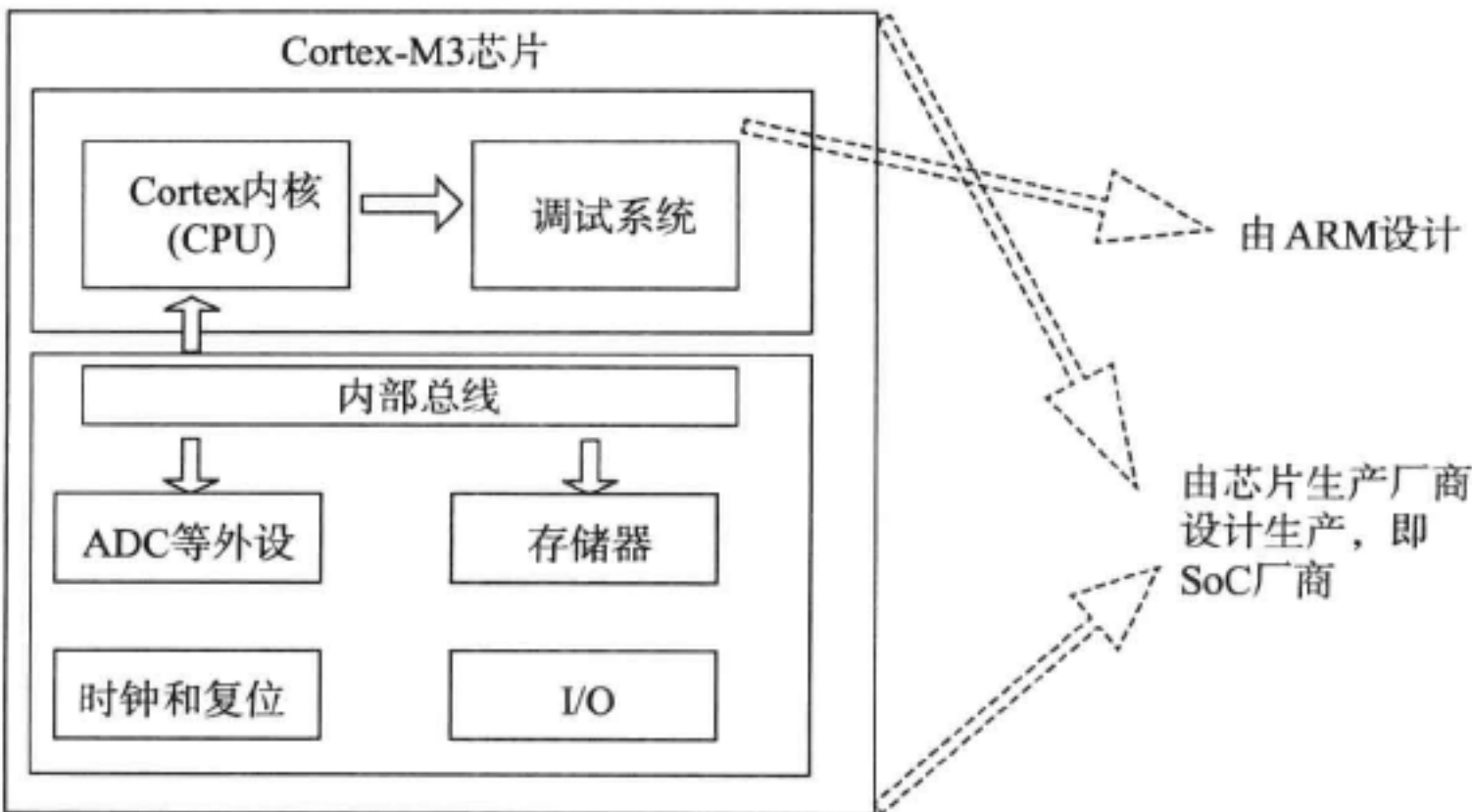


图 2-2 内核与外设的关系

因为基于 Cortex 的某系列芯片采用的内核都是相同的, 区别主要为核外的片上外设的差异, 这些差异却导致软件在同内核、不同外设的芯片上移植困难。为了解决不同芯片厂商生产的 Cortex 微控制器软件的兼容性问题, ARM 与芯片厂商建立了 CMSIS 标准 (Cortex Microcontroller Software Interface Standard)。

所谓 CMSIS 标准, 实际是新建了一个软件抽象层, 见图 2-3。

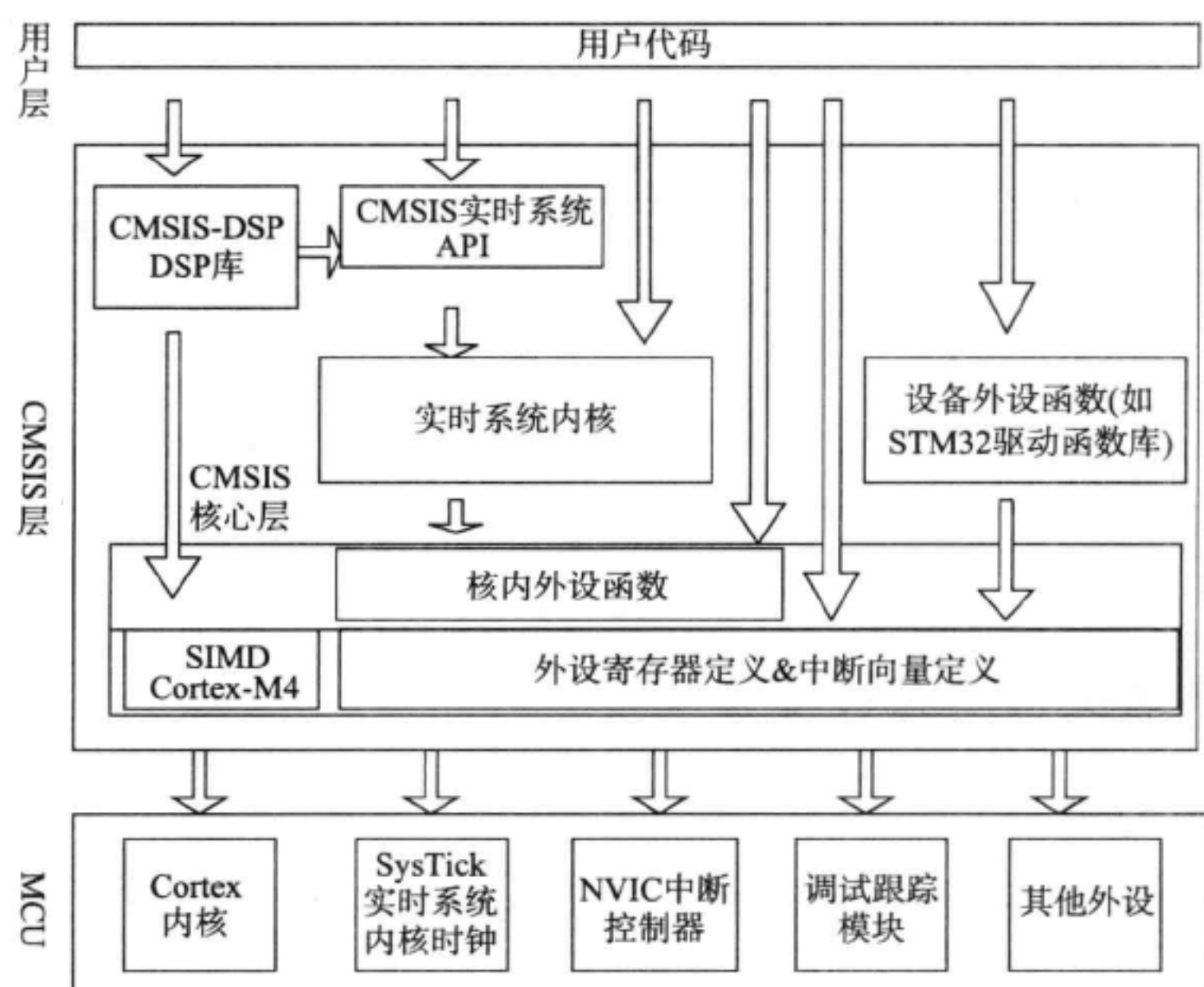


图 2-3 CMSIS 架构

CMSIS 标准中最主要的是 CMSIS 核心层, 它包括:

- 内核函数层: 其中包含用于访问内核寄存器的名称、地址定义, 主要由 ARM 公司提供。
- 设备外设访问层: 提供了片上的核外外设的地址和中断定义, 主要由芯片生产商提供。

可见 CMSIS 层位于硬件层与操作系统或用户层之间, 提供了与芯片生产商无关的硬件抽象层, 可以为接口外设、实时操作系统提供简单的处理器软件接口, 屏蔽了硬件差异, 这对软件的移植有极大的好处。STM32 固件库就是按照 CMSIS 标准建立的。

2.2.2 库目录、文件简介

STM32 的 3.5 版库可以从官网获得, 也可以直接从本书的随书光盘得到。本书讲解的例程全部采用最新的 3.5 版库文件。因为 3.5 版与 3.0 版的库文件兼容性很好, 光盘中附带的其他例程仍然保留了一些使用 3.0 版的代码。

解压库文件后进入其目录:

```
stm32f10x_stdperiph_lib\STM32F10x_StdPeriph_Lib_V3.5.0
```

各文件夹内容说明见图 2-4。

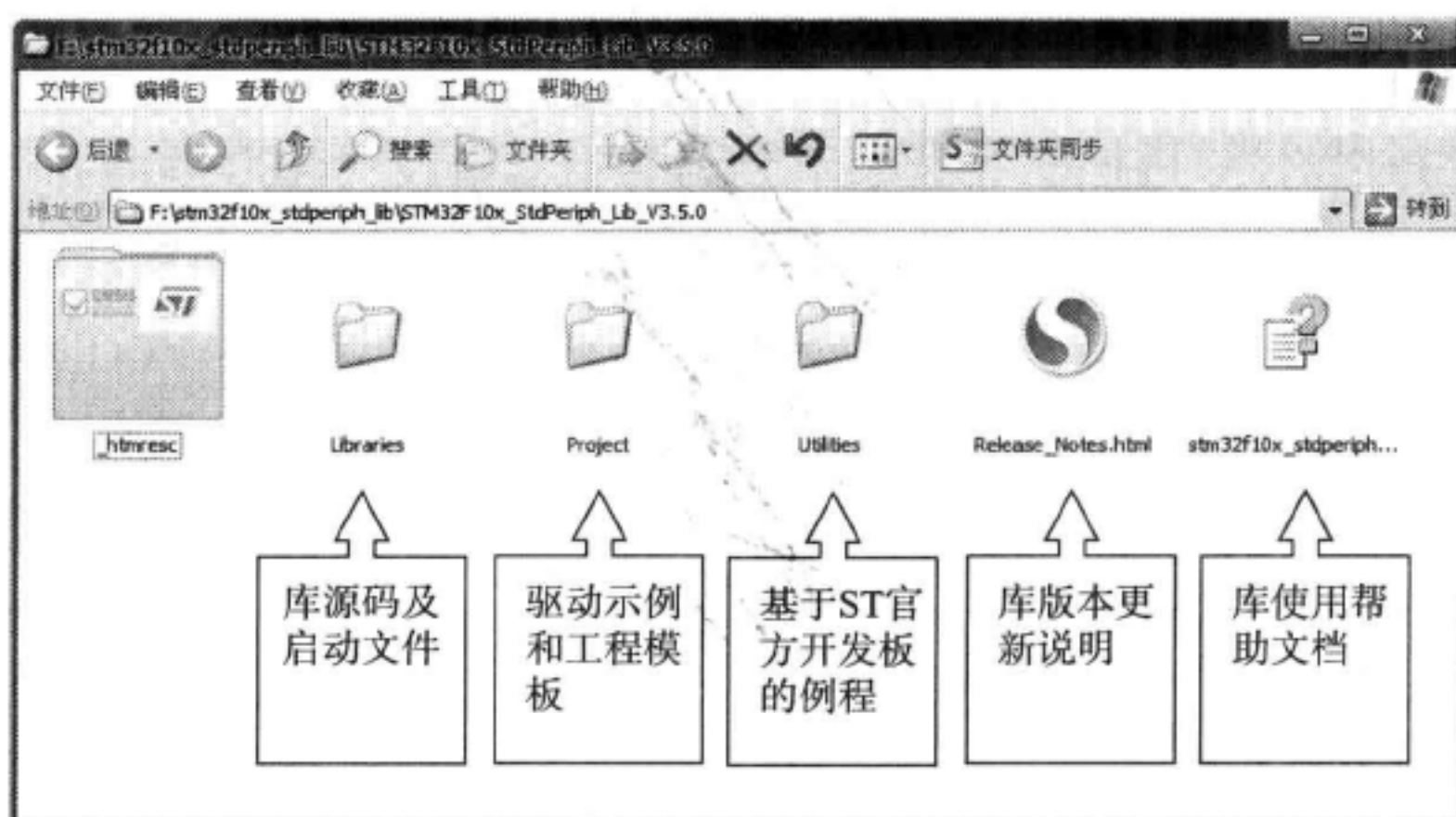


图 2-4 STM32 固件库目录

- Libraries 文件夹下是驱动库的源代码及启动文件。
- Project 文件夹下是用驱动库写的例子和一个工程模板。
- 库帮助文档，这是一个已经编译好的 HTML 文件，主要讲述如何使用驱动库来编写自己的应用程序。

在使用库开发时，我们需要把 Libraries 目录下的库函数文件添加到工程中，并查阅库帮助文档来了解 ST 提供的库函数，这个文档说明了每一个库函数的使用方法。

进入 Libraries 文件夹看到，关于内核与外设的库文件分别存放在 CMSIS 和 STM32F10x_StdPeriph_Driver 文件夹中。Libraries\CMSIS\CM3 文件夹下又分为 CoreSupport 和 DeviceSupport 文件夹。

1. core_cm3.c 文件

在 CoreSupport 文件夹中的是位于 CMSIS 标准的核内设备函数层的 CM3 核通用的源文件 core_cm3.c 和头文件 core_cm3.h，它们的作用是为采用 Cortex-M3 核设计 SoC 的芯片商设计的芯片外设提供一个进入 CM3 内核的接口。对于其他公司的 CM3 系列芯片这两个文件也是相同的。至于这些功能是怎样用源码实现的，我们先不用理会，我们只需把这个文件加进我们的工程文件即可，有兴趣的朋友可以深究。

core_cm3.c 文件还有一些与编译器相关的条件编译语句，用于屏蔽不同编译器的差异，我们在开发时不需要知道，有兴趣的读者可以了解一下。里面包含了一些与编译器相关的信息，如：RealView Compiler（本书采用的 MDK）、ICC Compiler（IAR）、GNU Compiler。见代码清单 2-1。

代码清单 2-1 core_cm3.c 文件中对编译器差异的屏蔽

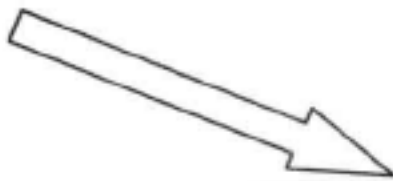
```
1. /* define compiler specific symbols */
2. #if defined ( __CC_ARM )
3.     #define __ASM          __asm
4.     #define __INLINE       __inline
5.
```

使用 MDK 编译器时的嵌入汇编与内联函数的关键字形式


```

6. #elif defined ( __ICCARM__ )
7.     #define __ASM          __asm
8.     #define __INLINE       inline
9. #elif defined ( __GNUC__ )
10.    #define __ASM           __asm
        #define __INLINE     inline
11.
12. #elif defined ( __TASKING__ )
13.    #define __ASM           __asm
14.    #define __INLINE       inline
15. #endif

```



使用 IAR 编译器时的形式

较重要的是在 core_cm3.c 文件中包含了 stdio.h 这个头文件，这是一个 ANSI C 文件，是独立于处理器之外的，就像我们熟知的 C 语言头文件 stdio.h 文件一样。它位于 RVMDK 这个软件的安装目录下，主要作用是提供一些新类型定义。见代码清单 2-2。

代码清单 2-2 core_cm3.c 文件中的类型定义

```

1. /* exact-width signed integer types */
2. typedef signed char int8_t;
3. typedef signed short int int16_t;
4. typedef signed int int32_t;
5. typedef signed __int64 int64_t;
6.
7. /* exact-width unsigned integer types */
8. typedef unsigned char uint8_t;
9. typedef unsigned short int uint16_t;
10. typedef unsigned int uint32_t;
11. typedef unsigned __int64 uint64_t;

```

这些新类型定义屏蔽了在不同芯片平台时，出现的诸如 int 的大小是 16 位还是 32 位的差异。所以在我们以后的程序中，都将使用新类型如 uint8_t、uint16_t 等。

在稍旧版的程序中还经常会出现如 u8、u16、u32 这样的类型，分别表示的无符号的 8 位、16 位、32 位整型。初学者碰到这样的旧类型感觉一头雾水，它们定义的位置在 STM32f10x.h 文件中。建议在以后的新程序中尽量使用 uint8_t、uint16_t 类型的定义。

core_cm3.c 与启动文件一样都是底层文件，都是由 ARM 公司提供的，遵守 CMSIS 标准，即所有 CM3 芯片的库都带有这个文件，这样软件在不同的 CM3 芯片的移植工作就得以简化。

2. system_stm32f10x.c 文件

在 DeviceSupport 文件夹下的是启动文件、外设寄存器定义和中断向量定义层的一些文件，这是由 ST 公司提供的，见图 2-5。

system_stm32f10x.c 文件是由 ST 公司提供的，遵守 CMSIS 标准，该文件的功能是设置系统时钟和总线时钟。STM32 比 51 单片机复杂得多，并不是说我们外部给一个 8M 的晶振，STM32 整个系统就以 8M 为时钟协调整个处理器的工作。我们还要通过 CM3 核的核内寄存器来对 8M 的时钟进行倍频、分频，或者使用芯片内部的时钟。所有的外设都与时钟的频率有关，所以这个文件的时钟配置是很关键的。



图 2-5 DeviceSupport 文件夹内容

system_stm32f10x.c 在实现系统时钟的时候要用到 PLL（锁相环），这就需要操作寄存器，寄存器都是以存储器映射的方式来访问的，所以该文件中包含了 stm32f10x.h 这个头文件。

3. stm32f10x.h 文件

stm32f10x.h 这个文件非常重要，是一个非常底层的文件。它包含了 STM32 中寄存器地址和结构体类型定义，在使用到 STM32 固件库的地方都要包含这个头文件。

4. 启动文件

(1) 启动文件的类型

Libraries\CMSIS\Core\CM3\startup\arm 文件夹下是由汇编语言编写的系统启动文件，不同的文件对应不同的芯片型号，在使用时要注意，见图 2-6。

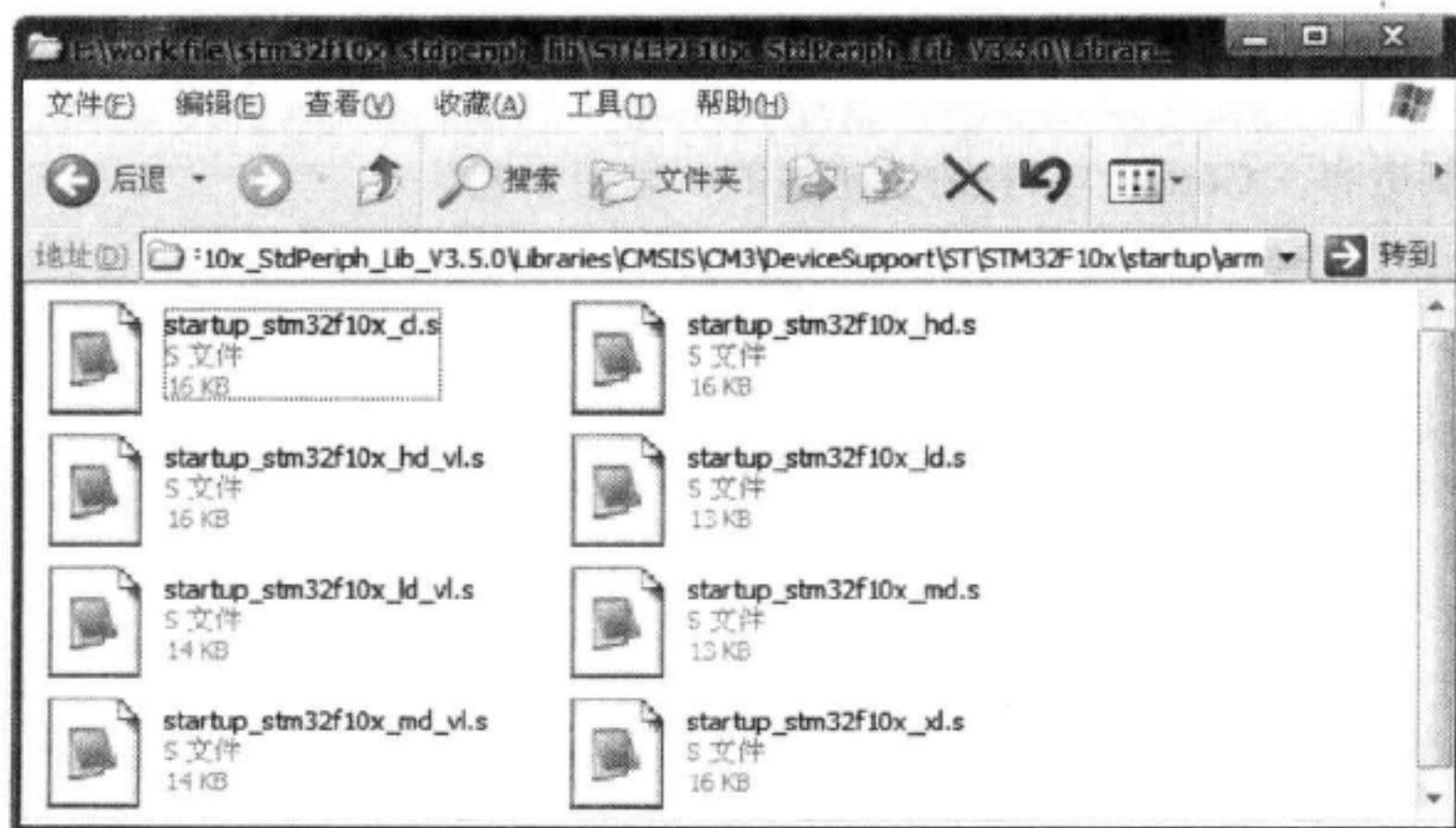


图 2-6 启动文件

文件名的英文缩写的意义如下：

□ cl：互联型产品，stm32f105/107 系列。

- ❑ vl: 超值型产品, stm32f100 系列。
- ❑ xl: 超高密度(容量)产品, stm32f101/103 系列。
- ❑ ld: 低密度产品, Flash 小于 64KB。
- ❑ md: 中等密度产品, Flash 等于 64KB 或 128KB。
- ❑ hd: 高密度产品, Flash 大于 128KB。

配套 STM32 开发板中用的芯片是 STM32F103VET6, 64KB RAM 和 512KB ROM, 是属于高密度产品, 所以启动文件要选择 startup_stm32f10x_hd.s。

(2) 启动文件的作用

启动文件是任何处理器在上电复位之后最先运行的一段汇编程序。在我们编写的 C 语言代码运行之前, 需要由汇编语言为 C 语言的运行建立一个合适的环境, 接下来才能运行我们的程序。所以我们要把启动文件添加到我们的工程中。

总的来说, 启动文件的作用是:

- ❑ 初始化堆栈指针 SP。
- ❑ 初始化程序计数器指针 PC。
- ❑ 设置堆、栈的大小。
- ❑ 设置异常向量表的入口地址。
- ❑ 配置外部 SRAM 作为数据存储器(这个由用户配置, 一般的开发板没有外部 SRAM)。
- ❑ 设置 C 库的分支入口 __main (最终用来调用 main 函数)。
- ❑ 3.5 版的启动文件还调用了在 system_stm32f10x.c 文件中的 SystemIni() 函数配置系统时钟, 在旧版本的工程中要用户进入 main 函数自己调用 SystemIni() 函数。

5. STM32F10x_StdPeriph_Driver 文件夹

Libraries\STM32F10x_StdPeriph_Driver 文件夹下有 inc (include 的缩写) 和 src (source 的缩写) 这两个文件夹, 这都属于 CMSIS 的设备外设函数部分。src 里面是每个设备外设的驱动程序, 这些外设是芯片制造商在 Cortex-M3 核外加进去的, 见图 2-7。

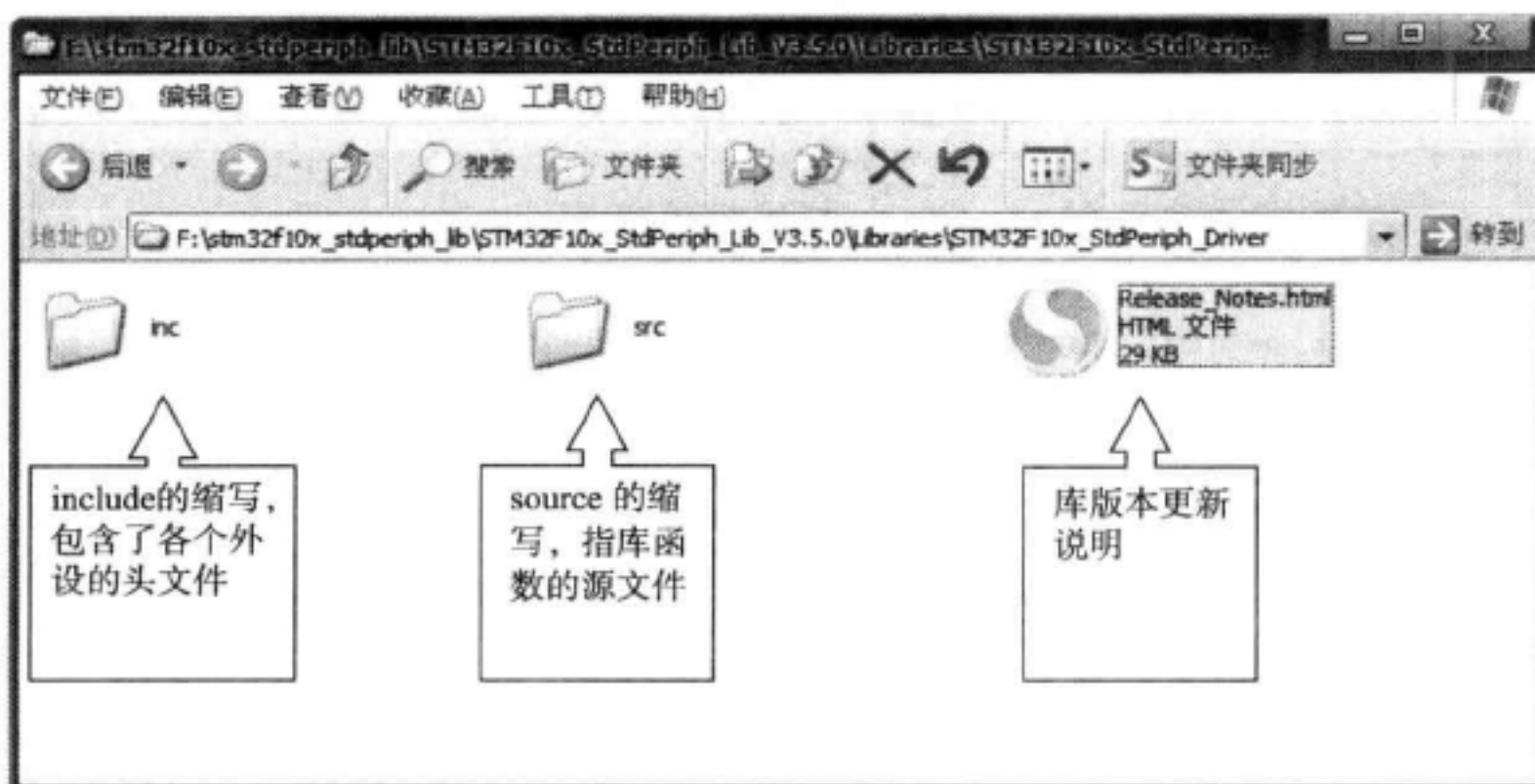


图 2-7 外设驱动

在 src 和 inc 文件夹里的就是 ST 公司针对每个 STM32 外设而编写的库函数文件，每个外设对应一个 .c 和 .h 后缀的文件。我们把这类外设文件统称为：stm32f10x_ppp.c 或 stm32f10x_ppp.h 文件，ppp 表示外设名称。

如针对模数转换（ADC）外设，在 src 文件夹下有一个 stm32f10x_adc.c 源文件，在 inc 文件夹下有一个 stm32f10x_adc.h 头文件，若我们开发的工程中用到了 STM32 内部的 ADC，则至少要把这两个文件包含到工程里，见图 2-8。



图 2-8 驱动源文件及头文件

这两个文件夹中还有一个很特别的 misc.c 文件，这个文件提供了外设对内核中的 NVIC（中断向量控制器）的访问函数，在配置中断时，我们必须把这个文件添加到工程中。

6. stm32f10x_it.c 和 stm32f10x_conf.h 文件

在库目录的 Project\STM32F10x_StdPeriph_Template 目录下，存放了官方的一个库工程模板，我们在用库建立一个完整的工程时，还需要添加这个目录下的 stm32f10x_it.c、stm32f10x_it.h 和 stm32f10x_conf.h 这三个文件。

stm32f10x_it.c 是专门用来编写中断服务函数的，在我们修改前，这个文件已经定义了一些系统异常的接口，其他普通中断服务函数由我们自己添加。但是我们怎么知道这些中断服务函数的接口如何写呢？是不是可以自定义呢？答案当然不是的，这些都可以在汇编启动文件中找到，具体内容大家可查阅库启动文件的源码。

stm32f10x_conf.h 文件被包含进 stm32f10x.h 文件，是用来配置使用了什么外设的头文件，用

这个头文件我们可以很方便地增加或删除上面 Driver 目录下的外设驱动函数库。如代码清单 2-3 的代码配置表示使用了 gpio、rcc、spi、usart 的外设库函数，其他注释掉的部分，表示没有用到。

代码清单 2-3 stm32f10x_conf.h 文件配置固件库

```

1. /* Includes -----*/
2. /* Uncomment/Comment the line below to enable/disable peripheral header file inclusion */
3. // #include "stm32f10x_adc.h"
4. // #include "stm32f10x_bkp.h"
5. // #include "stm32f10x_can.h"
6. // #include "stm32f10x_cec.h"
7. // #include "stm32f10x_crc.h"
8. // #include "stm32f10x_dac.h"
9. // #include "stm32f10x_dbgmcu.h"
10. // #include "stm32f10x_dma.h"
11. // #include "stm32f10x_exti.h"
12. // #include "stm32f10x_flash.h"
13. // #include "stm32f10x_fsmc.h"
14. #include "stm32f10x_gpio.h"
15. // #include "stm32f10x_i2c.h"
16. // #include "stm32f10x_iwdg.h"
17. // #include "stm32f10x_pwr.h"
18. #include "stm32f10x_rcc.h"
19. // #include "stm32f10x_rtc.h"
20. // #include "stm32f10x_sdio.h"
21. #include "stm32f10x_spi.h"
22. // #include "stm32f10x_tim.h"
23. #include "stm32f10x_usart.h"
24. // #include "stm32f10x_wwdg.h"
25. // #include "misc.h" /* High level functions for NVIC and SysTick (add-on to CMSIS functions) */

```

stm32f10x_conf.h 这个文件还可配置是否使用“断言”编译选项，在开发时使用“断言”可由编译器检查库函数传入的参数是否正确，软件编写成功后，去掉“断言”编译选项可使程序全速运行。可通过定义 USE_FULL_ASSERT 或取消定义来配置是否使用“断言”。

2.2.3 STM32 固件库文件间的关系

前面向大家简单介绍了各个库文件的作用，库文件直接包含进工程即可，丝毫不用修改，而有的文件就要我们在使用的时候根据具体的需要进行配置。接下来从整体上把握一下各个文件在库工程中的层次或关系，这些文件都对应到 CMSIS 标准架构上，见图 2-9。

图 2-9 描述了 STM32 固件库各文件之间的调用关系，这个图省略了 DSP 核（Cortex-M3 没有 DSP 核）和实时系统层的文件关系。在实际使用库开发工程的过程中，我们把位于 CMSIS 层的文件包含进工程，丝毫不用修改，也不建议修改。

对于位于用户层的几个文件，就是我们在使用库的时候，针对不同的应用对库文件进行增删（用条件编译的方法增删）和改动的文件。

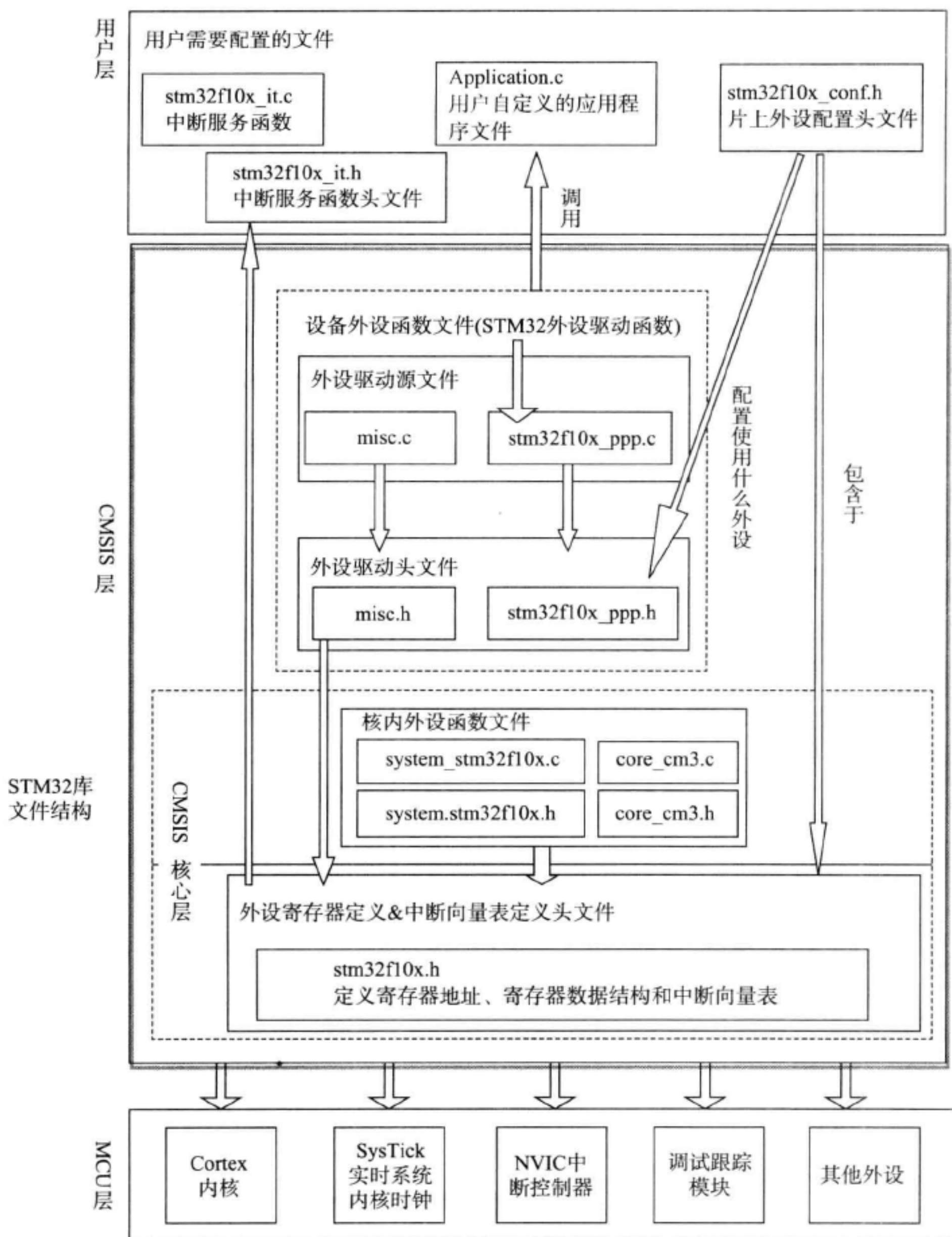


图 2-9 库各文件之间的关系

2.2.4 使用库帮助文档

授之以鱼不如授之以渔，官方资料是所有关于 STM32 知识的源头，所以在本节介绍如何使用官方资料。官方的帮助手册是最好的教程，几乎包含了所有在开发过程中遇到的问题。这些资

料已整理到随书光盘。

1. 常用官方资料

□ stm32f10x_stdperiph_lib_um.chm

这个就是前面提到的库帮助文档，在使用库函数时，我们最好通过查阅此文件来了解库函数原型或库函数调用的方法，也可以直接阅读源码里面的函数说明。

□ STM32 参考手册 .pdf

这个文件相当于 STM32 的 datasheet，它把 STM32 的时钟、存储器架构及各种外设、寄存器都描述得清清楚楚。当我们对 STM32 库函数的实现方式感到困惑时，可查阅这个文件。若以直接配置寄存器方式开发，查阅这个文档的频率会更高，但这样效率太低了。

□ 《ARM Cortex-M3 权威指南》

该手册是由 ARM 公司提供的，它详细讲解了 Cortex 内核的架构和特性，要深入了解 Cortex-M3 内核，这个文档是首选。

2. 初识库函数

所谓库函数，就是 STM32 固件库文件中为我们编写好的函数接口，我们只要调用这些库函数，就可以对 STM32 进行配置，达到控制目的。我们可以不知道库函数是如何实现的，但我们调用函数必须要知道函数的功能、可传入的参数及其意义和函数的返回值。

于是，有读者就问那么多函数我怎么记呀？我们的回答是：会查就行！所以我们学会查阅库帮助文档是很有必要的。

打开库帮助文档 stm32f10x_stdperiph_lib_um.chm，见图 2-10。

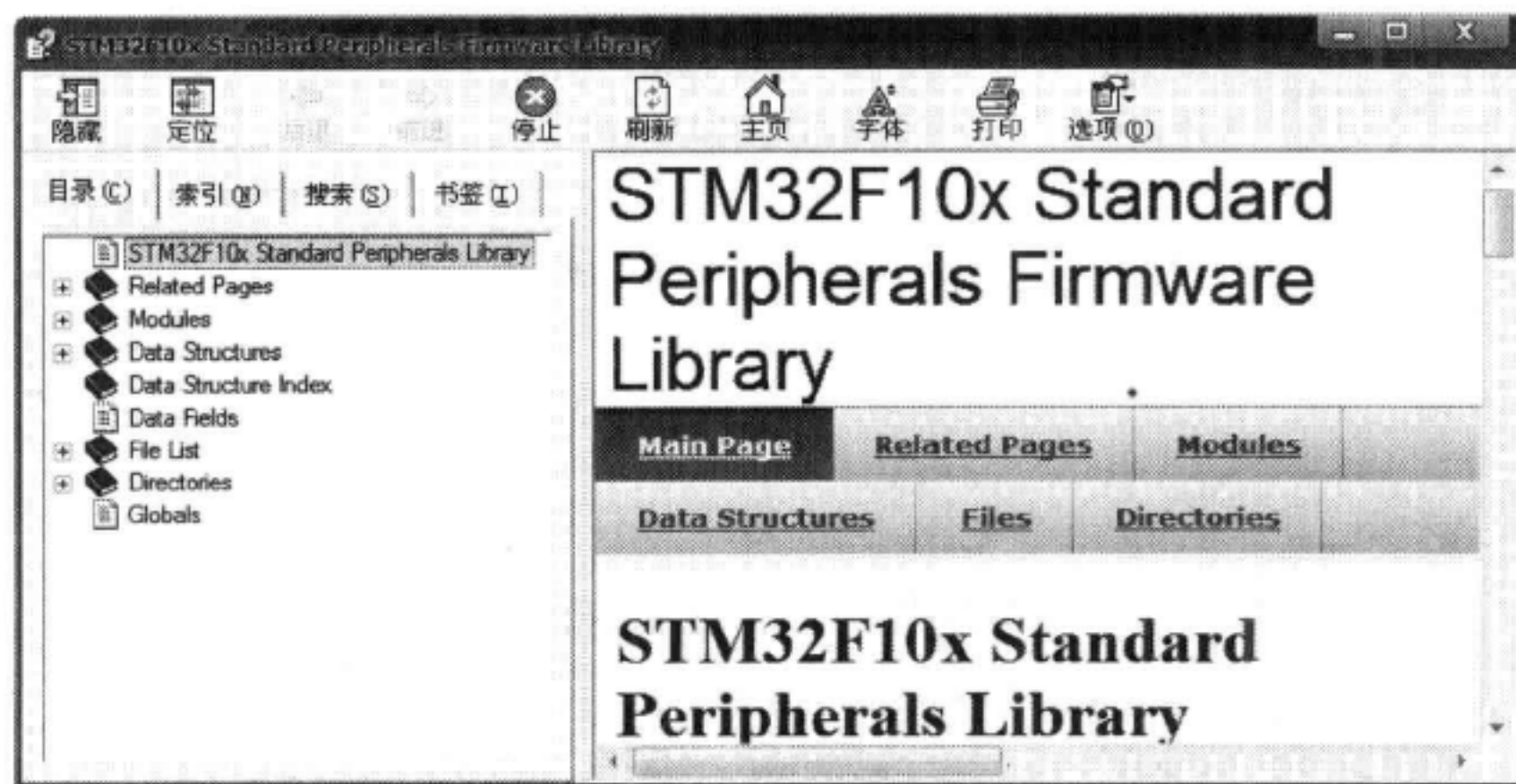


图 2-10 库帮助文档

层层打开文档的目录标签 Modules\STM32F10x_StdPeriph_Driver，可看到该标签下有很多外设驱动文件的名称，如 MISC、ADC、BKP、CAN 等。我们试着查看 ADC 的初始化库函数（ADC_Init）看看，继续打开标签 \ADC\ADC_Exported_Functions\Functions\ADC_Init，见图 2-11。



图 2-11 库帮助文档的函数说明

利用这个文档，我们即使没有去看它的具体代码，也知道要怎么利用它了。

如 ADC_Init 的功能是：以 ADC_InitStruct 参数配置 ADC，进行初始化。原型为：

```
void ADC_Init(ADC_TypeDef * ADCx , ADC_Init_TypeDef * ADC_InitStruct)
```

其中输入的参数 ADCx 和 ADC_InitStruct 均为库文档中的自定义数据类型，这两个传入参数均为结构体指针。初学时，我们并不知道如 ADC_TypeDef 这样的类型是什么意思，可以点击函数原型中带下划线的 ADC_TypeDef 就可以查看这是什么类型了。

就这样初步了解一下库函数，读者就可以发现 STM32 的库写得很优美。每个函数和数据类型都符合见名知义的原则，当然，这样的名称写起来特别长，而且对于我们来说要输入这么长的英文很容易出错，所以在开发软件的时候，在用到库函数的地方，直接把库帮助文档中函数名称复制粘贴到工程文件就可以了。



第 3 章

GPIO 入门之流水灯

本章为读者讲解如何建立工程模板和编译下载程序。接触过 Linux 的朋友都知道，在开始学习 GCC 编程时都喜欢以 Hello World 来作为第一个入门程序，在单片机中我们则常常以点亮 LED 灯来作为入门程序。所以在这个章节，演示下载一个 LED 流水灯的工程，先让代码在 STM32 上跑起来！详细的代码分析将在第 4 章展开。

3.1 安装 MDK

在新建工程之前我们先要把 MDK 这个软件安装好，这里用的版本是 V4.12，在安装完成之后可以在工具栏 help->about μ Vision 选项卡中查看到版本信息。 μ Vision 是一个集代码编辑、编译、链接及下载于一体的集成开发环境（IDE），其支持我们常见的 ARM7、ARM9 和 ARM 最新内核的 CM3 系列，其前身就是 51 中的大名鼎鼎的 Keil，相信大家会很快入手这个 IDE 的。如果大家要把 MDK 用在商业行为，需购买正版软件，这里仅用作教学演示。

MDK 安装过程如下所示：

- 1) 点击 Next 按钮，如图 3-1 所示。
- 2) 勾选复选框，点击 Next 按钮，如图 3-2 所示。

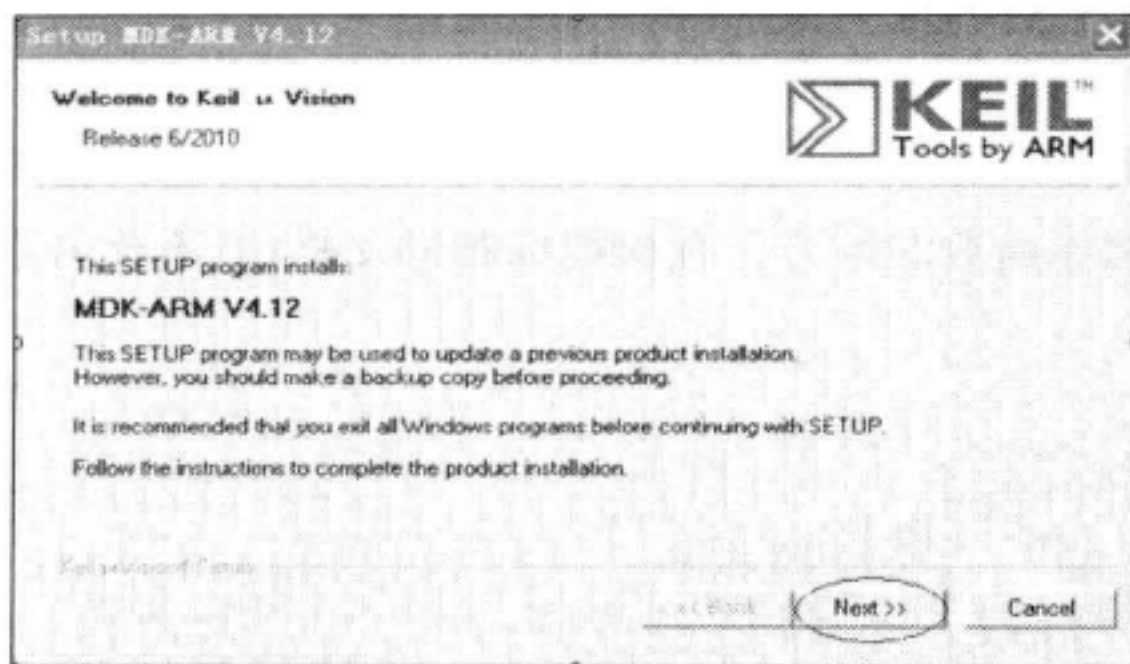


图 3-1 安装步骤 1

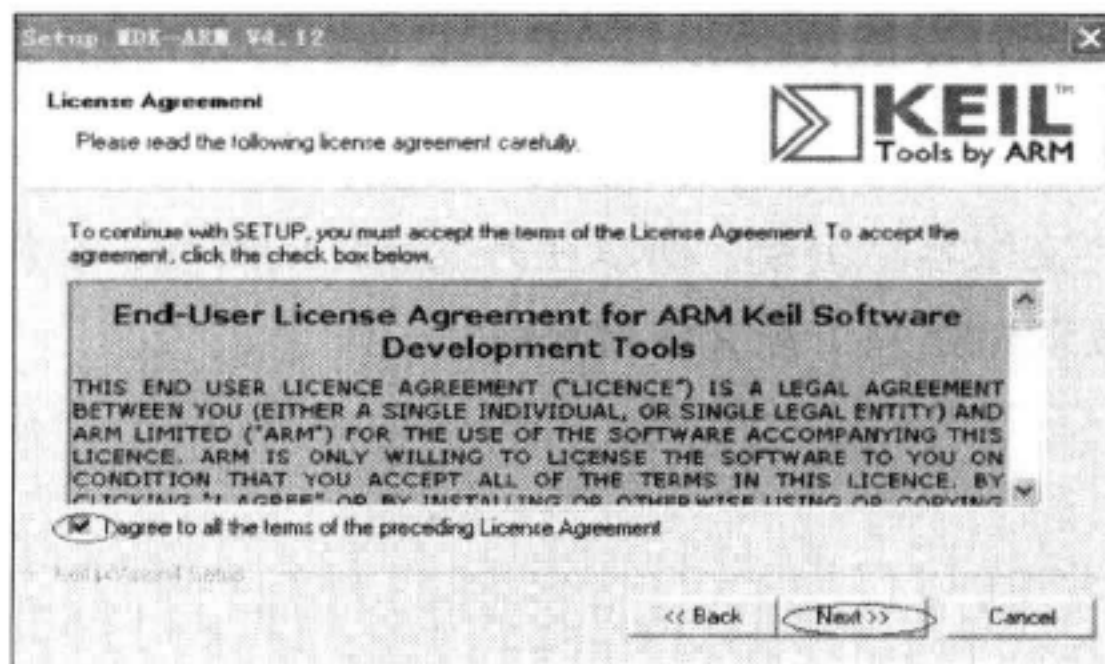


图 3-2 安装步骤 2

- 3) 点击 Next 按钮，默认安装在 C:\Keil 目录下，如图 3-3 所示。
- 4) 在用户名中填入名字，在邮件地址中填入邮件地址（可随便写，可空格），点击 Next 按钮，如图 3-4 所示。

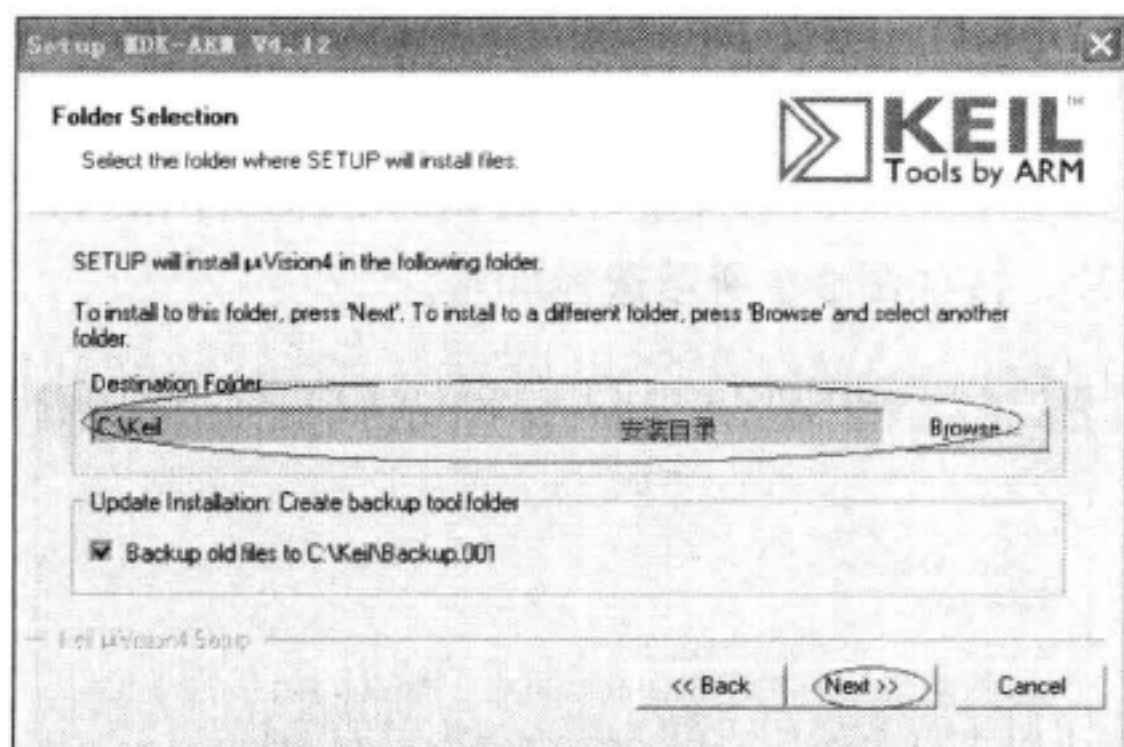


图 3-3 安装步骤 3

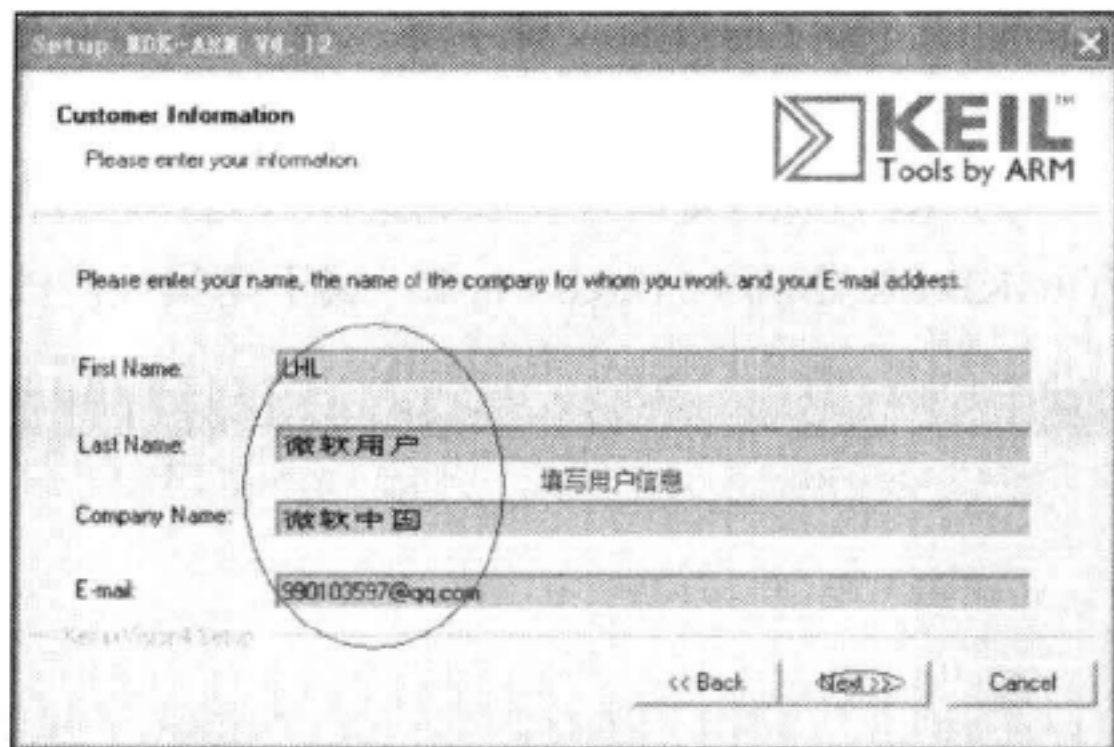


图 3-4 安装步骤 4

- 5) 正在安装, 如图 3-5 所示, 请耐心等待。
- 6) 点击 Finish 按钮, 安装完成, 如图 3-6 所示。

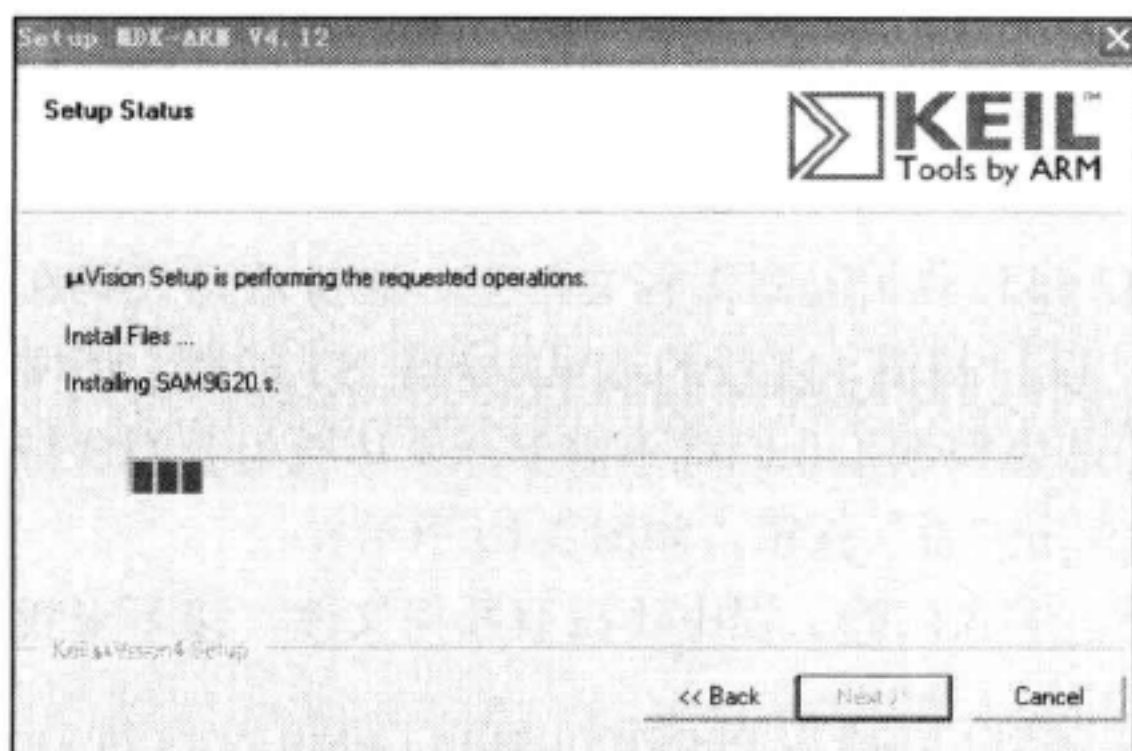


图 3-5 安装步骤 5

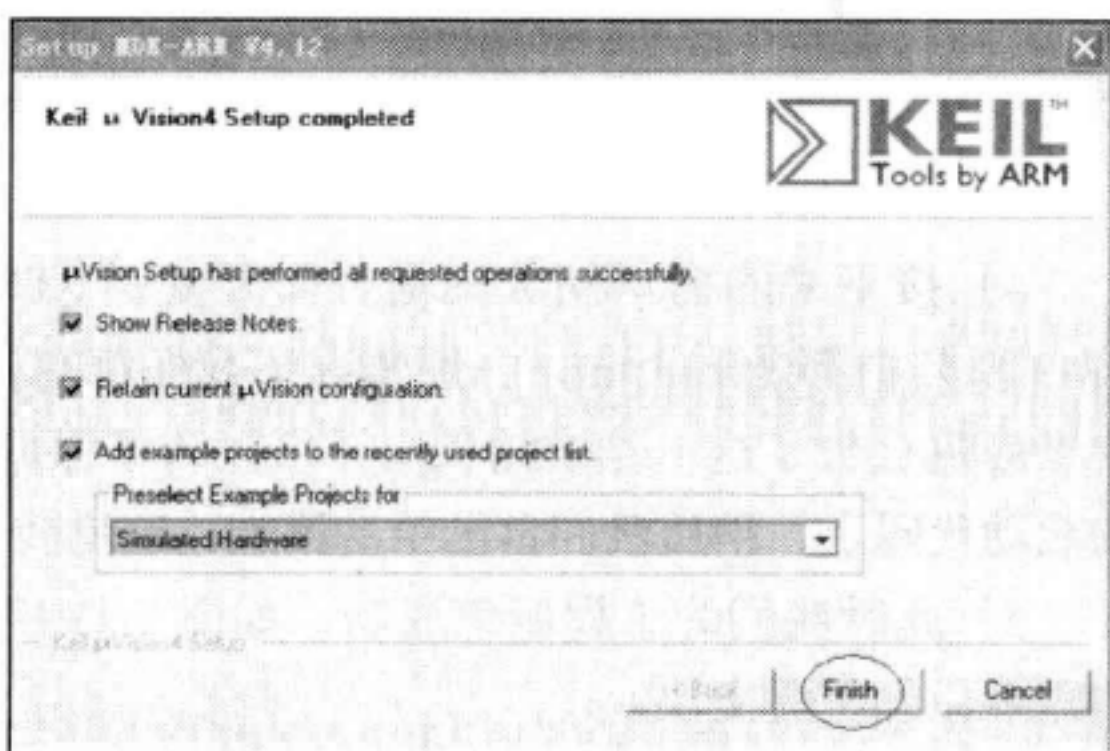


图 3-6 安装步骤 6

- 7) 此时就可在桌面看到 μ Vision 的快捷图标, 如图 3-7 所示。



图 3-7 快捷方式图标

3.2 建立工程模板

安装完 MDK 之后, 紧接着我们开始利用 STM32 的官方库来构建自己的工程模板。以后我们就用自己建立的模板来新建工程, 方便快捷。

3.2.1 新建工程

- 1) 点击桌面 μ Vision4 图标, 启动软件。如果是第一次使用的话会打开一个自带的工程文件, 我们可以通过工具栏 Project->Close Project 选项把它关掉。
- 2) 在工具栏 Project->New μ Vision Project 新建我们的工程文件, 我们将新建的工程文件保存

在桌面的 TEST/USER 文件夹下，文件取名为 STM-DEMO（英文 DEMO 的意思是例子），名字可以随便取，点击“保存”按钮，如图 3-8 所示。

3) 接下来的窗口是让我们选择公司和芯片的型号，我们用的芯片是 ST 公司的 STM32F103VE，有 64KB SRAM 和 512KB Flash，属于高集成度的芯片。按如图 3-9 所示选择即可。



图 3-8 保存工程到 USER 目录

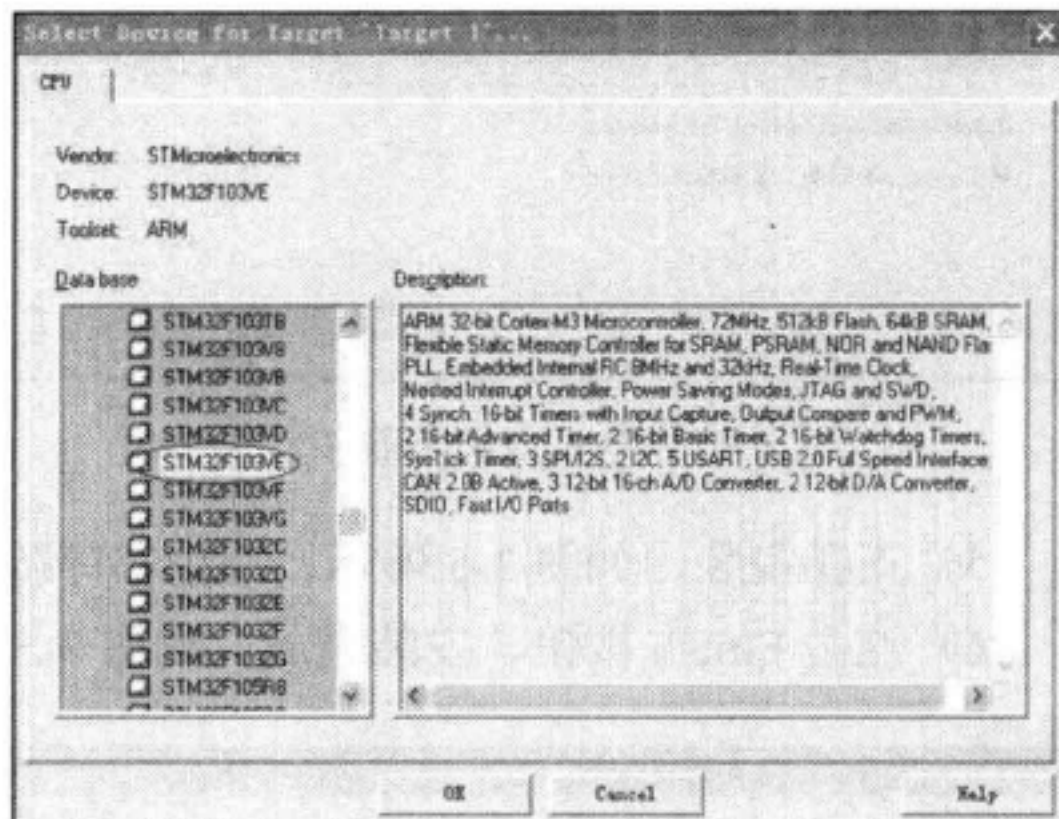


图 3-9 选择芯片型号

4) 接下来的窗口问我们是否需要复制 STM32 的启动代码到工程文件中，这份启动代码在 CM3 系列中都是适用的，一般情况下我们都点击“是”按钮，但我们这里用的是 ST 的库，库文件里面也自带了这一份启动代码，所以为了保持库的完整性，我们就不需要开发环境为我们自带的启动代码了，稍后我们自己手动添加，这里我们点击“否”按钮，如图 3-10 所示。

5) 此时我们的工程新建成功，如图 3-11 所示。但我们的工程中还没有任何文件，接下来我们需要添加所需文件。



图 3-10 不使用 Keil 自带的启动代码

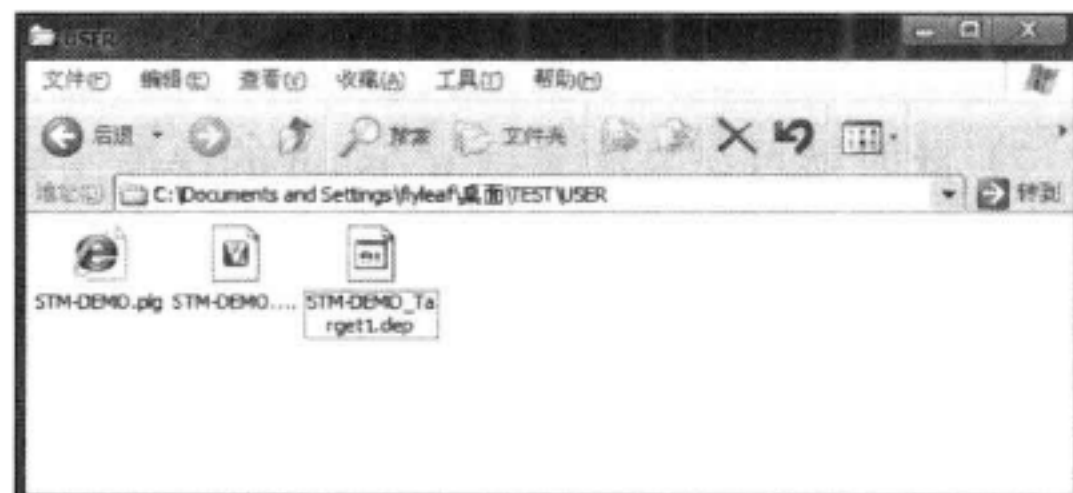


图 3-11 新建工程的效果图

6) 在 TEST 文件夹中，新建 5 个文件夹，分别为 USER、FWlib、CMSIS、Output、Listing，其中 USER 已经存在，就不需再建了，见图 3-12。

- ❑ USER 用来存放工程文件和用户层代码，包括主函数 main.c。
- ❑ FWlib 用来存放 STM32 库里面的 inc 和 src 这两个文件夹，这两个文件包含了芯片上的所有驱动，这两个文件夹下的文件我们不需要修改。

❑ CMSIS 用来存放库为我们自带的启动文件和一些位于 CMSIS 层的文件。

❑ Output 文件夹用来保存软件编译后输出的文件。

❑ Listing 用来保存编译后生成的链接文件。

7) 把库 \Libraries\STM32F10x_StdPeriph_Driver 文件夹下的 inc 跟 src 这两个文件夹复制到 FWlib 文件夹中, 见图 3-13。

8) 把库文件夹 Project\STM32F10x_StdPeriph_Template 下的 main.c、stm32f10x_conf.h、stm32f10x_it.h 和 stm32f10x_it.c 复制到 USER 目录下。这 4 个文件是用户在编程时需要修改的文件, 其他库文件一般不需要修改, 见图 3-14。

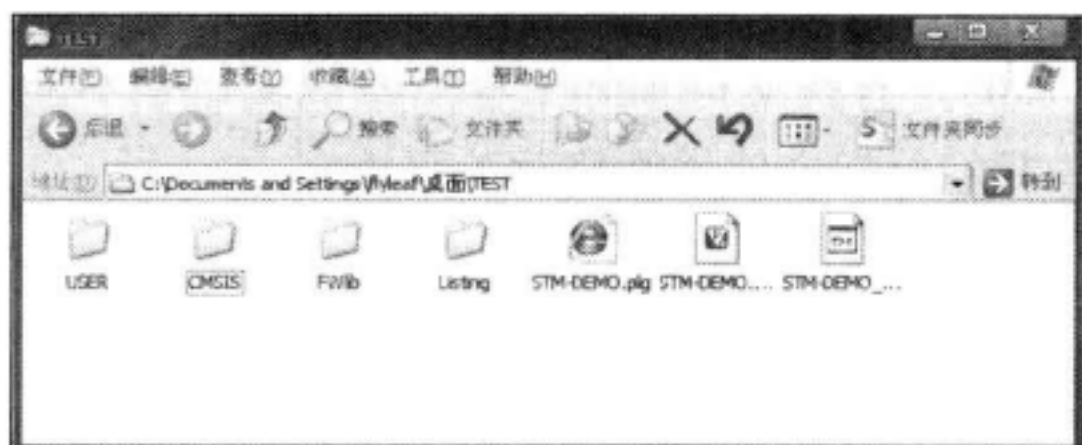


图 3-12 创建各种文件夹

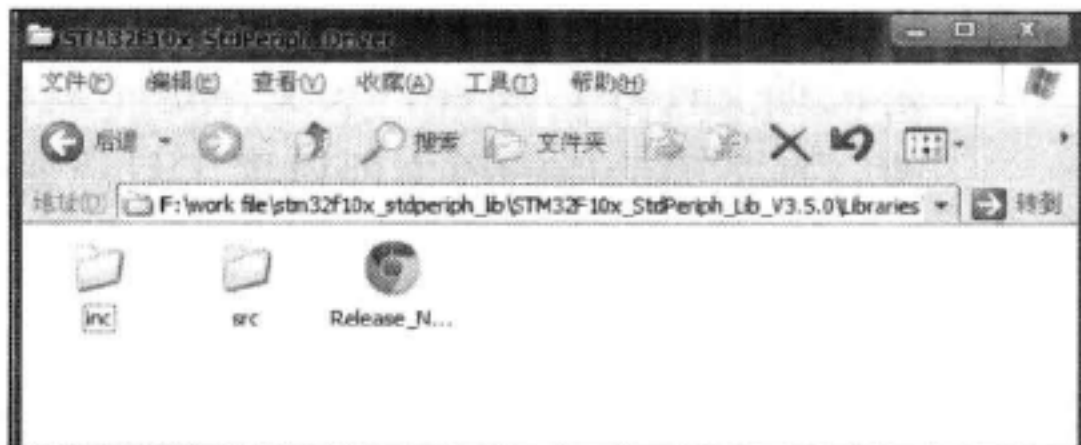


图 3-13 复制固件库

9) 将库文件 \Libraries\CMSIS\CM3 文件夹下的全部文件复制到 CMSIS 文件夹中。

Startup\arm 里面有 6 个启动文件, 每个文件都对应着不同的 STM32 型号芯片, 请参考 2.2.2 节。我们这里以配套 STM32V3 开发板为例, 这个开发板用的是 STM32F103VE 型号, 具有 512KB 的 Flash, 属于大容量, 所以在下面我们把 startup_stm32f10x_hd.s 启动文件添加到我们的工程中。据 ST 的官方资料: Flash 在 16 ~ 32 KB 为小容量, 64 ~ 128 KB 为中容量, 256 ~ 512 KB 为大容量, 不同大小的 Flash 对应的启动文件不一样, 这点要注意。

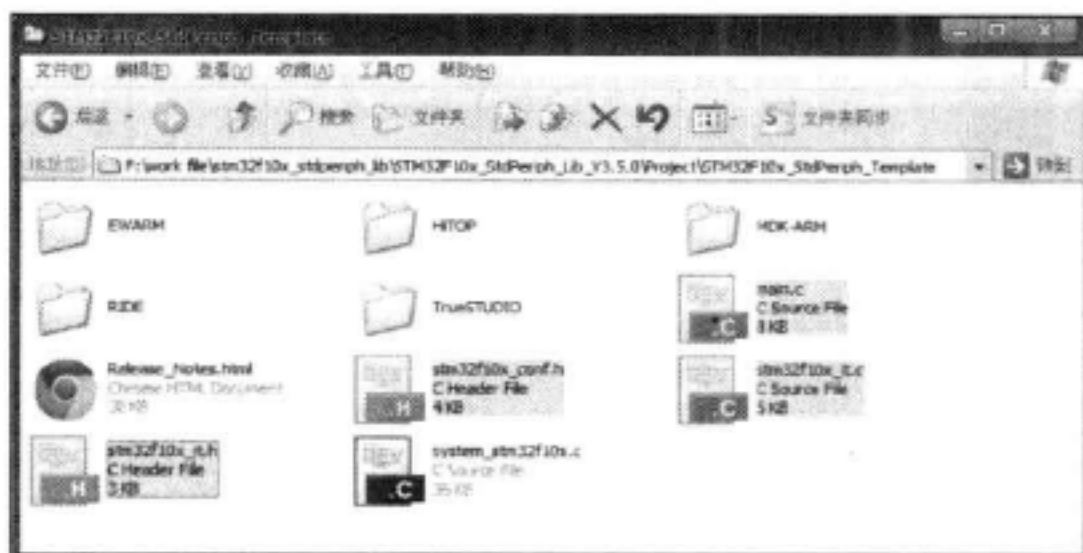


图 3-14 复制库工程文件

其他几个文件, system_stm32f10x.h、system_stm32f10x.c、stm32f10x.h 这几个位于 CMSIS 层的文件也是放到 CMSIS 文件夹中。这个步骤完成后如图 3-15 所示。

10) 回到我们的工程中, 选中 Target 右键选中 Add Group 选项新建 4 个组, 分别命名为 STARTCODE、USER、FWlib、CMSIS, 见图 3-16。

❑ STARTCODE 从名字就可以看出我们是用它来放我们的启动代码的。

❑ USER 用来存放用户自定义的应用程序。

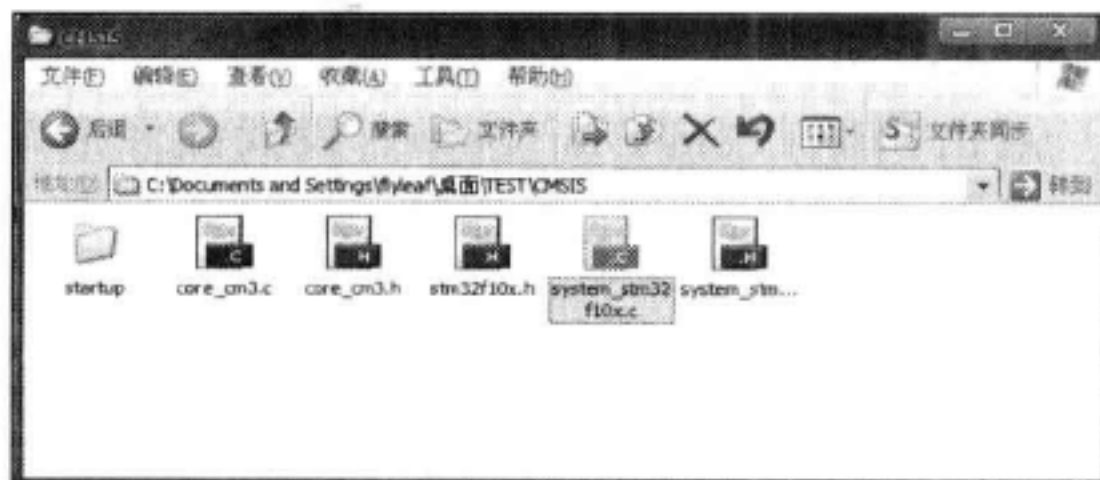


图 3-15 CMSIS 文件夹下的文件

- ❑ FWlib 用来存放库外设驱动文件。
 - ❑ CMSIS 用来存放 M3 系列单片机通用的文件。
- 11) 接下来我们往这些新建的组中添加文件，双击哪个组就可以往哪个组里面添加文件。
- ❑ 在 STARTCODE 组里面添加 startup_stm32f10x_hd.s 启动文件。
 - ❑ 在 USER 组里面添加 main.c 和 stm32f10x_it.c 这两个文件。
 - ❑ 在 FWlib 组里面添加 src 文件夹里面的全部驱动文件，当然 src 里面的驱动文件也可以按需添加。这里将它们全部添加进去是为了后续开发的方便，况且我们可以通过配置 stm32f10x_conf.h 这个头文件来选择性添加，只有在 stm32f10x_conf.h 文件中配置的文件才会被编译。
 - ❑ 在 CMSIS 里面添加 core_cm3.c 和 system_stm32f10x.c 文件。注意，这些组里面添加的都是汇编文件和 C 文件，头文件是不需要添加的。有些文件加到工程后发现文件是不能修改，那是因为库的原文件设置了只读属性，我们要把文件属性修改为可读写。
- 添加完文件后，最终效果见图 3-17。

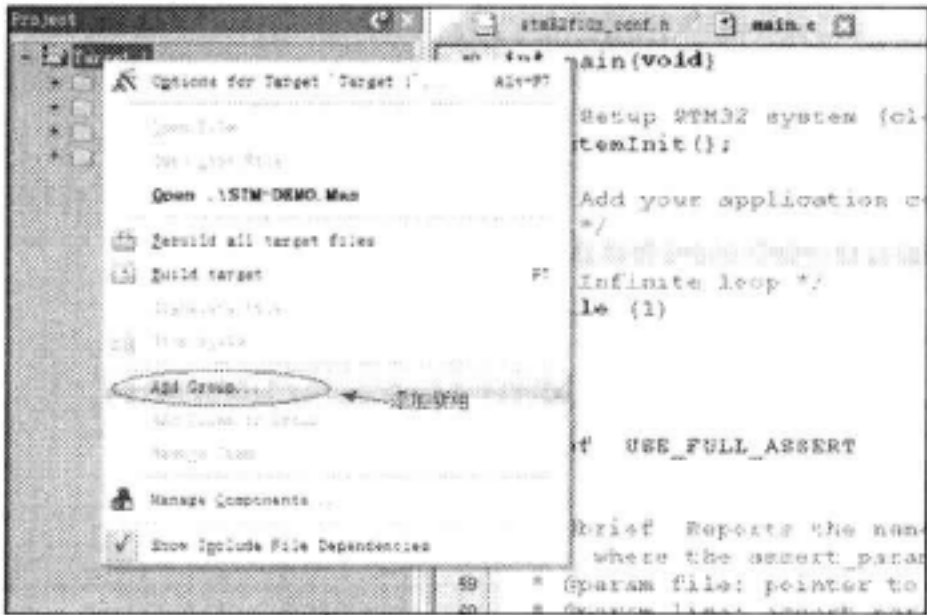


图 3-16 添加文件组

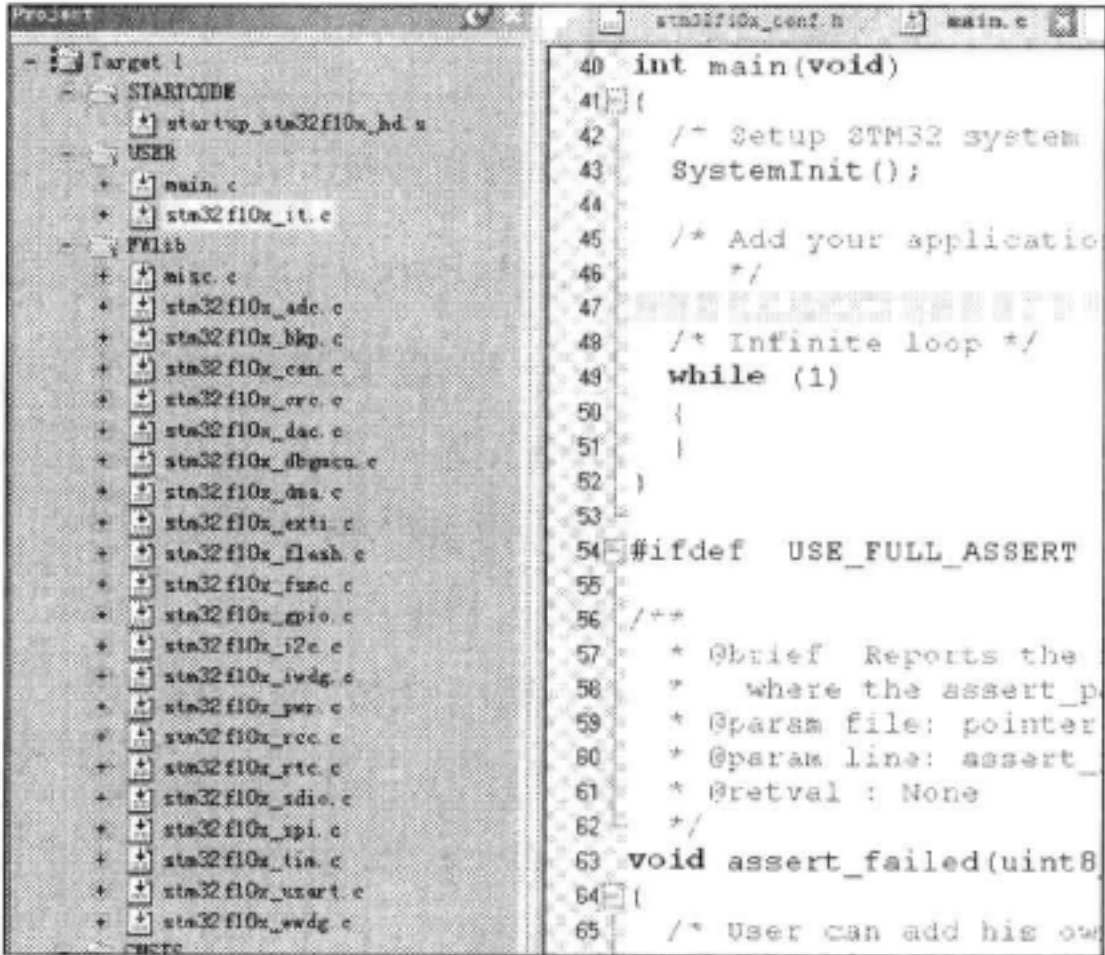



图 3-17 工程文件

12) 至此，我们的工程已经基本建好，接下来我们来配置 MDK 的某些选项卡，点击工具栏中的魔术棒按钮 ，在弹出来的窗口中选中“Output|”选项卡，见图 3-18。

点击 Select Folder for Objects 设置编译后输出的文件保存的位置，这里选择 Output 文件夹，见图 3-19。

13) 回到选项卡，选中“Listing”选项卡，点击 Select Folder for Listings 选择 Listing 文件夹，用来保存生成的链接文件，见图 3-20。

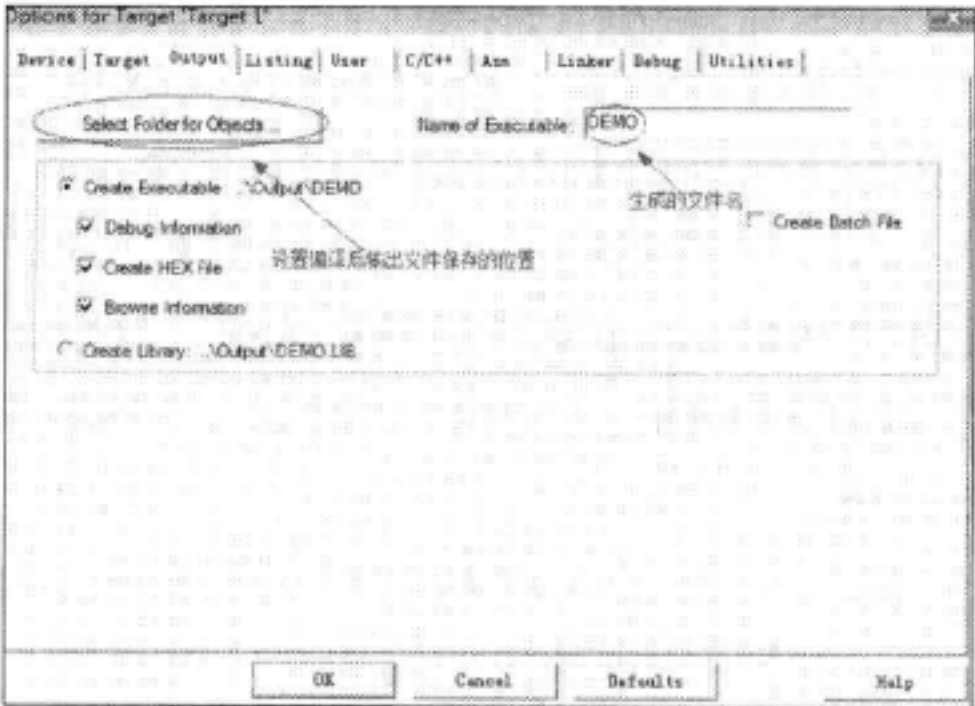


图 3-18 Output 选项卡



图 3-19 设置输出文件保存位置

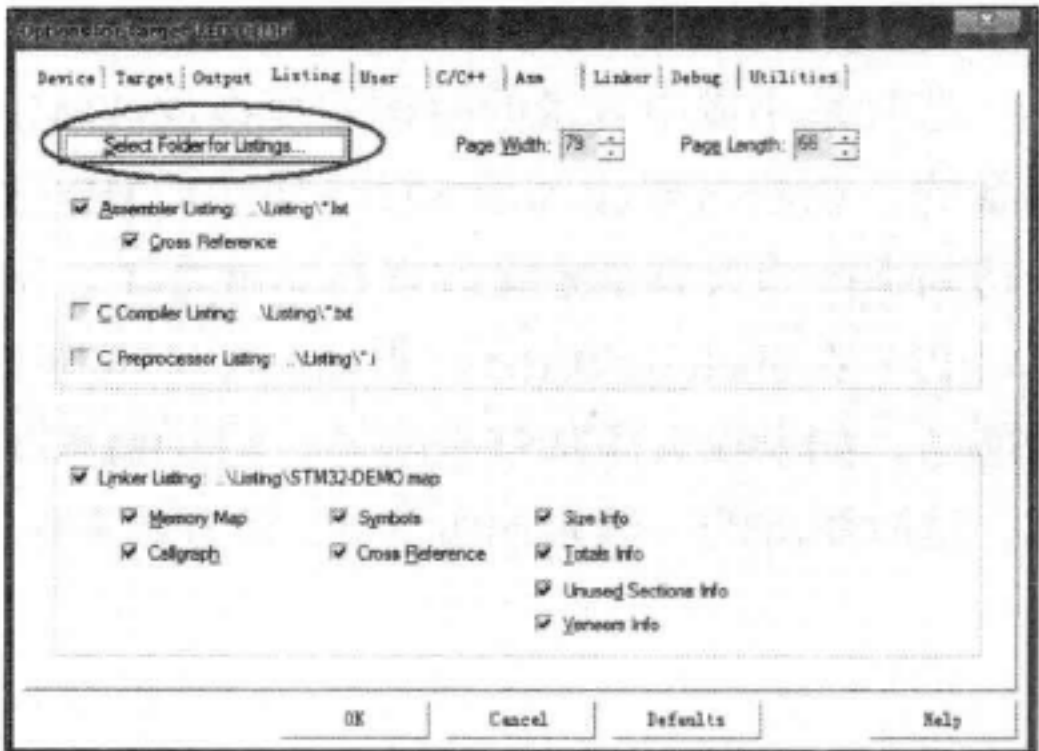



图 3-20 Listing 选项卡

14) 把原来从库函数复制来的 main.c 文件里的内容全部删除，输入代码清单 3-1 的基本代码。

代码清单 3-1 main 文件基本代码

```
1. #include "stm32f10x.h"
2.
3. int main (void)
4. {
5.     while (1);
6.     // add your code here ^_^。
7. }
```

15) 现在点击工具栏图标来编译一下，结果发现了非常多的错误，如图 3-21 所示。究其原因是编译器在编译时搜索的默认库路径是：C:\Keil\ARM\INC\ST\STM32F10x，这里面也有 STM32 官方驱动库的头文件，里面的文件与我们的 inc 文件夹下的内容差不多，只是版本旧了点，在编译我们新版本库时存在不兼容。为了解决这个问题，我们需要屏蔽掉编译器默认库的搜索路径。

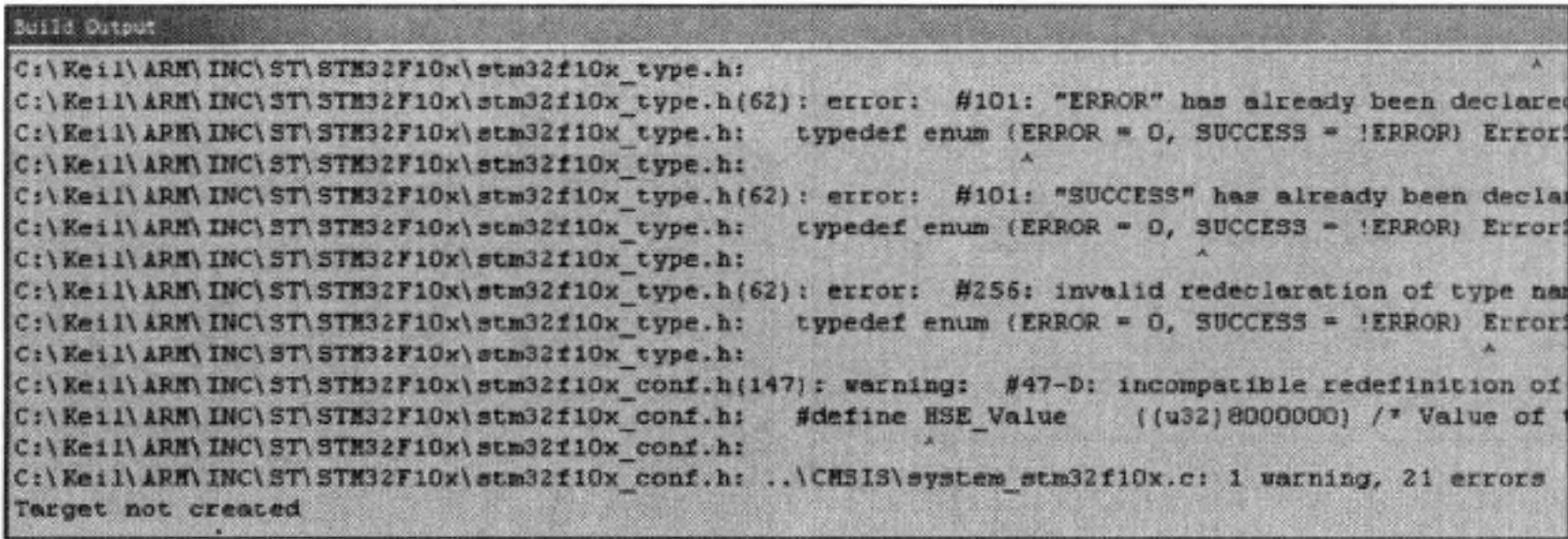


图 3-21 编译失败

16) 点击工具栏中的魔术棒按钮, 在弹出来的窗口中选中“C/C++”选项卡，在 Define 文本框里面添加两个宏定义：USE_STDPERIPH_DRIVER, STM32F10X_HD，见图 3-22。

添加 USE_STDPERIPH_DRIVER 是为了使用 ST 官方库，添加 STM32F10X_HD 宏定义是因为我们用的芯片是大容量的，添加了这个宏之后，我们就可以用库文件里面为大容量定义的寄存

21) 至此，大功告成，我们就可以在 main.c 函数中写自己的程序了，见图 3-29。

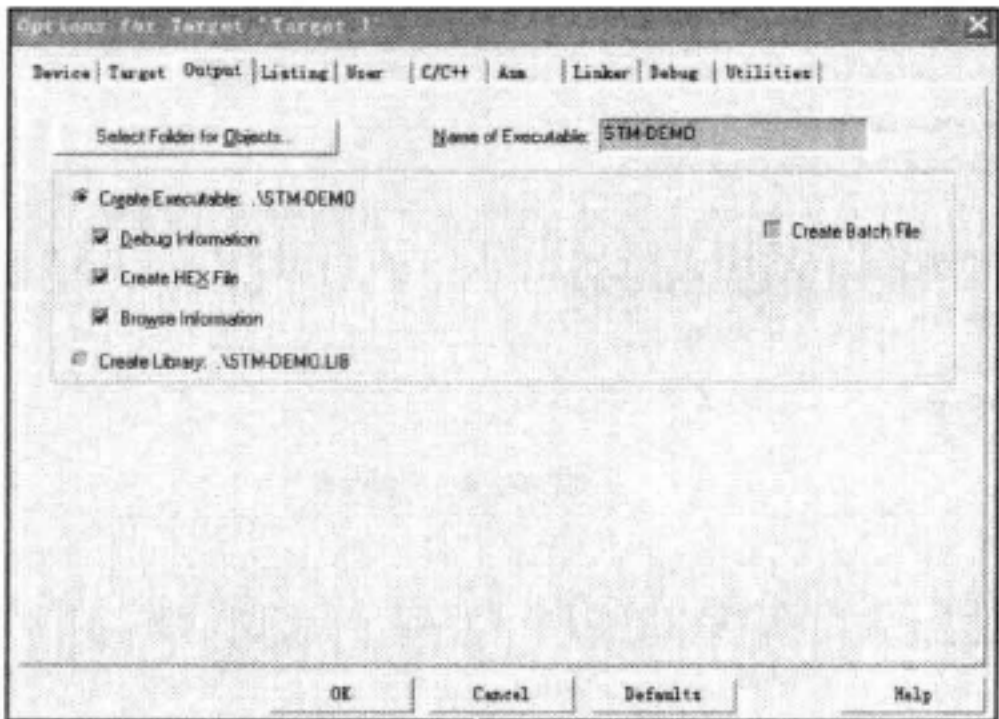


图 3-26 Output 选项卡设置

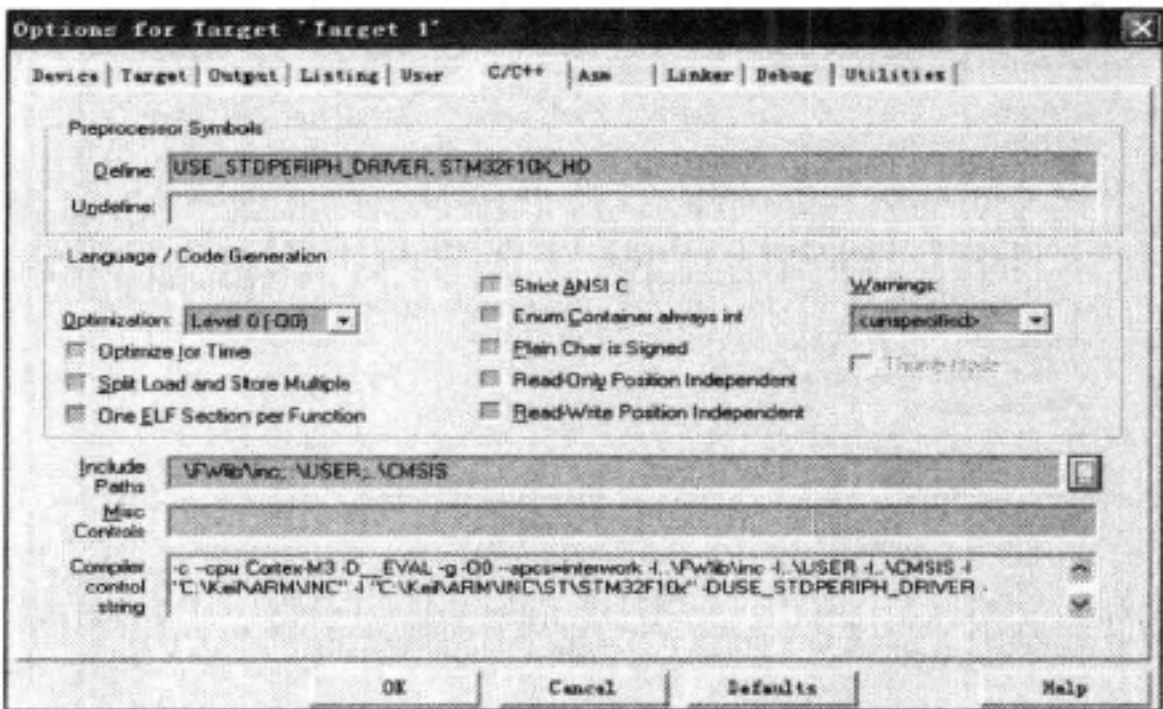


图 3-27 C/C++ 选项卡设置

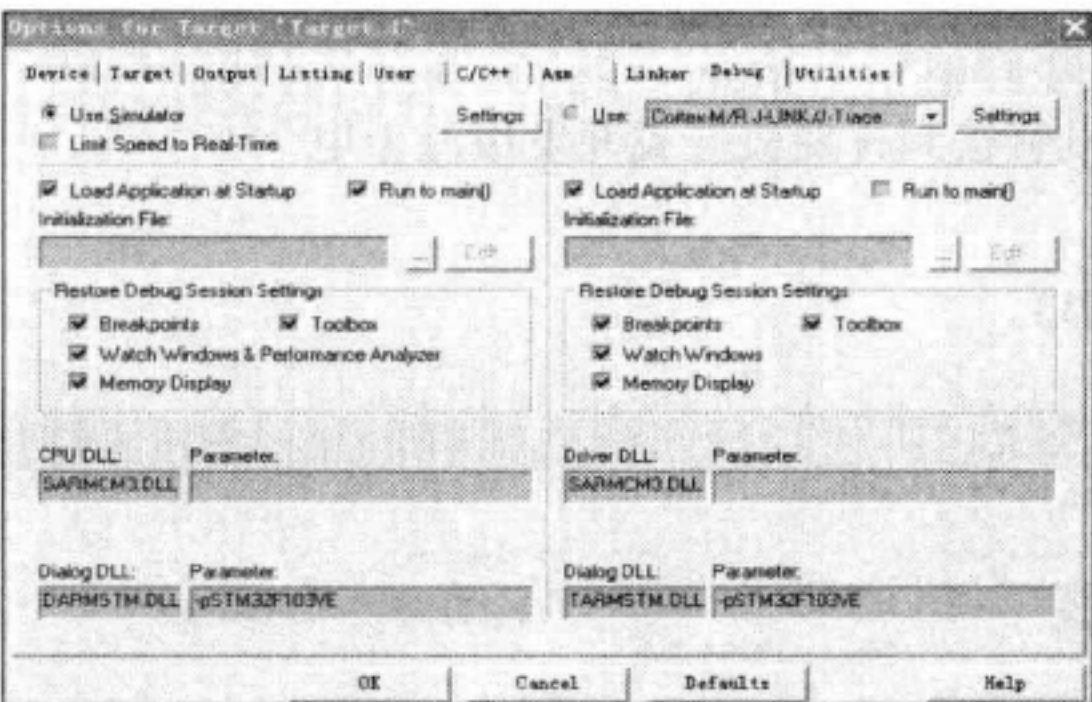


图 3-28 Debug 选项卡设置



图 3-29 编写 main 文件内容


小结：学会新建工程是进行后续程序开发的一个非常重要的工作，在建立工程时需要注意的是：

- 1) 因为我们用的是 ST 官方的新版本库，与编译器自带的库会存在不兼容性，所以我们需要修改库的搜索路径。
- 2) 这个工程我们是设置成软件仿真，如果是用开发板加 J-LINK 调试的话，还需要在开发环境中做修改。

有关 J-LINK 硬件仿真和下载的设置 3.2.2 节中讲解。如果还出现问题的话请参考光盘中的例子：工程模板 3.5。

3.2.2 配置 J-LINK 硬件调试

使用 J-LINK 可以在我们真正的硬件平台上，进行代码单步运行，这对调试有很大的帮助，实际上，我们开发程序的时候 80% 都是在硬件上调试的。

具体配置如图 3-30、图 3-31 所示：点击，在 Debug 和 Utilities 选项里设置。

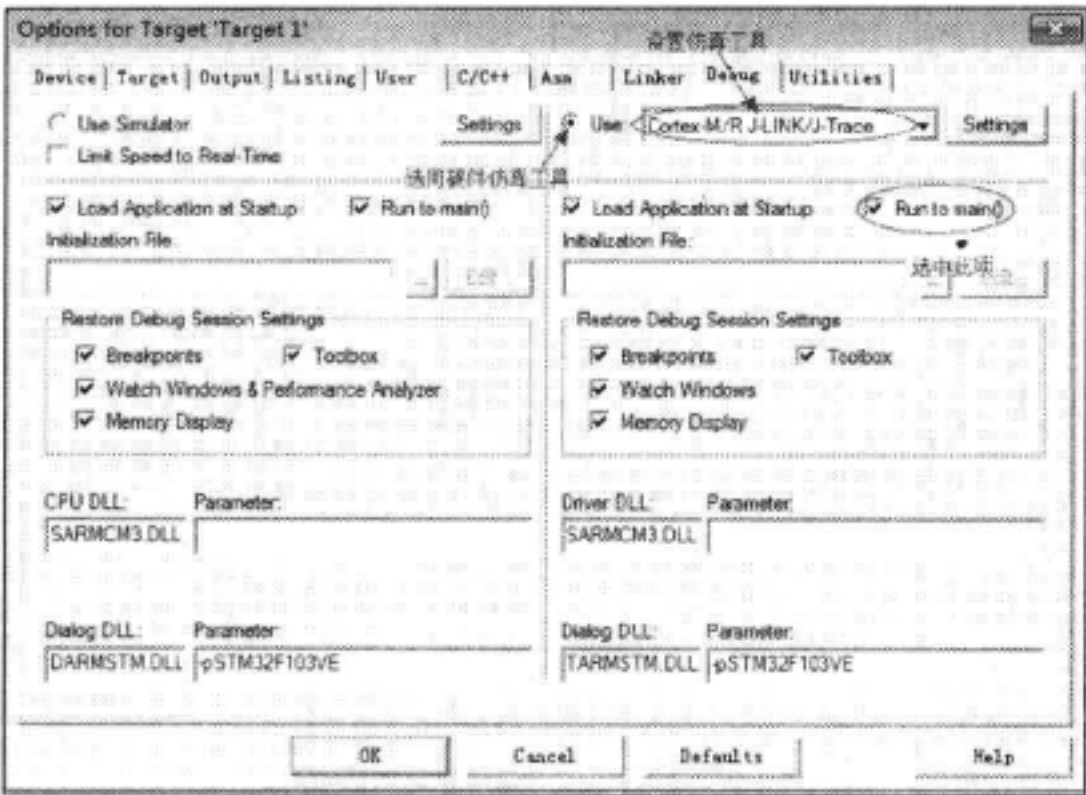


图 3-30 选择硬件仿真



图 3-31 设置仿真工具

初学的朋友经常会遇到 J-LINK 错误，总结如下：

- 1) 提示没有检测到 J-LINK，原因：J-LINK 没有接到开发板上或开发板没上电。
- 2) 没检测到 CPU 的 ID 错误，见图 3-32。
- 3) 没有添加 CPU 支持的 Flash 错误，见图 3-33。

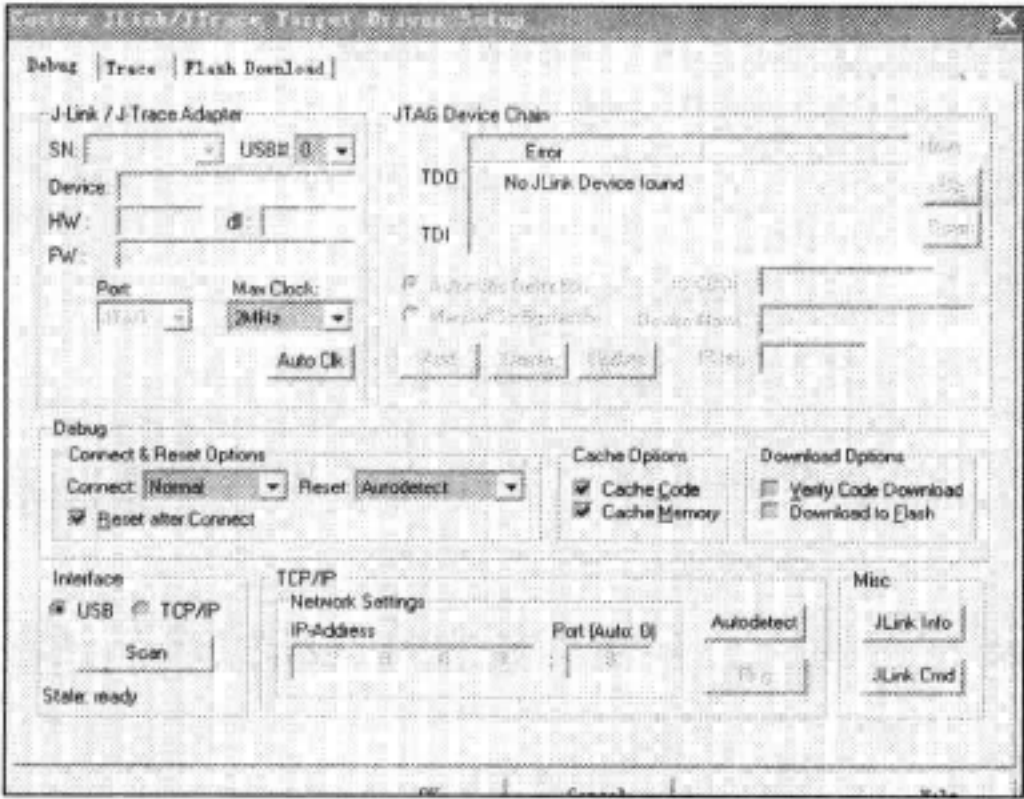


图 3-32 检测不到 CPU 的 ID 错误时的配置界面

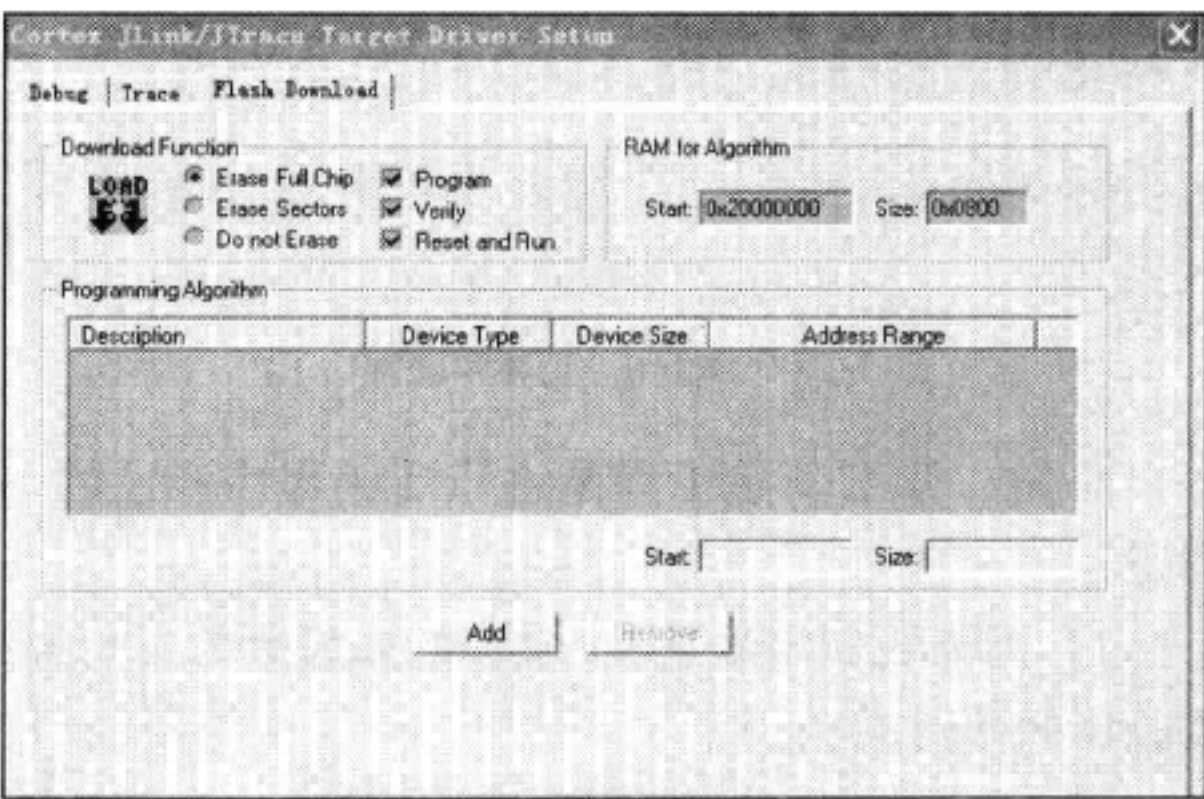


图 3-33 没有添加 CPU 支持 Flash 的错误界面

以下是设置正确，J-LINK 接到开发板且上电时的截图。

- 1) 检测到 CPU 的 ID，见图 3-34。
 - 2) 添加 CPU 支持的 Flash，这一步很重要，否则程序将无法下载，见图 3-35。
- 至此，一个真正完整的工程模板就建成了。

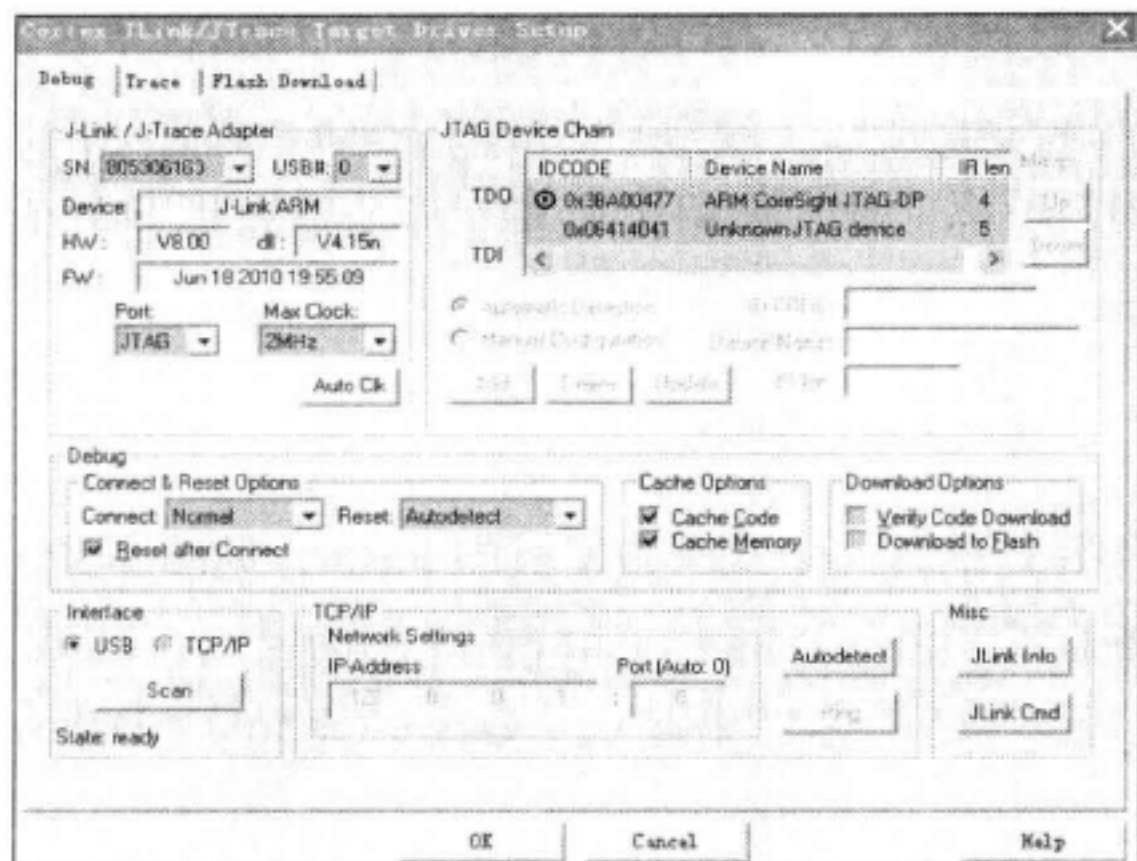


图 3-34 配置正确界面

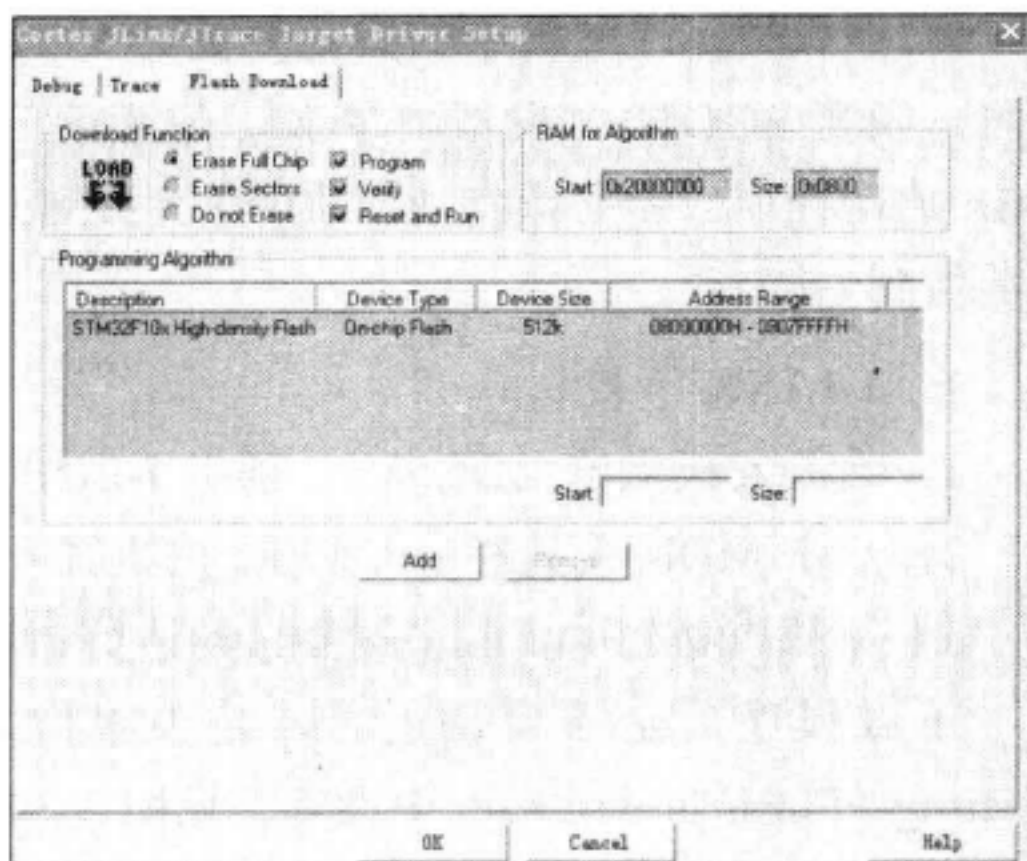


图 3-35 正确添加 Flash 类型时的界面

3.3 如何编译和下载程序

新建完工程模板后，若手头上有一个可以用的工程文件，接下来我们就可以体验一下如何编译和下载程序了，这里使用配套 STM32 V3 开发板自带例程中的 LED 例程进行演示。

3.3.1 如何编译程序

首先打开一个 MDK 工程，在界面左边的工具栏中有 3 个按钮，我们从左向右来介绍一下这 3 个按钮的功能，见图 3-36。

- ❑ Translate 按钮：就是编译当下修改过的文件，即检查一下有没有语法错误，既不链接库文件，也不会生成可执行文件。
- ❑ Build 按钮：就是编译当下修改过的工程，它包含了语法检查、链接动态库文件、生成可执行文件。
- ❑ Rebuild 按钮：即重新编译整个工程，与 Build 这个按钮实现的功能是一样的，但有所不同的是它是重新编译整个工程的所有文件，耗时巨大。

综上：当我们编辑好程序之后，只需要用 Build 按钮就可以，既方便又省时。第一个和第三个按钮用得比较少。

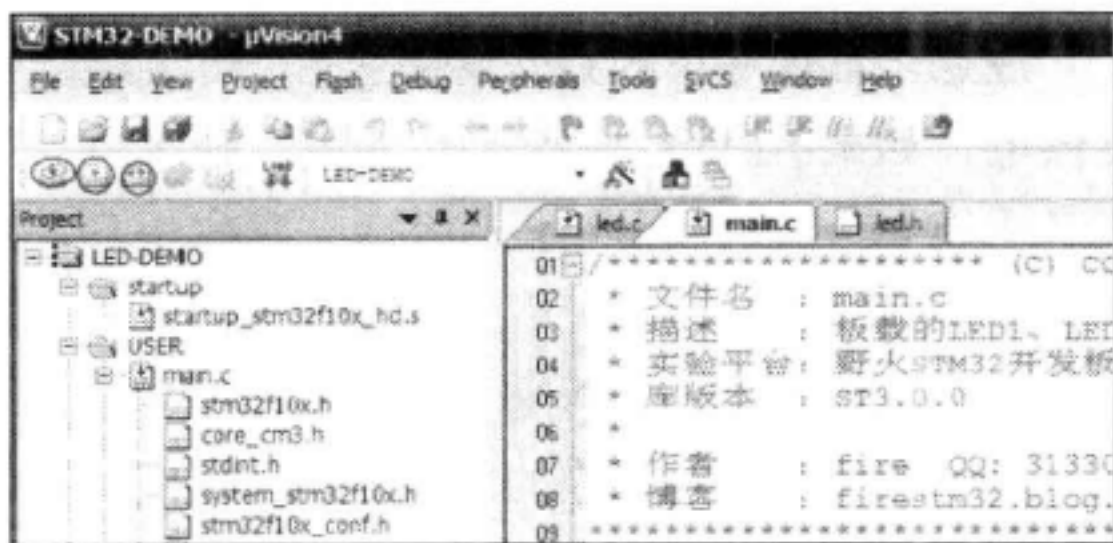


图 3-36 编译程序按钮

3.3.2 如何下载程序

配套 STM32 开发板有两种下载方式，JLINKV8 下载和串口下载（即 ISP）。要注意的是，

J-LINK 下载时，开发板中的拨动开关 BOOT0 既可以拨到 VCC 也可拨到 GND；在用串口下载程序时，须将 BOOT0 开发拨到 VCC。这两种下载方式都是把程序烧写到内部的 Flash，要想从内部 Flash 启动程序，就必须把 BOOT0 开关拨到 GND。所以，在用串口下载完程序后需要把 BOOT0 开关拨到 GND。

1. J-LINK 下载

- 1) 给开发板供电 (DC5V)，插上 J-LINK。
- 2) 点击 MDK 工具栏中的 Load 按钮就可将编译好的程序下载到开发板的 Flash，见图 3-37。
- 3) 下载成功之后，程序就会自动运行。见图 3-38。

下载程序后是否自动运行，是由我们自己设定的，这个在选项：Target Options->Debug->Setting->Flash DownLoad 中设置，见图 3-39。

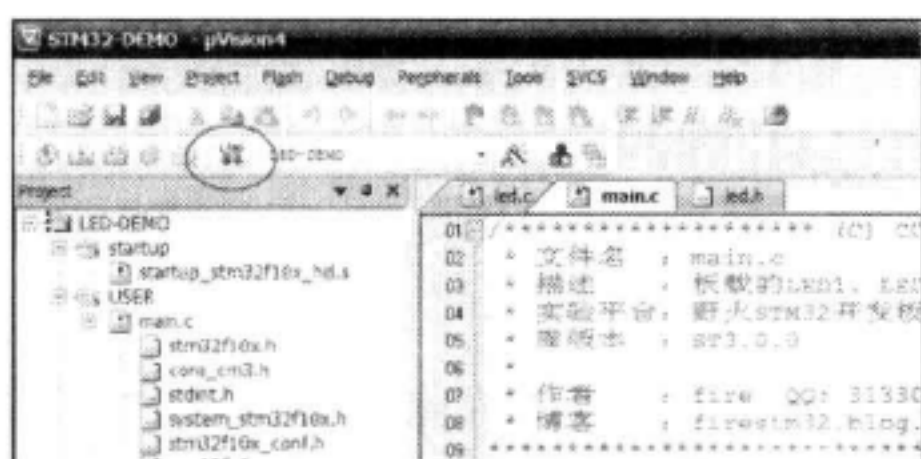


图 3-37 程序下载按钮

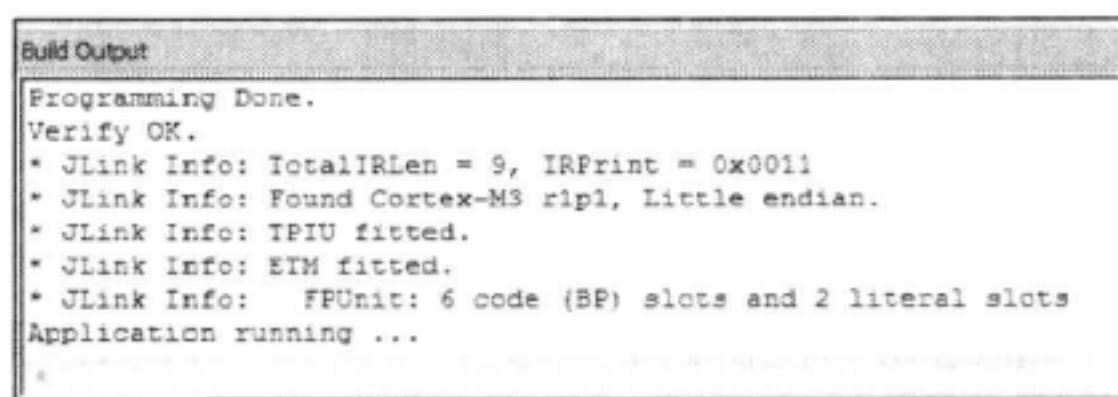


图 3-38 下载成功的 Keil 提示界面

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

注意 在程序下载到开发板之后，开发板要供电，如果 J-LINK 一端连开发板，则另一端必须连 PC，这样程序才能运行。有些用户在下载程序之后，第二次用的时候只是给开发板供电，J-LINK 的一端只连了开发板而没有连 PC，这样程序是不能工作的。要想只在供电的情况下要程序运行，只需把 J-LINK 从开发板中拔掉即可，即只连接电源，不接 J-LINK。

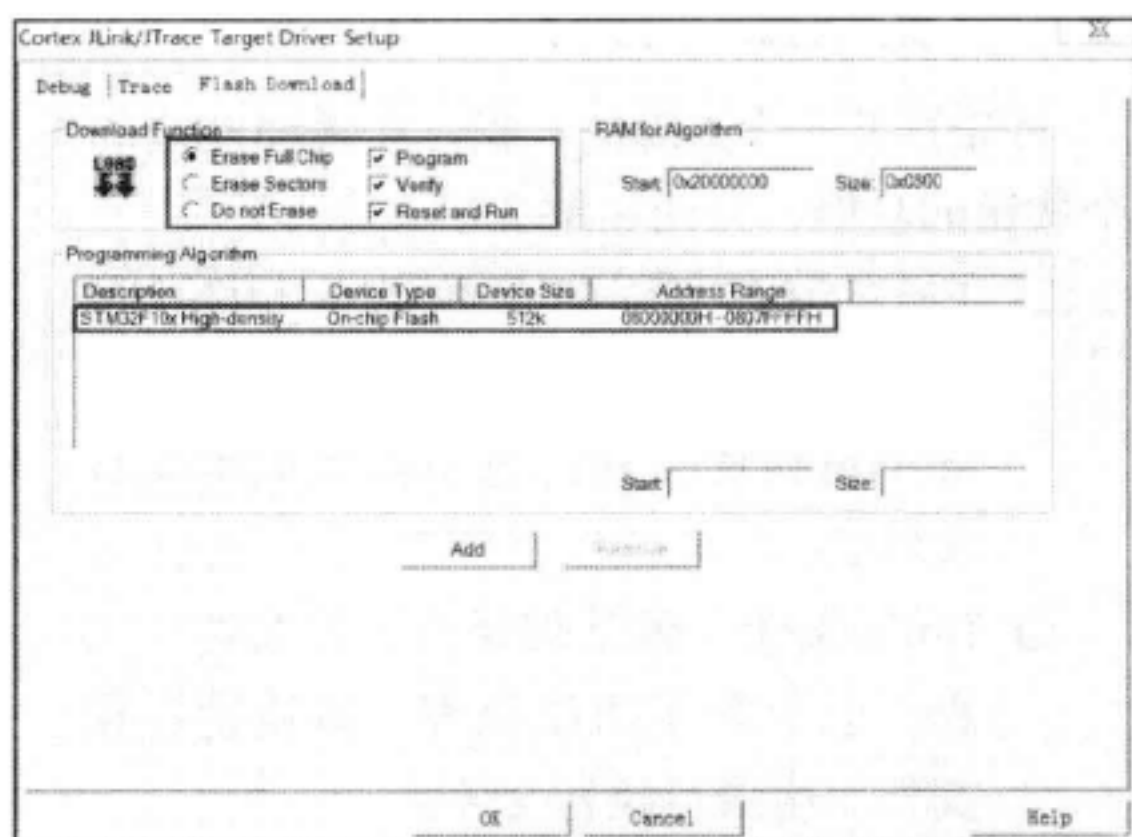


图 3-39 下载后自动运行的设置

2. 串口下载

- 1) 给开发板供电 (DC5V)，拔掉 J-LINK，插上串口线（注意是两头都是母的交叉线），接的是串口 1，串口 2 是下载不了程序的，再将 BOOT0 接到 VCC。
- 2) 在这里我们用的串口下载软件是 mcuisp，这是一个绿色软件，可从网上自由下载，我们也在光盘资料里面提供了这个软件。
- 3) 打开 mcuisp，mcuisp 是很智能的，只要开发板上电且接好了串口，它就会自动搜索串口，这里用的是计算机主板后面的串口，这个串口都会被默认为是串口 1。假如你是笔记本用户，用

的是 USB 转串口，那么端口号可能就不是 COM1，需要自己查看。

4) 设置波特率为 115200。选择要下载的程序。在开发板自带的例程中，可执行文件（hex 文件）都在工程目录下 Output 这个文件下，见图 3-40。

5) 点击“开始编程”按钮，如果程序下载成功后则会打印出如图 3-41 所示右侧框中的信息。

6) 程序下载成功之后，可是在开发板上还看不到实验现象。怎么办，是不是出什么问题了？这是因为我们是通过串口将我们的程序烧写到 Flash 中了，而我们想要从 Flash 里面执行程序的话，就需要将我们的“J-LNK- 串口”下载模式选择开关打到 J-LNK 这个位置，然后按下我们的“复位”按钮就可以看到实验现象了。

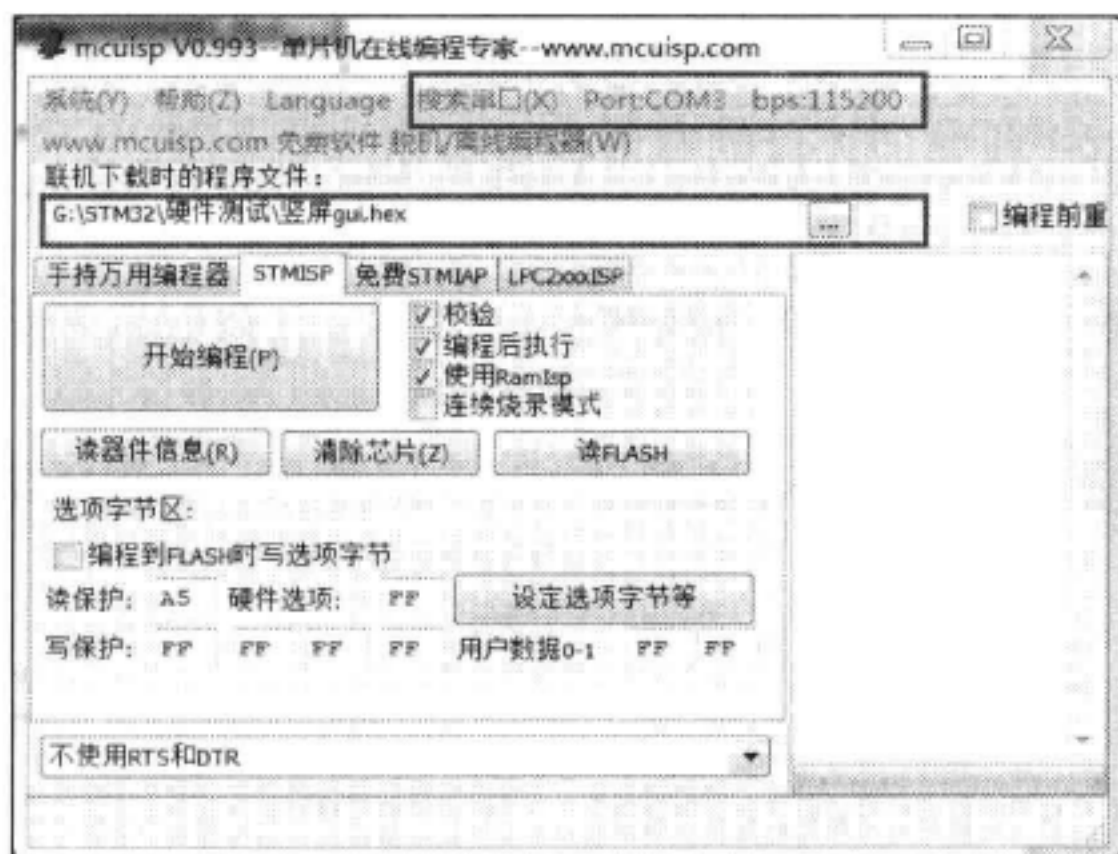


图 3-40 ISP 下载软件

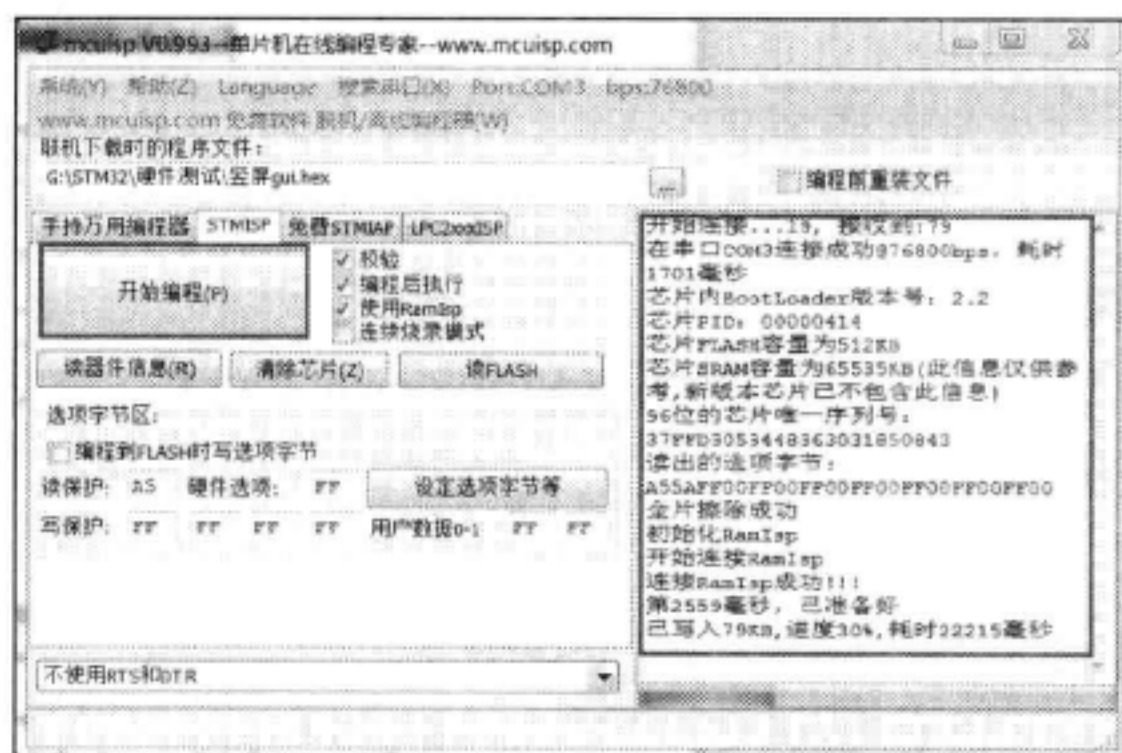


图 3-41 ISP 下载程序

7) 在我们点击“开始编程”按钮时，有时还会出现 mcuisp 一直处于连接的状态，导致程序下载不了，见图 3-42。解决的方法是按一下开发板中的“复位”按钮。

在没有 J-LINK 的情况下，我们可以选择用串口下载。但是用串口下载有一些缺点：下载速度慢、需要多次设置 BOOT0 开关以及不能够进行硬件在线仿真。鉴于这几个缺点我们还是建议用 J-LINK，既可下载又可仿真。

把这个例程烧录进开发板后，至此，我们的点灯仪式就完成了。但是我们还没有写过一句程序，感觉就是在喝稀粥，没什么成就感。别急，下一章将教会你如何磨枪上阵。

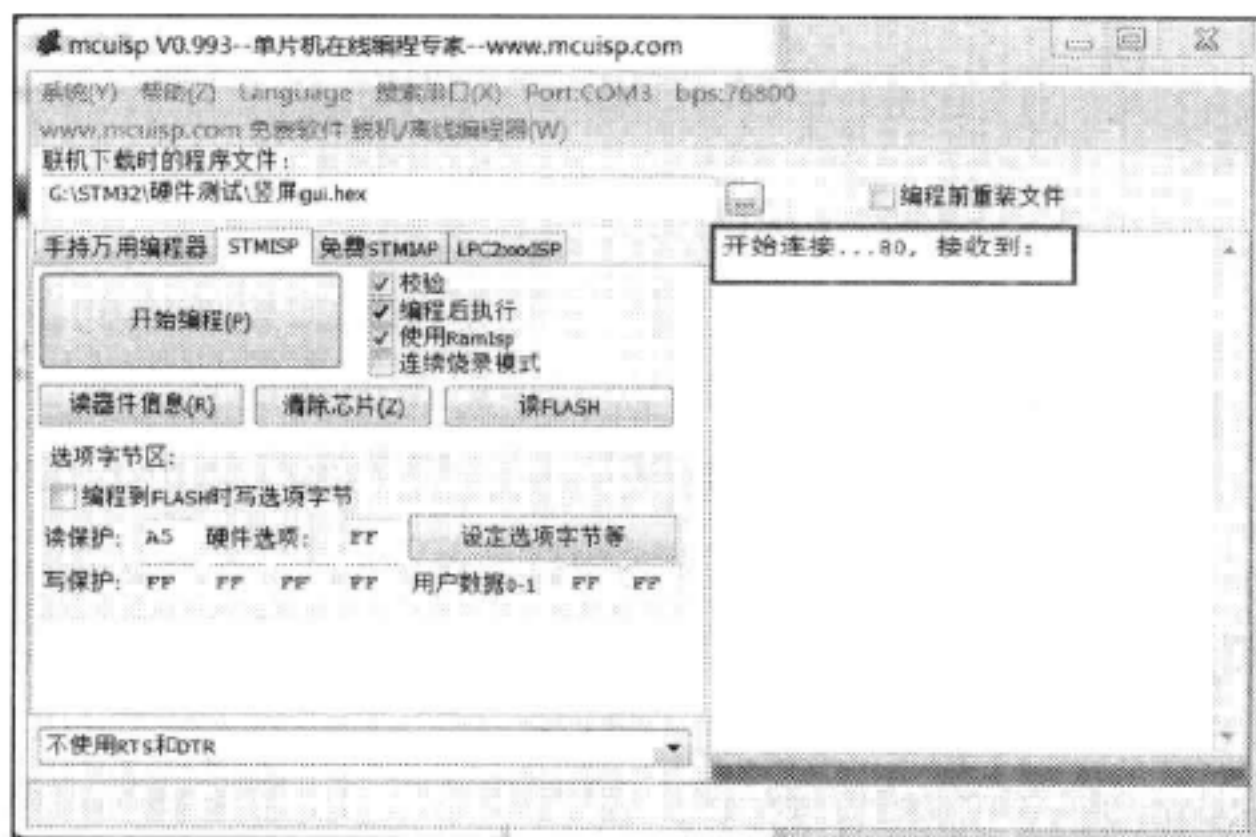
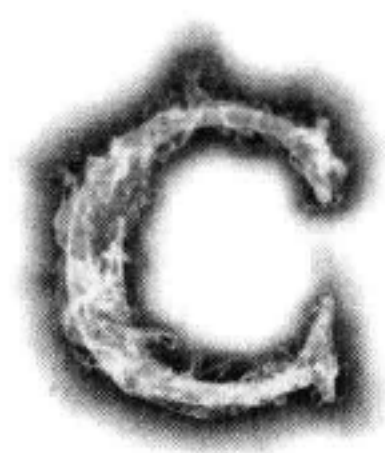


图 3-42 ISP 下载失败界面



第 4 章

深入分析流水灯例程

通过前面两章的学习，读者对库仅仅建立了一个非常模糊的印象。作为第一个 STM32 例程，只有进行足够深入的分析，才能从根本上解除读者对使用库函数的困惑。而且，只要读者利用这个 LED 例程真正领会了库开发的流程以及原理，再进行其他外设的开发就变得相当简单了。

本章的任务是：

- 1) 从 STM32 固件库的实现原理上解答库到底是什么、为什么要用库、用库与直接配置寄存器的区别等问题。
- 2) 让读者了解具体利用库的开发流程，熟悉库函数的结构，达到举一反三的效果，达到所学即所用的效果。

4.1 STM32 的 GPIO

想要控制 LED 灯，当然是通过控制 STM32 芯片的 I/O 引脚电平的高低来实现。在 STM32 芯片上，I/O 引脚可以被软件设置成各种不同的功能，如输入或输出，所以又被称为 GPIO (General-Purpose I/O)。而 GPIO 引脚又被分为 GPIOA、GPIOB、…、GPIOG 不同的组，每组端口分为 0 ~ 15 共 16 个不同的引脚，对于不同型号的芯片，端口的组和引脚的数量不同，具体请参考相应芯片型号的 datasheet。

于是，控制 LED 的步骤就自然整理出来了：

- 1) GPIO 端口引脚多——就要选定需要控制的特定引脚。
- 2) GPIO 功能如此丰富——配置需要的特定功能。
- 3) 控制 LED 的亮和灭——设置 GPIO 输出电压的高低。

继续思考，要控制 GPIO 端口，就要涉及控制相关的寄存器。这时我们就要查一查与 GPIO 相关的寄存器了，可以通过《STM32 参考手册》来查看，见图 4-1。



图 4-1 《STM32 参考手册》的 GPIO 相关目录

图 4-1 中的 7 个寄存器，相应功能在文档上有详细说明。它们可以分为以下 4 类，其功能简要概括如下：

- 1) 配置寄存器：选定 GPIO 的特定功能，最基本的如选择作为输入还是输出端口。
- 2) 数据寄存器：保存了 GPIO 的输入电平或将要输出的电平。
- 3) 位控制寄存器：设置某引脚的数据为 1 或 0，控制输出的电平。
- 4) 锁定寄存器：设置某锁定引脚后，就不能修改其配置。

注意 要想知道其功能严谨、详细的描述，请读者养成习惯，在正式使用时要以官方的 datasheet 为准，在这里只是简单地概括其功能。

关于寄存器名称如 GPIOx_CRL、GPIOx_CRH 上的标号 x，其取值可以为图中括号内的值 (A…E)，表示这些寄存器也跟 GPIO 一样，是分组的。也就是说，对于端口 GPIOA 和 GPIOB，它们都有互不相干的一组寄存器，如控制 GPIOA 的寄存器名为 GPIOA_CRL、GPIOA_CRH 等，而控制 GPIOB 的则是不同的被命名为 GPIOB_CRL、GPIOB_CRH 等的寄存器。

我们的程序代码以配套 STM32 V3 开发板为例，根据其硬件连接图来分析，见图 4-2 和图 4-3。

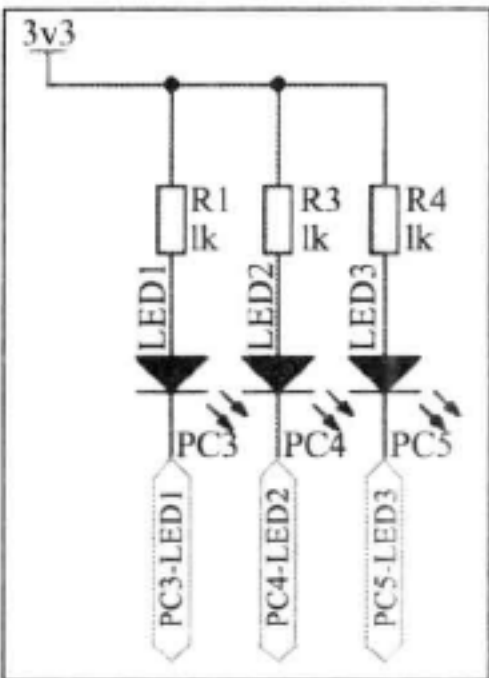


图 4-2 配套开发板 LED 灯原理图

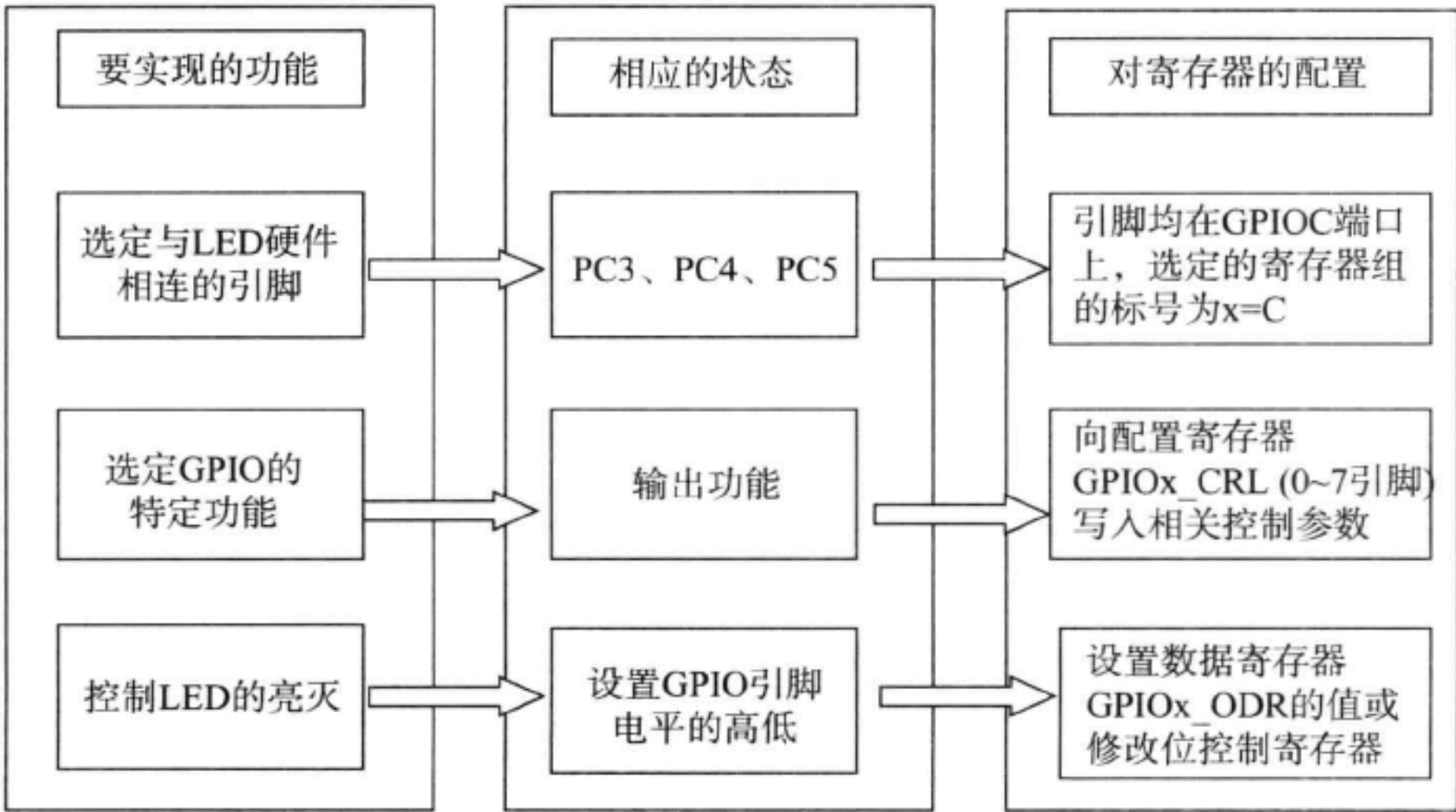


图 4-3 点亮 LED 的方法

从这两个图我们可以知道，STM32 的功能实际上也是通过配置寄存器来实现的。配置寄存器的具体参数，需要参考《STM32 参考手册》的寄存器说明，见表 4-1。

表 4-1 《STM32 参考手册》的端口配置高寄存器（GPIOx_CRH）说明

端口配置高寄存器（GPIOx_CRH）（x=A..E）															
地址偏移：0x04															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15		MODE15		CNF14		MODE14		CNF13		MODE13		CNF12		MODE12	
[1:0]		[1:0]		[1:0]		[1:0]		[1:0]		[1:0]		[1:0]		[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		MODE11		CNF10		MODE10		CNF9		MODE9		CNF8		MODE8	
[1:0]		[1:0]		[1:0]		[1:0]		[1:0]		[1:0]		[1:0]		[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位 数		描 述													
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），软件通过这些位配置相应的 I/O 端口。 在输入模式（MODE[1:0]=00）时： 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）时： 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													
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 = 8..15），软件通过这些位配置相应的 I/O 端口。 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													

如表 4-1 所示，对于 GPIO 端口，每个端口有 16 个引脚，每个引脚的模式由寄存器的 4 个位控制，每 4 位又分为两位控制引脚配置（CNFy[1:0]），另外两位控制引脚的模式及最高速度（MODEy[1:0]），其中 y 表示第 y 个引脚。这个表是 GPIOx_CRH 寄存器的说明，配置 GPIO 引脚模式的一共有两个寄存器，CRH 是高寄存器，用来配置高 8 位引脚：pin8 ~ pin15；还有一个称为 CRL 寄存器，如果我们要配置 pin0 ~ pin7 引脚，则要在寄存器 CRL 中进行配置。

举例说明对 CRH 寄存器的配置：当将 GPIOx_CRH 寄存器的第 28 至 29 位设置为参数“11”，并将第 30 至 31 位设置为参数“00”时，则把 x 端口第 15 个引脚的模式配置成了“输出的最大速度为 50MHz 的通用推挽输出模式”，其他引脚可通过其 GPIOx_CRH 或 GPIOx_CRL 的其他寄存器位来配置，这里的 x 是指端口 GPIOA 或 GPIOB，取决于不同的寄存器基址，这将在后面分析。

我们先来分析，控制引脚电平高低需要对寄存器进行什么具体的操作，见表 4-2。

表 4-2 《STM32 参考手册》的端口位设置 / 清除寄存器 (GPIOx_BSRR) 说明

端口位设置/清除寄存器 (GPIOx_BSRR) (x=A..E)															
地址偏移: 0x10															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR	BR	BR	BR	BR	BR	BR	BR	BR	BR	BR	BR	BR	BR	BR	BR
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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
BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
位 数	描 述														
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 注意: 如果同时设置了 BSy 和 BRy 的对应位, BSy 位起作用														
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														

由寄存器说明表可知, 一个引脚 y 的输出数据由 GPIOx_BSRR 寄存器位的两个位来控制, 分别为 BRy (Bit Reset y) 和 BSy (Bit Set y), BRy 位用于写 1 清零, 使引脚输出低电平, BSy 位用来写 1 置 1, 使引脚输出高电平。而对这两个位进行写零都是无效的。(还可以通过设置寄存器 ODR 来控制引脚的输出。)

例如: 对 x 端口的寄存器 GPIOx_BSRR 的第 0 位 (BS0) 进行写 1, 则 x 端口的第 0 引脚被设置为 1, 输出高电平, 若要令第 0 引脚再输出低电平, 则需要向 GPIOx_BSRR 的第 16 位 (BR0) 写 1。

4.2 STM32 的地址映射

4.2.1 温故而知新——stm32f10x.h 文件

首先请大家回顾一下在 51 单片机上点亮 LED 是怎样实现的。这很简单, 用几行代码就完成了, 见代码清单 4-1。

代码清单 4-1 使用 51 单片机点亮 LED 灯

```

1. #include<reg52.h>
2. int main (void)
3. {
4.     P0=0;
5.     while(1);
6. }
```


以上代码就可以点亮 P0 端口与 LED 阴极相连的 LED 灯了，当然，这里省略了启动代码。为什么这个“P0 =0”句子就能控制 P0 端口为低电平？很多刚入门 51 单片机的读者还真解释不来，关键之处在于这个代码所包含的头文件 <reg52.h>。该文件部分内容见代码清单 4-2。

代码清单 4-2 <reg 52.h> 头文件中的地址映射

```
1. /* BYTE Registers */
2. sfr P0      = 0x80;
3. sfr P1      = 0x90;
4. sfr P2      = 0xA0;
5. sfr P3      = 0xB0;
6. sfr PSW     = 0xD0;
7. sfr ACC     = 0xE0;
8. sfr B       = 0xF0;
9. sfr SP      = 0x81;
10. sfr DPL     = 0x82;
11. sfr DPH     = 0x83;
12. sfr PCON   = 0x87;
13. sfr TCON   = 0x88;
14. sfr TMOD   = 0x89;
15. sfr TL0    = 0x8A;
16. sfr TL1    = 0x8B;
17. sfr TH0    = 0x8C;
18. sfr TH1    = 0x8D;
19. sfr IE     = 0xA8;
20. sfr IP     = 0xB8;
21. sfr SCON   = 0x98;
22. sfr SBUF   = 0x99;
```

这些定义被称为地址映射，见图 4-4。

所谓地址映射，就是将芯片上的存储器甚至 I/O 等资源与地址建立一一对应的关系。如果某地址对应着某寄存器，我们就可以运用 C 语言的指针来寻址并修改这个地址上的内容，从而实现修改该寄存器的内容。

正是因为 <reg52.h> 头文件中有了对于各种寄存器和 I/O 端口的地址映射，我们才可以在 51 单片机程序中方便地使用“P0 =0xFF”、“TMOD =0xFF”等赋值句子对寄存器进行配置，从而控制单片机。

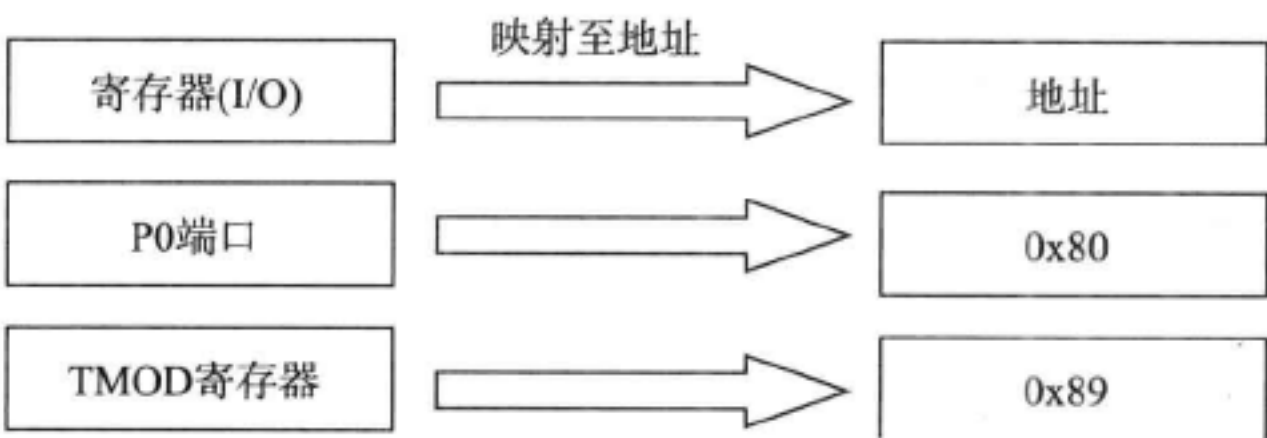


图 4-4 寄存器映射

Cortex-M3 的地址映射也是类似的。Cortex-M3 有 32 根地址线，所以它的寻址空间大小为 2³² bit=4 GB。ARM 公司设计时，预先把这 4 GB 的寻址空间大致地分配好了。它把从 0x40000000 至 0x5FFFFFFF（512 MB）的地址分配给片上外设。通过把片上外设的寄存器映射到这个地址区，就可以简单地以访问内存的方式，访问这些外设的寄存器，从而控制外设的工作。这样，片上外设可以使用 C 语言来操作。CM3 存储器映射见图 4-5。

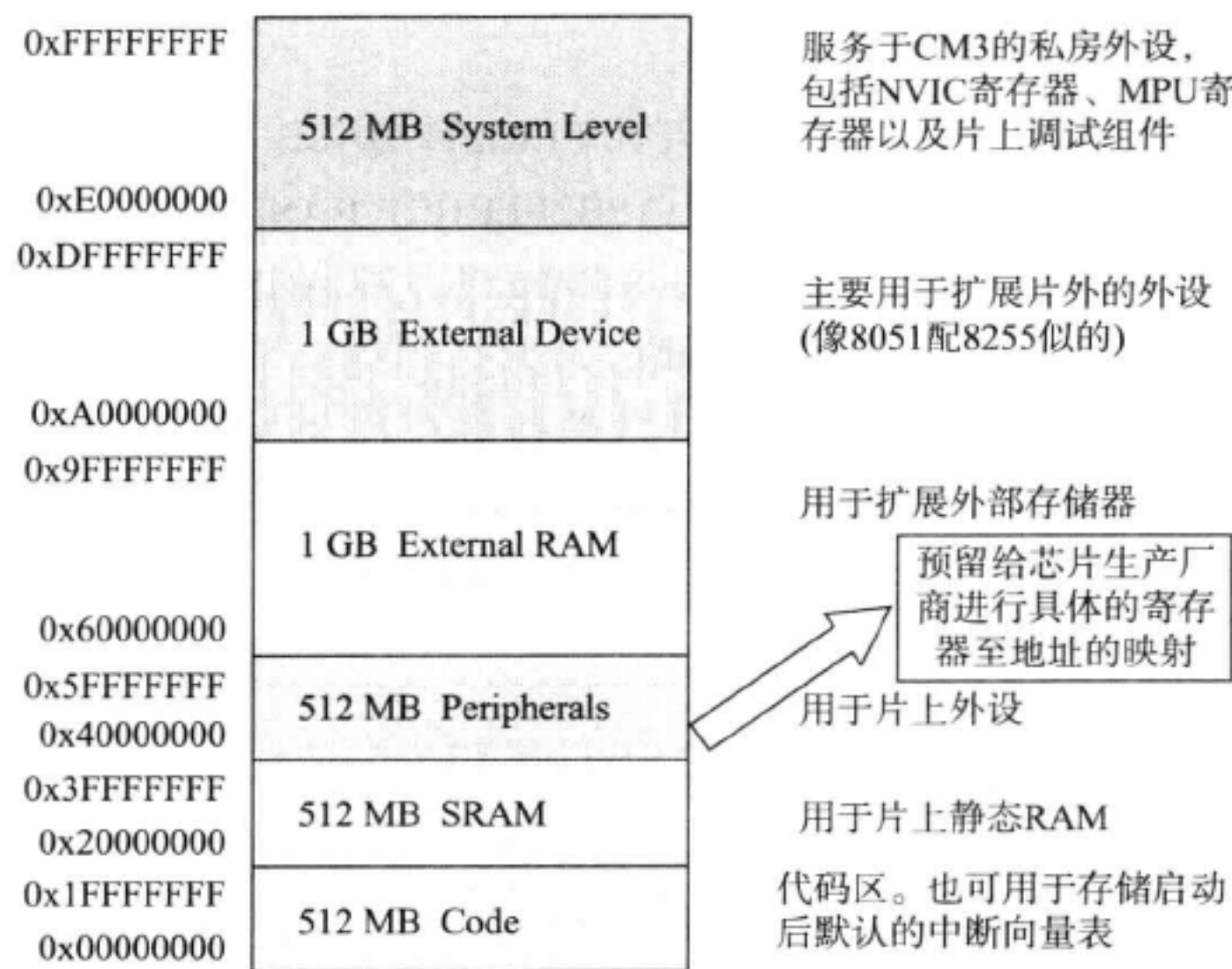


图 4-5 CM3 寻址空间映射

stm32f10x.h 这个文件中重要的内容就是把 STM32 的所有寄存器进行地址映射。如同 51 单片机的 <reg52.h> 头文件一样，stm32f10x.h 像一个大表格，我们在使用的时候就是通过宏定义进行类似查表的操作，大家想象一下没有这个文件的话，我们要怎样访问 STM32 的寄存器？有什么缺点？

不进行这些宏定义的缺点有：

- 1) 地址容易写错。
- 2) 我们需要查大量的手册来确定哪个地址对应哪个寄存器。
- 3) 看起来还不好看，且容易造成编程的错误，效率低，影响开发进度。

当然，这些工作都是由 ST 的固件工程师来完成的，只有设计 CM3 的人才是最了解 CM3 的，才能写出完美的库。

在这里我们以外接了 LED 灯的外设 GPIOC 为例，在这个文件中一系列宏实现了地址映射，见代码清单 4-3。

代码清单 4-3 stm32f10x.h 文件中对 GPIO 寄存器的地址映射

```
1. #define GPIOC_BASE (APB2PERIPH_BASE + 0x1000)
2. #define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
3. #define PERIPH_BASE ((uint32_t)0x40000000)
```

这几个宏定义是从文件中的几个部分抽离出来的，具体的内容读者可参考 stm32f10x.h 源码。

4.2.2 外设基地址

首先看到 PERIPH_BASE 这个宏，宏展开为 0x40000000，并把它强制转换为 uint32_t 的 32 位类型数据，这是因为 STM32 的地址是 32 位的，是不是觉得 0x40000000 这个地址很熟？是的，这是 Cortex-M3 核分配给片上外设 512MB 寻址空间中的第一个地址，我们把 0x40000000 称为外设基地址。

4.2.3 总线外设基地址

接下来是宏 APB2PERIPH_BASE，宏展开为 PERIPH_BASE（外设基地址）加上偏移地址 0x10000，即指向的地址为 0x40010000。这个 APB2PERIPH_BASE 宏是什么地址呢？STM32 不同的外设是挂载在不同的总线上的，见图 4-6。STM32 芯片有 AHB 总线、APB2 总线和 APB1 总线，挂载在这些总线上的外设有一定的地址范围。

其中像 GPIO、串口 1、ADC 及部分定时器是挂载在称为 APB2 的总线上，挂载到 APB2 总线上的外设地址空间是从 0x40010000 至 0x40013FFF 地址。这里的第一个地址，也就是 0x40010000，称为 APB2PERIPH_BASE（APB2 总线外设基地址）。

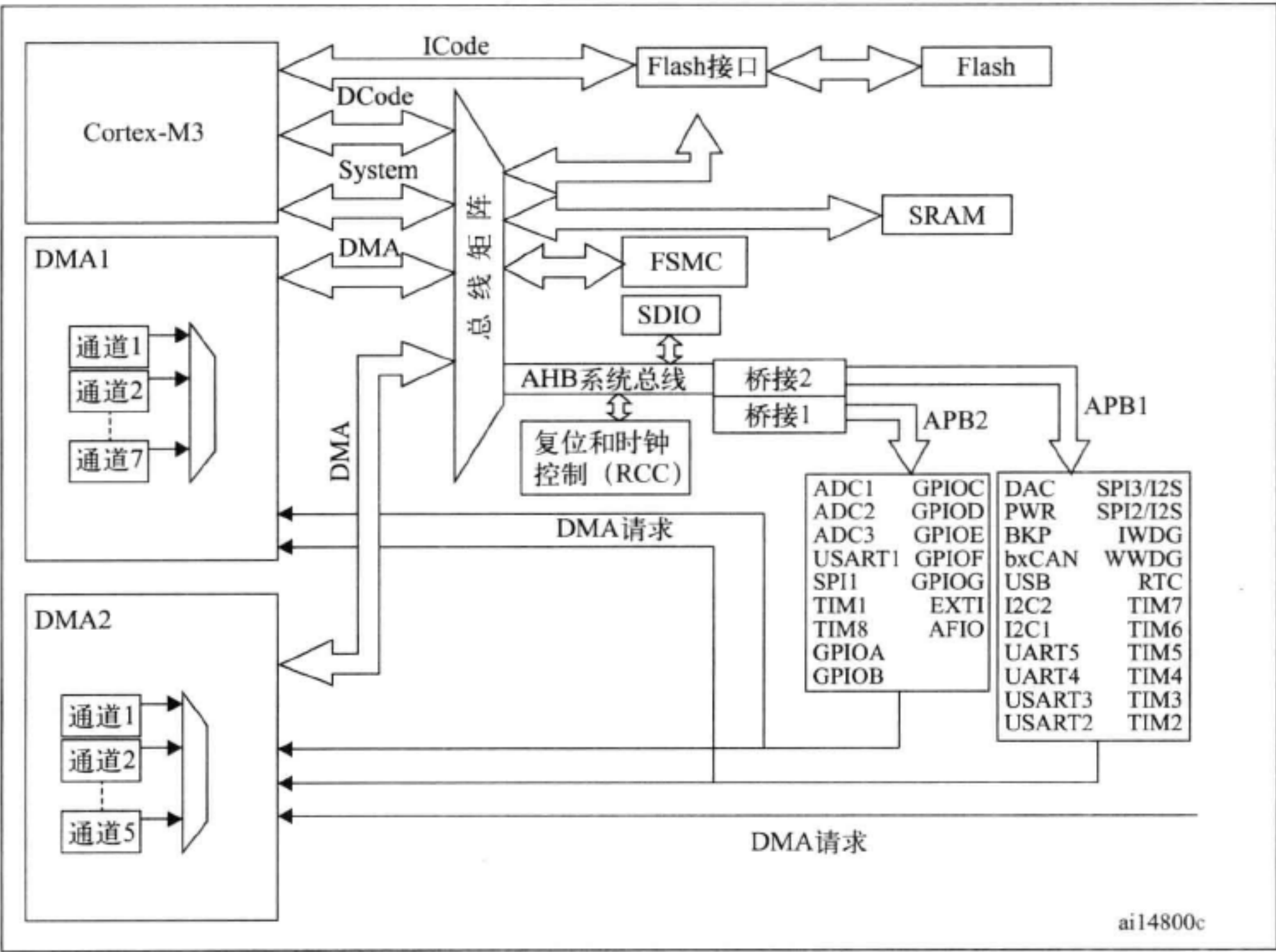


图 4-6 STM32 各外设与总线的关系

而 APB2 总线基地址相对于外设基地址的偏移量为 0x10000 个地址，即为 APB2 相对外设基地址的偏移地址，见表 4-3。

表 4-3 基地址、偏移量

地址范围	总线	总线基地址	总线基地址相对外设基地址的偏移量
0x4001 8000 - 0x5003FFFF	AHB	0x40018000	0x18000
0x4001 0000 - 0x40017FFF	APB2	0x40010000	0x10000
0x4000 0000 - 0x4000FFFF	APB1	0x40000000	0x00000

由这个表我们可以知道，stm32f10x.h 这个文件中必然含有用于定义总线外设基地址的宏，见代码清单 4-4。

代码清单 4-4 APB1 外设基地址的宏

1. #define APB1PERIPH_BASE	PERIPH_BASE
----------------------------	-------------

因为偏移量为零，所以 APB1 的地址直接就等于外设基地址。

4.2.4 寄存器组基地址

最后到了宏 GPIOC_BASE，宏展开为 APB2PERIPH_BASE（APB2 总线外设的基地址）加上相对 APB2 总线外设基地址的偏移量 0x1000 得到了 GPIOC 端口的寄存器组的基地址。这个所谓的寄存器组又是什么呢？它包括什么寄存器？

细看 stm32f10x.h 文件，我们还可以发现有关各个 GPIO 基地址的宏，见代码清单 4-5。

代码清单 4-5 GPIO 基地址宏

1. #define GPIOA_BASE	(APB2PERIPH_BASE + 0x0800)
2. #define GPIOB_BASE	(APB2PERIPH_BASE + 0x0C00)
3. #define GPIOC_BASE	(APB2PERIPH_BASE + 0x1000)
4. #define GPIOD_BASE	(APB2PERIPH_BASE + 0x1400)

除了 GPIOC 寄存器组的地址，还有 GPIOA、GPIOB 和 GPIOD 的地址，并且这些地址是不一样的。前面提到，每组 GPIO 都对应着独立的一组寄存器，查看 STM32 的 datasheet，看到寄存器说明如表 4-4 所示。

表 4-4 《STM32 参考手册》的端口配置寄存器（GPIOx_CRH）说明

端口配置高寄存器（GPIOx_CRH）（x=A..E）															
地址偏移：0x04															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15	MODE15		CNF14	MODE14		CNF13	MODE13		CNF12	MODE12					
[1:0]	[1:0]		[1:0]	[1:0]		[1:0]	[1:0]		[1:0]	[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	MODE11		CNF10	MODE10		CNF9	MODE9		CNF8	MODE8					
[1:0]	[1:0]		[1:0]	[1:0]		[1:0]	[1:0]		[1:0]	[1:0]					
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

注意到这个说明中有一个偏移地址：0x04，这里的偏移地址是相对哪个地址的偏移呢？下面进行举例说明。

对于 GPIOC 组的寄存器，GPIOC 含有的端口配置高寄存器（GPIOC_CRH）地址为：GPIOC_BASE+0x04。假如是 GPIOA 组的寄存器，则 GPIOA 含有的端口配置高寄存器（GPIOA_CRH）地址为：GPIOA_BASE+0x04。

也就是说，这个偏移地址，就是该寄存器相对所在寄存器组基地址的偏移量。

于是，读者可能会想，大概这个文件含有一个类似如下的宏：

```
1. #define GPIOC_CRH (GPIOC_BASE + 0x04)
```

这个宏定义了 GPIOC_CRH 寄存器的具体地址，然而在 stm32f10x.h 文件中并没有这样的宏。ST 公司的工程师采用了更巧妙的方式来确定这些地址，请看下一节。

4.3 STM32 固件库对寄存器的封装

ST 的工程师用结构体的形式封装了寄存器组，C 语言结构体学得不好的读者，可以在这里补补课了。在 stm32f10x.h 文件中，关于 GPIO 的封装见代码清单 4-6。

代码清单 4-6 stm32f10x.h 文件中对 GPIO 的封装

1. #define GPIOA	((GPIO_TypeDef *) GPIOA_BASE)
2. #define GPIOB	((GPIO_TypeDef *) GPIOB_BASE)
3. #define GPIOC	((GPIO_TypeDef *) GPIOC_BASE)

有了这些宏，我们就可以定位到具体的寄存器地址，在这里发现了一个陌生的类型 GPIO_TypeDef，追踪它的定义，可以在 stm32f10x.h 文件中找到一段代码，见代码清单 4-7。

代码清单 4-7 GPIO_TypeDef 的定义

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

其中 __IO 也是一个 ST 官方库定义的宏，其宏定义展开见代码清单 4-8。

代码清单 4-8 __O 和 __IO 的定义

1. #define __O	volatile /*!< defines 'write only' permissions */
2. #define __IO	volatile /*!< defines 'read / write' permissions */

volatile 是 C 语言的一个关键字，不了解的话可以暂时忽略，详见第 11 章。

回到 GPIO_TypeDef 这段代码，这个代码用 typedef 关键字声明了名为 GPIO_TypeDef 的结构体类型，结构体内又定义了 7 个 __IO uint32_t 类型的变量。这些变量都是 32 位，即每个变量占内存空间 4 个字节。在 C 语言中，结构体内变量的存储空间是连续的，也就是说假如我们定义了一个 GPIO_TypeDef，这个结构体的首地址（变量 CRL 的地址）若为 0x40011000，那么结构体中第二个

变量（CRH）的地址即为 0x40011000 + 0x04，加上的 0x04 正是代表 4 个字节地址的偏移量。

细心的读者会发现，这个 0x04 偏移量正是 GPIOx_CRH 寄存器相对于所在寄存器组的偏移地址，见图 4-7。同理，GPIO_TypeDef 结构体内其他变量的偏移量，也与相应的寄存器偏移地址相符。于是，只要我们匹配了结构体的首地址，就可以确定各寄存器的具体地址了。

有了这些准备，就可以分析本节的第一段代码了，请读者重新看看代码清单 4-6，GPIOA_BASE 在上一节已解析，是一个代表 GPIOA 组寄存器的基地址。

“(GPIO_TypeDef *)”在这里的作用则是把 GPIOA_BASE 地址转换为 GPIO_TypeDef 结构体指针类型。

有了这样的宏，以后我们写代码的时候，如果要修改 GPIO 的寄存器，就可以用代码清单 4-9 的方式来实现。代码分析见注释。

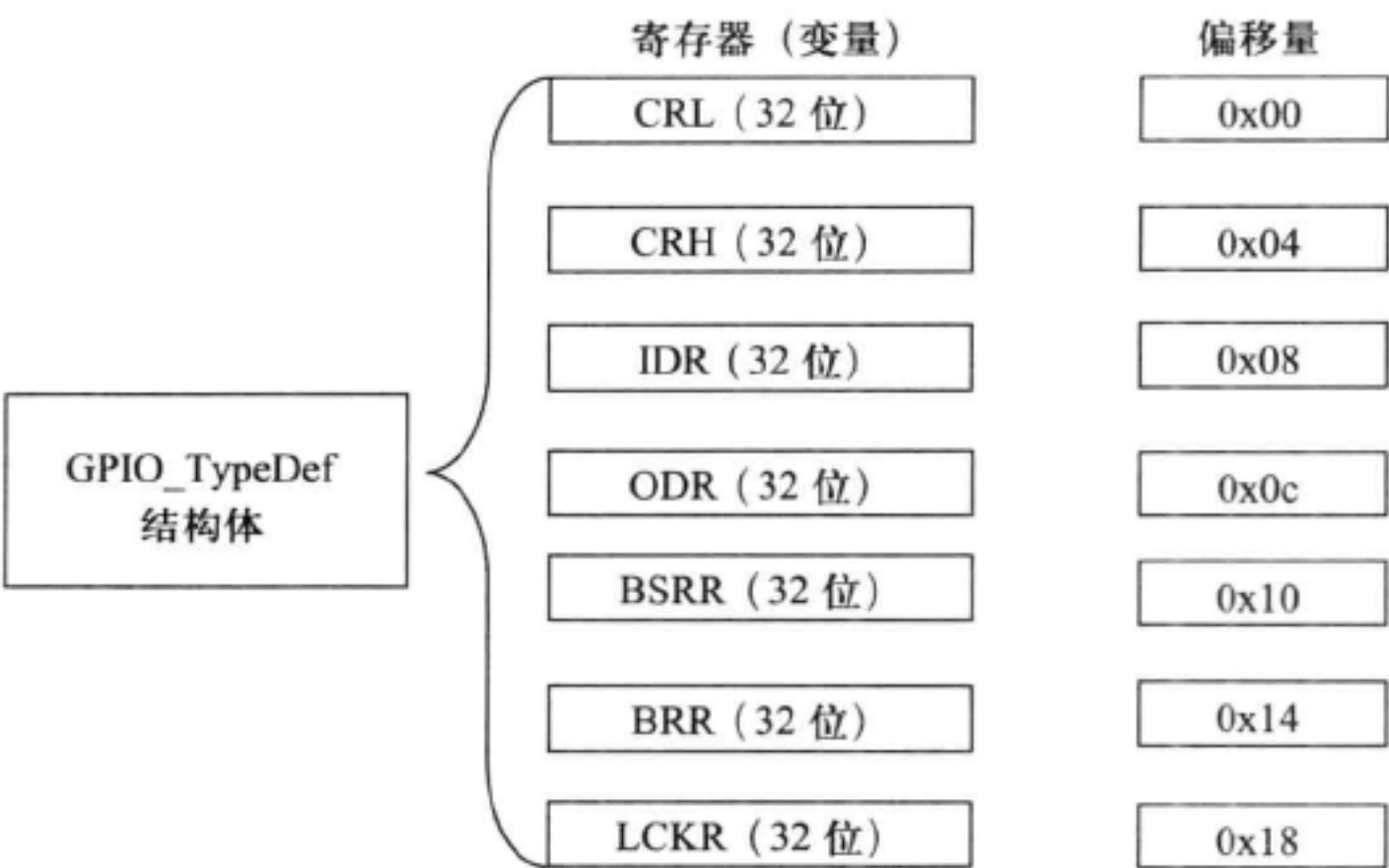


图 4-7 GPIO_TypeDef 结构体成员偏移量

代码清单 4-9 修改 GPIO 寄存器

```
1. GPIO_TypeDef * GPIOx;           // 定义一个 GPIO_TypeDef 型结构体指针 GPIOx
2. GPIOx = GPIOA;                  // 把指针地址设置为宏 GPIOA 地址
3. GPIOx->CRL = 0xffffffff;         // 通过指针访问并修改 GPIOA_CRL 寄存器
```

通过类似的方式，我们就可以给具体的寄存器写上适当的参数以控制 STM32 了。是不是觉得很巧妙？但这只是库开发的皮毛，而且实际上我们并不是这样使用库的，库为我们提供了更简单的开发方式。STM32 的库可谓尽情绽放了 C 的魅力，如果你是单片机初学者、C 语言初学者，那么请你不要放弃与 STM32 库邂逅的机会。

4.4 STM32 的时钟系统

STM32 芯片为了实现低功耗，设计了一个功能完善但却非常复杂的时钟系统。普通的 MCU 一般只要配置好 GPIO 的寄存器就可以使用了，但 STM32 还有一个步骤，就是开启外设时钟。

4.4.1 时钟树 & 时钟源

首先，让我们从整体上了解 STM32 的时钟系统。具体见图 4-8。

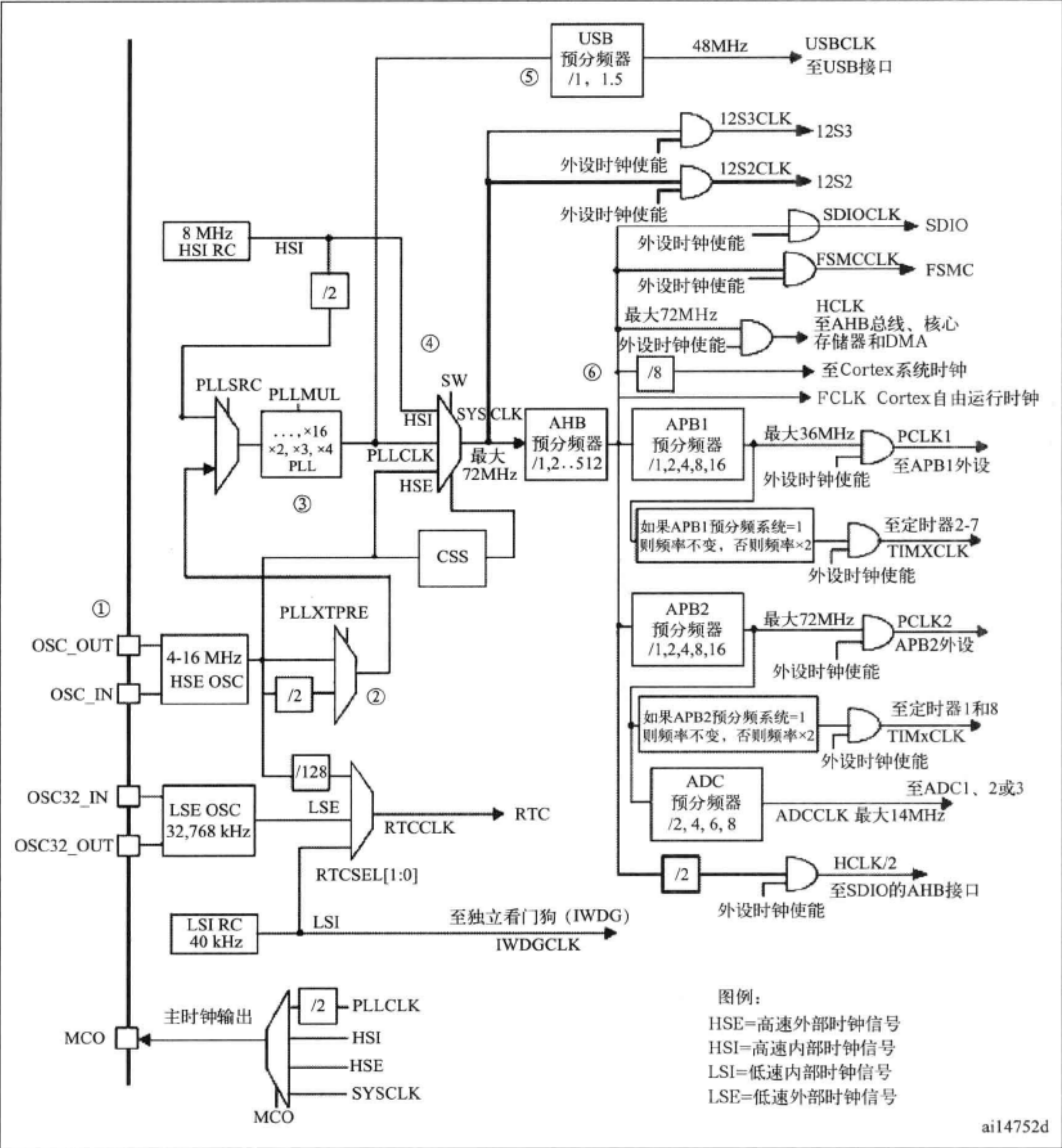


图 4-8 STM32 时钟树

这个图说明了 STM32 的时钟走向，从图的左边开始，从时钟源一步步分配到外设时钟。

从时钟频率来说，又分为高速时钟和低速时钟，高速时钟是提供给芯片主体的主时钟，而低速时钟只是提供给芯片中的 RTC（实时时钟）及独立看门狗使用。

从芯片角度来说，时钟源分为内部时钟与外部时钟源，内部时钟是由芯片内部 RC 振荡器产生的，起振较快，所以时钟在芯片刚上电的时候，默认使用内部高速时钟。而外部时钟信号是由

外部的晶振输入的，在精度和稳定性上都有很大优势，所以上电之后我们再通过软件配置，转而采用外部时钟信号。

所以，STM32 有以下 4 个时钟源：

1) 高速外部时钟 (HSE)：以外部晶振作为时钟源，晶振频率可取范围为 4 ~ 16 MHz，我们一般采用 8 MHz 的晶振。

2) 高速内部时钟 (HSI)：由内部 RC 振荡器产生，频率为 8 MHz，但不稳定。

3) 低速外部时钟 (LSE)：以外部晶振作为时钟源，主要提供给实时时钟模块，所以一般采用 32.768 kHz。配套 STM32 实验板上用的是 32.768 kHz、6p 负载规格的晶振。

4) 低速内部时钟 (LSI)：由内部 RC 振荡器产生，也主要提供给实时时钟模块，频率大约为 40 kHz。

4.4.2 高速外部时钟

我们以最常用的高速外部时钟 (HSE) 为例分析，首先假定我们在外部提供的晶振的频率为 8 MHz。

1) 从左端的 OSC_OUT 和 OSC_IN 开始，这两个引脚分别接到外部晶振的两端。

2) 8 MHz 的时钟遇到了第一个分频器 PLLXTPRE (HSE divider for PLL entry)，在这个分频器中，可以通过寄存器配置，选择它的输出。它的输出时钟可以是对输入时钟的二分频或不分频。本例中，我们选择不分频，所以经过 PLLXTPRE 后，还是 8MHz 的时钟。

3) 8MHz 的时钟遇到开关 PLLSRC (PLL entry clock source)，我们可以选择其输出，输出为外部高速时钟 (HSE) 或是内部高速时钟 (HSI)。这里选择输出为 HSE，接着遇到锁相环 PLL，具有倍频作用，在这里我们可以输入倍频因子 PLLMUL (PLL multiplication factor)。经过 PLL 的时钟称为 PLLCLK。倍频因子我们设定为 9 倍频，也就是说，经过 PLL 之后，我们的时钟从原来 8MHz 的 HSE 变为 72 MHz 的 PLLCLK。

4) 紧接着又遇到了一个开关 SW，经过这个开关之后就是 STM32 的系统时钟 (SYSCLK) 了。通过这个开关，可以切换 SYSCLK 的时钟源，可以选择 HSI、PLLCLK 或 HSE。我们选择 PLLCLK 时钟，所以 SYSCLK 就为 72 MHz 了。

5) PLLCLK 在输入到 SW 前，还流向了 USB 预分频器，这个分频器输出为 USB 外设的时钟 (USBCLK)。

6) 回到 SYSCLK，SYSCLK 经过 AHB 预分频器，分频后再输入到其他外设。如输出到称为 HCLK、FCLK 的时钟，还直接输出到 SDIO 外设的 SDIOCLK 时钟、存储器控制器 FSMC 的 FSMCCLK 时钟，以及作为 APB1、APB2 的预分频器的输入端。本例设置 AHB 预分频器不分频，即输出的频率为 72 MHz。

7) GPIO 外设是挂载在 APB2 总线上的，APB2 的时钟是 APB2 预分频器的输出，而 APB2 预分频器的时钟来源是 AHB 预分频器。因此，把 APB2 预分频器设置为不分频，我们就可以得到 GPIO 外设的时钟也等于 HCLK，即 72 MHz。

4.4.3 HCLK、FCLK、PCLK1、PCLK2

从时钟树的分析，看到经过一系列的倍频、分频后得到了几个与我们开发密切相关的时钟。

1) SYSCLK：系统时钟，是 STM32 大部分器件的时钟来源，主要由 AHB 预分频器分配到各个部件。

2) HCLK：由 AHB 预分频器直接输出得到，它是高速总线 AHB 的时钟信号，提供给存储器、DMA 及 Cortex 内核，是 Cortex 内核运行的时钟，CPU 主频就是这个信号，它的大小与 STM32 运算速度、数据存取速度密切相关。

3) FCLK：同样由 AHB 预分频器输出得到，是内核的“自由运行时钟”。“自由”表现在它不来自时钟 HCLK，因此在 HCLK 时钟停止时 FCLK 也继续运行。它的存在可以保证，在处理器休眠时也能够采样到中断和跟踪休眠事件，它与 HCLK 互相同步。

4) PCLK1：外设时钟，由 APB1 预分频器输出得到，最大频率为 36 MHz，提供给挂载在 APB1 总线上的外设。

5) PCLK2：外设时钟，由 APB2 预分频器输出得到，最大频率可为 72 MHz，提供给挂载在 APB2 总线上的外设。

为什么 STM32 的时钟系统如此复杂？因为有倍频、分频以及一系列外设时钟的开关。需要倍频是考虑到电磁兼容性，如果外部直接提供一个 72 MHz 的晶振，太高的振荡频率可能会给制作电路板带来一定的难度。分频是因为 STM32 既有高速外设又有低速外设，各种外设的工作频率不尽相同，如同 PC 上的南北桥，把高速和低速的设备分开来管理。最后，每个外设都配备了外设时钟的开关，当我们不使用某个外设时，可以把这个外设时钟关闭，从而降低 STM32 的整体功耗。所以，当我们使用外设时，一定要记得开启外设的时钟。

4.5 LED 具体代码分析

有了以上 STM32 存储器映像、时钟系统，以及基本的库函数知识，我们就可以分析 LED 例程的代码了。

4.5.1 实验描述及工程文件清单

1. 实验描述

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

2. 硬件连接

☐ PC3 – LED1

☐ PC4 – LED2

☐ PC5 – LED3

3. 用到的库文件

3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c

4. 用户编写的文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/led.c 及 led.h

4.5.2 配置工程环境

LED 实验中用到了 GPIO 和 RCC（用于设置外设时钟）这两个片上外设，所以在操作 I/O 之前我们需要把关于这两个外设的库文件添加到工程模板之中。它们分别为 stm32f10x_gpio.c 和 stm32f10x_rcc.c 文件。其中 stm32f10x_gpio.c 用于操作 I/O，而 stm32f10x_rcc.c 用于配置系统时钟和外设时钟，由于每个外设都要配置时钟，所以它是每个外设都需要用到的库文件。

在添加完这两个库文件之后立即编译的话会出错，因为每个外设库对应于一个 stm32f10x_XXX.c 文件的同时还对应着一个 stm32f10x_XXX.h 头文件，头文件包含了相应外设的 C 语言函数实现的声明，只有把相应的头文件也包含进工程才能够使用这些外设库。在库中有一个专门的文件 stm32f10x_conf.h 来管理所有库的头文件，stm32f10x_conf.h 源码见代码清单 4-10。

代码清单 4-10 stm32f10x_conf.h 文件源码

```

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

```

```

19./* #include "stm32f10x_sdio.h" */
20./* #include "stm32f10x_spi.h" */
21./* #include "stm32f10x_tim.h" */
22./* #include "stm32f10x_usart.h" */
23./* #include "stm32f10x_wwdg.h" */
24./* #include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
on to CMSIS functions) */

```

这是没有修改过的代码，默认情况下所有外设的头文件包含都被注释掉了。当我们需要用到某个外设驱动时直接把相应的注释去掉即可，非常方便。如本 LED 实验中我们用到了 RCC 和 GPIO 这两个外设，所以我们应取消其注释，使第 13、17 行的代码生效，修改后如代码清单 4-11 所示。

代码清单 4-11 修改后的 stm32f10x_conf.h 文件

```

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

```

到这里，我们就可以用库自带的函数来操作 I/O 口了，这时我们可以编译一下，会发现没有警告和错误。

4.5.3 编写用户文件

前期工程环境设置完毕，接下来就可以专心编写自己的应用程序了。我们把应用程序放在 USER 这个文件夹下，这个文件夹下至少包含了 main.c、stm32f10x_it.c 和 xxx.c 这三个源文件。其中 main 函数就位于 main.c 这个 C 文件中，只是用来测试我们的应用程序。stm32f10x_it.c 为我们提供了 STM32 所有中断函数的入口，默认情况下这些中断服务程序都为空，需要用户自己编

写。所以现在我们把 stm32f10x_it.c 包含到 USER 这个目录就可以了。

而 xxx.c 就是由用户编写的文件，xxx 是应用程序的名字，用户可自由命名。我们把应用程序的具体实现放在了文件之中，程序的实现和应用分别在不同的文件中，这样可以实现很好的封装性。本书的例程都严格遵从这个规则，每个外设的用户文件都由独立的源文件与头文件构成，这样可以更方便地实现代码重用了。

于是，我们在工程中新建两个文件，分别为 led.c 和 led.h，保存在 USER 目录下，并把 led.c 添加到工程之中。led.c 文件中输入代码清单 4-12。

代码清单 4-12 led.c 文件代码

```

1. /***** (C) COPYRIGHT 2012 WildFire Team *****/
2.  * 文件名   : led.c
3.  * 描述     : led 应用函数库
4.  * 实验平台 : 野火 STM32 开发板
5.  * 硬件连接 : -----
6.  *          |   PC3 - LED1   |
7.  *          |   PC4 - LED2   |
8.  *          |   PC5 - LED3   |
9.  *          | -----
10. * 库版本   : ST3.5.0
11. * 作者     : wildfire team
12. * 论坛     : www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
13. * 淘宝     : http://firestm32.taobao.com
14. *****/
15. #include "led.h"
16.
17. /*
18.  * 函数名 : LED_GPIO_Config
19.  * 描述   : 配置 LED 用到的 I/O 口
20.  * 输入   : 无
21.  * 输出   : 无
22.  */
23. void LED_GPIO_Config(void)
24. {
25.     /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
26.     GPIO_InitTypeDef GPIO_InitStructure;
27.
28.     /* 开启 GPIOC 的外设时钟 */
29.     RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE);
30.
31.     /* 选择要控制的 GPIOC 引脚 */
32.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5;
33.
34.     /* 设置引脚模式为通用推挽输出 */
35.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
36.
37.     /* 设置引脚速率为 50MHz */
38.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
39.

```



```

40.    /* 调用库函数, 初始化 GPIOC */
41.    GPIO_Init(GPIOC, &GPIO_InitStructure);
42.
43.    /* 关闭所有 LED 灯 */
44.    GPIO_SetBits(GPIOC, GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
45. }
46.
47.
48. /***** (C) COPYRIGHT 2012 WildFire Team *****/END OF FILE*****/

```

在这个文件中, 我们定义了一个函数 `LED_GPIO_Config()`, 在这个函数里, 实现了所有为点亮 LED 的配置。

4.5.4 初始化结构体——GPIO_InitTypeDef 类型

在代码清单 4-12 的 `LED_GPIO_Config()` 函数中, 即文件的第 26 行有这样的代码: “`GPIO_InitTypeDef GPIO_InitStructure;`” 这是利用库, 定义了一个名为 `GPIO_InitStructure` 的结构体, 结构体类型为 `GPIO_InitTypeDef`。 `GPIO_InitTypeDef` 类型与前面介绍的库对寄存器的封装类似, 是库文件利用关键字 `typedef` 定义的新类型。追踪其定义原型, 知道它位于 `stm32f10x_gpio.h` 文件中, 见代码清单 4-13。

代码清单 4-13 GPIO_InitTypeDef 类型定义

```

1. typedef struct
2. {
3.     uint16_t GPIO_Pin;           /* 指定将要进行配置的 GPIO 引脚 */
4.     GPIO_Speed_TypeDef GPIO_Speed; /* 指定 GPIO 引脚可输出的最高频率 */
5.     GPIOMode_TypeDef GPIO_Mode;  /* 指定 GPIO 引脚将要配置成的工作状态 */
6. }GPIO_InitTypeDef;

```

于是我们知道, `GPIO_InitTypeDef` 类型的结构体有三个成员, 分别为 `uint16_t` 类型的 `GPIO_Pin`、`GPIO_Speed_TypeDef` 类型的 `GPIO_Speed` 以及 `GPIOMode_TypeDef` 类型的 `GPIO_Mode`。

`uint16_t` 类型的 `GPIO_Pin` 为我们将要选择配置的引脚, 在 `stm32f10x_gpio.h` 文件中有关于 `GPIO_Pin` 的宏定义, 见代码清单 4-14。

代码清单 4-14 GPIO 引脚宏定义

```

1. #define GPIO_Pin_0      ((uint16_t)0x0001) /*!< Pin 0 selected */
2. #define GPIO_Pin_1      ((uint16_t)0x0002) /*!< Pin 1 selected */
3. #define GPIO_Pin_2      ((uint16_t)0x0004) /*!< Pin 2 selected */
4. #define GPIO_Pin_3      ((uint16_t)0x0008) /*!< Pin 3 selected */

```

这些宏的值, 就是允许我们给结构体成员 `GPIO_Pin` 赋的值, 如我们给 `GPIO_Pin` 赋值为宏 `GPIO_Pin_0`, 表示我们选择了 GPIO 端口的第 0 个引脚, 在后面会通过一个函数把这些宏的值进行处理, 设置相应的寄存器, 实现我们对 GPIO 端口的配置。如 `led.c` 代码中的第 32 行, 意思是我们将要选择 GPIO 的 `Pin3`、`Pin4`、`Pin5` 引脚进行配置。

GPIO_Speed_TypeDef 和 GPIOMode_TypeDef 又是两个库定义的新类型, GPIO_Speed_TypeDef 原型见代码清单 4-15。

代码清单 4-15 GPIO_Speed_TypeDef 类型定义

```

1. typedef enum
2. {
3.     GPIO_Speed_10MHz = 1, // 枚举常量, 值为 1, 代表输出速率最高为 10MHz
4.     GPIO_Speed_2MHz,      // 对不赋值的枚举变量, 自动加 1, 此常量值为 2
5.     GPIO_Speed_50MHz      // 常量值为 3
6. }GPIO_Speed_TypeDef;

```

这是一个枚举类型, 定义了 3 个枚举常量, 即 GPIO_Speed_10MHz=1、GPIO_Speed_2MHz=2、GPIO_Speed_50MHz=3。这些常量可用于标识 GPIO 引脚可以配置成的各个最高速度。所以我们在为结构体中的 GPIO_Speed 赋值的时候, 就可以直接用这些含义清晰的枚举标识符了。如代码清单 4-12 中的第 38 行, 给 GPIO_Speed 赋值为 3, 意思是使其最高频率可达到 50 MHz。

同样, GPIOMode_TypeDef 也是一个枚举类型定义符, 原型见代码清单 4-16。

代码清单 4-16 GPIOMode_TypeDef 枚举类型定义

```

1. typedef enum
2. { GPIO_Mode_AIN = 0x0,           // 模拟输入模式
3.     GPIO_Mode_IN_FLOATING = 0x04, // 浮空输入模式
4.     GPIO_Mode_IPD = 0x28,         // 下拉输入模式
5.     GPIO_Mode_IPU = 0x48,         // 上拉输入模式
6.     GPIO_Mode_Out_OD = 0x14,      // 开漏输出模式
7.     GPIO_Mode_Out_PP = 0x10,      // 通用推挽输出模式
8.     GPIO_Mode_AF_OD = 0x1C,       // 复用功能开漏输出
9.     GPIO_Mode_AF_PP = 0x18        // 复用功能推挽输出
10.} GPIOMode_TypeDef;

```

这个枚举类型也定义了很多含义清晰的枚举常量, 是用来帮助配置 GPIO 引脚的模式, 如 GPIO_Mode_AIN 为模拟输入、GPIO_Mode_IN_FLOATING 为浮空输入模式。代码清单 4-12 中的第 35 行是指把引脚设置为通用推挽输出模式。

于是, 我们可以总结 GPIO_InitTypeDef 类型结构体的作用, 即整个结构体包含 GPIO_Pin、GPIO_Speed 和 GPIO_Mode 三个成员, 我们对这三个成员赋予不同的数值可以对 GPIO 端口进行不同的配置, 而这些可配置的数值, 已经由 ST 的库文件封装成见名知义的枚举常量, 这使我们编写代码变得非常简便。

4.5.5 初始化库函数——GPIO_Init()

在前面我们已经接触到 ST 库文件, 以及各种各样由 ST 库定义的新类型, 但所有的这些都只是为库函数服务的。在代码清单 4-12 的第 41 行, 我们用到了第一个用于初始化的库函数 GPIO_Init()。

在我们应用库函数的时候, 只需要知道它的功能、输入什么类型的参数以及允许的参数值就足够了, 这些我们都可以通过查找库帮助文档获得, 详细方法见 2.2.4 节。见图 4-9。

1. 启动文件及 SystemInit() 函数分析

在 startup_stm32f10x_hd.s 启动文件中，有一段启动代码，见代码清单 4-18。

代码清单 4-18 启动代码

```

1. ;Reset_Handler 子程序开始
2. Reset_Handler PROC
3.
4. ; 输出子程序 Reset_Handler 到外部文件
5.                EXPORT Reset_Handler                [WEAK]
6.
7. ; 从外部文件引入 __main 函数
8.                IMPORT __main
9.
10. ; 从外部文件引入 SystemInit 函数
11.               IMPORT SystemInit
12.
13. ; 把 SystemInit 函数调用地址加载到通用寄存器 r0
14.               LDR    R0, =SystemInit
15.
16. ; 跳转到 r0 中保存的地址执行程序 (调用 SystemInit 函数)
17.               BLX    R0
18.
19. ; 把 main 函数调用地址加载到通用寄存器 r0
20.               LDR    R0, =__main
21.
22. ; 跳转到 r0 中保存的地址执行程序 (调用 main 函数)
23.               BX     R0
24.
25. ;Reset_Handler 子程序结束
26.               ENDP

```

注意 这是一段汇编代码，对汇编比较陌生的读者请配以“；”后面的注释来阅读，“；”表示注释其后的单行代码，相当于 C 语言中的“//”和“/* */”。

当芯片被复位（包括上电复位）时，将开始运行这一段代码，运行过程是先调用 SystemInit() 函数，再进入 C 语言中的“__main”（注意与 main 的区别）函数执行，这是一个 C 标准库的初始化函数，执行这个函数后，最终跳转到用户文件中的“main”函数入口，开始运行主程序。

也就是说，在进入 main 函数之前调用了一个名为 SystemInit() 的函数。这个函数的定义在 system_stm32f10x.c 文件之中，它的作用是设置系统时钟 SYSCLK。函数的执行流程是先将与配置时钟相关的寄存器都复位为默认值，复位寄存器后，调用了另外一个函数 SetSysClock()，SetSysClock() 代码见代码清单 4-19。

代码清单 4-19 SetSysClock() 代码

```

1. static void SetSysClock(void)
2. {
3.     #ifdef SYSCLK_FREQ_HSE
4.         SetSysClockToHSE();

```

```

5. #elif defined SYSCLK_FREQ_24MHz
6.     SetSysClockTo24();
7. #elif defined SYSCLK_FREQ_36MHz
8.     SetSysClockTo36();
9. #elif defined SYSCLK_FREQ_48MHz
10.    SetSysClockTo48();
11. #elif defined SYSCLK_FREQ_56MHz
12.    SetSysClockTo56();
13. #elif defined SYSCLK_FREQ_72MHz
14.    SetSysClockTo72();
15. #endif
16.
17. /* If none of the define above is enabled, the HSI is used as System clock
    source (default after reset) */
18.}

```

从 SetSysClock() 代码可以知道，它是根据我们设置的条件编译宏来进行不同的时钟配置的。在 system_stm32f10x.c 文件的开头，已经默认有了条件编译定义，见代码清单 4-20。

代码清单 4-20 关于 SYSCLK_FREQ 的宏定义

```

1. #if defined (STM32F10X_LD_VL) || (defined STM32F10X_MD_VL) || (defined STM32F10X_HD_VL)
2. /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
3. #define SYSCLK_FREQ_24MHz  24000000
4. #else
5. /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
6. /* #define SYSCLK_FREQ_24MHz  24000000 */
7. /* #define SYSCLK_FREQ_36MHz  36000000 */
8. /* #define SYSCLK_FREQ_48MHz  48000000 */
9. /* #define SYSCLK_FREQ_56MHz  56000000 */
10. #define SYSCLK_FREQ_72MHz  72000000
11. #endif

```

在第 10 行定义了 SYSCLK_FREQ_72MHz 条件编译的标识符，所以在 SetSysClock() 函数中将调用 SetSysClockTo72() 函数把芯片的系统时钟 SYSCLK 设置为 72 MHz。当然，前提是输入的外部时钟源 HSE 的振荡频率应为 8 MHz。

其中的 SetSysClockTo72() 函数就是最底层的库函数了，那些与寄存器打交道的工作都是由它来完成的，如果大家想知道我们的系统时钟是如何配置成 72 MHz 的话，可以研究这个函数的源码。但大可不必这样，我们应该抛开传统的跟寄存器打交道来学单片机的方法，而是直接用 ST 的库给我们提供的上层接口，这样会简化很多工作，还能提高开发产品的效率。对这一类直接跟寄存器打交道的函数，在 4.6 节以 GPIO_Init() 函数为例来分析。

注意 3.5 版本的库在启动文件中调用了 SystemInit()，所以不必在 main() 函数中再次调用。但如果使用的是 3.0 版本的库则必须在 main 函数中调用 SystemInit()，以设置系统时钟，因为在 3.0 版本的启动代码中并没有调用 SystemInit() 函数。

2. 开启外设时钟

SYSCLK 由 SystemInit() 配置好了，而 GPIO 所用的时钟 PCLK2 我们采用默认值，也为 72 MHz。

我们采用默认值可以不修改分频器，但外设时钟默认是处在关闭状态的。所以外设时钟一般会在初始化外设的时候设置为开启（根据设计的产品功耗要求，也可以在使用的时候才打开）。开启和关闭外设时钟也有封装好的库函数 `RCC_APB2PeriphClockCmd()`。在 `led.c` 文件中的第 29 行，我们调用了这个函数。查看其使用手册，见图 4-10。

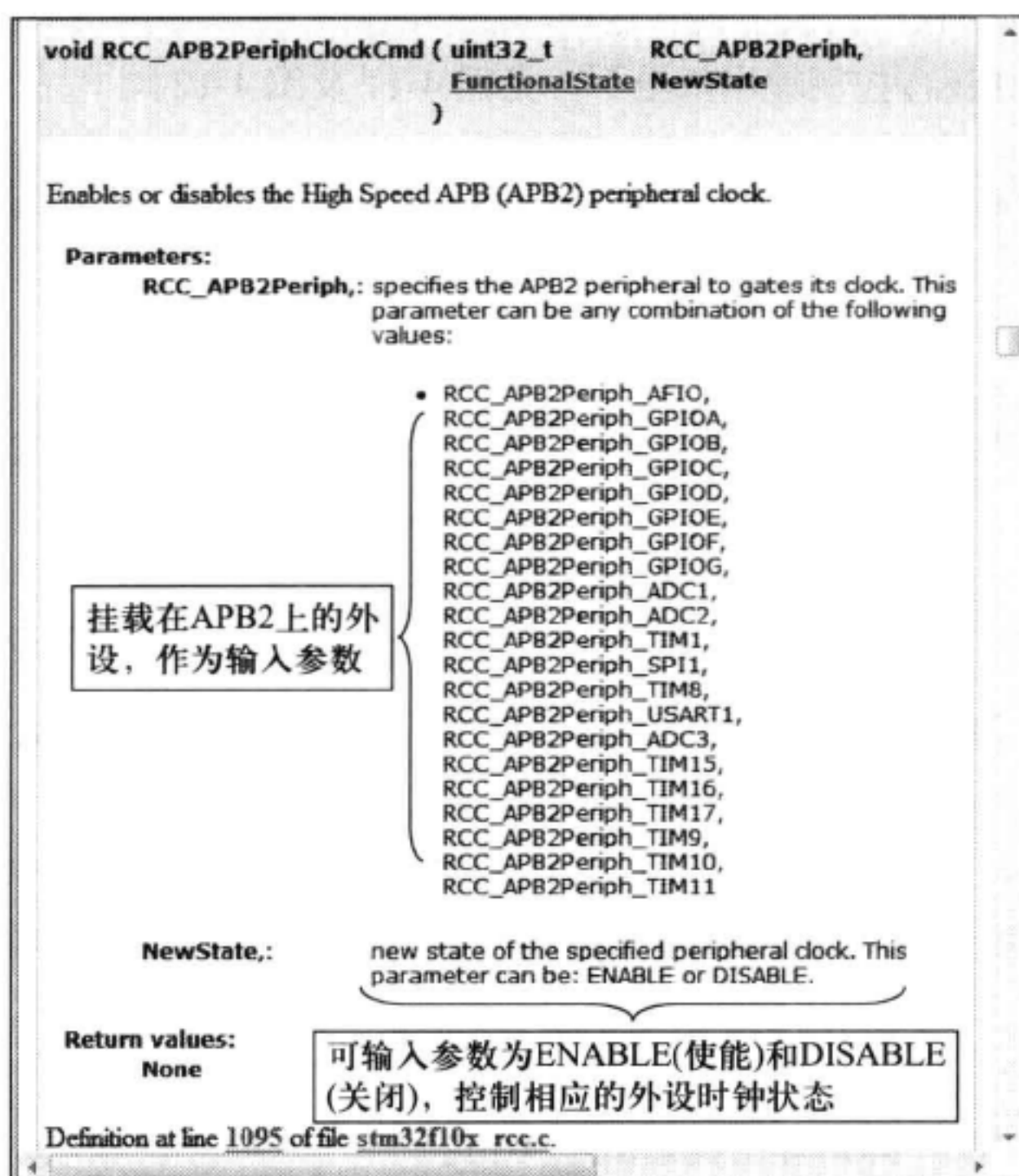


图 4-10 库帮助文档中的 APB2 时钟使能函数说明

调用时需要向它输入两个参数，一个参数为将要控制的挂载在 APB2 总线上的外设时钟，第二个参数为选择要开启还是关闭该时钟。

`led.c` 文件中对它的调用如下：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
```

这表示将要 ENABLE（使能）GPIOC 外设时钟。在这里强调一点，如果我们用到了 I/O 的引脚复用功能，还要开启其复用功能时钟。如 GPIOC 的 Pin4 还可以作为 ADC1 的输入引脚，现在我们把它作为 ADC1 来使用，除了开启 GPIOC 时钟外，还要开启 ADC1 的时钟，如下：

```
RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE);  
RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC1, ENABLE);
```

我们知道有的外设是挂载在高速外设总线 APB2 上，使用 PCLK2 时钟；还有的是挂载在低速外设总线 APB1 上，使用 PCLK1 时钟。既然时钟源是不同的，当然也就由另一个函数来开启 APB1 总线外设的时钟：`RCC_APB1PeriphClockCmd()` 函数。这两个函数正是根据其挂载的总线命名的。可输入的参数自然也就不一样，使用的时候要注意区分。其中所有的 GPIO 都是挂载在 APB2 上的。

4.5.7 控制 I/O 输出高、低电平

前面我们选择好了引脚，配置了其功能及开启了相应的时钟，我们终于可以正式控制 I/O 口的电平高低了，从而实现控制 LED 灯的亮与灭。

前面提到过，要控制 GPIO 引脚的电平高低，只要在 GPIOx_BSRR 寄存器相应的位写入控制参数就可以了。ST 库也为我们提供了具有这样功能的函数，可以分别用 GPIO_SetBits() 控制输出高电平，用 GPIO_ResetBits() 控制输出低电平。见图 4-11 及图 4-12。

```
void GPIO_SetBits ( GPIO_TypeDef * GPIOx,
                    uint16_t      GPIO_Pin
                  )

Sets the selected data port bits.

Parameters:
  GPIOx,:   where x can be (A..G) to select the GPIO peripheral.
  GPIO_Pin,: specifies the port bits to be written. This parameter can be
             any combination of GPIO_Pin_x where x can be (0..15).

Return values:
  None
```

图 4-11 库帮助文档中的 GPIO 引脚置 1 函数说明

```
void GPIO_ResetBits ( GPIO_TypeDef * GPIOx,
                      uint16_t      GPIO_Pin
                    )

Clears the selected data port bits.

Parameters:
  GPIOx,:   where x can be (A..G) to select the GPIO peripheral.
  GPIO_Pin,: specifies the port bits to be written. This parameter can be
             any combination of GPIO_Pin_x where x can be (0..15).

Return values:
  None
```

图 4-12 库帮助文档中的 GPIO 引脚清零函数说明

输入参数有两个，第一个为将要控制的 GPIO 端口：GPIOA…GPIOG，第二个为要控制的引脚号：Pin0 ~ Pin15。

在 led.c 文件的第 44 行 LED_GPIO_Config() 函数中，我们在调用 GPIO_Init() 函数之后就调用了 GPIO_SetBits() 函数，从而让这几个引脚输出高电平，使三盏 LED 初始化后都处于灭状态。

4.5.8 led.h 文件

接下来分析 led.h 文件。其内容见代码清单 4-21。

代码清单 4-21 led.h 文件内容

```
1. #ifndef __LED_H
2. #define __LED_H
3.
4. #include "stm32f10x.h"
5.
6. /* the macro definition to trigger the led on or off
7.  * 1 - off
8.  * 0 - on
9.  */
10. #define ON 0
11. #define OFF 1
12.
13. // 带参宏，可以像内联函数一样使用
14. #define LED1(a) if (a) \
15.                 GPIO_SetBits(GPIOC,GPIO_Pin_3);\
16.                 else \
17.                 GPIO_ResetBits(GPIOC,GPIO_Pin_3)
18.
```

```

19. #define LED2(a) if (a) \
20.             GPIO_SetBits(GPIOC,GPIO_Pin_4);\
21.             else \
22.             GPIO_ResetBits(GPIOC,GPIO_Pin_4)
23.
24. #define LED3(a) if (a) \
25.             GPIO_SetBits(GPIOC,GPIO_Pin_5);\
26.             else \
27.             GPIO_ResetBits(GPIOC,GPIO_Pin_5)
28.
29. void LED_GPIO_Config(void);
30.
31. #endif /* __LED_H */

```

这个头文件的内容不多，但也把它独立成一个头文件，方便以后扩展或移植使用。希望读者养成良好的工程习惯，在写头文件的时候加上类似以下这样的条件编译：

```

#ifndef __LED_H
#define __LED_H
...
#endif

```

这样可以防止头文件重复包含，使得工程的兼容性更好。为什么要加两个下划线“__”？在这里加两个下划线可以避免这个宏标识符与其他定义重名，因为在其他部分代码定义的宏或变量，一般都不会出现这样有下划线的名字。

在 led.h 头文件的部分，首先包含了前面提到的最重要的 ST 库必备头文件 stm32f10x.h。有了它我们才可以使用各种库定义、库函数。

在 led.h 文件的第 14 ~ 27 行，是我们利用 GPIO_SetBits()、GPIO_ResetBits() 库函数编写的带参宏定义，带参宏与 C++ 中的内联函数作用类似。在编译过程中，编译器会把带参宏展开，在相应的位置替换为宏展开代码。其中的反斜杠符号“\”称为续行符，用来连接上下行代码，表示下面一行代码属于“\”所在的代码行，这在 ST 库经常出现。“\”的语法要求极其严格，在它的后面不能有空格、注释等一切“杂物”，在论坛上经常有读者反映遇到编译错误，却不知道正是错在这里。

最后，led.h 文件中的第 29 行代码声明了我们在 led.c 源文件定义的 LED_GPIO_Config() 用户函数。因此，我们要使用 led.c 文件定义的函数时，只要把 led.h 包含到调用该函数的文件中就可以了。

4.5.9 main 文件

写好了 led.c 和 led.h 两个文件，我们控制 LED 灯的驱动程序就全部完成了。接下来，就可以利用写好的驱动文件，在 main 文件中编写应用程序代码。本 LED 例程的 main 文件内容见代码清单 4-22。

代码清单 4-22 LED 例程 main 文件内容

```

1. /***** (C) COPYRIGHT 2012 WildFire Team *****/
2. * 文件名   : main.c
3. * 描述     : LED 流水灯，频率可调……
4. * 实验平台 : 野火 STM32 开发板
5. * 库版本   : ST3.5.0
6. *

```

```

7.  * 作者      : wildfire team
8.  * 论坛      : www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
9.  * 淘宝      : http://firestm32.taobao.com
10. *****/
11. #include "stm32f10x.h"
12. #include "led.h"
13.
14. void Delay(__IO u32 nCount);
15.
16. /*
17. * 函数名: main
18. * 描述  : 主函数
19. * 输入  : 无
20. * 输出  : 无
21. */
22. int main(void)
23. {
24.     /* LED 端口初始化 */
25.     LED_GPIO_Config();
26.
27.     while (1)
28.     {
29.         LED1( ON );           // 亮
30.         Delay(0xFFFFEF);
31.         LED1( OFF );          // 灭
32.
33.         LED2( ON );
34.         Delay(0xFFFFEF);
35.         LED2( OFF );
36.
37.         LED3( ON );
38.         Delay(0xFFFFEF);
39.         LED3( OFF );
40.     }
41. }
42.
43. void Delay(__IO u32 nCount) // 简单的延时函数
44. {
45.     for(; nCount != 0; nCount--);
46. }
47.
48.
49. /***** (C) COPYRIGHT 2012 WildFire Team *****/END OF FILE*****/

```

main 文件的开头部分首先包含所需的头文件：stm32f10x.h 和 led.h。

在第 14 行还声明了一个简单的延时函数，其定义在 main 文件的末尾。它是利用 for 循环实现的，用作短暂的、对精度要求不高的延时，延时的时间与输入的参数并无准确的计算公式，请不要深究。需要精准延时的时候，我们会采用定时器来精确控制。

在芯片上电（复位）后，经过启动文件中的 SystemInit() 函数配置好了时钟，就进入 main 函数了。接下来，从 main 函数开始分析代码的执行。

首先，调用了在 led.c 文件编写好的 LED_GPIO_Config() 函数，完成了对 GPIOC 的 Pin3、Pin4 和 Pin5 的初始化。紧接着就在 while 死循环里不断执行在 led.h 文件中编写的带参宏代码，并加上延时函数，使各盏 LED 轮流亮灭。当然，在 LED 控制的部分，如果不习惯带参宏的方式，读者也可以直接使用 GPIO_SetBits() 和 GPIO_ResetBits() 函数实现对 LED 的控制。

如果使用的是 3.0 版本的库，由于启动文件中没有调用 SystemInit() 函数，所以要在初始化 GPIO 等外设之前，也就是在 main 函数的第 1 行代码，就调用 SystemInit() 函数，以完成对系统时钟的配置。

到此，我们整个控制 LED 灯工程的讲解就完成了。

4.6 GPIO_Init() 函数的实现

在我们控制 LED 灯的工程中，调用了很多库函数，有 SystemInit()、GPIO_Init()、GPIO_SetBits()、GPIO_ResetBits() 等。虽说为了提高开发速度，我们只管函数的功能和如何调用就行了，但免不了有种不踏实的感觉。

所以在本节以 GPIO_Init() 函数实现的分析为例，帮助读者理解 ST 库的本质，让读者在使用库开发的时候心里更有底。

4.6.1 规范的位操作方法

由于库函数的实现涉及不少位操作，首先为读者介绍一下几个常用的位操作方法，排除阅读代码的障碍。

□ 将 char 型变量 a 的第 7 位 (bit6) 清零，其他位不变。见代码清单 4-23。

代码清单 4-23 对某位清 0 范例

```
1. a &= ~(1<<6); // 括号内 1 左移 6 位，得二进制数：0100 0000
2.                // 按位取反，得 1011 1111，所得的数与 a 做 "位与 (&)" 运算
3.                // a 的第 7 位 (bit6) 被置零，而其他位不变
```

□ 同理，将变量 a 的第 7 位 (bit6) 置 1，其他位不变的方法见代码清单 4-24。

代码清单 4-24 对某位置 1 范例

```
1. a |= (1<<6); // 把第 7 位 (bit6) 置 1，其他位不变
```

□ 将变量 a 的第 7 位 (bit6) 取反，其他位不变的方法见代码清单 4-25。

代码清单 4-25 对某位取反范例

```
1. a ^= (1<<6); // 把第 7 位 (bit6) 取反，其他位不变
```

4.6.2 GPIO_Init() 实现代码分析

有了上面的位操作知识准备后，就可以分析 GPIO_Init() 函数的定义代码了。见代码清单 4-26。

代码清单 4-26 GPIO_Init() 函数定义源代码

```

1. void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
2. {
3.     uint32_t currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x00;
4.     uint32_t tmpreg = 0x00, pinmask = 0x00;
5.     /* 断言，用于检查输入的参数是否正确 */
6.     assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
7.     assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
8.     assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
9.
10. /*----- GPIO 的模式配置 -----*/
11. /* 把输入参数 GPIO_Mode 的低四位暂存在 currentmode */
12.
13.     currentmode = ((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x0F);
14.
15. /* 判断是否为输出模式，若是输出模式，可输入参数中输出模式的 bit4 位都是 1 */
16.
17.     if (((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x10)) != 0x00)
18.
19.     {
20.         /* 检查输入参数 */
21.         assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
22.         /* 输出模式，所以要配置 GPIO 的速率：00(输入模式) 01(10MHz) 10(2MHz) 11 */
23.         currentmode |= (uint32_t)GPIO_InitStruct->GPIO_Speed;
24.     }
25. /*----- 配置 GPIO 的 CRL 寄存器 -----*/
26.
27.     /* 判断要配置的是否为 pin0 ~ pin7 */
28.     if (((uint32_t)GPIO_InitStruct->GPIO_Pin & ((uint32_t)0x00FF)) != 0x00)
29.     {
30.         /* 备份原 CRL 寄存器的值 */
31.         tmpreg = GPIOx->CRL;
32.         /* 循环，一个循环设置一个寄存器位 */
33.         for (pinpos = 0x00; pinpos < 0x08; pinpos++)
34.         {
35.             /* pos 的值为 1 左移 pinpos 位 */
36.             pos = ((uint32_t)0x01) << pinpos;
37.             /* 令 pos 与输入参数 GPIO_PIN 做位与运算，为下面的判断做准备 */
38.             currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
39.             /* 判断，若 currentpin=pos，说明 GPIO_PIN 参数中含的第 pos 个引脚需要配置 */
40.             if (currentpin == pos)
41.             {
42.                 /* pos 的值左移两位 (乘以 4)，因为寄存器中 4 个寄存器位配置一个引脚 */
43.                 pos = pinpos << 2;
44.                 /* 以下两个句子，把控制这个引脚的 4 个寄存器位清零，其他寄存器位不变 */
45.                 pinmask = ((uint32_t)0x0F) << pos;
46.                 tmpreg &= ~pinmask;
47.                 /* 向寄存器写入将要配置的引脚模式 */

```

```

48.         tmpreg |= (currentmode << pos);
49.         /* 复位 GPIO 引脚的输入输出默认值 */
50. /* 判断是否为下拉输入模式 */
51.         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
52.         {
53. /* 下拉输入模式, 引脚默认置 0, 对 BRR 寄存器写 1 可对引脚置 0 */
54.             GPIOx->BRR = (((uint32_t)0x01) << pinpos);
55.         }
56.         else
57.         {
58.             /* 判断是否为上拉输入模式 */
59.             if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
60.             {
61. /* 上拉输入模式, 引脚默认值为 1, 对 BSRR 寄存器写 1 可对引脚置 1 */
62.                 GPIOx->BSRR = (((uint32_t)0x01) << pinpos);
63.             }
64.         }
65.     }
66. }
67. /* 把前面处理后的暂存值写入到 CRL 寄存器之中 */
68.     GPIOx->CRL = tmpreg;
69. }
70. /*----- 以下部分是对 CRH 寄存器配置的 -----
71. ----- 当要配置的引脚为 pin8 ~ pin15 的时候, 配置 CRH 寄存器, -----
72. ----- 这个过程和配置 CRL 寄存器类似 -----
73. ----- 读者可自行分析, 看看自己是否了解了上述过程 --^_^----- */
74. /* Configure the eight high port pins */
75. if (GPIO_InitStruct->GPIO_Pin > 0x00FF)
76. {
77.     tmpreg = GPIOx->CRH;
78.     for (pinpos = 0x00; pinpos < 0x08; pinpos++)
79.     {
80.         pos = (((uint32_t)0x01) << (pinpos + 0x08));
81.         /* Get the port pins position */
82.         currentpin = ((GPIO_InitStruct->GPIO_Pin) & pos);
83.         if (currentpin == pos)
84.         {
85.             pos = pinpos << 2;
86.             /* Clear the corresponding high control register bits */
87.             pinmask = ((uint32_t)0x0F) << pos;
88.             tmpreg &= ~pinmask;
89.             /* Write the mode configuration in the corresponding bits */
90.             tmpreg |= (currentmode << pos);
91.             /* Reset the corresponding ODR bit */
92.             if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
93.             {
94.                 GPIOx->BRR = (((uint32_t)0x01) << (pinpos + 0x08));
95.             }
96.             /* Set the corresponding ODR bit */

```

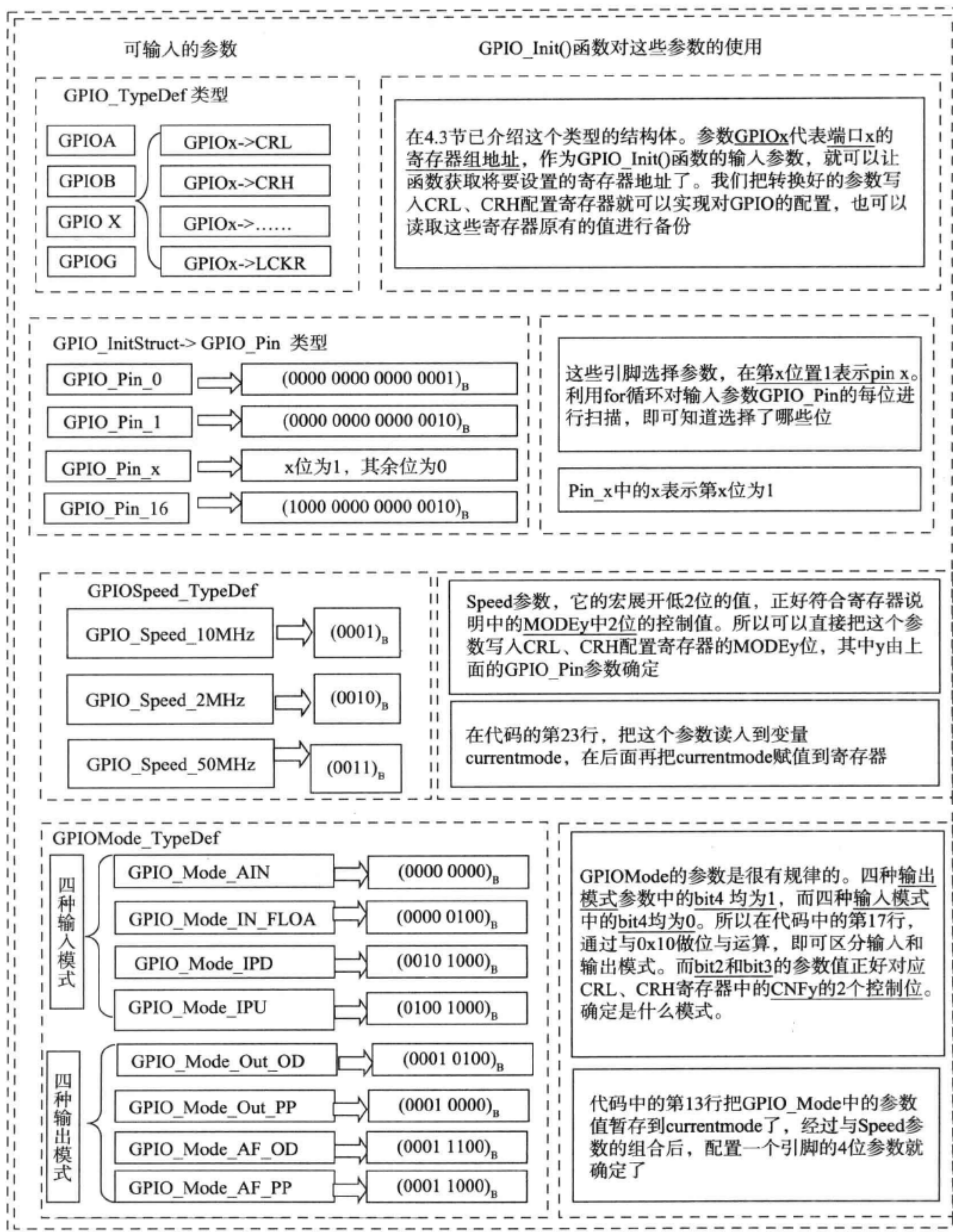



图 4-13 GPIO_Init() 函数说明

以 led.c 文件中对 GPIO_Init() 函数的调用为例。在调用函数前有这样的流程：

1) led.c 代码的第 32 行, 对 GPIO_InitStructure.GPIO_Pin 结构体成员赋值为 GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5, 宏展开为 (0000 0000 0011 1000)_B, 表明我们将要对这三个引脚进行配置。

2) 第 35 行, 对 .GPIO_Mode 赋值为 GPIO_Mode_Out_PP, 宏展开为 (0001 0000)_B, 表明我们要把这三个引脚都设置为通用推挽模式。

3) 第 38 行, 对 .GPIO_Speed 赋值为 GPIO_Speed_50MHz, 宏展开为 (0011)_B, 表明我们设置这三个引脚的输出最大速度都为 50 MHz。

led.c 的第 41 行调用 GPIO_Init() 的时候, 就把 GPIOC 和上面这三个参数输入到函数了, 经过这个函数处理, 最终它向 GPIOC 组的 CRL 配置寄存器写入了一个值:

```
1. GPIOC->CRL = 0x44333444;
2. // 二进制表示为 (0100 0100 0011 0011 0011 0100 0100 0100)
```

把这个值化为二进制为: (0100 0100 0011 0011 0011 0100 0100 0100)_B。这个值的每 4 个二进制位代表一组引脚的控制值。Pin3、Pin4、Pin5 的控制值都是 (0011)_B, 有心的读者可以对比一下 CRL 寄存器的说明, 这些控制值正好可以把 GPIO 设置为符合我们输入参数要求的状态, 即最大速率为 50 MHz 的通用推挽输出模式。

4.6.3 再论开发方式

了解库函数的实现后, 我们现在就可以用实例来分析使用库函数与直接配置寄存器的区别了。用直接配置寄存器的方法, 只需要一个语句:

```
1. GPIOC->CRL = 0x44333444;
```

这样直接向寄存器赋值就完成了, 以这样的方式配置是内核执行效率最高的方式, 内核的工作是简单了, 但我们为实现所需的配置来确定这样的一个值却是一件麻烦事, 工程量大的时候缺点就显而易见了。

配置寄存器还可以用一些相对缓和的方法, 如前面提到的三种位操作方式。见代码清单 4-27。

代码清单 4-27 位操作方式修改寄存器

```
1. GPIOC->CRL &= ~ (uint32_t) (1111<<4*3); // 清空 Pin3 的 4 个控制位
2. GPIOC->CRL |= (uint32_t) (0011<<4*3); // 配置 Pin3 的 4 个控制位
3. GPIOC->CRL &= ~ (uint32_t) (1111<<4*4); // 清空 Pin4 的 4 个控制位
4. GPIOC->CRL |= (uint32_t) (0011<<4*4); // 配置 Pin4 的 4 个控制位
5. GPIOC->CRL &= ~ (uint32_t) (1111<<4*5); // 清空 Pin5 的 4 个控制位
6. GPIOC->CRL |= (uint32_t) (0011<<4*5); // 配置 Pin5 的 4 个控制位
```

这个方法也可以实现我们所需的配置, 而且修改起来比较容易。

最后就是调用库函数的方法, 从内核的执行效率上看, 首先库函数在被调用的时候要耗费调用时间; 在函数内部, 把输入参数转换为可以直接写入到寄存器的值也耗费了一些运算时间。而其他的宏、枚举等解释操作是在编译过程完成的, 这部分并不消耗内核的时间。而优点则是: 我们可以快速上手 STM32 微控制器; 配置外设状态时, 不需要再纠结要向寄存器写入什么数值; 交流方便, 查错简单。这就是我们选择库开发方式的原因。

现在处理器的主频越来越高, 我们需不需要担心 CPU 耗费那么多时间来干活会不会被累倒?

我们要告诉你的是：不需要，还是担心一下自己字字查询 datasheet 会不会被累倒吧！

至此，我们就把 GPIO_Init() 库函数的实现分析完毕了。分析它纯粹是为了学习其编程的方式、思想，这对提高我们的编程水平是很有好处的，再就是顺便感受一下 ST 官方库设计的严谨性，我们认为这样的代码不仅严谨且华丽优美，相信你也有这样的感受。

我们在以后开发的工程中，一般不会去分析库函数的实现了。因为这些库函数都是把原来封装好的宏或枚举标识符转化成相应的值，写入到寄存器之中。这是十分枯燥和机械的工作，既然我们已经知道它的原理，又有现成的函数可供调用，就没必要再去探究了。

4.7 开发步骤总结

第 3、4 章引导读者建立第一个工程的过程，由于涉及面较广，知识点分散，在此做一个总结。

- 1) 为控制 LED 灯，知道要使用 GPIO 外设。
- 2) 了解 GPIO 外设有什么功能，要如何使用。
- 3) 获知 GPIO 的地址映射，知道它所挂载的总线 APB2。
- 4) 了解 ST 官方库对寄存器的封装。
- 5) 了解时钟树，查看 GPIOC 的时钟来源，即 PCLK2。
- 6) 在 stm32f10x_conf.h 文件中包含用到的头文件 stm32f10x_gpio.h、stm32f10x_rcc.h。
- 7) 在工程模板的基础上添加 led.c、led.h 用户文件。
- 8) 编写驱动初始化函数 LED_GPIO_Config()。

9) 开启外设 GPIOC 时钟，分析由 SystemInit() 函数配置的默认的 Sysclk=72MHz 的时钟频率是否符合工程要求。

10) 根据控制要求，定义并填充初始化结构体 GPIO_InitStructure，向相应的结构体成员写入适当的参数。

- 11) 调用初始化函数 GPIO_Init() 初始化 GPIOC。
- 12) 编写相应的 led.h 头文件。
- 13) 针对不同的应用要求，编写 main 应用程序。
- 14) 调试程序、完成。

不总结不知道，一总结吓一跳。我们只是写了一个简单的控制 LED 灯工程就如此复杂？非也，控制 LED 灯并不是我们学习的目的。以上这些步骤引导读者从整体上了解整个 STM32 芯片，真正有价值的并不是这个 LED 灯工程，而是使读者熟悉 STM32 的存储器架构、地址映射、时钟树、库文件、利用库的方法及开发工程的步骤，建立 STM32 的开发思想。从以后的工程开始，我们就可以真正地利用库来快速开发工程了。

对 LED 工程的步骤总结，希望能有举一反三的效果。在利用库开发其他工程的时候，其开发步骤类似，只是针对不同的外设，由于其功能的不同而稍微有点改变。这些改变，就体现在不同的库函数之中。对库函数的使用，就是填充结构体而已。如同编写 Linux 的驱动程序一样，所有驱动实质上都是向各种结构体成员写入适当的控制参数，填充完结构体，驱动开发就基本完成了。




第 5 章 调 试 程 序

编写代码少不了调试，调试也是能够快速提高 C 语言编程能力的一种途径。本章为读者详细介绍软件仿真及硬件调试的方法，并介绍一些 MDK 软件的使用技巧。

5.1 MDK 软件仿真调试

软件调试就是利用 MDK 软件对 STM32 进行仿真，执行我们工程中的代码。以下为进行软件调试的步骤：

- 1) 首先进行仿真环境配置，点击 ，弹出选择菜单，配置为使用软件仿真，见图 5-1。
- 2) 设置仿真运行的外部晶振频率为 8 MHz，见图 5-2。

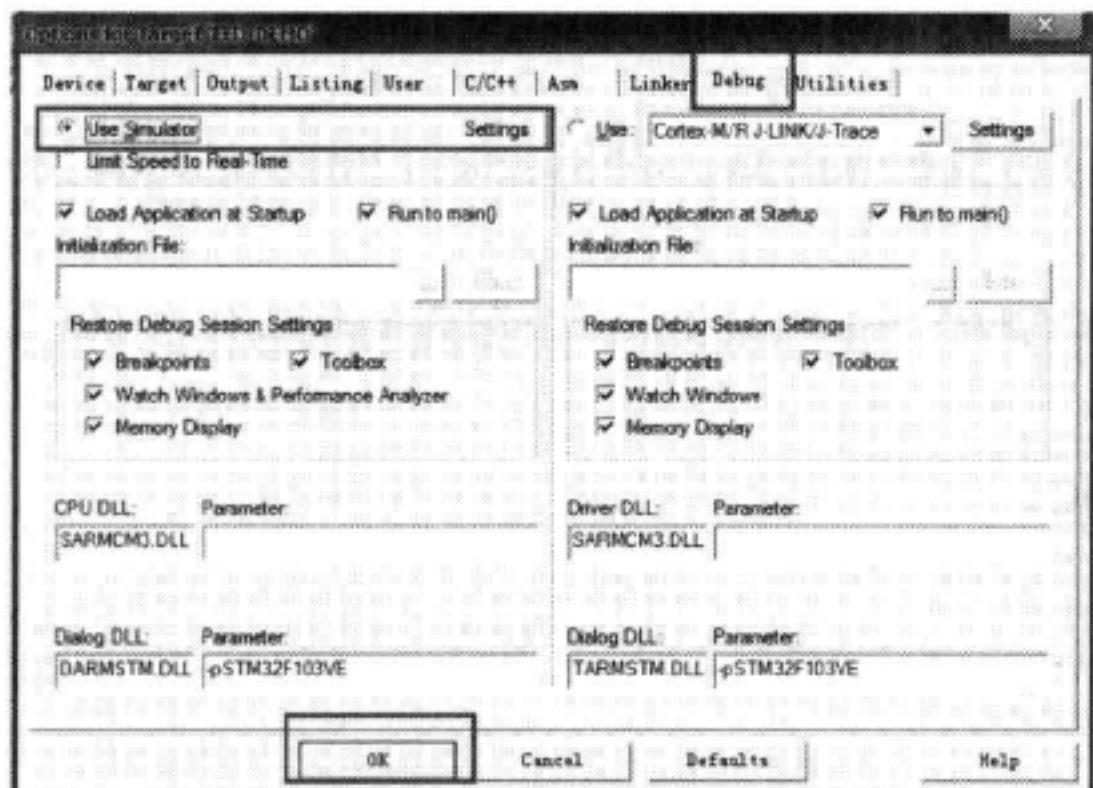


图 5-1 选择软件仿真

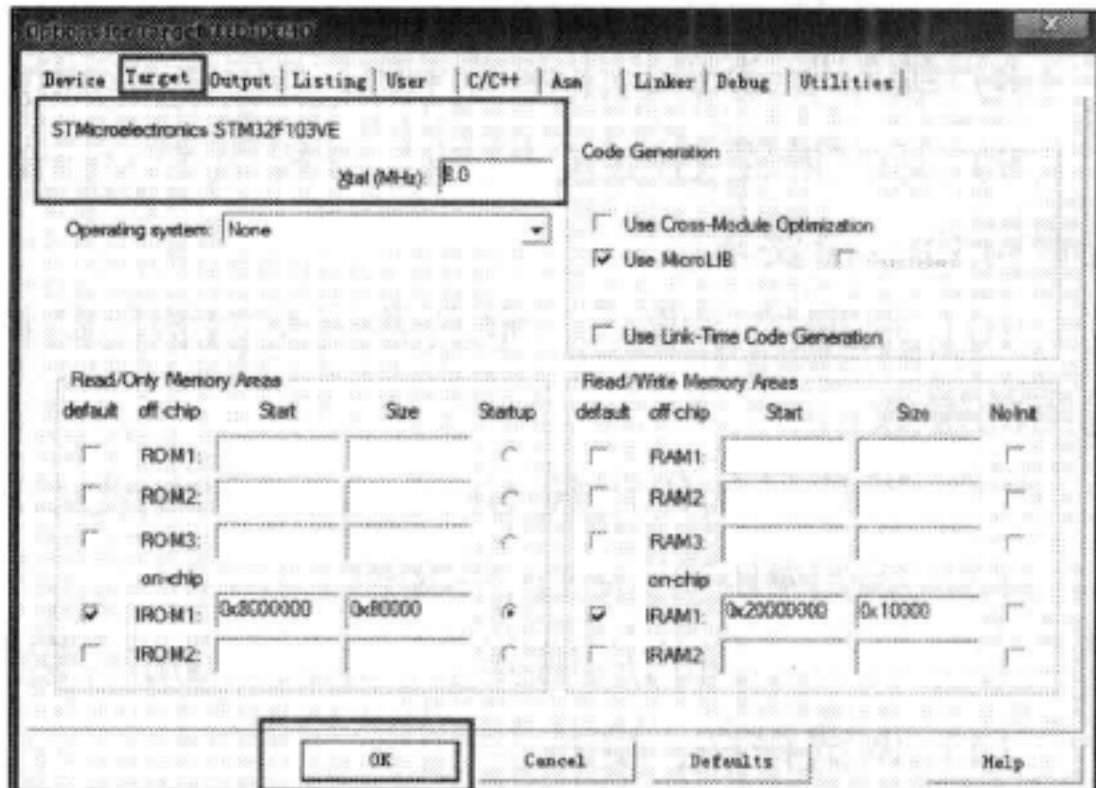










图 5-2 设置仿真的晶振频率

- 3) 点击 Debug  按钮，开始调试，弹出了调试界面，见图 5-3。
- 4) 接下来可以点击        这一栏工具，分别进行复位、全速运行、结束运行、单步运行、运行到下一行、运行至跳出函数、运行至断点，双击代码行可设置断点，前提是代码行左端要有灰色块。在全速运行的时候会运行至断点处暂停。
- 5) 点击外设状态标签可弹出相应的外设状态查看窗口，见图 5-4。
- 6) 点击 watch1 工具，弹出 watch1 窗口，输入变量名可查看代码中的变量值，见图 5-5。

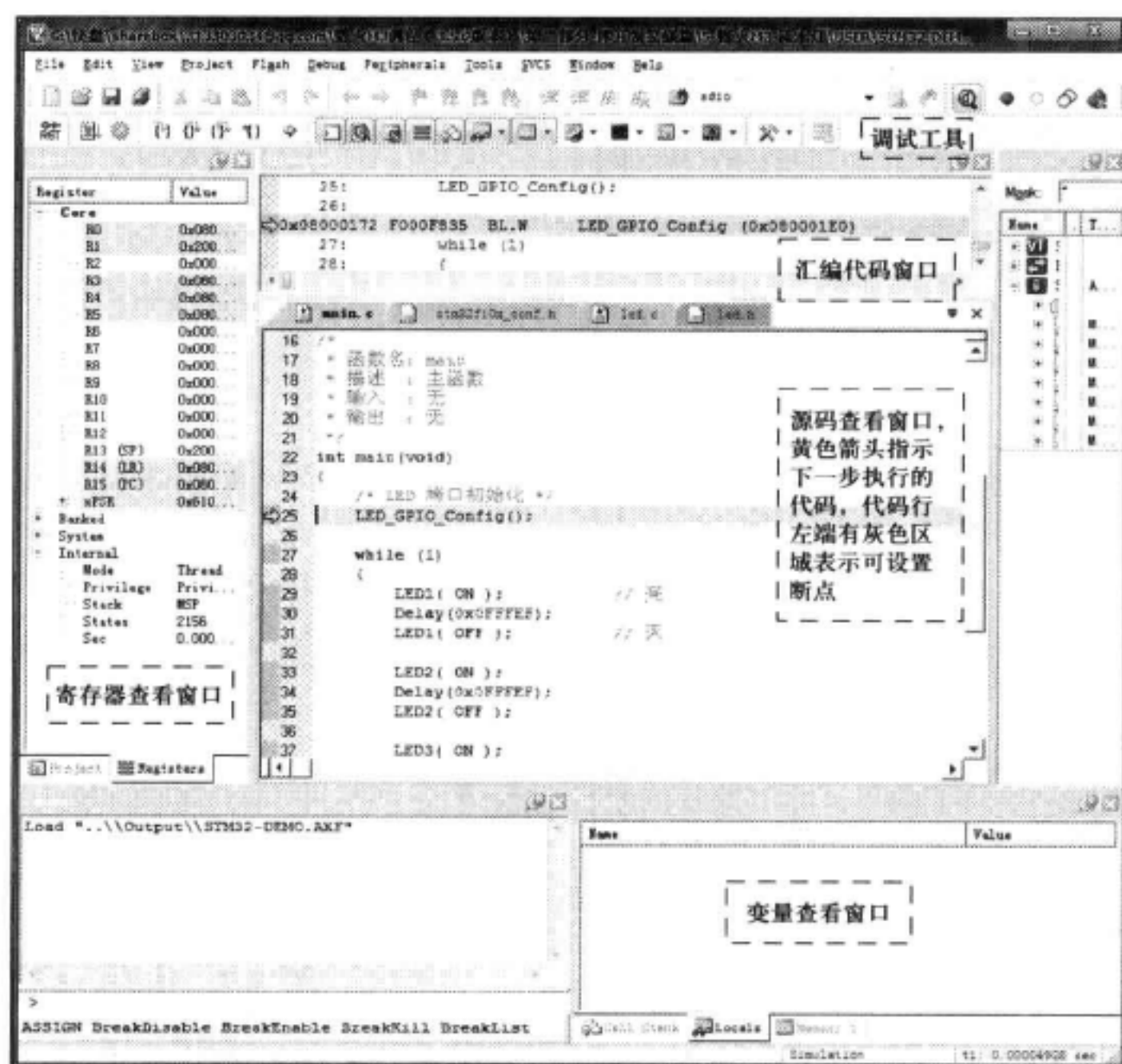


图 5-3 调试界面

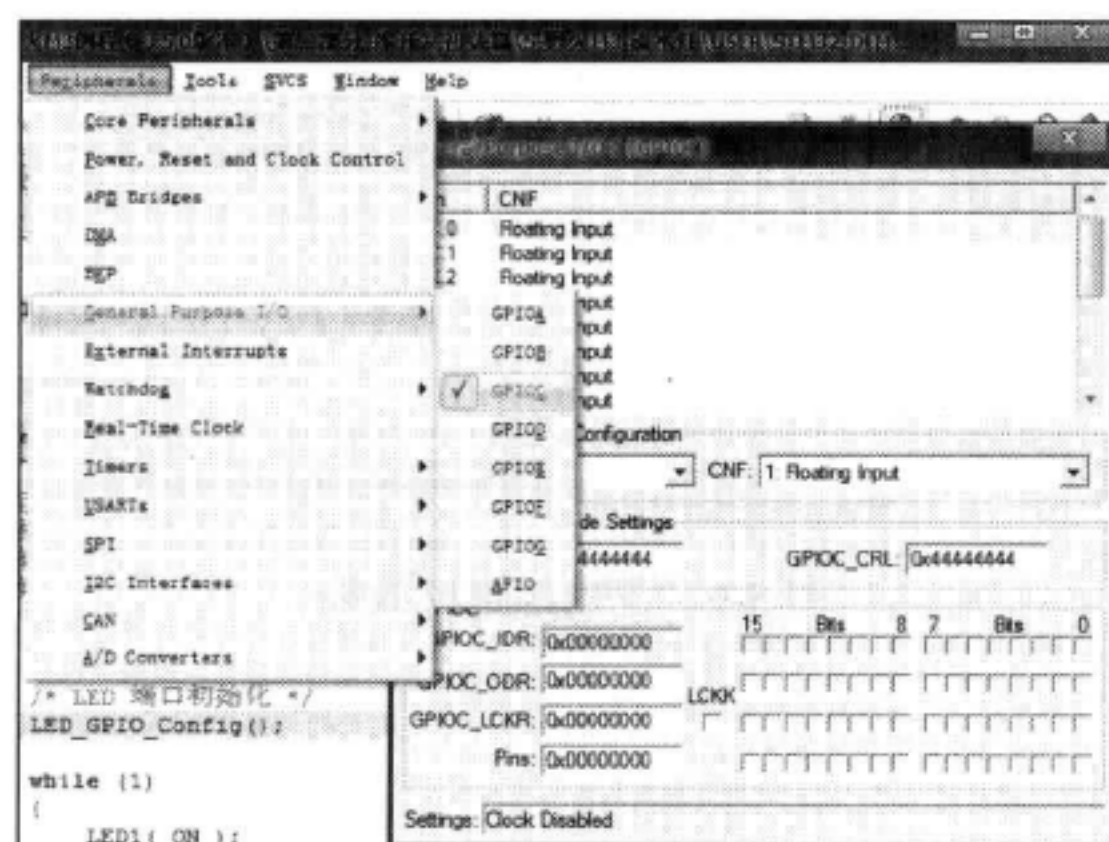


图 5-4 查看 IO 状态

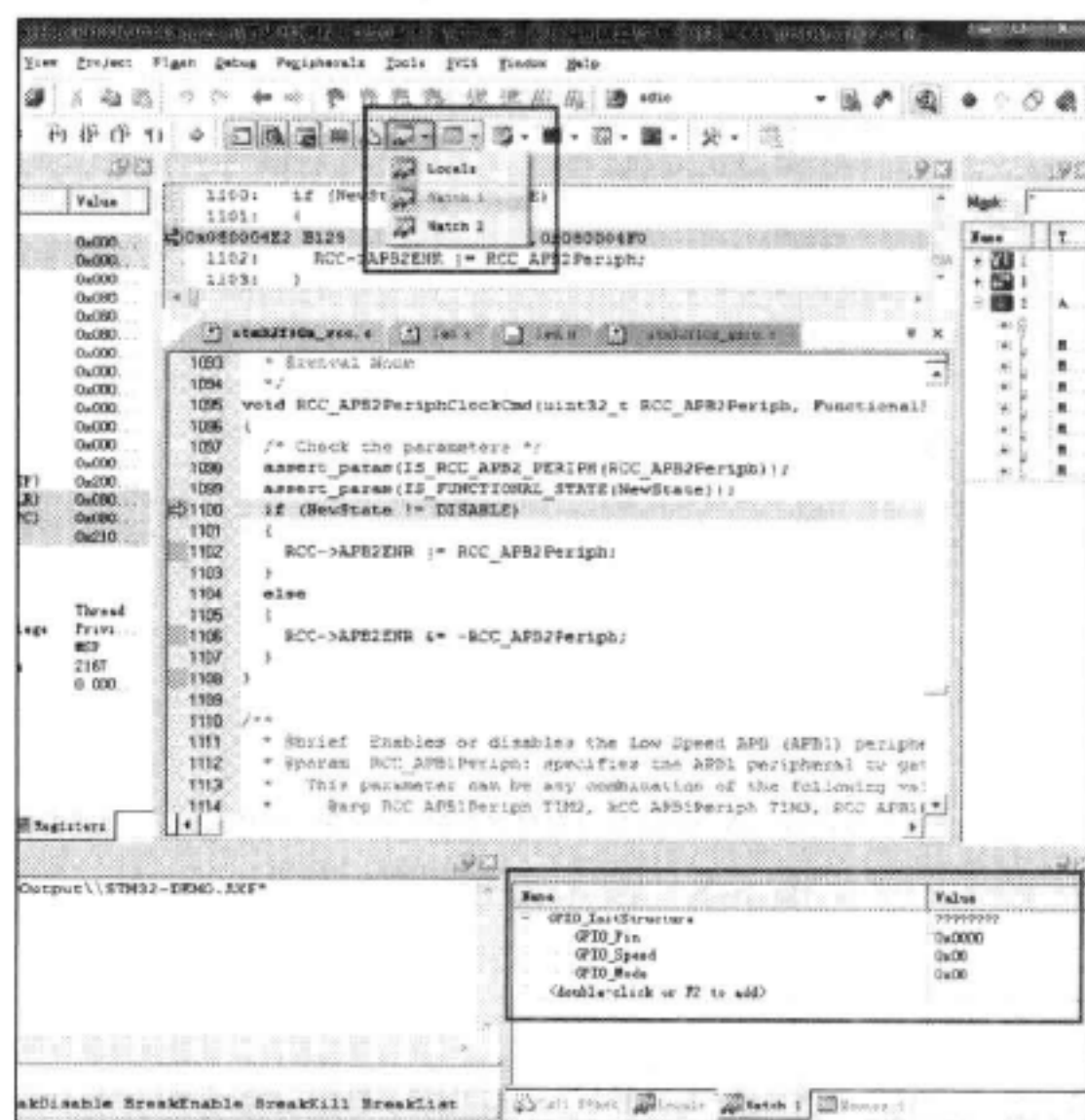




图 5-5 查看变量值

7) 点击 USART #1 窗口, 可弹出串口调试终端, 可模拟串口软件终端 (截图为串口调试例程), 见图 5-6。

8) 点击  选中  Logic Analyzer 逻辑分析仪。点击 Setup 输入要分析的引脚，见图 5-7。

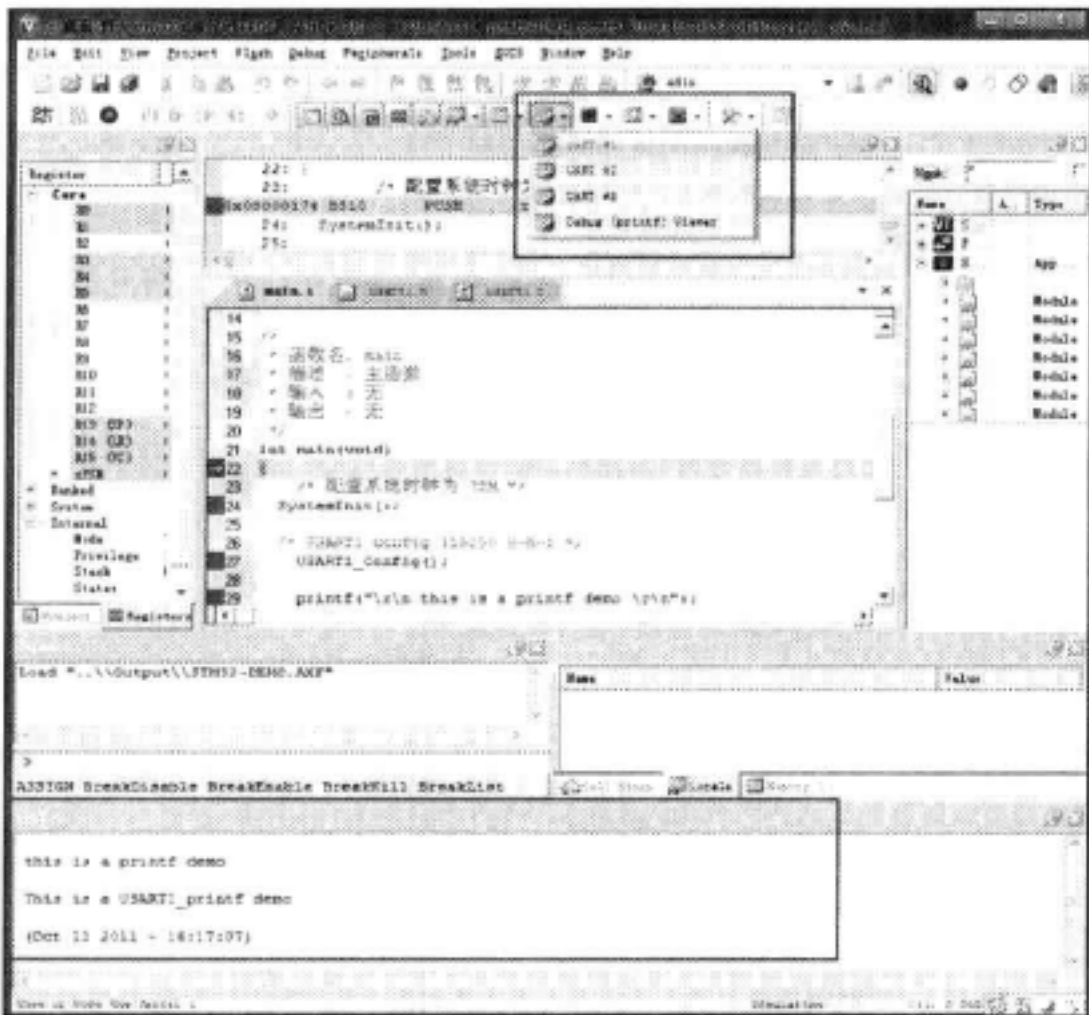


图 5-6 仿真串口

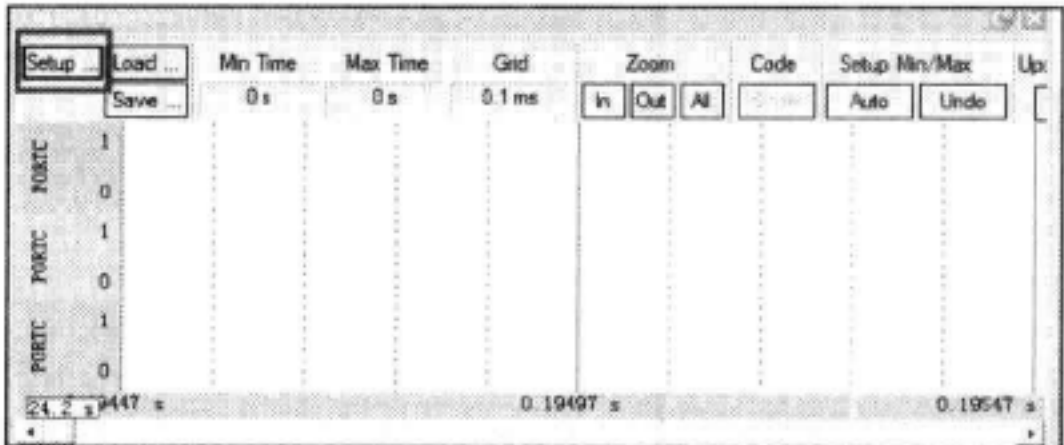


图 5-7 仿真逻辑分析仪

9) 在弹出的窗口中输入 PORTC.3、PORTC.4、PORTC.5，查看其运行效果，见图 5-8。

10) 全速运行。调节 Zoom 的 In/Out 可进行放大缩小时间轴，见图 5-9。

软件仿真还有很多强大的功能，就不在此一一介绍了，这些介绍都可以在 MDK 的帮助手册中找到。

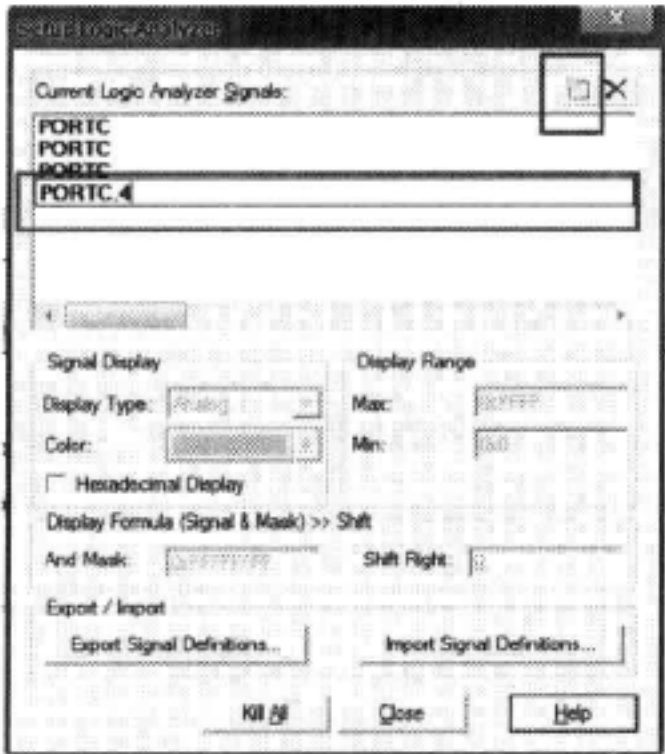


图 5-8 输入要查看的引脚

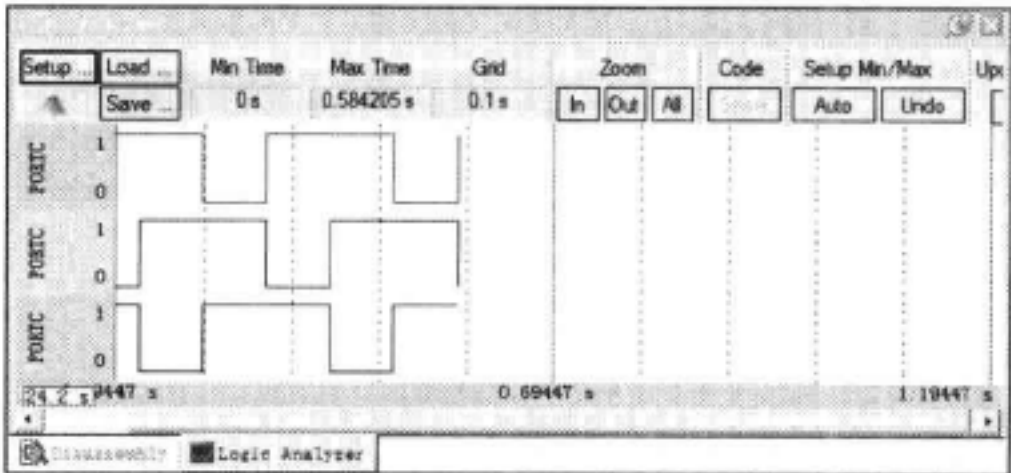


图 5-9 逻辑仿真仪效果


5.2 使用 J-LINK 进行硬件调试

5.2.1 硬件调试

MDK 仿真工具很强大，但更多时候我们是使用硬件调试的，硬件调试是让代码在真实的项目环境中运行，如项目要求用 ADC 采样、控制电机、驱动液晶屏等场合，软件仿真的缺陷就非常

明显了，一般建议采用硬件调试程序。

硬件调试和软件仿真的使用大部分都是相似的，如查看变量值等。但串口调试终端的仿真工具就无法使用了。这要求我们给硬件提供真实的工作环境，可以直接把串口接到 PC 上。

硬件调试的配置在前面已经做了讲解。配置好了硬件调试的环境，编译好代码，插上 J-LINK，就可以直接点击  按钮，然后就可以像使用软件仿真那样进行硬件调试了，查看变量、外设状态、寄存器值等都可以。

5.2.2 软件编译过程

读者问：为什么不用点击下载代码就可以调试了？这需要了解工程的编译过程。

当我们按下 MDK 的编译按钮之后，它会完整地执行完如图 5-10 所示的所有过程。首先把我们工程中的 C 语言源文件经过 C 编译器生成相应的后缀为 .o 的目标文件，把汇编源文件（startup_stm32f10x_hd.s 启动文件）也编译生成相应的 .o 目标文件，最后通过连接器把各目标文件及存储器布局设置（在 option for target 菜单设置）连接起来，生成后缀为 .axf 的可执行映像文件，这个映像文件可转化为二进制的程序映像文件 .bin，也可以转化为十六进制文件 .hex。

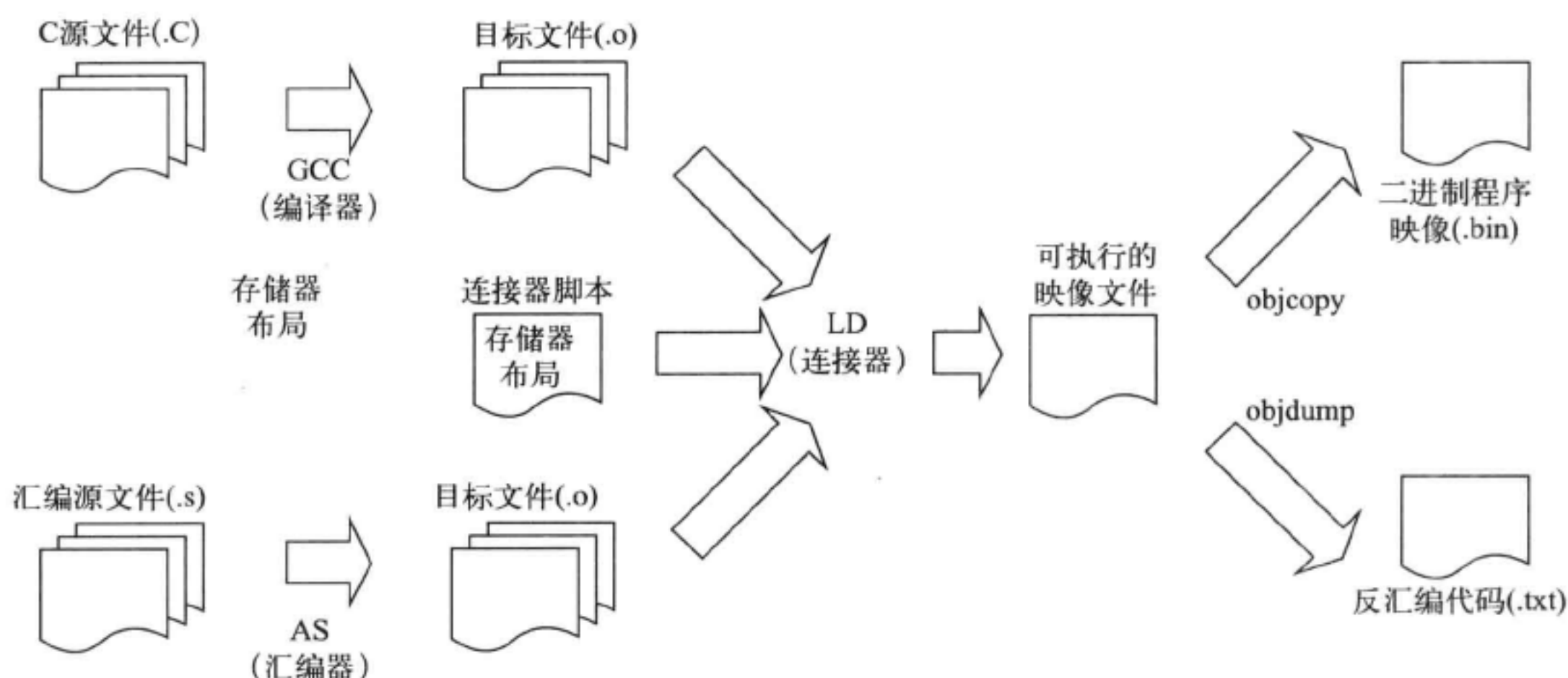


图 5-10 编译过程

平时我们下载到芯片 Flash 中的代码就是 .hex 文件，上电后，内核把在 Flash 中的代码加载到 SRAM，就可以开始执行代码了（Flash 相当于 PC 的硬盘，SRAM 相当于内存，在内存的代码可以直接运行，但在硬盘的代码却必须加载到内存上运行）。

而 .axf 文件，即所谓的可执行映像文件，如果我们把它直接加载到芯片的 SRAM，那么芯片就可以直接运行我们保存在 .axf 上的代码了。当我们按下硬件调试的 Debug 按钮时，Keil MDK 就通过 J-LINK 把编译时生成的可执行映像文件加载到芯片的 SRAM 上运行，我们就可以调试代码了。

5.3 MDK 使用小技巧

Keil MDK 是一个很强大的工具，使用它编写或阅读代码效率很低的话，那一定是不太熟悉它。在此向大家介绍一些使用 MDK 的小技巧，以提高开发效率。

1) 调试必备，大量代码注释与取消注释，选中代码按右键，进入图 5-11 所示界面选择“Comment Selection”可注释，“Uncomment Selection”可以取消注释。

2) 有读者问，宏定义怎么这么多？如何快速查找宏或函数的定义？对着要查找的宏或函数按右键，选择“GO To Definition of ‘××’”就可以快速跳转。见图 5-12。



图 5-11 批量注释



图 5-12 查看变量或宏定义

这个方法不仅可以查找宏，还可以查找函数、变量的原定义。若查看完它们的定义想返回到原来的代码中，可以点击工具栏中的“←”返回。见图 5-13。

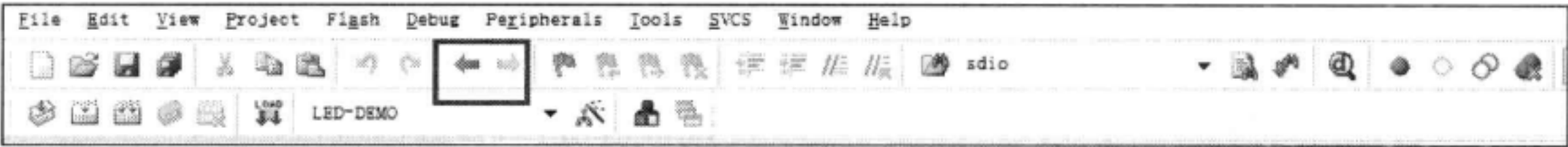


图 5-13 返回上一次代码跳转

3) 函数太多了，如何在函数的海洋中把它找出来？根据标签查找函数。点击图 5-14 中左侧的函数名即可跳转到相应的函数定义处。

4) 白色的代码调试环境太伤眼？调节代码框背景颜色。步骤见图 5-15 和图 5-16。

以上是使用 MDK 时常用的技巧，在这里介绍出来是为了抛砖引玉，其他更强大的功能就由读者去发现，让自己成为高手吧！



图 5-14 查找工程中的函数

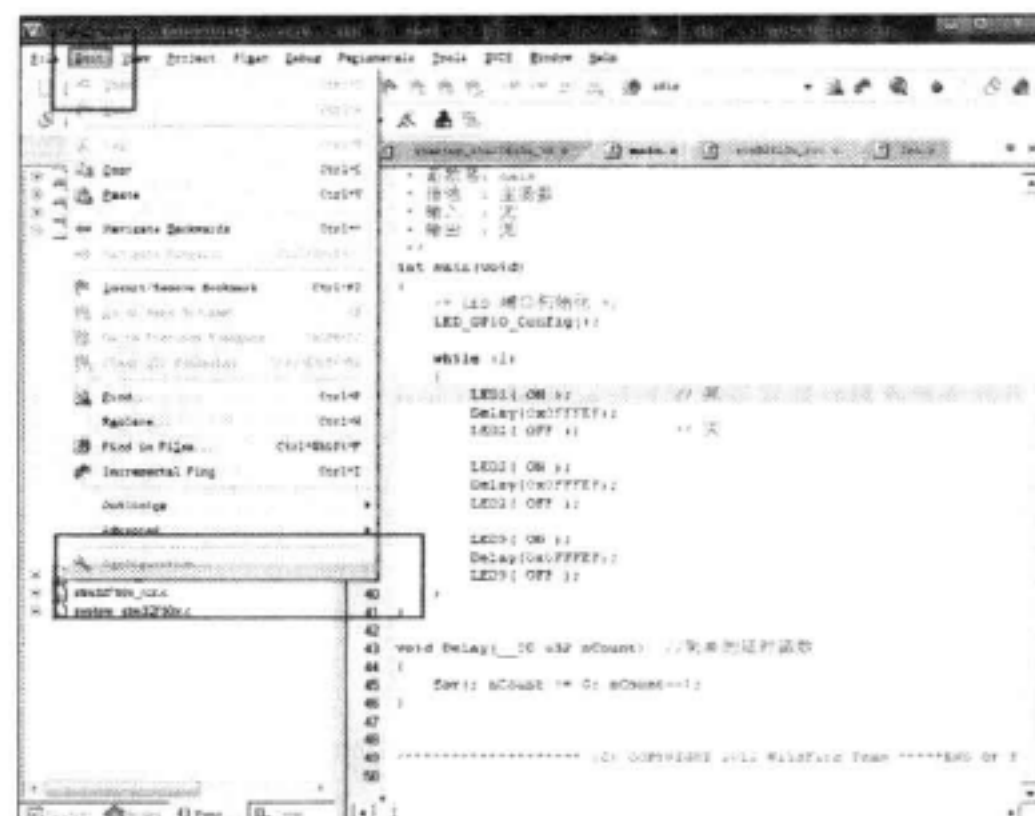


图 5-15 打开 configuration 选项

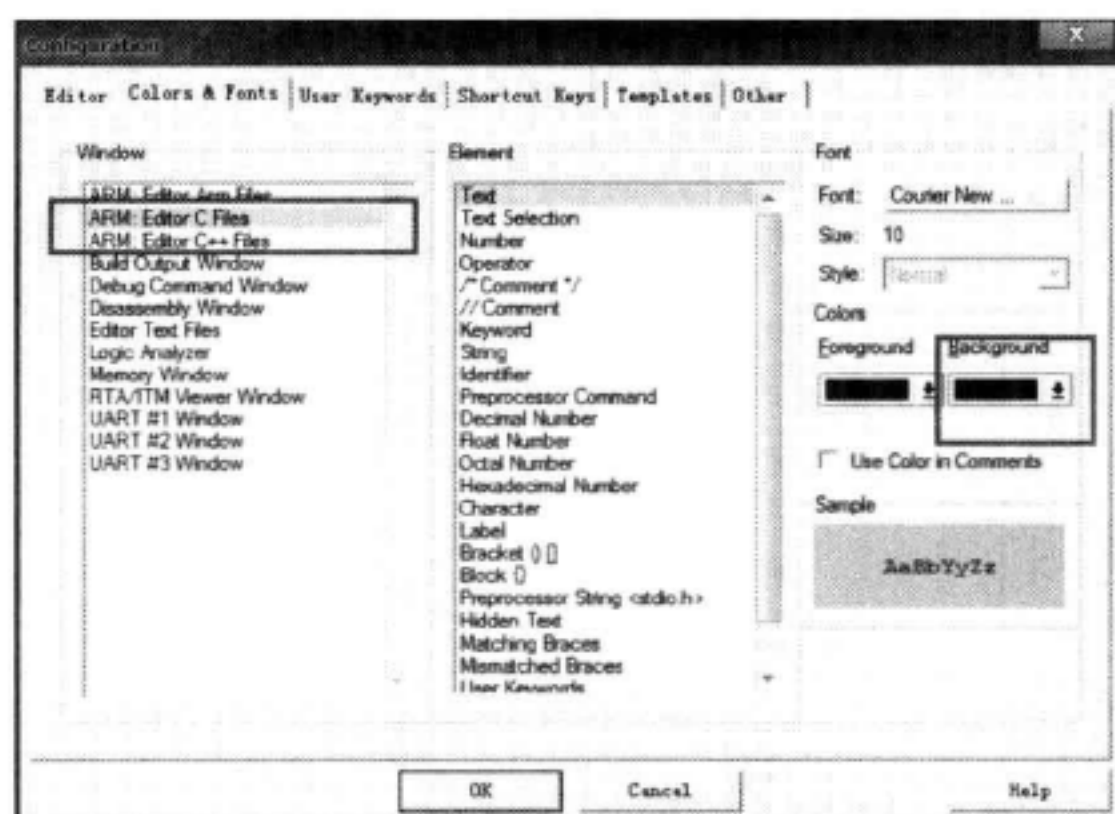
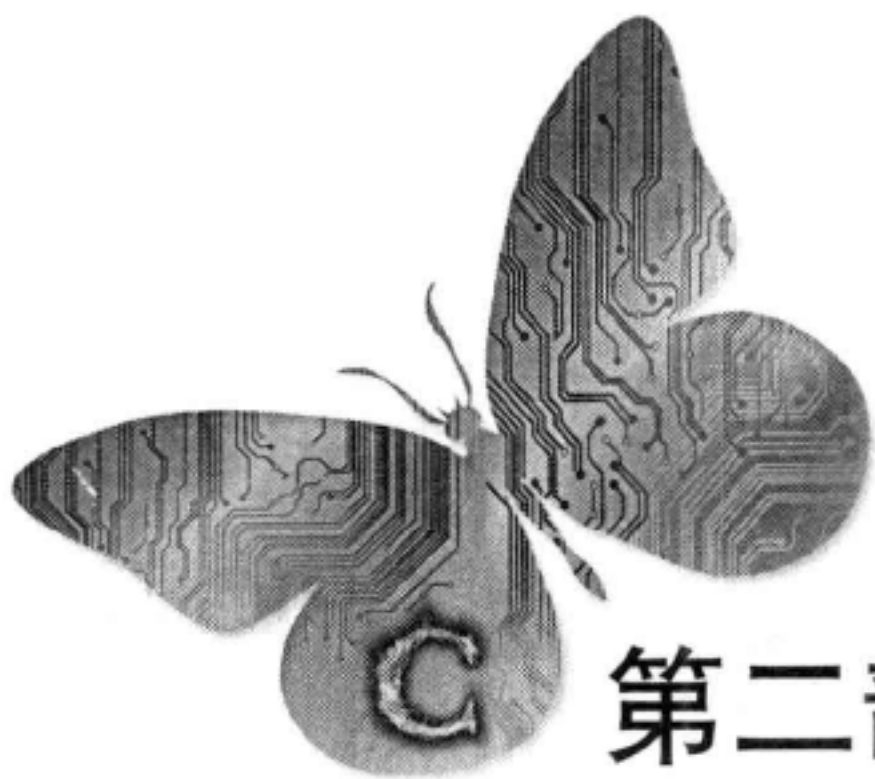


图 5-16 设置 C 文件背景颜色



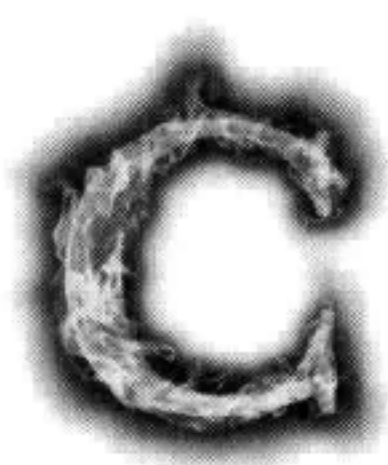
第二部分 库开发中级篇

初级篇从原理上讲解了固件库，带领读者踏进了 STM32 的世界，中级篇将针对 STM32 的各种片上外设，着重从应用的角度讲解库函数。在实际的工程代码中，会因不同的外设或需求，而使用不同的库函数。

中级篇的每个章节主要讲解一种外设，有 I²C 通信、SPI 通信、CAN 通信、ADC 应用等，各章节之间的内容基本独立，读者可以根据当前使用的实际需要进行选择性阅读，学习时，要注意结合相应的通信协议来理解。

相信通过中级篇大量的应用例程学习，读者对 STM32 和固件库的理解会更上一层楼。

- ❑ 第6章 GPIO 再举例之按键实验
- ❑ 第7章 EXTI 之按键中断实验
- ❑ 第8章 串口通信 (USART)
- ❑ 第9章 库函数开发小结
- ❑ 第10章 DMA——为CPU减负
- ❑ 第11章 ADC 实验 (DMA 方式)
- ❑ 第12章 SysTick (系统滴答定时器)
- ❑ 第13章 STM32定时器
- ❑ 第14章 I²C接口
- ❑ 第15章 SPI模块
- ❑ 第16章 CAN控制器



第 6 章

GPIO 再举例之按键实验

在 LED 灯例程中我们已经简单体验了 GPIO 的强大之处。本书配套的硬件平台使用的芯片型号是 STM32F103VET6，具有 100 个引脚，除去晶振输入、电源输入、Boot 引脚，剩下的 80 个引脚均为 GPIO。它们分布在 GPIOA ~ GPIOE 的 5 个端口组之中，每个小组有 16 个引脚，所有的 GPIO 引脚都可以用作外部中断源的输入，每个 GPIO 引脚可配置为 8 种模式，不同的引脚还有相应的复用功能、复用功能重映射等，足以满足应用需求，也足以把初学者弄得晕头转向。

本章以按键工程为例，着重分析 GPIO 的模式配置。

6.1 GPIO 的 8 种工作模式

在初始化 GPIO 的时候，根据我们的使用要求，必须把 GPIO 设置为相应的模式。如 LED 例程中的 GPIO 引脚如果配置为模拟输入模式是必然会导致错误的。

我们配合 GPIO 结构图，来看看 GPIO 的 8 种模式及其应用场合，见图 6-1。

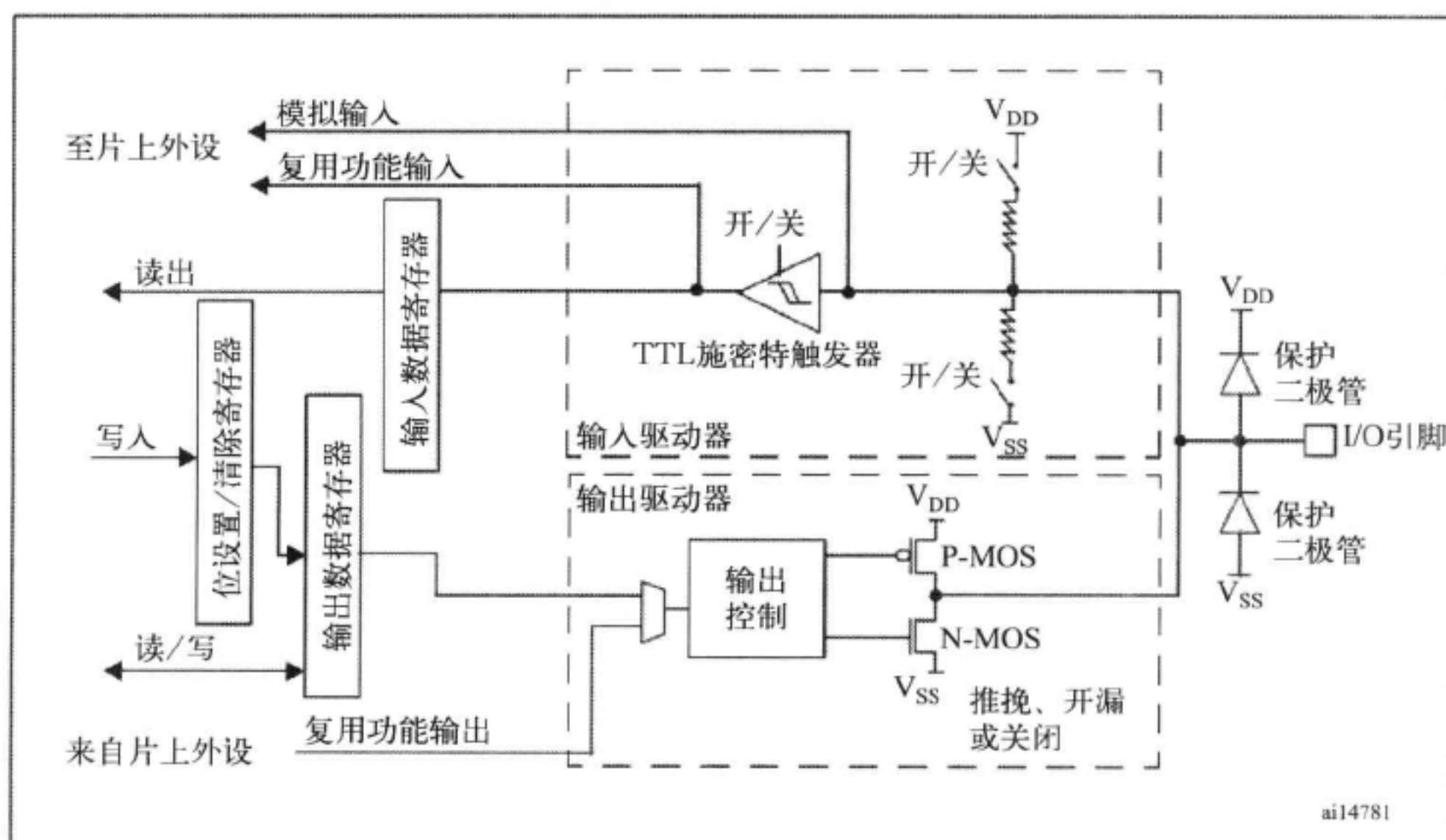


图 6-1 GPIO 结构图

图 6-1 的最右端为 I/O 引脚，左端的器件位于芯片内部。I/O 引脚并联了两个用于保护的二极管。

6.1.1 4 种输入模式

结构图的上半部分为输入模式结构，分为上拉输入模式、下拉输入模式、浮空输入模式和模拟输入模式。

接下来就遇到了两个开关和电阻，与 V_{DD} 相连的为上拉电阻，与 V_{SS} 相连的为下拉电阻。再连接到 TTL 施密特触发器就把电压信号转化为 0、1 的数字信号存储在输入数据寄存器 (IDR)。我们可以通过设置配置寄存器 (CRL、CRH) 来控制这两个开关，于是就可以得到 GPIO 的上拉输入模式 (GPIO_Mode_IPU) 和下拉输入模式 (GPIO_Mode_IPD) 了。

从它的结构我们就可以理解，若 GPIO 引脚配置为上拉输入模式，在默认状态下 (GPIO 引脚无输入)，读取得的 GPIO 引脚数据为 1，高电平。而下拉模式则相反，在默认状态下其引脚数据为 0，低电平。

而 STM32 的浮空输入模式 (GPIO_Mode_IN_FLOATING) 在芯片内部既没有接上拉，也没有接下拉电阻，经由触发器输入。配置成这个模式直接用电压表测量其引脚电压为 1 点几伏，这是个不确定值。由于其输入阻抗较大，一般把这种模式用于标准的通信协议如 I²C、USART 的接收端。

模拟输入模式 (GPIO_Mode_AIN) 则关闭了施密特触发器，不接上、下拉电阻，经由另一线路把电压信号传送到片上外设模块。如传送至 ADC 模块，由 ADC 采集电压信号。所以使用 ADC 外设的时候，必须设置为模拟输入模式。

6.1.2 4 种输出模式

结构图的下半部分为输出模式结构，分为推挽输出模式、开漏输出模式、复用推挽输出模式和复用开漏输出模式。

线路经过一个由 P-MOS 管和 N-MOS 管组成的单元电路。而所谓推挽输出模式，则是根据其工作方式命名的。在输出高电平时，P-MOS 管导通；低电平时，N-MOS 管导通。两个管子轮流导通，一个负责灌电流，一个负责拉电流，使其负载能力和开关速度都比普通的方式有很大的提高。推挽输出的供电为 0 伏，高电平为 3.3 伏。

在开漏输出模式时，如果我们控制输出为 0，低电平，则使 N-MOS 管导通，使输出接地，若控制输出为 1 (无法直接输出高电平)，则既不输出高电平，也不输出低电平，为高阻态。为正常使用时必须在外部接上一个上拉电阻。它具有“线与”特性，即很多个开漏模式引脚连接到一起时，只有当所有引脚都输出高阻态，才由上拉电阻提供高电平，此高电平的电压为外部上拉电阻所接电源的电压。若其中一个引脚为低电平，那线路就相当于短路接地，使得整条线路都为低电平，0 伏。

STM32 的 GPIO 输出模式就分为普通推挽输出 (GPIO_Mode_Out_PP)、普通开漏输出

(GPIO_Mode_Out_OD) 及复用推挽输出 (GPIO_Mode_AF_PP)、复用开漏输出 (GPIO_Mode_AF_OD)。

普通推挽输出模式一般应用在输出电平为 0 和 3.3 伏的场合。而普通开漏输出模式一般应用在电平不匹配的场合，如需要输出 5 伏的高电平，就需要在外部接一个上拉电阻，电源为 5 伏，把 GPIO 设置为开漏模式，当输出高阻态时，由上拉电阻和电源向外输出 5 伏的电平。

对于相应的复用模式，则是根据 GPIO 的复用功能来选择的，如 GPIO 的引脚用作串口的输出，则使用复用推挽输出模式。如果用在 IC、SMBUS 这些需要线与功能的复用场合，就使用复用开漏模式。其他不同复用场合的复用模式引脚配置将在具体的例子中讲解。

在使用任何一种开漏模式时，都需要接上拉电阻。

6.2 按键实验分析

了解了 GPIO 的 8 种工作模式之后，立即进行一下小小的测试。如果采用如图 6-2 所示电路，我们的按键 GPIO 端口应该如何进行配置？

有两个方案可以选择，一是采用上拉输入模式，因为按键在没按下的时候，是默认为高电平的，采用内部上拉模式正好符合这个要求。

第二个方案是直接采用浮空输入模式，因为按照这个硬件电路图，在芯片外部接了上拉电阻，其实就没必要再配置成内部上拉输入模式了，因为在外部分上拉与内部上拉效果是一样的。

见图 6-2，配套 STM32 开发板按键硬件原理图。

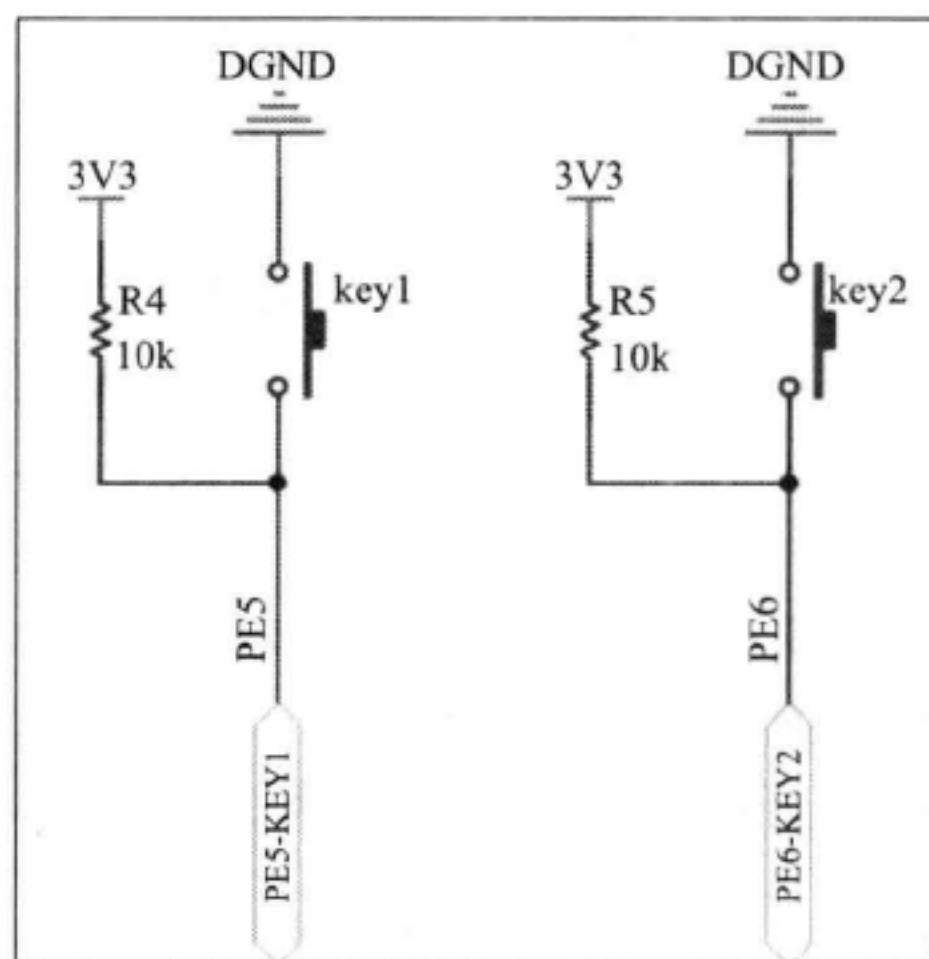


图 6-2 配套 STM32 按键硬件图

6.3 按键代码分析

6.3.1 实验描述及工程文件清单

1. 实验描述

PE5 连接到 key1，用扫描的方式查询是否有按键按下，key1 按下时，LED1 状态取反。

2. 硬件连接

□ PE5-key1

□ PE6-key2

3. 库文件

3.5 版本固件库：

□ startup/start_stm32f10x_hd.c

- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/led.c
- ☐ USER/key.c

6.3.2 配置工程环境

本按键实验中用到了 GPIO 和 RCC 片上外设，所以要把外设函数库文件 FWlib/stm32f10x_gpio.c 和 FWlib/stm32f10x_rcc.c 文件添加到工程模板之中。实验中还使用了 LED 灯，为了重用代码，我们把在前面写好的 led.c 和 led.h 用户文件复制到 USER 目录下，并添加到工程之中。配置工程环境最重要的一步就是别忘记在 stm32f10x_conf.h 文件中把使用到的外设头文件包含进来。见代码清单 6-1。

代码清单 6-1 按键例程的 stm32f10x_conf.h 文件配置

```

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

```

6.3.3 main 文件

顺着代码的执行流程，从 main 函数开始分析，这样阅读和分析别人写的代码更有条理。见代码清单 6-2。

代码清单 6-2 按键例程的 main 函数

```

1. /*
2.  * 函数名：main
3.  * 描述   ：主函数
4.  * 输入   ：无
5.  * 输出   ：无

```

```

6.  */
7. int main(void)
8. {
9.     /* config the led */
10.    LED_GPIO_Config();
11.    LED1( ON );
12.
13.    /*config key*/
14.    Key_GPIO_Config();
15.
16.    while(1)
17.    {
18.        if( Key_Scan(GPIOE,GPIO_Pin_5) == KEY_ON )
19.        {
20.            /*LED1 反转 */
21.            GPIO_WriteBit(GPIOC, GPIO_Pin_3,
22.                (BitAction)((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
23.
24.        }
25.    }
26.}

```

由于采用的为 3.5 版本的库，上电后，启动文件已经调用了 SystemInit() 将我们的系统时钟 SYSCLK 配置为 72 MHz。接着进入到 main 函数，第一步先调用了在 LED 灯例程中编写的 LED_GPIO_Config()，配置 LED 用到的 I/O。再使用 LED1 (ON) 宏把 LED 设置为点亮状态。为了使用 LED 这部分代码，我们只要把前面写的 led.c 和 led.h 文件复制一份，放到本工程目录下，把 led.c 添加到工程就可以了，这样重用代码使开发变得非常方便。关于这部分的具体分析可参考 LED 代码讲解部分。

6.3.4 GPIO 初始化配置

现在我们分析一下紧接下来调用到的 Key_GPIO_Config() 函数。见代码清单 6-3。

代码清单 6-3 Key_GPIO_Config() 函数代码

```

1.  /*
2.  * 函数名：Key_GPIO_Config
3.  * 描述   ：配置按键用到的 I/O 口
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7. void Key_GPIO_Config(void)
8. {
9.     GPIO_InitTypeDef GPIO_InitStructure;
10.
11.    /* 开启按键端口 (PE5) 的时钟 */
12.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE,ENABLE);
13.
14.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;

```

```

15.// GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
16.   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
17.
18.   GPIO_Init(GPIOE, &GPIO_InitStructure);
19.}

```

Key_GPIO_Config() 与 LED 的 GPIO 初始化函数 LED_GPIO_Config() 类似, 区别只是在这个函数中, 要开启的 GPIO 外设时钟为 GPIOE 的时钟, 并且把检测按键用的引脚 PE5 的模式设置为适合按键应用的上拉输入模式 (由于接了外部上拉电阻, 也可以使用浮空输入, 读者可自行修改代码做实验)。在这个函数的第 15 行, 读者注意到这行代码是被注释掉的, 若 GPIO 被设置为输入模式, 不需要设置 GPIO 端口的最大输出速度, 当然, 如果配置了这个速度也没关系, GPIO_Init() 函数会自动忽略它。

在 RCC_APB2PeriphClockCmd() 和 GPIO_InitStructure.GPIO_Pin 的输入参数设置之中, 我们可以用符号 “|”, 同时配置多个参数。如:

```
1. RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE|RCC_APB2Periph_GPIOC, ENABLE);
```

输入参数为 RCC_APB2Periph_GPIOE|RCC_APB2Periph_GPIOC, 这样调用之后, 就把 GPIOE 和 GPIOC 的时钟都开启了。

```
1. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6;
```

以上代码则表示将要同时配置 GPIO 端口的 Pin5 和 Pin6。

6.3.5 利用固件库的数据类型

回到 main 函数中的应用代码, 初始化了按键的 GPIO 之后, 就在死循环里不断调用一个函数 Key_Scan(), 用于扫描按键是否被按下。我们在 MDK 使用技巧中介绍的 “GO To Definition of” 功能追踪它的定义, 见代码清单 6-4。

代码清单 6-4 Key_Scan() 函数代码

```

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

```



```

19.             return KEY_ON;
20.         }
21.     else
22.         return KEY_OFF;
23. }
24. else
25.     return KEY_OFF;
26. }

```

相信延时消抖的原理大家在学习其他单片机时就已经了解了，本函数的功能就是扫描输入参数中指定的引脚，检测其电平变化，并作延时消抖处理，最终对按键消息进行确认。

1) 利用 GPIO_ReadInputDataBit() 读取输入数据，若从相应引脚读取的数据等于 0 (KEY_ON)，低电平，表明可能有按键按下，调用延时函数。否则返回 KEY_OFF，表示按键没有被按下。

2) 延时之后再次利用 GPIO_ReadInputDataBit() 读取输入数据，若依然为低电平，表明确实有按键被按下了。否则返回 KEY_OFF，表示按键没有被按下。

3) 循环调用 GPIO_ReadInputDataBit() 一直检测按键的电平，直至按键被释放，被释放后，返回表示按键被按下的标志 KEY_ON。

以上是按键消抖的流程，调用了库函数 GPIO_ReadInputDataBit()。输入参数为要读取的端口、引脚，返回引脚的输入电平状态，高电平为 1，低电平为 0。见图 6-3。

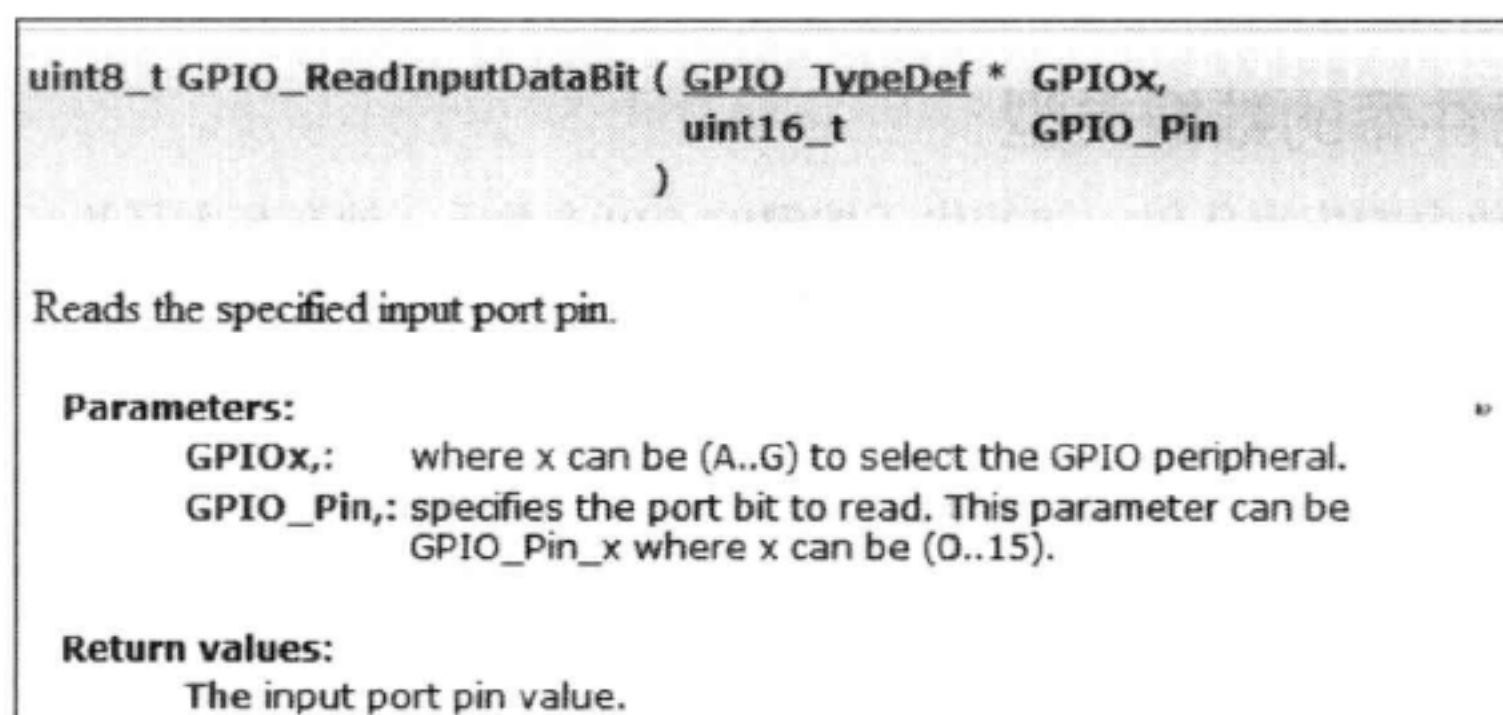


图 6-3 GPIO 输入数据读取函数

但按键消抖并不是本节的重点，而且这样的消抖在实际的工程应用中并无价值。重点是教会大家如何利用 ST 库定义的新数据类型来编写用户函数。在实际的工程应用中，是使用一个周期中断服务函数（在 STM32 可用 SysTick），每隔一段时间调用该函数来检测设备上所有的按键，把按下的键用不同的全局变量标志位记录下来（按键值）。在检测时也是使用短暂延时来消抖（滤波），这个延时比本例程中的延时要短，而且是在中断程序中定时调用的，不是在 main 中死循环调用。

其实 Key_Scan() 函数的形参与 GPIO_ReadInputDataBit() 的形参是一样的，都是 (GPIO_TypeDef* GPIOx, uint16 GPIO_Pin)，如果再在 Key_Scan() 的定义中加入断言，用于输入参数检查，看起来是不是很像 ST 官方的库函数？其实这是我们写的一个用户函数。

这个例子告诉大家，在 stm32f10x.h 文件中的新数据类型，我们不但可以利用它们来定义变

量，还应善于利用这些数据类型来编写用户函数。如这个 Key_Scan() 函数，由于使用了这些引脚类型形参，在其他不同的工程之中，我们就可以在调用时通过输入不同的实参来检测其他按键的引脚了。

如在调用 Key_Scan() 函数时，把实参改成 (GPIOE, GPIO_Pin_6)，就可以用 Key2 来控制 LED1 了（当然，GPIO_Pin_6 要在 Key_GPIO_Config() 中初始化）。是不是很方便？利用官方的库，我们可以很方便地开发出这一类用户函数，这就是库的魅力！

6.3.6 实现 LED 反转

在 main 函数中，检测到有按键被按下之后，就开始执行 LED 反转的操作。见代码清单 6-5。

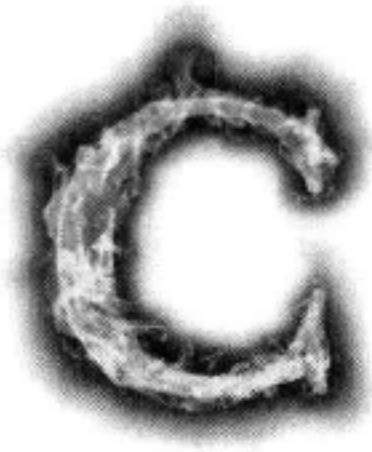
代码清单 6-5 LED 反转操作

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

这一段代码首先调用了 GPIO_ReadOutputDataBit() 函数，读取 PC3 的当前输出电平，然后再用 1 减去读取的电平数据状态，相当于获取一个与当前输出相反的状态，再把这个相反的状态利用 GPIO_WriteBit() 函数写入到 PC3，从而实现了输出状态取反的功能。大家会发现，这实在太复杂了，我们只不过是要取反输出，在 51 单片机可以直接利用“PA0= ~ PA0”就可以完成了，而在这里使用库的时候，我们竟然要先读取状态再计算出反状态，最后再写入新状态。能不能也像单片机那样使用呢？答案是肯定的，我们可以采用 Cortex-M3 的位带操作方式来实现同样的功能。

6.3.7 实验现象

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



第 7 章

EXTI 之按键中断实验

EXTI (External Interrupt) 就是指外部中断，通过 GPIO 检测输入脉冲，引起中断事件，打断原来的代码执行流程，进入到中断服务函数中进行处理，处理完后再返回到中断之前的代码中执行。

前面提到，STM32 的所有 GPIO 都可以用作外部中断源的输入端，利用这个特性，我们可以把按键轮询检测改为由中断来处理，大大提高软件执行的效率。

7.1 STM32 的中断和异常

Cortex 内核具有强大的异常响应系统，它把能够打断当前代码执行流程的事件分为异常 (exception) 和中断 (interrupt)，并把它们用一个表管理起来，编号为 0 ~ 15 的称为内核异常，而 16 以上的则称为外部中断 (外是相对内核而言)，这个表就称为中断向量表。

而 STM32 对这个表重新进行了编排，把编号从 -3 至 6 的中断向量定义为系统异常，编号为负的内核异常不能被设置优先级，如复位 (Reset)、不可屏蔽中断 (NMI)、硬错误 (Hardfault)。从编号 7 开始的为外部中断，这些中断的优先级都是可以自行设置的。详细的 STM32 中断向量表见表 7-1。

表 7-1 STM32 中断向量表

优先级	优先级类型	名 称	说 明	地 址
-	-	-	保留	0x0000_0000
-3	固定	Reset	复位	0x0000_0004
-2	固定	NMI	不可屏蔽中断 RCC 时钟安全系统 (CSS) 联接到 NMI 向量	0x0000_0008
-1	固定	硬件失效 (HardFault)	所有类型的失效	0x0000_000C
0	可设置	存储管理 (MemManage)	存储器管理	0x0000_0010
1	可设置	总线错误 (BusFault)	预取指失败，存储器访问失败	0x0000_0014
2	可设置	错误应用 (UsageFault)	未定义的指令或非法状态	0x0000_0018
-	-	-	保留	0x0000_001C
3	可设置	SVCall	通过 SWI 指令的系统服务调用	~ 0x0000_002B
4	可设置	调试监控 (DebugMonitor)	调试监控器	0x0000_002C

(续)

优先级	优先级类型	名 称	说 明	地 址
-	-	-	保留	0x0000_0030
5	可设置	PendSV	可挂起的系统服务	0x0000_0034
6	可设置	SysTick	系统嘀嗒定时器	0x0000_0038
7	可设置	WWDG	窗口定时器中断	0x0000_003C
8	可设置	PVD	连到 EXTI 的电源电压检测 (PVD) 中断	0x0000_0040
9	可设置	TAMPER	侵入检测中断	0x0000_0044
10	可设置	RTC	实时时钟 (RTC) 全局中断	0x0000_0048
11	可设置	FLASH	闪存全局中断	0x0000_004C
12	可设置	RCC	复位和时钟控制 (RCC) 中断	0x0000_0054
13	可设置	EXTI0	EXTI 线 0 中断	0x0000_0058
14	可设置	EXTI1	EXTI 线 1 中断	0x0000_005C
15	可设置	EXTI2	EXTI 线 2 中断	0x0000_0060
16	可设置	EXTI3	EXTI 线 3 中断	0x0000_0064
17	可设置	EXTI4	EXTI 线 4 中断	0x0000_0068
18	可设置	DMA1 通道 1	DMA1 通道 1 全局中断	0x0000_006C
19	可设置	DMA1 通道 2	DMA1 通道 2 全局中断	0x0000_0070
20	可设置	DMA1 通道 3	DMA1 通道 3 全局中断	0x0000_0074
21	可设置	DMA1 通道 4	DMA1 通道 4 全局中断	0x0000_0078
22	可设置	DMA1 通道 5	DMA1 通道 5 全局中断	0x0000_007C
23	可设置	DMA1 通道 6	DMA1 通道 6 全局中断	0x0000_0080
24	可设置	DMA1 通道 7	DMA1 通道 7 全局中断	0x0000_0084
25	可设置	ADC1_2	ADC1 和 ADC2 的全局中断	0x0000_0088
26	可设置	USB_HP_CAN_TX	USB 高优先级或 CAN 发送中断	0x0000_008C
27	可设置	USB_LP_CAN_RX0	USB 低优先级或 CAN 接收 0 中断	0x0000_0090
28	可设置	CAN_RX1	CAN 接收 1 中断	0x0000_0094
29	可设置	CAN_SCE	CAN SCE 中断	0x0000_0098
30	可设置	EXTI9_5	EXTI 线 [9:5] 中断	0x0000_009C
31	可设置	TIM1_BRK	TIM1 刹车中断	0x0000_00A0
32	可设置	TIM1_UP	TIM1 更新中断	0x0000_00A4
33	可设置	TIM1_TRG_COM	TIM1 触发和通信中断	0x0000_00A8
34	可设置	TIM1_CC	TIM1 捕获比较中断	0x0000_00AC
35	可设置	TIM2	TIM2 全局中断	0x0000_00B0

(续)

优先级	优先级类型	名 称	说 明	地 址
36	可设置	TIM3	TIM3 全局中断	0x0000_00B4
37	可设置	TIM4	TIM4 全局中断	0x0000_00B8
38	可设置	I2C1_EV	I ² C1 事件中断	0x0000_00BC
39	可设置	I2C1_ER	I ² C1 错误中断	0x0000_00C0
40	可设置	I2C2_EV	I ² C2 事件中断	0x0000_00C4
41	可设置	I2C2_ER	I ² C2 错误中断	0x0000_00C8
42	可设置	SPI1	SPI1 全局中断	0x0000_00CC
43	可设置	SPI2	SPI2 全局中断	0x0000_00D0
44	可设置	USART1	USART1 全局中断	0x0000_00D4
45	可设置	USART2	USART2 全局中断	0x0000_00D8
46	可设置	USART3	USART3 全局中断	0x0000_00DC
47	可设置	EXTI15_10	EXTI 线 [15:10] 中断	0x0000_00E0
48	可设置	RTCAlarm	连到 EXTI 的 RTC 闹钟中断	0x0000_00E4
49	可设置	USB 唤醒	连到 EXTI 的从 USB 待机唤醒中断	0x0000_00E8
50	可设置	TIM8_BRK	TIM8 刹车中断	0x0000_00EC
51	可设置	TIM8_UP	TIM8 更新中断	0x0000_00F0
52	可设置	TIM8_TRG_COM	TIM8 触发和通信中断	0x0000_00F4
53	可设置	TIM8_CC	TIM8 捕获比较中断	0x0000_00F8
54	可设置	ADC3	ADC3 全局中断	0x0000_00FC
55	可设置	FSMC	FSMC 全局中断	0x0000_0100
56	可设置	SDIO	SDIO 全局中断	0x0000_0104
57	可设置	TIM5	TIM5 全局中断	0x0000_0108
58	可设置	SPI3	SPI3 全局中断	0x0000_010C
59	可设置	UART4	UART4 全局中断	0x0000_0110
60	可设置	UART5	UART5 全局中断	0x0000_0114
61	可设置	TIM6	TIM6 全局中断	0x0000_0118
62	可设置	TIM7	TIM7 全局中断	0x0000_011C
63	可设置	DMA2 通道 1	DMA2 通道 1 全局中断	0x0000_0120
64	可设置	DMA2 通道 2	DMA2 通道 2 全局中断	0x0000_0124
65	可设置	DMA2 通道 3	DMA2 通道 3 全局中断	0x0000_0128
66	可设置	DMA2 通道 4_5	DMA2 通道 4 和 DMA2 通道 5 全局中断	0x0000_012C

这个表可以从《STM32 参考手册》找到，但我们一般建议从启动文件 startup_stm32f10x_hd.s

中查找的，因为不同型号的 STM32 芯片，中断向量表稍微有点区别，在启动文件中，已经有相应芯片可用的全部中断向量。而且在编写中断服务函数时，需要从启动文件中定义的中断向量表查找中断服务函数名。

7.2 NVIC 中断控制器

STM32 的中断如此之多，配置起来并不容易，因此我们需要一个强大而方便的中断控制器 NVIC (Nested Vectored Interrupt Controller)。NVIC 是属于 Cortex 内核的器件，不可屏蔽中断 (NMI) 和外部中断都由它来处理，而 SYSTICK 不是由 NVIC 来控制的。见图 7-1。

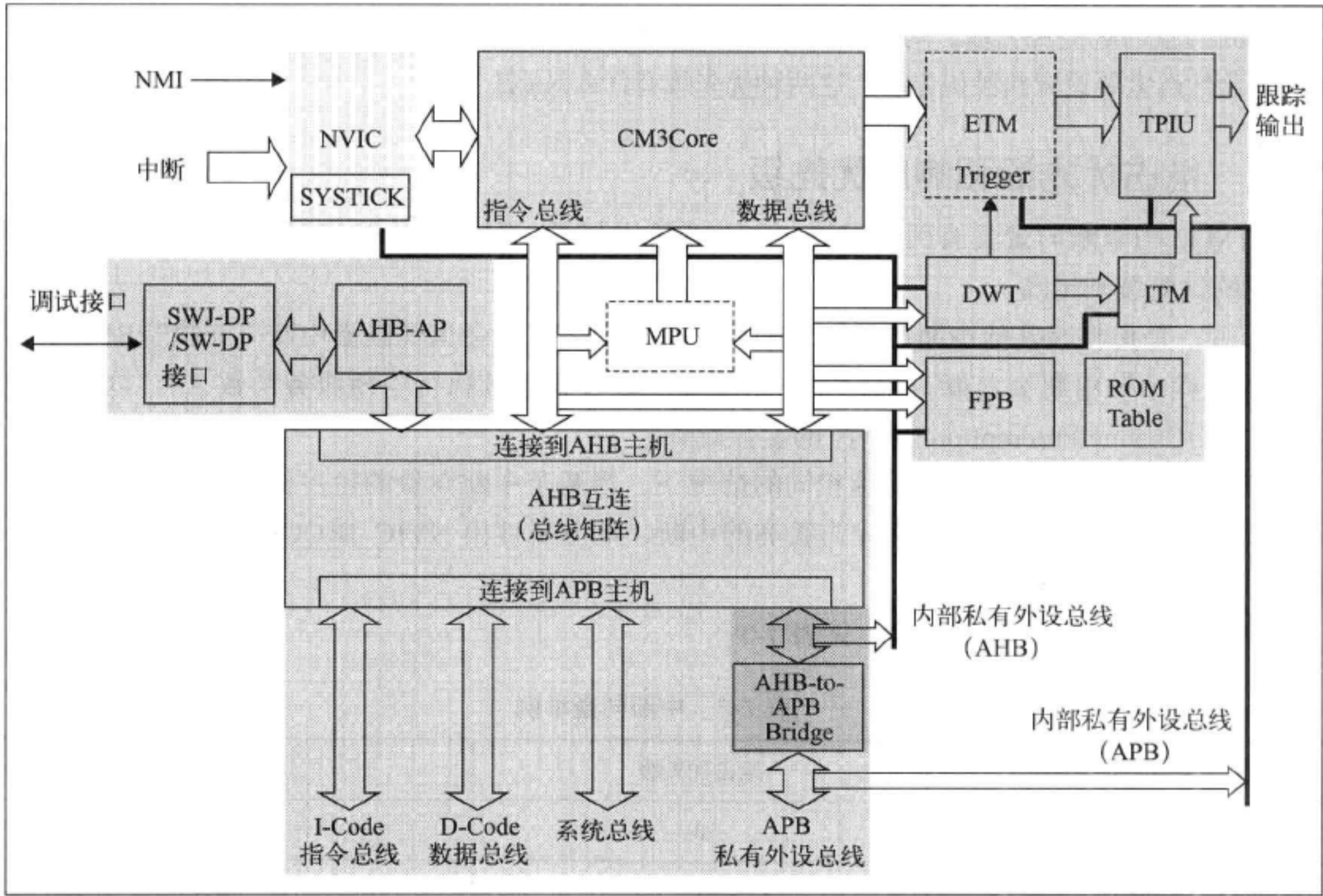


图 7-1 NVIC 在内核中的位置

7.2.1 NVIC 结构体成员

当我们要使用 NVIC 来配置中断时，自然想到 ST 库肯定也已经把它封装成库函数了。查找库帮助文档，发现在 Modules->STM32F10x_StdPeriph_Driver->misc 查找到一个 NVIC_Init() 函数。对 NVIC 初始化，首先要定义并填充一个 NVIC_InitTypeDef 类型的结构体。这个结构体有 4 个成员，见表 7-2。

表 7-2 NVIC 结构体成员

结构体成员名称	描 述
NVIC_IRQChannel	需要配置的中断向量
NVIC_IRQChannelCmd	使能或关闭相应中断向量的中断响应
NVIC_IRQChannelPreemptionPriority	配置相应中断向量抢占优先级
NVIC_IRQChannelSubPriority	配置相应中断向量的响应优先级

前面两个结构体成员都很好理解，首先要用 NVIC_IRQChannel 参数来选择将要配置的中断向量，用 NVIC_IRQChannelCmd 参数来进行使能（ENABLE）或关闭（DISABLE）该中断。在 NVIC_IRQChannelPreemptionPriority 成员要配置中断向量的抢占优先级，在 NVIC_IRQChannelSubPriority 需要配置中断向量的响应优先级。对于中断的配置，最重要的便是配置其优先级，但 STM32 的同一个中断向量为什么需要设置两种优先级？这两种优先级有什么区别？

7.2.2 抢占优先级和响应优先级

STM32 的中断向量具有两个属性，一个为抢占属性，另一个为响应属性，其属性编号越小，表明它的优先级别越高。

抢占，是指打断其他中断的属性，即因为具有这个属性会出现嵌套中断（在执行中断服务函数 A 的过程中被中断 B 打断，执行完中断服务函数 B 再继续执行中断服务函数 A），抢占属性由 NVIC_IRQChannelPreemptionPriority 的参数配置。

而响应属性则应用在抢占属性相同的情况下，当两个中断向量的抢占优先级相同时，如果两个中断同时到达，则先处理响应优先级高的中断，响应属性由 NVIC_IRQChannelSubPriority 参数配置。

例如，现在有三个中断向量，见表 7-3。

表 7-3 中断向量举例

中 断 向 量	抢占优先级	响应优先级
A	0	0
B	1	0
C	1	1

若内核正在执行 C 的中断服务函数，则它能被抢占优先级更高的中断 A 打断，由于 B 和 C 的抢占优先级相同，所以 C 不能被 B 打断。但如果 B 和 C 中断是同时到达的，内核就会首先响应响应优先级别更高的 B 中断。

7.2.3 NVIC 的优先级组

在配置优先级的时候，还要注意一个很重要的问题，即中断种类的数量。NVIC 只可以配置 16 种中断向量的优先级，也就是说，抢占优先级和响应优先级的数量由一个 4 位的数字来决定，把这个 4 位数字的位数分配成抢占优先级部分和响应优先级部分。有 5 组分配方式：

- 第 0 组：所有 4 位用来配置响应优先级。即 16 种中断向量具有都不相同的响应优先级。
- 第 1 组：最高 1 位用来配置抢占优先级，低 3 位用来配置响应优先级。表示有 $2^1=2$ 种级别的抢占优先级（0 级，1 级），有 $2^3=8$ 种响应优先级，即在 16 种中断向量之中，有 8 种中断，其抢占优先级都为 0 级，而它们的响应优先级分别为 0 ~ 7，其余 8 种中断向量的抢占优先级则都为 1 级，响应优先级分别为 0 ~ 7。
- 第 2 组：2 位用来配置抢占优先级，2 位用来配置响应优先级。即 $2^2=4$ 种抢占优先级， $2^2=4$ 种响应优先级。
- 第 3 组：高 3 位用来配置抢占优先级，最低 1 位用来配置响应优先级。即有 8 种抢占优先级，2 种响应优先级。
- 第 4 组：所有 4 位用来配置抢占优先级，即 NVIC 配置的 $2^4=16$ 种中断向量都是只有抢占属性，没有响应属性。

要配置这些优先级组，可以采用库函数 `NVIC_PriorityGroupConfig()`，可输入的参数为 `NVIC_PriorityGroup_0 ~ NVIC_PriorityGroup_4`，分别为以上介绍的 5 种分配组。

于是，有读者觉得疑惑了，如此强大的 STM32，所有 GPIO 都能够配置成外部中断，USART、ADC 等外设也有中断，而 NVIC 只能配置 16 种中断向量，那么在某个工程中使用超过 16 个中断怎么办呢？注意 NVIC 能配置的是 16 种中断向量，而不是 16 个，当工程中有超过 16 个中断向量时，必然有两个以上的中断向量是使用相同的中断种类，而具有相同中断种类的中断向量不能互相嵌套。

STM2 单片机的所有 I/O 端口都可以配置为 EXTI 中断模式，用来捕捉外部信号，可以配置为下降沿中断、上升沿中断和上升下降沿中断这三种模式。它们以如图 7-2 所示方式连接到 16 个外部中断 / 事件线上。

7.3 EXTI 外部中断

STM32 的所有 GPIO 都引入到 EXTI 外部中断线上，使得所有的 GPIO 都能作为外部中断的输入源。GPIO 与 EXTI 的连接方式见图 7-2。

观察图 7-2 可知，PA0 ~ PG0 连接到 EXTI0、PA1 ~ PG1 连接到 EXTI1、……、PA15 ~ PG15 连接到 EXTI15。这里大家要注意的是：PAx ~ PGx 端口的中断事件都连接到了 EXTIx，即同一时刻 EXTIx 只能响应一个端口的事件触发，不能够同一时间响应所有 GPIO 端口的事件，但可以分时复用。它可以配置为上升沿触发、下降沿触发或双边沿触发。EXTI 最普通的应用就是接上一个按键，设置为下降沿触发，用中断来检测按键。

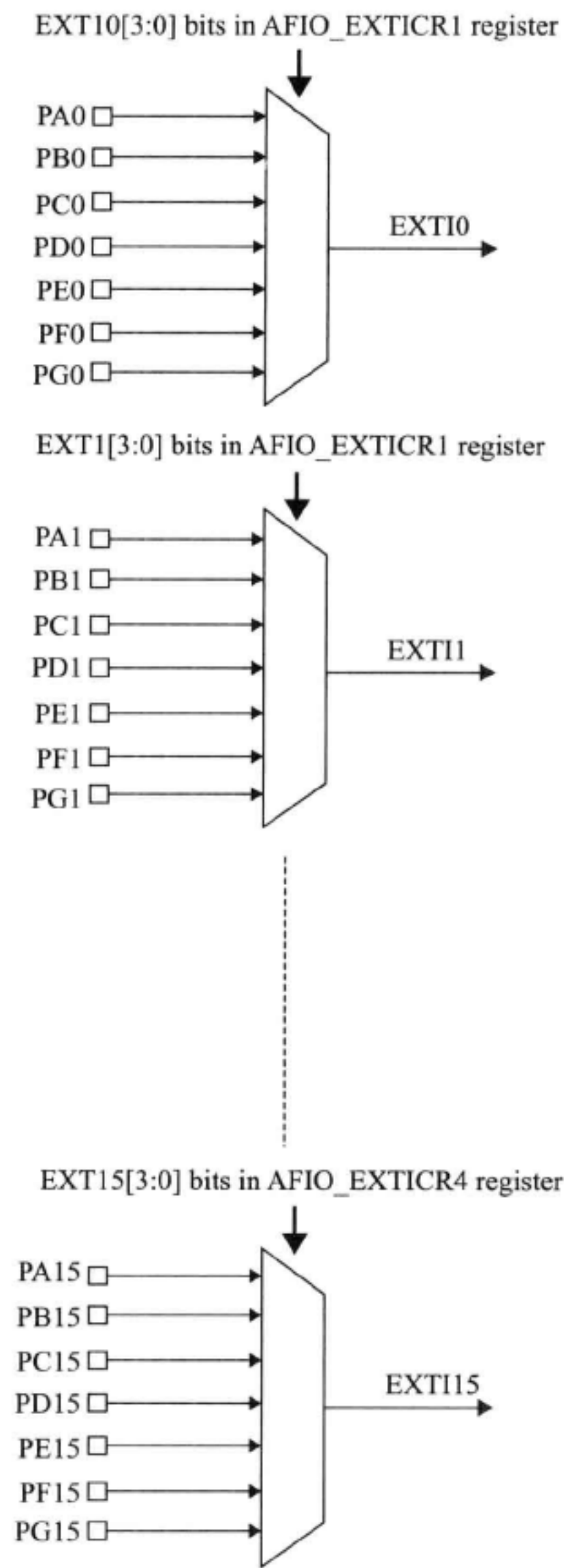


图 7-2 EXTI 与 GPIO 连接图

7.4 中断检测按键实验分析

7.4.1 实验描述及工程文件清单

1. 实验描述

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

2. 硬件连接

- ☐ PE5 – key1
- ☐ PE6 – key2

3. 库文件

3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_exti.c
- ☐ FWlib/misc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/led.c
- ☐ USER/exti.c

配套 STM32 开发板按键硬件原理图见图 7-3。

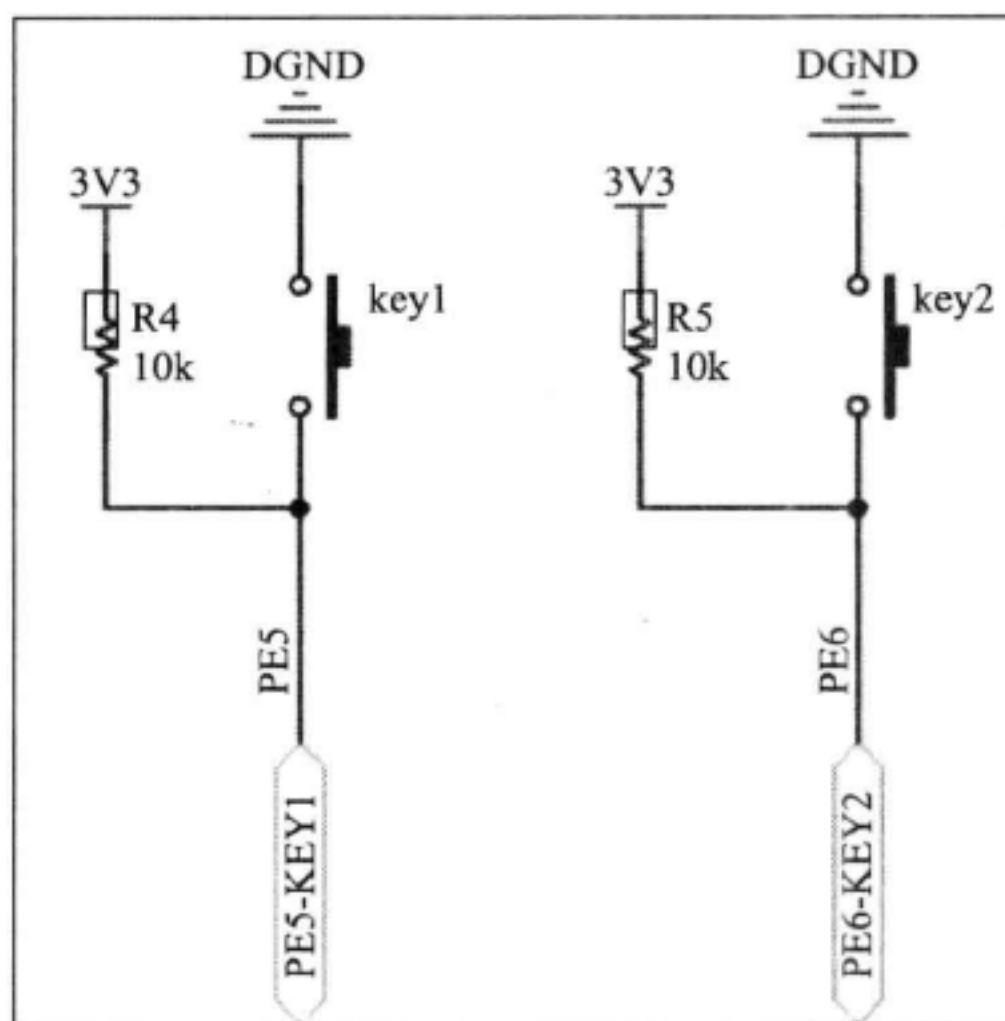


图 7-3 配套 STM32 开发板按键硬件图

7.4.2 配置工程环境

本中断检测按键实验照例使用了 GPIO 和 RCC 片上外设，由于还使用到了中断，所以比上一个按键实验要多使用两个库文件，分别为 FWlib/stm32f10x_exti.c 和 FWlib/misc.c，必须把这两个文件也添加到工程之中。其中 stm32f10x_exti.c 文件包含了支持 EXTI 配置和操作的相关库函数；而 misc.c 文件则包含了 NVIC 的配置函数。本实验中我们还会在 stm32f10x_it.c 文件中编写中断服务函数。

添加了所需要的库文件、用户文件之后，要在 stm32f10x_conf.h 文件中配置使用到的头文件，见代码清单 7-1。

代码清单 7-1 EXTI 例程的 stm32f10x_conf.h 文件配置

```

1. /*****
2.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
3.  * @author  MCD Application Team
4.  * @version V3.5.0
5.  * @date    08-April-2011
6.  * @brief   Library configuration file.
7.  *****/
8. #include "stm32f10x_exti.h"
9. #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "misc.h" /

```

7.4.3 main 文件

我们从 main 函数开始分析，见代码清单 7-2。

代码清单 7-2 EXTI 例程的 main 函数

```

1.  /*
2.   * 函数名：main
3.   * 描述   ：主函数
4.   * 输入   ：无
5.   * 输出   ：无
6.   */
7.  int main(void)
8.  {
9.      /* config the led */
10.     LED_GPIO_Config();
11.     LED1( ON );
12.
13.     /* exti line config */
14.     EXTI_PE5_Config();
15.
16.     /* wait interrupt */
17.     while(1)
18.     {
19.     }
20. }
```

使用 LED_GPIO_Config() 配置好 LED 用到的 I/O 后，调用 LED1 (ON) 点亮一盏 LED 灯。这两个函数的具体讲解可参考前面的教程。

7.4.4 配置外部中断

现在我们重点分析 EXTI_PE5_Config() 这个函数，这是一个在用户文件 exti.c 中实现的函数，它完成了配置一个 I/O 为 EXTI 中断的一般步骤，主要有以下功能：

- 1) 使能 EXTIx 线的时钟和第二功能 AFIO 时钟。
- 2) 配置 EXTIx 线的中断优先级。
- 3) 配置 EXTI 中断线 I/O。
- 4) 选定要配置为 EXTI 的 I/O 口线和 I/O 口的工作模式。
- 5) EXTI 中断线工作模式配置。

EXTI_PE5_Config() 代码见代码清单 7-3。

代码清单 7-3 EXTI_PE5_Config() 代码

```

1.  /*
2.   * 函数名：EXTI_PE5_Config
```

```

3.  * 描述   : 配置 PE5 为线中断口, 并设置中断优先级
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void EXTI_PE5_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     EXTI_InitTypeDef EXTI_InitStructure;
12.
13.     /* config the extiline(PE5) clock and AFIO clock */
14.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE | RCC_APB2Periph_AFIO, ENABLE);
15.
16.     /* config the NVIC(PE5) */
17.     NVIC_Configuration();
18.
19.     /* EXTI line gpio config(PE5) */
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
21.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD; // 上拉输入
22.     GPIO_Init(GPIOE, &GPIO_InitStructure);
23.
24.     /* EXTI line(PE5) mode config */
25.     GPIO_EXTILineConfig(GPIO_PortSourceGPIOE, GPIO_PinSource5);
26.     EXTI_InitStructure.EXTI_Line = EXTI_Line5;
27.     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
28.     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; // 下降沿中断
29.
30.     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
31.     EXTI_Init(&EXTI_InitStructure);
32.}

```

7.4.5 AFIO 时钟

代码清单 7-3 的第 14 行, 调用 `RCC_APB2PeriphClockCmd()` 时还输入了参数 `RCC_APB2Periph_AFIO`, 表示开启 AFIO 的时钟。见表 7-4。

AFIO (alternate-function I/O), 指 GPIO 端口的复用功能, GPIO 除了用作普通的输入输出 (主功能), 还可以作为片上外设的复用输入输出, 如串口、ADC, 这些就是复用功能。大多数 GPIO 都有一个默认复用功能, 有的 GPIO 还有重映射功能。重映射功能是指把原来属于 A 引脚的默认复用功能, 转移到 B 引脚进行使用, 前提是 B 引脚具有这个重映射功能。

当把 GPIO 用作 EXTI 外部中断或使用重映射功能的时候, 必须开启 AFIO 时钟, 而在使用默认复用功能的时候, 就不必开启 AFIO 时钟了。

表 7-4 《STM32 数据手册》中的 GPIO 引脚功能说明（部分）

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCSP64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
J7	G6	-	-	41	63	PE10	I/O	FT	PE10	FSMC_D7	TIM1_CH2N
H8	H6	-	-	42	64	PE11	I/O	FT	PE11	FSMC_D8	TIM1_CH2
J8	J6	-	-	43	65	PE12	I/O	FT	PE12	FSMC_D9	TIM1_CH3N
K8	K6	-	-	44	66	PE13	I/O	FT	PE13	FSMC_D10	TIM1_CH3
L8	G7	-	-	45	67	PE14	I/O	FT	PE14	FSMC_D11	TIM1_CH4
M8	H7	-	-	46	68	PE15	I/O	FT	PE15	FSMC_D12	TIM1_BKIN
M9	J7	G3	29	47	69	PB10	I/O	FT	PB10	I2C2_SCL/USART3_TX	TIM2_CH3
M10	K7	F3	30	48	70	PB11	I/O	FT	PB11	I2C2_SCL/USART3_RX	TIM2_CH4

7.4.6 NVIC 初始化配置

在 EXTI_PE5_Config() 代码的第 17 行调用了 NVIC_Configuration(), 这是用户编写的用来配置 NVIC 控制器的函数。其实现见代码清单 7-4。

代码清单 7-4 NVIC_Configuration()

```
1. /*
2.  * 函数名: NVIC_Configuration
3.  * 描述   : 配置嵌套向量中断控制器 NVIC
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void NVIC_Configuration(void)
9. {
10.  NVIC_InitTypeDef NVIC_InitStructure;
11.
12.  /* Configure one bit for preemption priority */
13.  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
14.
15.  /* 配置 P[A|B|C|D|E]5 为中断源 */
16.  NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;
17.  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
18.  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
19.  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
20.  NVIC_Init(&NVIC_InitStructure);
21. }
```

本代码的第 13 行调用了 NVIC_PriorityGroupConfig() 库函数，把 NVIC 中断优先级分组设置为第 1 组。接下来开始向 NVIC 初始化结构体写入参数 .NVIC_IRQChannel = EXTI9_5_IRQn，表示要配置的为 EXTI 第 5 ~ 9 线的中断向量。因为按键 PE5 对应的 EXTI 线为 EXTI5，而从

EXTI5 ~ EXTI9 线是使用同一个中断向量的，所以只能写入 EXTI9_5_IRQn 这个参数。这些可写入的参数可以在 stm32f10x.h 文件的 IRQn 类型定义中查找到。

然后配置抢占优先级和响应优先级，因为这个工程简单，就直接把它设置为最高级中断。填充完结构体，别忘记最后要调用 NVIC_Init() 函数来向寄存器写入参数。

7.4.7 EXTI 初始化配置

回到 EXTI_PE5_Config() 代码中，配置好 NVIC 后，还要对 GPIOE 进行初始化，这部分和按键轮询的设置类似。

接下来，调用 GPIO_EXTILineConfig() 函数把 GPIOE、Pin5 设置为 EXTI 输入线。见图 7-4。

选择好了 GPIO，开始填写 EXTI 的初始化结构体。从这些参数的名字，相信读者已经知道如何把它应用到按键检测中。

1) .EXTI_Line = EXTI_Line5；给 EXTI_Line 成员赋值。选择 EXTI_Line5 线进行配置，因为按键的 PE5 连接到了 EXTI_Line5。

2) .EXTI_Mode = EXTI_Mode_Interrupt；给 EXTI_Mode 成员赋值。把 EXTI_Line5 的模式设置为中断模式 (EXTI_Mode_Interrupt)。这个结构体成员也可以赋值为事件模式 EXTI_Mode_Event，这个模式不会立刻触发中断，而只是在寄存器上把相应的事件标志位置 1，应用这个模式需要不停地查询相应的寄存器。

3) .EXTI_Trigger = EXTI_Trigger_Falling；给 EXTI_Trigger 成员赋值。把触发方式 (EXTI_Trigger) 设置为下降沿触发 (EXTI_Trigger_Falling)。

4) .EXTI_LineCmd = ENABLE；给 EXTI_LineCmd 成员赋值。把 EXTI_LineCmd 设置为使能。

5) 最后调用 EXTI_Init() 把 EXTI 初始化结构体的参数写入寄存器。

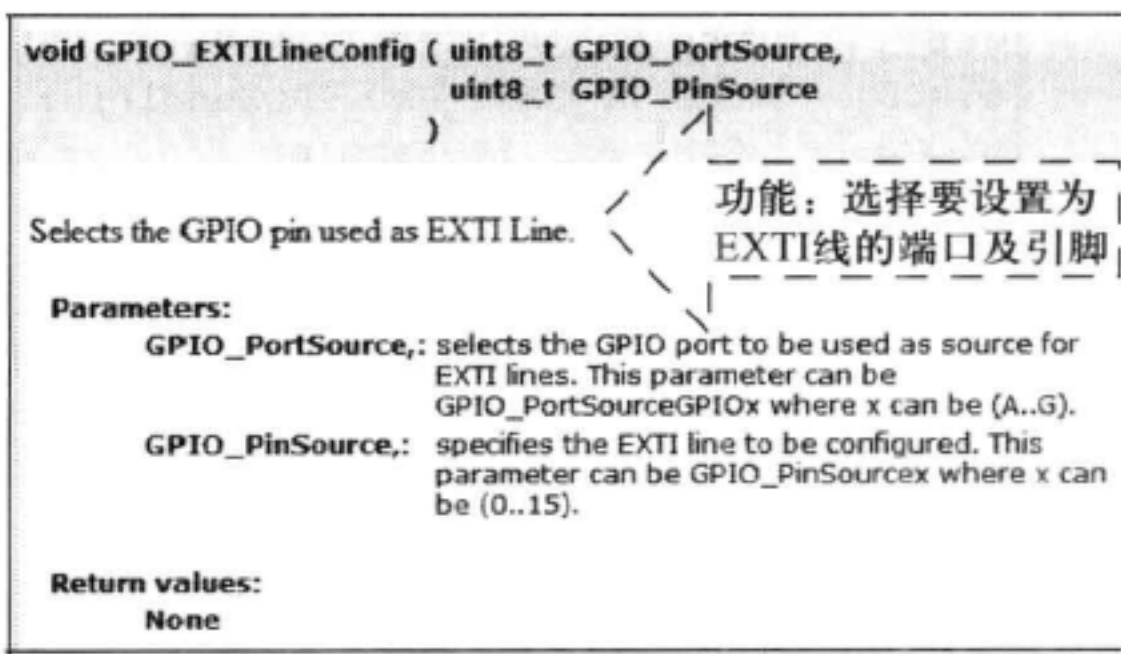


图 7-4 EXTI 中断源配置函数

7.4.8 编写中断服务函数

在这个 EXTI 设置中我们把 PE5 连接到内部的 EXTI5，GPIO 配置为上拉输入，工作在下降沿中断。在外围电路上我们将 PE5 接到了 key1 上。当按键没有按下时，PE5 始终为高，当按键按下时 PE5 变为低，从而 PE5 上产生一个下降沿跳变，EXTI5 会捕捉到这一跳变，并产生相应的中断，中断服务程序在 stm32f10x_it.c 中实现。

stm32f10x_it.c 文件是专门用来存放中断服务函数的。文件中默认只有几个关于系统异常的中断服务函数，而且都是空函数，在需要的时候自行编写。那么中断服务函数名是不是可以自己定义呢？不可以。中断服务函数的名字必须要与启动文件 startup_stm32f10x_hd.s 中的中断向量表定义一致。在启动文件中定义的部分向量表见代码清单 7-5。

代码清单 7-5 中断向量表

1. DCD	EXTI0_IRQHandler	; EXTI Line 0
2. DCD	EXTI1_IRQHandler	; EXTI Line 1
3. DCD	EXTI2_IRQHandler	; EXTI Line 2
4. DCD	EXTI3_IRQHandler	; EXTI Line 3
5. DCD	EXTI4_IRQHandler	; EXTI Line 4
6. DCD	DMA1_Channel1_IRQHandler	; DMA1 Channel 1
7. DCD	DMA1_Channel2_IRQHandler	; DMA1 Channel 2
8. DCD	DMA1_Channel3_IRQHandler	; DMA1 Channel 3
9. DCD	DMA1_Channel4_IRQHandler	; DMA1 Channel 4
10. DCD	DMA1_Channel5_IRQHandler	; DMA1 Channel 5
11. DCD	DMA1_Channel6_IRQHandler	; DMA1 Channel 6
12. DCD	DMA1_Channel7_IRQHandler	; DMA1 Channel 7
13. DCD	ADC1_2_IRQHandler	; ADC1 & ADC2
14. DCD	USB_HP_CAN1_TX_IRQHandler	; USB High Priority or CAN1 TX
15. DCD	USB_LP_CAN1_RX0_IRQHandler	; USB Low Priority or CAN1 RX0
16. DCD	CAN1_RX1_IRQHandler	; CAN1 RX1
17. DCD	CAN1_SCE_IRQHandler	; CAN1 SCE
18. DCD	EXTI9_5_IRQHandler	; EXTI Line 9..5
19. DCD	TIM1_BRK_IRQHandler	; TIM1 Break
20. DCD	TIM1_UP_IRQHandler	; TIM1 Update
21. DCD	TIM1_TRG_COM_IRQHandler	; TIM1 Trigger and Commutation
22. DCD	TIM1_CC_IRQHandler	; TIM1 Capture Compare
23. DCD	TIM2_IRQHandler	; TIM2
24. DCD	TIM3_IRQHandler	; TIM3
25. DCD	TIM4_IRQHandler	; TIM4
26. DCD	I2C1_EV_IRQHandler	; I2C1 Event
27. DCD	I2C1_ER_IRQHandler	; I2C1 Error
28. DCD	I2C2_EV_IRQHandler	; I2C2 Event
29. DCD	I2C2_ER_IRQHandler	; I2C2 Error
30. DCD	SPI1_IRQHandler	; SPI1
31. DCD	SPI2_IRQHandler	; SPI2
32. DCD	USART1_IRQHandler	; USART1
33. DCD	USART2_IRQHandler	; USART2
34. DCD	USART3_IRQHandler	; USART3
35. DCD	EXTI15_10_IRQHandler	; EXTI Line 15..10

在第 18 行, EXTI9_5_IRQHandler 表示为 EXTI5 ~ EXTI9 中断向量的服务函数名。于是,我们就可以在 stm32f10x_it.c 文件中加入名为 EXTI9_5_IRQHandler() 的函数,见代码清单 7-6。

代码清单 7-6 EXTI9_5_IRQHandler() 中断服务函数

```

1. /* I/O 线中断, 中断线为 PE5 */
2. void EXTI9_5_IRQHandler(void)
3. {
4.     if(EXTI_GetITStatus(EXTI_Line5) != RESET) // 确保是否产生了 EXTI Line 中断
5.     {
6.         // LED1 取反
7.         GPIO_WriteBit(GPIOC, GPIO_Pin_3,
8.             (BitAction)((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
9.         EXTI_ClearITPendingBit(EXTI_Line5); // 清除中断标志位
10.    }
11.}

```

其内容比较容易理解,进入中断后,调用库函数 EXTI_GetITStatus() 来重新检查是否产生了

EXTI_Line 中断, 接下来把 LED 取反, 操作完毕后, 调用 EXTI_ClearITPendingBit() 清除中断标志位再退出中断服务函数。这两个函数的解释见图 7-5 和图 7-6。

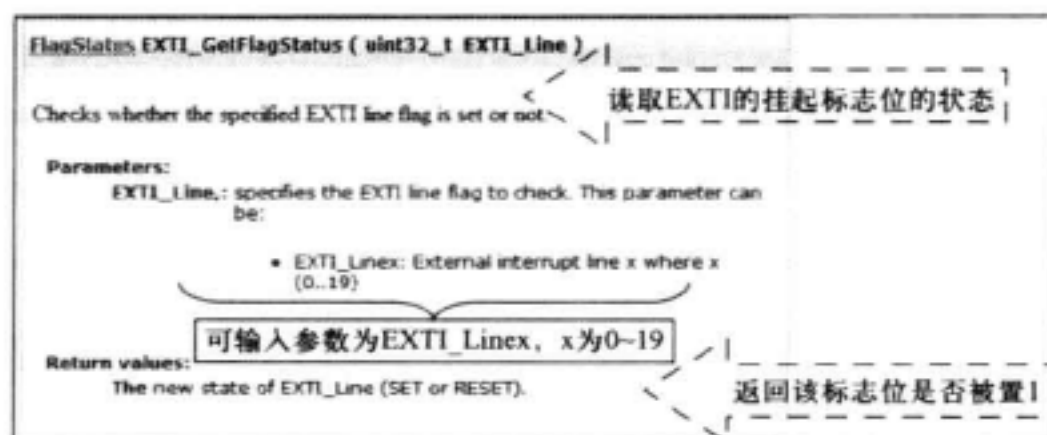


图 7-5 EXTI 状态检查函数



图 7-6 EXTI 清除标志位函数

这两种函数在 ST 库函数中经常见到, 当我们要读取某外设的状态时, 可调用该外设的 XXX_GetFlagStatus() 函数来获取该状态。一般也有 XXX_ClearFlag() 库函数可供调用, 进行相应的标志位清除。

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

- 1) EXTI0_IRQHandler ; EXTI Line 0
EXTI1_IRQHandler ; EXTI Line 1
EXTI2_IRQHandler ; EXTI Line 2
EXTI3_IRQHandler ; EXTI Line 3
EXTI4_IRQHandler ; EXTI Line 4
- 2) EXTI9_5_IRQHandler ; EXTI Line 9..5
EXTI15_10_IRQHandler ; EXTI Line 15..10

中断线在 5 之后的就不能像 0 ~ 4 那样只有单独一个函数名, 都必须写成 EXTI9_5_IRQHandler 和 EXTI15_10_IRQHandler。假如写成 EXTI5_IRQHandler、EXTI6_IRQHandler、...、EXTI15_IRQHandler 编译器是不会报错的, 不过中断服务程序不能工作。所以如果不知道这样的区别, 会浪费很多时间来查找错误。

7.4.9 实验现象

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



第 8 章

串口通信（USART）

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

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

8.1 异步串口通信协议

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

为实现最迫切的需求，利用串口来帮助我们调试程序，本章介绍串口最基本、最常用的方法：全双工、异步通信方式。图 8-1 为异步串口通信协议。

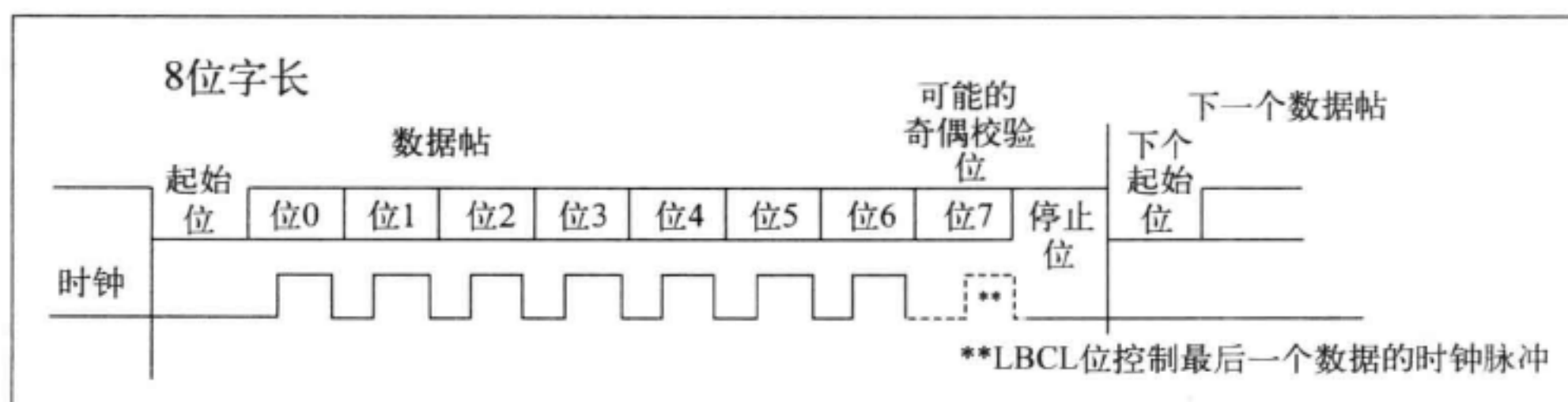


图 8-1 异步串口通信协议

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

8.2 直通线和交叉线

见图 8-2，这是配套 STM32 开发板的接线图，使用的为 MAX3232 芯片，把 STM32 的 PA10 引脚（复用功能为 USART1 的 Rx）接到了 DB9 接口的第 2 针脚，把 PA9 引脚（复用功能为 USART 1 的 Tx）连接到了 DB9 接口的第 3 针脚。

Tx（发送端）接第 3 针脚，Rx（接收端）接第 2 针脚。这种接法是跟 PC 的串口接法一样的，如果要想实现 PC 跟配套板子通信，就要使用两头都是母的交叉线。

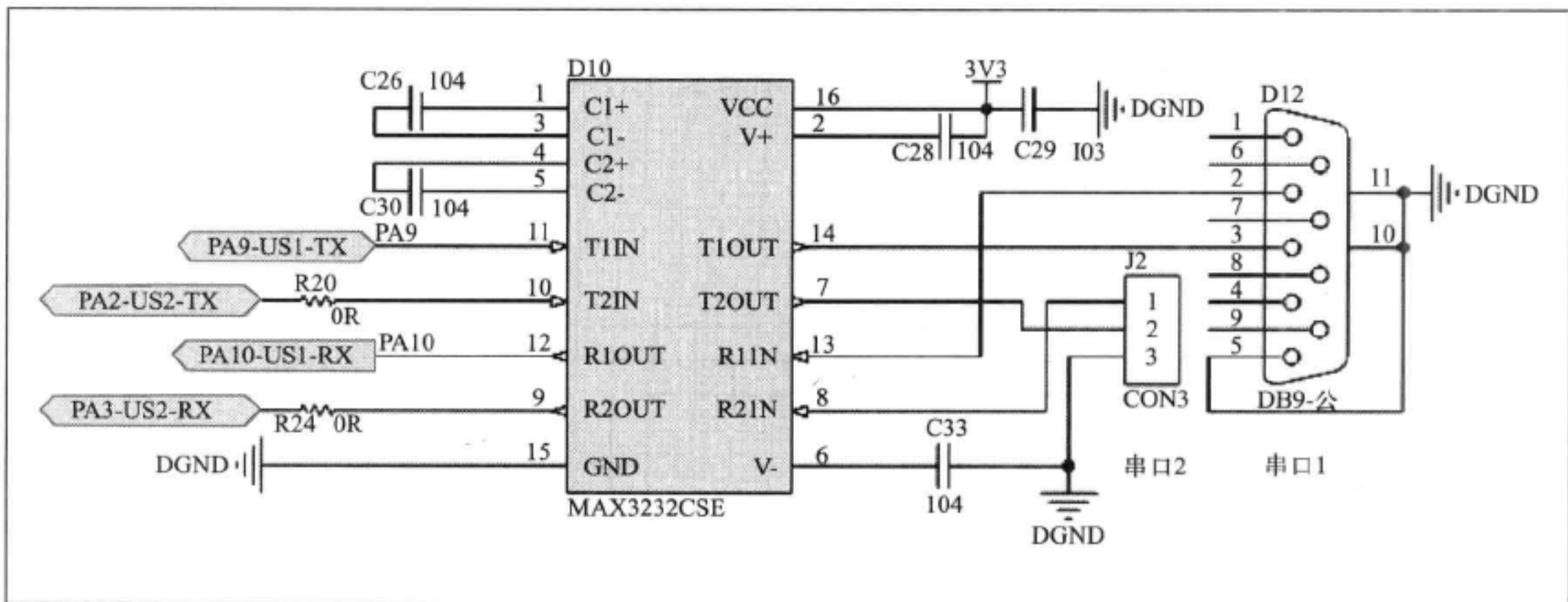


图 8-2 配套开发板串口硬件图

串口线主要分两种：直通线（平行线）和交叉线。它们的区别见图 8-3。假如 PC 与板子之间要实现全双工串口通信，必然是 PC 的 Tx 针脚要连接到板子的 Rx 针脚，而 PC 的 Rx 针脚则要连接至板子的 Tx 针脚。由于板子和 PC 的串口接法是相同的，就要使用交叉线来连接了。如果有的开发板是 Tx 连接至 DB9 的第 2 针脚，而 Rx 连接至第 3 针脚，这与 PC 接法是相反的，这样的板子与 PC 通信就需要使用直通线了。

为什么配套板子要使用 PC 的接法？假如使用非 PC 接法，由于板子与 PC 的接法相反，通信就要使用直通线；但两个板子之间想要进行串口通信时，由于接法相同就要使用交叉线。如果使用 PC 接法，板子与 PC 之间接法相同，通信使用交叉线；两个相同板子之间接法也相同，通信也是使用交叉线。

所以我们建议大家设计板子时，尽量采用与 PC 相同的标准串口接法。

为什么介绍直通线与交叉线的区别？一是我们发现某些读者因为线的问题而花费了大量宝贵时间。二是介绍串口线的 DIY 方法。要实现基本的全双工异步通信，只要 3 根线，分别为 Rx、Tx 和 GND。如果读者正在为直通线、交叉线、公头、母头不匹配而烦恼，可以根据自己的需要，参照图 8-3，用 3 根杜邦线连接即可。

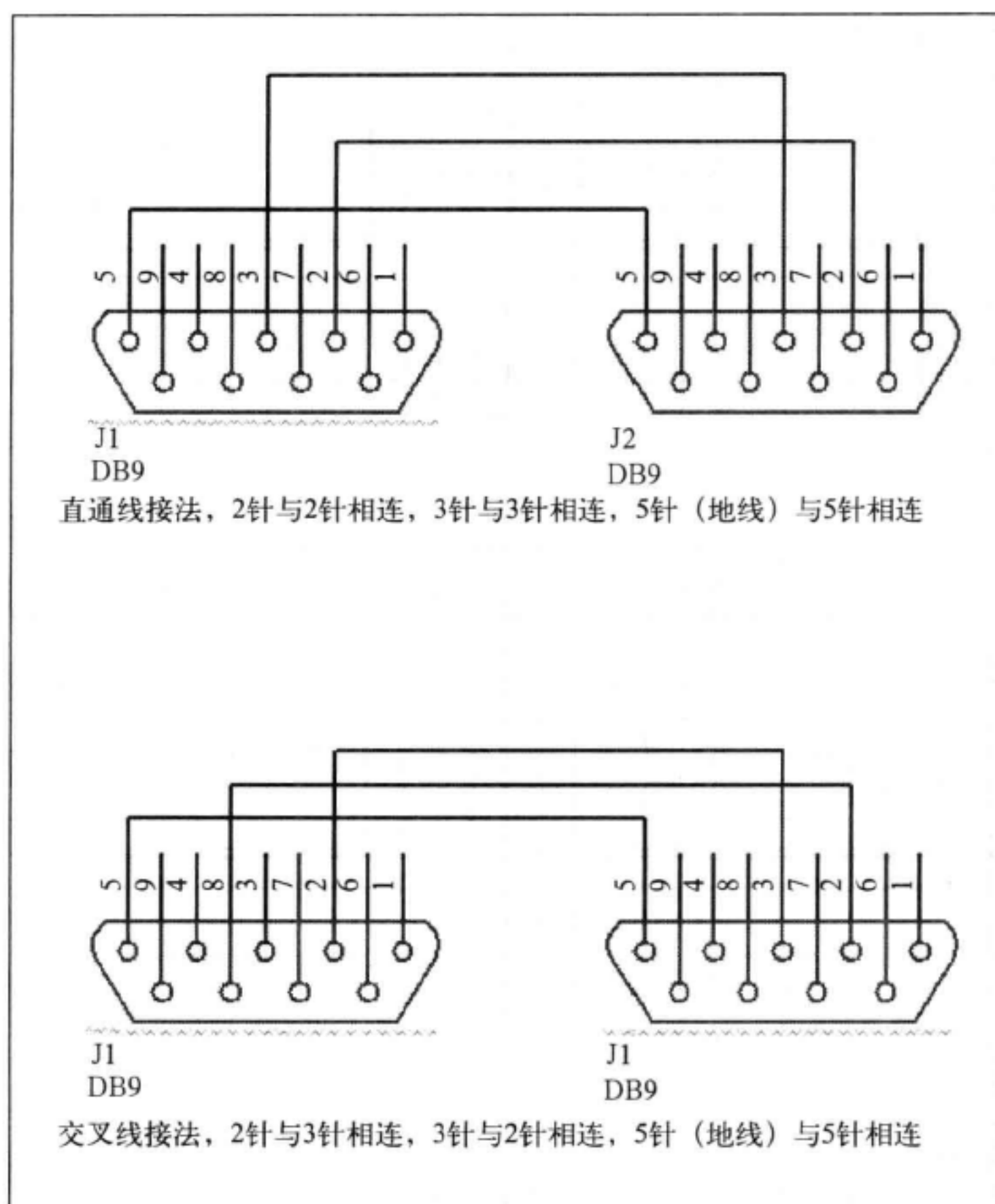


图 8-3 交叉线与直通线的区别

8.3 串口工作过程分析

串口外设的架构图（见图 8-4）看起来十分复杂，实际上对于软件开发人员来说，我们只需要大概了解串口发送的过程即可。

从下至上，我们看到串口外设主要由三个部分组成，分别是波特率控制、收发控制和数据存储转移。

8.3.1 波特率控制

波特率，即每秒传输的二进制位数；用 b/s (bps) 表示，通过对时钟的控制可以改变波特率。在配置波特率时，我们向波特比率寄存器 USART_BRR 写入参数，修改了串口时钟的分频值 USARTDIV。USART_BRR 寄存器包括两部分，分别是 DIV_Mantissa (USARTDIV 的整数部分) 和 DIV_Fraction (USARTDIV 的小数) 部分，最终，计算公式为 $USARTDIV = DIV_Mantissa + (DIV_Fraction / 16)$ 。

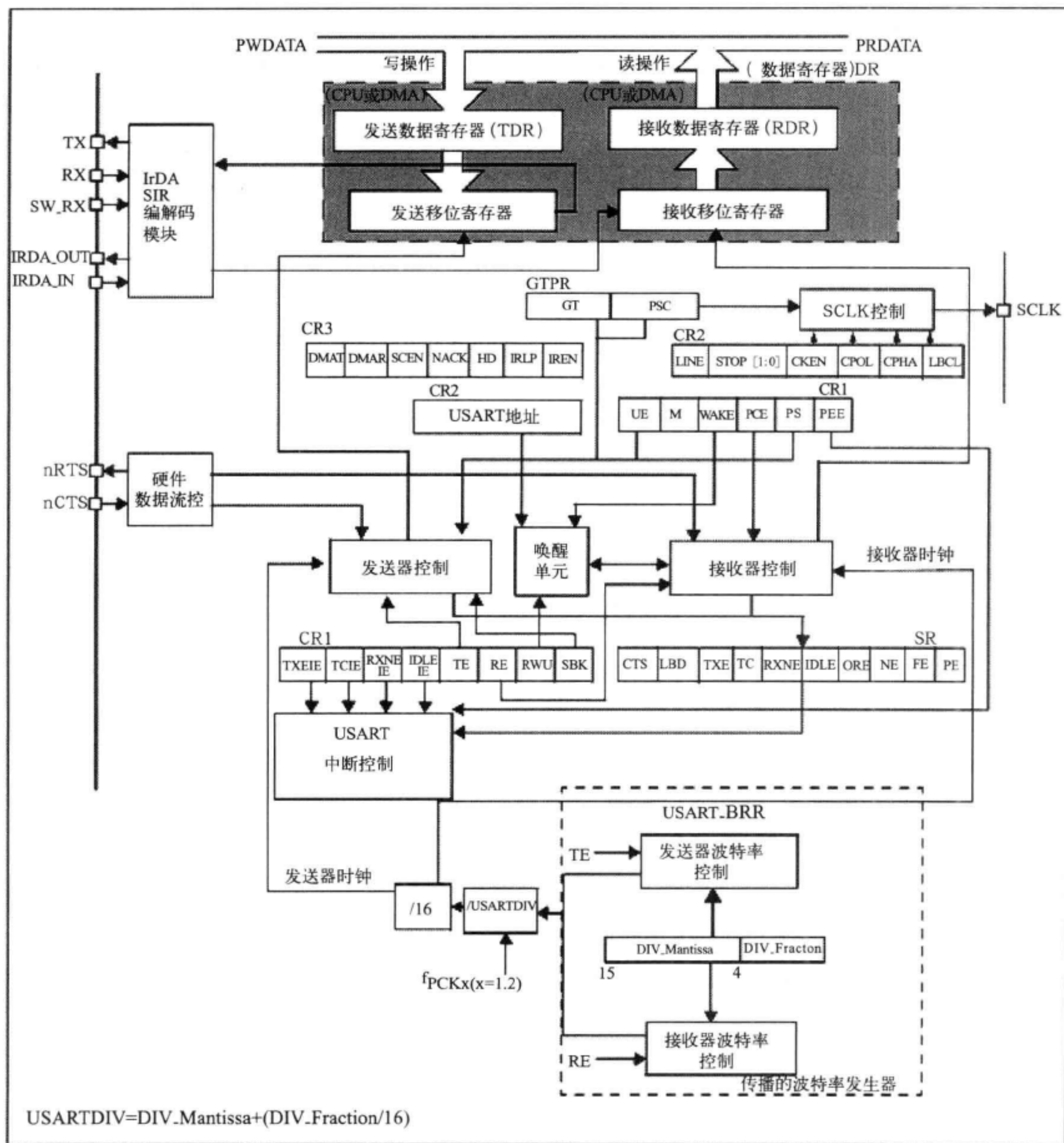


图 8-4 串口架构图

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

8.3.2 收发控制

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

8.3.3 数据存储转移

收发控制器根据我们的寄存器配置，对数据存储转移部分的移位寄存器进行控制。

当我们需要发送数据时，内核或 DMA 外设（一种数据传输方式，在第 10 章介绍）把数据从内存（变量）写入到发送数据寄存器 TDR 后，发送控制器将适时地自动把数据从 TDR 加载到发送移位寄存器，然后通过串口线 Tx，把数据一位一位地发送出去，当数据从 TDR 转移到移位寄存器时，会产生发送寄存器 TDR 已空事件 TXE，当数据从移位寄存器全部发送出去时，会产生数据发送完成事件 TC，这些事件可以在状态寄存器中查询到。

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

8.4 串口通信实验分析

8.4.1 实验描述及工程文件清单

1. 实验描述

重新实现 C 库中的 printf() 函数到串口 1，这样我们就可以像用 C 库中的 printf() 函数一样将信息通过串口打印到计算机，非常方便我们程序的调试。

2. 硬件连接

- ☐ PA9 – USART1 (Tx)
- ☐ PA10 – USART1 (Rx)

3. 库文件

3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/usart1.c

8.4.2 配置工程环境

串口实验中我们用到了 GPIO、RCC、USART 这三个外设的库文件 stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c，所以我们要先把这三个库文件添加进工程，新建用户文件 usart1.c，并在 stm32f10x_conf.h 中把相应头文件的注释去掉。见代码清单 8-1。

代码清单 8-1 串口例程的 stm32f10x_conf.h 文件配置

```

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

```

8.4.3 main 文件

配置好要用的库环境之后，我们就从 main 函数看起，层层剥离源代码。见代码清单 8-2。

代码清单 8-2 串口例程的 main 函数

```

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

```

```

17.
18.     USART1_printf(USART1, "\r\n (\"__DATE__ \" - \" __TIME__ \" ) \r\n");
19.
20.
21.     for(;;)
22.     {
23.
24.     }
25.}

```

首先调用函数 USART1_Config(), 函数 USART1_Config() 主要完成如下工作 :

- 1) 使能串口 1 的时钟。
- 2) 配置 usart1 的 I/O。
- 3) 配置 usart1 的工作模式, 具体为波特率为 115200 、 8 个数据位、 1 个停止位、 无硬件流控制。即 115200 8-N-1。

8.4.4 USART 初始化配置

具体的 USART1_Config() 在 usart1.c 这个用户文件中实现, 见代码清单 8-3。

代码清单 8-3 USART1_Config() 函数

```

1. /*
2.  * 函数名 : USART1_Config
3.  * 描述   : USART1 GPIO 配置 , 工作模式配置。115200 8-N-1
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void USART1_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     USART_InitTypeDef USART_InitStructure;
12.
13.     /* config USART1 clock */
14.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);
15.
16.     /* USART1 GPIO config */
17.
18.     /* Configure USART1 Tx (PA.09) as alternate function push-pull */
19.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
20.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
21.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
22.     GPIO_Init(GPIOA, &GPIO_InitStructure);
23.     /* Configure USART1 Rx (PA.10) as input floating */
24.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
25.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
26.     GPIO_Init(GPIOA, &GPIO_InitStructure);
27.

```

```

28.  /* USART1 mode config */
29.  USART_InitStructure.USART_BaudRate = 115200;
30.  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
31.  USART_InitStructure.USART_StopBits = USART_StopBits_1;
32.  USART_InitStructure.USART_Parity = USART_Parity_No ;
33.  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
34.  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
35.  USART_Init(USART1, &USART_InitStructure);
36.  USART_Cmd(USART1, ENABLE);
37.}

```

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

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

1. GPIO 初始化

在第6章提到，GPIO 具有默认的复用功能，在使用它的复用功能时，我们首先要将相应的 GPIO 进行初始化。此时我们使用的 GPIO 的复用功能为串口，但为什么是 PA9 和 PA10 用作串口的 Tx 和 Rx，而不是其他 GPIO 引脚呢？这是从《STM32F103CDE 增强型系列数据手册》的引脚功能定义中查询到的，见表 8-1。

表 8-1 GPIO 引脚说明（部分）

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCSP64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
D12	C9	D2	42	68	101	PA9	I/O	FT	PA9	USART1_TX/ TIM1_CH2	
D11	D10	D3	43	69	102	PA10	I/O	FT	PA10	USART1_RX/ TIM1_CH3	

选定了这两个引脚，并且 PA9 为 Tx、PA10 为 Rx，那么它们的 GPIO 模式要如何配置呢？Tx 为发送端，输出引脚，而且现在 GPIO 是使用复用功能，所以要把它配置为复用推挽输出模式（`GPIO_Mode_AF_PP`）；而 Rx 引脚为接收端，输入引脚，所以配置为浮空输入模式（`GPIO_Mode_IN_FLOATING`）。如果在使用复用功能时，对 GPIO 的模式不太确定的话，我们可以从《STM32 参考手册》的 GPIO 章节中查询得到，见表 8-2。

表 8-2 GPIO 复用功能模式设置

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

2. USART 初始化

从代码清单 8-3 的第 28 行开始，进行 USART1 的初始化，也就是填充 USART 的初始化结构体。这部分内容是根据串口通信协议来设置的。

1) .USART_BaudRate = 115200：波特率设置，利用库函数我们可以直接这样配置波特率，而不需要自行计算 USARTDIV 的分频因子。在这里把串口的波特率设置为 115200，也可以设置为 9600 等常用的波特率，在《STM32 参考手册》中列举了一些常用的波特率设置及其误差，见表 8-3。如果配置成 9600，那么在与 PC 通信的时候，也应把 PC 的串口传输波特率设置为 9600。通信协议要求两个通信器件之间的波特率、字长、停止位、奇偶校验位都相同。

表 8-3 STM32 常用波特率及其误差

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

2) .USART_WordLength = USART_WordLength_8b：配置串口传输的字长。本例程把它设置为最常用的 8 位字长，也可以设置为 9 位。

3) .USART_StopBits = USART_StopBits_1：配置停止位。把通信协议中的停止位设置为 1 位。

4) .USART_Parity = USART_Parity_No : 配置奇偶校验位。本例程不设置奇偶校验位。

5) .USART_HardwareFlowControl= USART_HardwareFlowControl_None : 配置硬件流控制。本例程不采用硬件流。

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

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

6) .USART_Mode = USART_Mode_Rx | USART_Mode_Tx : 配置串口的模式。为了配置双线全双工通信, 需要把 Rx 和 Tx 模式都开启。

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

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

8.4.5 printf() 函数重定向

在 main 文件中, 配置好串口之后, 就通过几行代码由串口往计算机超级终端打印信息, 打印的信息为一些字符串和当前的日期, 见代码清单 8-4。

代码清单 8-4 通过串口往计算机超级终端打印信息

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

下面是计算机超级终端的截图, 从图 8-5 可以看出程序运行正确。

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

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

重定向的这部分工作, 由 usart.c 文件中的 fputc (int ch, FILE *f) 这个函数来完成, 这个函数具体实现见代码清单 8-5。



图 8-5 串口实验 PC 终端现象

代码清单 8-5 fputc 函数重定向

```
1. /*
2.  * 函数名: fputc
3.  * 描述 : 重定向 c 库函数 printf 到 USART1
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 由 printf 调用
7.  */
8. int fputc(int ch, FILE *f)
9. {
10.     /* 将 Printf 内容发往串口 */
11.     USART_SendData(USART1, (unsigned char) ch);
12.     // while (!(USART1->SR & USART_FLAG_TXE));
13.     while( USART_GetFlagStatus(USART1,USART_FLAG_TC) != SET);
14.     return (ch);
15. }
```

这个代码中调用了两个 ST 库函数。USART_SendData() 和 USART_GetFlagStatus() 的说明见图 8-6 及图 8-7。

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

若使用 C 标准输出库函数，需要在 main.c 文件中把 stdio.h 这个头文件包含进来，还要在编译器中设置一个选项 Use MicroLIB (使用微库)，见图 8-8。这个微库是 Keil MDK 为嵌入式应用量身定做的 C 库，我们要先具有库，才能重定向，勾选使用之后，我们就可以使用 printf() 这个函数了。

void USART_SendData (USART_TypeDef * USARTx,
uint16_t Data
)

通过串口x发送一个数据

Transmits single data through the USARTx peripheral.

Parameters:
USARTx,: Select the USART or the UART peripheral. This parameter can be one of the following values: USART1, USART2, USART3, UART4 or UART5.
Data,: the data to transmit.

Return values:
None

Data参数为将要发送的数据

图 8-6 串口发送函数

FlagStatus USART_GetFlagStatus (USART_TypeDef * USARTx,
uint16_t USART_FLAG
)

检查USART的标志位

Checks whether the specified USART flag is set or not.

Parameters:
USARTx,: Select the USART or the UART peripheral. This parameter can be one of the following values: USART1, USART2, USART3, UART4 or UART5.
USART_FLAG,: specifies the flag to check. This parameter can be one of the following values:

- USART_FLAG_CTS: CTS Change flag (not available for UART4 and UART5)
- USART_FLAG_LBD: LIN Break detection flag
- USART_FLAG_TXE: Transmit data register empty flag
- USART_FLAG_TC: Transmission Complete flag
- USART_FLAG_RXNE: Receive data register not empty flag
- USART_FLAG_IDLE: Idle Line detection flag
- USART_FLAG_ORE: OverRun Error flag
- USART_FLAG_NE: Noise Error flag
- USART_FLAG_FE: Framing Error flag
- USART_FLAG_PE: Parity Error flag

Return values:
The new state of USART_FLAG (SET or RESET).

这些为可输入的标志位参数，如USART_FLAG_TXE表示发送区是否为空的标志位，USART_FLAG_TC表示是否发送完成的标志位

返回标志位检查结果

图 8-7 串口标志位检查函数

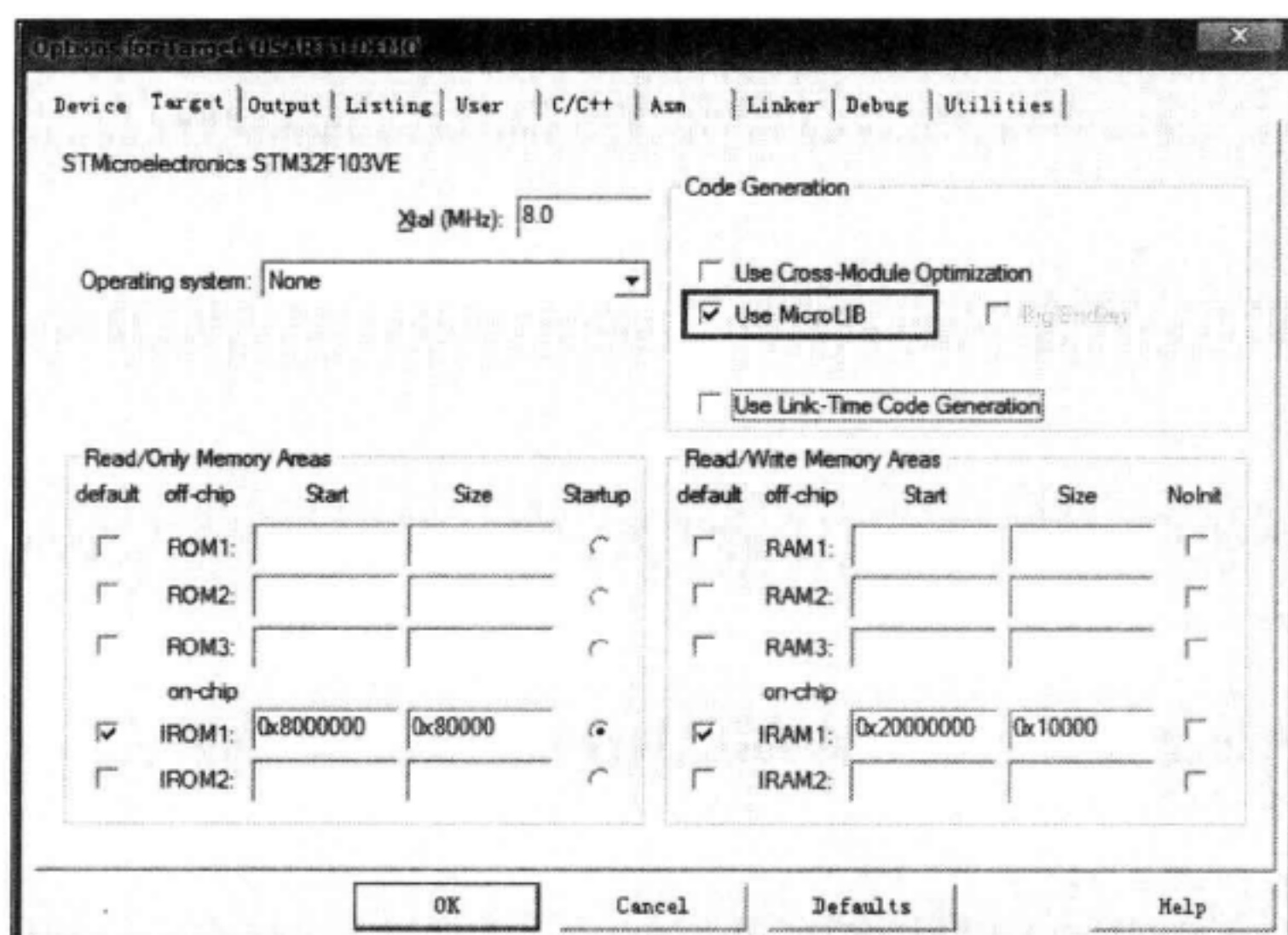


图 8-8 勾选使用 “_Micro LIB”

8.4.6 USART1_printf() 函数

除了重定向的方法，我们还可以自己编写格式输入输出函数。USART1_printf() 便是一个完全自定义的格式输出函数，它的功能与重定向之后的 printf() 类似。

让我们再来看看 USART1_printf (USART_TypeDef* USARTx, uint8_t *Data,...) 这个函数的实现，它调用了 itoa (int value, char *string, int radix) 函数。关于这两个函数的具体实现请看 usart.c 中的源代码。这两个函数中有些变量是定义在 stdarg.h 这个头文件中的，所以在 usart.c 中我们需要把这个头文件包含进来，这个头文件位于 KDE 的根目录下。我们可以在这个路径下找到它：C:\Keil\ARM\RV31\INC，见代码清单 8-6 和代码清单 8-7。

代码清单 8-6 itoa() 函数

```

1. /*
2.  * 函数名: itoa
3.  * 描述   : 将整型数据转换成字符串
4.  * 输入   : -radix =10 表示十进制，其他结果为 0
5.  *         -value 要转换的整型数
6.  *         -buf 转换后的字符串
7.  *         -radix = 10
8.  * 输出   : 无
9.  * 返回   : 无
10. * 调用   : 被 USART1_printf() 调用
11. */
12. static char *itoa(int value, char *string, int radix)
13. {
14.     int    i, d;
15.     int    flag = 0;

```

```

16.     char    *ptr = string;
17.
18.     /* This implementation only works for decimal numbers. */
19.     if (radix != 10)
20.     {
21.         *ptr = 0;
22.         return string;
23.     }
24.
25.     if (!value)
26.     {
27.         *ptr++ = 0x30;
28.         *ptr = 0;
29.         return string;
30.     }
31.
32.     /* if this is a negative value insert the minus sign. */
33.     if (value < 0)
34.     {
35.         *ptr++ = '-';
36.         /* Make the value positive. */
37.         value *= -1;
38.     }
39.     for (i = 10000; i > 0; i /= 10)
40.     {
41.         d = value / i;
42.         if (d || flag)
43.         {
44.             *ptr++ = (char)(d + 0x30);
45.             value -= (d * i);
46.             flag = 1;
47.         }
48.     }
49.
50.     /* Null terminate the string. */
51.     *ptr = 0;
52.     return string;
53. } /* NCL_Itoa */
54.

```

代码清单 8-7 USART1_printf() 函数

```

1.  /*
2.  * 函数名: USART1_printf
3.  * 描述   : 格式化输出, 类似于 C 库中的 printf, 但这里没有用到 C 库
4.  * 输入   : -USARTx 串口通道, 这里只用到了串口 1, 即 USART1
5.  *          -Data   要发送到串口的内容的指针
6.  *          -...     其他参数
7.  * 输出   : 无
8.  * 返回   : 无
9.  * 调用   : 外部调用
10. *        典型应用 USART1_printf( USART1, "\r\n this is a demo \r\n" );
11. *        USART1_printf( USART1, "\r\n %d \r\n", i );
12. *        USART1_printf( USART1, "\r\n %s \r\n", j );
13. */
14. void USART1_printf(USART_TypeDef* USARTx, uint8_t *Data,...)

```

```

15. {
16.   const char *s;
17.   int d;
18.   char buf[16];
19.   va_list ap;
20.   va_start(ap, Data);
21.   while ( *Data != 0)      // 判断是否到达字符串结束符
22.   {
23.       if ( *Data == 0x5c )  //'\'
24.       {
25.           switch ( *++Data )
26.           {
27.               case 'r' :      // 回车符
28.                   USART_SendData(USARTx, 0x0d);
29.                   Data ++;
30.                   break;
31.
32.               case 'n' :      // 换行符
33.                   USART_SendData(USARTx, 0x0a);
34.                   Data ++;
35.                   break;
36.
37.               default:
38.                   Data ++;
39.                   break;
40.           }
41.       }
42.       else if ( *Data == '%' )
43.       {                          //
44.           switch ( *++Data )
45.           {
46.               case 's' :      // 字符串
47.                   s = va_arg(ap, const char *);
48.                   for ( ; *s; s++)
49.                   {
50.                       USART_SendData(USARTx, *s);
51.                       while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
52.                   }
53.                   Data++;
54.                   break;
55.
56.               case 'd':      // 十进制
57.                   d = va_arg(ap, int);
58.                   itoa(d, buf, 10);
59.                   for (s = buf; *s; s++)
60.                   {
61.                       USART_SendData(USARTx, *s);
62.                       while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
63.                   }
64.                   Data++;
65.                   break;
66.               default:
67.                   Data++;
68.                   break;
69.           }
70.       } /* end of else if */
71.       else USART_SendData(USARTx, *Data++);

```



```
72. while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );  
73.     }  
74. }
```

这部分代码有点多，在格式编排上不是很好，推荐大家直接看源代码。

综上所述，我们已经可以用 `printf()` 和 `USART1_printf()` 这两个函数来打印信息了，但到底用哪个比较好呢？其实各有千秋，`printf()` 函数会受缓冲区大小的影响，有时候在用它打印的时候程序会发生莫名其妙的错误，而实际上就是由于使用 `printf()` 这个函数引起的，其优点就是这种情况很少见且支持的格式较多。而 `USART1_printf()` 则不会受缓冲区的影响，但其支持的格式较少。

8.4.7 实验现象

将配套 STM32 开发板供电（DC5V），插上 J_LINK，插上串口线（两头都是母的交叉线，没有的话就 DIY 一个吧），打开超级终端，配置超级终端为“115200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如图 8-9 所示信息。

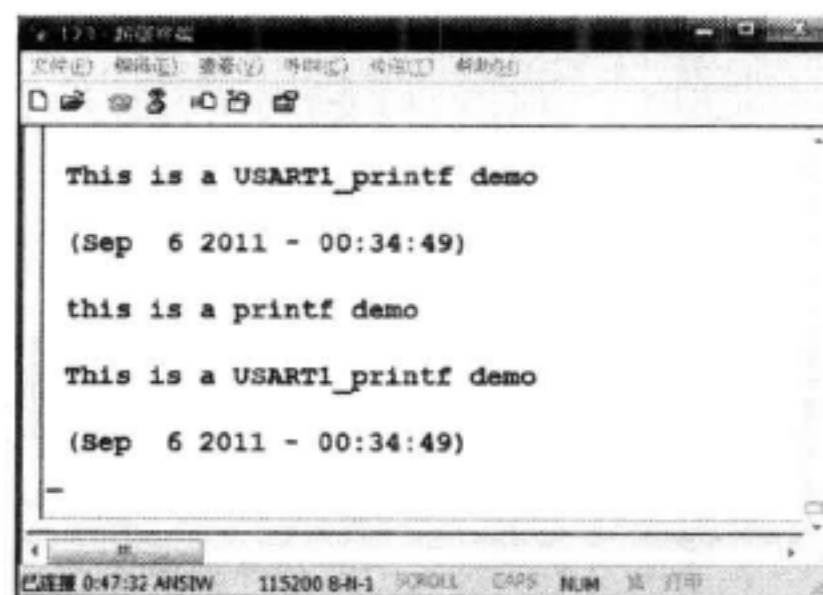


图 8-9 串口实验现象



第 9 章

库函数开发小结

从 LED 中了解了 STM32 的架构，接触了 ST 官方库的库函数，再经过 GPIO 按键轮询实验、中断检测实验、串口通信实验，相信读者已经总结出了一些用 ST 库来开发的步骤和共同点。本章正是为了与读者探讨这些使用 ST 官方库开发不同外设应用时的共同点。

在总结之前，给大家补充一些关于寄存器的概念。基本上所有外设都有以下几类寄存器：

- ❑ 控制寄存器 xxx_CR (Control/Configuration Register)：这类寄存器是用来配置、控制相应外设的工作方式的。如 GPIOx_CRL、GPIOx_CRH、AFIO_EXTICR1 ~ AFIO_EXTICR4，串口的 USART_CR1 ~ USART_CR3 等。
- ❑ 数据寄存器 xxx_DR (Data Register)：这类寄存器主要是存储了外设进行输出输入的数据。如 GPIOx_IDR、GPIOx_ODR、USART_DR 等。
- ❑ 状态寄存器 xxx_SR (Status Register)：这类寄存器主要存储了当前外设的运行状态，主要为一些标志位。如 USART_SR、ADC_SR 等。

9.1 初始化

回顾一下在前面几个例程中，使用过什么类型的初始化结构体？

我们总结如下：

- ❑ GPIO_InitTypeDef 型的 GPIO_InitStructure 用来配置 GPIO。
- ❑ NVIC_InitTypeDef 型的 NVIC_InitStructure 用来配置 NVIC。
- ❑ EXTI_InitTypeDef 型的 EXTI_InitStructure 用来配置 EXTI。
- ❑ USART_InitTypeDef 型的 USART_InitStructure 用来配置 USART。

这些初始化结构体的控制参数，一般就是与相应外设的控制寄存器 xxx_CR 对应的，在使用库开发的时候，有时我们查阅库帮助文档也搞不清楚该参数的作用时，可以直接到《STM32 参考手册》相应的外设控制寄存器 xxx_CR 的位说明中查找其参数意义。

如表 9-1 是 USART_CR1 的某些位说明，这是对到 USART_InitTypeDef 型结构体的 .USART_Mode 成员的，语句 .USART_Mode = USART_Mode_Rx | USART_Mode_Tx 表示把这两个寄存器都置 1，使能发送和接收。

表 9-1 USART_CR1 寄存器说明（部分）

位数	描 述
位 4	TE：发送使能（Transmitter Enable），该位使能发送器。该位由软件设置或清除。0：禁止发送；1：使能发送。注意：①在数据传输过程中，除了在智能卡模式下，如果 TE 位上有个 0 脉冲（即设置为“0”之后再设置为“1”），会在当前数据字传输完成后，发送一个“前导符”（空闲总线）。②当 TE 被设置后，在真正发送开始之前，有一个比特时间的延迟
位 3	RE：接收使能（Receiver Enable），该位由软件设置或清除。0：禁止接收；1：使能接收，并开始搜寻 RX 引脚上的起始位

使用 ST 库对外设进行初始化，一般有以下步骤：

- 1) 定义一个 xxx_InitTypeDef 类型的初始化结构体。
- 2) 根据使用需求，向这些初始化结构体的成员写入特定的控制参数。
- 3) 填充好结构体之后，把这个结构体作为输入参数调用相应的外设库函数 xxx_Init()，从而实现向寄存器写入控制参数，并配置好外设。

在以后其他外设开发中，我们还会遇到各种类型的初始化结构体及初始化函数。如 ADC_InitTypeDef，ADC_Init()；I2C_InitTypeDef，I2C_Init() 等。它们的应用方法都是相同的，区别在于不同的外设其结构体成员不一样，可输入参数相应也不同。只要理解了这些结构体成员所控制参数的意义，我们就能够轻松地使用一个全新的外设。

9.2 数据输入输出

对外设的使用，一般涉及其输入和输出数据，ST 官方库中有一类函数专门为此应用而生。如 GPIO 的输入输出函数：GPIO_ReadOutputDataBit()、GPIO_ReadInputData()、GPIO_SetBits()；还有 USART 的收发数据函数：USART_ReceiveData()、USART_SendData()，这类函数都是用于控制输入输出数据的。

这些函数控制相应外设数据寄存器 DR 的内容，达到控制输入输出的目的。使用这些函数的方法也是类似的。

- 1) 通过输入参数，向函数指定要使用的是什​​么外设，如用（GPIOA，GPIO_Pin_5）选定 PA5 进行控制，用（USART1）来指定使用串口 1 外设。
- 2) 若向外输出数据，则调用 Output 或 Send 函数，把将要输出的数据变量作为函数的输入参数。
- 3) 若为接收外部数据，则调用 Read 或 Receive 函数，读取函数的返回值来得到外部输入数据。

对于其他外设，也有类似的控制数据输入输出函数。如用 ADC_GetConversionValue() 函数来获取 ADC 转换所得到的数值；用 I2C_SendData() 函数来使用 I²C 接口进行发送数据。

9.3 状态位、标志位

当我们需要知道外设的工作状态时，就涉及一系列标志检查的 ST 官方库函数。

9.3.1 事件

当外设完成了某些工作或出现某些状态的时候，会触发一些事件，这些事件会在状态寄存器 SR 中，以不同的寄存器位来记录。这些寄存器位称为相应的事件标志位。

如串口发送完成后，会在 USART_SR 寄存器中的位 6 置 1，见表 9-2，作为发送完成的事件标志。若发送寄存器为空，则会相应地在位 7 置 1，作为发送寄存器已空的事件标志，如果我们不停地查询这个标志位，就可以得知串口的发送状态。

不停地查询标志位，会耗费内核宝贵的资源，ST 以中断的方式解决这个问题，大部分事件都可以被配置成中断。例如，若把串口发送完成事件配置为可触发中断后，当串口发送完成时，外设不仅在 USART_SR 寄存器中记录事件，还会触发串口中断，从而可以进入相应的中断服务函数，针对不同的事件进行具体的处理，而内核也省去了不停查询标志位的工作。

表 9-2 USART_SR 寄存器（部分）

位数	描 述
位 8	LBD：LIN 断开检测标志（LIN break detection flag），当检测到 LIN 断开时，该位由硬件置“1”，由软件清“0”（向该位写 0）。如果 USART_CR3 中的 LBDIE = 1，则产生中断。0：没有检测到 LIN 断开；1：检测到 LIN 断开。注意：若 LBDIE=1，当 LBD 为“1”时要产生中断
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，只有在多缓存通信中才推荐这种清除程序。0：发送还未完成；1：发送完成

9.3.2 标志位的检查与清除

假如我们把串口的发送完成事件、接收寄存器非空事件（串口接收到数据）都配置为可触发中断，因为它们触发的都是串口中断，所以中断时都是进入到同一个串口中断服务函数中处理的。那么我们在串口的中断服务函数之中，就要区分这个中断究竟是由发送完成事件触发的，还是由接收到数据事件触发的。

在这个时候，我们就必须进行一次标志位检查了。对标志位进行检查的库函数，一般命名为 xxx_GetFlagStatus() 或 xxx_GetITStatus()，功能分别为获取事件标志位状态和中断标志位状态。如前面使用过的读取串口标志位的函数 USART_GetFlagStatus()；EXTI 的获取 EXTI 线状态的函数 EXTI_GetFlagStatus()。

既然有标志位检查，自然也有清除标志位功能的函数。对标志位进行清除的 ST 库函数，一般命名为 xxx_ClearFlag() 或 xxx_ClearITPendingBit() 函数，功能分别为清除事件标志位和清除挂起的中断标志位。如串口的 USART_ClearFlag()，EXTI 的 USART_ClearITPendingBit()。

这四类函数都是对应到外设的 xxx_SR 寄存器的。

那么对事件标志操作的函数和对中断标志操作的函数有什么区别呢？实质上它们检查、清除的都是 SR 寄存器的标志，只是在检查中断标志位的时候，中断标志检查函数（xxx_GetITStatus）还会检查对应事件的中断屏蔽位是否开启（这位是由开发者配置的），如果事件中断被屏蔽（事件发生时不触发中断），那么在事件 A 发生的时候，用 xxx_GetFlag() 函数时检查结果返回事件 A 发生，但用 xxx_GetITStatus() 函数时检查结果是没有触发事件 A 的中断。

而清除标志位时，xxx_ClearFlag() 和 xxx_ClearITPendingBit() 结果都是对 xxx_SR 寄存器进行清除的，但在使用的时候，还是尽量在中断时使用 xxx_ClearITPendingBit()，在非中断时采用 xxx_ClearFlag() 要严谨一点。

这些对标志位进行操作的函数都有统一的方法。

输入参数就是要检查的标志，如串口的发送完成标志 USART_FLAG_TC、接收寄存器非空标志 USART_FLAG_RXNE。检查函数具有返回值，返回值是 SET 或 RESET，表示这个标志位被置位或没有被置位。而标志位清除函数就没有返回值了，调用函数后就直接把相应的标志位进行清除。

9.4 外设函数分类

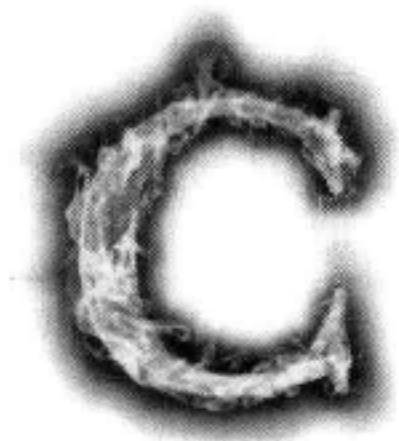
其实 ST 官方库的库函数还有很多共同点，如外设都有 xxx_Cmd() 函数，用来使能外设；如 xxx_ITConfig() 函数用来配置外设的事件触发中断。在此，对这些外设主要的共有函数进行了简单分类，一般外设都具有以下的函数，见表 9-3。

表 9-3 共同函数分类表

函 数 名	功 能	输 入 参 数	返 回 值	具体函数举例
XXX_Init()	对外设进行初始化	XXX_InitTypeDef 初始化类型结构体	void	GPIO_Init() USART_Init()
XXX_DeInit	以系统默认的形式对外设进行初始化	将要进行默认初始化的外设名	void	USART_DeInit() I2C_DeInit()
XXX_StructInit()	以默认数据填充初始化结构体	将要进行默认填充的 XXX_InitTypeDef 初始化类型结构体	Void	GPIO_StructInit()
XXX_SendData()	使用外设发送数据	XXX（相应的外设名）； 将要发送的数据	void	USART_SendData() I2C_SendData()
XXX_ReceiveData()	获取外设接收到的数据	XXX（相应的外设名）	返回接收到的数据	USART_ReceiveData() I2C_ReceiveData()
XXX_GetFlagStatus()	检查外设事件标志位	要检查的事件标志名	返回标志位状态 (SET 或 RESET)	USART_GetFlagStatus() SDIO_GetFlagStatus()
XXX_GetITStatus()	检查中断标志	要检查的中断标志名（大部分与事件标志相同）	返回标志位状态 (SET 或 RESET)	USART_GetITStatus() I2C_GetITStatus()

(续)

函 数 名	功 能	输 入 参 数	返 回 值	具体函数举例
XXX_ClearFlag()	清除事件标志位	要清除的事件标志名	void	USART_ClearFlag() RTC_ClearFlag()
XXX_ClearITPendingBit()	清除挂起的中断标志位	要检查的中断标志名（大部分与事件标志相同）	void	USART_ClearITPendingBit() RTC_ClearITPendingBit()
XXX_ITConfig()	设置外设的中断	XXX（相应外设的名字）；选择要开启外设的某种中断（如接收中断、发送完成中断）；ENABLE 或 DISABLE 中断	void	USART_ITConfig() ADC_ITConfig()
XXX_Cmd()	使能或关闭外设	要配置的外设名；ENABLE 或 DISABLE	void	USART_Cmd() TIM_Cmd()
XXX_DMAMCmd()	配置外是否可使用 DMA 请求	要配置的外设名；选择要配置的 DMA 请求（如 DMA 接收请求、DMA 发送请求）；ENABLE 或 DISABLE	void	USART_DMAMCmd() SDIO_DMAMCmd()



第 10 章

DMA——为 CPU 减负

10.1 DMA 功能简介

DMA (Direct Memory Access, 直接存储器存取), 是一种可以大大减轻 CPU 工作量的数据存取方式, 因而被广泛地使用。早在 8086 的应用中就已经有 Intel 的 8237 这种典型的 DMA 控制器, 而 STM32 的 DMA 则是以类似外设的形式添加到 Cortex 内核之外的。

在硬件系统中, 主要由 CPU (内核)、外设、内存 (SRAM)、总线等结构组成, 数据经常要在内存与外设之间转移, 或从外设 A 转移到外设 B。

例如: 当 CPU 需要处理由 ADC 外设采集回来的数据时, CPU 首先要把数据从 ADC 外设的寄存器读取到内存中 (变量), 然后进行运算处理, 这是一般的处理方法。

在转移数据的过程中会占用 CPU 十分宝贵的资源, 所以我们希望 CPU 更多地被用在数据运算或响应中断之中, 而数据转移的工作交由其他部件完成。DMA 正是为 CPU 分担了数据转移的工作。因为 DMA 的存在 CPU 才被解放出来, 它可以在 DMA 转移数据的过程中同时进行数据运算、响应中断, 大大提高效率。

10.2 DMA 工作分析

见图 10-1, 在这个图中我们可以清晰地看到 STM32 内核、存储器、外设及 DMA 的连接。

所有这些硬件结构最终都通过各种各样的线连接到总线矩阵之中, 硬件结构之间的数据转移都经过总线矩阵的协调, 使各个外设都能够和谐地使用总线来传输数据。

例如: 在不使用 DMA 的情况下, 内核通过 DCode 经过总线矩阵协调, 使用 AHB 把外设 ADC 采集的数据读取到内核, 然后内核 DCode 再通过总线矩阵协调, 把数据存放到内存 SRAM 中。

DMA 正好可以取代这样的工作。由 DMA 控制器的 DMA 总线与总线矩阵协调, 使用 AHB 把外设 ADC 的数据经由 DMA 通道存放到内存 SRAM。在这个数据传输的过程中, 不需要内核的全程参与, 所以内核可以同时进行数据运算。而且, DMA 方式是点到点的数据转移, 而不使用 DMA 方式还要以内核来作为中转站, 显然 DMA 传输方式的效率更高, 直接存储器存取中的“直接”不是徒有虚名的。

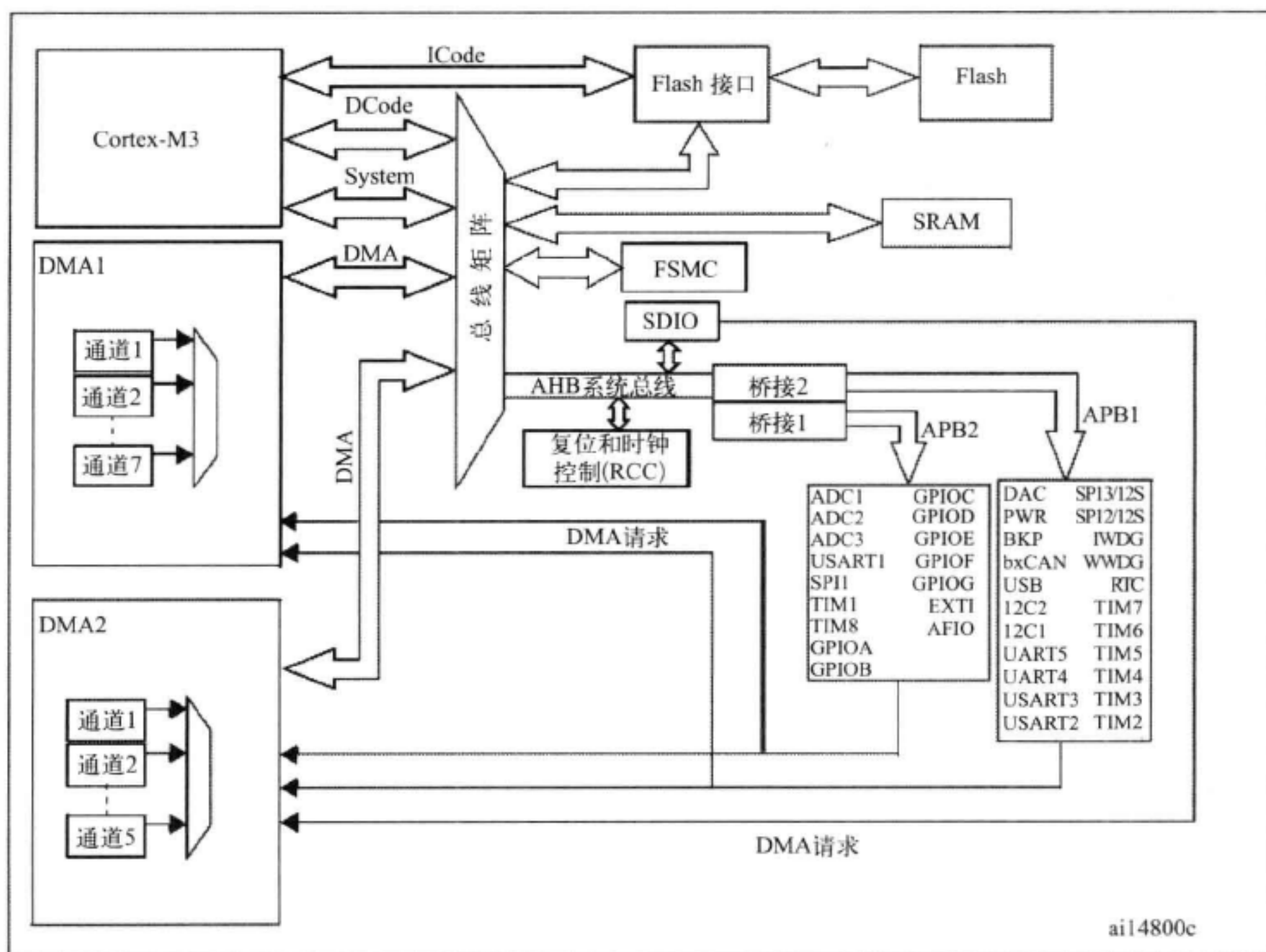


图 10-1 STM32 微控制器的系统结构

要使用 DMA，需要确定一系列的控制参数，如外设数据的地址、内存地址、传输方向等，在开启 DMA 传输前还要先发出 DMA 请求。

10.3 DMA 实例之串口通信

本节通过一个实例对 STM32 的 DMA 进行讲解，以 DMA 方式使用串口发送数据。实际上是利用 DMA 把数据（数组）从内存转移到外设（串口）。我们知道外设工作的时候，除了转移数据，实质上是不需要内核干预的，而数据转移的工作现在交给了 DMA，所以在串口发送数据的时候，内核同时还可以进行其他操作，如点亮 LED 灯。

10.3.1 实验描述及工程文件清单

1. 实验描述

利用 DMA 方式, 使用串口发送数据, 并利用 LED 进行检验在使用 DMA 传输的过程中, 内核是否可以同时进行其他操作 (点亮 LED 灯)。

2. 硬件连接

- PA9 – USART1 (Tx)

☐ PA10 – USART1 (Rx)

3. 库文件

3.5 版本固件库：

☐ startup/start_stm32f10x_hd.c

☐ CMSIS/core_cm3.c

☐ CMSIS/system_stm32f10x.c

☐ FWlib/stm32f10x_gpio.c

☐ FWlib/stm32f10x_rcc.c

☐ FWlib/stm32f10x_usart.c

☐ FWlib/stm32f10x_dma.c

☐ FWlib/misc.c

4. 用户文件

☐ USER/main.c

☐ USER/stm32f10x_it.c

☐ USER/usart1.c

10.3.2 配置工程环境

本串口 DMA 实验中我们用到了 GPIO、RCC、USART、DMA 外设及中断，所以我们先要把以下库文件添加到工程：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_dma.c、misc.c。本实验主要在 usart1.c 文件中添加 DMA 配置的代码，添加旧工程中的外设用户文件 led.c、usart1.c，并在 stm32f10x_conf.h 中把相应头文件的注释去掉。见代码清单 10-1。

代码清单 10-1 DMA 例程的 stm32f10x_conf.h 文件配置

```

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

```

10.3.3 main 文件

依然从 main 函数，按照代码的执行流程进行分析，见代码清单 10-2。

代码清单 10-2 DMA 例程的 main 函数

```

1. #include "stm32f10x.h"
2. #include "usart1.h"
3. #include "led.h"
4.
5. extern uint8_t SendBuff[SENDBUFF_SIZE];
6. uint16_t i;
7.
8.
9. /*
10. * 函数名: main
11. * 描述 : 主函数
12. * 输入 : 无
13. * 输出 : 无
14. */
15. int main(void)
16. {
17.     /* USART1 config 115200 8-N-1 */
18.     USART1_Config();
19.     DMA_Config();
20.     LED_GPIO_Config();
21.
22.
23.     /* 填充将要发送的数据 */
24.     for(i=0; i<SENDBUFF_SIZE; i++)
25.     {
26.         SendBuff[i] = 0xff;
27.     }
28.
29.     /* 串口向 DMA 发出请求 */
30.     USART_DMACmd(USART1, USART_DMAREq_Tx, ENABLE);
31.
32.     /* 在 DMA 尚未传送完成时, CPU 继续执行 main 函数中的代码 */
33.     /* 点亮了 LED 灯 */
34.     /* 而同时 DMA 在向串口运送数据, 当 DMA 发送完成时, 在中断函数关闭 LED 灯 */
35.
36.
37.     LED1(ON);
38.
39.
40.     while(1);
41. }

```

从 main 函数来看, 这个工程十分简单, 首先调用了用户函数 USART1_Config()、DMA_Config() 及 LED_GPIO_Config(), 分别配置好串口、DMA 及 LED 外设。其中 USART1_Config() 和 LED_GPIO_Config() 函数与前面的工程是完全相同的, 没有丝毫改动。

10.3.4 DMA 初始化

接下来我们开始分析本章的重点: DMA 的初始化配置。DMA 的初始化是在 DMA_Config() 函数中实现的, 其代码位于 usart1.c 文件, 见代码清单 10-3。

代码清单 10-3 DMA_Config() 函数

```

1.  /*
2.  * 函数名: DMA_Config
3.  * 描述   : DMA 串口的初始化配置
4.  * 输入    : 无
5.  * 输出    : 无
6.  * 调用    : 外部调用
7.  */
8.  void DMA_Config(void)
9.  {
10.     DMA_InitTypeDef DMA_InitStructure;
11.
12.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
13.     // 开启 DMA 时钟
14.
15.     NVIC_Config();           // 配置 DMA 中断
16.
17.     /* 设置 DMA 源: 内存地址 & 串口数据寄存器地址 */
18.     DMA_InitStructure.DMA_PeripheralBaseAddr = USART1_DR_Base;
19.
20.     /* 内存地址 (要传输的变量的指针) */
21.     DMA_InitStructure.DMA_MemoryBaseAddr = (u32)SendBuff;
22.
23.     /* 方向: 从内存到外设 */
24.     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;
25.
26.     /* 传输大小 DMA_BufferSize=SENDERBUFF_SIZE */
27.     DMA_InitStructure.DMA_BufferSize = SENDERBUFF_SIZE;
28.
29.     /* 外设地址不增 */
30.     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
31.     /* 内存地址自增 */
32.     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
33.
34.     /* 外设数据单位 */
35.     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
36.
37.     /* 内存数据单位 8bit */
38.     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
39.     /* DMA 模式: 一次传输, 循环 */
40.     DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
41.
42.     /* 优先级: 中 */
43.     DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;
44.
45.     /* 禁止内存到内存的传输 */
46.     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
47.
48.     /* 配置 DMA1 的 4 通道 */
49.     DMA_Init(DMA1_Channel4, &DMA_InitStructure);
50.
51.     DMA_Cmd(DMA1_Channel4, ENABLE);           // 使能 DMA
52.     DMA_ITConfig(DMA1_Channel4, DMA_IT_TC, ENABLE); // 配置 DMA 发送完成后产生中断
53.
54. }

```

初始化外设最主要的任务是什么？开启外设时钟、填充初始化结构体以及使能外设。这个函数正是完成了这三个任务。

下面重点分析如何填充 DMA 的初始化结构体，先跳过 NVIC_Config() 函数直接看给结构体成员赋值部分。

1) .DMA_PeripheralBaseAddr：这个成员保存的是外设数据寄存器的基地址，这个地址作为传输的源或目标。为什么说是“基”地址？因为 DMA 具有地址自增的功能，地址自增功使得可以方便地读取连续的数据单元。现在给这个成员的赋值为 USART1_DR_Base，是一个自定义的宏，宏展开后它是 USART1 的数据寄存器地址：

```
1. #define USART1_DR_Base 0x40013804
```

涉及的 USART 数据寄存器及其地址，见表 10-1。

表 10-1 USART 数据寄存器 (USART_DR)

数据寄存器 (USART_DR)															
地址偏移：0x04															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留							DR[8:0]								
位数		描述													
31:9		保留位，硬件强制为 0													
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，一个发送用 (TDR)，一个接收用 (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。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。当使能校验位 (USART_CR1 中 PCE 位被置位) 进行发送时，写到 MSB 的值 (根据数据的长度不同，MSB 是第 7 位或者第 8 位) 会被后来的校验位取代。当使能校验位进行接收时，读到的 MSB 位是接收到的校验位													

从《STM32 参考手册》可知，串口外设会自动把数据寄存器中的数据，送入它的移位寄存器，然后由硬件按照串口协议把该数据发送出去。

在本代码中我们把这个数据寄存器的地址作为外设地址，那么由 DMA 通道转移过来的内存数据就会被保存到这个寄存器中，然后串口就会自动进行发送了。

那么这个寄存器的地址又是怎么算出来的呢？请注意表 10-1 中的地址偏移为 0x04，前面的章节已经介绍过，这些寄存器的地址偏移是相对于它所在外设的基地址进行偏移的。本代码中使用的为外设串口 1，所以我们在《STM32 参考手册》的存储器映像中找到 USART1 (串口 1) 的外设基地址为 0x4001 3800，见表 10-2，串口 1 的外设基地址加上数据寄存器的地址偏移就是我们在 DMA 传输中需要的目标地址了：

$$0x4001\ 3804 = 0x4001\ 3800 + 0x04$$

表 10-2 存储器映射（部分）

起 止 地 址	外 设	总 线
0x5000 0000 – 0x5003 FFFF	USB OTG 全速	AHB
0x4003 0000 – 0x4FFF FFFF	保留	
0x4002 8000 – 0x4002 9FFF	以太网	
0x4002 3400 - 0x4002 3FFF	保留	AHB
0x4002 3000 - 0x4002 33FF	CRC	
0x4002 2000 - 0x4002 23FF	闪存存储器接口	
0x4002 1400 - 0x4002 1FFF	保留	
0x4002 1000 - 0x4002 13FF	复位和时钟控制（RCC）	
0x4002 0800 - 0x4002 0FFF	保留	
0x4002 0400 - 0x4002 07FF	DMA2	
0x4002 0000 - 0x4002 03FF	DMA1	
0x4001 8400 - 0x4001 7FFF	保留	
0x4001 8000 - 0x4001 83FF	SDIO	
0x4001 4000 - 0x4001 7FFF	保留	APB2
0x4001 3C00 - 0x4001 3FFF	ADC3	
0x4001 3800 - 0x4001 3BFF	USART1	
0x4001 3400 - 0x4001 37FF	TIM8 定时器	
0x4001 3000 - 0x4001 33FF	SPI1	
0x4001 2C00 - 0x4001 2FFF	TIM1 定时器	
0x4001 2800 - 0x4001 2BFF	ADC2	
0x4001 2400 - 0x4001 27FF	ADC1	
0x4001 2000 - 0x4001 23FF	GPIO 端口 G	
0x4001 2000 - 0x4001 23FF	GPIO 端口 F	
0x4001 1800 - 0x4001 1BFF	GPIO 端口 E	
0x4001 1400 - 0x4001 17FF	GPIO 端口 D	
0x4001 1000 - 0x4001 13FF	GPIO 端口 C	
0X4001 0C00 - 0x4001 0FFF	GPIO 端口 B	
0x4001 0800 - 0x4001 0BFF	GPIO 端口 A	
0x4001 0400 - 0x4001 07FF	EXTI	
0x4001 0000 - 0x4001 03FF	AFIO	

2) .DMA_MemoryBaseAddr：保存了内存的基地址，同样，这个地址可作为传输的源或目标。在使用时通常会给这个成员赋值为某个数组的基地址，然后利用 DMA 的地址自增功能把数

组一个个地填满。在本代码中向这个成员赋值为 SendBuff，这是一个自定义的数组变量名，它具有 5000 个数组元素：

```
1. #define SENDBUFF_SIZE 5000
2. uint8_t SendBuff[SENDBUFF_SIZE];
```

我们知道在 C 语言中数组名就是该数组的基地址，而数组（变量）是被保存到内存（SRAM）上的，所以我们实质上给 .DMA_MemoryBaseAddr 这个结构体成员赋予了一个内存地址。在本代码中，这个内存地址作为 DMA 传输的源地址。

3) .DMA_DIR：保存了 DMA 数据传输方向，可以选择是外设到内存还是内存到外设。本代码中该成员值为 DMA_DIR_PeripheralDST（外设作为目标地址），也就是说，这个成员的值决定了本次数据传输是从内存转移到外设。

4) .DMA_BufferSize：保存了 DMA 要传输的数据总大小，其单位为后面结构体成员 .DMA_PeripheralDataSize 中的参数。本代码对它赋值为自定义的宏 SENDBUFF_SIZE，宏展开为 5000。即本次 DMA 要传输 5000 个数据。

5) .DMA_PeripheralInc 和 .DMA_MemoryInc：分别为外设和内存的地址是否开启自增功能。本代码向这两个成员赋值为外设地址固定而内存地址自增，所以 DMA 在传输过程中，数组的元素 0 ~ 4999 被一个一个地转移到同样的串口数据寄存器地址。

6) .DMA_PeripheralDataSize 和 .DMA_MemoryDataSize：分别为外设和内存的数据单元大小。可以为字节、半字和字。本代码设置为字节大小，即每个传输数据的大小为 1 字节，加上前面对 .DMA_BufferSize 成员的配置，本次 DMA 传输的数据总大小即为 5000 字节。

7) .DMA_Mode：保存了 DMA 的模式，可以为循环模式或正常模式，循环模式即在传输完一轮数据之后再重新传输，这种方式很适合 ADC 不断采集数据的场合。本代码只做一次 DMA 传输，采用 DMA_Mode_Normal（正常模式）。

8) .DMA_Priority：DMA 通道的优先级，总线矩阵根据其 DMA 通道的优先级进行总线协调分配。本代码配置为 DMA_Priority_Medium（中等优先级），其实当只使用一个 DMA 通道时，配置为任何优先级都没有区别。

9) .DMA_M2M：保存了是否内存到内存的 DMA 传输。DMA 传输可以在外设与内存、外设与外设以及内存与内存之间进行。本代码是内存到外设的传输，所以要把本成员赋值为 DMA_M2M_Disable（禁止内存到内存的传输）。

填充完结构体后，就要调用其初始化函数及使能 DMA 外设，见代码清单 10-4。

代码清单 10-4 调用初始化函数和使能 DMA

```
1. DMA_Init(DMA1_Channel4, &DMA_InitStructure);
2.
3. DMA_Cmd (DMA1_Channel4, ENABLE);
```

在初始化的时候要选择恰当的 DMA 通道，本次函数使用的是 DMA1_Channel4 即 DMA1 的通道 4，这个通道并不是随便选择的，而是要根据 DMA 的请求映像来设置。见图 10-2。DMA 请求是指外设在使用 DMA 之前需要向 DMA 控制器发送请求信息，DMA 在接收到请求后才会

根据 DMA 配置进行数据转移。

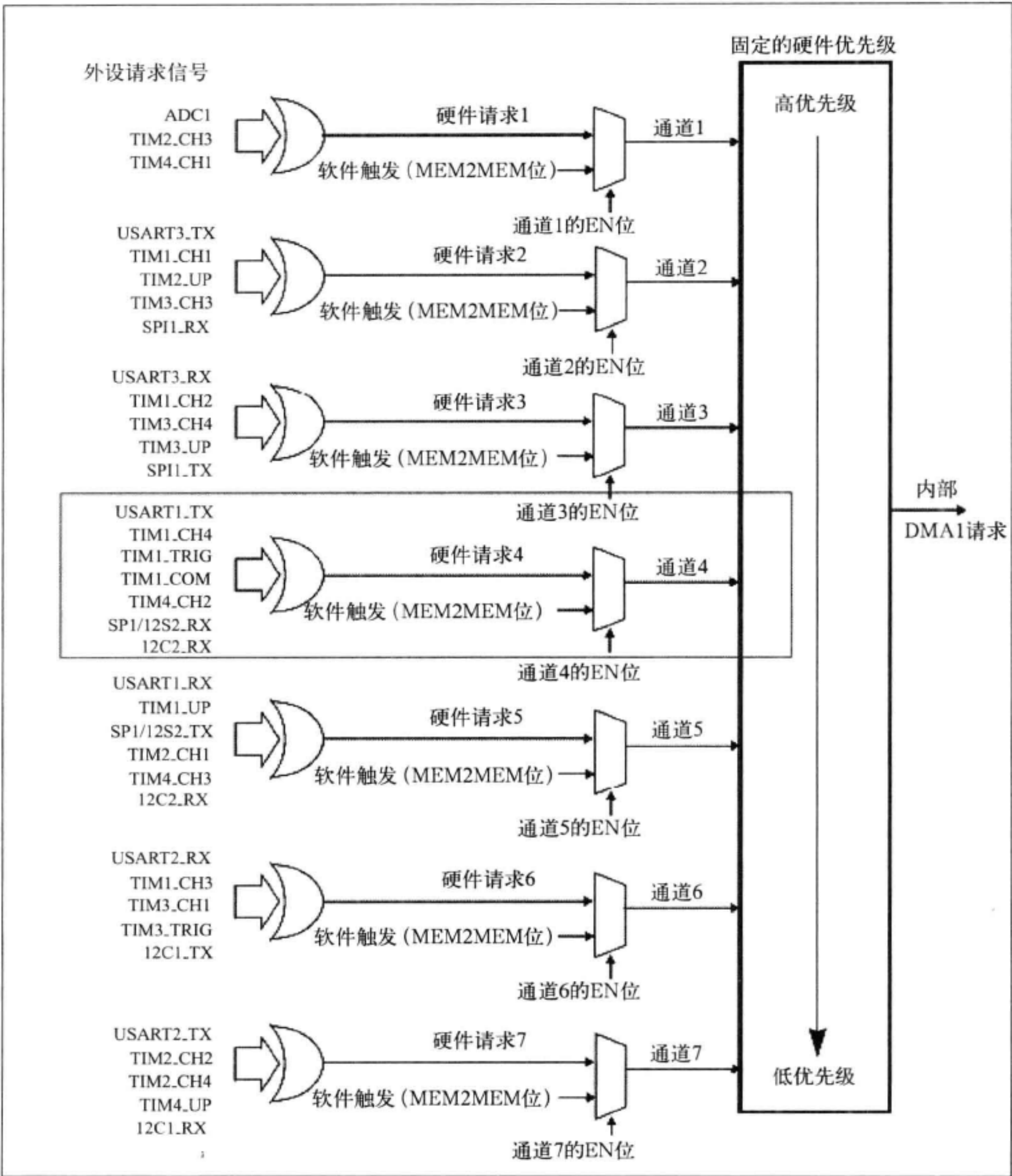


图 10-2 DMA1 请求映像

从图 10-2 中看到即使同样是外设串口 1，串口 1 的发送数据 DMA 请求和串口 1 的接收数据 DMA 请求通道都是不一样的，分别为 DMA1 通道 4 和 DMA1 的通道 5。

从整体来说，DMA 被配置为：数据传输方向是从内存（数组 SendBuff）到 USART1 外设的

数据寄存器 (USART1_DR_Base), 要传输的数据总量为 SENDBUFF_SIZE (5000) 个字节, 并且传输时内存地址自增、外设地址固定, DMA 模式为非循环模式, DMA 通道为 DMA1 的通道 4。

10.3.5 使用 DMA 中断

为了更好地看到实验现象, DMA_Config() 还配置了 DMA 发送完成中断, 在 stm32f10x_it.c 文件的中断服务函数对 LED 灯进行操作 (实际上不使用 DMA 中断就已经可以完成 DMA 传输了)。

在 DMA_Config() 代码中的第 15 行调用用户函数 NVIC_Config() 配置了 DMA 的中断优先级, 而用户函数 NVIC_Config() 也跟 EXTI 配置外部中断时用到的类似, 只是其中断通道换成了 DMA1_Channel4_IRQn (DMA1 的 4 通道中断)。NVIC_Config() 代码见代码清单 10-5。

代码清单 10-5 NVIC_Config() 函数

```

1.  /*
2.   * 函数名: NVIC_Config
3.   * 描述  : DMA 中断配置
4.   * 输入  : 无
5.   * 输出  : 无
6.   * 调用  : 外部调用
7.   */
8. static void NVIC_Config(void)
9. {
10.  NVIC_InitTypeDef NVIC_InitStructure;
11.
12.  /* Configure one bit for preemption priority */
13.  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
14.
15.  /* 配置 P[A|B|C|D|E]0 为中断源 */
16.  NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel4_IRQn;
17.  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
18.  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
19.  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
20.  NVIC_Init(&NVIC_InitStructure);
21. }
```

在 DMA_Config() 函数的第 52 行调用了 ST 库函数 DMA_ITConfig() 把 DMA 配置成 DMA_IT_TC (DMA 发送完成标志) 中断。DMA_ITConfig() 是用于配置 DMA 外设中断的函数, 在第 9 章已经介绍过类似函数了。

接下来还要在 stm32f10x_it.c 文件中编写中断服务函数, 见代码清单 10-6。

代码清单 10-6 DMA 中断服务函数

```

1. void DMA1_Channel4_IRQHandler(void)
2. {
3.  // 判断是否为 DMA 发送完成中断
4.  if(DMA_GetFlagStatus(DMA1_FLAG_TC4)==SET)
```

10.3.6 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上串口线，打开串口调试助手，配置调试助手为“115200 8-N-1”和“十六进制显示”。将编译好的程序下载到开发板，即可看到超级终端打印出图 10-4 所示信息，并且在 PC 端数据还没接收完的时候，板子上的 LED1 被点亮，在数据接收完成时，LED1 被关闭了，说明内核的确在 DMA 传输的过程中同时进行了其他工作。



图 10-4 串口 DMA 实验现象图



第 11 章

ADC 实验（DMA 方式）

11.1 ADC 简介

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

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

11.2 STM32 的 ADC 主要技术指标

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

□ 分辨率

12 位分辨率。不能直接测量负电压, 所以没有符号位, 即其最小量化单位 $LSB = V_{REF+} / 2^{12}$ 。

□ 转换时间

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

□ ADC 类型

ADC 类型决定了其性能的极限, STM32 的 ADC 是逐次比较型 ADC。

□ 参考电压范围

STM32 的 ADC 参考电压输入见表 11-1。

表 11-1 ADC 参考电压

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

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

从表 11-1 可知，它的参考电压负极是要接地的，即 $V_{\text{REF-}} = 0\text{V}$ 。而参考电压正极的范围为 $2.4\text{V} \leq V_{\text{REF}+} \leq 3.6\text{V}$ ，所以 STM32 的 ADC 是不能直接测量负电压的，而且其输入的电压信号的范围为： $V_{\text{REF-}} \leq V_{\text{IN}} \leq V_{\text{REF}+}$ 。当需要测量负电压或测量的电压信号超出范围时，要先经过运算电路进行平移或利用电阻分压。

11.3 ADC 工作过程分析

我们以 ADC 的规则通道转换来进行过程分析，见图 11-1。所有的器件都是围绕中间的模拟至数字转换器部分（下面简称 ADC 部件）展开的。它的左端为 V_{REF+}、V_{REF-} 等 ADC 参考电压，ADCx_IN0 ~ ADCx_IN15 为 ADC 的输入信号通道，即某些 GPIO 引脚。输入信号经过这些通道被送到 ADC 部件，ADC 部件需要受到触发信号才开始进行转换，如 EXTI 外部触发、定时器触发，也可以使用软件触发。ADC 部件接收到触发信号之后，在 ADCCLK 时钟的驱动下对输入通道的信号进行采样，并进行模数转换，其中 ADCCLK 是来自 ADC 预分频器的。

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

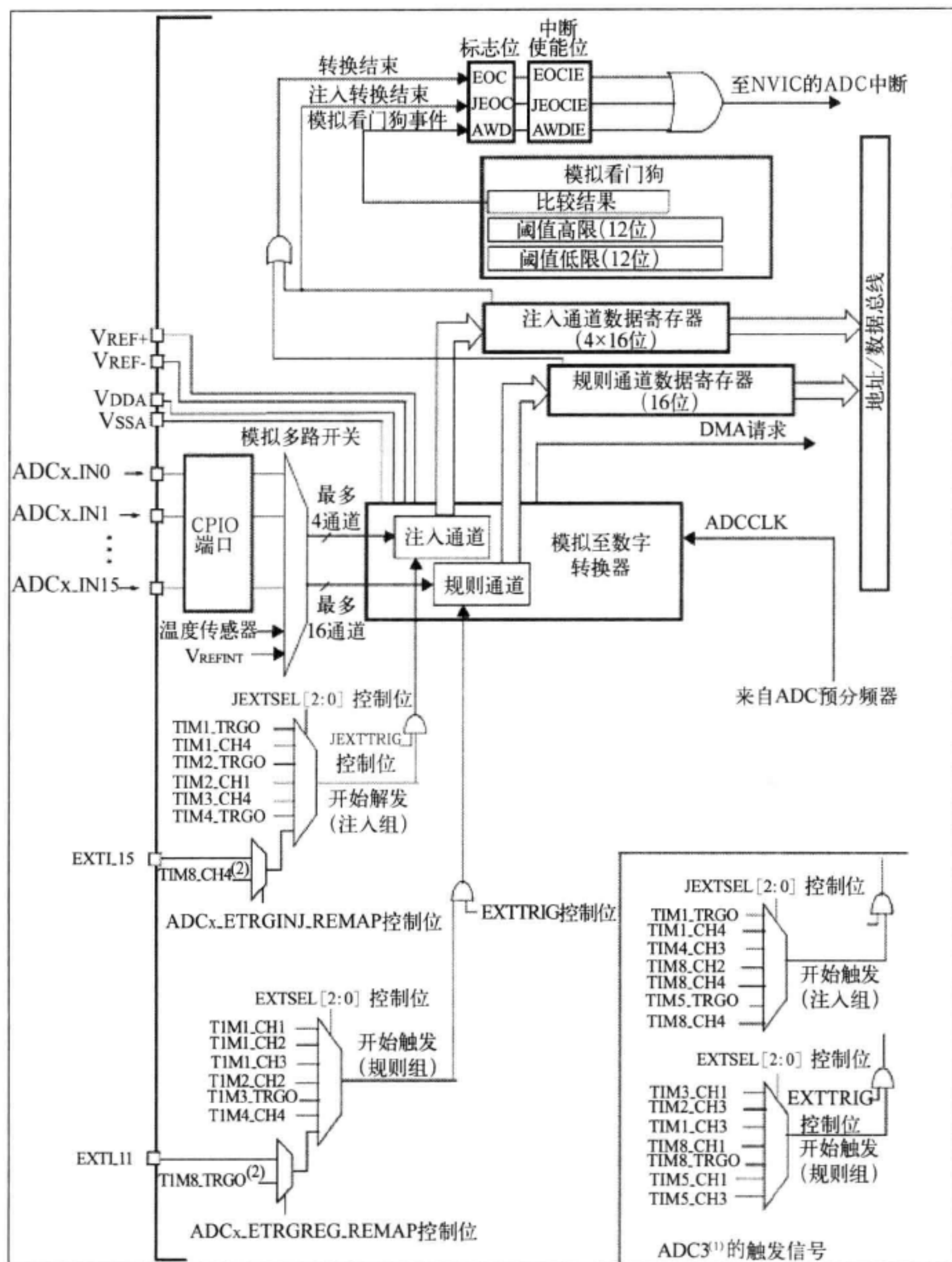


图 11-1 ADC 架构图

11.4 ADC 采集数据实例（采用 DMA 模式）

使用 ADC 时常常需要不间断采集大量的数据，在一般的器件中会使用中断进行处理，但使用中断的效率还是不够高。在 STM32 中，使用 ADC 时往往采用 DMA 传输方式，由 DMA 把 ADC 外设

转换的数据传输到 SRAM, 再进行处理, 甚至直接把 ADC 的数据转移到串口发送给上位机。本节对 ADC 的 DMA 方式采集数据实例进行讲解, 在讲解 ADC 的同时让读者进一步熟悉 DMA 的使用。

11.4.1 实验描述及工程文件清单

1. 实验描述

串口 1 (USART1) 向计算机的超级终端以一定的时间间隔打印当前 ADC1 的转换电压值。

2. 硬件连接

PC1 - ADC1 连接外部电压 (通过一个滑动变阻器分压而来)。图 11-2 为 ADC 硬件原理图。

3. 库文件

3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_adc.c
- ☐ FWlib/stm32f10x_dma.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/usart1.c
- ☐ USER/adc.c

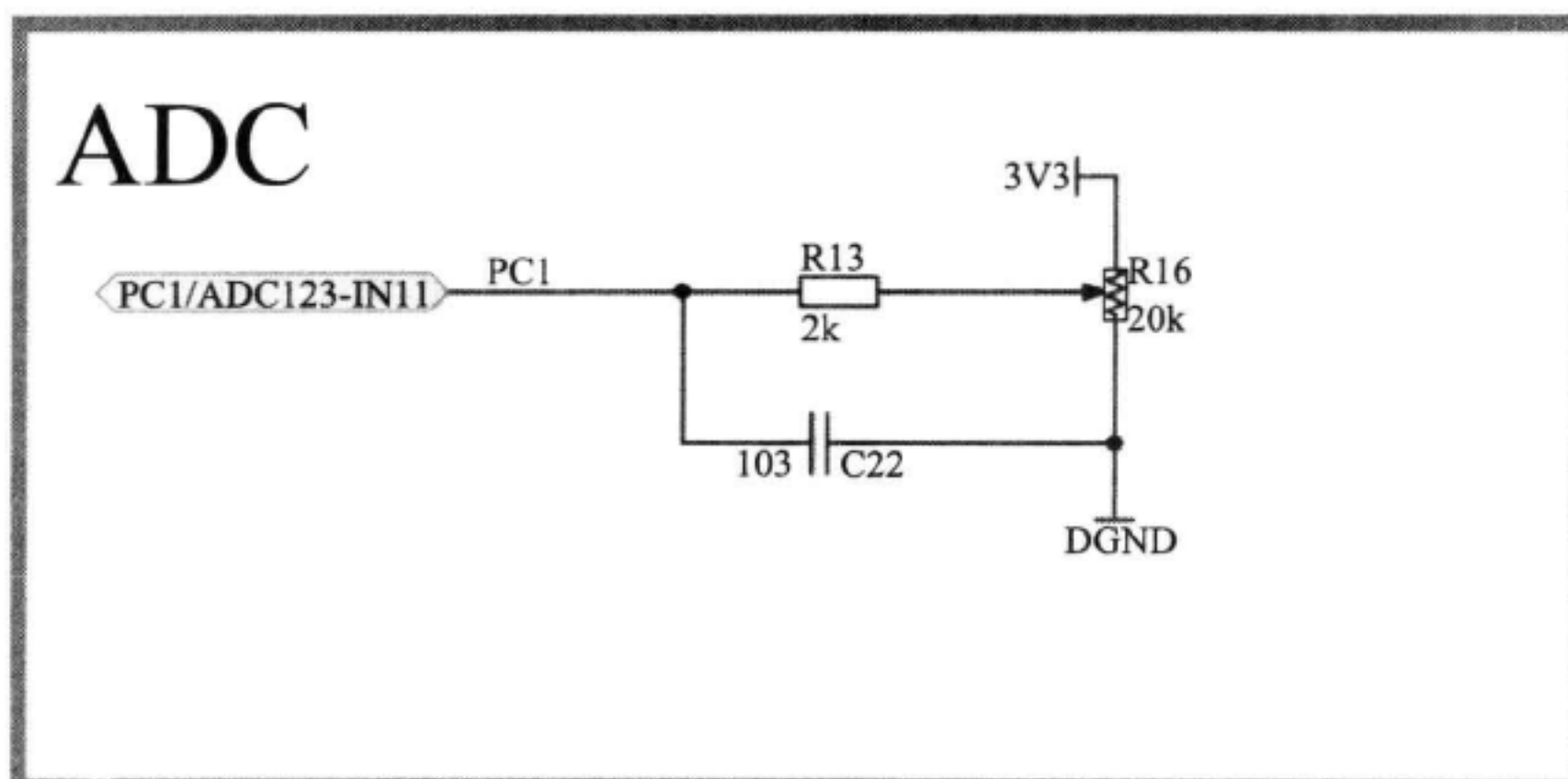


图 11-2 配套 STM32 开发板 ADC 硬件原理图

11.4.2 配置工程环境

本 ADC (DMA 方式) 实验中我们用到了 GPIO、RCC、USART、DMA 及 ADC 外设, 所以我们要把以下库文件添加到工程中: stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_dma.c、stm32f10x_adc.c, 添加旧工程中的外设用户文件 usart1.c, 新建 adc.c 及 adc.h 文件, 并在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉, 见代码清单 11-1。

代码清单 11-1 ADC 例程的 stm32f10x_conf.h 文件配置

```

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

```

11.4.3 main 文件

配置好工程环境之后, 我们就从 main 文件开始分析, 见代码清单 11-2。

代码清单 11-2 ADC 例程的 main 函数

```

1. #include "stm32f10x.h"
2. #include "usart1.h"
3. #include "adc.h"
4.
5. // ADC1 转换的电压值通过 DMA 方式传到 SRAM
6. extern __IO uint16_t ADC_ConvertedValue;
7.
8. // 局部变量, 用于保存转换计算后的电压值
9.
10. float ADC_ConvertedValueLocal;
11.
12. // 软件延时
13. void Delay(__IO uint32_t nCount)
14. {
15.     for(; nCount != 0; nCount--);
16. }
17.
18. /**
19.  * @brief  Main program.
20.  * @param  None
21.  * @retval : None

```

```

22.  */
23.
24. int main(void)
25. {
26.     /* USART1 config */
27.     USART1_Config();
28.
29.     /* enable adc1 and config adc1 to dma mode */
30.     ADC1_Init();
31.
32.     printf("\r\n ----- 这是一个 ADC 实验 ----- \r\n");
33.
34.     while (1)
35.     {
36.         ADC_ConvertedValueLocal = (float) ADC_ConvertedValue/4096*3.3; // 读取转换的 AD 值
37.
38.         printf("\r\n The current AD value = 0x%04X \r\n", ADC_ConvertedValue);
39.         printf("\r\n The current AD value = %f V \r\n", ADC_ConvertedValueLocal);
40.
41.         Delay(0xfffffee); // 延时
42.
43.
44.     }
45. }

```

浏览一遍 main 函数，在调用了用户函数 USART1_Config() 及 ADC1_Init() 配置好串口和 ADC 之后，就可以直接使用保存了 ADC 转换值的变量 ADC_ConvertedValue 了，在 main 函数中并没有对 ADC_ConvertedValue 重新赋值，这个变量是在什么时候改变的呢？除了可能在中断服务函数修改了变量值，就只有 DMA 有这样的能耐了。而且大家知道，在使用 DMA 传输时，由于不是内核执行的指令，所以修改变量值是绝对不会出现赋值语句的。

11.4.4 ADC 初始化

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

ADC1_Init() 函数使能了 ADC1，并使 ADC1 工作于 DMA 方式。ADC1_Init() 是在用户文件 adc.c 中实现的用户函数，见代码清单 11-3。

代码清单 11-3 ADC1_Init() 函数

```

1. /*
2.  * 函数名: ADC1_Init
3.  * 描述   : 无
4.  * 输入   : 无

```



```

5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void ADC1_Init(void)
9. {
10.     ADC1_GPIO_Config();
11.     ADC1_Mode_Config();
12. }

```

ADC1_Init() 调用了 ADC1_GPIO_Config() 和 ADC1_Mode_Config(), 这两个函数的作用分别是: 前者配置 ADC1 所用的 I/O 端口, 后者配置 ADC1 初始化和使用 DMA 模式运行。

1. 配置 GPIO 端口

关于 ADC 使用的 GPIO 口配置, 见代码清单 11-4。

代码清单 11-4 ADC1_GPIO_Config() 代码

```

1. /*
2.  * 函数名: ADC1_GPIO_Config
3.  * 描述   : 使能 ADC1 和 DMA1 的时钟, 初始化 PC.01
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void ADC1_GPIO_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* Enable DMA clock */
13.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
14.
15.     /* Enable ADC1 and GPIOC clock */
16.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOC, ENABLE);
17.
18.     /* Configure PC.01 as analog input */
19.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
20.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
21.     GPIO_Init(GPIOC, &GPIO_InitStructure); // PC1, 输入时不用设置速率
22. }

```

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

这里涉及 ADC 通道的知识, 每个 ADC 通道都对应一个 GPIO 引脚端口, GPIO 的引脚在设置为模拟输入模式后可用于模拟电压的输入。STM32F103VET6 有三个 ADC, 这三个 ADC 共用 16 个外部通道, 从《STM32 数据手册》的引脚定义可找到 ADC 通道与 GPIO 引脚的关系。见表 11-2。

表 11-2 ADC 通道引脚图（部分）

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCSP64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
H1	F1	E8	8	15	26	PC0	I/O		PC0	ADC123_IN10	
H2	F2	F8	9	16	27	PC1	I/O		PC1	ADC123_IN11	
H3	E2	D6	10	17	28	PC2	I/O		PC2	ADC123_IN12	
H4	F3	-	11	18	29	PC3	I/O		PC3	ADC123_IN13	

表中的引脚名称标注中出现的 ADC123_INx（x 表示 4 ~ 9 或 14 ~ 15 之间的整数），表示这个引脚可以是 ADC1_INx 或 ADC2_INx。例如：ADC12_IN9 表示这个引脚可以配置为 ADC1_IN9，也可以配置为 ADC2_IN9。

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

2. 配置 DMA

ADC 模式及其 DMA 传输方式都是在用户函数 ADC1_Mode_Config() 中实现的，见代码清单 11-5。

代码清单 11-5 ADC1_Mode_Config() 函数代码

```

1. /* 函数名：ADC1_Mode_Config
2. * 描述   ：配置 ADC1 的工作模式为 DMA 模式
3. * 输入   ：无
4. * 输出   ：无
5. * 调用   ：内部调用
6. */
7. static void ADC1_Mode_Config(void)
8. {
9.     DMA_InitTypeDef DMA_InitStructure;
10.    ADC_InitTypeDef ADC_InitStructure;
11.
12.    /* DMA channell configuration */
13.    DMA_DeInit(DMA1_Channel1);
14.    DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
15.    /*ADC 地址*/
16.    DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;
17.    /* 内存地址 */
18.    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; // 外设为数据源
19.    DMA_InitStructure.DMA_BufferSize = 1;
20.    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;

```

```

21. /* 外设地址固定 */
22.     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
23. /* 内存地址固定 */
24.     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; // 半字
25.     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
26.     DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; // 循环传输
27.     DMA_InitStructure.DMA_Priority = DMA_Priority_High;
28.     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
29.     DMA_Init(DMA1_Channel1, &DMA_InitStructure);
30.
31. /* Enable DMA channel1 */
32.     DMA_Cmd(DMA1_Channel1, ENABLE);
33.
34. /* ADC1 configuration */
35.
36.     ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
37. /* 独立ADC模式 */
38.     ADC_InitStructure.ADC_ScanConvMode = DISABLE ;
39. /* 禁止扫描模式, 扫描模式用于多通道采集 */
40.     ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
41. /* 开启连续转换模式, 即不停地进行ADC转换 */
42.     ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None; /* 不使用外部触发转换 */
43.     ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
44. /* 采集数据右对齐 */
45.     ADC_InitStructure.ADC_NbrOfChannel = 1; // 要转换的通道数目 1 */
46.     ADC_Init(ADC1, &ADC_InitStructure);
47.
48. /* 配置ADC时钟, 为PCLK2的8分频, 即9Hz */
49.     RCC_ADCCLKConfig(RCC_PCLK2_Div8);
50. /* 配置ADC1的通道11为55.5个采样周期 */
51.     ADC_RegularChannelConfig(ADC1, ADC_Channel_11, 1, ADC_SampleTime_55Cycles5);
52.
53. /* Enable ADC1 DMA */
54.     ADC_DMACmd(ADC1, ENABLE);
55.
56. /* Enable ADC1 */
57.     ADC_Cmd(ADC1, ENABLE);
58.
59. /* 复位校准寄存器 */
60.     ADC_ResetCalibration(ADC1);
61. /* 等待校准寄存器复位完成 */
62.     while(ADC_GetResetCalibrationStatus(ADC1));
63.
64. /* ADC校准 */
65.     ADC_StartCalibration(ADC1);
66. /* 等待校准完成 */
67.     while(ADC_GetCalibrationStatus(ADC1));
68.
69. /* 由于没有采用外部触发, 所以使用软件触发ADC转换 */
70.     ADC_SoftwareStartConvCmd(ADC1, ENABLE);
71. }

```

ADC 的 DMA 配置部分与串口 DMA 配置部分类似，它的 DMA 整体上被配置为：使用 DMA1 的通道 1，数据从 ADC 外设的数据寄存器（ADC1_DR_Address）转移到内存（ADC_ConvertedValue 变量），内存、外设地址都固定，每次传输的数据大小为半字（16 位），使用 DMA 循环传输模式。

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

DMA 传输的外设地址 ADC1_DR_Address 是一个自定义的宏：

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

ADC_DR 数据寄存器保存了 ADC 转换后的数值，以它作为 DMA 的传输源地址。它的地址是由 ADC1 外设的基地址（0x4001 2400）加上 ADC 数据寄存器（ADC_DR）的地址偏移（0x4c）计算得到的。见表 11-3 和表 11-4。

表 11-3 《STM32 参考手册》的 ADC1 起止地址说明

起 止 地 址	外 设
0x4001 2800 - 0x4001 2BFF	ADC2
0x4001 2400 - 0x4001 27FF	ADC1

表 11-4 ADC_DR 寄存器描述及其地址偏移

ADC规则数据寄存器（ADC_DR）															
地址偏移：0x4C															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADC2DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
位 数	描 述														
31:16	ADC2DATA[15:0] ：ADC2 转换的数据（ADC2 data） • 在 ADC1 中：双模式下，这些位包含了 ADC2 转换的规则通道数据 • 在 ADC2 和 ADC3 中：不使用这些位														
15:0	DATA[15:0] ：规则转换的数据（Regular data），这些位为只读，包含了规则通道的转换结果。数据是左对齐或右对齐														

3. 配置 ADC 模式

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

1) .ADC_Mode：STM32 具有多个 ADC，而不同的 ADC 又是共用通道的，当两个 ADC 采

集同一个通道的先后顺序、时间间隔不同，就演变出了各种各样的模式，如同步注入模式、同步规则模式等 10 种模式，应选择适合的模式以适应采集数据的要求。本实验用于测量电阻分压后的电压值，要求不高，只使用一个 ADC 就可以满足要求了，所以本成员被赋值为 ADC_Mode_Independent（独立模式）。

2) .ADC_ScanConvMode：当有多个通道需要采集信号时，可以把 ADC 配置为按一定的顺序来对各个通道进行扫描转换，即轮流采集各通道的值。若采集多个通道，必须开启此模式。本实验只采集一个通道的信号，所以 DISABLE（禁止）使用扫描转换模式。

3) .ADC_ContinuousConvMode：连续转换模式，此模式与单次转换模式相反，单次转换模式 ADC 只采集一次数据就停止转换。而连续转换模式则在上一次 ADC 转换完成后，立即开启下一次转换。本实验需要循环采集电压值，所以 ENABLE（使能）连续转换模式。

4) .ADC_ExternalTrigConv：ADC 需要在接收到触发信号后才开始进行模数转换，如外部中断触发（EXTI 线）、定时器触发，这两个为外部触发信号，如果不使用外部触发信号可以使用软件控制触发。本实验中使用软件控制触发，所以该成员被赋值为 ADC_ExternalTrigConv_None（不使用外部触发）。

5) .ADC_DataAlign：数据对齐方式。ADC 转换后的数值被保存到数据寄存器（ADC_DR）的 0 ~ 15 位或 16 ~ 32 位，数据宽度为 16 位，而 ADC 转换精度为 12 位。把 12 位的数据保存到 16 位的区域，就涉及左对齐和右对齐的问题。这里的左、右对齐跟 Word 文档中的文本左、右对齐是一样的意思。左对齐即 ADC 转换的数值最高位 D12 与存储区域的最高位 Bit15 对齐，存储区域的低 4 位无意义。右对齐则相反，ADC 转换的数值最低位 D0 保存在存储区域的最低位 Bit0，高 4 位无意义。见表 11-5（本例程使用的是规则组）。

表 11-5 ADC 数据对齐

数据右对齐															
注入组															
SEXT	SEXT	SEXT	SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
规则组															
0	0	0	0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
数据左对齐															
注入组															
SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0
规则组															
D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0

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

6) .ADC_NbrOfChannel：这个成员保存了要进行 ADC 数据转换的通道数，可以为 1 ~ 16 个。

本实验中只需要采集 PC1 这个通道, 所以把成员赋值为 1 就可以了。

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

4. ADC 转换时间配置

配置好了 ADC 的模式, 还要设置 ADC 的时钟 (ADCCLK), 见图 11-3。ADC 时钟频率越高, 转换速度也就越快, 但 ADC 时钟有上限值, 即不能超过 14 MHz。

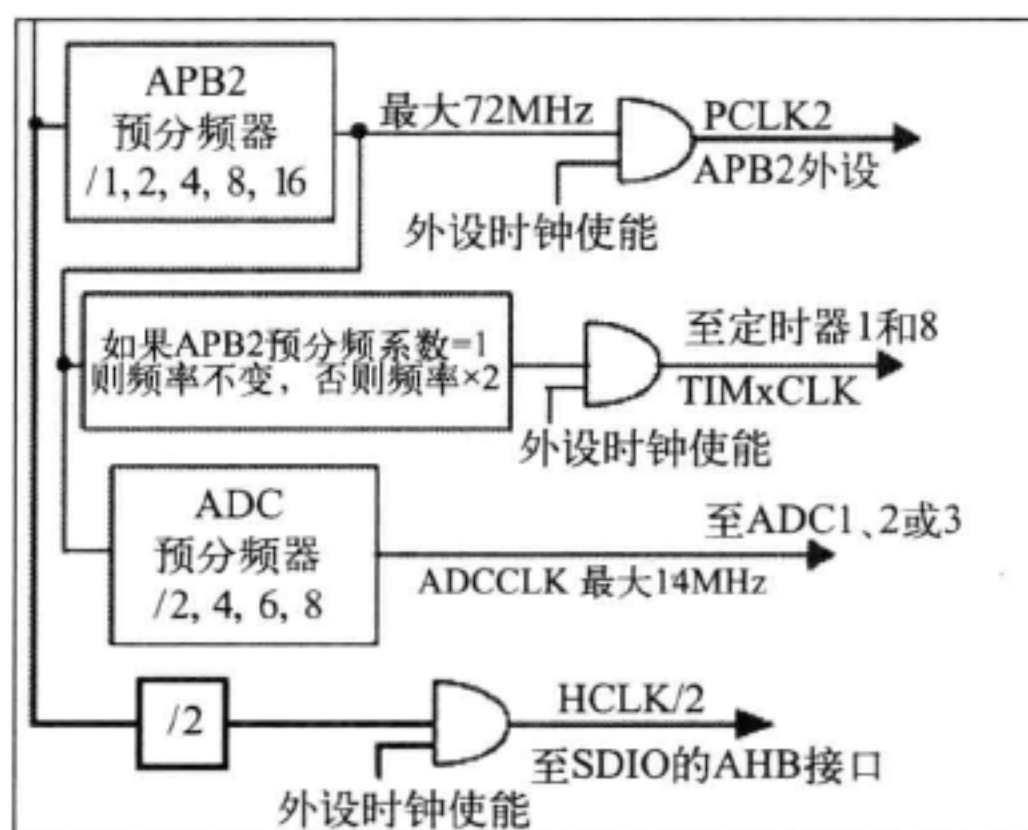


图 11-3 ADC 时钟

配置 ADC 时钟, 可使用函数 RCC_ADCCLKConfig() 配置, 见图 11-4。

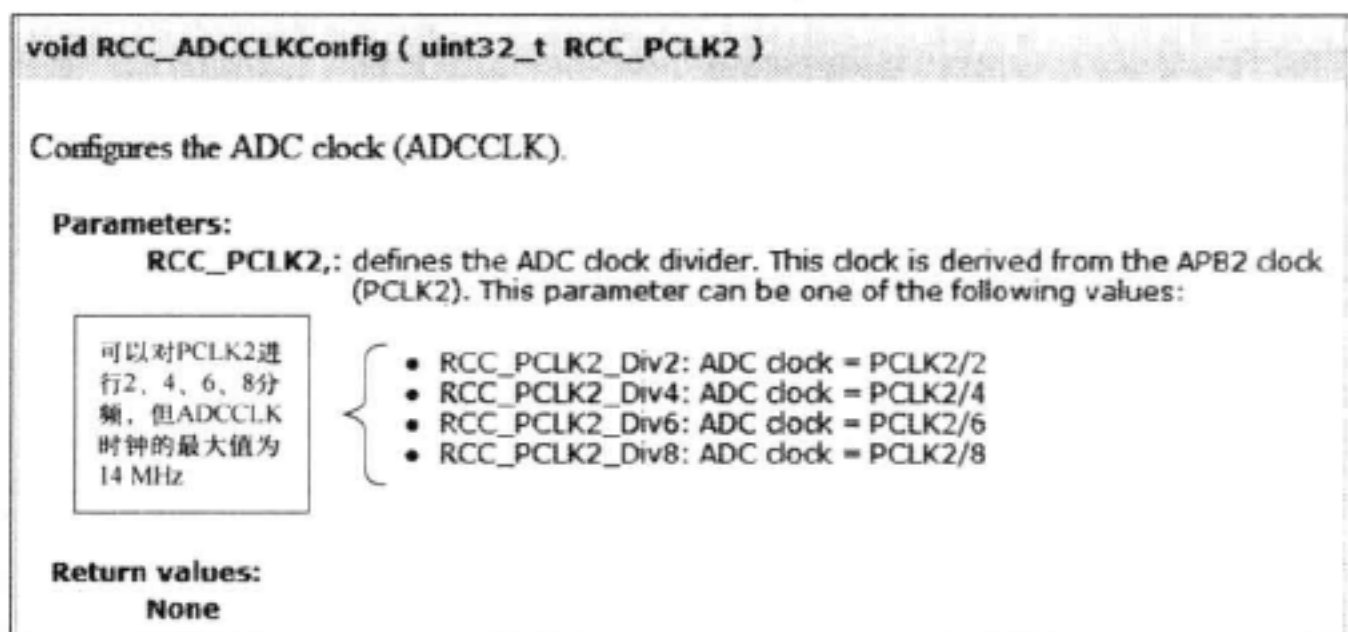


图 11-4 ADCCLK 时钟配置函数

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

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

根据 STM32 的 ADC 采样时间计算公式：

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

公式中的采样周期就是本函数中配置的 ADC_SampleTime，而后边加上的 12.5 个周期为固定的数值。所以，本实验中 ADC1 通道 11 的转换时间 $T_{\text{CONV}} = (55.5 + 12.5) \times 1/9 \approx 7.56\mu\text{s}$ 。

5. ADC 自校准

在开始 ADC 转换之前，需要启动 ADC 的自校准。ADC 有一个内置自校准模式，校准可大幅减小因内部电容器组的变化而造成的精度误差。在校准期间，在每个电容器上都会计算出一个误差修正码（数字值），这个码用于消除在随后的转换中每个电容器上产生的误差。见代码清单 11-6。这是在 ADC1_Mode_Config() 函数中的 ADC 自校准时调用的库函数和使用步骤。

代码清单 11-6 ADC 自校准

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

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

在校准完成后，就可以开始进行 ADC 转换了。本实验代码中配置的 ADC 模式为软件触发方式，我们可以调用库函数 ADC_SoftwareStartConvCmd() 来开启软件触发。其说明见图 11-6。

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

6. volatile 变量

现在我们来认识 ADC_ConvertedValue 这个变量：

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

ADC_ConvertedValue 在文件 adc.c 中定义。这里要注意一点的是，这个变量要用 C 语言的 volatile 关键字来修饰，为的是让编译器不要去优化这个变量。这样每次用到这个变量时都要回到

```
void ADC_SoftwareStartConvCmd (ADC_TypeDef * ADCx,
                               FunctionalState NewState)
{
    /* Enable or disable the selected ADC software start conversion */
}

Parameters:
  ADCx: where x can be 1, 2 or 3 to select the ADC peripheral.
  NewState: new state of the selected ADC software start conversion. This
            parameter can be: ENABLE or DISABLE.

Return values:
  None
```

图 11-6 ADC 软件触发控制函数

相应变量的内存中去取值，而 volatile 字面意思就是“可变的，不确定的”。

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

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

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

11.4.5 计算电压值

回到 main 函数中循环打印电压值部分见代码清单 11-7。

代码清单 11-7 计算并打印电压值

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

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

实际电压值 = ADC 转换值 × LSB

STM32 的 ADC 的精度为 12 位，而配套板子中 V_{REF+} 接的参考电压值为 3.3V，所以 $LSB = 3.3/2^{12}$ 。

11.4.6 实验现象

将配套 STM32 开发板供电（DC5V），插上 J_LINK，插上串口线（两头都是母的交叉线），打开超级终端，配置超级终端为“15200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息，见图 11-7。

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

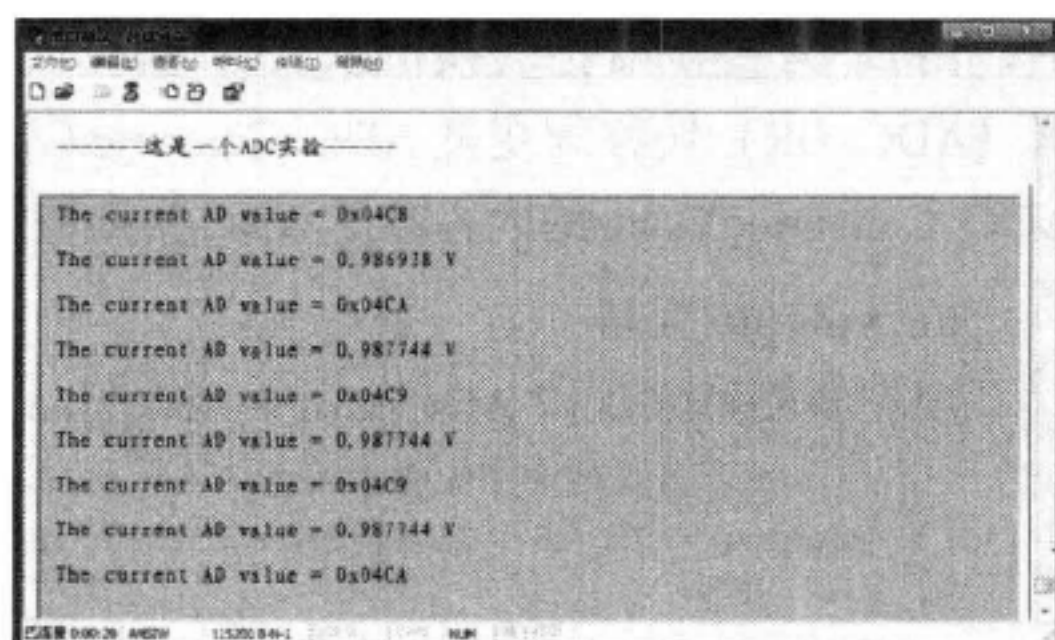
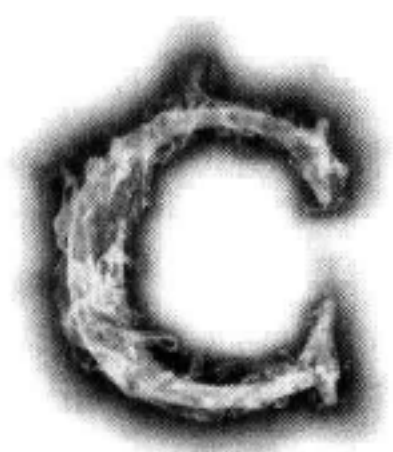


图 11-7 ADC 采集外部电压实验



第 12 章

SysTick（系统滴答定时器）

12.1 SysTick——操作系统的心跳

SysTick 定时器被捆绑在 NVIC 中，用于产生 SysTick 异常（异常号：15）。在以前，操作系统和所有使用了时基的系统都必须有一个硬件定时器来产生需要的“滴答”中断，作为整个系统的时基。滴答中断对操作系统尤其重要。例如，操作系统可以为多个任务分配不同数目的时间片，确保没有一个任务能霸占系统；或者将每个定时器周期的某个时间范围赐予特定的任务等，操作系统提供的各种定时功能都与这个滴答定时器有关。因此，需要一个定时器来产生周期性的中断，而且最好还让用户程序不能随意访问它的寄存器，以维持操作系统“心跳”的节律。

Cortex-M3 在内核部分包含了一个简单的定时器——SysTick。因为所有的 CM3 芯片都带有这个定时器，软件在不同芯片生产厂商的 CM3 器件间的移植工作就得以简化。该定时器的时钟源可以是内部时钟（FCLK，CM3 上的自由运行时钟），或者是外部时钟（CM3 处理器上的 STCLK 信号）。不过，STCLK 的具体来源则由芯片设计者决定，因此不同产品之间的时钟频率可能大不相同。因此，需要阅读芯片的使用手册来确定选择什么作为时钟源。在 STM32 中 SysTick 以 HCLK（AHB 时钟）或 HCLK/8 作为运行时钟，见图 12-1。

SysTick 定时器能产生中断，CM3 为它专门开出一个异常类型，并且在向量表中有它的一席之地。它使操作系统和其他系统软件在 CM3 器件间的移植变得简单多了，因为在所有 CM3 产品间，SysTick 的处理方式都是相同的。SysTick 定时器除了能服务于操作系统之外，还能用于其他目的，如作为一个闹铃、用于测量时间等。

Systick 定时器属于 Cortex 内核部件，可以参考《ARM Cortex-M3 权威指南》（（英）Joseph Yiu 著，宋岩译，北京航空航天大学出版社出版）或“STM32xxx-Cortex-M3programming manual”（这是 ST 官方提供的电子版编程手册，可以在 ST 官网下载）来了解。

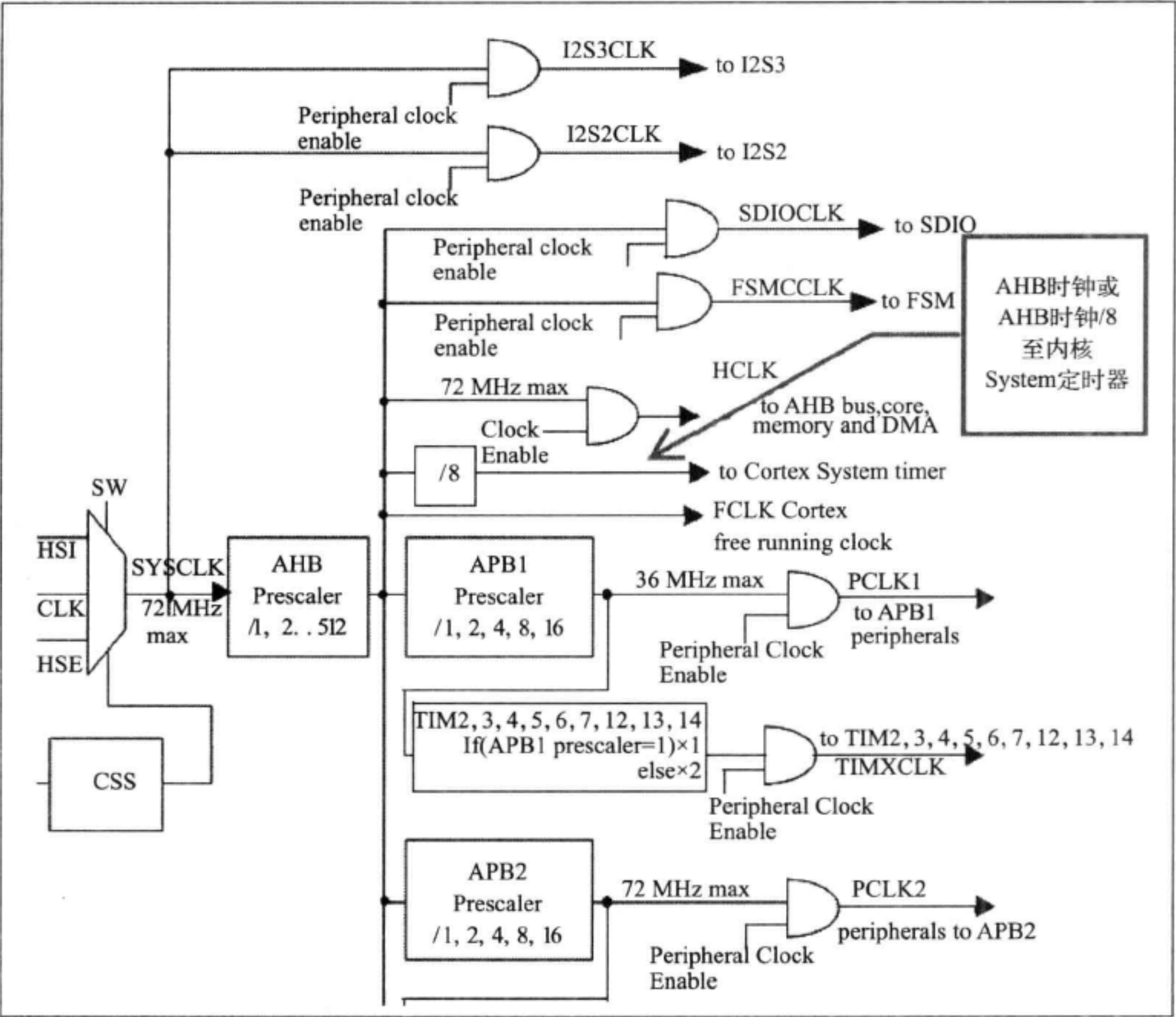


图 12-1 时钟树（部分）-SysTick timer 时钟来源

12.2 SysTick 工作分析

SysTick 是一个 24 位的定时器，即一次最多可以计数 2^{24} 个时钟脉冲，这个脉冲计数值被保存到当前计数值寄存器 STK_VAL (SysTick current value register) 中，只能向下计数，每接收到一个时钟脉冲 STK_VAL 的值就向下减 1，直至 0。当 STK_VAL 的值被减至 0 时，由硬件自动把重载寄存器 STK_LOAD (SysTick reload value register) 中保存的数据加载到 STK_VAL，重新向下计数。当 STK_VAL 的值被计数至 0 时，触发异常，就可以在中断服务函数中处理定时事件了。

当然，要使 SysTick 进行以上工作必须要进行 SysTick 配置。它的控制配置很简单，只有三个控制位和一个标志位，都位于寄存器 STK_CTRL (SysTick control and status register) 中，见表 12-1。

12.3.1 实验描述及工程文件清单

1. 实验描述

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

2. 硬件连接

- ❑ LED 灯与 GPIO 的连接
- ❑ PC3 – LED1
- ❑ PC4 – LED2
- ❑ PC5 – LED3

3. 库文件

使用 3.5 版本固件库：

- ❑ startup/start_stm32f10x_hd.c
- ❑ CMSIS/core_cm3.c
- ❑ CMSIS/system_stm32f10x.c
- ❑ FWlib/stm32f10x_gpio.c
- ❑ FWlib/stm32f10x_rcc.c

4. 用户文件

- ❑ USER/main.c
- ❑ USER/stm32f10x_it.c
- ❑ USER/led.c
- ❑ USER/SysTick.c

配套开发板 LED 硬件连接图见图 12-3。

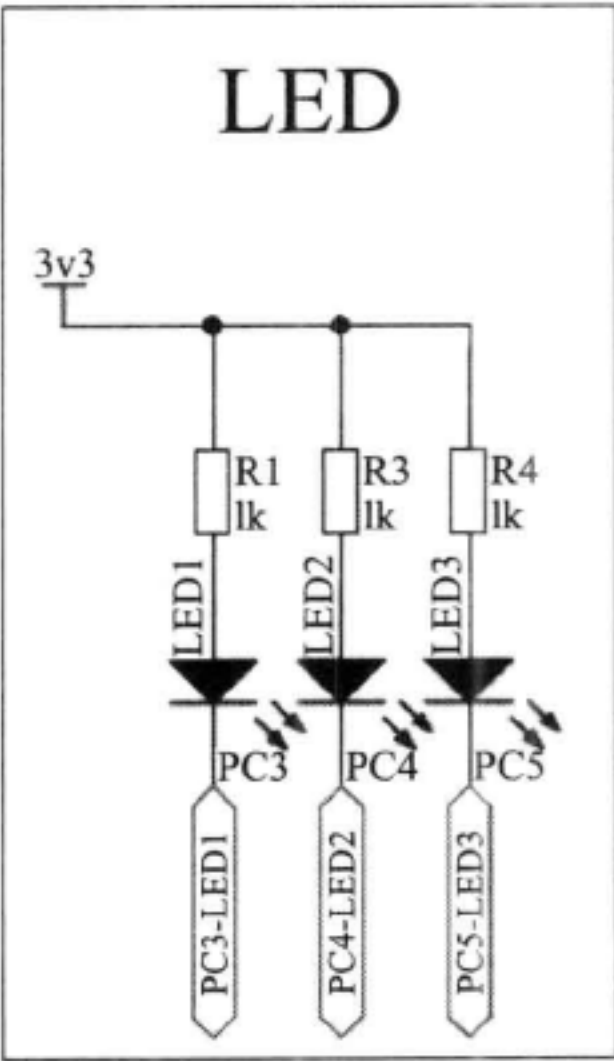


图 12-3 配套开发板 LED 硬件连接图

12.3.2 配置工程环境

本 SysTick 精确延时实验中我们用到了 GPIO、RCC 外设，所以我们先要把以下库文件添加到工程：stm32f10x_gpio.c、stm32f10x_rcc.c。

由于本实验中，SysTick 的中断是由文件 core_cm3.h 的函数配置的，没有使用 NVIC 来配置中断，所以可不添加 misc.c 文件。而 core_cm3.h 在包含 stm32f10x.h 头文件时已被添加进工程了。

接下来添加旧工程中的外设用户文件 led.c，新建 SysTick.c 及 SysTick.h 文件，并在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉，见代码清单 12-1。

代码清单 12-1 SysTick 例程中 stm32f10x_conf.h 的文件配置

```
1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
```

```

7.  * @brief   Library configuration file.
8.  *****/
9.
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"

```

12.3.3 main 文件

我们从 main 函数开始，见代码清单 12-2。

代码清单 12-2 SysTick 的 main 函数

```

1.  /*
2.  * 函数名：main
3.  * 描述   ：主函数
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7.  int main(void)
8.  {
9.      /* LED 端口初始化 */
10.     LED_GPIO_Config();
11.
12.     /* 配置 SysTick 为 10us 中断一次 */
13.     SysTick_Init();
14.
15.     for(;;)
16.     {
17.
18.         LED1( 0 );
19.         Delay_us(50000);           // 50000 * 10μs = 500ms
20.         LED1( 1 );
21.
22.         LED2( 0 );
23.         Delay_us(50000);           // 50000 * 10μs = 500ms
24.         LED2( 1 );
25.
26.         LED3( 0 );
27.         Delay_us(50000);           // 50000 * 10μs = 500ms
28.         LED3( 1 );
29.
30.     }
31.
32. }

```

在 main 函数中，SysTick_Init() 和 Delay_us() 这两个函数比较陌生，它们的功能分别是配置好 SysTick 定时器和进行精确延时。

整个 main 函数的流程就是初始化 LED 及 SysTick 定时器之后，就进入死循环，轮流点亮 LED1、LED2、LED3，点亮的时间为精确的 500 ms。

12.3.4 配置并启动 SysTick

接下来我们看一下 SysTick_Init() 这个函数，见代码清单 12-3。它是由用户在 SysTick.c 这个文件中实现的，其功能是启动系统滴答定时器 SysTick，并将 SysTick 配置为 10 μs 中断一次。

代码清单 12-3 SysTick_Init() 函数

```

1.  /*
2.   * 函数名: SysTick_Init
3.   * 描述   : 启动系统滴答定时器 SysTick
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8. void SysTick_Init(void)
9. {
10.     /* SystemFrequency / 1000    1ms 中断一次
11.      * SystemFrequency / 100000   10us 中断一次
12.      * SystemFrequency / 1000000  1us 中断一次
13.      */
14. // if (SysTick_Config(SystemFrequency / 100000)) // ST3.0.0 库版本
15.     if (SysTick_Config(SystemCoreClock / 100000)) // ST3.5.0 库版本
16.     {
17.         /* Capture error */
18.         while (1);
19.     }
20.     // 关闭滴答定时器
21.     SysTick->CTRL &= ~ SysTick_CTRL_ENABLE_Msk;
22. }

```

本函数实际上只是调用了 SysTick_Config() 函数，它是属于内核层的 Cortex-M3 通用函数，位于 core_cm3.h 文件中。若调用 SysTick_Config() 配置 SysTick 不成功，则进入死循环，初始化 SysTick 成功后，先关闭定时器，在需要的时候再开启。

SysTick_Config() 函数无法在《STM32 外设固件库帮助手册 .chm》文件中找到其使用方法。所以我们在 Keil 环境下直接跟踪这个函数到 core_cm3.h 文件，查看函数的定义，见代码清单 12-4。

代码清单 12-4 SysTick_Config() 函数

```

1. /**
2.  * @brief Initialize and start the SysTick counter and its interrupt.
3.  *
4.  * @param ticks    number of ticks between two interrupts
5.  * @return 1 = failed, 0 = successful
6.  *
7.  * Initialise the system tick timer and its interrupt and start the
8.  * system tick timer / counter in free running mode to generate
9.  * periodical interrupts.
10. */
11. static __INLINE uint32_t SysTick_Config(uint32_t ticks)
12. {
13.     /* Reload value impossible */
14.     if (ticks > SysTick_LOAD_RELOAD_Msk) return (1);
15.
16.     /* set reload register */
17.     SysTick->LOAD = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;
18.
19.     /* set Priority for Cortex-M0 System Interrupts */
20.     NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
21.
22.     /* Load the SysTick Counter Value */

```

```

23. SysTick->VAL = 0;
24.
25. SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
26.                 SysTick_CTRL_TICKINT_Msk |
27.                 SysTick_CTRL_ENABLE_Msk;
28. /* Enable SysTick IRQ and SysTick Timer */
29. return (0);
30. /* Function successful */
31. }

```

在这个函数定义的前面有关于它的注释，如果我们不想去研究它的具体实现，可以根据这段注释了解函数的功能：这个函数启动了 SysTick；并把它配置为计数至 0 时引起中断；输入的参数 ticks 为两个中断之间的脉冲数，即相隔 ticks 个时钟周期会引起一次中断；配置 SysTick 成功时返回 0，出错时返回 1。

但是，这段注释并没有告诉我们它把 SysTick 的时钟设置为 AHB 时钟还是 AHB/8，这是一个十分关键的问题，于是，我们将对这个函数的具体实现进行分析，与大家再分享一下如何分析底层库函数。

分析底层库函数，要有 12.2 节关于 SysTick 定时器工作分析的知识准备。

1. 检查输入参数

SysTick_Config() 第 14 行代码是检查输入参数 ticks，因为 ticks 是脉冲计数值，要被保存到重载寄存器 STK_LOAD 寄存器中，再由硬件把 STK_LOAD 值加载到当前计数值寄存器 STK_VAL 中使用，STK_LOAD 和 STK_VAL 都是 24 位的，所以当输入参数 ticks 大于其可存储的最大值时，将由这行代码检查出错误并返回。

2. 位指示宏及位屏蔽宏

检查 ticks 参数没有错误后，就稍稍处理一下把 ticks-1 赋值给 STK_LOAD 寄存器，要注意的是减 1，若 STK_VAL 从 ticks-1 向下计数至 0，实际上就经过了 ticks 个脉冲。这句赋值代码使用了宏 SysTick_LOAD_RELOAD_Msk，与其他库函数类似，这个宏是用来指示寄存器的特定位置或进行位屏蔽的。它以及类似的宏定义见代码清单 12-5。

代码清单 12-5 宏 SysTick_LOAD_RELOAD_Msk

```

1. /* SysTick Control / Status Register Definitions */
2. #define SysTick_CTRL_COUNTFLAG_Pos    16    /*!< SysTick CTRL: COUNTFLAG Position
   */
3. #define SysTick_CTRL_COUNTFLAG_Msk    (1ul << SysTick_CTRL_COUNTFLAG_Pos)
   /*!< SysTick CTRL: COUNTFLAG Mask */
4.
5. #define SysTick_CTRL_CLKSOURCE_Pos     2     /*!< SysTick CTRL: CLKSOURCE Position */
6. #define SysTick_CTRL_CLKSOURCE_Msk    (1ul << SysTick_CTRL_CLKSOURCE_Pos)
   /*!< SysTick CTRL: CLKSOURCE Mask */
7.
8. #define SysTick_CTRL_TICKINT_Pos       1     /*!< SysTick CTRL: TICKINT Position */
9. #define SysTick_CTRL_TICKINT_Msk      (1ul << SysTick_CTRL_TICKINT_Pos)
   /*!< SysTick CTRL: TICKINT Mask */
10.
11. #define SysTick_CTRL_ENABLE_Pos        0     /*!< SysTick CTRL: ENABLE Position */
12. #define SysTick_CTRL_ENABLE_Msk       (1ul << SysTick_CTRL_ENABLE_Pos)
   /*!< SysTick CTRL: ENABLE Mask */
13.
14. /* SysTick Reload Register Definitions */

```

```

15. #define SysTick_LOAD_RELOAD_Pos 0 /*!< SysTick LOAD: RELOAD Position */
16. #define SysTick_LOAD_RELOAD_Msk (0xFFFFFul << SysTick_LOAD_RELOAD_Pos)
    /*!< SysTick LOAD: RELOAD Mask */
17.
18. /* SysTick Current Register Definitions */
19. #define SysTick_VAL_CURRENT_Pos 0 /*!< SysTick VAL: CURRENT Position */
20. #define SysTick_VAL_CURRENT_Msk (0xFFFFFul << SysTick_VAL_CURRENT_Pos)
    /*!< SysTick VAL: CURRENT Mask */
21.
22. /* SysTick Calibration Register Definitions */
23. #define SysTick_CALIB_NOREF_Pos 31 /*!< SysTick CALIB: NOREF Position */
24. #define SysTick_CALIB_NOREF_Msk (1ul << SysTick_CALIB_NOREF_Pos)
    /*!< SysTick CALIB: NOREF Mask */
25.
26. #define SysTick_CALIB_SKEW_Pos 30 /*!< SysTick CALIB: SKEW Position */
27. #define SysTick_CALIB_SKEW_Msk (1ul << SysTick_CALIB_SKEW_Pos)
    /*!< SysTick CALIB: SKEW Mask */
28.
29. #define SysTick_CALIB_TENMS_Pos 0 /*!< SysTick CALIB: TENMS Position */
30. #define SysTick_CALIB_TENMS_Msk (0xFFFFFul << SysTick_VAL_CURRENT_Pos) /*!<
    SysTick CALIB: TENMS Mask */
31. /* @} */ /* end of group CMSIS_CM3_SysTick */

```

其中寄存器位指示宏：SysTick_xxx_Pos，宏展开后即为 xxx 在相应寄存器中的位置，如控制 SysTick 时钟源的 SysTick_CTRL_CLKSOURCE_Pos，宏展开为 2，这个寄存器位正是寄存器 STK_CTRL 中的 Bit2。

而寄存器位屏蔽宏：SysTick_xxx_Msk，宏展开是 xxx 的位全部置 1 后，左移 SysTick_xxx_Pos 位。如控制 SysTick 时钟源的 SysTick_CTRL_CLKSOURCE_Msk，宏展开为“1ul << SysTick_CTRL_CLKSOURCE_Pos”，把无符号长整型数值 (ul) 1 左移 2 位，得到了一个只有 Bit2：CLKSOURCE 位被置 1，其他位为 0 的数值，这样的数值配合位操作 &（按位与）、|（按位或）可以很方便地修改寄存器的某些位。假如控制 CLKSOURCE 需要 4 个寄存器位，这个宏就应该被改为 (0xf ul << SysTick_CTRL_CLKSOURCE_Pos)，这样就会得到一个关于 CLKSOURCE 的 4 位被置 1 的值，这些宏的参数就是这样被确定的。

寄存器位指示宏和位屏蔽宏在操作寄存器的代码（大部分库函数）中用得十分广泛，在前面 GPIO_Init() 函数分析时也遇到很多，为了方便以后再使用，我们就给这两类宏取了这两个名字。

3. 配置中断向量及重置 STK_VAL 寄存器

回到 SysTick_Config() 函数，接下来调用了 NVIC_SetPriority() 函数并配置了 SysTick 中断，这就是为什么我们在外部没有再使用 NVIC 配置 SysTick 中断的原因。配置好 SysTick 中断后把 STK_VAL 寄存器重新赋值为 0（在使能 SysTick 时，硬件会把存储在 STK_LOAD 寄存器中的 ticks 值加载给它）。

4. 配置 SysTick 时钟为 AHB

在这段代码最后，向 STK_CTRL 寄存器写入了 SysTick 的控制参数，配置为使用 AHB 时钟，使能计数至 0 时引起中断，使能 SysTick。执行了这行代码，SysTick 就开始运行并进行脉冲计数了。

若读者想要使用 AHB/8 作为时钟，可以调用库函数 SysTick_CLKSourceConfig() 进行修改，也可以直接对 SysTick_Config() 函数的代码进行修改。

5. 使能、关闭定时器

由于调用 SysTick_Config() 函数之后，SysTick 定时器就被开启了，但我们在初始化的时候并不希望这样，而是根据需要再开启。所以在 SysTick_Init() 函数中，调用完 SysTick_Config() 并配置好后，应先把定时器关闭了。SysTick 的开启和关闭由寄存器 STK_CTRL 的 Bit0 : ENABLE 位来控制，使用位屏蔽宏以操作寄存器的方式实现，见代码清单 12-6。

代码清单 12-6 使能、关闭定时器

```
1. // 使能滴答定时器
2. SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
3. // 失能滴答定时器
4. SysTick->CTRL &= ~ SysTick_CTRL_ENABLE_Msk;
```

12.3.5 定时时间的计算

现在回到函数 SysTick_Init()，在调用 SysTick_Config() 函数时，向它输入的参数为 SystemCoreClock / 100000，SystemCoreClock 为定义了系统时钟（SYSCLK）频率的宏，即等于 AHB 的时钟频率。在本书的所有例程中 AHB 都是被配置为 72 MHz 的，也就是这个 SystemCoreClock 宏展开为数值 7200 0000。

根据前面对 SysTick_Config() 函数的介绍，它的输入参数为 SysTick 将要计时的脉冲数，经过 ticks 个脉冲（经过 ticks 个时钟周期）后将触发中断，触发中断后又重新开始计数。

由此我们可以算出定时的时间，下面为计算公式：

$$T = \text{ticks} \times (1/f)$$

其中，T 为要定时的总时间；ticks 为 SysTick_Config() 的输入参数；1/f 即为 SysTick 使用的时钟源的时钟周期，f 为该时钟源的时钟频率，当时钟源确定后为常数。

例如：本实验例子中，使用时钟源为 AHB 时钟，其频率被配置为 72 MHz。调用函数时，把 ticks 赋值为 ticks=SystemFrequency / 10 000 = 720，表示 720 个时钟周期中断一次；1/f 是时钟周期的时间，此时 (1/f=1/72 μs)，所以最终定时总时间 T=720 × (1/72)，为 720 个时钟周期，正好是 10 μs。

SysTick 定时器的定时时间（配置为触发中断，即为中断周期）由 ticks 参数决定，最大定时周期不能超过 2²⁴ 个。以下是几种常用的中断周期配置，是根据上面的公式计算出来的。见代码清单 12-7。

代码清单 12-7 定时时间设置

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

12.3.6 编写中断服务函数

回到 main 函数，我们使 LED 工作在一个无限循环中，在 LED 的开与关之间调用了 Delay_us() 函数，见代码清单 12-8。

代码清单 12-8 使 LED 点亮和关闭的过程

```

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

```

一旦我们调用了 Delay_us() 函数, SysTick 定时器就被开启, 按照设定好的定时周期递减计数, 当 SysTick 的计数寄存器的值减为 0 时, 就进入中断函数, 当中断函数执行完毕之后重新计时, 如此循环, 除非它被关闭。Delay_us() 函数实现见代码清单 12-9。

代码清单 12-9 Delay_us() 函数

```

1. /*
2.  * 函数名: Delay_us
3.  * 描述   : us 延时程序, 10us 为一个单位
4.  * 输入   : - nTime
5.  * 输出   : 无
6.  * 调用   : Delay_us( 1 ) 则实现的延时为 1 * 10us = 10us
7.  *       : 外部调用
8.  */
9.
10. void Delay_us(__IO u32 nTime)
11. {
12.     TimingDelay = nTime;
13.
14.     // 使能滴答定时器
15.     SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
16.
17.     while(TimingDelay != 0);
18. }

```

使能了 SysTick 之后, 就使用 while (TimingDelay != 0) 语句等待 TimingDelay 变量变为 0, 这个变量是在中断服务函数中被修改的。

因此, 我们需要编写相应的中断服务程序, 在本实验室中我们配置为 10μs 中断一次, 每次中断把 TimingDelay 减 1。中断程序在 stm32f10x_it.c 中实现, 见代码清单 12-10。

代码清单 12-10 SysTick 中断服务函数

```

1. /*** @brief This function handles SysTick Handler.
2.  * @param None
3.  * @retval : None
4.  */
5. void SysTick_Handler(void)
6. {

```

```

7.     TimingDelay_Decrement();
8. }

```

SysTick 中断属于系统异常向量，在 stm32f10x_it.c 文件中已经默认有了它的中断服务函数 SysTick_Handler()，但内容为空。我们找到这个函数，其调用了用户函数 TimingDelay_Decrement()。后者是由用户编写的一个应用程序，在 SysTick.c 中实现，见代码清单 12-11。

代码清单 12-11 每次中断 TimingDelay 减 1

```

1. /*
2.  * 函数名: TimingDelay_Decrement
3.  * 描述  : 获取节拍程序
4.  * 输入  : 无
5.  * 输出  : 无
6.  * 调用  : 在 SysTick 中断函数 SysTick_Handler() 调用
7.  */
8. void TimingDelay_Decrement(void)
9. {
10.     if (TimingDelay != 0x00)
11.     {
12.         TimingDelay--;
13.     }
14. }

```

每次进入 SysTick 中断就调用一次 TimingDelay_Decrement() 函数，使全局变量 TimingDelay 自减一次。用户函数 Delay_us() 在 TimingDelay 被减至 0 时，才退出延时循环，即我们对 TimingDelay 赋的值为要中断的次数。

所以总的延时时间：

$$T_{\text{延时}} = T_{\text{中断周期}} \times \text{TimingDelay}$$

至此，SysTick 的精确延时功能讲解完毕。

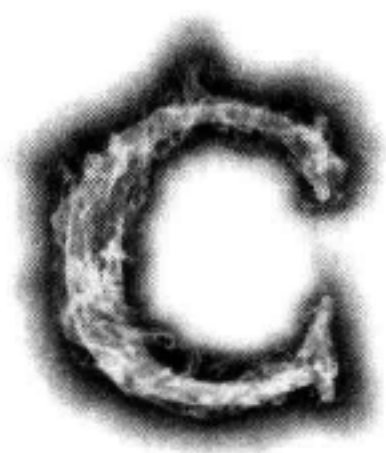
12.3.7 使用 SysTick 测量时间的功能

稍微改变一下用法，我们就可以利用 SysTick 进行时间测量。

当我们开启 SysTick 定时器后，定时器开始工作，我们可以定义一个变量 a 来对中断次数进行记录，在定时器进入中断时这个变量就 a++，当我们关闭定时器后，将变量的数值乘以定时器的中断周期就等于测量时间。一般利用该功能测量程序的运行时间，特别是涉及算法的程序，这对于优化算法有非常大的帮助。假如你的算法是 μs 级别的，那么 SysTick 就应该设定为 μs 级中断；如果是 ms 级别的，就将 SysTick 设定为 ms 级中断。

12.3.8 实验现象

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



第 13 章

STM32 定时器

13.1 定时器功能简介

区别于 SysTick 一般只用于系统时钟的计时，STM32 的定时器外设功能强大得超出了想像，《STM32 参考手册》中仅对定时器的介绍就已经占了 100 多页。STM32 一共有 8 个都为 16 位的定时器。其中 TIM6、TIM7 是基本定时器；TIM2、TIM3、TIM4、TIM5 是通用定时器；TIM1 和 TIM8 是高级定时器。这些定时器使 STM32 具有定时、信号的频率测量、信号的 PWM 测量、PWM 输出、三相 6 步电机控制及编码器接口等功能，都是专门为工控领域量身定做的。

13.2 定时器工作分析

13.2.1 基本定时器

基本定时器 TIM6 和 TIM7 只具备最基本的定时功能，就是累加的时钟脉冲数超过预定值时，能触发中断或触发 DMA 请求。由于在芯片内部与 DAC 外设相连，可通过触发输出驱动 DAC，也可以作为其他通用定时器的时钟基准。见图 13-1。

这两个基本定时器使用的时钟源都是 TIMxCLK，时钟源经过 PSC 预分频器输入至脉冲计数器 TIMx_CNT，基本定时器只能工作在向上计数模式，在重载寄存器 TIMx_ARR 中保存的是定时器的溢出值。

工作时，脉冲计数器 TIMx_CNT 由时钟触发进行计数，当 TIMx_CNT 的计数值 X 等于重载寄存器 TIMx_ARR 中保存的数值 N 时，产生溢出事件，可触发中断或 DMA 请求。然后 TIMx_CNT 的值重新被置为 0，重新向上计数。

13.2.2 通用定时器

相比之下，通用定时器 TIM2 ~ TIM5 就比基本定时器复杂得多了。除了基本的定时，它主要用在测量输入脉冲的频率、脉冲宽与输出 PWM 脉冲的场合，还具有编码器的接口。见图 13-5。

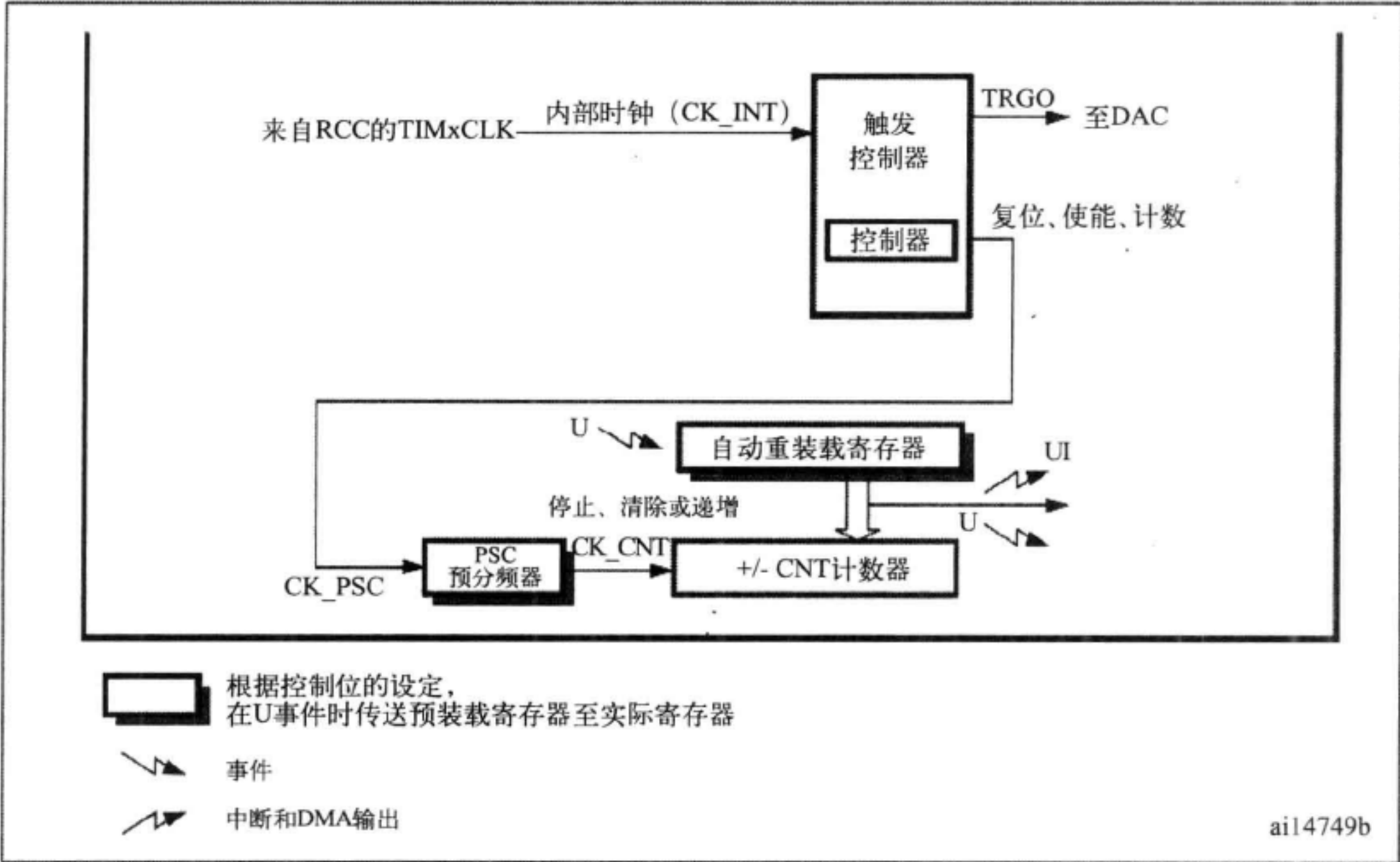


图 13-1 基本定时器结构

1. 捕获 / 比较寄存器

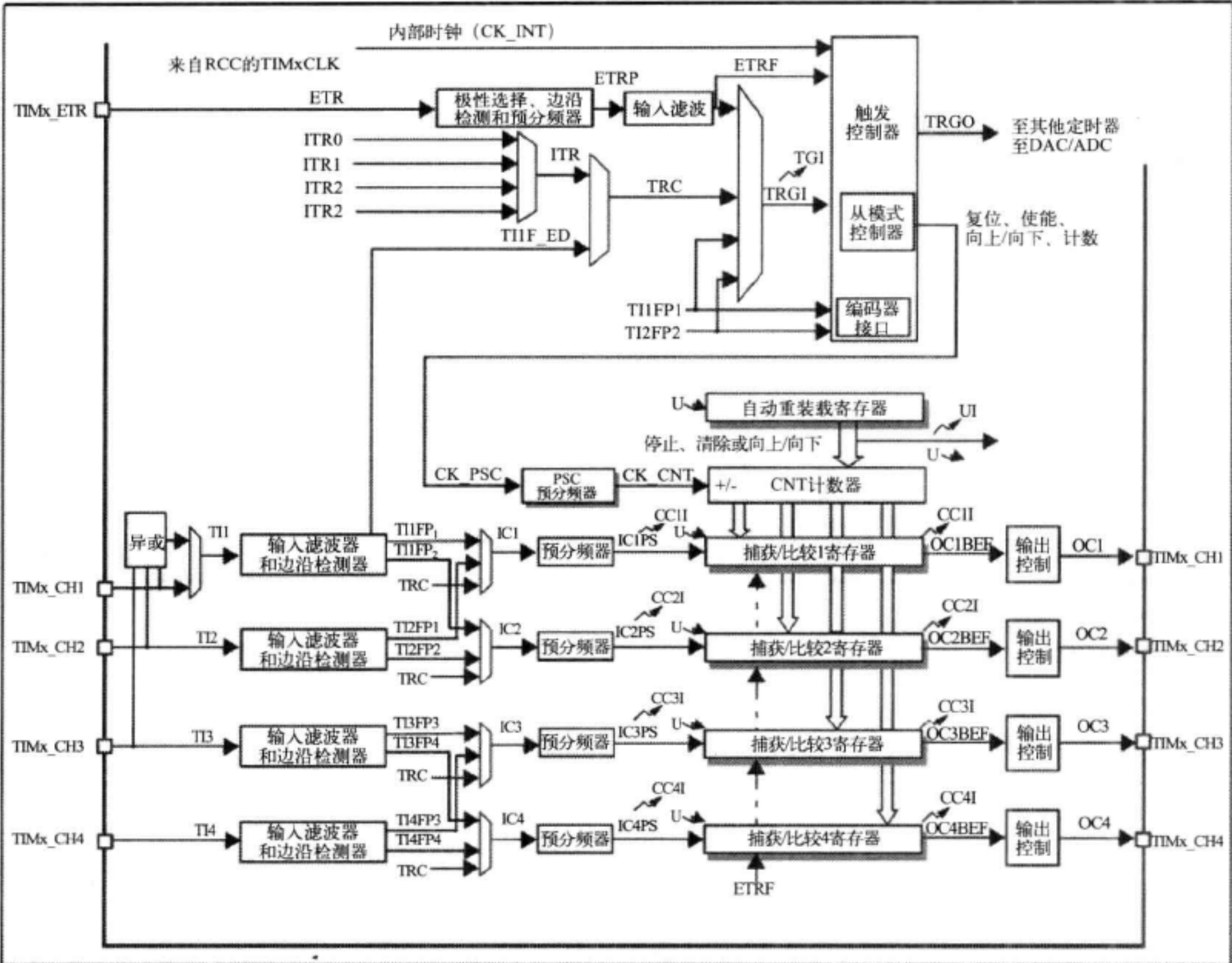
通用定时器的基本计时功能与基本定时器的工作方式是一样的，同样把时钟源经过预分频器输出到脉冲计数器 TIMx_CNT 累加，溢出时就产生中断或 DMA 请求。

而通用定时器比基本定时器多出的强大功能，就是因为通用定时器多出了一种寄存器——捕获 / 比较寄存器 TIMx_CCR (capture/compare register)，它在输入时被用于捕获（存储）输入脉冲在电平发生翻转时脉冲计数器 TIMx_CNT 的当前计数值，从而实现脉冲的频率测量；在输出时被用来存储一个脉冲数值，把这个数值用于与脉冲计数器 TIMx_CNT 的当前计数值进行比较，根据比较结果进行不同的电平输出。

2. PWM 输出过程分析

通用定时器可以利用 GPIO 引脚进行脉冲输出，在配置为比较输出、PWM 输出功能时，捕获 / 比较寄存器 TIMx_CCR 被用作比较功能，下面把它简称为比较寄存器。

这里直接举例说明定时器的 PWM 输出工作过程：若配置脉冲计数器 TIMx_CNT 为向上计数，而重载寄存器 TIMx_ARR 被配置为 N，即 TIMx_CNT 的当前计数值数值 X 在 TIMxCLK 时钟源的驱动下不断累加，当 TIMx_CNT 的数值 X 大于 N 时，会重置 TIMx_CNT 数值为 0 并重新计数。






注：  根据控制位的设定，在 U 事件时传送预加载寄存器的内容至工作寄存器
 事件
 中断和 DMA 输出

图 13-2 通用定时器结构

而在 TIMx_CNT 计数的同时，TIMx_CNT 的计数值 X 会与比较寄存器 TIMx_CCR 预先存储的数值 A 进行比较。当脉冲计数器 TIMx_CNT 的数值 X 小于比较寄存器 TIMx_CCR 的值 A 时，输出高电平（或低电平）；相反地，当脉冲计数器的数值 X 大于或等于比较寄存器的值 A 时，输出低电平（或高电平）。

如此循环，得到的输出脉冲周期就为重载寄存器 TIMx_ARR 存储的数值 (N+1) 乘以触发脉冲的时钟周期，其脉冲宽度则为比较寄存器 TIMx_CCR 的值 A 乘以触发脉冲的时钟周期，即输出 PWM 的占空比为 $A/(N+1)$ 。

见图 13-3，图中为重载寄存器 TIMx_ARR 被配置为 N=8，向上计数；比较寄存器 TIMx_CCR 的值被设置为 4、8、大于 8、等于 0 时的输出时序图。图中 OCxREF 即为 GPIO 引脚的输出时序、CCxIF 为触发中断的时序。

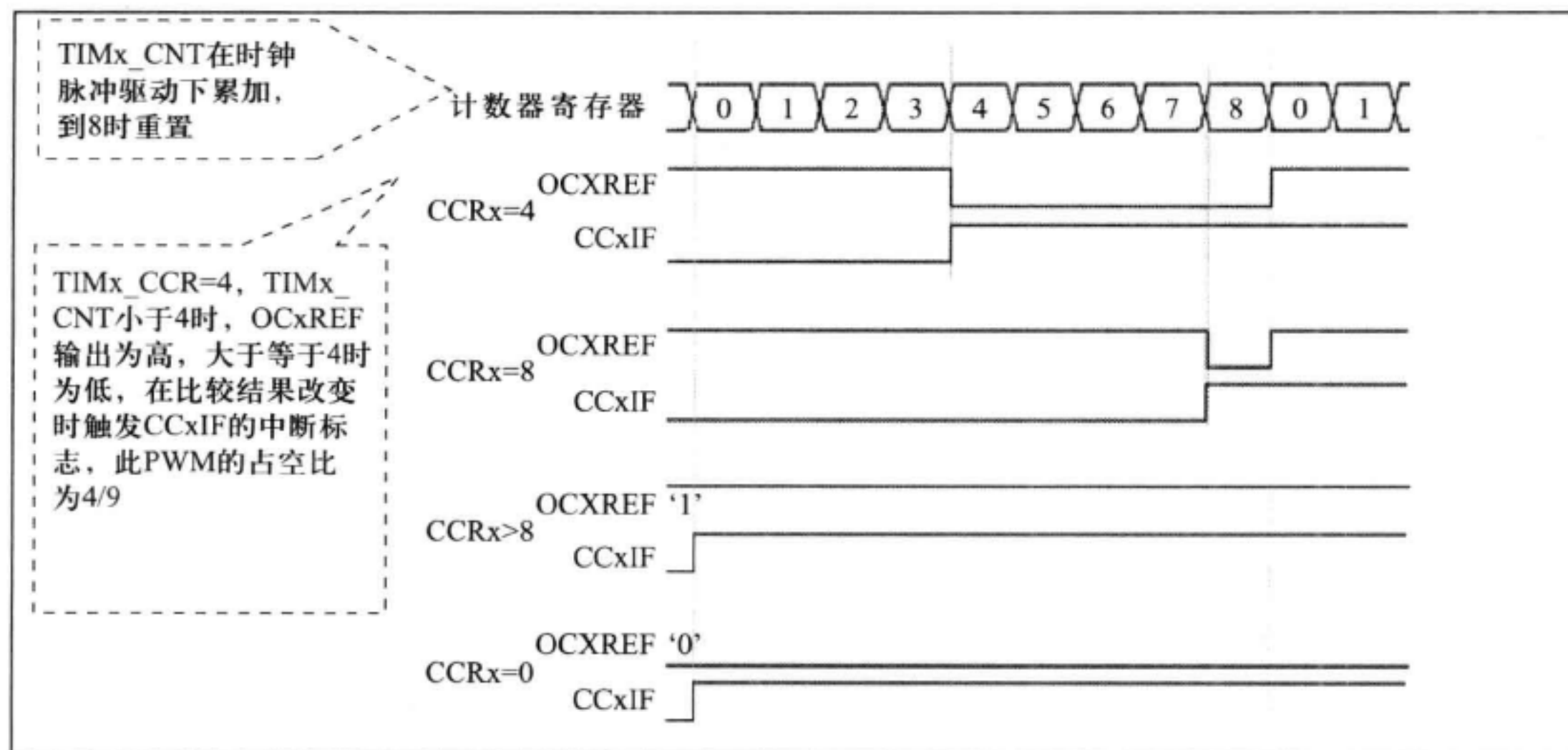


图 13-3 PWM 输出模式

3. PWM 输入过程分析

而当定时器被配置为输入功能时，可以用于检测输入到 GPIO 引脚的信号（频率检测、输入 PWM 检测），此时捕获 / 比较寄存器 TIMx_CCR 被用作捕获功能，下面把它简称为捕获寄存器。

见图 13-4，为 PWM 输入时的脉冲宽检测时序图。

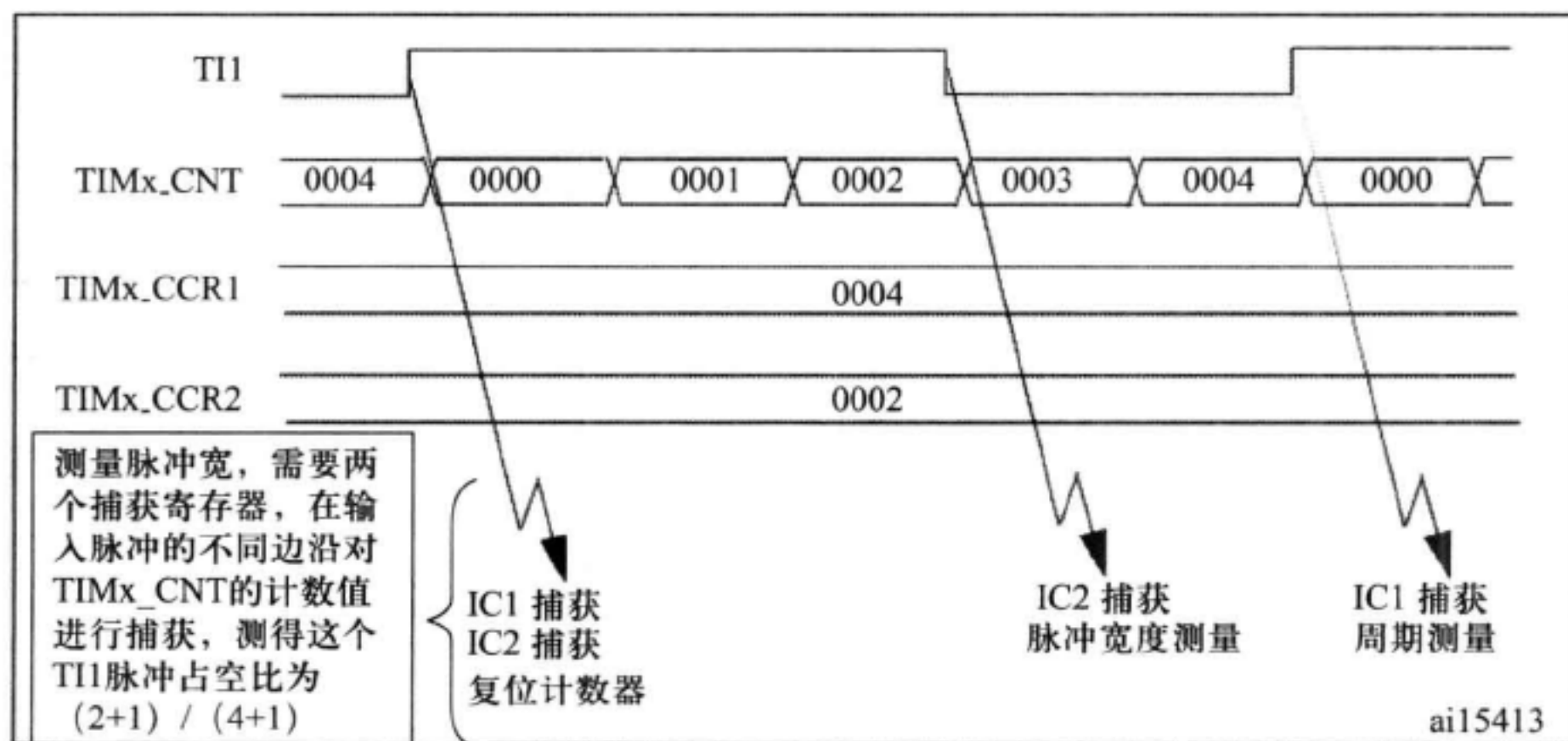


图 13-4 PWM 输入检测

按照如图 13-4 所示时序图来分析 PWM 输入脉冲宽检测的工作过程：要测量的 PWM 脉冲通过 GPIO 引脚输入到定时器的脉冲检测通道，其时序为图中的 TI1。把脉冲计数器 TIMx_CNT 配置为向上计数，重载寄存器 TIMx_ARR 的 N 值配置为足够大。

在输入脉冲 TI1 的上升沿到达时，触发 IC1 和 IC2 输入捕获中断，这时把脉冲计数器 TIMx_CNT 的计数值复位为 0，于是 TIMx_CNT 的计数值 X 在 TIMxCLK 的驱动下从 0 开始不断累加，直到 TI1 出现下降沿，触发 IC2 捕获事件，此时捕获寄存器 TIMx_CCR2 把脉冲计数器 TIMx_CNT 的当前值 2 存储起来，而 TIMx_CNT 继续累加，直到 TI1 出现第二个上升沿，触发了 IC1 捕获事件，此时 TIMx_CNT 的当前计数值 4 被保存到 TIMx_CCR1。

很明显 TIMx_CCR1 (加 1) 的值乘以 TIMxCLK 的周期，即为待检测的 PWM 输入脉冲周期，TIMx_CCR2 (加 1) 的值乘以 TIMxCLK 的周期，就是待检测的 PWM 输入脉冲的高电平时间，有了这两个数值就可以计算出 PWM 脉冲的频率、占空比了。

可以看出，正因为捕获 / 比较寄存器的存在，才使得通用定时器变得如此强大。

4. 定时器的时钟源

从时钟源方面来说，通用定时器比基本定时器多了一个选择，它可以使用外部脉冲作为定时器的时钟源。使用外部时钟源时，要使用寄存器进行触发边沿、滤波器带宽的配置。如果选择内部时钟源的话则与基本定时器一样，也为 TIMxCLK。但要注意的是，所有定时器（包括基本、通用和高级）使用内部时钟时，定时器的时钟源都被称为 TIMxCLK，但 TIMxCLK 的时钟来源并不是完全一样的，见图 13-5。

TIM2 ~ 7 也就是基本定时器和通用定时器，TIMxCLK 的时钟来源是 APB1 预分频器的输出。当 APB1 的分频系数为 1 时，则 TIM2 ~ 7 的 TIMxCLK 直接等于该 APB1 预分频器的输出，而 APB1 的分频系数不为 1 时，TIM2 ~ 7 的 TIMxCLK 则为 APB1 预分频器输出的 2 倍。

如在常见的配置中，AHB=72 MHz，而 APB1 预分频器的分频系数被配置为 2，则 PCLK1 刚好达到最大值 36 MHz，而此时 APB1 的分频系数不为 1，则 TIM2 ~ TIM7 的时钟 $TIMxCLK = (AHB/2) \times 2 = 72 \text{ MHz}$ 。

而对于 TIM1 和 TIM8 这两个高级定时器，TIMxCLK 的时钟来源则是 APB2 预分频器的输出，同样它 also 根据分频系数分为两种情况。

常见的配置中 AHB=72 MHz，APB2 预分频器的分频系数被配置为 1，此时 PCLK2 刚好达到最大值 72 MHz，而 TIMxCLK 则直接等于 APB2 分频器的输出，即 TIM1 和 TIM8 的时钟 $TIMxCLK = AHB = 72 \text{ MHz}$ 。

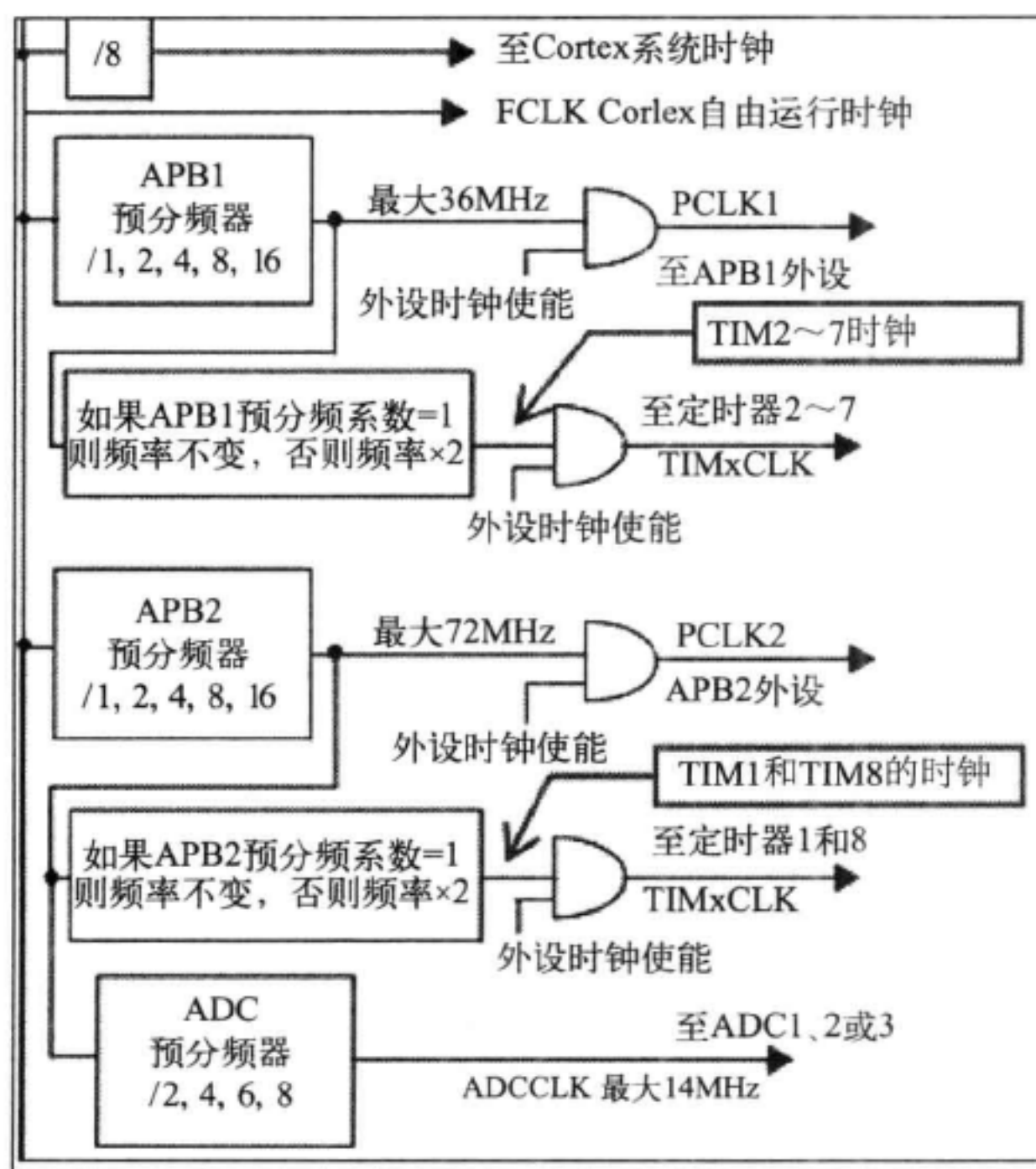


图 13-5 时钟树 (TIMxCLK 部分)

虽然这种配置下最终 TIMxCLK 的时钟频率相等,但必须清楚实质上它们的时钟来源是有区别的。还要强调的是:TIMxCLK 是定时器内部的时钟源,但在时钟输出到脉冲计数器 TIMx_CNT 前,还经过一个预分频器 PSC,最终用于驱动脉冲计数器 TIMx_CNT 的时钟频率根据预分频器 PSC 的配置而定。

13.2.3 高级定时器

TIM1 和 TIM8 是两个高级定时器,它们具有基本、通用定时器的所有功能,还具有三相 6 步电机的接口、刹车功能 (break function) 及用于 PWM 驱动电路的死区时间控制等,使得它非常适合于电机的控制。如图 13-6 所示为高级定时器结构。

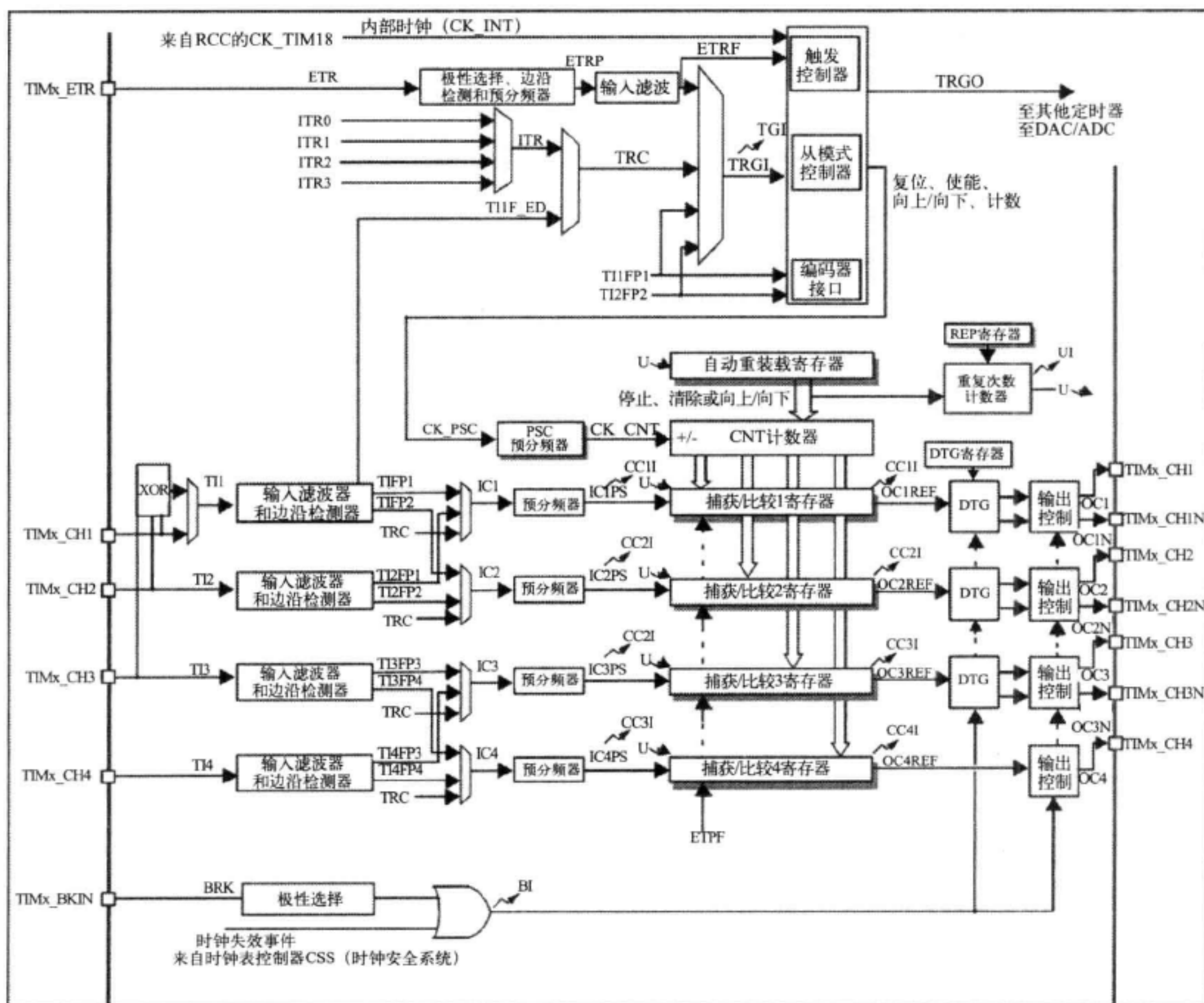


图 13-6 高级定时器结构

见图 13-6,相比于通用定时器,主要多出了 BRK、DTG 两个结构,因而具有了死区时间的控制功能。

首先，死区时间是什么呢？在 H 桥、三相桥的 PWM 驱动电路中，上下两个桥臂的 PWM 驱动信号是互补的，即上下桥臂轮流导通，但实际上为了防止出现上下两个臂同时导通（会造成短路），在上下两臂切换时留一小段时间，上下臂都施加关断信号，这个上下臂都关断的时间称为死区时间。

STM32 的高级定时器可以配置出输出互补的 PWM 信号，并且在这个 PWM 信号中加入死区时间，为电机的控制提供了极大的便利。见图 13-7。图中的 OCxREF 为参考信号（可理解为原信号），OCx 和 OCxN 为定时器通过 GPIO 引脚输出的 PWM 互补信号。

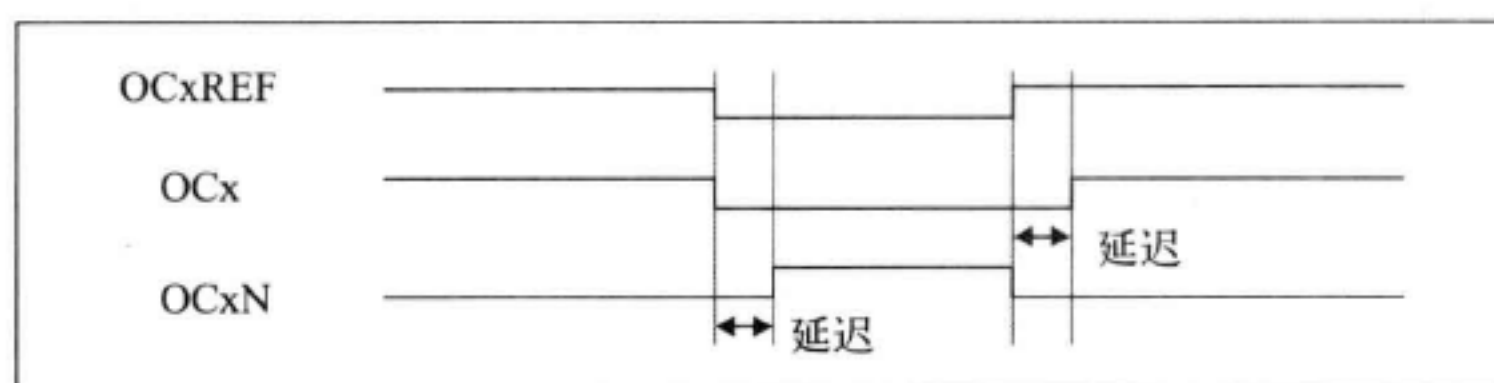


图 13-7 插入死区时间

若不加入死区时间，当 OCxREF 出现下降沿，OCx 同时输出下降沿，OCxN 则同时输出相反的上升沿，即这三个信号的跳变是同时的。

加入死区时间后，当 OCxREF 出现下降沿，OCx 同时输出下降沿，但 OCxN 则过了一小段延迟再输出上升沿，OCxREF 出现上升沿后，OCx 要经过一段延时再输出上升沿。假如 OCx、OCxN 分别控制上、下桥臂，有了延迟后，就不容易出现上、下桥臂同时导通的情况。这个延迟时间与 PWM 信号驱动的电子器件特性相关，从事工控领域的读者对此应该比较熟悉。

在保证不出现短路的情况下，死区时间越短越好。见图 13-8、图 13-9。

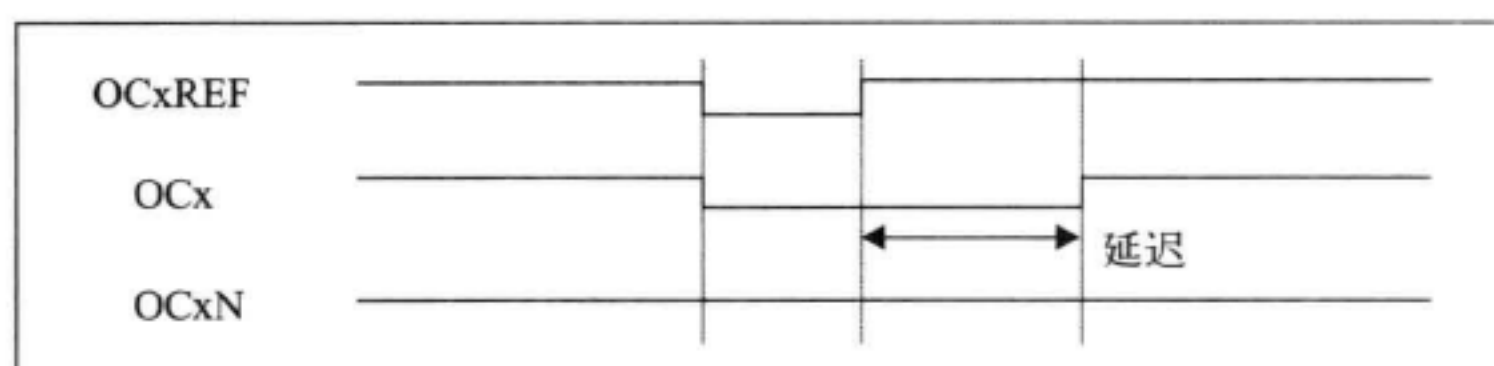


图 13-8 死区时间太长，OCxN 输出不正常

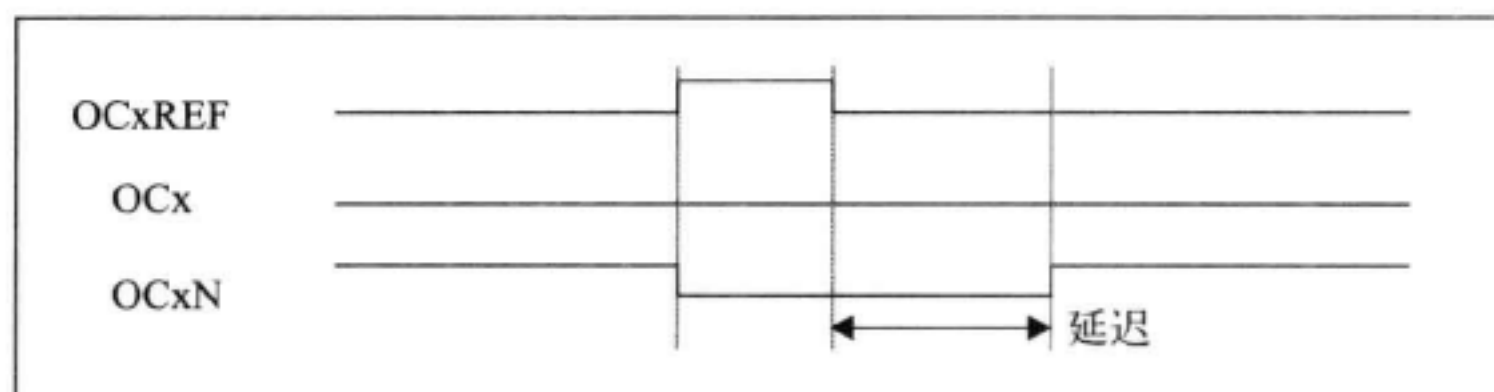


图 13-9 死区时间太长，OCx 输出不正常

13.3 PWM 输出实例分析

PWM 在电力电子技术中占据着重要的地位，被广泛地应用在逆变电路之中。利用 STM32 定时器的 PWM 输出功能，可以直接获取 PWM 波。根据面积等效原理，利用规则采样法、查表法可以调制出 SPWM 波及各种调制 PWM 波形。

本实验向大家讲解如何输出占空比固定的 PWM 波形。

13.3.1 实验描述及工程文件清单

1. 实验描述

通用定时器 TIM3 产生 4 路不同占空比的 PWM 波。

- ☐ TIM3 Channel1 duty cycle = $(\text{TIM3_CCR1} / \text{TIM3_ARR}) * 100 = 50\%$
- ☐ TIM3 Channel2 duty cycle = $(\text{TIM3_CCR2} / \text{TIM3_ARR}) * 100 = 37.5\%$
- ☐ TIM3 Channel3 duty cycle = $(\text{TIM3_CCR3} / \text{TIM3_ARR}) * 100 = 25\%$
- ☐ TIM3 Channel4 duty cycle = $(\text{TIM3_CCR4} / \text{TIM3_ARR}) * 100 = 12.5\%$

2. 硬件连接

- ☐ PA.06 : (TIM3_CH1)
- ☐ PA.07 : (TIM3_CH2)
- ☐ PB.00 : (TIM3_CH3)
- ☐ PB.01 : (TIM3_CH4)

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_tim.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/pwm_output.c

13.3.2 配置工程环境

本定时器 PWM 输出实验中我们用到了 GPIO、RCC、TIM 外设，没有使用中断，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_tim.c。

新建 pwm_output.c 及 pwm_output.h 文件，并在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 13-1。

代码清单 13-1 PWM 例程中的 stm32f10x_conf.h 文件配置

```

1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****/
9. #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_tim.h"

```

13.3.3 main 文件

我们先从 main 文件开始看本实验的执行流程，见代码清单 13-2。

代码清单 13-2 PWM 例程的 main 函数

```

1. #include "stm32f10x.h"
2. #include "pwm_output.h"
3.
4. /*
5.  * 函数名: main
6.  * 描述  : 主函数
7.  * 输入  : 无
8.  * 输出  : 无
9.  */
10. int main(void)
11. {
12.
13.
14.     /* TIM3 PWM 波输出初始化，并使能 TIM3 PWM 输出 */
15.     TIM3_PWM_Init();
16.
17.     while (1)
18.     {}
19. }
20.
21.
22. /***** (C) COPYRIGHT 2012 WildFire Team *****/END OF FILE*****/

```

我们看到，代码的执行流程十分简单，调用用户函数 TIM3_PWM_Init() 把 TIM 初始化成 PWM 输出模式后，内核就把所有工作都交给 TIM 外设，完全由 TIM 来控制 GPIO 引脚输出 PWM 波。

13.3.4 定时器初始化

接下来看看 TIM3_PWM_Init() 函数是怎样配置 TIM 外设的，它是在 pwm_output.c 文件中实现的，见代码清单 13-3。

代码清单 13-3 TIM3_PWM_Init() 函数

```

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

首先调用了 TIM3_GPIO_Config() 对作为 TIM 外设通道复用的 GPIO 引脚进行初始化，再用 TIM3_Mode_Config() 对 TIM 外设进行初始化。

1. GPIO 初始化

分析其中 TIM3_GPIO_Config() 代码，见代码清单 13-4。

代码清单 13-4 TIM3_GPIO_Config() 函数

```

1. /*
2.  * 函数名: TIM3_GPIO_Config
3.  * 描述   : 配置 TIM3 复用输出 PWM 时用到的 I/O
4.  * 输入    : 无
5.  * 输出    : 无
6.  * 调用    : 内部调用
7.  */
8. static void TIM3_GPIO_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* TIM3 clock enable */
13.     //PCLK1 经过 2 倍频后作为 TIM3 的时钟源等于 72MHz
14.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
15.
16.     /* GPIOA and GPIOB clock enable */
17.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB, ENABLE);
18.
19.     /*GPIOA Configuration: TIM3 channel 1 and 2 as alternate function push-pull */
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
21.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; // 复用推挽输出
22.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

```
23.  
24.  GPIO_Init(GPIOA, &GPIO_InitStructure);  
25.  
26.  /*GPIOB Configuration: TIM3 channel 3 and 4 as alternate function push-pull */  
27.  GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_0 | GPIO_Pin_1;  
28.  
29.  GPIO_Init(GPIOB, &GPIO_InitStructure);  
30.)
```

在 TIM3_GPIO_Config() 函数中，我们使能了 TIM3 外设的时钟，并对 TIM3 通道相应的 GPIO 引脚做了相应的配置，使能 GPIO 时钟，分别是 PA6、PA7、PB0 和 PB1。它们被配置为复用输出，翻转频率为 50 MHz。

同样，关于 TIM3 通道的 GPIO 引脚映射可以在《STM32 数据手册》的引脚定义表找到，GPIO 复用模式配置可从《STM32 参考手册》的 GPIO 章节找到。见表 13-1 及表 13-2。

表 13-1 TIM 外设 GPIO 引脚定义（部分）

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCSP64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
L3	J3	G5	22	31	42	PA6	I/O		PA6	SPI1_MISO/TIM8_BKIN ADC12_IN6/TIM3_CH1	TIM1_BKIN
M3	K3	G4	23	32	43	PA7	I/O		PA7	SPI1_MOSI/TIM8_CH1N ADC12_IN7/TIM3_CH2	TIM1_CH1N
C6	B5	B5	58	92	136	PB6	I/O	FT	PB6	I2C1_SCL/TIM4_CH1	USART1_TX
D6	A5	C5	59	93	137	PB7	I/O	FT	PB7	I2C1_SDA/FSMC_NADV TIM4_CH2	USART1_RX

表 13-2 TIM 的 GPIO 模式配置

TIM2/3/4/5引脚	配 置	GPIO配置
TIM2/3/4/5_CHx	输入捕获通道 x	浮空输入
	输出比较通道 x	推挽复用输出
TIM2/3/4/5_ETR	外部触发时钟输入	浮空输入

2. 时基初始化

配置好 GPIO 后，再来看看 TIM3_Mode_Config() 函数是如何配置 TIM 模式的，函数的代码有点多，但主要是因为配置了 4 个通道，见代码清单 13-5。

代码清单 13-5 TIM3_Mode_Config() 函数

```

1.  /*
2.  * 函数名: TIM3_Mode_Config
3.  * 描述 : 配置 TIM3 输出的 PWM 信号的模式, 如周期、极性、占空比
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 内部调用
7.  */
8. static void TIM3_Mode_Config(void)
9. {
10.     TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
11.     TIM_OCInitTypeDef TIM_OCInitStructure;
12.
13.     /* PWM 信号电平跳变值 */
14.     u16 CCR1_Val = 500;
15.     u16 CCR2_Val = 375;
16.     u16 CCR3_Val = 250;
17.     u16 CCR4_Val = 125;
18.
19. /* -----
20.     TIM3 Configuration: generate 4 PWM signals with 4 different duty cycles:
21.     TIM3CLK = 72 MHz, Prescaler = 0x0, TIM3 counter clock = 72 MHz
22.     TIM3 ARR Register = 999 => TIM3 Frequency = TIM3 counter clock/(ARR + 1)
23.     TIM3 Frequency = 72 kHz.
24.     TIM3 Channell duty cycle = (TIM3_CCR1/ TIM3_ARR) * 100 = 50%
25.     TIM3 Channel2 duty cycle = (TIM3_CCR2/ TIM3_ARR) * 100 = 37.5%
26.     TIM3 Channel3 duty cycle = (TIM3_CCR3/ TIM3_ARR) * 100 = 25%
27.     TIM3 Channel4 duty cycle = (TIM3_CCR4/ TIM3_ARR) * 100 = 12.5%
28. ----- */
29.
30.     /* Time base configuration */
31.     TIM_TimeBaseStructure.TIM_Period = 999;
32. // 当定时器从 0 计数到 999, 即为 1000 次, 为一个定时周期
33.     TIM_TimeBaseStructure.TIM_Prescaler = 0;
34. // 设置预分频: 不预分频, 即为 72MHz
35.     TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1 ;
36. // 设置时钟分频系数: 不分频
37.     TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
38. // 向上计数模式
39.
40.     TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
41.
42.     /* PWM1 Mode configuration: Channell */
43.     TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
44. // 配置为 PWM 模式 1
45.     TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
46.     TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
47. // 设置跳变值, 当计数器计数到这个值时, 电平发生跳变
48.     TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
49. // 当定时器计数值小于 CCR1_Val 时为高电平
50.

```

```

51. TIM_OC1Init(TIM3, &TIM_OCInitStructure);    // 使能通道 1
52.
53. TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);
54.
55. /* PWM1 Mode configuration: Channel2 */
56. TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
57. TIM_OCInitStructure.TIM_Pulse = CCR2_Val;
58. // 设置通道 2 的电平跳变值, 输出另外一个占空比的 PWM
59.
60. TIM_OC2Init(TIM3, &TIM_OCInitStructure);    // 使能通道 2
61.
62. TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);
63.
64. /* PWM1 Mode configuration: Channel3 */
65. TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
66. TIM_OCInitStructure.TIM_Pulse = CCR3_Val;
67. // 设置通道 3 的电平跳变值, 输出另外一个占空比的 PWM
68.
69. TIM_OC3Init(TIM3, &TIM_OCInitStructure);    // 使能通道 3
70.
71. TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);
72.
73. /* PWM1 Mode configuration: Channel4 */
74. TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
75. TIM_OCInitStructure.TIM_Pulse = CCR4_Val;
76. // 设置通道 4 的电平跳变值, 输出另外一个占空比的 PWM
77.
78. TIM_OC4Init(TIM3, &TIM_OCInitStructure);    // 使能通道 4
79.
80. TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Enable);
81.
82. TIM_ARRPreloadConfig(TIM3, ENABLE);
83. // 使能 TIM3 重载寄存器 ARR
84.
85. /* TIM3 enable counter */
86. TIM_Cmd(TIM3, ENABLE);                      // 使能定时器 3
87. }

```

TIM3_Mode_Config() 函数被分为三个部分, 分别为时基初始化、输出模式初始化和装载捕获/比较寄存器的数值。

首先是时基初始化, 使用了一个 TIM_TimeBaseInitTypeDef 类型的结构体。时基初始化即配置基本定时器只具有的那部分功能, 下面分析初始化结构体的成员:

1) .TIM_Period: 定时周期, 实质是存储到重载寄存器 TIMx_ARR 的数值, 脉冲计数器从 0 累加到这个值上溢或从这个值自减至 0 下溢。这个数值加 1 然后乘以时钟源周期就是实际定时周期。本实验中向该成员赋值为 999, 即定时周期为 $(999+1) \times T$, T 为时钟源周期。

2) .TIM_Prescaler: 对定时器时钟 TIMxCLK 的预分频值, 分频后作为脉冲计数器 TIMx_CNT 的驱动时钟, 得到脉冲计数器的时钟频率为: $f_{CK_CNT} = f_{TIMxCLK} / (N+1)$, 其中 N 即为赋给本成员的时钟分频值。本实验给 .TIM_Prescaler 成员赋值为 0, 即不对 TIMxCLK 分频, 已知 AHB 时

钟频率为 72 MHz、TIMxCLK 为 72 MHz，所以输出到脉冲计数器 TIMx_CNT 的时钟频率为 $f_{CK_CNT}=72\text{ MHz}/1=72\text{ MHz}$ 。

3) .TIM_ClockDivision：时钟分频因子。怎么又出现一个配置时钟分频的呢？要注意这个 TIM_ClockDivision 与上面的 TIM_Prescaler 是不一样的。TIM_Prescaler 预分频配置是对 TIMxCLK 进行分频，分频后的时钟被输出到脉冲计数器 TIMx_CNT 中，而 TIM_ClockDivision 虽然也是对 TIMxCLK 进行分频，但它分频后的时钟频率为 f_{DTS} ，是被输出到定时器的 ETRP 数字滤波器部分，会影响滤波器的采样频率。TIM_ClockDivision 可以被配置为 1 分频 ($f_{DTS}=f_{TIMxCLK}$)、2 分频及 4 分频。ETRP 数字滤波器的作用是对外部时钟 TIMxETR 进行滤波。本实验中是使用内部时钟 TIMxCLK 作为定时器时钟源的，所以配置 TIM_ClockDivision 为任何数值都没有影响。

4) .TIM_CounterMode：本成员配置的为脉冲计数器 TIMx_CNT 的计数模式，分别为向上计数、向下计数及中央对齐模式。向上计数即 TIMx_CNT 从 0 向上累加到 TIM_Period 中的值（重载寄存器 TIMx_ARR 的值），产生上溢事件；向下计数则 TIMx_CNT 从 TIM_Period 的值累减至 0，产生下溢事件。而中央对齐模式则为向上、向下计数的合体，TIMx_CNT 从 0 累加到 TIM_Period 的值减 1 时，产生一个上溢事件，然后向下计数到 1 时，产生一个计数器下溢事件，再从 0 开始重新计数。本实验中 .TIM_CounterMode 成员被赋值为 TIM_CounterMode_Up（向上计数模式）。

填充完配置参数后，调用库函数 TIM_TimeBaseInit() 把这些控制参数写到寄存器中，定时器的时基配置就完成了。

3. 输出模式配置

通用定时器的输出模式由 TIM_OCInitTypeDef 类型结构体的以下几个成员来配置，本实验未介绍使用 TIM1 或 TIM8 还有其他成员。

1) .TIM_OCMode：输出模式配置，主要使用的为 PWM1 和 PWM2 模式。

PWM1 模式是：在向上计数时，当 $TIMx_CNT < TIMx_CCRN$ （比较寄存器，其数值等于 TIM_Pulse 成员的内容）时，通道 n 输出为有效电平，否则为无效电平；在向下计数时，当 $TIMx_CNT > TIMx_CCRN$ 时通道 n 为无效电平，否则为有效电平。PWM2 模式与 PWM1 模式相反。其中有效电平和无效电平并不是固定地对应高电平和低电平，也是需要配置的，由下面介绍的 .TIM_OCPolarity 成员配置。本实验中使用 PWM1 输出模式。

2) .TIM_OutputState：配置输出模式的状态，使能或关闭输出。本实验中向该成员赋值为 TIM_OutputState_Enable（使能输出）。

3) .TIM_OCPolarity：有效电平的极性，把 PWM 模式中的有效电平设置为高电平或低电平。本实验中向该成员赋值为 TIM_OCPolarity_High（有效电平为高电平），因为上面把输出模式配置为 PWM1 模式，向上计数，所以在 $TIMx_CNT < TIMx_CCRN$ 时，通道 n 输出为高电平，否则为低电平。

4) .TIM_Pulse：直译为跳动，本成员的参数值即为比较寄存器 TIMx_CCR 的数值，当脉冲计数器 TIMx_CNT 与 TIMx_CCR 的比较结果发生变化时，输出脉冲将发生跳变。

本实验中向 1、2、3、4 通道的该成员分别赋值为 500、375、250、125。而定时器向上计数、PWM1 模式、有效电平为高，定时周期为 1000（.TIM_Period=999），所以当 TIMx_CNT 计时值小于

TIM_Pulse 值时, 输出高电平, 否则为低电平, 即各通道输出 PWM 的占空比为 $D = \text{TIM_Pulse} / (\text{TIM_Period} + 1)$, 即分别为 50%、37.5%、25%、12.5%。(注: 上面占空比的计算公式仅针对本实验的配置。)

填充完输出模式初始化结构体后, 要调用输出模式初始化函数 TIM_OCxInit() 对各个通道进行初始化, 其中 x 表示定时器的通道。如 TIM_OC1Init() 用来初始化定时器的通道 1, TIM_OC2Init() 用来初始化定时器的通道 2, 在调用各个通道的初始化函数前, 需要对初始化结构体的 .TIM_Pulse 成员重新赋值, 这是因为本实验中其他成员的配置都一样, 而占空比不同。

实验中还用 TIM_OCxPreloadConfig() 配置了各通道的比较寄存器 TIM_CCR 预装载使能; 使用 TIM_ARRPreloadConfig() 把重载寄存器 TIMx_ARR 使能, 最后用 TIM_Cmd() 使能定时器 TIM3, 定时器外设就开始工作了。

由于定时器需要配置的参数较多, 下面为大家总结一下:

- ☐ 设定 TIM 信号周期
- ☐ 设定 TIM 预分频值 (TIM_Prescaler)
- ☐ 设定 TIM 分频系数 (TIM_ClockDivision)
- ☐ 设定 TIM 计数模式
- ☐ 根据 TIM_TimeBaseInitStruct 这个结构体里面的值初始化 TIM
- ☐ 设定 TIM 的 OC 模式
- ☐ TIM 输出使能
- ☐ 设定电平跳变值
- ☐ 设定 PWM 信号的极性
- ☐ 使能 TIM 信号通道
- ☐ 使能 TIM 比较寄存器 CCRx 重载
- ☐ 使能 TIM 重载寄存器 ARR
- ☐ 使能 TIM 计数器

本实验中设置输出的为占空比固定的 PWM 波, 若想利用定时器输出 SPWM 波, 则要不断改变 PWM 的占空比, 我们可以利用库函数 TIM_SetCompare() 查表修改比较寄存器中的值 (即脉冲宽度) 来达到目的。

13.3.5 实验现象

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

以前我们都是通过 J_LINK 直接将我们的代码烧到开发板的 Flash 中去调试, 现在要换成软件仿真, 得首先设置一下我们的开发环境, 按照如下步骤所示:

- 1) 点击 Target Options 选项图标, 见图 13-10。
- 2) 选中 Debug 选项卡, 见图 13-11。

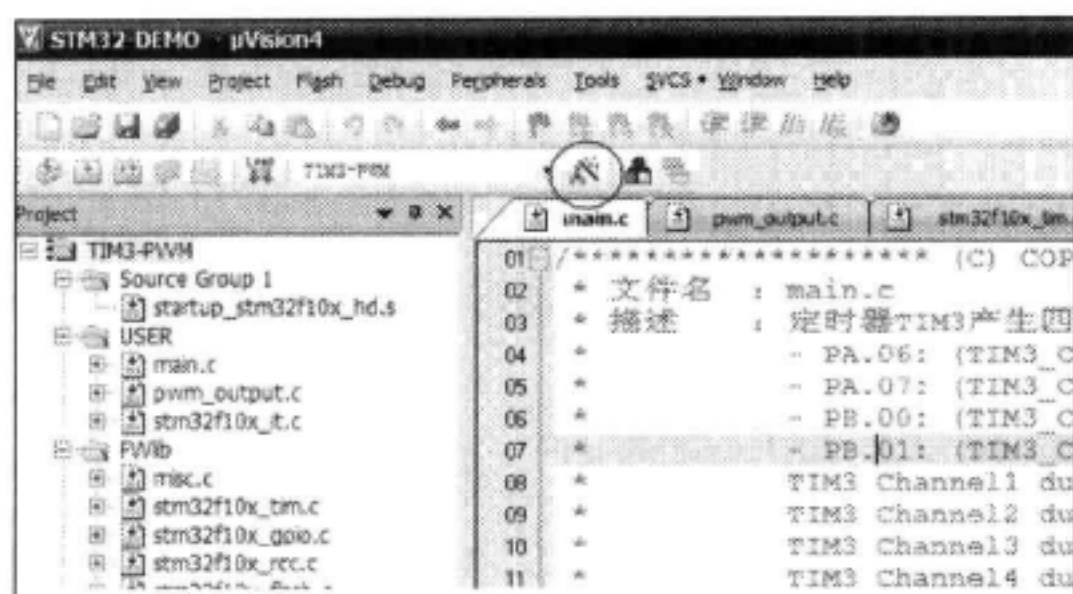


图 13-10 打开 Target Option 设置

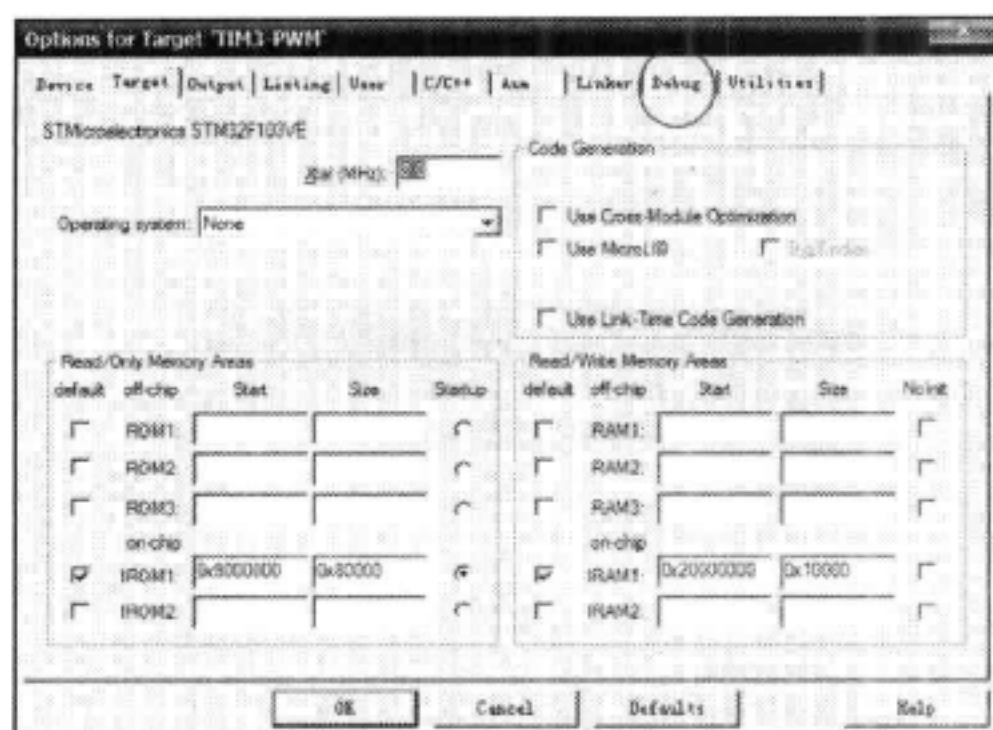


图 13-11 Debug 选项卡

3) 选中 Use Simulator 选项, 然后点击“OK”按钮即可, 见图 13-12。

下面我们开始进行软件仿真, 按照如下步骤进行:

1) 点击 Start/Stop Debug Session 选项图标, 见图 13-13。

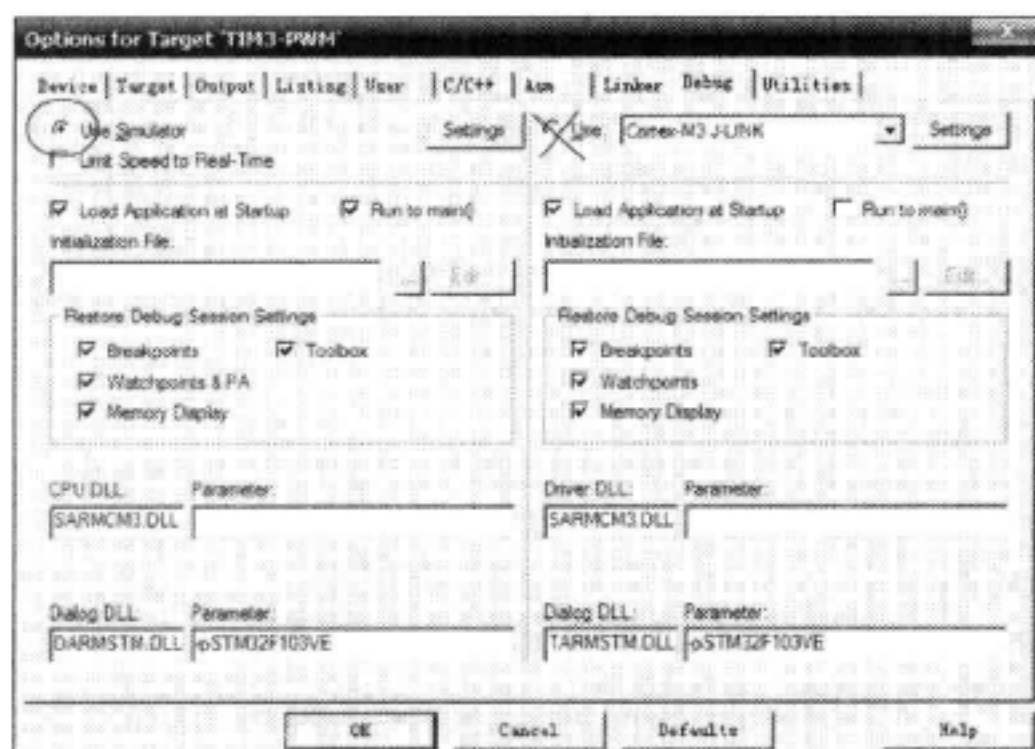


图 13-12 选择软件仿真

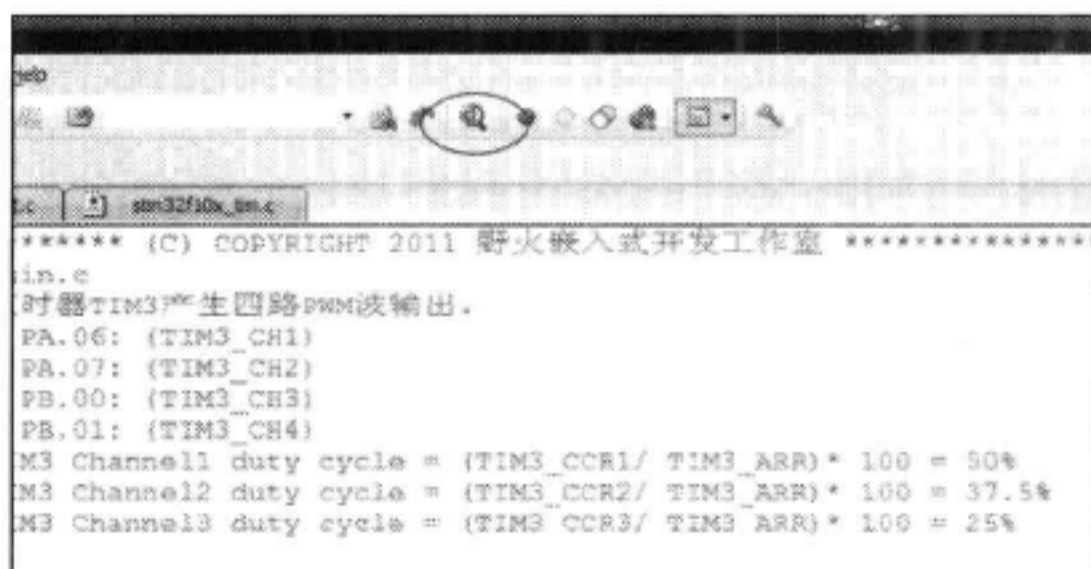


图 13-13 点击 Start Debug Session 图标

2) 点击 Analysis Windows 选项图标, 见图 13-14。

3) 点击 Setup 选项卡, 见图 13-15。

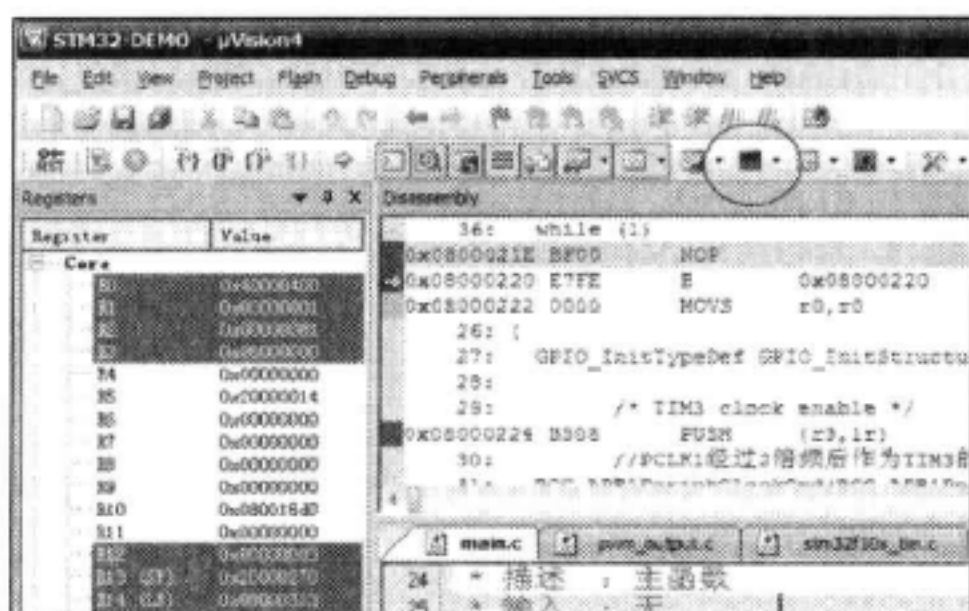


图 13-14 点击 Analysis Windows

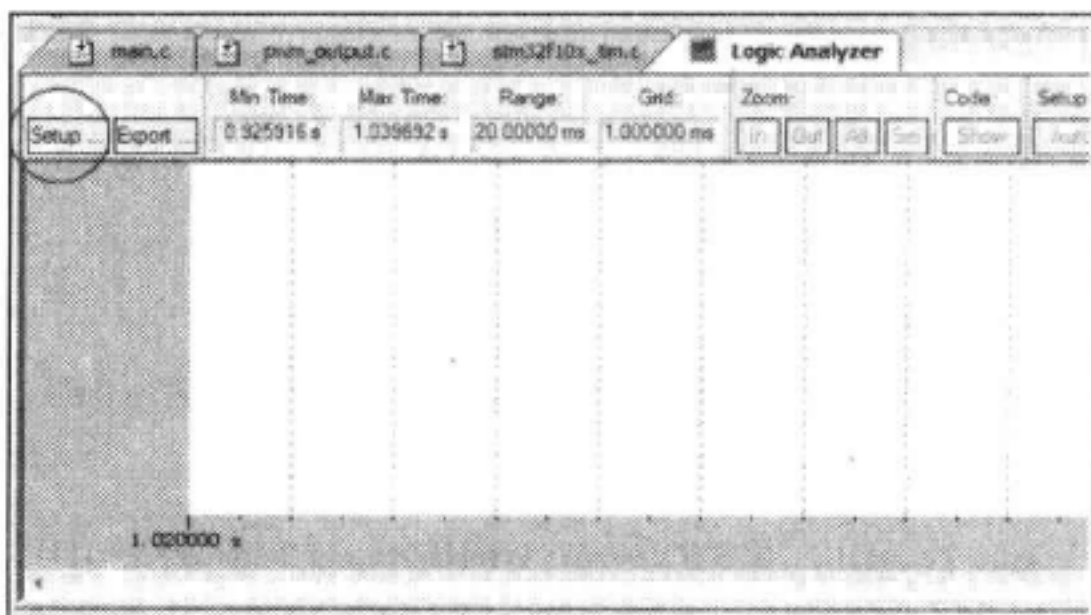


图 13-15 点击 Setup 选项卡

4) 点击 NEW (Insert) 图标, 见图 13-16, 在下面的文本框中输入 TIM3 的 PWM 通道, 这里分别是: PORTA.6、PORTA.7、PORTB.0、PORTB.1。然后点击“Close”按钮。

5) 点击运行图标, 见图 13-17。

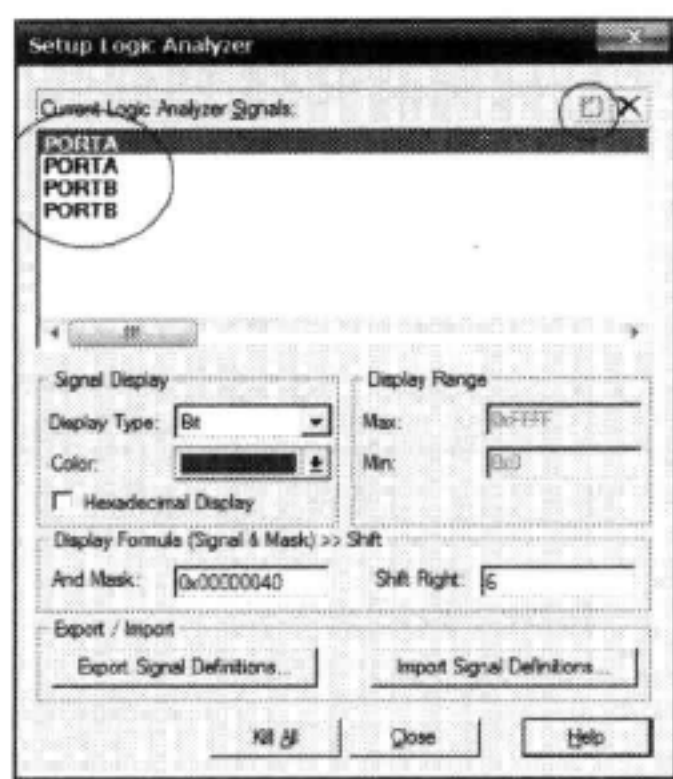


图 13-16 输入要查看的引脚

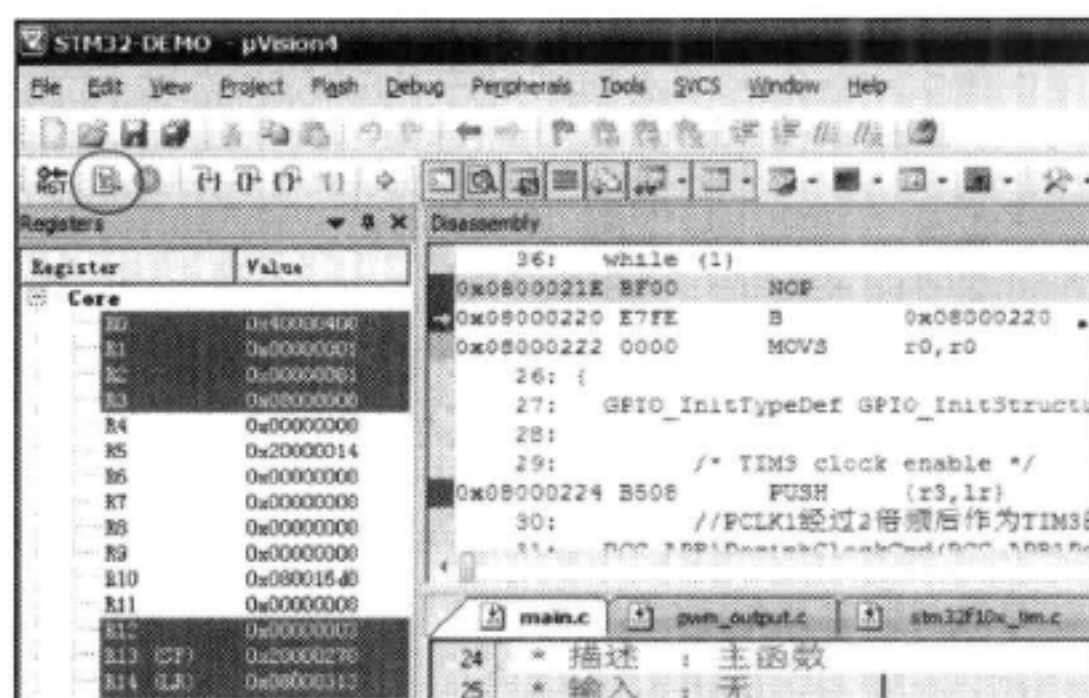



图 13-17 全速运行

6) 这时候正常的话则是出现如图 13-18 所示的 PWM 信号, 若没有出现逻辑分析仪界面可点击工具栏中的  按钮。

不正常的话则出现如图 13-19 所示情况, 根本看不到 PWM 信号。这时我们不免会有些抓狂, 但请大家放心, 接下来看看我们是怎么解决的吧。

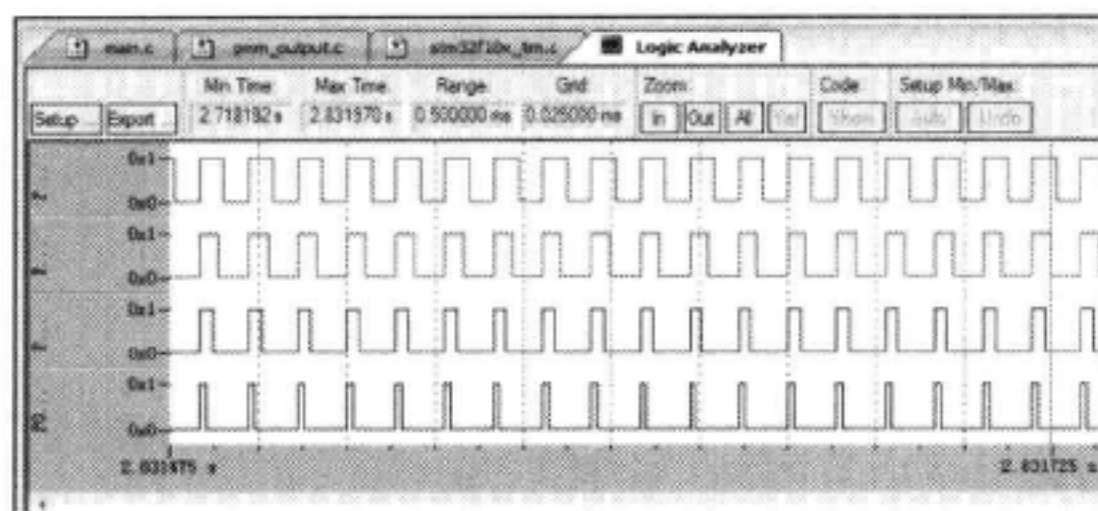


图 13-18 PWM 实验现象

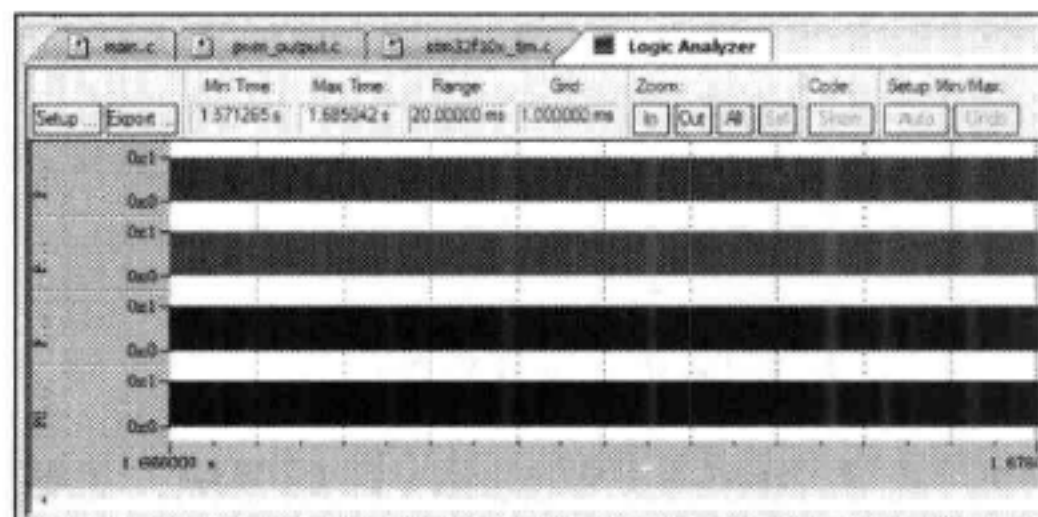


图 13-19 没有调整好时间轴的现象

7) 其实出现上面的情况是因为我们显示 PWM 信号时没有放大的缘故, 我们可以点击 In 这个按钮来将 PWM 信号显示得大点, 如图 13-20 所示, 一切搞定。

8) 我们可以点击停止按钮, 让 PWM 信号静止显示。见图 13-21。

9) 选中分析仪界面中的 Cursor 选项, 出现时间标签, 可以用来测量周期、频率。见图 13-22。

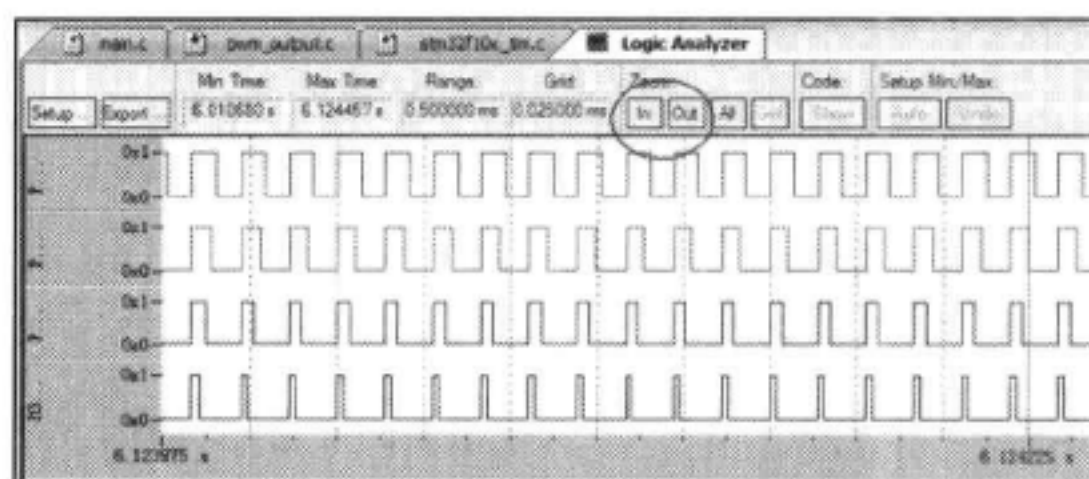


图 13-20 设置 Zoon 的 In 和 Out 调整时间轴

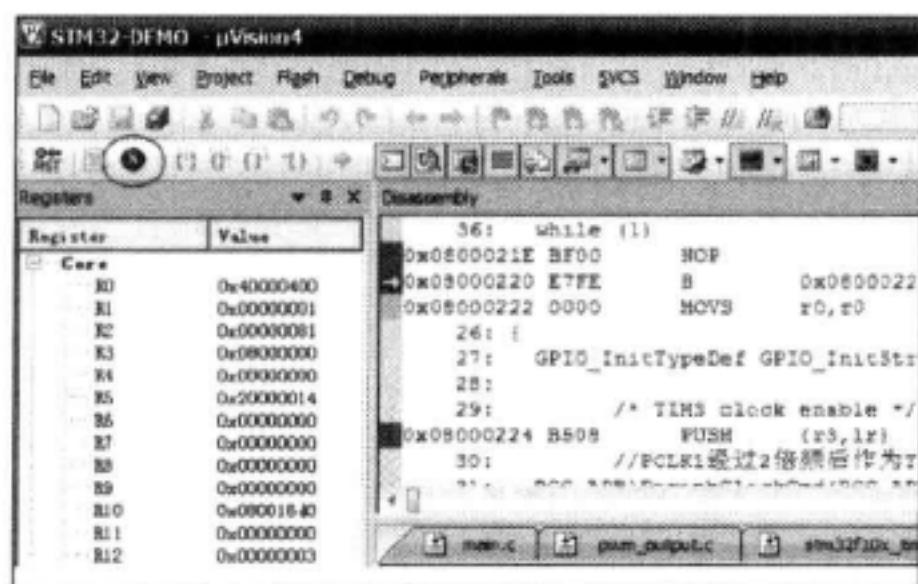


图 13-21 停止运行

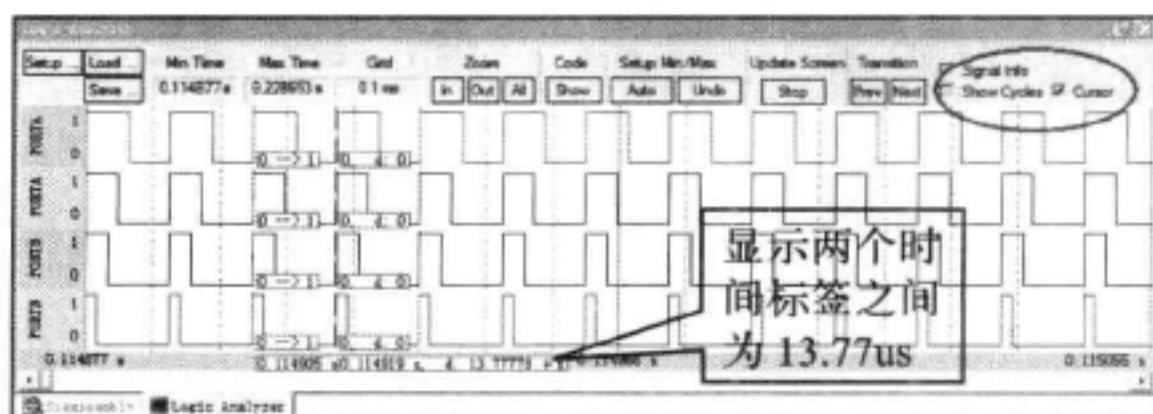


图 13-22 查看周期

我们可以看到，这样测量出来的一个脉冲周期约为 $13.77\mu\text{s}$ ，与我们配置的 $13.88\mu\text{s}$ 一致（误差是因为时间标签的位置放得不够准确）。

10) 仿真完毕之后，点击 Start/Stop Debug Session 图标就可以回到正常的代码编辑模式。见图 13-23。

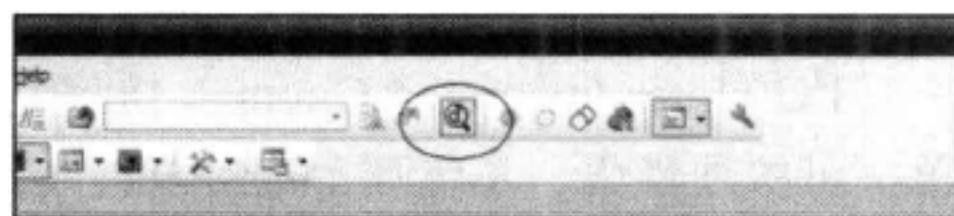


图 13-23 回到代码编辑模式

11) 要是有了示波器的话，看到的效果则更真实、准确。



第 14 章

I²C 接口

14.1 I²C 协议简介

I²C (Inter-Integrated Circuit) 协议是由 Philips 公司开发的, 由于它具备引脚少、硬件实现简单、可扩展性强、不需要如 USART、CAN 的外部收发设备等特点, 现在被广泛地使用在系统内多个集成电路 (IC) 间的通信。

根据 “I²C 总线协议版本 2.1-2000” 的说明, 我们可以更详细地了解 I²C 协议。

14.1.1 物理层

- 1) 它只使用两条总线线路: 一条双向串行数据线 (SDA), 一条串行时钟线 (SCL)。见图 4-1。
- 2) 每个连接到总线的设备都有一个独立的地址, 主机可以利用这个地址进行不同设备之间的访问。
- 3) 多主机同时使用总线时, 为了防止数据冲突, 会利用仲裁方式决定由哪个设备占用总线。
- 4) 具有三种传输模式: 标准模式的传输速率为 100 Kbit/s, 快速模式为 400 Kbit/s, 高速模式下可达 3.4 Mbit/s, 但目前大多 I²C 设备尚不支持高速模式。
- 5) 片上的滤波器可以滤去总线数据线上的毛刺波以保证数据完整。
- 6) 连接到相同总线的 IC 数量受到总线的最大电容 400 pF 限制。

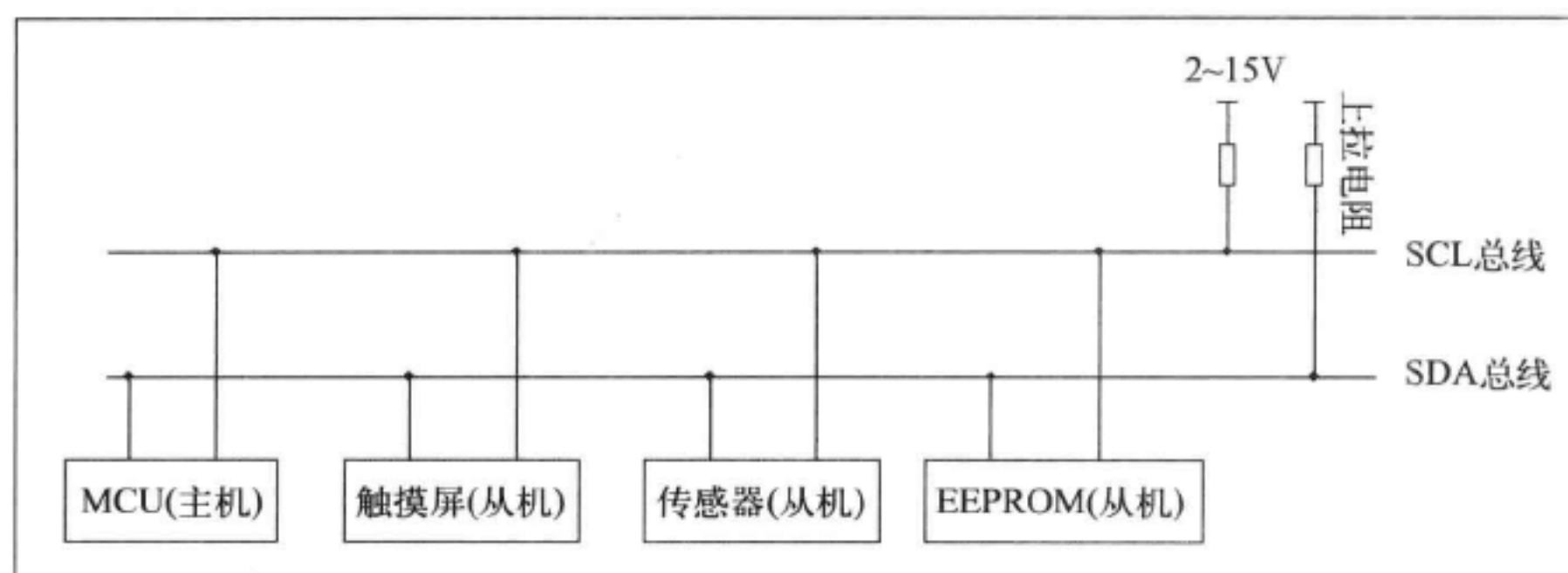


图 14-1 常见的 I²C 通信系统

14.1.2 协议层

I²C 的协议包括起始和停止条件、数据有效性、响应、仲裁、时钟同步和地址广播等环节，由于我们使用的是 STM32 集成的硬件 I²C 接口，并不需要用软件去模拟 SDA 和 SCL 线的时序，所以我们直接以 I²C 通信的流程为大家讲解。见图 14-2 和图 14-3。

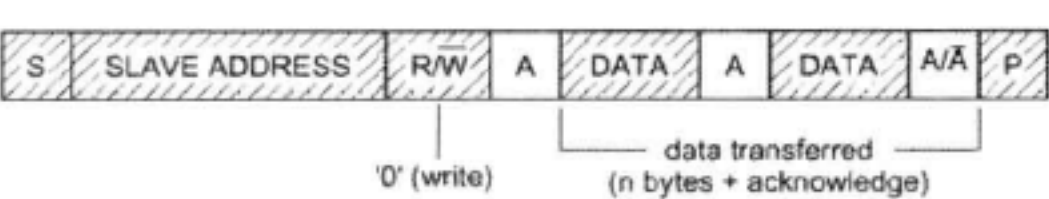


图 14-2 主机写数据到从机

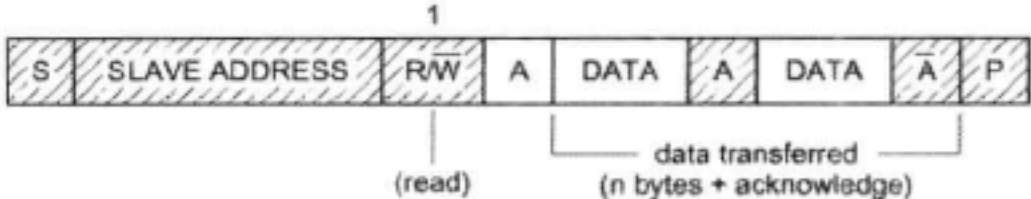


图 14-3 主机由从机中读数据

图 14-2 和图 14-3 的图例如下：

- ▨

 数据由主机传输至从机
- 数据由从机传输至主机
- S：传输开始信号
- SLAVE_ADDRESS：从机地址
- R/W：传输方向选择位，1 为读，0 为写
- A/A：应答或非应答信号
- P：停止传输信号

这两幅图表示的是主机和从机通信时 SDA 线的数据包序列。

其中 S 表示由主机的 I²C 接口产生的传输起始信号 (S)，这时连接到 I²C 总线上的所有从机都会接收到这个信号。

起始信号产生后，所有从机就开始等待主机紧接下来广播的从机地址信号 (SLAVE_ADDRESS)，在 I²C 总线上，每个设备的地址都是唯一的。当主机广播的地址与某个设备地址相同时，这个设备就被选中了，没被选中的设备将会忽略之后的数据信号。根据 I²C 协议，这个从机地址可以是 7 位或 10 位。

在地址位之后，是传输方向的选择位，该位为 0 时，表示后面的数据传输方向是由主机传输至从机。该位为 1 时，则相反。

从机接收到匹配的地址后，主机或从机会返回一个应答 (A) 或非应答 (\overline{A}) 信号，只有接收到应答信号后，主机才能继续发送或接收数据。

若配置的方向传输位为写数据，广播完地址，接收到应答信号后，主机开始正式向从机传输数据 (DATA)，数据包的大小为 8 位。主机每发送完一个数据，都要等待从机的应答信号 (A)，重复这个过程，可以向从机传输 N 个数据，这个 N 没有大小限制。当数据传输结束时，主机向从机发送一个停止传输信号 (P)，表示不再传输数据。

若配置的方向传输位为读数据，广播完地址，接收到应答信号后，从机开始向主机返回数据 (DATA)，数据包大小也为 8 位。从机每发送完一个数据，都会等待主机的应答信号 (A)，重复这个过程，可以返回 N 个数据，这个 N 也没有大小限制。当主机希望停止接收数据时，就向从机返回一个非应答信号 (\overline{A})，则从机自动停止数据传输。

14.2 STM32 的 I²C 特性及架构

14.2.1 I²C 接口特性

- 1) STM32 的中等容量和大容量型号的芯片均有多达两个的 I²C 总线接口。
- 2) 能够工作于多主模式或从模式，分别为主接收器、主发送器、从接收器及从发送器。
- 3) 支持标准模式 100 Kbit/s 和快速模式 400 Kbit/s，不支持高速模式。
- 4) 支持 7 位或 10 位寻址。
- 5) 内置了硬件 CRC 发生器 / 校验器。
- 6) I²C 的接收和发送都可以使用 DMA 操作。
- 7) 支持系统管理总线 (SMBus) 2.0 版。

14.2.2 I²C 架构

见图 14-4，我们可以看到，I²C 的所有硬件架构就是根据 SCL 线和 SDA 线展开的（其中 SMBALERT 线用于 SMBus）。

SCL 线的时序即为 I²C 协议中的时钟信号，它由 I²C 接口根据时钟控制寄存器 (CCR) 控制，控制的参数主要为时钟频率。

而 SDA 线的信号则通过一系列数据控制架构，在将要发送的数据的基础上，根据协议添加各种起始信号、应答信号、地址信号，实现以 I²C 协议的方式发送出去。读取数据时则从 SDA 线上的信号中取出接收到的数据值。发送和接收的数据都被保存在数据寄存器 (DR) 上。

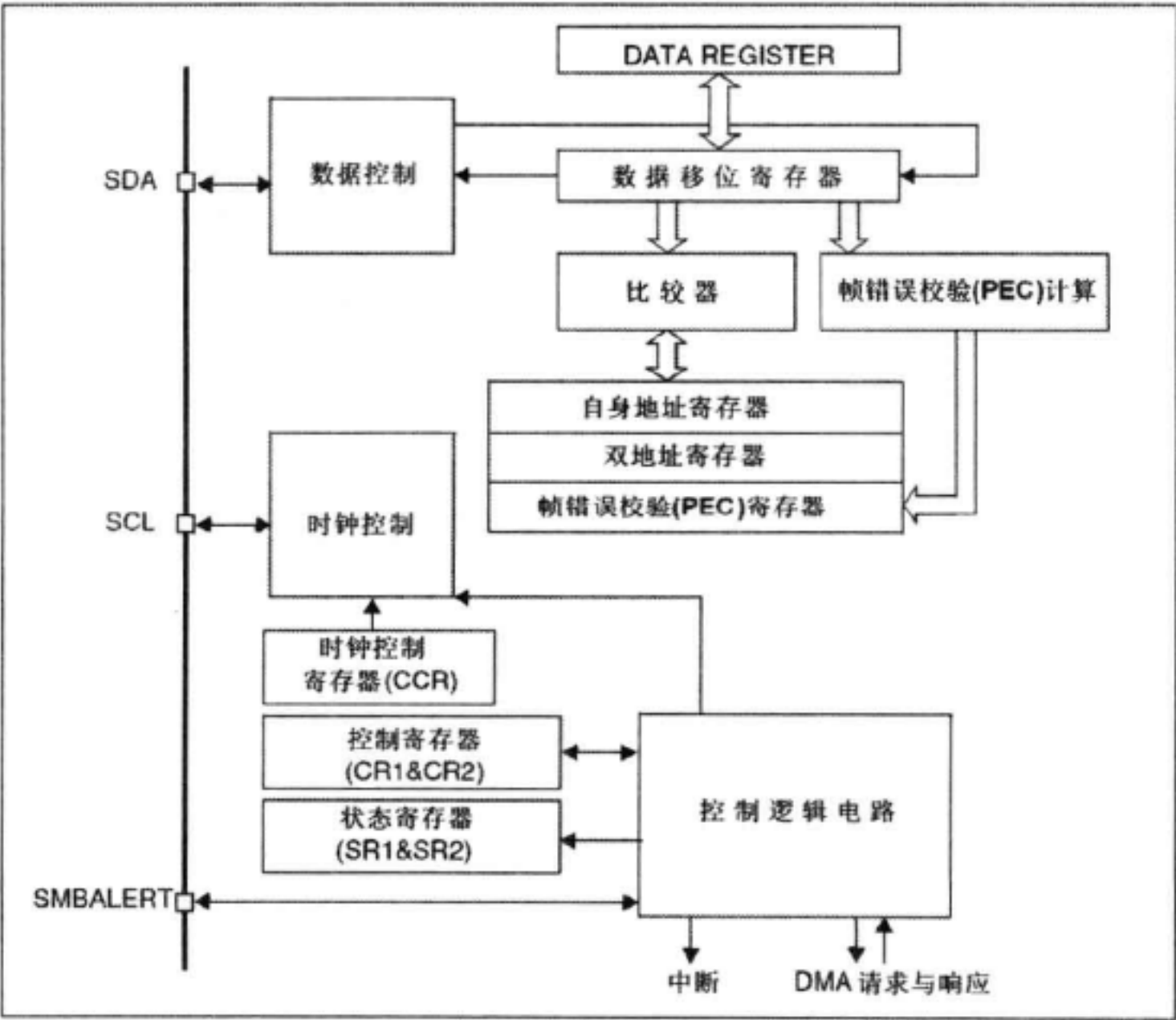


图 14-4 I²C 架构图

14.3 I²C 接口读写 EEPROM 实验

对 EEPROM 读写最常用的方式是使用 I²C 协议。本节以 EEPROM 的读写实验为大家讲解 STM32 的 I²C 使用方法。

配套 STM32 开发板用的是 STM32F103VET6，它有两个 I²C 接口。本实验使用 I²C1，对应地连接到 EEPROM（型号：AT24C02）的 SCL 和 SDA 线，实现 I²C 通信，对 EEPROM 进行读写。

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

14.3.1 实验描述及工程文件清单

1. 实验描述

向 EEPROM 写入数据，再读取出来，进行校验，通过串口打印写入与读取出来的数据，并输出校验结果。

2. 硬件连接

☐ PB6 – I2C1_SCL

☐ PB7 – I2C1_SDA

3. 库文件

使用 3.5 版本固件库：

☐ startup/start_stm32f10x_hd.c

☐ CMSIS/core_cm3.c

☐ CMSIS/system_stm32f10x.c

☐ FWlib/stm32f10x_gpio.c

☐ FWlib/stm32f10x_rcc.c

☐ FWlib/stm32f10x_usart.c

☐ FWlib/stm32f10x_i2c.c

4. 用户文件

☐ USER/main.c

☐ USER/stm32f10x_it.c

☐ USER/usart1.c

☐ USER/i2c_ee.c

图 14-5 为该实验硬件原理图。

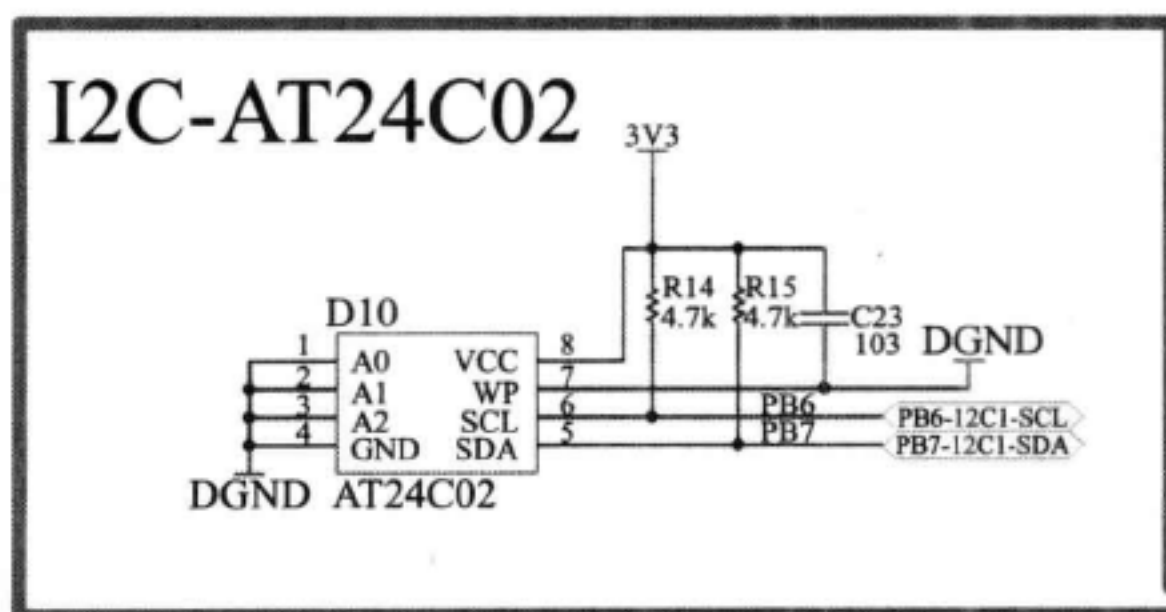


图 14-5 配套 STM32 开发板 I²C-EEPROM 硬件原理图

14.3.2 配置工程环境

本 I²C-EEPROM 实验中我们用到了 GPIO、RCC、USART 及 I²C 外设，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_i2c.c。本实验中没有使用中断，使用轮询的方式来完成 I²C 的收发数据。

接下来添加旧工程中的外设用户文件 `usart.c`，以便调试和观察实验效果。新建 `i2c_ee.c` 及 `i2c_ee.h` 文件，并在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉。见代码清单 14-1。

代码清单 14-1 I²C 接口实例的 `stm32f10x_conf.h` 文件配置

```

1.  /**
2.      *****
3.      * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.      * @author  MCD Application Team
5.      * @version V3.5.0
6.      * @date    08-April-2011
7.      * @brief   Library configuration file.
8.      *****/  *
9.
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_i2c.h"
12. #include "stm32f10x_rcc.h"
13. #include "stm32f10x_usart.h"

```

14.3.3 main 文件

配置好所需的库文件之后，我们就从 `main` 函数开始分析，见代码清单 14-2。

代码清单 14-2 I²C 接口实例的 `main` 函数

```

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

```

代码中的系统库函数 `SystemInit()` 被注释掉了，这是在升级代码时修改的，3.5 版本的代码不需要在 `main` 函数中再调用 `SystemInit()` 函数来初始化时钟。接下来调用用户函数 `USART1_Config()` 和 `I2C_EE_Init()` 以配置串口及初始化 I²C 接口。

初始化完成后由串口向终端输出调试信息，接着调用用户函数 `I2C_Test()`，它利用配置好的 I²C 接口向 EEPROM 写入数据，再把这些数据读取出来进行校验，检查我们的 I²C 是否正常工作。

14.3.4 I²C 接口初始化

我们先来分析 I²C 接口的初始化函数 `I2C_EE_Init()`。它是用户编写的函数，与其他对 GPIO 复用的外设一样，它先调用了用户函数 `I2C_GPIO_Config()` 配置好 I²C 所用的 I/O 端口，然后再调用用户函数 `I2C_Mode_Configu()` 设置 I²C 的工作模式，并使能相关外设的时钟。见代码清单 14-3。

代码清单 14-3 I2C_EE_Init() 函数

```

1.  /*
2.   * 函数名：I2C_EE_Init
3.   * 描述   ：I2C 外设 (EEPROM) 初始化
4.   * 输入    ：无
5.   * 输出    ：无
6.   * 调用    ：外部调用
7.   */
8.  void I2C_EE_Init(void)
9.  {
10.
11.   I2C_GPIO_Config();
12.
13.   I2C_Mode_Configu();
14.
15.  /* 根据头文件 i2c_ee.h 中的定义来选择 EEPROM 要写入的地址 */
16.  #ifdef EEPROM_Block0_ADDRESS
17.   /* 选择 EEPROM Block0 来写入 */
18.   EEPROM_ADDRESS = EEPROM_Block0_ADDRESS;
19. #endif
20.
21.  #ifdef EEPROM_Block1_ADDRESS
22.   /* 选择 EEPROM Block1 来写入 */
23.   EEPROM_ADDRESS = EEPROM_Block1_ADDRESS;
24. #endif
25.
26.  #ifdef EEPROM_Block2_ADDRESS
27.   /* 选择 EEPROM Block2 来写入 */
28.   EEPROM_ADDRESS = EEPROM_Block2_ADDRESS;
29. #endif
30.
31.  #ifdef EEPROM_Block3_ADDRESS
32.   /* 选择 EEPROM Block3 来写入 */
33.   EEPROM_ADDRESS = EEPROM_Block3_ADDRESS;
34. #endif
35. }
```

1. EEPROM 的地址

我们知道，每一个连接到 I²C 总线上的设备都需要有唯一的地址，本实验中的 EEPROM 地址是如何决定的呢？阅读 EEPROM 的 datasheet，我们可以看到如图 14-6 所示说明。

本实验中使用的 EEPROM 型号为 AT24C02，容量为 2 Kbits，说明中已经对 EEPROM 地址的高四位做了硬性规定，而最低位又为 R/W（传输方向选择位）。在制作硬件时，我们可以根据需要改变的是地址位中的 A2、A1、A0 位。查看本实验中的硬件接法图可知 EEPROM 的 A2、A1、A0 位均接地，所以它的地址为：0xA0 或 0xA1，为什么有两个呢？因为最低位为数据方向选择位，在我们使用这个地址变量时，调用的库函数还会对数据方向位重新处理，所以最低位的数据并不影响我们实际的使用，但这也不代表 EEPROM 的实际地址有两个。如果不是使用库函数，那就得小心使用了。

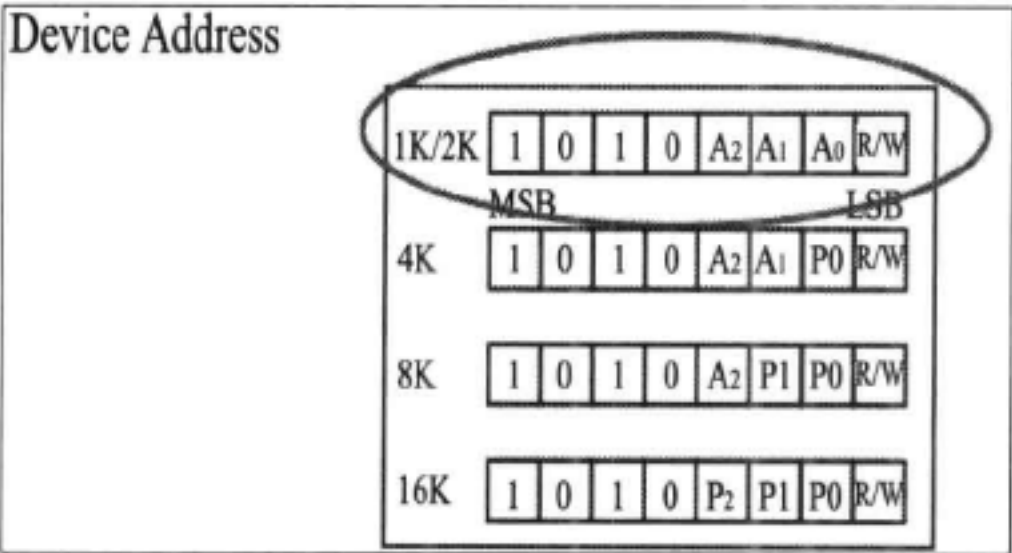


图 14-6 EEPROM 地址说明

在 I2C_EE_Init() 函数中，用条件编译确定了 EEPROM 的器件地址，这个宏在 i2c_ee.h 文件中进行了定义，见代码清单 14-4。若大家对 EEPROM 的编址不一样，就要使用其他宏。

代码清单 14-4 EEPROM 器件地址宏

```
1. /* EEPROM Addresses defines */
2. #define EEPROM_Block0_ADDRESS 0xA0 /* E2 = 0 */
3. // #define EEPROM_Block1_ADDRESS 0xA2 /* E2 = 0 */
4. // #define EEPROM_Block2_ADDRESS 0xA4 /* E2 = 0 */
5. // #define EEPROM_Block3_ADDRESS 0xA6 /* E2 = 0 */
```

条件编译中的变量值 EEPROM_ADDRESS 在正式使用时我们再来分析。

2. GPIO 端口初始化

现在先来看看 GPIO 的初始化，它在 I2C_GPIO_Config() 函数中实现了，见代码清单 14-5。

代码清单 14-5 I2C_GPIO_Config() 函数

```
1. /*
2.  * 函数名：I2C_GPIO_Config
3.  * 描述：I2C1 I/O 配置
4.  * 输入：无
5.  * 输出：无
6.  * 调用：内部调用
7.  */
8. static void I2C_GPIO_Config(void)
9. {
10.  GPIO_InitTypeDef GPIO_InitStructure;
11.
12.  /* 使能与 I2C1 有关的时钟 */
13.  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
14.  RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
```



```
15.
16. /* PB6-I2C1_SCL、PB7-I2C1_SDA*/
17. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
18. GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
19. GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD; // 开漏输出
20. GPIO_Init(GPIOB, &GPIO_InitStructure);
21.}
```

在本函数中，我们开启了 GPIO 时钟、I²C1 的时钟，并且对 I²C1 复用的两个引脚进行初始化配置，模式设置为开漏输出。若不清楚 I²C1 的复用引脚和复用模式，可分别查询《STM32 数据手册》的引脚定义部分和《STM32 参考手册》的 GPIO 章节。见表 14-1 和表 14-2。

表 14-1 I²C1 引脚定义

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCSP64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
B6	C5	A5	57	91	135	PB5	I/O		PB5	I2C1_SMBA/ SPI3_MOSI I2S3_SD	TIM3_CH2/ SPI1_MOSI
C6	B5	B5	58	92	136	PB6	I/O	FT	PB6	I2C1_SCL/TIM4_CH1	USART1_TX
D6	A5	C5	59	93	137	PB7	I/O	FT	PB7	I2C1_SDA /FSMC_NADV TIM4_CH2	USART1_RX

表 14-2 I²C 复用 GPIO 模式设置

I ² C 引脚	配置	GPIO 配置
I2Cx_SCL	I2C 时钟	开漏复用输出
I2Cx_SDA	I2C 数据	开漏复用输出

3. I²C 模式初始化

接下来我们再来分析一下在 I2C_Mode_Configu() 函数中进行的 I²C 模式初始化，见代码清单 14-6。

代码清单 14-6 I2C_Mode_Configu() 函数

```
1. /*
2. * 函数名：I2C_Configuration
3. * 描述：I2C 工作模式配置
4. * 输入：无
5. * 输出：无
6. * 调用：内部调用
7. */
8. static void I2C_Mode_Configu(void)
```

```

9. {
10.  I2C_InitTypeDef  I2C_InitStructure;
11.
12.  /* I2C 配置 */
13.  I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
14.  I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
15.  I2C_InitStructure.I2C_OwnAddress1 = I2C1_OWN_ADDRESS7;
16.  I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
17.  I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
18.  I2C_InitStructure.I2C_ClockSpeed = I2C_Speed;
19.
20.  /* I2C1 初始化 */
21.  I2C_Init(I2C1, &I2C_InitStructure);
22.  /* 使能 I2C1 */
23.  I2C_Cmd(I2C1, ENABLE);
24.}

```

与其他外设初始化一样，I²C 模式的初始化还是对它的初始化结构体进行参数配置。我们来分析它这些结构体成员：

1) I2C_Mode：本成员是选择 I²C 的使用方式，有 I²C 模式（I2C_Mode_I2C）和 SMBus 模式。（I2C_Mode_SMBusDevice、I2C_Mode_SMBusHost）。本实验中使用的是 I²C 模式。

2) I2C_DutyCycle：本成员设置的是 I²C 的 SCL 线时钟的占空比。SCL 线的时钟信号的高电平时间与低电平时间不必相同，由于 SDA 线在 SCL 线维持在高电平时读取或写入数据，而在 SCL 的低电平期间 SDA 的数据发生变化，所以高电平时间较长就不容易出现数据错误。根据 I²C 协议，在快速模式和高速模式下 SCL 的高低电平时间可以不同。在 STM32 的 I²C 占空比配置中有两个选择，分别为高电平时间和低电平时间之比为 16 : 9（I2C_DutyCycle_16_9）和 2 : 1（I2C_DutyCycle_2）。本实验中使用的是 2 : 1。

3) I2C_OwnAddress1：本成员配置的是 STM32 的 I²C 设备自己的地址，每个连接到 I²C 总线上的设备都要有一个自己的地址，作为主机也不例外。就如同前面介绍的连接到 I²C 的 EEPROM 的设备地址一样，只不过 STM32 的 I²C 地址可软件编程。这个地址可以被配置为 7 位和 10 位地址。本实验中把这个地址设置为 0x0A（自定义宏 I2C1_OWN_ADDRESS7 的值）。

4) I2C_Ack_Enable：本成员关于 I²C 应答设置，设置为使能则每接收到一个字节就返回一个应答信号。本实验配置为允许应答（I2C_Ack_Enable），这是绝大多数遵循 I²C 标准的设备通信的要求，改为禁止应答（I2C_Ack_Disable）往往会导致通信错误。

5) I2C_AcknowledgeAddress：本成员选择 I²C 的寻址模式是 7 位还是 10 位地址。这需要根据实际连接到 I²C 总线上设备的地址进行选择。本实验是与 EEPROM 进行通信，使用的为 7 位寻址模式（I2C_AcknowledgedAddress_7bit）。

6) I2C_ClockSpeed：本成员设置的是 I²C 的传输速率，在调用初始化函数时，函数会根据我们输入的数值经过运算后把分频值写入到 I²C 的时钟控制寄存器。而我们写入的这个参数值不得高于 400 kHz。本实验向这个成员写入参数为 400000（自定义宏 I2C_Speed）。实际上由于 I²C 使

用的 APB1 时钟为 36 MHz，不是 10 MHz 的整数倍，因此最终分频后输出的 SCL 线时钟并不是精确的 400 kHz。

对结构体成员赋值完成后，我们调用库函数 I2C_Init() 根据我们的配置对 I²C 进行初始化，并调用库函数 I2C_Cmd() 使能 I²C 外设。

14.3.5 对 EEPROM 的读写操作

对 I²C 接口初始化完成后，我们就可以利用 I²C 接口，根据 EEPROM 设备的要求对它进行读写操作。在本实验中的 I2C_Test() 函数就是一个对 EEPROM 进行读写的例子，见代码清单 14-7。

代码清单 14-7 I2C_Test() 函数

```

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

```



```

38.     }
39.     printf("0x%02X ", I2c_Buf_Read[i]);
40.     if(i%16 == 15)
41.         printf("\n\r");
42.
43. }
44. printf("I2C(AT24C02) 读写测试成功 \n\r");
45.}

```

这个函数实现的功能是把数值 0 ~ 255 按顺序填入缓冲区数组，并通过串口打印到终端，接着通过用户函数 I2C_EE_BufferWrite() 把缓冲区的数据写入 EEPROM。写入成功之后，利用用户函数 I2C_EE_BufferRead() 把数据读取出来，进行校验，判断数据是否被正确写入。

其中的 I2C_EE_BufferWrite() 和 I2C_EE_BufferRead() 函数就是读写 EEPROM 的重点，编写这个函数主要是根据所使用的 EEPROM 说明。我们通过介绍这个 I2C_EE_BufferWrite() 函数让大家熟悉 STM32 的 I²C 通信流程，而类似的用户函数 I2C_EE_BufferRead() 则留给读者进行分析。

I2C_EE_BufferWrite() 具有三个输入参数：pBuffer 为缓冲区指针，WriteAddr 将写入数据到 EEPROM 的存储地址，NumByteToWrite 为要写入 EEPROM 的字节数。我们在调用时的形式如下 I2C_EE_BufferWrite (I2c_Buf_Write, EEP_Firstpage, 256)，前两个参数定义见代码清单 14-8。I2C_EE_BufferWrite() 函数见代码清单 14-9。

代码清单 14-8 I2C_EE_BufferWrite 的输入参数

```

1. u8 I2c_Buf_Write[256];
2. #define EEP_Firstpage    0x00

```

代码清单 14-9 I2C_EE_BufferWrite() 函数

```

1. /*
2.  * 函数名：I2C_EE_BufferWrite
3.  * 描述   ：将缓冲区中的数据写到 I2C EEPROM 中
4.  * 输入   ：-pBuffer 缓冲区指针
5.  *          -WriteAddr 接收数据的 EEPROM 的地址
6.  *          -NumByteToWrite 要写入 EEPROM 的字节数
7.  * 输出   ：无
8.  * 返回   ：无
9.  * 调用   ：外部调用
10. */
11. void I2C_EE_BufferWrite(u8* pBuffer, u8 WriteAddr, u16 NumByteToWrite)
12. {
13.     u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0;
14.
15.     Addr = WriteAddr % I2C_PageSize;
16.     count = I2C_PageSize - Addr;
17.     NumOfPage = NumByteToWrite / I2C_PageSize;
18.     NumOfSingle = NumByteToWrite % I2C_PageSize;
19.

```

```

20.  /* If WriteAddr is I2C_PageSize aligned */
21.  if(Addr == 0)
22.  {
23.      /* If NumByteToWrite < I2C_PageSize */
24.      if(NumOfPage == 0)
25.      {
26.          I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
27.          I2C_EE_WaitEepromStandbyState();
28.      }
29.      /* If NumByteToWrite > I2C_PageSize */
30.      else
31.      {
32.          while (NumOfPage--)
33.          {
34.              I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
35.              I2C_EE_WaitEepromStandbyState();
36.              WriteAddr += I2C_PageSize;
37.              pBuffer += I2C_PageSize;
38.          }
39.
40.          if(NumOfSingle!=0)
41.          {
42.              I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
43.              I2C_EE_WaitEepromStandbyState();
44.          }
45.      }
46.  }
47.  /* If WriteAddr is not I2C_PageSize aligned */
48.  else
49.  {
50.      /* If NumByteToWrite < I2C_PageSize */
51.      if(NumOfPage== 0)
52.      {
53.          I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
54.          I2C_EE_WaitEepromStandbyState();
55.      }
56.      /* If NumByteToWrite > I2C_PageSize */
57.      else
58.      {
59.          NumByteToWrite -= count;
60.          NumOfPage = NumByteToWrite / I2C_PageSize;
61.          NumOfSingle = NumByteToWrite % I2C_PageSize;
62.
63.          if(count != 0)
64.          {
65.              I2C_EE_PageWrite(pBuffer, WriteAddr, count);
66.              I2C_EE_WaitEepromStandbyState();
67.              WriteAddr += count;
68.              pBuffer += count;
69.          }
70.
71.          while (NumOfPage--)

```

```

72.    {
73.        I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
74.        I2C_EE_WaitEepromStandbyState();
75.        WriteAddr += I2C_PageSize;
76.        pBuffer += I2C_PageSize;
77.    }
78.    if(NumOfSingle != 0)
79.    {
80.        I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
81.        I2C_EE_WaitEepromStandbyState();
82.    }
83.    }
84.    }
85.)

```

首先，我们要了解 EEPROM 的数据组织形式，EEPROM 设备把它的存储矩阵进行了分页处理，见图 14-7。

型号为 AT24C02 的 EEPROM 分为 32 页，每页可存储 8 个字节的数据，若在同一页写入超过 8 字节，则超过的部分会被写在该页的起始地址，这样部分数据会被覆盖。

Memory Organization	AT24C01A, 1K SERIAL EEPROM: Internally organized with 16 pages of 8 bytes each, the 1K requires a 7-bit data word address for random word addressing.
	AT24C02, 2K SERIAL EEPROM: Internally organized with 32 pages of 8 bytes each, the 2K requires an 8-bit data word address for random word addressing.
	AT24C04, 4K SERIAL EEPROM: Internally organized with 32 pages of 16 bytes each, the 4K requires a 9-bit data word address for random word addressing.
	AT24C08A, 8K SERIAL EEPROM: Internally organized with 64 pages of 16 bytes each, the 8K requires a 10-bit data word address for random word addressing.
	AT24C16A, 16K SERIAL EEPROM: Internally organized with 128 pages of 16 bytes each, the 16K requires an 11-bit data word address for random word addressing.

图 14-7 EEPROM 分页方式

因此，为了把连续的缓冲区数组按页写入 EEPROM，就需要对缓冲区进行分页处理。I2C_EE_BufferWrite() 函数根据我们输入的缓冲区大小参数 NumByteToWrite，计算出我们需要写入多少页，并计算写入位置。

分页处理好之后，调用 I2C_EE_PageWrite() 函数，这个函数是与 EEPROM 进行 I²C 通信的最底层函数，它与 STM32 的 I²C 库函数使用密切相关。见代码清单 14-10。

代码清单 14-10 I2C_EE_PageWrite() 函数

```

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

```



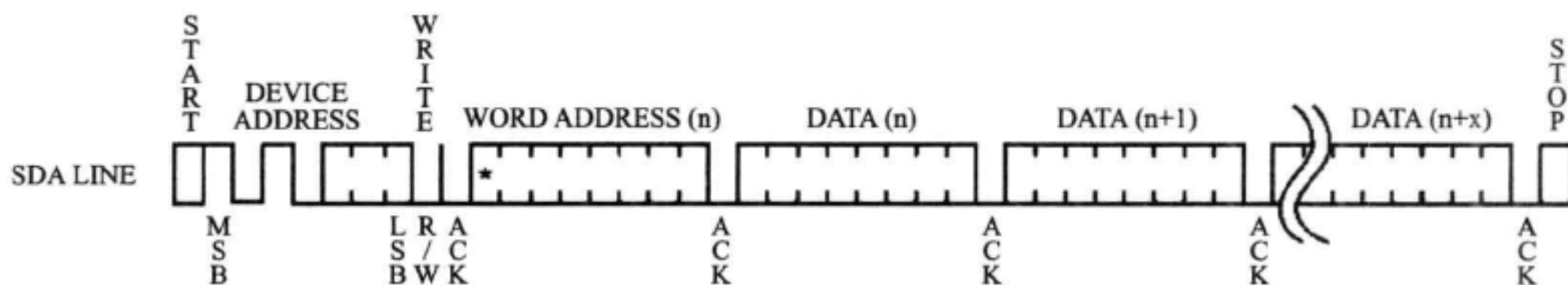
```

17. I2C_GenerateSTART(I2C1, ENABLE);
18.
19. /* Test on EV5 and clear it */
20. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
21.
22. /* Send EEPROM address for write */
23. I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);
24.
25. /* Test on EV6 and clear it */
26. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
27.
28. /* Send the EEPROM's internal address to write to */
29. I2C_SendData(I2C1, WriteAddr);
30.
31. /* Test on EV8 and clear it */
32. while(! I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
33.
34. /* While there is data to be written */
35. while(NumByteToWrite--)
36. {
37.     /* Send the current byte */
38.     I2C_SendData(I2C1, *pBuffer);
39.
40.     /* Point to the next byte to be written */
41.     pBuffer++;
42.
43.     /* Test on EV8 and clear it */
44.     while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
45. }
46. /* Send STOP condition */
47. I2C_GenerateSTOP(I2C1, ENABLE);
48.}

```

1. EEPROM 的页写入时序

这个页写入的函数是根据 EEPROM 的页写入时序来编写的，见图 14-8。



(* = DON'T CARE bit for 1K)

图 14-8 EEPROM 页写入时序

在 I2C_EE_PageWrite() 函数中，我们先不去理会使用 while 语句循环检测事件的语句。这样我们就可以清晰地看到这个函数的代码执行流程就是 EEPROM 的页写入时序流程。

1) 第 17 行，调用库函数 I2C_Generate START() 产生 I²C 的通信起始信号 S。其库函数说明见图 14-9。



图 14-9 I2C GenerateSTART 函数说明

2) 第 23 行, 调用库函数 I2C_Send7bitAddress() 把前面条件编译中赋值的变量 EEPROM_ADDRESS 地址通过 I²C1 接口发送出去, 数据传输方向为 STM32 的 I²C 发送数据 (I2C_Direction_Transmitter)。其中的 EEPROM_ADDRESS 地址是指 EEPROM 作为挂载在 I²C 总线上设备寻址, 而不是 EEPROM 内存储矩阵的地址。本库函数的说明见图 14-10。

3) 第 29 行, 调用库函数 `I2C_SendData()`, 请注意这个库函数的输入参数为 `WriteAddr`, 根据 EEPROM 的页写入时序, 发送完 I²C 的地址后的第一个数据并不就是要写入 EEPROM 的数据, EEPROM 对这个数据解释为将要对存储矩阵写入的地址, 这个参数 `WriteAddr` 是在我们调用 `I2C_EE_PageWrite()` 函数时作为参数输入的, 在本实验中 `WriteAddr` 的值由 `I2C_EE_BufferWrite()` 计算得出。I2C `SendData()` 函数说明见图 14-11。

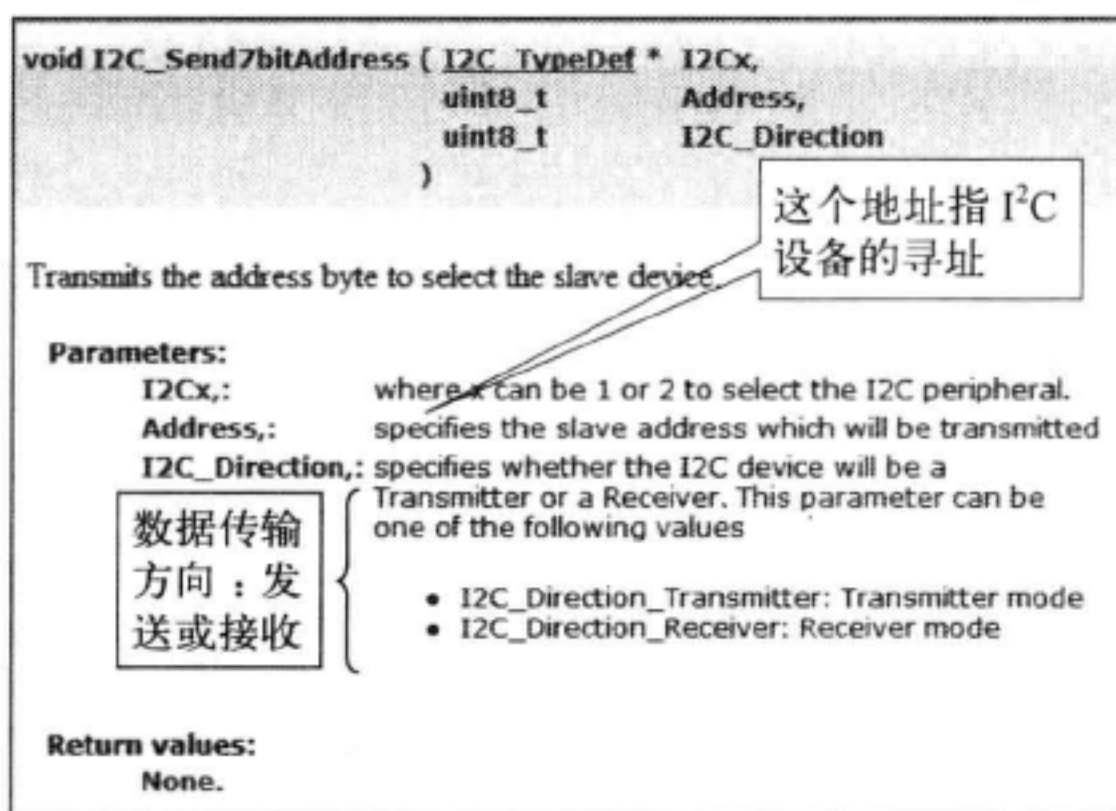


图 14-10 I2C Send7bitAddress 函数说明

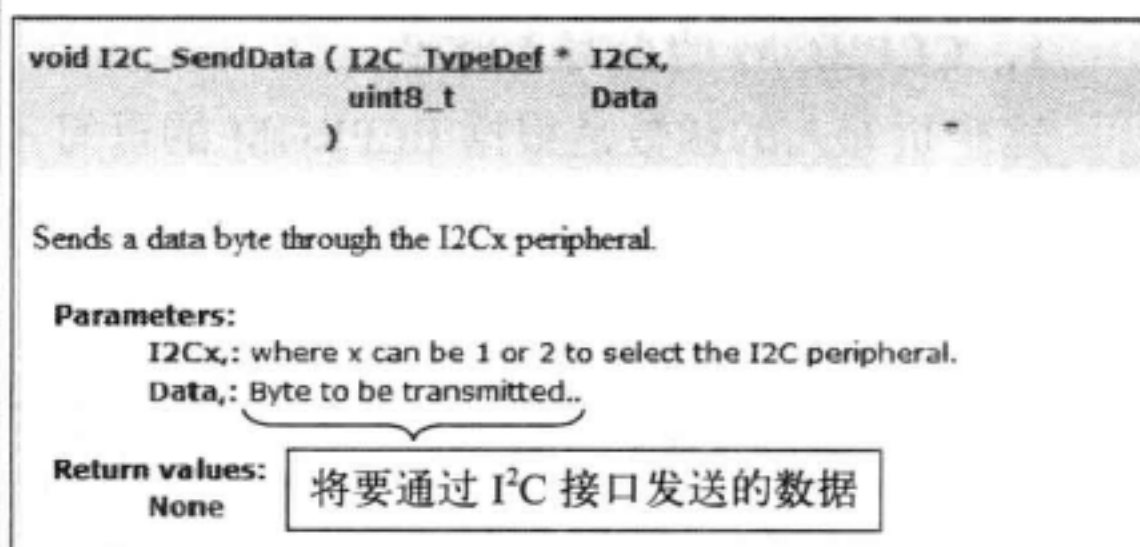


图 14-11 I2C SendData 函数说明

这个库函数实际上是把数据传输到数据寄存器，再由 I²C 模块根据 I²C 协议发送出去。

4) 第 35 ~ 45 行, 调用 I2C_SendData() 函数, 向 EEPROM 发送要写入的数据, 根据 EEPROM 的页写入时序, 这些数据将会被写入到前面发送的页地址中, 若连续写入超过一页的最大字节数 (本实验为 8 个), 则多出来的数据会重新从该页的起始地址连续写入, 覆盖前面的数据。

5) 第 47 行, 调用库函数 I2C_Generate_STOP() 产生 I²C 传输结束信号, 完成一次 I²C 通信。其库函数说明见图 14-12。

2. I²C 事件检测

在以上的分析中，我们忽略了 I²C 的事件检测，而在 STM32 的 I²C 通信中，事件的检测是必不可少的。从《STM32 参考手册》的序列图可以看到（见图 14-13），在 I²C 的通信过程中，会产生一系列的事件，出现事件后在相应的寄存器中会产生标志位。

图 14-13 的意思是：若发出了起始信号，会产生事件 5（EV5），即 STM32 的 I²C 成为主机模式；继续发送完 I²C 设备寻址并得到应答后，会产生 EV6，即 STM32 的 I²C 成为数据发送端；之后发送数据完成会产生 EV8 等。

我们在做出 I²C 通信操作时，可以通过循环调用库函数 I2C_CheckEvent() 进行事件查询，以确保上一操作完成后才进行下一操作。

如：在确定 SDA 总线空闲之后，作为主发送器的 STM32 发出起始信号，若成功，这时会产生 EV5，我们在第 20 行使用语句 while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT)) 来检测这个事件，确保检测到之后再执行下一操作。其中的参数 “I2C_EVENT_MASTER_MODE_SELECT” 在固件函数说明中可以查到这就是 “EV5” 的宏，后面对 EV6、EV8 的相应检测操作类似。

其中库函数 I2C_CheckEvent() 的参数说明见图 14-14。

```
void I2C_GenerateSTOP ( I2C_TypeDef * I2Cx,
                        FunctionalState NewState
                      )

Generates I2Cx communication STOP condition.

Parameters:
  I2Cx:      where x can be 1 or 2 to select the I2C peripheral.
  NewState:  new state of the I2C STOP condition generation. This
             parameter can be: ENABLE or DISABLE.

Return values:
  None.
```

图 14-12 I2C_GenerateSTOP 函数说明

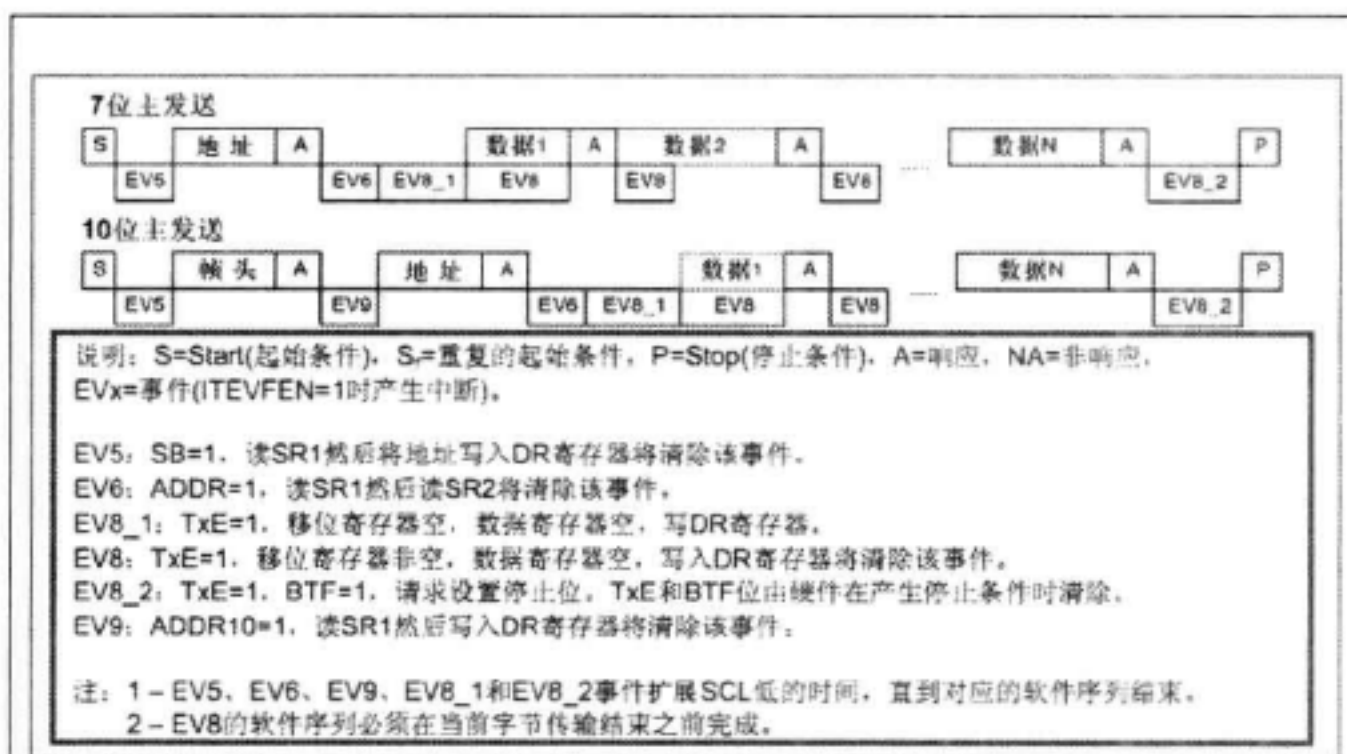


图 14-13 I²C 主发送通信过程产生的事件

Parameters:
 I2Cx: where x can be 1 or 2 to select the I2C peripheral.
 I2C_EVENT: specifies the event to be checked. This parameter can be one of the following values:

各种I²C事件
的参数及对应的事
件编号。如
“I2C_EVENT_
MASTER_
MODE_SELECT”
的事件编号见
下划线标示处

- I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED : EV1
- I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED : EV1
- I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MATCHED : EV1
- I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MATCHED : EV1
- I2C_EVENT_SLAVE_GENERALCALLADDRESS_MATCHED : EV1
- I2C_EVENT_SLAVE_BYTE_RECEIVED : EV2
- (I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_FLAG_DUALF) : EV2
- (I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_FLAG_GENCALL) : EV2
- I2C_EVENT_SLAVE_BYTE_TRANSMITTED : EV3
- (I2C_EVENT_SLAVE_BYTE_TRANSMITTED | I2C_FLAG_DUALF) : EV3
- (I2C_EVENT_SLAVE_BYTE_TRANSMITTED | I2C_FLAG_GENCALL) : EV3
- I2C_EVENT_SLAVE_ACK_FAILURE : EV3_2
- I2C_EVENT_SLAVE_STOP_DETECTED : EV4
- I2C_EVENT_MASTER_MODE_SELECT : EV5
- I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED : EV6
- I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED : EV6
- I2C_EVENT_MASTER_BYTE_RECEIVED : EV7
- I2C_EVENT_MASTER_BYTE_TRANSMITTING : EV8
- I2C_EVENT_MASTER_BYTE_TRANSMITTED : EV8_2
- I2C_EVENT_MASTER_MODE_ADDRESS10 : EV9

图 14-14 I2C_CheckEvent 参数说明

3. 等待 EEPROM 内部写入完成

现在我们重新回到 I2C_EE_BufferWrite() 这个函数，在每次调用完 I2C_EE_PageWrite() 后，都调用了一个用户函数 I2C_EE_WaitEepromStandbyState()，见代码清单 14-11。

代码清单 14-11 I2C_EE_WaitEepromStandbyState() 函数

```

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

```

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

4. EEPROM 读取函数

关于 EEPROM 的数据读取函数 I2C_EE_BufferRead()，它的实现与写入函数类似的，请读者尝试亲自分析吧，见代码清单 14-12。

代码清单 14-12 I2C_EE_BufferRead() 函数

```

1.  /*
2.  * 函数名: I2C_EE_BufferRead
3.  * 描述  : 从 EEPROM 里面读取一块数据。
4.  * 输入  : -pBuffer 存放从 EEPROM 读取的数据的缓冲区指针。
5.  *          -WriteAddr 接收数据的 EEPROM 的地址。
6.  *          -NumByteToWrite 要从 EEPROM 读取的字节数。
7.  * 输出  : 无
8.  * 返回  : 无
9.  * 调用  : 外部调用
10. */
11. void I2C_EE_BufferRead(u8* pBuffer, u8 ReadAddr, u16 NumByteToRead)

```

```

12. {
13.     /*((u8 *)0x4001080c) |=0x80;
14.     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)); // Added by Najoua 27/08/2008
15.
16.
17.     /* Send START condition */
18.     I2C_GenerateSTART(I2C1, ENABLE);
19.     /*((u8 *)0x4001080c) &= ~0x80;
20.
21.     /* Test on EV5 and clear it */
22.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
23.
24.     /* Send EEPROM address for write */
25.     I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);
26.
27.     /* Test on EV6 and clear it */
28.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
29.
30.     /* Clear EV6 by setting again the PE bit */
31.     I2C_Cmd(I2C1, ENABLE);
32.
33.     /* Send the EEPROM's internal address to write to */
34.     I2C_SendData(I2C1, ReadAddr);
35.
36.     /* Test on EV8 and clear it */
37.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
38.
39.     /* Send STRAT condition a second time */
40.     I2C_GenerateSTART(I2C1, ENABLE);
41.
42.     /* Test on EV5 and clear it */
43.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
44.
45.     /* Send EEPROM address for read */
46.     I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Receiver);
47.
48.     /* Test on EV6 and clear it */
49.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
50.
51.     /* While there is data to be read */
52.     while(NumByteToRead)
53.     {
54.         if(NumByteToRead == 1)
55.         {
56.             /* Disable Acknowledgement */
57.             I2C_AcknowledgeConfig(I2C1, DISABLE);
58.
59.             /* Send STOP Condition */
60.             I2C_GenerateSTOP(I2C1, ENABLE);
61.         }
62.
63.         /* Test on EV7 and clear it */
64.         if(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED))
65.         {
66.             /* Read a byte from the EEPROM */
67.             *pBuffer = I2C_ReceiveData(I2C1);
68.
69.             /* Point to the next location where the byte read will be saved */
70.             pBuffer++;
71.

```

```

72.      /* Decrement the read bytes counter */
73.      NumByteToRead--;
74.  }
75.  }
76.
77.  /* Enable Acknowledgement to be ready for another reception */
78.  I2C_AcknowledgeConfig(I2C1, ENABLE);
79.}

```

这个读 EEPROM 函数与写的类似，也是利用 I2C_CheckEvent() 来确保通信正常进行，要注意的是读取数据时遵循 I²C 的标准，主发送器 STM32 要发出两次起始 I²C 信号才能建立通信。在这个函数中的第 18 行和第 40 行各产生了一次起始信号。

14.3.6 使用 I²C 读写 EEPROM 流程总结

最后，总结一下在 STM32 如何建立与 EEPROM 的通信。

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

2) 写：

- ① 检测 SDA 是否空闲。
- ② 按 I²C 协议发出起始信号。
- ③ 发出 7 位器件地址和写模式。
- ④ 要写入的存储区首地址。
- ⑤ 用页写入方式或字节写入方式写入数据。
- ⑥ 发送 I²C 通信结束信号。

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

3) 读：

- ① 检测 SDA 是否空闲。
- ② 按 I²C 协议发出起始信号。
- ③ 发出 7 位器件地址和写模式（伪写）。
- ④ 发出要读取的存储区首地址。
- ⑤ 重发起始信号。
- ⑥ 发出 7 位器件地址和读模式。
- ⑦ 接收数据。

类似写操作，每个操作之后要检测“事件”是否成功。

14.3.7 实验现象

将配套 STM32 开发板供电（DC5V），插上 J-LINK，插上串口线（两头都是母的交叉线），打开超级终端，配置超级终端为“115200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如图 14-15 的信息。

读出的数据见图 14-16。校验结果：读取的数据与写入的一致，实验成功！



图 14-15 写入数据图



图 14-16 读出数据图



第 15 章

SPI 模块

15.1 SPI 协议简介

SPI 协议 (Serial Peripheral Interface), 即串行外围设备接口, 是一种高速全双工的通信总线, 它由摩托罗拉公司提出, 当前最新的为 V04.01—2004 版。它被广泛地使用在 ADC、LCD 等设备与 MCU 间通信的场合。

15.1.1 SPI 信号线

SPI 包含 4 条总线, 分别为 SS、SCK、MOSI、MISO。它们的作用介绍如下:

1) SS (Slave Select): 片选信号线, 当有多个 SPI 设备与 MCU 相连时, 每个设备的这个片选信号线是与 MCU 单独的引脚相连的, 而其他的 SCK、MOSI、MISO 线则为多个设备并联到相同的 SPI 总线上, 见图 15-1。当 SS 信号线为低电平时, 片选有效, 开始 SPI 通信。

2) SCK (Serial Clock): 时钟信号线, 由主通信设备产生, 不同的设备支持的时钟频率不一样, 如 STM32 的 SPI 时钟频率最大为 $f_{PCLK}/2$ 。

3) MOSI (Master Output, Slave Input): 主设备输出 / 从设备输入引脚。主机的数据从这条信号线输出, 从机由这条信号线读入数据, 即这条线上数据的方向为主机到从机。

4) MISO (Master Input, Slave Output): 主设备输入 / 从设备输出引脚。主机从这条信号线读入数据, 从机的数据则由这条信号线输出, 即在这条线上数据的方向为从机到主机。

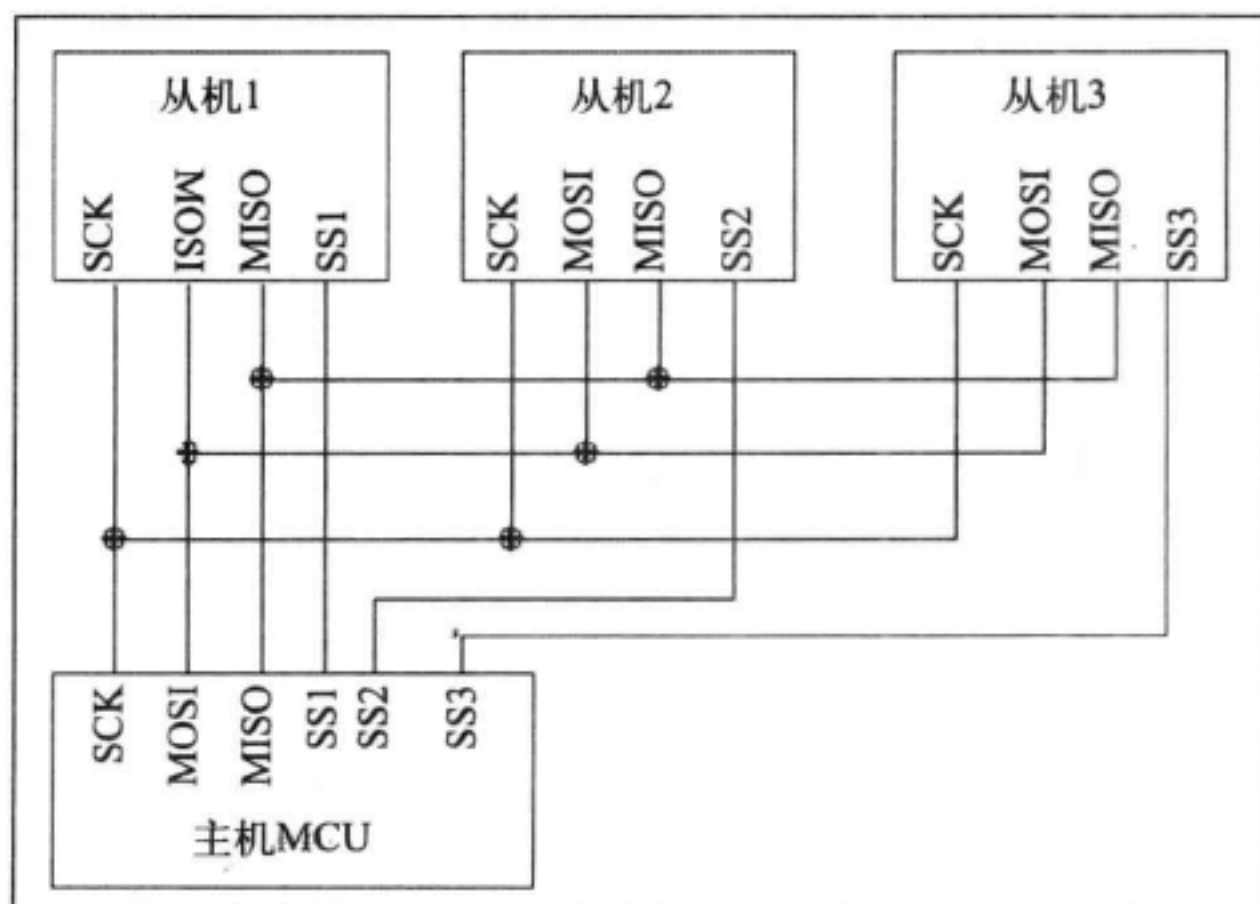


图 15-1 SPI 多设备通信

15.1.2 SPI 模式

根据 SPI 时钟极性 (CPOL) 和时钟相位 (CPHA) 配置的不同, 分为 4 种 SPI 模式。

时钟极性是指 SPI 通信设备处于空闲状态时 (也可以认为这是 SPI 通信开始时, 即 \overline{SS} 线为低电平时), SCK 信号线的电平信号。CPOL=0 时, SCK 在空闲状态时为低电平, CPOL=1 时则相反。

时钟相位是指数据采样的时刻, 当 CPHA=0 时, MOSI 或 MISO 数据线上的信号将会在 SCK 时钟线的奇数边沿被采样。当 CPHA=1 时, 数据线在 SCK 的偶数边沿采样。见图 15-2。

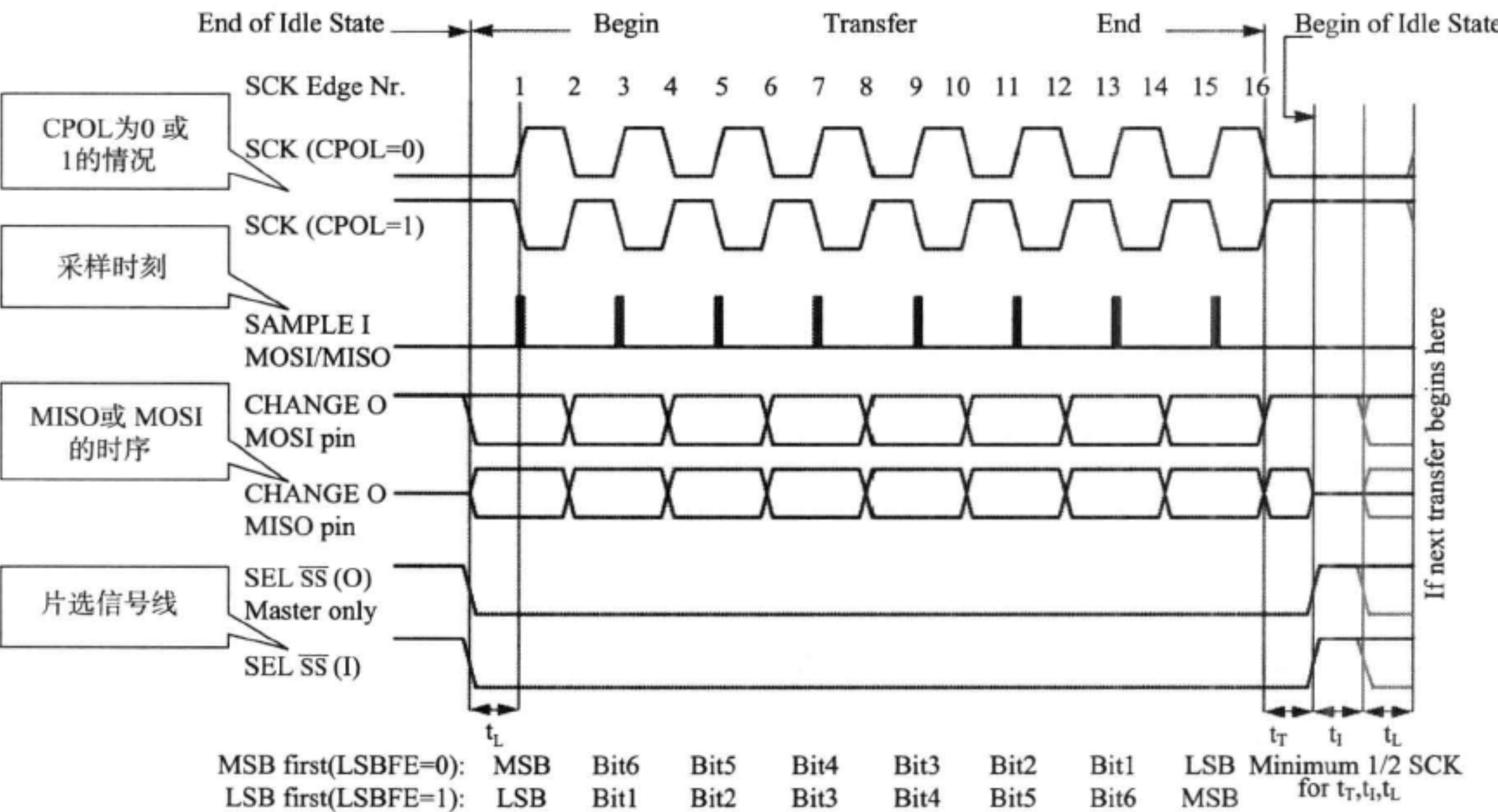


图 15-2 CPHA=0 时, SPI 时序

我们分析这个 CPHA=0 的时序图。

首先, 由主机把片选信号线 \overline{SS} 拉低, 即为图中的 \overline{SS} (O) 时序, 意为主机输出, \overline{SS} (I) 时序实际上也是 \overline{SS} 线信号, \overline{SS} (I) 时序表示从机接收到 \overline{SS} 片选被拉低的信号。

在 \overline{SS} 被拉低的时刻, SCK 分为两种情况。若我们设置为 CPOL=0, 则 SCK 时序在这个时刻为低电平, 若设置为 CPOL=1, 则 SCK 在这个时刻为高电平。

无论 CPOL 为 0 还是为 1, 因为我们配置的时钟相位 CPHA=0, 在采样时刻的时序中我们可以看到, 采样时刻都是在 SCK 的奇数边沿 (注意奇数边沿有时为下降沿, 有时为上升沿)。

因此, MOSI 和 MISO 数据线的有效信号在 SCK 的奇数边沿保持不变, 这个信号将在 SCK 奇数边沿时被采集, 在非采样时刻, MOSI 和 MISO 的有效信号才发生切换。

对于 CPHA=1 的情况也很类似, 但数据信号的采样时刻为偶数边沿, 其时序图见图 15-3。使用 SPI 协议通信时, 主机和从机的时序要保持一致, 即两者都选择相同的 SPI 模式。

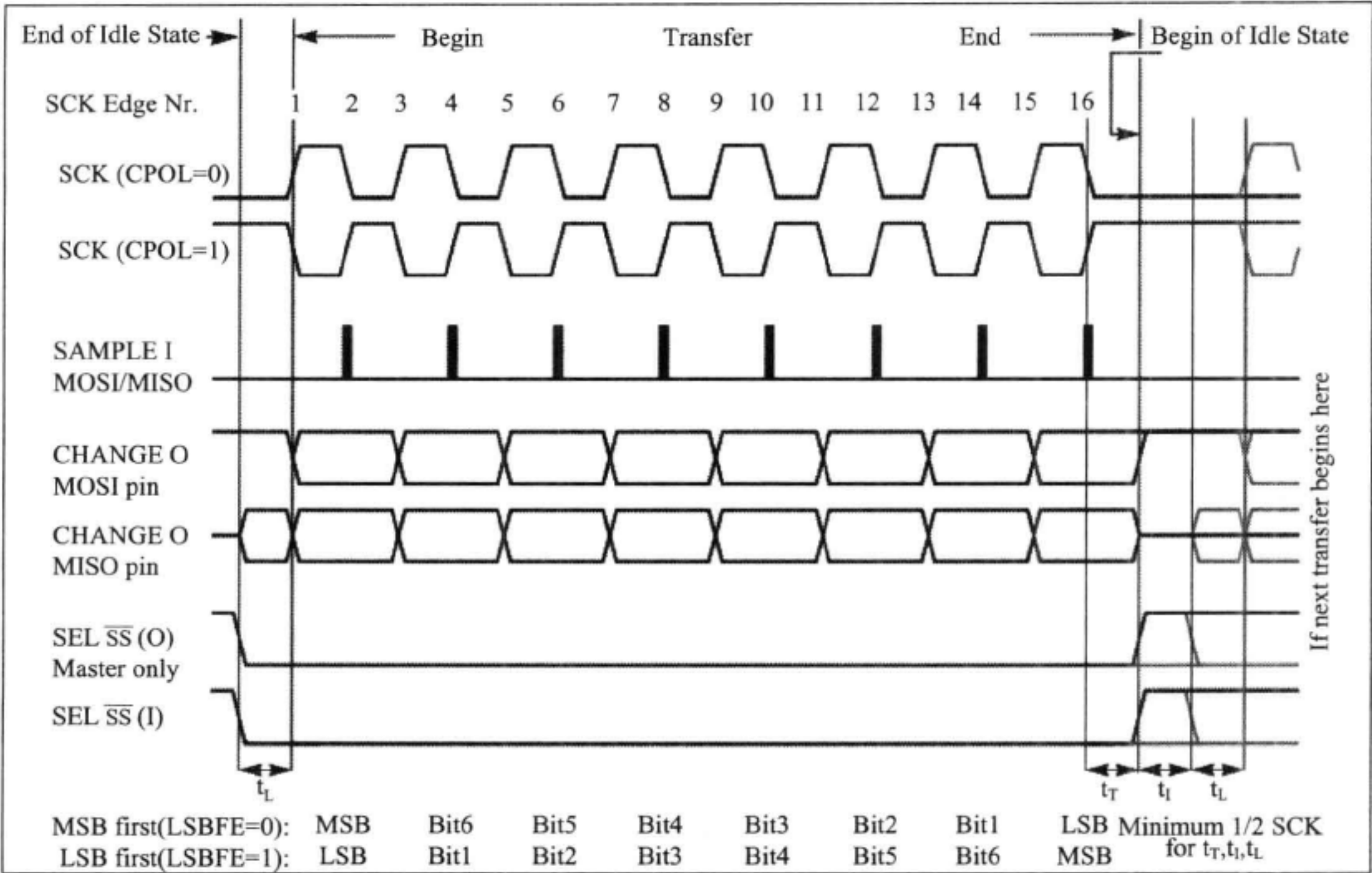


图 15-3 CPHA=1 时，SPI 时序

15.2 STM32 的 SPI 特性及架构

15.2.1 STM32 的 SPI 特性

STM32 的小容量产品有一个 SPI 接口，中容量的有两个，而大容量的则有 3 个。其特性如下：

- 单次传输可选择为 8 或 16 位。
- 波特率预分频系数（最大为 $f_{PCLK}/2$ ）。
- 时钟极性（CPOL）和相位（CPHA）可编程设置。
- 数据顺序的传输顺序可进行编程选择，MSB 在前或 LSB 在前。
- 可触发中断的专用发送和接收标志。
- 可以使用 DMA 进行数据传输操作。

15.2.2 STM32 的 SPI 架构分析

图 15-4 所示为 STM32 的 SPI 架构图，可以看到 MISO 数据线接收到的信号经移位寄存器处理后把数据转移到接收缓冲区，然后这个数据就可以由我们的软件从接收缓冲区读出了。

当要发送数据时，我们把数据写入发送缓冲区，硬件将会把它用移位寄存器处理后输出到 MOSI 数据线。

SCK 的时钟信号则由波特率发生器产生，我们可以通过波特率控制位（BR）来控制它输出的波特率。

控制寄存器 CR1 掌管着主控制电路，STM32 的 SPI 模块的协议设置（时钟极性、相位等）就是由它来制定的。而控制寄存器 CR2 则用于设置各种中断使能。

最后为 NSS 引脚，这个引脚扮演着 SPI 协议中的 S S 片选信号线的角色，如果我们把 NSS 引脚配置为硬件自动控制，SPI 模块能够自动判别它能否成为 SPI 的主机，或自动进入 SPI 从机模式。但实际上我们用得更多的是由软件控制某些 GPIO 引脚单独作为 S S 信号，这个 GPIO 引脚可以随便选择。

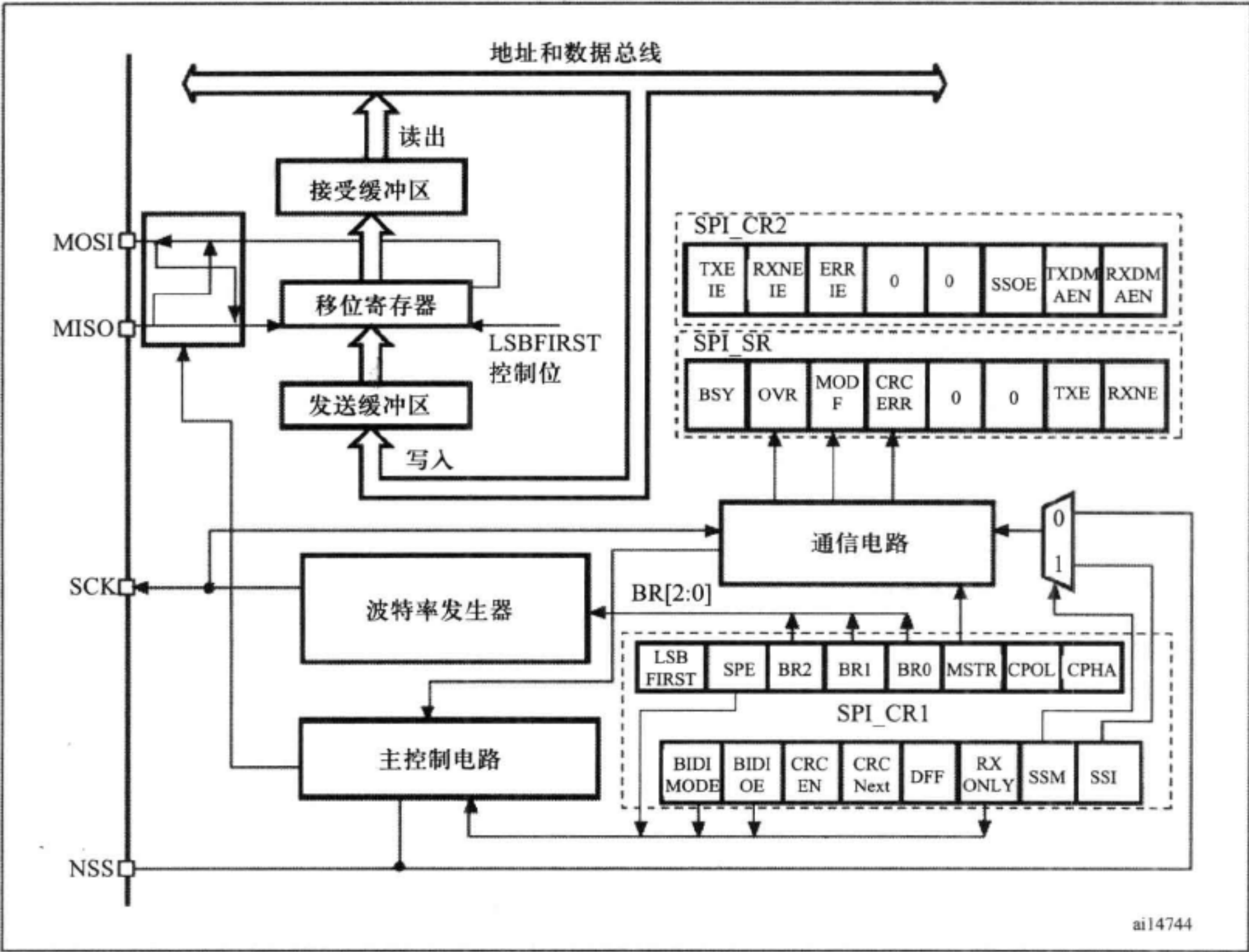


图 15-4 STM32 的 SPI 架构图

15.3 SPI 接口读取 Flash 实例分析

本章以 SPI 读写 Flash 的例程为大家讲解 SPI 的使用。

配套 STM32 开发板用的是 STM32F103VET6，它有 3 个 SPI 接口。本实验使用 SPI1，各信号线相应连接到 Flash（型号：W25X16）的 CS、CLK、DO 和 DIO 线，实现 SPI 通信，对 Flash 进行读写。

本实验采用主模式，全双工通信。通过查询发送数据寄存器和接收数据寄存器状态确保通信正常。

15.3.1 实验描述及工程文件清单

1. 实验描述

读取 Flash 的 ID 信息，写入数据，并读取出来进行校验，通过串口打印写入与读取出来的数据，输出测试结果。

2. 硬件连接

- ☐ PA4-SPI1-NSS : W25X16-CS
- ☐ PA5-SPI1-SCK : W25X16-CLK
- ☐ PA6-SPI1-MISO : W25X16-DO
- ☐ PA7-SPI1-MOSI : W25X16-DIO

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_spi.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/usart1.c
- ☐ USER/spi_flash.c

SPI-FLASH 硬件原理图见图 15-5。

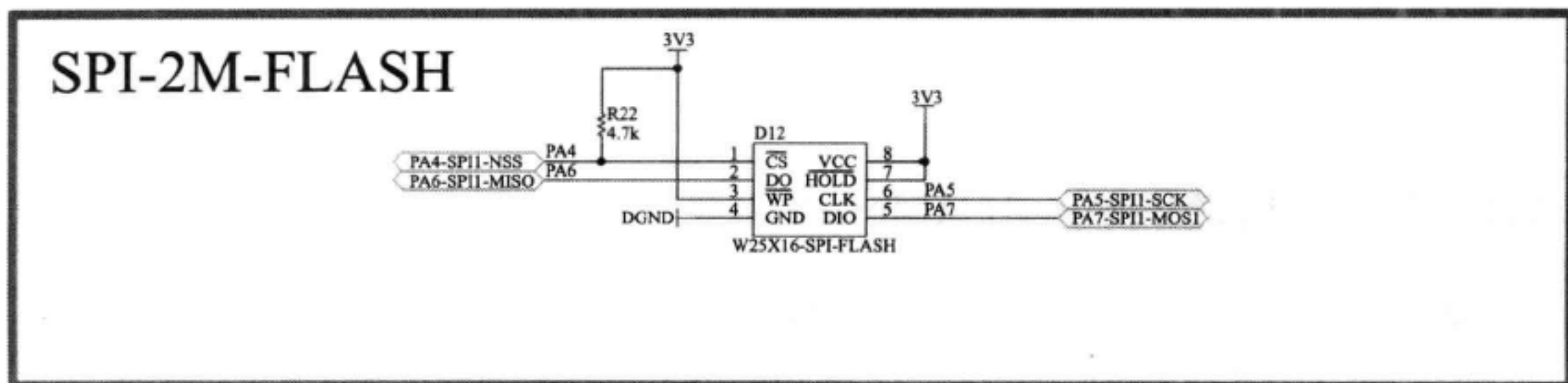


图 15-5 配套 STM32 开发板 SPI-FLASH 硬件原理图

15.3.2 配置工程环境

本 SPI-2M-FLASH 实验中我们用到了 GPIO、RCC、USART 及 SPI 外设，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_spi.c。本实验中没有使用中断，采用轮询标志位的方式来确保 SPI 正常通信。

接下来添加旧工程中的外设用户文件 usart.c，以便调试和观察实验效果。新建 spi_flash.c 及 spi_flash.h 文件，并在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 15-1。

代码清单 15-1 Flash 例程的 stm32f10x_conf.h 文件配置

```

1.  **
2.  ****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  ****/
9. #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_spi.h"
12. #include "stm32f10x_usart.h"

```

15.3.3 main 文件

配置好所需的库文件之后，我们就从 main 函数开始，分析见代码清单 15-2。

代码清单 15-2 Flash 例程的 main 函数

```

1. /*
2.  * 函数名：main
3.  * 描述   ：主函数
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7. int main(void)
8. {
9.
10.  /* 配置串口 1 为：115200 8-N-1 */
11.  USART1_Config();
12.  printf("\r\n 这是一个 2M 串行 flash (W25X16) 实验 \r\n");
13.
14.  /* 2M 串行 flash W25X16 初始化 */
15.  SPI_FLASH_Init();
16.
17.  /* Get SPI Flash Device ID */
18.  DeviceID = SPI_FLASH_ReadDeviceID();
19.
20.  Delay( 200 );
21.
22.  /* Get SPI Flash ID */

```

```

23. FlashID = SPI_FLASH_ReadID();
24.
25. printf("\r\n FlashID is 0x%X, Manufacturer Device ID is 0x%X\r\n", FlashID, DeviceID);
26.
27. /* Check the SPI Flash ID */
28. if (FlashID == sFLASH_ID) /* #define sFLASH_ID 0xEF3015 */
29. {
30.
31.     printf("\r\n 检测到华邦串行 flash W25X16 !\r\n");
32.
33.     /* Erase SPI FLASH Sector to write on */
34.     SPI_FLASH_SectorErase(FLASH_SectorToErase);
35.
36.     /* 将发送缓冲区的数据写到 flash 中 */
37.     SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, BufferSize);
38.     printf("\r\n 写入的数据为: %s \r\t", Tx_Buffer);
39.
40.     /* 将刚刚写入的数据读出来放到接收缓冲区中 */
41.     SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);
42.     printf("\r\n 读出的数据为: %s \r\n", Tx_Buffer);
43.
44.     /* 检查写入的数据与读出的数据是否相等 */
45.     TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);
46.
47.     if( PASSED == TransferStatus1 )
48.     {
49.         printf("\r\n 2M 串行 flash(W25X16) 测试成功!\n\r");
50.     }
51.     else
52.     {
53.         printf("\r\n 2M 串行 flash(W25X16) 测试失败!\n\r");
54.     }
55. } // if (FlashID == sFLASH_ID)
56. else
57. {
58.     printf("\r\n 获取不到 W25X16 ID!\n\r");
59. }
60.
61. SPI_Flash_PowerDown();
62. while(1);
63.}

```

这个例程的 main 函数很长，主要是因为后面有很多与实验验证有关的测试代码，实际上我们使用 SPI 读写 Flash 可以写得十分简洁。

沿着 main 函数的流程先走一遍，大家要清楚地认识到，本实验中 main 函数调用的所有函数都是用户函数。

- 1) 调用 USART1Config() 初始化串口。
- 2) 调用 SPI_FLASH_Init() 初始化 SPI 模块。
- 3) 调用 SPI_FLASH_ReadDeviceID() 读取 Flash 器件生产厂商的 ID 信息。
- 4) 调用 SPI_FLASH_ReadID() 读取 Flash 器件的设备 ID 信息。

读取器件的 ID 信息可以让我们知道设备与主机是否能够正常工作，也便于我们区分不同的器件。根据本实验使用的 datasheet 说明可以查询到器件的 ID，见表 15-1。我们可以在程序中进行验证。

表 15-1 Flash 数据手册的设备 ID 说明

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

5) 若读取得的 ID 正确，则调用 SPI_FLASH_SectorErase() 把 Flash 的内容擦除，擦除后调用 SPI_FLASH_BufferWrite() 向 Flash 写入数据，然后再调用 SPI_FLASH_BufferRead() 从刚刚写入的地址中读出数据。最后调用 Buffercmp() 函数对写入的数据与读取的数据进行比较，若写入的数据与读出的数据相同，则把标志变量 TransferStatus1 赋值为 PASSED（自定义的枚举变量）。

6) 根据标志变量 TransferStatus1 判断 Flash 数据的“擦除—写入—读取”是否正常，分情况输出测试信息到终端。

7) 若在读取 Flash 的 ID 信息时就出错，则直接向终端输出“检测不到 Flash”的信息。

8) 最后调用 SPI_Flash_PowerDown() 函数关闭 Flash 设备的电源，因为数据写入到 Flash 后并不会因断电而丢失，我们在使用它时才重新开启 Flash 的电源。

接下来我们详细分析 main 函数中调用的以上用户函数是怎样编写的。

15.3.4 SPI 初始化

SPI_FLASH_Init() 函数初始化了 SPI1 复用到的 GPIO 引脚，启动了 GPIO 及 SPI1 外设的时钟，并初始化了 SPI 的模式。实现见代码清单 15-3。

代码清单 15-3 SPI_FLASH_Init() 函数

```
1. /*****
2. * Function Name   : SPI_FLASH_Init
3. * Description    : Initializes the peripherals used by the SPI FLASH driver.
4. * Input          : None
5. * Output         : None
6. * Return         : None
7. *****/
8. void SPI_FLASH_Init(void)
9. {
10. SPI_InitTypeDef SPI_InitStructure;
11. GPIO_InitTypeDef GPIO_InitStructure;
12.
13. /* Enable SPI1 and GPIO clocks */
14. /*!< SPI_FLASH_SPI_CS_GPIO, SPI_FLASH_SPI_MOSI_GPIO,
```



```

15.     SPI_FLASH_SPI_MISO_GPIO, SPI_FLASH_SPI_DETECT_GPIO
16.     and SPI_FLASH_SPI_SCK_GPIO Periph clock enable */
17. RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOD, ENABLE);
18.
19. /*!< SPI_FLASH_SPI Periph clock enable */
20. RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
21.
22. /*!< Configure SPI_FLASH_SPI pins: SCK */
23. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
24. GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
25. GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
26. GPIO_Init(GPIOA, &GPIO_InitStructure);
27.
28. /*!< Configure SPI_FLASH_SPI pins: MISO */
29. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
30. GPIO_Init(GPIOA, &GPIO_InitStructure);
31.
32. /*!< Configure SPI_FLASH_SPI pins: MOSI */
33. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
34. GPIO_Init(GPIOA, &GPIO_InitStructure);
35.
36. /*!< Configure SPI_FLASH_SPI_CS_PIN pin: SPI_FLASH Card CS pin */
37. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
38. GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
39. GPIO_Init(GPIOA, &GPIO_InitStructure);
40.
41. /* Deselect the FLASH: Chip Select high */
42. SPI_FLASH_CS_HIGH();
43.
44. /* SPI1 configuration */
45. // W25X16: data input on the DIO pin is sampled on the rising edge of the CLK.
46. // Data on the DO and DIO pins are clocked out on the falling edge of CLK.
47. SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
48. SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
49. SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
50. SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
51. SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
52. SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
53. SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
54. SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
55. SPI_InitStructure.SPI_CRCPolynomial = 7;
56. SPI_Init(SPI1, &SPI_InitStructure);
57.
58. /* Enable SPI1 */
59. SPI_Cmd(SPI1, ENABLE);
60.}

```

SPI_FLASH_Init() 分为两部分。从开始到第 42 行为 GPIO 配置部分，之后为 SPI 模式配置。

1. GPIO 端口初始化

我们根据《STM32 数据手册》和《STM32 参考手册》把 PA5 (SCK)、PA6 (MISO)、PA7 (MOSI) 配置成复用的推挽输出模式，而由于我们的 NSS 引脚使用的是软件模式，所以我们将 PA4 (NSS) 配置成通用推挽输出。见表 15-2 和表 15-3。

表 15-2 SPI1 引脚定义

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCS64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
J3	G3	H7	20	29	40	PA4	I/O		PA4	SPI1_NSS/USART2_CK DAC_OUT1/ADC12_IN4	
K3	H3	E5	21	30	41	PA5	I/O		PA5	SPI1_SCK DAC_OUT2/ ADC12_IN5	
L3	J3	G5	22	31	42	PA6	I/O		PA6	SPI1_MISO /TIM8_BKIN ADC12_IN6/TIM3_CH1	TIM1_BKIN
M3	K3	G4	23	32	43	PA7	I/O		PA7	SPI1_MOSI /TIM8_CH1N ADC12_IN7/TIM3_CH2	TIM1_CH1N

表 15-3 SPI 的 GPIO 引脚复用设置

SPI引脚	配 置	GPIO配置
SPIx_SCK	主模式	推挽复用输出
	从模式	浮空输入
SPIx_MOSI	全双工模式 / 主模式	推挽复用输出
	全双工模式 / 从模式	浮空输入或带上拉输入
	简单的双向数据线 / 主模式	推挽复用输出
	简单的双向数据线 / 从模式	未用，可作为通用 I/O
SPIx_MISO	全双工模式 / 主模式	浮空输入或带上拉输入
	全双工模式 / 从模式	推挽复用输出
	简单的双向数据线 / 主模式	未用，可作为通用 I/O
	简单的双向数据线 / 从模式	推挽复用输出
SPIx_NSS	硬件主 / 从模式	浮空输入或带上拉输入或带 下拉输入
	硬件主模式 /NSS 输出使能	推挽复用输出
	软件模式	未用，可作为通用 I/O

在本函数中第 42 行的 SPI_FLASH_CS_HIGH() 实际上是一个自定义的宏，如下：

```
1. #define SPI_FLASH_CS_HIGH()      GPIO_SetBits(GPIOA, GPIO_Pin_4)
```

实际上这个宏是用来把 SPI 的 NSS（PA4）引脚拉高，从而禁止 SPI 的通信，在我们需要进行 SPI 通信时，我们会使用下面的宏把 NSS（PA4）引脚置低：

```
1. #define SPI_FLASH_CS_LOW()      GPIO_ResetBits(GPIOA, GPIO_Pin_4)
```

这样控制片选信号的方式，就是前面提到的软件控制方式，如果我们连接了多个 SPI 设备，这个片选信号可以由其他 GPIO 引脚产生。

2. SPI 模式初始化

从代码的第 48 行至第 56 行，为 SPI 模式的初始化。对 STM32 的 SPI 初始化配置，是根据将要与之通信的 Flash 设备的 SPI 特性来制定的。初始化时，有如下相关的结构体成员。

1) SPI_Mode : STM32 的 SPI 设备可以工作于主机模式 (SPI_Mode_Master) 或从机模式 (SPI_Mode_Slave)，这两个模式的重大区别为 SPI 的 SCK 信号线的时序，SCK 的时序是由通信中的主机产生的。若被配置为从机模式，STM32 的 SPI 模块将接受外来的 SCK 信号。本实验中 STM32 作为 SPI 通信中的主机，我们向这个成员赋值为主机模式 (SPI_Mode_Master)。

2) SPI_DataSize : 这个成员可以选择 SPI 每次通信的数据大小 (称为数据帧) 为 8 位还是 16 位。从 Flash 的数据手册我们可以查到，本 Flash 通信的数据帧大小为 8 位，STM32 的 SPI 模块设置要与之相同。

3) SPI_CPOL 和 SPI_CPHA : 这两个成员即配置 SPI 的时钟极性 (CPOL) 和时钟相位 (CPHA)，这两个配置影响到 SPI 的通信模式，该设置要符合将要互相通信的设备的要求。CPOL 分别可以取 SPI_CPOL_High (SPI 通信空闲时 SCK 为高电平) 和 SPI_CPOL_Low (SPI 通信空闲时 SCK 为低电平)。CPHA 则可以取 SPI_CPHA_1Edge (在 SCK 的奇数边沿采集数据) 和 SPI_CPHA_2Edge (在 SCK 的偶数边沿采集数据)。

查阅本 Flash 的使用手册，见图 15-6。可以了解到这个 Flash 支持以 SPI 的模式 0 和模式 3 通信。即在 SPI 空闲时，SCK 为低电平，奇数边沿采样 (模式 0)；也可以在 SPI 空闲时，SCK 为高电平，偶数边沿采样 (模式 3)。即无论 CPOL 的状态是什么，Flash 的数据采样时刻为 SCK 的上升沿。

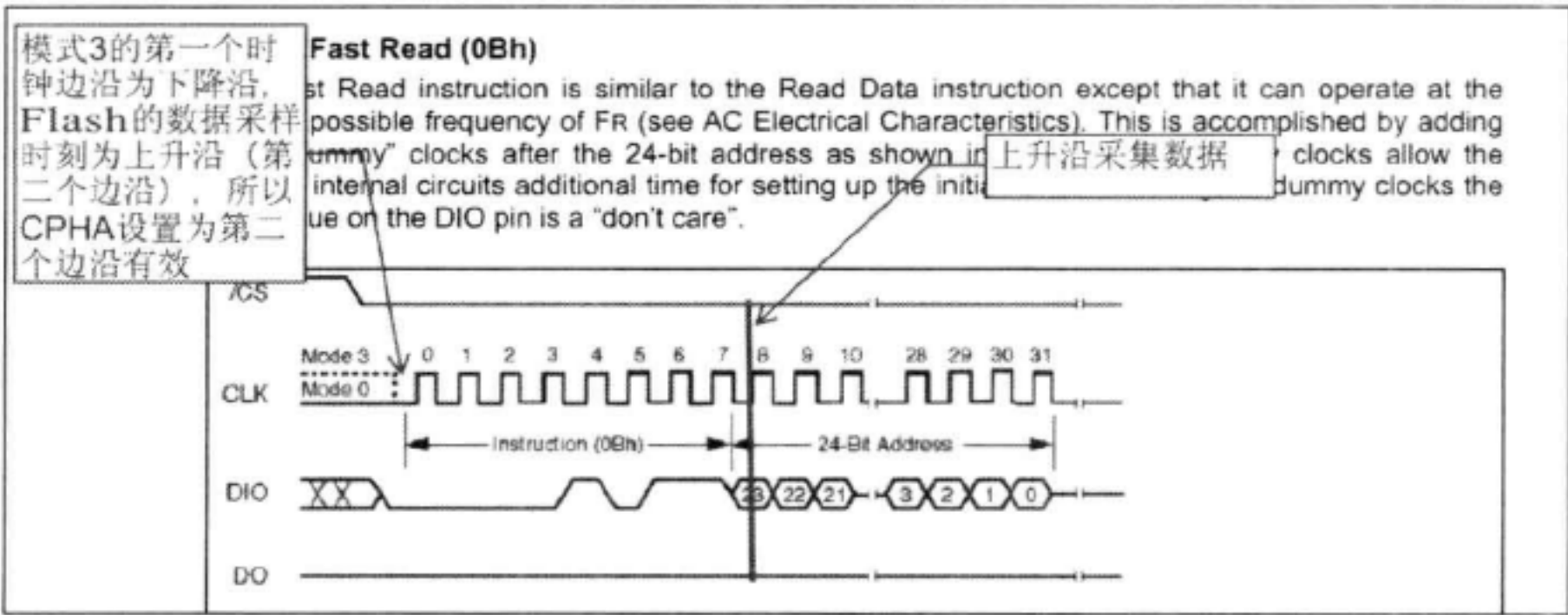


图 15-6 Flash 时序说明图

我们在本实验配置使用它的模式 3，即把 CPOL 赋值为 SPI_CPOL_High；CPHA 赋值为 SPI_CPHA_2Edge。

4) SPI_NSS : 本成员配置 NSS 引脚的使用模式，可以选择为硬件模式 (SPI_NSS_Hard) 与软件模式 (SPI_NSS_Soft)，在硬件模式中的 SPI 片选信号由硬件自动产生，而软件模式则需要我

们亲自把相应的 GPIO 端口拉高或置低产生非片选和片选信号。如果外界条件允许，硬件模式还会自动将 STM32 的 SPI 设置为主机。本实验使用软件模式，向这个成员赋值为 SPI_NSS_Soft。

5) SPI_BaudRatePrescaler：本成员设置波特率分频值，分频后的时钟即为 SPI 的 SCK 信号线的时钟频率。这个成员参数可设置为 f_{PCLK} 的 2、4、6、8、16、32、64、128、256 分频。本实验向这个成员赋值为 SPI_BaudRatePrescaler_4，即 f_{PCLK} 的 4 分频。

6) SPI_FirstBit：所有串行的通信协议都会有 MSB 先行（高位数据在前）还是 LSB 先行（低位数据在前）的问题，而 STM32 的 SPI 模块可以通过这个结构体成员，对这个特性编程控制。据 Flash 的通信时序，我们向这个成员赋值为 MSB 先行（SPI_FirstBit_MSB）。

7) SPI_CRCPolynomial：这是 SPI 的 CRC 校验中的多项式，若我们使用 CRC 校验时，就使用这个成员的参数（多项式）来计算 CRC 的值。由于本实验的 Flash 不支持 CRC 校验，所以我们向这个结构体成员赋值为 7 实际上是没有意义的。

配置完这些结构体成员后，我们要调用 SPI_Init() 函数把这些参数写入寄存器中，实现 SPI 的初始化，然后调用 SPI_Cmd() 来使能 SPI1 外设。

15.3.5 控制 Flash 的命令

实际上，编写设备的驱动都有一定的规律可循。

首先我们要确定设备使用的是什么通信协议。如第 14 章的 EEPROM 使用的是 I²C，本章的 Flash 使用的是 SPI。那么我们就先根据它的通信协议，选择好 STM32 的硬件模块，并进行相应的 I²C 或 SPI 模块初始化。

因为不同的设备都会相应的有不同的指令，如 EEPROM 中会把第一个数据解释为存储矩阵的地址（实质就是指令）。而 Flash 则定义了更多的指令，有写指令、读指令、读 ID 指令等。

对主机来说，这些指令只是它遵守最基本的通信协议发送出的数据。但设备把这些数据解释成不同的意义（指令编码），所以才成为指令。在我们配置好 STM32 的协议模块后，想要控制设备，就要遵守相应设备所定义的命令规则。

所以写驱动并不难：就是确定通信协议模块，通过协议收发命令、数据，进而驱动设备。我们把这个称为驱动编写原理。这也是设备厂商的设计准则。

我们先来查看 Flash 的各种指令和命令解释时序，见表 15-4。

表 15-4 Flash 命令

指 令	第一字节 (指令编码)	第二字节	第三字节	第四字节	第五字节	第六字节	第七-N 字节
Write Enable	06h						
Write Disable	04h						
Read Status Register	05h	(S7-S0)					
Write Status Register	01h	S7-S0					
Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0)	(Next byte)	continuous

(续)

指 令	第一字节 (指令编码)	第二字节	第三字节	第四字节	第五字节	第六字节	第七-N 字节
Fast Read	0Bh	A23-A16	A15-A8	A7-A0	dummy	(D7-D0)	(Next Byte) continuous
Fast Read Dual Output	3Bh	A23-A16	A15-A8	A7-A0	dummy	I/O = (D6,D4,D2,D0) O = (D7,D5,D3,D1)	(one byte per 4 clocks, continuous)
Page Program	02h	A23-A16	A15-A8	A7-A0	(D7-D0)	(Next byte)	Up to 256 bytes
Block Erase (64KB)	D8h	A23-A16	A15-A8	A7-A0			
Sector Erase (4KB)	20h	A23-A16	A15-A8	A7-A0			
Chip Erase	C7h						
Power-down	B9h						
Release Power- down / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0)		
Manufacturer/ Device ID	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)	
JEDEC ID	9Fh	(M7-M0) 生产厂商	(ID15-ID8) 存储器类型	(ID7-ID0) 容量			

指令表中的 A0 ~ A23 指地址, M0 ~ M7 为器件的制造商 ID (MANUFACTURERID), D0 ~ D7 为数据。

在命令列表中了解到读取设备 ID 的命令 (Device ID) 编码为 ABh、dummy、dummy、dummy。表示此命令由这四个字节组成, 其中 dummy 意为任意编码, 表示这些编码可以发送任意数据。命令列表中带括号的字节数据表示由 Flash 返回给主机的响应, 可以看到 Release Power-down/Device ID 命令的第 5 个字节为从机返回的响应, (ID7 ~ ID0) 即返回设备的 ID 号。

使用 Device ID 命令时的时序图见图 15-7, 可以看到主机首先通过 MOSI 线 (即 Flash 的 DIO 线) 发送第一个字节为 ABh 编码, 紧接着三个字节的 dummy 编码, 然后 Flash 就忽略 DIO 线上的信号, 通过 MISO 线 (即 Flash 的 DO 线) 把它的 Flash 设备 ID 发送给主机。

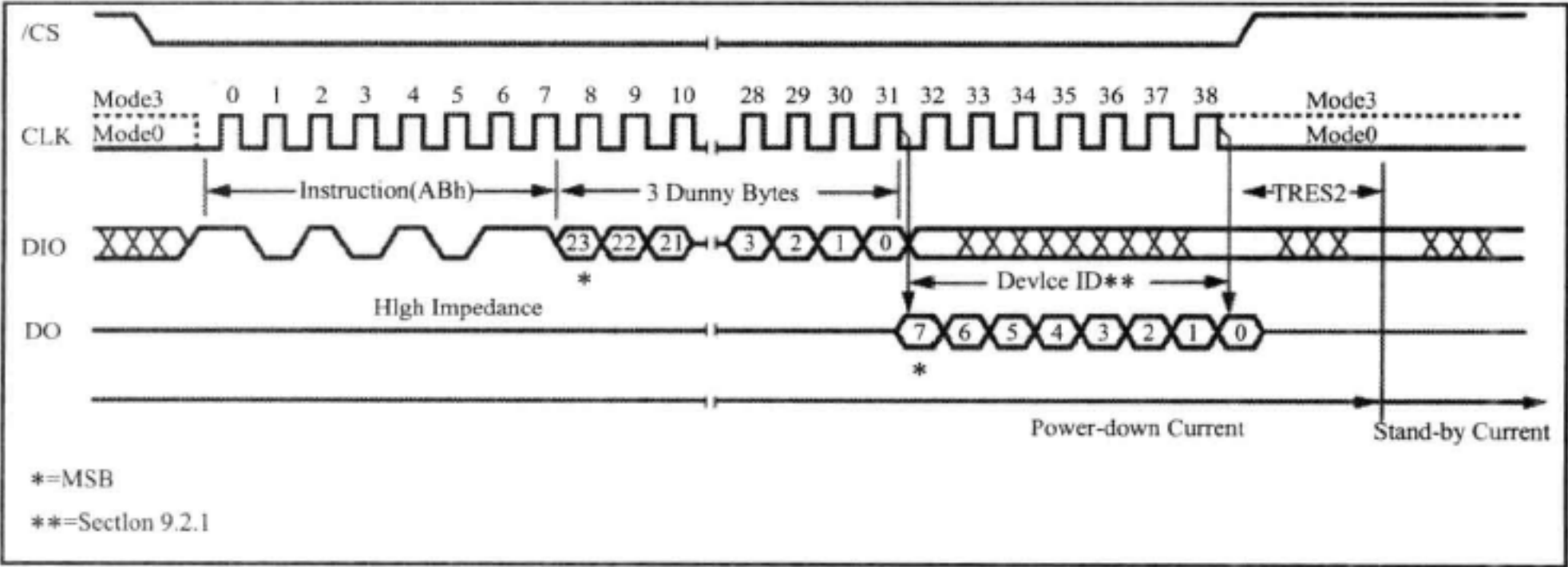


图 15-7 Flash 读 ID 命令解释时序

了解了 Device ID 命令及其时序，我们分析 SPI_FLASH_ReadDeviceID() 函数来说明驱动编写原理。这个函数实现了读取 Flash 的 ID 的功能，见代码清单 15-4。

代码清单 15-4 SPI_FLASH_ReadDeviceID() 函数

```

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

```

这个函数的代码流程严格遵从 Device ID 命令的时序：

1) 第 12 行，调用宏 SPI_FLASH_CS_LOW()，这是一个自定义的宏，使 Flash 的片选有效，以使能 Flash 设备。

2) 第 14 行，利用用户函数 SPI_FLASH_SendByte() 来向 Flash 发送第一个命令字节编码“W25X_DeviceID”（自定义的宏：0xAB）。

3) 根据指令表，发送完这个指令后，后面紧跟着三个字节的“dummy byte”，我们把 Dummy_Byte 宏定义为“0xff”，实际上改成其他宏编码也无影响。

4) 第 19 行，完整的命令在前面已经发送完毕，根据时序，在第 5 个字节 Flash 通过 DO 端口输出它的器件 ID，我们调用函数 SPI_FLASH_SendByte() 接收返回的数据，并赋值给 Temp 变量。SPI_FLASH_ReadDeviceID() 函数的返回值即为读取得的器件 ID。

5) 把片选信号拉高，结束通信。

这就完成了读 Flash 的 ID。在这个读 Flash ID 函数中多次调用了一个相对底层的用户函数，它实现了利用 SPI 发送和接收数据的功能，见代码清单 15-5。

代码清单 15-5 SPI_FLASH_SendByte() 函数

```

1.  /*****
2.  * Function Name   : SPI_FLASH_SendByte
3.  * Description    : Sends a byte through the SPI interface and return the byte

```

```

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

```

-
- 1) 调用库函数 SPI_I2S_GetFlagStatus() 等待发送数据寄存器清空。
 - 2) 发送数据寄存器准备好后，调用库函数 SPI_I2S_SendData() 向从机发送数据。
 - 3) 调用库函数 SPI_I2S_GetFlagStatus() 等待接收数据寄存器非空。
 - 4) 接收寄存器非空时，调用 SPI_I2S_ReceiveData() 获取接收寄存器中的数据并作为函数的返回值，这个数据即由从机发送给主机的数据。

这是最底层的发送数据和接收数据的函数，利用了库函数的标志检测确保通信正常。

15.3.6 读取厂商 ID

对于其他函数，编写的方法是类似的，如读取厂商 ID 的函数 SPI_FLASH_ReadID() 见代码清单 15-6。

代码清单 15-6 SPI_FLASH_ReadID() 函数

```

1. /***/
2. * Function Name : SPI_FLASH_ReadID
3. * Description  : Reads FLASH identification.
4. * Input       : None
5. * Output      : None
6. * Return      : FLASH identification
7. *****/
8. u32 SPI_FLASH_ReadID(void)
9. {
10.     u32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;
11.
12.     /* Select the FLASH: Chip Select low */
13.     SPI_FLASH_CS_LOW();
14.
15.     /* Send "RDID " instruction */
16.     SPI_FLASH_SendByte(W25X_JedecDeviceID);
17.
18.     /* Read a byte from the FLASH */

```

```

19. Temp0 = SPI_FLASH_SendByte(Dummy_Byte);
20.
21. /* Read a byte from the FLASH */
22. Temp1 = SPI_FLASH_SendByte(Dummy_Byte);
23.
24. /* Read a byte from the FLASH */
25. Temp2 = SPI_FLASH_SendByte(Dummy_Byte);
26.
27. /* Deselect the FLASH: Chip Select high */
28. SPI_FLASH_CS_HIGH();
29.
30. Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;
31.
32. return Temp;
33.}

```

这个函数根据命令流程，发送一个字节的命令编码“JEDEC ID (9Fh)”后，从机就通过 DO 线返回厂商 ID 及 0 ~ 16 位的设备 ID。图 15-8 为读厂商 ID 的时序图。

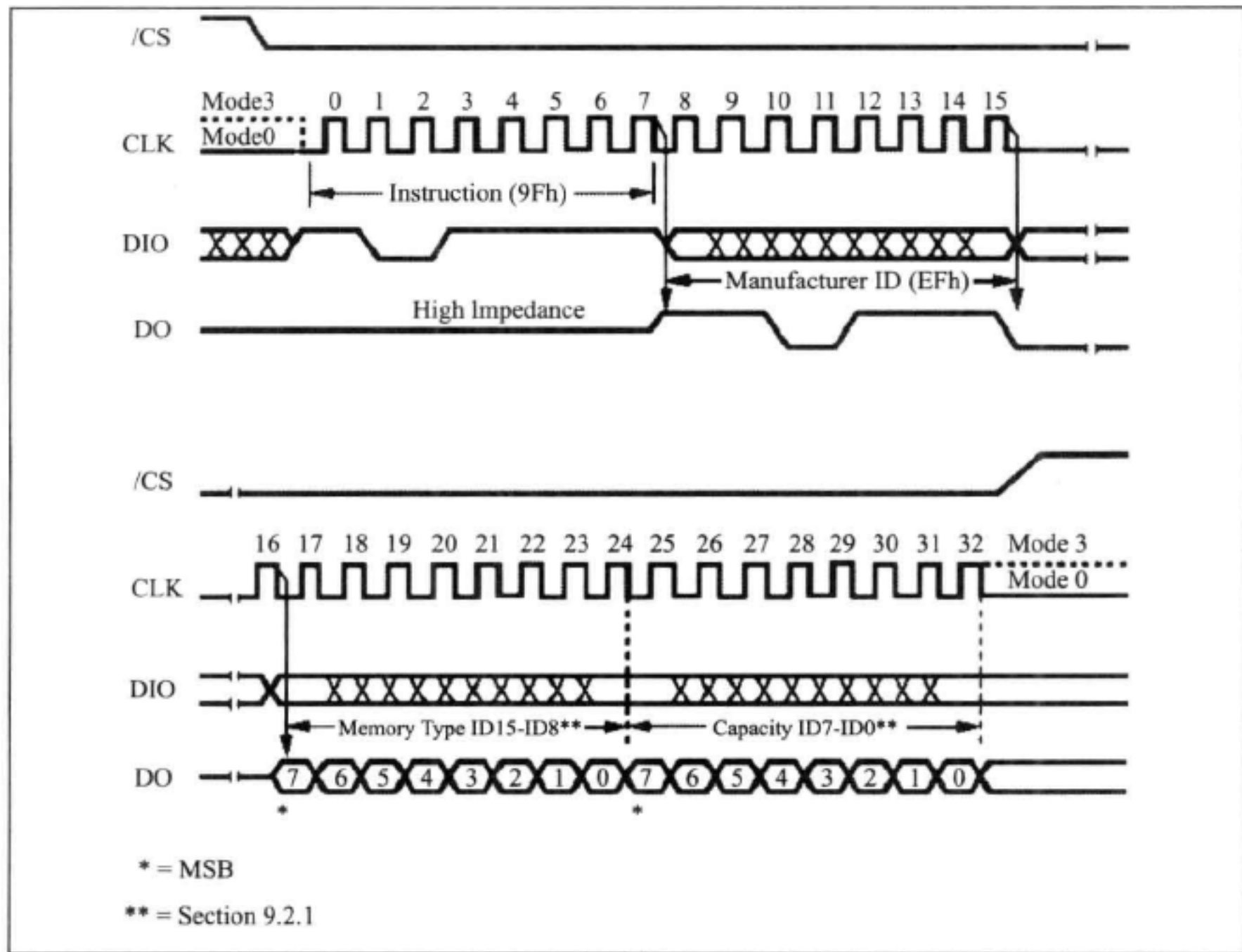


图 15-8 读厂商 ID 时序图

15.3.7 擦除 Flash 内容

1. 扇区擦除

根据 Flash 的存储原理，在写入数据前要先对存储区域进行擦除，所以执行 SPI_FLASH_SectorErase() 函数对要写入的扇区进行擦除，也称为预写。见代码清单 15-7。

代码清单 15-7 SPI_FLASH_SectorErase() 函数

```
1. /*****
2. * Function Name   : SPI_FLASH_SectorErase
3. * Description    : Erases the specified FLASH sector.
4. * Input          : SectorAddr: address of the sector to erase.
5. * Output         : None
6. * Return         : None
7. *****/
8. void SPI_FLASH_SectorErase(u32 SectorAddr)
9. {
10. /* Send write enable instruction */
11. SPI_FLASH_WriteEnable();
12. SPI_FLASH_WaitForWriteEnd();
13. /* Sector Erase */
14. /* Select the FLASH: Chip Select low */
15. SPI_FLASH_CS_LOW();
16. /* Send Sector Erase instruction */
17. SPI_FLASH_SendByte(W25X_SectorErase);
18. /* Send SectorAddr high nibble address byte */
19. SPI_FLASH_SendByte((SectorAddr & 0xFF0000) >> 16);
20. /* Send SectorAddr medium nibble address byte */
21. SPI_FLASH_SendByte((SectorAddr & 0xFF00) >> 8);
22. /* Send SectorAddr low nibble address byte */
23. SPI_FLASH_SendByte(SectorAddr & 0xFF);
24. /* Deselect the FLASH: Chip Select high */
25. SPI_FLASH_CS_HIGH();
26. /* Wait the end of Flash writing */
27. SPI_FLASH_WaitForWriteEnd();
28. }
```

先忽略其中的 SPI_FLASH_WriteEnable() 和 SPI_FLASH_WaitForWriteEnd() 函数，从第 15 行至第 25 行是纯粹的关于 Flash 擦除操作，扇区擦除命令时序如图 15-9 所示。

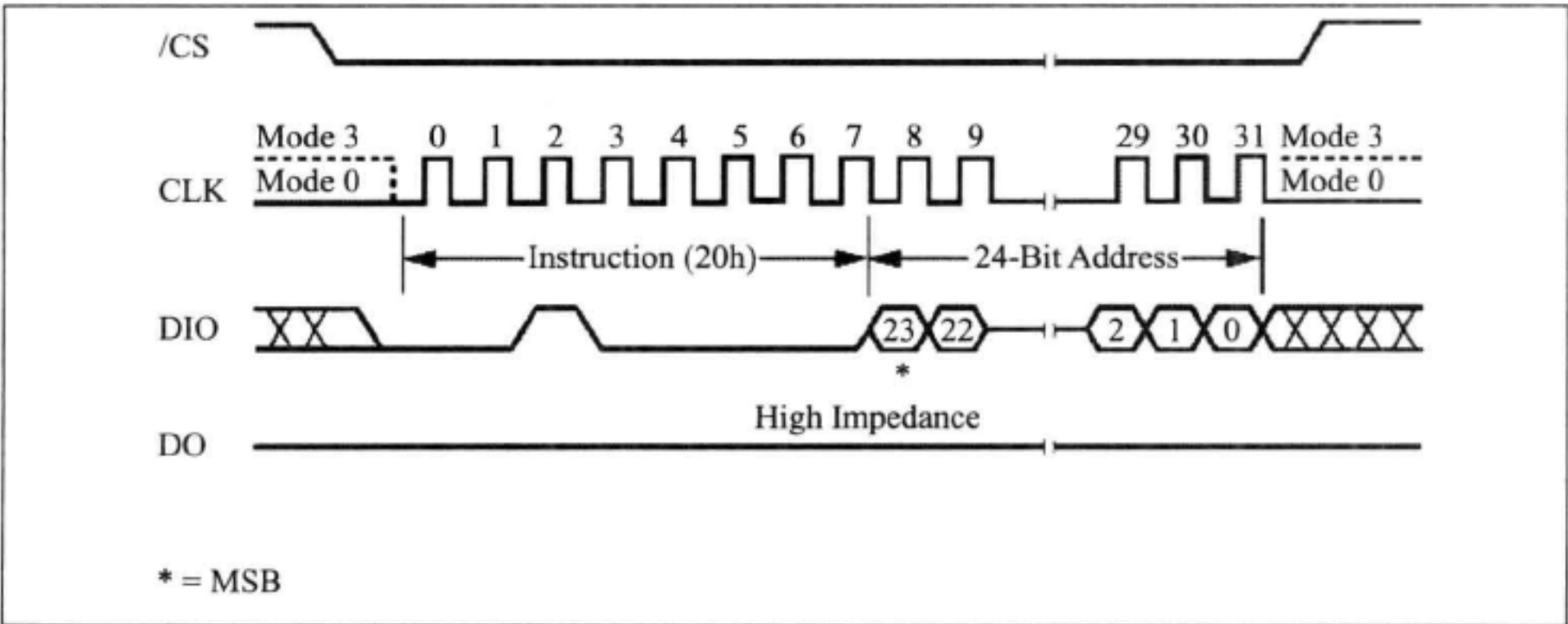


图 15-9 扇区擦除时序

其中第一个字节为扇区擦除命令编码（20h），紧跟其后的为要进行擦除的 24 位起始地址。根据 Flash 的说明，它把整个存储矩阵分为块区和扇区，每块（Block）的大小为 64 KB，每个扇区

(Sector) 的大小为 4 KB，对存储矩阵进行擦除时，最小的单位为扇区。由于篇幅问题，就不显示它的地址分布图了。

2. 写使能

根据 Flash 的读写要求，在进行页写入、扇区擦除、块擦除、整片擦除及写状态寄存器前，都需要先发送写使能命令。在 SPI_FLASH_SectorErase() 函数的第 11 行就调用了用户函数 SPI_FLASH_WriteEnable()，它的具体实现见代码清单 15-8。

代码清单 15-8 SPI_FLASH_WriteEnable() 函数

```
1. /*****
2.  * Function Name   : SPI_FLASH_WriteEnable
3.  * Description    : Enables the write access to the FLASH.
4.  * Input          : None
5.  * Output         : None
6.  * Return         : None
7. *****/
8. void SPI_FLASH_WriteEnable(void)
9. {
10.  /* Select the FLASH: Chip Select low */
11.  SPI_FLASH_CS_LOW();
12.
13.  /* Send "Write Enable" instruction */
14.  SPI_FLASH_SendByte(W25X_WriteEnable);
15.
16.  /* Deselect the FLASH: Chip Select high */
17.  SPI_FLASH_CS_HIGH();
18. }
```

本函数十分简单，就是根据写使能命令的时序，发送写使能命令 write enable (06h)。见图 15-10。

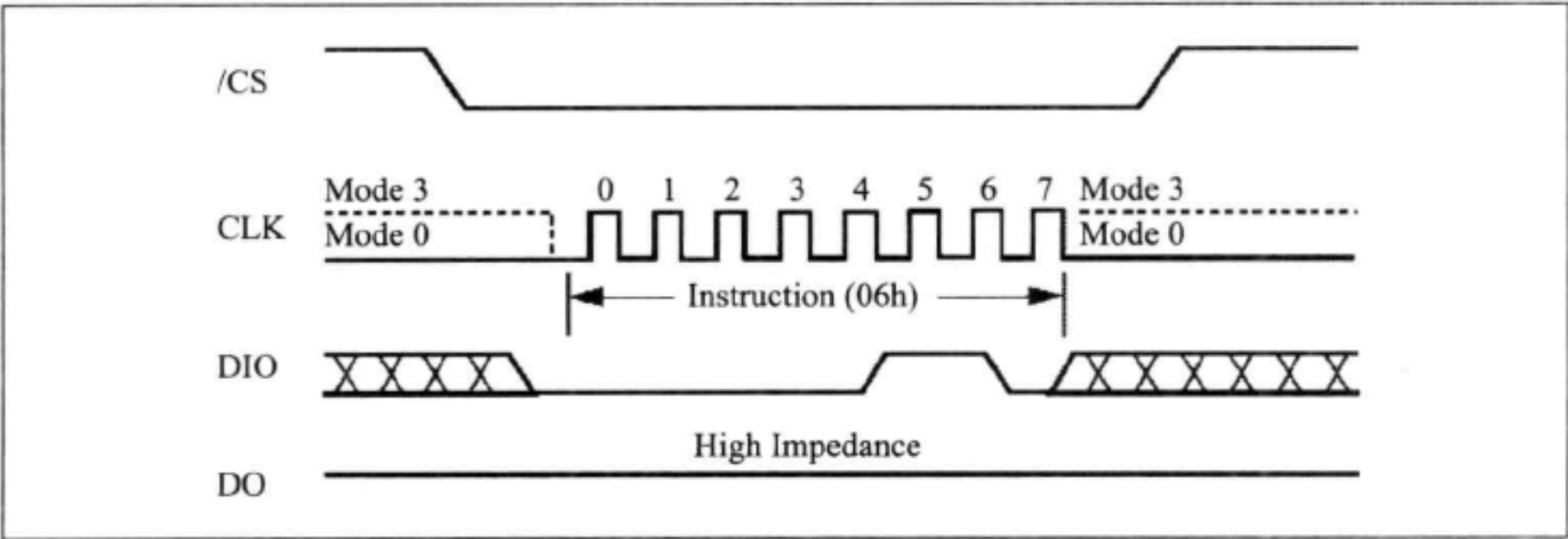


图 15-10 写使能命令时序

3. 读 Flash 状态

在擦除函数 SPI_FLASH_SectorErase() 中，还调用用户函数 SPI_FLASH_WaitFor WriteEnd() 来确保在 Flash 不忙碌的时候，才发送命令或数据。这个函数通过读取 Flash 的状态寄存器来获知它的工作状态，实现见代码清单 15-9。

代码清单 15-9 SPI_FLASH_WaitForWriteEnd() 函数

```

1.  /*****
2.  * Function Name   : SPI_FLASH_WaitForWriteEnd
3.  * Description    : Polls the status of the Write In_Progress (WIP) flag in the
4.  *                  FLASH's status register and loop until write operation
5.  *                  has completed.
6.  * Input          : None
7.  * Output         : None
8.  * Return         : None
9.  *****/
10. void SPI_FLASH_WaitForWriteEnd(void)
11. {
12.     u8 FLASH_Status = 0;
13.
14.     /* Select the FLASH: Chip Select low */
15.     SPI_FLASH_CS_LOW();
16.
17.     /* Send "Read Status Register" instruction */
18.     SPI_FLASH_SendByte(W25X_ReadStatusReg);
19.
20.     /* Loop as long as the memory is busy with a write cycle */
21.     do
22.     {
23.         /* Send a dummy byte to generate the clock needed by the FLASH
24.         and put the value of the status register in FLASH_Status variable */
25.         FLASH_Status = SPI_FLASH_SendByte(Dummy_Byte);
26.     }
27.     while ((FLASH_Status & WIP_Flag) == SET); /* Write in progress */
28.
29.     /* Deselect the FLASH: Chip Select high */
30.     SPI_FLASH_CS_HIGH();
31. }

```

本函数实质是不断循环地检测 Flash 状态寄存器的 Busy 位，直到 Flash 的内部写时序完成，从而确保下一通信操作正常。主机通过发送读状态寄存器命令 Read Status Register (05h)，返回它的 8 位状态寄存器的值，这个命令的时序及状态寄存器位的说明见图 15-11 和图 15-12。

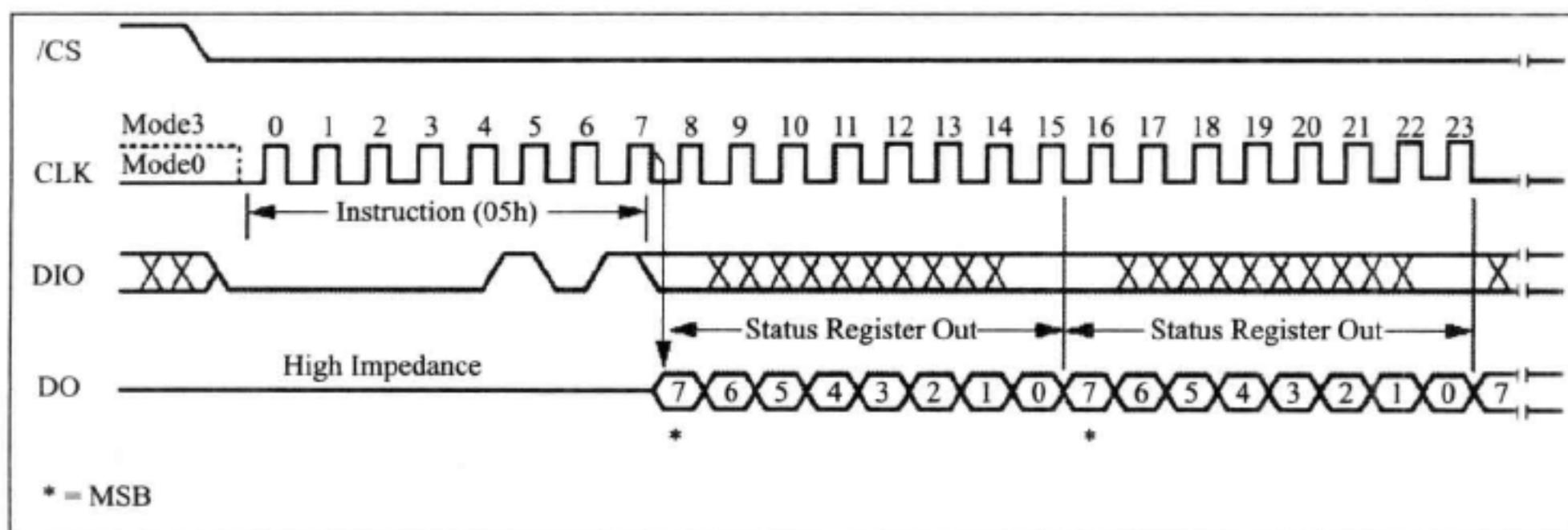


图 15-11 读状态寄存器时序

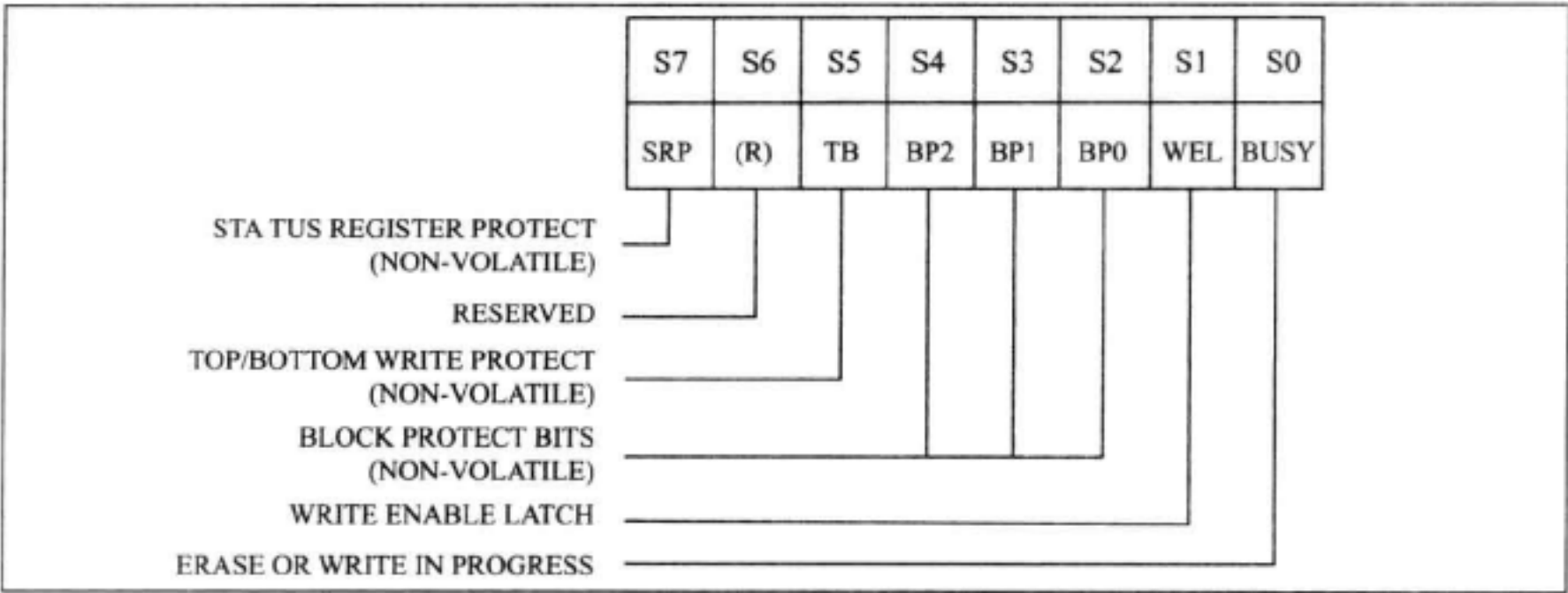


图 15-12 状态寄存器位说明

本函数检测的就是状态寄存器的 Bit0，即 BUSY 位。Flash 在执行内部写时序的时候，除了读状态寄存器命令，其他一切命令都会忽略，并且 BUSY 位保持为 1，即我们需要等待 BUSY 位为 0 的时候，再向 Flash 发送其他命令。

15.3.8 向 Flash 写入数据

对 Flash 写入数据，最小单位是 256 字节，厂商把这个单位称为页。写入时，一般也只有页写入的方式。因而我们为了方便地把一个很长的数组写入 Flash 中，一般需要进行转换，把数组按页分好，再写入 Flash 中，如同 I²C 例子中对 EEPROM 的页写入一样，只是页的大小不同而已，这个函数见代码清单 15-10。

代码清单 15-10 SPI_FLASH_BufferWrite() 函数

```
1.
2. /*****
3. * Function Name   : SPI_FLASH_BufferWrite
4. * Description    : Writes block of data to the FLASH. In this function, the
5. *                  number of WRITE cycles are reduced, using Page WRITE sequence.
6. * Input          : - pBuffer : pointer to the buffer containing the data to be
7. *                  written to the FLASH.
8. *                  - WriteAddr : FLASH' s internal address to write to.
9. *                  - NumByteToWrite : number of bytes to write to the FLASH.
10.* Output         : None
11.* Return        : None
12.*****/
13.void SPI_FLASH_BufferWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
14.{
15.    u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0, temp = 0;
16.
17.    Addr = WriteAddr % SPI_FLASH_PageSize;
18.    count = SPI_FLASH_PageSize - Addr;
19.    NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
20.    NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;
21.
22.    if (Addr == 0) /* WriteAddr is SPI_FLASH_PageSize aligned */
23.    {
24.        if (NumOfPage == 0) /* NumByteToWrite < SPI_FLASH_PageSize */
25.        {
```



```

26.     SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumByteToWrite);
27. }
28. else /* NumByteToWrite > SPI_FLASH_PageSize */
29. {
30.     while (NumOfPage--)
31.     {
32.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, SPI_FLASH_PageSize);
33.         WriteAddr += SPI_FLASH_PageSize;
34.         pBuffer += SPI_FLASH_PageSize;
35.     }
36.
37.     SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumOfSingle);
38. }
39. }
40. else /* WriteAddr is not SPI_FLASH_PageSize aligned */
41. {
42.     if (NumOfPage == 0) /* NumByteToWrite < SPI_FLASH_PageSize */
43.     {
44.         if (NumOfSingle > count) /* (NumByteToWrite + WriteAddr) > SPI_FLASH_PageSize */
45.         {
46.             temp = NumOfSingle - count;
47.
48.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
49.             WriteAddr += count;
50.             pBuffer += count;
51.
52.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, temp);
53.         }
54.         else
55.         {
56.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumByteToWrite);
57.         }
58.     }
59.     else /* NumByteToWrite > SPI_FLASH_PageSize */
60.     {
61.         NumByteToWrite -= count;
62.         NumOfPage = NumByteToWrite / SPI_FLASH_PageSize;
63.         NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;
64.
65.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
66.         WriteAddr += count;
67.         pBuffer += count;
68.
69.         while (NumOfPage--)
70.         {
71.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, SPI_FLASH_PageSize);
72.             WriteAddr += SPI_FLASH_PageSize;
73.             pBuffer += SPI_FLASH_PageSize;
74.         }
75.
76.         if (NumOfSingle != 0)
77.         {
78.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumOfSingle);
79.         }
80.     }
81. }
82. }

```

在 SPI_FLASH_BufferWrite() 中，对数组进行分页后，它调用了用户函数 SPI_FLASH_PageWrite() 来对数据进行按页写入，见代码清单 15-11。

代码清单 15-11 SPI_FLASH_PageWrite() 函数

```

1.  /*****
2.  * Function Name   : SPI_FLASH_PageWrite
3.  * Description    : Writes more than one byte to the FLASH with a single WRITE
4.  *                  cycle(Page WRITE sequence). The number of byte can't exceed
5.  *                  the FLASH page size.
6.  * Input          : - pBuffer : pointer to the buffer containing the data to be
7.  *                  written to the FLASH.
8.  *                  - WriteAddr : FLASH's internal address to write to.
9.  *                  - NumByteToWrite : number of bytes to write to the FLASH,
10. *                  must be equal or less than "SPI_FLASH_PageSize" value.
11. * Output         : None
12. * Return         : None
13. *****/
14. void SPI_FLASH_PageWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
15. {
16.     /* Enable the write access to the FLASH */
17.     SPI_FLASH_WriteEnable();
18.
19.     /* Select the FLASH: Chip Select low */
20.     SPI_FLASH_CS_LOW();
21.     /* Send "Write to Memory " instruction */
22.     SPI_FLASH_SendByte(W25X_PageProgram);
23.     /* Send WriteAddr high nibble address byte to write to */
24.     SPI_FLASH_SendByte((WriteAddr & 0xFF0000) >> 16);
25.     /* Send WriteAddr medium nibble address byte to write to */
26.     SPI_FLASH_SendByte((WriteAddr & 0xFF00) >> 8);
27.     /* Send WriteAddr low nibble address byte to write to */
28.     SPI_FLASH_SendByte(WriteAddr & 0xFF);
29.
30.     if(NumByteToWrite > SPI_FLASH_PerWritePageSize)
31.     {
32.         NumByteToWrite = SPI_FLASH_PerWritePageSize;
33.         //printf("\n\r Err: SPI_FLASH_PageWrite too large!");
34.     }
35.
36.     /* while there is data to be written on the FLASH */
37.     while (NumByteToWrite--)
38.     {
39.         /* Send the current byte */
40.         SPI_FLASH_SendByte(*pBuffer);
41.         /* Point on the next byte to be written */
42.         pBuffer++;
43.     }
44.
45.     /* Deselect the FLASH: Chip Select high */
46.     SPI_FLASH_CS_HIGH();
47.
48.     /* Wait the end of Flash writing */
49.     SPI_FLASH_WaitForWriteEnd();
50. }

```

其页写入时序如图 15-13，发送完页写入命令 Page Program (02h) 及地址之后，可以连续写入最多 256 个字节的数据。在发送完数据之后，我们调用 SPI_FLASH_WaitFor WriteEnd() 等待 Flash 内部写时序完成再退出函数。

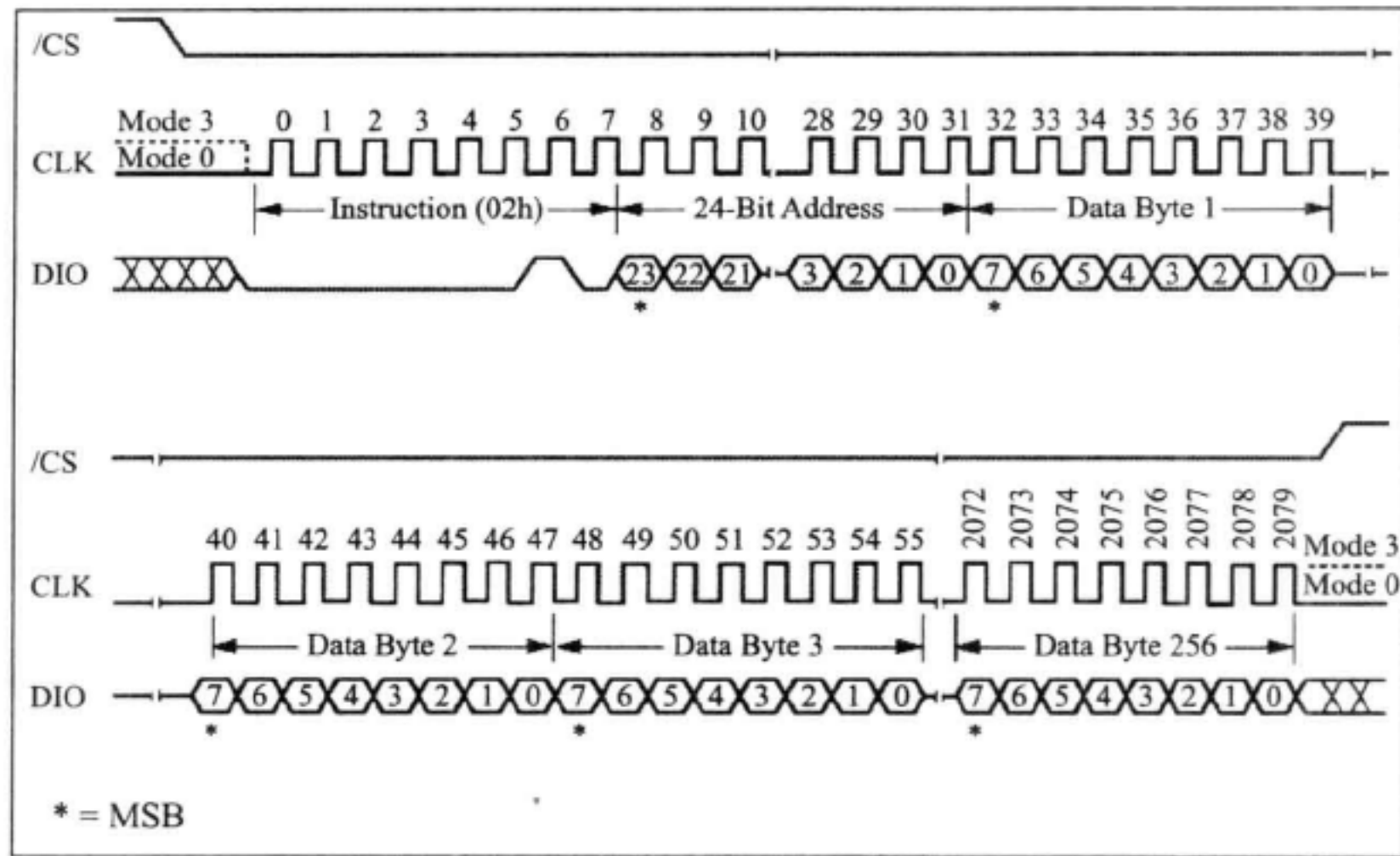


图 15-13 页写入时序

15.3.9 从 Flash 读取数据

对于读数据，发出一个命令后可以无限制地一直把整个 Flash 的数据都读取完。若认为读取的数据量足够了，可以拉高片选信号以表示读取数据结束，我们使用的读数据函数 `SPI_FLASH_BufferRead()` 见代码清单 15-12。

代码清单 15-12 `SPI_FLASH_BufferRead()` 函数

```

1.  /*******
2.  * Function Name   : SPI_FLASH_BufferRead
3.  * Description    : Reads a block of data from the FLASH.
4.  * Input          : - pBuffer : pointer to the buffer that receives the data read
5.  *                  : from the FLASH.
6.  *                  : - ReadAddr : FLASH' s internal address to read from.
7.  *                  : - NumByteToRead : number of bytes to read from the FLASH.
8.  * Output         : None
9.  * Return         : None
10. *****/
11. void SPI_FLASH_BufferRead(u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
12. {
13.     /* Select the FLASH: Chip Select low */
14.     SPI_FLASH_CS_LOW();
15.
16.     /* Send "Read from Memory " instruction */
17.     SPI_FLASH_SendByte(W25X_ReadData);
18.
19.     /* Send ReadAddr high nibble address byte to read from */
20.     SPI_FLASH_SendByte((ReadAddr & 0xFF0000) >> 16);
21.     /* Send ReadAddr medium nibble address byte to read from */
22.     SPI_FLASH_SendByte((ReadAddr & 0xFF00) >> 8);
23.     /* Send ReadAddr low nibble address byte to read from */
24.     SPI_FLASH_SendByte(ReadAddr & 0xFF);
25.
26.     while (NumByteToRead--) /* while there is data to be read */

```



```

27. {
28.     /* Read a byte from the FLASH */
29.     *pBuffer = SPI_FLASH_SendByte(Dummy_Byte);
30.     /* Point to the next location where the byte read will be saved */
31.     pBuffer++;
32. }
33.
34. /* Deselect the FLASH: Chip Select high */
35. SPI_FLASH_CS_HIGH();
36.}

```

读数据命令时序见图 15-14。

SPI_FLASH_BufferRead() 函数首先发送了读数据命令 Read Data (03h)，紧接着发送 24 位的读数据起始地址，STM32 再通过 DO 线接收数据，并把它们使用指针的方式记录起来。

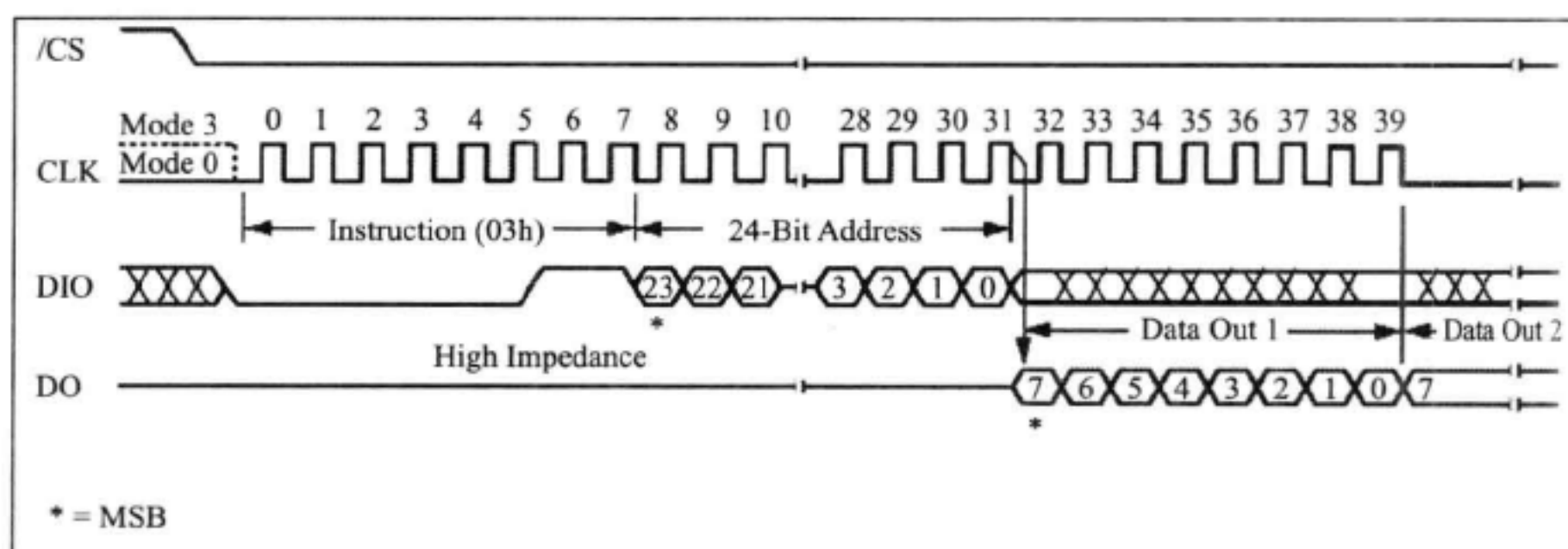


图 15-14 读数据命令时序

15.3.10 小结

通过以上的函数，我们就可以实现对 Flash 的擦除、写入和读取操作了，还有一小部分函数没有分析到，大家根据需求在使用的时候再分析一下源代码吧。

最后，总结一下在 STM32 如何建立与 SPI-FLASH 的通信。

- 1) 配置 I/O 端口，使能 GPIO。
- 2) 根据将要进行通信器件的 SPI 模式，配置 STM32 的 SPI，使能 SPI 时钟。
- 3) 配置好 SPI 后，根据各种 Flash 定义的命令控制对它的读写。

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

15.3.11 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上串口线（两头都是母的交叉线），打开超级终端，配置超级终端为“115200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如图 15-15 所示信息。



图 15-15 Flash 实验现象



第 16 章

CAN 控制器

控制器局域网（Controller Area Network, CAN）是由德国研发和生产汽车电子产品著称的 BOSCH 公司开发的，并最终成为国际标准（ISO11519），是国际上应用最广泛的现场总线之一。

在北美和西欧，CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议，专为大型货车和重工机械车辆设计的 J1939 协议。近年来，其所具有的高可靠性和良好的错误检测能力受到重视，被广泛应用于汽车计算机控制系统和环境恶劣、电磁辐射强和振动大的工业环境。

16.1 CAN 协议简介

16.1.1 物理层

与 I²C、SPI 等具有时钟信号的通信方式不同，CAN 通信并不是以时钟信号来进行同步的。它只具有 CAN_High 和 CAN_Low 两条信号线，共同构成一组差分信号线，所以 CAN 是以差分信号的形式进行通信的。见图 16-1。

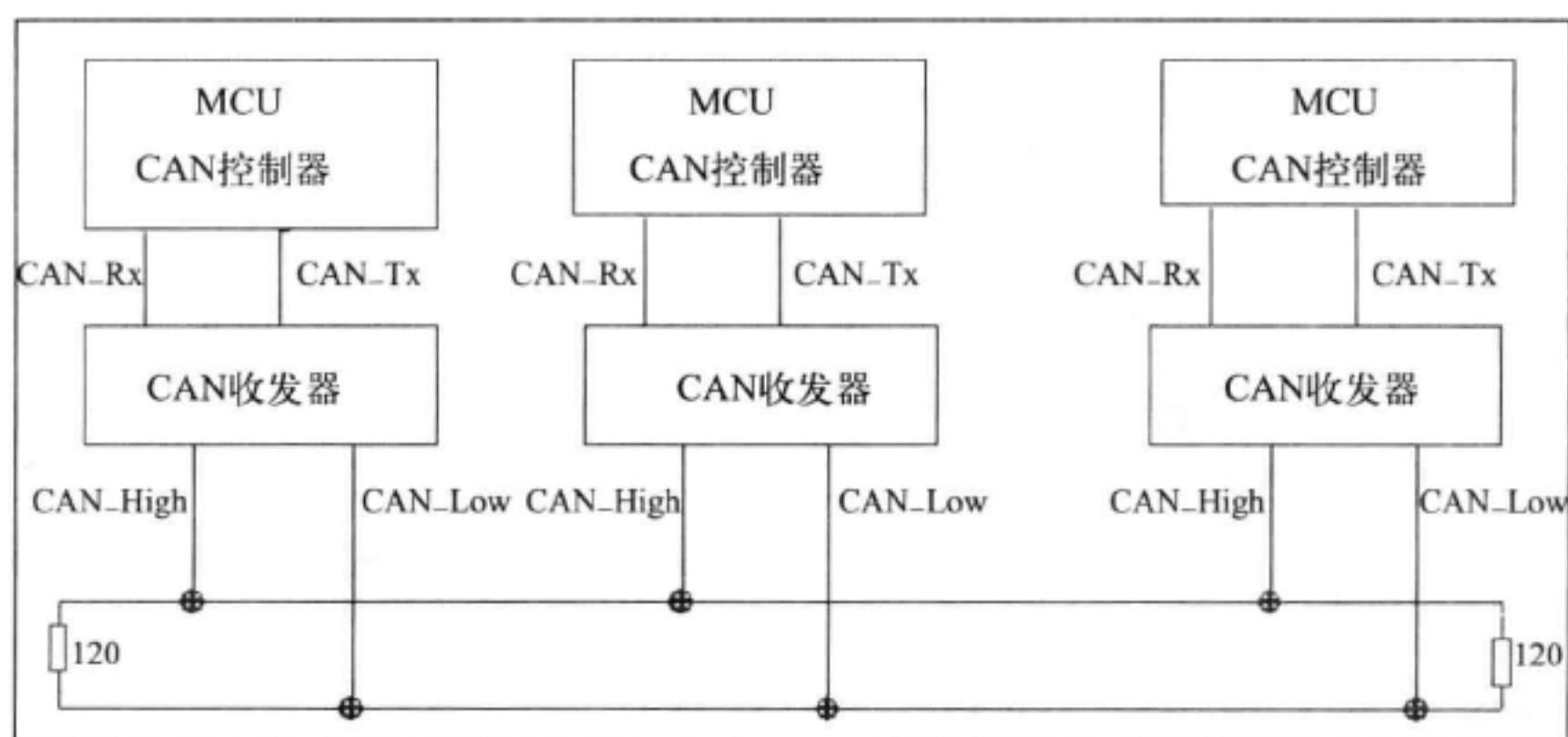


图 16-1 CAN 闭环总线通信网络

图中的 CAN 通信网络是遵循 ISO11898 标准的高速短距离闭环网络，它的总线最大长度为 40 m，通信速度最高为 1 Mbit/s。还有一种是遵循 ISO11519-2 标准的低速远距离开环网络，它最大传输距离为 1 km，最高通信速率为 125 kbit/s。

从 CAN 的通信网络图可以了解到，它的通信节点由一个 CAN 控制器、一个 CAN 收发器组成。如 STM32 的 CAN 接口即为 CAN 控制器，为了构成完整的节点，还要给它外接一个 CAN 收发器。

在发送数据时，CAN 控制器把要发送的二进制编码通过 CAN_Tx 线发送到 CAN 收发器，然后由收发器把这个普通的逻辑电平信号转化成差分信号，通过差分线 CAN_High 和 CAN_Low 线输出到 CAN 总线网络。在接收数据时，这个过程相反。

差分信号是什么意思呢？即信号的逻辑 0 和逻辑 1 由两根差分信号线的电压差来表示。ISO11898 规定 CAN 协议中处于逻辑 1（隐性电平）时，CAN_High 和 CAN_Low 线上的电压均为 2.5 V，即它们的电压差 $V_h - V_l = 0V$ 。而在逻辑 0（显性电平）时，CAN_High 的电平为 3.5 V，CAN_Low 线的电平为 1.5 V，即它们的电压差为 $V_h - V_l = 2V$ 。

如当 CAN 收发器从 CAN_Tx 线接收到来自 CAN 控制器的低电平信号时，它会将该信号转化，使 CAN_High 输出 3.5 V，同时 CAN_Low 输出 1.5 V，即显性电平。如同串口中的 MAX3232 用作电平转换，CAN 收发器的作用则是用作差分信号的转换。

在 CAN 总线中，必须处于隐性电平（逻辑 1）或显性电平（逻辑 0）中的一个状态。假如有两个 CAN 通信节点，在同一时间，一个输出隐性电平，另一个输出显性电平，总线的“线与”特性将使它处于显性电平状态，即可以认为显性具有优先的意味。

16.1.2 CAN 的报文种类及结构

在 SPI 通信中，片选、时钟信号、数据输入及数据输出这四个信号都有单独的信号线。而 CAN 使用的是两条差分信号线，只能表达一个信号。简洁的物理层决定了 CAN 必然要配上一套更复杂的协议。如何用一个信号通道实现同样甚至更强大的功能呢？答案是对数据或操作命令进行打包。

1. 报文的种类

在原始数据段的前面加上传输起始标签、片选（识别）标签、控制标签，在数据的尾段加上 CRC 校验标签、应答标签和传输结束标签。把这些内容按特定的格式打包好，就可以用一个通道表达各种信号了，各种各样的标签就如同 SPI 中各种通道上的信号，起到了协同传输的作用。当整个数据包被传输到其他设备时，只要这些设备按格式去解读，就能还原出原始数据。类似这样的数据包就被称为 CAN 的数据帧。为了更有效地控制通信，CAN 一共规定了 5 种类型的帧，帧也称为报文，它们的类型及用途说明见表 16-1。

表 16-1 帧的种类及其用途

帧	帧 用 途
数据帧	用于发送单元向接收单元传送数据的帧
遥控帧	用于接收单元向具有相同 ID 的发送单元请求数据的帧
错误帧	用于当检测出错误时向其他单元通知错误的帧
过载帧	用于接收单元通知其尚未做好接收准备的帧
帧间隔	用于将数据帧及遥控帧与前面的帧分离开来的帧

2. 数据帧的结构

数据帧是在 CAN 通信中最主要、最复杂的报文，我们来了解一下它的结构，见图 16-2。

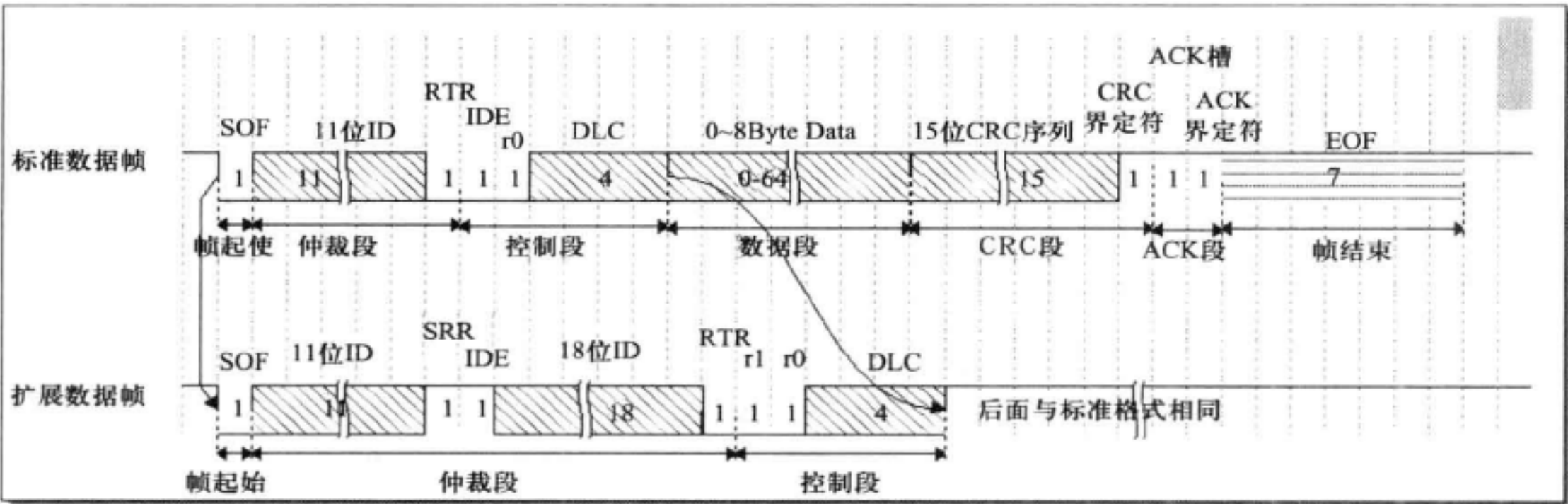


图 16-2 数据帧的结构

数据帧以一个显性位（逻辑 0）开始，以 7 个连续的隐性位（逻辑 1）结束。在它们之间，分为仲裁段、控制段、数据段、CRC 段和 ACK 段。

(1) 仲裁段

仲裁段的内容主要为本数据帧的 ID 信息。数据帧分为标准格式和扩展格式两种，区别就在于 ID 信息的长度；标准格式的 ID 为 11 位，扩展格式的 ID 为 29 位。

在 CAN 协议中，ID 起着重要的作用，它决定着数据帧发送的优先级，也决定着其他设备是否会接收这个数据帧。CAN 协议不对挂载在它之上的设备分配优先级，对总线的占有权是由信息的重要性决定的，即对于重要的信息，我们会给它打包上一个优先级高的 ID，使它能够及时地发送出去。也正因为它这样的优先级分配原则，使得 CAN 的扩展性大大加强，在总线上增加或减少节点并不影响其他设备。

报文的优先级，是通过对 ID 的仲裁来确定的。前面对物理层的分析我们知道如果总线上同时出现显性电平和隐性电平，总线的状态会被置为显性电平，优先级的仲裁就根据这个特性实现的。

见图 16-3。若两个节点同时竞争 CAN 总线的占有权，在它们发送报文时，若首先出现隐性电平，则会失去对总线的占有权，进入接收状态。在开始阶段，两个设备发送的电平一样，所以它们一直继续发送数据。到了图中箭头所指的时序处，节点 1 发送的为隐性时序，而此时节点 2 发送的为显性时序，因此节点 2 竞争总线成功，这个报文得以继续发送出去。

仲裁段 ID 的优先级也影响着接收设备对报文的反应。因为在 CAN 总线上数据是以广播的形式发送的，所有连接在 CAN 总线的节点都会收到所有其他节点发出的有效数据，因而我们的 CAN 控制器大多具有根据 ID 过滤报文的功能，即只接收某些 ID 的报文。

仲裁段除了报文 ID 外，还有 RTR、IDE、SRR 位。其中 RTR（Remote Transmission Request）位是用于区分数据帧和遥控帧的，在数据帧里这一位为显性（逻辑 0）。IDE（Identifier Extension）位是用于区分标准格式与扩展格式的，在标准格式中为显性，在扩展格式里为隐性。SRR（Substitute Remote Request）位只存在于扩展格式，它用于替代标准格式中的 RTR 位。SRR 位为隐性位，由于

RTR 在数据帧为显性位，所以在两个 ID 相同的标准格式报文与扩展格式报文中，标准格式的优先级较高。

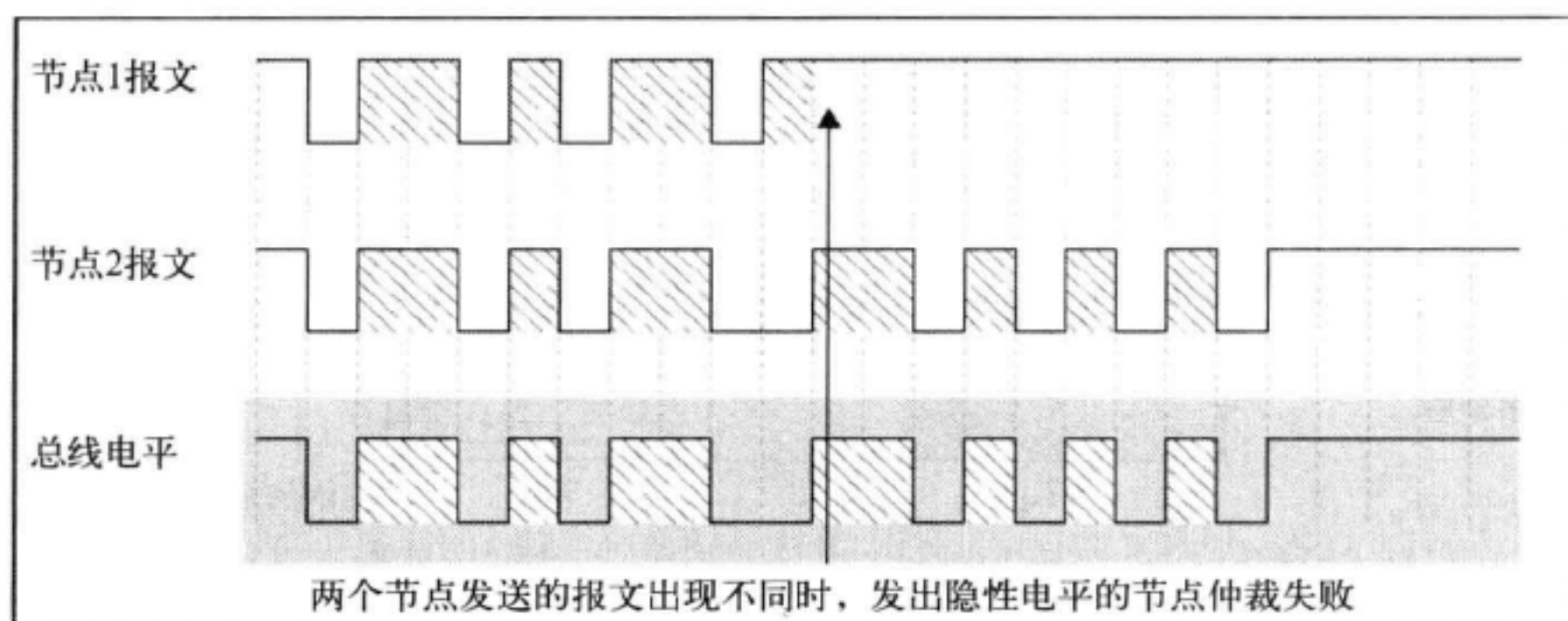


图 16-3 仲裁过程

(2) 控制段

在控制段中的 r1 和 r0 为保留位，默认设置为显性位。最主要的为 DLC 段，DLC 段由 4 位组成，MSB 先行，它的二进制编码用于表示本报文中的数据段含有多少个字节，DLC 段表示的数字为 0 ~ 8。

(3) 数据段

数据段为数据帧的核心内容，它由 0 ~ 8 个字节组成，MSB 先行。

(4) CRC 段

为了保证报文的正确传输，CAN 的报文包含了一段 15 位的 CRC 校验码，一旦接收端计算出的 CRC 码跟接收到的 CRC 码不同，则会向发送端反馈出错信息以及重新发送。CRC 部分的计算和出错处理一般由 CAN 控制器硬件完成或由软件控制最大重发数。

在 CRC 校验码之后，有一个 CRC 界定符，它为隐性位，主要作用是把 CRC 校验码与后面的 ACK 段隔开。

(5) ACK 段

ACK 段包括一个 ACK 槽位和 ACK 界定符位。在 ACK 槽位中，发送端发送的为隐性位，而接收端在这一位中发送显性位以示应答。在 ACK 槽和帧结束之间由 ACK 界定符隔开。

(6) 帧结束段

帧结束段由发送端发送 7 个隐性位表示结束。

CAN 其他种类的帧结构请参阅 CAN 标准协议。

16.1.3 同步

由于 CAN 没有时钟信号线，而且它的报文中并没有包含用于同步的标志，所以要使用位同步的方式来确保通信时序，以及对总线的电平进行正确采样。

1. 位时序分解

为了实现位同步，CAN 协议把每一位的时序分解成如图 16-4 所示的 SS 段、PTS 段、PBS1

段和 PBS2 段，这四段的长度加起来即为一个 CAN 数据位的长度。分解后最小的时间单位是 T_q ，而一个完整的位由 8 ~ 25 个 T_q 组成。图中的一位为 19 T_q 。

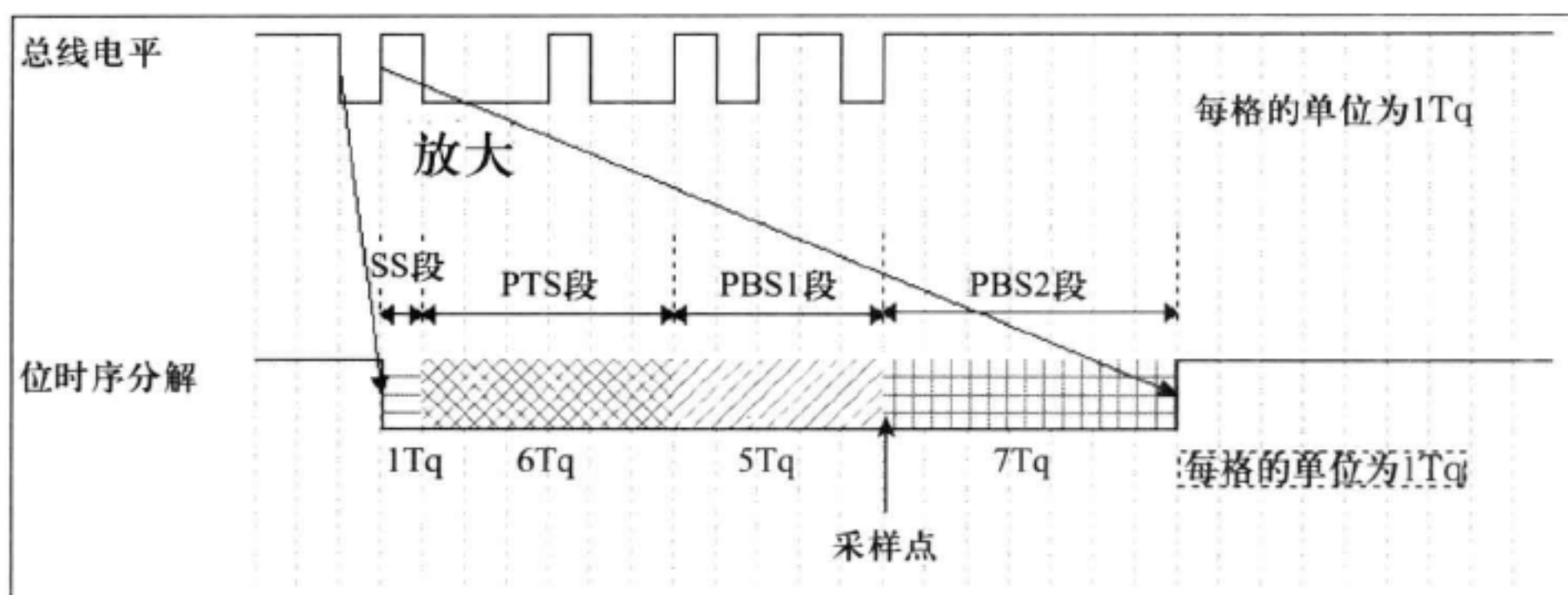


图 16-4 CAN 位时序分解图

每位中的各段作用如下：

- SS 段（SYNC SEG），译为同步段，若总线的跳变沿被包含在 SS 段的范围之内，则表示节点与总线的时序同步。节点与总线同步时，采样点采集到的总线电平即可被确定为该位的电平。如当总线上出现帧起始信号（SOF）时，其他节点上的控制器根据总线上的这个下降沿，对自己的位时序进行调整，把该下降沿包含到 SS 段内，这样根据起始帧来进行同步的方式称为硬同步。其中 SS 段的大小为 1 T_q 。
- PTS 段（PROP SEG），译为传播时间段，这个时间段用于补偿网络的物理延时时间，是总线上输入比较器延时和输出驱动器延时总和的两倍。PTS 段的大小为 1 ~ 8 T_q 。
- PBS1 段（PHASE SEG1），译为相位缓冲段，主要用来补偿边沿阶段的误差，它的时间长度在重新同步的时候可以加长。PBS1 段的初始大小可以为 1 ~ 8 T_q 。
- PBS2 段（PHASE SEG2），这是另一个相位缓冲段，也是用来补偿边沿阶段误差的，它的时间长度在重新同步时可以缩短。PBS2 段的初始大小可以为 2 ~ 8 T_q 。

在重新同步的时候，PBS1 和 PBS2 段的允许加长或缩短的时间长度定义为：重新同步补偿宽度（reSynchronization Jump Width, SJW）。

2. 同步过程分析

CAN 的同步分为硬同步和重新同步。因为硬同步时只是在有帧起始信号时起作用，无法确保后续一连串的位时序都是同步的，所以 CAN 还引入了重新同步的方式。在检测到总线上的时序与节点使用的时序有相位差时（即总线上的跳变沿不在节点时序的 SS 段范围），通过延长 PBS1 段或缩短 PBS2 段来获得同步，这样的方式称为重新同步。见图 16-5。

可以看到在总线出现帧起始信号时，该节点原来的位时序与总线时序不同步，因而这个状态的采样点采集得的数据是不正确的。所以节点以硬同步的方式调整，把自己的位时序中的 SS 段平移至总线出现下降沿的部分，获得同步，这时采样点采集得的为正确数据。

重新同步的方式分为两种情况，如图 16-6 所示。第一种，节点从总线的边沿跳变中，检测到它的时序比总线的时序相对滞后 $2T_q$ ，这时控制器在下一个位时序中的 PBS1 段增加 $2T_q$ 的时间长度，使得节点与总线时序重新同步。第二种，节点从总线的边沿跳变中，检测到它的时序比总线的时序相对超前 $2T_q$ ，这时控制器在前一个位时序中的 PBS2 段减少 $2T_q$ 的时间长度，获得同步。这里设置的 PBS1 和 PBS2 能够增减的最大时间长度为 $SJW=2T_q$ ，若 SJW 设置得太小则重新同步的调整速度慢，若设置得太大则影响传输速率。

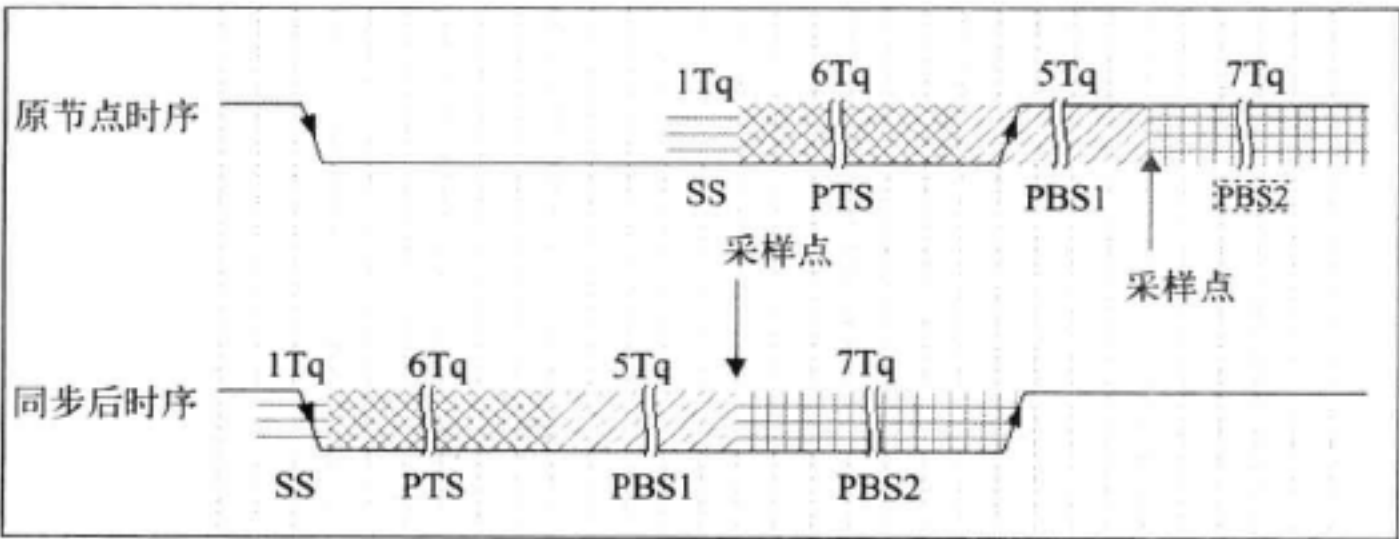


图 16-5 硬同步过程图

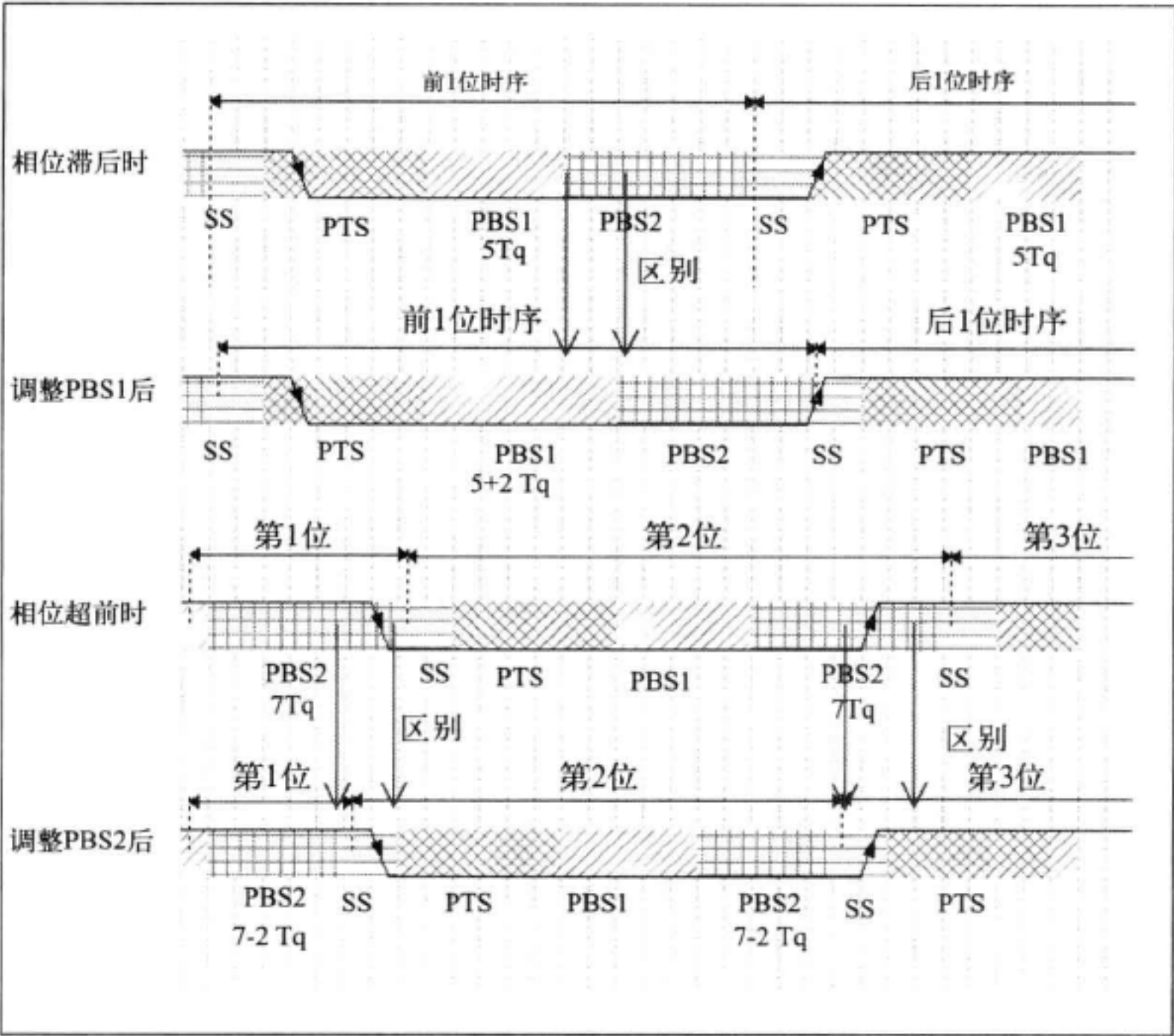


图 16-6 重新同步时的两种情况

16.2 STM32 的 CAN 特性及架构

16.2.1 CAN 特性

STM32 的所有型号芯片中都具有 bxCAN 控制器 (Basic Extended CAN)，它支持 CAN 协议 2.0A 和 2.0B。bxCAN 接口可以自动地接收和发送 CAN 报文，支持标准标识符和扩展标识符。

它具有 3 个发送邮箱，发送报文的优先级可以使用软件，可以记录发送的时间。有两个 3 级深度的接收 FIFO，可以使用过滤功能只接收或不接收某些 ID 号的报文。可以配置成自动重发。不支持使用 DMA 进行数据收发。

16.2.2 CAN 架构

图 16-7 为 STM32 的 CAN 架构，图中左边的 Control / Status / Configuration 的具体细节是关于 CAN 的各种配置。这个图右边有三个专业名词。

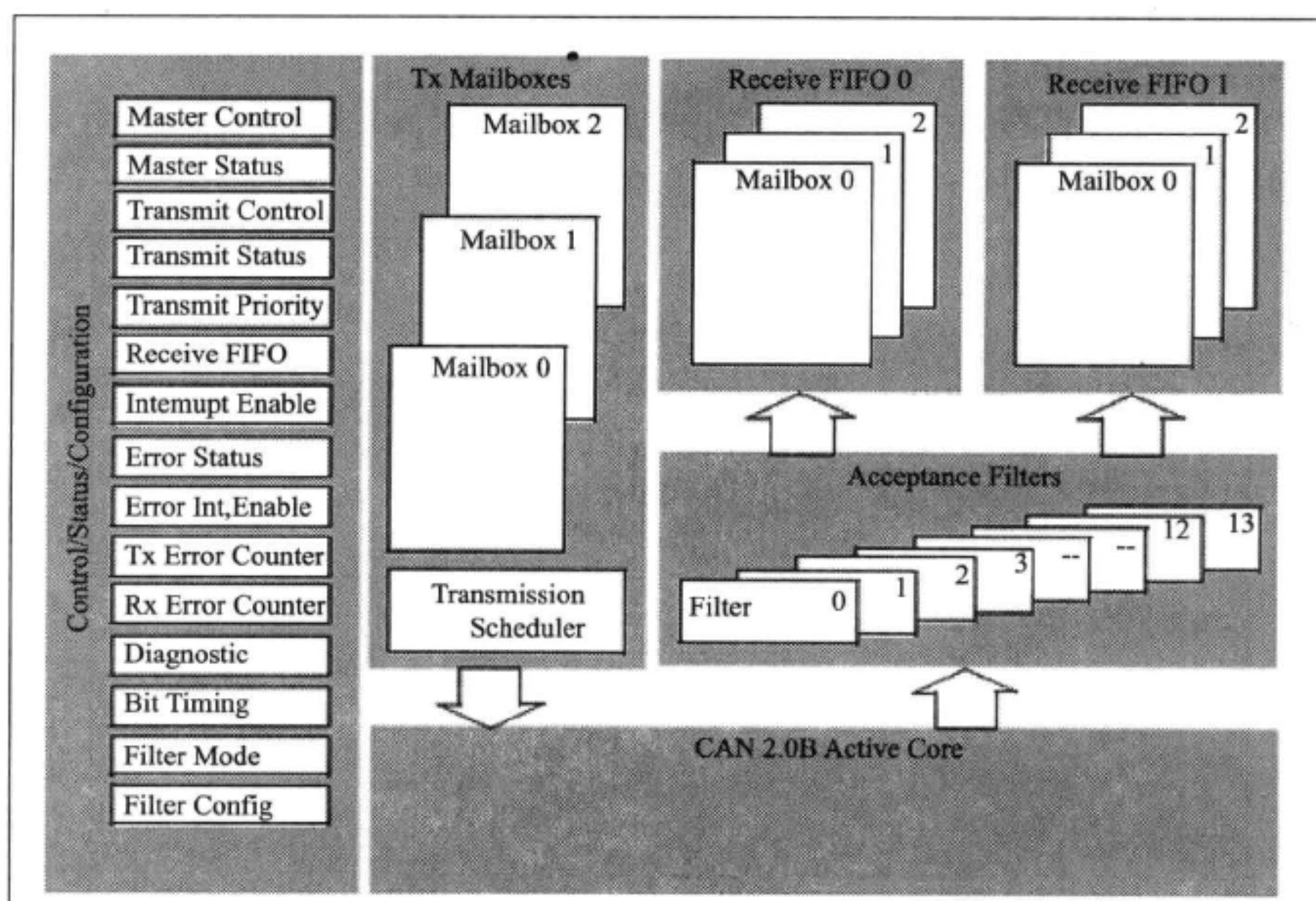


图 16-7 STM32 的 CAN 架构

□ Tx Mailboxes（发送邮箱）

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

□ Acceptance Filters（接收过滤器）

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

□ Receive FIFO（接收 FIFO）

STM32 的 CAN 中共有两个接收 FIFO，每个 FIFO 都可以存放 3 个完整的报文。它们完全由硬件来管理。

CAN 的工作就是围绕这三部分展开的，我们通过实例进行详细讲解。

16.3 双 CAN 通信实验分析

本节讲解的实验是在两个开发板之间使用 CAN 接口进行通信。STM32 的 CAN 接口还可以使用回环模式，进行 CAN 的自检。若只有一个开发板的读者可以使用回环模式来进行测试。

16.3.1 实验描述及工程文件清单

1. 实验描述

双 CAN 测试实验（中断模式），采用两块实验板充当某次 CAN 通信过程的主动发送节点和被动接收节点。主动设备首先发送信号“ABCD”，若被动设备接收到这个信号后，会发送出“DCBA”。若主设备成功收到答复信号“DCBA”，则通过 USART1 向终端打印信息。本次实验中，主动设备在被动设备之后运行。

2. 硬件连接

☐ PB8-CAN-RX

☐ PB9-CAN-TX

3. 库文件

使用 3.5 版本固件库：

☐ startup/start_stm32f10x_hd.c

☐ CMSIS/core_cm3.c

☐ CMSIS/system_stm32f10x.c

☐ FWlib/stm32f10x_gpio.c

☐ FWlib/stm32f10x_rcc.c

☐ FWlib/stm32f10x_usart.c

☐ FWlib/stm32f10x_can.c

☐ FWlib/misc.c

4. 用户文件

☐ USER/main.c

☐ USER/stm32f10x_it.c

☐ USER/led.c

☐ USER/usart.c

☐ USER/can.c

CAN 硬件原理见图 16-8。

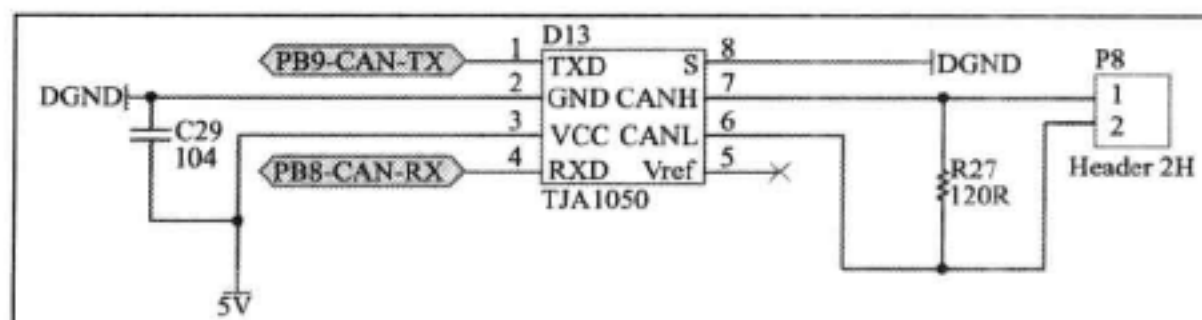


图 16-8 CAN 硬件原理图

配套开发板使用的 STM32 型号为 STM32F103VET6，它自带了一个 CAN 控制器（即我们说的 CAN 接口）。具体 I/O 定义为 PB8-CAN-RX、PB9-CAN-TX。板载的 CAN 外接了一个 TJA1050 型号的 CAN 收发器，它的 RXD 和 TXD 引脚分别连接到 STM32 的 PB8 和 PB9，它的 CANH 和 CANL 引脚是外接到排针上的，并且在 CANH 和 CANL 之间连接了一个 120Ω 的电阻。

在使用双 CAN 通信时，我们可以利用双绞线或两条杜邦线把相应的信号连接起来，由两个板上的电阻和双绞线就构成了 CAN 总线网络。而两块板就成为接入这个 CAN 网络的两个节点了。见图 16-9。

若只是用了 CAN 的回环测试，就不需要挂接外部的 CAN 节点。当我们用 CAN 的回环测试时，硬件会在内部将 TX 和 RX 连接起来，实现内部的收和发，从而达到测试的目的。见图 16-10。

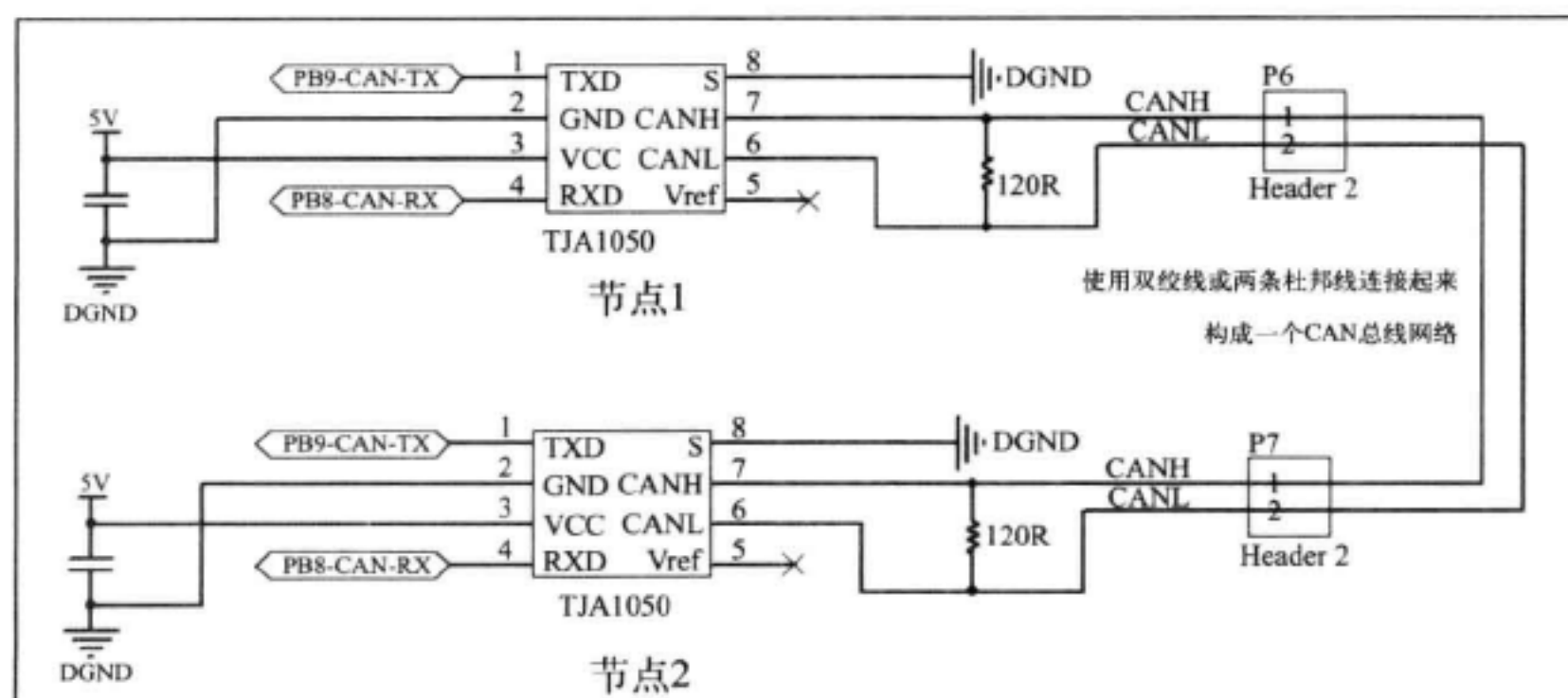


图 16-9 双 CAN 通信连接图

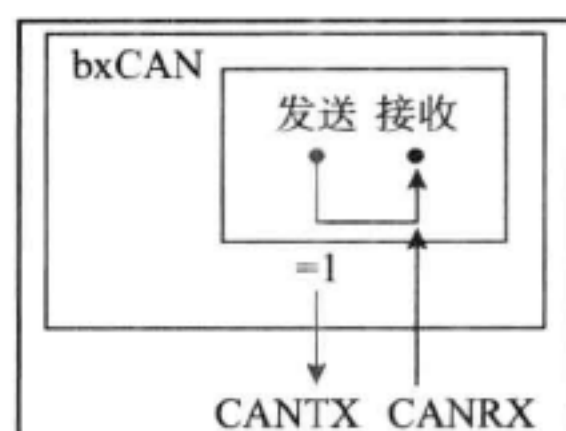


图 16-10 回环测试模式

16.3.2 配置工程环境

本双 CAN 通信中我们创建了两个工程，分别用于节点 1（主机）和节点 2（从机），它们的内容大体相似，讲解时主要使用主设备的代码进行讲解。

在实验环境配置部分，这两个工程是完全一样的。实验中我们用到了 GPIO、RCC、USART 及 CAN 外设，所以我们先要把以下库文件添加到主、从设备的工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_can.c。本实验中使用了中断来读取 CAN 接收邮箱中的数据，所以我们还要添加 misc.c 文件。

接下来添加旧工程中的外设用户文件 usart.c，以便调试和观察实验效果。新建 can.c 及 can.h 文件，并在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 16-1。

代码清单 16-1 CAN 例程中的 stm32f10x_conf.h 文件配置

```
1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****/
9. #include "stm32f10x_can.h"
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
12. #include "stm32f10x_usart.h"
13. #include "misc.h"
```

16.3.3 main 文件

在本实验中，主机和从机的区别主要就体现在 main 函数中，即代码的执行流程不同。不同的 CAN 节点实际具有同等的地位，只是在本实验中为了方便区分而把它们分别称为主机和从机。本实验中的流程图如图 16-11 所示。

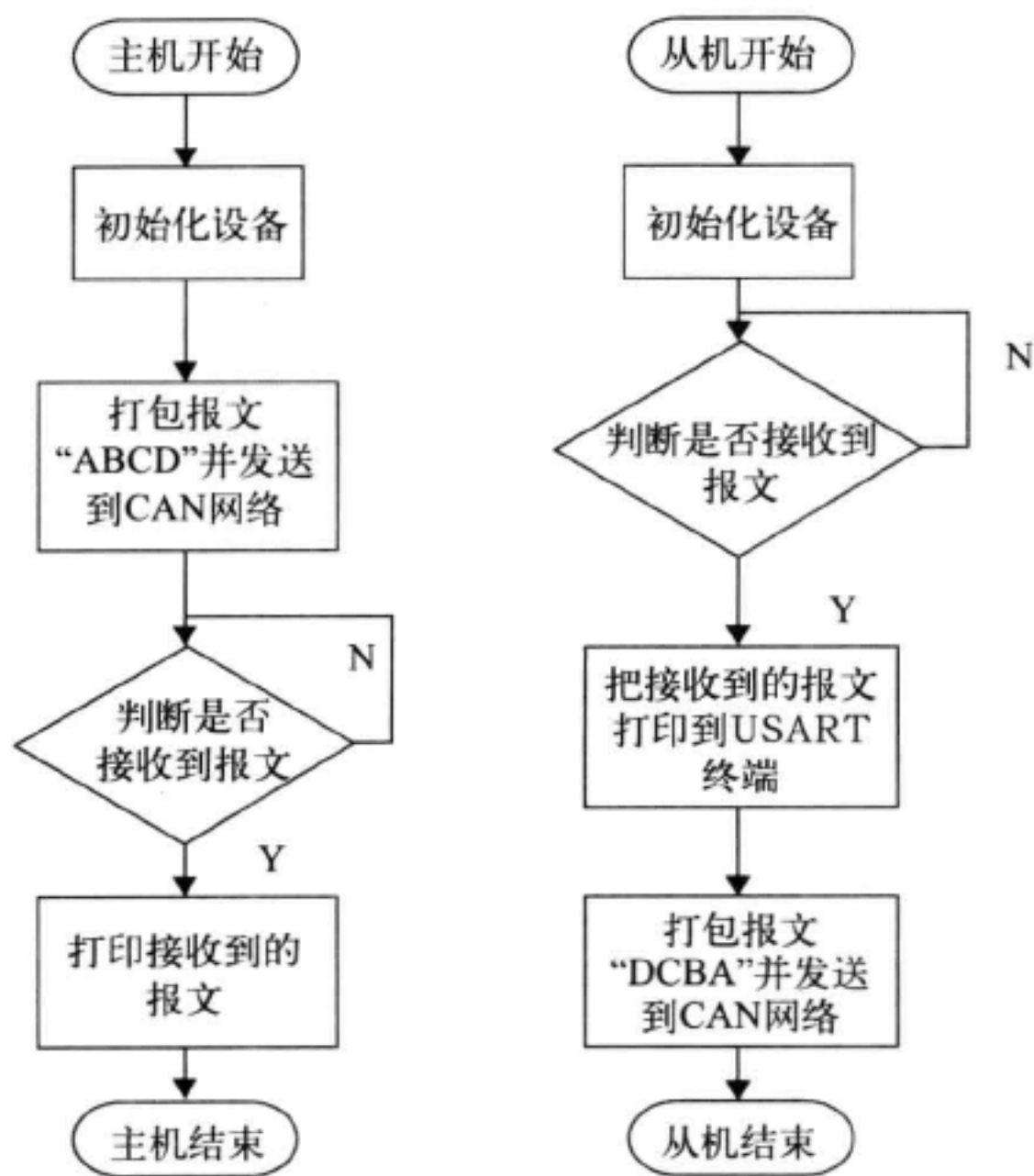


图 16-11 主机及从机的执行流程

1. 主机

首先我们来分析主机的执行流程代码，见代码清单 16-2。

代码清单 16-2 CAN 例程的主机 main 函数

```
1. /*
2.  * 函数名：main
3.  * 描述   ："主机"的主函数
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7. int main(void)
8. {
9.     /* 初始化串口模块 */
10.    USART1_Config();
11.
12.    /* 配置 CAN 接口 */
13.    CAN_Config();
14.
15.    printf( "\r\n***** 这是一个双 CAN 通讯实验 ***** \r\n");
16.    printf( "\r\n 这是 "主机端" 的反馈信息： \r\n");
17.
```

```

18.    /* 设置要通过 CAN 发送的信息 */
19.    CAN_SetMsg();
20.
21.    printf("\r\n 将要发送的报文内容为: \r\n");
22.    printf("\r\n 扩展 ID 号 ExtId: 0x%x", TxMessage.ExtId);
23.    printf("\r\n 数据段的内容: Data[0]=0x%x , Data[1]=0x%x \r\n", TxMessage.Data[0], TxMessage.
    Data[1]);
24.
25.    /* 发送消息 "ABCD" */
26.    CAN_Transmit(CAN1, &TxMessage);
27.
28.
29.    while( flag == 0xff );           //flag =0 , success
30.
31.    printf( "\r\n 成功接收到 " 从机 " 返回的数据 \r\n ");
32.    printf("\r\n 接收到的报文为: \r\n");
33.    printf("\r\n 扩展 ID 号 ExtId: 0x%x", RxMessage.ExtId);
34.    printf("\r\n 数据段的内容: Data[0]= 0x%x , Data[1]=0x%x \r\n", RxMessage.Data[0], RxMessage.
    Data[1]);
35.
36.    while(1);
37.
38.)

```

在主机中，首先调用用户函数 `USART1_Config()` 和 `CAN_Config()` 初始化串口及配置 CAN 接口。接着调用用户函数 `CAN_SetMsg()` 把我们要发送的数据打包成报文并利用 `printf()` 输出这个报文的 ID 及数据信息。生成报文后，调用库函数 `CAN_Transmit()` 把这个报文广播到 CAN 网络上。发送完毕后进入等待状态，轮询标志变量 `flag` 直到它变为 0（这个变量在本 CAN 接口收到数据时在中断服务函数被赋值为 0），本 CAN 接收器接到数据后，把接收到的报文信息打印到终端上。

2. 从机

从机的 main 函数代码见代码清单 16-3。

代码清单 16-3 CAN 例程的从机 main 函数

```

1.  /*
2.  * 函数名: main
3.  * 描述   : "从机" 的主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.
10.     /* USART1 config */
11.     USART1_Config();
12.
13.     /* 配置 CAN 接口 */
14.     CAN_Config();
15.
16.     printf( "\r\n***** 这是一个双 CAN 通讯实验 ***** \r\n");
17.     printf( "\r\n 这是 " 从机端 " 的反馈信息: \r\n");
18.
19.     /* 等待主机端的数据 */

```



```

20.     while( flag == 0xff );
21.
22.     printf( "\r\n 成功接收到 " 主机 " 返回的数据 \r\n " );
23.     printf( "\r\n 接收到的报文为: \r\n" );
24.     printf( "\r\n 扩展 ID 号 ExtId: 0x%x", RxMessage.ExtId );
25.     printf( "\r\n 数据段的内容: Data[0]= 0x%x , Data[1]=0x%x \r\n", RxMessage.Data[0], RxMessage.
    Data[1] );
26.
27.     /* 设置要通过 CAN 发送的信息 */
28.     CAN_SetMsg();
29.
30.     printf( "\r\n 将要发送的报文内容为: \r\n" );
31.     printf( "\r\n 扩展 ID 号 ExtId: 0x%x", TxMessage.ExtId );
32.     printf( "\r\n 数据段的内容: Data[0]=0x%x , Data[1]=0x%x \r\n", TxMessage.Data[0], TxMessage.
    Data[1] );
33.
34.     /* 发送消息 "CDAB" */
35.     CAN_Transmit( CAN1, &TxMessage );
36.
37.     while(1);
38.
39. }

```

从机中配置好了串口和 CAN 接口后, 就开始等待标志变量 flag 被更新 (接收到报文), 若接收到报文则把报文的信息打印到终端上, 接下来成为发送节点, 向 CAN 网络广播 “DCBA” 数据。主机若接收到该报文则会在主机端的 USART 打印报文信息。

16.3.4 配置 CAN 接口

了解代码的执行流程后, 我们仔细分析在 main 函数中调用的函数。被调用的 CAN_Config() 具有初始化 CAN 控制器的 GPIO, 配置 CAN 中断、CAN 模式、CAN 接收过滤器的功能, 这几个功能又被封装成一个个模块。CAN_Config() 函数及它所调用的函数模块, 在本实验的主、从机工程中都是完全一样的。它的实现见代码清单 16-4。

代码清单 16-4 CAN_Config() 函数

```

1.  /*
2.  * 函数名: CAN_Config
3.  * 描述   : 完整配置 CAN 的功能
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void CAN_Config(void)
9. {
10.     CAN_GPIO_Config();
11.     CAN_NVIC_Config();
12.     CAN_Mode_Config();
13.     CAN_Filter_Config();
14. }

```

1. GPIO 初始化及其重映射功能

CAN_Config() 函数调用了 CAN_GPIO_Config() 函数开启了 GPIO 时钟、CAN 外设时钟, 并把 CAN 复用的引脚配置成相应的模式。见代码清单 16-5。

代码清单 16-5 CAN_GPIO_Config() 函数

```
1. /*
2.  * 函数名: CAN_GPIO_Config
3.  * 描述 : CAN 的 GPIO 配置, PB8 上拉输入, PB9 推挽输出
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 内部调用
7.  */
8. static void CAN_GPIO_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* 外设时钟设置 */
13.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOB, ENABLE);
14.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
15.
16.     /* IO 设置 */
17.     GPIO_PinRemapConfig(GPIO_Remap1_CAN1, ENABLE);
18.
19.     /* Configure CAN pin: RX PB8 */
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
21.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; // 上拉输入
22.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
23.     GPIO_Init(GPIOB, &GPIO_InitStructure);
24.
25.     /* Configure CAN pin: TX PB9 */
26.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
27.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; // 复用推挽输出
28.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
29.     GPIO_Init(GPIOB, &GPIO_InitStructure);
30.
31. }
```

关于 GPIO 的复用功能引脚号及其复用功能配置，见表 16-2 和表 16-3。

表 16-2 GPIO 引脚复用功能

脚 位						引脚名称	类型	I/O 电平	主功能	可选的复用功能	
BGA144	BGA100	WLCSP64	LQFP64	LQFP100	LQFP144					默认复用功能	重映射功能
C5	B4	D5	61	95	139	PB8	I/O	FT	PB8	TIM4_CH3/SDIO_D4	I2C1_SCL/CAN_RX
B5	A4	B6	62	96	140	PB9	I/O	FT	PB9	TIM4_CH4/SDIO_D5	I2C1_SDA/CAN_TX

表 16-3 CAN 的 GPIO 复用配置

BxCAN引脚	GPIO配置
CAN_TX	推挽复用输出
CAN_RX	浮空输入或带上拉输入


```
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void CAN_NVIC_Config(void)
9. {
10.     NVIC_InitTypeDef NVIC_InitStructure;
11.     /* Configure one bit for preemption priority */
12.     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
13.     /* 中断设置 */ /*CAN1 RX0 中断 */
14.     NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
15.
16.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;           // 抢占优先级 0
17.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; // 子优先级为 0
18.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
19.     NVIC_Init(&NVIC_InitStructure);
20. }
```

中断向量配置在第 7 章已详细解释，与本实验中的主要为 NVIC 的通道配置区别。本实验中的第 12 行的 .NVIC_IRQChannel 被赋值为 USB_LP_CAN1_RX0_IRQn，即 CAN 的 RX0 中断。这个中断号 USB_LP_CAN1_RX0_IRQn 可以从启动文件 startup_stm32f10x_hd.s 中的中断向量列表中查找到。

那么 RX0 中断又是什么呢？见图 16-13。

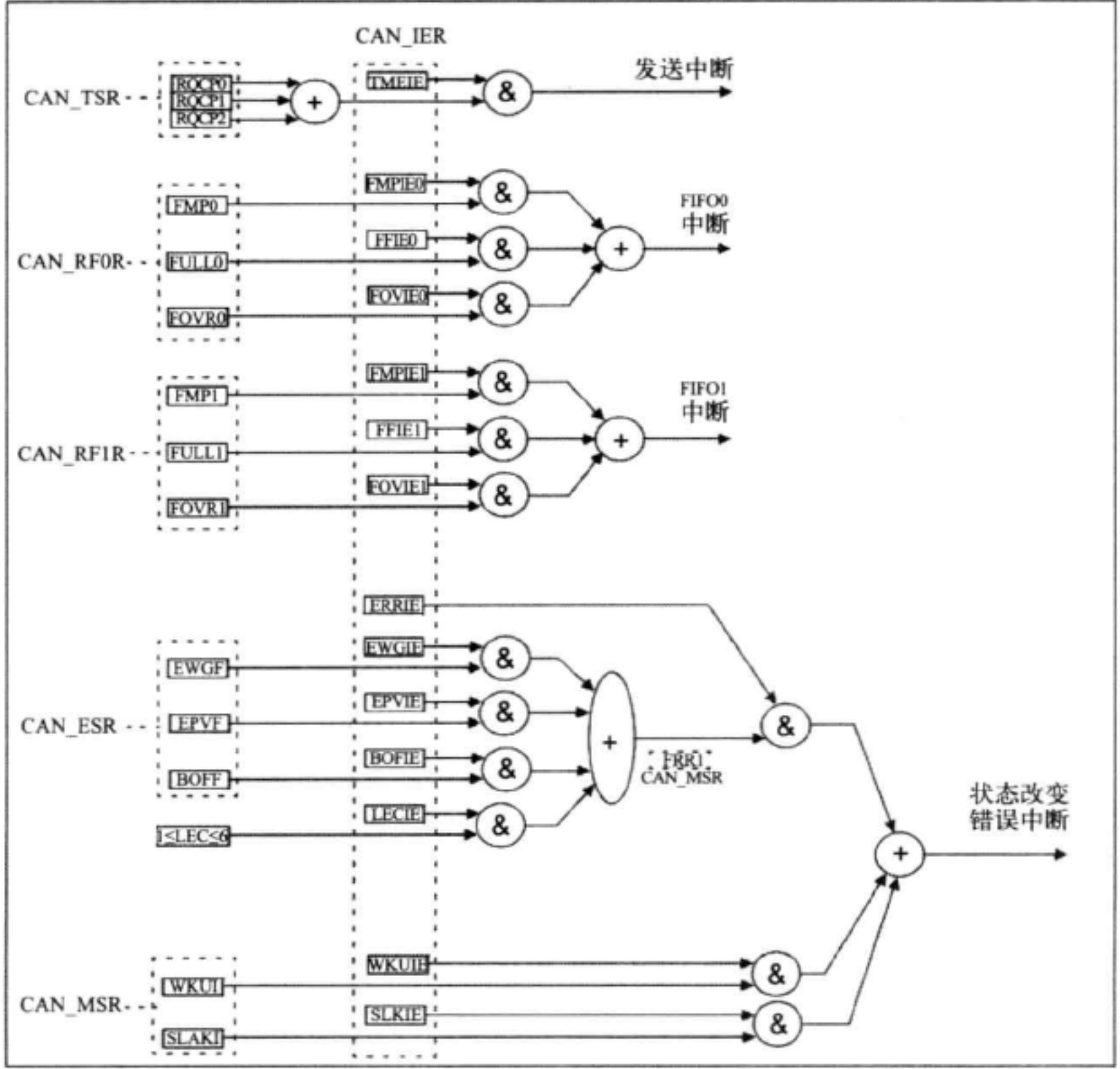


图 16-13 CAN 中断示意图

CAN 的中断由发送中断、接收 FIFO 中断和错误中断构成。发送中断由三个发送邮箱任意一个为空的事件构成。接收 FIFO 中断分为 FIFO0 和 FIFO1 的中断，接收 FIFO 收到新的报文或报文溢出的事件可以引起中断。本实验中使用的 RX0 中断通道即为 FIFO0 中断通道，当 FIFO0 收到新报文时，引起中断，我们就在相应的中断服务函数读取这个新报文。

3. CAN 模式配置

CAN 的模式我们是通过调用用户函数 CAN_Mode_Config() 来实现的，见代码清单 16-7。

代码清单 16-7 CAN_Mode_Config() 函数

```

1. /*
2.  * 函数名：CAN_Mode_Config
3.  * 描述   ：CAN 的模式 配置
4.  * 输入   ：无
5.  * 输出   ：无
6.  * 调用   ：内部调用
7.  */
8. static void CAN_Mode_Config(void)
9. {
10.     CAN_InitTypeDef      CAN_InitStructure;
11.     /******CAN 通信参数设置 *****/
12.     /*CAN 寄存器初始化 */
13.     CAN_DeInit(CAN1);
14.     CAN_StructInit(&CAN_InitStructure);
15.     /*CAN 单元初始化 */
16.     CAN_InitStructure.CAN_TTCM=DISABLE;
17. /*MCR-TTCM 关闭时间触发通信模式使能 */
18.     CAN_InitStructure.CAN_ABOM=ENABLE;
19. /*MCR-ABOM 自动离线管理 */
20.     CAN_InitStructure.CAN_AWUM=ENABLE;
21. /*MCR-AWUM 自动唤醒模式 */
22.     CAN_InitStructure.CAN_NART=DISABLE;
23. /*MCR-NART 禁止报文自动重传  DISABLE= 自动重传 */
24.     CAN_InitStructure.CAN_RFLM=DISABLE;
25. /*MCR-RFLM 接收 FIFO 锁定模式  DISABLE- 溢出时新报文会覆盖原有报文 */
26.     CAN_InitStructure.CAN_TXFP=DISABLE;
27. /*MCR-TXFP 发送 FIFO 优先级  DISABLE- 优先级取决于报文标示符 */
28.     CAN_InitStructure.CAN_Mode = CAN_Mode_Normal; /* 正常工作模式 */
29.     CAN_InitStructure.CAN_SJW=CAN_SJW_2tq;
30. /*BTR-SJW 重新同步跳跃宽度 2 个时间单元 */
31.     CAN_InitStructure.CAN_BS1=CAN_BS1_6tq;
32. /*BTR-TS1 时间段 1 占用了 6 个时间单元 */
33.     CAN_InitStructure.CAN_BS2=CAN_BS2_3tq;
34. /*BTR-TS1 时间段 2 占用了 3 个时间单元 */
35.     CAN_InitStructure.CAN_Prescaler =4;
36. /*BTR-BRP 波特率分频器 定义了时间单元的时间长度 36/(1+6+3)/4=0.8Mbps */
37.     CAN_Init(CAN1, &CAN_InitStructure);
38. }

```

STM23 的 CAN 接口具有非常强大的功能，因而它的初始化结构体也有大量的参数可供配置。

CAN 初始化结构体具有以下成员：

1) CAN_TTCM：本成员用于配置 CAN 的时间触发通信模式（time triggered communication mode）。在此模式下，CAN 使用它内部定时器产生时间戳，被保存在 CAN_RDTxR、CAN_TDTxR 寄存器中。内部定时器在每个 CAN 位时间累加，在接收和发送的帧起始位被采样，并生

成时间戳。本实验不使用时间触发模式。

2) CAN_ABOM: 当 CAN 检测到发送错误 (TEC) 或接收错误 (REC) 超过一定值时, 会自动进入离线状态。在离线状态中, CAN 不能接收或发送报文。其中的发送错误或接收错误的计算原则由 CAN 协议规定, 是 CAN 硬件自动检测的, 不需要软件干预。软件可干预的是通过此 CAN_ABOM 参数选择是否使用自动离线管理 (automatic bus-off management), 决定 CAN 硬件在什么条件下可以退出离线状态。

若我们把此成员赋值为 ENABLE, 则使用硬件自动离线管理。一旦硬件检测到 128 次 11 位连续的隐性位, 则自动退出离线状态。若我们把此成员赋值为 DISABLE, 离线状态由软件管理。首先由软件对 CAN_MCR 寄存器的 INRQ 位进行置“1”随后清“0”, 再等到硬件检测到 128 次 11 位连续的隐性位, 才退出离线状态。本实验使用硬件自动离线管理。

3) CAN_AWUM: 本成员选择是否开启自动唤醒功能 (automatic wakeup mode)。若使能了自动唤醒功能, 并且 CAN 处于睡眠模式, 检测到 CAN 总线活动时会自动进入正常模式, 以便收发数据。若禁止此功能, 则只能由软件配置才可以使 CAN 退出睡眠模式。本实验使用自动唤醒模式。

4) CAN_NART: 本成员用于选择是否禁止报文自动重传 (no automatic retransmission)。按照 CAN 的标准, CAN 发送失败时会自动重传至成功为止。向本参数赋值 ENABLE, 即禁止自动重传, 若赋值为 DISABLE, 则允许自动重传功能。本实验允许 CAN 的自动重传。

5) CAN_RFLM: 本成员用于配置接收 FIFO 是否锁定 (receive FIFO locked mode)。若选择 ENABLE, 则当 FIFO 溢出时会丢弃下一个接收的报文。若选择 DISABLE, 当 FIFO 溢出时下一个接收到的报文会覆盖原报文。本实验选择非锁定模式。

6) CAN_TXFP: 本成员用于选择 CAN 报文发送优先级的判定方法。STM32 的 CAN 接口可以对它邮箱内的几个将要发送的报文按照优先级进行处理。对于这个优先级的判定可以设置为按照报文标识符来决定 (DISABLE), 或按照报文的请求顺序来决定 (ENABLE)。本实验发送报文的优先级按照报文标识符来决定。

7) CAN_Mode: 本成员用于选择 CAN 是处于工作模式状态还是测试模式状态。它有四个可赋值参数, 分别是一个正常工作模式 (CAN_Mode_Normal), 以及静默模式 (CAN_Mode_Silent)、回环模式 (CAN_Mode_LoopBack) 和静默回环模式 (CAN_Mode_Silent_LoopBack) 三个测试模式。本实验使用的是正常的两个 CAN 节点间的通信, 所以向本成员赋值为正常工作模式。

8) CAN_SJW、CAN_BS1、CAN_BS2 及 CAN_Prescaler: 这几个成员是用来配置 CAN 通信的位时序的。它们分别代表 CAN 协议中的 SJW 段 (重新同步跳跃宽度)、PBS1 段 (相位缓冲段 1)、PBS2 段 (相位缓冲段 2) 及时钟分频。

在 STM32 的 CAN 接口配置中, SS 段 (同步段) 被固定为 1 Tq, PTS (物理缓冲段) 被省略。所以一个正常的位时间只由 SS 段 (固定为 1 Tq)、PBS1 段 (CAN_BS1 成员) 及 PBS2 段 (CAN_BS2 成员) 组成, PBS1 和 PBS2 的重新同步跳跃宽度由成员 CAN_SJW 决定。而时间单元 Tq 则

由 CAN_Prescaler 成员决定，它决定 CAN 使用的时钟是由 APB1 的多少分频得到。STM32 位时间段见图 16-14。

我们知道 PBS1 与 PBS2 之间是采样点，一般配置在位时间段的 75% ~ 80% 的位置，保证总线上不同节点数据同步。在本实验中我们把采样点设置在 70% 处。即 $SS=1 T_q$ ， $CAN_BS1=6 T_q$ ， $CAN_BS2=3 T_q$ 。为提高同步调整的速度，把 CAN_SJW 配置为 $2 T_q$ 。利用图 16-14 中的公式。时间单位 T_q 根据成员 CAN_Prescaler 的值（4 分频）及 APB1 的时钟频率（36 MHz）计算得出。 $T_q=1/(36\text{ MHz}/4)$ 秒，即实际上每一个 CAN 位的时间为 $10 T_q (1+6+3T_q)$ ，波特率为 $36\text{MHz}/4/10=0.8\text{Mbit/s}$ 。

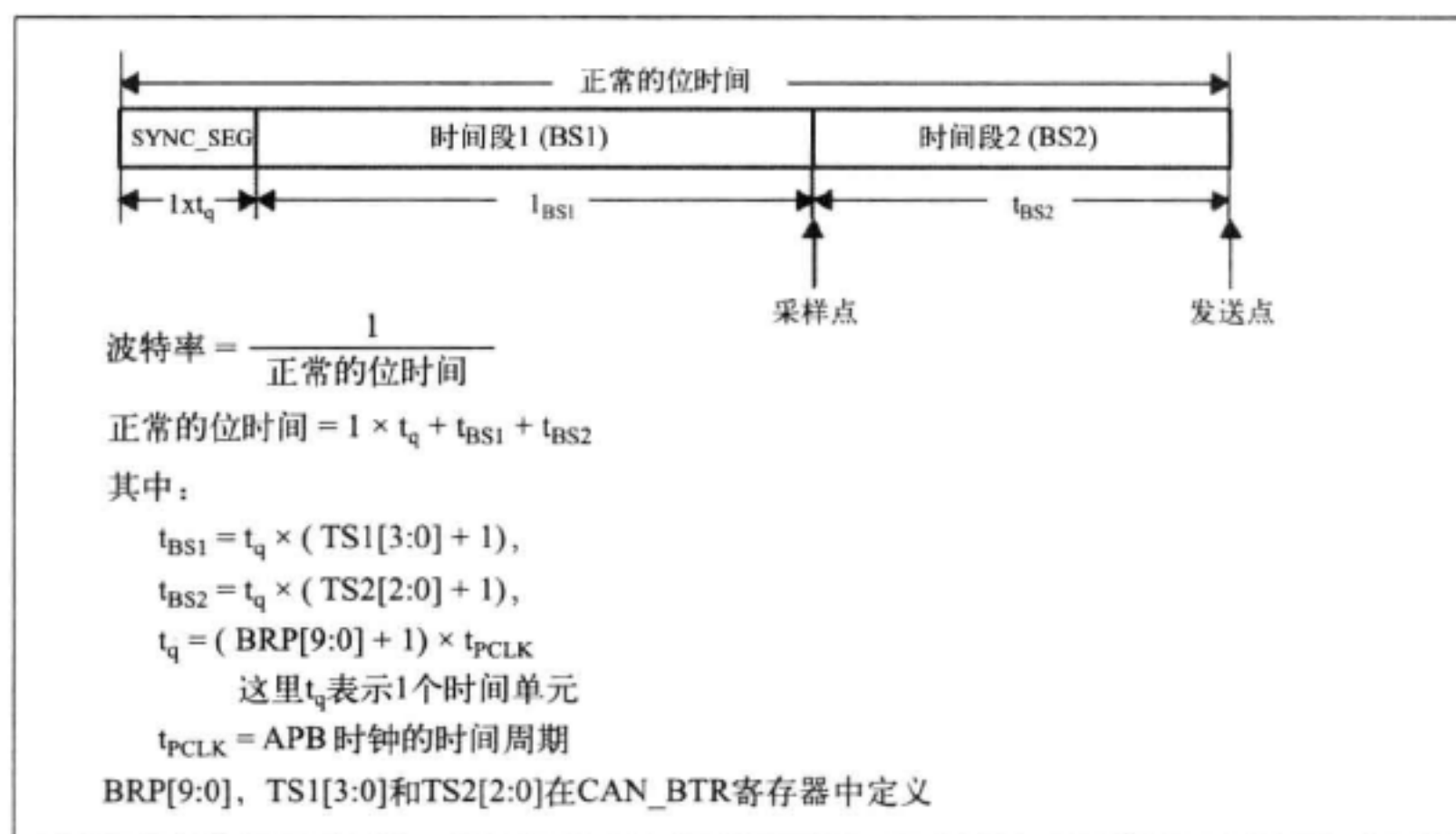


图 16-14 STM32 位时间段

对于这些与 CAN 位时序相关的配置，在 CAN 网络上的各节点应设置成相同或相似以确保 CAN 通信的正常。

配置完这些成员后，我们调用库函数 CAN_Init() 把这些参数写进寄存器。

4. CAN 过滤器配置

在配置完 CAN 的 GPIO、中断和模式后，我们还要对 CAN 的过滤器进行配置，使接收 FIFO 只接收特定的报文。所谓的过滤就是 CAN 接口根据收到报文的 ID，选择是否把该报文保存到接收 FIFO 中。

STM32 的 ID 过滤方式有两种。一种为标识符列表模式，它把要接收报文的 ID 列成一个表，要求报文 ID 与列表中的某一个标识符完全相同才可以接收，可以理解为白名单管理。另一种称为标识符屏蔽模式，它把可接收报文 ID 的某几位作为列表，这几位被称为屏蔽位，可以把它理解成关键字搜索，只要屏蔽位（关键字）相同就符合要求。即这种模式只要求报文 ID 的屏蔽位与列表中标识符相应屏蔽位相同，报文就被保存到接收 FIFO。见图 16-15。

在非互联型号的 STM32 中的 CAN 接口具有 14 个过滤器组，每组过滤器由两个标识符寄存器组成。图 16-15 中左端表示过滤器的配置，FSCx、FBMx 是属于过滤器的配置寄存器（CAN_FS1R、CAN_FM1R）中的寄存器位。FSCx 用来配置过滤器组 x 的寄存器长度为 16 位还是 32 位，本图只解释 32 位

的情况，即 FSCx=1。而 FBMx 则用于选择过滤器组 x 是用作标识符屏蔽模式还是标识符列表模式。

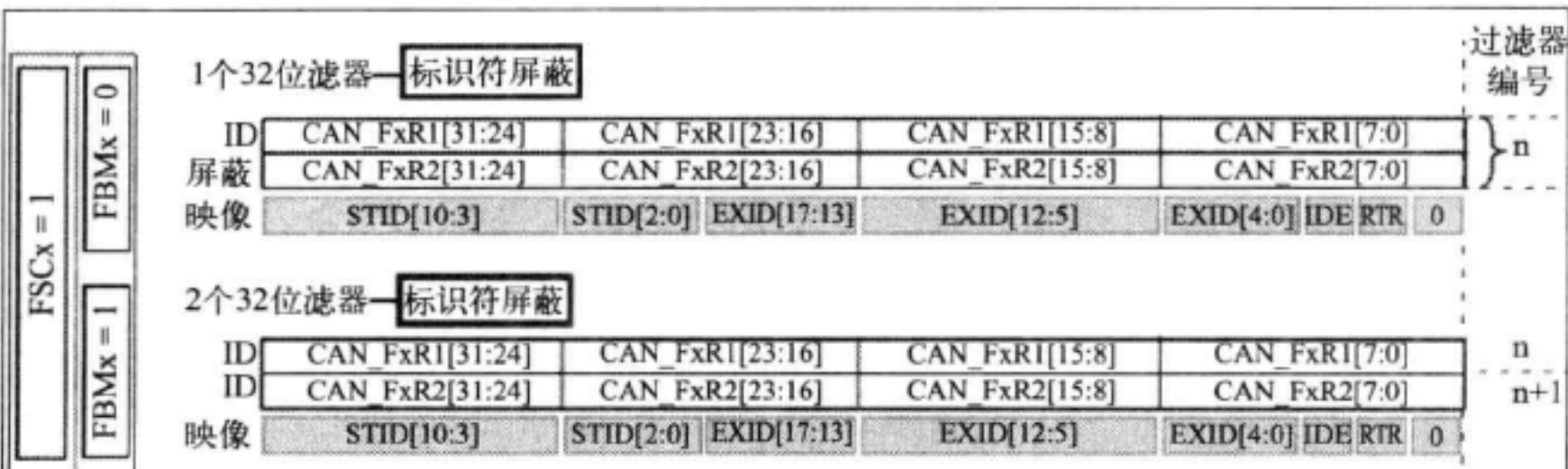


图 16-15 过滤器组

在标识符屏蔽模式下，过滤器组 x 的第 1 个标识符寄存器（CAN_FxR1）用来保存用于与报文 ID 比较的完整标识符。第 2 个标识符寄存器（CAN_FxR2）则用来表示屏蔽位，即表明报文 ID 要与第 1 个标识符寄存器中的哪几位比较。

在标识符列表模式下，过滤器组 x 的两个标识符寄存器都用于保存完整的标识符。报文 ID 若与其中一个标识符完全相同，可被保存到接收 FIFO。

从过滤器的寄存器映像中看到，无论用作哪个模式，标识符寄存器的第 0 位保留，第 1 位为报文的 RTR 位，第 2 位是报文的 IDE 位，报文的扩展 ID 保存在第 3 ~ 20 位，而报文的标准 ID 则保存在第 21 ~ 32 位。使用的时候，我们要根据它的意义写入参数。

下面我们对具体的代码进行分析，CAN 的过滤器设置在用户函数 CAN_Filter_Config() 中实现，见代码清单 16-8。

代码清单 16-8 CAN_Filter_Config() 函数

```
1. /*
2.  * 函数名：CAN_Filter_Config
3.  * 描述   ：CAN 的过滤器 配置
4.  * 输入   ：无
5.  * 输出   ：无
6.  * 调用   ：内部调用
7.  */
8. static void CAN_Filter_Config(void)
9. {
10.     CAN_FilterInitTypeDef  CAN_FilterInitStructure;
11.
12.     /*CAN 过滤器初始化 */
13.     CAN_FilterInitStructure.CAN_FilterNumber=0;          // 过滤器组 0
14.     CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
15. /* 工作在标识符屏蔽位模式 */
16.     CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;
17. /* 过滤器位宽为单个 32 位。 */
18. /* 使能报文标示符过滤器按照标示符的内容进行比对过滤，扩展 ID 不是如下的就抛弃掉，是的话，会存入 FIFO0 */
19.
20.     CAN_FilterInitStructure.CAN_FilterIdHigh= (((u32)0x1314<<3)&0xFF
        FF0000)>>16;          /* 要过滤的 ID 高位 */
```



```

21.    CAN_FilterInitStructure.CAN_FilterIdLow= (((u32)0x1314<<3)|CAN_ID_EXT|CAN_RTR_
      DATA)&0xFFFF; /* 要过滤的 ID 低位 */
22.    CAN_FilterInitStructure.CAN_FilterMaskIdHigh= 0xFFFF;
23./* 过滤器高 16 位每位必须匹配 */
24.    CAN_FilterInitStructure.CAN_FilterMaskIdLow= 0xFFFF;
25./* 过滤器低 16 位每位必须匹配 */
26.    CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO0 ; /* 过滤器被
      关联到 FIFO0 */
27.    CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;
28./* 使能过滤器 */
29.    CAN_FilterInit(&CAN_FilterInitStructure);
30.    /*CAN 通信中断使能 */
31.    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
32.}

```

过滤器的初始化是使用过滤器初始化结构体与库函数 CAN_FilterInit() 实现的。过滤器初始化结构体成员如下：

1) CAN_FilterNumber：本成员用于选择要配置的过滤器组，其参数值可以为 0 ~ 13，分别与过滤器组 0 ~ 13 一一对应。本实验选择过滤器组 0。

2) CAN_FilterMode：本成员用于配置过滤器的工作模式，分别为标识符列表模式 (CAN_FilterMode_IdList) 和标识符屏蔽模式 (CAN_FilterMode_IdMask)。本实验使用标识符屏蔽模式。

3) CAN_FilterScale：本成员用于配置过滤器的长度，可以分别设置为 16 位 (CAN_FilterScale_16bit) 和 32 位 (CAN_FilterScale_32bit)。本实验设置与过滤器解说图 16-15 的一样，为 32 位模式。

4) CAN_FilterIdHigh 和 CAN_FilterIdLow：这两个成员分别为过滤器组中的第 1 个标识符寄存器的高 16 位和低 16 位。本实验中配置要接收的报文使用了扩展 ID，其标准 ID 部分 (11 位) 为 0x000，其扩展部分 (18 位) ID 为 0x01314。

IDE 位为隐性位 (宏 CAN_ID_EXT：(uint32_t) 0x00000004) 并且报文为数据帧，即 RTR 位为显性位 (宏 CAN_RTR_DATA：(uint32_t) 0x00000000)。

我们把这些信息整理好，写入到相应的寄存器位中，使过滤器比较的报文 ID 为扩展格式的数据帧。赋值时标识符 0x0000 1314 左移三位是因为寄存器的低三位意义为保留位、RTR 位和 IDE 位。

```
1. CAN_FilterIdHigh= (((u32)0x1314<<3)&0xFFFF0000)>>16
```

CAN_FilterIdHigh 成员中 (0x00001314) 左移后再与 0xFFFF0000 作 “&” 运算是为了取这个标识符的高 16 位，最后再右移 16 位赋值给 CAN_FilterIdHigh 成员。调用库函数时，函数会把本成员的值写入到标识符寄存器的高 16 位。

```
1. CAN_FilterIdLow= (((u32)0x1314<<3)|CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF
```

CAN_FilterIdLow 成员的赋值运算类似，(0x00001314) 左移后与宏 CAN_ID_EXT 和宏 CAN_RTR_DATA 作 “|” 运算，即把报文的 IDE 位、RTR 位也加入到过滤器进行过滤比较。

5) CAN_FilterMaskIdHigh 和 CAN_FilterMaskIdLow：这两个成员为过滤器组中的第 2 个标识符寄存器的高 16 位与低 16 位。若在标识符列表模式中，这个寄存器保存的内容也是用于过滤的标识符。在标识符屏蔽模式下，这个寄存器保存的内容是屏蔽位。

本实验中过滤器模式是屏蔽模式。本寄存器中，值为 1 的寄存器位就是屏蔽位，值为 0 的寄存器位不参与比较。如我们向这两个成员的参数赋值都为 0xFFFF，即本寄存器位的值全部为 1，表示我们在上面配置的 CAN_FilterIdHigh 和 CAN_FilterIdLow 每一位都要进行比较。其实在正常的屏蔽位模式使用中，我们一般不会要求所有标识符都参与比较的（因为这样的比较结果与标识符列表模式相同），而是选择某些位进行比较，从而过滤出一组报文。

6) CAN_FilterFIFOAssignment：本成员用于设置过滤器与接收 FIFO 的关联，即过滤成功后报文的存储位置，可配置为 FIFO0（CAN_Filter_FIFO0）和 FIFO1（CAN_Filter_FIFO1）两个接收位置。本实验设置存储位置为 FIFO0。

7) CAN_FilterActivation：本成员用于使能或关闭过滤器，默认为关闭。因而使用过滤器时，我们要向它赋值 ENABLE。

对过滤器成员参数赋值完毕后我们调用库函数 CAN_FilterInit() 将它们写入寄存器。在本实验中由于我们使用中断来读取 FIFO 数据，所以在函数 CAN_Filter_Config() 的最后我们调用库函数 CAN_ITConfig() 来开启 FIFO0 消息挂号中断使能（CAN_IT_FMP0），在 CAN 接口的 FIFO0 收到报文时，我们在中断服务函数中从 FIFO0 读数据到内存。库函数 CAN_ITConfig() 说明见图 16-16。

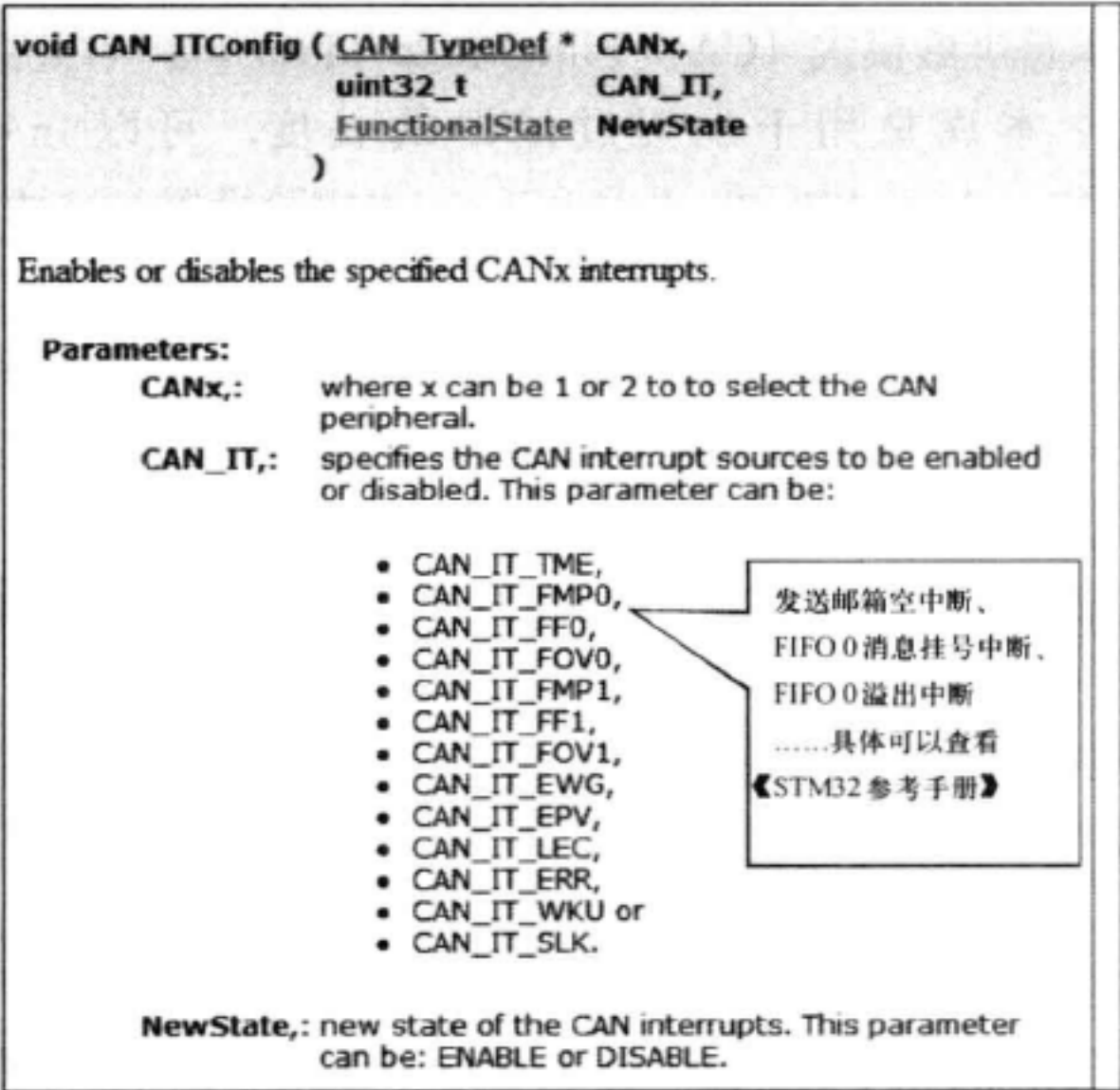


图 16-16 库函数 CAN_ITConfig() 说明

16.3.5 打包报文

配置好 CAN 接口后，我们就可以复用它来发送数据了。利用 CAN 发送数据，要先把数据打包成完整的 CAN 报文格式。我们使用用户函数 CAN_SetMsg() 来实现，见代码清单 16-9。

代码清单 16-9 CAN_SetMsg() 函数

```

1.  /*
2.  * 函数名: CAN_SetMsg
3.  * 描述  : "主机" 的 CAN 通信报文内容设置
4.  * 输入  : 无
5.  * 输出  : 无
6.  * 调用  : 外部调用
7.  */
8.  void CAN_SetMsg(void)
9.  {
10. // TxMessage.StdId=0x00;
11. TxMessage.ExtId=0x1314;           // 使用的扩展 ID
12. TxMessage.IDE=CAN_ID_EXT;        // 扩展模式
13. TxMessage.RTR=CAN_RTR_DATA;      // 发送的是数据
14. TxMessage.DLC=2;                 // 数据长度为 2 字节
15. TxMessage.Data[0]=0xAB;
16. TxMessage.Data[1]=0xCD;
17. }

```

若清楚 CAN 协议的数据帧报文格式, 阅读本代码就没有难度了。本函数使用的结构体变量 TxMessage 在 main 文件以全局变量的形式定义, 见代码清单 16-10。

代码清单 16-10 结构体变量 TxMessage 和 RxMessage

```

1. CanTxMsg TxMessage;           // 发送缓冲区
2. CanRxMsg RxMessage;          // 接收缓冲区

```

TxMessage 的类型为 CanTxMsg, 而接收报文时, 我们使用 CanRxMsg 类型。它们都是由库文件定义的结构体类型。

我们先以 CAN_SetMsg() 函数分析怎样利用类型 CanTxMsg 打包发送报文, 它有如下成员:

- 1) StdId: 本成员保存的是报文的标准 ID。在本实验中我们使用扩展 ID, 所以不需要对它赋值。
- 2) ExtId: 报文的扩展 ID。在实验中我们设置为 0x01314。
- 3) IDE: 本成员用于设置 CAN 报文的 IDE 位, 即使用标准 ID (CAN_Id_Standard) 还是使用扩展 ID (CAN_ID_EXT)。本实验使用扩展 ID。
- 4) RTR: 本成员用于设置 CAN 报文的 RTR 位, 即确定是数据帧 (CAN_RTR_DATA) 还是遥控帧 (CAN_RTR_REMOTE)。本实验发送的为数据帧。
- 5) DLC: 用于设置 CAN 报文控制段中的 DLC 段, DLC 的大小为数据段的字节长度, 可取值范围为 0 ~ 8。本实验要发送的数据为 2 个字节, 所以向 DLC 赋值为 2。
- 6) Data[0] ~ Data[1]: 这些 Data 数组即为 CAN 报文中数据段的内容, Data 的数量要与 DLC 段的一致。本实验中主机的数据段设置为 Data[0]=0xAB, Data[1]=0xCD。在从机的数据段则设置为 Data[0]=0xDC, Data[1]=0xBA。

从机端的报文打包函数如下, 与主机仅有数据段的内容区别, 见代码清单 16-11。

代码清单 16-11 从机端报文打包函数

```

1.  /*
2.  * 函数名: CAN_SetMsg

```

```

3.  * 描述   : "从机"CAN 通信报文内容设置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void CAN_SetMsg(void)
9. {
10. //TxMessage.StdId=0x00;
11. TxMessage.ExtId=0x1314;           // 使用的扩展 ID
12. TxMessage.IDE=CAN_ID_EXT;       // 扩展模式
13. TxMessage.RTR=CAN_RTR_DATA;     // 发送的是数据
14. TxMessage.DLC=2;                // 数据长度为 2 字节
15. TxMessage.Data[0]=0xDC;
16. TxMessage.Data[1]=0xBA;
17.}

```

16.3.6 发送报文

回到主机的 main 函数中, 在初始化完毕后, 调用了函数 CAN_SetMsg() 来打包报文, 然后调用了库函数 CAN_Transmit(), 由这个函数把我们打包好的报文发送到 CAN 网络。见代码清单 16-12。

代码清单 16-12 发送报文到 CAN 网络

```

1. /* 发送消息 "ABCD" */
2. CAN_Transmit(CAN1, &TxMessage);

```

其库函数说明见图 16-17。

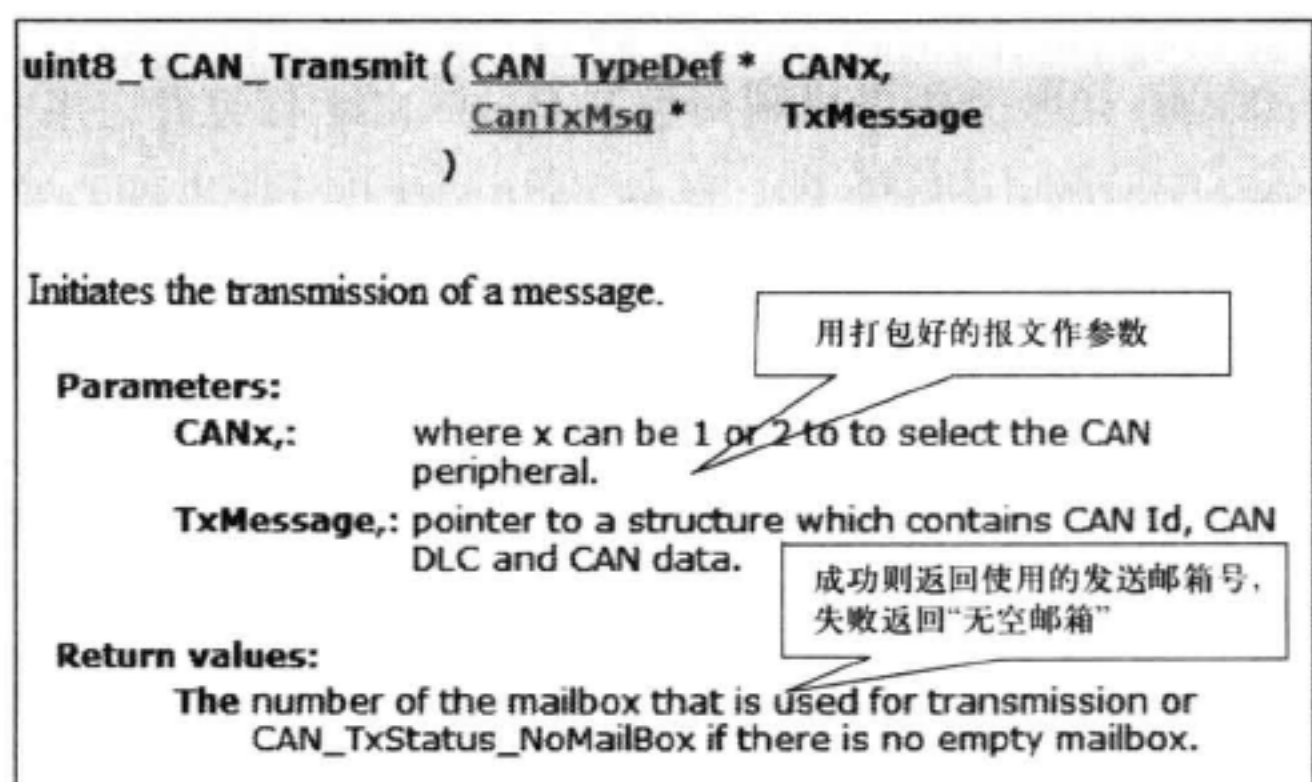


图 16-17 库函数 CAN_Transmit 说明

16.3.7 接收报文、编写中断服务函数

根据我们的实验配置, 主机初始化完成后发送报文, 而从机初始化完成后一直等待接收报文。我们来看从机端是如何接收的。

在从机的 main 函数中, 初始化完成后就一直轮询标志变量 flag, 见代码清单 16-13。

代码清单 16-13 轮询标志变量 flag

```

1. /* 等待主机端的数据 */
2. while( flag == 0xff );
3.
4. printf( "\r\n 成功接收到 " 主机 " 返回的数据 \r\n " );
5. printf( "\r\n 接收到的报文为: \r\n" );
6. printf( "\r\n 扩展 ID 号 ExtId: 0x%x", RxMessage.ExtId );

```

当从机的 FIFO0 收到报文时, 将会进入中断, 在中断服务函数中我们修改 flag 的值, 中断服务函数结束后, main 函数中这句轮询代码因 flag 的改变而结束, 执行紧接下来的 printf() 打印信息到串口。

我们来看看从机的中断服务函数是如何编写的, 见代码清单 16-14。

代码清单 16-14 从机中断服务函数

```

1. void USB_LP_CAN1_RX0_IRQHandler(void)
2. {
3.
4.   CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
5.   /* 比较是否是发送的数据和 ID */
6.   if((RxMessage.ExtId==0x1314) && (RxMessage.IDE==CAN_ID_EXT)
7.       && (RxMessage.DLC==2) && ((RxMessage.Data[1]|RxMessage.Data[0]<<8)==0xABCD))
8.   {
9.       flag = 0;                                // 接收成功
10.
11.   }
12.   else
13.   {
14.       flag = 0xff;                             // 接收失败
15.   }
16. }

```

我们可以从启动文件 startup_stm32f10x_hd.s 中查得 CAN1 的 RX0 的中断服务函数名为 USB_LP_CAN1_RX0_IRQHandler, 然后在 stm32f10x_it.c 文件中添加这个函数。

在中断服务函数中, 我们调用了库函数 CAN_Receive() 把 FIFO0 的报文读取到 main 文件的 CanRxMsg 类型全局变量 RxMessage 中。CAN_Receive() 库函数说明见图 16-18。

接收报文结构体 CanRxMsg 类型与报文的 结构体类似, 见代码清单 16-15。

<pre> void CAN_Receive (CAN_TypeDef * CANx, uint8_t FIFONumber, CanRxMsg * RxMessage) </pre>	
Receives a message.	三个参数分别为: CAN 号、接收 FIFO 号及保存报文的结构体指针
Parameters:	
CANx,:	where x can be 1 or 2 to select the CAN peripheral.
FIFONumber,:	Receive FIFO number, CAN_FIFO0 or CAN_FIFO1.
RxMessage,:	pointer to a structure receive message which contains CAN Id, CAN DLC, CAN datas and FMI number.

图 16-18 CAN_Receive 库函数说明

代码清单 16-15 结构体 CanRxMsg 的定义

```

1. typedef struct
2. {
3.     uint32_t StdId; /* 接收报文的标准 ID */
4.     uint32_t ExtId; /* 接收报文的扩展 ID */
5.     uint8_t IDE; /* 报文的 IDE 位 */
6.     uint8_t RTR; /* 报文的 RTR 位 */
7.     uint8_t DLC; /* 报文的 DLC 段 */
8.     uint8_t Data[8]; /* 报文的数据段 */
9.     uint8_t FMI; /* 过滤器匹配序号 */
10.} CanRxMsg;

```

使用 CAN_Receive() 函数接收了报文后, 我们使用 if 语句判断接收到的报文的 ID 信息、IDE 位、DLC 位及数据段是否等于 0xABCD, 若接收到的报文与我们主机预定发送的报文一样, 则把 flag 位置“0”, 退出中断服务函数。之后由 main 函数中的 printf() 把接收到的报文信息打印到终端。

这样就完成了一次从主机发送报文到从机的过程。第一个主机发送过程结束后, 主机进入轮询状态, 等待从机向它发送数据。而从机把接收到的报文打印到终端后, 打包好数据段为 0xDCBA 的报文发送到 CAN 网络 (这部分的流程详见主从机的 main 函数代码)。当主机收到报文后, 进入它的中断服务函数, 判断接收是否正确, 若正确则把 flag 值置为“0”, 退出中断函数, 由 main 函数的 printf() 打印报文信息到终端, 实验结束。其中主机的中断服务函数见代码清单 16-16。

代码清单 16-16 主机中断服务函数

```

1. void USB_LP_CAN1_RX0_IRQHandler(void)
2. {
3.
4.     /* 从邮箱中读出报文 */
5.     CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
6.
7.     /* 比较 ID 和数据是否为 0x1314 及 DCBA */
8.     if((RxMessage.ExtId==0x1314) && (RxMessage.IDE==CAN_ID_EXT)
9.        && (RxMessage.DLC==2) && ((RxMessage.Data[1]|RxMessage.Data[0]<<8)==0xDCBA))
10.    {
11.        flag = 0; /* 接收成功 */
12.    }
13.    else
14.    {
15.        flag = 0xff; /* 接收失败 */
16.    }
17.}

```

它与从机中断服务函数的区别仅在于数据段的内容判别, 为 0xDCBA。

16.3.8 实验小结

在本节总结一下 STM32 的 CAN 通信配置。

- 1) 初始化 CAN 的 GPIO。
- 2) 配置好 CAN 的工作模式, 主要是对 CAN 的位时序的配置。
- 3) 若需要过滤报文, 要设置 CAN 的过滤器, 根据需求选用标识符列表或标识符屏蔽模式。

4) 在发送报文前需要先对报文打包。

只要对 CAN 协议有清晰的了解, STM32 的 CAN 配置实际上并不复杂, 我们建议读者进一步阅读 CAN 2.0 的标准协议。

16.3.9 实验现象

将配套 STM32 开发板供电 (DC5V), 插上 J-LINK, 插上串口线 (两头都是母的交叉线), 打开超级终端, 配置超级终端为 “115200 8-N-1”, 将编译好的程序分别下载到两块开发板, 根据连接图把两块板子的 CAN 信号线连接起来, 先开启从机电源, 再开启主机电源 (这是由我们实验代码的执行流程决定的)。

若串口线接到 “主机” 板子上, 超级终端打印出如图 16-19 所示信息。

若串口线接到 “从机” 板子上, 超级终端打印出如图 16-20 所示信息。

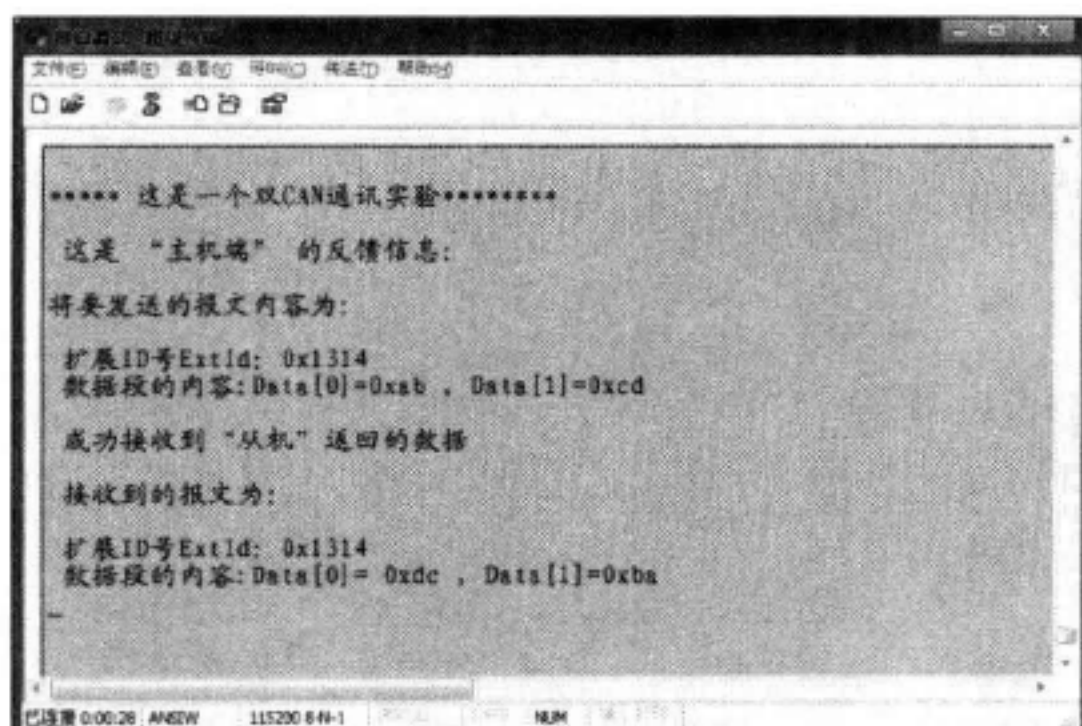


图 16-19 双 CAN 通信主机端信息

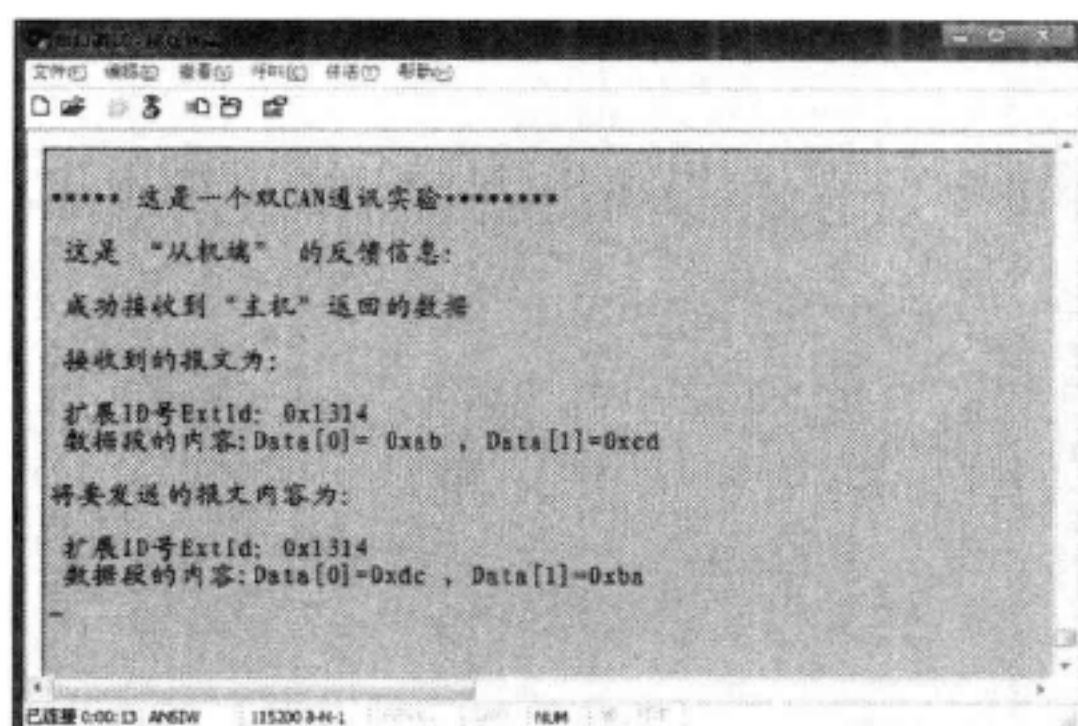
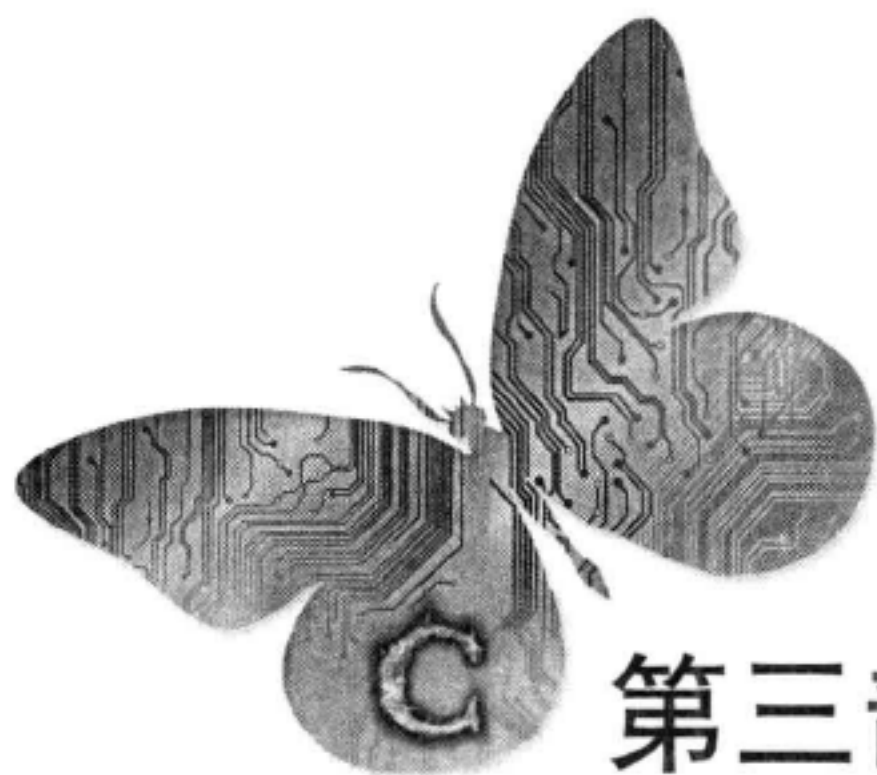


图 16-20 双 CAN 通信从机端信息



第三部分 库开发高级篇

通过初级、中级篇内容的学习，读者对 STM32 库函数的开发已经相当熟悉了，并且也能够利用它进行常用外设、通信接口的开发。在高级篇部分则讲解更复杂的外设接口和工程，如 SDIO 接口、USB 接口、FSMC 及以太网接口，这些通信接口的复杂之处不在于库函数的使用，而在于它们涉及的协议栈、文件系统、接口模拟等内容。

该部分的前三章有着十分紧密的联系，它们是一个完整的 MP3 例程，包括了最底层的 SDIO 驱动、文件系统的移植及应用层 MP3 文件的播放，从这个例程读者可以完成一次从驱动到系统再到应用层的旅行，感受代码之美。当然，除了美，还可以把它投入到实际的应用中。

- ❑ 第 17 章 SDIO 之 SD 卡驱动
- ❑ 第 18 章 文件系统之 FATFS_R0.09
- ❑ 第 19 章 MP3 播放器
- ❑ 第 20 章 USB 大容量存储器实例
- ❑ 第 21 章 LCD 触摸屏画板
- ❑ 第 22 章 字库及 BMP 图片显示
- ❑ 第 23 章 OV7670 摄像头驱动
- ❑ 第 24 章 以太网及 LwIP 协议栈移植
- ❑ 第 25 章 Wi-Fi 模块 EMW3180 驱动



第 17 章

SDIO 之 SD 卡驱动

17.1 SD 协议简介

大多数人原来没有了解过 SD 协议，又看到 SDIO 的驱动有 2000 多行，感觉无从下手。下面结合 STM32 的 SDIO 接口分析 SD 协议，让大家对它先有个大概了解，更具体的说明在后面代码中展开。

17.1.1 卡的种类

首先要清楚，可以使用 SDIO 接口通信的不只是我们常见的单纯用于存储数据的 SD 存储卡，还有 SDIO 卡、MMC 卡。其中 SDIO 卡与 SD 存储卡是有区别的，SDIO 卡实际上就是利用 SDIO 接口的一些模块，插入 SD 的插槽中，以扩展设备的功能，如 SDIO Wi-Fi SDIO CMOS 相机等。见图 17-1。

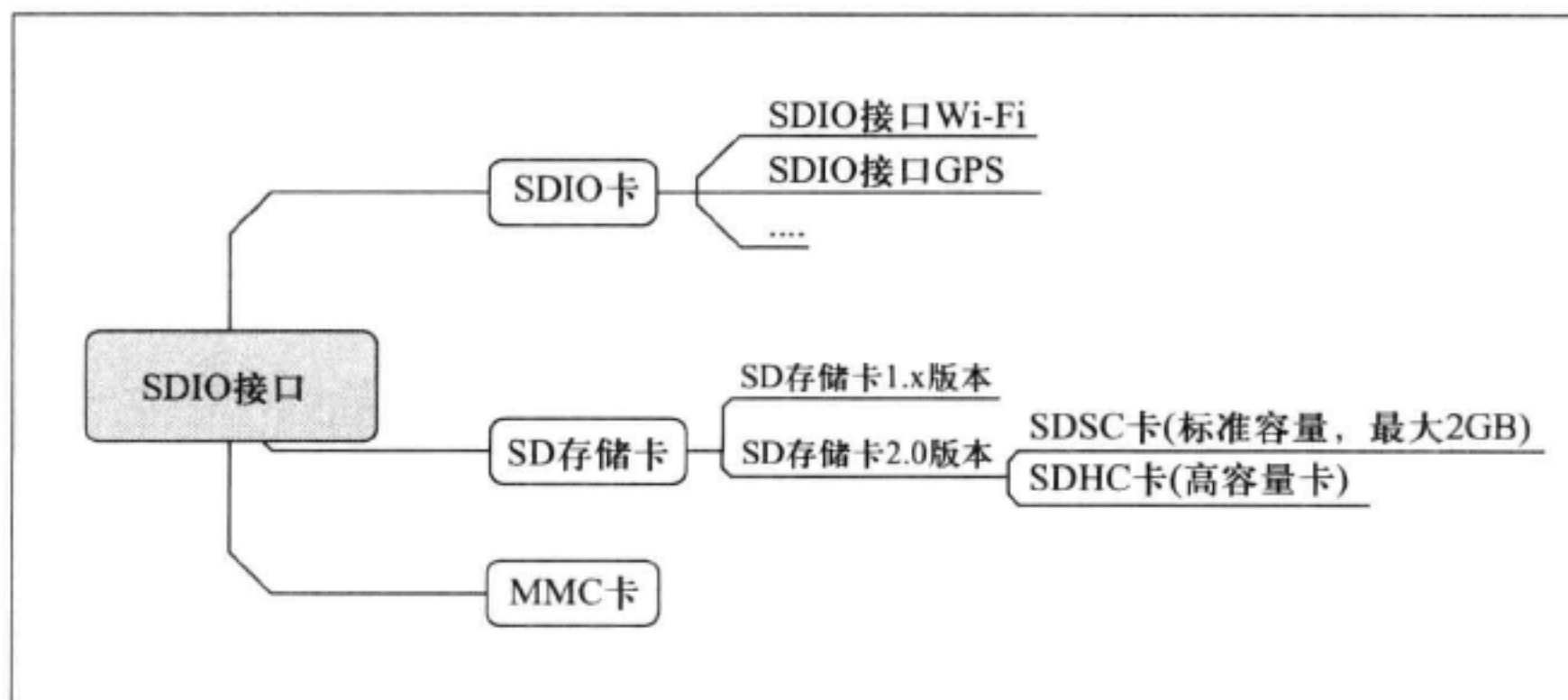


图 17-1 使用 SDIO 接口的设备种类

本章只讲解针对 SD 存储卡（后面简称 SD 卡）的使用，SD 卡主要分为两个版本，有 1.x 版和现在普遍使用的 2.0 版，2.0 版是为了适应大容量 SD 卡提出的标准，这个标准把 SD 卡分为 SDSC 卡和 SDHC 卡。其中 SDSC 卡为标准容量，容量最大不超过 2GB，超过 2GB 的都属于高容量的 SDHC 卡。

由于 SDIO 可支持各种设备，所以对 SDIO 接口进行初始化，上电后就要对它接入的卡进行识别、分类，这个过程是主机向卡发送一系列不同的命令，根据卡不同的响应来进行分类。

17.1.2 SDIO 基本架构

图 17-2 为一个 SDIO 主机与两个 SD、SDIO 类型卡的连接方式。SDIO 接口包含 CLK、CMD 及 4 条 DAT[3:0] 信号线。这 6 条信号线都是共用总线，即新加入的设备可以并联接入 SDIO 接口。SDIO 主机是通过命令和 SD 从设备的响应来寻址的，所以不需要片选信号线。

(1) CLK

CLK 是卡的时钟信号线，由主机产生时钟信号，SD 卡和 SDIO 卡的时钟频率可为 0 ~ 25 MHz。在命令和数据线上，每个时钟周期传输 1 位命令或数据。

(2) CMD

CMD 为命令信号线，SDIO 的所有由主机发出的命令及从机对命令的响应都是在这条信号线上传输的。

(3) DAT[3:0]

DAT[3:0] 表示 4 条数据线，主机和从机的数据信号在这 4 条线上传输。

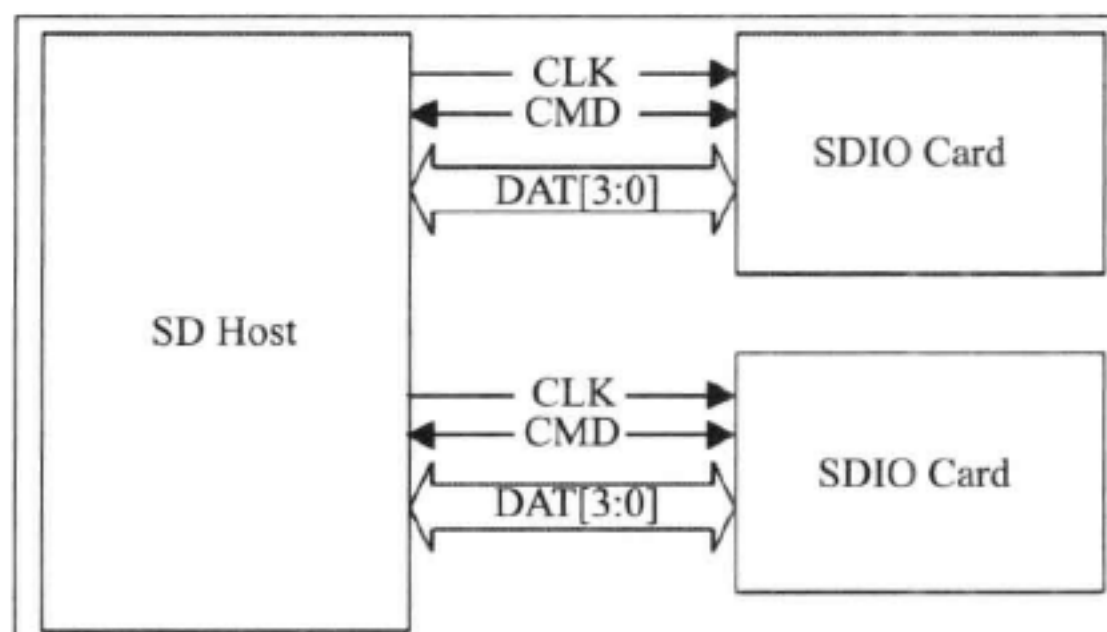


图 17-2 SDIO 连接图

17.2 STM32 的 SDIO 接口

STM32 的 SDIO 接口与标准 SDIO 有稍许区别，区别在于数据线的数量。见图 17-3，它具有 8 条数据线——SDIO_D[7:0]，STM32 这样设计是为了支持部分 MMC 存储卡，这些 MMC 卡使用的是 8 条数据线。

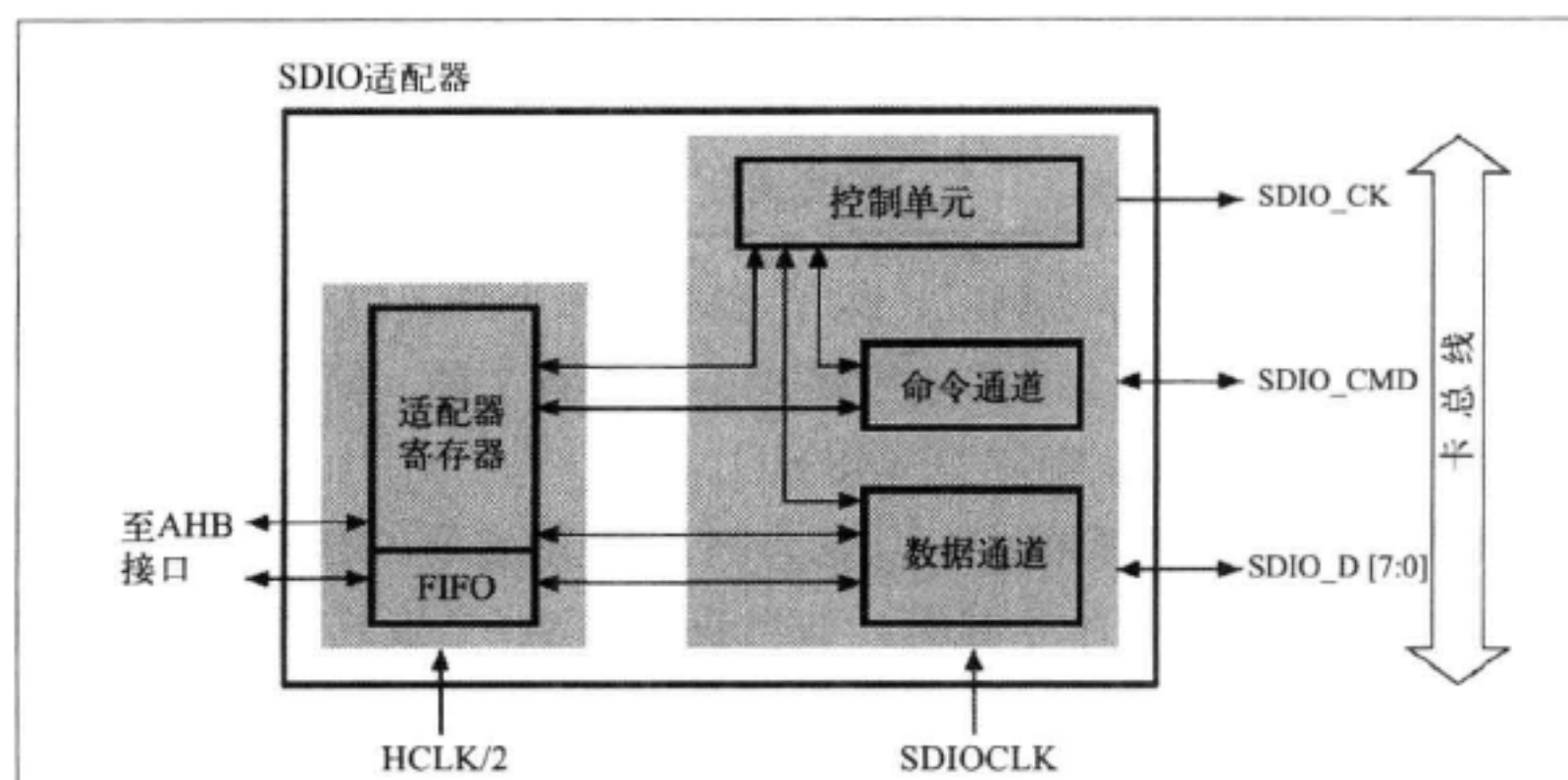


图 17-3 STM32 的 SDIO 接口

17.2.1 从 SDIO 的时钟说起

SDIO_CK 是 SDIO 接口与 SD 卡用于同步的时钟信号。它的来源可以是 HCLK 二分频后的时钟或直接从 SDIOCLK 得到。

SDIO 接口挂载到 AHB 总线上，若使用 HCLK/2 输入到适配器得到 SDIO_CK 的时钟，实际上 SDIO_CK 还会对 HCLK/2 分频，最终 $SDIO_CK = HCLK / (2 + CLKDIV)$ 。其中 CLKDIV 是我们初始化时在寄存器 SDIO_CLK 中配置的分频位。

若使用 SDIOCLK 作为 SDIO_CK 的时钟来源，可以通过设置 bypass 模式直接得到，这时 $SDIO_CK = SDIOCLK = HCLK$ 。

17.2.2 SDIO 的命令格式

SDIO 的所有命令及命令响应，都是通过 SDIO_CMD 引脚来传输的，且命令只能由主机即 STM32 的 SDIO 控制器发出。

SDIO 协议规定了非常多的命令，把这些命令分类别来整理也有 11 种之多，包括基本命令、块读取命令、块写入命令、写保护命令、擦除命令、卡上锁命令、应用指定命令、I/O 模式命令、功能选择命令及特殊的应用命令 ACMD。其中，在使用 ACMD 命令前，要首先向卡发送编号为 CMD55 的应用指定命令。

参照命令格式表，见表 17-1。一个命令包含了 6 个段，分别为起始位、传输位、命令索引、参数、CRC7 和结束位，其中除了命令索引段和参数段是需要我们在软件配置的时候设置的，其他段都由硬件完成。命令索引段是指 SD 协议规定的命令编码，如命令 CMD0、CMD1…的编码为 0、1…。有的命令会包含参数，如读命令的参数为要读取数据的地址，这些命令参数存放在参数段。

表 17-1 SDIO 命令格式

位	47	46	[45:40]	[39:8]	[7:1]	0
宽度（位）	1	1	6	32	7	1
数值	'0'	'1'	x	x	x	'1'
描述	起始位	传输位	命令索引	参数	CRC7	结束位

SD 卡对主机的各种命令回复称为响应，除了 CMD0 命令外，SD 卡在接收到命令后都会返回一个响应。对于不同的命令，会有不同的响应格式，一共有 7 种，简称 R1 ~ R7。按响应的字节长度又分为长响应型（136 bit）和短响应型（48 bit）。

见表 17-2，这是响应 6（R6）的格式。以它为例，R6 的主要内容为命令索引段和参数段。

命令索引段的内容为它响应的命令编码，如当我们向 SD 卡发送 CMD3（编码为 000011）命令时，它返回的响应 R6 的命令索引段内容即为 000011。

参数段的内容为命令响应参数，如这个响应 R6 的内容即为 RCA（卡的相对地址）及 card status（卡的状态）。

表 17-2 SDIO 命令响应格式 (R6)

位	47	46	[45:40]	[39:8] 参数段		[7:1]	0
宽度 (位)	1	1	6	16	16	7	1
数值	'0'	'0'	x	x	x	x	'1'
描述	起始位	传输位	命令索引 ('000011')	要重新设置的卡 相对地址 RCA[31:16]	[15:0] 卡状态 位: 23,22,19,12:0	CRC7	结束位

所以当我们需要知道 RCA 和卡状态时,可以向卡发送 CMD3 命令,然后等待 SD 卡对命令的响应。SDIO 接口通过 CMD 信号线接收到响应后,由硬件去除响应的头尾信息,把命令索引段的内容保存到 SDIO_RESPCMD 寄存器,把参数段内容存储到 SDIO_RESPx 寄存器中。然后通过软件读取这两个寄存器即可获得所需信息。

17.2.3 数据传输格式

SD 卡的数据写入、读取的最小单位是块,每块的大小为 512 字节。见图 17-4,为多个数据块的写入过程。首先软件通过 SDIO 接口的 CMD 信号线发送多块写入的命令,接收到正常的响应后,要写入的数据线从 4 根 DAT 信号线传输出去,每块结束后是 CRC 校验码。接着要检测忙状态,数据传输到 SD 卡后,SD 卡启动内部时序保存数据,这时 SD 卡会把 DAT0 信号线拉低,表示处于“忙”状态,忙状态结束后,主机才能发送下一个数据块的数据。

对 SD 卡的操作一般是大吞吐量的数据传输,可以采用 DMA 来提高效率,SDIO 采用的是 DMA2 中的通道 4。

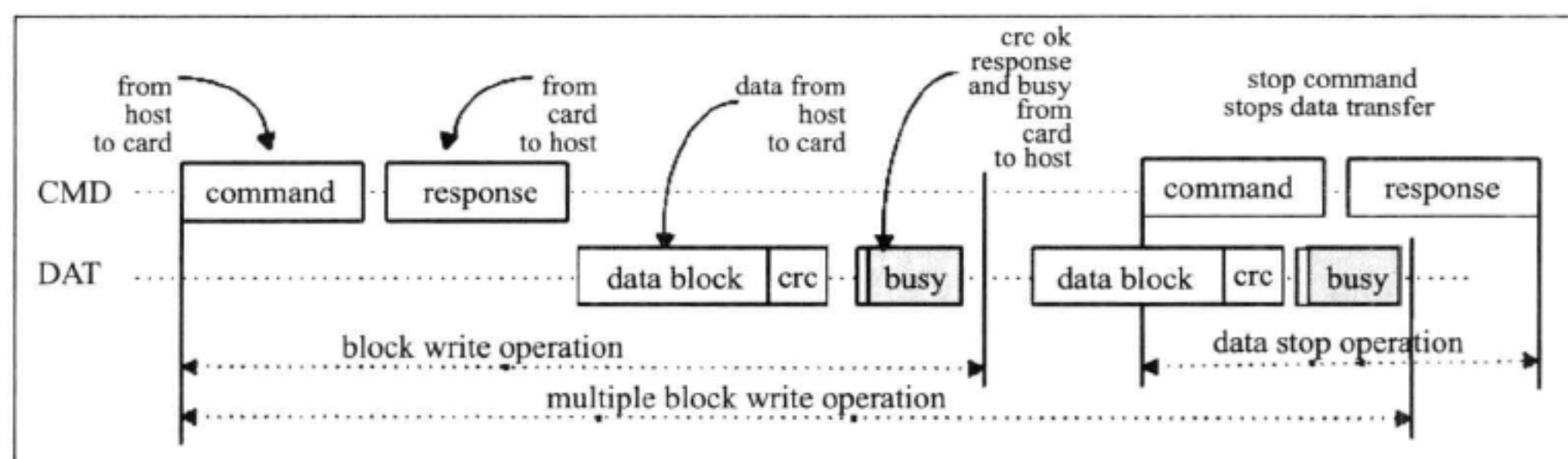


图 17-4 多个数据块写入时序

17.3 SD 卡读写实验分析

17.3.1 实验描述及工程文件清单

1. 实验描述

MicroSD 卡 (SDIO 模式) 测试实验,采用 4bit 数据线、DMA 模式。没有使用文件系统,只

是单纯地读 block 并将测试信息通过串口 1 在计算机的超级终端上打印出来。本实验是使用文件系统的基础。

2. 硬件连接

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

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_sdio.c
- ☐ FWlib/stm32f10x_dma.c
- ☐ FWlib/misc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/usart1.c
- ☐ USER/sdio_sdcard.c (官方 SDIO 库的 4.5 版)

MicroSD 卡硬件原理图见图 17-5。

17.3.2 配置工程环境

本 SD 卡读写实验中我们用到了 GPIO、RCC、USART、DMA 及 SDIO 外设，所以先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_dma.c、stm32f10x_sdio.c。本实验中使用了中断，在数据传输完成时会进入中断服务程序，所以还要添加 misc.c 文件。

接下来添加旧工程中的外设用户文件 usart.c，以便调试和观察实验效果。新建 sdio_sdcard.c 及 sdio_sdcard.h 文件，并在 stm32f10x_conf.h 中把用到的 ST 库的头文件注释去掉。见代码清单 17-1。

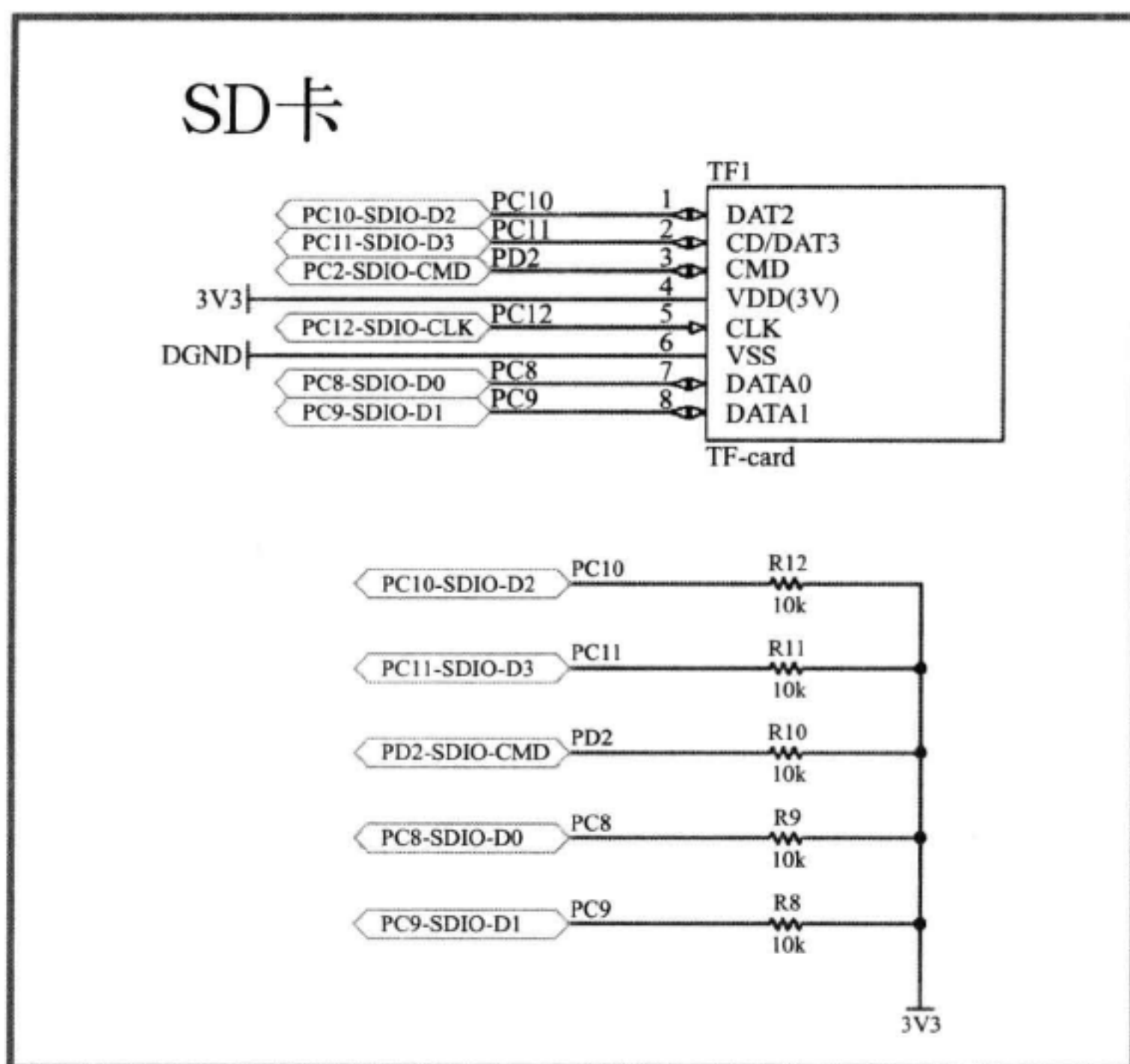


图 17-5 配套 STM32 开发板 MicroSD 卡硬件原理图

代码清单 17-1 SDIO 例程的 stm32f10x_conf.h 文件配置

```

1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****/
9. #include "stm32f10x_dma.h"
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
12. #include "stm32f10x_sdio.h"
13. #include "stm32f10x_usart.h"
14. #include "misc.h"

```

利用官方驱动例程

部分读者打开本工程的 sdio_sdcard.c 文件，看了一眼，发现单单是这个文件就有 2500 行！然后很无奈地在论坛发帖子：“野火，SD 例程的这 2500 行代码都要自己写吗？要不要去理解它？”

显然，这些读者还没发现利用 STM32 库开发的一个小秘密，在我们下载的官方库文件 stm32f10x_stdperiph_lib 中，除了常用的帮助文档、外设库文件，还有官方的例程。对于如本书前

两部分简单的外设，我们可以轻松地独立开发，但如 SDIO、USB 这些复杂的外设，它们的通信协议相当庞大，要自行编写完整、严谨的驱动不是一件轻松的事情，这时就可以利用官方例程的驱动文件，所以并不需要完全自己编写本驱动。

那要不要理解这个例程呢？我们认为有必要理解编写本驱动的思想、流程。若要求再低一点，则至少要会调用它的函数。在本 SD 卡的原版官方例程中有一点错误，绝大部分时候无法实现正常地读写 SD 卡，我们在它的基础上添加了部分必要的代码，使它成为真正可用的工程。

原版的官方驱动在哪里找呢？见图 17-6 及图 17-7。



图 17-6 SDIO-SD 卡原版官方驱动

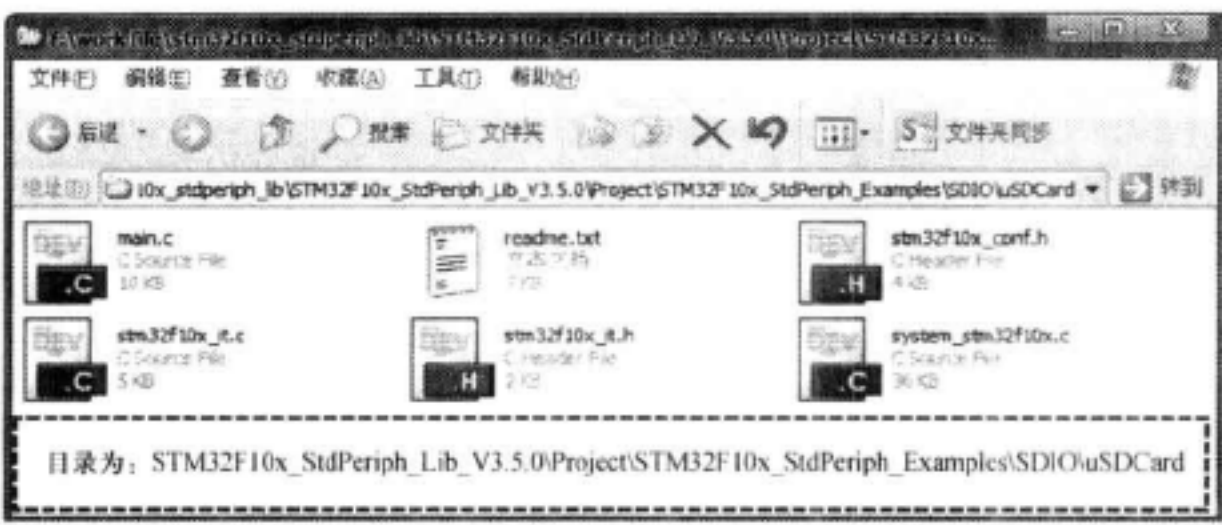


图 17-7 原版驱动工程的其他文件

官方的驱动例程都只保留了源文件，还需要我们把它整合为工程。它的驱动源文件 stm32_eval_sdio_sd.c 存放在 STM32 库的 Utilities 文件夹下，工程所需的 main.c、stm32f10x_it.c 文件存放在 STM32 库的 Project 文件夹下，利用这些文件整理出来的工程是以 STM32 官方的 EVAL 评估开发板为平台的。官方对库文件和驱动例程做了不同的版本编号，我们使用的库是 3.5 版本，这个 SDIO 驱动例程是在相应的 3.5 版本的库中，而官方对本 SDIO 驱动例程文件编号为 4.5 版。

17.3.3 main 文件

由于本实验代码庞大，我们挑选重点部分来分析。首先仍从 main 函数开始，见代码清单 17-2。

代码清单 17-2 SDIO 例程的 main 函数

```
1. int main(void)
2. {
3.
4.     /* 进入到main函数前，启动文件 startup(startup_stm32f10x_xx.s) 已经调用了在    system_
       stm32f10x.c 中的 SystemInit()，配置好了系统时钟，在外部晶振 8MHz 的条件下，设置 HCLK = 72MHz */
5.
6.     /* Interrupt Config */
7.     NVIC_Configuration();
8.
9.     /* USART1 config */
10.    USART1_Config();
11.
12.    /*----- SD Init ----- */
13.    Status = SD_Init();
14.
15.    printf( "\r\n 这是一个MicroSD卡实验（没有跑文件系统）.....\r\n " );
```



```

16.
17.
18.  if (Status == SD_OK) // 检测初始化是否成功
19.  {
20.      printf( " \r\n SD_Init 初始化成功 \r\n " );
21.  }
22.  else
23.  {
24.      printf("\r\n SD_Init 初始化失败 \r\n" );
25.      printf("\r\n 返回的 Status 的值为: %d \r\n", Status );
26.  }
27.
28.  printf( " \r\n CardType is : %d ", SDCardInfo.CardType );
29.  printf( " \r\n CardCapacity is : %d ", SDCardInfo.CardCapacity );
30.  printf( " \r\n CardBlockSize is : %d ", SDCardInfo.CardBlockSize );
31.  printf( " \r\n RCA is : %d ", SDCardInfo.RCA);
32.  printf( " \r\n ManufacturerID is : %d \r\n", SDCardInfo.SD_cid.
    ManufacturerID );
33.
34.  SD_EraseTest(); // 擦除测试
35.
36.  SD_SingleBlockTest(); // 单块读写测试
37.
38.  SD_MultiBlockTest(); // 多块读写测试
39.
40.  while (1)
41.  {}
42.}

```

若忽略输出调试信息的 printf() 语句, main 函数的流程将简单明了:

1) 第 7 行, 调用 NVIC_Configuration() 初始化 SDIO 的中断。

2) 第 10 行, 调用 USART1_Config() 配置用于返回调试信息的串口。

3) 第 13 行, 调用 SD_Init() 开始进行 SDIO 的初始化, 并把返回值赋给 Status 变量, 若 Status 变量等于枚举元素 SD_OK 则初始化成功。在 SD_Init() 函数中还对全局结构体变量 SDCardInfo 进行了赋值, 所以在后面可以使用 printf() 来输出调试信息。

4) 第 34 ~ 38 行, 分别用 SD_EraseTest()、SD_SingleBlockTest() 和 SD_MultiBlockTest() 进行擦除测试、单块读写测试、多块读写测试。

17.3.4 SDIO 初始化

下面我们先进入 SDIO 驱动函数中最复杂的函数 SD_Init() 进行分析, 见代码清单 17-3。

代码清单 17-3 SD_Init() 函数

```

1.  /*
2.   * 函数名: SD_Init
3.   * 描述   : 初始化 SD 卡, 使卡处于就绪状态 (准备传输数据)
4.   * 输入   : 无
5.   * 输出   : -SD_Error SD 卡错误代码
6.   *         成功时则为 SD_OK
7.   * 调用   : 外部调用

```

```

8.  */
9. SD_Error SD_Init(void)
10. {
11.     /* 重置 SD_Error 状态 */
12.     SD_Error errorstatus = SD_OK;
13.
14.     /* SDIO 外设底层引脚初始化 */
15.     GPIO_Configuration();
16.
17.     /* 对 SDIO 的所有寄存器进行复位 */
18.     SDIO_DeInit();
19.
20.     /* 上电并进行卡识别流程, 确认卡的操作电压 */
21.     errorstatus = SD_PowerON();
22.
23.     /* 如果上电, 识别不成功, 返回“响应超时”错误 */
24.     if (errorstatus != SD_OK)
25.     {
26.         /*!< CMD Response TimeOut (wait for CMDSENT flag) */
27.         return(errorstatus);
28.     }
29.
30.     /* 卡识别成功, 进行卡初始化 */
31.     errorstatus = SD_InitializeCards();
32.
33.     if (errorstatus != SD_OK) // 失败返回
34.     {
35.         /*!< CMD Response TimeOut (wait for CMDSENT flag) */
36.         return(errorstatus);
37.     }
38.
39.     /*!< Configure the SDIO peripheral
40.     上电识别、卡初始化都完成后, 进入数据传输模式, 提高读写速度
41.     速度若超过 24MHz 要进入 bypass 模式
42.     !< on STM32F2xx devices, SDIOCLK is fixed to 48MHz
43.     !< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_TRANSFER_CLK_DIV) */
44.     SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
45.     SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
46.     // 上升沿采集数据
47.     SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;
48.     // 时钟频率若超过 24MHz, 要开启此模式
49.     SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable; // 若开启
        此功能, 在总线空闲时关闭 SD_CLK 时钟
50.     SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b; // 1 位模式
51.     SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
        Disable; // 硬件流, 若开启, 在 FIFO 不能进行发送和接收数据时, 数据传输暂停
52.     SDIO_Init(&SDIO_InitStructure);
53.
54.     if (errorstatus == SD_OK)
55.     {
56.         /*----- Read CSD/CID MSD registers -----*/
57.         errorstatus = SD_GetCardInfo(&SDCardInfo); // 用来读取 CSD/CID 寄存器
58.
59.     }
60.
61.     if (errorstatus == SD_OK)
62.     {

```

```

63.    /*----- Select Card -----*/
64.    errorstatus = SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));    // 通过
    CMD7 ,RCA 选择要操作的卡
65.    }
66.
67.    if (errorstatus == SD_OK)
68.    {
69.        errorstatus = SD_EnableWideBusOperation(SDIO_BusWide_4b);
70.    // 开启 4bit 模式
71.    }
72.
73.    return(errorstatus);
74.}

```

先从整体上了解这个 SD_Init() 函数：

1) 本函数的类型 SD_Error 是一个自定义的枚举变量，它对 40 多种 SD 错误类型进行了编号，若它的值为枚举元素 SD_OK，则表明 SDIO 初始化成功。

2) 第 15 行，调用函数 GPIO_Configuration() 进行 SDIO 的复用引脚、外设时钟初始化。

3) 第 21 ~ 37 行，分别调用了 SD_PowerON() 和 SD_InitializeCards() 函数，这两个函数共同实现了卡的上电、检测、识别的流程。

4) 第 44 ~ 50 行，填充 SDIO 外设的初始化结构体，调用库函数 SDIO_Init() 初始化 SDIO。配置 SDIO 使用 HCLK/2 为时钟源，再对分频因子赋值为宏 SDIO_TRANSFER_CLK_DIV（数值为 1），所以最终 SDIO_CK 的频率为 24MHz。并把数据线宽度设置为 1 位（在后面的代码再配置为 4 位），开启硬件流（在读写数据的时候，开启硬件流是很有必要的，可以减少出错）。

5) 第 57 行，调用函数 SD_GetCardInfo() 获取 SD 卡的 CSD 寄存器中的内容，它具有 128 位，主要包含了卡的性能、容量等信息。在 main 函数里输出到串口的调试信息有一部分就是这个时候从卡读取得到的。

6) 第 64 行，调用 SD_SelectDeselect() 选定后面即将要控制的 SD 卡，输入的参数 RCA 是卡的相对地址（relative address）。若 SDIO 连接着多个 SD 卡，主机在初始化 SDIO 时会给不同的 SD 卡分配不同的编号（SD_InitializeCards() 函数中），之后主机对 SD 卡寻址时就使用这个编号，这个编号即为卡相对地址。

7) 调用函数 SD_EnableWideBusOperation() 开启 4bit 数据线模式。

如果 SD_Init() 函数能够执行完整流程，并且返回值是 SD_OK 则说明初始化成功，接下来就可以开始进行擦除、读写的操作了。

17.3.5 卡的上电识别流程

下面分析在 SD_Init() 函数中调用的 SD_PowerON() 函数，分析完这个函数大家就能了解 SDIO 如何接收、发送命令了。见代码清单 17-4。

代码清单 17-4 SD_PowerON() 函数

```

1.  /*
2.   * 函数名：SD_PowerON

```



```

3.  * 描述   : 确保 SD 卡的工作电压和配置控制时钟
4.  * 输入   : 无
5.  * 输出   : -SD_Error SD 卡错误代码
6.  *         成功时则为 SD_OK
7.  * 调用   : 在 SD_Init() 调用
8.  */
9. SD_Error SD_PowerON(void)
10. {
11.     SD_Error errorstatus = SD_OK;
12.     uint32_t response = 0, count = 0, validvoltage = 0;
13.     uint32_t SDType = SD_STD_CAPACITY;
14.
15.     /*!< Power ON Sequence -----*/
16.     /*!< Configure the SDIO peripheral */
17.     /*!< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_INIT_CLK_DIV) */
18.     /*!< on STM32F2xx devices, SDIOCLK is fixed to 48MHz */
19.     /*!< SDIO_CK for initialization should not exceed 400 kHz */
20.     /* 初始化时的时钟不能大于 400kHz*/
21.     SDIO_InitStructure.SDIO_ClockDiv = SDIO_INIT_CLK_DIV;
22.     /* HCLK = 72MHz, SDIOCLK = 72MHz, SDIO_CK = HCLK/(178 + 2) = 400 kHz */
23.     SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
24.     SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;
25.     // 不使用 bypass 模式, 直接用 HCLK 进行分频得到 SDIO_CK
26.     SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
27.     // 空闲时不关闭时钟电源
28.     SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;
29.     // 1 位数据线
30.     SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
        Disable; // 硬件流
31.     SDIO_Init(&SDIO_InitStructure);
32.
33.     /*!< Set Power State to ON */
34.     SDIO_SetPowerState(SDIO_PowerState_ON);
35.
36.     /*!< Enable SDIO Clock */
37.     SDIO_ClockCmd(ENABLE);
38.
39.     /* 下面发送一系列命令, 开始卡识别流程 */
40.     /*!< CMD0: GO_IDLE_STATE -----*/
41.     /*!< No CMD response required */
42.     SDIO_CmdInitStructure.SDIO_Argument = 0x0;
43.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_GO_IDLE_STATE; //CMD0
44.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_No; // 无响应
45.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
46.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
47.     // 则 CPSM 在开始发送命令之前等待数据传输结束
48.     SDIO_SendCommand(&SDIO_CmdInitStructure); // 写命令进命令寄存器
49.
50.     errorstatus = CmdError(); // 检测是否正确接收到 CMD0
51.
52.     if (errorstatus != SD_OK) // 命令发送出错, 返回
53.     {
54.         /*!< CMD Response TimeOut (wait for CMDSENT flag) */
55.         return(errorstatus);
56.     }
57.

```

```

58.  /*!< CMD8: SEND_IF_COND -----*/
59.  /*!< Send CMD8 to verify SD card interface operating condition */
60.  /*!< Argument: - [31:12]: Reserved (shall be set to '0')
61.                  - [11:8]: Supply Voltage (VHS) 0x1 (Range: 2.7-3.6 V)
62.                  - [7:0]: Check Pattern (recommended 0xAA) */
63.  /*!< CMD Response: R7 */
64.  SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN;
65.  // 接收到命令 SD 会返回这个参数
66.  SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND; //CMD8
67.  SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //R7
68.  SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
69. // 关闭等待中断
70.  SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
71.  SDIO_SendCommand(&SDIO_CmdInitStructure);
72.
73.  /* 检查是否接收到命令 */
74.  errorstatus = CmdResp7Error();
75.
76.  if (errorstatus == SD_OK) // 有响应则卡遵循 SD 协议 2.0 版本
77.  {
78.      CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0;
79.  /*!< SD Card 2.0, 先把它定义为 SDSC 类型的卡 */
80.      SDType = SD_HIGH_CAPACITY;
81.  // 这个变量用作 ACMD41 的参数, 用来询问是 SDSC 卡还是 SDHC 卡
82.  }
83.  else // 无响应, 说明是 1.x 的或 MMC 的卡
84.  {
85.      /*!< CMD55 */
86.      SDIO_CmdInitStructure.SDIO_Argument = 0x00;
87.      SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
88.      SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
89.      SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
90.      SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
91.      SDIO_SendCommand(&SDIO_CmdInitStructure);
92.      errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
93.  }
94.  /*!< CMD55 */
95.  // 发送 CMD55, 用于检测是 SD 卡还是 MMC 卡, 或是不支持的卡
96.      SDIO_CmdInitStructure.SDIO_Argument = 0x00;
97.      SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
98.      SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //R1
99.      SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
100.      SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
101.      SDIO_SendCommand(&SDIO_CmdInitStructure);
102.      errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
103.      // 是否响应, 无响应的是 MMC 或不支持的卡
104.
105.      /*!< If errorstatus is Command TimeOut, it is a MMC card */
106.      /*!< If errorstatus is SD_OK it is a SD card: SD card 2.0 (voltage range mismatch) or SD card 1.x */
107.      if (errorstatus == SD_OK)
108.      // 响应了 CMD55, 是 SD 卡, 可能为 1.x, 可能为 2.0
109.      {
110.          /* 下面开始循环地发送 SDIO 支持的电压范围, 循环一定次数 */
111.

```

```

112.         /*!< SD CARD */
113.         /*!< Send ACMD41 SD_APP_OP_COND with Argument 0x80100000 */
114.         while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL))
115.         {
116.
117.             // 因为下面要用到 ACMD41, 是 ACMD 命令, 在发送 ACMD 命令前都要先向卡发送 CMD55
118.             /*!< SEND CMD55 APP_CMD with RCA as 0 */
119.             SDIO_CmdInitStructure.SDIO_Argument = 0x00;
120.             SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;    //CMD55
121.             SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
122.             SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
123.             SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
124.             SDIO_SendCommand(&SDIO_CmdInitStructure);
125.
126.             errorstatus = CmdResplError(SD_CMD_APP_CMD); // 检测响应
127.
128.             if (errorstatus != SD_OK)
129.             {
130.                 return(errorstatus); // 没响应 CMD55, 返回
131.             }
132.
133.             //ACMD41, 命令参数由支持的电压范围及 HCS 位组成, HCS 位置 1 来区分卡是 SDSC 还是 SDHC
134.             SDIO_CmdInitStructure.SDIO_Argument = SD_VOLTAGE_WINDOW_
SD | SDType; // 参数为主机可供电电压范围及 HCS 位
135.             SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_APP_OP_COND;
136.             SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //R3
137.             SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
138.             SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
139.             SDIO_SendCommand(&SDIO_CmdInitStructure);
140.
141.             errorstatus = CmdResp3Error(); // 检测是否正确接收到数据
142.             if (errorstatus != SD_OK)
143.             {
144.                 return(errorstatus); // 没正确接收到 ACMD41, 出错, 返回
145.             }
146.             /* 若卡需要电压在 SDIO 的供电电压范围内, 会自动上电并标志 pwr_up 位 */
147.             response = SDIO_GetResponse(SDIO_RESP1);
148.             // 读取卡寄存器、卡状态
149.             validvoltage = (((response >> 31) == 1) ? 1 : 0);
150.             // 读取卡的 OCR 寄存器的 pwr_up 位, 看是否已工作在正常电压
151.             count++; // 计算循环次数
152.         }
153.         if (count >= SD_MAX_VOLT_TRIAL) // 循环检测超过一定次数还没上电
154.         {
155.             errorstatus = SD_INVALID_VOLTRANGE; //SDIO 不支持卡的供电电压
156.
157.             return(errorstatus);
158.         }
159.         /* 检查卡返回信息中的 HCS 位 */
160.         if (response &= SD_HIGH_CAPACITY)
161.             // 判断 OCR 中的 CCS 位, 如果是 SDSC 卡则不执行下面的语句
162.             {
163.                 CardType = SDIO_HIGH_CAPACITY_SD_CARD;
164.                 // 把卡类型从初始化的 SDSC 改为 SDHC
165.             }
166.

```



```
167.         } /*!< else MMC Card */
168.
169.         return(errorstatus);
170.     }
```

本函数的代码虽然很长，但很多部分是类似的，只是发送的命令或接收的命令响应不同。下面对这几个部分进行总体分析：

- 1) 第 21 ~ 31 行，调用库函数 SDIO_Init()，做好 SDIO 外设最基本的配置，在这部分要注意初始化的时钟信号频率。SDIO_CK 的时钟分为两个阶段，在初始化阶段 SDIO_CK 的频率要小于 400 kHz，初始化完成后可把 SDIO_CK 调整成高速模式，高速模式时超过 24 MHz 要开启 bypass 模式，对于 SD 存储卡即使开启 bypass 模式，最高频率也不能超过 25 MHz。
- 2) 第 33 ~ 37 行，调用库函数 SDIO_SetPowerState() 和 SDIO_ClockCmd() 使 SDIO 上电，并使能它的外设时钟。
- 3) 发送命令部分。上电之后就开始对 SD 卡发送各种不同的命令，发送命令时是通过调用库函数 SDIO_SendCommand() 把填充的命令结构体发送出去的。该命令结构体包含了 5 个成员：命令编码（command index 段）、命令参数（argument 段）、命令的响应类别（长、短或无响应）、是否等待中断及是否使用 CPSM 发送命令，如第 37 ~ 43 行为发送 CMD0 命令。
- 4) 处理命令响应部分。大多 SD 命令被发出后，SD 卡都会有不同的响应返回到主机，根据不同的响应使用不同的函数进行处理。如第 50 行的 CmdError() 用于处理 CMD0 命令的响应，第 74 行的 CmdResp7Error() 函数用于处理类型为 R7 的响应，还有其他针对 R1 ~ R6 的函数，这些并非库函数，而是官方例程利用库编写的。

这个函数调用了 SDIO_SetPowerState() 函数对 SDIO 上电后，开始发送的各种命令及其流程见图 17-8。

代码中所有的判断语句都是根据这个图的各个识别走向展开的，最终把卡区分为 1.0 版的 SD 存储卡、2.0 版的 SDSC 卡和 2.0 版的 SDHC 卡。

这个代码流程的重点为 CMD8 及 ACMD41 命令。

(1) CMD8 命令（见表 17-3）

表 17-3 CMD8 命令格式

位	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
宽度 (位)	1	1	6	20	4	8	7	1
数值	'0'	'1'	'001000'	'00000h'	x	x	x	'1'
描述	起始位	传输位	命令索引	保留位	供给电压 (VHS)	检测模式	CRC7	结束位

CMD8 命令中的 VHS 用来确认主机 SDIO 是否支持卡的工作电压，它的命令响应为 R7，见表 17-4。命令中的第 8 ~ 15 位（检测模式段）可以是任何数值，由我们使用的软件设置，若 SDIO 支持卡的工作电压，卡会把接收到的检测模式数值在命令响应中原样返回给主机。

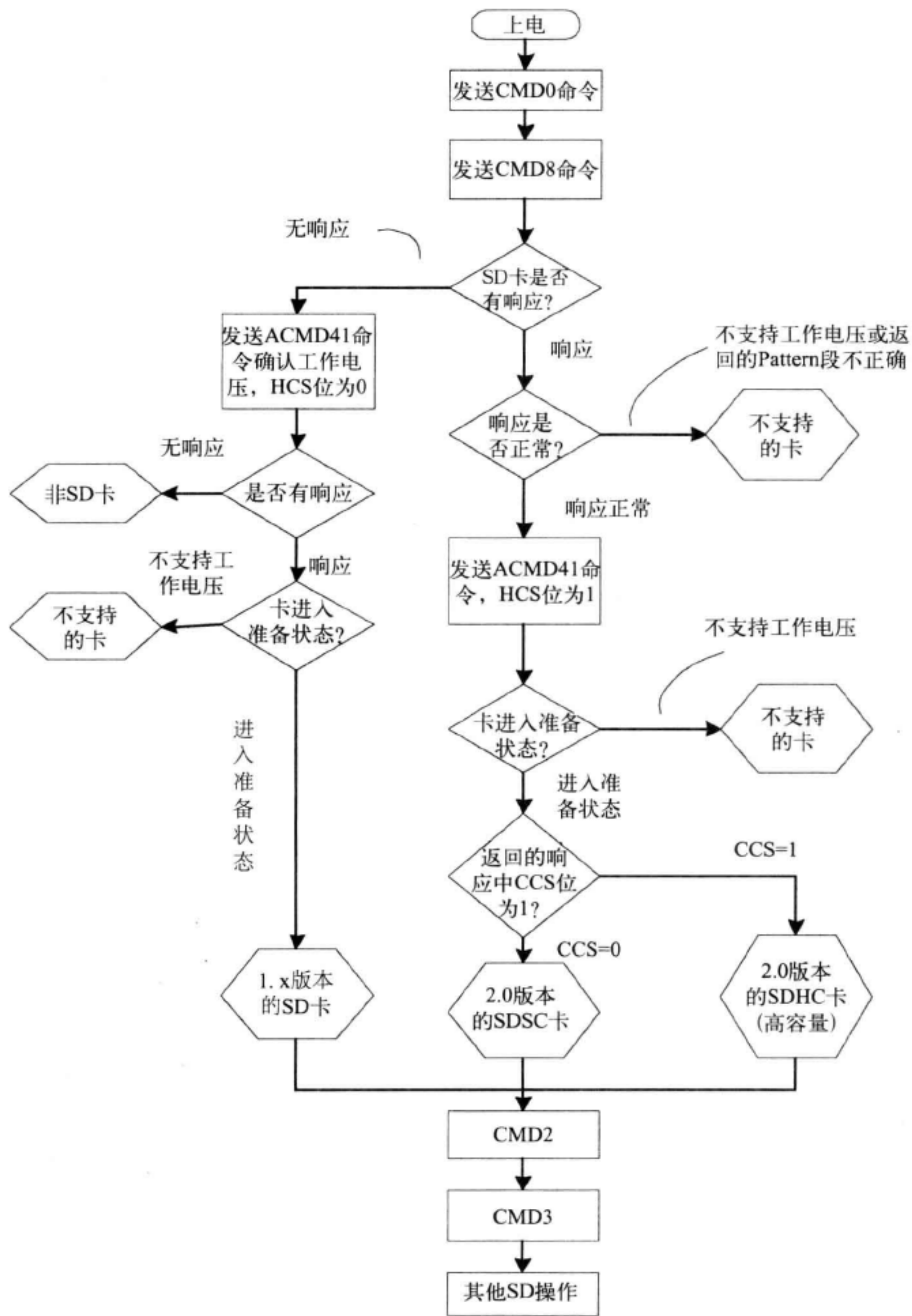


图 17-8 SD 卡上电识别流程

表 17-4 CMD8 命令的响应格式 R7

位	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
宽度 (位)	1	1	6	20	4	8	7	1
数值	'0'	'0'	'001000'	'00000h'	x	x	x	'1'
描述	起始位	传输位	命令索引	保留位	电压值	检测响应	CRC7	结束位

在驱动程序中调用了 CmdResp7Error() 来接收卡的响应，若卡对 CMD8 无响应，则说明接入的是 1.x 版本的 SD 卡或 MMC 卡，有响应则为 2.0 版本的 SD 卡。

(2) ACMD41 命令 (见表 17-5)

表 17-5 ACMD41 命令格式

ACMD 索引	类型	参 数	响应模式	缩 写	命令描述
ACMD41	bcr	[31] 保留位 [30]HCS (OCR[30]) [29:24] 保留位 [23:0] V _{DD} 电压窗口 (OCR[23:0])	R3	SD_SEND_OP_COND	发送主机的容量支持信息 (HCS) 到被访问的卡，并利用 CMD 线询问其工作状态寄存器 (OCR) 中的内容。当卡接收 SEND_IF_COND 的命令时，HCS 位是有效的，而保留位应设置为 '0'。CCS 位将被设置成 OCR[30] 中的内容。

协议要求在使用 ACDM41 之前必须先发送 CMD8 命令。它是用来进一步检查 SDIO 是否支持卡的工作电压的，另外还可以通过其命令参数中的 HCS 位来区分是 SDHC (高容量) 卡还是 SDSC 卡。它的命令响应为 R3，见表 17-6。

把 ACMD41 命令中的命令参数 HCS 位置“1”，然后循环地把这个命令发送出去，得到 SD 卡对 ACMD41 命令的响应。命令响应中包含了 SD 卡的 OCR 寄存器内容，这个寄存器保存了 SD 卡可支持的工作电压、是 SDHC 卡还是 SDSC 卡的信息，我们使用 CmdResp3Error() 函数来接收响应，读取这些信息。

另外，因为 ACMD41 命令属于 ACMD 命令，所以在发送 ACMD 命令前都要先发送 CMD55。

表 17-6 为 ACMD41 命令的响应 R3，返回的第 8 ~ 39 位是 OCR 寄存器的内容。

表 17-6 ACMD41 命令响应 R3

位	47	46	[45:40]	[39:8]	[7:1]	0
宽度 (位)	1	1	6	32	7	1
数值	'0'	'0'	'111111'	x	'1111111'	'1'
描述	起始位	传输位	保留	OCR 寄存器	保留	结束位

OCR 寄存器的格式如表 17-7,OCR 的第 15 ~ 23 位分别对应着一个电压值，表示可支持电压。若返回的命令响应中 OCR 的这些位为 1，则表示这个 SD 卡支持该位对应的电压值。第 30 位即为 CCS 位，若响应中这一位为 1 则表示这个 SD 卡为 SDHC 卡 (高容量)，否则为 SDSC 卡。

表 17-7 OCR 寄存器的内容

OCR 位	OCR 段定义
0-3	保留
4	保留
5	保留
6	保留
7	为低电压范围保留
8	保留
9	保留
10	保留
11	保留
12	保留
13	保留
14	保留
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-29	保留
30	卡容量状态 (CCS) 1
31	卡供电状态位 (busy) 2

17.3.6 卡的初始化流程

SD 卡上电确认成功、识别好卡的类别后，进入 SD_InitializeCards() 函数，见代码清单 17-5。

代码清单 17-5 SD_InitializeCards() 函数

```
1. /*
2.  * 函数名：SD_InitializeCards
3.  * 描述   ：初始化所有的卡或者单个卡进入就绪状态
4.  * 输入   ：无
5.  * 输出   ：-SD_Error SD卡错误代码
6.  *          成功时则为 SD_OK
7.  * 调用   ：在 SD_Init() 调用，在调用 power_on() 上电、卡识别完毕后，调用此函数进行卡初始化
8.  */
9. SD_Error SD_InitializeCards(void)
10. {
```

```

11.  SD_Error errorstatus = SD_OK;
12.  uint16_t rca = 0x01;
13.
14.  if (SDIO_GetPowerState() == SDIO_PowerState_OFF)
15.  {
16.      errorstatus = SD_REQUEST_NOT_APPLICABLE;
17.      return(errorstatus);
18.  }
19.
20.  if (SDIO_SECURE_DIGITAL_IO_CARD != CardType) // 判断卡的类型
21.  {
22.      /*!< Send CMD2 ALL_SEND_CID */
23.      SDIO_CmdInitStructure.SDIO_Argument = 0x0;
24.      SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ALL_SEND_CID;
25.      SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
26.      SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
27.      SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
28.      SDIO_SendCommand(&SDIO_CmdInitStructure);
29.
30.      errorstatus = CmdResp2Error();
31.
32.      if (SD_OK != errorstatus)
33.      {
34.          return(errorstatus);
35.      }
36.
37.      CID_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
38.      CID_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
39.      CID_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
40.      CID_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
41.  }
42.
43.      /* 下面开始 SD 卡初始化流程 */
44.  if ((SDIO_STD_CAPACITY_SD_CARD_V1_1 == CardType) || (SDIO_STD_CAPACITY_SD_CARD_
V2_0 == CardType) || (SDIO_SECURE_DIGITAL_IO_COMBO_CARD == CardType)
45.      || (SDIO_HIGH_CAPACITY_SD_CARD == CardType)) // 使用的是 2.0 的卡
46.
47.  {
48.      /*!< Send CMD3 SET_REL_ADDR with argument 0 */
49.      /*!< SD Card publishes its RCA. */
50.      SDIO_CmdInitStructure.SDIO_Argument = 0x00;
51.      SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR; //CMD3
52.      SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //R6
53.      SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
54.      SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
55.      SDIO_SendCommand(&SDIO_CmdInitStructure);
56.
57.      errorstatus = CmdResp6Error(SD_CMD_SET_REL_ADDR, &rca);
58. // 把接收到的卡相对地址存起来
59.
60.      if (SD_OK != errorstatus)
61.      {
62.          return(errorstatus);
63.      }
64.  }

```

```
65.
66.  if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
67.  {
68.      RCA = rca;
69.
70.      /*!< Send CMD9 SEND_CSD with argument as card's RCA */
71.      SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16);
72.      SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
73.      SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
74.      SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
75.      SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
76.      SDIO_SendCommand(&SDIO_CmdInitStructure);
77.
78.      errorstatus = CmdResp2Error();
79.
80.      if (SD_OK != errorstatus)
81.      {
82.          return(errorstatus);
83.      }
84.
85.      CSD_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
86.      CSD_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
87.      CSD_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
88.      CSD_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
89.  }
90.
91.  errorstatus = SD_OK; /*!< All cards get intialized */
92.
93.  return(errorstatus);
94.}
```

这个函数向卡发送了 CMD2 和 CMD3 命令。

(1) CMD2 命令

CMD2 命令是要求卡返回它的 CID 寄存器的内容，CID 寄存器的内容包括厂商 ID、卡的名称、生产日期等信息。CMD2 命令的响应格式为 R2，见表 17-8。

表 17-8 CMD2 命令响应格式 R2

位	135	134	[133:128]	[127:1]	0
宽度（位）	1	1	6	127	1
数值	‘0’	‘0’	‘111111’	x	‘1’
描述	起始位	传输位	保留	CID 或 CSD 寄存器，包括内部 CRC7 码	结束位

命令格式是 136 位的，属于长响应。硬件处理后，这些响应被 STM32 接收后分成几段保存在它的 SDIO_RESPx 寄存器上，由软件接收这些寄存器的信息，共有 128 位。读取的时候通过库函数 SDIO_GetResponse() 中的不同参数来获取 CID 中不同数据段的数据，如 SDIO_RESP1 接收响应的 96 ~ 127 位、SDIO_RESP2 则接收响应的 95 ~ 64 位。见表 17-9，这是 STM32 寄存器与响应中对应的数据位描述。

表 17-9 SDIO_RESPx 寄存器

寄 存 器	短 响 应	长 响 应
SDIO_RESP1	卡状态 [31:0]	卡状态 [127:96]
SDIO_RESP2	不用	卡状态 [95:64]
SDIO_RESP3	不用	卡状态 [63:32]
SDIO_RESP4	不用	卡状态 [31:1]

(2) CMD3 命令

CMD3 命令是要求卡向主机发送卡的相对地址。在连接多个卡的时候，主机要求接口上的卡重新发一个相对地址，这个地址与卡的实际 ID 不一样。比如接口上接了 5 个卡，这 5 个卡的相对地址就分别为 1、2、3、4、5。以后主机 SDIO 对这几个卡寻址就直接使用相对地址。相对地址比卡的 ID 要短得多，所以这个相对地址的作用就是为了寻址更加简单。

分析到这里大家应该对 SDIO 的命令发送和响应比较清楚了。接下来我们回到 SD_Init() 函数，在 SD_InitializeCards() 之后的 SD_GetCardInfo()、SD_SelectDeselect() 和 SD_EnableWideBusOperation() 的具体实现就不再详细分析了，实际就是发送相应的命令，获取相应的卡信息，控制卡的模式。

17.3.7 对 SD 卡进行读写

接下来分析 main 函数中的 SD_MultiBlockTest() 多块数据读写测试函数，让大家了解 SDIO 是怎样传输数据的。见代码清单 17-6。

代码清单 17-6 SD_MultiBlockTest() 函数

```
1. /*
2.  * 函数名：SD_MultiBlockTest
3.  * 描述   ：   多数据块读写测试
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7. void SD_MultiBlockTest(void)
8. {
9.     /*----- Multiple Block Read/Write -----*/
10.    /* Fill the buffer to send */
11.    Fill-Buffer (Buffer_MultiBlock_Tx, MULTI_BUFFER_SIZE, 0x0);
12.
13.    if (Status == SD_OK)
14.    {
15.        /* Write multiple block of many bytes on address 0 */
16.        Status = SD_WriteMultiBlocks(Buffer_MultiBlock_Tx, 0x00, BLOCK_SIZE, NUMBER_
            OF_BLOCKS);
17.        /* Check if the Transfer is finished */
18.        Status = SD_WaitWriteOperation();
19.        while(SD_GetStatus() != SD_TRANSFER_OK);
20.    }
21.
```

```

22.  if (Status == SD_OK)
23.  {
24.      /* Read block of many bytes from address 0 */
25.      Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, BLOCK_SIZE, NUMBER_OF_
        BLOCKS);
26.      /* Check if the Transfer is finished */
27.      Status = SD_WaitReadOperation();
28.      while (SD_GetStatus() != SD_TRANSFER_OK);
29.  }
30.
31.  /* Check the correctness of written data */
32.  if (Status == SD_OK)
33.  {
34.      TransferStatus2 = Buffercmp(Buffer_MultiBlock_Tx, Buffer_MultiBlock_Rx, MULTI_
        BUFFER_SIZE);
35.  }
36.
37.  if (TransferStatus2 == PASSED)
38.      printf("\r\n 多块读写测试成功!  ");
39.
40.  else
41.      printf("\r\n 多块读写测试失败!  ");
42.
43.}

```

这个函数只是一个用于测试代码是否正常的函数，流程如下：

1) 第 11 行，它首先调用了 Fill_Buffer() 填充数组。

2) 第 16 行，调用 SD_WriteMultiBlocks() 函数，本函数用于向 SD 卡写入多个数据块，例程中把填充好的数组发送到 SD 卡。

3) 第 18 ~ 19 行，在调用了上面的 SD_WriteMultiBlocks() 这一类写操作的函数后，一定要调用函数 SD_WaitWriteOperation() 和 SD_GetStatus() 来确保数据写入已经结束再进行其他操作。其中 SD_WaitWriteOperation() 用来等待 DMA 把缓冲的数据传输到 SDIO 的 FIFO。而 SD_GetStatus() 则是用于等待卡与 SDIO 之间传输数据完毕。

4) 第 25 行，调用 SD_ReadMultiBlocks() 函数，用于从 SD 卡读取多个数据块，例程中把读取出的数据存放在接收数组中。

5) 第 27 ~ 28 行，同样，调用了上面的 SD_ReadMultiBlocks() 读操作函数后，调用 SD_WaitReadOperation() 和 SD_GetStatus() 来等待数据读取完毕。

6) 第 34 行，调用 Buffercmp() 函数对写数组和读数组的数据进行比较，若完全相同，表明对 SD 卡的读写正常。

多块数据写入

下面以 SD_WriteMultiBlocks() 这个多数据块写入函数为例，分析本驱动如何读写 SD 卡，其他类似函数 SD_ReadBlock()、SD_ReadMultiBlocks()、SD_WriteBlock() 在编写方法上是一样的。SD_WriteMultiBlocks() 函数的具体代码见代码清单 17-7。

代码清单 17-7 SD_WriteMultiBlocks() 函数

```

1. /*
2.  * 函数名: SD_WriteMultiBlocks
3.  * 描述  : 从输入的起始地址开始, 向卡写入多个数据块,
4.            只能在 DMA 模式下使用这个函数
5.            注意: 调用这个函数后一定要调用
6.                  SD_WaitWriteOperation() 来等待 DMA 传输结束
7.                  和 SD_GetStatus() 检测卡与 SDIO 的 FIFO 间是否已经完成传输
8.  * 输入  :
9.  * @param WriteAddr: Address from where data are to be read.
10. * @param writebuff: pointer to the buffer that contain the data to be transferred.
11. * @param BlockSize: the SD card Data block size. The Block size should be 512.
12. * @param NumberOfBlocks: number of blocks to be written.
13. * 输出  : SD 错误类型
14. */
15. SD_Error SD_WriteMultiBlocks(uint8_t *writebuff, uint32_t WriteAddr, uint16_t
    BlockSize, uint32_t NumberOfBlocks)
16. {
17.     SD_Error errorstatus = SD_OK;
18.     __IO uint32_t count = 0;
19.
20.     TransferError = SD_OK;
21.     TransferEnd = 0;
22.     StopCondition = 1;
23.
24.     SDIO->DCTRL = 0x0;
25.
26.     if (CardType == SDIO_HIGH_CAPACITY_SD_CARD)
27.     {
28.         BlockSize = 512;
29.         WriteAddr /= 512;
30.     }
31.
32.     /******add, 没有这一段容易卡死在 DMA 检测中 *****/
33.     /*!< Set Block Size for Card, CMD16, 若是 SDSC 卡, 可以用来设置块大小, 若是 SDHC 卡, 块
        大小为 512 字节, 不受 CMD16 影响 */
34.     SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
35.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
36.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //R1
37.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
38.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
39.     SDIO_SendCommand(&SDIO_CmdInitStructure);
40.
41.     errorstatus = CmdResplError(SD_CMD_SET_BLOCKLEN);
42.
43.     if (SD_OK != errorstatus)
44.     {
45.         return(errorstatus);
46.     }
47.     /*******/
48.
49.     /*!< To improve performance */
50.     SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) (RCA << 16);
51.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD; // cmd55
52.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
53.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;

```



```

54.  SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
55.  SDIO_SendCommand(&SDIO_CmdInitStructure);
56.
57.
58.  errorstatus = CmdResplError(SD_CMD_APP_CMD);
59.
60.  if (errorstatus != SD_OK)
61.  {
62.      return(errorstatus);
63.  }
64.  /*!< To improve performance */
    // pre-erased, 在多块写入时可发送此命令进行预擦除
65.  SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)NumberOfBlocks;
    // 参数为将要写入的块数目
66.  SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCK_COUNT; //ACMD23
67.  SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
68.  SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
69.  SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
70.  SDIO_SendCommand(&SDIO_CmdInitStructure);
71.
72.  errorstatus = CmdResplError(SD_CMD_SET_BLOCK_COUNT);
73.
74.  if (errorstatus != SD_OK)
75.  {
76.      return(errorstatus);
77.  }
78.
79.
80.  /*!< Send CMD25 WRITE_MULT_BLOCK with argument data address */
81.  SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)WriteAddr;
82.  SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_WRITE_MULT_BLOCK;
83.  SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
84.  SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
85.  SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
86.  SDIO_SendCommand(&SDIO_CmdInitStructure);
87.
88.  errorstatus = CmdResplError(SD_CMD_WRITE_MULT_BLOCK);
89.
90.  if (SD_OK != errorstatus)
91.  {
92.      return(errorstatus);
93.  }
94.
95.  SDIO_DataInitStructure.SDIO_DataTimeOut = SD_DATATIMEOUT;
96.  SDIO_DataInitStructure.SDIO_DataLength = NumberOfBlocks * BlockSize;
97.  SDIO_DataInitStructure.SDIO_DataBlockSize = (uint32_t) 9 << 4;
98.  SDIO_DataInitStructure.SDIO_TransferDir = SDIO_TransferDir_ToCard;
99.  SDIO_DataInitStructure.SDIO_TransferMode = SDIO_TransferMode_Block;
100.      SDIO_DataInitStructure.SDIO_DPSM = SDIO_DPSM_Enable;
101.      SDIO_DataConfig(&SDIO_DataInitStructure);
102.
103.      SDIO_ITConfig(SDIO_IT_DATAEND, ENABLE);
104.      SDIO_DMAMCmd(ENABLE);
105.      SD_DMA_TxConfig((uint32_t *)writebuff, (NumberOfBlocks * BlockSize));
106.
107.      return(errorstatus);
108.  }

```

本函数为了实现写操作，在发送正式的多块写入命令 CMD25 前发送了“预写入”命令 ACMD23。预写入即预先擦除后面将要写入数据的空间，这样有利于提高写入的速度。

SD_WriteMultiBlocks() 函数中的第 95 ~ 105 行，对将要发送的数据进行结构体填充。其成员包括超时时间长度、数据大小（单位为字节，但必须是 512 的整数倍）、块大小（STM32 的块大小可以自由设置，但 SD 协议规定块大小为 512 字节）、数据传输方向、传输模式、是否使用 DPSM 状态机，填充完成后调用库函数 SDIO_DataConfig() 把这些控制信息写入寄存器。

配置好数据的控制信息后，在第 103 行，调用库函数 SDIO_ITConfig() 开启了数据传输结束中断，SDIO 的数据传输结束中断就是在这个时候开启的。数据传输结束时，就进入到 stm32f10x_it.c 文件的中断服务函数 SDIO_IRQHandler() 中处理，中断服务函数主要就是负责清中断。

第 104 ~ 105 行调用 SDIO_DMACmd() 允许发送 DMA 请求，调用用户函数 SD_DMA_TxConfig() 配置 DMA 传输的控制信息。

17.3.8 原版官方驱动例程的 bug

最后讲解官方原版驱动中的一个 bug。在 SD_WriteMultiBlocks() 的第 32 ~ 47 行是我们在原版官方例程中添加的代码，即在正式的写操作前，补充发送了 CMD16 命令，该命令用于设置读写 SD 卡的块大小，见代码清单 17-8。

代码清单 17-8 SD_WriteMultiBlocks() 函数部分代码

```

1.  /*** 本段代码摘自 SD_WriteBlock() 的 32 ~ 47 行 *****/
2.  /*****add, 没有这一段容易卡死在 DMA 检测中 *****/
3.  /*CMD16, 若是 SDSC 卡, 可以用来设置块大小, 若是 SDHC 卡, 块大小为 512 字节, 不受 CMD16 影响 */
4.  SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
5.  SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
6.  SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;    //R1
7.  SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
8.  SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
9.  SDIO_SendCommand(&SDIO_CmdInitStructure);
10.
11. errorstatus = CmdResplError(SD_CMD_SET_BLOCKLEN);
12.
13. if (SD_OK != errorstatus)
14. {
15.     return(errorstatus);
16. }
```

缺少这个命令会导致程序运行时卡死在循环检测 DMA 传输结束的代码中，大部分工程师在直接移植 ST 官方例程时，用 3.5 版库函数和这个 4.5 版 SDIO 驱动例程移植失败，就是缺少了这段用 CMD16 设置块大小的代码。在原版例程的 SD_ReadBlock()、SD_ReadMultiBlocks()、SD_WriteBlock() 和 SD_WriteMultiBlocks() 这一类读写操作的函数中，也都缺少这段关键代码，添加发送 CMD16 命令之后就没有问题了。

本实验于此讲解完毕，由于代码庞大，想要更深入地了解还是要在具体的工程文件中，请利用这个例程中的注释和附带资料 SD2.0 协议——Simplified_Physical_Layer_Spec.pdf 好好研究一番！

17.3.9 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上串口线 (两头都是母的交叉线)，插上 MicroSD 卡 (我们使用的是 1GB，经测试本驱动也适用于 2GB 以上的卡 (SDHC 卡))，打开超级终端，配置超级终端为“115200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如图 17-9 所示信息。

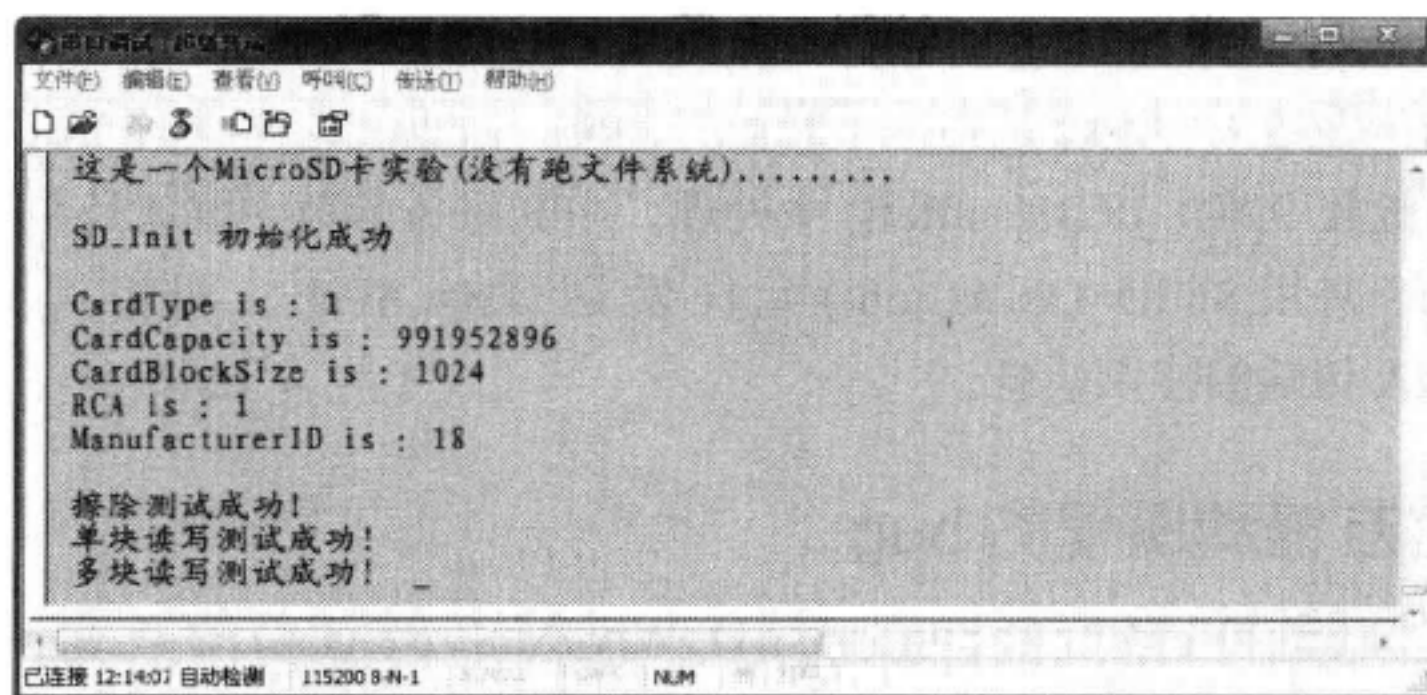


图 17-9 SD 卡实验调试信息

注意 这个例程没有跑文件系统，而是直接就去读写卡的 block，这样的话就会破坏卡的分区。在实验完成之后，读者再把卡插到计算机上时，计算机会提示你要重新格式化卡，这是正常现象，并不是本实验把你的卡弄坏了。重新格式化后，卡就能如平常一样使用。但卡里原来的资料将会被销毁，所以如果卡原来有资料请先把数据备份了再进行测试。但跑文件系统时就不会出现这种问题，想知道原因请看第 18 章，有关文件系统的操作将在此章讲解。



第 18 章

文件系统之 FATFS_R0.09

18.1 什么是文件系统

即使读者可能不了解文件系统，读者也一定对“文件”这个概念十分熟悉。数据在 PC 上是以文件的形式存储在磁盘中，这些数据的形式一般为 ASCII 码或二进制形式。在第 17 章我们已经写好了 SD 卡的驱动，也就是说已经可以对 SD 卡进行读写数据。如需要记录本书的书名，可以首先把这些文字转化成 ASCII 码，存储在数组中，然后调用块写入函数 `SD_WriteBlock()`，把数组内容写入 SD 卡的某个地址，在需要的时候从该地址把数据读取出来，再对读出来的数据以 ASCII 码的格式进行解读。

但是，这样直接存储数据会带来极大的不便，如难以记录有效数据的位置，难以确定存储介质的剩余空间，以及应以何种格式来解读数据。就如同一个巨大的图书馆无人管理，杂乱无章地存放着各种书籍，难以查找所需的文档。想象一下图书馆的采购人员购书后，把书籍往馆内一扔，拍拍屁股走人，当有人来借阅某本书的时候，就不得不一本本地查找。这样直接存储数据的方式对于小容量的存储介质如 EEPROM 还可以接受，但对于 SD 卡这类大容量设备，我们需要一种高效的方式来管理它的存储内容。

这些管理方式即为文件系统，它是为了存储和管理数据，而在存储介质上建立的一种组织结构，这些结构包括操作系统引导区、目录和文件。常见的 Windows 下的文件系统格式包括 FAT32、NTFS、exFAT。在使用文件系统前，要先对存储介质进行格式化。格式化之后，存储介质会创建一个文件分配表和目录。这样，文件系统就可以记录数据存放的物理地址和剩余空间。

使用文件系统时，数据都以文件的形式存储。写入新文件时，先在目录中创建一个文件索引，它指示了文件存放的物理地址，再把数据存储到该地址中。当需要读取数据时，可以从目录中找到该文件的索引，进而在相应的地址中读取数据。具体还涉及逻辑地址、簇大小、不连续存储等一系列辅助结构或处理过程。

文件系统的存在使我们在存取数据时，不再是简单地向某物理地址直接读写，而是要遵循它的读写格式。如经过逻辑转换，一个完整的文件可能被分成多段并存储到不连续的物理地址，以及使用目录或链表的方式来获知下一段的位置。

第 17 章的 SD 卡驱动只完成了向物理地址写入数据的工作，而根据文件系统格式的逻辑转换

部分则需要额外的代码来完成。实质上,这个逻辑转换部分可以理解为当我们需要写入一段数据时,由它来求解向什么物理地址写入数据、以什么格式写入及写入一些原始数据以外的信息(如目录)。这个逻辑转换部分代码我们也习惯称之为文件系统。

18.2 FATFS 文件系统简介

上面提到的逻辑转换部分代码(文件系统)即为本章的要点,文件系统庞大而复杂,它需要根据应用的文件系统格式而编写,而且一般与驱动层分离开来,以方便移植,所以工程应用中一般是移植现成的文件系统源码。

FATFS 是面向小型嵌入式系统的一种通用的 FAT 文件系统。它完全是由 ANSI C 语言编写并且完全独立于底层的 I/O 介质。因此它可以很容易地不加修改地移植到其他处理器当中,如 8051、PIC、AVR、SH、Z80、H8、ARM 等。FATFS 支持 FAT12、FAT16、FAT32 等格式,所以我们利用前面写好的 SDIO 驱动,把 FATFS 文件系统代码移植到工程之中,就可以利用文件系统的各种函数,对已格式化的 SD 卡的文件进行读写了。

FATFS 文件系统的源码可以从 FATFS 官网下载:

http://elm-chan.org/fsw/ff/00index_e.html

18.2.1 FATFS 的目录结构

在移植 FATFS 文件系统之前,我们先要到 FAT 的官网获取源码,版本为 R0.09。解压之后可看到里面有 doc 和 src 这两个文件夹。doc 文件夹里面是一些使用文档,src 里面是文件系统的源码。见图 18-1。

18.2.2 FATFS 帮助文档

打开 doc 文件夹,可看到如图 18-2 所示文件目录。

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

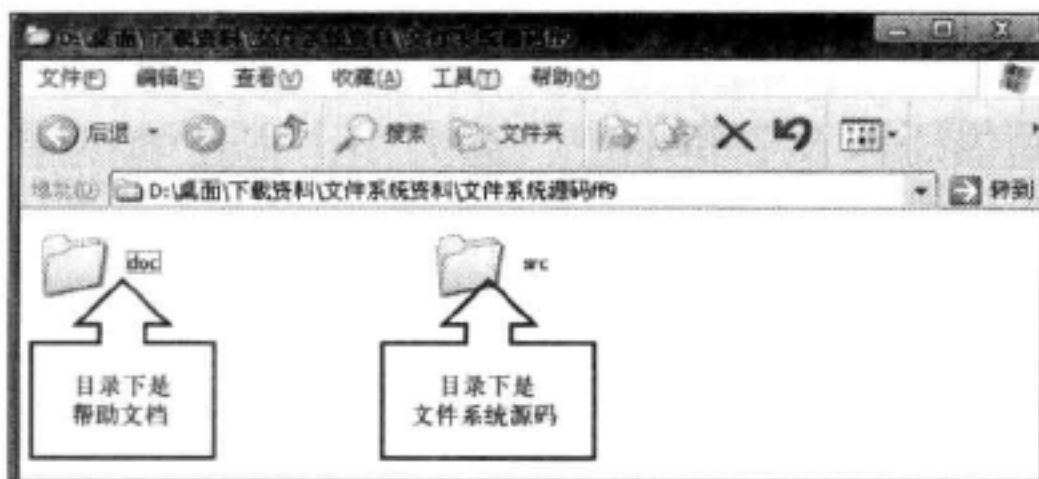


图 18-1 FATFS 源码结构



图 18-2 doc 文件夹下的内容

18.2.3 FATFS 源码

打开 src 文件夹，可看到如图 18-3 所示目录。

option 文件夹下是一些可选的外部 C 文件，包含了多语言支持需要用到的文件和转换函数。

在新的 R0.09 版源码中没有了 diskio.c 文件，但有 diskio.h。在 diskio.h 源码中有一些关于底层硬件接口的函数声明，我们可以从旧版本的源码中把 diskio.c 的函数定义搬过来，自己新建一个 diskio.c 文件。diskio.c 文件是移植中最关键的文件，它为文件系统提供了最底层访问 SD 卡的方法，即调用了 SD 驱动函数。



图 18-3 src 文件夹下的源码文件

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

src 文件夹下的源码文件功能简介如下：

- ❑ integer.h：文件中包含了一些数值类型定义。
- ❑ diskio.c：包含底层存储介质的操作函数，这些函数需要用户自己实现，主要添加底层驱动函数。
- ❑ ff.c：独立于底层介质操作文件的函数，利用这些函数实现文件的读写。
- ❑ cc936.c：本文件在 option 目录下，是简体中文支持所需要添加的文件，包含了简体中文的 GBK 和转换函数。
- ❑ ffconf.h：这个头文件包含了对文件系统的各种配置，类似 stm32f10x_conf.h 在 STM32 库中的功能。如需要支持简体中文，需要把 ffconf.h 中的 _CODE_PAGE 的宏改成 936，并把上面的 cc936.c 文件加入到工程之中。

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

阅读文件系统源码 ff.c 文件需要一定的功底，建议读者先阅读 FAT32 的文件格式，再去分析 ff.c 文件。若仅为使用文件系统，则只需要理解 integer.h 及 diskio.c 文件，并会调用 ff.c 文件中的函数就可以了。本章主要讲解如何把 FATFS 文件系统移植到配套开发板上。

18.3 移植 FATFS 文件系统实验

18.3.1 实验描述及工程文件清单

1. 实验描述

MicroSD 卡文件系统 FATFS_R0.09 测试实验。在 MicroSD 卡里面创建一个 DEMO.TXT 文本文件，并在文件里面写入字符串“感谢您选用 野火 STM32 开发板！”，然后通过串口将这些内容

打印在计算机的超级终端上。支持简体中文和长文件名，采用 SDIO 的 4bit+DMA 模式。关于文件系统的使用将在紧接下来的 MP3 例程中进行演示。

2. 硬件连接

与 SDIO 实验相同。

- ☐ PC8-SDIO-D0 : DATA0
- ☐ PC9-SDIO-D1 : DATA1
- ☐ PC10-SDIO-D2 : DATA2
- ☐ PC11-SDIO-D3 : CD/DAT3
- ☐ PC12-SDIO-CLK : CLK
- ☐ PD2-SDIO-CMD : CMD

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_sdio.c
- ☐ FWlib/stm32f10x_dma.c
- ☐ FWlib/misc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/usart.c
- ☐ USER/sdio_sdcard.c

5. 文件系统源文件

使用 FATFS_R0.09 版本源码：

- ☐ ff9/diskio.c
- ☐ ff9/ff.c
- ☐ ff9/cc936.c

MicroSD 卡硬件原理图见图 18-4。

本实验是在第 17 章的基础上讲解的，只有第 17 章的实验成功了，文件系统才能跑起来。有关卡的底层的初始化，不再详述，本章着重讲解文件系统的移植。

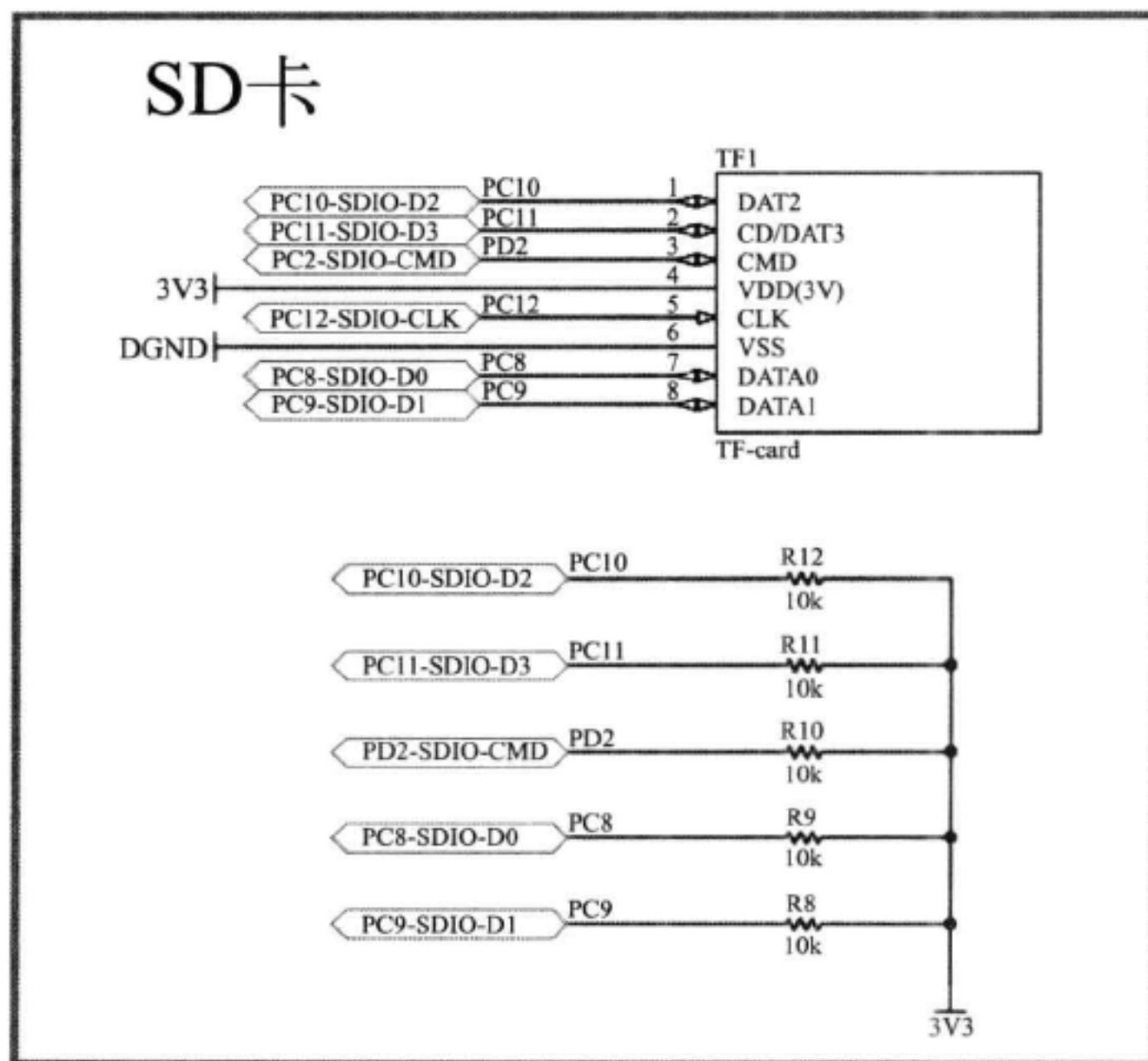


图 18-4 配套 STM32 开发板 MicroSD 卡硬件原理图

18.3.2 配置工程环境

首先要获取一个完全没有修改过的文件系统源码，添加到 SD 卡读写实验中，在该工程的基础上进行文件系统移植。我们在移植这个文件系统的过程中会尽量保持文件系统源码的纯净，尽量做到在修改最少量源码的情况下移植成功。

- 1) 由于本实验与 SD 卡读写实验使用的外设完全一样，首先复制一份 SD 读写实验的工程。
- 2) 直接将整个解压后的文件系统源码复制到工程目录下，见图 18-5。
- 3) 返回到 MDK 界面下，添加 ff9 项目组，并把 ff9\src\ 目录下的 ff.c、ff9\src\option 目录下的 cc936.c 和新建的 diskio.c 这三个文件添加到 ff9 项目组下，见图 18-6。

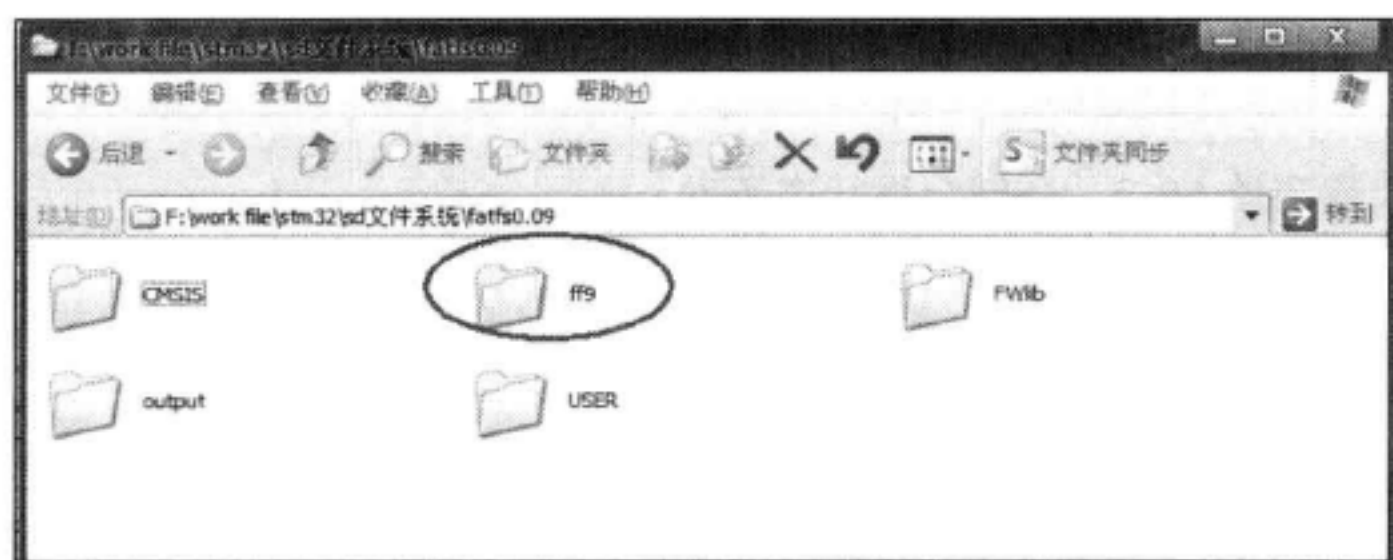


图 18-5 复制文件系统源码到工程目录

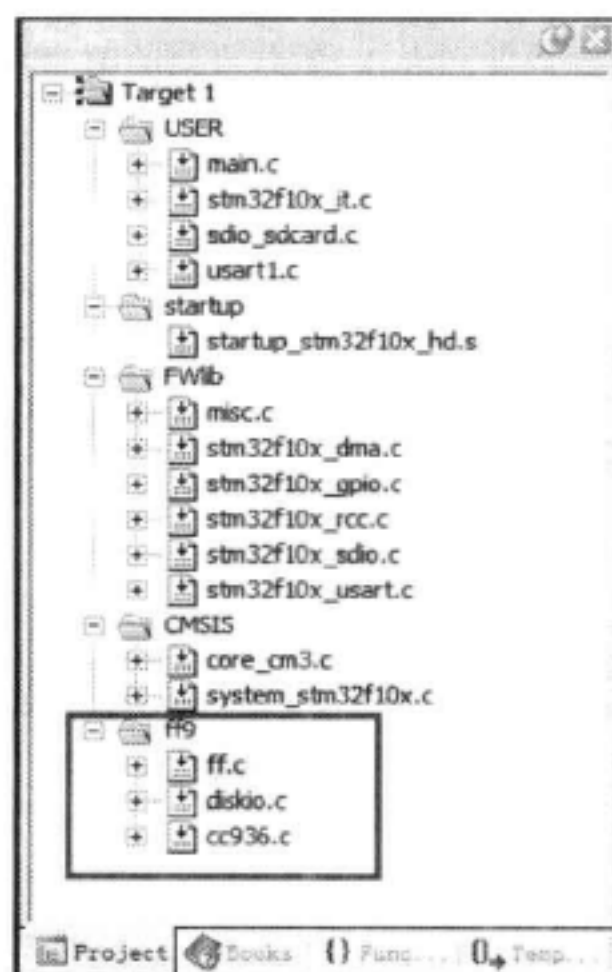
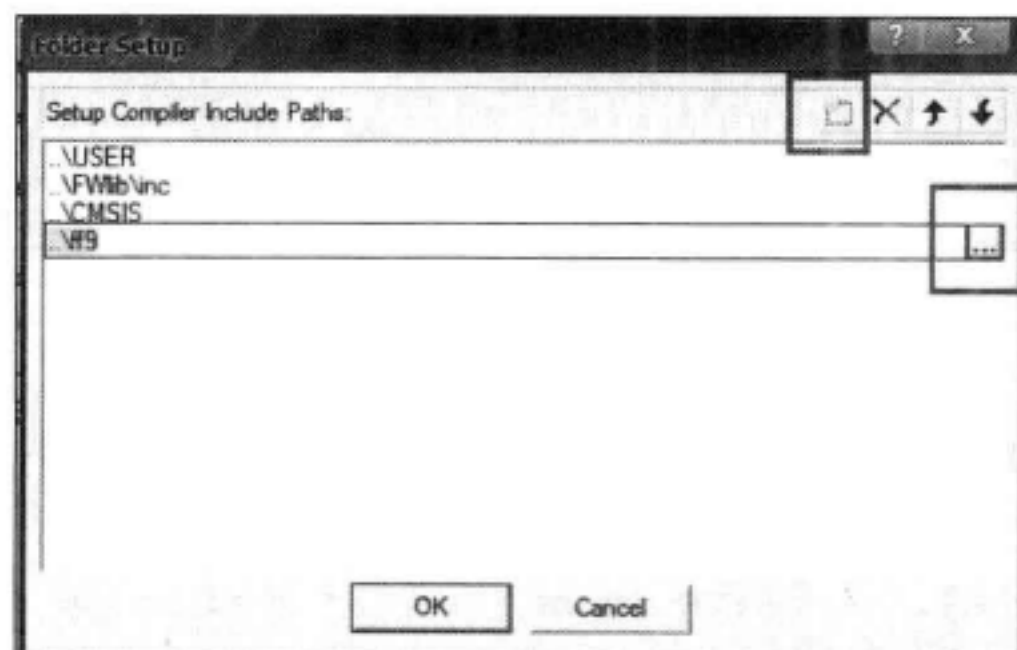


图 18-6 添加源文件到工程

- 4) 在 Target Options 的 C++ 编译器选项中添加文件包含路径，见图 18-7。



a)



b)

图 18-7 添加头文件搜索路径

接下来就可以直接编译，可以发现，新版本的文件系统源码与 STM32 的库文件没有冲突，这样我们就不用像使用旧版本文件系统那样修改重复定义的变量，省了不少事。如果你使用的还是旧版本的文件系统源码，可以参照之前的例程处理重复定义了变量，这里就不详述了。

18.3.3 为文件系统添加底层驱动

FATFS 文件系统与底层介质的驱动分离开来，对底层介质的操作都要交给用户去实现，它仅仅是提供了一个函数接口而已，函数为空，要用户添加代码。因此我们要把 diskio.c 中的函数接口与前面写的 SD 实验驱动连接起来。

根据 FATFS 帮助文档的说明，用户需要提供的几个函数的原型如下，它们在 diskio.c 中定义，见代码清单 18-1 至代码清单 18-5。

代码清单 18-1 存储介质初始化函数

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

代码清单 18-2 存储介质状态函数

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

代码清单 18-3 扇区读取函数

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

代码清单 18-4 扇区写入函数

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

代码清单 18-5 其他控制功能

```

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

```

这些函数都是操作底层介质的函数，都需要用户自己实现，然后 FATFS 的应用函数就可以调用这些函数来操作我们的卡了。

1. 初始化函数接口

以下是文件系统初始化函数接口，文件系统的一些函数会调用这个 disk_initialize() 接口来进行底层存储介质的初始化，所以我们在这里加入 SDIO 的 SD_Init() 初始化函数，见代码清单 18-6。

代码清单 18-6 disk_initialize() 接口

```

1. DSTATUS disk_initialize (
2.     BYTE drv          /* Physical drive number (0..) */
3. )
4. {
5.     SD_Error  Status;
6.     /* Supports only single drive */
7.     if (drv)
8.     {
9.         return STA_NOINIT;
10.    }
11. /*----- SD Init ----- */
12.    Status = SD_Init();
13.    if (Status!=SD_OK )
14.    {
15.        return STA_NOINIT;
16.    }
17.    else
18.    {
19.        return RES_OK;
20.    }
21.
22.}

```

这个初始化函数接口调用了 SDIO 的 SD_Init() 函数，返回初始化成功或失败的参数，当文件系统的上层函数调用 disk_initialize() 时，实质调用了 SD_Init() 函数对 SD 卡进行初始化。

在使用文件系统前，还要在 main 函数里调用 SDIO 驱动层的一个函数 NVIC_Configuration()，用来配置 SDIO 的中断优先级，其实也可以把这个函数放到 disk_initialize() 里。注意如果有多个中断要处理时，必须调整 SDIO 的优先级符合工程的应用，不然错了也不知道从哪里找。

2. 扇区读取函数

接下来是 disk_read() 函数，它是文件系统读取 SD 卡数据会调用的一个函数，所以我们在这

个函数中加入 SDIO 驱动的读取函数的接口。见代码清单 18-7。

代码清单 18-7 disk_read() 接口

```

1. DRESULT disk_read (
2.     BYTE drv,          /* Physical drive number (0..) */
3.     BYTE *buff,        /* Data buffer to store read data */
4.     DWORD sector,      /* Sector address (LBA) */
5.     BYTE count         /* Number of sectors to read (1..255) */
6. )
7. {
8.
9.     if (count > 1)
10.    {
11.        SD_ReadMultiBlocks(buff, sector*BLOCK_SIZE, BLOCK_SIZE, count);
12.
13.        /* Check if the Transfer is finished */
14.        SD_WaitReadOperation(); // 循环查询 DMA 传输是否结束
15.
16.        /* Wait until end of DMA transfer */
17.        while(SD_GetStatus() != SD_TRANSFER_OK);
18.
19.    }
20.    else
21.    {
22.
23.        SD_ReadBlock(buff, sector*BLOCK_SIZE, BLOCK_SIZE);
24.
25.        /* Check if the Transfer is finished */
26.        SD_WaitReadOperation(); // 循环查询 DMA 传输是否结束
27.
28.        /* Wait until end of DMA transfer */
29.        while(SD_GetStatus() != SD_TRANSFER_OK);
30.
31.    }
32.    return RES_OK;
33.}

```

这里分了两个部分，分为单块读取和多块数据读取。因为用一个多块读取函数 SD_ReadMultiBlocks() 比用多个单块读取函数 SD_ReadBlock() 要简便快捷得多，所以我们加入了一个判断函数来进行区分。由于文件系统的读写都是以块（512 字节）为单位的，所以只需要提供 512 字节或 512*N 字节的 SD 卡驱动（N 为块的数目）。

要强调的是，在调用了读取函数之后要调用 SD_WaitReadOperation() 查询 SDIO 是否发送完毕，以及使用语句 while (SD_GetStatus() != SD_TRANSFER_OK) 等待 DMA 传输是否结束，这是底层驱动结构决定的。

3. 扇区写入函数

扇区写入函数与扇区读取函数十分类似，也是根据写入的扇区数目是一个还是多个来分别调用不同的 SD 数据块写入函数。在调用了写入函数之后，也要等待 SDIO 发送完毕及 DMA 传输结束。其具体函数代码见代码清单 18-8。

代码清单 18-8 disk_write() 函数

```

1. DRESULT disk_write (
2.     BYTE drv,           /* Physical drive number (0..) */
3.     const BYTE *buff,   /* Data to be written */
4.     DWORD sector,       /* Sector address (LBA) */
5.     BYTE count           /* Number of sectors to write (1..255) */
6. )
7. {
8.
9.     if (count > 1)
10.    {
11.        SD_WriteMultiBlocks((uint8_t *)buff, sector*BLOCK_SIZE, BLOCK_
        SIZE, count);
12.
13.        /* Check if the Transfer is finished */
14.        SD_WaitWriteOperation();           // 等待 DMA 传输结束
15.        while(SD_GetStatus() != SD_TRANSFER_OK);
16.    // 等待 SDIO 到 SD 卡传输结束
17.    }
18.    else
19.    {
20.        SD_WriteBlock((uint8_t *)buff, sector*BLOCK_SIZE, BLOCK_SIZE);
21.
22.        /* Check if the Transfer is finished */
23.        SD_WaitWriteOperation();           // 等待 DMA 传输结束
24.        while(SD_GetStatus() != SD_TRANSFER_OK);
25.    // 等待 SDIO 到 SD 卡传输结束
26.    }
27.    return RES_OK;
28.}

```

4. 时间接口函数

在 diskio.c 文件的最后我们还得提供获取时间的函数，因为 ff.c 中调用了它，用于记录文件的创建、修改时间，而 FATFS 库又没有给出这个函数的原型，所以需要用户实现，否则会发生编译出错。对于这部分，函数体为空，提供无意义的返回值 0 即可，也可以为它加载 STM32 的 RTC 驱动。若加载 RTC 驱动，返回值需要按照如下格式组织数据：

- ❑ bit31:25 —— 从 1980 至今是多少年，范围是 (0 ~ 127)
- ❑ bit24:21 —— 月份，范围为 (1 ~ 12)
- ❑ bit20:16 —— 该月份中的第几日，范围为 (1 ~ 31)
- ❑ bit15:11 —— 时，范围为 (0 ~ 23)
- ❑ bit10:5 —— 分，范围为 (0 ~ 59)
- ❑ bit4:0 —— 秒 / 2，范围为 (0 ~ 29)

在本实验中没有添加 RTC 时间驱动，因此没有进行修改，见代码清单 18-9。

代码清单 18-9 空的 RTC 时间驱动

```

1. /*-----*/
2. /* Get current time */
3. /*-----*/

```

```

4. DWORD get_fattime(void)
5. {
6.
7.     return 0;
8.
9. }

```

18.3.4 添加简体中文和长文件名支持

添加完底层的驱动，实际上文件系统已经可以正常使用了，为了支持简体中文和长文件名，需要对文件系统配置。

优秀的工程一般都具备一个源文件配套一个头文件，再外加一个头文件，包含对工程进行裁剪、配置的条件编译宏。如 STM32 的库文件中就有 stm32f10x_conf.h 文件可对使用的库文件进行配置。FATFS_R0.09 文件系统也是一个很优秀的工程，我们可以在 ffconf.h 头文件中对文件系统进行裁剪。

在这个文件中，修改如下的宏，这是 ffconf.h 中的一部分，见代码清单 18-10。

代码清单 18-10 ffconf.h 文件中的宏

```

1. /*-----/
2. / Locale and Namespace Configurations
3. /-----*/
4.
5. #define _CODE_PAGE 936
6. #define _USE_LFN 2 /* 0 to 3 */
7. #define _MAX_LFN 255 /* Maximum LFN length to handle (12 to 255) */
8. #define _LFN_UNICODE 0 /* 0:ANSI/OEM or 1:Unicode */
9. #define _FS_RPATH 0 /* 0 to 2 */

```

- 1) 把 _CODE_PAGE 宏配置成简体中文的 code_page，936。
- 2) 把 _USE_LFN 宏配置成 2，表示开启长文件名支持。
- 3) 把 _MAX_LFN 宏配置成 255，表示可支持的长文件名的最大字节数。

这些配置参数可以从源文件中的注释或帮助文档中查找到，由于篇幅问题，我们省略了源文件中对于以上几个宏的注释。由于使用简体中文支持后，添加的字库会使代码变得非常庞大，下载代码的时候会花上较长的时间，建议读者在调试完成基本功能后，再添加简体中文支持。

按这样的配置就可以支持简体中文和长文件名，但在具体配置的时候要注意堆栈大小问题，使用长文件名与使用默认长度文件名不同。关于这部分在第 19 章会详细解释。在本章中我们先把文件系统跑起来，感受一下。

到这里我们算是把文件系统移植成功了，接下来的任务就是调用文件系统的函数来操作我们的卡，往 SD 卡里读写文件。

18.3.5 main 文件

实现好底层介质的操作函数之后，我们就可以回到应用层了，下面我们从 main 文件开始，见代码清单 18-11。

代码清单 18-11 FATFS 例程的 main 函数

```

1. /* Includes -----*/
2. #include "stm32f10x.h"
3. #include "sdio_sdcard.h"
4. #include "usart1.h"
5. #include "ff.h"
6.
7. int res;           // 读写文件的返回值
8. int a;
9.
10. FIL fsrc, fdst;    // 文件系统结构体, 包含文件指针等成员
11. FATFS fs;          // 记录文件系统盘符信息的结构体
12. UINT br, bw;       // File R/W count
13. BYTE buffer[512];  // file copy buffer
14. BYTE textFileBuffer[] = "感谢您选用 野火 STM32 开发板 ! ^_^ \r\n";
15.
16.
17. int main(void)
18. {
19.
20.
21.     /* USART1 config */
22.     USART1_Config();
23.
24.     /* Interrupt Config */
25.     NVIC_Configuration();
26.
27.     printf("\r\n 这是一个MicroSD卡文件系统实验 (FATFS R0.09)\n");
28.
29.     printf ( "\r\n disk_initialize starting.....\n " );
30.
31.     f_mount(0, &fs);
32.
33.     res = f_open(&fdst, "0:/Demo.TXT", FA_CREATE_NEW | FA_WRITE);
34.
35.     if ( res == FR_OK )
36.     {
37.         /* 将缓冲区的数据写到文件中 */
38.         res = f_write(&fdst, textFileBuffer, sizeof(textFileBuffer), &bw);
39.         printf( "\r\n 文件创建成功 \n" );
40.         /* 关闭文件 */
41.         f_close(&fdst);
42.     }
43.     else if ( res == FR_EXIST )
44.     {
45.         printf( "\r\n 文件已经存在 \n" );
46.     }
47.
48.     /*----- 将刚刚新建的文件里面的内容打印到超级终端 -----*/
49.     /* 以只读的方式打开刚刚创建的文件 */
50.     res = f_open(&fdst, "0:/Demo.TXT", FA_OPEN_EXISTING | FA_READ);
51.     /* 打开文件 */
52.     br = 1;
53.     a = 0;
54.     for (;;)
55.     {
56.         for ( a=0; a<512; a++ )      /* 清缓冲区 */
57.             buffer[a]=0;
58.
59.         res = f_read( &fdst, buffer, sizeof(buffer), &br );
60.         /* 将文件里面的内容读到缓冲区 */
61.         printf("\r\n %s ", buffer);

```

```

62.     if (res || br == 0) break;          /* 错误或者到了文件尾 */
63.
64. }
65.     f_close(&fdst);                    /* 关闭打开的文件 */
66.
67.
68. while (1)
69. {
70. }
71.
72. /***** (C) COPYRIGHT 2012 野火嵌入式开发工作室 **END OF FILE****/

```

首先因为 main 函数中要用到 ff.c 文件的函数接口，所以在 main.c 中将该文件对应的头文件 ff.h 包含进来。

1. 写入文件

定义了一系列文件系统需要用到的变量后，进入 main 函数，初始化了串口、NVIC，开始写文件流程，这个流程调用了一系列文件系统的函数。

1) 第 31 行，调用函数 f_mount() 创建一个工作区，它的另一个功能是调用了底层的 disk_initialize() 函数，进行 SDIO 接口的初始化，将我们的底层硬件初始化，这一步非常重要，如果不成功，接下来什么都干不了。

2) 第 33 行，调用 f_open() 函数在刚刚开辟的工作区的盘符 0 下打开一个名为 Demo.TXT 的文件，以创建新文件或写入的方式打开（参数“FA_CREATE_NEW | FA_WRITE”），如果文件不存在的话则创建这个文件。本函数还将 Demo.TXT 这个文件关联到 fsrc 这个结构指针，以后我们操作文件就是通过这个结构指针来完成的，可以把它理解为文件指针。

3) 第 35 行，对 f_open() 函数的返回值进行检查，若返回值 res 等于宏 FR_OK，表示创建新文件成功，若返回值 res 等于宏 FR_EXIST，表明文件已经存在。

4) 第 38 行，创建文件成功，调用 f_write() 将缓冲区的数组变量 textFileBuffer 的内容写到刚刚打开的 Demo.TXT 文件中。写完之后必须调用 f_close() 函数关闭已打开的文件，否则前面写入的数据无效，甚至可能导致其他错误。

2. 读取文件

从第 48 行开始，演示的是读文件流程，从刚刚写入的文件中读取数据。

1) 第 50 行，调用函数 f_open() 以只读的方式打开刚创建的文件。

2) 第 59 行，调用函数 f_read() 将文件的内容读到缓冲区 buffer 数组变量中，然后调用 printf() 将读取的数据打印到计算机超级终端。

3) 第 62 行，根据 f_read() 函数的返回值和调用它时使用的参数 br 来判断是否读取到了文件尾。

4) 第 65 行，调用 f_close() 关闭文件。当被打开的文件操作完成之后都要调用 f_close() 将它关闭，这如同一块动态分配的内存存在用完之后都要调用 free() 来将它释放。

这里涉及了 FATFS 文件系统库函数的操作，如果读者学习过 Linux 系统调用，操作这些函数将非常简单。如果没有学过，可以把本实验当作学习 Linux 文件操作的基础。在 FATFS 源码目录 doc 这个文件夹中提供了每个应用函数的用法，见图 18-8。

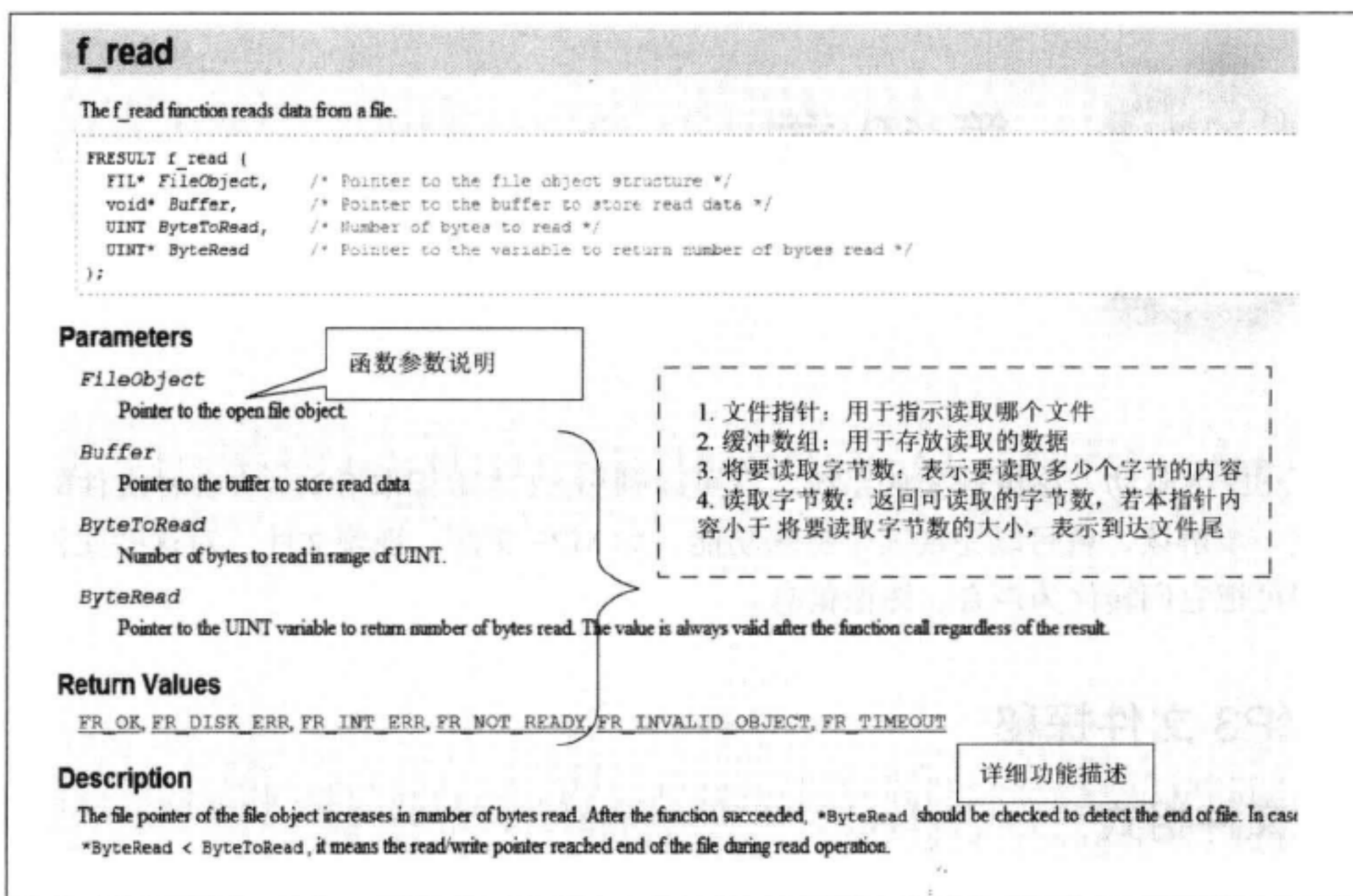


图 18-8 f_read 帮助文档说明

18.3.6 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上串口线（两头都是母的交叉线），插上已格式化成 FAT32 格式的 MicroSD 卡（我们用的是 1GB，4GB 的也已经测试通过，理论上 SDSC、SDHC 卡均可正常使用），打开超级终端，配置超级终端为“115200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如图 18-9 所示信息，完成后可以用 PC 查看 SD 卡内容，看看卡里是不是多了一个文件。

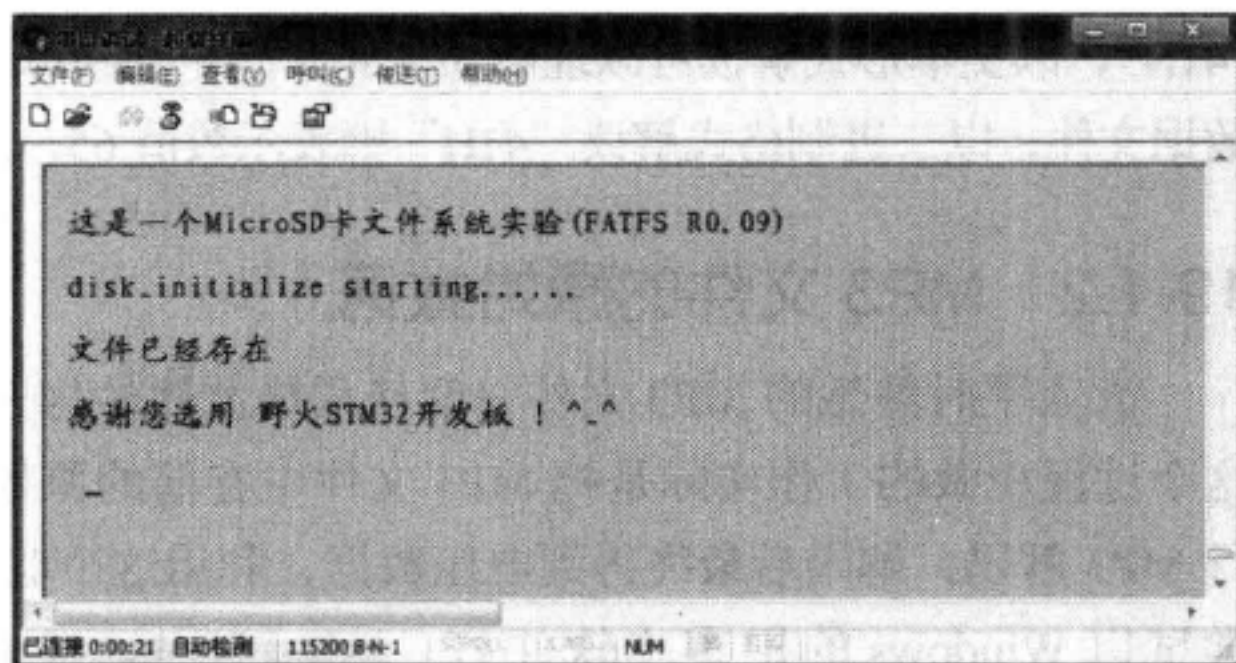


图 18-9 文件系统移植实验



第 19 章

MP3 播放器

具备了 SD 卡驱动，移植好文件系统，就可以利用 STM32 读取 SD 卡中各种文件的数据，对这些数据进一步解读，就可以完成某个特殊功能。如 MP3 文件、视频文件，对这些文件的数据进行解码，即可把它们转化为声音、图像信息。

19.1 MP3 文件探秘

19.1.1 文件格式

保存到存储介质的文件，有的是图像数据，有的是音频数据，还有的为程序数据等，为了正常地记录并有效地读取这些信息，人们建立了不同的标准。这些不同的标准呈现在用户眼中为不同的文件格式，在 Windows 系统下更直接地以后缀名作为区别。

因为文件信息是以 0、1 的形式存储的，以不同的格式解读会有不一样的意义。文件的类型最简单的区分可为纯粹的 ASCII 码文本信息或二进制数据信息。常见的后缀为 TXT 的文本文件存储的信息均为 ASCII 码，把文件中的这些数据转化后打印到屏幕即为文字信息，如文件中存储了数字“41H”，以文本形式解读可以理解为 ASCII 码中的字母“A”。而以 BIN 为后缀的文件存储的是二进制数据文件，以二进制格式解读“41H”则表示数值 65，应用程序可直接利用这个数值进行数值运算。

19.1.2 MP3 文件的原始数据

大家平时熟悉的 MP3 文件一般使用播放软件打开，然后就可以听到相应的音乐。音乐软件在这个过程中做的工作实际是把 MP3 文件中存储的原始数据按照一定格式读取出来，对音频数据进行 MP3 解码，解码后最终得到电压数据，利用这个电压数据通过 DAC 输出到音响设备。如果读者尝试以 Windows 的记事本软件打开会出现什么状况呢？见图 19-1。

是的，乱码！绝大部分是我们看不懂的中文。但是，文件的开头还是有一部分是有意义的“ID3—@ fE TIT2 预感 TPE1 陈奕迅 TALB 1997-2007 跨世纪国语

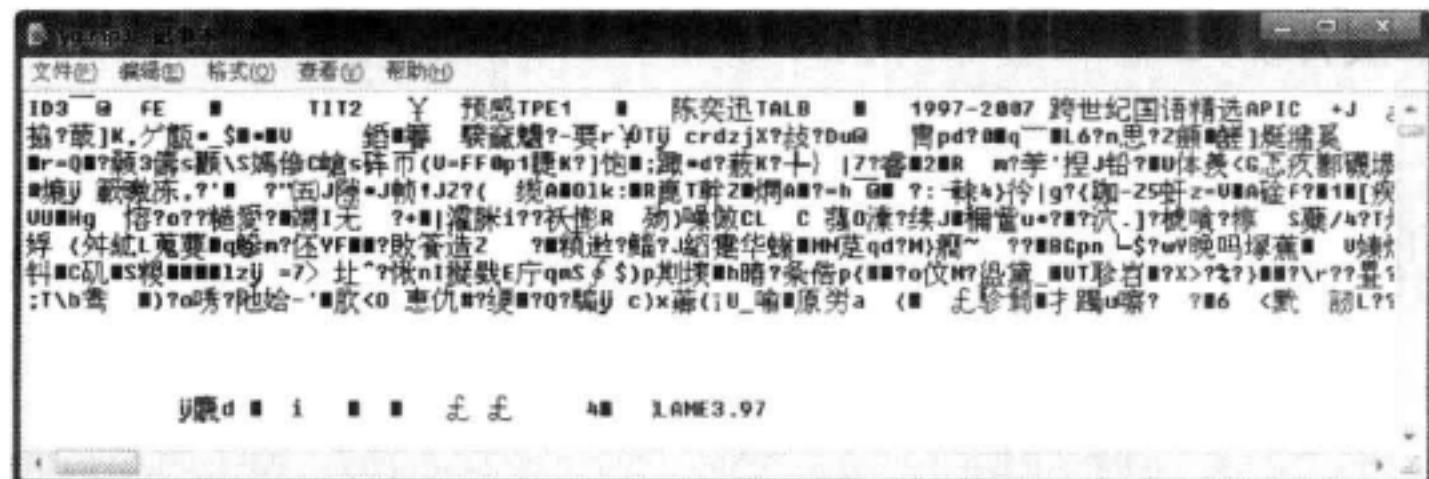


图 19-1 以记事本软件打开 MP3 文件

精选 APIC”。从这部分我们可以知道本 MP3 的名字为“预感”，歌手名为“陈奕迅”，所属专辑为“1997-2007 跨世纪国语精选”。因为记事本软件以 ASCII 码解读 MP3 文件，而 MP3 文件的开头正是以 ASCII 码保存这些音乐信息，所以能够正确解读出来。而后面存储的为压缩后的音频数据，正常的解析方式需要按 MP3 压缩格式解码成电压数据。

如果读者安装有 UltraEdit 软件可以直接查看到最原始的数据，这个软件把文件中的数据按字节读取出来，以十六进制显示出来。见图 19-2，图中最左侧的为地址，中间为文件的数据，最右侧为以 ASCII 码解读数据的结果。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	49	44	33	03	00	40	00	00	66	45	00	00	00	06	00	00	; ID3...@...FE.....
00000010h:	00	00	00	00	54	49	54	32	00	00	00	05	00	00	00	D4	;TIT2.....?
00000020h:	A4	B8	D0	54	50	45	31	00	00	00	07	00	00	00	B3	C2	; 陈PE1.....陈
00000030h:	DE	C8	D1	B8	54	41	4C	42	00	00	00	19	00	00	00	31	; 奕迅TALB.....1
00000040h:	39	39	37	2D	32	30	30	37	20	BF	E7	CA	CO	BC	CD	B9	; 997-2007 跨世纪?
00000050h:	FA	D3	EF	BE	AB	D1	A1	41	50	49	43	00	00	2B	4A	00	; 裸 PIC...+J.
00000060h:	00	01	69	6D	61	67	65	2F	6A	70	65	67	00	00	FF	FE	; ..image/jpeg.. ?

图 19-2 用 UltraEdit 打开 MP3 文件

19.1.3 MP3 文件格式

根据 MP3 的文件标准，它的格式见表 19-1。

表 19-1 MP3 文件格式

ID3V2	包含了作者、作曲、专辑等信息，长度不固定，扩展了 ID3V1 的信息量
Frame	一系列的帧，个数由文件大小和帧长决定，每个帧的长度可能不固定，也可能固定，由位率决定，每个帧又分为帧头和数据实体两部分，帧头记录了 MP3 的位率、采样率、版本等信息，每个帧之间相互独立
ID3V1	包含了作者、作曲、专辑等信息，长度为 128 字节

MP3 文件大体可以分为 3 部分。开头部分是 ID3V2 的一些歌曲信息，文件尾部为旧版本的 ID3V1 信息，现在的 MP3 文件大多没有 ID3V1 部分。文件的其余部分为以帧为单位的音频数据，每帧的开头又含有该帧的比特位速率、采样率等信息，帧的实体数据部分为经压缩后的音乐数据。

19.2 VS1003 硬件解码芯片

对于 MP3 文件的音频数据信息的解码，可以使用软件根据 MP3 压缩格式经过运算得到电压数据，再由 DAC 输出到音响转换成声音。另外一种方法是使用专用的 MP3 解码芯片，只需要把 MP3 文件中的数据读取出来传送到解码芯片中，由它把数据转换成声音。因为使用软件解码的方式要涉及大量的运算，占用资源较多，所以本章讲解的实验是采用硬件芯片解码的方式。

本实验的解码部分采用 VS1003-MP3/WMA 音频解码器，再将解码产生的电压信号送至 TDA1308 功放芯片放大后由音频接口外播出来。

19.2.1 VS1003 芯片简介

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

VS1003 通过一个串行接口来接收输入的比特流, 它可以作为一个系统的从机。输入的比特流被解码, 然后通过一个数字音量控制器到达一个 18 位过采样多位 DAC。通过串行总线控制解码器。除了基本的解码, 在用户 RAM 中它还可以做其他特殊应用, 如 DSP 音效处理。VS1003 原理框图见图 19-3。

本实验中我们只用了圆圈中的那几个数据口, 这些数据口是串行模式的, 我们用到了开发板中的 SPI2 来控制。其中数据经 SI 接口进去, 经解码后由 L、R 这两个左右声道引脚出来, 因为 VS1003 内部集成了一个 DAC, 传输出来的信号都是模拟的, 可直接驱动耳机。

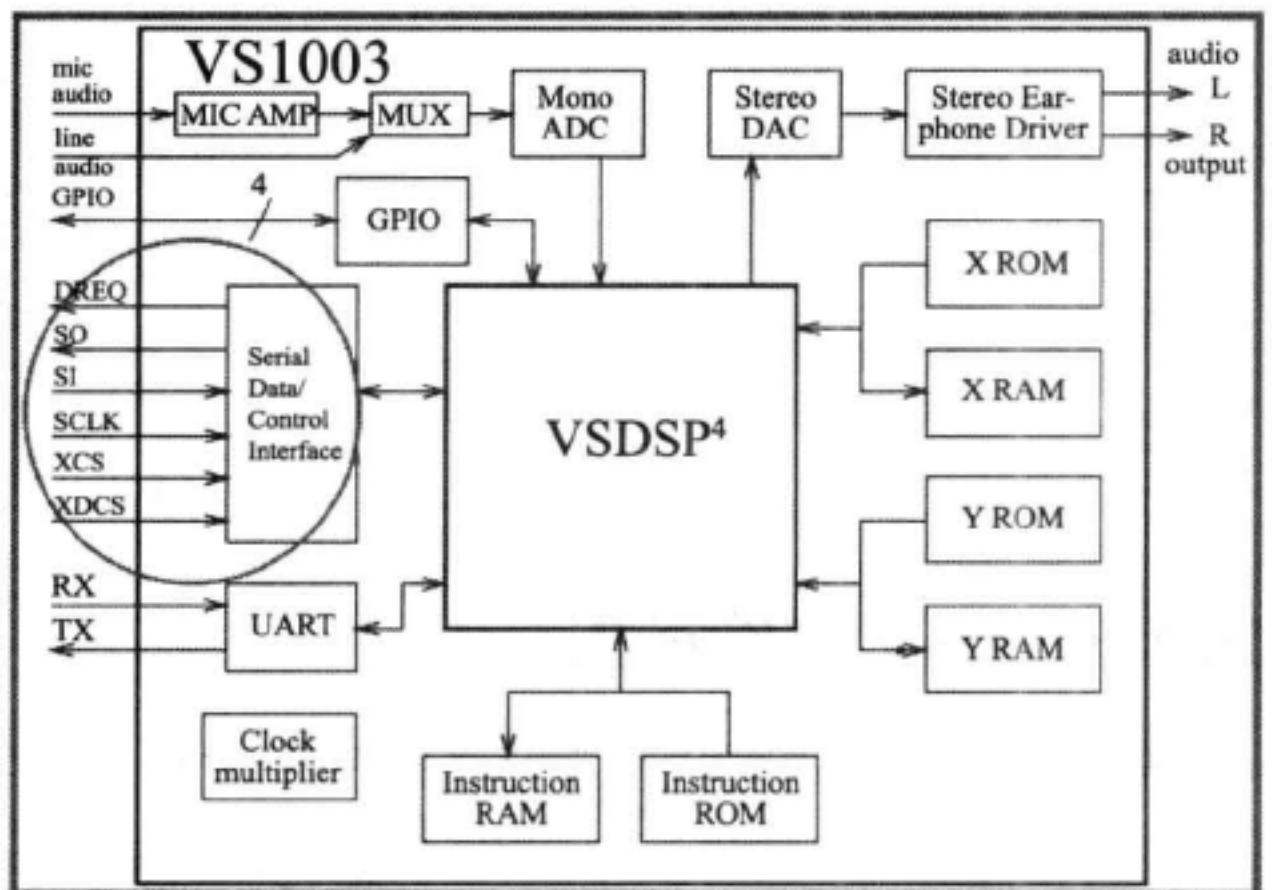


图 19-3 VS1003 原理框图

19.2.2 TDA1308 芯片

由于 VS1003 芯片的输出功率太小, 音效不佳, 所以我们将信号送往 TDA1308 放大后再通过耳机外放出来。现在市面上的 MP3 模块基于成本考虑都没加音频功放, 而是直接驱动音频耳机, 效果可想而知。

TDA1308 是一款双通道的立体耳机驱动器, 是一款专门用于声音驱动的功放。其原理见图 19-4。把由 VS1003 的左、右声道电压信号输入 INA- 和 INB- 接口可得到放大的信号。

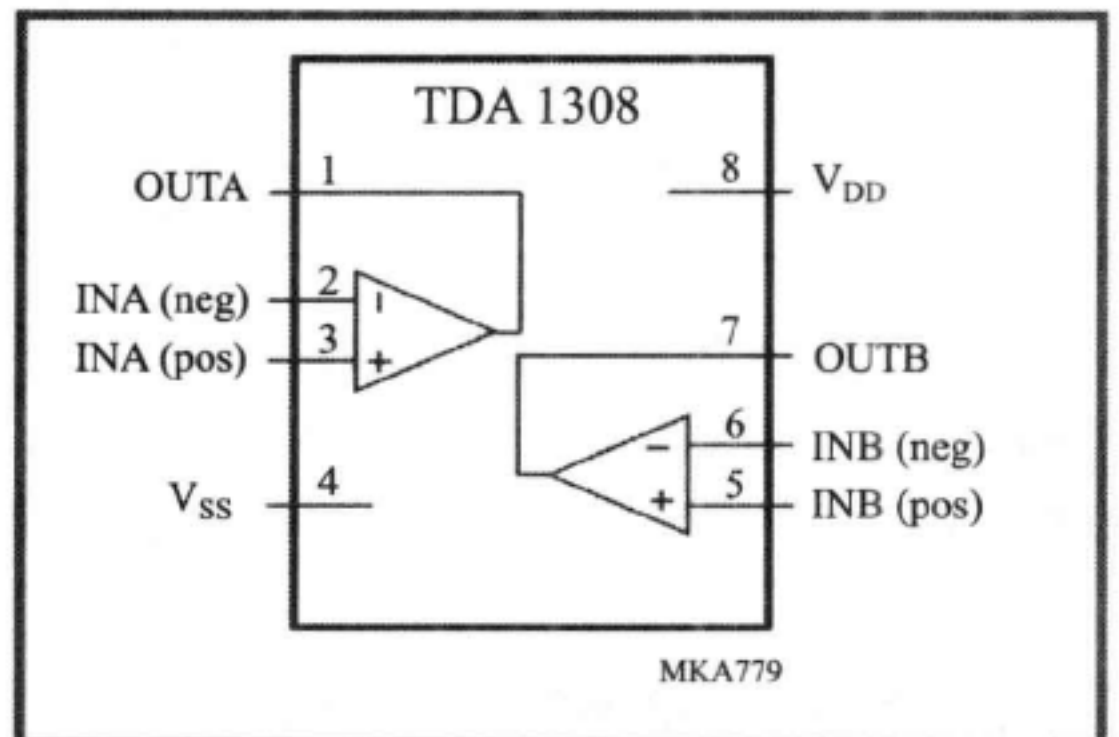


图 19-4 TDA1308 原理框图

19.3 MP3 播放器实验

19.3.1 实验描述及工程文件清单

1. 实验描述

将 MicroSD 卡 (以文件系统 FATFS 访问) 里面的 MP3 文件通过 VS1003B 解码, 然后将解

码后的信号送到功放 TDA1308 后通过耳机播放出来。本例程支持中文、长文件名、4G 的 SD 卡，可以播放 MP3、WMA、MID 和部分 WAV 格式的音频文件。

2. 硬件连接

- ☐ PB13-SPI2_SCK : VS1003B-SCLK
- ☐ PB14-SPI2_MISO : VS1003B-SO
- ☐ PB15-SPI2_MOSI : VS1003B-SI
- ☐ PB12-SPI2_NSS : VS1003B-XCS
- ☐ PB11 : VS1003B-XRET
- ☐ PC6 : VS1003B-XDCS
- ☐ PC7 : VS1003B-DREQ

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_sdio.c
- ☐ FWlib/stm32f10x_dma.c
- ☐ FWlib/stm32f10x_spi.c
- ☐ FWlib/misc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/sdio_sdcard.c
- ☐ USER/usart1.c
- ☐ USER/mp3play.c
- ☐ USER/vs1003.c
- ☐ USER/SysTick.c

5. 文件系统文件

使用 FATFS 0.09 版本源文件：

- ☐ ff9/diskio.c
- ☐ ff9/ff.c
- ☐ ff9/cc936.c

MP3 的硬件原理图见图 19-5。

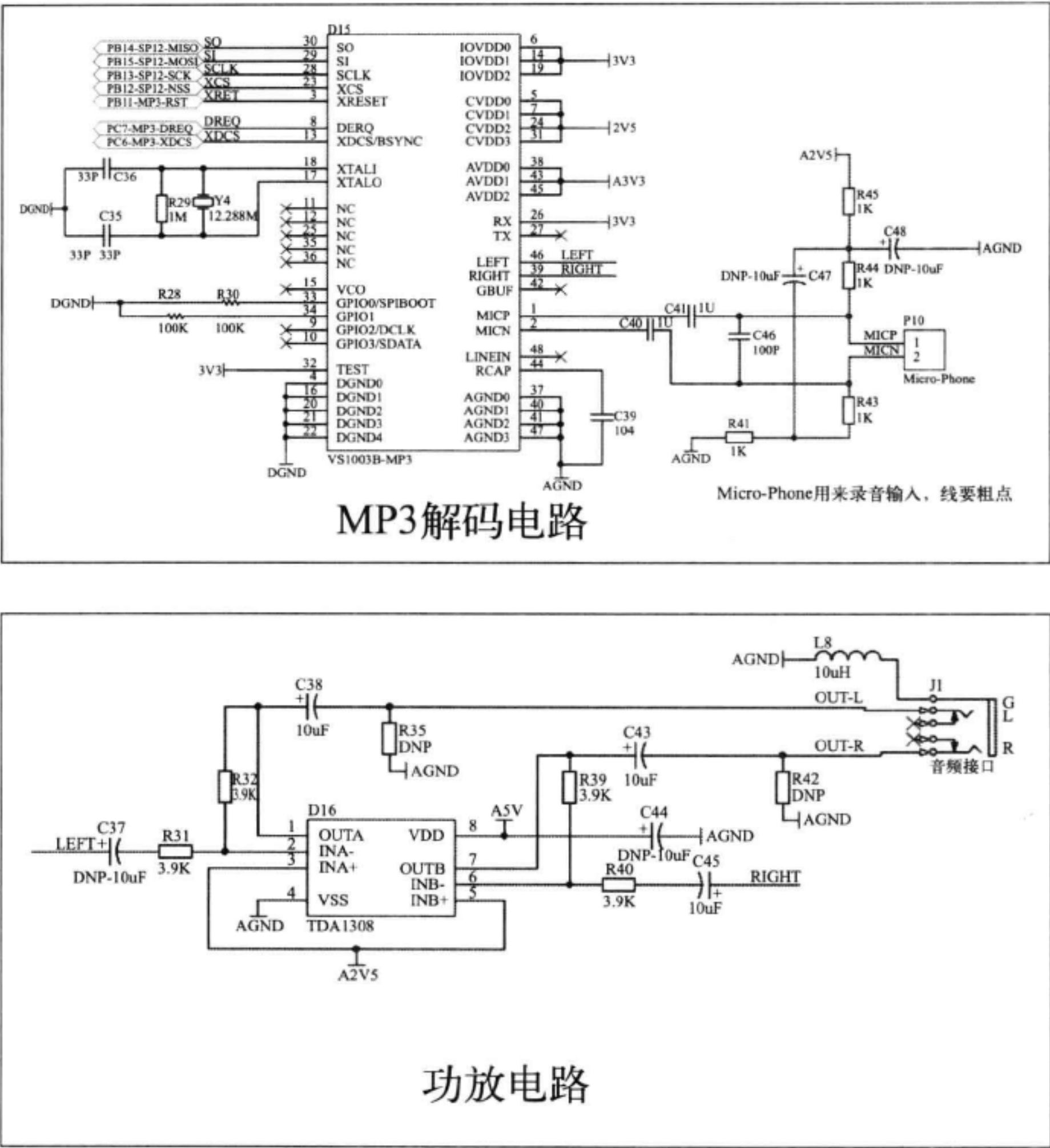


图 19-5 配套 STM32 开发板中 MP3 的硬件原理图

19.3.2 配置工程环境

本 MP3 播放器实验中我们用到了 GPIO、RCC、USART、SDIO、DMA、SPI 外设，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_sdio.c、stm32f10x_dma.c、stm32f10x_spi.c。在 SDIO 中还使用了中断，所以要把 misc.c 文件添加进工程。本实验是在第 18 章文件系统的基础上建立的，先把旧工程中的外设用户文件 sdio_sdcard.c、ff.c、usart1.c、SysTick.c 添加到新工程之中，并新建 mp3play.c、vs1003.c 文件。最后在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 19-1。

代码清单 19-1 MP3 例程的 stm32f10x_conf.h 文件配置

```

1.  /*****
2.   * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
3.   * @author    MCD Application Team
4.   * @version    V3.5.0
5.   * @date      08-April-2011
6.   * @brief     Library configuration file.
7.   *****/
8.
9.
10. #include "stm32f10x_dma.h"
11. #include "stm32f10x_gpio.h"
12. #include "stm32f10x_rcc.h"
13. #include "stm32f10x_sdio.h"
14. #include "stm32f10x_spi.h"
15. #include "stm32f10x_usart.h"
16. #include "misc.h"

```

19.3.3 main 文件

在配置好库的环境之后我们从 main 函数开始分析，见代码清单 19-2。

代码清单 19-2 MP3 例程的 main 函数

```

1. int main(void)
2. {
3.     SysTick_Init();           /* 配置 SysTick 为 10us 中断一次 */
4.     USART1_Config();          /* 配置串口 1 115200 8-N-1 */
5.
6.     /* Interrupt Config, 配置 sdio 的中断优先级, */
7.     NVIC_Configuration();
8.
9.     printf(" \r\n 这是一个 MP3 测试例程 !\r\n " );
10.
11.     VS1003_SPI_Init();        /* MP3 硬件 I/O 初始化 */
12.
13.     MP3_Start();              /* MP3 就绪, 准备播放, 在 vs1003.c 实现 */
14.
15.     MP3_Play();               /* 播放 SD 卡 (FATFS) 里面的音频文件 */
16.
17.     /* Infinite loop */
18.     while (1)
19.     {
20.     }
21. }

```

main 文件执行流程：

- 1) 调用 SysTick_Init() 函数，把 SysTick 配置为 10 μ s 中断一次，SysTick 用于后面应用的延时函数。
- 2) 调用 USART1_Config() 函数配置串口 1 波特率为 115200、8 个数据位、1 个停止位、无硬件流控制。
- 3) 调用 NVIC_Configuration() 函数用于配置 MicroSD 卡的中断优先级。

4) 调用 VS1003_SPI_Init() 用于初始化 MP3 解码芯片 VS1003B 需要用到的 I/O 口, 包括数据口 (SPI2) 和控制 I/O。VS1003_SPI_Init() 由用户在 vs1003.c 中实现。假如我们要将数据口换成 SPI1 或者改变其他控制 I/O, 只需改变这个函数即可, 移植性非常强。关于 STM32 的 SPI 接口详细使用教程可参照前面的 SPI 章节。

5) 调用 MP3_Start() 使 MP3 进入就绪模式 (Standby), 随时播放音乐。MP3_Start() 在 vs1003.c 中实现。

6) 调用 MP3_Play() 函数, 这个函数逐个扫描我们卡里面的音频文件, 把根目录下的所有音频文件播放一次, 若音频文件放在其他目录, 可以通过修改代码中的文件路径来实现。

19.3.4 控制 VS1003 进入准备状态

1. SCI 和 SDI 时序

VS1003 芯片使用 SPI 接口进行通信, 但它与普通的通信形式又有稍许区别。根据传送的为普通数据还是命令分为 SDI 时序和 SCI 时序, 见图 19-6 和图 19-7。

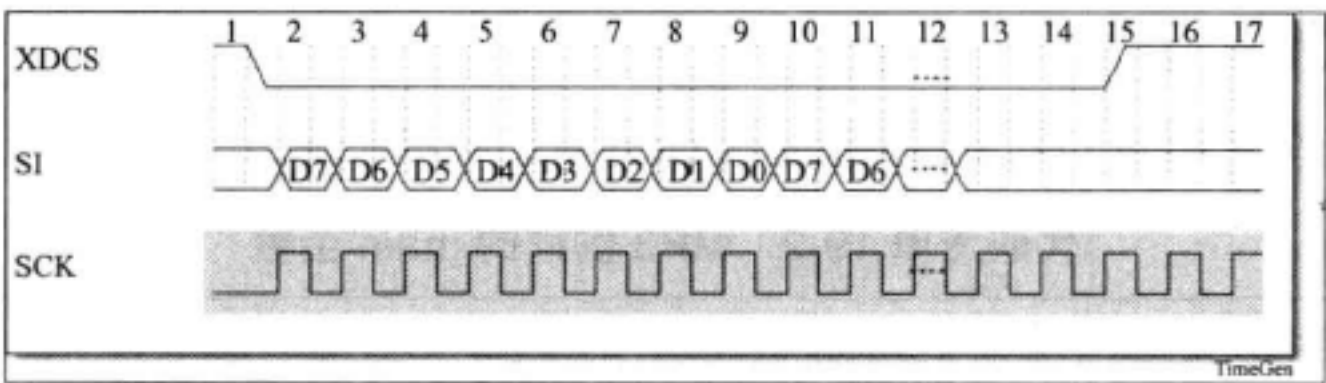


图 19-6 SDI 时序图

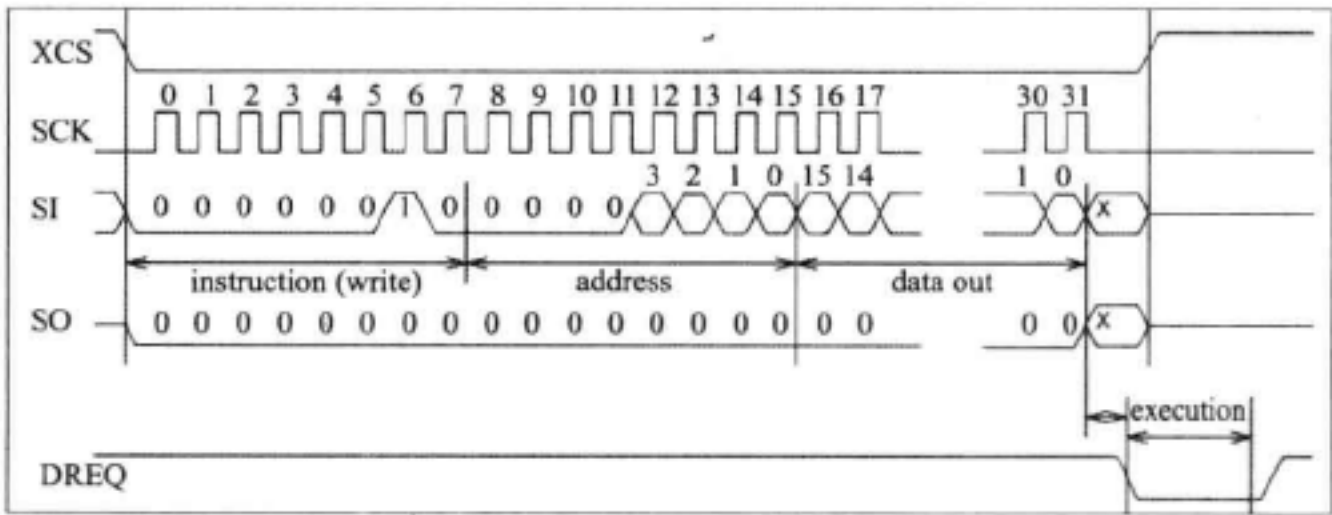


图 19-7 SCI 时序图

在两个时序图中, 主要区别为信号线 XDCS 和 XCS。当主机向 VS1003 传送的为数据时, 使用 SDI 时序, 传送过程中把 XDCS 引脚拉低, 开始传输数据, 这些数据主要为待解码的 MP3 数据流。当主机向 VS1003 传送的为命令时, 使用 SCI 时序, 传送过程中把 XCS 引脚拉低。这些命令主要是写入 VS1003 的寄存器, 控制它的工作方式。

2. 写入控制参数

在 main 文件的执行流程中, 把各种 STM32 外设接口初始化完成后, 调用了 MP3_Start() 函数, 见代码清单 19-3。它主要调用用户函数 Mp3WriteRegister() 通过 SPI 接口对 VS1003 芯片的寄存器写入控制参数, 达到控制 VS1003 芯片的目的, 这些控制参数都是从 VS1003 的芯片手册查阅得到的。

代码清单 19-3 MP3_Start() 函数

```

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

```

本函数写入的均为命令, 所以使用的是 SCI 时序。函数中的第 11 ~ 18 行有三个特殊的宏: TRST_SET()、TXDCS_SET() 和 TCS_SET(), 它们分别用于控制 VS1003 的复位引脚、XDCS 引脚、CS 引脚, 它们具体的定义见代码清单 19-4。

代码清单 19-4 控制引脚宏定义

```

1. /*
2. * VS1003 (音频解码芯片) 控制接口宏定义, 这些 I/O 都是普通 I/O, 如需改变 I/O,
3. * 只要改变这写宏定义即可, 其他应用函数不用改变, 可方便移植。
4. * 这些接口分别为:
5. * 1-XCS   VS1003 片选
6. * 2-RST   VS1003 复位
7. * 3-XDCS  VS1003 数据命令选择
8. * 4-DREQ  VS1003 数据中断
9. */
10.
11. #define TCS    (1<<12) // PB12-XCS
12.
13. #define TCS_SET(x)  GPIOB->ODR=(GPIOB->ODR&~TCS)|(x?TCS:0)
14.
15. #define RST     (1<<11) // PB11-RST
16. #define TRST_SET(x) GPIOB->ODR=(GPIOB->ODR&~RST)|(x?RST:0)
17.

```

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

有了这几个宏，我们就可以方便地改变这几个引脚的电平从而控制不同的时序，如 TXDCS_SET(1) 即把 XDCS 引脚拉高，TXDCS_SET(0) 即把该引脚拉低。

MP3_Start() 函数中第 11 ~ 18 行调用这几个宏把 VS1003 复位后，先把 CS、XDCS 都拉高，即复位后暂时不传输任何数据或命令。

之后调用 Mp3WriteRegister() 开始向 VS1003 写入各种控制参数，该函数定义见代码清单 19-5。

代码清单 19-5 Mp3WriteRegister() 函数

```
1. /*
2.  * 函数名：Mp3WriteRegister
3.  * 描述   ：写 VS1003 寄存器
4.  * 输入   ：-addressbyte 寄存器地址
5.  *          -highbyte   高 8 位
6.  *          -lowbyte    低 8 位
7.  * 输出   ：无
8.  * 调用   ：内 / 外部调用
9.  */
10. void Mp3WriteRegister(u8 addressbyte, u8 highbyte, u8 lowbyte)
11. {
12.     TXDCS_SET(1);
13.     TCS_SET(0);
14.
15.     VS1003_WriteByte( VS_WRITE_COMMAND );
16.     VS1003_WriteByte( addressbyte );
17.     VS1003_WriteByte( highbyte );
18.     VS1003_WriteByte( lowbyte );
19.
20.     TCS_SET(1);
21. }
```

从本函数可以看到，它是完全根据 SCI 时序来编写的。把 xCS 引脚拉低后，调用 VS1003_WriteByte() 按时序向 VS1003 写入命令、寄存器地址、控制参数，最后再把 xCS 引脚拉高，完成一次参数写入。

19.3.5 播放 MP3 文件

现在我们来大概分析一下 MP3_Play() 这个函数，见代码清单 19-6。这里涉及一些文件系统操作函数，关于这部分函数的操作大家可参考前面的内容或者阅读 FATFS 的官方文档。

代码清单 19-6 MP3_Play() 函数

```
1. /*
2.  * 函数名：MP3_Play
3.  * 描述   ：读取 SD 卡里面的音频文件，并通过耳机播放出来
4.  *          支持的格式：mp3,mid,wma, 部分的 wav
5.  * 输入   ：无
6.  * 输出   ：无
7.  * 说明   ：已添加支持长中文文件名
8.  */
```

```

9. void MP3_Play(void)
10. {
11.
12.     FATFS fs;           // Work area (file system object) for logical drive
13.     FRESULT res;
14.     UINT br;            /* 读取出的字节数, 用于判断是否到达文件尾 */
15.     FIL fsrc;           // file objects
16.     FILINFO finfo;      /* 文件信息 */
17.     DIR dirs;
18.     uint16_t count = 0;
19.
20.     char lfn[70];        /* 为支持长文件的数组, [] 最大支持 255 */
21.     char j = 0;
22.     char path[100] = {" "}; /* MicroSD 卡根目录 */
23.     char *result1, *result2, *result3, *result4;
24.
25.     BYTE buffer[512];    /* 存放读取出的文件数据 */
26.
27.     finfo.lfname = lfn;   /* 为长文件名分配空间 */
28.     finfo.lfsize = sizeof(lfn); /* 空间大小 */
29.
30.     f_mount(0, &fs);     /* 挂载文件系统到 0 区 */
31.
32.     if (f_opendir(&dirs, path) == FR_OK) /* 打开根目录 */
33.     {
34.         while (f_readdir(&dirs, &finfo) == FR_OK) /* 依次读取文件名 */
35.         {
36.
37.             if (finfo.fattrib & AM_ARC) /* 判断是否为存档型文档 */
38.             {
39.                 if(finfo.lfname[0] == NULL && finfo.fname != NULL)
40. /* 当长文件名称为空, 短文件名非空时转换 */
41.                     finfo.lfname = finfo.fname;
42.
43.
44.                 if( !finfo.lfname[0] )
45. /* 文件名为空即到达了目录的末尾, 退出 */
46.                     break;
47.
48.                 printf( " \r\n 文件名为: %s \r\n", finfo.lfname );
49.
50.                 result1 = strstr( finfo.lfname, ".mp3" );
51. /* 判断是否为音频文件 */
52.                 result2 = strstr( finfo.lfname, ".mid" );
53.                 result3 = strstr( finfo.lfname, ".wav" );
54.                 result4 = strstr( finfo.lfname, ".wma" );
55.
56.                 if ( result1!=NULL || result2!=NULL || result3!=NULL || result4!=NULL )
57.                 {
58.
59.                     if(result1 != NULL) /* 若是 mp3 文件则读取 mp3 的信息 */
60.                     {
61.                         res = f_open( &fsrc, finfo.lfname, FA_OPEN_EXISTING | FA_
62. READ ); /* 以只读方式打开 */
63.
64.                         /* 获取歌曲信息 (ID3V1 tag / ID3V2 tag) */
65.                         if ( Read_ID3V1(&fsrc, &id3v1) == TRUE )
66.                         { // ID3V1 tag
67.                             printf( "\r\n 曲目      : %s \r\n", id3v1.title );
68.                             printf( "\r\n 艺术家   : %s \r\n", id3v1.artist );
69.                             printf( "\r\n 专辑      : %s \r\n", id3v1.album );
70.                         }

```

```

71.             else
72.             {
73. // 有些MP3文件没有ID3V1 tag, 只有ID3V2 tag
74.             res = f_lseek(&fsrc, 0);
75.             Read_ID3V2(&fsrc, &id3v2);
76.
77.             printf( "\r\n 曲目      : %s \r\n", id3v2.title );
78.             printf( "\r\n 艺术家   : %s \r\n", id3v2.artist );
79.             }
80.         }
81. /* 使文件指针 fsrc 重新指向文件头, 因为在调用 Read_ID3V1/Read_ID3V2 时,
82.    fsrc 的位置改变了 */
83.     res = f_open( &fsrc, finfo.lfname, FA_OPEN_EXISTING | FA_READ );
84.     res = f_lseek(&fsrc, 0);
85.
86.
87.     br = 1; /* br 为全局变量 */
88.     TXDCS_SET( 0 ); /* 选择 VS1003 的数据接口 */
89. /* ----- 一曲开始 ----- */
90.     printf( " \r\n 开始播放 \r\n" );
91.     for (;;)
92.     {
93.         res = f_read( &fsrc, buffer, sizeof(buffer), &br );
94.         if ( res == 0 )
95.         {
96.             count = 0; /* 512 字节完重新计数 */
97.
98.             Delay_us( 1000 ); /* 10ms 延时 */
99.             while ( count < 512 )
100.            /* SD卡读取一个 sector, 一个 sector 为 512 字节 */
101.            {
102.                if ( DREQ != 0 )
103.                /* 等待 DREQ 为高, 请求数据输入 */
104.                {
105.                    for (j=0; j<32; j++ )
106.                    /* VS1003 的 FIFO 只有 32 个字节的缓冲 */
107.                    {
108.                        VS1003_WriteByte( buffer[count] );
109.                        count++;
110.                    }
111.                }
112.            }
113.
114.            if (res || br == 0) break;
115.            /* 出错或者到了MP3文件尾 */
116.        }
117.        printf( " \r\n 播放结束 \r\n" );
118.        /* ----- 一曲结束 ----- */
119.        count = 0;
120.        /* 根据 VS1003 的要求, 在一曲结束后需发送 2048 个 0 来确保下一首的正常播放 */
121.        while ( count < 2048 )
122.        {
123.            if ( DREQ != 0 )
124.            {
125.                for ( j=0; j<32; j++ )
126.                {
127.                    VS1003_WriteByte( 0 );
128.                    count++;
129.                }
130.            }
131.        }
132.        count = 0;

```



```

133.          TXDCS_SET( 1 );
134.      /* 关闭 VS1003 数据端口 */
135.          f_close(&fsrc); /* 关闭打开的文件 */
136.      }
137.  }
138.  } /* while (f_readdir(&dirs, &finfo) == FR_OK) */
139.  } /* if (f_opendir(&dirs, path) == FR_OK) */
140.  } /* end of MP3_Play */

```

本段代码十分长，建议读者配合如图 19-8 所示流程图，在具体的工程中使用 MDK 进行阅读。

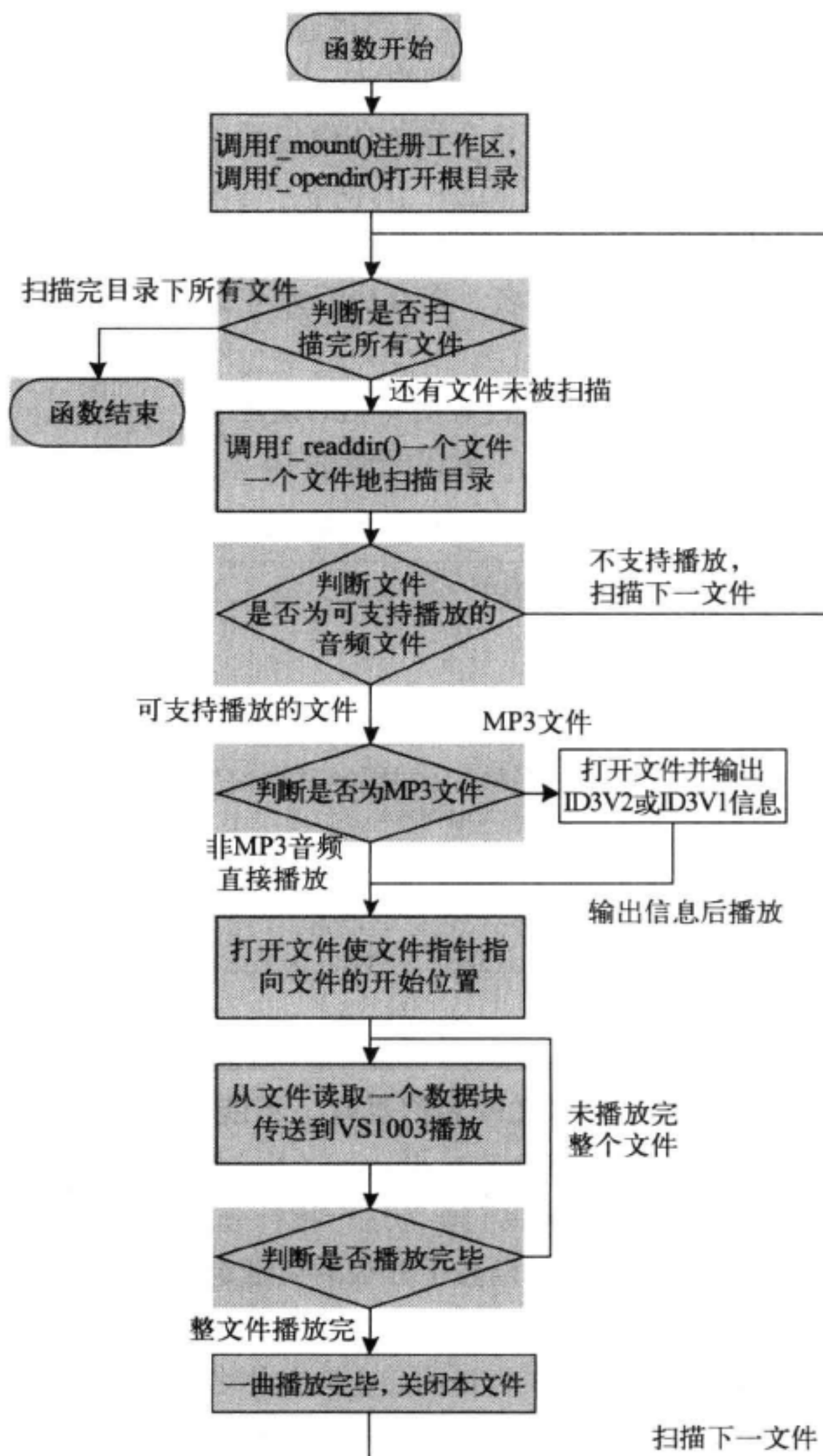


图 19-8 MP3 文件播放流程

其流程解释如下：

1) 第 30 行，调用函数 `f_mount()` 为我们在文件系统中注册一个工作区，并初始化盘符的名为 0。这个函数调用了底层的 `disk_initialize()`，进行 SDIO 的初始化，所以在文件操作之前必须调用这个函数。不建议在 `main` 函数直接调用 `disk_initialize()` 来对 SDIO 进行初始化，要尽量使用封装好的脱离硬件层的函数，这样能使代码的可移植性更好。

2) 第 32 行，调用函数 `f_opendir()` 用于打开卡的根目录，并将这个根目录关联到 `dirs` 这个结构指针，然后我们就可以通过这个结构指针来操作这个目录了，其实这个结构指针就类似 Linux 下系统编程中的文件描述符，不论是操作还是目录都得通过文件描述符才能操作。

3) 第 34 行，调用 `f_readdir()` 函数通过刚刚的 `dirs` 结构指针来读取目录里面的信息，并将目录的信息存储在 `finfo` 这个结构体变量中。这个结构体中包括了文件名、文件大小、文件类型、修改时间等信息。

4) 第 37 ~ 54 行，根据读取得的 `finfo` 结构体值判断文件的属性，如果是存档型文件的话就将文件名打印出来，然后比较文件的后缀名，查看是否为可支持播放的音频文件，支持的音频格式有 MP3、MID、WAV、WMA。

5) 其中第 39 ~ 41 行代码，是为了支持长文件名的设置。

根据文件系统的 `finfo` 结构体类型定义，见代码清单 19-7。

代码清单 19-7 `finfo` 结构体类型定义

```

1. /* File status structure (FILINFO) */
2.
3. typedef struct {
4.     DWORD    fsize;           /* File size */
5.     WORD     fdate;           /* Last modified date */
6.     WORD     ftime;           /* Last modified time */
7.     BYTE     fattrib;         /* Attribute */
8.     TCHAR    fname[13];       /* Short file name (8.3 format) */
9.     #if _USE_LFN
10.    TCHAR*    lfname;          /* Pointer to the LFN buffer */
11.    UINT      lfname_size;      /* Size of LFN buffer in TCHAR */
12. #endif
13. } FILINFO;

```

可知为支持使用长文件名，在使用文件名信息时，不能再使用结构体中的成员 `FILINFO->fname`（短文件名数组），而应该使用 `FILINFO->lfname`（长文件名指针）。而且长文件名在结构体中定义的是一个指针类型变量，在使用前我们要为这个指针分配内存空间，注意不要使用野指针。具体的分配空间方法可以参照 `MP3_Play()` 函数中开头的变量定义和赋初值部分。

另外，如果读取的文件名的长度不超过 13 个字节的空间时（短文件名），文件名的信息只会保存在短文件名数组中，而这时 `FILINFO->lfname`（长文件名指针）的值将会是空的（NULL），为了统一使用 `FILINFO->lfname`，在代码中加了这一段判断语句，把文件名信息无论长短都存储在 `FILINFO->lfname` 成员中。

6) 第 56 ~ 80 行，判断该可支持文件是否为 MP3 类型的文件，对于 MP3 类型文件，在第

61 行调用 `f_open()` 打开文件，再调用 `Read_ID3V1()` 和 `Read_ID3V2()` 来读取 MP3 的文件信息。这些文件信息是属于 MP3 文件的内部数据，可以参照 MP3 文件存储格式来理解这两个函数，实质就是把文件记录的数据，按格式把相应的信息整合到结构体里便于使用而已，这里读取出来的信息包括歌曲名、歌手名、专辑等。

7) 第 83 行，调用 `f_open()` 打开这个音频文件，准备播放。对于非 MP3 的音频文件，没有执行第 5 个步骤，所以是第一次打开本文件。若为 MP3 文件，重新打开这个文件使文件指针重新指向文件的头部。

8) 第 84 ~ 116 行，进入一个大循环中播放我们的 MP3 文件。我们把读取到的音频数据直接通过 SPI 接口（使用 SDI 时序）送入 VS1003，就可以进行各种音频数据的解码了。

9) 第 93 行，在播放的大循环中，调用函数 `f_read()` 从文件中读取 512 个字节的数据到缓冲区中，这是因为卡的一个扇区是 512 个字节，一次至少要读取一个扇区。

10) 第 105 ~ 110 行，本段代码仍为播放循环中的一部分，调用函数 `VS1003_WriteByte()` 把从 MP3 文件读出的扇区数据写入 VS1003 的数据缓冲区。在这里每次只能把扇区数据的 32 个字节写入 VS1003，这是因为它的接收 FIFO 的大小为 32 个字节，写多了无效。

11) 第 114 行，根据 `f_read()` 函数中的 `br` 变量判断，当文件出错或者一曲播放完毕时就跳出播放循环，并打印出“播放结束”的调试信息。

12) 第 121 ~ 131 行，根据 VS1003 的要求，在一曲结束后需发送 2048 个 0 来确保下一首的正常播放。

13) 第 131 行至函数结尾，一曲播放完毕我们关闭 VS1003 的数据端，并关闭打开的文件，等待下一曲的播放，直到目录下的音频文件播放完为止。

本函数里面涉及了 VS1003 操作的一些特性，需大家参考 VS1003 的 datasheet 来帮助理解。

19.3.6 STM32 的堆栈

在本节详细讲解一下为支持中文长文件名的文件系统配置。

为支持这些特性，需要在文件系统 `ffconf.h` 文件中的 Namespace configuration 宏配置中设定，见代码清单 19-8。

代码清单 19-8 Namespace configuration 宏配置

```

1. /*-----*/
2. / Locale and Namespace Configurations
3. /-----*/
4.
5. #define _CODE_PAGE 936
6.
7.
8. #define _USE_LFN 2 /* 0 to 3 */
9. #define _MAX_LFN 255 /* Maximum LFN length to handle (12 to 255) */
10.
11. #define _LFN_UNICODE 0 /* 0:ANSI/OEM or 1:Unicode */

```

```
12.  
13. #define _FS_RPATH      0    /* 0 to 2 */
```

1. Code page

修改的第一个宏配置是 _CODE_PAGE，改成简体中文的 936。

Code page 是什么？我们知道 ASCII 码的前 7 位定义的是我们常用的标准字符集，于是 128 位以下的用处达成了共识，而 ASCII 码中的第 8 位没有被使用，对于 128 位以上的可能有不同的解释，这些不同的解释就称为 code_page，我们使用 936 这个宏就是调用了简体中文的 code_page。所以要支持中文，还要添加 FATFS 源文件中 option 目录下的 cc936.c 文件到工程中。

2. 修改栈空间大小

接下来还要修改 _USE_LFN 和 MAX_LFN 的宏，这两个是长文件名支持的配置。

1) 宏 MAX_LFN 定义了最大文件名长度，单位为 Byte。我们把它设置为 255，即最大支持长度为 255 字节的文件名。

2) 当宏 _USE_LFN 的值大于或等于 1 时，表示开启长文件支持。不同的参数值又表示不同的存储方式：

- ❑ 1 表示长文件名的存储在静态存储区。
- ❑ 2 表示长文件名的存储在栈区。
- ❑ 3 表示长文件名的存储在堆区。

在本工程中对这个宏赋值为 2，也就是把长文件名存储的空间设置在内存中的栈区。这里涉及变量的存储分布问题，见图 19-9。

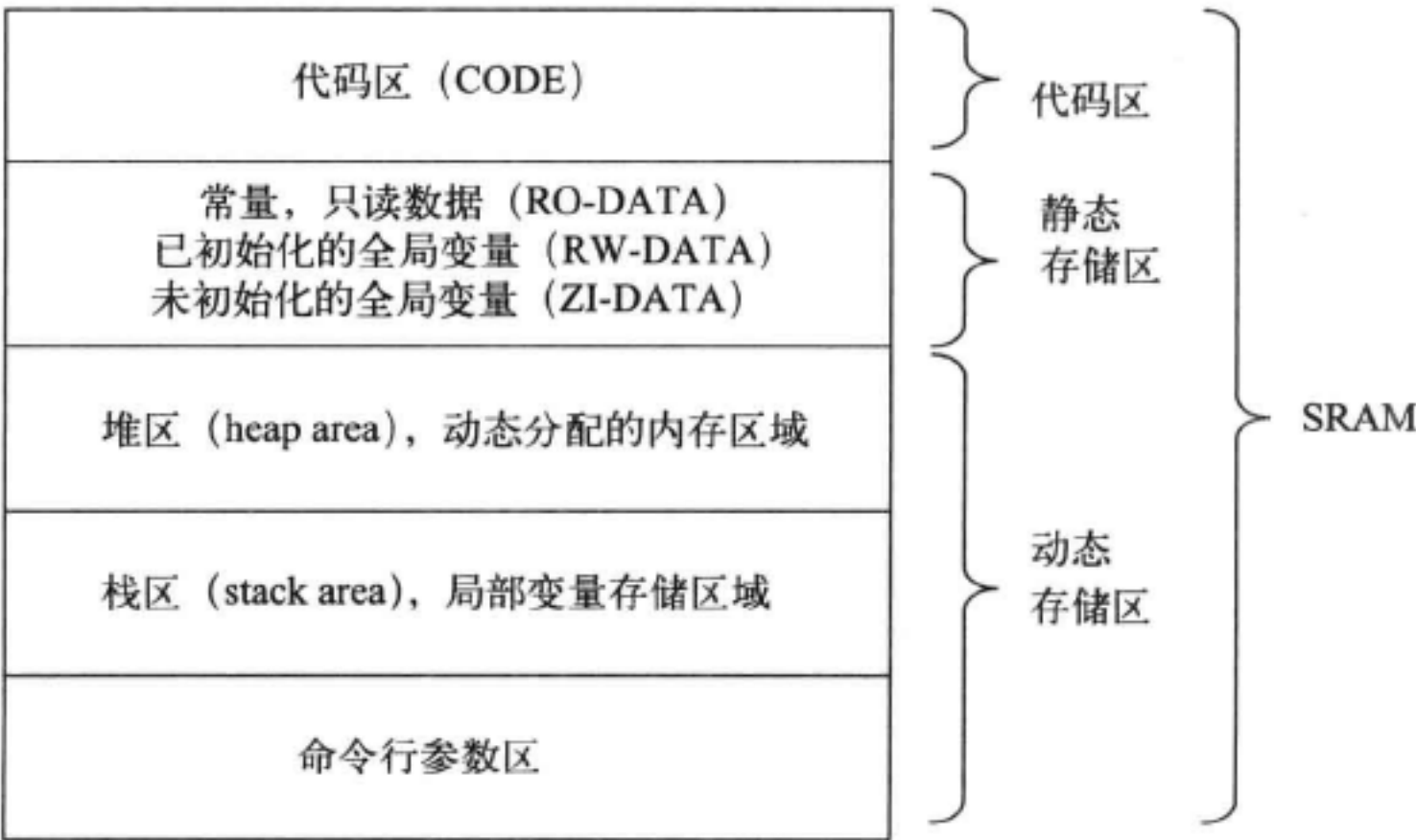
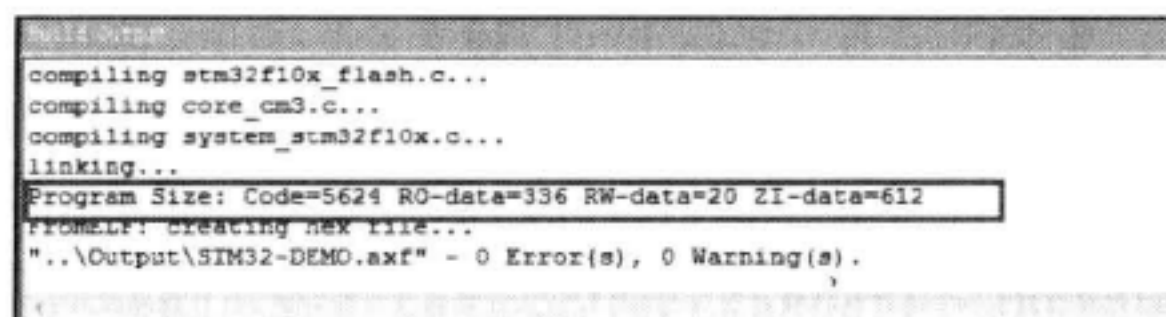


图 19-9 SRAM 内存变量分布图

使用 MDK 编译完工程之后，编译的提示信息会告诉我们本工程使用的内存分布，见图 19-10。

编译信息中的 Code 表示代码段大小，上电运行后，它被加载到内存的代码区。



```

compiling stm32f10x_flash.c...
compiling core_cm3.c...
compiling system_stm32f10x.c...
linking...
Program Size: Code=5624 RO-data=336 RW-data=20 ZI-data=612
FROMELF: creating HEX file...
"..\\Output\\SIM32-DEMO.axf" - 0 Error(s), 0 Warning(s)

```

图 19-10 编译工程后的内存分布信息

RO-data 为工程中的常量, RW-data 为已初始化的全局变量, ZI-data 为未被初始化的全局变量, 这三部分存放在静态存储区。在 C 语言中, 存储在代码区和静态存储区的内存空间是不会被回收的。可被回收的是动态存储区, 它包括堆区和栈区。

堆区是在调用 ANSI C 标准的 malloc() 或 calloc() 函数时, 动态申请内存时使用的, 这些函数获得的内存空间就位于堆区。这部分内存空间的释放需要调用 free() 函数。

栈区是调用子函数时局部变量存储的空间, 子函数结束时会自动释放内存空间。

对于内存空间的分配, 可以在工程中的启动文件 startup_stm32f10x_hd.s 中进行修改, 在启动文件的开头部分, 就是关于堆、栈区的地址定义, 见代码清单 19-9。

代码清单 19-9 启动文件中堆、栈区的地址定义

```

1. ; <h> Stack Configuration
2. ;   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
3. ; </h>
4.
5. Stack_Size      EQU      0x00000200
6.
7.                AREA      STACK, NOINIT, READWRITE, ALIGN=3
8. Stack_Mem       SPACE    Stack_Size
9. __initial_sp
10.
11. __initial_spTop EQU  0x20000400 ; stack used for SystemInit_ExtMemCtl
12.
13.                ; always internal RAM used
14.
15.; <h> Heap Configuration
16.;   <o> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
17.; </h>
18.
19.Heap_Size        EQU      0x00000000
20.
21.                AREA      HEAP, NOINIT, READWRITE, ALIGN=3
22. __heap_base
23.Heap_Mem         SPACE    Heap_Size
24. __heap_limit

```

代码中第 5 ~ 11 行, 把栈区的大小设置为 0x0000 0200, 把栈区基地址设置为 0x2000 0400。在第 19 ~ 24 行, 把堆区的大小设置为 0, 也就是说, 这样的配置不能使用动态内存分配。

由于栈区存放的内容在子函数调用后会被释放, 而且少用全局变量会让代码的移植性更好, 把文件系统长文件名的存储空间放在栈区。

另外, 因为我们的 MP3_Play() 函数中定义了很多局部变量, 而且为了演示代码, 把长文件

0 的支持设置到了极限大，为 255 字节（实际上可以设置小一点），使得占用的栈空间太大，导致 STM32 硬中断（harddefault）。用 JTAG 调试时发现会执行到 harddefault 的中断服务函数中，这种错误是应该避免的。所以要使本工程正常运行，我们要在启动文件 startup_stm32f10x_hd.s 的开始部分修改栈空间大小，见代码清单 19-10。

代码清单 19-10 修改栈空间大小

1. Stack_Size	EQU	0x00000f00	;Stack_Size, 标号。EQU 定义
2.	AREA	STACK, NOINIT, READWRITE, ALIGN=3	
3. __initial_sp			

在这个文件中把原来的：

```
Stack_size EQU 0x00000400
```

改成了：

```
Stack_size EQU 0x0000f00
```

如果不修改启动文件的这部分，编译工程是没有提示错误的，但在 STM32 上实际运行时，却无法得到正常的运行结果。

当调试程序的时候会发现代码莫名奇妙地卡在 harddefault 的硬中断里，这时可以检查一下是不是在启动文件中把栈大小设置得太保守了，可以根据实际需要把它设置得大一点，这种调试方式同样适用于其他工程，如在 STM32 上运行操作系统，一般也需要对它的堆栈大小做出修改。

SDIO 驱动、文件系统的移植、MP3 文件的播放，这三章内容讲解了一个完整的工程，包括底层驱动、系统层和应用层，做完这个实验，相信读者对软件的分层概念又有了更深刻的认识。

19.3.7 实验现象

将配套 STM32 开发板供电（DC5V），插上 J_LINK，插上串口线（两头都是母的交叉线），插上 MicroSD 卡（我们用的是 1GB，也可支持 4GB 的），在卡的根目录下要有 MP3 文件，打开超级终端，配置超级终端为“115200 8-N-1”，将编译好的程序下载到开发板，即可看到超级终端打印出如图 19-11 所示信息。

在卡的根目录下存放 1 个 MP3 文件、1 个 WMA 文件。可以看到，这个代码支持了超长的中文文件名；也支持了 WMA 的格式，根据 VS1003 的 datasheet 说明，还可以支持 MID 和部分的 WAV 音频，大家可以尝试一下。插上耳机，音质堪比电脑，音量可通过耳机来调节，前提是你的耳机要能调节音量才行。

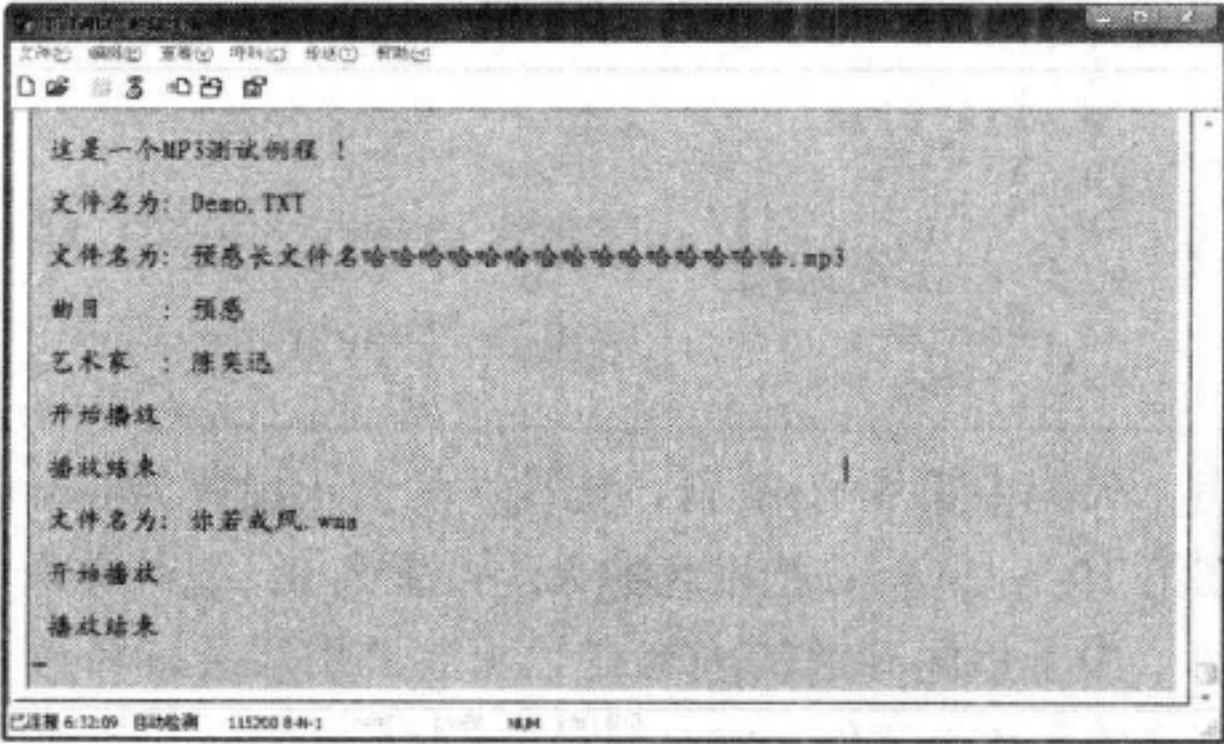


图 19-11 MP3 实验现象



第 20 章

USB 大容量存储器实例

USB 作为电子设备中最常用连接方式，读者一定不会觉得陌生。USB (Universal Serial Bus)，译作通用串行总线，由于它易于扩展、价格低廉、易于升级、速度快和支持热插拔等优点，被广泛用于与 PC 相连的设备中。

20.1 USB 协议分析


20.1.1 协议版本

曾被广泛使用的 USB 协议版本有 USB 1.1、USB 2.0，而目前公布的最新 USB 协议为 USB 3.0，主要由于数据线的增加，USB 3.0 数据传输速度有了很大的提高。USB 1.1 协议支持低速模式 (1.5Mb/s) 和全速模式 (12Mb/s)，而 USB 2.0 协议还支持高速模式 (480Mb/s)。不同的器件对 USB 协议的支持是不一样的，如本书采用的 STM32 芯片仅支持 USB 2.0 中的全速模式 (12Mb/s)。在本文中关于 USB 协议的解释都是基于 USB 2.0 全速模式的，在不同的协议或不同的模式下会有一些区别。

20.1.2 USB 电气特性

1. USB 接口

USB 协议规定的 USB 接口主要有 A 和 B 两个类型，每种类型有对应的插头和插座，见图 20-1。A 型接口主要用于 USB 主机，B 型则主要用于 USB 集线器或 USB 设备。

拿起一根 USB 线缆，使 USB 标志  朝上，可以看到有四根露出的金属触点，对于 A 型插头，从左往右：第 1 脚为 V_{BUS} 、2 脚为 D⁻、3 脚为 D⁺、4 脚为 GND，而对于 B 型插头，左上角的为第 1 脚，按顺时针方向数为 2、3、4 针脚。

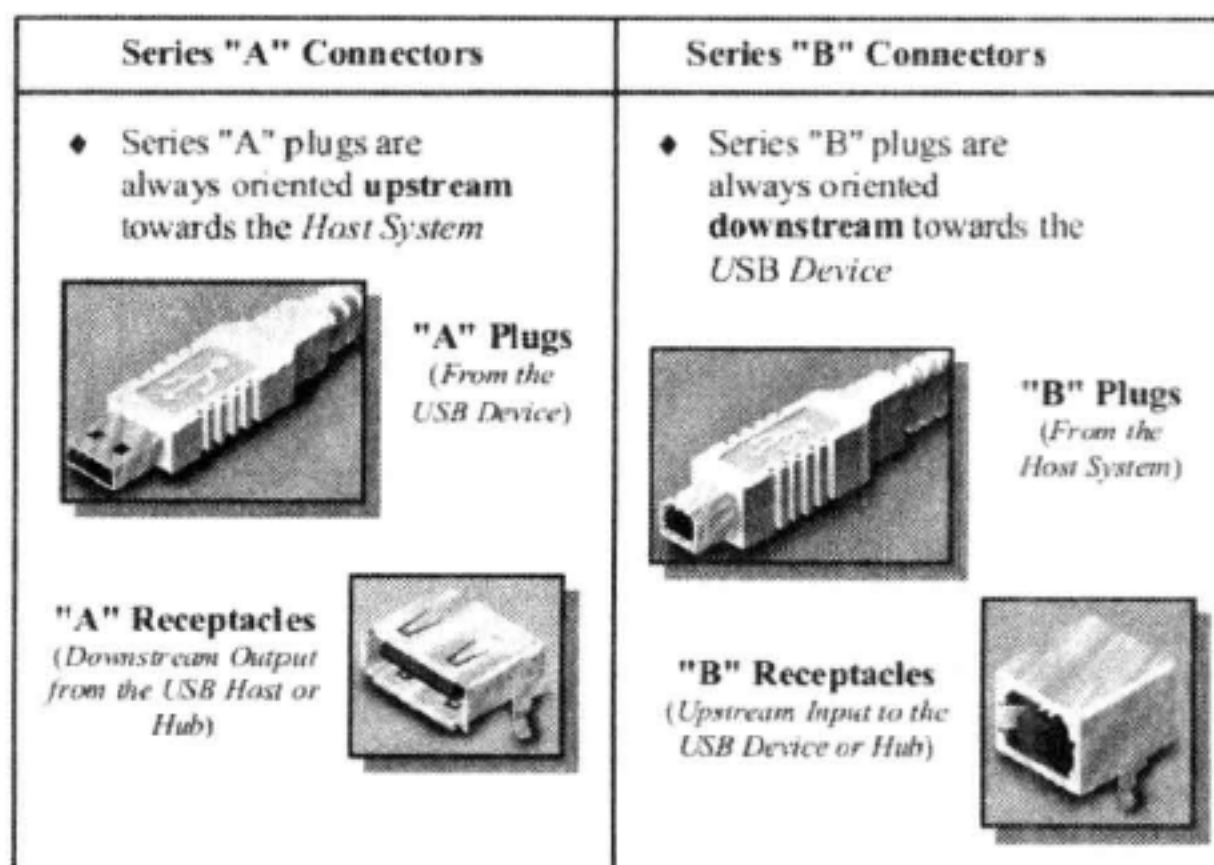


图 20-1 USB 的接口类型

图中 USB 系统架构模型的分层思想与其他通信系统是很类似的。对主机端和 USB 设备端的通信结构分为三层：PC 主机端的客户端软件（Client SW）与 USB 的应用程序（Function）属于功能层，USB 系统软件（USB System SW）与 USB 逻辑设备（USB Logical Device）属于设备层，USB 主机控制器（USB Host Controller）与 USB 设备总线接口（USB Bus Interface）属于总线接口层。实际的数据通信都通过总线接口层传输，软件封装使得主机、设备在设备层、功能层的通信可以忽略下层对数据包的处理，从而感觉主机与设备的通信是在同一个层次的，这样处理也更为简单、可扩展性强。

总线接口层主要负责包括 NRZI 编码、传输速率等，为功能层提供 USB 通信中的端点、通道、缓冲区等物理支持。它可以理解为物理层。

设备层主要负责解释各种通信包，进行各种 USB 事务管理，针对不同的情况按命令状态机进行状态切换等。它可以理解为逻辑层。

功能层主要就是根据不同的应用而开发的，需要主机的客户端软件进行配合。它可以理解为应用层。

2. USB 主机与设备

USB 主机与设备的连接、通信方式是 USB 逻辑通信结构的基础。在 USB 的通信模型中要严格区分 USB 主机（Host）与 USB 设备（Device），因为在 USB 的通信中，有且只有一个主机。当检测到有 USB 设备接入时，首先由主机发起第一次通信。

以 PC 中的 USB 通信模型为例，见图 20-4。PC 作为 USB 通信中的主机，它拥有 USB 主控制器，在内部它与 USB 根集线器（Root hub）相连，向外给 USB 设备提供 USB 插座。USB 设备包括 USB 鼠标、键盘、音响等，还有用于扩展 USB 接口的特殊 USB 设备——USB 集线器（USB hub），如常见的“一拖五”。一个 USB 主控制器同时最多可支持 127 个设备，这是由于 USB 协议中规定 USB 设备接入后，主控制器会给它分配一个 7 位的设备地址，以后就使用该地址进行通信，所以使用 USB 集线器能扩展接口，但无法扩展主控制器可支持设备数目的上限。

大部分 USB 控制器只支持 USB 设备的功能，如 STM32F103 系列芯片的 USB 控制器外设仅可以作为 USB 设备的控制器，而不能用于 USB 主机控制器。在 STM32F105、STM32F107 系列芯片中的 USB 控制器称为“USB OTG 全速控制器（OTG FS）”，它带有 OTG（on the go）功能。OTG 是 USB 2.0 协议中规定的双重角色功能，这种控制器不仅可作为 USB 的主机端，也可以作为 USB 通信中的设备端，这在手机、平板电脑中应用十分广泛。在这些设备中使用的一般为 USB MINI 接口，虽然是 MINI，但它比 A、B 型 USB 接串口多出了一条用于区别主机和设备的 ID 线。

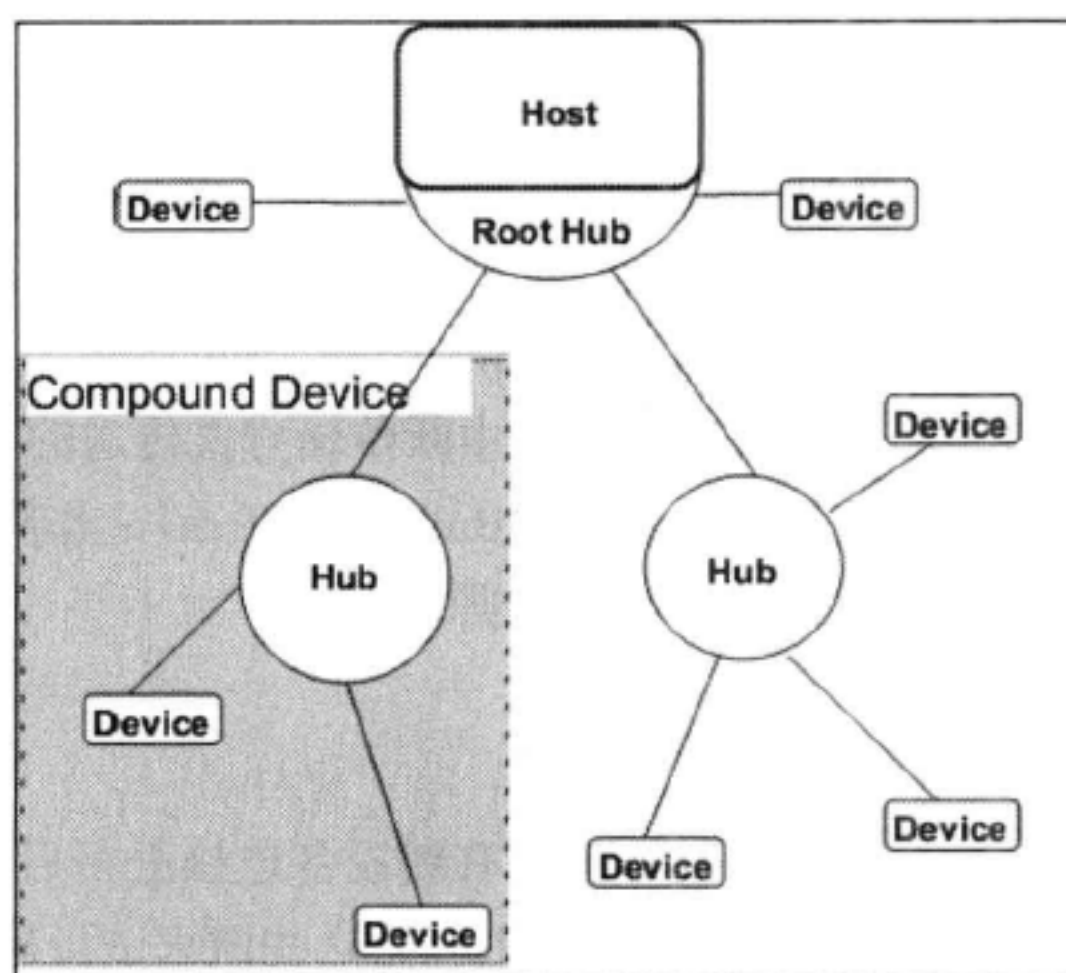


图 20-4 USB 主机、设备连接模型

3. USB 端点

在 USB 通信模型中，端点 (Endpoints) 是一个非常重要的概念，见图 20-5。

在一个 USB 接口 (Interface) 中，含有多个端点，它是 USB 主机与设备通信的节点。每个端点都有特定的编号、传输方向、最大数据包长度等属性。在主机和设备上的两个端点构成了一个数据通道 (Pipe)，数据就在这通道间转移。通常在特定的端点中 (即对应的通道)，传输指定的数据。如端点 0 是特殊端点，默认用于接收和发送初始化控制信息，利用它在通信双方进行基本的协调，之后才能使用其他端点进行通信。在 USB 存储器应用中，端点 0 用于初始化，建立 USB 通信，USB 主机的端点 1 用于批量接收数据，端点 2 用于批量输出数据。

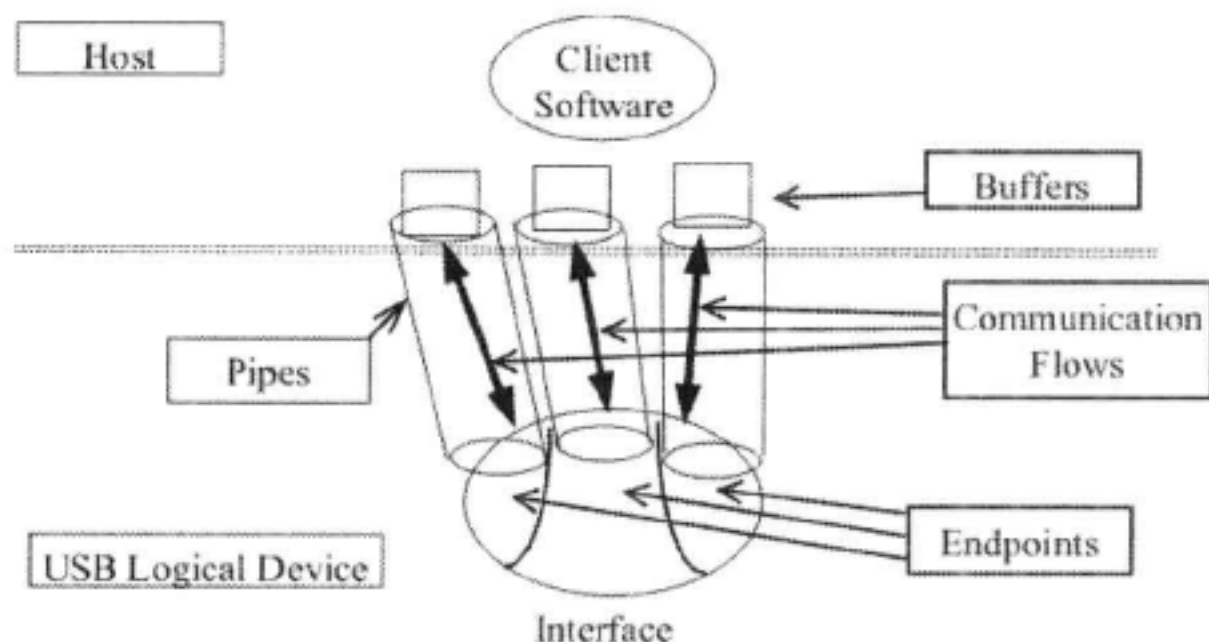


图 20-5 USB 端点、通道

通道、接口的关系可以与高速公路的收费站进行类比，一个收费站 (接口) 拥有多个收费通道，要注意 USB 虽然划分了多个端点和通道，但由于只有一组差分数据线，所以任一时刻至多只传输一个通道的数据 (命令)。

4. USB 传输类型

USB 协议定义了四种传输类型，它用于描述端点和通道的传输特性，就如同公路规定的大车道、小车道一样，这样规定有利于管理和提高效率。

- ❑ 控制传输类型 (Control Transfers)：突发、非周期的传输，主要用于传输命令和状态消息。端点 0 一般就被配置为控制传输。
- ❑ 同步传输类型 (Isochronous Transfers)：周期性、持续的传输，常用于与时间相关的事件、信息的传送。如音频设备中的音频数据。
- ❑ 中断传输类型 (Interrupt Transfers)：传输频率低、数据量小的传输。如鼠标、键盘设备的控制信息就是以中断传输方式传送的。
- ❑ 批量传输类型 (Bulk Transfers)：非周期性的、大数据量的传输。如 USB 存储器主要使用这种方式传输数据。

20.1.4 USB 枚举

当 USB 主机检测到有新设备连接进来后，主机将利用端点 0，以控制传输的方式向设备发送各种请求，USB 设备对主机的这些请求进行应答，以使主机识别该设备，这个识别过程称为枚举。枚举成功，才能建立起正式的 USB 通信。枚举过程简介如下：

主机需要知道接入的是什么设备、使用的协议版本、端点 0 的最大数据包长度等信息，这些信息由 USB 协议规定的设备描述符来进行记录。它是一串数据，在程序中可以用结构体来表示。

协议具体规定哪几个字节表示设备使用的 USB 协议版本、哪几个字节表示端点 0 的最大数据包长度等等。记录设备其他的信息还有配置描述符、接口描述符、端点描述符等。

1) 为了获得设备描述符, 主机首先使用地址 0, 向接入的设备发送 USB 标准请求: Get_Device_Descriptor (获取设备描述符)。正常时, 设备会给主机返回它自己的设备描述符, 但由于第一次通信不知道端点 0 支持数据包的最大长度, 所以主机只能通过设备描述符中的第 8 个字节了解设备端点 0 的最大数据包长度, 这 8 个字节以外的信息还没法了解。

2) 主机为设备分配一个新地址, 把这个地址存放到标准请求 Set_Address (设置地址) 中, 发送这个请求给设备, 设备保存该地址, 以后的通信就使用这个新地址。

3) 主机重新向设备发送 Get_Device_Descriptor (获取设备描述符) 请求, 这次主机会完全读取设备返回的设备描述符, 了解设备的信息。

4) 主机向循环设备发送 Get_Device_Configuration (获取配置描述符) 请求, 获得设备的配置描述符、接口描述符、类特殊描述符、端点描述符。

5) 主机发送 Get_Device_String (描述字符集) 获取厂商 ID、产品描述、型号等信息。

6) 若 USB 能提供该设备的驱动, 主机向设备发送 Set_Configuration (选择设备配置) 请求设备进入某个配置状态。

7) 建立通信。

20.2 STM32 的 USB 控制器

STM32F103 系列的 USB 控制器外设 (不含 OTG 功能) 可用于制作 USB 设备。其主要特性如下:

- 1) 符合 USB 2.0 全速设备的技术规范, 即速度为 12 Mb/s。
- 2) 可配置 1 到 8 个 USB 端点, 其中的 8 个端点均为双向端点, 若配置的都是单向端点, 则可配置 16 个。
- 3) CRC (循环冗余校验) 生成 / 校验, 反向不归零 (NRZI) 编码 / 解码和位填充。
- 4) 支持同步传输。
- 5) 支持批量 / 同步端点的双缓冲区机制, 为实现 USB 存储器提供了良好的支持。
- 6) 支持 USB 挂起 / 恢复操作, 这有利于降低功耗。

见图 20-6, 这是 STM32F103 系列的 USB 控制器的结构图。它主要具有如下结构。

USB 收发器: 整个控制器通过 DP (D+)、DM (D-) 与 USB 收发器相连, USB 收发器的作用跟 CAN 的收发器作用是类似的, 主要负责一些协议相关的数模转换工作和设备接入检测。

串行接口控制器 (SIE): 主要负责 USB 帧的同步域识别、位填充、CRC 校验等。根据 USB 事件、传输结束、正确接收或端点相关的事件产生各种信号, 这些信号可用于触发 USB 中断事件。

注意: 图 20-6 表示的 USB 模块分为两个区域。SIE 部件、数据包缓冲器接口、Suspend 定时器工作于 USB 48 MHz 的时钟下, 它由 PLLCLK 经 USB 预分频器得到, 其他部分的时钟则由 PCLK1 提供。

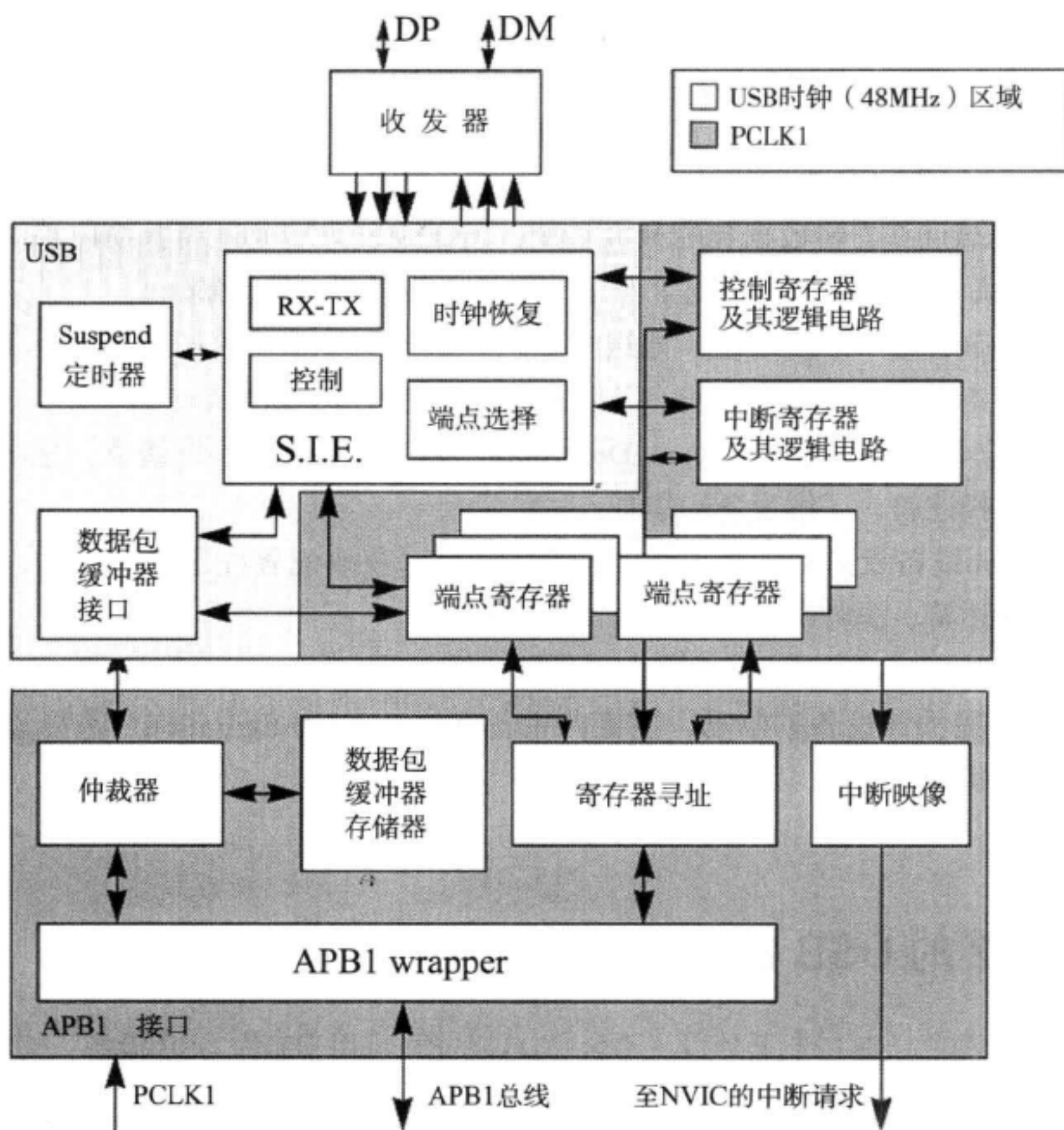


图 20-6 STM32 USB 控制器结构

通过配置和读取与 SIE 交互的控制寄存器、中断寄存器、端点寄存器，我们就可以利用 USB 外设进行 USB 通信了。

USB 控制器还为 PC 主机和微控制器所实现的功能之间提供了符合 USB 规范的通信连接。PC 主机和微控制器之间的数据传输是通过共享一个专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512 字节缓冲区，最多可用于 16 个单向或 8 个双向端点。

开发 STM32 的 USB 驱动程序，主要是利用不同的中断信号，通过读状态寄存器确定哪个端点需要得到服务、正在进行哪种类型的数据传输。

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

20.3 USB 读取 SD 卡——模拟 U 盘实验

20.3.1 实验描述及工程文件清单

1. 实验描述

这是一个 USB Mass Storage (USB 存储器) 实验, 用 USB 线连接 PC 与开发板, 在 PC 上就可以像操作普通 U 盘那样来操作开发板中的 MicroSD 卡, 并在超级终端中打印出相应的调试信息。在这里我们用的 MicroSD 卡的容量是 1GB 的, 4GB 的 SD 卡也已通过测试。

2. 硬件连接

- ☐ PE3-USB-MODE (PE3 为普通 I/O)
- ☐ PA11-USBDM (D-)
- ☐ PA12-USBDP (D+)

3. 库文件

使用 3.5 版本固件库:

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_dma.c
- ☐ FWlib/stm32f10x_sdio.c
- ☐ FWlib/misc.c

4. USB 库

- ☐ usb_core.c
- ☐ usb_init.c
- ☐ usb_mem.c
- ☐ usb_regs.c
- ☐ usb_desc.c
- ☐ usb_endp.c
- ☐ usb_bot.c
- ☐ usb_scsi.c
- ☐ scsi_data.c

5. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c

径, 并添加 USB 库的源文件: usb_core.c、usb_init.c、usb_int.c、usb_mem.c、usb_regs.c、usb_sil.c。其中 USB 固件库文件名中有的带“OTG”字符, 这些文件只在 STM32F105 或 STM32F107 系列的芯片中才用到, 在本实验中不需要添加。

USB 固件库中还有一些 STM32 官方提供的 USB 应用例程, 我们可以利用它来进行开发, 这些例程在它的 project 文件夹下。

打开下载的 USB 库目录: STM32_USB-FS-stm32_usb-fs-device_lib\STM32_USB-FS-Device_Lib_V3.4.0\Project\Mass_Storage, 这是 USB Mass Storage 例程, 找到该目录下的 src 和 inc 文件夹, 复制这些文件到自己的工程, 删除重复文件, 最终添加了: hw_config.c、mass_mal.c、memory.c、scsi_data.c、usb_bot.c、usb_desc.c、usb_endp.c、usb_istr.c、usb_prop.c、usb_pwr.c、usb_scsi.c 文件 (别忘记添加头文件及其路径)。

完成后的 USB 工程目录见图 20-8。

最后在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉即可 (USB 的固件库不在 stm32f10x_conf.h 文件的管理范围)。见代码清单 20-1。

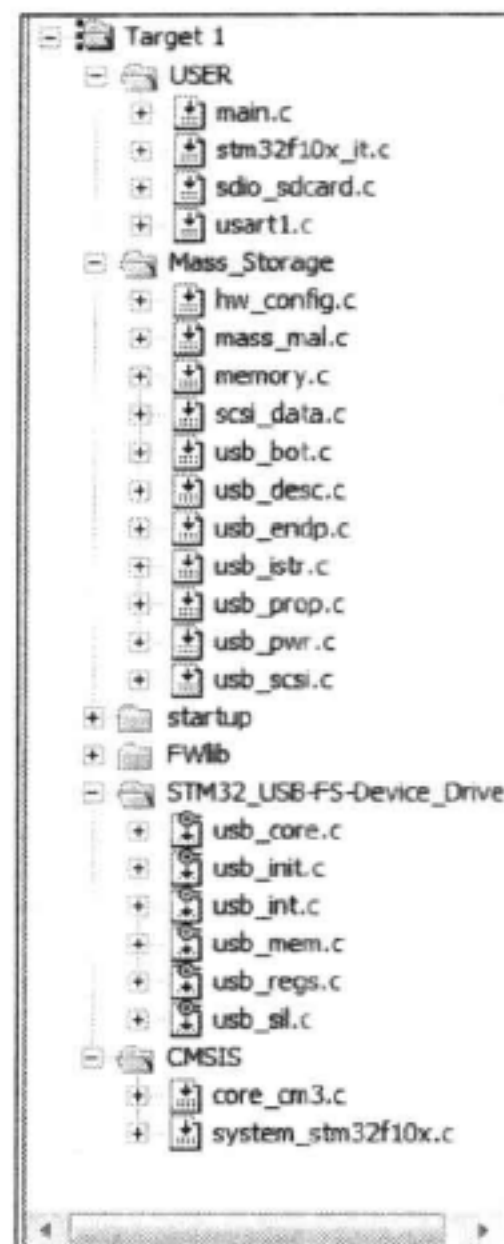


图 20-8 USB 工程目录

代码清单 20-1 USB 例程的 stm32f10x_conf.h 文件配置

```

1. /*****
2.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
3.  * @author  MCD Application Team
4.  * @version V3.5.0
5.  * @date    08-April-2011
6.  * @brief   Library configuration file.
7.  *****/
8. #include "stm32f10x_dma.h"
9. #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_sdio.h"
12. #include "stm32f10x_usart.h"
13. #include "misc.h"

```

20.3.3 USB 固件库说明

USB 固件库包含非常多的文件, 它们的关系见图 20-9。

该图清晰地说明了各文件所处的位置及作用。它把这些文件按三层整理如下:

□ 最底层

1) usb_int.c, 本文件包含两个中断服务函数 CTR_LP() 和 CTR_HP(), 在接收发送中断时使用这两个函数进行处理, 这两个函数又调用各个 USB 端点的处理函数, 端点处理函数由用户自定义。

2) usb_regs.c, 本文件是硬件抽象层。它含有各种用于读取或设置 USB 寄存器的函数, 相当于对寄存器操作方法的封装。

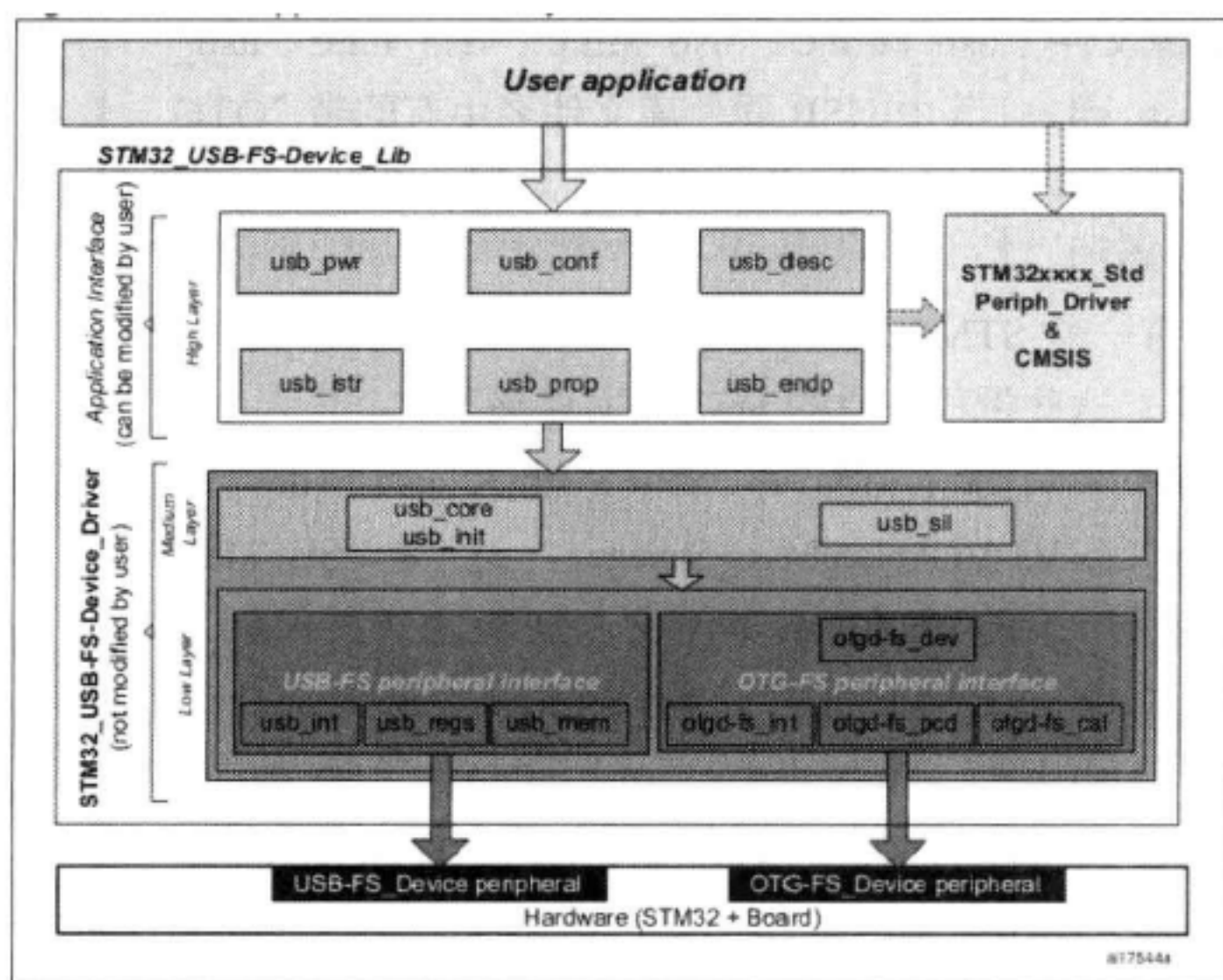


图 20-9 USB 固件库文件关系

3) `usb_mem.c`，本文件主要负责进行数据传输处理。它主要负责数据在用户区域（内存）或 USB 数据包缓冲区（参考图 20-6）的互相转移。

□ 中间层

- 1) `usb_core.c`，本文件主要负责 USB 协议管理，如 USB 标准请求、各种端点 0 的控制信息处理。
- 2) `usb_init.c`，本文件主要包含了 USB 初始化的函数。
- 3) `usb_sil.c`，本文件包含精简接口的初始化或向端点读写的操作函数。

□ 上层

- 1) `usb_pwr.c`，该文件中包含处理上电、调电、挂起和恢复事件的函数。
- 2) `usb_istr.c`，该文件中只有一个函数，即 USB 中断的 `USB_Istr` 函数，该函数对各类引起 USB 中断的事件做轮询处理。
- 3) `hw_config.c` (`usb_conf.c`)，该文件中包含系统配置的函数，包含基本的时钟配置、中断配置和存储器初始化函数。
- 4) `usb_desc.c`，该文件包含一些与 USB 相关的设备描述符、配置描述符等，以数组形式存储，在 USB 主机请求的时候这些信息将发送给主机。
- 5) `usb_prop.c`，该文件用于实现相关设备的 USB 协议，如初始化、SETUP 包、IN 包、OUT 包等。
- 6) `usb_endp.c`，本文件包含端点收、发送的处理函数，由用户根据不同的 USB 设备进行不同的定义。

USB 的应用软件在这三层之上，在不同的 USB 应用中，用户可能需要对 USB 固件库的上层部分文件配置进行修改（主要为 `usb_prop.c` 和 `usb_endp.c` 文件）。而中间层和底层封装得非常好，对这些文件用户是不需要进行改动的。由于本实验的文件较多，采用 Source Insight 软件来分析代码有利于帮助读者理清各函数的调用关系。

20.3.4 main 文件

在配置好我们需要用到的库文件之后，我们就从 main 函数开始分析。见代码清单 20-2。

代码清单 20-2 USB 例程的 main 函数

```

1. /* Includes -----*/
2. #include "stm32f10x.h"
3. #include "sdio_sdcard.h"
4. #include "usart1.h"
5. #include "hw_config.h"
6. #include "usb_lib.h"
7. #include "usb_pwr.h"
8.
9.
10.
11.
12. int main(void)
13. {
14.
15.     /* 进入到 main 函数前，启动文件 startup(stm32f10x_xx.s) 已经调用了在 system_
        stm32f10x.c 中的 SystemInit()，配置好了系统时钟，在外部晶振 8MHz 的条件下，设置 HCLK = 72MHz */
16.
17.
18.     /* USART1 config */
19.     USART1_Config();
20.
21.     /* 初始化 SD 卡 */
22.     Set_System();
23.
24.     /* 设置 USB 时钟为 48MHz */
25.     Set_USBClock();
26.
27.     /* 配置 USB 中断 (包括 SDIO 中断) */
28.     USB_Interrupts_Config();
29.
30.     /* USB 初始化 */
31.     USB_Init();
32.
33.     while (bDeviceState != CONFIGURED); // 等待配置完成
34.
35.     printf("野火 STM32 USB MASS STORAGE 实验 \r\n");
36.
37.
38.     while (1)
39.     {}
40. }

```

在 main 函数中，调用 USART1_Config() 函数初始化调试用的串口，调用 Set_System() 函数初始化 SD 卡。接下来调用 Set_USBClock()、USB_Interrupts_Config()、USB_Init() 分别用于设置 USB 时钟、配置 USB 的接收和发送中断及 SDIO 中断、对 USB 进行初始化。

在 main 函数初始化完成后，剩下任务都在中断服务函数中完成了，当使用 USB 线把开发板与 PC 连接起来后，作为 USB 主机的 PC 就会开始发送请求给作为 USB 设备的开发板，触发它的中断服务函数，开始进行 USB 数据传输。

20.3.5 基本配置

1. SDIO 初始化

Set_System() 主要用于初始化系统中各个外设，在本实验中把它简化成只初始化 SD 卡，Set_System() 经过层层函数调用，最终调用了 MAL_Init() 函数，见代码清单 20-3。

代码清单 20-3 MAL_Init() 函数

```

1.  /*****
2.  * Function Name   : MAL_Init
3.  * Description     : Initializes the Media on the STM32
4.  * Input           : None
5.  * Output          : None
6.  * Return          : None
7.  *****/
8.  uint16_t MAL_Init(uint8_t lun)
9.  {
10.     uint16_t status = MAL_OK;
11.
12.     switch (lun)
13.     {
14.         case 0:
15.             Status = SD_Init();
16.             break;
17.         default:
18.             return MAL_FAIL;
19.     }
20.     return status;
21. }
```

当输入参数 $lun = 0$ 时，本函数调用 SD_Init() 进行 SD 卡的初始化，在原版官方的本函数中还有当 $lun = 1$ 时，进行 NAND FLASH 的初始化，但这需要硬件提供支持。从前面的例程知道调用 SD_Init() 之后，我们就可以进行 SD 卡的读写了。本实验中的 STM32 不需要文件系统，这是因为在 USB Mass Storage 的应用中，文件系统的支持是由 USB 主机提供的，即 PC。

2. USB 时钟

接下来，在 main 函数调用 Set_USBClock() 函数，定义见代码清单 20-4。

代码清单 20-4 Set_USBClock() 函数

```

1.  /*****
2.  * Function Name   : Set_USBClock
3.  * Description     : Configures USB Clock input (48MHz)
4.  * Input           : None.
5.  * Return          : None.
6.  *****/
7.  void Set_USBClock(void)
8.  {
9.
10.     RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);
11. }
```

```

12.  /* Enable the USB clock */
13.  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
14.
15.}

```

这个函数用于配置 USB 的时钟，调用库函数 `RCC_USBCLKConfig()` 把 USB 的时钟预分频设置为 1.5，USB 的时钟是由 PLLCLK 分频得来的，我们实验配置的 PLLCLK 为 72 MHz，1.5 分频后正好为 48 MHz 作为 USB 外设的时钟。

调用 `RCC_APB1PeriphClockCmd()` 函数使能 USB 外设的时钟，这时 USB 外设就运行起来了，不需要再配置 USB 外设使用到的引脚模式，在《STM32 参考手册》GPIO 复用功能配置小节已说明，见表 20-1。

表 20-1 USB 复用引脚配置

USB引脚	GPIO配置
USB_DM / USB_DP	一旦使能了 USB 模块，这些引脚会自动连接到内部 USB 收发器

3. 中断

在 main 中调用用于配置中断的 `USB_Interrupts_Config()` 函数，定义见代码清单 20-5。

代码清单 20-5 USB_Interrupts_Config() 函数

```

1.  /*****
2.  * Function Name : USB_Interrupts_Config
3.  * Description : Configures the USB interrupts
4.  * Input : None.
5.  * Return : None.
6.  *****/
7.  void USB_Interrupts_Config(void)
8.  {
9.      NVIC_InitTypeDef NVIC_InitStructure;
10.
11.      NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
12.
13.
14.      NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
15.      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
16.      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
17.      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
18.      NVIC_Init(&NVIC_InitStructure);
19.
20.      NVIC_InitStructure.NVIC_IRQChannel = USB_HP_CAN1_TX_IRQn;
21.      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
22.      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
23.      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
24.      NVIC_Init(&NVIC_InitStructure);
25.
26.
27.
28.      NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn;

```

```

29. NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
30. NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
31. NVIC_Init(&NVIC_InitStructure);
32.
33.
34.
35.}

```

这个函数配置了 SDIO 及 USB 的低等级 (LP) 和高等级 (HP) 中断的优先级，低优先级一般在控制、中断传输模式下使用，而高优先级中断在快速传输（同步、批量传输）中使用。

20.3.6 USB 初始化

在 main 函数中最后调用 USB_Init() 就完成了 USB 的初始化，它的定义见代码清单 20-6。

代码清单 20-6 USB_Init() 函数

```

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

```

本函数中有多个我们不认识的结构体指针，它们作为全局变量，定义见代码清单 20-7。

代码清单 20-7 结构体指针定义

```

1. DEVICE_INFO *pInformation;
2. DEVICE_PROP *pProperty;
3. USER_STANDARD_REQUESTS *pUser_Standard_Requests;

```

这些结构体指针实际上指向各种用户接口的函数（根据不同的应用定义用户），我们需要把自定义的接口函数指针赋值给以上几个变量，以供 usb_core.c 中的函数（可理解为 USB 内核函数）引用，在那些 USB 内核函数中会进行 USB 协议密切相关的处理——这是一种回调函数的机制。

先来分析一下 USB_Init() 中第 12 行使用的结构体指针，它的类型为 DEVICE_PROP，这是一种由 usb_core.h 文件中定义的结构体，它的结构体成员大多为指向函数的指针变量，见代码清单 20-8。

代码清单 20-8 DEVICE_PROP 结构体类型指针

```

1. typedef struct _DEVICE_PROP
2. {

```

```

3.  void (*Init)(void);          /* Initialize the device */
4.  void (*Reset)(void);         /* Reset routine of this device */
5.
6.  void (*Process_Status_IN)(void);
7.  void (*Process_Status_OUT)(void);
8.
9.  RESULT (*Class_Data_Setup)(uint8_t RequestNo);
10. RESULT (*Class_NoData_Setup)(uint8_t RequestNo);
11.
12. RESULT (*Class_Get_Interface_Setting)(uint8_t Interface, uint8_t AlternateSetting);
13.
14. uint8_t* (*GetDeviceDescriptor)(uint16_t Length);
15. uint8_t* (*GetConfigDescriptor)(uint16_t Length);
16. uint8_t* (*GetStringDescriptor)(uint16_t Length);
17. void* RxEP_buffer;
18. uint8_t MaxPacketSize;
19.
20. }DEVICE_PROP;

```

看第一个结构体成员 `void (*Init)(void)`，这是一个指向 USB 设备初始化函数的指针。在应用开始时需要被调用一次，回到 `USB_Init()` 函数中的最后一行：

```
1. pProperty->Init();
```

我们知道 `pProperty` 是一个 `DEVICE_PROP` 类型结构体，它的 `pProperty->Init()` 表示要引用它的结构体成员 `Init`，而这个成员是一个初始化函数指针，也就是说它以这样的方式调用了这个初始化函数，完成了 USB 设备的初始化，但是这个指针指向的函数具体是谁呢？回头看 `USB_Init()` 中第 12 行：

```
1. pProperty = &Device_Property;
```

这个语句使 `pProperty` 指向同类型的结构体变量 `Device_Property`，而这个结构体在 `usb_prop.c` 文件中赋值，见代码清单 20-9。

代码清单 20-9 结构体变量 `Device_Property` 赋值

```

1. DEVICE_PROP Device_Property =
2.  {
3.      MASS_init,
4.      MASS_Reset,
5.      MASS_Status_In,
6.      MASS_Status_Out,
7.      MASS_Data_Setup,
8.      MASS_NoData_Setup,
9.      MASS_Get_Interface_Setting,
10.     MASS_GetDeviceDescriptor,
11.     MASS_GetConfigDescriptor,
12.     MASS_GetStringDescriptor,
13.     0,
14.     0x40 /*MAX PACKET SIZE*/
15. };

```

`Device_Property` 的第一个结构体成员即是初始化函数指针，它的函数名为 `MASS_init`，具体

在 usb_prop.c 文件定义，见代码清单 20-10。

代码清单 20-10 MASS_init 函数

```

1. void MASS_init()
2. {
3.     /* Update the serial number string descriptor with the data from the unique
4.     ID*/
5.     Get_SerialNum();
6.
7.     pInformation->Current_Configuration = 0;
8.
9.     /* Connect the device */
10.    PowerOn();
11.
12.    /* Perform basic device initialization operations */
13.    USB_SIL_Init();
14.
15.    bDeviceState = UNCONNECTED;
16.}

```

就是这个函数，完成了 USB 的初始化，它调用了 Get_SerialNum() 获取 USB 字符串描述符、调用 PowerOn() 给 USB 外设上电，调用 USB_SIL_Init() 进行基本初始化，提供基本的 USB 设备操作。

在 USB_Init() 函数中的 pUser_Standard_Requests 指针指向关于 USB 标准请求的处理函数；在 DEVICE_PROP 结构体中也有一些函数用于向主机返回设备描述符、配置描述符，由读者自行分析。

20.3.7 中断服务函数

STM32 USB 控制器的中断分为高优先级中断和低优先级中断，前面已介绍过它们的区别，由于低优先级中断包含控制传输（主要为端点 0）的处理，比较复杂，这里我们使用高优先级中断服务函数进行分析，它在 stm32f10x_it.c 文件定义，见代码清单 20-11。

代码清单 20-11 USB 中断服务程序

```

1. /*
2.  * 函数名：USB_HP_CAN1_TX_IRQHandler
3.  * 描述   ：USB 高优先级中断请求
4.  * 输入    ：无
5.  * 输出    ：无
6.  */
7. void USB_HP_CAN1_TX_IRQHandler(void)
8. {
9.     CTR_HP();
10.}

```

调用的函数 CTR_HP() 见代码清单 20-12。

代码清单 20-12 CTR_HP() 函数

```

1. /*****

```

```

2. * Function Name   : CTR_HP.
3. * Description    : High Priority Endpoint Correct Transfer interrupt's service
4. *                : routine.
5. * Input          : None.
6. * Output         : None.
7. * Return         : None.
8. *****/
9. void CTR_HP(void)
10. {
11.     uint32_t wEPVal = 0;
12.
13.     while (((wIstr = _GetISTR()) & ISTR_CTR) != 0)
14.     {
15.         _SetISTR((uint16_t)CLR_CTR); /* clear CTR flag */
16.
17.         /* extract highest priority endpoint number */
18.         EPindex = (uint8_t)(wIstr & ISTR_EP_ID);
19.
20.         /* process related endpoint register */
21.
22.         wEPVal = _GetENDPOINT(EPindex);
23.         if ((wEPVal & EP_CTR_RX) != 0)
24.         {
25.             /* clear int flag */
26.             _ClearEP_CTR_RX(EPindex);
27.
28.             /* call OUT service function */
29.             (*pEpInt_OUT[EPindex-1])();
30.
31.         } /* if((wEPVal & EP_CTR_RX) */
32.         else if ((wEPVal & EP_CTR_TX) != 0)
33.         {
34.             /* clear int flag */
35.             _ClearEP_CTR_TX(EPindex);
36.
37.             /* call IN service function */
38.             (*pEpInt_IN[EPindex-1])();
39.
40.
41.         } /* if((wEPVal & EP_CTR_TX) != 0) */
42.
43.     } /* while(...) */
44. }

```

分析如下：

- 1) 第 13 行，调用 `_GetISTR()` 函数读取 ISTR 寄存器的内容到变量 `wIstr` 中。
- 2) 第 15 行，调用 `_SetISTR()` 函数清除正确传输标志 CTR（正确传输会引起 USB 中断）。
- 3) 第 18 行，读取触发中断的端点号。
- 4) 第 22 行，读取相应端点号的端点寄存器内容，它包含了发送或接收信息。
- 5) 第 23 行，判断端点是否接收到数据，在第 29 行调用回调函数 `(*pEpInt_OUT[EPindex-1])()` 对接收到的数据进行处理。
- 6) 第 32 行，判断端点发送数据是否成功，在第 38 行调用回调函数 `(*pEpInt_IN[EPindex-1])()`

进行处理。

以上两个回调函数究竟是什么呢？这真的是函数吗？它们是在 `usb_istr.c` 文件中定义的，见代码清单 20-13。

代码清单 20-13 pEpInt_IN 和 pEpInt_OUT 回调函数

```

1. /* function pointers to non-control endpoints service routines */
2.
3. void (*pEpInt_IN[7])(void) =
4. {
5.     EP1_IN_Callback,
6.     EP2_IN_Callback,
7.     EP3_IN_Callback,
8.     EP4_IN_Callback,
9.     EP5_IN_Callback,
10.    EP6_IN_Callback,
11.    EP7_IN_Callback,
12. };
13.
14. void (*pEpInt_OUT[7])(void) =
15. {
16.     EP1_OUT_Callback,
17.     EP2_OUT_Callback,
18.     EP3_OUT_Callback,
19.     EP4_OUT_Callback,
20.     EP5_OUT_Callback,
21.     EP6_OUT_Callback,
22.     EP7_OUT_Callback,
23. };

```

查看定义我们发现它们是函数指针数组，听这名字就不简单，这样定义是为了方便根据 USB 端点调用不同的端点处理函数。但在本实验中，并没有使用这么多的 USB 端点，以上函数指针数组中，仅有 `EP1_IN_Callback` 和 `EP2_OUT_Callback` 指针是指向实体函数的，分别被中断服务函数调用来处理 USB 主机输入数据和 USB 主机输出数据时的情况（函数名中的 IN 和 OUT 是针对 USB 主机来说的）。这两个函数定义见代码清单 20-14。

代码清单 20-14 EP1_IN_Callback 和 EP2_OUT_Callback 指针

```

1.
2. void EP1_IN_Callback(void)
3. {
4.     Mass_Storage_In();
5. }
6.
7. void EP2_OUT_Callback(void)
8. {
9.     Mass_Storage_Out();
10.}

```

它们分别调用了 `Mass_Storage_In()` 和 `Mass_Storage_Out()` 函数，理解它们须先了解 BOT 和 SCSI 协议。

20.3.8 BOT 和 SCSI 协议

1. BOT

BOT (Bulk Only Transport, 仅批量传输协议) 是 USB 的子类协议。它只使用 USB 传输模式中的批量传输通道进行命令、状态、数据的传输, 没有中断和控制通道。建立了传输后 (即枚举后), 默认的端点 0 只用于清除批量管道的状态和发送 Mass Storage 复位、Get Max Lun 这两个请求。

在本实验中就只配置了两个端点用于批量传输: 端点 1 用于 USB 主机输入类的传输 (BulkIn), 端点 2 用于 USB 主机输出类的传输 (BulkOut)。

在 BOT 协议中使用 CBW 格式发送命令, 用 CSW 格式发送命令状态 (详细格式说明请参考 BOT 协议书)。

2. SCSI

SCSI 命令集为硬盘、磁带、大容量存储设备提供了高效的点对点操作, 目前大多数 USB 存储设备都使用 SCSI 命令集。在 USB 设备中主要用到的 SCSI 命令如表 20-2。

表 20-2 SCSI 命令集

命令名称	操作码	命令支持 (M=强制的, O=可选的)	命令描述
Inquiry	0x12	M	得到设备信息
Read Format Capacities	0x23	M	报告当前媒介支持的媒介能力和可格式化能力
Mode Sense (6)	0x1A	M	向主机报告参数
Mode Sense (10)		M	向主机报告参数
Prevent\ Allow MediumRemoval	0x1E	M	阻止或者允许从可移动媒介设备的移除媒体
Read (10)	0x28	M	从媒介向主机传输二进制数据
Read Capacity (10)	0x25	M	报告当前媒介支持的媒介能力和可格式化能力
RequestSense	0x03	O	向主机传输状态传感数据
Start Stop Unit	0x1B	M	使能或禁止媒介访问操作和控制特定电源情况的逻辑单元
Test Unit Ready	0x00	M	请求设备报告是否准备好
Verify (10)	0x2F	M	验证媒介上的数据
Write (10)	0x2A	M	从主机向媒介传输二进制数据

我们主要给大家讲解的是 Read(10) 和 Write(10) 命令, 这两个命令分别用于从媒介 (存储器) 向主机传输二进制数据和从主机向媒介传输数据。

3. 传输状态机

以图 20-10 分析, 结合 BOT 协议和 SCSI 协议。当 USB 接口处于准备状态时, 主机使用以

BOT 中的 CBW 格式向设备发送包含 Write (10) 的 SCSI 命令，USB 设备接收到该命令后，使用 CSW 格式向主机说明命令通过，接着主机使用端点 2 BulkOut 传输大量要被写入 USB 存储器的数据（实际上，发送 CBW 时也是使用端点 2）。当要从 USB 存储器读取数据时，该过程类似。

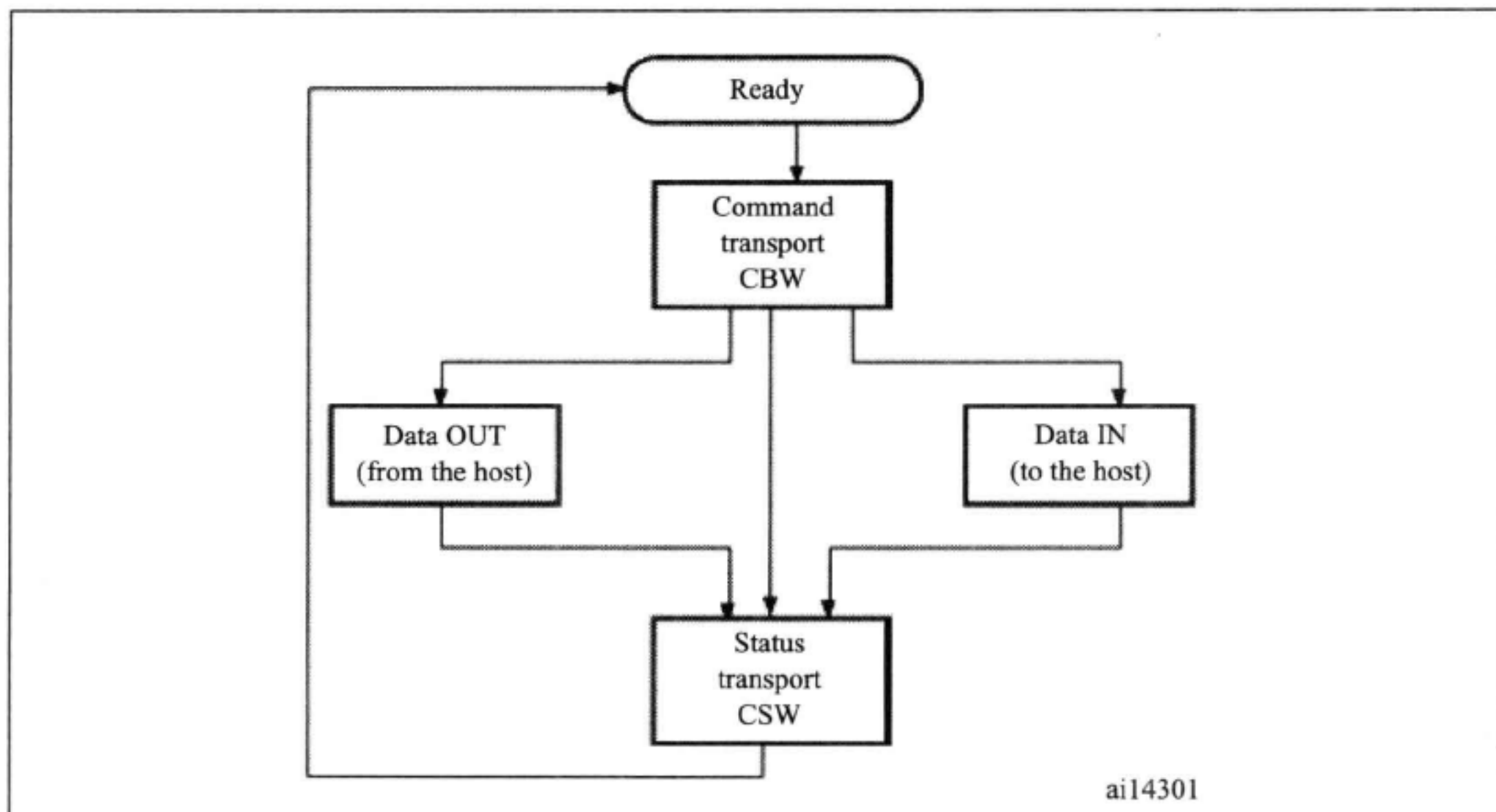


图 20-10 传输状态机

4. 代码实现

以端点 2 调用的 `Mass_Storage_Out()` 函数为例，作为 USB 设备，端点 2 是用于接收数据和命令的。见代码清单 20-15。

代码清单 20-15 `Mass_Storage_Out()` 函数

```

1. /*****
2. * Function Name : Mass_Storage_Out
3. * Description   : Mass Storage OUT transfer.
4. * Input        : None.
5. * Output       : None.
6. * Return       : None.
7. *****/
8. void Mass_Storage_Out (void)
9. {
10.  uint8_t CMD;
11.  CMD = CBW.CB[0];
12.
13.  Data_Len = USB_SIL_Read(EP2_OUT, Bulk_Data_Buff);
14.
15.  switch (Bot_State)
16.  {
17.      case BOT_IDLE:

```

```

18.     CBW_Decode();
19.     break;
20. case BOT_DATA_OUT:
21.     if (CMD == SCSI_WRITE10)
22.     {
23.         SCSI_Write10_Cmd(CBW.bLUN, SCSI_LBA, SCSI_BlklLen);
24.         break;
25.     }
26.     Bot_Abort(DIR_OUT);
27.     Set_Scsi_Sense_Data(CBW.bLUN, ILLEGAL_REQUEST, INVALID_FIELED_IN_COMMAND);
28.     Set_CSW (CSW_PHASE_ERROR, SEND_CSW_DISABLE);
29.     break;
30. default:
31.     Bot_Abort(BOTH_DIR);
32.     Set_Scsi_Sense_Data(CBW.bLUN, ILLEGAL_REQUEST, INVALID_FIELED_IN_COMMAND);
33.     Set_CSW (CSW_PHASE_ERROR, SEND_CSW_DISABLE);
34.     break;
35. }
36.}

```

函数的第 11 行把接收到的 CBW 格式块中的 CB 成员赋值给 CMD 变量, CMD 变量中的内容即为接收到的 SCSI 命令编码。在第 21 行, 判断得 CMD 内容与宏 SCSI_WRITE10 相同时, 表明接收到的是写入命令, 于是调用 SCSI_Write10_Cmd() 函数进行写入数据, 它进行一些处理后调用 Write_Memory() 函数, 再调用 MAL_Write() 函数, 见代码清单 20-16。

代码清单 20-16 MAL_Write() 函数

```

1. uint16_t MAL_Write(uint8_t lun, uint32_t Memory_Offset, uint32_t *Writebuff, uint16_t
   Transfer_Length)
2. {
3.
4.     switch (lun)
5.     {
6.         case 0:
7.             Status = SD_WriteBlock((uint8_t*)Writebuff, Memory_Offset, Transfer_Length);
8.             Status = SD_WaitWriteOperation();
9.             while(SD_GetStatus() != SD_TRANSFER_OK);
10.            if ( Status != SD_OK )
11.            {
12.                return MAL_FAIL;
13.            }
14.            break;
15.        default:
16.            return MAL_FAIL;
17.    }
18.    return MAL_OK;
19.}

```

我们看到在 MAL_Write() 中最终调用了我们熟悉的 SD_WriteBlock() 函数, 进行 SD 卡的块写入, 到这里, 我们终于把 USB 通信与 SD 卡的读写联系起来了。

在本章的讲解中, 舍去了大部分细节, 这些细节需要读者在实践中去理解、研究。

20.3.9 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上 MicroSD 卡，插上方口的 USB 线，将编译好的程序下载到开发板，如果程序运行成功，则可在电脑上看到开发板上的 U 盘（我们用的是 1GB 的卡，4GB 的也已测试通过），如图 20-11。

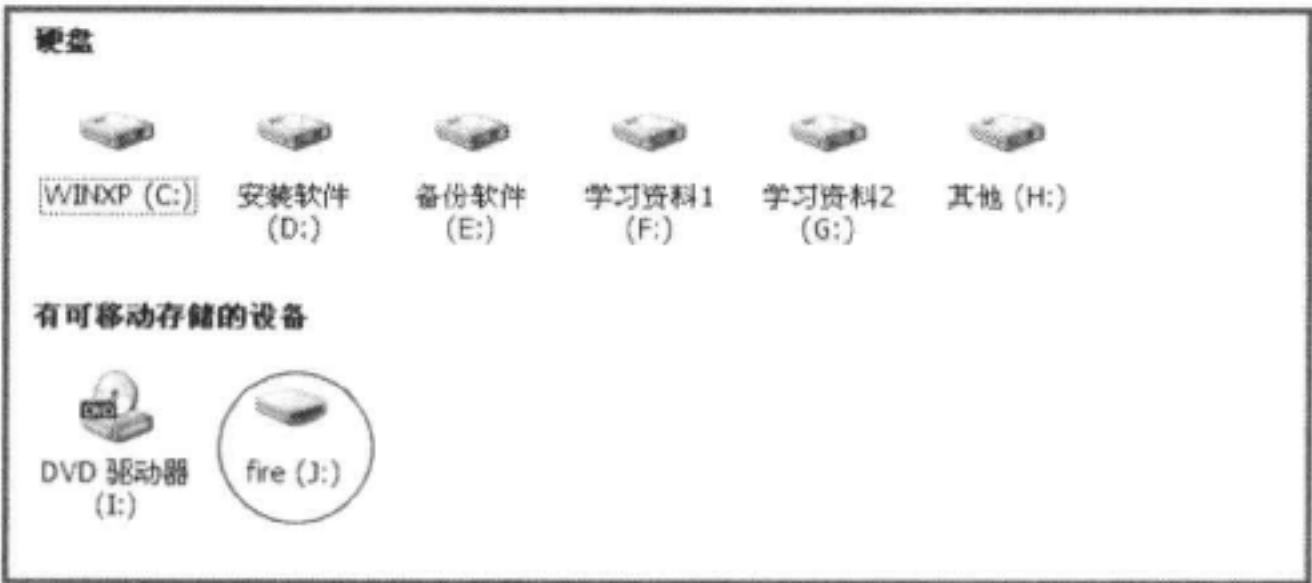
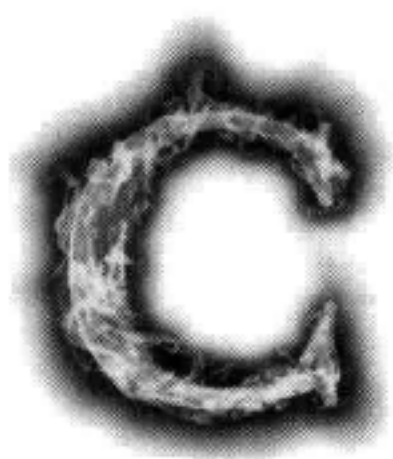


图 20-11 模拟 U 盘实验现象

打开 U 盘，可看到里面的文件，见图 20-12，还可以进行新建、复制、粘贴、格式化等操作，与操作我们的普通 U 盘没什么区别。



图 20-12 查看 SD 卡的内容



第 21 章

LCD 触摸屏画板

本章向大家介绍如何使用 STM32 的 FSMC 接口驱动 LCD 屏，及使用触摸屏控制器检测触点坐标。

21.1 LCD 控制器简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

因为 STM32 内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路（液晶屏和触摸屏的驱动电路是相互独立的），STM32 芯片通过驱动芯片来控制液晶屏和触摸屏。以配套的 3.2 寸液晶屏（ 240×320 ）为例，它使用 ILI9341 芯片控制液晶屏，通过 TSC2046 芯片控制触摸屏。

21.1.1 ILI9341 控制器结构

液晶屏的控制芯片内部结构非常复杂，见图 21-1。最主要的是位于中间的 GRAM（Graphics RAM），可以理解为显存。GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。

框图的左上角为 ILI9341 的主要控制信号线和配置引脚，根据其不同状态设置可以使芯片工作在不同的模式，如每个像素点的位数是 6、16 还是 18 位；使用 SPI 接口还是 8080 接口与 MCU 进行通信；使用 8080 接口的哪种模式。MCU 通过 SPI 或 8080 接口与 ILI9341 进行通信，从而访问它的控制寄存器（CR）、地址计数器（AC）及 GRAM。

在 GRAM 的左侧还有一个 LED 控制器（LED Controller）。LCD 为非发光性的显示装置，它需要借助背光源才能达到显示功能，LED 控制器就是用来控制液晶屏中的 LED 背光源。

21.1.2 像素点的数据格式

图像数据的像素点由红（R）、绿（G）、蓝（B）三原色组成，三原色根据其深浅程度被分为 0 ~ 255 个级别，它们按不同比例的混合可以得出各种色彩。例如，R : 255，G : 255，B : 255 混

合后为白色。根据描述像素点数据的长度，主要分为 8、16、24 及 32 位。如以 8 位来描述的像素点可表示 $2^8=256$ 色，16 位描述的为 $2^{16}=65536$ 色，称为真彩色，也称为 64K 色。实际上受人眼对颜色的识别能力的限制，16 位色与 12 位色已经难以分辨了。

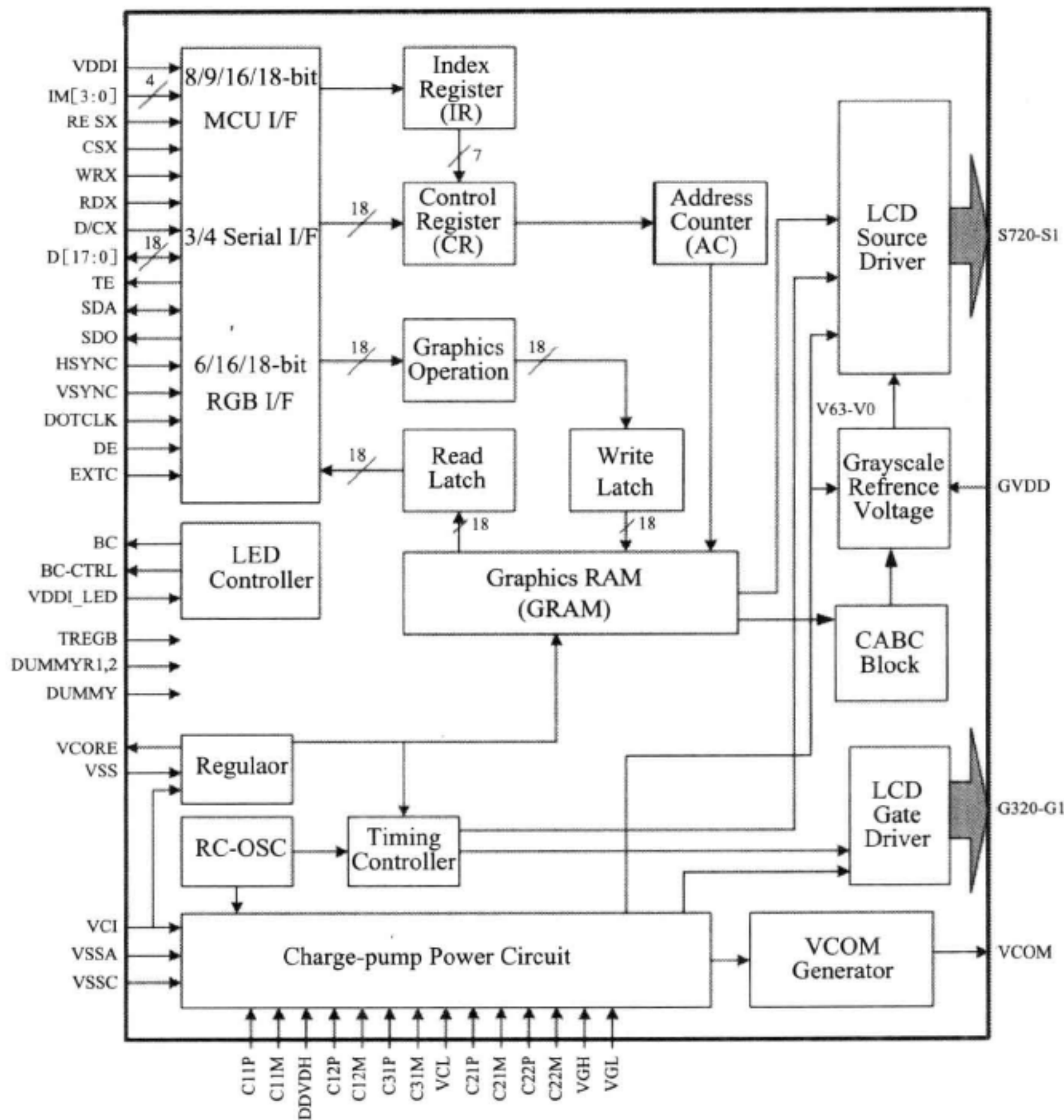


图 21-1 ILI9341 控制器内部框图

ILI9341 最高能够控制 18 位的 LCD，但为了数据传输简便，采用它的 16 位控制模式，以 16 位描述像素点。按照标准格式，16 位像素点的三原色描述的位数为 R : G : B = 5 : 6 : 5，描述绿色的位数较多是因为人眼对绿色更为敏感。16 位的像素点格式见表 21-1。

表 21-1 16 位像素点格式

D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R[4]	R[3]	R[2]	R[1]	R[0]		G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	B[4]	B[3]	B[2]	B[1]	B[0]	

表 21-1 中默认 18 条数据线时，像素点三原色的分配状况：D1 ~ D5 为蓝色，D6 ~ D11 为绿色，D13 ~ D17 为红色。这样分配 D0 和 D12 位是无效的。若使用 16 根数据线传送像素点的数据，则 D0 ~ D4 为蓝色，D5 ~ D10 为绿色，D11 ~ D15 为红色，使得刚好使用完整的 16 位。

RGB 比例为 5 : 6 : 5 是一个十分通用的颜色标准，在 GRAM 相应的地址中填入该颜色的编码，即可控制 LCD 输出该颜色的像素点。如黑色的编码为 0x0000，白色的编码为 0xffff，红色为 0xf800。

21.1.3 ILI9341 的通信时序

目前，大多数的液晶控制器都使用 8080 或 6800 接口与 MCU 进行通信，它们的时序十分相似，我们以 ILI9341 使用的 8080 通信时序进行分析，实际上 ILI9341 也可以使用 SPI 接口来控制。

ILI9341 的 8080 接口有 5 条基本的控制信号线：

- 1) 用于片选的 CSX 信号线。
- 2) 用于写使能的 WRX 信号线。
- 3) 用于读使能的 RDX 信号线。
- 4) 用于区分数据和命令的 D/CX 信号线。
- 5) 用于复位的 RESX 信号线。

其中带 X 的表示低电平有效。除了控制信号，还有数据信号线，它的数目不定，可根据 ILI9341 框图中的 IM[3:0] 来设定，这部分一般由制作液晶屏的厂家完成。为便于传输像素点数据，我们将液晶屏设定为 16 条数据线 D[15:0]。使用 8080 接口的写命令时序图见图 21-2。

由图 21-2 可知，写命令时序由 CSX 信号线拉低开始，D/CX 信号线也置低电平表示写入的是命令地址（可理解为命令编码，如软件复位命令 0x01），以 WRX 信号线为低、RDX 信号线为高表示数据传输方向为写入，同时，在数据线 [17:0] 输出命令地址，在第二个传输阶段传送的为命令的参数，所以 D/CX 要置高电平，表示写入的是命令数据。

当我们需要向 GRAM 写入数据的时候，把 CSX 信号线拉低后，把 D/CX 信号线置为高电平，这时由 D[17:0] 传输的数据则会被 ILI9341 保存至它的 GRAM 中。

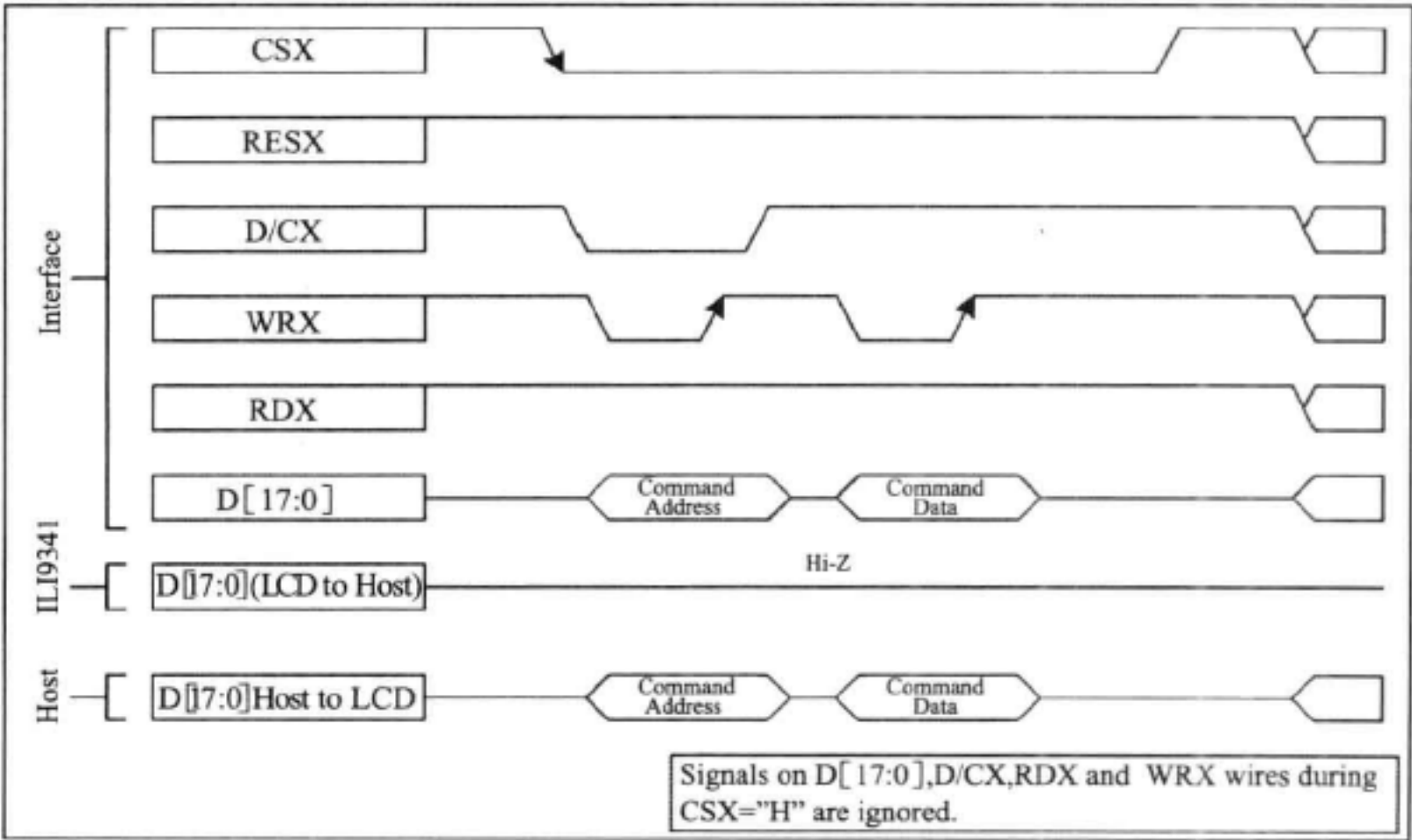


图 21-2 使用 18 条数据线的 8080 接口写命令时序

21.2 用 STM32 驱动 LCD

ILI9341 的 8080 通信接口时序可以由 STM32 使用普通 I/O 接口进行模拟，但这样效率较低，它提供了一种特别的控制方法——使用 FSMC 接口。

21.2.1 FSMC 简介

FSMC (Flexible Static Memory Controller, 静态存储控制器) 可用于 STM32 芯片控制 NOR FLASH、PSRAM 和 NAND FLASH 存储芯片，其结构见图 21-3。

我们使用 FSMC 的 NOR\PSRAM 模式控制 LCD，所以我们重点分析框图中 NOR FLASH 控制信号线部分。控制 NOR FLASH 主要使用到表 21-2 的信号线。

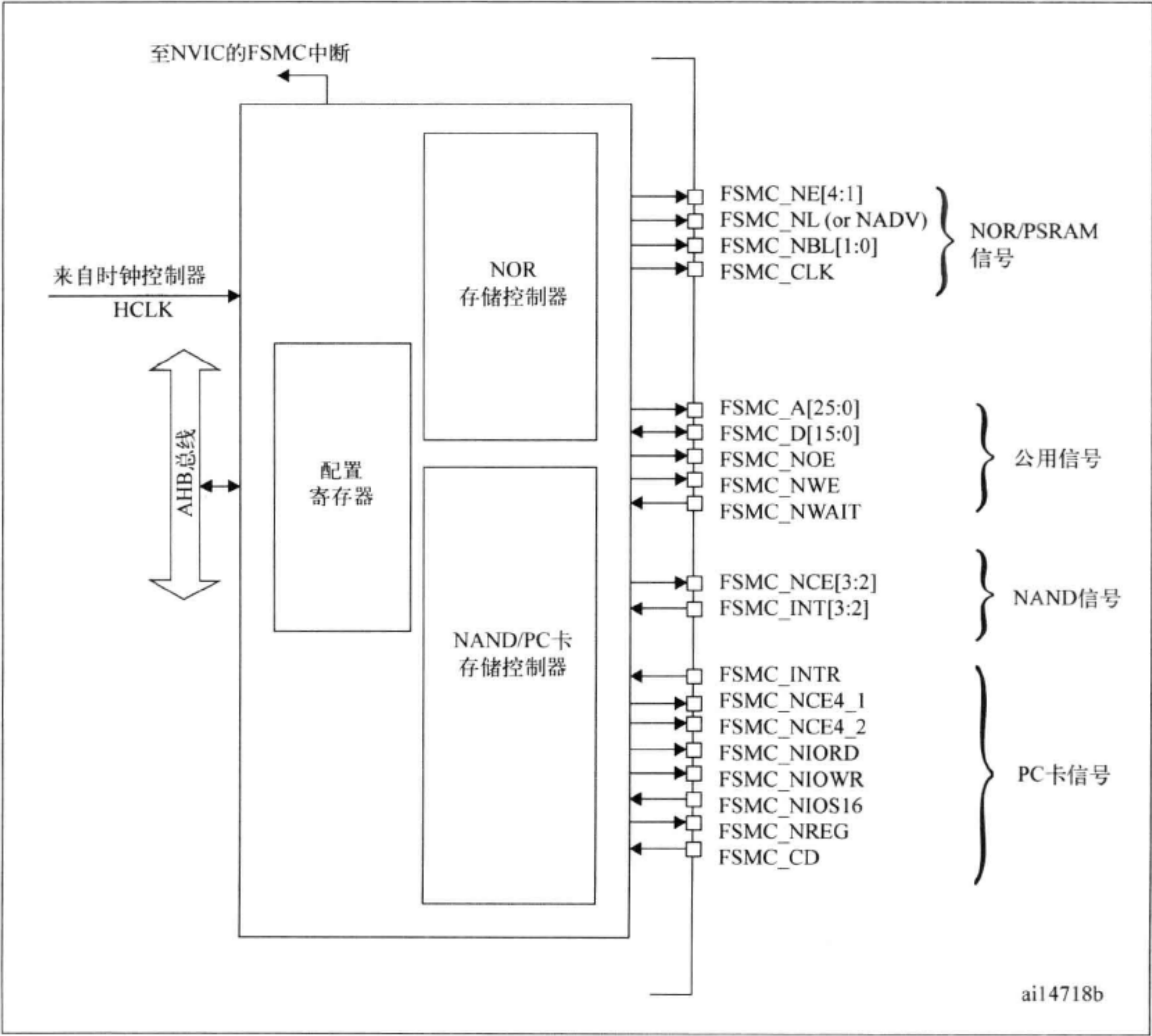


图 21-3 FSMC 结构图

表 21-2 FSMC 控制 NOR FLASH 的信号线

FSMC信号名称	信号方向	功 能
CLK	输出	时钟（同步突发模式使用）
A[25:0]	输出	地址总线
D[15:0]	输入 / 输出	双向数据总线
NE[x]	输出	片选，x = 1...4
NOE	输出	输出使能
NWE	输出	写使能
NWAIT	输入	NOR 闪存要求 FSMC 等待的信号

根据 STM32 对寻址空间的地址映射，见图 4-5，地址 0x6000 0000 ~ 0x9FFF FFFF 是映射到外部存储器的，而其中的 0x6000 0000 ~ 0x6FFF FFFF 则是分配给 NOR FLASH、PSRAM 这类可直接寻址的器件。当 FSMC 外设被配置为正常工作，并且外部连接了 NOR FLASH，这时若向 0x6000 0000 地址写入数据 0xffff，FSMC 会自动在各信号线上产生相应的电平信号，写入数据。该过程的时序图见图 21-4。

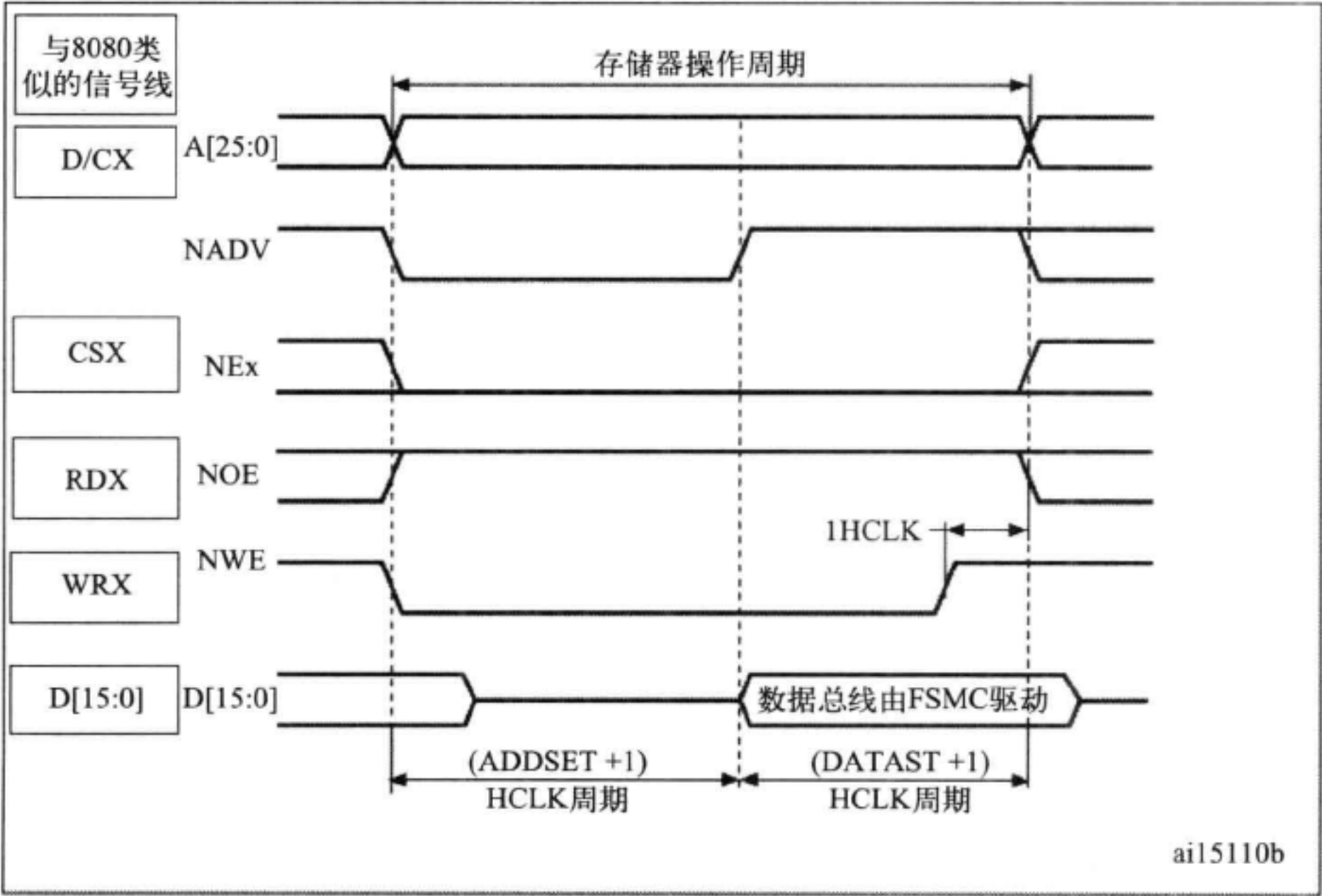


图 21-4 FSMC 写 NOR 时序图

它会控制片选信号 NE[X] 选择相应的某块 NOR 芯片，然后使用地址线 A[25:0] 输出 0x6000 0000，在 NWE 写使能信号线上发出写使能信号，而要写入的数据信号 0xffff 则从数据线 D[15:0] 输出，然后数据就被保存到 NOR FLASH 中了。

21.2.2 用 FSMC 模拟 8080 时序

在图 21-4 的时序图中 NADV 信号是在地址、信号线复用作为锁存信号的，在此我们忽略它。然后读者会发现，这个 FSMC 写 NOR 时序与 8080 接口的时序（见图 21-2）是十分相似的，对它们的信号线对比见表 21-3。

表 21-3 FSMC 的 NOR 与 8080 信号线对比

8080信号线	功 能	FSMC-NOR信号线	功 能
CSX	片选信号	NEx	片选
WRX	写使能	NWR	写使能
RDX	读使能	NOE	读使能
D[15:0]	数据信号	D[15:0]	数据信号
D\CX	数据 / 命令选择	A[25:0]	地址信号

前四种信号线都是完全一样的，仅在 8080 的数据 \ 命令选择线与 FSMC 的地址信号线有区别。为了模拟出 8080 时序，我们把 FSMC 的 A0 地址线（也可以使用其他地址线）连接 8080 的 D\CX，即 A0 为高电平时，数据线 D[15:0] 的信号会被理解为数值，若 A0 为低电平时，传输的信号则会被理解为命令。

也就是说，当向地址为 0x6xxx xxx1、0x6xxx xxx3、0x6xxx xxx5……这些奇数地址写入数据时，地址线 A0（D/CX）会为高电平，这个数据被理解为数值；若向 0x6xxx xxx0、0x6xxx xxx2、0x6xxx xxx4……这些偶数地址写入数据时，地址线 A0（D/CX）会为低电平，这个数据会被理解为命令。

有了这个基础，只要我们在代码中利用指针变量，向不同的地址单元写入数据，就能够由 FSMC 模拟出的 8080 接口向 ILI9341 写入控制命令或 GRAM 的数据了。

21.3 触摸屏感应原理

触摸屏常与液晶屏配套使用，组合成为一个可交互的输入输出系统。除了熟悉的电阻、电容屏外，触摸屏的种类还有超声波屏、红外屏。由于电阻屏的控制系统简单、成本低，且能适应各种恶劣环境，因此被广泛采用。

电阻触摸屏的基本原理为分压，它由一层或两层阻性材料组成，在检测坐标时，在阻性材料的一端接参考电压 V_{ref} ，另一端接地，形成一个沿坐标方向的均匀电场。当触摸屏受到挤压时，阻性材料与下层电极接触，阻性材料被分为两部分，因而在触摸点的电压反映了触摸点与阻性材料的 V_{ref} 端的距离，而且为线性关系，而该触点的电压可由 ADC 测得。更改电场方向，以同样的方法，可测得另一方向的坐标。

21.4 TSC2046 触摸屏控制器

TSC2046 是专用在四线电阻屏的触摸屏控制器，MCU 可通过 SPI 接口向它写入控制字，由它测得 X、Y 方向的触点电压返回给 MCU。见图 21-5。

图 21-5 中,电阻屏两层阻性材料的两端分别接入 TSC2046 的 X+、X- 和 Y+、Y-。当要测量 X 坐标时,MCU 通过 SPI 接口写命令到 TSC2046,使它通过内部的模拟开关令 X+、X- 接通电源,于是在电阻屏的 X 方向上产生一个匀强电场;把 Y+、Y- 连接到 TSC2046 的 ADC。当电阻屏被触摸时,上、下两层的阻性材料接触,在 PENIRQ 引脚产生一个中断信号,通知 MCU。该触点的电压由 Y+ 或 Y- (此时的 Y+、Y- 电阻很小,可忽略)引入到 ADC 进行测量,MCU 读取该电压,进行软件转换,就可以测得触点 X 方向的坐标。同理可以测得 Y 方向的坐标。

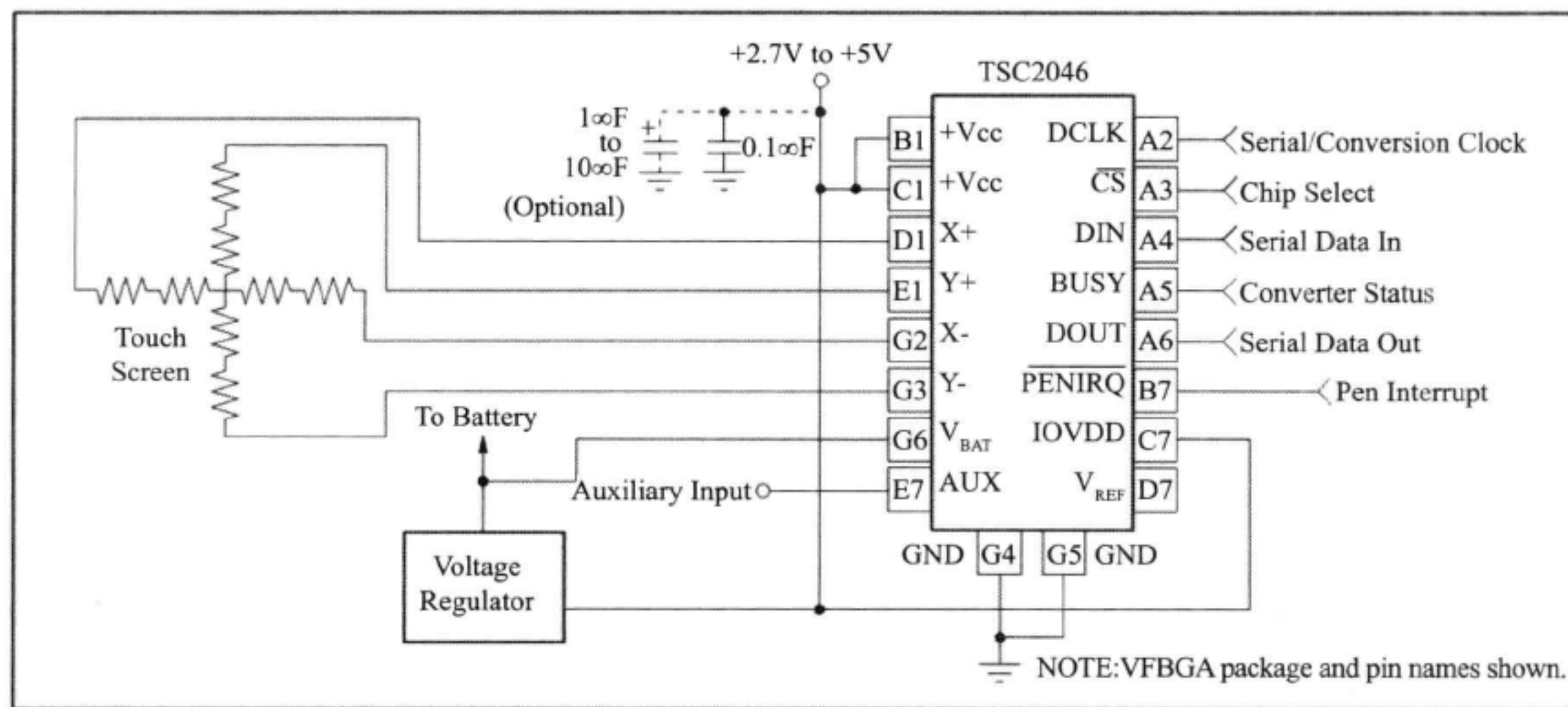


图 21-5 TSC2046 与电阻屏的连接图

21.5 LCD 触摸屏画板实验

21.5.1 实验描述及工程文件清单

1. 实验描述

配套 STM32 开发板驱动配套的 3.2 寸液晶、触摸屏，使用 FSMC 接口控制该屏幕自带的液晶控制器 ILI9341，使用 SPI 接口与触摸屏控制器 TSC2046 通信。驱动成功后可在屏幕上使用基本的触摸绘图功能。

2. 硬件连接

TFT 数据线：

□ PD14-FSMC-D0 : LCD-DB0

- ☐ PD15-FSMC-D1 : LCD-DB1
- ☐ PD0-FSMC-D2 : LCD-DB2
- ☐ PD1-FSMC-D3 : LCD-DB3
- ☐ PE7-FSMC-D4 : LCD-DB4
- ☐ PE8-FSMC-D5 : LCD-DB5
- ☐ PE9-FSMC-D6 : LCD-DB6
- ☐ PE10-FSMC-D7 : LCD-DB7
- ☐ PE11-FSMC-D8 : LCD-DB8
- ☐ PE12-FSMC-D9 : LCD-DB9
- ☐ PE13-FSMC-D10 : LCD-DB10
- ☐ PE14-FSMC-D11 : LCD-DB11
- ☐ PE15-FSMC-D12 : LCD-DB12
- ☐ PD8-FSMC-D13 : LCD-DB13
- ☐ PD9-FSMC-D14 : LCD-DB14
- ☐ PD10-FSMC-D15 : LCD-DB15

TFT 控制线 :

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

触摸屏 TSC2046 控制线 :

- ☐ PA5-SPI1-SCK : TSC2046-SPI -SCK
- ☐ PA7-SPI1-MOSI : TSC2046-SPI - MOSI
- ☐ PA6-SPI1-MISO : TSC2046-SPI - MISO
- ☐ PB7-I2C1-SDA : TSC2046-SPI-CS
- ☐ PB6-I2C1-SCL : TSC2046- INT_IRQ

3. 库文件

使用 3.5 版本固件库 :

- ☐ Startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/misc.c
- ☐ FWlib/stm32f10x_spi.c

- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_exti.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_fsmc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/lcd.c
- ☐ USER/SysTick.c
- ☐ USER/lcd_botton.c
- ☐ USER/Touch.c

2.4 寸和 3.2 寸的 LCD 接口图见图 21-6。



图 21-6 2.4 寸和 3.2 寸的 LCD 接口图

21.5.2 配置工程环境

本 LCD 触摸屏画板实验中我们用到了 GPIO、RCC、SPI、EXTI、FSMC 外设，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_spi.c、stm32f10x_exti.c、stm32f10x_fsmc.c。由于在 TSC2046 的触摸检测中使用了中断，所以还要把 misc.c 文件添加进工程。

本工程使用了旧的用户文件 SysTick.c 用作定时，把它添加到新工程之中，并新建 lcd_botton.c、lcd.c、Touch.c 及相应的头文件。其中 lcd_botton.c 文件定义了最底层的 LCD 控制函数，LCD 上层的函数如画点、显示字符等位于 lcd.c 文件中。

最后在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 21-1。

代码清单 21-1 LCD 例程的 stm32f10x_conf.h 文件配置

```

1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****/
9.
10. #include "stm32f10x_exti.h"
11. #include "stm32f10x_fsmc.h"
12. #include "stm32f10x_gpio.h"
13. #include "stm32f10x_rcc.h"
14. #include "stm32f10x_spi.h"
15. #include "misc.h"

```

21.5.3 main 文件

从本工程的 main 文件分析代码的执行流程，见代码清单 21-2。

代码清单 21-2 LCD 例程的 main 函数

```

1. /*
2.  * 函数名：main
3.  * 描述   ：主函数
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7. int main(void)
8. {
9.     SysTick_Init();           /*systick 初始化*/
10.    LCD_Init();                /*LCD 初始化*/
11.    Touch_init();              /* 触摸初始化*/
12.
13.    while(Touch1_Calibrate() !=0); /* 等待触摸屏校准完毕*/
14.    Init_Palette();            /* 画板初始化*/
15.
16.    while (1)
17.    {
18.        if(touch_flag == 1)    /* 如果触笔按下了*/
19.        {
20.                                /* 获取点的坐标*/
21.            if(Get_touch_point(&display, Read_2046_2(), &touch_para) !=DISABLE)
22.            {
23.                                /* 画点*/
24.                Palette_draw_point(display.x,display.y);           /* 画点*/
25.            }
26.        }
27.    }
28.}

```

其执行流程如下：

- 1) 调用 SysTick_Init()、LCD_Init()、Touch_init() 初始化 STM32 的 SysTick、FSMC、SPI 外设，并用 FSMC 和 SPI 接口初始化 ILI9341 和 TSC2046 控制器。
- 2) 调用 Touch1_Calibrate() 函数进行触摸屏校准，使得触摸屏与液晶屏的坐标匹配。
- 3) 调用 Init_Palette() 函数初始化触摸画板的应用程序，使得在 LCD 上显示画板界面，并能够正常响应触摸屏的信号。
- 4) 第 16 ~ 27 行的 while 循环，通过不断检测触笔按下标志 touch_flag，判断触摸屏是否被触笔按下。触摸屏控制器 TSC2046 检测到触笔信号时，由它的 PENIRQ 引脚触发 STM32 的中断，在中断服务函数中对 touch_flag 标志置“1”。
- 5) 第 21 行，若检测到触笔按下，调用 Get_touch_point() 函数读取 TSC2046 的寄存器，获得与触点的 X、Y 坐标相关的电压信号，转化成 LCD 的 X、Y 坐标。
- 6) 获取了触点坐标后，使液晶屏在该坐标点显示对应的颜色。
- 7) 循环触点捕捉、画点过程，就实现了触摸画板的功能。

21.5.4 初始化 FSMC 模式

1. 初始化液晶屏流程

在 main 函数中调用了 LCD_Init() 函数，它对液晶控制器 ILI9341 用到的 GPIO、FSMC 接口进行了初始化，并且向该控制器写入命令参数，配置好液晶屏的基本功能。其函数定义位于 lcd_botton.c 文件，见代码清单 21-3。

代码清单 21-3 LCD_Init() 函数

```

1.  /*****
2.   * 函数名：LCD_Init
3.   * 描述   ：LCD 控制 I/O 初始化
4.   *         LCD FSMC 初始化
5.   *         LCD 控制器 HX8347 初始化
6.   * 输入   ：无
7.   * 输出   ：无
8.   * 举例   ：无
9.   * 注意   ：无
10. *****/
11. void LCD_Init(void)
12. {
13.     unsigned long i;
14.
15.     LCD_GPIO_Config();           // 初始化使用到的 GPIO
16.     LCD_FSMC_Config();           // 初始化 FSMC 模式
17.     LCD_Rst();                   // 复位 LCD 液晶屏
18.     Lcd_init_conf();             // 写入命令参数，对液晶屏进行基本的初始化配置
19.     Lcd_data_start();            // 发送写 GRAM 命令
20.     for(i=0; i<(320*240); i++)
21.     {

```



```

22.         LCD_WR_Data(GBLUE);    // 发送颜色数据, 初始化屏幕为 GBLUE 颜色
23.
24.     }
25. }

```

LCD_Init() 函数执行后, 最直观的结果是使 LCD 整个屏幕显示编码为 0X07FF 的 GBLUE 颜色。

函数中调用的 LCD_GPIO_Config() 主要工作是把液晶屏 (不包括触摸屏) 中使用到的 GPIO 引脚初始化和使能外设时钟, 除了背光、复位用的 PD13 和 PD1 设置为通用推挽输出外, 其他与 FSMC 接口相关的地址信号、数据信号、控制信号的端口均设置为复用推挽输出。

2. 初始化 FSMC 模式

接下来 LCD_Init() 函数调用 LCD_FSMC_Config() 设置 FSMC 的模式, 我们的目的是使用它的 NOR FLASH 模式模拟出 8080 接口, 在 LCD 接口中我们使用 FSMC 地址线 A16 作为 8080 的 D/CX 命令选择信号。LCD_FSMC_Config() 具体代码见代码清单 21-4。

代码清单 21-4 LCD_FSMC_Config() 函数

```

1.  /*****
2.  * 函数名: LCD_FSMC_Config
3.  * 描述  : LCD  FSMC 模式配置
4.  * 输入  : 无
5.  * 输出  : 无
6.  * 举例  : 无
7.  * 注意  : 无
8.  *****/
9.  void LCD_FSMC_Config(void)
10. {
11.     FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
12.     FSMC_NORSRAMTimingInitTypeDef  p;
13.
14.
15.     p.FSMC_AddressSetupTime = 0x02;           // 地址建立时间
16.     p.FSMC_AddressHoldTime = 0x00;           // 地址保持时间
17.     p.FSMC_DataSetupTime = 0x05;             // 数据建立时间
18.     p.FSMC_BusTurnAroundDuration = 0x00;     // 总线恢复时间
19.     p.FSMC_CLKDivision = 0x00;              // 时钟分频
20.     p.FSMC_DataLatency = 0x00;              // 数据保持时间
21.     p.FSMC_AccessMode = FSMC_AccessMode_B;
22.     // 在地址数\数据线不复用的情况下, ABCD 模式的区别不大
23.     // 本成员配置只有使用扩展模式才有效
24.
25.
26.     FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1; //NOR FLASH 的 BANK1
27.     FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_
Disable;    // 数据线与地址线不复用
28.     FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_
NOR;        // 存储器类型 NOR FLASH
29.     FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;
// 数据宽度为 16 位
30.     FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_
Disable;    // 使用异步写模式, 禁止突发模式
31.     FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_

```

```

        Low;          // 本成员的配置只在突发模式下有效, 等待信号极性为低
32.   FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_    Disable;
        // 禁止非对齐突发模式
33.   FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_
        BeforeWaitState;
        // 本成员配置仅在突发模式下有效。NWAIT 信号在什么时期产生
34.   FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_
        Disable;          // 本成员的配置只在突发模式下有效, 禁用 N WAIT 信号
35.   FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_
        Disable;          // 禁止突发写操作
36.   FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_
        Enable;           // 写使能
37.   FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_
        Disable;          // 禁止扩展模式, 扩展模式可以使用独立的读、写模式
38.   FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p;
        // 配置读写时序
39.   FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p;      // 配置写时序
40.
41.
42.   FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
43.
44.   /* 使能 FSMC Bank1_SRAM Bank */
45.   FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);
46.)

```

本函数主要使用了两种类型的结构体对 FSMC 进行配置, 第一种为 FSMC_NORSRAMInitTypeDef 类型的结构体, 主要用于 NOR FLASH 的模式配置, 包括存储器类型、数据宽度等。另一种类型为 FSMC_NORSRAMTimingInitTypeDef, 用于配置 FSMC 的 NOR FLASH 模式下读写时序中的地址建立时间、地址保持时间等, 代码中用它定义了结构体 p, 第二种类型的结构体在前一种结构体中被指针调用。

下面先来分析 FSMC_NORSRAMInitTypeDef 的结构体成员。

(1) FSMC_Bank

用于选择外接存储器的区域 (或地址), 见图 21-7, STM32 的存储器映射中, 把 0x6000 0000 ~ 0x9fff ffff 的地址都映射到被 FSMC 控制的外存储器中, 其中属于 NOR FLASH 的为 0x6000 0000 ~ 0x6fff ffff。而属于 NOR FLASH 的这部分地址空间又被分为 4 份, 每份大小为 64 MB, 编号为 BANK1 ~ BANK4。分块是由 FSMC 寻址范围决定的, 该接口的地址线最多为 26 条, 即最大寻址空间为 $2^{26}=64$ MB。为了扩展寻址空间, 可把地址与数据线与多片 NOR FLASH 并联, 由不同的片选信号 NE[3:0] 区分不同的块。

在本实验中, 我们使用的是 FSMC 的信号线 NE1 作为控制 8080 的 CSX 片选信号, 所以我们把本成员配置为 FSMC_Bank1_NORSRAM1 (NE1 片选 BANK1)。

(2) FSMC_DataAddressMux

本成员用于配置 FSMC 的数据线与地址线是否复用。FSMC 支持数据线与地址线复用或非复用两种模式。在非复用模式下 16 位数据线及 26 位地址线分开使用; 复用模式则低 16 位数据 / 地址线复用。在复用模式下, 推荐使用地址锁存器以区分数据与地址。当 NADV 信号线为低时, 复用信号线 ADx (x=0~15) 上出现地址信号 Ax; 当 NADV 变高时, ADx 上出现数据信号 Dx。

本实验中用 FSMC 模拟 8080 接口，地址线 A16 提供 8080 的 D/CX 信号，实际上就只使用了这一条地址线，IO 资源并不紧张，所以把本成员配置为 FSMC_DataAddressMux_Disable (非复用模式)。

(3) FSMC_MemoryType

本成员用于配置 FSMC 外接存储器的类型，可被配置为 NOR FLASH 模式、PSARM 模式及 SRAM 模式。

在本实验的应用中，由于 NOR FLASH 模式的时序与 8080 更接近，所以本结构体被配置为 FSMC_MemoryType_NOR (NOR FLASH 模式)。

(4) FSMC_MemoryDataWidth

本成员用于设置 FSMC 接口的数据宽度，可被设置为 8 位或 16 位。在 STM32 地址映射到 FSMC 接口的结构中，HADDR 信号线是需要连接到外部存储器的内部 AHB 地址线，是字节地址。

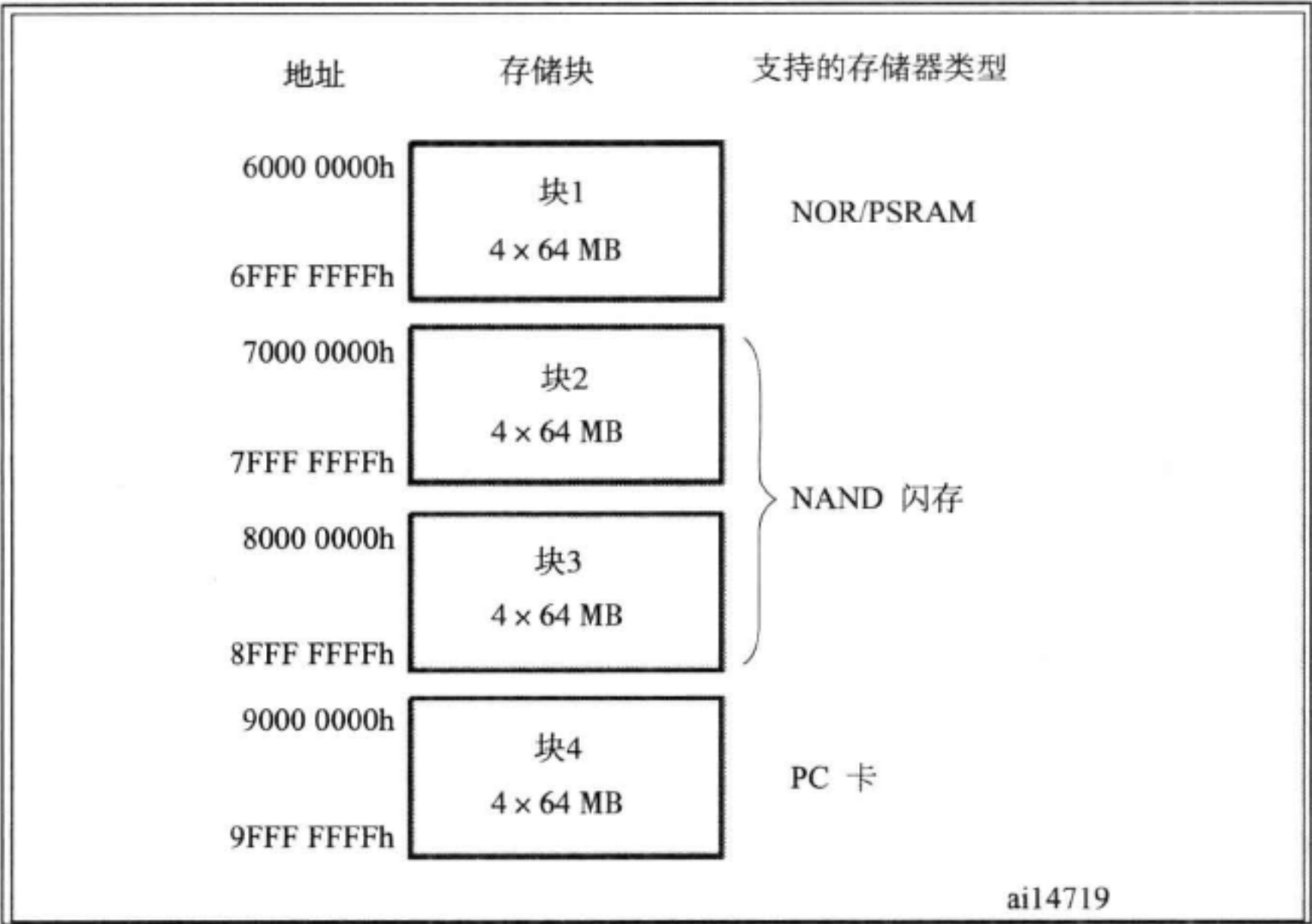


图 21-7 FSMC 存储块

若存储器的数据线宽为 8 位，FSMC 的 26 条地址信号线 FSMC_A[25:0] 可以直接引入与 AHB 相连的 HADDR[25:0]，26 条字节地址信号线最大寻址空间为 64 MB。见表 21-4。

表 21-4 外部存储器地址

数据宽度	连到存储器的地址线	最大访问存储器空间
8 位	HADDR[25:0] 与 FSMC_A[25:0] 对应相连	64MB × 8 = 512 Mb
16 位	HADDR[25:1] 与 FSMC_A[24:0] 对应相连	64MB/2 × 16 = 512 Mb

若存储器的数据线宽 16 位，则存储器的地址信号线是半字地址（16 位）。为了使 HADDR 的字节地址信号线与存储器匹配，FSMC 的 25 条地址信号线 FSMC_A[24:0] 与 HADDR[25:1] 相

连, 由于变成了半字地址, 仅需要 25 条半字地址信号线就达到最大寻址空间 64 MB。正因地址线的不对称相连, 16 位数据线宽下, 实际的访问地址为右移一位之后的地址。

本实验中 8080 接口采用 16 位模式, 所以我们把本成员配置为 `FSMC_MemoryDataWidth_16b`, 由于地址线不对称相连, 这会影响到我们用地址信号线 A16 控制的 8080 接口的 D/CX 信号。

(5) `FSMC_BurstAccessMode`

本成员用于配置访问模式。FSMC 对存储器的访问分为异步模式和突发模式 (同步模式)。在异步模式下, 每次传送数据都需要产生一个确定的地址, 而突发模式可以在开始时提供一个地址之后, 把数据成组地连续写入。

本实验中使用 FSMC 模拟 8080 端口, 更适合使用异步模式, 因而向本成员赋值为 `FSMC_WriteBurst_Disable`。

(6) 突发模式参数配置

代码中的 30 ~ 34 行, 都是关于使用突发模式时的一些参数配置, 这些成员为: `FSMC_WaitSignalPolarity` (配置等待信号极性)、`FSMC_WrapMode` (配置是否使用非对齐方式)、`FSMC_WaitSignalActive` (配置等待信号什么时期产生)、`FSMC_WaitSignal` (配置是否使用等待信号)、`FSMC_WriteBurst` (配置是否允许突发写操作), 这些成员均需要在突发模式开启后配置才有效。

本实验使用的是异步模式, 所以这些成员的参数没有意义。

(7) `FSMC_WriteOperation`

本成员用于配置写操作使能, 如果禁止了写操作, FSMC 不会产生写时序, 但仍可从存储器中读出数据。

本实验需要写时序, 所以向本成员赋值为 `FSMC_WriteOperation_Enable` (写使能)。

(8) `FSMC_ExtendedMode`

本成员用于配置是否使用扩展模式, 在扩展模式下, 读时序和写时序可以使用独立时序模式。如读时序使用模式 A, 写时序使用模式 B, 类似 A、B、C、D 模式实际上差别不大, 主要是在使用数据 / 地址线复用的情况下, NADV 信号产生的时序不一样, 具体的时序图可查阅《STM32 参考手册》。

本实验中数据 / 地址线不复用, 所以读写时序中不同的 NADV 信号没有影响, 禁止使用扩展模式 `FSMC_ExtendedMode_Disable`。

(9) `FSMC_ReadWriteTimingStruct` 和 `FSMC_WriteTimingStruct`

这两个参数分别用来设置 FSMC 的读时序及写时序的时间参数。若使用了扩展模式, 则前者配置的是读时序, 后者为写时序; 若禁止了扩展模式, 则读写时序都使用 `FSMC_ReadWriteTimingStruct` 结构体中的参数。

在配置这两个参数时, 使用的是类型 `FSMC_NORSRAMTimingInitTypeDef` 时序初始化结构体, 对这种类型结构体的成员进行赋值。它的成员分别有: `FSMC_AddressSetupTime` (地址建立时间)、`FSMC_AddressHoldTime` (地址保持时间)、`FSMC_DataSetupTime` (数据建立时间)、`FSMC_DataLatency` (数据保持时间)、`FSMC_BusTurnAroundDuration` (总线恢复时间)、`FSMC_CLKDivision` (时钟分频)、`FSMC_AccessMode` (访问模式)。对以上各个时间成员赋的数值 X 表示 X 个时钟周期, 它的时钟是由 HCLK 经过成员时钟分频得到的, 该分频值在成员 `FSMC_`

CLKDivision (时钟分频) 中设置。其中 FSMC_AccessMode (访问模式) 成员的设置只有在开启了扩展模式才有效, 而且开启了扩展模式后, 读时序和写时序的设置可以是独立的。

本实验中的时序设置是根据 ILI9341 的 datasheet 设置的, 调试的时候可以先把这些值设置得大一些, 然后慢慢靠近 datasheet 要求的最小值, 这样会取得比较好的效果。时序的参数设置对 LCD 的显示效果有一定的影响。

配置完初始化结构体后, 要调用库函数 FSMC_NORSRAMInit() 把这些配置参数写到控制寄存器, 还要调用 FSMC_NORSRAMCmd() 使能 BANK1。如果是使用 FSMC 配置其他存储器如 NAND FLASH, 要使用其他库函数及初始化结构体。

21.5.5 FSMC 模拟 8080 读写参数、命令

回到 LCD_Init() 函数的执行流程, 初始化完成 FSMC 接口后, 就可以使用它控制 ILI9341 了。在 LCD_Init() 中调用 Lcd_init_conf() 函数向 ILI9341 写入一系列控制参数, 见代码清单 21-5。

代码清单 21-5 Lcd_init_conf() 函数

```

1.  /*****
2.   * 函数名: Lcd_init_conf
3.   * 描述  : ILI9341 LCD 寄存器初始配置
4.   * 输入  : 无
5.   * 输出  : 无
6.   * 举例  : 无
7.   * 注意  : 无
8.   *****/
9.  void Lcd_init_conf(void)
10. {
11.     DEBUG_DELAY();
12.     LCD_ILI9341_CMD(0xCF);
13.     LCD_ILI9341_Parameter(0x00);
14.     LCD_ILI9341_Parameter(0x81);
15.     LCD_ILI9341_Parameter(0x30);
16.
17.     DEBUG_DELAY();
18.     LCD_ILI9341_CMD(0xED);
19.     LCD_ILI9341_Parameter(0x64);
20.     LCD_ILI9341_Parameter(0x03);
21.     LCD_ILI9341_Parameter(0x12);
22.     LCD_ILI9341_Parameter(0x81);
23.
24.     DEBUG_DELAY();
25.     LCD_ILI9341_CMD(0xE8);
26.     LCD_ILI9341_Parameter(0x85);
27.     LCD_ILI9341_Parameter(0x10);
28.     LCD_ILI9341_Parameter(0x78);
29.     // .....此处省略几十行.....

```

本函数十分长, 由于篇幅问题, 以上只是该函数其中的一部分, 省略部分的代码也是这样的模板, 只是写入的命令和参数不一样而已, 这些命令和参数设置了像素点颜色格式、屏幕扫描方

式、横屏\竖屏等初始化配置，这些命令的意义从 ILI9341 的 datasheet 命令列表中可以查到。该函数通过调用 LCD_ILI9341_CMD() 写入命令，用 LCD_ILI9341_Parameter() 写入参数。它们实质是两个宏，见代码清单 21-6。

代码清单 21-6 LCD_ILI9341_CMD() 和 LCD_ILI9341_Parameter() 宏

```

1. /*****/
2. #define LCD_ILI9341_CMD(index)      LCD_WR_REG(index)
3. #define LCD_ILI9341_Parameter(val)  LCD_WR_Data(val)
4.
5. /**** 为了移植方便，上面的宏只是封装，以下才是最底层的宏 *****/
6. /* 选择 BANK1-BORSRAM1 连接 TFT，地址范围为 0X60000000 ~ 0X63FFFFFF
7.  * FSMC_A16 接 LCD 的 DC (寄存器 / 数据选择) 脚
8.  * 16 bit => FSMC[24:0] 对应 HADDR[25:1]
9.  * 寄存器基地址 = 0X60000000
10. * RAM 基地址 = 0X60020000 = 0X60000000+2^16*2 = 0X60000000 + 0X20000 = 0X60020000
11. * 当选择不同的地址线时，地址要重新计算。
12. */
13.
14. #define Bank1_LCD_D      ((u32)0x60020000)      //Disp Data ADDR
15. #define Bank1_LCD_C      ((u32)0x60000000)      //Disp Reg ADDR
16.
17. /* 选定 LCD 指定寄存器 (命令编码) */
18. #define LCD_WR_REG(index)      ((*(__IO u16 *) (Bank1_LCD_C)) = ((u16)index))
19.
20. /* 往 LCD 写入数据 */
21. #define LCD_WR_Data(val)      ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))

```

这部分是 FSMC 模拟 8080 接口的精髓。

1. 读写参数、命令

先来看第 21 行的宏，LCD_WR_Data (val)，这是一个带参宏，用于向 LCD 控制器写入参数，参数为 val。它的宏展开为：

```
1. ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))
```

宏展开中的 (Bank1_LCD_D) 是一个在第 14 行定义的宏，它的值为 0x6002 0000，实质是一个地址，这个地址的计算在后面介绍。

(__IO u16 *) (Bank1_LCD_D) 表示把 (Bank1_LCD_D) 强制转换成一个 16 位的地址。((* (__IO u16 *) (Bank1_LCD_D)) 表示再对这个地址做 “*” 指针运算，取该指针对象的内容，并把它的内容赋值为 = ((u16) (val)))。所以整个宏的操作就是：把参数 val 写入地址为 0x6002 0000 的地址空间。

由于这个地址被 STM32 映射到外存储器，所以会由 FSMC 外设以访问 NOR FLASH 的形式、时序，在地址线上发出 0x6002 0000 地址信号，在数据线上发出 val 数据信号，写入参数到外存储器中。而 FSMC 接口又被模拟成了 8080 接口，最终 val 被 8080 接口理解为参数，传输到 ILI9341 控制器中。

2. 计算地址

见图 21-8。计算地址前，再明确一下在本实验中，使用的是 FSMC_NE1 作为 8080_CS 片选信号（即图 21-8 中的 NEx 线），以 FSMC_A16 作为 8080_D/CX 数据/命令信号（图 21-8 中为 RS，意义相同）。

按照这种连接，FSMC_NE1 为低电平、FSMC_A16 为高电平，表示通过 D[15:0] 发送\接收的数据被 8080 接口解释为参数（数值），当我们访问 0x6002 0000 这个地址的时候，正好符合这个条件。该地址的计算过程如下。

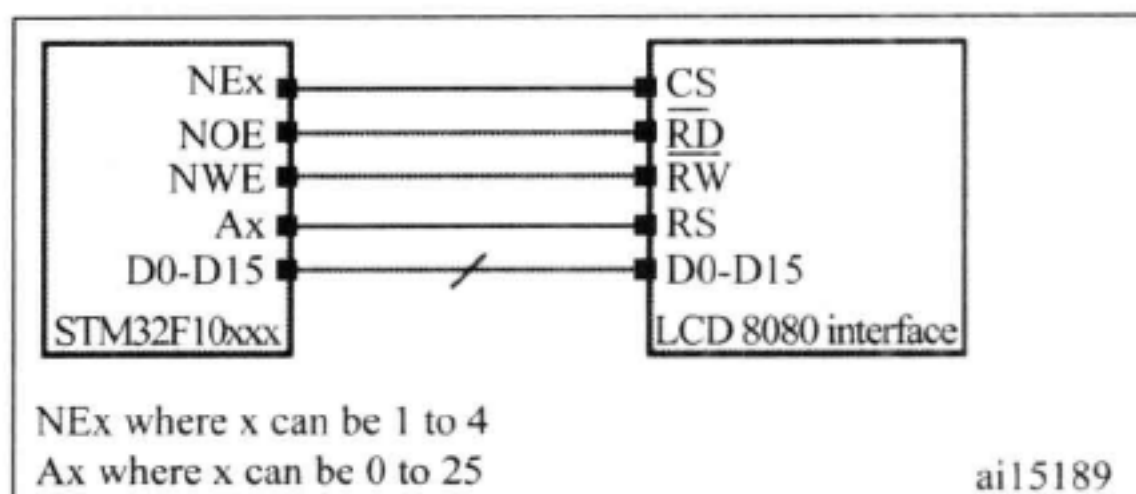


图 21-8 FSMC 与 8080 端口连接简图

由于选择的是使用 FSMC_NE1 片选信号线，片选的为 BANK1，所以基地址为 0x6000 0000。要把地址线 FSMC_A16 置为高电平，可以采用下列算式：

```
1. 0x6000 0000 |= 1<<16; // 结果 = 0x6001 0000
```

但是，这样计算出来的地址只是数据线为 8 位模式下的字节地址。由于我们采用的是 16 位数据线，FSMC[24:0] 与 HADDR[25:1] 对齐，HADDR 地址要左移一位才是 FSMC 的访问地址。因此为了把 FSMC 中的 FSMC_A16 (D/CX 线) 置 1，实际上要对应到 HADDR 地址 (AHB 地址) 的 HADDR_A17，即正确的计算地址公式应为：

```
1. 0x6000 0000 |= 1<<(16+1); // 此为正确结果 = 0x6002 0000
```

对于 16 位数据线模式，能使 FSMC_NE1 为低电平，FSMC_A16 为高电平的地址，并不只有 0x6002 0000 一个，只要是属于 0x6000 0000 ~ 0x63FF FFFF 范围内 (BANK1 地址范围)，HADDR_A17 位为高电平的地址均可，这是因为我们只采用了 FSMC_A16 用于 8080 的 D/CX 信号，所以地址线的电平状态并无影响。如 0x6002 0001 地址，在本实验中是与 0x6002 0000 等价的。

若修改这个地址，使 FSMC_NE1 为低电平，FSMC_A16 为低电平，即 D/CX 被置 0，8080 会把由数据线传输的信号理解为命令。以同样的方式计算，符合这样要求的其中一个地址为 0x6000 0000，向这个地址空间赋值，这个值最终会被 8080 接口解释为命令。宏 LCD_WR_REG (index) 就是这样实现的。

3. 给整个屏幕上色

再次回到 LCD_Init() 函数，它调用完 Lcd_init_conf() 初始化了液晶屏后，使用 Lcd_data_start() 函数向液晶屏发送了一个命令——写 GRAM 内容，即后面发送的数据都被解析为显示到屏幕像素点的数据。代码中使用 for 循环把语句 LCD_WR_Data (GBLUE) 执行了 320×240 次，即把所有像素点都显示为 GBLUE 颜色。

21.5.6 液晶屏画点函数

初始化了液晶屏后，就可以控制液晶屏上每个像素点的颜色了。如果能够实现一个画点函数，在指定的 (x,y) 坐标像素点上显示指定的颜色，那么就能够实现一切液晶屏最复杂的显示

功能,如在液晶屏指定位置显示形状、文字、图像,都可以通过调用画点函数或以类似的方式控制液晶屏的像素点。本实验中的画点函数 LCD_Point() 在 lcd.c 文件中定义,见代码清单 21-7。

代码清单 21-7 LCD_Point() 函数

```

1.  /*****
2.   * 函数名: LCD_Point
3.   * 描述  : 在指定坐标处显示一个点
4.   * 输入  : -x 横向显示位置 0 ~ 319
5.             -y 纵向显示位置 0 ~ 239
6.   * 输出  : 无
7.   * 举例  :    LCD_Point(100,200);
8.             LCD_Point(10,200);
9.             LCD_Point(300,220);
10.  * 注意  :    (0,0) 位置为液晶屏左上角 已测试
11. *****/
12. void LCD_Point(u16 x,u16 y)
13. {
14.     LCD_open_windows(x,y,1,1);
15.     LCD_WR_Data(POINT_COLOR);
16. }
```

本函数首先调用了一个液晶显示窗口函数 LCD_open_windows() 用于开辟液晶屏上的显示区域,后面写入的颜色数据将被显示到该区域中。示窗函数按 (x,y,1,1) 的参数调用时,该函数开辟了一个坐标为 (x,y) 的像素点。接着调用 LCD_WR_Data() 向 ILI9341 写入颜色数据,像素的坐标即为示窗函数开辟的显示坐标。于是 LCD_Point() 函数就实现了控制特定坐标的像素点。

接下来分析 LCD_open_windows() 函数是如何开辟显示区域的,见代码清单 21-8。

代码清单 21-8 LCD_open_windows() 函数

```

1.  /*****
2.   * 函数名: LCD_open_windows
3.   * 描述  : 开窗 (以 x,y 为坐标起点, 长为 len, 高为 wid)
4.   * 输入  : -x    窗户起点
5.             -y    窗户起点
6.             -len  窗户长
7.             -wid  窗户宽
8.   * 输出  : 无
9.   * 举例  : 无
10.  * 注意  : 无
11. *****/
12. void LCD_open_windows(u16 x,u16 y,u16 len,u16 wid)
13. {
14.
15.     if(display_direction == 0)        /* 如果是横屏选项 */
16.     {
17.
18.         LCD_ILI9341_CMD(0X2A);
19.         LCD_ILI9341_Parameter(x>>8);           //start 起始位置的高 8 位
20.         LCD_ILI9341_Parameter(x-((x>>8)<<8));    // 起始位置的低 8 位

```

```

21.         LCD_ILI9341_Parameter((x+len-1)>>8);           //end 结束位置的高8位
22. LCD_ILI9341_Parameter((x+len-1)-(((x+len-1)>>8)<<8));
    // 结束位置的低8位
23.
24.         LCD_ILI9341_CMD(0X2B);
25.         LCD_ILI9341_Parameter(y>>8);           //start
26.         LCD_ILI9341_Parameter(y-((y>>8)<<8));
27.         LCD_ILI9341_Parameter((y+wid-1)>>8);       //end
28.         LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-1)>>8)<<8));
29.
30.     }
31.     else
32.     {
33.         LCD_ILI9341_CMD(0X2B);
34.         LCD_ILI9341_Parameter(x>>8);
35.         LCD_ILI9341_Parameter(x-((x>>8)<<8));
36.         LCD_ILI9341_Parameter((x+len-1)>>8);
37.         LCD_ILI9341_Parameter((x+len-1)-(((x+len-1)>>8)<<8));
38.
39.         LCD_ILI9341_CMD(0X2A);
40.         LCD_ILI9341_Parameter(y>>8);
41.         LCD_ILI9341_Parameter(y-((y>>8)<<8));
42.         LCD_ILI9341_Parameter((y+wid-1)>>8);
43.         LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-1)>>8)<<8));
44.
45.     }
46.
47.     LCD_ILI9341_CMD(0x2c);
48. }

```

本函数主要使用了三个 ILI9341 的控制命令：

(1) 0x2A 列地址控制命令

用于设置显示区域的列像素区域。它有四个参数，分别为 SC 的高 8 位、SC 的低 8 位及 EC 的高 8 位、EC 的低 8 位。SC 和 EC 的意义见图 21-9。SC 表示要控制的显示区域的列起始坐标，EC 表示显示区域的列结束坐标。

(2) 0x2B 页地址控制命令

用于设置显示区域的页（行）像素区域。与上一命令类似，也有四个参数，分别为 SP 的高 8 位、SP 的低 8 位及 EP 的高 8 位、EP 的低 8 位。SP 和 EP 的意义见图 21-10。SP 表示要控制的显示区域的行起始坐标，EP 表示显示区域的行结束坐标。

(3) 0x2C 写 RAM 命令

本命令用于表示开始写入像素显示数据，紧跟着本命令后面的即为写入 GRAM 的 RGB 5:6:5 的颜色数据。在初始化液晶屏函数中也调用到本命令。使用本命令，后面的颜色数据按照一个接着一个预设的扫描方式（扫描方式决定了是横屏和竖屏）写入由 0x2A 和 0x2B 设置的显示区域中。横屏扫描方式为从左到右，扫描完一行（页）像素点再扫描下一行（页）。竖屏扫描为从上到下，扫描完一列再扫描下一列。

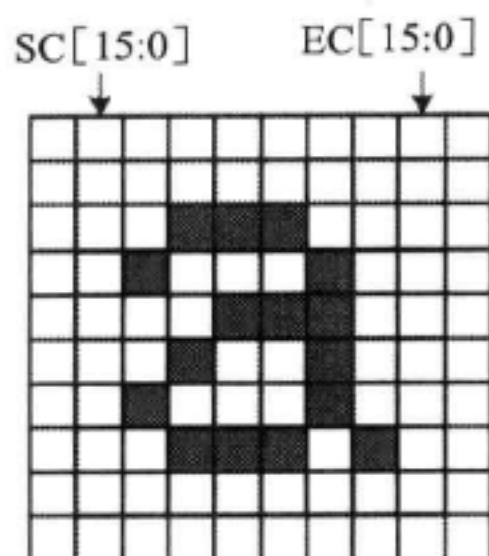


图 21-9 列控制命令 SC 和 EC

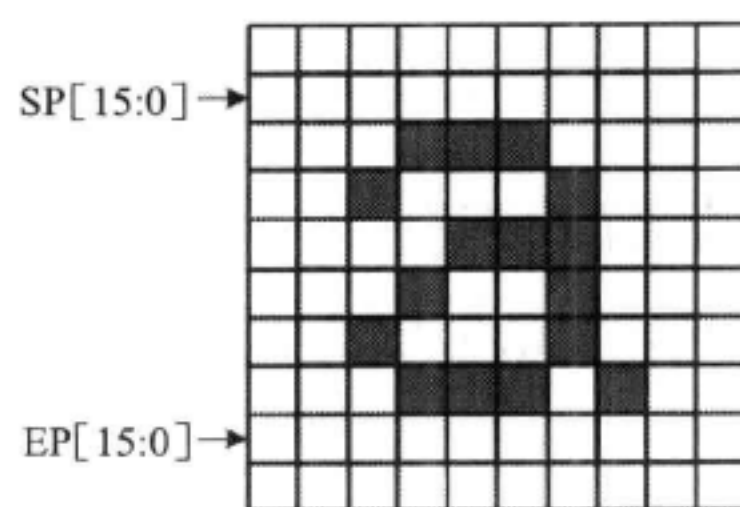


图 21-10 页控制命令 SP 和 EP

了解这三个命令后，就知道示窗函数的执行流程了，见图 21-11。

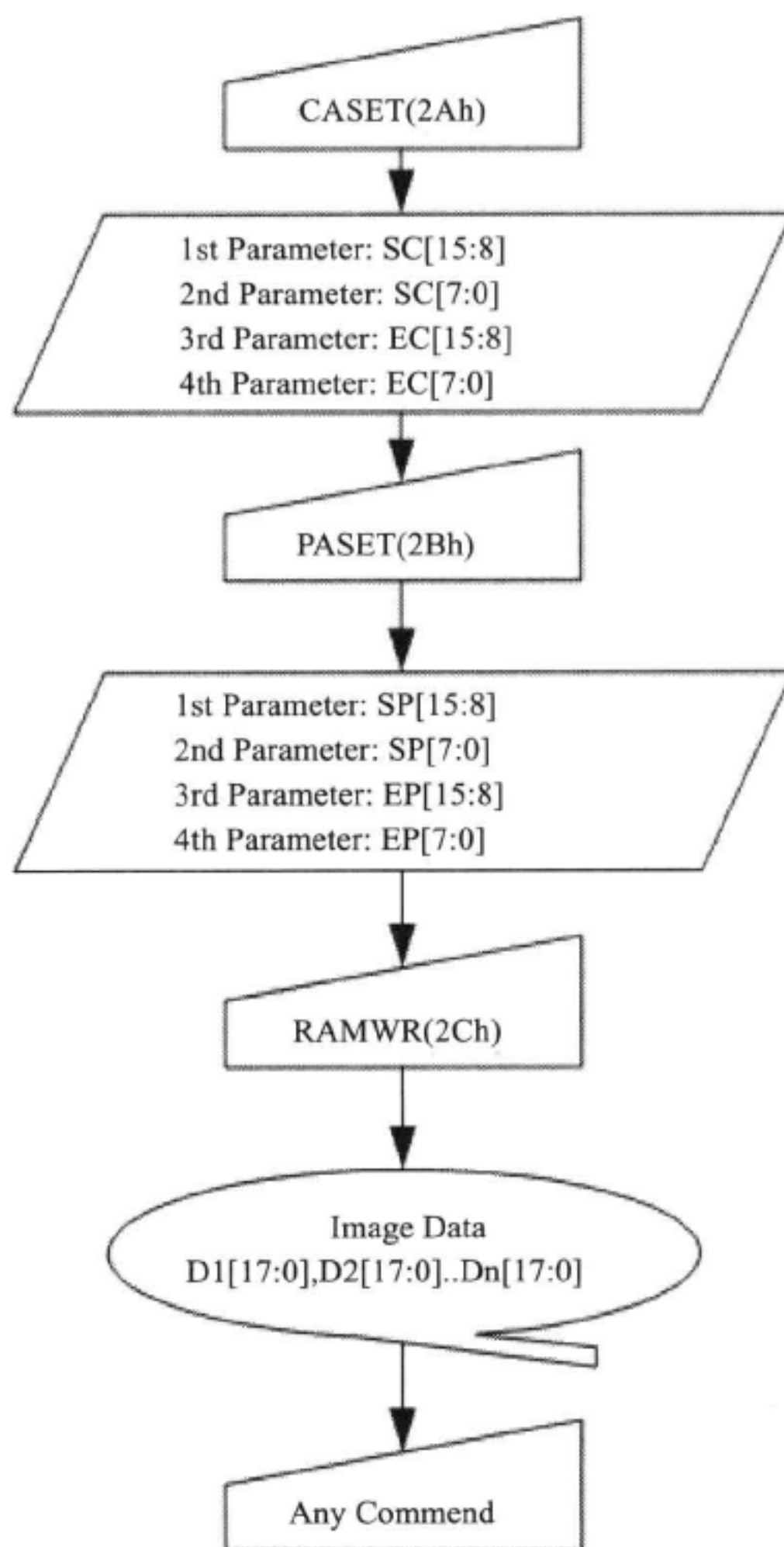


图 21-11 示窗函数执行流程

调用本示窗函数后, 就可以开辟一个起始坐标为 (x,y), 长为 len, 宽为 wid 的矩形显示窗口了。特别地, 当 len=wid=1 时, 开辟的为一个坐标为 (x,y) 的像素点。

21.5.7 触摸屏校正

在 main 函数初始化 LCD 之后, 调用了 Touchl_Calibrate() 函数进行触摸屏校正, 见代码清单 21-9。

代码清单 21-9 Touchl_Calibrate() 函数

```

1.  /*****
2.  * 函数名: Touchl_Calibrate
3.  * 描述  : 触摸屏校正函数
4.  * 输入  : 无
5.  * 输出  : 0    --- 校正成功
6.           1    --- 校正失败
7.  * 举例  : 无
8.  * 注意  : 无
9.  *****/
10. int Touchl_Calibrate(void)
11. {
12.     uint8_t i;
13.     ul6 test_x=0, test_y=0;
14.     ul6 gap_x=0, gap_y=0;
15.     Coordinate * Ptr;
16.     // delay_init();
17.     Set_direction(0);    // 设置为横屏
18.     for(i=0;i<4;i++)
19.     {
20. LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);
        // 使整个屏幕显示背景颜色
21.         LCD_Str_6x12_O(10, 10, "Touch Calibrate", 0);    // 显示提示信息
22.         LCD_Num_6x12_O(10,25, i+1, 0);                    // 显示触点次数
23.
24.         delay_ms(500);
25. DrawCross(DisplaySample[i].x, DisplaySample[i].y);
        // 显示校正用的 "十" 字
26.         do
27.         {
28.             Ptr=Read_2046();    // 读取 TSC2046 数据到变量 ptr
29.         }
30.         while( Ptr == (void*)0 );    // 当 ptr 为 0 时表示没有触点被按下
31. ScreenSample[i].x= Ptr->x;
        // 把读取的原始数据存放到 ScreenSample 结构体
32.         ScreenSample[i].y= Ptr->y;
33.
34.     }
35.
36.     /* 用原始参数计算出 原始参数与坐标的转换系数。 */
37.     Cal_touch_para( &DisplaySample[0], &ScreenSample[0], &touch_para );

```

```

38.
39.     /* 计算 X 值 */
40.     test_x = ( (touch_para.An * ScreenSample[3].x) +
41.               (touch_para.Bn * ScreenSample[3].y) +
42.               touch_para.Cn
43.               ) / touch_para.Divider ;
44.
45.     /* 计算 Y 值 */
46.     test_y = ( (touch_para.Dn * ScreenSample[3].x) +
47.               (touch_para.En * ScreenSample[3].y) +
48.               touch_para.Fn
49.               ) / touch_para.Divider ;
50.
51.     gap_x = (test_x > DisplaySample[3].x)?(test_x - DisplaySample[3].x):
              (DisplaySample[3].x - test_x);
52.     gap_y = (test_y > DisplaySample[3].y)?(test_y - DisplaySample[3].y):
              (DisplaySample[3].y - test_y);
53.
54.
55.     LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);
56.     if((gap_x>11)|| (gap_y>11))
57.     {
58.
59.         LCD_Str_6x12_O(100, 100,"Calibrate fail", 0);
60.         LCD_Str_6x12_O(100, 120," try again ", 0);
61.         delay_ms(2000);
62.         return 1;
63.     }
64.
65.
66.     aa1 = (touch_para.An*1.0)/touch_para.Divider;
67.     bb1 = (touch_para.Bn*1.0)/touch_para.Divider;
68.     cc1 = (touch_para.Cn*1.0)/touch_para.Divider;
69.
70.     aa2 = (touch_para.Dn*1.0)/touch_para.Divider;
71.     bb2 = (touch_para.En*1.0)/touch_para.Divider;
72.     cc2 = (touch_para.Fn*1.0)/touch_para.Divider;
73.
74.     LCD_Str_6x12_O(100, 100,"Calibrate Success", 0);
75.     delay_ms(1000);
76.
77.     return 0;
78. }

```

本函数的主要作用是在指定的几个液晶屏坐标（逻辑坐标）显示“十”字交叉点，由用户使用触笔点击触摸屏交叉点，读取由 TSC2046 测得的触点电压（物理坐标）。采集 4 个不同位置的触点电压（物理坐标），然后根据触摸校准算法把逻辑坐标与物理坐标转换公式的系数 A、B、C、D、E、F 计算出来。

若使用此函数校准成功后，用户再点击触摸屏时，可把测量出的触点电压（物理坐标）代入

已知系数的转换公式，计算出对应的液晶屏坐标（逻辑坐标）。

转换公式的系数为以上代码第 66 ~ 72 行中的 aa1、bb1、cc1、aa2、bb2、cc2 这几个全局变量，如果把这几个数据保存在非易失性存储器（SD 卡、EEPROM 等）中，上电后向这几个变量赋值，就不需要每次上电都进行一次触屏校准了。

本函数中大部分都是关于触摸屏校准算法的数学运算，有兴趣的读者可查阅其他相关资料来理解。在代码中的第 18 ~ 30 行，与获取触摸屏的触点电压有关，分析如下：

1) 第 20 ~ 25 行，调用 LCD_Rectangle()、LCD_Str_6x12_O()、LCD_Num_6x12_O()、DrawCross() 由液晶屏显示背景、提示信息及校准用的“十”字。这些函数都与液晶的画点函数原理类似，关于字符显示的内容在下一章进行说明。

2) 第 28 行，调用 Read_2046() 函数获取触点的电压，该函数向 TSC2046 控制器发送控制命令：若触笔点击触摸屏时采集触点的电压，采集 10 个电压取平均值，结果返回给变量 Ptr；若没有触点，则 Ptr 的值为 0，由 do-while 循环等待至采集到数据为止。Ptr 中保存的电压数据在后面被用于校准算法计算。Read_2046() 函数定义见代码清单 21-10。

代码清单 21-10 Read_2046() 函数

```

1.  /*****
2.  * 函数名：Read_2046
3.  * 描述   ：得到滤波之后的 X Y
4.  * 输入   ：无
5.  * 输出   ：Coordinate 结构体地址
6.  * 举例   ：无
7.  * 注意   ：速度相对比较慢
8.  *****/
9.  Coordinate *Read_2046(void)
10. {
11.     static Coordinate  screen;
12.     int m0,m1,m2,TP_X[1],TP_Y[1],temp[3];
13.     uint8_t count=0;
14.
15.     /* 坐标 X 和 Y 进行 9 次采样 */
16.     int buffer[2][9]={0},{0};
17.     do
18.     {
19.         Touch_GetAdXY(TP_X,TP_Y);
20.         buffer[0][count]=TP_X[0];
21.         buffer[1][count]=TP_Y[0];
22.         count++;
23.
24.     } /* 用户点击触摸屏时即 TP_INT_IN 信号为低 并且 count<9 */
25.     while(!INT_IN_2046&& count<9);
26.
27. // 由于篇幅问题，此处省略很多行，省略部分主要为计算 10 个采样电压的平均值
28.
29.
30.

```

在 Read_2046() 函数中，调用了 Touch_GetAdXY()，它用于获取一次触点 (x,y) 电压。实际上，驱动 TSC2046 最底层的是命令 WR_CMD (CHX) 和 WR_CMD (CHY)，发送了这两个命令后，TSC2046 开始采集相应的触点电压，通过 SPI 传送触点电压数据到 STM32。

命令语句中的 CHX 宏展开为 0xd0，CHY 为 0x90，它们是根据 TSC2046 的命令格式设定的。驱动 TSC2046 的命令控制字格式见表 21-5。

表 21-5 TSC2046 命令控制字

位7 (MSB)	位6	位5	位4	位3	位2	位1	位0 (LSB)
S	A2	A1	A0	MODE	SER DFR	PD1	PD0

其中 S 为数据传输起始标志位，该位必为 1，A2 ~ A0 进行通道选择，MODE 用于转换精度选择，1 为 8 位精度，0 为 12 位精度。表 21-6 为通道选择。

表 21-6 通道选择

A2	A1	A0	+REF	-REF	Y-	X+	Y+	Y-位置	X-位置	Z1-位置	Z2-位置	驱 动
0	0	1	Y+	Y-		+IN		M				Y+ 9Y-
0	1	1	Y+	X-		+IN				M		Y+ 9X-
1	0	0	Y+	Y+	+IN						M	Y+ 9X-
1	0	1	X+	X-			+IN		M			X+ 9X-

所谓通道选择即为检测哪一个通道的坐标。如 A2 ~ A0 为 001 时，即命令控制字为 0x90，根据表格知，芯片会给触摸屏的 Y 阻性材料层的两端提供 Y+、Y- 的电压，若有触笔点击，则 Y 触点电压可经过 X+ 利用 ADC 读取。同理，命令控制字为 0xd0 时，A2 ~ A0 为 101，即给 X 阻性材料层提供电压，触点电压经过 Y+ 由 ADC 读取。这就是 TSC2046 采集触点电压的原理。

21.5.8 检测触点、画点

回到 main 函数。触摸屏也校准好后，剩下的就是应用程序代码了，调用 Init_Palette() 使液晶屏显示出画板的界面，见图 21-12（由于印刷原因，无法分辨具体颜色）。

该画板的界面左侧为各种颜色方块，右侧为提供给用户进行绘画的空间。显示完该界面后，循环检测 touch_flag 标志，它在 stm32f10x_it.c 文件的中断服务函数中被赋值。当触摸屏被按下时会进入该函数，对 touch_flag 赋值，见代码清单 21-11。

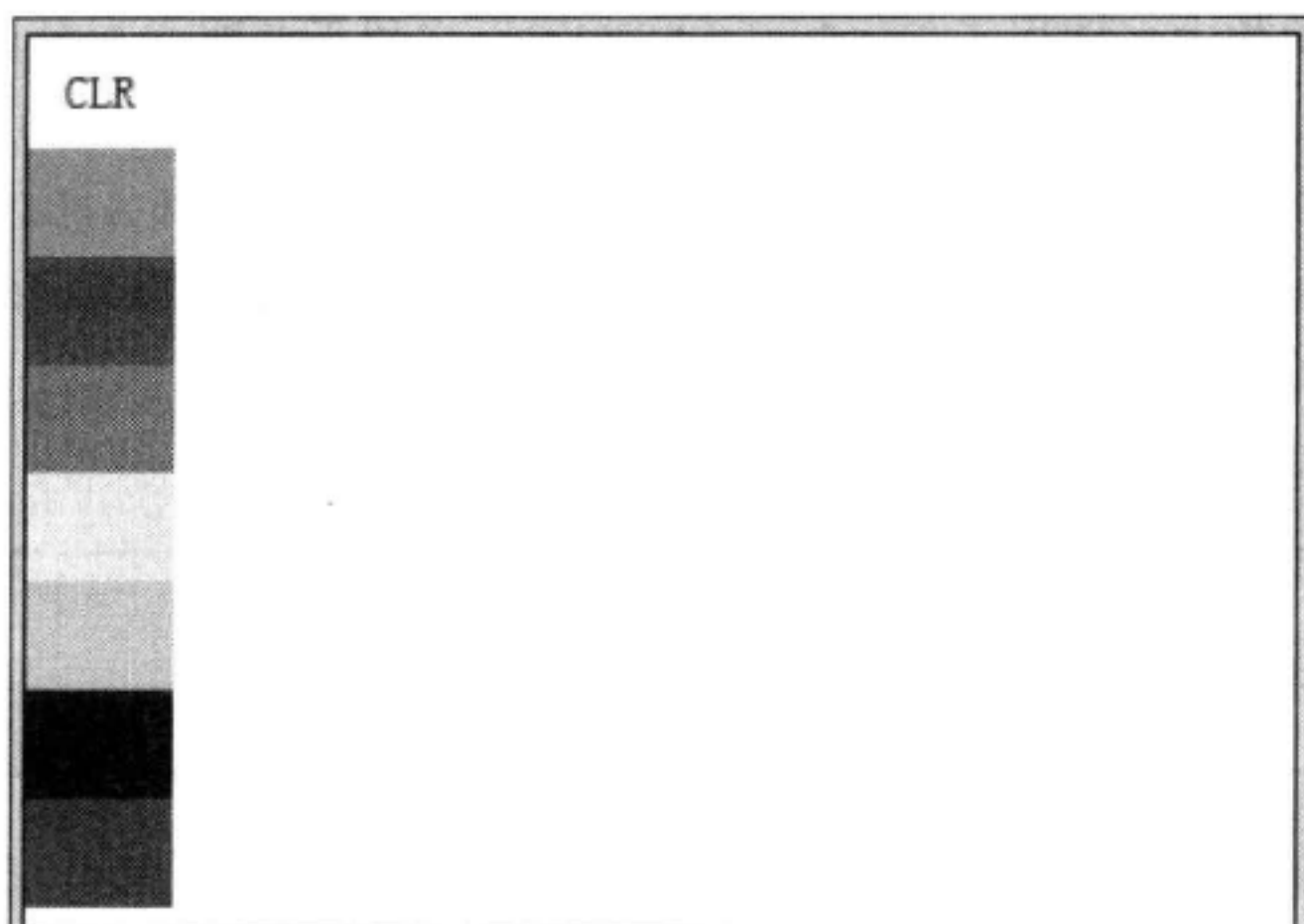


图 21-12 画板界面

代码清单 21-11 触摸屏的中断服务函数

```

1. void EXTI9_5_IRQHandler(void)
2. {
3.
4.
5.     if(EXTI_GetITStatus(EXTI_Line6) != RESET)
6.     {
7.         GPIO_ResetBits(GPIOB, GPIO_Pin_5);
8.
9.         touch_flag=1;
10.
11.     EXTI_ClearITPendingBit(EXTI_Line6);
12. }
13.}

```

若 touch_flag 标志为 1，即触摸屏被按下，main 中调用函数 Get_touch_point()，该函数通过 Read_2046_2() 获取触点电压，根据公式把电压转换为液晶坐标，并保存到 display 结构体中。得到液晶坐标后，main 调用 Palette_draw_point() 对相应的坐标点进行处理，若触点位于画板界面的颜色方块中，则使画笔变为该颜色，其后在空白界面的触点将显示该颜色的笔迹。

21.5.9 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上串口线（两头都是母的交叉线），接上液晶屏，将编译好的程序下载到开发板。运行后，可在 LCD 屏幕看到提示信息“Touch Calibrate”和触摸屏校正用的“十”符号。点击校正符号进行校正，若校正成功后会出现画板界面，在画板界面的右侧可进行绘画，点击左侧的颜色块可选择笔迹的颜色。



第 22 章

字库及 BMP 图片显示

在第 21 章中，我们已经成功地实现了驱动 LCD 和触摸屏，并制作了触摸画板小应用，但是若要显示文字或图片文件，则还需要利用文件系统，读取保存在 SD 卡中的字库文件、图片文件。

22.1 什么是字模

我们知道其实液晶屏就是一个由像素点组成的点阵，若要显示文字，则需要由很多像素点共同构成。见图 22-1，图中是两个由 16×16 的点阵显示的两个汉字。

如果我们规定：每个汉字都由这样 16×16 的点阵来显示，把笔迹经过的像素点以“1”表示，没有笔迹的点以“0”表示，每个像素点的状态以一个二进制位来记录，用 $16 \times 16 / 8 = 32$ 个字节就可以把这个字记录下来。这 32 个字节数据就称为该文字的字模，还有其他常用字模是 24×24 、 32×32 的。 16×16 的“字”的字模数据见代码清单 22-1。

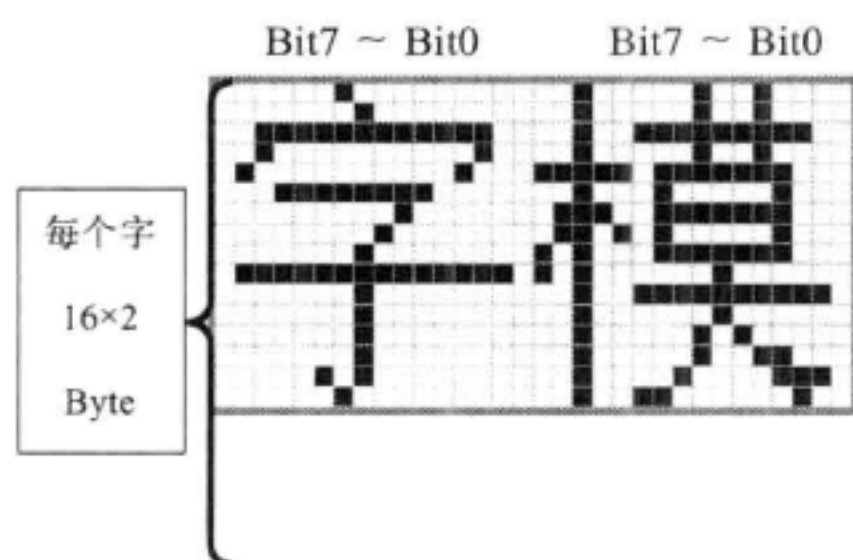


图 22-1 字模


代码清单 22-1 “字”的字模数据

```
1. /* 字 */
2. unsigned char code Bmp003[]=
3. {
4. /*-----
5. ; 源文件 / 文字 : 字
6. ; 宽×高(像素): 16×16
7. ; 字模格式/大小 : 单色点阵液晶字模, 横向取模, 字节正序 / 32 字节
8. -----*/
9.
10. 0x02, 0x00, 0x01, 0x00, 0x3F, 0xFC, 0x20, 0x04, 0x40, 0x08, 0x1F, 0xE0, 0x00, 0x40, 0x00, 0x80,
11. 0xFF, 0xFF, 0x7F, 0xFE, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x05, 0x00, 0x02, 0x00,
12. };
```

在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模。如果使用 LCD 的画点函数，按位来扫描这些字模数据，把为 1 的位以黑色来显示（也可以使用其他颜色），即可把整个点阵还原出来，显示在液晶屏上。

22.2 制作字模

我们采用“字模 III- 增强版 V3.91”软件来制作中文字库，步骤如下：

- 1) 打开字模软件，界面见图 22-2。
- 2) 点击“自动批量生成字库”按钮选项。软件界面左下角将出现几个按钮选项，见图 22-3。
- 3) 点击选择“二级汉字库”按钮。在“输入批量字符”框里面将会列出二级汉字的所有汉字，其中共收录了 6768 个汉字字符，非特殊情况下都能够满足大家的要求，见图 22-4。
- 4) 点击“字库智能生成”按钮，弹出“字库批量参数确认”对话框。

我们在“源字体”选项里面做如图 22-5 所示设置。需要注意的是大小问题，因为我们本次的设计目标是实现 16×16 的汉字，所以在此选择“小四”字体。设置好之后如图 22-6 所示。

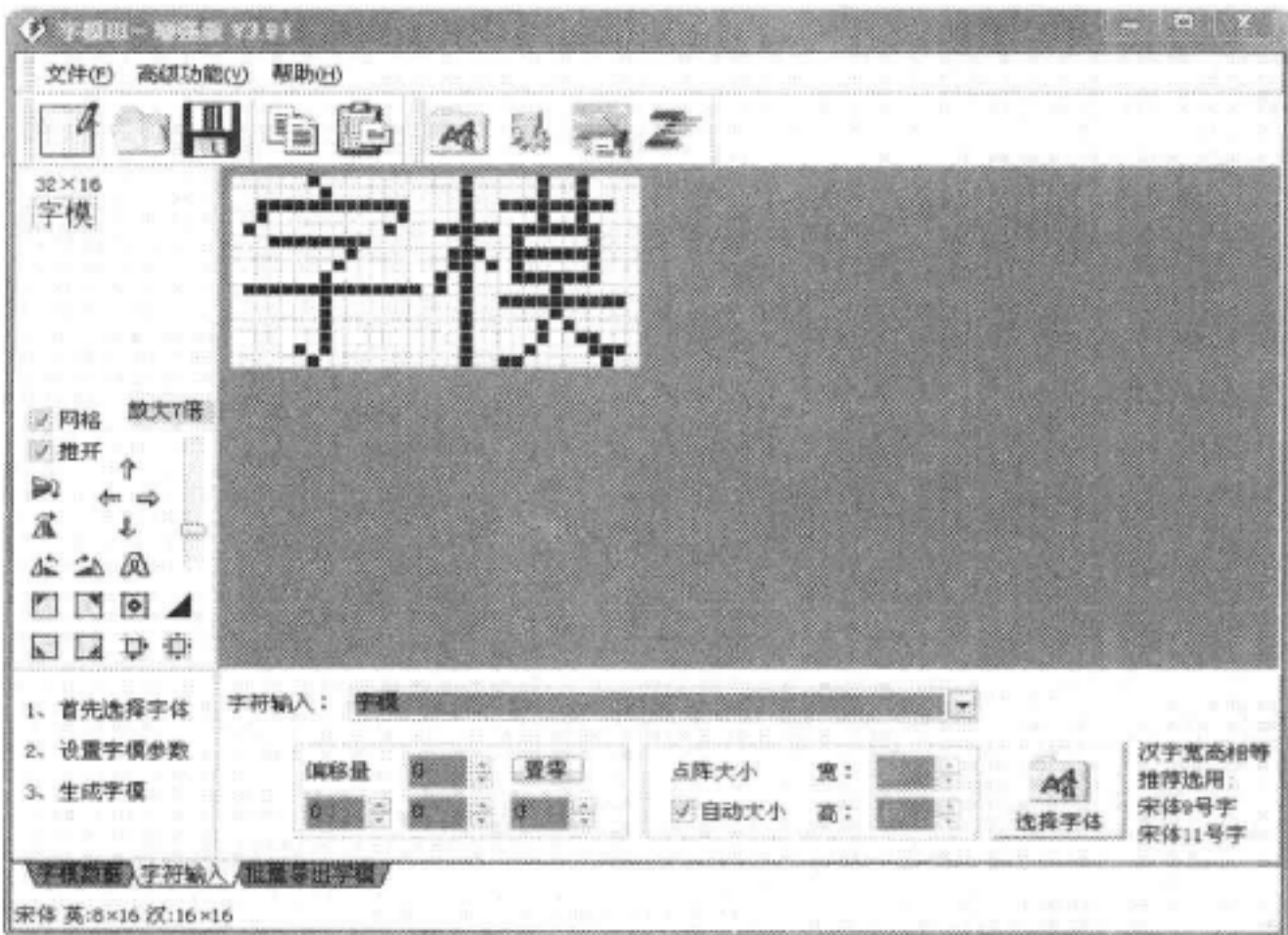


图 22-2 字模软件界面



图 22-3 字库选项

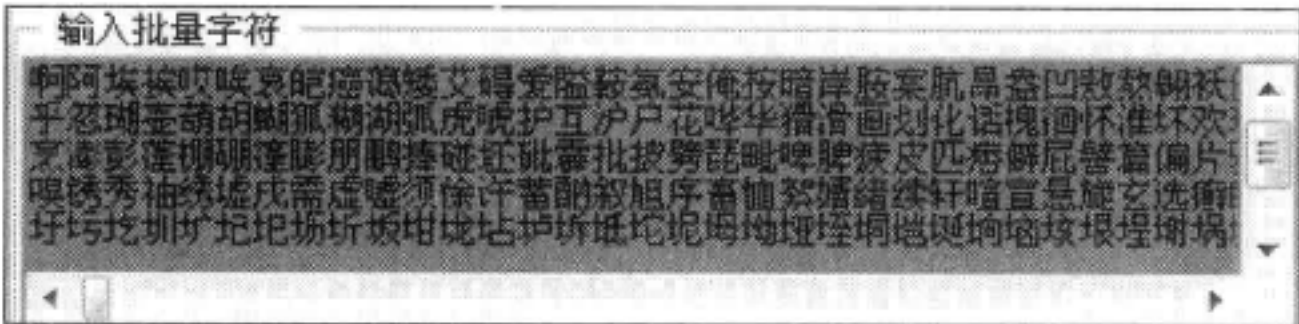


图 22-4 字库

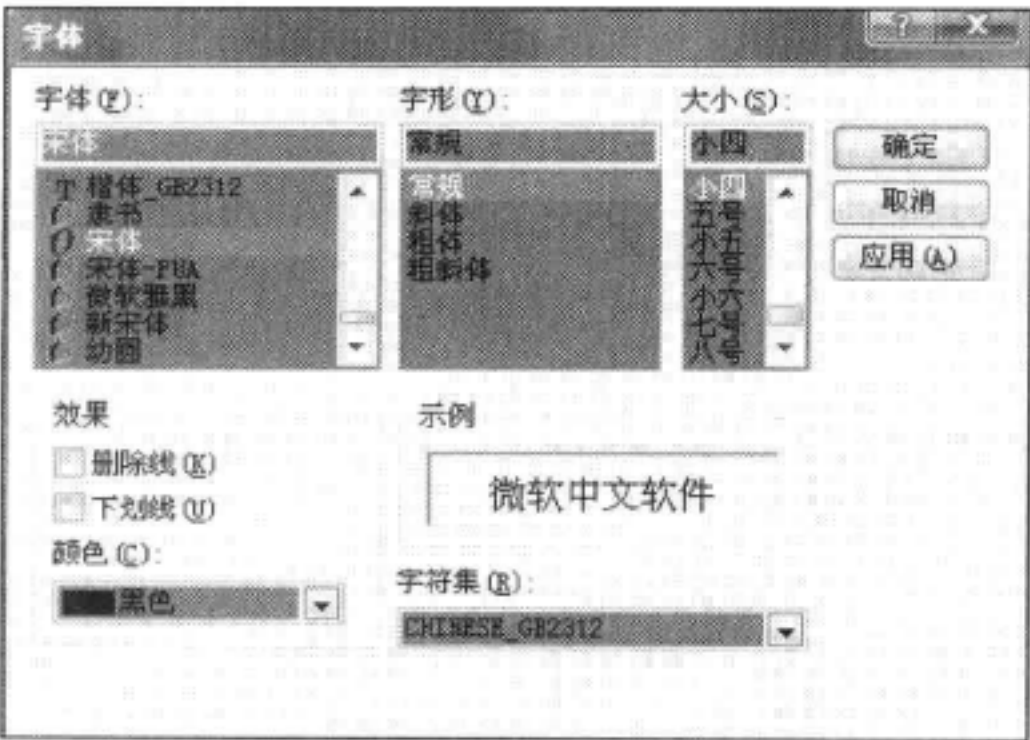


图 22-5 设置字体

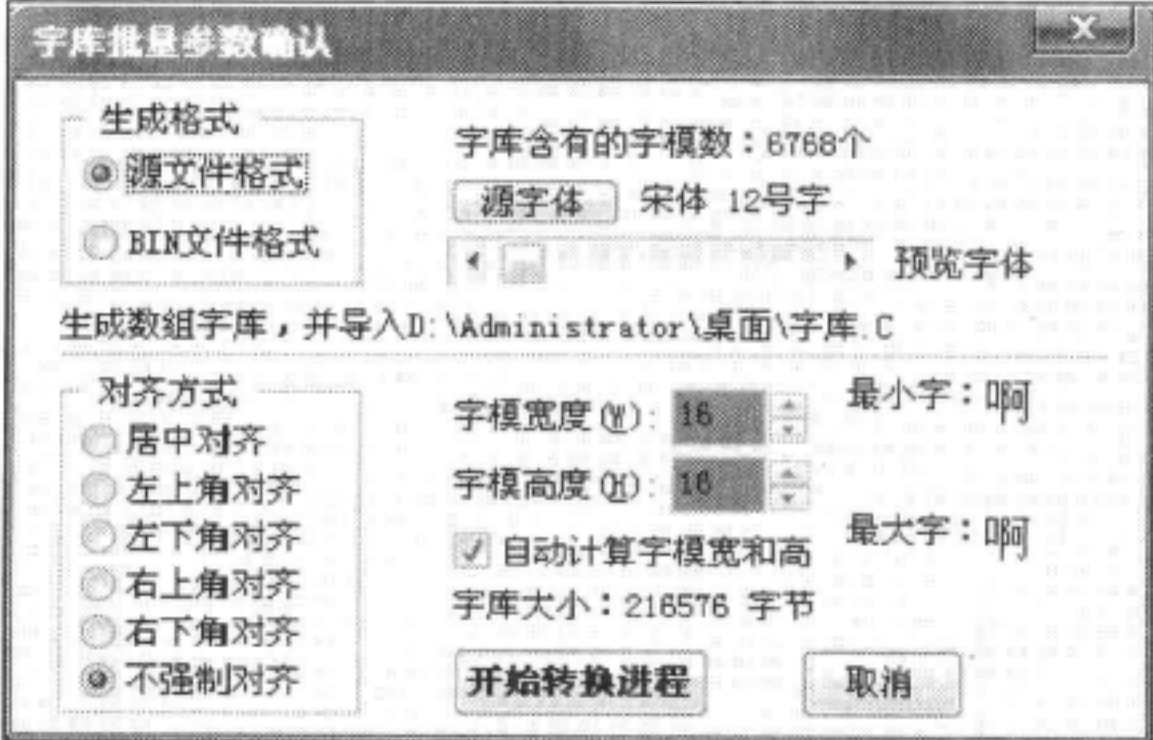


图 22-6 转换字库

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

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

在这里我们取“当”字符的数据来测试, 见图 22-7。

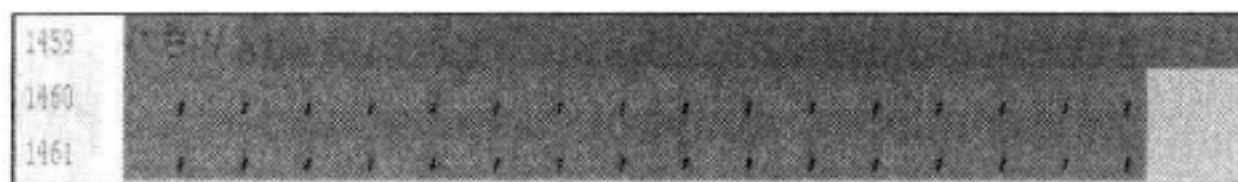


图 22-7 “当”字的字模

VC 6.0 测试源码如下, 该代码实现了把字模中为 1 的点都用数字“8”来表示, 见代码清单 22-2。

代码清单 22-2 “当”字字模示例

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



```
33.
34.         if( kk & 0x80)           // 如果最高位为 1
35.         {
36.             printf("8");
37.         }
38.         else
39.         {
40.             printf(" ");
41.         }
42.
43.     }
44.
45.     printf("\n");
46.
47. }
48. printf("\n\n");
49.
50. }
51.
52.
```

测试结果见图 22-8。

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

7) 回到“字模 III- 增强版 V3.91”软件，采用与之前同样的方式生成 bin 格式的字库文件 (即“生成格式”选项设置为“BIN 文件格式”)，见图 22-9。

在软件安装目录下会生成 Font.dat 文件，我们用 WinHex 软件查看具体内容，与刚才制作的 .c 字库的文件内容是一致的，对比如图 22-10。



图 22-8 测试结果

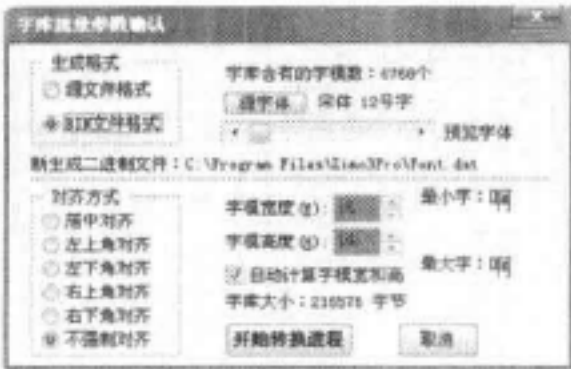


图 22-9 生成 bin 文件格式

将生成的汉字字库复制到 SD 卡根目录下并重命名为“HZLIB.bin”。把该文件保存到 SD 卡中，STM32 芯片通过文件系统读取文件即可获得字库的数据。

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

图 22-12 测试图片的原始数据

1. 文件头部信息部分

见图 22-13，阴影部分是文件头部信息（前面 54 字节）。它可以分为两块：BMP 文件头和位图信息头。

(1) 0 ~ 1 字节

BMP 文件的 0 和 1 字节用于表示文件的类型。如果是位图文件类型，必须分别为 0x42 和 0x4D，0x424D='BM'。

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

图 22-13 文件头部信息

(2) 3 ~ 14 字节

3 到 14 字节的意义可以用一个结构体来描述。见代码清单 22-3。

代码清单 22-3 BMP 文件信息数据结构体

```
1. typedef struct tagBITMAPFILEHEADER
2. {
3.     //attention: sizeof(DWORD)=4   sizeof(WORD)=2
4.     DWORD bfSize;           // 文件大小
5.     WORD bfReserved1;      // 保留字，不考虑
6.     WORD bfReserved2;      // 保留字，同上
7.     DWORD bfOffBits;       // 实际位图数据的偏移字节数，即前三个部分长度之和
8. } BITMAPFILEHEADER,tagBITMAPFILEHEADER;
```

(3) 14 ~ 53 字节

头部信息剩下的部分就是位图信息头，第 14 到第 53 字节内容的意义见代码清单 22-4。

代码清单 22-4 位图信息头内容

```

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

```

由上述分析与 WinHex 软件的分析内容结合得到该图片的信息如下:

- ☐ 文件大小: 438
- ☐ 保留字: 0
- ☐ 保留字: 0
- ☐ 实际位图数据的偏移字节数: 54
- ☐ 结构体的长度: 40
- ☐ 位图宽: 15
- ☐ 位图高: 8
- ☐ biPlanes 平面数: 1
- ☐ biBitCount 采用颜色位数: 24
- ☐ 压缩方式: 0
- ☐ biSizeImage 实际位图数据占用的字节数: 0
- ☐ X 方向分辨率: 3780
- ☐ Y 方向分辨率: 3780
- ☐ 使用的颜色数: 0
- ☐ 重要颜色数: 0

2. 图像像素数据部分

如果是 24 位真彩色, 则 54 字节之后就是像素部分, 见图 22-14。

有些读者可能已经发现, 在像素部分夹杂着一些值为 0 的数据信息, 如图 22-15 灰色部分所示。

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

图 22-14 图像像素数据

00000096	31	31	31	00	00	00	31	31	31	31	31	31	31	31	31
----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

图 22-15 像素中夹杂的 0 数据信息

本例子中整张图片都是灰色的，为什么会有 0 像素的出现呢？

对齐的数据更容易被操作系统或者硬件调入 Cache，使得 Cache 的命中率提高，从而提高访问效率。也就是基于性能上的考虑，Windows 规定一个扫描行所占的字节数必须是 4 的倍数（即以 long 为单位），不足的以 0 填充。由前面位图信息头的分析可知，位图宽为 15 个像素点，由于是 24 位图，每个像素点由 3 个字节构成。因此，未补充字节前字节数为 15×3 等于 45 字节，要到 4 的倍数，必须向上取 4 的倍数即 48，所以就有了补上 3 个字节 0 的结果。

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

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

下面我们修改图片来验证一下：我们将左下角画上白色，右上角画上黑色，图片放大之后如图 22-16，图片数据分析见图 22-17。

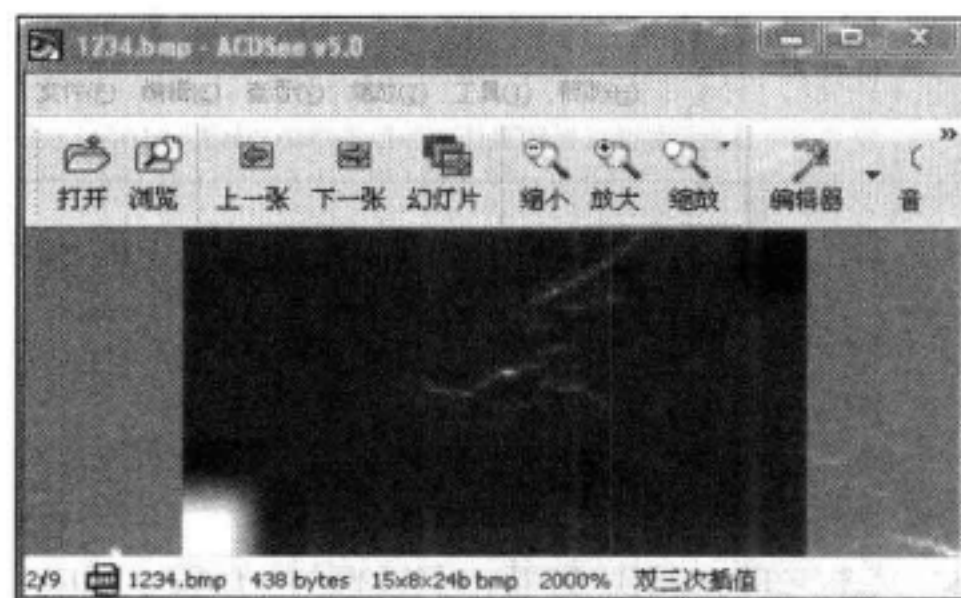


图 22-16 图片放大结果

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

图 22-17 图像数据分析

我们可以看到像素开始部分是白色像素（灰色部分），对应我们图像的左下角，见图 22-18。

00000048	00	00	00	00	00	00	FF	FF	FF	31	31	31	31	31	31	31	31
----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

图 22-18 白色像素

后尾部分是黑色像素（灰色部分）对应我们图像的右上角，见图 22-19。

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

图 22-19 黑色像素

对 BMP 图像文件内容有了具体的了解之后就可以开始编写我们的应用程序，根据图片的头部信息，合理地读出其中的像素部分，把读出的像素点数据送到 LCD 屏，就可以显示出该图片了。

22.4 显示中英文及 BMP 图片实验

22.4.1 实验描述及工程文件清单

1. 实验描述

使用软件制作自定义类型的字库，然后将字库放入 SD 卡中，并且在 SD 卡中放入三张 BMP 图片。最后调用截屏函数截取 LCD 背景并保存为 BMP 图片。

2. 硬件连接

本实验的硬件连接包括 MicroSD 卡控制信号、TFT 控制信号线及触摸屏 TSC2046 控制线，这些连接与前面章节的一样，可参照前面的电路图。

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/misc.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ Fwlib/stm32f10x_systick.c
- ☐ FWlib/stm32f10x_exti.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_sdio.c
- ☐ FWlib/stm32f10x_dma.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_fsmc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/systick.c
- ☐ USER/usart1.c
- ☐ USER/lcd.c
- ☐ USER/ff.c
- ☐ USER/sdio_sdcard.c
- ☐ USER/lcd_botton.c
- ☐ USER/Sd_bmp.c
- ☐ USER/sd_fs_app.c

5. 文件系统源文件

使用 FATFS0.09 版本源码：

- ☐ ff9/diskio.c
- ☐ ff9/ff.c
- ☐ ff9/cc936.c

22.4.2 配置工程环境

本实验需要制作字库，其文件名为 HZLIB.bin，3 个 BMP 图片文件，文件名为 pic1.bmp、pic2.bmp、pic3.bmp，把这 4 个文件保存到 SD 卡中，再把该 SD 卡插入开发板的 SD 卡接口。

本实验中要把旧文件：systick.c、usart1.c、lcd.c、ff.c、sdio_sdcard.c、lcd_botton.c 文件添加进新工程，新建 Sd_bmp.c、sd_fs_app.c 文件，分别用于编写 BMP 文件相关的函数和字模获取函数。

22.4.3 main 文件

从 main 函数开始，它调用了很多函数，用于显示 BMP 图和沿各种方向排列的文字。见代码清单 22-5。

代码清单 22-5 图片显示例程的 main 文件

```

1. int main(void)
2. {
3.
4.     /* USART1 config */
5.     USART1_Config();
6.     SysTick_Init();
7.     LCD_Init();           /* LCD 初始化 */
8.     sd_fs_init();
9.
10.    /* 显示图像 */
11.    Lcd_show_bmp(0, 0, "/pic3.bmp");

```

```

12.    Lcd_show_bmp( 0,0,"/pic2.bmp");
13.    Lcd_show_bmp( 0,0,"/pic1.bmp");
14.
15.    /* 横屏显示 */
16.    LCD_Str_O(20, 10, "LCD_DEMO",0);
17.    LCD_Str_CH(20,30,"阿莫论坛野火专区",0,0xffff);
18.    LCD_Str_CH_O(20,50,"阿莫论坛野火专区",0);
19.    LCD_Num_6x12_O(20, 70, 65535, BLACK);
20.    LCD_Str_6x12_O(20, 90,"LOVE STM32", BLACK);
21.
22.    /* 竖屏显示 */
23.    LCD_Str_O_P(300, 10, "Runing", 0);
24.    LCD_Str_CH_P(280,10,"阿莫论坛野火专区欢迎你",0xff,0xffff);
25.    LCD_Str_CH_O_P(260,10,"阿莫论坛野火专区",0);
26.    LCD_Str_6x12_O_P(240, 10,"LOVE STM32", 0);
27.    LCD_Str_ENCH_O_P(220,10,"欢迎使用野火stm32开发板",0);
28.
29.    /* 截图 */
30.    LCD_Str_CH(20,150,"正在截图",0,0xffff);
31.    Screen_shot(0, 0, 240, 320, "/myScreen");
32.    LCD_Str_CH(20,150,"截图完成",0,0xffff);
33.
34.
35.    while (1)
36.    {}
37.}

```

22.4.4 显示汉字

这里先说汉字字符的显示，第17行：

LCD_Str_CH (20,30,"阿莫论坛野火专区",0,0xffff)；

该函数功能是显示汉字字符串，源码见代码清单22-6。

代码清单22-6 LCD_Str_CH() 函数

```

1.  /*****
2.  * 函数名：LCD_Str_CH
3.  * 描述   ：在指定坐标处显示16*16大小的指定颜色汉字字符串
4.  * 输入    ：    - x：显示位置横向坐标
5.  *          - y：显示位置纵向坐标
6.  *          - str：显示的中文字符串
7.  *          - Color：字符颜色
8.  *          - bkColor：背景颜色
9.  * 输出    ：无
10. * 举例    ：    LCD_Str_CH(0,0,"阿莫论坛野火专区",0,0xffff);
11.                LCD_Str_CH(50,100,"阿莫论坛野火专区",0,0xffff);
12.                LCD_Str_CH(320-16*8,240-16,"阿莫论坛野火专区",0,0xffff);
13. * 注意    ：    字符串显示方向为横向 已测试
14. *****/
15. void LCD_Str_CH(u16 x,u16 y,const u8 *str,u16 Color,u16 bkColor)

```

```

16.{
17.
18.    Set_direction(0);
19.    while(*str != '\0')
20.    {
21.        if(x>(320-16))
22.        {
23.            /* 换行 */
24.            x =0;
25.            y +=16;
26.
27.        }
28.        if(y >(240-16))
29.        {
30.            /* 重新归零 */
31.            y =0;
32.            x =0;
33.        }
34.        LCD_Char_CH(x,y,str,Color,bkColor);
35.        str += 2 ;
36.        x += 16 ;
37.    }
38.}

```

该函数其实没做什么工作，对超出屏幕范围的显示坐标进行换行处理，并把字符串中的汉字一个个提取出来，调用单字符显示函数 LCD_Char_CH() 显示出来，LCD_Char_CH() 函数的源码见代码清单 22-7。

代码清单 22-7 LCD_Char_CH() 函数

```

1.  /*****
2.   * 函数名：LCD_Char_CH
3.   * 描述   ：显示单个汉字字符
4.   * 输入    ：    x: 0 ~ (319-16)
5.   *          y: 0 ~ (239-16)
6.   *          str: 中文字符串首址
7.   *          Color: 字符颜色
8.   *          bkColor: 背景颜色
9.   * 输出    ：无
10.  * 举例    ：    LCD_Char_CH(200,100,"好",0,0);
11.  * 注意    ：如果输入大于1的汉字字符串，显示将会截断，只显示最前面一个汉字
12. *****/
13. void LCD_Char_CH(u16 x,u16 y,const u8 *str,u16 Color,u16 bkColor)
14. {
15.
16. #ifndef NO_CHNISEST_DISPLAY          /* 如果汉字显示功能没有关闭 */
17.     u8 i,j;
18.     u8 buffer[32];
19.     u16 tmp_char=0;
20.
21.

```



```

22.  GetGBKCode_from_sd(buffer,str);    /* 取字模数据 */
23.
24.  for (i=0;i<16;i++)
25.  {
26.      tmp_char=buffer[i*2];
27.      tmp_char=(tmp_char<<8);
28.      tmp_char|=buffer[2*i+1];
29.      for (j=0;j<16;j++)
30.      {
31.          if ( (tmp_char >> 15-j) & 0x01 == 0x01)
32.          {
33.              LCD_ColorPoint(x+j,y+i,Color);
34.          }
35.          else
36.          {
37.              LCD_ColorPoint(x+j,y+i,bkColor);
38.          }
39.      }
40.  }
41.
42. #endif
43. }

```

函数中的条件编译“#ifndef NO_CHNISEST_DISPLAY”，是用于开关汉字显示功能的，若定义了NO_CHNISEST_DISPLAY，则本函数为空，关闭了显示汉字的功能。

在LCD_Char_CH()这个函数中，首先调用GetGBKCode_from_sd()从SD卡中读出我们需要显示在LCD上的指定汉字的字模数据。

接着第22～40行的代码根据字模数据来描写，将字模中为1的数据位，在LCD屏的像素点中使用画点函数LCD_ColorPoint()显示特定的颜色。思路与前面VC测试部分用数字“8”来显示“当”字是一样的。

查找字模

读者现在可能在想，字库里面保存着大量的汉字字幕信息，现在输入GetGBKCode_from_sd(buffer,str)就能够复制这个字符的字模数据，它是怎样定位字模信息所在位置的呢？换句话说，假如现在要显示“吾”字，怎样根据这个字来确定“吾”字符在字库中保存位置的呢？其实这里面有一定的映射关系，那就是接下来要讲的汉字“区码”和“位码”。

在国标GB2312—80中规定，所有的国标汉字及符号在字库中的存储形式是：分配在一个94行94列的阵列中，阵列的每一行称为一个“区”，共有01区到94区；每一列称为一个“位”，共有01位到94位，阵列中的每一个汉字和符号所在的区号和位号组合在一起形成的四个阿拉伯数字就是它们的“区位码”。区位码的前两位是它的区号，后两位是它的位号。01～09区是特殊符号，10～15区未编码，16～55区是一级汉字，56～87区是二级汉字。例如，汉字“啊”的区位码是1601，汉字“阿”的区位码为1602。

在前面生成的HZLIB.bin文件中，生成的字库是按国标GB2312编排的，包含了一级汉字和二级汉字，但不包含01～15区。所以，汉字“啊”存储在该文件的第0个位置中，汉字“阿”

存储在该文件的第 1 个位置中。

汉字的机内码是指在计算机中表示一个汉字的编码。为避免与 ASCII 码混淆。用机内码的两个字节表示一个汉字，这两个字节分别称为高位字节和低位字节。

高位字节 = 区码 + 20H + 80H (或区码 + A0H) 低位字节 = 位码 + 20H + 80H (或位码 + A0H)

因此，我们就可以通过汉字的机内码，运算得出汉字在字库中的区位码，由区位码和文件中的偏移地址查找出该汉字的字模。

下面以 VC 6.0 的测试源码来说明机内码、区位码的关系，见代码清单 22-8。

代码清单 22-8 机内码、区位码的关系

```

1. #include <stdio.h>
2. void main ()
3. {
4.     unsigned char * s , * e = "A" , * c = "古" ;
5.     unsigned char high_byte,lower_byte;          // 内码高字节, 内码低字节
6.     printf ( " 字母 '%s' 的 ASCII 码 '=", e ) ;
7.     s = e ;
8.
9.     while ( * s != 0 )                            //C 的字符串以 0 为结束符 *
10.    {
11.        printf ( "%3d," , *s ) ;
12.        s ++ ;
13.    }
14.    printf ( "\n 汉字内码 (10 进制) '%s'=", c ) ;
15.
16.    s = c ;
17.    while ( *s != 0 )
18.    {
19.        printf ( "%3d," , * s );
20.        s ++ ;
21.    }
22.
23.    printf ( "\n 汉字内码 (16 进制) '%s'=", c ) ;
24.
25.    s = c ;
26.    while ( *s != 0 )
27.    {
28.        printf ( "%0X," , * s );
29.        s ++ ;
30.    }
31.
32.
33.    s = c ;
34.    high_byte = *s;
35.
36.    s ++ ;
37.    lower_byte = *s;
38.

```

```

39.     printf("\n\n 汉字 '%s' 对应的\n 内码高字节:%d\n 内码低字节:%d\n",c,high_byte,lower_byte);
40.     printf("\n\n 汉字 '%s' 对应的\n 区码为:%d-160 = %d\n 位码为:%d-160 = %d\n",c,high_
    byte,high_byte-160,lower_byte,lower_byte-160);
41.
42.     printf("\n\n 汉字 '%s' 在区位码表中的位置为 %d%d\n",c,high_byte-160,lower_byte-160);
43.     printf(" 汉字区位码表可参考网站:http://cs.scu.edu.cn/~wangbo/others/quweima.htm\n");
44.     printf(" 通过在线查阅, 编号为 %d%d对应的汉字刚好就是 '%s'\n\n",high_byte-160,lower_
    byte-160,c);
45.
46.}

```

测试结果见图 22-20。

打开汉字区位码表在线查询网站：

<http://www.jscj.com/index/gb2312.php>,
查询“古”汉字的区位码刚好如计算
所得。

上面的测试结果说明了每一个汉字的内码具体作用。回到本实验工程中获取字模函数 `GetGBKCode_from_sd()` 中, 它的具体定义见代码清单 22-9。



图 22-20 区位码测试效果

代码清单 22-9 `GetGBKCode_from_sd()` 函数

```

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

```



```

22.    {
23.        f_lseek (&myfsrc, pos);                // 制定读取位置
24.        myres = f_read( &myfsrc, pBuffer, 32, &mybr );    //16*16 大小的汉字其字模
        占用 16*2 个字节
25.        f_close(&myfsrc);
26.
27.        return 0;
28.    }
29.
30.    else
31.        return -1;
32.
33.}

```

第 17 行的： $pos = ((High8bit-0xb0) \times 94 + Low8bit-0xa1) \times 2 \times 16$ ，该语句就是根据约定的映射关系，由汉字内码求得该汉字字模在字库文件中的存放位置，之后就到指定的位置去复制字模数据就可以了。

以汉字“啊”为例，解释该公式的含义。“啊”是 GB2312 标准中的第一个汉字，它的高位字节码 (High8bit) 为 16，低位字节码 (Low8bit) 为 01，而它的字模存放在我们的字库文件 HZLIB.bin 中的第 0 个位置。由 High8bit-0xa0 从机内码的高位字节求得其区码 16，再由区码 16 减去 0x10 (即公式中 High8bit-0xb0)，求得它在字库文件中的区码部分位置是 0。公式中还要乘以 94，94 表示一个区有 94 位，由 Low8bit-0xa0 从机内码的低位字节求得其位码 1，再由位码减去 0x01 (即公式中的 Low8bit-0xa1)，求得它在字库文件中的位码部分位置是 0，利用公式求得“啊”字的字模位置。

以上就是利用 SD 卡字库实现 LCD 显示汉字的具体流程。对于 ASCII 码 (包括英文字符) 的显示，实际上原理也是一样的，由于 ASCII 码占用空间较少，所以我们直接把它字模数据以数组的形式存储在代码中，这些数组在文件 ascii.h 和 asc_font.h 中定义。

22.4.5 在 SD 卡上读取与保存 BMP 图像

1. 显示 BMP 图

再回到前面的 main 函数，其中调用了一个 BMP 图片显示函数 Lcd_show_bmp()，其定义见代码清单 22-10。

代码清单 22-10 Lcd_show_bmp() 函数

```

1.  /*****
2.  * 函数名：Lcd_show_bmp
3.  * 描述   ：LCD 显示 RGB888 位图图片
4.  * 输入   ：x                -- 显示横坐标 (0-319)
5.             y                -- 显示纵坐标 (0-239)
6.  *             pic_name       -- 图片名称
7.  * 输出   ：无
8.  * 举例   ：Lcd_show_bmp(0, 0, "/test.bmp");
9.  * 注意   ：图片位 24 位真彩色位图图片
10.             图片宽度不能超过 320
11.             图片在 LCD 上的粘贴范围为：纵向： [x, x+ 图像高度]    横向 [Y, Y+ 图像宽度]
12.             当图片为 320*240 时 -- 建议 X 输入 0   y 输入 0

```

```

13. *****/
14. void Lcd_show_bmp(unsigned short int x, unsigned short int y, unsigned char *pic_name)
15. {
16.     int i, j, k;
17.     int width, height, l_width;
18.
19.     BYTE red, green, blue;
20.     BITMAPFILEHEADER bitHead;
21.     BITMAPINFOHEADER bitInfoHead;
22.     WORD fileType;
23.
24.     unsigned int read_num;
25.     unsigned char tmp_name[20];
26.     sprintf((char*)tmp_name, "0:%s", pic_name);
27.     f_mount(0, &bmpfs[0]);
28.
29.     BMP_DEBUG_PRINTF("file mount ok \r\n"); // 使用串口输出调试信息
30.
31.     bmpres = f_open(&bmpfsrc, (char *)tmp_name, FA_OPEN_EXISTING | FA_READ);
32.     Set_direction(0);
33.
34.     if(bmpres == FR_OK)
35.     {
36.         BMP_DEBUG_PRINTF("Open file success\r\n");
37.
38.         // 读取位图文件头信息
39.         f_read(&bmpfsrc, &fileType, sizeof(WORD), &read_num);
40.
41.         if(fileType != 0x4d42)
42.         {
43.             BMP_DEBUG_PRINTF("file is not .bmp file!\r\n");
44.             return;
45.         }
46.         else
47.         {
48.             BMP_DEBUG_PRINTF("Ok this is .bmp file\r\n");
49.         }
50.
51.         f_read(&bmpfsrc, &bitHead, sizeof(tagBITMAPFILEHEADER), &read_num);
52.
53.         showBmpHead(&bitHead);
54.         BMP_DEBUG_PRINTF("\r\n");
55.
56.         // 读取位图信息头信息
57.         f_read(&bmpfsrc, &bitInfoHead, sizeof(BITMAPINFOHEADER), &read_num);
58.         showBmpInforHead(&bitInfoHead);
59.         BMP_DEBUG_PRINTF("\r\n");
60.     }
61.     else
62.     {
63.         BMP_DEBUG_PRINTF("file open fail!\r\n");
64.         return;

```

```

65.     }
66.
67.     width = bitInfoHead.biWidth;
68.     height = bitInfoHead.biHeight;
69.
70.     l_width = WIDTHBYTES(width* bitInfoHead.biBitCount);           // 计算位图的实际宽度
        并确保它为 32 的倍数
71.
72.     if(l_width>960)
73.     {
74.         BMP_DEBUG_PRINTF("\nSORRY, PIC IS TOO BIG (<=320)\n");
75.         return;
76.     }
77.
78.     if(bitInfoHead.biBitCount>=24)
79.     {
80.
81.         bmp_lcd(x,240-y-height,width, height);           //LCD 参数相关设置
82.
83.         for(i=0;i<height+1; i++)
84.         {
85.
86.             for(j=0; j<l_width; j++)                       // 将一行数据全部读入
87.             {
88.
89.                 f_read(&bmpfsrc,pColorData+j,1,&read_num);
90.             }
91.
92.             for(j=0;j<width;j++)                           // 一行有效信息
93.             {
94.                 k = j*3;                                     // 一行中第 K 个像素的起点
95.                 red = pColorData[k+2];
96.                 green = pColorData[k+1];
97.                 blue = pColorData[k];
98.                 LCD_WR_Data( RGB24TORGB16(red,green,blue)); // 写入 LCD-GRAM
99.             }
100.        }
101.        bmp_lcd_reset();   //LCD 扫描方向复原
102.    }
103.    else
104.    {
105.        BMP_DEBUG_PRINTF("SORRY, THIS PIC IS NOT A 24BITS REAL COLOR");
106.        return ;
107.    }
108.
109.    f_close(&bmpfsrc);
110. }

```

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

另外一点就是：RGB24TORGB16 是个宏定义，因为图像数据是 RGB888 即 24 位真彩色，而

我们的 LCD 是 RGB565 即 16 位色度的, 所以我们需要按比例将 24 位真彩色压缩为 16 位。宏定义见代码清单 22-11。

代码清单 22-11 图像数据压缩宏

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

2. LCD 截图功能

为了实现截图功能, 我们可以根据用户截屏范围的宽和高来构造 BMP 文件的信息头, 并且根据位图宽与 4 字节对齐的关系来补充 0。在 main 函数中, 我们调用了 Screen_shot() 这个函数来截图。保存的图片是 24 位的真彩色。Screen_shot() 的源码见代码清单 22-12。

代码清单 22-12 Screen_shot() 函数

```
1. /*****
2.  * 函数名: Screen_shot
3.  * 描述 : 截取 LCD 指定位置、指定宽高的像素, 保存为 24 位真彩色 bmp 格式图片
4.  * 输入 : x          --- 水平位置
5.  *       y          --- 竖直位置
6.  *       Width      --- 水平宽度
7.  *       Height     --- 竖直高度
8.  *       filename   --- 文件名
9.  * 输出 : 0          --- 成功
10. *        -1        --- 失败
11. *        8         --- 文件已存在
12. * 举例 : Screen_shot(0, 0, 320, 240, "/myScreen");----- 全屏截图
13. * 注意 : x 范围 [0,319] y 范围 [0,239] Width 范围 [0,320-x] Height 范围 [0,240-y]
14. *          如果文件已存在, 将直接返回
15. *****/
16. int Screen_shot(unsigned short int x, unsigned short int y, unsigned short int
    Width, unsigned short int Height, unsigned char *filename)
17. {
18.     unsigned char header[54] =
19.     {
20.         0x42, 0x4d, 0, 0, 0, 0,
21.         0, 0, 0, 0, 54, 0,
22.         0, 0, 40, 0, 0, 0,
23.         0, 0, 0, 0, 0, 0,
24.         0, 0, 1, 0, 24, 0,
25.         0, 0, 0, 0, 0, 0,
26.         0, 0, 0, 0, 0,
27.         0, 0, 0, 0, 0,
28.         0, 0, 0, 0, 0,
29.         0, 0, 0
30.     };
31.     int i;
32.     int j;
33.     long file_size;
34.     long width;
35.     long height;
```

```

36. unsigned short int tmp_rgb;
37. unsigned char r,g,b;
38. unsigned char tmp_name[30];
39. unsigned int mybw;
40. char kk[4]={0,0,0,0};
41.
42.
43. // if(!(Width%4))
44. //     file_size = (long)Width * (long)Height * 3 + 54;
45. //else
46. file_size = (long)Width * (long)Height * 3 + Height*(Width%4) + 54;
47. // 宽 * 高 + 补充的字节 + 头部信息
48. header[2] = (unsigned char)(file_size & 0x000000ff);
49. header[3] = (file_size >> 8) & 0x000000ff;
50. header[4] = (file_size >> 16) & 0x000000ff;
51. header[5] = (file_size >> 24) & 0x000000ff;
52.
53.
54. width=Width;
55. header[18] = width & 0x000000ff;
56. header[19] = (width >> 8) & 0x000000ff;
57. header[20] = (width >> 16) & 0x000000ff;
58. header[21] = (width >> 24) & 0x000000ff;
59.
60. height = Height;
61. header[22] = height & 0x000000ff;
62. header[23] = (height >> 8) & 0x000000ff;
63. header[24] = (height >> 16) & 0x000000ff;
64. header[25] = (height >> 24) & 0x000000ff;
65.
66. sprintf((char*)tmp_name, "0:%s.bmp", filename);
67. f_mount(0, &bmpfs[0]);
68.
69.
70. bmpres = f_open( &bmpfsrc , (char*)tmp_name, FA_CREATE_NEW | FA_WRITE);
71.
72. f_close(&bmpfsrc); // 新建文件之后要先关闭再打开才能写入
73. bmpres = f_open( &bmpfsrc , (char*)tmp_name, FA_OPEN_EXISTING | FA_WRITE);
74. if ( bmpres == FR_OK )
75. {
76.     bmpres = f_write(&bmpfsrc, header, sizeof(unsigned char)*54, &mybw);
77.     for(i=0;i<Height;i++) // 高
78.     {
79.         if(!(Width%4))
80.         {
81.             for(j=0;j<Width;j++) // 宽
82.             {
83.                 #ifdef HX8347
84.                 tmp_rgb = bmp4(j+y,Height-i+x);
85.                 #else
86.                 tmp_rgb = bmp4(Height-i+x,j+y);
87.                 #endif
88.

```

```

89.         r = GETR_FROM_RGB16(tmp_rgb);
90.         g = GETG_FROM_RGB16(tmp_rgb);
91.         b = GETB_FROM_RGB16(tmp_rgb);
92.
93.         bmpres = f_write(&bmpfsrc, &b, sizeof(unsigned char), &mybw);
94.         bmpres = f_write(&bmpfsrc, &g, sizeof(unsigned char), &mybw);
95.         bmpres = f_write(&bmpfsrc, &r, sizeof(unsigned char), &mybw);
96.
97.     }
98.
99. }
100.     else
101.     {
102.         for(j=0;j<Width;j++)
103.         {
104.             #ifdef HX8347
105.                 tmp_rgb = bmp4(j+y,Height-i+x);
106.             #else
107.                 tmp_rgb = bmp4(Height-i+x,j+y);
108.             #endif
109.
110.             r = GETR_FROM_RGB16(tmp_rgb);
111.             g = GETG_FROM_RGB16(tmp_rgb);
112.             b = GETB_FROM_RGB16(tmp_rgb);
113.
114.             bmpres = f_write(&bmpfsrc, &b, sizeof
(unsigned char), &mybw);
115.             bmpres = f_write(&bmpfsrc, &g, sizeof
(unsigned char), &mybw);
116.             bmpres = f_write(&bmpfsrc, &r, sizeof
(unsigned char), &mybw);
117.
118.
119.         }
120.
121.         bmpres = f_write(&bmpfsrc, kk, sizeof(unsigned char)*
(Width%4), &mybw);
122.
123.
124.     }
125. }
126.
127. f_close(&bmpfsrc);
128. return 0;
129. }
130. else if ( bmpres == FR_EXIST )           // 如果文件已经存在
131. {
132.     f_close(&bmpfsrc);
133.     return FR_EXIST;                       //8
134. }
135.
136. else
137. { f_close(&bmpfsrc);
138.   return -1;
139. }
140. }

```


该函数中，调用了 bmp4() 这个函数，该函数返回 LCD 上指定位置的像素信息。GETG_FROM_RGB16（绿色）、GETB_FROM_RGB16（蓝色）和 GETR_FROM_RGB16（红色）都是宏定义，将 RGB565 即 16 位色度抽取出其中的 RGB 数据并分别将其线性映射为 8 位数据即映射为 RGB888 真彩色。宏定义的内容见代码清单 22-13。

代码清单 22-13 抽取 RGB 数据宏

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

利用 Screen_shot() 函数，就实现了把屏幕的当前显示的图像转换为 BMP 文件的功能。

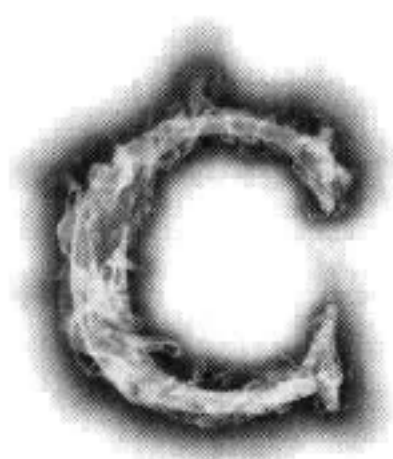
22.4.6 实验现象

把文件 HZLIB.bin、pic1.bmp、pic2.bmp 和 pic3.bmp 保存到 SD 卡中，再把该 SD 卡插入开发板的 SD 卡接口（也可直接把工程下的 SD 字库备份文件夹下的内容复制到 SD 卡的根目录），然后将配套 STM32 开发板供电（DC5V），插上 J-LINK，插上串口线（两头都是母的交叉线），接上液晶屏，将编译好的程序下载到开发板。

程序运行之后截图保存为“myScreen.bmp”，见图 22-21。（截图保存图片功能 + 摄像头模块就构成了一个完整的照相机啦！）



图 22-21 截图结果



第 23 章

OV7670 摄像头驱动

在各种信息中，图像含有最丰富的信息。作为机器视觉领域的核心部件，摄像头被广泛地应用在安防、探险、车牌检测等场合。STM32 的处理速度比传统的 8 位、16 位机快得多，所以使用它驱动摄像头采集图像信息并进行基本的加工处理更为适合，本章讲解使用 STM32 驱动 OV7670 型号的摄像头。

23.1 摄像头的分类

摄像头按照输出信号的类型可以分为数字摄像头和模拟摄像头，按照摄像头图像传感器材料构成可以分为 CCD 和 CMOS。

23.1.1 数字摄像头与模拟摄像头的区别

- 输出信号类型：数字摄像头输出信号为数字信号，模拟摄像头输出信号为标准的模拟信号。
- 接口类型：数字摄像头有 USB 接口（比如常见的 PC 端免驱摄像头）、IEEE1394 火线接口（由苹果公司领导的开发联盟开发的一种高速度传送接口，数据传输率高达 800 Mbps）、千兆网接口（网络摄像头）。模拟摄像头多采用 AV 视频端子（信号线+地线）或 S-VIDEO（即莲花头——SUPER VIDEO），是一种五芯的接口，由两路视频亮度信号、两路视频色度信号和一路公共屏蔽地线共五条芯线组成）。
- 分辨率：模拟摄像头的感光器件，其像素指标一般维持在 752（H）×582（V）左右的水平，像素数一般情况下维持在 41 万左右。数字摄像头分辨率一般从数十万到数百万甚至数千万（比如诺基亚 808 手机搭载了 4100 万像素，被人称为可以打电话的相机）。但这并不能说明数字摄像头的成像分辨率就比模拟摄像头的高，原因在于模拟摄像头输出的是模拟视频信号，一般直接输入至电视或监视器，其感光器件的分辨率与电视信号的扫描数呈一定的换算关系，图像的显示介质已经确定，因此模拟摄像头的感光器件分辨率不是不能做高，而是依据实际情况没必要这么高。

23.1.2 CCD 与 CMOS 的区别

CCD 与 CMOS 成像器主要区别如下：

- 成像原理：CCD 是“电荷耦合器件”（Charge Coupled Device）的简称，而 CMOS 是“互补金属氧化物半导体”（Complementary Metal Oxide Semiconductor）的简称。

- ❑ 功耗：由于 CCD 的像素由 MOS 电容构成，读取电荷信号时需使用电压相当大（至少 12V）的两相、三相或四相时序脉冲信号，才能有效地传输电荷。因此 CCD 的取像系统除了要有多个电源外，其外设电路也会消耗相当大的功率。有的 CCD 取像系统需消耗 2 ~ 5 W 的功率。而 CMOS 光电成像器件只需使用一个单电源 5V 或 3V，耗电量非常小，仅为 CCD 的 1/8 ~ 1/10，有的 CMOS 取像系统只消耗 20 ~ 50 mW 的功率。
- ❑ 成像质量：CCD 成像器件制作技术起步早，技术成熟，采用 PN 结或二氧化硅（SiO₂）隔离层隔离噪声，所以噪声低，成像质量好。与 CCD 相比，CMOS 的主要缺点是噪声高及灵敏度低，不过随着 CMOS 电路消噪技术的不断进步，为生产高密度优质的 CMOS 成像器件提供了良好的条件（现在高级的 CMOS 并不比一般 CCD 差，但是 CMOS 工艺还不是十分成熟，普通的 CMOS 分辨率低而成像较差，太容易出现杂色点），现在主流的单反相机普遍采用 CMOS 成像器。

23.2 OV7670 介绍

OV7670 是一个能够提供单片 VGA 摄像头和影像处理器的所有功能的图像传感器，它支持整帧输出、子采样、取窗口等模式，支持 8/10 位图像分辨率，支持的数据格式有很多种，包括 RAW RGB、RGB (GRB 4:2:2、RGB565/555/444) 以及 YCbCr (4:2:2) 等格式。它的体积小，工作电压低，可以对图像进行伽玛曲线、白平衡、饱和度、色度等处理。

23.2.1 OV7670 功能框架

OV7670 芯片的各个功能模块见图 23-1。

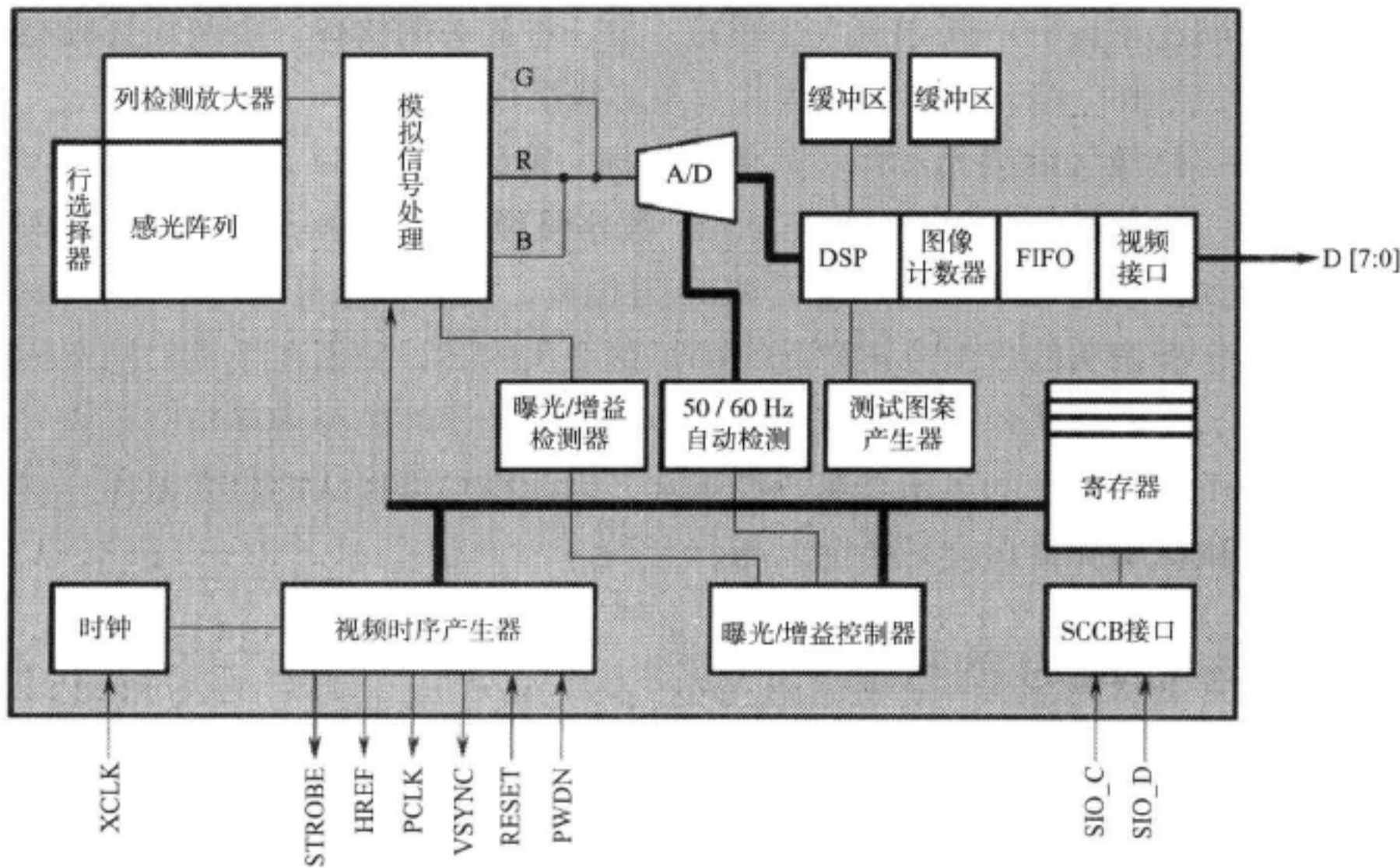


图 23-1 OV7670 功能框架图

OV7670 包含有 656×488 图像阵列，共包含 320128 个像素，但是实际有效的还是 640×480 个，边缘多出来的像素主要是为了去除一些影响边缘失真特性而设立的。图像阵列感知的原始信号（颜色 & 亮度）之后会输入模拟处理器进行处理（比如通常所说的曝光控制和增益控制），经处理后会被分成 G 和 BR 两路通道进入一个 A/D 转换器，转换成数字信号后送入 DSP 处理器做进一步处理。

当然，用户还可以通过相关寄存器来切换选择把测试图案发生器产生的图形数据送入 DSP 处理器处理。DSP 模块在整个框架中起着非常重要的作用，它控制着从原始信号插值到 RGB 信号的整个过程，并且控制图像质量包括边缘增强、颜色空间转换、伽玛控制等，这些操作的实现可能还会依赖于缓冲区的支持，这跟我们平时做大量数据的复杂运算时类似，在这里同样需要开辟一定空间的缓冲区，否则很多中间数据无法安置。

图像处理后会放入一个图像缩放模块，缩放模块按照默认配置或者用户配置的要求，支持将 YUV/RGB 信号从 VGA 缩小到 CIF 以下的任何尺寸，再进入一个 FIFO，最后通过视频端口将数据传递给用户。另外，我们还可以看到，系统时钟经过视频时序发生器后，根据系统或者用户配置，产生了视频采集常见的三种信号：行同步信号、场同步信号、像素时钟；图像传感器内部的寄存器可以通过 SCCB 接口来访问，该内容在 23.3 节我们会单独描述。

23.2.2 OV7670 管脚封装

OV7670 管脚图见图 23-2，主要管脚的定义见表 23-1。

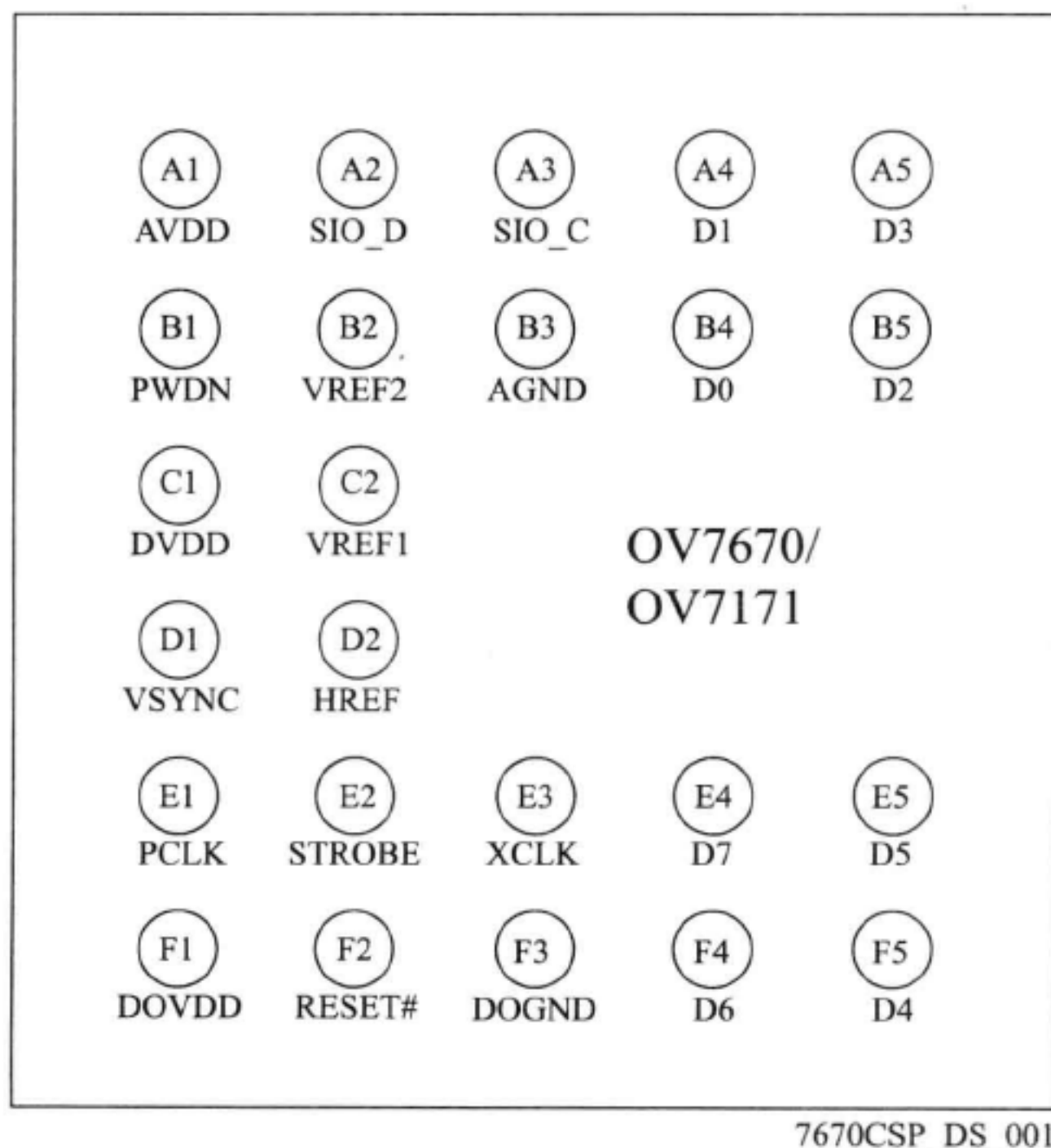


图 23-2 OV7670 管脚图

表 23-1 OV7670 管脚

名 称	类 型	功能 / 说明	名 称	类 型	功能 / 说明
AVDD	电源	模拟电源	VSYNC	输出	帧同步
SIO_D	输入 / 输出	SCCB 数据口	HREF	输出	行同步
SIO_C	输入	SCCB 时钟口	PCLK	输出	像素时钟
D1	输出	数据位 1	STROBE	输出	闪光灯控制输出
D3	输出	数据位 3	XCLK	输入	系统时钟输入
PWDN	输入	POWER DOWN 模式选择： 0：工作；1：关闭	D7	输出	数据位 7
VREF2	参考	参考电压 - 并 0.1μF 电容	D5	输出	数据位 5
AGND	电源	模拟地	DOVDD	电源	I/O 电源，电压（1.7 ~ 3.0）
D0	输出	数据位 0	RESET#	输入	初始化所有寄存器到默认值： 0：复位模式；1：一般模式
D2	输出	数据位 2	DOGND	电源	数字地
DVDD	电源	核电压 +1.8VDC	D6	输出	数据位 6
VREF1	参考	参考电压 - 并 0.1μF 电容	D4	输出	数据位 4

当图像传感器配置为输出 RAW RGB 格式数据时，数字端口 D0 到 D9 都要用到，D0 作为最低有效位，D9 作为最高有效位。当 SENSOR 配置为输出 YUV 或者 RGB 565/RGB 555 格式时，数字端口的 D2 到 D9 才有效，D2 作为最低有效位，D9 作为最高有效位。

23.3 SCCB 总线

SCCB 总线全称为 Serial Camera Control Bus，它的工作方式与 I²C 十分类似，是由 OV 公司定义的 3 线串行摄像头控制总线，可以控制大部分 OV 系列图像传感器。SCCB 亦可以工作在 2 线串行模式（SIOC 与 SIOD），一条 SCCB 总线下可以挂载多个从设备（通过从设备地址来区分），另外 SCCB 还可以附带一根 PWDN 用于关闭或者开启从设备系统。

23.3.1 SCCB 接口定义

□ 标准的 3 线 SCCB 接口定义见表 23-2 和图 23-3。

表 23-2 3 线 SCCB 接口定义

信号名称	信号类型	描 述
SCCB_E	输出	片选信号。当总线处于空闲状态，主机应该将其逻辑电平设为 1；该信号的逻辑电平从 1 变成 0 标志着一个起始信号，从 0 变成 1 标志着一个终止信号
SIO_C	输出	时钟信号。标志着一个位的传输，当总线处于空闲状态，主机应该将其逻辑电平设为 1；当 SCCB_E 信号的裸机电平为 0 时，该管脚的 0、1 变换标志着位的传输

(续)

信号名称	信号类型	描述
SIO_D	I/O	数据信号。当总线为空闲状态时将其设为浮空状态，当系统为挂起状态时将其设为裸机 0。在数据传输阶段，该管脚的逻辑电平变换只能发生在 SIO_C 为低电平时
PWDN	输出	控制从设备状态（附加的信号）

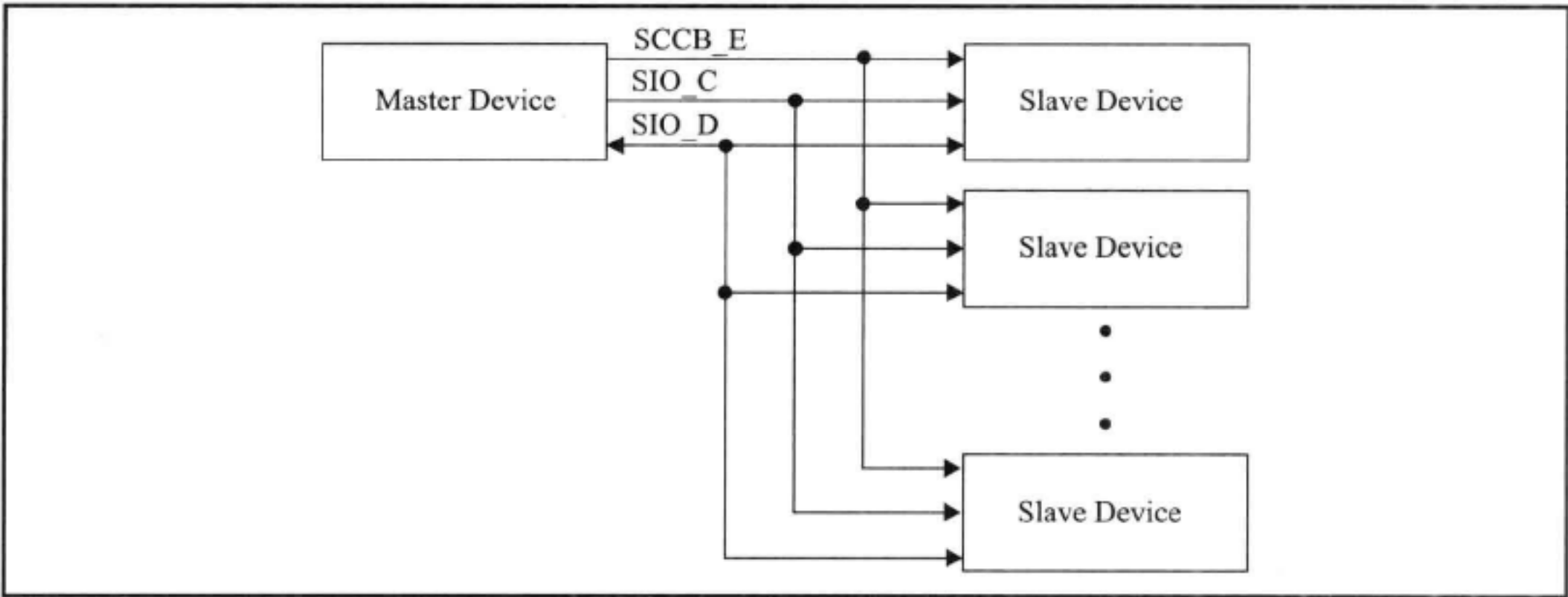


图 23-3 3 线 SCCB 设备连接范例

□ 简化的 2 线 SCCB 接口定义见表 23-3 和图 23-4。

表 23-3 2 线的 SCCB 接口定义

信号名称	信号类型	描述
SIO_C	输出	时钟信号。空闲状态下该管脚的逻辑电平为 1。逻辑 0、1 变换标志着位的传输
SIO_D	I/O	数据信号。当 SIO_C 是高电平时，SIO_D 线由高电平向低电平切换表示起始传输；当 SIO_C 是高电平时，SIO_D 线由低电平向高电平切换表示停止传输。因此在数据传输过程中，只有当 SIO_C 的电平为 0 时，该管脚逻辑电平才能发生改变

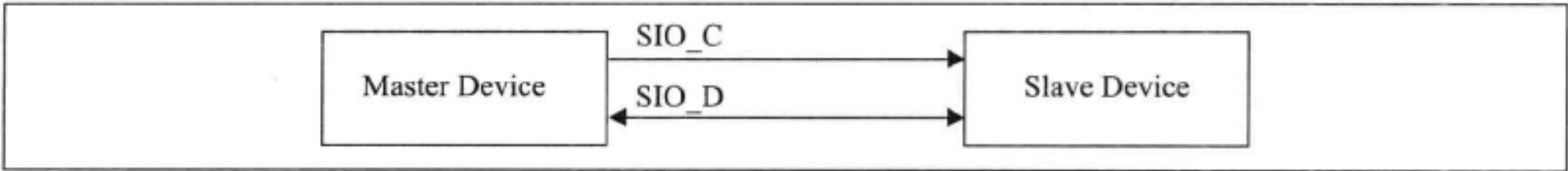


图 23-4 2 线 SCCB 设备连接范例

从表格描述可以看出，2 线 SCCB 接口同样可以满足控制要求，如果需要控制从设备的工作状态，我们可以外加 GPIO 来控制从设备即可。

23.3.2 SCCB 时序描述

读者可以参考 I²C 协议与 SCCB 的时序进行对比，SCCB 时序见图 23-5。

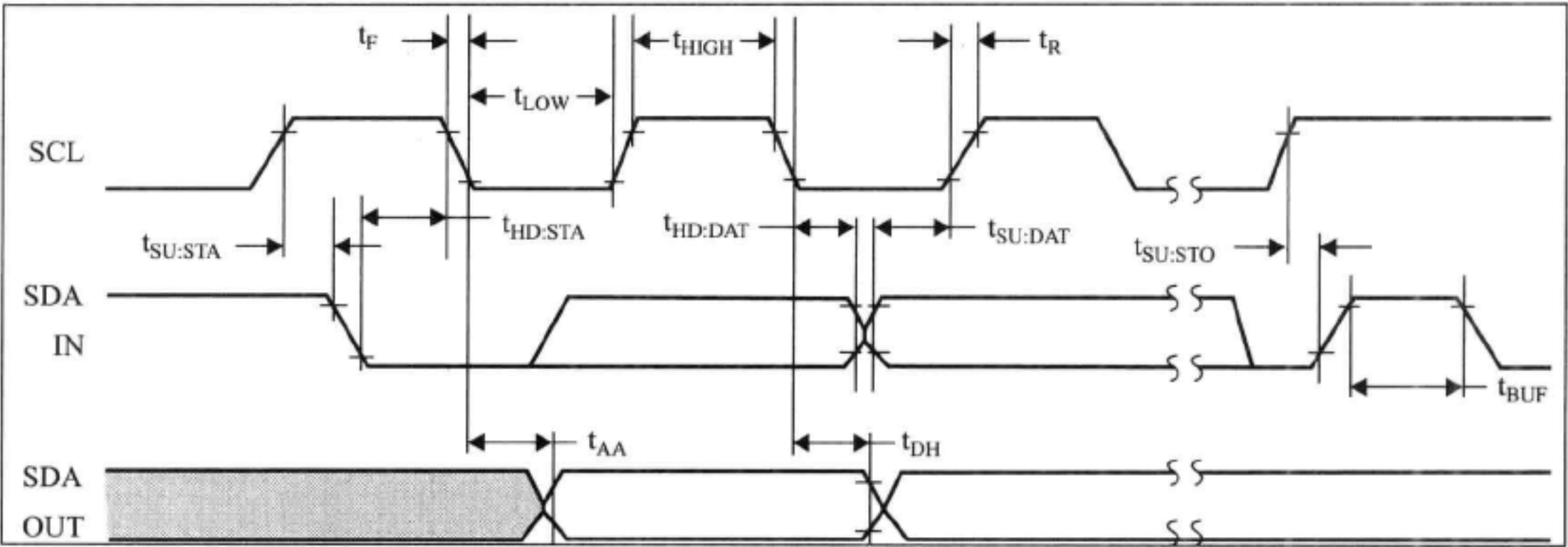


图 23-5 SCCB 总线时序

图 23-5 中所引用的一些符号描述见表 23-4。

表 23-4 时序图描述

符 号	描 述
$t_{SU:STA}$	“START”条件的建立时间
t_F 与 t_R	SCCB 的上升或者下降时间
t_{BUF}	新的“START”信号前总线的空闲时间
$t_{HD:STA}$	“START”条件的保持时间
$t_{SU:STA}$	“START”条件的创建时间
$t_{HD:DAT}$	Data-in 保持时间
$t_{SU:DAT}$	Data-in 建立时间
t_{DH}	Data-out 保持时间

- 起始信号：在 SIO_C 为高电平时，SIO_D 出现一个下降沿，则 SCCB 开始传输。见图 23-6。
- 停止信号：在 SIO_C 为高电平时，SIO_D 出现一个上升沿，则 SCCB 停止传输。见图 23-7。
- 数据传输：除了开始和停止状态，当数据传输时，当 SIO_C 为高电平时，必须保证 SIO_D 上的数据稳定，也就是说，SIO_D 上的电平变换只能发生在 SIO_C 为低电平时。

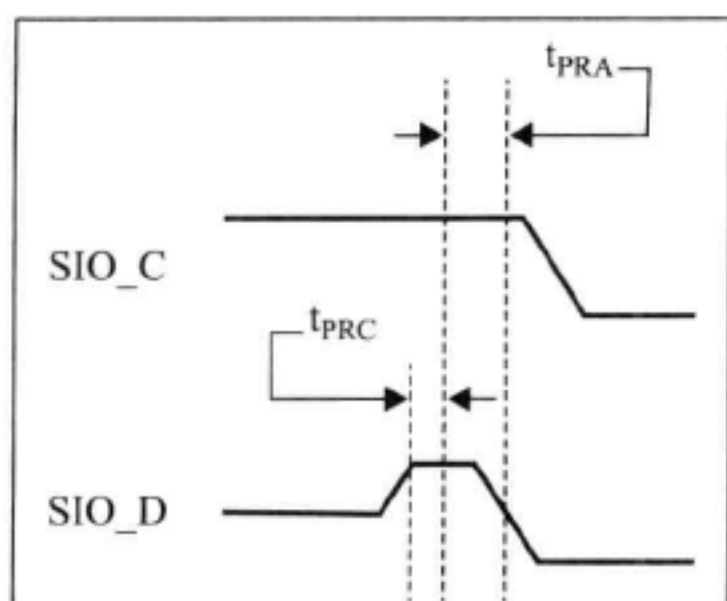


图 23-6 SCCB 起始信号

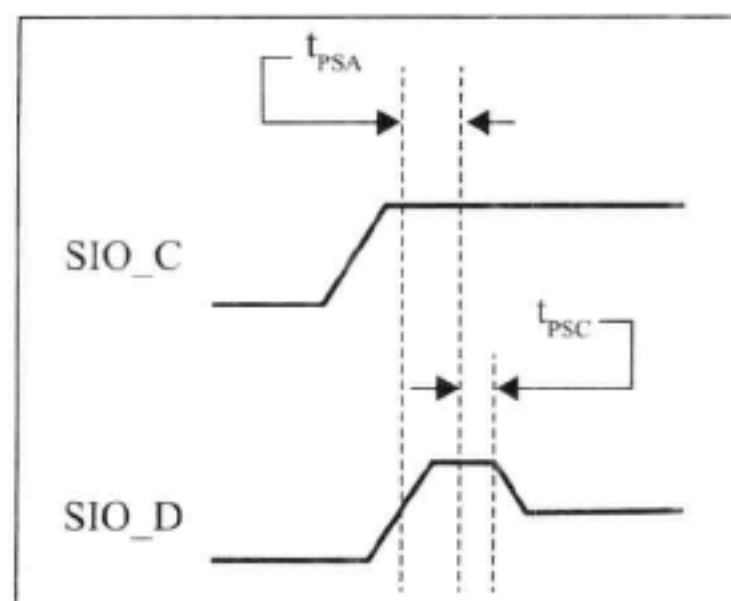


图 23-7 SCCB 停止信号

与 I²C 总线类似，SCCB 基本传输格式如图 23-8 所示，完整的数据传输包括两个或者三个阶段。每一阶段包含 9 位二进制数据，其中高 8 位为所要传输的 8 位数据。第一阶段的 bit0 根据主器件的数据传输是读操作还是写操作而确定。在进行主器件写操作时，全部阶段的最低位均是自由位（Don't care bit），自由位中主机输出高电平，从机输出低电平响应（有些从机不会响应自由位）；读操作时，第一阶段的最低位是自由位，第二阶段的最低位为 NA——主器件驱动为高电平有效。

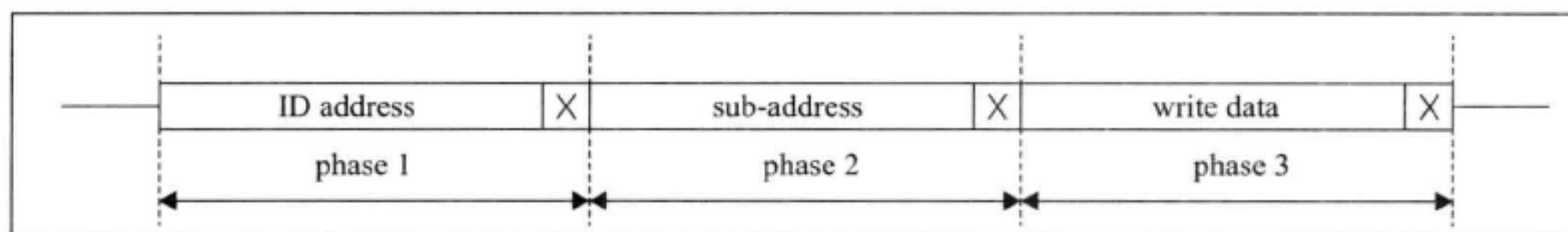


图 23-8 SCCB 的三相写操作

SCCB 协议定义了两种写操作，即三相写操作和两相写操作。三相写操作是向从器件的一个目的寄存器中写入数据，见图 23-8。在三相写操作中，第一阶段写从器件的 8 位 IDW（器件地址 + 写方向标志）和自由位，第二阶段写从器件目标寄存器的 8 位地址和自由位，第三阶段写要求写入寄存器的 8 位数据和自由位；两相写操作没有第三阶段，即只向从器件传输了器件 ID 和目的寄存器的地址，见图 23-9。

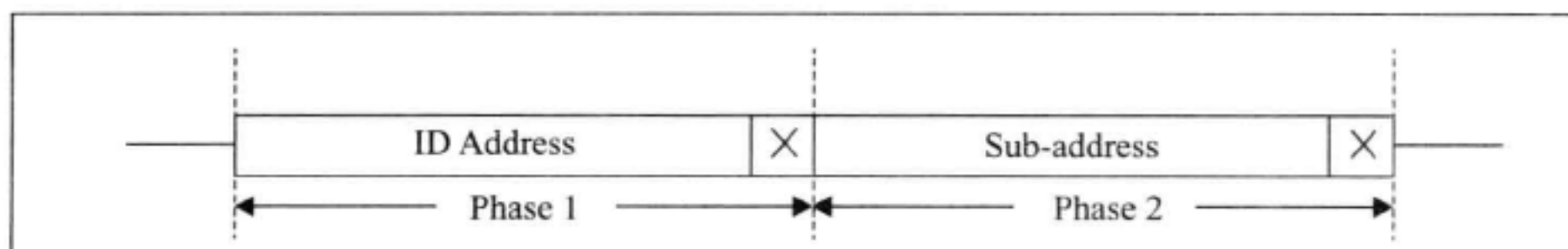


图 23-9 SCCB 的两相写操作

SCCB 协议定义了两相读操作，它用于读取从器件目的寄存器中的数据，见图 23-10。在第一阶段中写从件读操作 8 位 IDR（器件地址 + 读方向标志）和自由位，在第二阶段中读取寄存器中的 8 位数据和写 NA 位。由于两相读操作没有确定目的寄存器的地址，所以在读操作前，必须

有一个两相或三相的写循环操作，以提供读操作中的寄存器地址（该过程类似 EEPROM 例程中的“伪写”）。

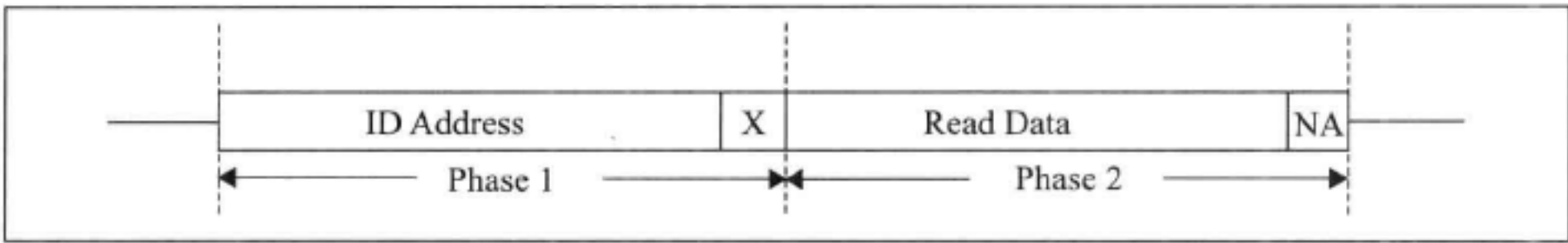


图 23-10 SCCB 的两相读操作

23.4 摄像头模块

23.4.1 摄像头模块硬件介绍

配套 STM32 开发板具有一个 CAMERA 接口，可用于扩展摄像头模块，该接口可兼容野火的 OV7670、OV7725 及市面上其他基于 OV 系列传感器的模块。考虑到一些性能比较低的单片机也能够搭载野火开发的摄像头模块，我们在 OV7670 后端搭载了 FIFO 来降低对单片机的性能依赖——当前模块对处理器的硬件要求仅仅为一个中断，几个 GPIO 管脚即可。

23.4.2 OV7670 输出时序

摄像头采集的图像，也就是 OV7670 芯片的输出，它采用 VGA 时序，通过 VSYNC、HREF/HSYNC 和 PCLK 引脚输出图像。因此我们必须正确地理解 VGA 工作时序图。

VGA 最早是指一种 480 × 640 像素显示器的显示模式，它显示时将一行一行地对图像的像素进行扫描，所以它的时序分为行时序和帧时序。

□ 行时序图

行时序即输出每一行像素的时序，见图 23-11。它以 PCLK 输出像素时钟，以 D0 ~ D9 输出像素数据，以 HREF 输出行起始信号和行结束信号。

从图 23-11 中可以看出，当 HREF 为高电平时，摄像头数据端口随着像素时钟 PCLK 的运转，先后输出一行的像素数据，当一行数据传输完成时，HREF 转为低电平。在 HREF 为高电平期间，每一个 PCLK 时钟就输出一个基本数据单元，而且数据在 PCLK 在上升沿阶段维持稳定，因此，主控芯片配置中断时，应该配置为上升沿中断，在上升沿读取数据。

□ 帧时序图

VGA 显示图像时以一幅图像为一帧，所以其帧时序（也称为场时序）由多个行时序组成，见图 23-12。帧时序与行时序的区别主要是多了一条信号线 VSYNC，用于表示帧起始信号和帧结束信号。图 23-12 中的 HSYNC 信号线与 HREF 作用相同，我们使用 HREF 即可。

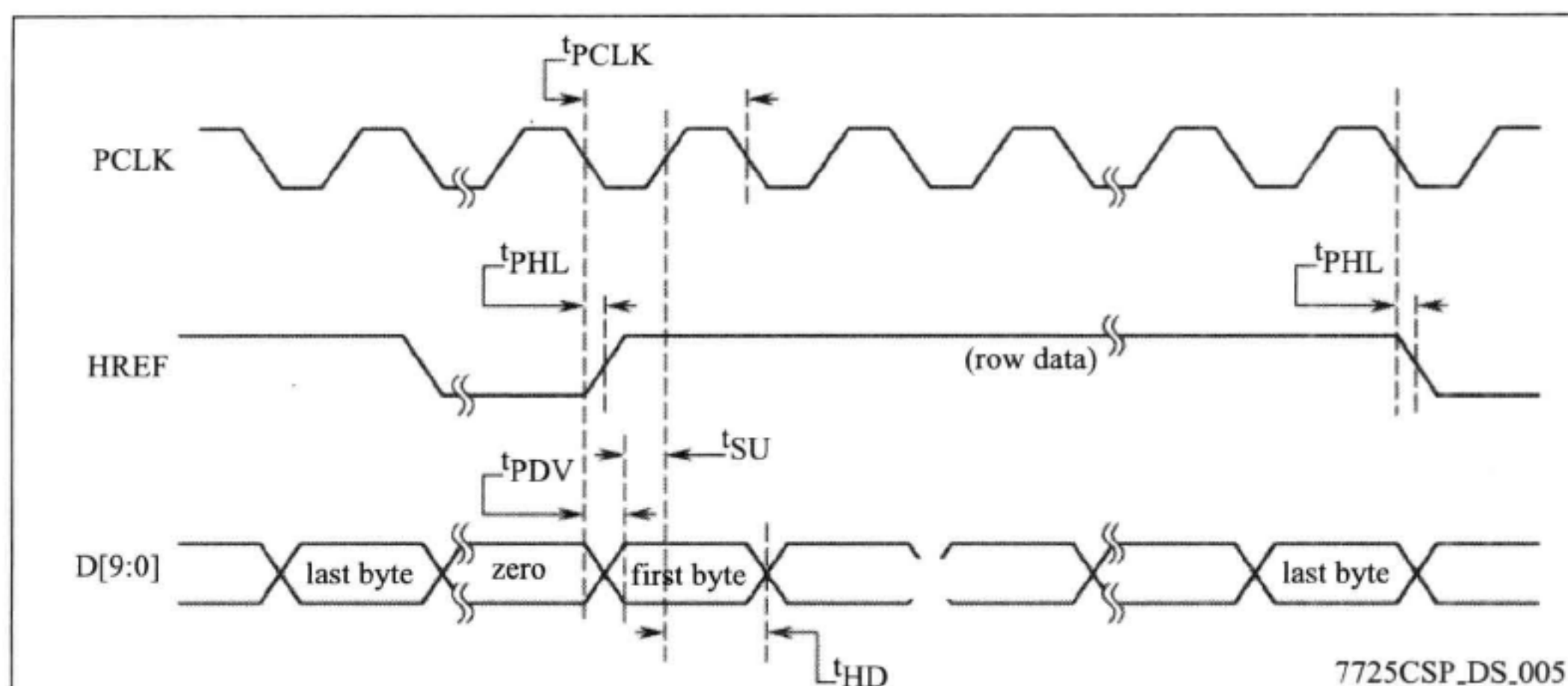


图 23-11 OV7670 输出的 VGA 行时序

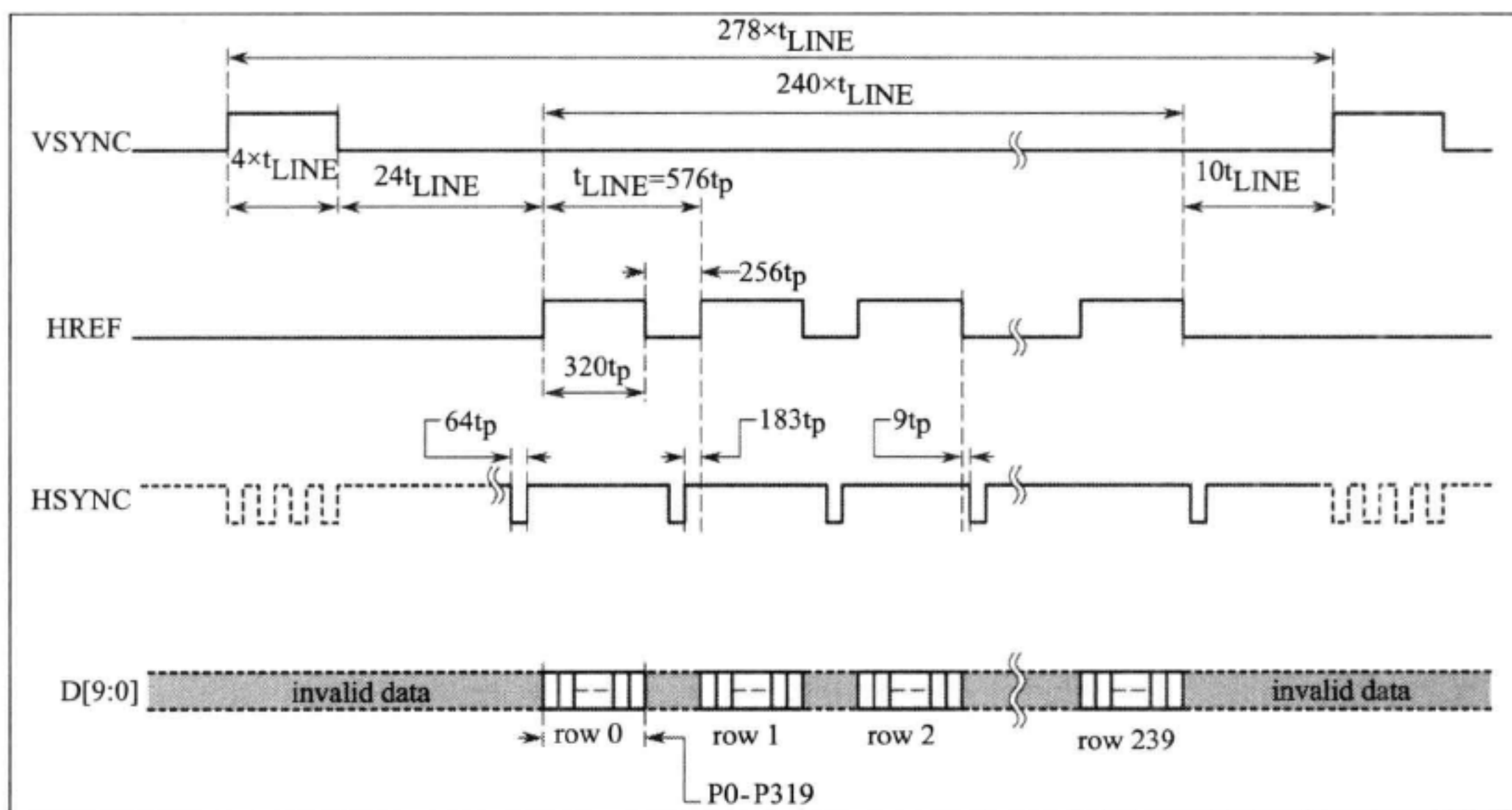


图 23-12 QVGA 帧时序（与 VGA 时序类似）

当 VSYNC 为低电平时，表明正在传送一幅图像，当 VSYNC 由低电平变为高电平时，表明一幅图像的数据已经传送完成。而在 VSYNC 为低电平期间，我们可以根据 HREF 信号来换行，当 HREF 由高电平变为低电平时，表明一行数据已经传送完毕；当 HREF 由低电平变为高电平时，表明新的一行数据开始传输。信号 HSYNC 同样可用于行同步，它与 HREF 有一点区别（HSYNC 和 HREF 共用一个管脚，可以通过软件来配置），由图 23-12 可知，HREF 为高电平时，摄像头输出的数据都是有效的；而 HSYNC 为高电平期间，在后面有一段时间是无意义的，即图中灰色部

分。结合 FIFO 的特性，我们选用了 HREF 作为行同步。

OV7670 实际输出图像时，不是标准的 VGA 时序，而可以选择使用 QVGA、CIF、QCIF 时序（可通过配置 OV7670 的 COM7 寄存器选择）。QVGA 意为 Quarter VGA，译为 VGA 的四分之一，即 240×320 像素，这正好符合配套 3.2 寸屏的像素大小，所以我们采用 QVGA 时序。见图 23-12 中的 QVGA 时序，可以看到输出每行像素的时间（HREF 高电平的时间）为 $320 t_p$ ，即 320 个像素点的时间， $240 \times t_{LINE}$ 就是指输出 240 行像素的基本时间。

□ 像素输出时序

像素输出时序即每个像素点数据的组织形式和输出方式。我们选择的是 RGB565 格式，所以，我们看看该模式下的像素输出时序，见图 23-13。

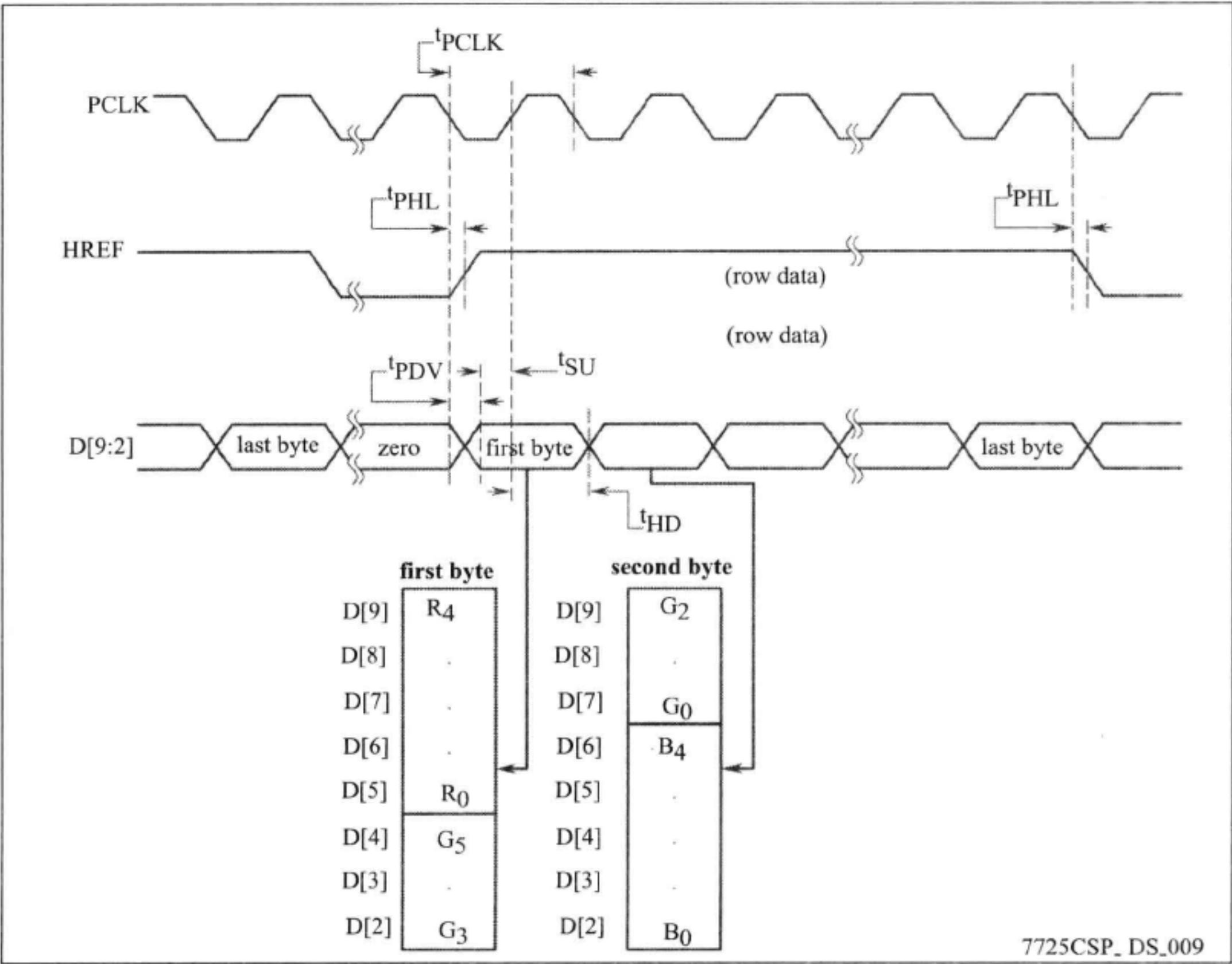


图 23-13 摄像头 RGB565 像素数据输出时序图

像素值从摄像头数据端口的 D2 到 D9 管脚输出，每一个像素由 RGB565 格式来表示，共 16 位，分两次来传输，之后用户组合起来即可。当用户收到 HREF 上升沿中断之后，开始捕捉到 PCLK 上升沿，该时刻读取到的数据即为第一字节有效像素数据。当收到 HREF 下降沿时，表明整行像素已经全部输出。

23.4.3 FIFO 时序

摄像头采集分辨率我们配置为 240×320 ，每个像素用 RGB565 格式表示，即每个像素占用两个字节，因此，整幅图像占用的空间大小为 $240 \times 320 \times 2$ ，等于 153600 字节。摄像头模块中采用的 FIFO 型号为 AL422B，其容量高达 384 KB，完全符合我们的空间要求。

AL422B 以下几个操作比较重要：读 FIFO、写 FIFO、读指针复位、写指针复位等。

□ 读 FIFO 时序

读 FIFO 时序见图 23-14。

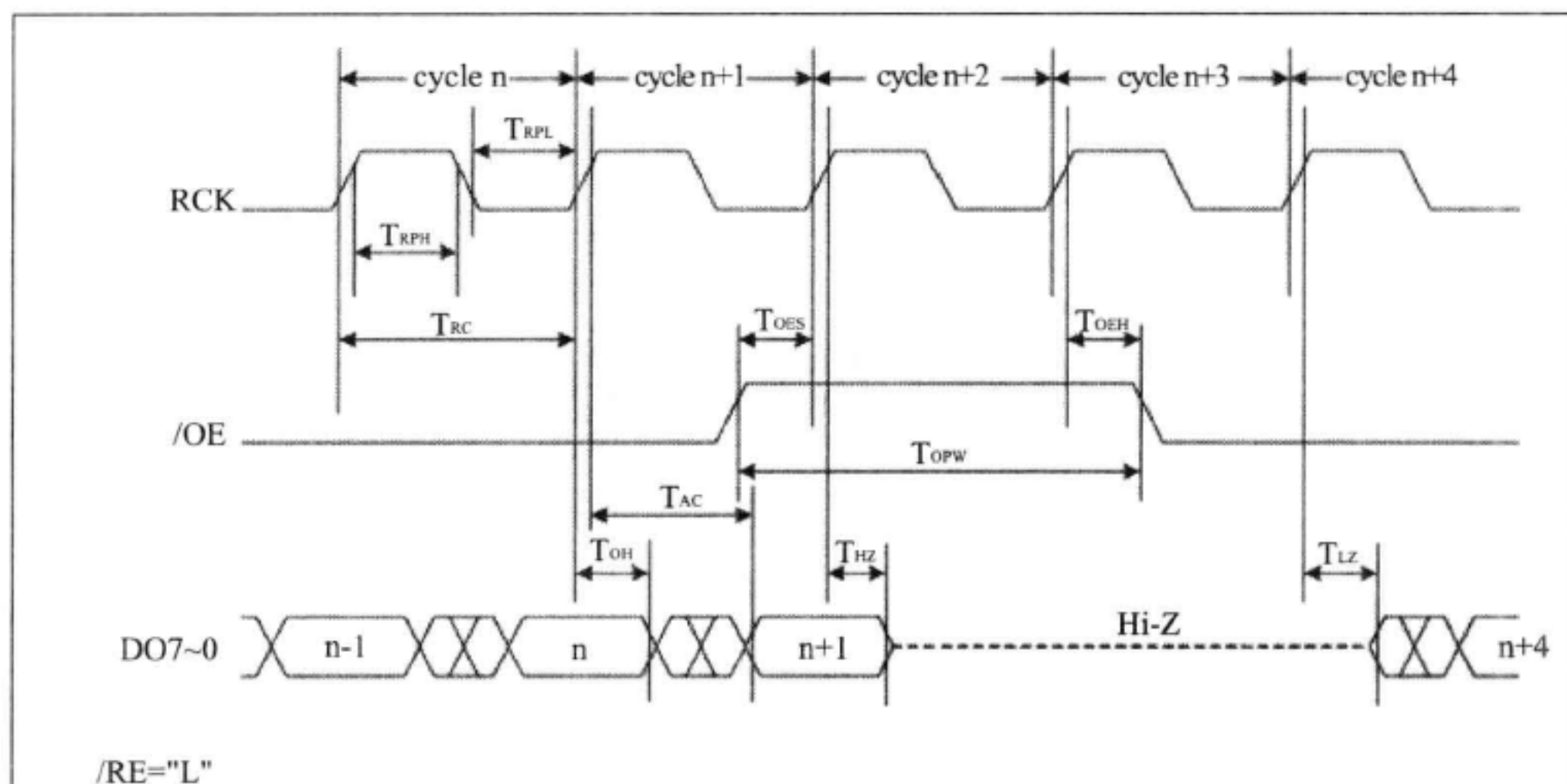


图 23-14 读 FIFO 时序

摄像头模块的 FIFO 硬件中已经将 RE 管脚设置为低电平，即硬件已经激活读操作。从时序图可知，当 OE 管脚为低电平时，输出处于使能状态，随着读时钟 RCK 的运转，数据输出管脚 DO7 ~ DO0 就会按地址递增的方式输出数据；当 OE 管脚为高电平时，关闭输出，随着读时钟 RCK 的运转，数据输出管脚 DO7 ~ DO0 会呈现高阻态，即在这个时候，单片机读取到的端口值是不稳定的、无效的，但是 FIFO 读指针的地址值仍会随着 RCK 的运转递增。

□ 写 FIFO 时序

写 FIFO 时序见图 23-15。

在写时序中，当 WE 管脚为低电平时，FIFO 写入处于使能状态，随着写时钟 WCK 的运转，数据管脚 DI7 ~ DI0 的电平值将会按地址递增的方式存入 FIFO；当 WE 管脚为高电平时，关闭输入，数据端口管脚 DI7 ~ DI0 的电平值不会随着写时钟 WCK 的运转被捕捉保存入 FIFO。

□ 读指针复位操作

当 WRST 为低电平时，读指针就复位到从 0 地址开始读取数据。

□ 写指针复位操作

当 WRST 为低电平时，写指针就复位到从 0 地址开始写入数据。

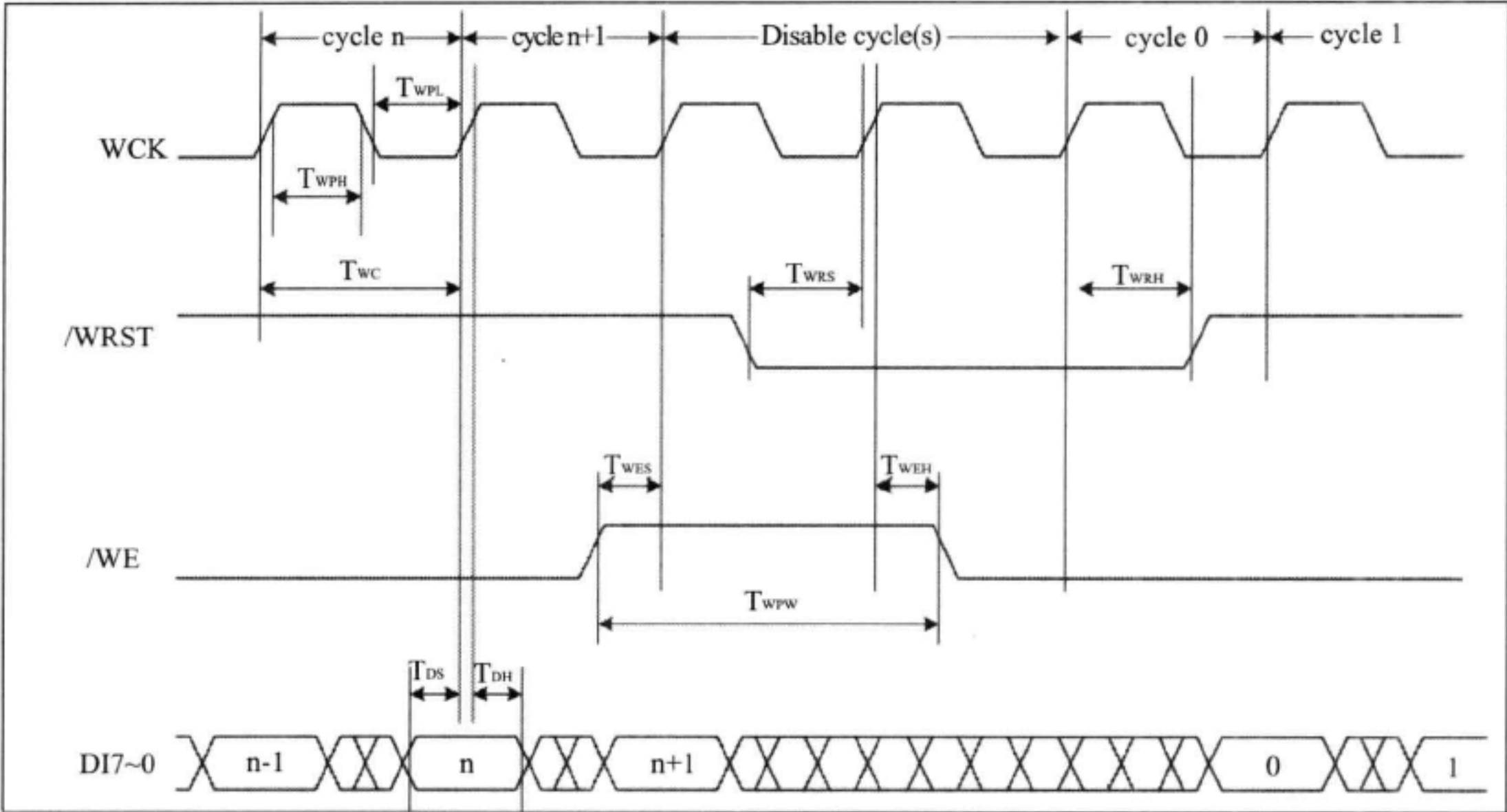


图 23-15 写 FIFO 时序

23.4.4 摄像头的驱动原理

摄像头模块的 OV7670 和 FIFO AL422B 的原理图见图 23-16 和图 23-17。

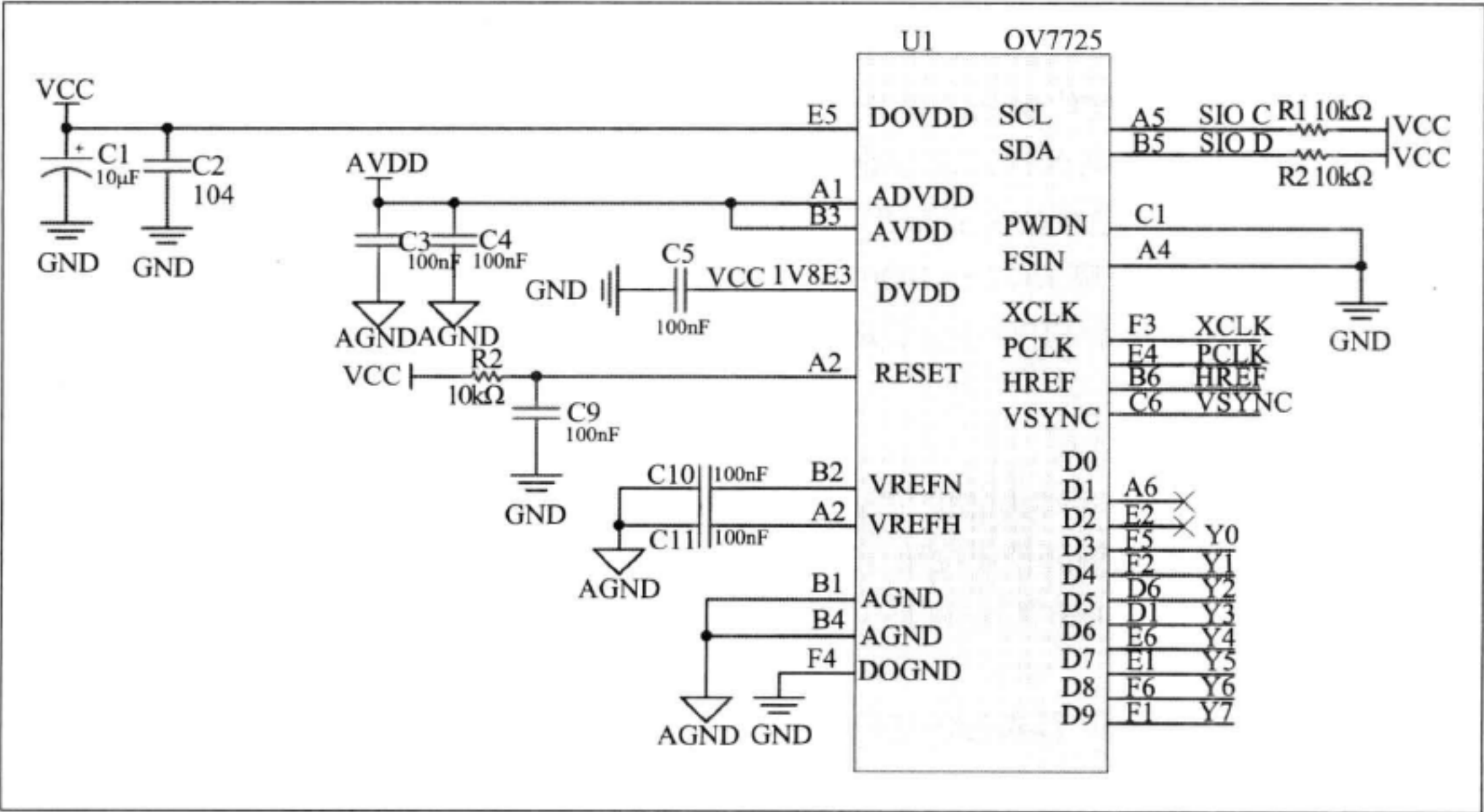


图 23-16 OV7670 硬件驱动电路图

- ☐ PD6: CAMERA-OE; PB8: CAMERA-D0
- ☐ PE0: CAMERA-RRST; PB9: CAMERA-D1
- ☐ PE2: CAMERA-RCLK; PB10: CAMERA-D2
- ☐ PA8: CAMERA-XCLK; PB11: CAMERA-D3
- ☐ PA0: CAMERA-VSYNC; PB12: CAMERA-D4
- ☐ PB5: CAMERA-WRST; PB13: CAMERA-D5
- ☐ PD3: CAMERA-WEN; PB14: CAMERA-D6
- ☐ PA1: CAMERA-HREF; PB15: CAMERA-D7
- ☐ PC6: CAMERA-SIO-C; PC6: CAMERA-SIO-D

3. 库文件

使用 3.5 版本固件库：

- ☐ Startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/misc.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_exti.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_fsmc.c
- ☐ FWlib/stm32f10x_usart.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/lcd.c
- ☐ USER/SysTick.c
- ☐ USER/lcd_botton.c
- ☐ USER/usart1.c
- ☐ SENSOR/OV7670.c
- ☐ SENSOR/SCCB.c
- ☐ SENSOR/SensorConfig.c
- ☐ SENSOR/Imag_App.c

配套开发板上的 CAMERA 接口兼容 OV7670、OV7725 等系列的摄像头模块，原理图见图 23-18。

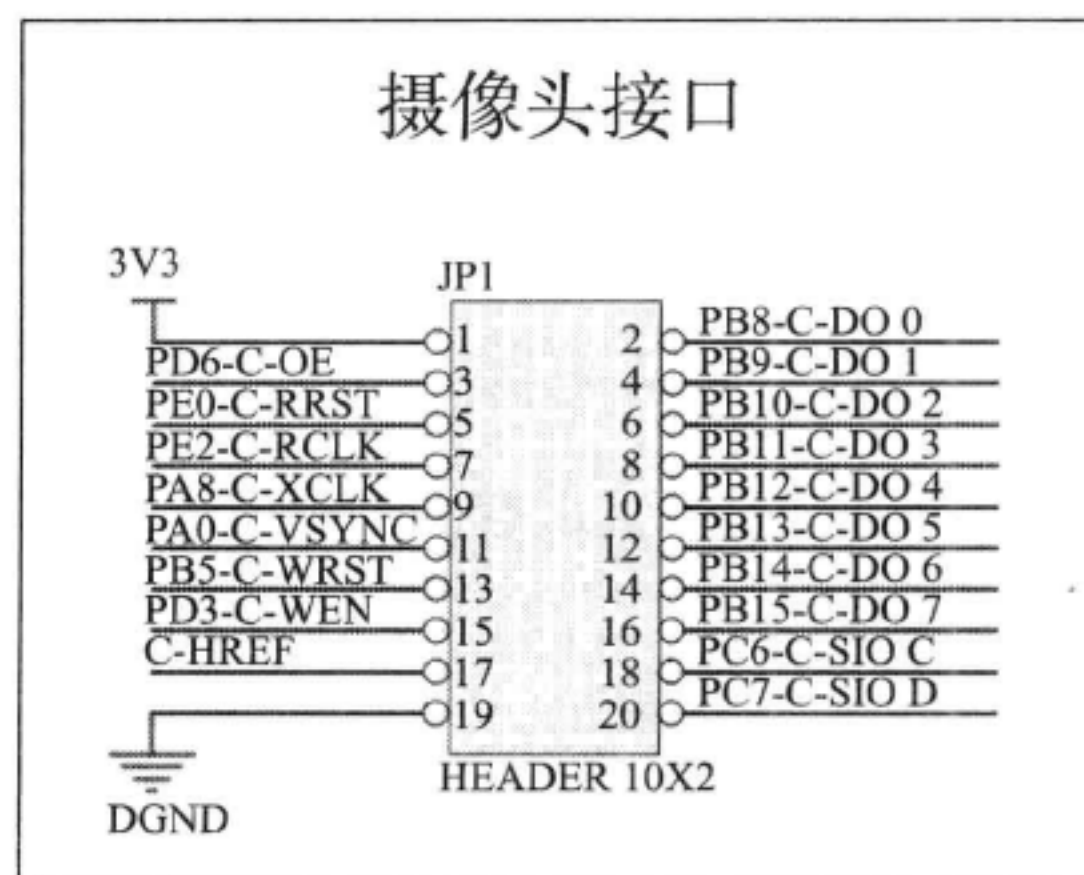


图 23-18 CAMERA 接口原理图

23.5.2 配置工程环境

本摄像头驱动实验中我们用到了 GPIO、RCC、EXTI、FSMC、USART 外设，所以我们要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_exti.c、stm32f10x_fsmc.c、stm32f10x_usart.c。由于在场中断 VSYNC 的检测中使用了中断，所以还要把 misc.c 文件添加进工程。

本工程使用了旧的用户文件 SysTick.c 用作定时，还使用了 lcd_botton.c 文件驱动 LCD。最后在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 23-1。

代码清单 23-1 摄像头例程的 stm32f10x_conf.h 文件配置

```

16. /**
17.  *****
18.  * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
19.  * @author    MCD Application Team
20.  * @version    V3.5.0
21.  * @date      08-April-2011
22.  * @brief     Library configuration file.
23.  *****/
24.
25. #include "stm32f10x_exti.h"
26. #include "stm32f10x_fsmc.h"
27. #include "stm32f10x_gpio.h"
28. #include "stm32f10x_rcc.h"
29. #include "stm32f10x_usart.h"
30. #include "misc.h"

```

23.5.3 main 文件

从本工程的 main 文件分析代码的执行流程，见代码清单 23-2。

代码清单 23-2 摄像头例程的 main 函数

```

1.
2. #include "stm32f10x.h"
3. #include "lcd_botton.h"
4. #include "OV7670.h"
5. #include "Imag_App.h"
6. #include "SysTick.h"
7. #include "usart1.h"
8. #include "SCCB.h"
9. extern volatile u8 Frame_Count;
10.
11.
12. /* 使用 3.2 寸 ILI9341 LCD */
13. int main(void)
14. {
15.
16.     SysTick_Init();
17.     USART1_Config();
18.     LCD_Init();

```

```

19.     SCCB_GPIO_Configuration();
20.     FIFO_GPIO_Configuration();
21.     while(OV7670_Init() != SUCCESS);
22.     VSYNC_Init();
23.     LCD_open_windows(0, 0, 320, 240);
24.     OV7670_VSYNC = 0;
25.     while(1)
26.     {
27.         if( 1 )                                // Ov7670_vsync == 2
28.         {
29.             Delay_us(5);
30.             FIFO_PREPARE;                        /*FIFO 准备*/
31.             Get_imag_and_discor();               /*采集并显示*/
32.             OV7670_VSYNC = 0;
33.             Frame_Count++;                       /*帧计数器加1*/
34.         }
35.     }
36. }
37.
38. /***** (C) COPYRIGHT 2012 WildFire Team *****/
39.

```

main 函数的执行流程说明如下：

1) main 函数首先调用了 SysTick_Init()、USART1_Config()、LCD_Init() 函数初始化了 SysTick、串口及 LCD，这些函数在前面的章节已经介绍。

2) 接下来 SCCB_GPIO_Configuration() 和 FIFO_GPIO_Configuration() 函数用于初始化 GPIO，这些 GPIO 分别用来模拟 SCCB 时序和读写 FIFO。

3) 初始化 SCCB 相关的引脚，就可向 OV7670 芯片写入配置参数。本例程中是通过调用 OV7670_Init() 函数实现的，该函数利用 SCCB 通信协议，根据 OV7670 的寄存器说明，向寄存器写入相关的配置参数。

4) 由于需要 VSYNC 提供中断，所以作为 VSYNC 引脚的 GPIO PA0 要设置 EXTI 中断，并配置中断优先级，这些工作由 VSYNC_Init() 函数完成。

5) 调用 LCD_open_windows() 使 LCD 开窗，做好显示的准备。

6) 根据 OV7670_VSYNC 标志，判断 FIFO 是否接收完一幅图像。在中断服务程序中，若检测到两次 VSYNC 的上升沿（表示接收完一幅图像），会把 OV7670_VSYNC 变量设置为 2。

7) 判断接收完一幅图像后，调用宏 FIFO_PREPARE 使读 FIFO 指针复位，使 Get_imag_and_discor() 读取 FIFO 时，读取正确的数据并显示到 LCD 屏上。

8) 记录帧数目的变量 Frame_Count 加 1，这个变量用来统计帧率，由 SysTick 的中断服务函数通过串口输出。最后把 OV7670_VSYNC 置 0，使重新开始计数。

23.5.4 SCCB 总线的软件实现

由于 STM32 没有 SCCB 的接口，我们需要用普通的 GPIO 引脚来模拟 SCCB 时序。首先需要 SCCB_GPIO_Configuration() 函数初始化 SCCB 总线所用到的 GPIO 引脚，再根据 SCCB 时序编写读、写字节的函数，OV7670_Init() 利用这些函数，向 OV7670 相应的寄存器写入配置参数，

初始化摄像头。本节介绍的与 SCCB 时序相关函数都位于 SCCB.c 文件中。

(1) 初始化 SCCB 用到的 GPIO

在本实验中,摄像头的 SCL 和 SDA 连接到 STM32 的 PC6 和 PC7,使用 SCCB_GPIO_Configuration() 函数初始化这两个引脚,其代码见代码清单 23-3。

代码清单 23-3 SCCB_GPIO_Configuration() 函数

```

1.  /*****
2.  * 函数名: SCCB_Configuration
3.  * 描述  : SCCB 管脚配置
4.  * 输入  : 无
5.  * 输出  : 无
6.  * 注意  : 无
7.  *****/
8.  void SCCB_GPIO_Configuration(void)
9.  {
10.     GPIO_InitTypeDef  GPIO_InitStructure;
11.     /* SCL(PC6)、SDA(PC7) 管脚配置 */
12.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
13.     GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_6 | GPIO_Pin_7;
14.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
16.     GPIO_Init(GPIOC, &GPIO_InitStructure);
17. }
```

(2) SCCB 起始与结束时序

SCCB 通信需要有起始与结束信号,这分别由 SCCB_Start() 和 SCCB_Stop() 函数实现。SCCB_Start() 代码见代码清单 23-4。

代码清单 23-4 SCCB_Start() 函数

```

1.  /*****
2.  * 函数名: SCCB_Start
3.  * 描述  : SCCB 起始信号
4.  * 输入  : 无
5.  * 输出  : 无
6.  * 注意  : 内部调用
7.  *****/
8.  static int SCCB_Start(void)
9.  {
10.     SDA_H;
11.     SCL_H;
12.     SCCB_delay();
13.     if(!SDA_read)
14.         return DISABLE; /* SDA 线为低电平则总线忙,退出 */
15.     SDA_L;
16.     SCCB_delay();
17.     if(SDA_read)
18.         return DISABLE; /* SDA 线为高电平则总线出错,退出 */
19.     SDA_L;
20.     SCCB_delay();
21.     return ENABLE;
22. }
```

因此，在数据传输的第 9 位，主机使用 SCCB_WaitAck() 函数来等待从机的应答，见代码清单 23-6。

代码清单 23-6 SCCB_WaitAck() 函数

```

1.  /*****
2.   * 函数名：SCCB_WaitAck
3.   * 描述   ：SCCB 等待应答
4.   * 输入   ：无
5.   * 输出   ：返回为：=1 有 ACK，=0 无 ACK
6.   * 注意   ：内部调用
7.   *****/
8.  static int SCCB_WaitAck(void)
9.  {
10.     SCL_L;
11.     SCCB_delay();
12.     SDA_H;
13.     SCCB_delay();
14.     SCL_H;
15.     SCCB_delay();
16.     if(SDA_read)
17.     {
18.         SCL_L;
19.         return DISABLE;
20.     }
21.     SCL_L;          //SDA 低电平应答
22.     return ENABLE;
23. }
```

该函数让主机把 SDA 线设置为高电平，延时一段时间后再检测 SDA 线的电平，若为低则返回 ENABLE 表示接收到从机的应答，反之返回 DISABLE。

最后，整个三相写过程由函数 SCCB_WriteByte() 实现，它的具体代码见代码清单 23-7。

代码清单 23-7 SCCB_WriteByte() 函数

```

1.  /*****
2.   * 函数名：SCCB_WriteByte
3.   * 描述   ：写一字节数据
4.   * 输入   ：- WriteAddress: 待写入地址    - SendByte: 待写入数据    - DeviceAddress: 器件类型
5.   * 输出   ：返回为：=1 成功写入，=0 失败
6.   * 注意   ：无
7.   *****/
8.  int SCCB_WriteByte( u16 WriteAddress , u8 SendByte )
9.  {
10.     if(!SCCB_Start())
11.     {
12.         return DISABLE;
13.     }
14.     SCCB_SendByte( DEV_ADR );          /* 器件地址，第一阶段 */
15.     if( !SCCB_WaitAck() )
16.     {
17.         SCCB_Stop();
18.         return DISABLE;
19.     }
```

```

20.   SCCB_SendByte((u8)(WriteAddress & 0x00FF));
21.                                   /* 设置低起始地址, 第二阶段 */
22.   SCCB_WaitAck();
23.   SCCB_SendByte(SendByte);        /* 写内容, 第三阶段 */
24.   SCCB_WaitAck();
25.   SCCB_Stop();
26.   return ENABLE;
27.}

```

SCCB_WriteByte() 函数调用 SCCB_Start() 产生起始信号, 接着调用 SCCB_SendByte() 把器件地址 DEV_ADR (这是一个宏, 数值是 0x42) 一位一位地发送出去, 在第 9 位时, 调用 SCCB_WaitAck() 函数检测从机的应答, 若接收到应答则进入第二阶段——发送目的寄存器地址, 再进入第三阶段——发送要写入的内容。在第二、三阶段没有加条件语句判断是否接收到从机的应答, 因为 SCCB 规定在数据传输阶段允许从机不应答 (实际上, OV7670 芯片在这两个阶段都会有应答信号)。最后, 在三阶段都传输结束时, 要调用 SCCB_Stop() 函数结束本次 SCCB 传输。

与自由位相对应的非应答信号用在两相读操作的第二阶段第 9 位, 见图 23-20。在这第 9 位中, 从机把 SDA 线置为高电平, 而主机把 SDA 线拉低表示非应答, 接着本次读数据的操作就结束了 (SCCB 不支持连续读取数据)。

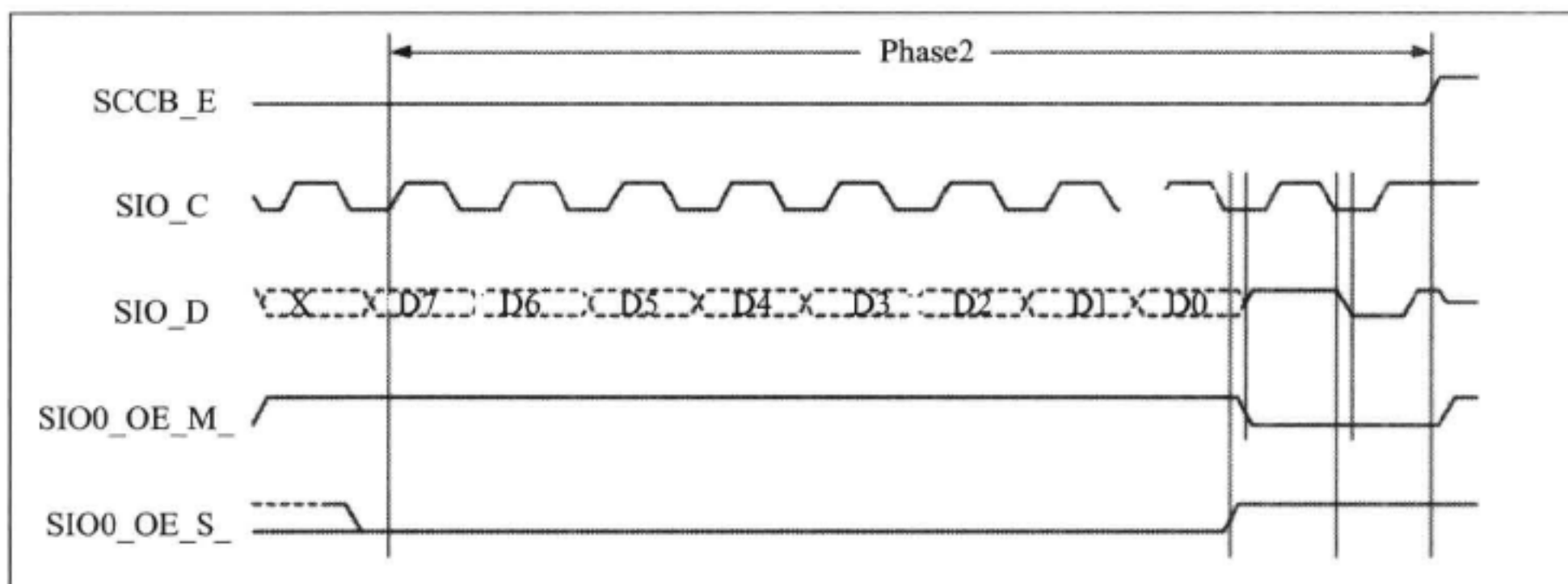


图 23-20 两相读操作第二阶段 (读寄存器内容)

主机的非应答信号由 SCCB_NoAck() 函数实现, 其代码见代码清单 23-8。

代码清单 23-8 SCCB_NoAck() 函数

```

1.  /*****
2.   * 函数名: SCCB_NoAck
3.   * 描述  : SCCB 无应答方式
4.   * 输入  : 无
5.   * 输出  : 无
6.   * 注意  : 内部调用
7.   *****/
8. static void SCCB_NoAck(void)
9. {
10.    SCL_L;
11.    SCCB_delay();

```



```

12.    SDA_H;      // 高电平为非应答信号
13.    SCCB_delay();
14.    SCL_H;
15.    SCCB_delay();
16.    SCL_L;
17.    SCCB_delay();
18.}

```

最后，整个读寄存器的过程由 SCCB_ReadByte() 函数完成，其代码见代码清单 23-9。

代码清单 23-9 SCCB_ReadByte() 函数

```

1.  /*****
2.  * 函数名: SCCB_ReadByte
3.  * 描述  : 读取一串数据
4.  * 输入  : - pBuffer: 存放读出数据    - ReadAddress: 待读出地址    - DeviceAddress: 器件
      类型
5.  * 输出  : 返回为 :=1 成功读入, =0 失败
6.  * 注意  : 无
7.  *****/
8.  int SCCB_ReadByte(u8* pBuffer, u8 ReadAddress)
9.  {
10.
11.    if(!SCCB_Start())
12.    {
13.        return DISABLE;
14.    }                                     /* 在读操作前要有写操作以提供起始地址 */
15.
16.    SCCB_SendByte( DEV_ADR );              /* 器件地址, 写操作第一阶段 */
17.    if( !SCCB_WaitAck() )
18.    {
19.        SCCB_Stop();
20.        return DISABLE;
21.    }
22.    SCCB_SendByte( ReadAddress );          /* 设置低起始地址, 写操作第二阶段 */
23.
24.    SCCB_WaitAck();
25.    SCCB_Stop();
26.
27.    if(!SCCB_Start())
28.    {
29.        return DISABLE;
30.    }
31.    SCCB_SendByte( DEV_ADR + 1 );          /* 器件地址, 读操作第一阶段 */
32.    if(!SCCB_WaitAck())
33.    {
34.        SCCB_Stop();
35.        return DISABLE;
36.    }
37.
38.    *pBuffer = SCCB_ReceiveByte();         /* 读取内容, 读操作第二阶段 */
39.    SCCB_NoAck();
40.
41.    SCCB_Stop();
42.    return ENABLE;
43.}

```

本函数在两相读操作前，加入了一个两相写操作，用于向从机发送要读取的寄存器地址。在两相读操作的第一个阶段（第 31 行），先使用 SCCB_SendByte() 函数发送的数据是 DEV_ADR+1（即 0x43），这与写操作中发送的 DEV_ADR 有区别。这是因为在第一阶段发送的这个器件地址的最低位用于表示数据传送方向，最低位为 0 表示主机写数据，最低位为 1 表示主机读数据，所以 DEV_ADR+1 就表示读数据了。读操作的第二阶段是使用 SCCB_ReceiveByte() 函数一位一位地接收数据，然后存放到 pBuffer 指向的单元中。接收完 8 位数据后，主机调用 SCCB_NoAck() 发送非应答位，最后调用 SCCB_Stop() 结束本次读操作。

23.5.5 初始化 OV7670

在上一节编写了模拟 SCCB 的读写寄存器的时序后，就可以向 OV7670 的寄存器发送配置参数并对其进行初始化了。该过程由 OV7670.c 文件中的 Sensor_Init() 函数完成，代码见代码清单 23-10。

代码清单 23-10 Sensor_Init() 函数

```

1.
2. /*****
3.  * 函数名：Sensor_Init
4.  * 描述   ：Sensor 初始化
5.  * 输入   ：无
6.  * 输出   ：返回 1 成功，返回 0 失败
7.  * 注意   ：无
8.  *****/
9. ErrorStatus OV7670_Init(void)
10. {
11.     u16 i = 0;
12.     u8 Sensor_IDCode = 0;
13.
14.     if( 0 == SCCB_WriteByte ( 0x12, 0x80 ) ) /* 复位 sensor */
15.     {
16.
17.         Delay_ms(50);
18.         return ERROR ;
19.     }
20.     Delay_ms(50);
21.     if( 0 == SCCB_ReadByte( &Sensor_IDCode, 0x0b ) )
22.         /* 读取 sensor ID 号 */
23.     {
24.         return ERROR;
25.     }
26.
27.
28.     if(Sensor_IDCode == OV7670_ID)
29.     {
30.         for( i = 0 ; i < OV7670_REG_NUM ; i++ )
31.         {
32.             if( 0 == SCCB_WriteByte(Sensor_Config[i].Address, Sensor_Config[i].Value) )
33.             {
34.                 return ERROR;

```

```
35.         }
36.     }
37. }
38. else
39. {
40.     return ERROR;
41. }
42.     return SUCCESS;
43. }
44.
```

这个函数的执行流程如下：

1) 调用 SCCB_WriteByte() 向地址为 0x12 的寄存器写入数据 0x80，进行复位操作。见表 23-5，根据 OV7670 的数据手册，把该寄存器的位 7 置 1，可控制 SCCB 的寄存器复位。

表 23-5 OV7670 部分寄存器

地址	寄 存 器 名	默认值	读/写	描 述															
11	CLKRC	80	读写	内部时钟 位 [7]：保留；位 [6]：直接使用外部时钟（没有预分频）；位 [5：0]：内部时钟分频 F（内部时钟）=F（输入时钟）/（位 [5：0]+1） 范围：[0000] ~ [1111]															
12	COM7	00	读写	通用控制 7 位 [7]：SCCB 寄存器复位，0 不复位，1 复位 位 [6]：保留 位 [5]：输出格式 -CIF 位 [4]：输出格式 -QVGA 位 [3]：输出格式 -QCIF 位 [2]：输出格式 -RGB（见下面 COM7[2] 和 COM7[0] 配置） 位 [1]：彩色条，0 非使能，1 使能 位 [5]：输出格式 -Raw RGB（见下面 COM7[2] 和 COM7[0] 配置） <table><tr><td></td><td>COM7[2]</td><td>COM7[0]</td></tr><tr><td>YUV</td><td>0</td><td>0</td></tr><tr><td>RGB</td><td>0</td><td>1</td></tr><tr><td>Bayer RAW</td><td>1</td><td>0</td></tr><tr><td>Processed Bayer RAW</td><td>1</td><td>1</td></tr></table>		COM7[2]	COM7[0]	YUV	0	0	RGB	0	1	Bayer RAW	1	0	Processed Bayer RAW	1	1
	COM7[2]	COM7[0]																	
YUV	0	0																	
RGB	0	1																	
Bayer RAW	1	0																	
Processed Bayer RAW	1	1																	
13	COM8	8F	读写	通用控制 8 位 [7]：使能快速 AGC/AEC 算法 位 [6]：AEC- 步长限制，0 表示步长限制与垂直同步，1 表示不限制步长 位 [5]：条纹滤波器打开 / 关闭 - 打开条纹滤波器，BD50ST（0x9D）或者 BD60ST（0x9E）要设成 1。0：关；1：开 位 [4:3]：保留 位 [2]：AGC 使能 位 [1]：AWB 使能 位 [0]：AEC 使能															

2) 调用 `SCCB_ReadByte()` 函数从地址为 `0x0b` 的寄存器读取出 `OV7670` 芯片的 ID 号, 并与默认值进行对比, 这个操作可以用来确保 `OV7670` 是否正常工作。

3) 利用 `for` 语句循环调用 `SCCB_WriteByte()` 函数, 向各个寄存器写入配置参数, 其中 `SCCB_WriteByte()` 的输入参数为 `Sensor_Config[i].Address` 和 `Sensor_Config[i].Value`, 这是自定义的结构体数组, 分别对应于要配置的寄存器地址和寄存器配置参数。`Sensor_Config` 数组是在 `SensorConfig.c` 文件定义的, 首先定义了 `Register_Info` 结构体, 它包含地址和寄存器两个结构体成员, 见代码清单 23-11。

代码清单 23-11 Register_Info 结构体

```
1. typedef struct Register
2. {
3.     u8 Address;          /* 寄存器地址 */
4.     u8 Value;            /* 寄存器值 */
5. }Register_Info;
```

再利用这个自定义的结构体定义结构体数组, 每组的内容就表示了寄存器地址及相应的配置参数。若要修改对 `OV7670` 的配置, 可参考 `OV7670` 数据手册的说明, 修改相应地址的内容即可, `SCCB_WriteByte()` 函数会把这些参数写入 `OV7670` 芯片, 下面是结构体数组的部分代码, 省略了部分寄存器, 完整部分请参考源代码, 见代码清单 23-12。

代码清单 23-12 部分寄存器控制参数

```
1.
2. Register_Info Sensor_Config[] =
3. {
4.
5.     /* 以下为 OV7670 QVGA RGB565 参数 */
6.     {0x3a, 0x04}, //
7.     {0x40, 0x10},
8.     {0x12, 0x14},
9.     {0x32, 0x80},
10.    {0x17, 0x16},
11.
12.    {0x18, 0x04}, //5
13.    {0x19, 0x02},
14.    /****** 此处省略 N 行 *****/
15.    {0x6f, 0x9f}, //0x9e for advance AWB
16.    {0x55, 0x00}, // 亮度
17.    {0x56, 0x45}, // 对比度
18.    {0x57, 0x80}, //0x40, change according to Jim's request
19.
20.};
```

23.5.6 采集并显示图像

初始化 `OV7670` 后, `OV7670` 芯片开始正常工作, 由于我们使用的摄像头模块自带 FIFO, `OV7670` 采集的图像将保存到 FIFO, 我们只需要使用 STM32 检测摄像头模块的 `VSYNC` 输出的帧

结束信号，然后从 FIFO 中读取图像数据即可。

(1) 初始化 VSYNC 引脚

由于我们使用中断的方式来检测 VSYNC 的信号，所以先要把相应的引脚初始化并为它配置 EXTI 中断。

在 main 函数中，调用的 VSYNC_Init() 函数通过调用其他三个子函数来完成所有初始化流程，见代码清单 23-13。

代码清单 23-13 VSYNC_Init() 函数

```

1.  /*****
2.   * 函数名：VSYNC_Init
3.   * 描述   ：OV7670 VSYNC 中断相关配置
4.   * 输入   ：无
5.   * 输出   ：无
6.   * 注意   ：无
7.   *****/
8.  void VSYNC_Init(void)
9.  {
10.     VSYNC_GPIO_Configuration();
11.     VSYNC_EXTI_Configuration();
12.     VSYNC_NVIC_Configuration();
13. }
```

子函数 VSYNC_GPIO_Configuration() 完成了 GPIO 引脚模式、开启时钟的基本 I/O 配置工作，本实验中使用的是 PA0 引脚检测 VSYNC 信号。VSYNC_GPIO_Configuration() 代码见代码清单 23-14。

代码清单 23-14 VSYNC_GPIO_Configuration() 函数

```

1.  /*****
2.   * 函数名：VSYNC_GPIO_Configuration
3.   * 描述   ：OV7670 GPIO 配置
4.   * 输入   ：无
5.   * 输出   ：无
6.   * 注意   ：无
7.   *****/
8.  static void VSYNC_GPIO_Configuration(void)
9.  {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); /*PA0-VSYNC*/
12.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
13.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
14.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15.     GPIO_Init(GPIOA, &GPIO_InitStructure);
16. }
```

子函数 VSYNC_EXTI_Configuration() 为 PA0 引脚配置成上升沿触发的 EXTI 中断，根据前面关于 OV7670 输出时序的介绍，在 VSYNC 的两个上升沿期间，OV7670 输出了一幅完整的图像。VSYNC_EXTI_Configuration() 代码见代码清单 23-15。

代码清单 23-15 VSYNC_EXTI_Configuration() 函数

```

1.  /*****
2.   * 函数名: VSYNC_EXTI_Configuration
3.   * 描述  : OV7670 VSYNC 中断管脚配置
4.   * 输入  : 无
5.   * 输出  : 无
6.   * 注意  : 无
7.   *****/
8.  static void VSYNC_EXTI_Configuration(void)
9.  {
10.     EXTI_InitTypeDef EXTI_InitStructure;
11.
12.     GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);
13.     EXTI_InitStructure.EXTI_Line = EXTI_Line0;
14.     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
15.     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising ;/* 上升沿触发 */
16.     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
17.     EXTI_Init(&EXTI_InitStructure);
18.     EXTI_GenerateSWInterrupt(EXTI_Line0);    /* 中断挂到 EXTI_Line0 线 */
19. }

```

设置好了中断, 还需要为它分配优先级, 这由子函数 VSYNC_NVIC_Configuration() 完成, 见代码清单 23-16。

代码清单 23-16 VSYNC_NVIC_Configuration() 函数

```

1.  /*****
2.   * 函数名: VSYNC_NVIC_Configuration
3.   * 描述  : VSYNC 中断配置
4.   * 输入  : 无
5.   * 输出  : 无
6.   * 注意  : 无
7.   *****/
8.  static void VSYNC_NVIC_Configuration(void)
9.  {
10.     NVIC_InitTypeDef NVIC_InitStructure;
11.     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
12.     NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
13.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
14.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
15.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
16.     NVIC_Init(&NVIC_InitStructure);
17. }

```

(2) 编写检测 VSYNC 的中断服务函数

由于 VSYNC 出现两次上升沿才表示 FIFO 保存了一幅图像, 所以在检测 VSYNC 上升沿的中断服务函数中, 我们使用一个变量 OV7670_VSYNC 作为标志。OV7670_VSYNC 标志的初始值为 0, 当检测到第一次上升沿时, 我们控制 FIFO 的相应 GPIO 引脚, 允许 OV7670 向 FIFO 写入图像数据, 并把标志值设置为 1; 检测到第二次上升沿时, 禁止 OV7670 写 FIFO, 把标志设置为 2。

在 main 函数的循环判断中, 当 OV7670_VSYNC=2 时, STM32 开始从 FIFO 读取数据并显示, 读取完毕后把 OV7670_VSYNC 标志复位为 0, 重新开始下一幅图像的采集。

中断服务函数 EXTI0_IRQHandler() 位于 stm32f10x_it.c 文件, 见代码清单 23-17。

代码清单 23-17 EXTI0_IRQHandler() 中断服务函数

```

1.
2. void EXTI0_IRQHandler(void)
3. {
4.     if ( EXTI_GetITStatus(EXTI_Line0) != RESET )
5.         // 检查 EXTI_Line0 线路上的中断请求是否发送到了 NVIC
6.     {
7.         if( OV7670_VSYNC == 0 )
8.         {
9.             FIFO_WRST_L();    // 拉低使 FIFO 写 (数据 from 摄像头) 指针复位
10.            FIFO_WE_H();      // 拉高使 FIFO 写允许
11.            OV7670_VSYNC = 1;
12.            FIFO_WE_H();      // 使 FIFO 写允许
13.            FIFO_WRST_H();    // 允许使 FIFO 写 (数据 from 摄像头) 指针运动
14.        }
15.        else if( OV7670_VSYNC == 1 )
16.        {
17.            OV7670_VSYNC = 2;
18.            FIFO_WE_L();      // 拉低使 FIFO 写暂停
19.        }
20.        EXTI_ClearITPendingBit(EXTI_Line0);
21.        // 清除 EXTI_Line0 线路挂起标志位
22.    }
23.
24.}

```

(3) 读 FIFO 并显示图像

采集的图像数据都保存到摄像头模块的 FIFO 中, 在 OV7670_VSYNC 标志变为 2 时, STM32 即可读取它并显示到 LCD 上。与 FIFO 相关的函数有 FIFO_GPIO_Configuration()、FIFO_PREPARE 和 Get_imag_and_discor()。

□ FIFO_GPIO_Configuration() 类似 SCCB_GPIO_Configuration() 函数, 完成了基本的 GPIO 初始化, 见代码清单 23-18。

代码清单 23-18 FIFO_GPIO_Configuration() 函数

```

1. /*****
2.  * 函数名: FIFO_GPIO_Configuration
3.  * 描述  : FIFO GPIO 配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 注意   : 无
7.  *****/
8. void FIFO_GPIO_Configuration(void)
9. {
10.    GPIO_InitTypeDef GPIO_InitStructure;

```

```

11.     RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC | RCC_
        APB2Periph_GPIOD | RCC_APB2Periph_GPIOE, ENABLE);
12.
13.     /*PB5(FIFO_WRST--FIFO 写复位)*/
14.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
15.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
16.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
17.     GPIO_Init(GPIOB, &GPIO_InitStructure);
18.
19.     /*PD3(FIFO_WEN--FIFO 写使能) PD6(FIFO_REN--FIFO 读使能)*/
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_6;
21.     GPIO_Init(GPIOD, &GPIO_InitStructure);
22.
23.     /*PE0(FIFO_RRST--FIFO 读复位) PE2(FIFO_RCLK--FIFO 读时钟)*/
24.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_2;
25.     GPIO_Init(GPIOE, &GPIO_InitStructure);
26.
27.     /*PB8-PB15(FIFO_DATA--FIFO 输出数据)*/
28.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_
        Pin_11 | GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
29.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
30.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
31.     GPIO_Init(GPIOB, &GPIO_InitStructure);
32.
33.     FIFO_CS_L();                                /* 拉低使 FIFO 输出使能 */
34.     FIFO_WE_H();                                /* 拉高使 FIFO 写允许 */
35.
36. }

```

□ FIFO_PREPAGE 实际是一个宏，它是在 main 函数中判断到接收完一幅图像后被调用的，它的作用是把 FIFO 读指针复位，使后面的数据从 FIFO 的 0 地址开始读取，见代码清单 23-19（在工程中，要把宏定义写在同一行或使用续行符）。

代码清单 23-19 FIFO_PREPAGE 宏

```

1.
2. #define FIFO_PREPAGE \
3. do{\
4. FIFO_RRST_L();\
5. FIFO_RCLK_L();\
6. FIFO_RCLK_H();\
7. FIFO_RRST_H();\
8. FIFO_RCLK_L();\
9. FIFO_RCLK_H();\
10.}while(0)
11.

```

□ Get_imag_and_discor() 函数完成了从 FIFO 读取图像及显示到 LCD 的工作，每当 OV7670 输出完一幅图像，它就被调用一次。在调用前，要用上面的 FIFO_PREPAGE 宏复位 FIFO 读指针。见代码清单 23-20。

代码清单 23-20 Get_imag_and_discor() 函数

```

1.
2. void Get_imag_and_discor(void)
3. {
4.     ul6 i, j;
5.     ul6 Camera_Data;
6.     for(i = 0; i < 240; i++)
7.     {
8.         for(j = 0; j < 319; j++)
9.         {
10.
11.             READ_FIFO_PIXEL(Camera_Data);
12.             // 从 FIFO 读出一个 RGB565 像素到 Camera_Data 变量
13.             LCD_WR_Data(Camera_Data);
14.         }
15.         READ_FIFO_PIXEL(Camera_Data);
16.         LCD_WR_Data(0);
17.     }
18. }
19.

```

在代码中, 循环调用了宏 READ_FIFO_PIXEL 读取 FIFO 数据, 并使用 LCD_WR_Data() 函数把该图像数据显示到 LCD 上。见代码清单 23-21。

代码清单 23-21 READ_FIFO_PIXEL 宏

```

1.
2. #define READ_FIFO_PIXEL(RGB565) \
3. do{RGB565=0; \
4. FIFO_RCLK_L(); \
5. RGB565 = (GPIOB->IDR) & 0xff00; \
6. FIFO_RCLK_H(); \
7. FIFO_RCLK_L(); \
8. RGB565 |= (GPIOB->IDR >>8) & 0x00ff; \
9. FIFO_RCLK_H(); \
10. }while(0)
11.

```

这个宏把 FIFO 读取的数据按 RGB565 格式处理, 保存到一个 16 位的变量中, LCD_WR_Data() 函数就可以直接利用这个数据, 显示到 LCD 上了。

回顾一下摄像头的驱动流程, 首先要设计 SCCB 时序, 根据 OV7670 数据手册进行配置, 然后利用 VSYNC 中断, 利用标志变量做状态标记, 最后在 main 函数中判断 OV7670 输出完一幅图像后, 调用 FIFO 相关函数进行读取并显示。

23.5.7 实验现象

将 OV7670 摄像头模块接入配套 STM32 开发板的 CAMERA 接口, 然后给开发板供电 (DC5V), 插上 J-LINK, 将编译好的程序下载到开发板。即可在 LCD 上输出摄像头拍到的图像, 若图片显示不够清晰, 可调整镜头进行调焦, 使得到清晰的图像。



第 24 章

以太网及 LwIP 协议栈移植

互联网技术对人类社会的影响不言而喻，当今大部分电子设备都能以不同的方式接入互联网（Internet）。在家庭中 PC 常见的互联网接入方式是使用路由器（Router）组建小型局域网（LAN），利用调制解调器（Modem）经过电话线网络连接到互联网服务提供商（ISP），由互联网服务提供商把用户的局域网接入互联网。而企业或学校的局域网规模较大，常使用交换机组成局域网，经过路由以不同的方式接入到互联网中。

24.1 互联网模型

在互联网技术的发展过程中，曾出现过 OSI 七层模型和 TCP/IP 四层模型，两种模型分别有对应的协议，这两种模型和协议都有各自的优缺点。而最终 TCP/IP 改进模型（混合模型）及其协议被广泛使用。

设计网络时，为了降低网络设计的复杂性，对组成网络的硬件、软件进行封装、分层，这些分层即构成了网络体系模型。在两个设备相同层之间的对话、通信约定，构成了层级协议。设备中使用的所有协议加起来统称协议栈。这三个概念是互联网技术的核心。

虽然当前互联网的模型没有被明确规定，但我们一般可以使用五层的混合参考模型来理解，见图 24-1。

本书开篇便强调了分层在解决计算机科学问题中的强大作用，在各个章节中都有不同程度的体现，网络体系模型对网络传输问题的分层解决则再次印证了分层思想的作用。在这个网络模型中，每一层完成不同的任务，都提供接口供上一层访问。而在每层的内部，可以使用不同的方式来实现接口，因而内部的改变不会影响其他层。

在本参考模型中，数据链路层又被分为 LLC 层（逻辑链路层）和 MAC 层（媒体介质访问层）。目前，对于普通的接入网络终端的设备，LLC 层和

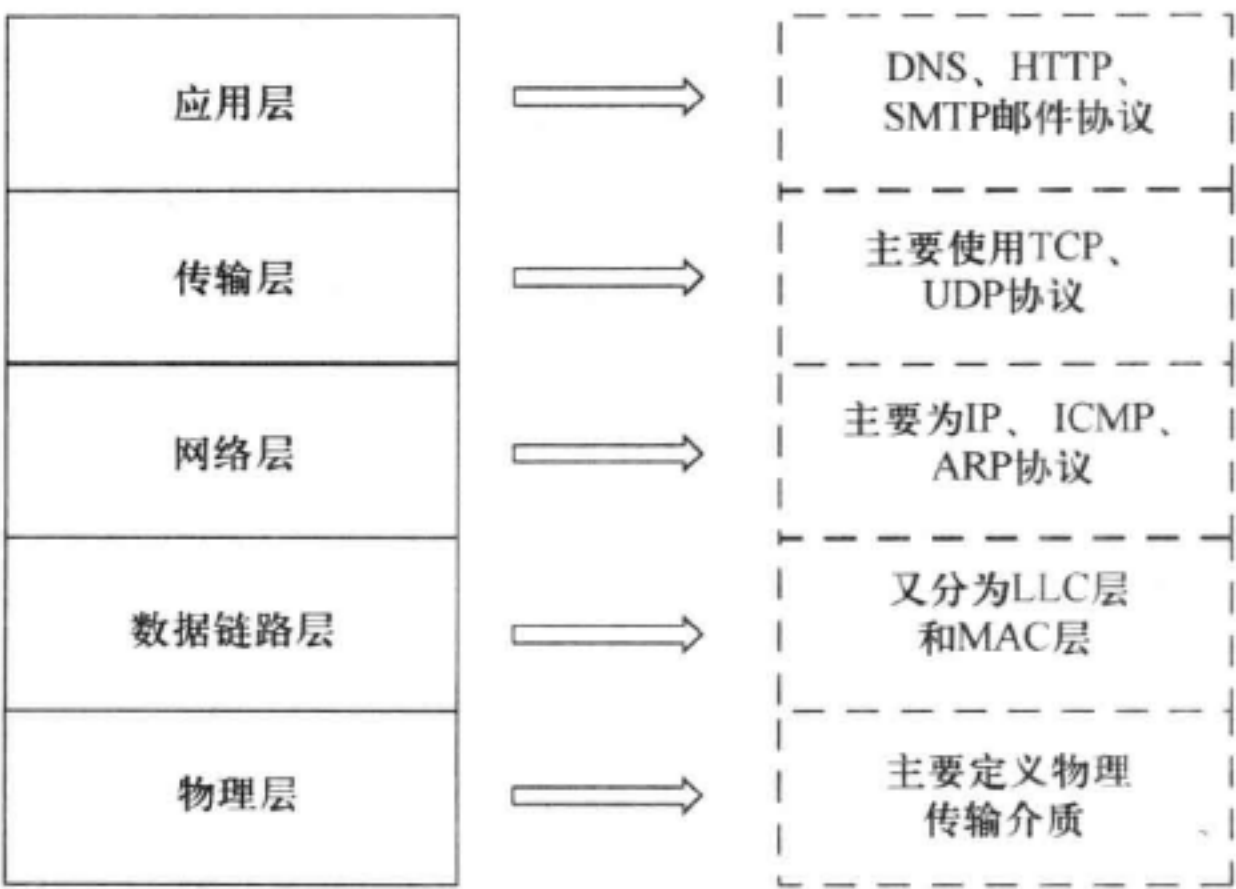


图 24-1 混合参考模型

MAC 层是软、硬件的分界线。如 PC 的网卡主要负责实现参考模型中的 MAC 子层和物理层，在 PC 的软件系统中则有一套庞大程序实现了 LLC 层及以上的所有网络层次的协议。

由硬件实现的物理层和 MAC 子层在不同的网络形式中有很大的区别，如以太网和 Wi-Fi，这是由物理传输方式决定的。但由软件实现的其他网络层次通常不会有太大区别，在 PC 上也许能实现完整的功能，支持所有协议，而在嵌入式领域则按需要进行裁剪。本章网络模型的硬件部分用以太网为例讲解，软件部分以 LwIP 协议栈讲解。

24.2 以太网

以太网（Ethernet）是互联网技术的一种，由于它在组网技术中占的比例最高，很多人直接把以太网理解为互联网。

以太网是指遵守 IEEE 802.3 标准组成的局域网，由 IEEE 802.3 标准规定的主要是位于参考模型的物理层（PHY）和数据链路层中的媒体接入控制子层（MAC）。在家庭、企业和学校所组建的 PC 局域网形式一般也是以太网，其标志是使用水晶头网线来连接（当然还有其他形式）。IEEE 还有其他局域网标准，如 IEEE 802.11 是无线局域网，俗称 Wi-Fi。IEEE 802.15 是个人域网，即蓝牙技术，其中的 802.15.4 标准则是当前非常热门的 ZigBee 技术。

渗透到工业控制、环境监测、智能家居的嵌入式设备产生了接入互联网的需求，利用以太网技术，嵌入式设备可以非常容易地接入到现有的计算机网络中。

24.2.1 PHY 层

在物理层，由 IEEE 802.3 标准规定了以太网使用的传输介质、传输速度、数据编码方式和冲突检测机制。

1. 传输介质

传输介质包括同轴电缆、双绞线（水晶头网线是一种双绞线）、光纤。根据不同的传输速度和距离要求，基于这三类介质的信号线又衍生出很多不同的种类。最常用的“五类线”适用于 100BASE-T 和 10BASE-T 的网络，它们的网络速率分别为 100 Mbps 和 10 Mbps。

2. 编码

为了让接收方在没有外部时钟参考的情况下也能确定每一位的起始、结束和中间位置，在传输信号时不直接采用二进制编码。在 10BASE-T 的传输方式中采用曼彻斯特编码，在 100BASE-T 中则采用 4B/5B 编码。

曼彻斯特编码把每一个二进制位的周期分为两个间隔，在表示“1”时，前半周期为高电平，后半周期为低电平；表示“0”时则相反，见图 24-2。

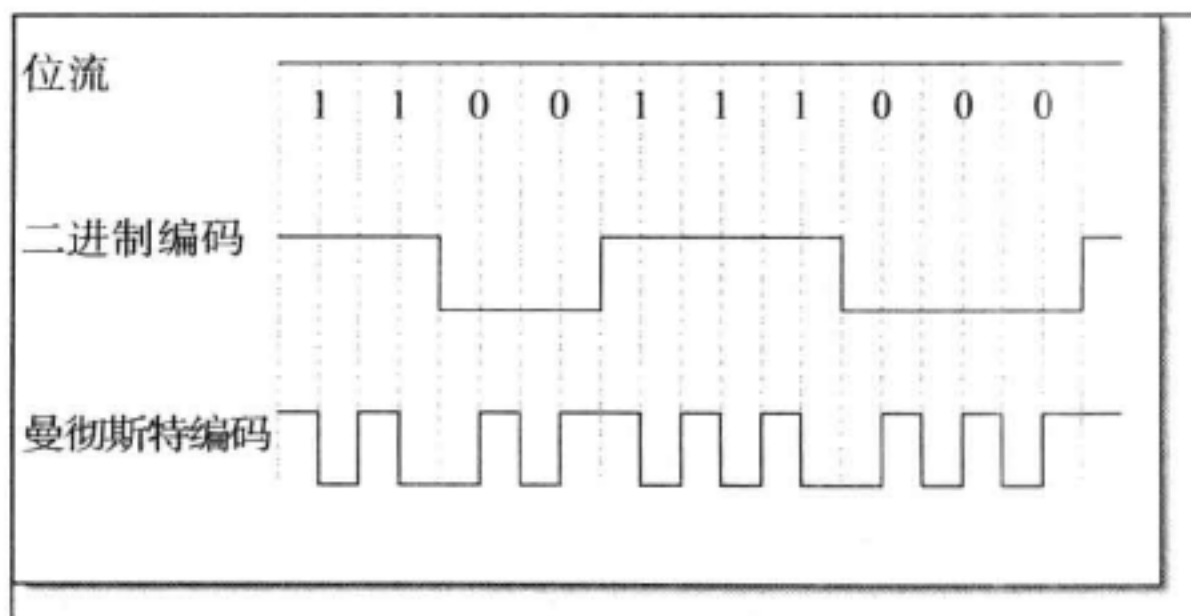


图 24-2 曼彻斯特编码

采用曼彻斯特编码在每个位周期都有电压变化，便于同步，但这样的编码方式效率太低。

在 100BASE-T 采用的 4B/5B 编码是把待发送数据位流的每 4 位分为一组，以特定的 5 位编码来表示，这些特定的 5 位编码能使数据流有足够多的跳变，达到同步的目的，而且效率也从曼彻斯特编码的 50% 提高到了 80%。

3. CSMA/CD 冲突检测

早期的以太网大多是多个节点连接到同一条网络总线上（总线型网络），存在信道竞争问题，因而每个连接到以太网上的节点都必须具备冲突检测功能。以太网具备 CSMA/CD 冲突检测机制，如果多个节点同时利用同一条总线发送数据，则会产生冲突，总线上的节点可通过将接收到的信号与原始发送的信号进行比较检测是否存在冲突，若存在冲突则停止发送数据，随机等待一段时间再重传。

现在大多数局域网组建的时候很少采用总线型网络，大多是一个设备接入到一个独立的路由或交换机接口，组成星型网络，不会产生冲突。但为了兼容，新出的产品还是带有冲突检测机制。

24.2.2 MAC 子层

1. MAC 的功能

MAC 子层是属于数据链路层的下半部分，它主要负责与物理层进行数据交接，如是否可以发送数据、发送的数据是否正确、对数据流进行控制等。它自动给来自上层的数据包加上一些控制信号，交给物理层。接收方得到正常数据时，自动去除 MAC 控制信号，把该数据包交给上层。

2. MAC 数据包

IEEE 对以太网上传输的数据包格式也进行了统一规定，见图 24-3。该数据包被称为 MAC 数据包。

MAC 数据包由前导字段、帧起始定界符、目标地址、源地址、数据包类型、数据域、填充域、校验和域组成。

- ❑ 前导字段，这是一段方波，用于使收发节点的时钟同步。而帧起始定界符则用于区分前导段与数据段的。前导字段和帧起始定界符在 MAC 收到数据包后会自动过滤掉。

- ❑ DA（目标地址）：要传

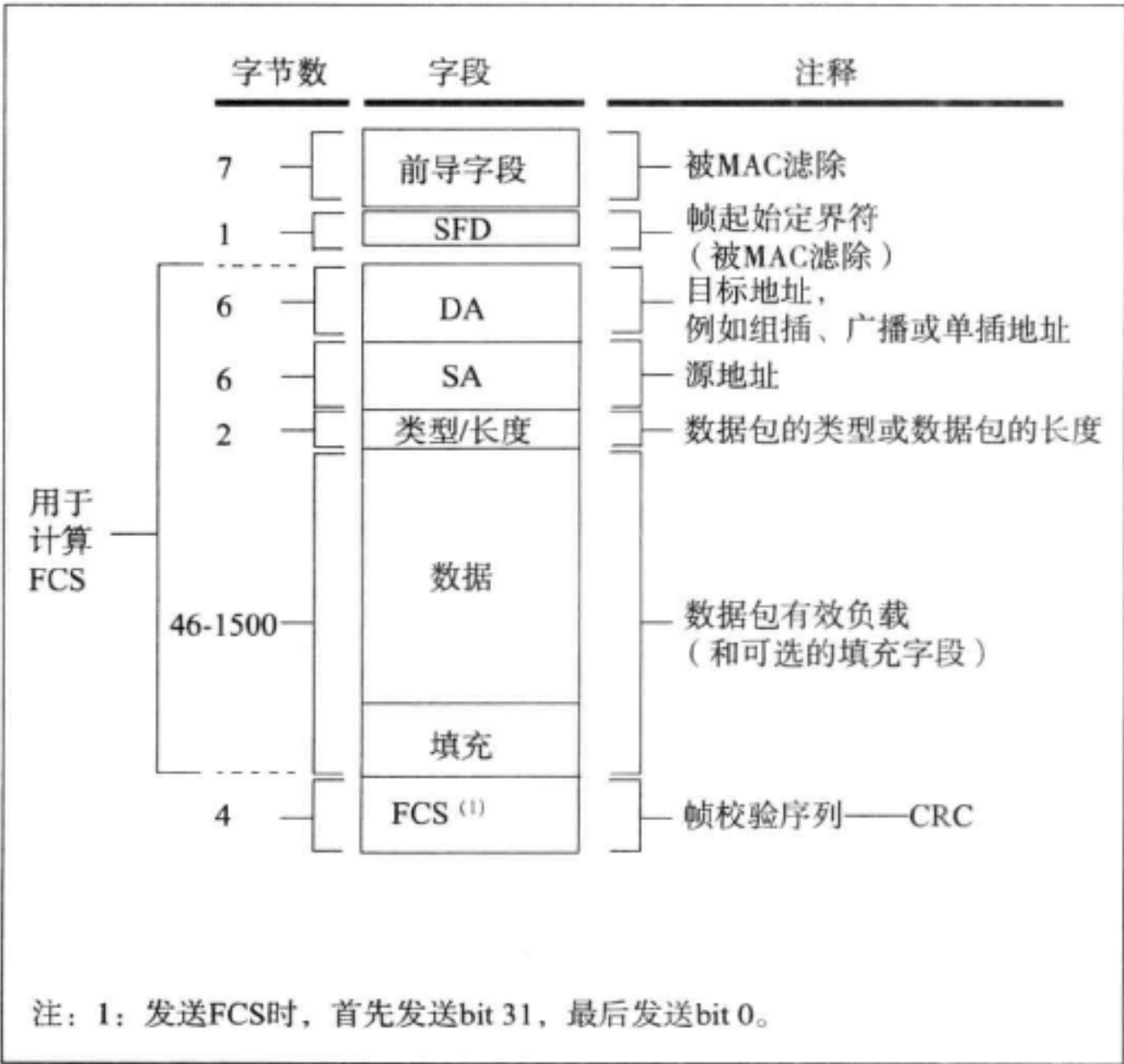


图 24-3 MAC 数据包格式

输的目标 MAC 地址。MAC 地址由 48 位数字组成，它是网卡的物理地址，以太网传输的最底层就是根据 MAC 地址来收发数据的。部分 MAC 地址用于广播和多播，在同一个网络里不能有两个相同的 MAC 地址。PC 的网卡在出厂时已经设置好了自身的 MAC 地址，但也可以通过一些软件进行修改，在嵌入式的以太网控制器中可由程序进行配置。

- ❑ SA（源地址）：自身的 MAC 地址。
- ❑ 数据包类型：本区域可以用来描述本 MAC 数据包是属于 TCP/IP 协议层的 IP 包、ARP 包还是 SNMP 包，也可以用来描述本 MAC 数据包数据段的长度。
- ❑ 数据段：数据段是 MAC 包的核心内容，它包含的数据来自 MAC 的上层。其长度可以从 0 ~ 1500 字节间变化。
- ❑ 填充域：由于协议要求整个 MAC 数据包的长度至少为 64 字节（接收到的数据包如果少于 64 字节会被认为发生冲突，数据包被自动丢弃），当数据段的字节少于 46 字节时，在填充域会自动填上无效数据，以使数据包符合长度要求。
- ❑ 校验和域：MAC 数据包的尾部是校验和域，它保存了 CRC 校验序列，用于检错。

24.2.3 以太网控制器

1. 集成 MAC 控制器的 MCU 接入方案

PHY 层和 MAC 层的实现是由以太网控制器实现的。部分 MCU 如 STM32F107 型号的芯片，集成了 MAC 控制器外设，在芯片外再外接一个 PHY 控制器和以太网变压器接口即可实现以太网功能。它们的关系可以类比 STM32 的 CAN 通信模型，STM32 集成了 CAN 控制器，但要实现 CAN 通信还需要外接 CAN 收发器芯片。以太网变压器的功能是增强、隔离信号及与 RJ45 水晶头进行连接。

STM32 的 MAC 控制器外设可通过 IEEE 协议规定的 MII 或 RMII、SMI 与 PHY 芯片进行通信。STM32F107 型号芯片使用 MII 接口与 PHY 构成的以太网接口，见图 24-4（省略了以太网变压器）。

2. 外接以太网控制器方案

对于没有集成以太网控制器的 MCU，可通过外接以太网控制器芯片接入以太网。本书使用的 STM32F103VET6 型号的 MCU 没有集成以太网控制器，所以采用 STM32 外接常用的嵌入式以太网控制器 ENC28J60 接入网络。

ENC28J60 芯片兼容 IEEE 802.3 的以太网控制器，集成 MAC 控制器和 10BASE-T PHY 控制器，自带缓冲区、DMA，使用 SPI 接口与 MCU 进行通信。MCU 使用 SPI 接口对 ENC28J60 芯片的寄存器写入控制参数和接收数据，实现以太网的功能。它与 MCU 连接组成的以太网接口见图 24-5。

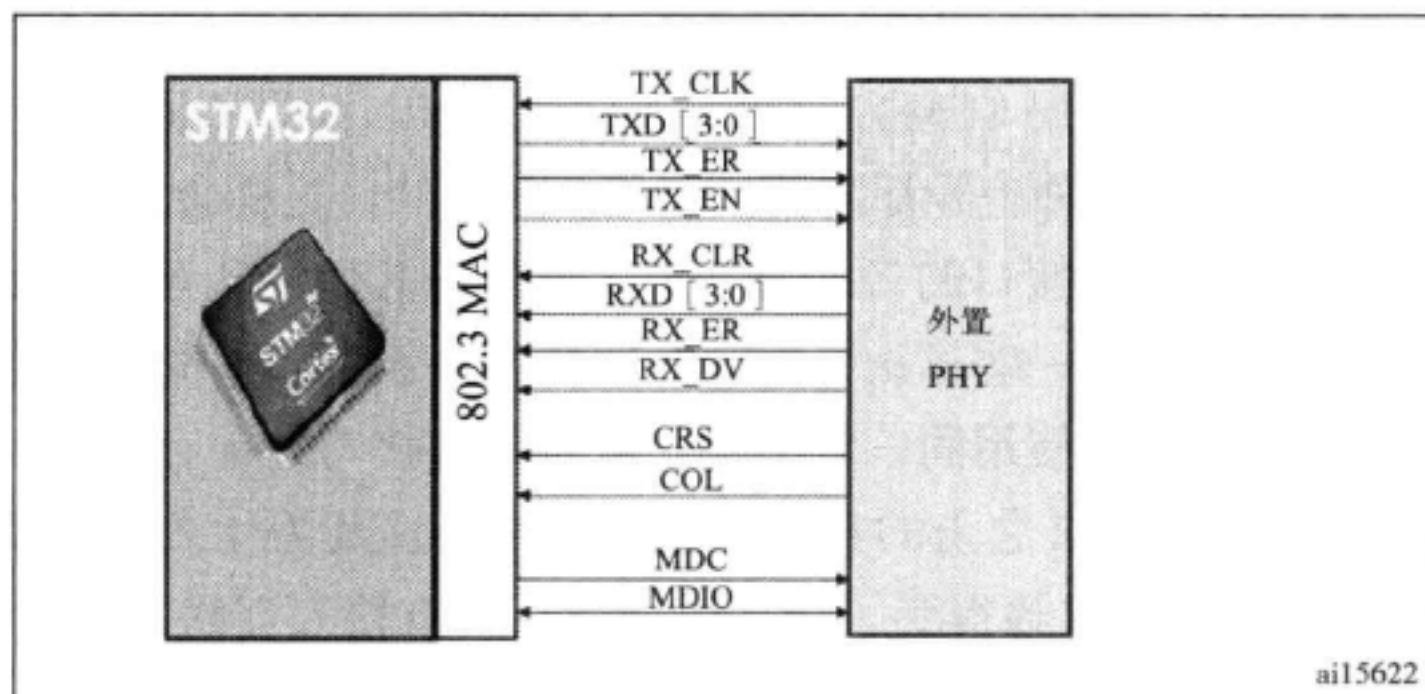


图 24-4 STM32F107 与 PHY 芯片构成以太网接口

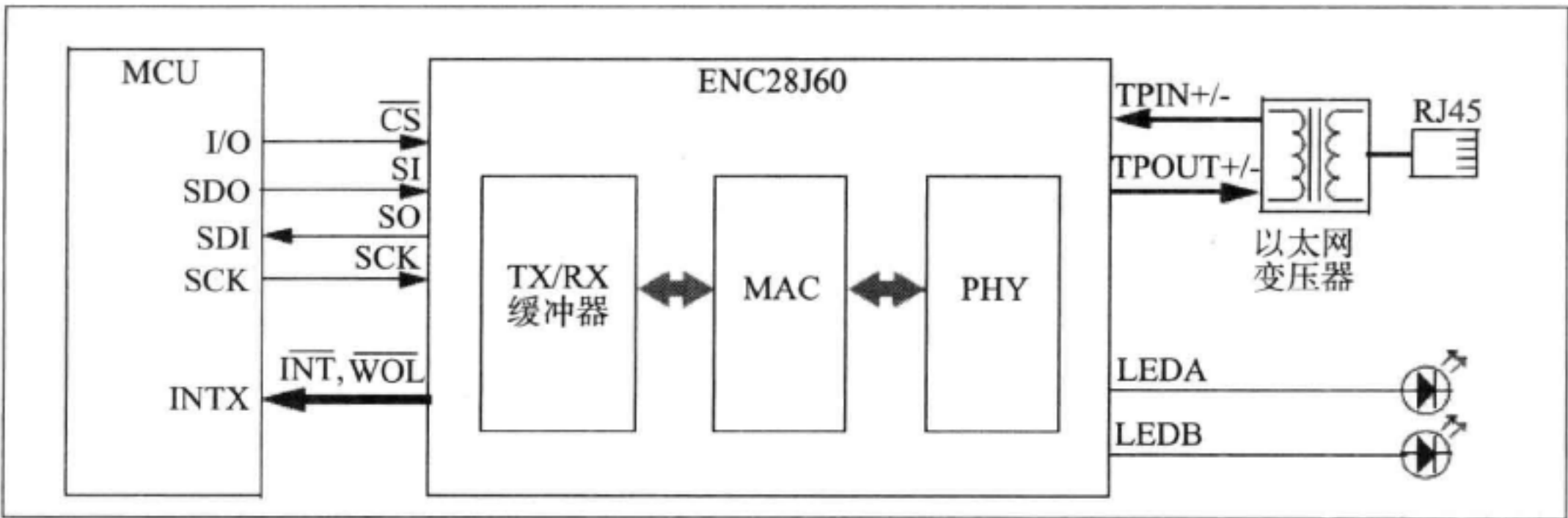


图 24-5 MCU 与 ENC28J60 组成以太网接口

使用以太网控制器，实现了混合模型的 MAC 子层和物理层，就可以利用它进行基本的收发 MAC 数据包了。

24.3 MAC 之上的网络层

24.3.1 为什么在 MAC 之上还有分层

如果两个 STM32 开发板，都实现了以太网接口的 MAC 子层和物理层，我们使用网线把这两个板子连接起来。针对接收端和发送端写两个代码，使得在 MAC 控制器收取到 MAC 数据包时，把位于数据段的内容提取出来，毫无疑问这是可以实现数据传输的，如果读者想尝试这样的功能，可以参考随书光盘的 NRF24L01 无线传输例程。

但是如果使用以太网仅仅是为了实现这样的功能真是大材小用了，而且还是有线传输，实在是鸡肋。想实现用 STM32 开发板与 PC 通信，如果不同的 PC 使用不同的系统或者设备使用嵌入式系统，针对每种差异都需要分别写一套软件，这样会过于复杂。

使用以太网接口的目的就是为了方便与其他设备互联，如果所有设备都约定使用一种互联方式，在软件上加一些层次来封装，这样不同系统、不同的设备通信就变得相对容易了。而且只要新加入的设备也使用同一种方式，就可以直接与之前存在于网络上的其他设备通信。这就是为什么产生了在 MAC 之上的其他层次的网络协议及为什么要使用协议栈的原因。又由于在各种协议栈中 TCP/IP 协议栈得到了最广泛使用，所有接入互联网的设备都遵守 TCP/IP 协议。所以，想方便地与其他设备互联通信，需要提供对 TCP/IP 协议的支持。

24.3.2 TCP/IP 协议中各层次的功能

用以太网和 Wi-Fi 作为例子，它们的 MAC 子层和物理层有较大的区别，但在 MAC 之上的 LLC 层、网络层、传输层和应用层的协议，是基本上同的，这几层协议由软件实现，并对各层进行封装。根据 TCP/IP 协议，各层要实现的功能如下：

LLC 层：处理传输错误；调节数据流，协调收发数据双方速度，防止发送方发送得太快而接

收方丢失数据。主要使用数据链路协议。

网络层：本层也被称为 IP 层。LLC 层负责把数据从线的一端传输到另一端，但很多时候不同的设备位于不同的网络中（并不是简单的网线的两头）。此时就需要网络层来解决子网路由拓扑问题、路径选择问题。在这一层主要有 IP 协议、ICMP 协议。

传输层：由网络层处理好了网络传输的路径问题后，端到端的路径就建立起来了。传输层就负责处理端到端的通信。在这一层中主要有 TCP、UDP 协议。

应用层：经过前面三层的处理，通信完全建立。应用层可以通过调用传输层的接口来编写特定的应用程序。而 TCP/IP 协议一般也会包含一些简单的应用程序，如 Telnet 远程登录、FTP 文件传输、SMTP 邮件传输协议。

实际上，在发送数据时，经过网络协议栈的每一层，都会给来自上层的数据添加上一个数据包的头，再传递给下一层。在接收方收到数据时，一层层地再把所在层的数据包的头去掉，向上层递交数据。见图 24-6。

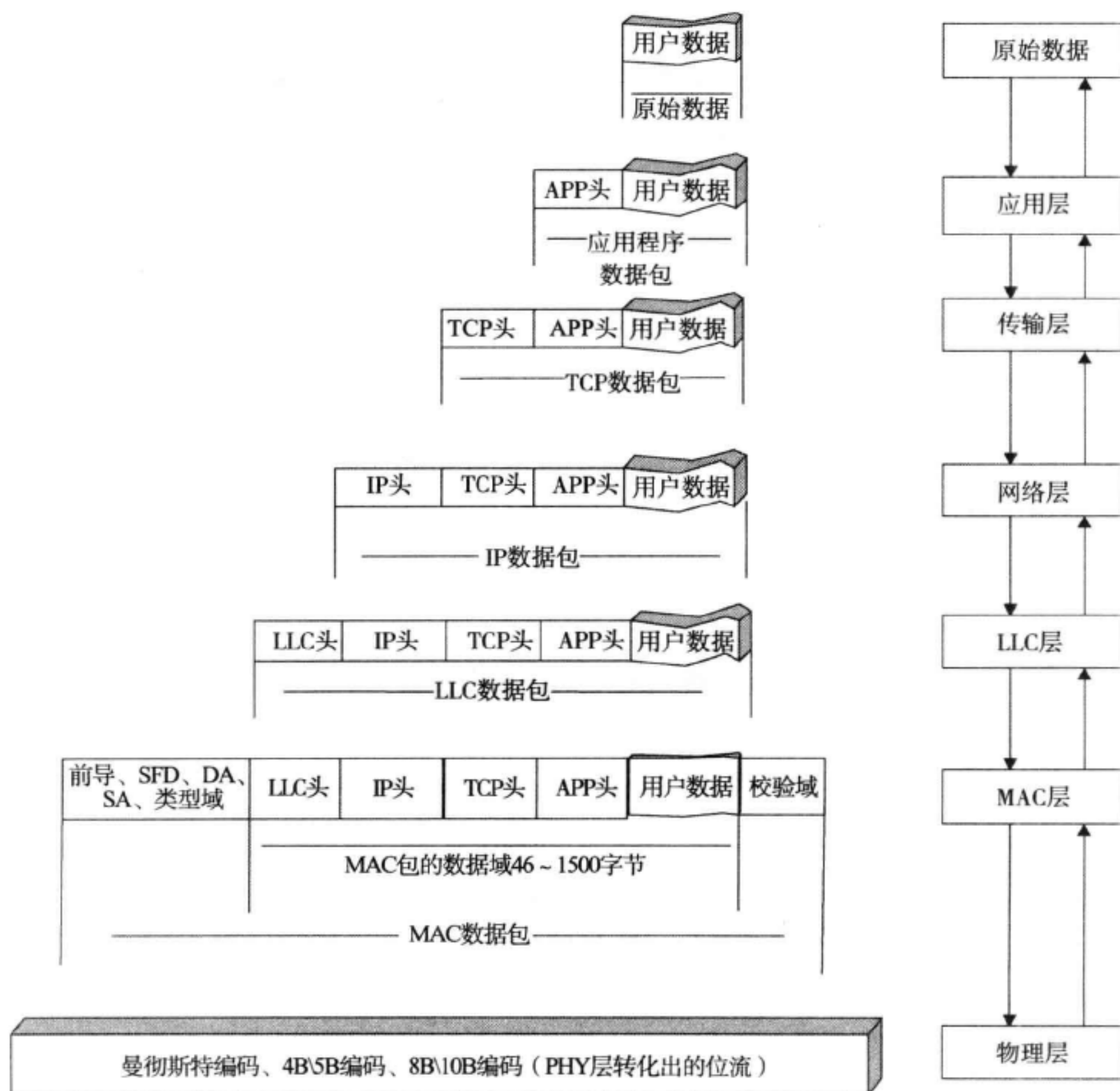


图 24-6 数据经过每一层的封装和还原

24.3.3 LwIP 协议栈

1. LwIP 协议栈简介

要实现 TCP/IP 协议栈，把每一层都处理好，这样的代码当然可以自己亲自写，但我们一般会移植更加稳定和性能优良的代码。最著名的实现了 TCP/IP 协议栈的代码是 BSD TCP/IP 协议栈，大多数专业的协议栈都是由它派生出来的，由它提供的接口 Socket 连接模式也几乎成为了通用标准。但 BSD 协议栈比较庞大，大多数小型嵌入式设备不宜使用，所以人们还开发出了 μ c/IP、LwIP、 μ IP 等适用于嵌入式设备使用的协议栈。

LwIP 是 Light Weight Internet Protocol 的缩写，是由瑞士计算机科学院 Adam Dunkels 等开发的适用于嵌入式领域的轻量级 TCP/IP 协议栈。它可以移植到含有操作系统的平台中，也可以在没有操作系统的平台下运行。由于它开源，占用的 RAM 和 ROM 比较少，支持较为完整的 TCP/IP 协议，且十分便于裁剪、调试，被广泛应用在中低端的 32 位控制器平台。

2. 获取 LwIP 协议栈

LwIP 协议栈是开源的，可以在网站 <http://download.savannah.gnu.org/releases/lwip/> 下载得到。目前最新版本 1.4.0 版刚推出不久，尚不清楚其稳定性如何，本章采用 LwIP 的 1.3.2 版本进行讲解。从该网站中可以下载到 lwip-1.3.2.zip 文件和相应版本的 contrib-1.3.0.zip 文件，它们包含了 LwIP 协议栈的核心源代码及应用、移植例程。

解压 lwip-1.3.2.zip 文件后，其目录 \lwip-1.3.2\src 的内容见图 24-7。

在 src 目录下的 api 文件夹保存的文件中，包含了适用于具有操作系统平台调用的应用层接口函数。core 文件夹下的文件内容为 LwIP 协议栈对于各种协议的实现。netif 下的文件则保存了与硬件底层关系比较紧密的函数。这三个文件夹下的都是 C 源文件，它们的头文件都被保存到 include 文件夹中。在实际应用中，可根据需要进行裁剪。

在移植的时候，我们也常常需要利用 contrib-1.3.0.zip 中的文件。解压后，在 \contrib-1.3.0_\contrib\ports 目录下有一些针对特定平台移植时使用的文件，选择进入其中一个目录，如 \contrib-1.3.0_\contrib\ports old\6502\include\arch，其内容见图 24-8。

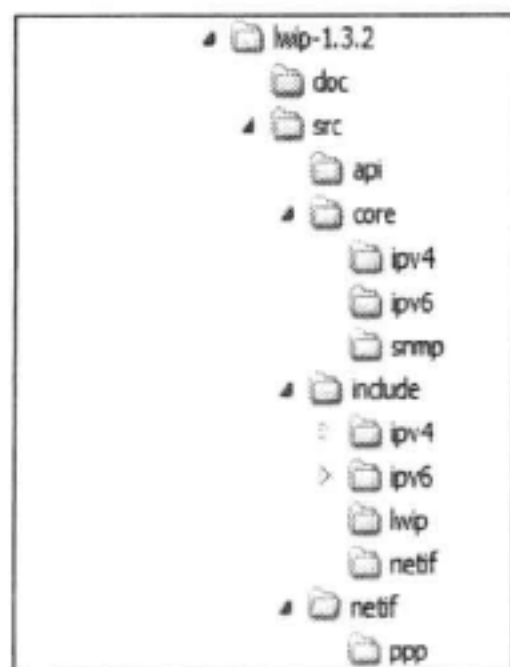


图 24-7 lwip 的 src 目录



图 24-8 特定头文件

我们需要用到的是 cc.h、perf.h 和 sys_arch.h 文件，通常把它复制出来存放到自己工程中的 arch 文件夹中。cc.h 包含了 LwIP 对于基本数据类型的定义，sys_arch.h 定义了与系统有关的信号量、邮箱及线程。

24.4 ENC28J60+LwIP 以太网实验

24.4.1 实验描述及工程文件清单

1. 实验描述

使用配套开发板利用 ENC28J60 以太网控制器接入局域网，移植 LwIP 协议栈，并利用协议栈提供的函数在 STM32 上建立服务器和创建 Telnet 应用。

建立服务器后可以使用 PC 的浏览器访问网页，通过鼠标点击网页按钮控制开发板上的 LED 灯。Telnet 应用即利用 PC 上的 Telnet 程序向 STM32 输入命令，控制 LED 灯。

本实验分析十分复杂，对网络应用不熟悉的读者，务必先阅读本章的实验步骤和实验现象，这样既能有个大概了解，也能激起学习的兴趣。

2. 硬件连接

- ☐ PB13: ENC28J60-INT
- ☐ PA6-SPI1-MISO: ENC28J60-SO
- ☐ PA7-SPI1-MOSI: ENC28J60-SI
- ☐ PA5-SPI1-SCK: ENC28J60-SCK
- ☐ PA4-SPI1-NSS: ENC28J60-CS
- ☐ PE1: ENC28J60-RST

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_spi.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_usart.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/SysTick.c
- ☐ USER/usart1.c

- ☐ USER/led.c
- ☐ USER/ENC28J60.c
- ☐ USER/SPI.c
- ☐ USER/httpd.c
- ☐ USER/netconfig.c
- ☐ USER/cmd.c

5. LwIP 文件

使用 1.3.2 版本 LwIP 协议栈，以及 lwip-1.3.2.zip 文件解压后 src 中的源文件，具体见工程环境配置。

该实验硬件连接见图 24-9。

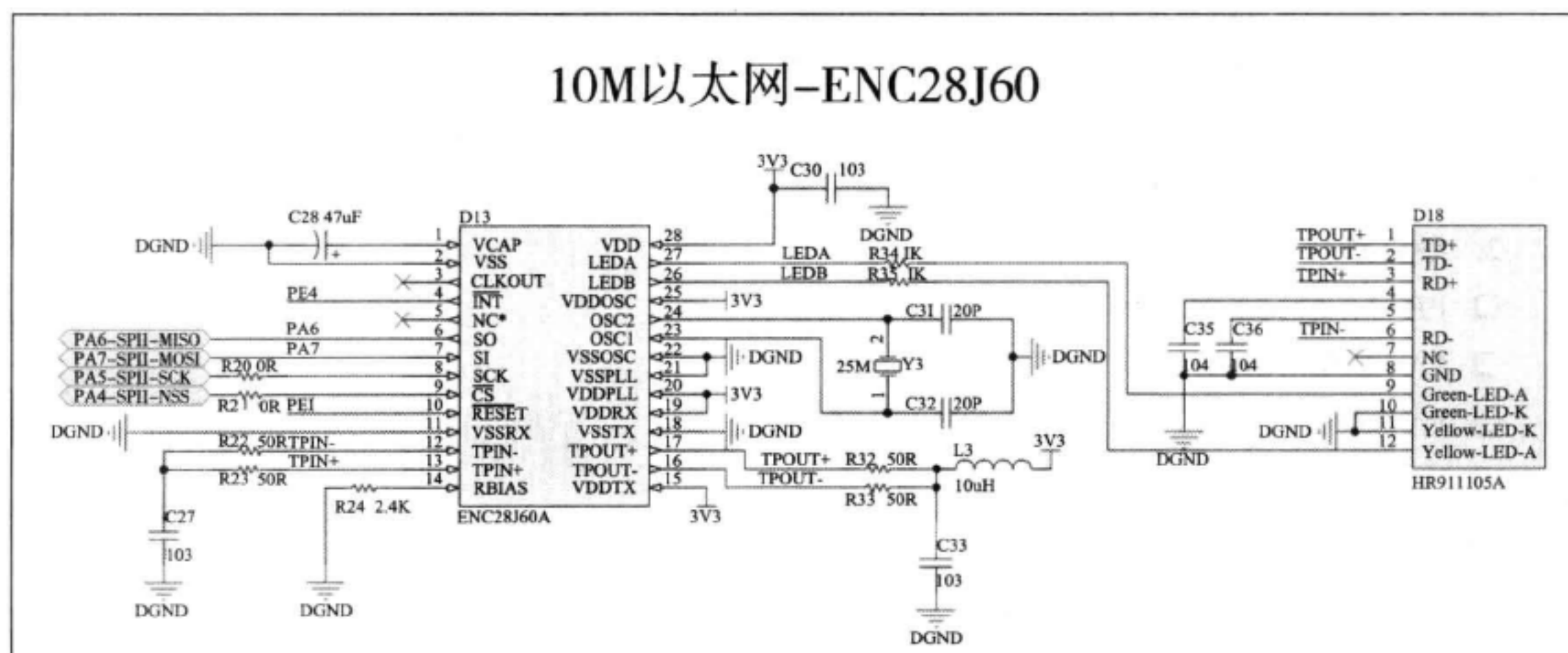


图 24-9 ENC28J60 与 STM32 硬件连接图

24.4.2 配置工程环境

以太网实验中我们用到了 GPIO、RCC、SPI、USART 外设，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_spi.c、stm32f10x_usart.c。实验中还使用了旧工程中的 led.c 和 systick.c 文件，用于网络控制 LED 和给 LwIP 协议栈提供定时功能。

接下来就要新建文件编写 SPI 初始化函数和编写 ENC28J60 的驱动文件了，由于本次实验使用的文件十分多，所以把 SPI.C 文件和 ENC28J60.C 文件放到新的分组中管理。

在应用层则创建了三个文件用来配置网卡、服务器和 Telnet 应用程序，这三个文件分别为 httpd.c、netconfig.c 和 cmd.c。

最后，将 LwIP 协议栈整个文件夹 lwip-1.3.2 复制到目录下，从 contrib-1.3.0.zip 中复制 cc.h 等头文件保存到新建的 ARCH 文件夹下。把使用到的 LwIP 协议栈的源文件都按它在 src 的分组添加到工程，添加后的结果见图 24-10。

其中带有“+”号的 LwIP 组下，都包含 LwIP 协议文件包中相应文件夹的所有源文件（文件太多，无法一一列出，请参照例程），没有“+”号的 ipv6 和 ppp 工程组相应的源文件则因为没有被使用到而未被添加进工程。

添加完工程后，还要在 C/C++ 编译器选项中添加头文件路径，主要为 LwIP 的头文件，见图 24-11。

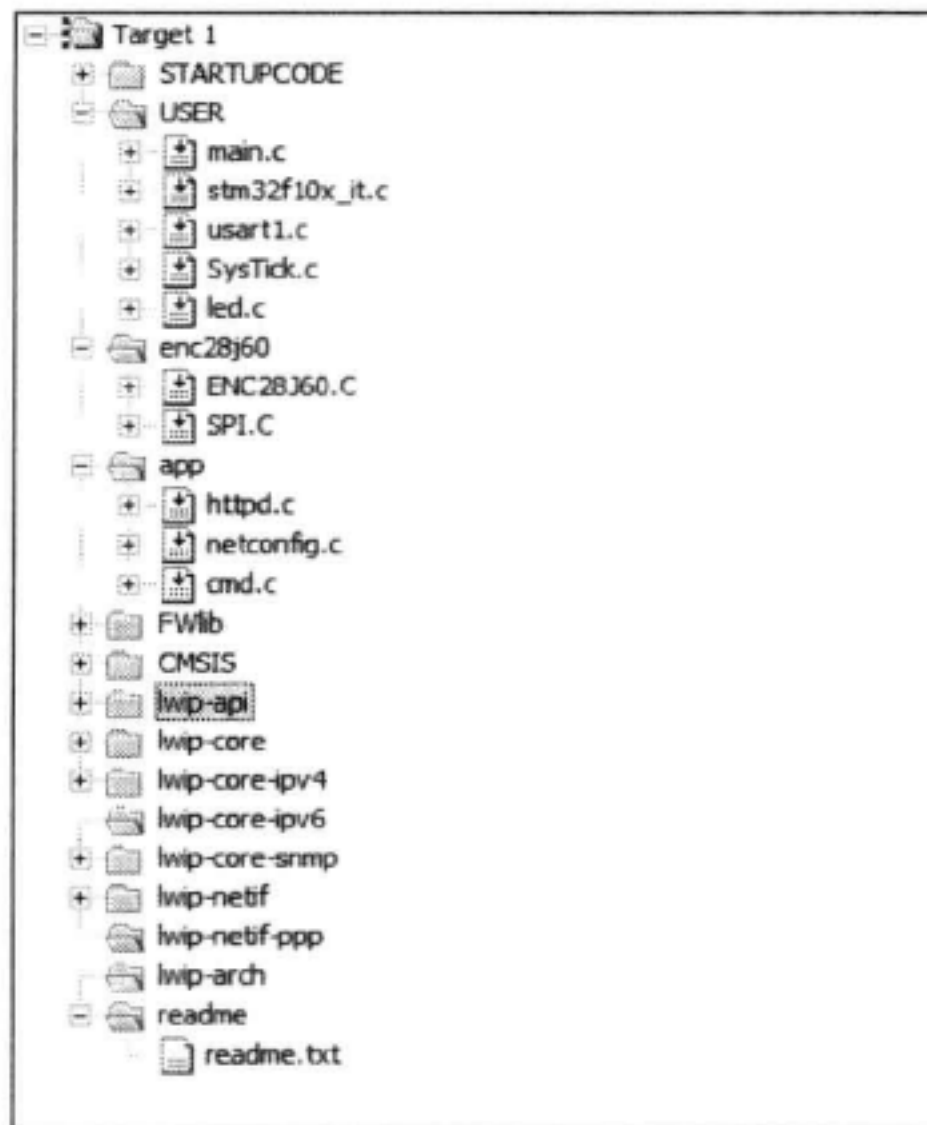


图 24-10 添加工程文件

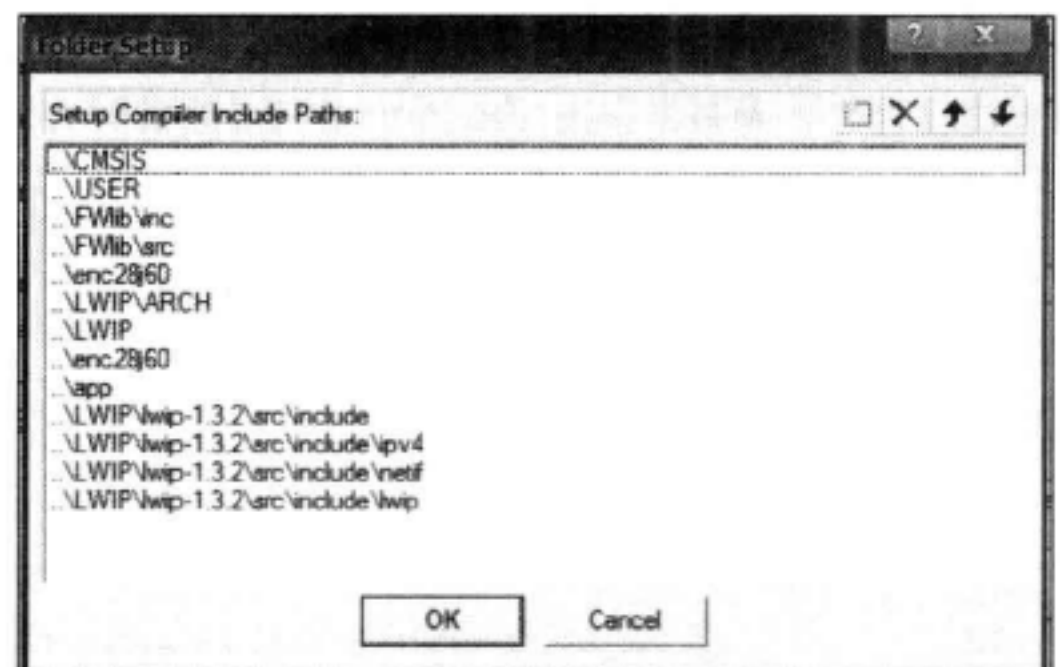


图 24-11 添加头文件路径

24.4.3 main 文件

无论工程多么复杂，坚持从 main 文件理解不动摇。本工程中的 main 文件主要内容见代码清单 24-1。

代码清单 24-1 以太网例程的 main 函数

```

1.  /*****
*****
2.
3.  __IO uint32_t LocalTime = 0;
4.  /* 该变量每 10ms 改变一次，增量为 10*/
5.
6.  /*
7.   * 函数名：main
8.   * 描述   ：主函数
9.   * 输入    ：无
10.  * 输出    ：无
11.  * 调用    ：
12.  */

```

```

13.int main(void)
14.{
15.    /* 初始化串口 */
16.    USART1_Config();
17.
18.    /* 初始化 以太网 SPI 接口 */
19.    ENC_SPI_Init();
20.
21.    /* 初始化 systick, 用于定时轮询输入或给 LWIP 提供定时 */
22.    SysTick_Init();
23.
24.    /* 初始化 LED 使用的端口 */
25.    LED_GPIO_Config();
26.
27.    printf("\r\n***** 野火 STM32_enc8j60+lwip 移植实验 *****\r\n");
28.
29.    /* 初始化 LWIP 协议栈 */
30.    LwIP_Init();
31.
32.    /* 初始化 web server 显示网页程序 */
33.    httpd_init();
34.
35.    /* 初始化 telnet 远程控制 程序 */
36.    CMD_init();
37.
38.    /* Infinite loop */
39.    while ( 1 )
40.    {
41.        /* 轮询 */
42.        LwIP_Periodic_Handle(LocalTime);
43.    }
44.}
45.

```

在 main 函数中，使用 USART1_Config() 配置了串口驱动；用 ENC_SPI_Init() 配置好了与 ENC28J60 进行通信的 SPI 接口；SysTick_Init() 配置了用于给 LwIP 定时的 SysTick；LED_GPIO_Config() 则对控制 LED 的 GPIO 进行了初始化。

LwIP_Init() 初始化了 LwIP 协议栈，它设置了网络接口的 IP、子网掩码、网关，并使能了 ENC28J60。httpd_init() 完成了建立服务器所需的状态，如监听网络端口、配置连接后调用的各种函数，以完成网页控制 LED 的功能。类似地，CMD_init() 完成了 Telnet 控制 LED 的准备。

最后，就在 while 死循环中调用 LwIP_Periodic_Handle()，根据 LocalTime 变量的计时值，轮询网络接口和给 LwIP 协议栈系统提供定时。

24.4.4 LwIP 对底层数据结构的封装

了解了 main 函数的执行流程后，读者可能会觉得奇怪，为何没有关于 ENC28J60 接口驱动的初始化呢（ENC_SPI_Init() 函数仅仅完成了 SPI 接口的 GPIO、SPI 模式配置）？实际上它是在 LwIP_Init() 函数完成的，但由于封装的层次比较多，所以不仔细分析很难了解它是如何完成对底层的 ENC28J60 进行初始化的。

由于硬件底层使用的设备十分容易发生变化,同时也为了使 LwIP 的适用范围更广,就像文件系统的底层驱动是留给移植者完成一样,LwIP 协议栈给底层的驱动提供了接口,如初始化网卡、发送数据包、接收数据包等最底层的操作。这些接口定义位于 \lwip-1.3.2\src\netif 目录下的 ethernetif.c 文件,该文件已经被添加进了工程中,接口的具体实现因使用的网络接口不同而有所区别,需要移植者来完成。

在编写这些与接口相关的内容时,涉及两个关系密切的 LwIP 的底层数据结构: pbuf 和 netif 结构体。

1. pbuf 结构体

LwIP 的底层接口涉及大量对 pbuf、netif 结构体赋值的内容,我们先来分析它们的结构体成员。

当 MAC 层收到 MAC 数据包的时候,它会把 MAC 包中的前导字段和帧起始定界符过滤掉,把剩余的数据段都保存起来,交给 LwIP 的 pbuf 结构体类型的变量,由 LwIP 协议栈进行处理。pbuf 结构体在 pbuf.h 文件中定义见代码清单 24-2。

代码清单 24-2 pbuf 结构体定义

```

1. struct pbuf {
2.     /** next pbuf in singly linked pbuf chain */
3.     struct pbuf *next;
4.
5.     /** pointer to the actual data in the buffer */
6.     void *payload;
7.
8.     /**
9.      * total length of this buffer and all next buffers in chain
10.     * belonging to the same packet. */
11.     u16_t tot_len;
12.
13.     /** length of this buffer */
14.     u16_t len;
15.
16.     /** pbuf_type as u8_t instead of enum to save space */
17.     u8_t type; /*pbuf_type*/
18.
19.     /** misc flags */
20.     u8_t flags;
21.
22.     /**
23.      * the reference count always equals the number of pointers
24.      * that refer to this pbuf. This can be pointers from an application, * the stack itself,
25.      * or pbuf->next pointers from a chain.
26.     */
27.     u16_t ref;
28. };

```

其成员作用如下:

1) next。这是一个指向 pbuf 类型的指针,它用于指向下一个 pbuf 结构体。由于每个 MAC 包的数据段最大只是 1500 字节,而来自上层的数据包往往很大,LwIP 把大数据包分装到多个

pbuf 结构体，并把这些 pbuf 组成链表，通过 next 指针来索引。

2) payload。这是一个指向实际数据的指针。当 MAC 控制器接收到 MAC 数据包时，整个 MAC 包（包括 MAC 地址）直接被存储到 payload 指向的存储区。当需要发送数据时，LwIP 一层层地给原始数据加上数据头，最终组装成 MAC 包的格式，也保存到 payload 指向的存储区，然后把这些数据交给底层的 MAC 控制器发送出去。

3) tot_len 和 len。len 成员表示本 pbuf 结构体的长度，tot_len 表示本 pbuf 及下一个 pbuf (next 指向的 pbuf) 长度的和。例如，pbuf A 的 next 成员指向 pbuf B，若 pbuf A 的 len 值为 500，pbuf B 的 len 值为 600，那么 pbuf A 的 tot_len 长度就等于 (500+600)，而 pbuf B 的 tot_len 长度就等于 600 加上下一个 pbuf 的 len。若 pbuf C 是 pbuf 链表中的最后一个，那么 pbuf C 的 len 就等于它的 tot_len，从而就可以判断链表的结尾。

4) type 和 flags。type 表示 pbuf 的存储类型，共有四种，分别为 PBUF_RAM、PBUF_ROM、PBUF_REF 和 PBUF_POOL，针对不同的应用而使用不同的存储方式。flags 在存储分配时被赋值为 0，作用不明，可能是作者为了兼容而保留的。

5) ref。ref 用于记录 pbuf 的访问次数，在 LwIP 中存在根据访问次数来决定内存释放的机制。从上面的分析可以了解到 LwIP 利用 pbuf 结构体进行与底层硬件的交接，用 payload 存储数据包的指针，为了便于管理，使用 next、len、tot_len、type、ref 成员来定义 pbuf 的属性。

最后，参考图 24-12 可以让我们更加清晰地了解 pbuf 的结构，该图是存储形式为 PBUF_POOL 类型的结构体存储分布，其他类型的 pbuf 也是类似的。

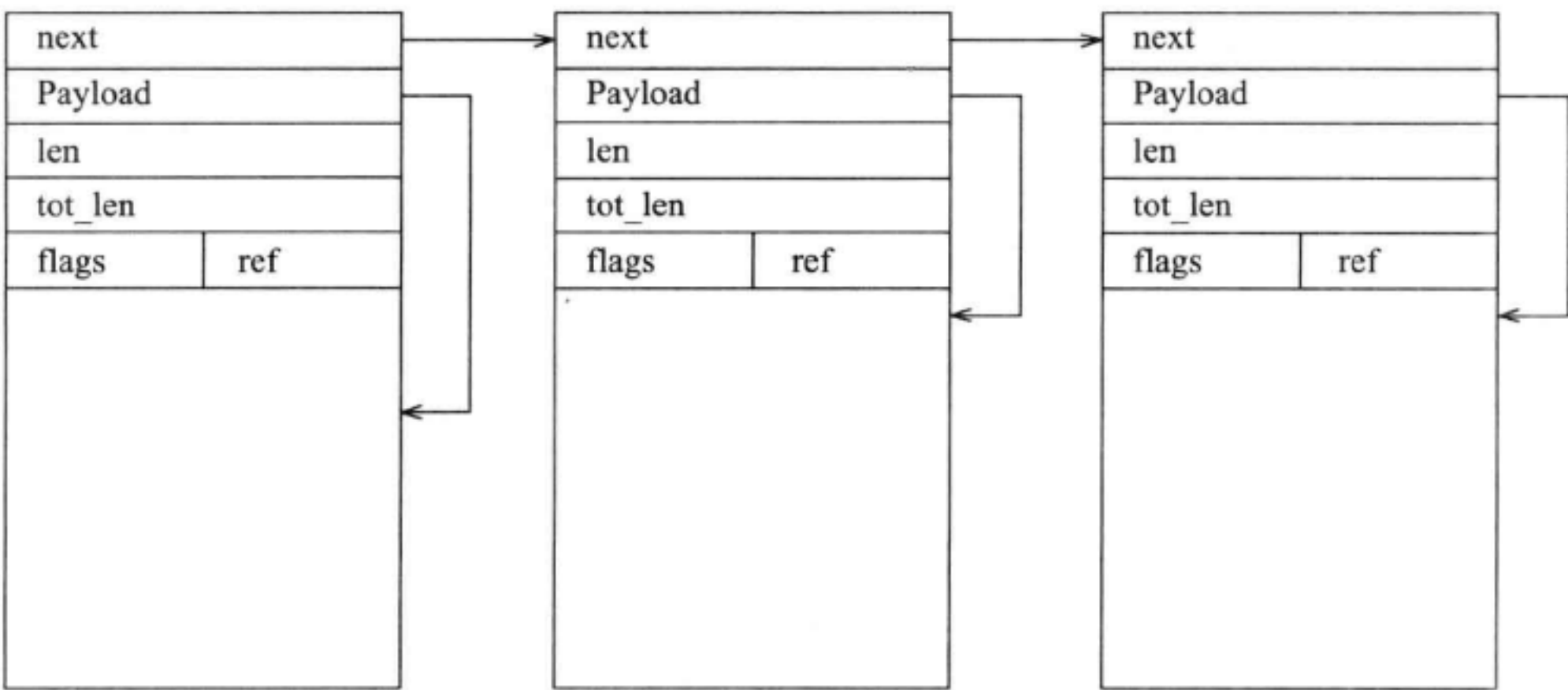


图 24-12 PBUF_POOL 类型的存储方式

2. netif 结构体

LwIP 通过 pbuf 建立了与底层硬件收发数据包的数据结构，可以实现数据的管理。但很多时候，收发数据还需要清楚网卡的状态，特别是网卡地址、IP 地址、网关等设置，有的设备上还可能有多网卡，这就需要一个结构体来保存这些信息，让 LwIP 进行管理和使用。这些信息被保存在 netif 结构体中，它的定义在 netif.h 文件，见代码清单 24-3。（为了便于讲解，在初学时，只

分析重点代码，我们只保留了重点与网卡相关的结构体成员，条件编译部分都被删除了，想了解完整版的读者可参考源码。)

代码清单 24-3 netif 结构体

```

1. struct netif {
2.     /** pointer to next in linked list */
3.     struct netif *next;
4.
5.     /** IP address configuration in network byte order */
6.     struct ip_addr ip_addr;
7.     struct ip_addr netmask;
8.     struct ip_addr gw;
9.
10.    /** This function is called by the network device driver
11.     *   to pass a packet up the TCP/IP stack. */
12.    err_t (* input)(struct pbuf *p, struct netif *inp);
13.
14.    /** This function is called by the IP module when it wants
15.     *   to send a packet on the interface. This function typically
16.     *   first resolves the hardware address, then sends the packet. */
17.
18.    err_t (* output)(struct netif *netif, struct pbuf *p,
19.                     struct ip_addr *ipaddr);
20.
21.    /** This function is called by the ARP module when it wants
22.     *   to send a packet on the interface. This function outputs
23.     *   the pbuf as-is on the link medium. */
24.    err_t (* linkoutput)(struct netif *netif, struct pbuf *p);
25.
26.    /** This field can be set by the device driver and could point
27.     *   to state information for the device. */
28.
29.    void *state;
30.
31.    /** maximum transfer unit (in bytes) */
32.    u16_t mtu;
33.
34.    /** number of bytes used in hwaddr */
35.    u8_t hwaddr_len;
36.
37.    /** link level hardware address of this interface */
38.
39.    u8_t hwaddr[NETIF_MAX_HWADDR_LEN];
40.    /** flags (see NETIF_FLAG_ above) */
41.
42.    u8_t flags;
43.    /** descriptive abbreviation */
44.
45.    char name[2];
46.    /** number of this interface */
47.
48.    u8_t num;
49.
50.};

```

各成员的作用如下：

1) next。这个 next 指针是 netif 结构体类型，它与 pbuf 的 next 指针作用类似，也是用作链表。netif 结构体是用来存储网卡属性的，由 netif 结构体构成的链表即表示同一个设备上不同网卡的属性，LwIP 就通过该 next 指针访问这个链表。

2) ip_addr、netmask 和 gw。这三个结构体成员分别用来存储本网卡的 IP 地址、子网掩码和网关。

3) err_t (*input)(struct pbuf *p, struct netif *inp)。这个结构体成员第一眼看上去有点奇怪，与以往碰到的都不一样，实际上这是一个函数指针。这个函数用来接收底层硬件输入的数据包。函数的返回值是 err_t 类型，它是由 LwIP 定义的一种用于表示操作成功、失败等参数的变量。函数的输入参数为 pbuf 类型的 p 指针和 netif 类型的 inp 指针，表示我们要接收 inp 网卡通过底层硬件接收到的数据包，并把数据包保存到结构体 p 中。

4) err_t (*output)(struct netif *netif, struct pbuf *p, struct ip_addr *ipaddr)。本成员也是一个函数指针，它指向一个用于 IP 层输出的函数。在 IP 层有数据包需要发送时，就通过该指针调用 output 函数。本函数输入参数有 netif、pbuf 和 ip_addr 三个参数，表示使用 netif 网卡把 pbuf 结构体的内容发送到 IP 地址为 ip_addr 的设备中。但由于在硬件底层收发数据时是使用 MAC 地址的，所以本函数把 IP 地址经过转化得到 MAC 地址后，调用下一个成员——linkoutput 函数。

5) err_t (*linkoutput)(struct netif *netif, struct pbuf *p)。这个函数指针是指向最底层的网卡发送数据包函数。它是被上面的 output 函数调用的。在 output 函数中的 pbuf 数据包实际上缺少了 MAC 包中的 DA 段（目标 MAC 地址），output 函数把它的输入参数 IP 地址转化成 MAC 地址后，添加到 pbuf 的 payload 成员，组成完整的 MAC 包，再由本函数 linkoutput 输出到网络中。

6) state。state 指针指向用户感兴趣的信息，这部分可以由用户自行配置，也可以不使用。

7) mtu。本成员记录了最大传输单元，我们知道 MAC 数据段的最大长度为 1500 字节，所以本成员我们一般会直接赋值为 1500。

8) hwaddr_len 和 hwaddr[NETIF_MAX_HWADDR_LEN]。hwaddr_len 存储的是 MAC 地址长度，hwaddr[] 数组则存储了本网卡的 MAC 完整地址。

9) flags。本成员保存了网卡允许使用的功能，如使用广播地址、ARP 功能等。

10) name 和 num。name 用于保存网络接口的名字，可以自由设置，名字是两个字节的 ASCII 编码。当出现相同网络接口名字时，使用 num 的数值进行区分。

24.4.5 初始化协议栈

了解了 pbuf、netif 结构体后，我们进入被 main 函数调用的 LwIP_Init() 的具体代码中（删减了部分扩展功能的代码），该函数位于自定义的 netconfig.c 文件，见代码清单 24-4。

代码清单 24-4 LwIP_Init() 函数

```

1. /*
2.  * 函数名：LwIP_Init
3.  * 描述   ：初始化 LwIP 协议栈，主要是把 ENC28J60 与 LwIP 连接起来。
4.             包括 IP、MAC 地址，接口函数

```



```

5.  * 输入   : 无
6.  * 输出   : 无
7.  * 调用   : 外部调用
8.  */
9. void LwIP_Init( void )
10. {
11.     struct ip_addr ipaddr;
12.     struct ip_addr netmask;
13.     struct ip_addr gw;
14.
15.     /* 调用 LWIP 初始化函数,
16.      初始化网络接口结构体链表、内存池、pbuf 结构体 */
17.     lwip_init();
18.
19.     IP4_ADDR(&ipaddr, 192, 168, 1, 18);           // 设置网络接口的 IP 地址
20.     IP4_ADDR(&netmask, 255, 255, 255, 0);         // 子网掩码
21.     IP4_ADDR(&gw, 192, 168, 1, 1);                // 网关
22.
23.     /* 初始化 ENC28J60 与 LwIP 的接口, 参数为网络接口结构体、IP 地址、
24.      子网掩码、网关、网卡信息指针、初始化函数、输入函数 */
25.     netif_add(&enc28j60, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &ethernet_
        input);
26.
27.     /* 把 ENC28J60 设置为默认网卡 */
28.     netif_set_default(&enc28j60);
29.
30.     /* When the netif is fully configured this function must be called*/
31.     netif_set_up(&enc28j60); // 使能 ENC28J60 接口
32. }

```

函数执行流程如下：

1) 第 11 ~ 13 行, 定义了三个 ip_addr 类型的变量, 用于存储 IP 地址、子网掩码和网关, 这些变量在第 19 ~ 21 行被赋值, 本实验的网卡这三个属性就是由它们配置的。

2) 第 17 行, 调用 lwip_init() 函数 (名字跟 LwIP_Init() 很相似, 注意区分), lwip_init() 是由 LwIP 协议栈定义的函数, 在使用 LwIP 协议栈相关的内容前, 要先调用它对网卡结构体链表、内存池、pbuf 进行初始化。

3) 第 25 行, 调用 netif_add() 函数, 它把我们配置好的网卡属性赋值到 netif 类型的 enc28j60 变量中, 它是在本文件的开头定义的 (被省略了), 如 IP 地址、子网掩码、网关地址、网卡初始化函数 ethernetif_init()、网卡输入函数 ethernet_input()。在本函数之后的 LwIP 协议栈操作, 只要使用到了相关的底层服务, 都通过这个 netif 类型的 enc28j60 变量来访问。代码中的 28 ~ 31 行调用两个函数的作用分别是设置 enc28j60 结构体指向的网卡为默认网卡, 并使能。

netif_add() 函数

netif_add() 是上面提到的重点函数, 它是由 LwIP 协议栈定义的, 它的使用方法与我们使用 STM32 固件库初始化外设时类似, 先把初始化结构体配置好, 然后调用 xxx_init 函数把这些配置写入寄存器。netif_add() 函数则把我们对网卡属性的配置写入该函数的输入参数 netif 中, 本实验

调用时该参数是变量 enc28j60。netif_add() 在 netif.c 文件的定义见代码清单 24-5（有删减）。

代码清单 24-5 netif_add() 函数

```

1. struct netif *
2. netif_add(struct netif *netif, struct ip_addr *ipaddr, struct ip_addr *netmask,
3.   struct ip_addr *gw,
4.   void *state,
5.   err_t (* init)(struct netif *netif),
6.   err_t (* input)(struct pbuf *p, struct netif *netif))
7. {
8.   static u8_t netifnum = 0;
9.
10.  /* reset new interface configuration state */
11.  netif->ip_addr.addr = 0;    // 复位结构体成员
12.  netif->netmask.addr = 0;
13.  netif->gw.addr = 0;
14.  netif->flags = 0;
15.
16.  /* remember netif specific state information data */
17.  netif->state = state;
18.  netif->num = netifnum++;
19.  netif->input = input;    // 输入函数
20.
21.  netif_set_addr(netif, ipaddr, netmask, gw);    // 设置三个地址
22.
23.  /* call user specified initialization function for netif */
24.  if (init(netif) != ERR_OK) {
25.    return NULL;
26.  }    // 初始化函数
27.
28.  /* add this netif to the list */
29.  netif->next = netif_list;    // 加入链表
30.  netif_list = netif;    //
31.  snmp_inc_iflist();
32.
33.  return netif;
34.}

```

从它的源码中可以看出这个函数就完成了对 netif 结构体的赋值，在这里有一个问题，调用 netif_add() 函数时（见 LwIP_Init() 中的调用），输入的参数 ethernetif_init 和 ethernet_input 分别是网卡初始化函数和输入函数的指针，这两个指针指向的函数具体是如何定义的呢？

24.4.6 LwIP 对底层操作的封装

要解决上一节的问题，需要了解 LwIP 对底层操作的封装。对这些操作封装后给上层提供的接口呈现在 LwIP 协议栈中就是上述的函数指针形式，LwIP 协议栈内部的函数通过这些指针来调用函数。这些接口都保存在 \lwip-1.3.2\src\netif 文件夹下的 ethernetif.c 文件中。它已被添加至本实验工程的 lwip-netif 组中。

1. ethernetif_init 指针

首先看看网卡初始化函数 `ethernetif_init()`，它并不是最底层的，该函数的具体定义都是由 LwIP 协议栈已经完全编写好。其定义见代码清单 24-6。

代码清单 24-6 `ethernetif_init()` 函数

```

1. err_t ethernetif_init( struct netif *netif )
2. {
3.     struct ethernetif *ethernetif;
4.
5.     ethernetif = mem_malloc( sizeof(struct ethernetif) ); // 分配内存
6.
7.     if( ethernetif == NULL )                               // debug 时使用的输出
8.     {
9.         LWIP_DEBUGF( NETIF_DEBUG, ("ethernetif_init: out of memory\n\r") );
10.        return ERR_MEM;
11.    }
12.
13.    netif->state = ethernetif;
14.    netif->name[0] = IFNAME0;                               // 网卡名字
15.    netif->name[1] = IFNAME1;
16.    netif->output = etharp_output;                           // IP 层输出函数
17.    netif->linkoutput = low_level_output;                   // 底层硬件输出函数
18.
19.    ethernetif->ethaddr = ( struct eth_addr * ) &( netif->hwaddr[0] );
                                                                // MAC 地址
20.
21.    low_level_init( netif );                                // 最底层初始化函数
22.
23.    return ERR_OK;
24.}

```

在上一节的 `netif_add()` 只是对 `netif` 部分的结构体成员进行了赋值，还有一些成员如 `state`、`name`、`output` 函数指针、`linkoutput` 函数指针及 `MAC` 地址成员，就是在本函数中进行赋值的，见第 13 ~ 21 行。通过这个函数，我们终于发现了最底层的输出函数指针 `low_level_output` 和最底层的初始化函数指针 `low_level_init`。

2. 底层输出函数

LwIP 协议栈的源文件只是提供了 `low_level_output` 函数的模型，该函数的具体实现见代码清单 24-7。

代码清单 24-7 `low_level_output()` 函数

```

1. /*
2.  * low_level_output(): Should do the actual transmission of the packet. The
3.  * packet is contained in the pbuf that is passed to the function. This pbuf
4.  * might be chained.
5.  */
6. static err_t low_level_output( struct netif *netif, struct pbuf *p ) /* 底层发送数

```

```

    据函数 */
7. {
8.     struct pbuf *q;
9.     int i = 0;
10.
11.     err_t xReturn = ERR_OK;
12.
13.     /* Parameter not used. */
14.     for(q = p; q != NULL; q = q->next)
15.     {
16.         // 复制 pbuf 内容
17.         memcpy(&Tx_Data_Buf[i], (u8_t*)q->payload, q->len);
18.         i = i + q->len;
19.     }
20.     enc28j60PacketSend(i, Tx_Data_Buf); // 发送数据包
21.
22.     return xReturn;
23. }

```

本函数看上去十分简单，在上层需要发送数据的时候，以 pbuf 作为输入参数调用 low_level_output() 函数。而 pbuf 链表的所有 payload 成员就是需要发送的 MAC 数据包，每个数据包的长度为 len 成员的值。函数中利用 for 循环调用 memcpy() 把 q->payload 链表的数据都取出来存放在 Tx_Data_Buf 数组中，最后调用 enc28j60PacketSend() 函数把 Tx_Data_Buf 数组发送出去。enc28j60PacketSend() 是在 ENC28J60.C 文件编写好的 enc28j60 驱动，它使用 SPI 接口控制 ENC28J60 芯片发送数据。

3. 底层初始化函数

最底层的网卡初始化函数 low_level_init() 的定义见代码清单 24-8。

代码清单 24-8 low_level_init() 函数

```

1. static void low_level_init( struct netif *netif )
2. {
3.     /* set MAC hardware address length */
4.     netif->hwaddr_len = 6;
5.
6.     /* set MAC hardware address */
7.     /* MAC 地址 */
8.     netif->hwaddr[0] = macaddress[0];
9.     netif->hwaddr[1] = macaddress[1];
10.    netif->hwaddr[2] = macaddress[2];
11.    netif->hwaddr[3] = macaddress[3];
12.    netif->hwaddr[4] = macaddress[4];
13.    netif->hwaddr[5] = macaddress[5];
14.
15.    /* maximum transfer unit */
16.    /* 最大传输单元 */
17.    netif->mtu = netifMTU;
18.
19.    /* broadcast capability */
20.    netif->flags =

```

```

21.NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP | NETIF_FLAG_LINK_UP;
22.
23.     enc28j60Init(netif->hwaddr);    // 初始化 enc28j60
24.     enc28j60clkout(1);              // change clkout from 6.25MHz to 12.5MHz
25.
26.}

```

本初始化函数中先对 netif 结构体的 MAC 地址成员 netif->hwaddr 赋值为 macaddress 数组的值。macaddress 数组是在 netconfig.c 文件中定义的，定义的时候已经设置了它的初始值，在本函数中复制到 netif->hwaddr 后作为网卡的 MAC 地址。本函数中对最大传输单元结构体成员 netif->mtu 赋值为 1500，对 netif->flags 赋值使其允许使用广播地址、使能 ARP 功能。函数的最后，像上面的 low_level_output 函数一样，调用了在 ENC28J60.C 文件中编写的驱动。分别是初始化函数 enc28j60Init() 和时钟输出函数 enc28j60clkout()。

4. ethernet_input 指针

分析完 netif_add() 输入参数 ethernetif_init 指针，再来看看另一个作为它的输入参数的函数指针 ethernet_input，它是由 LwIP 协议栈自带的完整函数，一般不需要我们修改，见代码清单 24-9。

代码清单 24-9 ethernet_input() 函数

```

1. /*
2.  * ethernetif_input(): This function should be called when a packet is ready to
3.  * be read from the interface. It uses the function low_level_input() that
4.  * should handle the actual reception of bytes from the network interface.
5.  */
6. err_t ethernetif_input(struct netif *netif)
7. {
8.     err_t err = ERR_OK;
9.     struct pbuf *p = NULL;
10.
11.     /* move received packet into a new pbuf */
12.     p = low_level_input(netif);
13.
14.     if (p == NULL) return ERR_MEM;
15.
16.     err = netif->input(p, netif);
17.     if (err != ERR_OK)
18.     {
19.         LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n"));
20.         pbuf_free(p);
21.         p = NULL;
22.     }
23.
24.     return err;
25.}

```

由于本实验中的 ENC28J60 驱动没有使用中断接收数据包，所以 ethernetif_input() 需要被 main 函数中 while 循环的 LwIP_Periodic_Handle() 函数不停地调用，通过 ethernetif_input() 函数的返回值来判断是否接收到数据包。本函数执行时，先调用了最底层的输入函数 low_level_input()，它在底层又调用了 ENC28J60 接收数据包的驱动。若接收到数据包，返回值赋值到 pbuf 类型指针

(pbuf 非空)，然后在本函数的第 16 行，把 pbuf 输入到 netif 结构体的 input 函数指针，通过 input 指针把数据包传递给上层的 LwIP 协议栈代码进行处理，完成数据包的收取。(读者也可以修改 ENC28J60 的驱动，使用中断方式接收数据包，就不需要采用轮询的方式了。)

5. 底层输入函数

上面的 ethernetif_input() 调用了最底层的输入函数 low_level_input()，我们需要在这个函数中实现 LwIP 接收数据包的功能，主要是通过调用 ENC28J60 来实现的。见代码清单 24-10。

代码清单 24-10 low_level_input() 函数

```

1.  /*
2.   * low_level_input(): Should allocate a pbuf and transfer the bytes of the
3.   * incoming packet from the interface into the pbuf.
4.   */
5.
6.  static struct pbuf *low_level_input( struct netif *netif )
7.  {
8.
9.      struct pbuf *q, *p = NULL;
10.     u16 Len = 0;
11.
12.     int i = 0;
13.
14.         // 接收数据包，并返回接收到的数据包长度
15.     Len = enc28j60PacketReceive(1520 * 4, Data_Buf);
16.
17.     if ( Len == 0 ) return 0;
18.
19.     p = pbuf_alloc(PBUF_RAW, Len, PBUF_POOL);
20.
21.     if (p != NULL)
22.     {
23.
24.         for (q = p; q != NULL; q = q->next)
25.         {
26.             memcpy((u8_t*)q->payload, (u8_t*)&Data_Buf[i], q->len);
27.
28.             i = i + q->len;
29.         }
30.         if( i != p->tot_len ){ return 0; } // 相等时表明到了数据尾
31.     }
32.
33.     return p;
34. }

```

在本函数的第 15 行，就是调用 ENC28J60.C 文件中的 ENC28J60 接收数据包的驱动，即 enc28j60PacketReceive() 函数，它把接收到的数据存放在它的输入参数 Data_Buf 数组中，并把该数据包的长度以返回值的形式赋值给变量 Len，若 Len 的长度非 0 时，即接收到数据包，就调用 LwIP 的内存管理函数 pbuf_alloc() 申请一个长度为 Len 的 pbuf 存储空间，在 for 循环中把数组 Data_Buf

的内容转移到 pbuf 的 q->payload 成员中，最后以返回值的形式把 pbuf 交给调用它的函数。

对于最底层 ENC28J60 驱动函数的具体实现，请读者利用源码进行分析，它们都是在 ENC28J60.C 文件中定义的。编写的思想就是利用 SPI 接口根据 datasheet 发送不同的命令，控制 ENC28J60 芯片，收发数据包时则往它的缓冲区读取或写入数据。

24.4.7 轮询和计时

确保底层驱动正确，并且把 LwIP 接口与驱动关联起来后，还需要在循环中轮询输入函数，给 LwIP 协议栈提供计时。这是在 main 函数中循环调用的 LwIP_Periodic_Handle() 函数实现的。调用它时，它的输入参数 LocalTime 变量由 SysTick 定时器每 10 ms 更新一次，每次加 10。其具体定义见代码清单 24-11。

代码清单 24-11 LwIP_Periodic_Handle() 函数

```

1. /*
2.  * 函数名: LwIP_Periodic_Handle
3.  * 描述  : lwip 协议栈要求周期调用一些函数
4.          tcp_tmr etharp_tmr dhcp_fine_tmr dhcp_coarse_tmr
5.  * 输入  : 无
6.  * 输出  : 无
7.  * 调用  : 外部调用
8.  */
9. void LwIP_Periodic_Handle(__IO uint32_t localtime)
10. {
11.     //err_t err;
12.
13.         /* 接收数据包 */
14.         ethernetif_input(&enc28j60);    // 轮询是否接收到数据
15.
16.
17.     /* TCP periodic process every 250 ms */
18.     if (localtime - TCPTimer >= TCP_TMR_INTERVAL)
19.     {
20.         TCPTimer = localtime;
21.         tcp_tmr();    // 每 250ms 调用一次
22.     }
23.     /* ARP periodic process every 5s */
24.     if (localtime - ARPTimer >= ARP_TMR_INTERVAL)
25.     {
26.         ARPTimer = localtime;
27.         etharp_tmr();    // 每 5s 调用一次
28.     }
29.
30. }
```

本函数在第 14 行调用了 ethernetif_input() 函数，用于查询是否接收到数据包，若接收到数据包则通过内部的机制把它传递给 LwIP 的上层。接下来检查当前 LocalTime 变量与上一次调用了 tcp_tmr() 或 etharp_tmr() 的差值，若时间差达到 250 ms 或 5 s 时，则调用一次 tcp_tmr() 或 etharp_tmr() 一

次。这两个函数分别用于 TCP 协议层和 ARP 模块，这些模块涉及一些超时重传或其他与时间有关的操作。

24.4.8 opt.h 文件和 debug

1. 裁剪和配置 LwIP

正确完成了 LwIP 与底层的接口，轮询也配置完毕，还需要配置 LwIP 的 opt.h 文件，它如同 FATFS 文件系统中的 ffconf.h 文件。opt.h 用于裁剪、配置 LwIP 协议栈，如是否使用操作系统、上层的应用是使用 API 还是 RAW 函数等。部分关于 TCP 配置的一些宏见代码清单 24-12。

代码清单 24-12 TCP 配置宏

```

1.  /*
2.  -----
3.  ----- TCP options -----
4.  -----
5.  */
6.  /**
7.   * LWIP_TCP==1: Turn on TCP.
8.   */
9.  #ifndef LWIP_TCP
10. #define LWIP_TCP                1
11. #endif
12.
13. /**
14.   * TCP_TTL: Default Time-To-Live value.
15.   */
16. #ifndef TCP_TTL
17. #define TCP_TTL                  (IP_DEFAULT_TTL)
18. #endif
19.
20. /**
21.   * TCP_WND: The size of a TCP window. This must be at least
22.   * (2 * TCP_MSS) for things to work well
23.   */
24. #ifndef TCP_WND
25. #define TCP_WND                  (4 * TCP_MSS)
26. #endif

```

由于 opt.h 文件中这些宏已经配置了默认值，所以大部分是不需要修改的。又由于这里的每个宏都包含条件编译的设置，所以我们在移植 LwIP 协议栈时，通常新建一个名为 lwipopts.h 的头文件，在该文件中修改需要的配置。

如上面的 LWIP_TCP 宏用于使能 TCP 协议或关闭 TCP 协议，它的默认值为 1，表示使能。若我们不需要使用 LwIP 的 TCP 协议时，我们就在 lwipopts.h 文件中添加这个宏，把 LWIP_TCP 宏定义为 0。

由于 opt.h 文件中的 LWIP_TCP 宏是在条件编译“#ifndef”中，若在其他地方定义了相同的

宏，如 lwipopts.h 文件中定义了 LWIP_TCP 宏，则该宏的参数以 lwipopts.h 文件的定义为准，若在其他地方没有定义相同名字的宏，则以 opt.h 文件的为准。在本工程实验的 lwipopts.h 文件中对 LwIP 配置见代码清单 24-13。

代码清单 24-13 lwipopts.h 文件配置

```

1. /*****
2. **-----Socket 参数配置: Socket options-----
3. *****/
4. #define LWIP_SOCKET 0
5. // #define LWIP_COMPAT_SOCKETS 0
6. #define NO_SYS 1 // 不使用操作系统
7.
8. /*****
9. -----Sequential layer options-----
10. *****/
11. #define LWIP_NETCONN 0
12.
13. #define NO_SYS_NO_TIMERS 1
14.
15. #define LWIP_DHCP 0
16.
17. /*****
18. #define ETHARP_TMR_INTERVAL 5000 /* Time in milliseconds to perform
    ARP processing */
19.
20. /*****
21.
22. /* ----- Memory options ----- */
23. /* MEM_ALIGNMENT: should be set to the alignment of the CPU for which
24.  lwIP is compiled. 4 byte alignment -> define MEM_ALIGNMENT to 4, 2
25.  byte alignment -> define MEM_ALIGNMENT to 2. */
26.
27. #define MEM_ALIGNMENT 4
28. /* Align memory on 4 byte boundery (32-bit) */
29.
30. /* MEM_SIZE: the size of the heap memory. If the application will send
31. a lot of data that needs to be copied, this should be set high. */
32.
33. #define MEM_SIZE (4*1024)
34.
35. /* MEMP_NUM_PBUF: the number of memp struct pbufs. If the application
36.  sends a lot of data out of ROM (or other static memory), this
37.  should be set high. */
38.
39. #define MEMP_NUM_PBUF 10
40.
41. /* MEMP_NUM_UDP_PCB: the number of UDP protocol control blocks. One
42.  per active UDP "connection". */
43.
44. #define MEMP_NUM_UDP_PCB 6

```



```

45.
46./* MEMP_NUM_TCP_PCB: the number of simulatenously active TCP
47.   connections. */
48.
49.#define MEMP_NUM_TCP_PCB          10
50./* MEMP_NUM_TCP_PCB_LISTEN: the number of listening TCP
51.   connections. */
52.
53.#define MEMP_NUM_TCP_PCB_LISTEN  6
54.
55./* MEMP_NUM_TCP_SEG: the number of simultaneously queued TCP
56.   segments. */
57.
58.#define MEMP_NUM_TCP_SEG          12
59.
60./* MEMP_NUM_SYS_TIMEOUT: the number of simulateously active
61.   timeouts. */
62.
63.#define MEMP_NUM_SYS_TIMEOUT      3
64.
65.
66./* ----- Pbuf options ----- */
67./* PBUF_POOL_SIZE: the number of buffers in the pbuf pool. */
68.
69.#define PBUF_POOL_SIZE            10
70.
71./* PBUF_POOL_BUFSIZE: the size of each pbuf in the pbuf pool. */
72.
73.#define PBUF_POOL_BUFSIZE         1500
74./******
75.**-----TCP 参数配置: TCP options-----
76.*****
77.
78. #define MEMP_NUM_REASSDATA          5
79./* ----- TCP options ----- */
80.#define LWIP_TCP                    1
81.#define TCP_TTL                     255
82.
83./* Controls if TCP should queue segments that arrive out of
84.   order. Define to 0 if your device is low on memory. */
85.
86.#define TCP_QUEUE_OOSEQ            0
87.
88./* TCP Maximum segment size. */
89.
90.#define TCP_MSS                     (1500 - 40) /* TCP_MSS = (Ethernet MTU - IP header
   size - TCP header size) */
91.
92./* TCP sender buffer space (bytes). */
93.
94.#define TCP_SND_BUF                 (2*TCP_MSS)
95.

```

```

96./* TCP sender buffer space (pbufs). This must be at least = 2 *
97.   TCP_SND_BUF/TCP_MSS for things to work. */
98.
99.#define TCP_SND_QUEUELEN      (6 * TCP_SND_BUF)/TCP_MSS
100.
101.   /* TCP receive window. */
102.
103.   #define TCP_WND              (2*TCP_MSS)
104.
105.
106.   /* ----- ICMP options ----- */
107.   #define LWIP_ICMP            1
108.

```

完成了这部分后（不需要例程 main 中的 `httpd_init()` 和 `CMD_init()` 函数），下载程序，接上网线就可以 ping 通了，表示 LwIP 移植初步成功。但由于没有建立服务器和 Telnet 响应的程序，还不能访问服务器和使用 Telnet 的远程登录功能。

2. LwIP 的 debug 功能

大部分初学者不熟悉 TCP/IP 协议，移植 LwIP 十分不容易，而 LwIP 协议栈有一个十分贴心的输出调试信息的功能。

如在前面提到的 `ethernet_input()` 函数的第 19 行有一句宏：

```
1. LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n"));
```

如果我们设置了输出调试信息使能，若 `ethernet_input()` 中出现错误时就会输出调试信息。这样类似的宏由 LwIP 的设计者精心安排在各种容易出错的场合，以便于移植和调试。宏 `LWIP_DEBUGF` 是在 LwIP 的 `debug.h` 文件中定义的，见代码清单 24-14。

代码清单 24-14 LWIP_DEBUGF 宏

```

1. #define LWIP_DEBUGF(debug, message) do { \
2.     if ( \
3.         ((debug) & LWIP_DBG_ON) && \
4.         ((debug) & LWIP_DBG_TYPES_ON) && \
5.         ((s16_t)((debug) & LWIP_DBG_MASK_LEVEL) >= LWIP_
   DBG_MIN_LEVEL)) { \
6.         LWIP_PLATFORM_DIAG(message); \
7.         if ((debug) & LWIP_DBG_HALT) { \
8.             while(1); \
9.         } \
10.    } \
11.    } while(0)

```

这个宏看似十分复杂，其功能就是当符合某些条件的时候，调用第 6 行中的宏 `LWIP_PLATFORM_DIAG` 输出信息。具体如何输出信息需要我们去实现，本工程在 `cc.h` 文件对该宏设置如下：

```
1. #define LWIP_PLATFORM_DIAG(x) do {printf x;} while(0)
```

即调用 printf() 函数来输出这些调试信息，而 printf() 函数已经被我们重定义到串口，所以 PC 的终端就可以接收调试信息了。

在默认的情况下是不输出调试信息的，详见 opt.h 文件相关的 debug 宏设置，可以配置不输出调试信息或输出部分调试信息。本实验工程已调试完毕，所以我们把大部分输出调试信息的功能都关了。本例程在 lwipopts.h 文件中设置的与调试信息相关的宏，见代码清单 24-15。

代码清单 24-15 与调试信息相关的宏

```

1.  /*****
2.  #define LWIP_DEBUG                      1
3.  #define LWIP_DBG_TYPES_ON              LWIP_DBG_OFF
4.
5.  /**
6.   * ETHARP_DEBUG: Enable debugging in etharp.c.
7.   */
8.  #define ETHARP_DEBUG                    LWIP_DBG_OFF
9.
10. /**
11.  * NETIF_DEBUG: Enable debugging in netif.c.
12.  */
13. #define NETIF_DEBUG                      LWIP_DBG_OFF
14.
15. /**
16.  * PBUF_DEBUG: Enable debugging in pbuf.c.
17.  */
18. #define PBUF_DEBUG                       LWIP_DBG_OFF
19.
```

若需要输出调试信息，必须把宏 LWIP_DEBUG 设置为 1，其他的如 NETIF_DEBUG、ETHARP_DEBUG 可根据调试的情况决定是否开启。

24.4.9 LwIP 应用

1. 网络应用软件架构

构成网络应用的软件有不同的结构，有 B/S 结构（浏览器 / 服务器）和 C/S 结构（客户端 / 服务器）。

基于 B/S 结构的应用程序以网页形式存放于服务器上，用户运行应用程序时通过 Web 调用服务器的应用程序，并通过浏览器把结果显示给用户，用户只需要浏览器即可运行所有服务器中提供的应用模块。

而基于 C/S 结构的软件需要针对不同的应用对客户端进行更改。本实验工程中的 HTTP 服务器是基于 B/S 结构的，而 Telnet 程序是基于 C/S 结构的。

2. TCP 网络应用

基于 TCP 协议的网络应用十分常见。TCP 应用的服务器流程如图 24-13，例如要把 STM32 作为网页服务器，其程序就是根据该流程编写的。

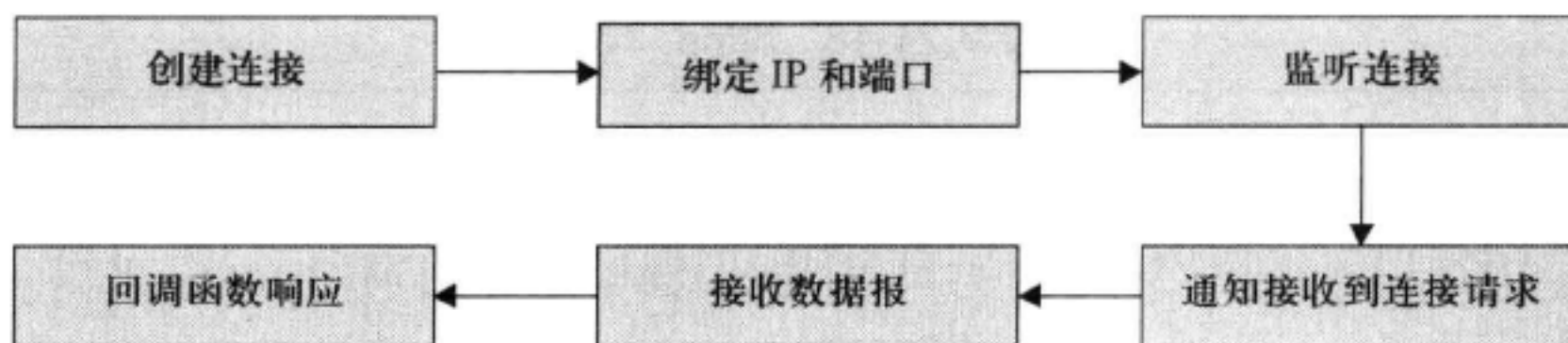


图 24-13 建立 TCP 应用流程

当服务器开始监听连接时，客户端或浏览器就可以向服务器提出连接请求，然后服务器做出响应。

3. LwIP 的应用函数

移植好 LwIP 后，就可以利用它来编写应用层的程序了。LwIP 协议栈支持类似 BSD 协议栈的 Socket 应用程序接口，这种被称为 LwIP 的 API 函数，API 函数更适合用在具有操作系统的平台上。

另一种方式是直接调用 LwIP 协议栈内部的函数，这些函数被称为 RAW 函数，RAW 函数适合用在没有操作系统的平台，效率高，但开发起来较复杂。由于本例程中没有使用操作系统，所以我们利用 LwIP 的 RAW 函数进行应用层编程。

利用 LwIP 编写的 TCP 类型应用程序，它在各个层次的处理过程见图 24-14。

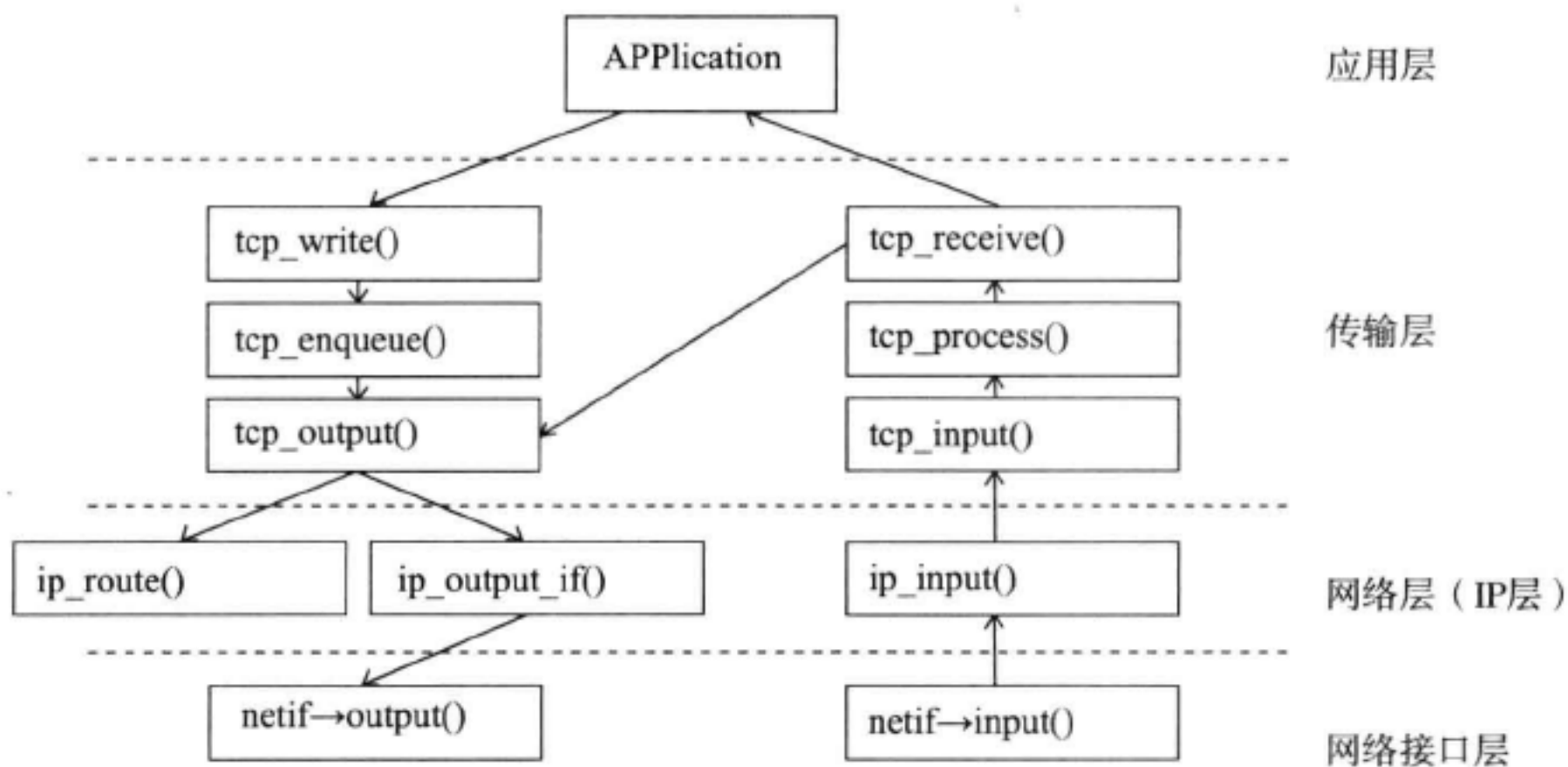


图 24-14 TCP 处理过程

在移植的时候，我们主要完成的就是图 24-14 中网络接口层的 `netif->output` 和 `netif->input` 函数。移植完成后，即可利用 LwIP 的应用层接口编写应用程序，而网络层和传输层则由 LwIP 协议栈完成。不过，大部分 RAW 函数都是位于传输层的。

24.4.10 网页服务器

利用 LwIP 编写网页服务器是由 `main` 函数中调用的 `httpd_init()` 函数实现的。该函数的源码见代码清单 24-16。

代码清单 24-16 httpd_init() 函数

```

1.  /*
2.  * 函数名: httpd_init
3.  * 描述   : 初始化 web server, 初始化后才能显示网页
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8.  void httpd_init(void)
9.  {
10.   struct tcp_pcb *pcb;
11.
12.   pcb = tcp_new();           /* 建立通信的 TCP 控制块 (pcb) */
13.
14.   tcp_bind(pcb, IP_ADDR_ANY, 80); /* 绑定本地 IP 地址和端口号 */
15.
16.   pcb = tcp_listen(pcb);     /* 进入监听状态 */
17.
18.   tcp_accept(pcb, http_accept); /* 设置有连接请求时的回调函数 */
19.
20. }
```

httpd_init() 函数只有几行, 它每调用一个函数, 就完成了 TCP 服务器流程图中的一个步骤。分析如下:

(1) tcp_new()

本函数的原型为:

```
1. struct tcp_pcb * tcp_new(void)
```

它用于创建一个 TCP 连接。它向 LwIP 协议栈申请分配一个 TCP 控制块, 若分配成功, 则返回控制块的指针, 否则返回 NULL。TCP 控制块结构体用 tcp_pcb 进行定义, 它是用于描述 TCP 连接的, 包含了非常丰富的信息。

本实验中调用 tcp_new() 后, 把它的返回值赋给 TCP 控制块类型的指针 pcb。

(2) tcp_bind()

本函数的原型为:

```
1. err_t
2. tcp_bind(struct tcp_pcb *pcb, struct ip_addr *ipaddr, u16_t port)
```

它用于把一个本地的 IP 和端口进行绑定, 它的输入参数分别为要进行绑定的连接、绑定的 IP 地址、端口号。

本实验中把 tcp_new() 新建的连接 pcb 进行绑定, 绑定的 IP 地址为 IP_ADDR_ANY, 这个宏展开为 0x00000000; 绑定的端口为 80, 表示把本地的任意 IP 地址都与 80 端口绑定。网页浏览器默认使用 80 端口。

(3) tcp_listen()

它实际是一个宏，宏展开是一个函数：

```
1.  #define    tcp_listen(pcb)    tcp_listen_with_backlog(pcb, TCP_DEFAULT_LISTEN_
    BACKLOG)
```

函数的原型如下：

```
1.  struct tcp_pcb *
2.  tcp_listen_with_backlog(struct tcp_pcb *pcb, u8_t backlog)
```

它用于监听端口，若接收到连接，则会调用 tcp_accept() 函数中指定的回调函数进行响应，返回值为新的 TCP 控制块，用于回调函数的输入参数。

本实验调用 tcp_listen() 对以上创建的 80 端口进行监听。

(4) tcp_accept()

函数的原型为：

```
1.  void
2.  tcp_accept(struct tcp_pcb *pcb, err_t (* accept)(void *arg, struct tcp_
    pcb *newpcb, err_t err))
```

它的作用是用于指定连接建立后调用的回调函数。第一个输入参数为 TCP 控制块，用于指定是哪个连接；第二个参数是回调函数的指针。

本实验中调用 tcp_accept() 指定回调函数为 http_accept。

当我们在 PC 浏览器的地址栏输入：http://192.168.1.18:80 (:80 表示 80 端口，可省略)，由于服务器 STM32 的监听设置，连接成功，就会调用 tcp_accept() 指定的 http_accept 函数。http_accept() 函数定义见代码清单 24-17。

代码清单 24-17 http_accept() 函数

```
1.  /*
2.   * 函数名：http_accept
3.   * 描述   ：http web server 的回调函数，建立连接后调用
4.   * 输入   ：tcp_arg 设置的参数、pcb、err
5.   * 输出   ：err
6.   * 调用   ：内部调用
7.   */
8.  static err_t http_accept(void *arg, struct tcp_pcb *pcb, err_t err)
9.  {
10. /* 设置回调函数优先级，当存在几个连接时特别重要，此函数必须调用 */
11.  tcp_setprio(pcb, TCP_PRIO_MIN);
12.
13.  tcp_recv(pcb, http_recv);          /* 设置 TCP 段到时的回调函数 */
14.
15.  return ERR_OK;
16. }
```

```

44. // 提取用户名。sizeof 字符串包括终止符 '\0'
45.         PassWord += sizeof("PassWord=") - 1;
46. // 提取密码
47.
48.         if (strcmp(UserName, "wildfire") == 0 && strncmp
(PassWord, "123456", 6) == 0) /* 输入的用户名和密码正确 */
49.         {
50.             LED1(ON);
51. // printf("\r\n 提取出的用户名 =%s\r\n 提取出的密 =%s", UserName, PassWord);
52.
53. tcp_write(pcb, http_html_hdr, sizeof(http_html_hdr), 0); /* 发送 http 协议头部信息 */
54. tcp_write(pcb, led_ctrl_on, sizeof(led_ctrl_on), 0); /* 发送 led 控制网页信息 */
55.
56.         }
57.         else /* 输入的用户名和密码错误 */
58.         {
59.             tcp_write(pcb, http_html_hdr, sizeof(http_html_hdr), 0); /* 发送 http 协议头部信
息 */
60.             tcp_write(pcb, login, sizeof(login), 0); /* 发送登录网页信息 */
61.         }
62.         }
63.         else if (LED_CMD != NULL) /* 接收到 LED 控制命令 */
64.         {
65.             if (strstr(LED_CMD, "LED_CTRL=ON")) /* 检测是哪个命令：开 \ 关 */
66.             {
67.                 LED1(ON);
68.                 tcp_write(pcb, http_html_hdr, sizeof(http_html_hdr), 0); /* 发送 http 协
议头部信息 */
69.                 tcp_write(pcb, led_ctrl_on, sizeof(led_ctrl_on), 0); /* 发送 led 控制网
页信息 */
70.             }
71.             else if (strstr(LED_CMD, "LED_CTRL=OFF"))
72.             {
73.                 LED1(OFF);
74.                 tcp_write(pcb, http_html_hdr, sizeof(http_html_hdr), 0); /* 发送 http 协议头
部信息 */
75.                 tcp_write(pcb, led_ctrl_off, sizeof(led_ctrl_off), 0); /* 发送 led 控制网页
信息 */
76.             }
77.         }
78.     }
79.
80.     pbuf_free(p); /* 释放该 pbuf 段 */
81. }
82. tcp_close(pcb); /* 关闭这个连接 */
83. err = ERR_OK;
84.
85. return err;
86. }

```

该函数看似十分复杂，其实只是重复判断接收到什么数据，然后根据数据的内容调用 `tcp_write()` 函数发送不同的网页。其流程如下：

1) 第 17 行，把接收到的数据包 `p->payload` 的指针赋值给 `data`，在后面使用 `data` 来判断接收到的数据。

2) 第 19 行，若数据包非空，则开始判断数据包的内容。

3) 当我们在浏览器输入服务器的 IP 地址并访问时，浏览器会向服务器发送一些数据，我们可以使用一些网络抓包工具查看这些数据包。其数据内容见图 24-15 中的箭头所指的区域。数据包中开头的内容为“GET”。在服务器中我们以这三个字符作为第一次与服务器连接的标志，当确定为第一次连接时，调用 LwIP 的 `tcp_write()` 函数输出网页。

4) 第 25 行，调用 `tcp_write()` 函数，输出参数 `http_html_hdr` 数组的内容，这部分内容的大小为 `sizeof(http_html_hdr)`。`http_html_hdr` 数组是 http 网页的文件头部信息，在输出网页的时候我们需要先发送这部分内容。见代码清单 24-19。

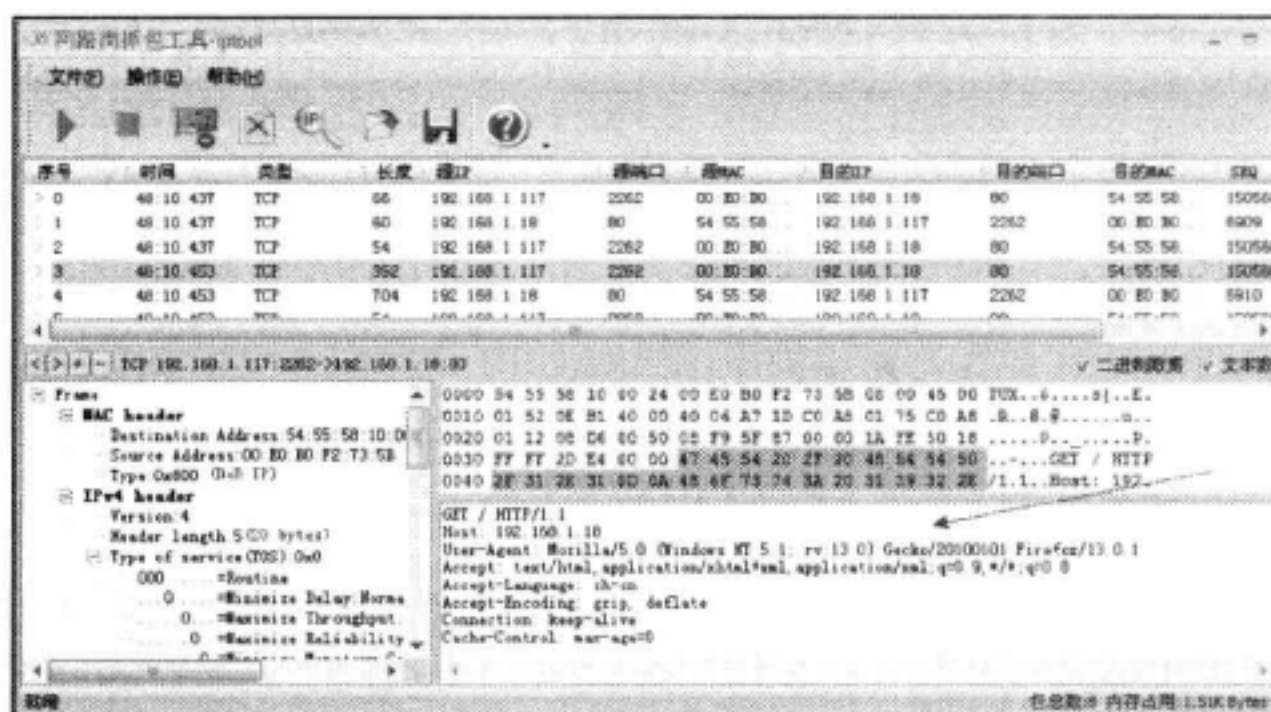


图 24-15 浏览器发送的数据

代码清单 24-19 http 文件头

```
/*http 文件头 */
#define
HTTP_HEAD "HTTP/1.1 200 OK\r\nContent-type: text/html\r\n\r\n"
```

5) 第 26 行，调用 `tcp_write()` 函数，输出 `login` 数组内容，`login` 数组存储的就是登录页面。由于没有使用文件系统，所以在本实验中 HTML 格式的网页文件都是以 C 语言的数组存储起来的。

6) 登录页面后的代码作用是类似的，只是浏览器向服务器发送的数据不同，而服务器响应的网页也有所区别而已。服务器接收到特定的 LED 控制的信息后，STM32 对 LED 进行控制。建议读者了解一下 HTML 语言，本实验中重点用到的是 HTML 语言中用于提交表单的 GET 方法和 POST 方法。

经过以上步骤，我们就成功建立了一个网页服务器，实验中还有一个针对 Telnet 程序的服务，它是在 `CMD_init()` 函数中完成的，编写的思路和搭建网页服务器十分相似。

24.4.11 实验现象

将配套 STM32 开发板供电 (DC5V)，插上 J-LINK，插上串口线（两头都是母的交叉线），利用网线把 STM32 开发板接入与 PC 相同的路由，也可以直接利用网线把开发板和 PC 相连，其

实验的操作是相同的（这样可以排除路由的问题），但在进行浏览网页实验时，图片可能无法正常显示。把本工程文件编译后烧录到开发板上，在程序运行框输入 cmd 命令进入 DOS 模式。

1) ping 实验，见图 24-16。在命令提示符窗口输入命令并回车：ping 192.168.1.18。

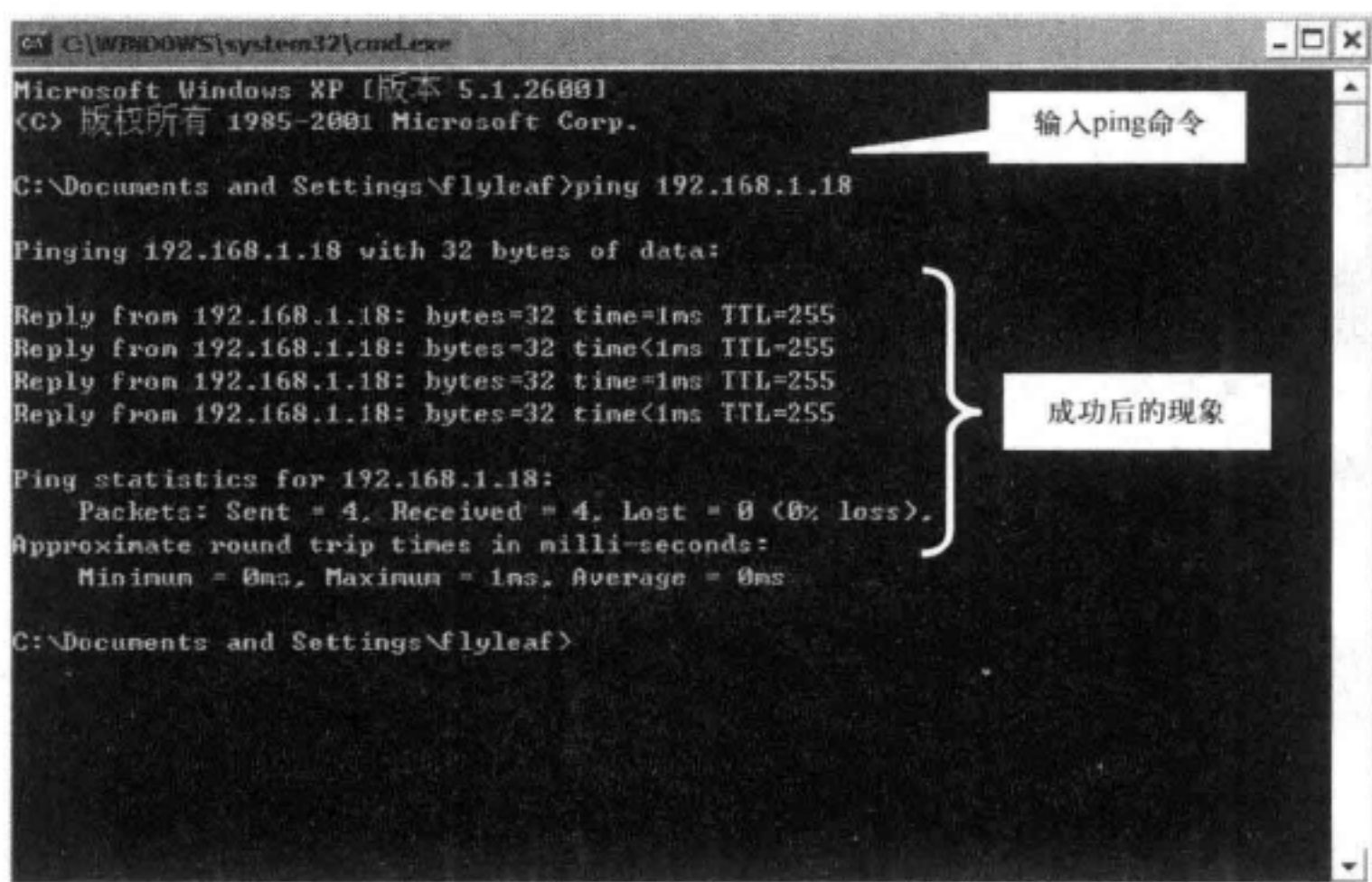


图 24-16 ping 192.168.1.18

2) Telnet 实验。

① 如果使用 Windows 7 系统，系统没有 Telnet 程序，需要自行下载安装。使用 Windows XP 系统的用户，在命令提示符窗口输入命令并回车：telnet 192.168.1.18。输入命令后弹出窗口如图 24-17 所示。

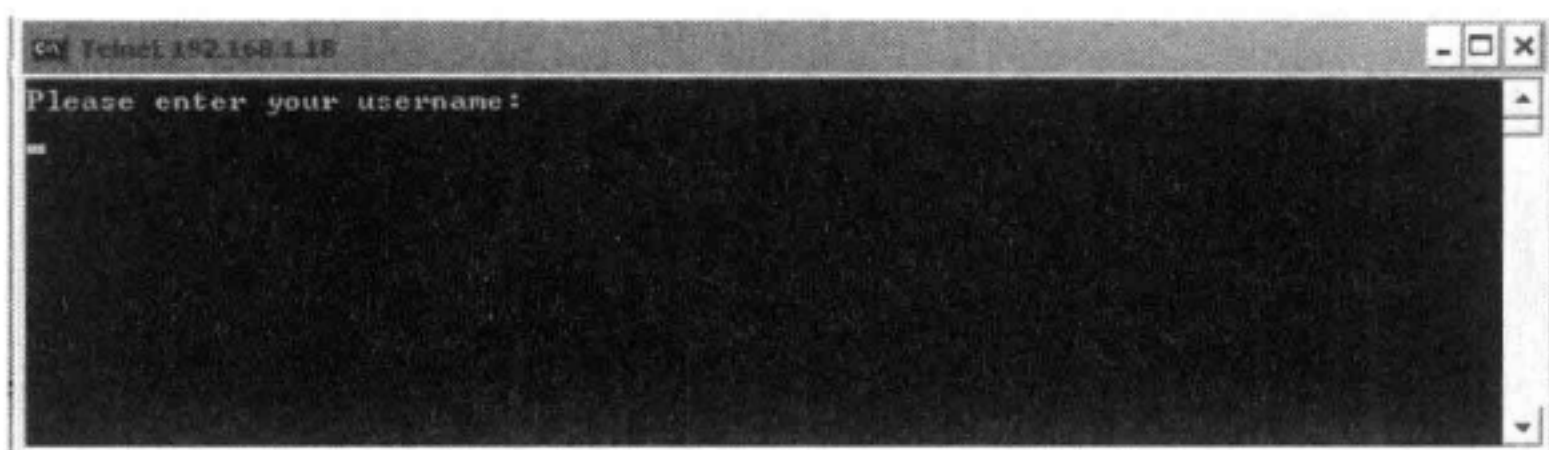


图 24-17 进入 Telnet 程序

② 见图 24-18，在弹出的窗口下输入用户名并回车：wildfire。

③ 若用户名正确，程序提示输入密码，键入密码并回车：123456。

④ 若密码正确，提示输入命令，本工程只允许两条命令，分别为 LED1_ON 和 LED1_OFF，用于控制 LED1 的亮和灭。输入命令：LED1_ON，板上的 LED1 灯会被点亮，窗口会弹出控制成功的信息，并且提示输入命令。输入命令：LED1_OFF，板上的 LED1 灯会被关灭，窗口弹出控制成功信息，再次提示输入命令。

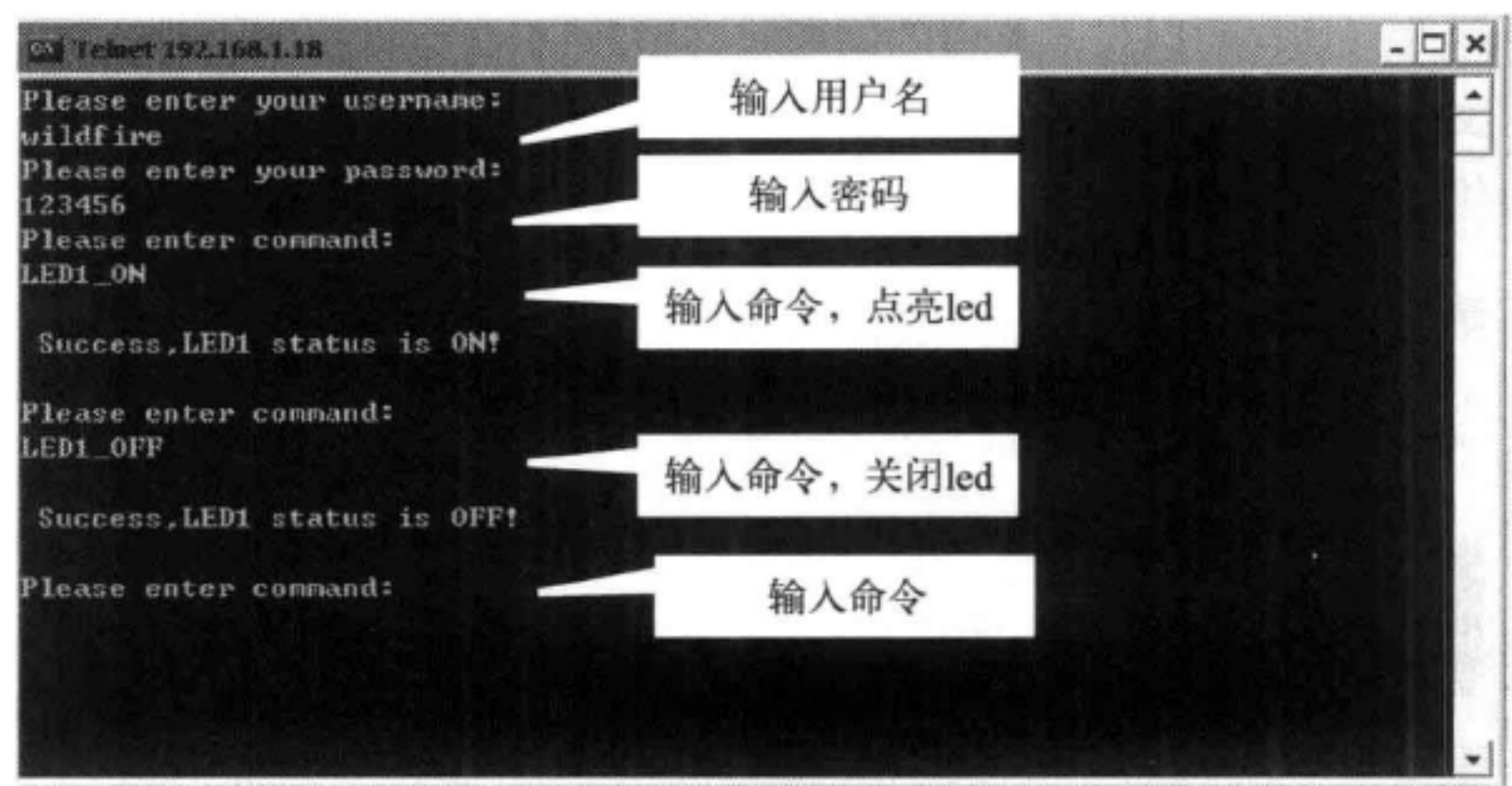


图 24-18 Telnet 控制流程

若用户输入的用户名、密码不正确或输入不存在的命令，会出现各种提示，并可以重新输入。
3) 网页浏览实验。(若 PC 没有接入互联网，图片可能无法正常显示。)

① 打开浏览器，在地址栏输入 IP 并回车：192.168.1.18，在弹出的网页中输入用户名和密码：wildfire、123456，见图 24-19。



图 24-19 网页登录

② 点击登录后，出现如图 24-20 所示界面，且开发板上的 LED 灯被点亮。

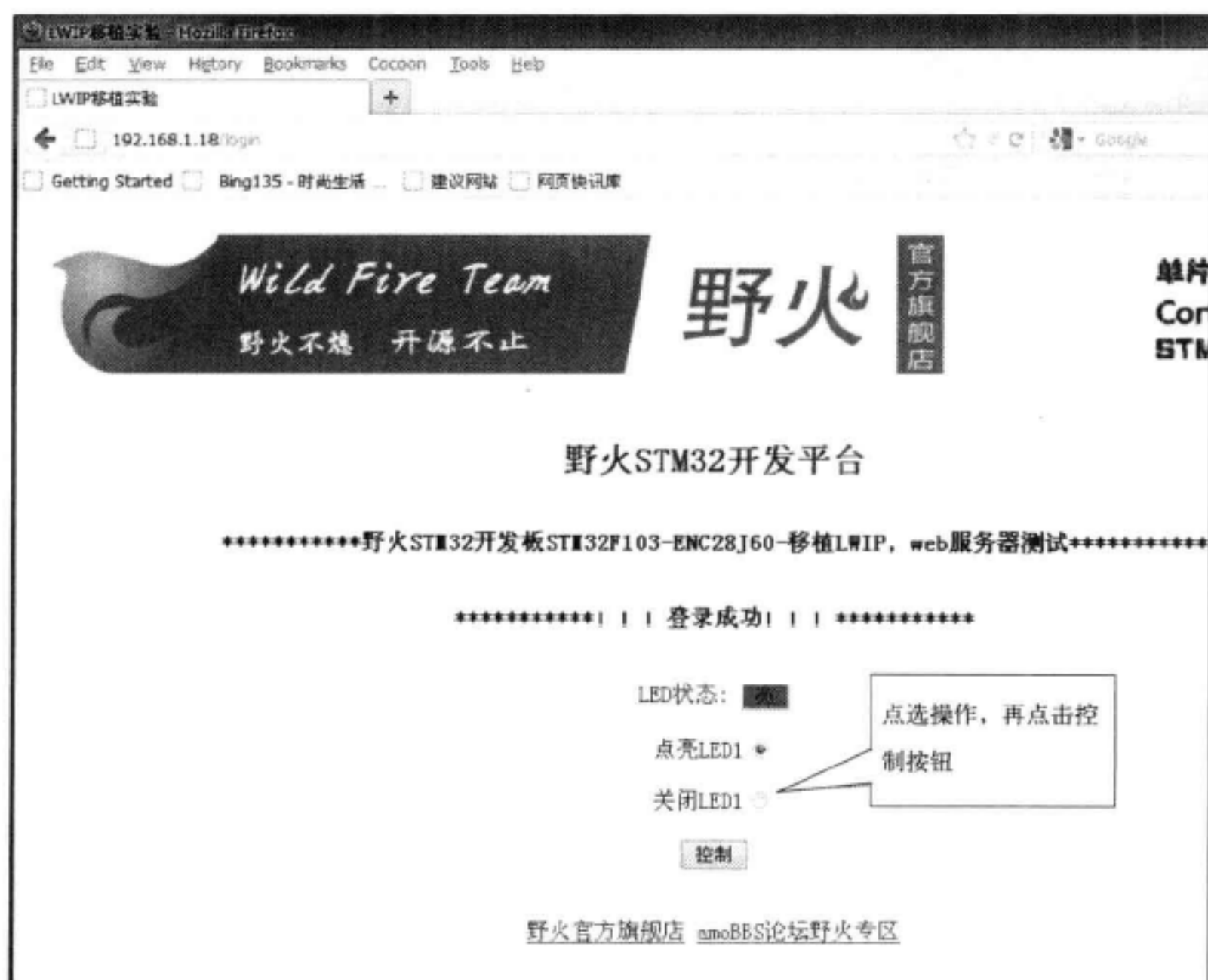


图 24-20 登录后的页面

③ 点选“关闭 LED1”，并点击“控制”按钮，网页显示的 LED 状态改变，板上的 LED 灯也被关灭。见图 24-21。



图 24-21 关闭 LED



第 25 章

Wi-Fi 模块 EMW3180 驱动

现在很多设备需要无线接入网络，这就需要使用 Wi-Fi 了。本章介绍一个 STM32 利用 Wi-Fi 模块 EMW3180 连接互联网的例子，本模块使用串口方式与主机通信，具有利用简单的串口进行透传的功能。

25.1 资料与工具下载

为了更好地理解和使用 Wi-Fi 模块的各项功能，首先需要学习以下资料，这些文档资料需要通过 Internet 下载或从附带光盘资料中找到：

- ❑ AN0003_EMW_DataTransferExample.pdf：透明传输模块使用范例，详细描述了模块在各种模式下透明传输的使用方法。
- ❑ EMW3X80_RM01050272：EMW 模块使用说明，详细描述了模块的各项功能。
- ❑ RM0002_EMWToolBox：EMW 模块配置软件使用说明，详细描述了如何配置模块的各项参数。
- ❑ EMW Tool Box：PC 端配置 Wi-Fi 模块参数工具软件。
- ❑ TCP/UDP 测试工具：用于在 PC 端与 Wi-Fi 模块建立 TCP/UDP 连接（使用串口调试助手收发数据）。

25.2 EMW3180 简介

EMW3180 是上海庆科公司开发的高速率串口 Wi-Fi 透传模块，集成了 TCP/IP 协议栈和 Wi-Fi 驱动，用户可轻松实现串口设备的无线网络功能，可用于楼宇自动化、智能家电、医疗和个人保健系统、汽车电子等领域。其实物见图 25-1，原理见图 25-2。

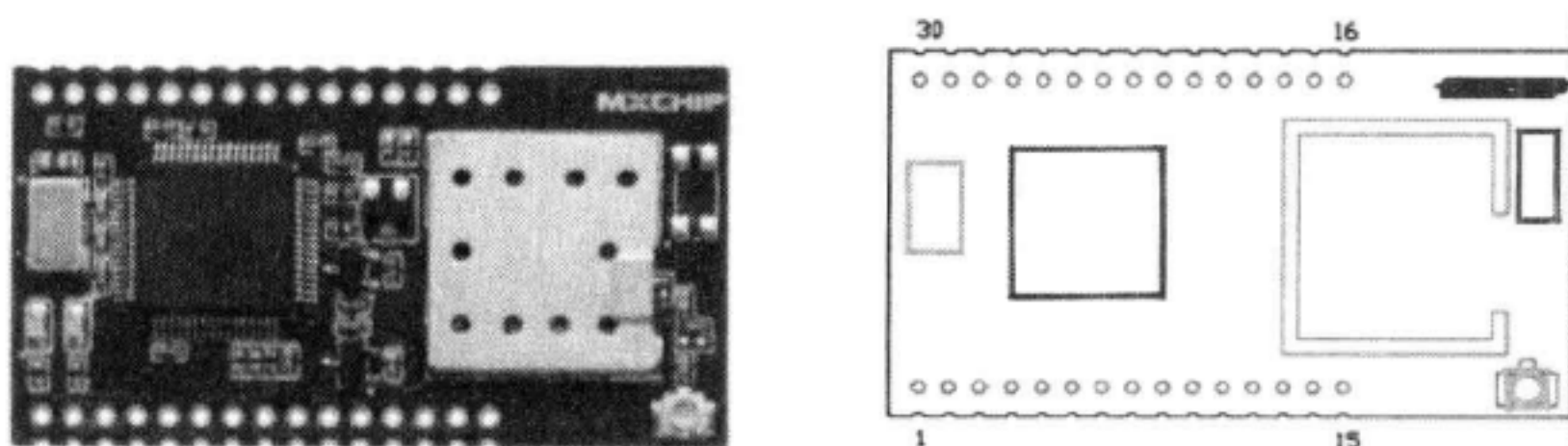


图 25-1 EMW3180 模块实物及引脚图

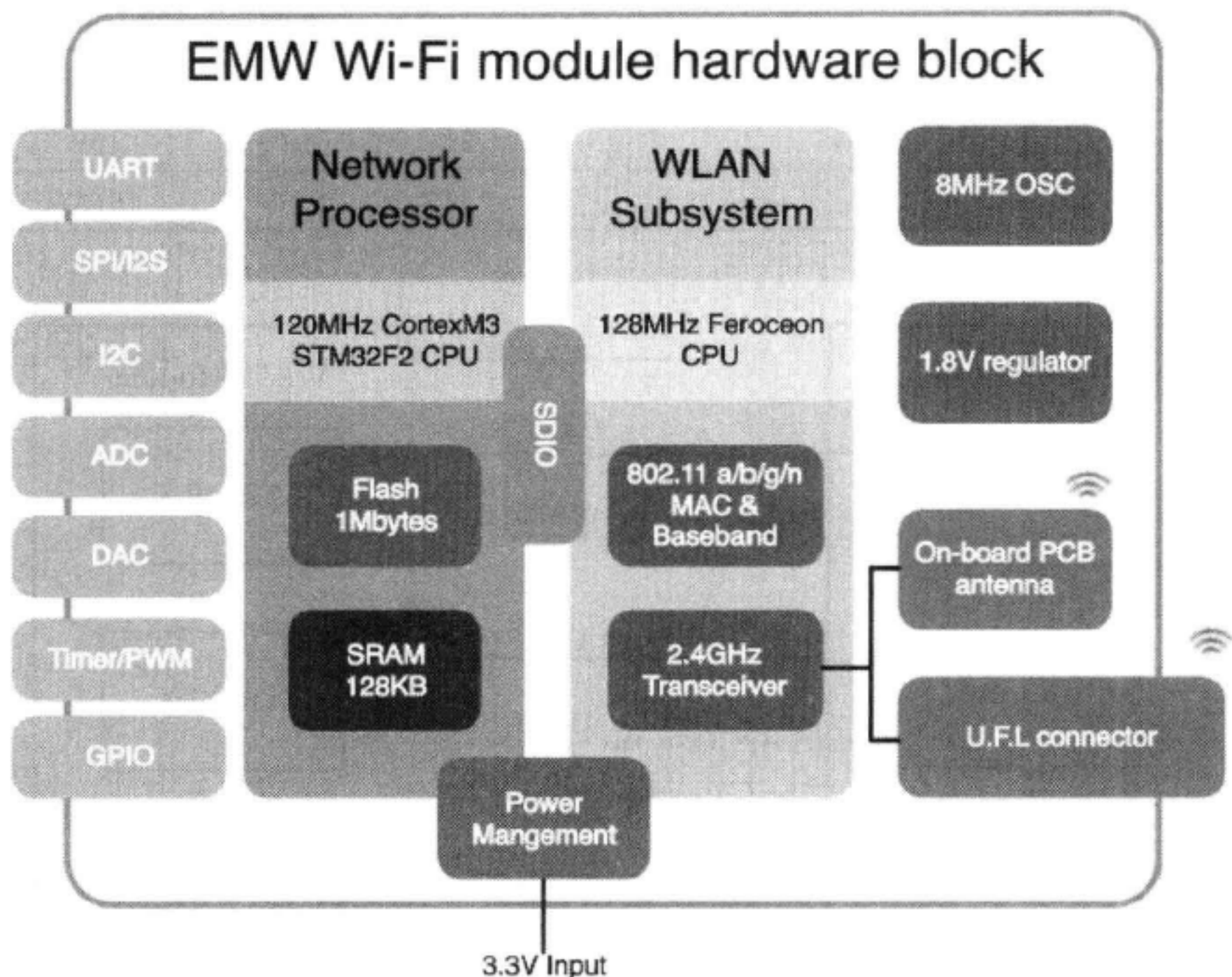


图 25-2 EMW3280 模块原理框图（与 EMW3180 的原理图类似）

从原理图可以看到，该模块是由 STM32 芯片和一个使用 SDIO 接口的 Wi-Fi 芯片构成。Wi-Fi 的驱动被保存在这个 STM32 芯片的 Flash 中，Wi-Fi 芯片通信收发的过程及数据都由板上的 STM32 芯片处理，外部设备则通过原理框图左侧中的 UART 接口即可控制各收发网络数据。所以大部分数据过程都由模块中的 STM32 芯片完成，外部设备不需要编写复杂的 Wi-Fi 驱动和 TCP/IP 协议栈，只要调用厂家提供的 API 就可以完成 Wi-Fi 通信的任务。

将 EMW3180 模块与 EMW3280 模块相比，EMW3180 使用的是 STM32F1xx 芯片，且没有引出 ADC、SPI 等引脚，只有 UART 接口。其引脚定义见表 25-1，硬件连接图见图 25-3。

表 25-1 EMW3180 模块引脚定义

引 脚 号	名 称	引 脚 号	名 称
1-5, 7-14	NC	19	NC
6	NC	20	nUART_RTS
15	GND	21	nUART_CTS
16	nWi-Fi LED BOOT	22	UART_TXD
17	nRESET	23	UART_RXD
18	IO1	24	VDD

(续)

引脚号	名称	引脚号	名称
25	GND	29	nWAKE_UP
26 ~ 28	NC	30	STATUS

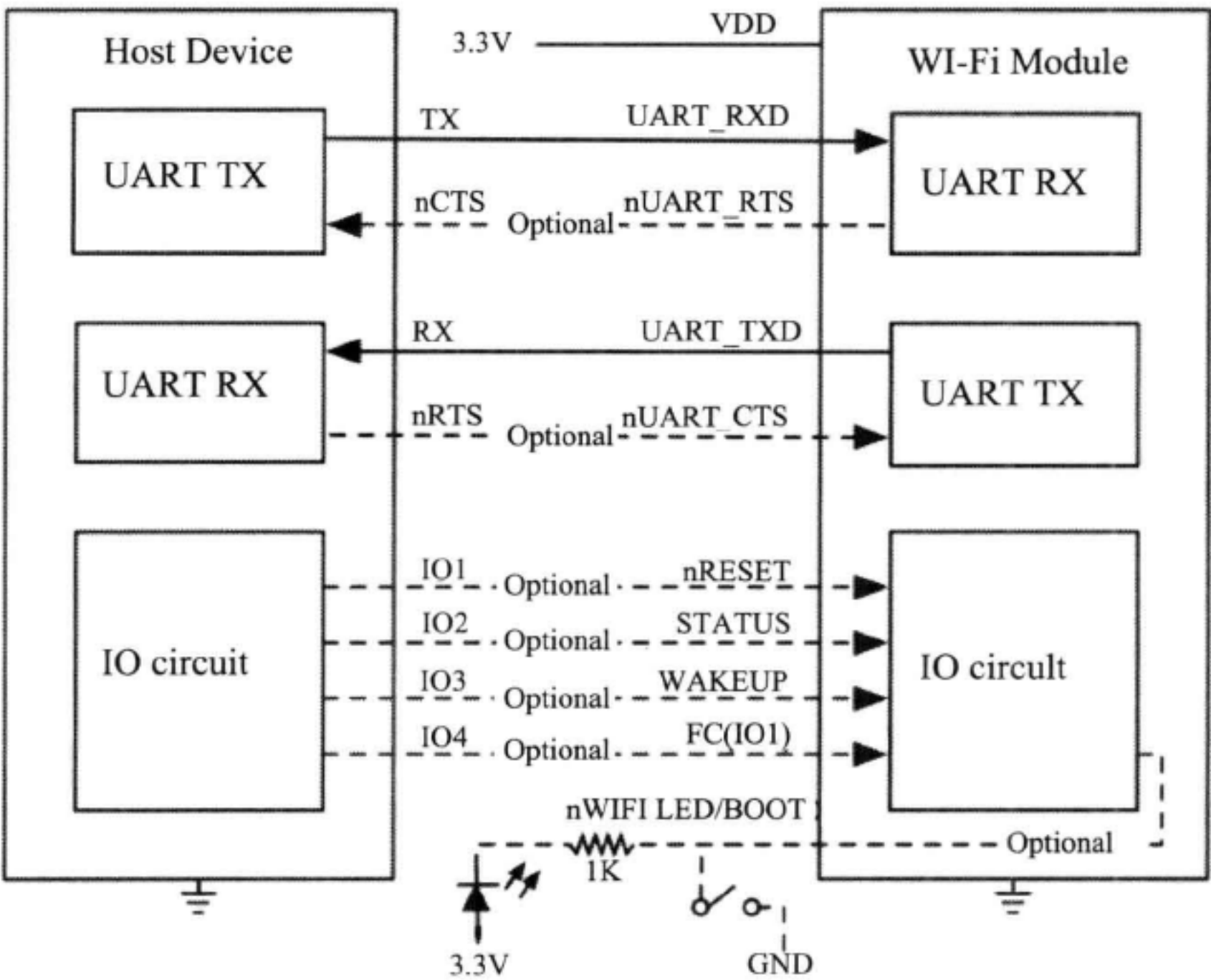


图 25-3 EMW3180 模块硬件连接图

从模块连接图看，模块的引脚主要分为两个部分，它与主机之间使用 UART 接口传输命令和数据（TX、RX、nCTS、nRTS），利用其他引脚来改变模块的工作模式（nRESET、BOOT、STATUS、WAKE_UP、IO1）。

EMW3180 模块有四种工作模式：

- ❑ 固件升级模式：升级模块的固件。
- ❑ 命令控制模式：使用厂家定义的 EMSP 命令配置模块，命令通过串口发送到模块中。
- ❑ 强制命令控制模式：与命令控制模式类似，但不需要通过 STATUS 切换模式。
- ❑ 透明传输模式：进入该模式后，模块会自动把串口接收到的数据打包通过 Wi-Fi 发送出去，对于主机来说，相当于用串口连接到网络一样。

这四个工作模式的切换过程见图 25-4。

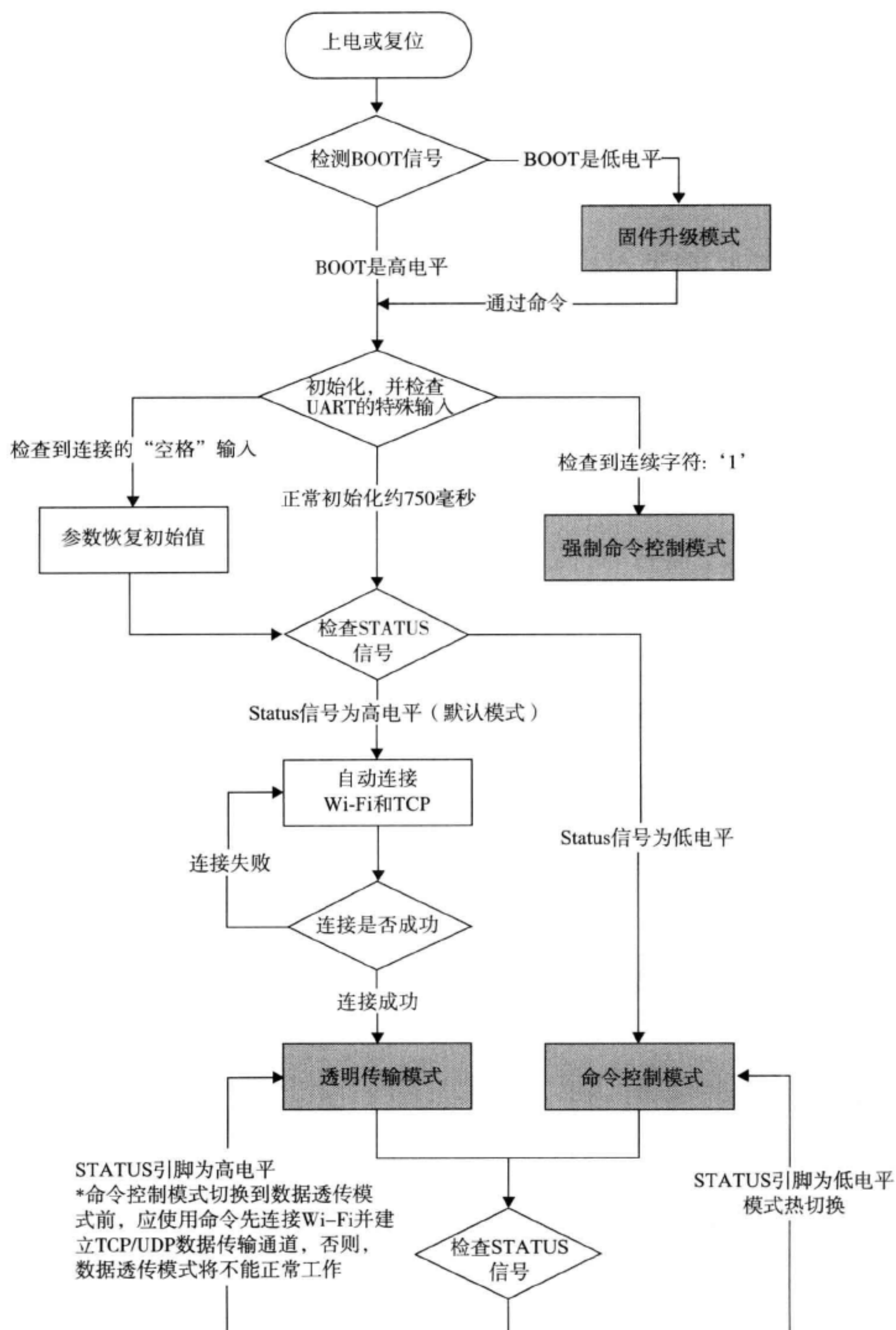


图 25-4 EMW3180 工作模式切换图

25.3 EMW3180 驱动实验

初步了解 EMW3180 后，通过具体代码向读者讲解该模块。

25.3.1 实验描述及工程文件清单

1. 实验描述

运用 Mxchip 提供的 EMSP_API 函数来配置 Wi-Fi 模块的参数，连接到无线网络，与同网段中的 PC 建立 TCP 连接，并打开 PC 端安装的 TCP/UDP 测试工具，发送数据，配套 STM32 开发板通过 Wi-Fi 将接收到的数据返回给 PC，达到回显的功能。

2. 硬件连接

- ☐ PA9: RXD
- ☐ PA10: TXD
- ☐ PA12: CTS
- ☐ PA11: RTS
- ☐ PB12: STATUS
- ☐ PB13: WAKE UP
- ☐ PB14: IO1

3. 库文件

使用 3.5 版本固件库：

- ☐ startup/start_stm32f10x_hd.c
- ☐ CMSIS/core_cm3.c
- ☐ CMSIS/system_stm32f10x.c
- ☐ FWlib/stm32f10x_rcc.c
- ☐ FWlib/stm32f10x_gpio.c
- ☐ FWlib/stm32f10x_usart.c
- ☐ FWlib/stm32f10x_fsmc.c
- ☐ FWlib/misc.c

4. 用户文件

- ☐ USER/main.c
- ☐ USER/stm32f10x_it.c
- ☐ USER/SysTick.c
- ☐ USER/lcd_botton.c
- ☐ USER/lcd.c

5. EMSP (EMW API) 文件

- ☐ EMSP.c

- ❑ EMSP_API.c
- ❑ em380c_hal.c

Wi-Fi 模块硬件连接见图 25-5。

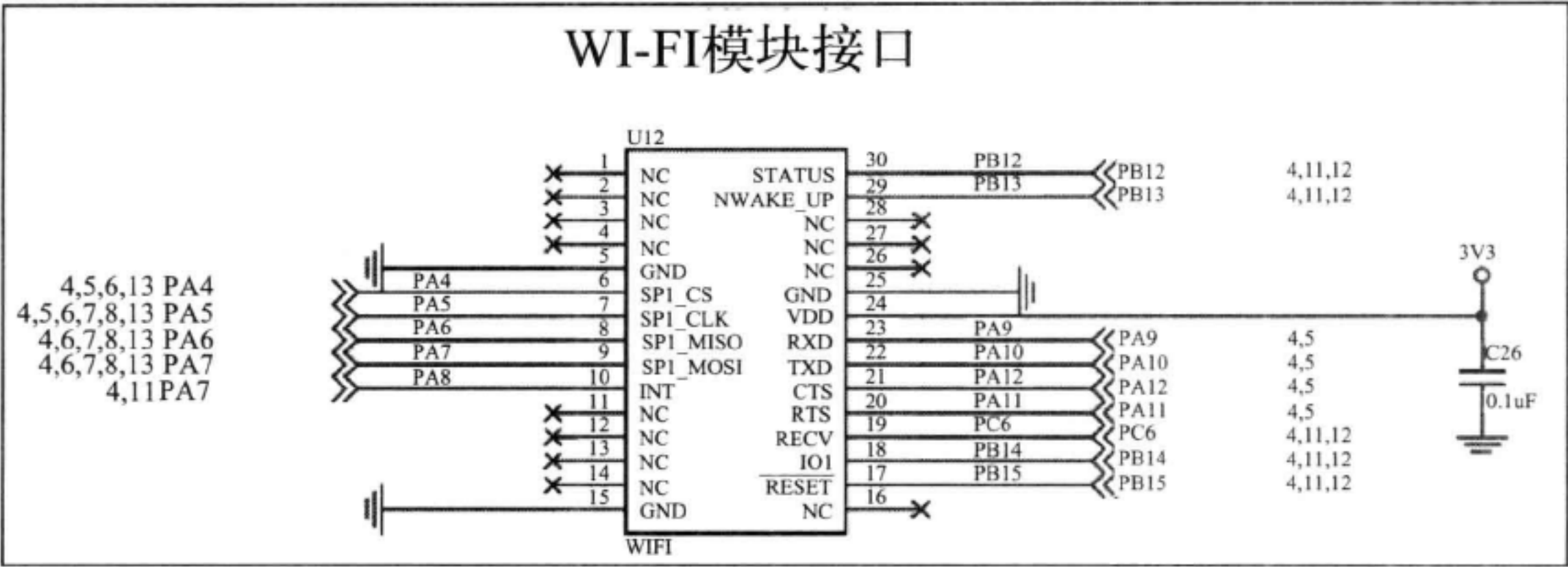


图 25-5 Wi-Fi 模块硬件连接图

25.3.2 配置工程环境

本实验中我们用到了 GPIO、RCC、FSMC、USART 外设，所以我们先要把以下库文件添加到工程中：stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c、stm32f10x_fsmc.c。实验中还使用了旧工程中的 lcd_botton.c、lcd.c 和 systick.c 文件用于 LCD 显示和实现定时功能。

另外，本工程需要使用 EMW3180 的 API，所以要把 EMSP.c、EMSP_API.c 和 em380c_hal.c 文件及其相应的头文件添加到工程中。

25.3.3 EMSP_API 函数

EMSP_API 接口函数提供了一系列 API 函数，用户通过调用这些函数可以轻松地在各种嵌入式设备上实现对 EMW 系列 Wi-Fi 模块的控制和数据传输。现在该接口函数随 Wi-Fi 资料和配套 STM32 开发板例程一并提供，见图 25-6。

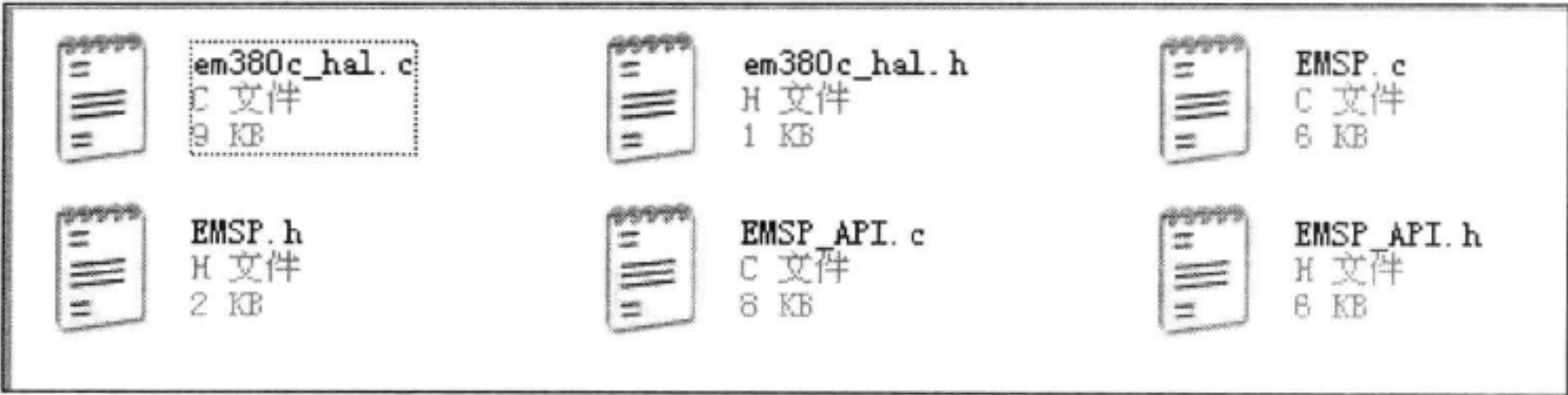


图 25-6 EMSP_API 文件

如果大家购买了配套 STM32 开发板和 Mxchip EMW 系列 Wi-Fi 模块，就可以在开发板上调试这些例程。

EMSP_API 函数库由标准 C 编写而成，可以直接加入常用的嵌入式开发环境，如 KEIL、IAR 等。EMSP_API 函数由以下三个 C 语言文件及其对应的头文件构成。

❑ emw38x_hal.c

该代码实现了 EMW 系列模块和嵌入式设备之间的硬件接口。用户需要根据自己的硬件环境实现相应的函数。

❑ EMSP.c

该代码实现了 EMSP 命令的协议处理。

❑ EMSP_API.c

该代码提供给用户用于操控模块的 API 函数，用户只需调用这些函数，就可以实现对模块的配置和操作。

25.3.4 API 函数一览

以下这些 EMW 的 API 函数位于 EMSP_API.c 文件中，见表 25-2 ~ 表 25-11。

表 25-2 模块初始化函数

函 数 名	vs8 EM380C_Init (void)
功 能	用于初始化模块以及与模块通信的 UART 接口，并使模块处于能够响应状态
返 回 值	-1：执行命令失败 0：执行命令成功

表 25-3 获取固件版本函数

函 数 名	vs8 EM380C_Get_ver (u32* version)
功 能	用于获得 EMW 系列模块的固件版本号
返 回 值	-1：执行命令失败 0：执行命令成功

表 25-4 获取网络连接状态函数

函 数 名	vs8 EMSP_Get_status (EM380C_status_TypeDef* EM380C_status)
功 能	用于获得 Wi-Fi 模块的网络连接状态
输 入	用于存放 Wi-Fi 的网络连接状态结构体地址： typedef struct { EM380C_TCPstatus_TypeDef TCPstatus; EM380C_Wi-Fistatus_TypeDef Wi-Fistatus; } EM380C_status_TypeDef;

(续)

函 数 名	vs8 EMSP_Get_status (EM380C_status_TypeDef* EM380C_status)
返 回 值	-1 : 执行命令失败
	0 : 执行命令成功

表 25-5 获取信号强度函数

函 数 名	vs8 EM380C_Get_APLst (EM380C_APLst_TypeDef* EM380C_APLst)
功 能	用于获得区域内无线 AP 的 SSID 号和相应信号强度
输 入	用于存放无线 AP 的 SSID 号和相应信号强度的线性表的起始地址： typedef struct { char AP_NAME[20]; float AP_signal; } EM380C_APLst_TypeDef;
返 回 值	-1 : 执行命令失败
	≥ 0 : 执行命令成功, 获得的 AP 信息的数量

表 25-6 启动模块 TCP/IP 连接函数

函 数 名	vs8 EM380C_Startup (void)
功 能	启动 Wi-Fi 模块的 TCP/IP 网络连接
返 回 值	-1 : 执行命令失败
	0 : 执行命令成功

表 25-7 获取模块当前配置函数

函 数 名	vs8 EM380C_Get_RF_POWER(EM380C_RF_POWER_TypeDef* RF_POWER)
功 能	用于获得 Wi-Fi 模块当前的配置参数
输 入	参数结构体的地址, 成功执行命令后, 模块当前的参数会写入这个地址。参数结构体如下 typedef struct { // Wi-Fi 配置 u8 Wi-Fi_mode; u8 Wi-Fi_ssid[32]; u8 Wi-Fi_wepkey[16]; // 40 位和 104 位 u8 Wi-Fi_wepkeylen; // 5, 13 // TCP/IP 配置 u8 local_ip_addr[16]; u8 remote_ip_addr[16]; // 服务器地址, 若模块作为服务器, 不使用 u8 net_mask[16]; // 子网掩码: 默认 255.255.255.0 u8 gateway_ip_addr[16]; // 网关 IP 地址 u8 portH; // 高 8 位 u8 portL; // 低 8 位 u8 connect_mode; // 0: 服务器 1: 客户端 u8 use_dhcp; // 0: 关闭, 1: 使能 u8 use_udp; // 0: 使用 TCP, 1: 使用 UDP

(续)

函数名	vs8 EM380C_Get_RF_POWER(EM380C_RF_POWER_TypeDef*RF_POWER)
输入	<pre>// COM 串口配置 u8 UART_buadrate; // 0:9600, 1:19200, 2:38400, 3:57600, 4:115200 u8 DMA_buffersize; // 0:2, 1:16, 2:32, 3:64, 4:128, 5:256, 6:512 u8 use_CTS_RTS; // 0: 关闭, 1: 使能 u8 parity; // 0: 不检验, 1: 偶校验, 2: 奇校验 u8 data_length; // 0:8, 1:9 u8 stop_bits; // 0:1, 1:0.5, 2:2, 3:1.5 // 设备配置 u8 device_num; // 0 - 255 u8 IO_Control; // 0 - 255 u8 sec_mode; // 0 = wep 加密, 1=wpa 加密, 2= 不加密 u8 wpa_psk[32]; } EM380C_parm_TypeDef;</pre>
返回值	<p>-1: 执行命令失败</p> <p>0: 执行命令成功</p>

表 25-8 设置配置参数函数

函数名	vs8 EM380C_Set_Config (EM380C_parm_TypeDef* EM380C_Parm)
功能	用于设置 Wi-Fi 的配置参数
输出	参数结构体的地址, 成功执行命令后, 会将该地址上的数据写入 Wi-Fi 模块, 结构体与上面 GetConfig 参数一致
返回值	<p>-1: 执行命令失败</p> <p>0: 执行命令成功</p>

表 25-9 Wi-Fi 发送数据函数

函数名	u32 EM380C_Send_Data (u8* Data,u32 len)
功能	用于通过 Wi-Fi 模块发送数据
输出 1	保存发送数据的内存空间的起始地址
输出 2	发送的数据长度
返回值	<p>>0: 执行命令成功, 返回发送的数据长度</p> <p>0: 执行命令失败</p>

表 25-10 Wi-Fi 重启函数

函数名	vs8 EM380C_Reset (void)
功能	重启 Wi-Fi 模块, 配置参数后, 需重启模块, 参数才能生效
返回值	<p>-1: 执行命令成功, 返回发送的数据长度</p> <p>0: 执行命令失败</p>

表 25-11 Wi-Fi 模式设置

函 数 名	vs8 EM380C_Set_Mode (EM380C_mode_TypeDef mode)
功 能	设置 Wi-Fi 模块模式：命令模式和透传模式
输 入	用于存放 Wi-Fi 模块的模式结构体： <pre>typedef enum { config_mode = 0x0, // 命令模式 DTU_mode = 0x1, // 透传模式 } EM380C_mode_TypeDef;</pre>
返 回 值	-1：执行命令成功 0：执行命令失败

25.3.5 main 文件

按照 main 函数的执行流程来分析程序，见代码清单 25-1。

代码清单 25-1 Wi-Fi 实例的 main 函数

```

1.  /*
2.  * 函数名：main
3.  * 描述   ：主函数
4.  * 输入   ：无
5.  * 输出   ：无
6.  */
7.  int main(void)
8.  {
9.      SysTick_Init();                /* 配置 SysTick*/
10.     LCD_Init();                    /* LCD 初始化 */
11.     /* 第一步 */
12.     while(EM380C_Init(BaudRate_115200,WordLength_8b,StopBits_1,Parity_
        No,HardwareFlowControl_None,buffer_512bytes)==EM380ERROR);
13.     printf("EMW initialization completed\n");
14.
15.     /* 获取 Wi-Fi 模块固件版本 */
16.     while(EM380C_Get_ver(&version)==EM380ERROR);
17.     printf("Ver:%8.0x\n",version);
18.
19.     /* 第二步：配置 Wi-Fi 模块参数 */
20.     parm.Wi-Fi_mode = AP;
21.     strcpy((char*)parm.wifi_ssid,"MXCHIP");
22.     strcpy((char*)parm.wifi_wepkey,"");
23.     parm.wifi_wepkeylen = 0;
24.     strcpy((char*)parm.local_ip_addr,"192.168.2.11"); //192.168.2.11
25.     strcpy((char*)parm.remote_ip_addr,"192.168.2.108"); //192.168.2.108
26.     strcpy((char*)parm.net_mask,"255.255.255.0");
27.     strcpy((char*)parm.gateway_ip_addr,"192.168.2.1"); // 192.168.2.1
28.     parm.portH = 8080>>8;
29.     parm.portL = 8080;
30.     parm.connect_mode = TCP_Client;
31.     parm.use_dhcp = DHCP_Disable;
32.     parm.use_udp = TCP_mode;
```

```

33.    parm.UART_buadrate = BaudRate_115200;
34.    parm.DMA_buffersize = buffer_256bytes;
35.    parm.use_CTS_RTS = HardwareFlowControl_None;
36.    parm.parity = Parity_No;
37.    parm.data_length = WordLength_8b;
38.    parm.stop_bits = .StopBits_1;
39.    parm.IO_Control = IO1_Normal;
40.
41. #ifdef EMW_FIRMWARE_UART
42.    parm.sec_mode = Secure_WPA_WPA2_PSK;
43.    strcpy((char*)parm.wpa_psk, "str710fz2t6");
44. #endif
45.
46.    /* 设置 Wi-Fi 模块参数 */
47.    while(EM380C_Set_Config(&parm)==EM380ERROR);
48.
49.    /* 第三步：重启 Wi-Fi 模块，使能配置 */
50.    while(EM380C_Reset()==EM380ERROR);
51.
52.    /* 第四步：启动 Wi-Fi 连接到 AP */
53.    EM380C_Startup();
54.    /* 输出 Wi-Fi 模块参数到 LCD */
55.    printf_config();
56.    /* 获取列表 */
57.    EM380C_APLst_now = malloc(10*sizeof(EM380C_APLst_TypeDef));
58.    APlist_tmp = EM380C_APLst_now;
59.    apnum=EM380C_Get_APList(EM380C_APLst_now);
60.    LCD_Rectangle(0,0,320,240,WHITE);
61.    printf("search SSID and strength.....\n");
62.    printf("AP list:\n");
63.    if(apnum == 0)
64.        printf("NO AP detected \n");
65.
66.    for(;apnum>0;apnum--, APlist_tmp++)
67.    {
68.        printf("[%s]", APlist_tmp->AP_NAME);
69.        printf("%3.0f%%\n ", APlist_tmp->AP_signal);
70.    }
71.    free(EM380C_APLst_now);
72.
73.    /* 第五步：进入透传模式 */
74.    while(EM380C_Set_Mode(DTU_mode)==EM380ERROR);
75.
76.    USART_ITConfig(USART1, USART_IT_RXNE, DISABLE);
77.
78.    while(1)
79.    {
80.        /* 若 USART 收到数据，把原文发送回去 */
81.        if(USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == SET){
82.            USART_SendData(USART1, USART_ReceiveData(USART1));
83.        }
84.    }
85.}

```

main 函数比较长，针对 Wi-Fi 模块的操作主要分为 5 个步骤（见代码中的步骤说明）：

1) 调用 EMSP API 的 EM380C_Init() 初始化硬件接口和引脚，拉低 STATUS 引脚。

2) 设置 Wi-Fi 模块参数, 把配置参数填充到 parm 结构体, 并通过 EMSP API 的 EMSP_SET_CONFIG() 命令发送给 Wi-Fi 模块。这些配置参数包括 Wi-Fi 类型、模块 IP 地址、服务器 IP 地址、子网掩码、网关等参数。模块详细的功能可参考“Wi-Fi 模块 datasheet”文件 EMW_DataTransferExamples.pdf, 里面详细介绍了各种模式的数据透传。

3) 调用 EMSP API 的 EM380C_Reset() 重启 Wi-Fi 模块, 模块的参数配置好之后, 需要重启 Wi-Fi 模块才能生效。

4) 调用 EMSP API 的 EM380C_Startup() 启动 Wi-Fi 模块, 通过发送 EMSP_CMD_START 命令, 此时模块内部 TCP/IP 协议栈已经开启, 模块上红灯常亮。

5) 调用 EMSP API 的 EM380C_Set_Mode(), 模块进入透传模式。另外, 直接拉高 STATUS 引脚可直接启动 Wi-Fi, 跳过第 4 步。

对于这些在 main 函数调用的 EMSP API, 读者可查询 EMSP_API 函数的说明, 这些函数在 EMSP_API.c 文件中定义。而在底层还需要用户亲自编写一些函数来初始化 STM32 的硬件。

25.3.6 em380c_hal.c 文件

底层的硬件初始化都是在 em380c_hal.c 中定义的。在 EMSP API 中的初始化函数 EM380C_Init() 和默认初始化函数 EM380C_InitWithDefault() 需要底层实现, 这两个 API 函数分别调用 EM380C_HAL_Init() 函数和 EM380C_HAL_InitWithDefault() 函数, 所以用户需要在这两个函数中, 完成底层硬件的初始化。以 EM380C_HAL_Init() 函数为例, 见代码清单 25-2。

代码清单 25-2 EM380C_HAL_Init() 函数

```

1.  /*
2.   * 函数名: EM380C_HAL_Init
3.   * 描述   : 初始化 EMW3180 引脚, 波特率
4.   * 输入   :
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8.  vs8 EM380C_HAL_Init(uint32_t BaudRate, uint16_t WordLength, uint16_t
   StopBits, uint16_t Parity, uint16_t CTSRTS)
9.  {
10.     vs8 ret = -1, i;
11.     buff_pos = 0;
12.     //STM_EVAL_LEDInit(LED2);
13.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_USART1, ENABLE);
14.
15.     /* 配置 USART */
16.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
17.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
18.     GPIO_Init(GPIOA, &GPIO_InitStructure); /* RX 引脚 */
19.
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
21.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
22.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
23.     GPIO_Init(GPIOA, &GPIO_InitStructure); /* TX 引脚 */
24.

```



```

25.
26.  if(CTSRTS)
27.  {
28.      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
29.      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
30.      GPIO_Init(GPIOA, &GPIO_InitStructure);          /* CTS 引脚 */
31.
32.      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
33.      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
34.      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
35.      GPIO_Init(GPIOA, &GPIO_InitStructure);          /* RTS 引脚 */
36.  }
37.
38.  /* 配置 NVIC 优先级组 */
39.  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
40.
41.  /* 使能 UART 中断 */
42.  NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
43.  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
44.  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
45.  NVIC_Init(&NVIC_InitStructure);
46.
47.  USART_InitStructure.USART_BaudRate = UART_BaudRate[BaudRate];
48.  USART_InitStructure.USART_WordLength = UART_WordLength[WordLength];
49.  USART_InitStructure.USART_StopBits = UART_StopBits[StopBits];
50.  USART_InitStructure.USART_Parity = UART_Parity[Parity];
51.  USART_InitStructure.USART_HardwareFlowControl = UART_CTSRTS[CTSRTS];
52.  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;          /* 配置 USART
参数 */
53.  USART_Init(USART1, &USART_InitStructure);
54.
55.  USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
56.
57.  USART_Cmd(USART1, ENABLE);          /* 使能 USART */
58.
59.
60.  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC, ENABLE);
61.
62.
63.  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13;
64.  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
65.  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
66.  GPIO_Init(GPIOB, &GPIO_InitStructure);
67.
68.
69.  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
70.  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
71.  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
72.  GPIO_Init(GPIOC, &GPIO_InitStructure);
73.  RESET_STATUS_PIN();          /* 把 STATUS 引脚置 0 */
74.  SET_WAKEUP_PIN();          /* 把 WAKEUP 引脚拉高 */
75.
76.  //GPIO_ResetBits(GPIOA, GPIO_Pin_12);          /* RTS=0 */
77.
78.  if(!READ_INT_PIN())
79.      ret = 0;
80.  return ret;
81.}

```

与其他章节的例子一样，这个函数就是把 STM32 最基本的外设配置好，包括使用到的所有引脚、串口及其中断优先级，在最后还是调用了 SET_STATUS_PIN()、SET_WAKEUP_PIN() 等函数，这是封装好了的控制 GPIO 引脚电平的函数，见代码清单 25-3。

代码清单 25-3 SET_STATUS_PIN()、SET_WAKEUP_PIN() 等函数

```

1. /*STATUS 引脚置 1*/
2. void SET_STATUS_PIN(void)
3. {
4.     GPIO_SetBits(GPIOB, GPIO_Pin_12);
5. }
6.
7. /*STATUS 引脚置 0*/
8. void RESET_STATUS_PIN(void)
9. {
10.    GPIO_ResetBits(GPIOB, GPIO_Pin_12);
11.}
12./*WAKEUP 引脚置 1*/
13.void SET_WAKEUP_PIN(void)
14.{
15.    GPIO_SetBits(GPIOB, GPIO_Pin_13);
16.}
17./*WAKEUP 引脚置 0*/
18.void RESET_WAKEUP_PIN(void)
19.{
20.    GPIO_ResetBits(GPIOB, GPIO_Pin_13);
21.}

```

EMSP API 还有 send_cmd() 和 recv_cmd() 函数，它们分别需要底层实现用串口发送一串数据和接收一串数据的函数，这两个函数分别由用户在 em380c_hal.c 的 UART_send_buf() 和 UART_receive_buf() 函数实现，见代码清单 25-4。

代码清单 25-4 UART_receive_buf() 函数

```

1.
2. /*
3.  * 函数名：UART_send_buf
4.  * 描述   ：利用串口发送数据
5.  * 输入   ：数据地址，长度
6.  * 输出   ：
7.  * 调用   ：外部调用
8.  */
9. void UART_send_buf(u8 *buf, int len)
10.{
11.    int i;
12.    buff_pos=0;
13.    for (i = 0; i < len; i++)
14.    {
15.        USART_SendData(USART1,buf[i]);
16.        while(!USART_GetFlagStatus(USART1,USART_FLAG_TXE));

```

```

17.     }
18. }
19.
20. /*
21.  * 函数名: UART_receive_buf
22.  * 描述  : 接收串口缓冲区数据
23.  * 输入  : 存储数据的地址
24.  * 输出  : 数据指针
25.  * 调用  : 外部调用
26.  */
27. int UART_receive_buf(u8 *buf)
28. {
29.     memcpy(buf, EM380RxBuffer, buff_pos);
30.     return buff_pos;
31. }
32.
33.

```

最后，由于串口使用中断方式收发数据，所以还要编写中断服务函数，见代码清单 25-5。

代码清单 25-5 串口中断服务函数

```

1. /*
2.  * 函数名: USART1_IRQHandler
3.  * 描述  : 串口中断服务程序
4.  * 输入  :
5.  * 输出  :
6.  * 调用  : 外部调用
7.  */
8. void USART1_IRQHandler(void)
9. {
10.     if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
11.     {
12.         /* Read one byte from the receive data register */
13.         EM380RxBuffer[buff_pos++] = USART_ReceiveData(USART1);
14.     }
15. }

```

STM32 使用 UART1 接口与 Wi-Fi 模块通信，所以本例程中的 printf 函数并不是输出到串口再 PC 的终端的，它被重定向至 STM32 开发板上的 LCD 屏上，重定向代码在 lcd.c 文件中，见代码清单 25-6。

代码清单 25-6 fputc 函数重定向

```

1. /* 重定向 fputc 函数 */
2. #ifdef __GNUC__
3.     /* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
4.        set to 'Yes') calls __io_putchar() */
5.     #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
6. #else
7.     #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
8. #endif /* __GNUC__ */
9.

```



```

10.
11./**
12. * @brief Retargets the C library printf function to the LCD.
13. * @param None
14. * @retval None
15. */
16.PUTCHAR_PROTOTYPE
17.{
18.    static uint8_t x =0 , y = 0;
19.    u8 acsii;
20.    acsii = ch;
21.//    unsigned long i;
22.
23.    Set_direction(0);
24.
25.    switch(ch)
26.    {
27.        case ('\n'):
28.            x = 0;
29.            y += 12;
30.            if(y>228)
31.            {
32.                y = 0;
33.            }
34.            break;
35.        case ('\r'):
36.            x=0;
37.            break;
38.        default:
39.            if(x==320)
40.            {
41.                x=0;
42.                y+=12;
43.            }
44.            if(y>228)
45.            {
46.                y=0;
47.            }
48.
49.            LCD_Char_6x12_O( x, y, acsii, RED);
50.            x += 6;
51.            break;
52.    }
53.    return ch;
54.}

```

至此，本例程讲解完毕，由于直接使用模块，驱动程序比以太网的例程要简单得多。

25.3.7 实验现象

将配套 STM32 开发板供电（DC5V），插上 J-LINK，把本工程文件编译后烧录到开发板上。运行 TCP/UDP 测试工具，把 PC 接入到 STM32 即将要连接到的同一个路由。运行 STM32 开发板的程序。

看到实验现象如下（本实验运行较慢，请耐心等待）：

1) 读取到 Wi-Fi 模块的配置参数，并且显示到 LCD 屏，见图 25-7。

2) 读取搜索到的周围的无线网络和信号强度, 见图 25-8。

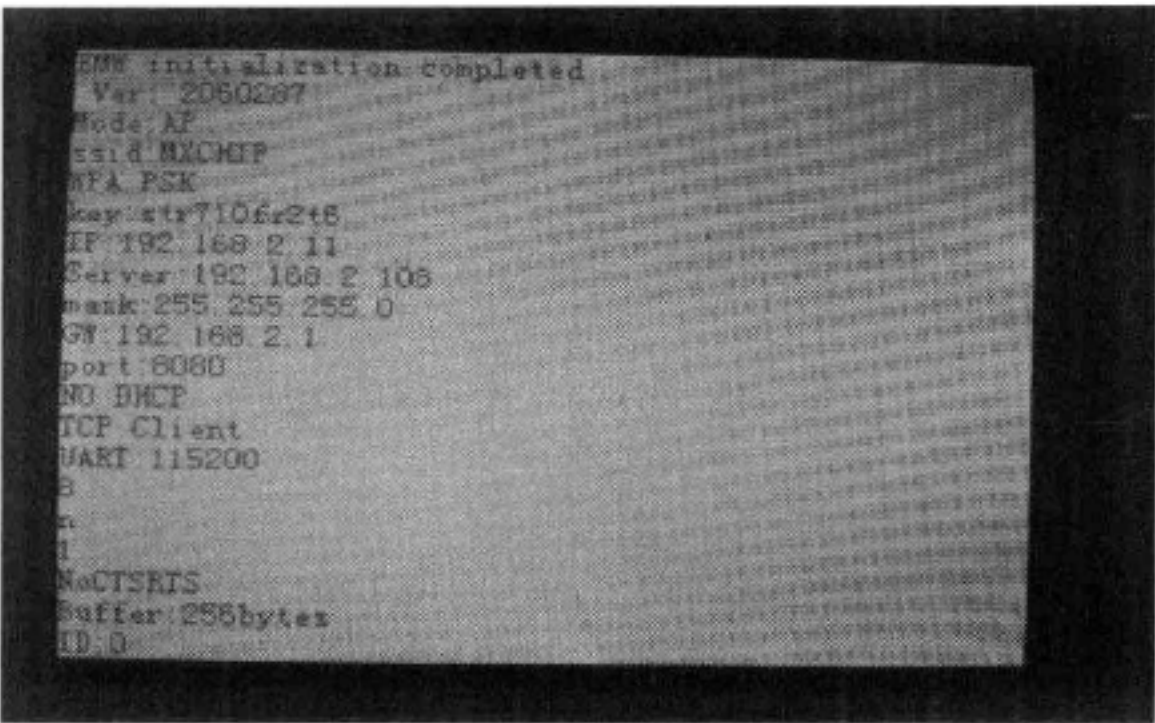


图 25-7 读取 Wi-Fi 模块参数到 LCD 屏

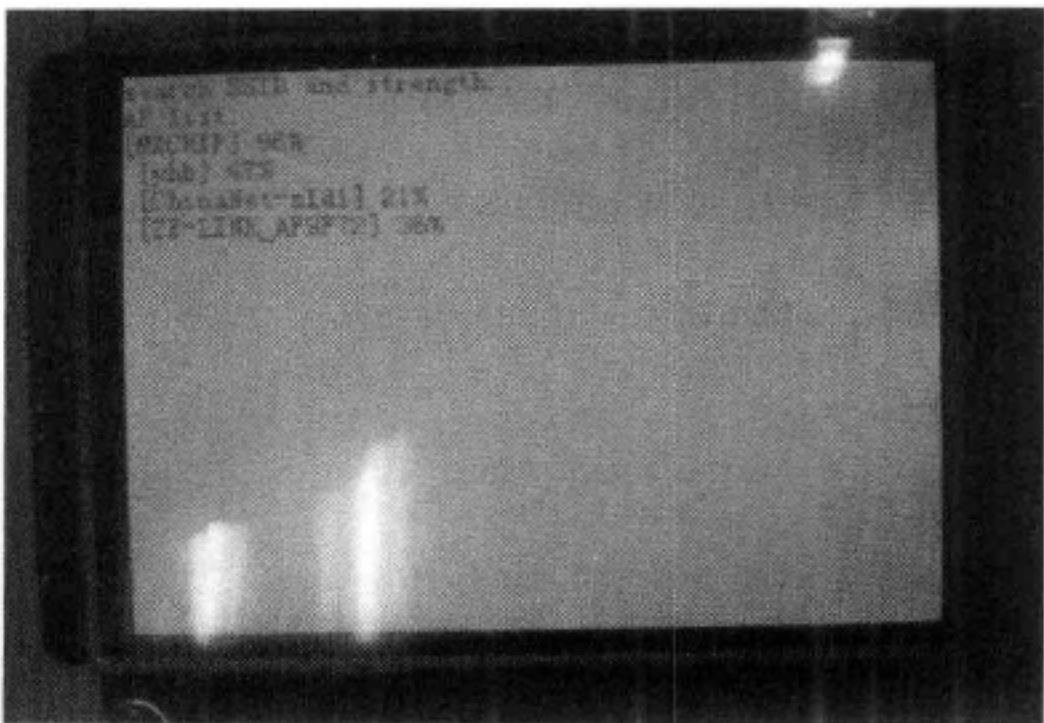


图 25-8 LCD 显示周围无线网络的信号强度

3) 在 PC 端 TCP/UDP 测试工具发送的数据通过 Wi-Fi 网络发送给开发板, 开发板接收数据并通过 Wi-Fi 模块发送给 PC, 达到回显的功能, 见图 25-9。

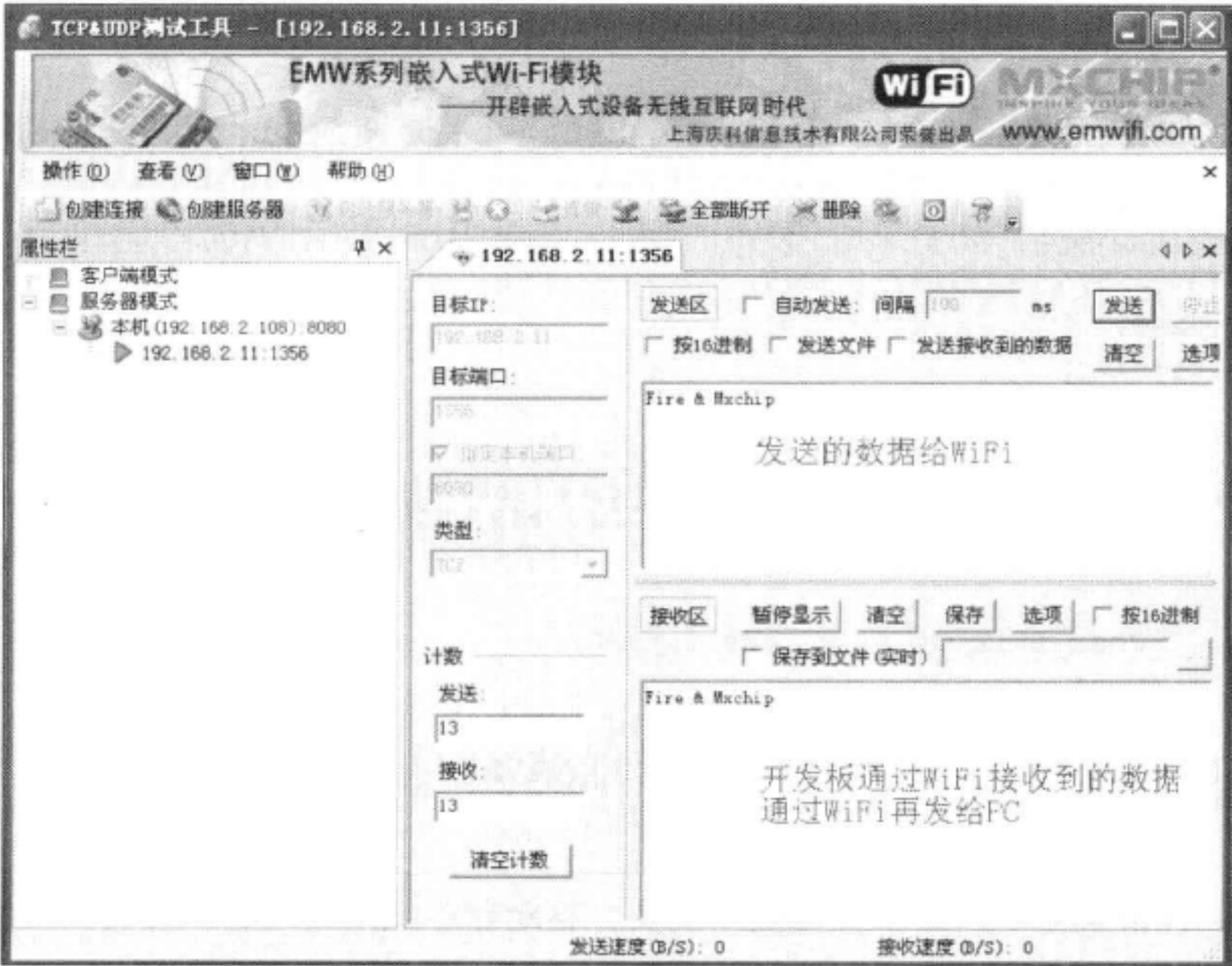


图 25-9 STM32 通过 Wi-Fi 回显字符串到 PC



第四部分 库开发系统篇

由于在裸机上编程可对 CPU、RAM 等硬件资源进行直接管理，在软件规模比较小时，可开发出运行效率很高的程序。但随着嵌入式系统的软件规模越来越大，软件的开发效率会变得越来越低，而同时嵌入式系统的硬件资源变得越来越充裕，因此开发人员引入操作系统层，以对下层嵌入式平台的硬件资源进行统一管理，简化上层应用软件的开发。

μ C/OS 操作系统是开源代码，又因为它小巧而严谨，具有很高的实时性，被很多高校用于讲解操作系统原理，为学习高级操作系统打下基础，而且在实际项目中也经常使用 μ C/OS 开发。STM32 作为控制领域的中高端 MCU，硬件资源充裕，可开发复杂的软件应用，尤其适合使用嵌入式系统。本书以 μ C/OS-III 为例，讲解如何在 STM32 平台上移植并使用嵌入式系统进行软件开发，进一步提升读者的嵌入式软件开发水平。

- 第26章 μ C/OS-III 及其源代码介绍
- 第27章 移植 μ C/OS-III 到 STM32
- 第28章 运行多任务



第 26 章

μC/OS-III 及其源代码介绍

26.1 μC/OS 简介

26.1.1 操作系统与裸机的区别

裸机运行的程序代码，一般由一个 main 函数中的 while 死循环和各种中断服务程序组成，平时 CPU 执行 while 循环中的代码，出现其他事件时，跳转到中断服务程序进行处理，没有多任务、线程的概念。

而引入操作系统后，程序运行时可以把一个应用流程分割为多个任务，每个任务完成一部分工作，并且每个任务都可以写成死循环。操作系统根据任务的优先级，通过调度器使 CPU 分时执行各个任务，保证每个任务都能够得到运行。若调度方法优良，则可使各任务看起来是并行执行的，减少了 CPU 的空闲时间，提高了 CPU 的利用率。由操作系统的任务管理衍生出相应的 CPU 管理、内存管理，它们分别负责分配任务对 CPU 的占有权和管理任务所占有的内存空间。在 Linux 等操作系统中，还具有文件管理、I/O 设备管理的功能。

26.1.2 μC/OS 实时操作系统

实时操作系统是指当外界事件或数据产生时，能够接收并以足够快的速度予以处理，其处理的结果又能在规定的时间内控制生产过程或对处理系统做出快速响应，并控制所有实时任务协调一致运行的操作系统。

实时操作系统要求严格的及时事件响应，并且还具有非常好的稳定性能。这样的系统是针对航空航天、工控等对响应时间有严格要求的应用场合而产生的，在这些场合，传统的 PC 及其相应系统是难以胜任的。

μC/OS 是一个由 Micrium 公司开发的微型实时操作系统，包括了一个操作系统最基本的一些特性，如任务调度、任务通信、内存管理、中断管理、定时管理等。而且这是一个代码完全开放的实时操作系统，简单明了的结构和严谨的代码风格，非常适合初涉嵌入式操作系统的人士学习。对于没有学习过操作系统的朋友，建议购买相关书籍学习下操作系统的基础知识，而有一定基础的可看看 Micrium 的官方文档《μC/OS-III : The Real-Time Kernel》。

在运行 $\mu\text{C}/\text{OS}$ 系统的设备上，当程序执行时，首先会初始化系统任务管理所需要的各种链表等数据结构，接着根据应用程序需求创建任务，最后由调度器管理各个任务，而中断可由操作系统使能和除能，若使能中断则可以在其他任务运行时跳转到中断服务程序。 $\mu\text{C}/\text{OS}$ 的运行流程见图 26-1。

各步骤的详细说明如下：

1) 初始化所有全局变量、数据结构，创建最低优先级空闲任务 OSTaskIdle （如果使用了统计任务，也在此创建），创建 6 个空数据链表：

- ☐ 空任务控制块链表
- ☐ 空事件控制块链表
- ☐ 空队列控制块链表
- ☐ 空标志组链表
- ☐ 空内存控制块链表
- ☐ 空闲定时器控制块链表

2) 至少创建一个任务。一般创建一个最高优先级别 TaskStart 任务（建议），任务调度后，在这个任务中再创建其他任务，初始化硬件并开中断。

3) 进入多任务管理阶段，将就绪表中最高优先级任务的栈指针加载到 SP 中，并强制中断返回。

4) $\mu\text{C}/\text{OS}$ 的任务调度工作。任务调度是内核的主要服务，是区分裸机和多任务系统的最大特点，好的调度策略能更好地发挥系统的效率。调度工作主要包括：查找就绪表中最高优先级任务和实现任务切换。而任务切换又分为两种，分别为任务级的调度器 OSSched 和中断级的调度器 OSIntExt 。

5) 运行用户任务，某些用户任务会因为主动让出 CPU、延时、请求临界资源或优先级不够高而挂起，由调度器调度运行其他任务。

6) $\mu\text{C}/\text{OS}$ 的任务调度是靠周期时钟中断来实现的，每个时钟节拍到来就会产生一次定时中断，中断后启动调度器，运行就绪表中优先级最高的任务（非抢占型内核中断后继续运行被中断任务）。即过一段时间就检测是否有重要任务需要运行，若是就转而运行更重要的任务，从而确保实时性（裸机程序就无法这样做了）。在 CM3 芯片平台上，这个周期时钟中断由 SysTick 提供。

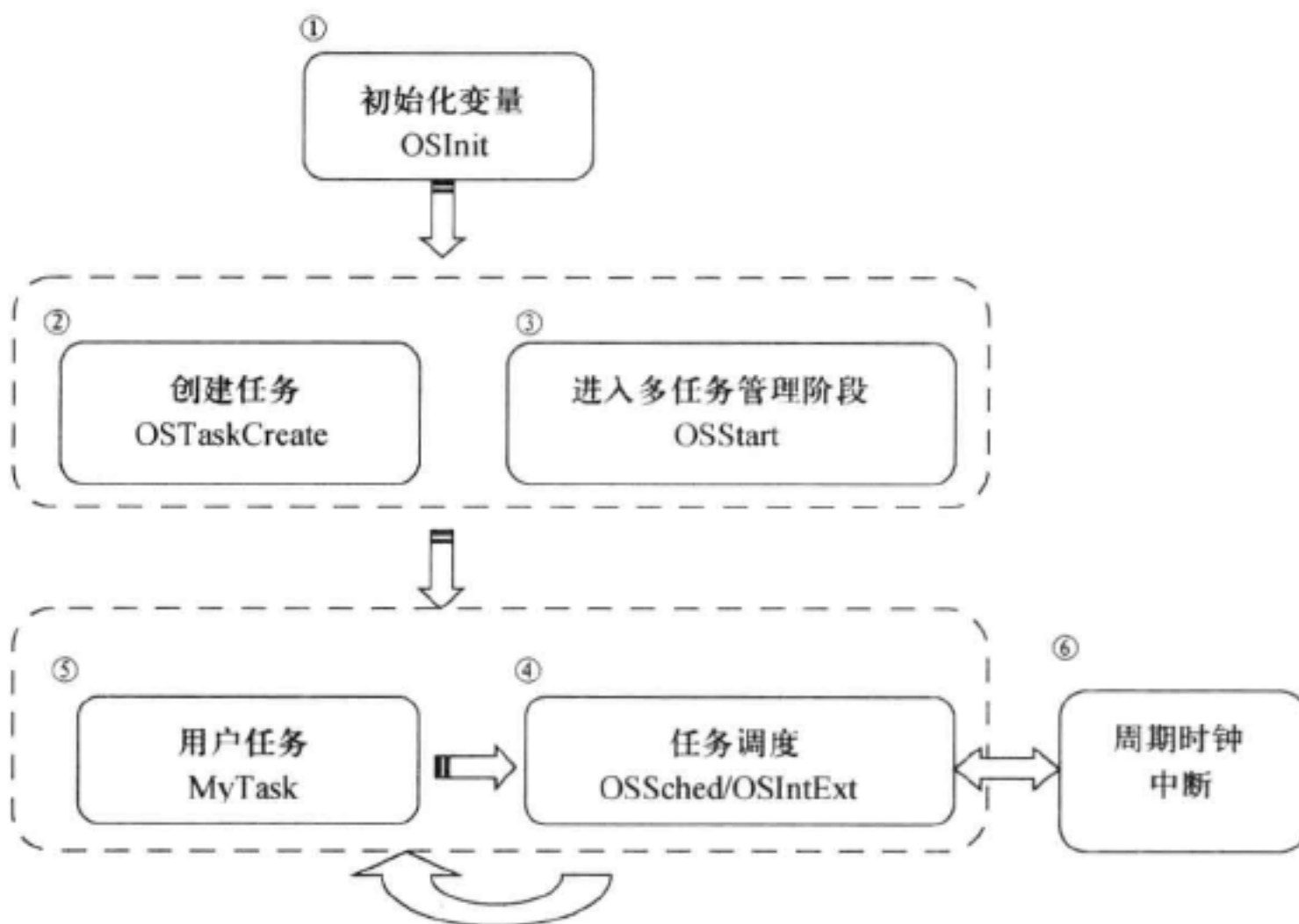


图 26-1 $\mu\text{C}/\text{OS}$ 工作流程图

26.2 μ C/OS-III与 μ C/OS-II的主要区别

相对于 μ C/OS-II， μ C/OS-III系统性能有了较大的改动和提升，主要区别见表 26-1。

表 26-1 μ C/OS-II 与 μ C/OS-III的主要区别

功 能	μ C/OS-II	μ C/OS-III
最大任务数	256	无限制
每个优先级的任务个数	1	无限制
时间片轮转法	不支持	支持
消息邮箱	支持	不支持（已经不需要了）
不使用信号量标记任务	否	是
不使用消息队列发消息给任务	否	是
在运行时配置	允许	不允许
嵌入的测量功能	少量	大量
时间戳	不支持	支持
汇编可优化	否	是
任务级的时基定时器处理	不支持	支持

最直观的感受是 μ C/OS-III的任务数只受相应平台的内存大小限制，由于支持了时间片轮转法，使得支持多个任务具有同样的优先级，以及同优先级的任务以时间片轮转方式运行。

26.3 μ C/OS-III源码

为近距离了解 μ C/OS-III，首先要了解其源代码，它可以从 Micrium 公司官方下载地址获取：
<http://micrium.com/page/downloads/ports/st/stm32>（下载资料需要注册账号）。

由于官方没有提供 MDK 平台下的移植例子，我们需要下载 IAR 开发平台下的 3.02 版本，读者也可使用光盘中的源码文件。

下载解压后可以看到 Micrium \Software 含有 4 个文件夹，见图 26-2。

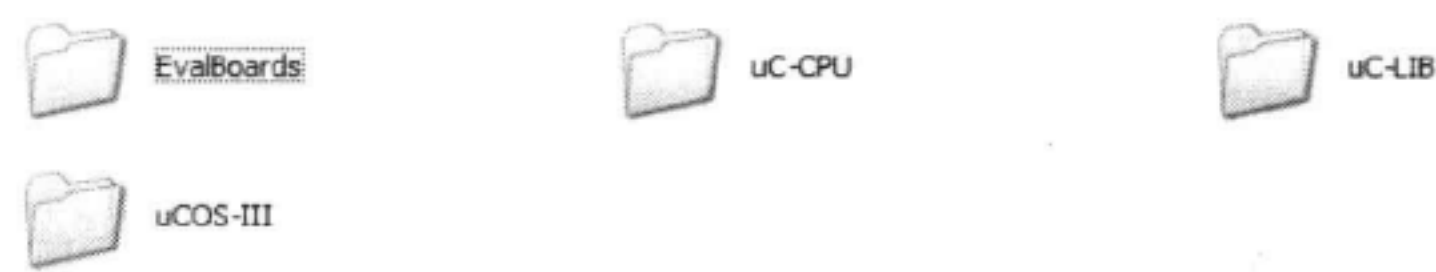


图 26-2 解压后的源码目录

其包含的主要文件及功能见表 26-2。

表 26-2 μ C/OS 源码文件内容

文 件 名	说 明		
Software	Software 下的文件是官方移植到评估板上的例子，包含了 μ C/OS 内核源码、CPU、C 函数库及板级文件相关文件。在移植过程中主要使用 uC-CPU、uC-LIB 及 uCOS-III 下的内容		
	uCOS-III	Doc	uC/OS 官方自带说明文档和版本信息
		Ports	官方移植到 STM32 的移植文件 (IAR 工程)
			os_cpu.h os_cpu_c 和 os_cpu_a 函数声明
			os_cpu_c.c 定义用户钩子函数，提供扩充软件功能的入口点（所谓钩子函数，是指那些插入某函数中拓展这些函数功能的函数）
			os_cpu_a.asm 与处理器相关的汇编函数，主要是任务切换函数
		Cfg	os_app_hooks.c 空定义的钩子函数
			os_app_hooks.h 钩子函数声明
			os_cfg_app.h 空闲堆栈、时钟速率、内存池
			os_cfg.h 配置 μ C/OS 的功能配置
		Source	μ C/OS 的内核源代码文件
			os.h 内部函数参数设置
			os_dbg.c 内核调试数据和函数
			os_core.c 内核结构管理， μ C/OS 的核心，包含了内核初始化、任务切换、事件块管理、事件标志组管理等功能
			os_time.c 时间管理，主要是延时
			os_tmr.c 定时器管理，设置定时时间，时间到了就进行一次回调函数处理
			os_task.c 任务管理
			os_mem.c 内存管理
			os_sem.c 信号量
			os_mutex.c 互斥信号量
			os_q.c 队列
			os_flag.c 事件标志组
			os_msg.c 消息处理

(续)

文 件 名	说 明			
Software	uCOS- III	Source	os_prio.c	任务优先级
			os_int.c	中断处理
			os_stat.c	统计任务
			os_pend_multi.c	多个信号量或消息队列
			os_cfg_app.c	声明变量和数组
			os_tick.c	任务延时、停止
			os_var.c	全局变量
	EvalBoards	Micrium 官方评估板的代码		
		μC/OS- III -Ex1	os_type.h	声明数据类型
	uC-CPU	基于 Micrium 官方评估板的 CPU 移植代码		
	uC-LIB	Micrium 官方的一个库代码		

我们下载的这个例子是可以在 Micrium 官方提供的目标板上运行的，部分文件是移植时在 μC/OS-III 内核的基础上添加的。如 uCOS-III 文件夹中的 Cfg 和 Source 文件夹内是纯 uC/OS-III 内核文件；Ports 文件夹内是官方移植到目标板时针对 CM3 内核编写的 CPU 相关文件，主要包含与任务切换相关的函数；uC-CPU 文件也是 CPU 相关文件，它以函数的形式封装了 CM3 特定的 CPU 功能；uC-LIB 是由 Micrium 提供的 C 语言函数库，作用与 keil 编译器提供的微库类似。

26.4 μC/OS-III 工程架构

操作系统是对硬件层的封装与抽象，所有的其他应用层软件都依赖于操作系统的服务，使用操作系统后，软件的架构与裸机相比有了很大的区别，见图 26-3。

从下至上，分别为硬件层，CPU、板级相关的抽象层，μC/OS 内核层，应用层。各部分按图中的编号解释如下：

- 1) Application Code：应用层代码，这部分代码架设在系统层之上，与内核代码分离，灵活度很高，是完全由用户建立的。
- 2) CPU：芯片厂家提供的用于控制 CPU 或 MCU 外设的固件库函数，在本书中即为 STM32 固件库。
- 3) BSP：板级支持包，即对目标板的初始化、控制相关的代码，如本书中相关工程的驱动代码 led.c、led.h、usart1.c、usart1.h 等。
- 4) μC/OS-III CPU Independent：μC/OS-III 的内核代码，这部分代码完全独立于具体的处理器，完成了操作系统的核心工作。
- 5) μC/OS-III CPU Specific：μC/OS-III 适应不同架构 CPU 的相关代码，在不同的 CPU 平台，

μ C/OS 内核任务切换的相关代码稍有区别, 这是移植的关键。Micrium 官方已经给出在 STM32 平台下移植的范例, 这部分文件保存在 Ports 文件夹中, 我们可以直接使用。

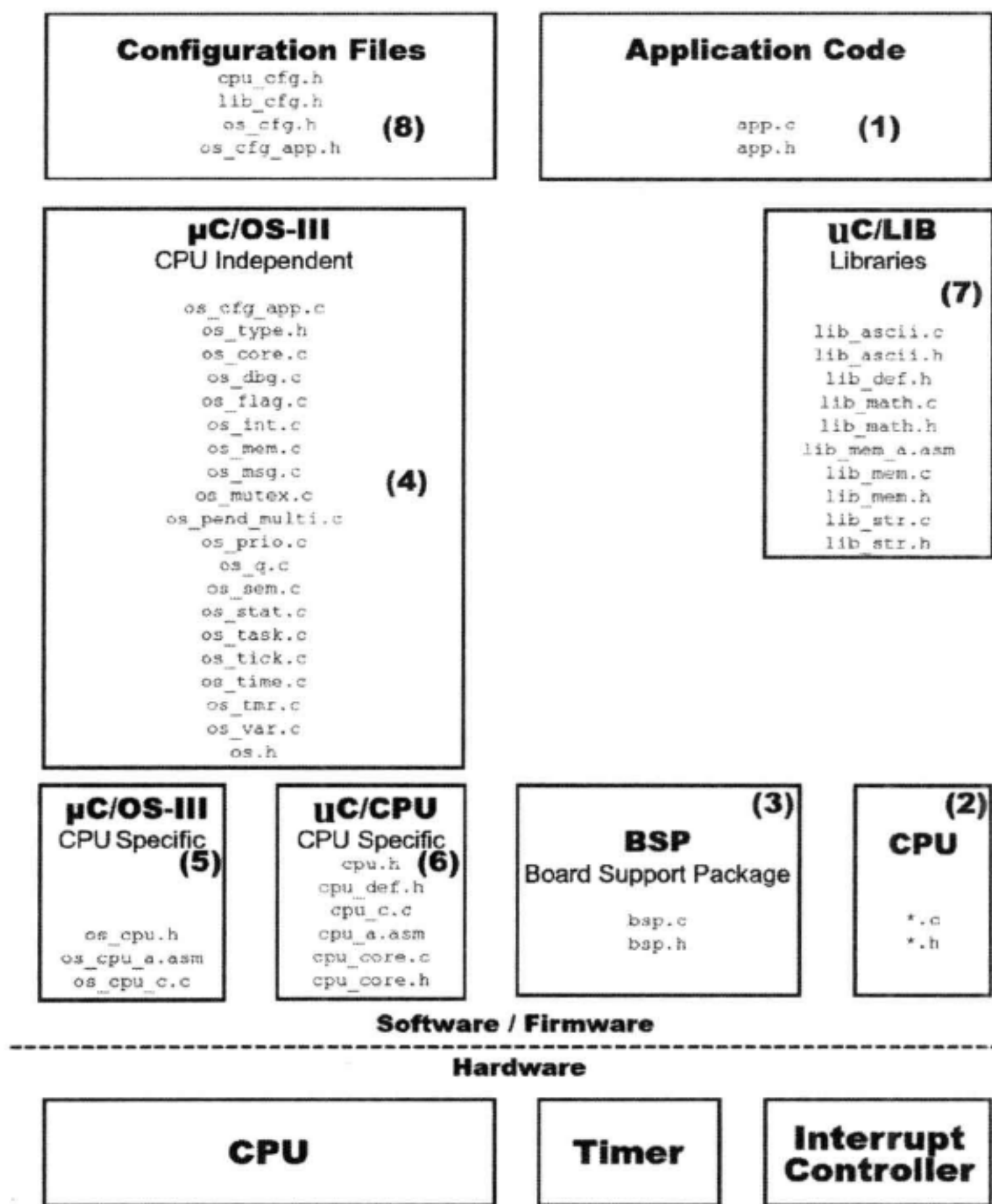


图 26-3 μ C/OS-III 工程架构

6) μ C/CPU CPU Specific : Micrium 官方针对不同 CPU 总结出的相关 CPU 的某些功能, 并封装到这部分文件的函数中。在 STM32 平台下可以直接使用 Micrium 官方提供的这些文件。

7) μ C/LIB : Micrium 提供的 C 语言函数库, 可用于代替编译器提供的微库, μ C/OS-III 内核本身不使用这些文件, 但 μ C/CPU 需要使用。

8) Configuration Files : μ C/OS-III 的功能配置文件, 对这些文件做合适的修改, 可以对 μ C/OS-III 系统进行裁剪。

从 μ C/OS 工程架构和前面提到的运行流程可知, 在移植时, 由于 CPU 平台的区别, 部分 CPU 相关文件需要修改, 为使系统内核的调度器工作, 要将 CPU 的定时器与内核关联起来。在应用时, 则需要充分利用多任务带来的便捷, 并处理好因多任务而出现的资源竞争等问题。



第 27 章

移植 μ C/OS-III 到 STM32

为了更深入地学习 μ C/OS，我们必须要以一个平台为载体。在本章中，我们将带领读者一步步把 μ C/OS-III 系统移植到配套 STM32 开发板，并在 μ C/OS 系统层之上，建立流水灯的基本任务以及讲解系统编程与裸机编程的区别。

27.1 搭建 μ C/OS 工程文件结构

我们的 μ C/OS-III 移植教程是建立在配套 STM32 的 LED 工程基础上的，在它的基础上按一定规则建立分类文件夹，将开发板的硬件驱动程序与 μ C/OS 系统文件分开，使得工程更有条理。

首先我们要提取配套 STM32 LED 工程的代码，从光盘资料中复制一份到一个新目录下，并重命名文件夹为“STM32+UC/OS+LED”。解压打开工程后，就会看到 LED 工程的文件结构如图 27-1 所示。

这是我们在前面开发裸机单片机程序时写的工程文件结构，但对于 μ C/OS-III 或者其他大型的软件工程，这样的文件结构就会显得很乱。所以，我们需要重新建立文件结构，见表 27-1（其他没显示出来的文件，按照原来位置不改变）。

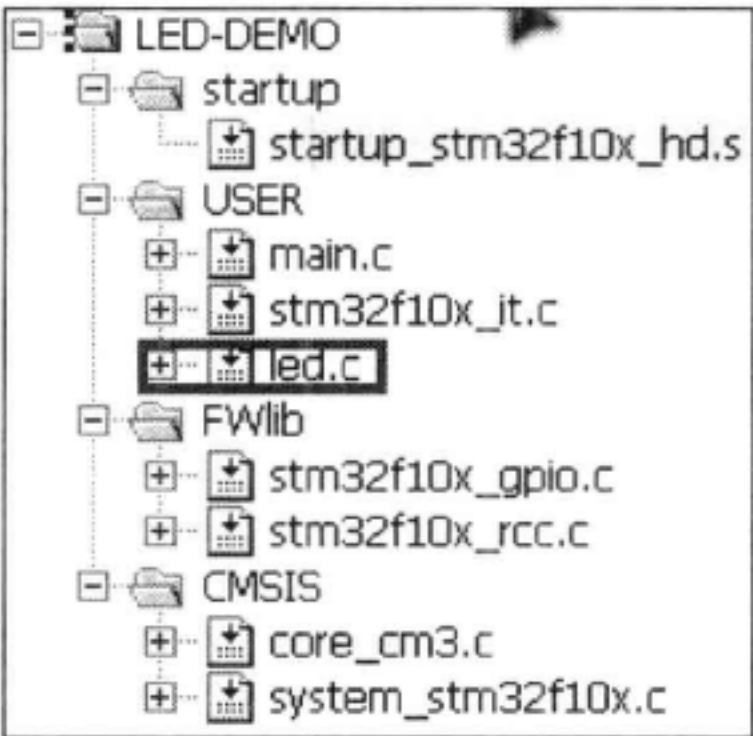


图 27-1 LED 工程文件结构

表 27-1 STM32+ μ C/OS+LED 结构

文 件 夹		文 件 名	说 明
USER		来自旧工程的文件夹	
		main.c	新建的文件
		includes.h	新建的文件
uCOS-III	Source	除 os_type.h 外，所有文件来自于下载附件的 Micrium\Software\uCOS-III	
		os_cfg_app.c	
		os_core.c	
		os_dbg.c	

(续)

文 件 夹	文 件 名	说 明
uCOS-III	os_flag.c	
	os_int.c	
	os_mem.c	
	os_msg.c	
	os_mutex.c	
	os_pend_multi.c	
	os_prio.c	
	os_q.c	
	os_sem.c	
	os_stat.c	
	os_task.c	
	os_tick.c	
	os_time.c	
	os_tmr.c	
	os_var.c	
	os.h	
	os_type.h	本文件来自目录：\Micrium\Software\EvalBoards\M..\uC..\IAR..\uCOS-III-Ex1..
	Ports	所有文件来自目录：\Micrium\Micrium\Software\uCOS-III\Ports 及其子文件夹
		os_cpu.h
		os_cpu_c.c
		os_cpu_a.asm
	新建 Cfg 文件夹	文件来自目录：\Micrium\Micrium\Software\uCOS-III\Cfg\Template
		os_app_hooks.c
		os_app_hooks.h
		os_cfg_app.h
		os_cfg.h
	新建 uC-CPU 文件夹	文件来自目录：Micrium\Micrium\Software\uC-CPU 及其子文件夹
		cpu_core.c
		cpu_core.h
		cpu_c.c
		cpu.h
		cpu_a.asm
		cpu_def.h
		cpu_cfg.h

(续)

文 件 夹		文 件 名	说 明
uCOS-III	新建 uC-LIB 文件夹	文件来自目录 : Micrium\Micrium\Software\ uC-LIB 及其子文件夹	
		lib_ascii.c	
		lib_ascii.h	
		lib_cfg.h	
		lib_math.c	
		lib_math.h	
		lib_mem.c	
		lib_mem.h	
		lib_str.c	
		lib_str.h	
		lib_def.h	
		lib_cfg.h	
新建 BSP 文件夹		led.c	来自原 USER 文件夹
		led.h	来自原 USER 文件夹
		BSP.c	新建文件
		BSP.h	新建文件
新建 APP 文件夹		除说明的文件外, 其他来自目录 : Mi..\Soft..\EvalBoards\M..\uC..\ \IAR\uCOS-III-Ex1	
		app.c	新建文件
		app.h	新建文件
		os_cfg_app.h	
		app_cfg.h	
		os_cfg.h	

为了方便初学者, 下面为具体步骤 :

- 1) 把 LED 工程所在的文件夹先改名为 STM32+μC/OS+LED (建议这样做, 避免与原来的 LED 工程混乱)。
- 2) 在 USER 文件夹下新建 includes.h 头文件。
- 3) 按照前面的 μC/OS-III 工程架构图 26-3, 在工程的根目录下建立 BSP 文件夹、APP 文件夹和 uCOS-III 文件夹。
□ BSP 文件夹 : 存放外设硬件驱动程序。

□ APP 文件夹：存放应用软件任务。

□ uCOS-III 文件夹： μ C/OS-III 的相关代码。

4) 把原 USER 文件夹下的 led.h 和 led.c 文件剪切到 BSP 文件夹里。在 BSP 文件夹里新建 BSP.c 和 BSP.h 文件。

5) 在 APP 文件夹下建立 app.h、app.c 文件。进入 μ C/OS-III 源代码附件目录 \Micrium\Software\EvalBoards\Micrium\uC-Eval-STM32F107\IAR\uCOS-III-Ex1，把其中的 os_cfg_app.h、app_cfg.h、os_cfg.h 文件复制到此目录。

6) 在 uCOS-III 文件夹中建立 Source 文件夹，复制 μ C/OS-III 源码中的 Source 文件夹下的全部文件。进入 μ C/OS-III 源代码目录 \Micrium\Software\EvalBoards\Micrium\uC-Eval-STM32F107\IAR\uCOS-III-Ex1，把其中的 os_type.h 文件复制到此目录。

7) 在 uCOS-III 文件夹中建立 Ports 文件夹，把 μ C/OS-III 源码目录 \Micrium\Software\uC/OS-III\Ports\ARM-Cortex-M3\Generic\IAR 下的文件复制到工程 uCOS-III 文件夹中新建的 Ports 文件夹里。

8) 在 uCOS-III 文件夹中建立 uC-LIB 文件夹，把 μ C/OS-III 源码目录 \Micrium\Micrium\Software\uC-LIB 及子文件夹中的文件复制到工程的该目录中。

9) 在 uCOS-III 文件夹中建立 uC-CPU 文件夹，把 μ C/OS-III 源码目录 \Micrium\Software\uC-CPU 及子文件夹中的文件复制到工程的该目录中。

10) 在 uCOS-III 文件夹中建立 Cfg 文件夹，把 μ C/OS-III 源码目录 Micrium\Software\uCOS-III\Cfg\Template 中的文件复制到工程的该目录中。

11) 最后，选中工程目录文件夹 STM32+ μ C//OS+LED，右键选择“属性”去掉只读属性，单击“确定”，将该更改应用于该文件夹、子文件夹和文件，以便我们修改源码文件。

到此，工程的目录结构就建立好了，下面需要修改工程设置。

12) 打开工程文件，会发现提示出错，原因是我们修改了 led.h 和 led.c 的路径，直接点击“确定”就可以了。见图 27-2。

然后需要在项目里手动删掉原来的 led.c，见图 27-3。

13) 建立 BSP、APP、uC/Source、uC/Ports、uC/LIB、uC/CPU 和 uC/Cfg 七个文件夹组，并添加进相应的文件，添加完毕后效果见图 27-4。

注意 要添加工程目录下的所有文件，别漏了在 uC\Ports 中添加汇编文件 os_cpu_a.asm 和 uC/CPU 中的汇编文件 cpu_a.asm。

14) 添加头文件路径，把各文件夹路径都添加进来，见图 27-5。

即将 Include Paths 设置为：..\CMSIS;..\USER;..\FWlib\inc;..\FWlib\src;..\APP;..\BSP;..\uC/OS-III\Cfg;..\uCOS-III\Ports;..\uC/OS-III\Source;..\uCOS-III\uC-CPU;..\uCOS-III\uC-LIB。

至此，已完成全部工程的设置，需要开始修改移植代码了！



图 27-2 错误提示

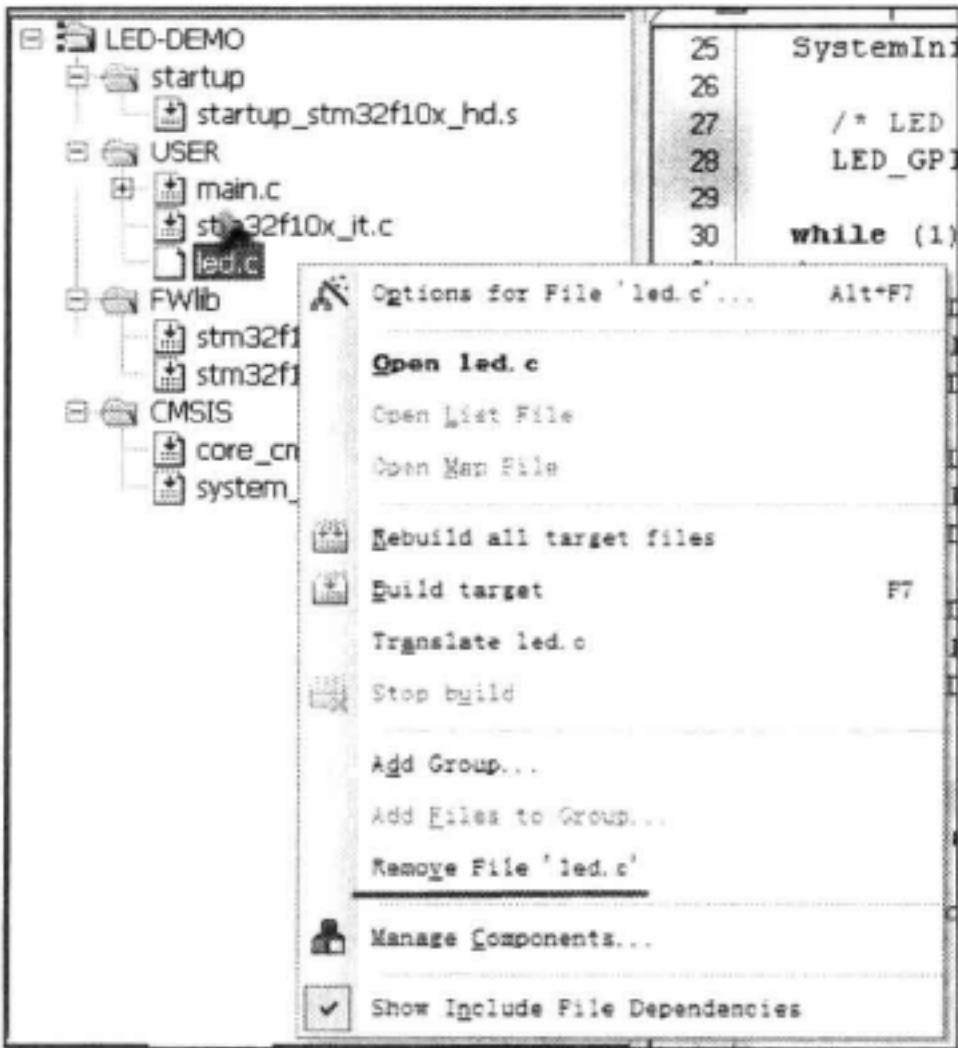


图 27-3 删除原工程目录的 led 文件

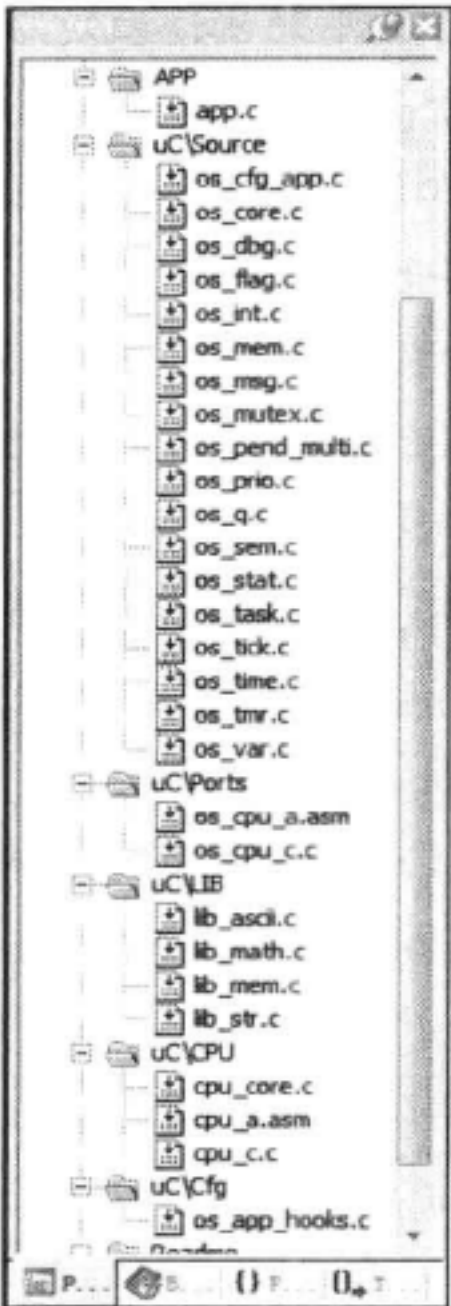


图 27-4 添加完整的工程目录

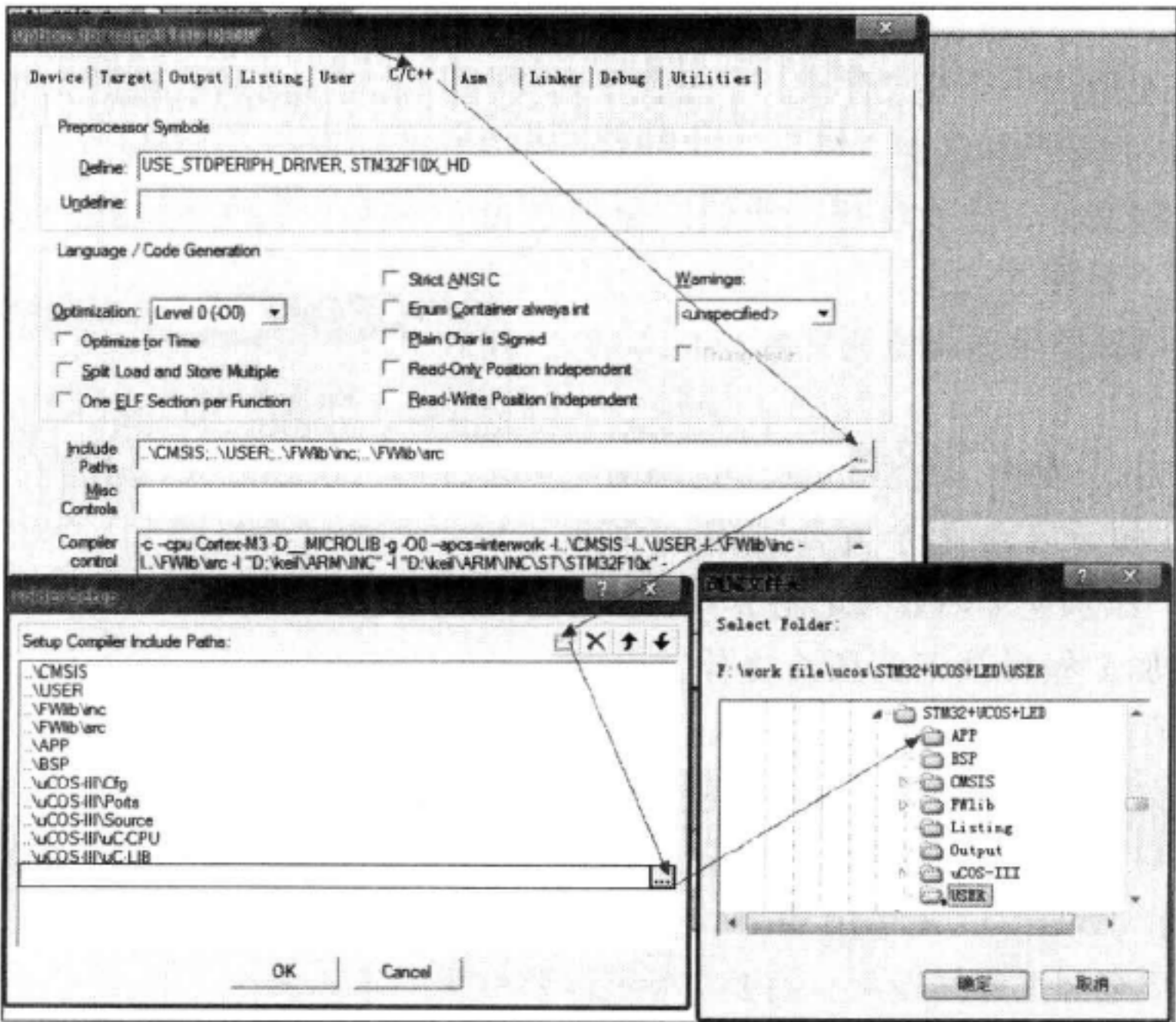



图 27-5 设置头文件路径

27.2 修改 μ C/OS 代码

读者设置完工程环境后，编译一下工程，出现错误提示的地方都是我们需要修改的。这些需要修改的文件大都与 CPU 相关，每修改完一个文件，就点击  按钮编译一下该文件（不要直接编译链接工程），具体按照下面的说明一步步修改即可。

27.2.1 修改 os_cpu.h 文件

μ C/OS-III 的内核使用一个周期时钟中断，以计算任务延时时间和进行任务调度，在 STM32 中，这样的时钟中断正适合由 SysTick 来提供。Micrium 官方提供的文件中包含了 SysTick 的中断服务函数，它在 os_cpu.h 文件中，见代码清单 27-1。

代码清单 27-1 Micrium 官方的 SysTick 的中断服务函数

```
1. void OS_CPU_SysTickHandler(void); // 系统定时中断处理函数，时钟节拍函数
2. void OS_CPU_SysTickInit(CPU_INT32U cnts); // 系统 SysTick 定时器初始化
```

这两个函数的作用见表 27-2。

表 27-2 μ C/OS 的 SysTick 相关函数

函 数 名	功 能
OS_CPU_SysTickHandler()	在 os_cpu.c 中定义，是 SysTick 中断的中断处理函数，而在 stm32f10x_it.c 中已经有该中断函数的定义 SysTick_Handler()
OS_CPU_SysTickInit()	定义在 os_cpu.c 中，用于初始化 SysTick 定时器

因为 OS_CPU_SysTickHandler() 函数与我们更熟悉的 stm32f10x_it.c 文件中的 SysTick_Handler() 函数功能一样，都是用于 SysTick 的中断处理，为便于理解，我们采用 SysTick_Handler() 函数。相应的 SysTick 初始化函数 OS_CPU_SysTickInit() 也可以用 ST 的库函数完成。

所以，我们对 os_cpu.h 文件的操作是：注释掉 OS_CPU_SysTickHandler() 和 OS_CPU_SysTickInit() 函数的声明。

27.2.2 修改 os_cpu.c

OS_CPU_SysTickHandler() 和 OS_CPU_SysTickInit() 函数的定义在 os_cpu.c 文件中，OS_CPU_SysTickHandler() 函数见代码清单 27-2。

代码清单 27-2 OS_CPU_SysTickHandler() 函数

```
1.
2. void OS_CPU_SysTickHandler (void)
3. {
4.     CPU_SR_ALLOC();
5.     CPU_CRITICAL_ENTER();
```

```

6.     OSIntNestingCtr++;
7.     CPU_CRITICAL_EXIT();
8.     OSTimeTick();
9.     OSIntExit();
10.}
11.

```

OS_CPU_SysTickInit() 函数见代码清单 27-3。

代码清单 27-3 OS_CPU_SysTickInit() 函数

```

1.
2. void OS_CPU_SysTickInit (CPU_INT32U cnts)
3. {
4.     CPU_INT32U prio;
5.     CPU_REG_NVIC_ST_RELOAD = cnts - 1u;
6.     prio = CPU_REG_NVIC_SHPRI3;
7.     prio &= DEF_BIT_FIELD(24, 0);
8.     prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);
9.
10.    CPU_REG_NVIC_SHPRI3 = prio;
11.    CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_CLKSOURCE |
12.                            CPU_REG_NVIC_ST_CTRL_ENABLE;
13.
14.    CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_TICKINT;
15.}
16.

```

前面提到我们采用 ST 官方库提供的函数来对 SysTick 进行中断处理，所以我们要把 os_cpu_c.c 的 OS_CPU_SysTickInit() 和 OS_CPU_SysTickHandler() 这两个函数注释掉。

27.2.3 修改 os_cpu_a.asm 文件

由于我们下载的 μ C/OS-III 移植工程是官方在 IAR 编译环境下建立的，IAR 在汇编的语法方面与我们使用的 MDK 编译器有一点区别，所以我们要修改汇编文件的部分指令。

在 os_cpu_a.asm 文件中，要将原来的 PUBLIC 指令改为 EXPORT，它们是等价的。即把代码清单 27-4 改为代码清单 27-5。

代码清单 27-4 要修改的代码 1

```

1. PUBLIC OSStartHighRdy          ; Functions declared in this file
2. PUBLIC OS_CPU_PendSVHandler

```

代码清单 27-5 修改后的代码 1

```

1. EXPORT OSStartHighRdy          ; Functions declared in this file
2. EXPORT OS_CPU_PendSVHandler

```

同样由于编译环境的原因，下面的代码也要进行修改，把原来的代码清单 27-6 改为代码清单 27-7。

代码清单 27-6 要修改的代码 2

```
1. RSEG CODE:CODE:NOROOT(2)
2. THUMB
```

代码清单 27-7 修改后的代码 2

```
1. PRESERVE8
2.
3. AREA |.text|, CODE, READONLY
4. THUMB
```

27.2.4 修改 cpu_a.asm 文件

在 cpu_a.asm 汇编文件中,也有不少因编译环境不同引起的错误,修改的方法一样。把代码清单 27-8 改为代码清单 27-9。

代码清单 27-8 要修改的代码 3

```
1. PUBLIC CPU_IntDis
2. PUBLIC CPU_IntEn
3.
4. PUBLIC CPU_SR_Save
5. PUBLIC CPU_SR_Restore
6.
7. PUBLIC CPU_CntLeadZeros
8. PUBLIC CPU_RevBits
9.
10. PUBLIC CPU_WaitForInt
11. PUBLIC CPU_WaitForExcept
```

代码清单 27-9 修改后的代码 3

```
1. EXPORT CPU_IntDis
2. EXPORT CPU_IntEn
3.
4. EXPORT CPU_SR_Save
5. EXPORT CPU_SR_Restore
6.
7. EXPORT CPU_CntLeadZeros
8. EXPORT CPU_RevBits
9.
10. EXPORT CPU_WaitForInt
11. EXPORT CPU_WaitForExcept
```

把代码代码清单 27-10 改为代码清单 27-11。

代码清单 27-10 要修改的代码 4

```
1. RSEG CODE:CODE:NOROOT(2)
2. THUMB
```

代码清单 27-11 修改后的代码 4

```
1. PRESERVE8
2.
3. AREA |.text|, CODE, READONLY
4. THUMB
```

除此之外，在 `cpu_a.asm` 文件中某些标号带有冒号，如“CPU_CntLeadZeros:”、“CPU_RevBits:”、“CPU_WaitForInt:”和“CPU_WaitForExcept:”，为适应编译环境，需要把这些标号中的冒号去掉。

27.2.5 修改 `startup_stm32f10x_hd.s` 文件

针对 `startup_stm32f10x_hd.s` 文件的修改是：将该文件中所有出现 `PendSV_Handler` 的地方替换成 `OS_CPU_PendSVHandler`。

操作系统通过 `SysTick` 中断进行任务调度，在任务切换时保存当前任务的上下文（如同裸机程序中，进入中断时要保护上下文一样）。但操作系统规定，若 `SysTick` 抢占了其他中断服务程序 `ISR`，不能进行任务的上下文切换，只能等待下一次的 `SysTick` 再次中断，这就带来任务调度延迟的问题，尤其是某 `ISR` 与 `SysTick` 的中断频率较为接近时。

在 `CM3` 内核中，有一个中断异常称为 `PendSV`，意为可挂起系统服务，它是专门用于解决以上问题的。若 `SysTick` 抢占了 `ISR`，就使用 `PendSV` 挂起任务的上下文切换，当 `ISR` 处理完急事务时，立即进行任务的上下文切换，而不用再等待到下一个 `SysTick` 中断，具体可参考《`ARM Cortex-M3` 权威指南》。

使用 `PendSV` 时，可在 `stm32f10x_it.c` 文件的 `PendSV_Handler` 函数中编写任务的上下文切换操作，但这部分必须使用汇编语言编写，难度较大，所以我们采用 `Micrium` 官方提供的 `OS_CPU_PendSVHandler` 函数，该函数的定义在 `os_cpu_a.asm` 文件中，见代码清单 27-12。

代码清单 27-12 `OS_CPU_PendSVHandler` 函数

```
1. OS_CPU_PendSVHandler
2.     CPSID     I
3.     MRS       R0, PSP
4.     CBZ       R0, OS_CPU_PendSVHandler_nosave
5.     SUBS      R0, R0, #0x20
6.     STM       R0, {R4-R11}
7.
8.     LDR       R1, =OSTCBCurPtr
9.     LDR       R1, [R1]
10.    STR       R0, [R1]
11.OS_CPU_PendSVHandler_nosave
```



```

12.    PUSH    {R14}
13.    LDR     R0, =OSTaskSwHook
14.    BLX     R0
15.    POP     {R14}
16.
17.    LDR     R0, =OSPrioCur
18.    LDR     R1, =OSPrioHighRdy
19.    LDRB    R2, [R1]
20.    STRB    R2, [R0]
21.
22.    LDR     R0, =OSTCBCurPtr
23.    LDR     R1, =OSTCBHighRdyPtr
24.    LDR     R2, [R1]
25.    STR     R2, [R0]
26.
27.    LDR     R0, [R2]
28.    LDM     R0, {R4-R11}
29.    ADDS    R0, R0, #0x20
30.    MSR     PSP, R0
31.    ORR     LR, LR, #0x04
32.    CPSIE   I
33.    BX      LR
34.

```

当产生 PendSV 异常时，要使程序跳转到 os_cpu_a.asm 的 OS_CPU_PendSVHandler 这段代码中进行上下文切换。但我们知道产生中断异常都是跳转到 stm32f10x_it.c 文件中相应的中断服务函数的（PendSV 异常跳转到 PendSV_Handler），如何使它跳转到 os_cpu_a.asm 的 OS_CPU_PendSVHandler 呢？在本书第 7 章讲解过中断向量表是在启动文件 startup_stm32f10x_hd.s 中定义的，只要我们修改中断向量表，使 PendSV 异常的中断向量由原来的 PendSV_Handler 改为指向 OS_CPU_PendSVHandler 即可。

27.2.6 修改 stm32f10x_it.c 文件

- 1) 把 stm32f10x_it.c 文件原有的 PendSV_Handler 空函数注释掉。
- 2) 更为重要的一点是编写 SysTick 中断服务函数，内容见代码清单 27-13。

代码清单 27-13 编写 SysTick 中断服务函数

```

1. #include "includes.h"
2.
3. void SysTick_Handler(void)
4. {
5.     OSIntEnter();    // 用于统计中断的嵌套层数，对嵌套层数 +1
6.     OSTimeTick();    // 统计时间，遍历任务，对延时任务计时减 1
7.     OSIntExit();     // 对嵌套层数减 1，在退出中断前启动任务调度
8. }
9.

```

在 stm32f10x_it.c 文件中，需要完成中断服务函数。通过前面的分析已经知道，原来的中断服务函数 PendSV_Handler 的作用已经被 OS_CPU_PendSVHandler 替换，所以我们要把 PendSV_

Handler 空函数注释掉。

在 SysTick 的服务函数中调用了 3 个函数，它们都是 $\mu\text{C}/\text{OS}$ 源码定义的函数，在移植 $\mu\text{C}/\text{OS}$ 时，可依葫芦画瓢，函数的具体内容可在读者理解 $\mu\text{C}/\text{OS}$ 内核时再学习，其基本功能如下：
 OSIntEnter() 函数，对用于表示中断嵌套层数的变量 OSIntNesting 加 1，它与 OSIntExit() 函数成对出现，在进入中断服务函数时，都应包含这两个函数，中断服务的内容位于这两个函数之间。OSIntExit() 函数除了对嵌套层数 OSIntNesting 减 1 表示退出中断外，还具有任务调度功能。OSTimeTick() 函数主要工作是对系统统计时间的变量 OSTime 加 1（就相当于钟表的秒针加 1 一样），另外，它还会遍历所有任务，对延时任务的延时时间减 1。

配置好这样的 SysTick 中断服务函数，每当产生 SysTick 中断时， $\mu\text{C}/\text{OS}$ 内核就会判断是否需要任务调度，所以这是系统实现多任务的基础。

对 $\mu\text{C}/\text{OS}$ 的工程文件针对编译环境做了修改，把 SysTick 中断修改到我们熟悉的 stm32f10x_it.c 文件后，就基本完成 $\mu\text{C}/\text{OS}$ 的移植了。剩下的大部分都是编写用户文件，如果读者觉得使用 MDK 编译环境太麻烦，可使用 IAR 编译环境，这样就不需要对 $\mu\text{C}/\text{OS}$ 文件做过多的修改，IAR 还可以使用 $\mu\text{C}/\text{OS}$ 提供的 PROBE 插件，可以为调试 $\mu\text{C}/\text{OS}$ 系统提供方便。

27.3 编写用户文件

修改完 $\mu\text{C}/\text{OS}$ 内核文件后，还需要编写用户文件。在这些用户文件中，仍然有部分内容与内核相关，如 SysTick 的初始化等，其余的都是为应用而编写的。在本节中编写的文件都是用户新建的文件，主要文件内容为驱动、板级初始化和应用层函数。

27.3.1 编写 includes.h 文件

在 $\mu\text{C}/\text{OS}$ 的工程中，我们使用 includes.h 文件包含所有头文件。阅读这个文件的内容，可以让大家整理思路，看看我们的工程包含了什么头文件，以便让大家知道 $\mu\text{C}/\text{OS}$ 的工程架构。本工程中 includes.h 文件内容见代码清单 27-14。

代码清单 27-14 includes.h 文件内容

```

1. #ifndef __INCLUDES_H__
2. #define __INCLUDES_H__
3.
4. #include "stm32f10x.h"
5. #include "os.h"           //μC/OS-III 系统函数头文件
6. #include "os_type.h"      //μC/OS-III 系统函数头文件
7. #include "BSP.h"          //与开发板相关的函数
8. #include "app.h"          //用户任务函数
9. #include "led.h"          //LED 驱动函数
10.
11. #endif //__INCLUDES_H__

```

它主要包含了 STM32 固件库所需的 stm32f10x.h, μ C/OS 系统的两个头文件 os.h、os_type.h 和用户编写的其他头文件, 如 BSP.h、app.h、led.h。

27.3.2 编写 BSP 相关文件

与目标板级相关的内容, 我们通常会把它编写到用户自建的 BSP.c 文件中, 并附有相应的头文件 BSP.h。

BSP.c 文件内容见代码清单 27-15, 它包含了 BSP_Init() 和 SysTick_init() 函数。

代码清单 27-15 BSP.c 文件内容

```

1. #include "includes.h"
2. #include "os_cfg_app.h"
3. /*
4.  * 函数名: BSP_Init
5.  * 描述  : 时钟初始化、硬件初始化
6.  * 输入  : 无
7.  * 输出  : 无
8.  */
9. void BSP_Init(void)
10. {
11.     SysTick_init();
12.     LED_GPIO_Config(); /* LED 端口初始化 */
13. }
14.
15. /*
16.  * 函数名: SysTick_init
17.  * 描述  : 配置 SysTick 定时器
18.  * 输入  : 无
19.  * 输出  : 无
20.  */
21. void SysTick_init(void)
22. {
23.     SysTick_Config(SystemCoreClock/OS_CFG_TICK_RATE_HZ); // 初始化并使能 SysTick 定时器
24.     if (SysTick_Config(SystemCoreClock/OS_CFG_TICK_RATE_HZ)) // ST3.5.0 库版本
25.     {
26.         /* Capture error */
27.         while (1);
28.     }
29. }

```

1. SysTick_init() 函数

在前面几小节的分析已知, SysTick 中断对于 μ C/OS 内核的任务调度十分重要, 但前面的文件中我们只编写了 SysTick 的中断服务函数, 还没有使能 SysTick 及设置其中断频率。在本文件中, SysTick_init() 函数完成了这些工作。这个函数从本书第 12 章的 SysTick 初始化驱动移植而来, 只是修改了其中 SysTick_Config() 函数的输入参数, 该输入参数表示多少个系统时钟周期 (频率为 72 MHz) 中断一次。在本工程调用时, 它的输入参数为 SystemCoreClock/OS_CFG_TICK_

RATE_HZ, 其中 SystemCoreClock 表示 STM32 的系统时钟频率 72 MHz, 而 OS_CFG_TICK_RATE_HZ 是 μ C/OS 内核在 os_cfg_app.h 文件定义的一个宏:

```
1.  #define OS_CFG_TICK_RATE_HZ          1000u
```

它用于表示 μ C/OS 内核时钟的频率, 在本工程即为 SysTick 中断的频率, 这个宏的值可以由用户修改。若设置得太大, CPU 花费在系统调度的时间过长; 若设置得太小, 则高优先级的就绪任务可能等待时间过长。对于 STM32 来说, 内核时钟频率设置为 1000 Hz 是比较合适的。

以“SystemCoreClock”与“OS_CFG_TICK_RATE_HZ”的比值作为 SysTick_Config() 函数的输入参数, 运算得该参数为 72000, 即使 SysTick 每 72000 个系统时钟周期 (频率为 72 MHz) 中断一次, 即 1 ms, 于是 μ C/OS 内核时钟的频率就被设置为 1000 Hz 了。

2. BSP_Init() 函数

BSP_Init() 函数是封装好的板级初始化函数, 我们通常会把工程中使用到的各种外设初始化函数都包含在该函数中, 如 SysTick 初始化、LED 初始化、串口初始化等。BSP_Init() 函数在 μ C/OS 系统启动前被调用, 完成设备的初始化。

在本工程中, BSP_Init() 函数调用了上面提到的 SysTick 初始化函数 SysTick_init() 和 LED 外设初始化函数 LED_GPIO_Config()。LED_GPIO_Config() 函数是原裸机 LED 工程文件 led.c 中定义的, 在这里不再重复说明。

3. BSP.h 头文件

板级相关的还有 BSP.h 文件, 这个文件主要是 BSP.c 文件中函数的声明, 内容见代码清单 27-16。

代码清单 27-16 BSP.c 文件中函数的声明

```
1. #ifndef __BSP_H
2. #define __BSP_H
3.
4. #include "os_cfg_app.h"
5.
6. void SysTick_init(void);
7. void BSP_Init(void);
8.
9.
10. #endif // __BSP_H
```

27.3.3 创建任务

与应用层相关的内容和任务, 我们通常会把它编写到用户自建的 app.c 文件中, 并附有相应的头文件 app.h。

1. 编写 app.c

本工程建立了一个 LED 流水灯的任务, app.c 文件内容见代码清单 27-17。

代码清单 27-17 LED 流水灯任务

```

1.
2. #include "includes.h"
3.
4. void Task_LED(void *p_arg)
5. {
6.     OS_ERR err;
7.     (void)p_arg;          // 'p_arg' 并没有用到, 防止编译器提示警告
8.
9.     while (1)
10.    {
11.        LED1( ON );          // 点亮 LED1
12.        OSTimeDlyHMSM(0, 0,1,0,OS_OPT_TIME_HMSM_STRICT,&err);
13.        // 做 1 秒的延时
14.        LED1( OFF);          // 关闭 LED1
15.
16.        LED2( ON );
17.        OSTimeDlyHMSM(0, 0,1,0,OS_OPT_TIME_HMSM_STRICT,&err);
18.        LED2( OFF);
19.
20.        LED3( ON );
21.        OSTimeDlyHMSM(0, 0,1,0,OS_OPT_TIME_HMSM_STRICT,&err);
22.        LED3( OFF);
23.    }
24.}
25.

```

这是一个简单的任务示例, 任务函数为 Task_LED(), 它调用了 μ C/OS 的延时函数 OSTimeDlyHMSM() 和 led.h 文件中的宏 LED1 等实现了流水灯的功能。

2. 编写 app.h

还要编写相应的 app.h 头文件, 内容见代码清单 27-18。

代码清单 27-18 app.h 头文件

```

1. #ifndef _APP_H_
2. #define _APP_H_
3.
4. /***** 用户任务声明 *****/
5. void Task_LED(void *p_arg);
6.
7. #endif // _APP_H_

```

3. 编写 main 文件

虽然移植了操作系统, 本工程的 C 语言程序仍然从 main 文件中开始执行, 内容见代码清单 27-19。

代码清单 27-19 μ C/OS 的 main 文件

```

1. #include "includes.h"
2.

```

```

3.
4.
5. static OS_TCB LED_TCB;           // 定义任务控制块
6. static CPU_STK LED_Stk[128];     // 定义任务堆栈
7.
8. /*
9.  * 函数名: main
10. * 描述   : 主函数
11. * 输入    : 无
12. * 输出    : 无
13. */
14. int main(void)
15. {
16.
17.     OS_ERR err;
18.     BSP_Init();                   // 板级初始化
19.     OSInit(&err);                 // 系统初始化
20.     /***** 创建任务 *****/
21.     OSTaskCreate((OS_TCB *) &LED_TCB, // 任务控制块指针
22.
23.                 (CPU_CHAR *) "LED", // 任务名称
24.                 (OS_TASK_PTR) Task_LED, // 任务代码指针
25.                 (void *) 0, // 传递给任务的参数 parg
26.                 (OS_PRIO) 2, // 任务优先级
27.                 (CPU_STK *) &LED_Stk[0], // 任务堆栈基地址
28.                 (CPU_STK_SIZE) 12, // 堆栈剩余警戒线
29.                 (CPU_STK_SIZE) 128, // 堆栈大小
30.                 (OS_MSG_QTY) 0, // 可接收的最大消息队列数
31.                 (OS_TICK) 0, // 时间片轮转时间
32.                 (void *) 0, // 任务控制块扩展信息
33.                 (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), // 任务选项
34.
35.                 (OS_ERR *) &err); // 返回值
36.     OSStart(&err);
37. }
38.
39. /***** (C) COPYRIGHT 2012 WildFire Team *****/
40.

```

我们在 main 函数中调用 BSP_Init() 函数初始化目标板, 调用 OSInit() 初始化 μ C/OS 系统, 调用 OSTaskCreate() 创建任务并使任务指针指向我们在 app.c 文件中定义的 Task_LED 任务函数, 最后调用 OSStart() 开始任务调度。

27.4 配置 μ C/OS-III

完成了修改、编写上述所有 μ C/OS 文件、用户文件后, 重新编译一下工程, 发现还是提示很多错误, 细心一看, 这些错误都是相同的, 见图 27-6。


```
error: #20: identifier "CPU_TS_TMR" is undefined
error: #20: identifier "CPU_TS_TMR" is undefined
error: #20: identifier "CPU_TS_TMR" is undefined
```

图 27-6 时间戳相关功能错误提示

这是一个与 μ C/OS 时间戳功能相关的错误，错误还提示我们需要使能时间戳，添加时间戳的函数。时间戳是 μ C/OS-III 的新增功能，可以用于测量调度锁的时间。在本实验中，我们不需要用到如此高级的功能，添加时间戳相关函数会使这个移植步骤变得更加复杂，于是我们关闭时间戳功能，就可以不编写这些函数了。

事实上，即使在实际应用的工程项目中，我们往往不需要使用 μ C/OS 提供的全部功能，这时我们就会把多余的模块裁剪掉，以减少内核体积，节约宝贵的程序空间，在以后有必要的时候再添加相应的功能。

os_cfg.h 文件是用于配置 μ C/OS 系统功能的，可以通过修改它来达到剪裁系统功能的目的。读者在动手配置 os_cfg.h 文件时，不仅能够从整体了解到 μ C/OS 系统具有哪些功能，还能知道哪些功能对于我们的应用是必要的。

修改 os_cfg.h 文件

与我们移植 FATFS 文件系统和 LwIP 时一样，对 μ C/OS 系统功能的配置也是由一个个宏组成的。一般来说，这一类可移植、扩展性强的代码都会有一个类似的文件，以一系列的宏，配合子功能模块中的条件编译语句实现功能裁剪。 μ C/OS 中这些与功能裁剪相关的宏位于 os_cfg.h 文件中，打开 os_cfg.h 文件，内容见代码清单 27-20。

代码清单 27-20 os_cfg.h 文件

```
1.
2. #ifndef OS_CFG_H
3. #define OS_CFG_H
4.
5.
6. /* ----- 杂项 ----- */
7. #define OS_CFG_APP_HOOKS_EN      1u    /* 钩子函数功能          */
8. #define OS_CFG_ARG_CHK_EN        1u    /* 参数检查功能          */
9. #define OS_CFG_CALLED_FROM_ISR_CHK_EN 1u    /* 中断调用系统函数功能 */
10. #define OS_CFG_DBG_EN            1u    /* debug 功能            */
11. #define OS_CFG_ISR_POST_DEFERRED_EN 1u    /* 关中断或锁调度器保护临界段 */
12.
13. #define OS_CFG_OBJ_TYPE_CHK_EN    1u    /* 对象类型检测功能      */
14. #define OS_CFG_TS_EN              1u    /* 时间戳功能            */
15.
16. #define OS_CFG_PEND_MULTI_EN      1u    /* 多事件等待功能        */
17.
18. #define OS_CFG_PRIO_MAX           64u    /* 最大优先级数量        */
19.
20. #define OS_CFG_SCHED_LOCK_TIME_MEAS_EN 1u    /* 调度锁时间检测功能    */
21. #define OS_CFG_SCHED_ROUND_ROBIN_EN 1u    /* 时间片轮转功能        */
```

```

22. #define OS_CFG_STK_SIZE_MIN          64u    /* 任务堆栈的最小值 */
23.
24. /* ----- 事件管理 ----- */
25. #define OS_CFG_FLAG_EN                1u     /* 事件标志功能 */
26. #define OS_CFG_FLAG_DEL_EN            1u     /* 事件标志删除功能 */
27. #define OS_CFG_FLAG_MODE_CLR_EN       1u     /* 清除等待事件标志功能 */
28.
29. #define OS_CFG_FLAG_PEND_ABORT_EN      1u     /* 终止标志事件等待功能 */
30.
31.
32.
33. /* ----- 内存管理 ----- */
34. #define OS_CFG_MEM_EN                  1u     /* 内存管理功能 */
35.
36.
37. /* ----- 互斥信号量 ----- */
38. #define OS_CFG_MUTEX_EN                1u     /* 互斥信号量功能 */
39. #define OS_CFG_MUTEX_DEL_EN            1u     /* 互斥信号量删除功能 */
40.
41. #define OS_CFG_MUTEX_PEND_ABORT_EN     1u     /* 终止互斥信号量等待功能 */
42.
43.
44.
45. /* ----- 消息队列 ----- */
46. #define OS_CFG_Q_EN                    1u     /* 消息队列功能 */
47. #define OS_CFG_Q_DEL_EN                1u     /* 队列删除功能 */
48. #define OS_CFG_Q_FLUSH_EN              1u     /* 消息队列刷新功能 */
49. #define OS_CFG_Q_PEND_ABORT_EN         1u     /* 终止队列等待功能 */
50.
51.
52. /* ----- 信号量 ----- */
53. #define OS_CFG_SEM_EN                  1u     /* 信号量功能 */
54. #define OS_CFG_SEM_DEL_EN              1u     /* 信号量删除功能 */
55. #define OS_CFG_SEM_PEND_ABORT_EN       1u     /* 终止等待信号量功能 */
56. #define OS_CFG_SEM_SET_EN              1u     /* 信号量设置功能 */
57.
58.
59. /* ----- 任务管理 ----- */
60. #define OS_CFG_STAT_TASK_EN            1u     /* 统计任务功能 */
61. #define OS_CFG_STAT_TASK_STK_CHK_EN    1u     /* 统计任务堆栈检测功能 */
62.
63. #define OS_CFG_TASK_CHANGE_PRIO_EN     1u     /* 优先级调节功能 */
64. #define OS_CFG_TASK_DEL_EN             1u     /* 任务删除功能 */
65. #define OS_CFG_TASK_Q_EN               1u     /* 任务消息队列功能 */
66. #define OS_CFG_TASK_Q_PEND_ABORT_EN    1u     /* 终止任务消息队列等待功能 */
67.
68. #define OS_CFG_TASK_PROFILE_EN          1u     /* 任务详细状态功能 */
69. #define OS_CFG_TASK_REG_TBL_SIZE       1u     /* 任务特殊功能寄存器 */
70. #define OS_CFG_TASK_SEM_PEND_ABORT_EN  1u     /* 终止任务信号量等待功能 */
71. #define OS_CFG_TASK_SUSPEND_EN         1u     /* 任务暂时终止和恢复功能 */
72.
73.

```

```

74.
75./* ----- 时间管理 ----- */
76.#define OS_CFG_TIME_DLY_HMSM_EN      1u    /* 时间延时函数功能 */
77.#define OS_CFG_TIME_DLY_RESUME_EN    1u    /* 时间延时取消功能 */
78.
79.
80./* ----- 定时器管理 ----- */
81.#define OS_CFG_TMR_EN                1u    /* 定时器功能 */
82.#define OS_CFG_TMR_DEL_EN            1u    /* 定时器删除功能 */
83.
84.#endif

```

这个文件把宏分成了多个小组，有事件管理、内存管理、互斥信号量、信号量、消息队列、任务管理、时间管理和定时器管理，这些是 μ C/OS 的主要功能模块，还包括一些杂项。配置 μ C/OS 内核时，我们只要把相应功能的宏改成 0 即可，默认时 μ C/OS 的这些宏都配置为 1，表示使能这些功能。

在本次实验中，我们必须关闭的宏是杂项组里的时间戳 OS_CFG_TS_EN 和调度锁时间检测功能 OS_CFG_SCHED_LOCK_TIME_MEAS_EN，如果读者想使用这个功能，需要添加支持时间戳的函数。修改后见代码清单 27-21。

代码清单 27-21 修改后的相关宏

```

1.  /* ----- 杂项 ----- */
2.  #define OS_CFG_APP_HOOKS_EN      1u    /* 钩子函数功能 */
3.
4.  #define OS_CFG_ARG_CHK_EN        1u    /* 参数检查功能 */
5.
6.  #define OS_CFG_CALLED_FROM_ISR_CHK_EN  1u    /* 中断调用系统函数功能 */
7.
8.  #define OS_CFG_DBG_EN            1u    /* debug 功能 */
9.
10. #define OS_CFG_ISR_POST_DEFERRED_EN  1u    /* 关中断或锁调度器保护临界段 */
11.
12. #define OS_CFG_OBJ_TYPE_CHK_EN    1u    /* 对象类型检测功能 */
13.
14. #define OS_CFG_TS_EN              0u    /* 修改 ** 关闭 ** 时间戳功能 *** */
15.
16.
17. #define OS_CFG_PEND_MULTI_EN      1u    /* 多事件等待功能 */
18.
19.
20. #define OS_CFG_PRIO_MAX           64u    /* 最大优先级数量 */
21.
22.
23. #define OS_CFG_SCHED_LOCK_TIME_MEAS_EN  0u    /* 修改 * 关闭 * 调度锁时间检测 */
24.
25. #define OS_CFG_SCHED_ROUND_ROBIN_EN  1u    /* 时间片轮转功能 */
26.
27. #define OS_CFG_STK_SIZE_MIN       64u    /* 任务堆栈的最小值 */
28.

```


修改这两个宏之后，重新编译工程，编译结果为“0 错误 0 警告”，我们把程序代码下载到开发板，发现实验现象与裸机的 LED 流水灯无异。至此，从功能上说，我们已经完成了 $\mu\text{C}/\text{OS-III}$ 在配套 STM32 开发板上的移植。

但是，我们再看看这个 $\mu\text{C}/\text{OS}+\text{LED}$ 的工程编译的输出结果，里面有以下语句：

```
Program Size: Code=32480 RO-data=864 RW-data=240 ZI-data=8032 (uC/OS+LED 工程)
```

我们知道，该语句中数字的单位为 Byte，其中 Code 是代码占用的空间，RO-data 是 Read Only 只读常量的大小，如 const 型，RW-data 是 Read Write 初始化了的可读写变量的大小，ZI-data 是 Zero Initialize 没有初始化的可读写变量的大小。ZI-data 不会被算作代码，因为它不会被初始化。

简单地说就是在烧写的时候是 Flash 中被占用的空间为：Code+RO Data+RW Data，程序运行的时候，芯片内部 RAM 使用的空间为：RW Data + ZI Data。

我们使用的 STM32f103VET6 型号芯片的 Flash 大小为 512 KB，SRAM 大小为 64 KB，虽然可以毫无顾忌地运行我们这个 $\mu\text{C}/\text{OS}+\text{LED}$ 工程代码，但我们打开一下前面实现同样功能的裸机 LED 流水灯代码，编译看到以下语句：

```
Program Size: Code=2540 RO-data=336 RW-data=40 ZI-data=1024 (裸机 LED 工程)
```

对比之下发现，使用 $\mu\text{C}/\text{OS}$ 系统，比裸机多了很多资源消耗，所以，我们可以修改 os_cfg.h 文件，对 $\mu\text{C}/\text{OS}$ 内核功能做进一步裁剪，如关闭钩子函数、信号量、消息队列等功能，经过简单的配置后， $\mu\text{C}/\text{OS}+\text{LED}$ 的工程消耗的资源变为：

```
Program Size: Code=15828 RO-data=864 RW-data=156 ZI-data=5220 (裁剪后的 uC/OS+LED 工程)
```

我们还没有对 $\mu\text{C}/\text{OS}$ 做最苛刻的裁剪，对 Flash 的消耗已经几乎降低了一半，而在实验现象上没有任何区别。所以在实际项目中，针对应用来裁剪 $\mu\text{C}/\text{OS}$ 内核是非常必要的。读者可以亲自尝试裁剪一下，看看结果如何。



第 28 章

运行多任务

使用操作系统的一大特点是它能够运行多任务，在第 27 章的工程中主要介绍 $\mu\text{C}/\text{OS}$ 的移植步骤，只创建了一个 LED 任务，丝毫没有体现出使用操作系统的优势。在本章中，通过另一个 LED 灯的例子，为读者展示如何使用 $\mu\text{C}/\text{OS}$ 多任务进行编程。在这里提到的单任务、多任务是对于用户任务而言的。上一章的实验中，实际上运行了 3 个任务，一个是用户创建的 LED 任务，另外两个是系统自带的空闲任务和统计任务。

28.1 创建用户任务

开始本章实验前，把第 27 章移植好的 $\mu\text{C}/\text{OS-III}$ 工程复制一份，在它的基础上修改即可，需要重新编写的是与应用层相关的 main.c、app.c、app.h 文件。

在移植好操作系统的平台上，最重要的就是编写用户任务。在多任务编程中，一般在 main 函数中创建一个初始任务 A，再通过任务 A 创建其他任务 B、C……本实验中，我们一共建立了四个用户任务，在初始任务 Task_Start 中建立 Task_LED1、Task_LED2 和 Task_LED3 控制三盏 LED 灯，分别以 100 ms、200 ms 和 300 ms 的时间间隔闪烁。

Task_Start 用户任务是在 main 函数中创建的，main 文件的内容见代码清单 28-1。

代码清单 28-1 多任务下的 main 文件

```
1.
2. #include "includes.h"
3.
4.
5.
6. OS_TCB StartUp_TCB;           // 定义任务控制块
7. CPU_STK StartUp_Stk[STARTUP_TASK_STK_SIZE]; // 定义任务堆栈
8.
9. /*
10. * 函数名: main
11. * 描述   : 主函数
12. * 输入   : 无
13. * 输出   : 无
14. */
15. int main(void)
16. {
```

```

17.
18.     OS_ERR err;
19.     BSP_Init();                                // 板级初始化
20.     OSInit(&err);                              // 系统初始化
21.     /* 创建任务 */
22.     OSTaskCreate((OS_TCB      *) &StartUp_TCB,    //1 任务控制块指针
23.
24.                 (CPU_CHAR    *) "StartUp",        //2 任务名称
25.                 (OS_TASK_PTR ) Task_Start,        //3 任务代码指针
26.                 (void        *) 0,                //4 传递给任务的参数 parg
27.
28.                 (OS_PRIO      ) STARTUP_TASK_PRIO, //5 任务优先级
29.                 (CPU_STK      *) &StartUp_Stk[0],  //6 任务堆栈基地址
30.                 (CPU_STK_SIZE) STARTUP_TASK_STK_SIZE/10, //7 堆栈警戒线
31.
32.                 (CPU_STK_SIZE) STARTUP_TASK_STK_SIZE, //8 堆栈大小
33.                 (OS_MSG_QTY   ) 0,                //9 可接收的最大消息队列数
34.
35.                 (OS_TICK      ) 0,                //10 时间片轮转时间
36.                 (void         *) 0,                //11 任务控制块扩展信息
37.                 (OS_OPT        ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), //12
           任务选项
38.                 (OS_ERR       *) &err);            //13 返回值
39.
40.     OSStart(&err);
41.
42.
43. }
44. /**** (C) COPYRIGHT 2012 WildFire Team *****/

```

main 文件中用到的一些宏在 app.h 文件中定义, 该文件主要包括设置任务的优先级、栈大小和函数声明, 内容见代码清单 28-2。

代码清单 28-2 app.h 文件中的宏

```

1.
2. #ifndef _APP_H_
3. #define _APP_H_
4.
5.
6. /***** 设置任务优先级 *****/
7. #define STARTUP_TASK_PRIO      3
8. #define TASK_LED1_PRIO        4
9. #define TASK_LED2_PRIO        5
10. #define TASK_LED3_PRIO        6
11. /***** 设置栈大小 (单位为 OS_STK) *****/
12. #define STARTUP_TASK_STK_SIZE  80
13. #define TASK_LED1_STK_SIZE     80
14. #define TASK_LED2_STK_SIZE     80
15. #define TASK_LED3_STK_SIZE     80
16.

```



```
17. /***** 用户任务声明 *****/
18.
19. void Task_Start(void *p_arg);
20. void Task_LED1(void *p_arg);
21. void Task_LED2(void *p_arg);
22. void Task_LED3(void *p_arg);
23.
24. #endif // _APP_H_
```

回到 main 函数中分析，与单任务中的流程无异，main 函数先调用了 BSP_Init() 函数进行板级的初始化，接着调用系统函数 OSInit() 初始化 μ C/OS 系统。在第三步中，调用函数 OSTaskCreate() 创建了 Task_Start 任务，之后在第 40 行中调用系统函数 OSStart() 函数使 μ C/OS 开始任务管理，首先进入并运行 Task_Start 任务。

读者第一次看到系统函数 OSTaskCreate() 会感觉很复杂，因为它一共有 13 个输入参数，这些参数作用见表 28-1。

表 28-1 OSTaskCreate 的输入参数

参数编号	参数类型	参数名称	功能	本实验的实参
1	OS_TCB *	任务控制块指针	μ C/OS 内核通过任务控制块管理任务，每个任务都有相应的任务控制块	指向在 main 文件头定义的 StartUp_TCB
2	CPU_CHAR *	任务名称	一组自定义的字符串，主要在 PROBE 工具调试时显示用	任务名称为 StartUp
3	OS_TASK_PTR	任务代码指针	是一个函数指针，指向本任务的函数代码	指向 Task_Start 任务代码
4	void *	传递给任务的参数	任务通过本指针获取参数，本指针可指向变量、复杂的结构体	设置为 0，空指针，不使用传递参数
5	OS_PRIO	任务优先级	μ C/OS 内核根据优先级来进行任务调度，该数值越大，优先级越低	优先级是在 app.h 自定义的宏：STARTUP_TASK_PRIO 数值为 3
6	CPU_STK *	任务堆栈基地址	每个任务都要分配一个堆栈，用于保存任务切换时的上下文	基地址设置为 &StartUp_Stk[0]，StartUp_Stk 是在 main 文件头定义的堆栈
7	CPU_STK_SIZE	堆栈剩余警戒线	当任务堆栈增长到本位置时，限制其增长，可以用于防止栈溢出	设置为 STARTUP_TASK_STK_SIZE/10，即为本任务堆栈大小的 1/10
8	CPU_STK_SIZE	堆栈大小	堆栈的大小取决于任务中的局部变量、要保存的 CPU 寄存器大小、中断服务函数中变量总和	设置为 STARTUP_TASK_STK_SIZE，其值为 80，即堆栈大小为 $80 \times 4=320$ 字节
9	OS_MSG_QTY	最大消息队列数	消息队列用于多任务间的信息交流	本任务不使用消息队列，设置参数为 0

(续)

参数编号	参 数 类 型	参 数 名 称	功 能	本实验的实参
10	OS_TICK	时间片轮转时间	μC/OS-III 支持当有多个同优先级的任务时，采用时间片调度	本任务不使用时间片，设置为 0
11	void *	任务扩展信息	本指针指向任务控制块的扩展信息，这些扩展信息可由用户自定义	本实验不使用任务控制块扩展信息
12	OS_OPT	任务选项	可以用于设置一些任务相关操作	本任务设置为 OS_OPT_TASK_STK_CHK 和 OS_OPT_TASK_STK_CLR，分别表示开启任务堆栈检查和使用前对任务堆栈清空
13	OS_ERR	运行信息	运行本函数后，检查本参数，可获知函数的运行信息、错误号等	本参数在 main 函数中定义为 err，调试时可在本函数后进行参数检查

28.2 编写用户代码

根据上一节的分析，在 main 函数创建了 Task_Start 的用户任务，这时还未进入该任务代码。当执行了 OSStart() 函数之后，就跳转到 Task_Start 任务代码中执行了。Task_Start 的任务代码定义在 app.c 文件中，文件内容见代码清单 28-3。

代码清单 28-3 Task_Start 的任务代码

```
1.
2. #include "includes.h"
3.
4. extern OS_TCB StartUp_TCB; // 任务堆栈
5.
6. static OS_TCB LED1_TCB; // 定义任务控制块
7. static CPU_STK LED1_Stk[TASK_LED1_STK_SIZE]; // 定义任务堆栈
8.
9. static OS_TCB LED2_TCB; // 定义任务控制块
10. static CPU_STK LED2_Stk[TASK_LED2_STK_SIZE]; // 定义任务堆栈
11.
12. static OS_TCB LED3_TCB; // 定义任务控制块
13. static CPU_STK LED3_Stk[TASK_LED3_STK_SIZE]; // 定义任务堆栈
14.

1. /*
2.  * 函数名：Task_Start
3.  * 描述   ： 启动任务，
4.             优先级为 3，
5.             创建 LED1、LED2 和 LED3 的任务
6.  * 输入   ：无
7.  * 输出   ：无
8.  */
9. void Task_Start(void *p_arg)
```

```

10. {
11.     OS_ERR err;
12.     (void)p_arg;          // 'p_arg' 并没有用到, 防止编译器提示警告
13.
14.
15.     /***** 创建任务 LED1 *****/
16.     OSTaskCreate((OS_TCB *) &LED1_TCB,          // 任务控制块指针
17.
18.                 (CPU_CHAR *) "LED1",            // 任务名称
19.                 (OS_TASK_PTR) Task_LED1,         // 任务代码指针
20.                 (void *) 0,                      // 传递给任务的参数 parg
21.
22.                 (OS_PRIO) TASK_LED1_PRIO,        // 任务优先级
23.                 (CPU_STK *) &LED1_Stk[0],        // 任务堆栈基地址
24.                 (CPU_STK_SIZE) TASK_LED1_STK_SIZE/10, // 堆栈剩余警戒线
25.
26.                 (CPU_STK_SIZE) TASK_LED1_STK_SIZE, // 堆栈大小
27.
28.                 (OS_MSG_QTY) 0,                  // 可接收的最大消息队列数
29.                 (OS_TICK) 0,                    // 时间片轮转时间
30.                 (void *) 0,                      // 任务控制块扩展信息
31.                 (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), // 任务选项
32.                 (OS_ERR *) &err);               // 返回值
33.
34.     /***** 创建任务 LED2 *****/
35.     OSTaskCreate((OS_TCB *) &LED2_TCB,          // 任务控制块指针
36.
37.                 (CPU_CHAR *) "LED2",            // 任务名称
38.                 (OS_TASK_PTR) Task_LED2,         // 任务代码指针
39.                 (void *) 0,                      // 传递给任务的参数 parg
40.                 (OS_PRIO) TASK_LED2_PRIO,        // 任务优先级
41.                 (CPU_STK *) &LED2_Stk[0],        // 任务堆栈基地址
42.                 (CPU_STK_SIZE) TASK_LED2_STK_SIZE/10, // 堆栈剩余警戒线
43.
44.                 (CPU_STK_SIZE) TASK_LED2_STK_SIZE, // 堆栈大小
45.                 (OS_MSG_QTY) 0,                  // 可接收的最大消息队列数
46.
47.                 (OS_TICK) 0,                    // 时间片轮转时间
48.                 (void *) 0,                      // 任务控制块扩展信息
49.                 (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), // 任务选项
50.                 (OS_ERR *) &err);               // 返回值
51.
52.     /***** 创建任务 LED3 *****/
53.     OSTaskCreate((OS_TCB *) &LED3_TCB,          // 任务控制块指针
54.
55.                 (CPU_CHAR *) "LED3",            // 任务名称
56.                 (OS_TASK_PTR) Task_LED3,         // 任务代码指针
57.                 (void *) 0,                      // 传递给任务的参数 parg
58.
59.                 (OS_PRIO) TASK_LED3_PRIO,        // 任务优先级
60.                 (CPU_STK *) &LED3_Stk[0],        // 任务堆栈基地址
61.                 (CPU_STK_SIZE) TASK_LED3_STK_SIZE/10, // 堆栈剩余警戒线
62.
63.                 (CPU_STK_SIZE) TASK_LED3_STK_SIZE, // 堆栈大小
64.                 (OS_MSG_QTY) 0,                  // 可接收的最大消息队列数
65.
66.                 (OS_TICK) 0,                    // 时间片轮转时间

```



```

67.          (void      *)0,          // 任务控制块扩展信息
68.          (OS_OPT     )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), // 任务选项
69.          (OS_ERR      *)&err);     // 返回值
70.
71.          OSTaskDel(&StartUp_TCB,&err); // 任务删除自己
72.
73.}
74.
15.

1. /*
2.  * 函数名: Task_LED1
3.  * 描述   : LED 任务 1,
4.             优先级为 4,
5.             以 100ms 的间隔点亮、关闭 LED1
6.  * 输入   : 无
7.  * 输出   : 无
8.  */
9. void Task_LED1(void *p_arg)
10.{
11.    OS_ERR err;
12.    (void)p_arg;
13.
14.    while (1)
15.    {
16.        LED1( ON );
17.        OSTimeDlyHMSM(0, 0,0,100,OS_OPT_TIME_HMSM_STRICT,&err);
18.        // 延时阻塞 100ms
19.        LED1( OFF);
20.        OSTimeDlyHMSM(0, 0,0,100,OS_OPT_TIME_HMSM_STRICT,&err);
21.
22.    }
23.}
24.

1. /*
2.  * 函数名: Task_LED2
3.  * 描述   : LED 任务 2,
4.             优先级为 5,
5.             以 200ms 的间隔点亮、关闭 LED2
6.  * 输入   : 无
7.  * 输出   : 无
8.  */
9. void Task_LED2(void *p_arg)
10.{
11.    OS_ERR err;
12.    (void)p_arg;
13.
14.    while (1)
15.    {
16.        LED2( ON );
17.        OSTimeDlyHMSM(0, 0,0,200,OS_OPT_TIME_HMSM_STRICT,&err);
18.        // 延时阻塞 200ms

```

```
19.         LED2( OFF);
20.         OSTimeDlyHMSM(0, 0,0,200,OS_OPT_TIME_HMSM_STRICT,&err);
21.     }
22.}
23.

1. /*
2.  * 函数名 : Task_LED3
3.  * 描述   :   LED 任务 3,
4.             优先级为 6,
5.             以 300ms 的间隔点亮、关闭 LED3
6.  * 输入   : 无
7.  * 输出   : 无
8.  */
9. void Task_LED3(void *p_arg)
10.{
11.    OS_ERR err;
12.    (void)p_arg;
13.
14.    while (1)
15.    {
16.        LED3( ON );
17.        OSTimeDlyHMSM(0, 0,0,300,OS_OPT_TIME_HMSM_STRICT,&err);
18.        // 延时阻塞 300ms
19.        LED3( OFF);
20.        OSTimeDlyHMSM(0, 0,0,300,OS_OPT_TIME_HMSM_STRICT,&err);
21.    }
22.}
23./***** (C) COPYRIGHT 2012 WildFire Team *****END OF FILE*****/
```

在 Task_Start 任务中，又创建了 3 个类似的 LED 任务，创建完这 3 个任务后，Task_Start 任务删除了自己，即不再运行。其他 3 个任务分别控制一盏 LED 灯以不同的频率闪烁，各任务描述见表 28-2。

表 28-2 任务描述

任 务	优 先 级	功 能 描 述
Task_Start	3	创建 3 个 LED 任务
Task_LED1	4	LED1 闪烁延时 100 ms
Task_LED2	5	LED2 闪烁延时 200 ms
Task_LED3	6	LED3 闪烁延时 300 ms

读者阅读 LED 任务的代码时，会发现它的延时既不是使用粗暴的 for 循环延时，也不是轮询定时器标志，而是调用了一个系统函数 OSTimeDlyHMSM()。该函数功能是阻塞任务一段时间，输入的参数分别为：时、分、秒、毫秒、操作选项和错误信息。

28.3 任务执行流程

从任务代码中可以分析出每个任务的执行流程：打开 LED 灯 → 延时阻塞任务向下执行 → 关闭 LED 灯 → 再延时阻塞任务 → 再打开 LED 灯，如此循环。那么，各任务之间的执行关系是怎

样的呢？如各任务的执行次序、执行时间等，见图 28-1。

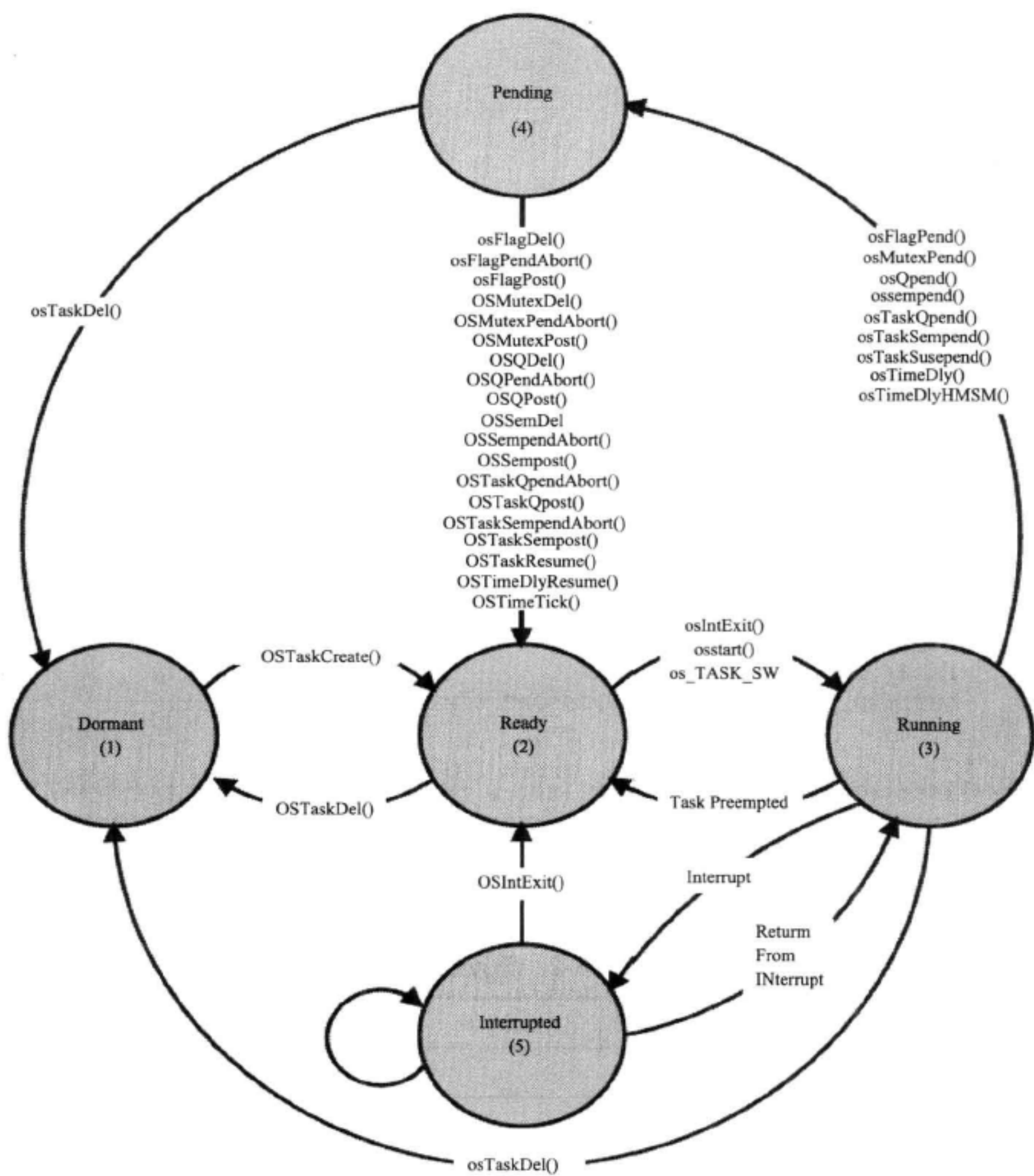


图 28-1 任务状态切换

在 $\mu\text{C}/\text{OS}$ 系统中，任务共分为 5 种状态，各个状态之间通过箭头中的函数，可以互相转换，本实验中各任务状态转换解说如下：

- 休眠态 (Dormant)：在任务未被 `OSTaskCreate()` 创建时，任务是处于休眠状态的，处于休眠状态的任务无法得到 CPU 运行。处于其他任务状态的任务可以通过 `OSTaskDel()` 删除自己，进入休眠态，通常对只运行一次的任务在运行完后删除自己，这样可释放该任务占据的内存空间。本实验中，`Task_Start` 创建完其他 3 个任务后，删除了自己，进入休眠态，不再获得 CPU。

- 就绪态 (Ready) : 就绪态的任务根据优先级排列在系统的就绪任务列表中, 处于就绪态的任务等待获得 CPU 的使用权, CPU 的使用权由系统的调度器根据任务的优先级分配。

本实验刚开始时, 所有 4 个任务都处于休眠态, 使用函数 `OSStart()` 开始任务调度后, 首先被 `OSTaskCreate()` 函数创建的 `Task_Start` 任务得到 CPU。接着 CPU 运行 `Task_Start` 任务代码, 创建其他 3 个 LED 任务。每创建完成一个任务, 相应的任务立即进入就绪态, 但由于这 3 个 LED 任务的优先级都比 `Task_Start` 的低, 所以在 `Task_Start` 运行结束前, 这 3 个 LED 任务一直在等待 CPU。

- 运行态 (Running) : 运行态的任务获得了 CPU, 并正在执行任务代码, 在 STM32 这样的单核 CPU 中, 一个时刻只能有一个任务处于运行态。

本实验在 `Task_Start` 任务删除自己后, 启动了任务调度, 这时 3 个 LED 任务都处于就绪态, 调度器根据优先级, 使最高优先级的 `Task_LED1` 任务得到运行, 该任务点亮 LED1 后, 紧接着调用 `OSTimeDlyHMSM()` 函数延时, 使自己进入挂起状态, 挂起状态的任务失去 CPU 的占有权, 再次由调度器分配 CPU。

- 挂起态 (Pending) : 挂起态的任务被放置在挂起任务列表中, 表明任务在等待某些事件的发生。挂起态的任务不占用 CPU。当事件发生时, 任务会从挂起态进入就绪态, 并且启动调度器, 若该任务的优先级高于当前运行的任务, 当前运行的任务会被抢占 (被放回就绪列表)。也就是说, 如果新的任务优先级最高, 那么它就会被立即运行。

本实验中 LED1 因为调用 `OSTimeDlyHMSM()` 延时进入挂起态, 等待延时结束, 放弃 CPU。此时的就绪任务列表中有 `Task_LED2` 和 `Task_LED3` 任务, 调度器把 CPU 分配给优先级较高的 `Task_LED2` 运行, 点亮 LED2 后, 该任务也进入挂起态, 最低优先级的 `Task_LED3` 因此得到了 CPU, 点亮 LED3, 然后进入挂起态。此时 3 个 LED 任务都处于挂起态, CPU 因为无事可做, 运行 $\mu\text{C}/\text{OS}$ 系统的空闲任务, 这个任务是用于统计 CPU 的使用率的, 若 CPU 在一个时间段内, 运行空闲任务的时间越长, 则表示 CPU 使用率越低。

- 被中断 (Interrupted) : 正在运行的任务会被中断, 失去 CPU, 中断结束后在中断中的 `OSIntExit()` 会启动任务调度, 若经过中断后有更高优先级的任务进入就绪态, 那么中断前正在运行的任务则被新的任务剥夺 CPU 占有权。若经过中断后该任务还是最高优先级, 则返回到该任务中运行。

本实验中, `SysTick` 是唯一的中断, 它每 1 ms 中断一次, 在中断中把各个 LED 任务的延时时间值减 1, 退出中断前启动任务调度, 若中断结束后没有就绪的用户任务, 则继续运行空闲任务。从 LED1 任务调用 `OSTimeDlyHMSM()` 开始, 经过 100 ms (即进入 100 次 `SysTick` 中断), LED1 任务重新进入就绪态, 继而得到 CPU, 执行后面关闭 LED1 的任务代码, 再进入挂起态。LED2、LED3 任务类似, 在 200 ms、300 ms 后重新运行, 关闭 LED 灯, 进入挂起态, 如此循环。

理解了本实验的任务状态转换后, 读者会发现这 3 个 LED 灯的任务几乎是并行的, 这为编写复杂的应用程序提供了方便。在更高级的系统应用中, 会把一个应用分割为多个任务, 因而使用到信号量、消息队列等系统功能, 编程时还需要注意临界区管理、可重入函数的编写, 这是使用操作系统与裸机编程的区别, 而这些区别产生的根源都是因为操作系统的多任务特性。

参 考 文 献

- [1] STMicroelectronics Ltd. UM0427 User manual: STM32F10x Standard Peripherals Firmware Library[J/OL]. <http://www.st.com>. 2009.
- [2] Joseph Yiu. ARM Cortex-M3 权威指南 [M]. 宋岩, 译. 北京: 北京航空航天大学出版社, 2009.
- [3] STMicroelectronics Ltd. RM0008 Reference manual: STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs Rev10[J/OL]. <http://www.st.com>. 2010.
- [4] STMicroelectronics Ltd. STM32F103xC, STM32F103xD STM32F103xE datasheet. Rev5[J/OL]. <http://www.st.com>. 2009.
- [5] Stephen Prata. C Primer Plus 中文版 [M]. 云巅工作室, 译. 5 版. 北京: 人民邮电出版社, 2005.
- [6] ARM Ltd. Cortex-M3 Revision: Technical Reference Manual. Rev r1p1[J/OL]. <http://www.arm.com>. 2006.
- [7] Technical Committee SD Card Association. SD Specifications Part 1 Physical Layer Simplified Specification. Rev2 [J/OL]. <http://www.sdcard.org>. 2006.
- [8] BOSCH Ltd. CAN Specification Rev 2 [J/OL]. <http://www.bosch.com>. 1991.
- [9] Compaq Computer Corporation. Universal Serial Bus Specification. Rev 2 [J/OL]. <http://www.usb.org>. 2000.
- [10] Andrew S Tanenbaum. 等, 计算机网络 [M]. 5 版, 影印版. 北京: 机械工业出版社, 2012.
- [11] Jean J Labrosse. μ C/OS-III – The Real-Time Kernel [J/OL]. <http://micrium.com/books/ucosiii>. 2011.

横亘在STM32初学者面前最大的一个问题就是选择用库的方式开发，还是用寄存器的方式开发。对此，曾在网上引发了一场大讨论，大家众说纷纭，仁者见仁，智者见智。从深层次看，这个问题其实并不是一个对开发方式进行选择的问题。

在ARM还没有打算进入MCU领域的时候，大多数单片机工程师的学习是从8位单片机开始的，而且以51单片机居多。由于51单片机的寄存器非常少，很容易记忆，那么使用操作寄存器的方式进行单片机程序开发不但容易，而且高效。随着ARM Cortex-M内核和STM32的发布，这个局面就被打破了。

在苹果公司旗舰产品iPhone和iPad的带领下，以触控屏应用为代表的人机交互方式席卷了全球，当然，中国也未能幸免。在中国，连最保守的工业领域也开始接受这种先进的人机交互方式。然而，这时候，工业界普遍应用的微控制器仍然是51单片机，它不能满足工业界对新型人机交互体验的需求。STM32的出现填补了工业界这个缺憾，从它发布开始，STM32迅速蹿红，成为MCU领域一颗耀眼的新星。原来熟悉了51单片机的工程师不得不开始学习新的基于Cortex-M架构的MCU。然而，他们很快发现一个问题，对32位MCU寄存器的操作是一个极富挑战的事情，因为寄存器太多了。为了解决这个问题，ST公司组织自己的工程技术人员开发了一个针对STM32的固件库。通过对寄存器的封装，开发人员只须简单地调用固件库函数，就能够完成复杂的开发任务。根据ST公司的统计，有大约一半的初学者选择了库开发的方式。他们并没有回过头去学习复杂的寄存器操作，而是选择直接从固件库入手，很快掌握了STM32的开发技术。关于寄存器的开发方式好，还是固件库的开发方式好的争议就在这两个人群中蔓延开了。

那么究竟哪种方式好呢？这个问题很难回答。如果你本身就是一个完全没有接触过单片机的“白丁”，那么建议你还是不要冒险体验寄存器的开发方式了。你可以在尝试掌握了库开发的方式之后再重新体验一下寄存器的开发方式。如果你已经掌握了某个单片机的开发，那么沿用原来的思路是一个省脑细胞的决定。但是如果你的志向不是仅仅屈居一个单片机码农，你若有更高的理想和抱负，建议你用批判和分析的目光学习一下固件库的开发方式。因为当你迈向更高阶的ARM Linux嵌入式开发的时候，你会发现这个功课是值得做的，它给了你一把打开嵌入式Linux开发大门的钥匙——自顶向下的开发方式和思想。

STM32嵌入式开发学习曲线



附光盘

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604

数字阅读：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com



上架指导：嵌入式

ISBN 978-7-111-42637-0



9 787111 426370 >

定价：69.00元（附光盘）