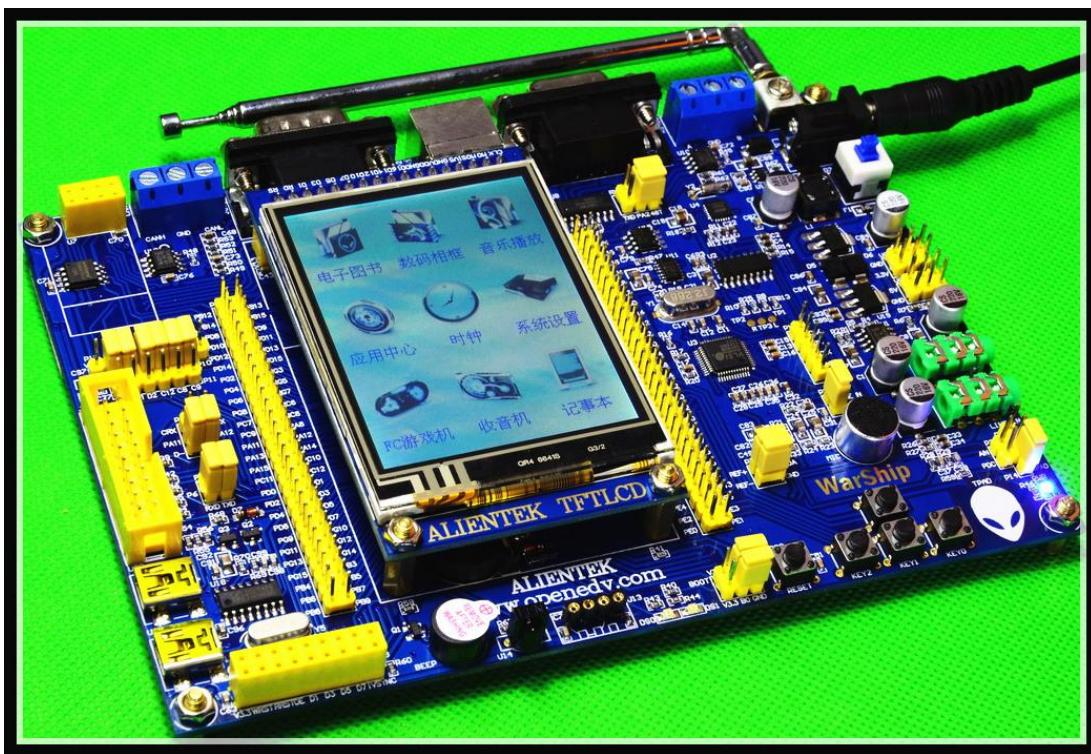




STM32 开发指南

V1.3

-ALIENTEK 战舰 STM32 开发板库函数教程



官方店铺 1: <http://shop62057469.taobao.com>

官方店铺 2: <http://shop62103354.taobao.com>

技术论坛: www.openedv.com



内容简介	I
前言	2
第一篇 硬件篇	4
第一章 实验平台简介	5
1.1 ALIENTEK 战舰 STM32 开发板资源初探	5
1.2 ALIENTEK 战舰 STM32 开发板资源说明	7
1.2.1 硬件资源说明	7
1.2.2 软件资源说明	12
第二章 实验平台硬件资源详解	14
2.1 开发板原理图详解	14
2.1.1 MCU	14
2.1.2 引出 IO 口	16
2.1.3 USB 串口/串口 1 选择接口	16
2.1.4 JTAG/SWD	17
2.1.5 SRAM	17
2.1.6 LCD/OLED 模块接口	18
2.1.7 复位电路	19
2.1.8 启动模式设置接口	19
2.1.9 RS232 串口	20
2.1.10 RS485 接口	20
2.1.11 CAN/USB 接口	21
2.1.12 EEPROM	21
2.1.13 游戏手柄接口	22
2.1.14 SPI FLASH	22
2.1.15 3D 加速度传感器	23
2.1.16 温湿度传感器接口	23
2.1.17 红外接收头	23
2.1.18 无线模块接口	24
2.1.19 LED	24
2.1.20 按键	25
2.1.21 TPAD 电容触摸按键	25
2.1.22 PS/2 接口	26



2.1.23 OLED/摄像头模块接口	26
2.1.24 有源蜂鸣器	27
2.1.25 SD 卡/以太网模块接口	28
2.1.26 多功能端口	29
2.1.27 音频选择	30
2.1.28 FM 收发	31
2.1.29 音频输出	31
2.1.30 音频编解码	32
2.1.31 电源	32
2.1.33 USB 串口	34
2.2 开发板使用注意事项	34
第二篇 软件篇	36
第三章 RVMDK 软件入门	37
3.1 STM32 官方固件库简介	37
3.1.1 库开发与寄存器开发的关系	37
3.1.2 STM32 固件库与 CMSIS 标准讲解	38
3.1.3 STM32 官方库包介绍	39
3.1.3.1 文件夹介绍:	40
3.1.3.2 关键文件介绍:	41
3.2 RVMDK3.80A 简介	42
3.3 新建基于固件库的 RVMDK 工程模板	43
3.3.1 MDK3.8a 安装步骤	43
3.3.2 添加 License Key	44
3.3.3 新建工程模板	46
3.4 MDK 下的程序下载与调试	65
3.4.1 STM32 软件仿真	65
3.4.2 STM32 程序下载	71
3.4.3 STM32 硬件调试	76
3.5 RVMDK 使用技巧	80
3.5.1 文本美化	80
3.5.2 代码编辑技巧	84
3.5.3 其他小技巧	89
3.5.4 调试技巧	90

第四章 STM32 开发基础知识入门	94
4.1 MDK 下 C 语言基础复习	94
4.1.1 位操作	94
4.1.2 define 宏定义	95
4.1.3 ifdef 条件编译	95
4.1.4 extern 变量申明	96
4.1.5 typedef 类型别名	97
4.1.6 结构体	97
4.2 STM32 系统架构	99
4.3 STM32 时钟系统	100
4.4 端口复用和重映射	104
4.4.1 端口复用功能	104
4.4.2 端口重映射	105
4.5 STM32 NVIC 中断优先级管理	106
4.6 MDK 中寄存器地址名称映射分析	109
4.7 MDK 固件库快速组织代码技巧	111
第五章 SYSTEM 文件夹介绍	117
5.1 delay 文件夹代码介绍	117
5.1.1 delay_init 函数	118
5.1.2 delay_us 函数	119
5.1.3 delay_ms 函数	121
5.2 sys 文件夹代码介绍	122
5.2.1 IO 口的位操作实现	122
5.2.2 中断分组设置函数	124
5.3 usart 文件夹介绍	124
5.3.1 printf 函数支持	125
5.3.2 uart_init 函数	125
5.3.3 USART1_IRQHandler 函数	128
第三篇 实战篇	131
第六章 跑马灯实验	132
6.1 STM32 IO 简介	133
6.2 硬件设计	140



6.3 软件设计	140
6.4 仿真与下载	145
第六章 蜂鸣器实验	148
7.1 蜂鸣器简介	149
7.2 硬件设计	149
7.3 软件设计	150
7.4 仿真与下载	153
第七章 按键输入实验	155
8.1 STM32 IO 口简介	156
8.2 硬件设计	156
8.3 软件设计	156
8.4 仿真与下载	159
第八章 串口实验	164
9.1 STM32 串口简介	165
9.2 硬件设计	167
9.3 软件设计	168
9.4 下载验证	171
第九章 外部中断实验	174
10.1 STM32 外部中断简介	175
10.2 硬件设计	178
10.3 软件设计	178
10.4 下载验证	180
第十章 独立看门狗（IWDG）实验	181
11.1 STM32 独立看门狗简介	182
11.2 硬件设计	183
11.3 软件设计	184
11.4 下载验证	185
第十一章 窗口看门狗（WWDG）实验	186
12.1 STM32 窗口看门狗简介	187
12.2 硬件设计	189
12.3 软件设计	190
12.4 下载验证	191



第十三章 定时器中断实验	192
13.1 STM32 通用定时器简介	193
13.2 硬件设计	198
13.3 软件设计	198
13.4 下载验证	200
第十四章 PWM 输出实验	201
14.1 PWM 简介	202
14.2 硬件设计	205
14.3 软件设计	205
14.4 下载验证	207
第十五章 输入捕获实验	209
15.1 输入捕获简介	210
15.2 硬件设计	214
15.3 软件设计	214
15.4 下载验证	218
第十六章 电容触摸按键实验	220
16.1 电容触摸按键简介	221
16.2 硬件设计	222
16.3 软件设计	222
16.4 下载验证	227
第十七章 OLED 显示实验	228
17.1 OLED 简介	229
17.2 硬件设计	235
17.3 软件设计	236
17.4 下载验证	243
第十八章 TFTLCD 显示实验	245
18.1 TFTLCD&FSMC 简介	246
18.1.1 TFTLCD 简介	246
18.1.2 FSMC 简介	250
18.2 硬件设计	258
18.3 软件设计	259
18.4 下载验证	270



第十九章 USMART 调试组件实验.....	271
19.1 USMART 调试组件简介	272
19.2 硬件设计	275
19.3 软件设计	275
19.4 下载验证	279
第二十章 RTC 实时时钟实验.....	283
20.1 STM32 RTC 时钟简介	284
20.2 硬件设计	290
20.3 软件设计	290
20.4 下载验证	297
第二十一章 待机唤醒实验	298
21.1 STM32 待机模式简介	299
21.2 硬件设计	302
21.3 软件设计	302
21.4 下载与测试	305
第二十二章 ADC 实验.....	306
22.1 STM32 ADC 简介	307
22.2 硬件设计	315
22.3 软件设计	315
22.4 下载验证	318
第二十三章 内部温度传感器实验	319
23.1 STM32 内部温度传感器简介	320
23.2 硬件设计	320
23.3 软件设计	321
23.4 下载验证	322
第二十四章 DAC 实验.....	324
24.1 STM32 DAC 简介	325
24.2 硬件设计	330
24.3 软件设计	331
24.4 下载验证	333
第二十五章 PWM DAC 实验	335
25.1 PWM DAC 简介	336



25.2 硬件设计	337
25.3 软件设计	338
25.4 下载验证	341
第二十六章 DMA 实验	344
26.1 STM32 DMA 简介	345
26.2 硬件设计	350
26.3 软件设计	350
26.4 下载验证	353
第二十七章 IIC 实验	356
27.1 IIC 简介	357
27.2 硬件设计	357
27.3 软件设计	358
27.4 下载验证	365
第二十八章 SPI 实验	367
28.1 SPI 简介	368
28.2 硬件设计	371
28.3 软件设计	372
28.4 下载验证	377
第二十九章 485 实验	379
29.1 485 简介	380
29.2 硬件设计	381
29.3 软件设计	382
29.4 下载验证	386
第三十章 CAN 通讯实验	388
30.1 CAN 简介	389
30.1.1 CAN 发送流程	398
30.1.2 CAN 接收流程	398
30.2 硬件设计	408
30.3 软件设计	409
30.4 下载验证	414
第三十一章 触摸屏实验	417
31.1 触摸屏简介	418

31.2 硬件设计	419
31.3 软件设计	419
31.4 下载验证	427
第三十二章 红外遥控实验	430
32.1 红外遥控简介	431
32.2 硬件设计	432
32.3 软件设计	433
32.4 下载验证	438
第三十三章 游戏手柄实验	440
33.1 游戏手柄简介	441
33.2 硬件设计	442
33.3 软件设计	444
33.4 下载验证	446
第三十四章 三轴加速度传感器实验	448
34.1 ADXL345 简介	449
34.2 硬件设计	451
34.3 软件设计	452
34.4 下载验证	458
第三十五章 DS18B20 数字温度传感器实验	461
35.1 DS18B20 简介	462
35.2 硬件设计	463
35.3 软件设计	464
35.4 下载验证	469
第三十六章 DHT11 数字温湿度传感器实验	470
36.1 DHT11 简介	471
36.2 硬件设计	473
36.3 软件设计	473
36.4 下载验证	477
第三十七章 无线通信实验	479
37.1 NRF24L01 无线模块简介	480
37.2 硬件设计	480
37.3 软件设计	481



37.4 下载验证	489
第三十八章 PS2 鼠标实验.....	491
38.1 PS/2 简介	492
38.2 硬件设计	494
38.3 软件设计	495
38.4 下载验证	505
第三十九章 FLASH 模拟 EEPROM 实验	506
39.1 STM32 FLASH 简介	507
39.2 硬件设计	513
39.3 软件设计	513
39.4 下载验证	517
第四十章 FM 收发实验	519
40.1 RDA5820 简介	520
40.2 硬件设计	521
40.3 软件设计	523
40.4 下载验证	532
第四十一章 摄像头实验	533
41.1 OV7670 简介	534
41.2 硬件设计	538
41.3 软件设计	540
41.4 下载验证	548
第四十二章 外部 SRAM 实验	550
42.1 IS62WV51216 简介	551
42.2 硬件设计	553
42.3 软件设计	553
42.4 下载验证	557
第四十三章 内存管理实验	559
43.1 内存管理简介	560
43.2 硬件设计	561
43.3 软件设计	561
43.4 下载验证	568
第四十四章 SD 卡实验	570

44.1 SD 卡简介	571
44.2 硬件设计	574
44.3 软件设计	575
44.4 下载验证	579
第四十五章 FATFS 实验	581
45.1 FATFS 简介	582
45.2 硬件设计	587
45.3 软件设计	587
45.4 下载验证	594
第四十六章 汉字显示实验	596
46.1 汉字显示原理简介	597
46.2 硬件设计	601
46.3 软件设计	601
46.4 下载验证	610
第四十七章 图片显示实验	612
47.1 图片格式简介	613
47.2 硬件设计	614
47.3 软件设计	615
47.4 下载验证	623
第四十八章 照相机实验	625
48.1 BMP 编码简介	626
48.2 硬件设计	628
48.3 软件设计	629
48.4 下载验证	635
第四十九章 音乐播放器实验	636
49.1 VS1053 简介	637
49.2 硬件设计	642
49.3 软件设计	642
49.4 下载验证	647
第五十章 录音机实验	649
50.1 WAV 简介	650
50.2 硬件设计	653

50.3 软件设计	653
50.4 下载验证	659
第五十一章 手写识别实验	662
51.1 手写识别简介	663
51.2 硬件设计	667
51.3 软件设计	667
51.4 下载验证	670
第五十二章 T9 拼音输入法实验	672
52.1 拼音输入法简介	673
52.2 硬件设计	675
52.3 软件设计	675
52.4 下载验证	682
第五十三章 串口 IAP 实验	685
53.1 IAP 简介	686
53.2 硬件设计	691
53.3 软件设计	692
53.4 下载验证	698
第五十四章 触控 USB 鼠标实验	701
54.1 USB 简介	702
54.2 硬件设计	703
54.3 软件设计	704
54.4 下载验证	709
第五十五章 USB 读卡器实验	712
55.1 USB 读卡器简介	713
55.2 硬件设计	713
55.3 软件设计	714
55.4 下载验证	717
第五十六章 USB 声卡实验	719
56.1 USB 声卡简介	720
56.2 硬件设计	720
56.3 软件设计	720
56.4 下载验证	722



第五十七章 ENC28J60 网络实验.....	724
57.1 ENC28J60 以及 uIP 简介	725
57.1.1 ENC28J60 简介.....	725
57.1.2 uIP 简介.....	727
57.2 硬件设计	730
57.3 软件设计	731
57.4 下载验证	744
第五十八章 UCOSII 实验 1-任务调度.....	749
58.1 UCOSII 简介	750
58.2 硬件设计	755
58.3 软件设计	755
58.4 下载验证	759
58.5 任务删除, 挂起和恢复测试	759
第五十九章 UCOSII 实验 2-信号量和邮箱.....	764
59.1 UCOSII 信号量和邮箱简介	765
59.2 硬件设计	767
59.3 软件设计	767
59.4 下载验证	773
第六十章 UCOSII 实验 3-消息队列、信号量集和软件定时器.....	774
60.1 UCOSII 消息队列、信号量集和软件定时器简介	775
60.2 硬件设计	782
60.3 软件设计	782
60.4 下载验证	790
第六十一章 战舰 STM32 开发板综合实验.....	792
61.1 战舰 STM32 开发板综合实验简介	793
61.2 战舰 STM32 开发板综合实验详解	794
61.2.1 电子图书	796
61.2.2 数码相框	798
61.2.3 音乐播放	799
61.2.4 应用中心	801
61.2.5 时钟	802
61.2.6 系统设置	803



61.2.7 FC 游戏机	810
61.2.8 收音机	812
61.2.9 记事本	813
61.2.10 运行器	815
61.2.11 3D	816
61.2.12 手写画笔	816
61.2.13 照相机	819
61.2.14 录音机	821
61.2.15 USB 连接	823
61.2.16 TOM 猫	824
61.2.17 无线传书	824
61.2.18 计算器	826

内容简介

本开发指南将由浅入深，带领大家进入 STM32 的世界。本指南总共分为三篇：1，硬件篇，主要介绍本指南的实验平台；2，软件篇，主要介绍 STM32 开发软件的使用以及一些下载调试的技巧，并详细介绍了几个常用的系统文件（程序）；3，实战篇，主要通过 56 个实例（固件库实现）带领大家一步步深入 STM32 的学习。

本指南为 ALIENTEK 战舰 STM32 开发板的固件库版本配套教程，在开发板配套的光盘里面，有详细原理图以及所有实例的完整代码，这些代码都有详细的注释，所有源码都经过我们严格测试，不会有任何警告和错误，另外，源码有我们生成好的 hex 文件，大家只需要通过串口下载到开发板即可看到实验现象，亲自体验实验过程。

本指南不仅非常适合广大学生和电子爱好者学习 STM32，其大量的实验以及详细的解说，也是公司产品开发的不二参考。



前言

Cortex-M3 采用 ARM V7 构架，不仅支持 Thumb-2 指令集，而且拥有很多新特性。较之 ARM7 TDMI，Cortex-M3 拥有更强劲的性能、更高的代码密度、位带操作、可嵌套中断、低成本、低功耗等众多优势。

国内 Cortex-M3 市场，ST（意法半导体）公司的 STM32 无疑是最大赢家，作为 Cortex-M3 内核最先尝蟹的两个公司（另一个是 Luminary（流明））之一，ST 无论是在市场占有率，还是在技术支持方面，都是远超其他对手。在 Cortex-M3 芯片的选择上，STM32 无疑是大家的首选。

现在 ST 公司又推出了 STM32F0 系列 Cortex M0 芯片以及 STM32F4 系列 Coretx M4 芯片，这两种芯片都已经量产，而且可以比较方便的购买到，但是本指南，我们只讨论 Cortex M3，（因为这个现在是性价比最高的 ^_^）有兴趣的读者可以自行了解一下。

STM32 的优异性体现在如下几个方面：

- 1, 超低的价格。以 8 位机的价格，得到 32 位机，是 STM32 最大的优势。
- 2, 超多的外设。STM32 拥有包括：FSMC、TIMER、SPI、IIC、USB、CAN、IIS、SDIO、ADC、DAC、RTC、DMA 等众多外设及功能，具有极高的集成度。
- 3, 丰富的型号。STM32 仅 M3 内核就拥有 F100、F101、F102、F103、F105、F107、F207、F217 等 8 个系列上百种型号，具有 QFN、LQFP、BGA 等封装可供选择。同时 STM32 还推出了 STM32L 和 STM32W 等超低功耗和无线应用型的 M3 芯片。
- 4, 优异的实时性能。84 个中断，16 级可编程优先级，并且所有的引脚都可以作为中断输入。
- 5, 杰出的功耗控制。STM32 各个外设都有自己的独立时钟开关，可以通过关闭相应外设的时钟来降低功耗。
- 6, 极低的开发成本。STM32 的开发不需要昂贵的仿真器，只需要一个串口即可下载代码，并且支持 SWD 和 JTAG 两种调试口。SWD 调试可以为你的设计带来很多的方便，只需要 2 个 IO 口，即可实现仿真调试。

学习 STM32 有两份不错的中文资料：

《STM32 参考手册》中文版 V10.0

《Cortex-M3 权威指南》中文版（宋岩 译）

前者是 ST 官方针对 STM32 的一份通用参考资料，内容翔实，但是没有实例，也没有对 Cortex-M3 构架进行多少介绍（估计 ST 是把读者都当成一个 Cortex-M3 熟悉者来写的），读者只能根据自己对书本的理解来编写相关代码。后者是专门介绍 Cortex-M3 构架的书，有简短的实例，但没有专门针对 STM32 的介绍。所以，在学习 STM32 的时候，必须结合这份资料来看。

STM32 拥有非常多的寄存器，对于新手来说，直接操作寄存器有很大的难度，所以 ST 官方提供了一套固件库函数，大家不需要再直接操作繁琐的寄存器，而是直接调用固件库函数即可实现操作寄存器的目的。当然，我们要了解一些外设的原理，必须对寄存器有一定的了解，这对以后开发和调试也是非常有帮助的，所以在我们手册中我们会保留一些重要寄存器的讲解，但是我们的实例代码基本都是调用固件库来实现的。有关寄存器操作的实例，大家可以参考我们寄存器版本的手册及代码。

本指南将结合《STM32 参考手册》和《Cortex-M3 权威指南》两者的优点，并从固件库级

别出发，深入浅出，向读者展示 STM32 的各种功能。总共配有 56 个实例，基本上每个实例在均配有软硬件设计，在介绍完软硬件之后，马上附上实例代码，并带有详细注释及说明，让读者快速理解代码。

这些实例涵盖了 STM32 的绝大部分内部资源，并且提供很多实用级别的程序，如：内存管理、拼音输入法、手写识别、图片解码、IAP 等。所有实例在 MDK3.80A 编译器下编译通过，大家只需下载程序到 ALIENTEK 战舰 STM32 开发板，即可验证实验。

不管你是一个 STM32 初学者，还是一个老手，本指南都非常适合。尤其对于初学者，本指南将手把手的教你如何使用 MDK，包括新建工程、编译、仿真、下载调试等一系列步骤，让你轻松上手。本指南适用于想通过库函数学习 STM32 的读者，大家可以结合官方提供的库函数实例对照学习。

本指南的实验平台是 ALIENTEK 战舰 STM32 开发板，有这款开发板的朋友则直接可以拿本指南配套的光盘上的例程在开发板上运行、验证。而没有这款开发板而又想要的朋友，可以上淘宝购买。当然你如果有了一款自己的开发板，而又不想再买，也是可以的，只要你的板子上有 ALIENTEK 战舰 STM32 开发板上的相同资源（需要实验用到的），代码一般都是可以通用的，你需要做的就只是把底层的驱动函数（一般是 IO 操作）稍做修改，使之适合你的开发板即可。

俗话说：人无完人。本指南也不例外，在编写过程中虽然得到了不少网友的指正，但难免不会再有出错的地方，如果大家发现指南中有什么错误的地方，还请告诉我们，我们的官方技术支持论坛为 www.openedv.com(开源电子网)，大家可以在论坛上发帖提问。在此先向各位朋友表示衷心的感谢。

ALIENTEK//广州市星翼电子科技有限公司

第一篇 硬件篇

实践出真知，要想学好 STM32，实验平台必不可少！本篇将详细介绍我们用来学习 STM32 的硬件平台：ALIENTEK 战舰 STM32 开发板，通过该篇的介绍，你将了解到我们的学习平台 ALIENTEK 战舰 STM32 开发板的功能及特点。

为了让读者更好的使用 ALIENTEK 战舰 STM32 开发板，本篇还介绍了开发板的一些使用注意事项，请读者在使用开发板的时候一定要注意。

本篇将分为如下两章：

- 1, 实验平台简介；
- 2, 实验平台硬件资源详解；



第一章 实验平台简介

本章，主要向大家简要介绍我们的实验平台：ALIENTEK 战舰 STM32 开发板。通过本章的学习，你将对我们后面使用的实验平台有个大概了解，为后面的学习做铺垫。

本章将分为如下两节：

- 1.1, ALIENTEK 战舰 STM32 开发板资源初探；
- 1.2, ALIENTEK 战舰 STM32 开发板资源说明；

1.1 ALIENTEK 战舰 STM32 开发板资源初探

在 ALIENTEK 战舰 STM32 开发板之前，ALIENTEK 推出过 MiniSTM32 开发板，在两年的时间里面，售出 10000 多套，连续一年多稳居淘宝 STM32 开发板销量之首（目前仍是销量第一）。而这款战舰 STM32 开发板，则是 MiniSTM32 开发板的超级加强版。下面我们开始介绍战舰 STM32 开发板。

ALIENTEK 战舰 STM32 开发板的资源图如图 1.1.1 所示：

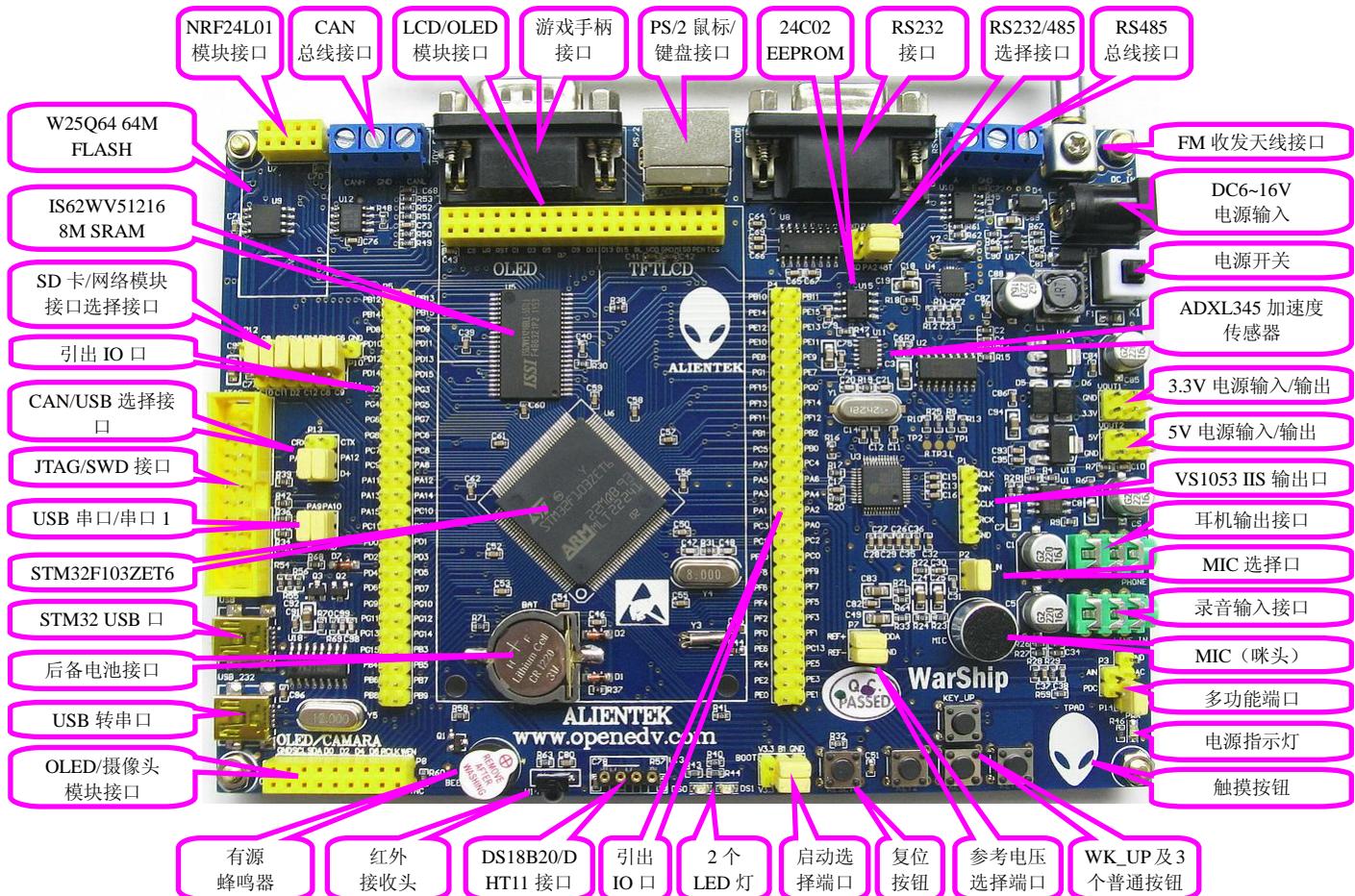


图 1.1.1 战舰 STM32 开发板资源图

从图 1.1.1 可以看出，ALIENTEK 战舰 STM32 开发板，资源十分丰富，并把 STM32F103 的内部资源发挥到了极致，基本所有 STM32F103 的内部资源，都可以在此开发板上验证，同时扩充丰富的接口和功能模块，整个开发板显得十分大气。



开发板的外形尺寸为 11.2cm*15.6cm 大小，板子的设计充分考虑了人性化设计，并结合广大客户对 Mini 板提出的改进意见，经过反复修改（在面市之前，硬件就改版了 8 次之多，目前版本为 V2.0），最终确定了这样的设计。

ALIENTEK 战舰 STM32 开发板板载资源如下：

- ◆ CPU: STM32F103ZET6, LQFP144, FLASH: 512K, SRAM: 64K;
- ◆ 外扩 SRAM: IS62WV51216, 1M 字节
- ◆ 外扩 SPI FLASH: W25Q64, 8M 字节
- ◆ 1 个电源指示灯（蓝色）
- ◆ 2 个状态指示灯（DS0: 红色, DS1: 绿色）
- ◆ 1 个红外接收头，并配备一款小巧的红外遥控器
- ◆ 1 个 EEPROM 芯片，24C02，容量 256 字节
- ◆ 1 个重力加速度传感器芯片，ADXL345
- ◆ 1 个高性能音频编解码芯片，VS1053
- ◆ 1 个 FM 立体声收发芯片，RDA5820
- ◆ 1 个 2.4G 无线模块接口（NRF24L01）
- ◆ 1 路 CAN 接口，采用 TJA1050 芯片
- ◆ 1 路 485 接口，采用 SP3485 芯片
- ◆ 1 路 RS232 接口，采用 SP3232 芯片
- ◆ 1 个 PS/2 接口，可外接鼠标、键盘
- ◆ 1 个游戏手柄接口，可以直接插 FC(红白机) 游戏手柄
- ◆ 1 路数字温湿度传感器接口，支持 DS18B20 /DHT11 等
- ◆ 1 个标准的 2.4/2.8/3.5 寸 LCD 接口，支持触摸屏
- ◆ 1 个摄像头模块接口
- ◆ 2 个 OLED 模块接口
- ◆ 1 个 USB 串口，可用于程序下载和代码调试（USMART 调试）
- ◆ 1 个 USB SLAVE 接口，用于 USB 通信
- ◆ 1 个有源蜂鸣器
- ◆ 1 个 FM 收发天线接口，并配天线
- ◆ 1 个 RS232/RS485 选择接口
- ◆ 1 个 CAN/USB 选择接口
- ◆ 1 个串口选择接口
- ◆ 1 个 SD 卡接口（在板子背面，支持 SPI/SDIO）
- ◆ 1 个 SD 卡/网络模块选择接口
- ◆ 1 个标准的 JTAG/SWD 调试下载口
- ◆ 1 个 VS1053 的 IIS 输出接口
- ◆ 1 个 MIC/LINE IN 选择接口
- ◆ 1 个录音头（MIC/咪头）
- ◆ 1 路立体声音频输出接口
- ◆ 1 路立体声录音输入接口
- ◆ 1 组多功能端口（DAC/ADC/PWM DAC/AUDIO IN/TPAD）
- ◆ 1 组 5V 电源供应/接入口
- ◆ 1 组 3.3V 电源供应/接入口
- ◆ 1 个参考电压设置接口



- ◆ 1个直流电源输入接口（输入电压范围：6~16V）
- ◆ 1个启动模式选择配置接口
- ◆ 1个RTC后备电池座，并带电池
- ◆ 1个复位按钮，可用于复位MCU和LCD
- ◆ 4个功能按钮，其中WK_UP兼具唤醒功能
- ◆ 1个电容触摸按键
- ◆ 1个电源开关，控制整个板的电源
- ◆ 独创的一键下载功能
- ◆ 除晶振占用的IO口外，其余所有IO口全部引出

ALIENTEK 战舰 STM32 开发板的特点包括：

- 1) 接口丰富。板子提供十来种标准接口，可以方便的进行各种外设的实验和开发。
- 2) 设计灵活。板上很多资源都可以灵活配置，以满足不同条件下的使用。我们引出了除晶振占用的IO口外的所有IO口，可以极大的方便大家扩展及使用。另外板载一键下载功能，可避免频繁设置B0、B1的麻烦，仅通过1根USB线即可实现STM32的开发。
- 3) 资源充足。外扩1M字节SRAM和8M字节FLASH，满足大内存需求和大数据存储。板载MP3和FM收发芯片，娱乐学习两不误。板载3D加速度传感器和各种接口芯片，满足各种应用需求。
- 4) 人性化设计。各个接口都有丝印标注，使用起来一目了然；接口位置设计安排合理，方便顺手。资源搭配合理，物尽其用。

1.2 ALIENTEK 战舰 STM32 开发板资源说明

资源说明部分，我们将分为两个部分说明：硬件资源说明和软件资源说明。

1.2.1 硬件资源说明

这里我们首先详细介绍战舰 STM32 开发板的各个部分（图 1.1.1 中的标注部分）的硬件资源，我们将按逆时针的顺序依次介绍。

1. W25Q64 64M FLASH

这是开发板外扩的 SPI FLASH 芯片，容量为 64Mbit，也就是 8M 字节，可用于存储字库和其他用户数据，满足大容量数据存储要求。当然如果觉得 8M 字节还不够用，你可以把数据存放在外部 SD 卡。

2. IS62WV51216 8M SRAM

这是开发板外扩的 SRAM 芯片，容量为 8M 位，也就是 1M 字节，这样，对大内存需求的应用（比如 GUI），就可以很好的实现了。

3. SD 卡/网络模块接口选择接口

这里是一个由 3 拍排针（在板上标号[下同]为：P10、P11 和 P12）组成的复合接口，当不用网络模块的时候，这个组合就变成了 SD 卡的接口选择接口，可以通过跳线帽选择 SDIO/SPI（我们默认是设置在 SPI 接口的）。但是，如果需要网络模块（网络模块接 P12），那么 SD 卡就只能用 SDIO 模式了。

4. 引出 IO 口

这里是一组 54 个 IO 口的引出(P5)，在它的右侧不远，是另外一组 54 个 IO 口的引出(P4)，这两组排针引出 108 个 IO，而 STM32F103ZET6 总共只有 112 个 IO，除去 RTC 晶振占用的 2 个 IO，还剩下 PA9 和 PA10 没有在这里引出（由 P6 引出）。



5. CAN/USB 选择接口

这是一个 USB/CAN 的选择接口 (P13)，因为 STM32 的 USB 和 CAN 是共用一组 IO (PA11 和 PA12)，所以我们通过跳线帽来选择不同的功能，以实现 USB/CAN 的实验。

6. JTAG/SWD 接口

这是 ALIENTEK 战舰 STM32 开发板板载的 20 针标准 JTAG 调试口 (JTAG)，该 JTAG 口直接可以和 ULINK、JLINK 或者 STLINK 等调试器(仿真器)连接，同时由于 STM32 支持 SWD 调试，这个 JTAG 口也可以用 SWD 模式来连接。

用标准的 JTAG 调试，需要占用 5 个 IO 口，有些时候，可能造成 IO 口不够用，而用 SWD 则只需要 2 个 IO 口，大大节约了 IO 数量，但他们达到的效果是一样的，所以我们强烈建议你的仿真器使用 SWD 模式！

7. USB 串口/串口 1

这是 USB 串口同 STM32F103ZET6 的串口 1 进行连接的接口 (P6)，标号 RXD 和 TXD 是 USB 转串口的 2 个数据口 (对 CH340G 来说)，而 PA9(TXD)和 PA10(RXD)则是 STM32 的串口 1 的两个数据口(复用功能下)。他们通过跳线帽对接，就可以和连接在一起了，从而实现 STM32 的程序下载以及串口通信。

设计成 USB 串口，是出于现在电脑上串口正在消失，尤其是笔记本，几乎清一色的没有串口。所以板载了 USB 串口可以方便大家下载代码和调试。而在板子上并没有直接连接在一起，则是出于使用方便的考虑。这样设计，你可以把 ALIENTEK 战舰 STM32 开发板当成一个 USB 串口，来和其他板子通信，而其他板子的串口，也可以方便地接到 ALIENTEK 战舰 STM32 开发板上。

8. STM32F103ZET6

这是开发板的核心芯片 (U5)，型号为：STM32F103ZET6。该芯片具有 64KB SRAM、512KB FLASH、2 个基本定时器、4 个通用定时器、2 个高级定时器、3 个 SPI、2 个 IIC、5 个串口、1 个 USB、1 个 CAN、3 个 12 位 ADC、1 个 12 位 DAC、1 个 SDIO 接口、1 个 FSMC 接口以及 112 个通用 IO 口。

9. STM32 USB 口

这是开发板板载的一个 MiniUSB 头 (USB)，用于 STM32 与电脑的 USB 通讯，通过此 MiniUSB 头，开发板就可以和电脑进行 USB 通信了。开发板总共板载了 2 个 MiniUSB 头，一个用于 USB 转串口，连接 CH340G 芯片；另外一个用于 STM32 内带的 USB。

同时开发板可以通过此 MiniUSB 头供电，板载两个 MiniUSB 头 (不共用)，主要是考虑了使用的方便性，以及可以给板子提供更大的电流（两个 USB 都接上）这两个因素。

10. 后备电池接口

这是 STM32 后备区域的供电接口，可以用来给 STM32 的后备区域提供能量，在外部电源断电的时候，维持后备区域数据的存储，以及 RTC 的运行。

11. USB 转串口

这是开发板板载的另外一个 MiniUSB 头 (USB_232)，用于 USB 连接 CH340G 芯片，从而实现 USB 转串口。同时，此 MiniUSB 接头也是开发板电源的主要提供口。

12. OLED/摄像头模块接口

这是开发板板载的一个 OLED/摄像头模块接口 (P8)，如果是 OLED 模块，靠左插即可 (右边两个孔位悬空)。如果是摄像头模块 (ALIENTEK 提供)，则刚好插满。通过这个接口，可以分别连接 2 个外部模块，从而实现相关实验。

13. 有源蜂鸣器

这是开发板的板载蜂鸣器 (BEEP)，可以实现简单的报警/闹铃。让开发板可以听得见。



14. 红外接收头

这是开发板的红外接收头（U14），可以实现红外遥控功能，通过这个接收头，可以接受市面上常见的各种遥控器的红外信号，大家甚至可以自己实现万能红外解码。当然，如果应用得当，该接收头也可以用来传输数据。

战舰 STM32 开发板给大家配了一个小巧的红外遥控器，该遥控器外观如图 1.2.1 所示：



图 1.2.1 红外遥控器

15. DS18B20/DHT11 接口

这是开发板的一个复用接口（U13），该接口由 4 个镀金排孔组成，可以用来接 DS18B20/DS1820 等数字温度传感器。也可以用来接 DHT11 这样的数字温湿度传感器。实现一个接口，2 个功能。不用的时候，大家可以拆下上面的传感器，放到其他地方去用，使用上是十分方便灵活的。

16. 2 个 LED 灯

这是开发板板载的两个 LED 灯（DS0 和 DS1），DS0 是红色的，DS1 是绿色的，主要是方便大家识别。这里提醒大家不要停留在 51 跑马灯的思维，搞这么多灯，除了浪费 IO 口，实在是想不出其他什么优点。

我们一般的应用 2 个 LED 足够了，在调试代码的时候，使用 LED 来指示程序状态，是非常不错的一个辅助调试方法。战舰 STM32 开发板几乎每个实例都使用了 LED 来指示程序的运行状态。

17. 启动选择端口

这是开发板板载的启动模式选择端口（BOOT），STM32 有 BOOT0（B0）和 BOOT1（B1）两个启动选择引脚，用于选择复位后 STM32 的启动模式，作为开发板，这两个是必须的。在开发板上，我们通过跳线帽选择 STM32 的启动模式。关于启动模式的说明，请看 2.1.8 小节。

18. 复位按钮

这是开发板板载的复位按键（RESET），用于复位 STM32，还具有复位液晶的功能，因为液晶模块的复位引脚和 STM32 的复位引脚是连接在一起的，当按下该键的时候，STM32 和液晶一并被复位。



19. 参考电压选择端口

这是 STM32 的参考电压选择端口 (P7)，我们默认是接开发板的 3.3V 和 GND。如果大家想设置其他参考电压，只需要把你的参考电压源接到 REF- 和 REF+ 上即可。

20. WK_UP 及 3 个普通按钮

这是开发板板载的 4 个机械式输入按键 (KEY0、KEY1、KEY2 和 WK_UP)，其中 WK_UP 具有唤醒功能，该按键连接到 STM32 的 WAKE_UP (PA0) 引脚，可用于待机模式下的唤醒，在不使用唤醒功能的时候，也可以做为普通按键输入使用。

其他 3 个是普通按键，可以用于人机交互的输入，这 3 个按键是直接连接在 STM32 的 IO 口上的。这里注意 WK_UP 是高电平有效，而 KEY0、KEY1 和 KEY2 是低电平有效，大家在使用的时候留意一下。

21. 触摸按钮

这是开发板板载的一个电容触摸输入按键 (TPAD)，用于实现触摸按键。现在触摸按键非常流行，所以我们在开发板上也设计了一个，咱得跟上时代的步伐。

22. 电源指示灯

这是开发板板载的一颗蓝色的 LED 灯 (PWR)，用于指示电源状态。在电源开启的时候 (通过板上的电源开关控制)，该灯会亮，否则不亮。通过这个 LED，可以判断开发板的上电情况。

23. 多功能端口

这里大家可别小看这 6 个排针，这可是本开发板设计的很巧妙的一个端口 (由 P3 和 P14 组成)，这组端口通过组合可以实现的功能有：ADC 采集、DAC 输出、PWM DAC 输出、外部音频输入、电容触摸按键、DAC 音频、PWM DAC 音频、DAC ADC 自测等，所有这些，你只需要 1 个跳线帽的设置，就可以逐一实现。

24. MIC (咪头)

这是开发板的板载录音输入口 (MIC)，该咪头直接接到 VS1053 的输入上，可以用来实现录音功能。

25. 录音输入接口

这是开发板板载的外部录音输入接口 (LINE_IN)，通过咪头我们只能实现单声道的录音，而通过这个 LINE_IN，我们可以实现立体声录音。

26. MIC 选择口

这是开发板板载录音的接入选择口 (P2)，如果使用 LINE_IN 录音的时候，我们把 P2 断开，以排除来自咪头的干扰信号，从而可以更好的立体声录音。而使用咪头录音的时候，我们短接 P2 即可。

27. 耳机输出接口

这是开发板板载的音频输出接口 (PHONE)，战舰 STM32 开发板有多个音频输出 (VS1053/ 收音机/PWM DAC 等)，通过 74HC4052 实现音频选择，输入到 TDA1308，再输出到该音频输出口，实现开发板的音频输出。

28. VS1053 IIS 输出口

这是 VS1053 的 IIS 输出接口 (P1)，该接口可以用来连接外部 DAC，实现更好的音质输出。其实我觉得 VS1053 本身的音频 DAC 已经很好了。这个接口适合发烧友使用。

29. 5V 电源输入/输出

这是开发板板载的一组 5V 电源输入输出排针 (2*3) (VOUT2)，用于给外部提供 5V 的电源，也可以用于从外部取 5V 的电源给板子供电。

大家在实验的时候可能经常会为没有 5V 电源而苦恼不已，有了 ALIENTEK 战舰 STM32 开发板，你就可以很方便的拥有一个简单的 5V 电源 (最大电流不能超过 500ma)。



30. 3.3V 电源输入/输出

这是开发板板载的一组 3.3V 电源输入输出排针 (2*3) (VOU1)，该排针用于给外部提供 3.3V 的电源，也可以用于从外部取 3.3V 的电源给板子供电。

同样大家在实验的时候可能经常会为没有 3.3V 电源而苦恼不已，ALIENTEK 充分考虑到了大家需求，有了这组 3.3V 排针，你就可以很方便的拥有一个简单的 3.3V 电源（最大电流不能超过 500ma）。

31. ADXL345 加速度传感器

这是开发板板载的一个 3 轴加速度传感器 (U11)，ADXL345 分辨率高 (13 位)，测量范围大 ($\pm 16g$)，可以通过 SPI/IIC 访问，战舰开发板采用 IIC 访问它。有了这个，大家就可以实现一些比较有意思的应用（比如测量倾角等）

32. 电源开关

这是开发板板载的电源开关 (K1)。该开关用于控制整个开发板的供电，如果切断，则整个开发板都将断电，电源指示灯 (PWR) 会随着此开关的状态而亮灭。

33. DC9~12V 电源输入

这是开发板板载的一个外部电源输入口 (DC_IN)，采用标准的直流电源插座。开发板板载了 DC-DC 芯片 (MP2359)，用于给开发板提供高效、稳定的 5V 电源。由于采用了 DC-DC 芯片，所以开发板的供电范围十分宽，大家可以很方便的找到合适的电源（只要输出范围在 DC6~16V 的基本都可以）来给开发板供电。特别注意：如果你使用的是战舰 V2.0 以前的版本，输入电压建议不要超过 9V！切记不能超过 12V！战舰 V2.0 及以后的版本才支持 DC6~16V 的宽输入范围。

34. FM 收发天线接口

这个是开发板板载 FM 收发芯片的天线接口 (ANT)，同时我们安装有天线在这个上面。通过这个天线，可以很好的实现 FM 收音和 FM 发射。

35. RS485 总线接口

这是开发板板载的 RS485 总线接口 (RS485)，通过 3 个端口和外部 485 设备连接。一般情况下，只需要连接 2 个端口即可，即 A 和 B，并不需要连接 GND。这里提醒大家，RS485 通信的时候，必须 A 接 A，B 接 B。否则可能通信不正常！

36. RS232/485 选择接口

这是开发板板载的 RS232/485 选择接口 (P9)，因为 RS485 基本上就是一个半双工的串口，为了节约 IO，我们把 RS232 和 RS485 共用一个串口，通过 P9 来设置当前是使用 RS232 还是 RS485。当然，这样的设计还有一个好处。就是我们的开发板既可以充当 RS232 到 TTL 串口的转换，又可以充当 RS485 到 TTL485 的转换。（注意，这里的 TTL 高电平是 3.3V）

37. RS232 接口

这是开发板板载的 RS232 接口 (COM)，通过一个标准的 DB9 母头和外部的串口连接。通过这个接口，我们可以连接带有串口的电脑或者其他设备，实现串口通信。

38. 24C02 EEPROM

这是开发板板载的 EEPROM 芯片 (U15)，容量为 2Kb，也就是 256 字节。用于存储一些掉电不能丢失的重要数据，比如系统设置的一些参数/触摸屏校准数据等。有了这个就可以方便的实现掉电数据保存。

39. PS/2 鼠标/键盘接口

这是开发板板载的一个标准 PS/2 母头 (PS/2)，用于连接电脑鼠标和键盘等 PS/2 设备。

通过 PS/2 口，我们仅仅需要 2 个 IO 口，就可以扩展一个键盘，所以大家没有必要对板上只有 4 个按键而感到担忧。ALIENTEK 提供了标准的鼠标驱动例程，方便大家学习 PS/2 协议。



40. 游戏手柄接口

这是开发板板载的一个 9 针游戏手柄接口 (JOY_PAD)，可以用来连接 FC 手柄 (红白机/小霸王游戏机手柄)，这样大家可以在开发板上编写游戏程序，直接通过手柄玩游戏了。我们的综合实验提供有一个简单的 NES 模拟器，大家可以直接从网上下载 nes 游戏，放到开发板上玩。

41. LCD/OLED 模块接口

这是战舰 STM32 开发板的又一个特色设计，一个接口，兼容多种模块。如果是 OLED 模块，请靠左侧插。如果是 LCD 模块，则靠右侧插。OLED 模块支持 ALIENTEK 的单色/双色 OLED 模块。LCD 模块则支持 ALIENTEK 的 2.4/2.8/3.5 寸 LCD 模块，并且支持触摸屏功能。

42. CAN 总线接口

这是开发板板载的 CAN 总线接口 (CAN)，通过 3 个端口和外部 CAN 总线连接。一般情况下，只需要连接 2 个端口即可，即 CANH 和 CANL，并不需要连接 GND。这里提醒大家，CAN 通信的时候，必须 CANH 接 CANH，CANL 接 CANL。否则可能通信不正常！

43. NRF24L01 模块接口

这是开发板板载的 NRF24L01 模块接口 (U7)，只要插入模块，我们便可以实现无线通信，从而使得我们板子具备了无线功能，但是这里需要 2 个模块和 2 个开发板同时工作才可以。如果只有 1 个开发板或 1 个模块，是没法实现无线通信的。

1.2.2 软件资源说明

上面我们详细介绍了 ALIENTEK 战舰 STM32 开发板的硬件资源。接下来，我们将向大家简要介绍一下战舰 STM32 开发板的软件资源。

战舰 STM32 开发板提供的标准例程多达 57 个，一般的 STM32 开发板仅提供库函数代码，而我们则提供寄存器和库函数两个版本的代码 (本指南以寄存器版本作为介绍)。我们提供的这些例程，基本都是原创，拥有非常详细的注释，代码风格统一、循序渐进，非常适合初学者入门。而其他开发板的例程，大都是来自 ST 库函数的直接修改，注释也比较少，对初学者来说不容易入门。战舰 STM32 开发板的例程列表如表 1.2.2.1 所示：

编号	实验名字	编号	实验名字
1	跑马灯实验	30	DS18B20 数字温度传感器实验
2	蜂鸣器实验	31	DHT11 数字温湿度传感器实验
3	按键输入实验	32	无线通信实验
4	串口实验	33	PS2 鼠标实验
5	外部中断实验	34	FLASH 模拟 EEPROM 实验
6	独立看门狗实验	35	FM 收发实验
7	窗口看门狗实验	36	摄像头实验
8	定时器中断实验	37	外部 SRAM 实验
9	PWM 输出实验	38	内存管理实验
10	输入捕获实验	39	SD 卡实验
11	电容触摸按键实验	40	FATFS 实验
12	OLED 实验	41	汉字显示实验
13	TFTLCD 实验	42	图片显示实验
14	USMART 调试实验	43	照相机实验
15	RTC 实验	44	音乐播放器实验
16	待机唤醒实验	45	录音机实验



17	ADC 实验	46	手写识别实验
18	内部温度传感器实验	47	T9 拼音输入法实验
19	DAC 实验	48	串口 IAP 实验
20	PWM DAC 实验	49	触控 USB 鼠标实验
21	DMA 实验	50	USB 读卡器实验
22	IIC 实验	51	USB 声卡实验
23	SPI 实验	52	ENC28J60 网络模块实验
24	485 实验	53	UCOSII 实验 1-任务调度
25	CAN 收发实验	54	UCOSII 实验 2-信号量和邮箱
26	触摸屏实验	55	UCOSII 实验 3-消息队列、信号量 集和软件定时器
27	红外遥控实验	56	战舰系统综合实验
28	游戏手柄实验	57	ucGUI 实验
29	三轴加速度传感器实验		

表 1.2.2.1 ALIENTEK 战舰 STM32 开发板例程表

从上表可以看出，ALIENTEK 战舰 STM32 开发板的例程基本上涵盖了 STM32F103ZET6 的所有内部资源，并且外扩展了很多有价值的例程，比如：FLASH 模拟 EEPROM 实验、IAP 实验、拼音输入法实验、手写识别实验、综合实验等。

而且从上表可以看出，例程安排是循序渐进的，首先从最基础的跑马灯开始，然后一步步深入，从简单到复杂，有利于大家的学习和掌握。所以，ALIENTEK 战舰 STM32 开发板是非常适合初学者的。当然，对于想深入了解 STM32 内部资源的朋友，ALIENTEK 战舰 STM32 开发板也绝对是一个不错的选择。

这里特别说明一下战舰系统综合实验，这个实验使得 ALIENTEK 战舰 STM32 开发板更像一个产品，而不单单是一个开发板了，它拥有目前市面上所有开发板中最复杂，最强大的功能，可玩性极高，它的实现，充分向大家展示了 ALIENTEK 战舰开发板的优势，同时也证明了 STM32 的强悍性能。解决了一部分人，STM32 能干啥的顾虑。

第二章 实验平台硬件资源详解

本章，我们将向大家详细介绍 ALIENTEK 战舰 STM32 开发板各部分的硬件原理图，让大家对该开发板的各部分硬件原理有个深入理解，并向大家介绍开发板的使用注意事项，为后面的学习做好准备。

本章将分为如下两节：

- 1.1, 开发板原理图详解;
- 1.2, 开发板使用注意事项;

2.1 开发板原理图详解

2.1.1 MCU

ALIENTEK 战舰 STM32 开发板选择的是 STM32F103ZET6 作为 MCU，该芯片是 STM32F103 里面配置非常强大的了，它拥有的资源包括：64KB SRAM、512KB FLASH、2 个基本定时器、4 个通用定时器、2 个高级定时器、3 个 SPI、2 个 IIC、5 个串口、1 个 USB、1 个 CAN、3 个 12 位 ADC、1 个 12 位 DAC、1 个 SDIO 接口、1 个 FSMC 接口以及 112 个通用 IO 口。该芯片的配置十分强悍，并且还带外部总线（FSMC）可以用来外扩 SRAM 和连接 LCD 等，通过 FSMC 驱动 LCD，可以显著提高 LCD 的刷屏速度，更重要的是其价格，23 元左右的零售价，足以秒杀很多其他芯片了。所以我们选择了它作为我们的主芯片。MCU 部分的原理图如图 2.1.1.1（请大家打开开发板光盘的原理图查看清晰版本）所示：

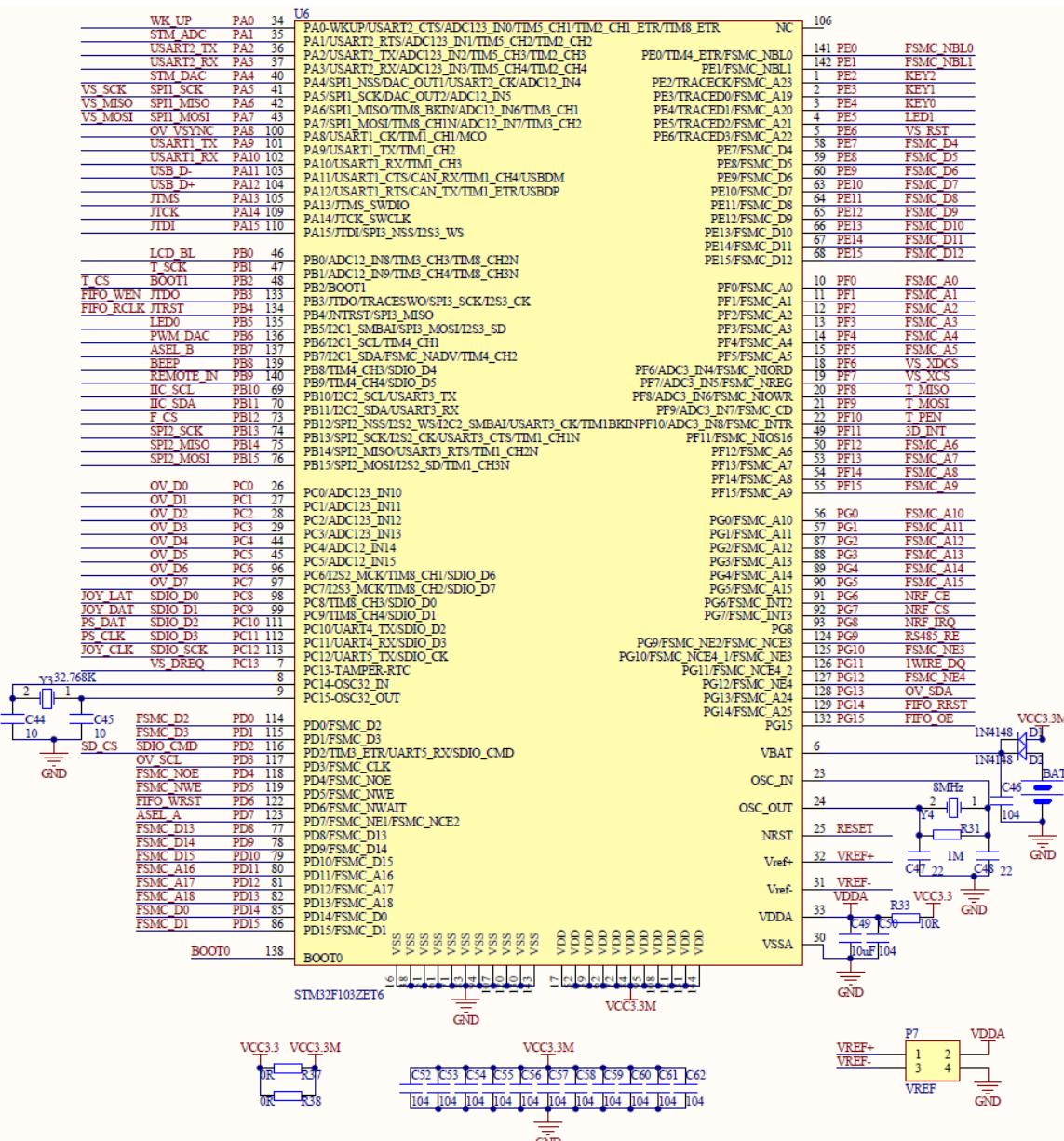


图 2.1.1.1 MCU 部分原理图

上图中 U5 为我们的主芯片：STM32F103ZET6。

这里主要讲解一下 3 个地方：

1，后备区域供电脚 VBAT 脚的供电采用 CR1220 纽扣电池和 VCC3.3 混合供电的方式，在有外部电源(VCC3.3)的时候，CR1220 不给 VBAT 供电，而在外部电源断开的时候，则由 CR1220 给其供电。这样，VBAT 总是有电的，以保证 RTC 的走时以及后备寄存器的内容不丢失。

2，图中的 R37 和 R38 用隔离 MCU 部分和外部的电源，这样的设计主要是考虑了后期维护，如果 3.3V 电源短路，可以断开这两个电阻，来确定是 MCU 部分短路，还是外部短路，有助于生产和维修。当然大家在自己的设计上，这两个电阻是完全可以去掉的。

3，图中 P7 是参考电压选择端口。我们开发板默认是接板载的 3.3V 作为参考电压，如果大家想用自己的参考电压，则把你的参考电压接入 VRFF- 和 VRFF+ 即可。



2.1.2 引出 IO 口

ALIENTEK 战舰 STM32 开发板引出了 STM32F103ZET6 的所有 IO 口，如图 2.1.2.1 所示：

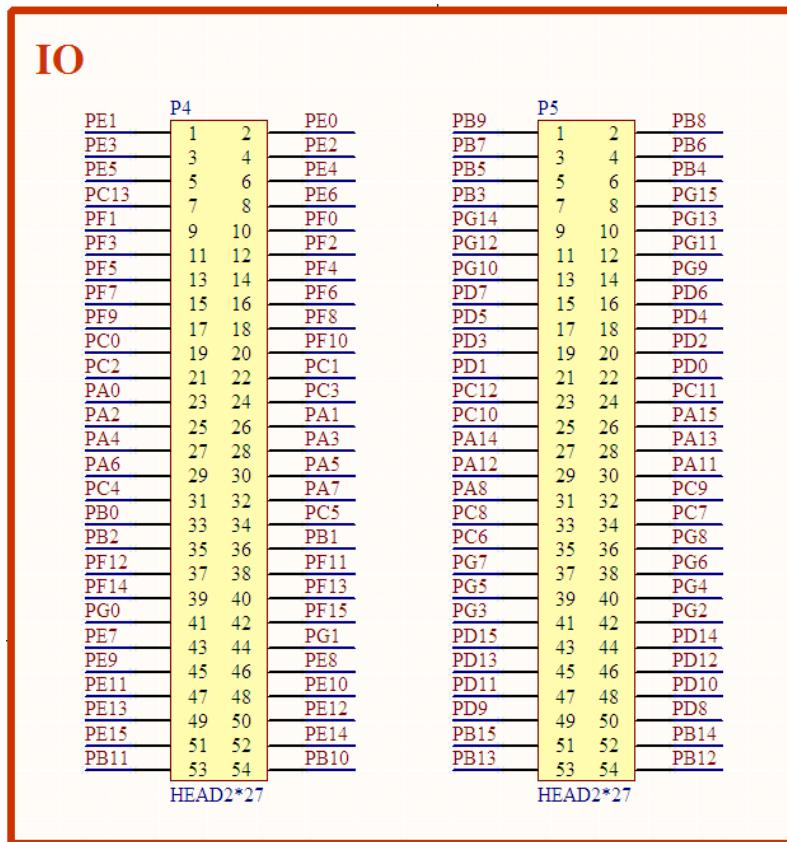


图 2.1.2.1 引出 IO 口

图中 P4 和 P5 为 MCU 主 IO 引出口，这两组排针每组引出 IO 数位 54 个，共 108 个 IO 从这里引出。STM32F103ZET6 总共有 112 个 IO，除去 RTC 晶振占用的 2 个，还剩 110 个，这两组 IO 引出除 PA9 和 PA10 以外的所有 IO 口。大家可以通过这两组 IO 引出口，方便的扩展自己的外设。(PA9 和 PA10 通过 P6 引出)

2.1.3 USB 串口/串口 1 选择接口

ALIENTEK 战舰 STM32 开发板板载的 USB 串口和 STM32F103ZET6 的串口是通过 P6 连接起来的，如图 2.1.3.1 所示：

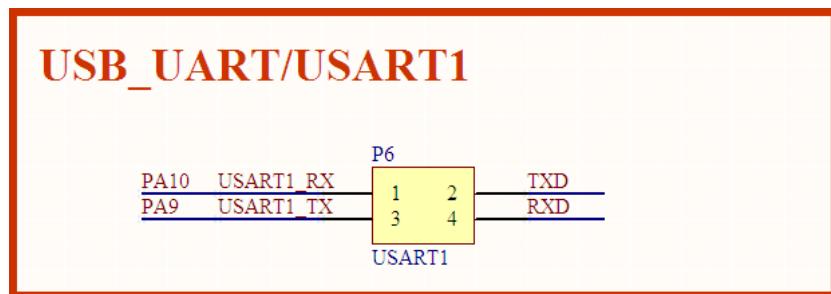


图 2.3.1.1 USB 串口/串口 1 选择接口

图中 TXD/RXD 是相对 CH340G 来说的，也就是 USB 串口的发送和接受脚。而 USART1_RX



和 USART1_TX 则是相对于 STM32F103ZET6 来说的。这样，通过对接，就可以实现 USB 串口和 STM32F103ZET6 的串口通信了。同时，P6 是 PA9 和 PA10 的引出口。

这样设计的好处就是使用上非常灵活。比如需要用到外部 TTL 串口和 STM32 通信的时候，只需要拔了跳线帽，通过杜邦线连接外部 TTL 串口，就可以实现和外部设备的串口通信了；又比如我有个板子需要和电脑通信，但是电脑没有串口，那么你就可以使用开发板的 RXD 和 TXD 来连接你的设备，把我们的开发板当成 USB 串口用了。

2.1.4 JTAG/SWD

ALIENTEK 战舰 STM32 开发板板载的标准 20 针 JTAG/SWD 接口电路如图 2.1.4.1 所示：

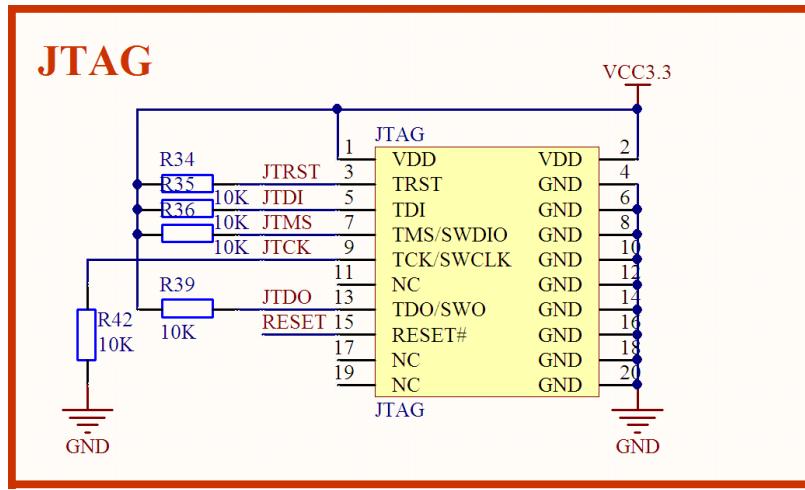


图 2.1.4.1 JTAG/SWD 接口

这里，我们采用的是标准的 JTAG 接法，但是 STM32 还有 SWD 接口，SWD 只需要最少 2 跟线（SWCLK 和 SWDIO）就可以下载并调试代码了，这同我们使用串口下载代码差不多，而且速度非常快，能调试。所以建议大家在设计产品的时候，可以留出 SWD 来下载调试代码，而摒弃 JTAG。STM32 的 SWD 接口与 JTAG 是共用的，只要接上 JTAG，你就可以使用 SWD 模式了（其实并不需要 JTAG 这么多线），当然，你的调试器必须支持 SWD 模式，JLINK V7/V8、ULINK2 和 ST LINK 等都支持 SWD 调试。

2.1.5 SRAM

ALIENTEK 战舰 STM32 开发板外扩了 1M 字节的 SRAM 芯片，如图 2.1.5.1 所示，注意图中的地址线标号，是以 IS61LV51216 为模版的，但是和 IS62WV51216 的 datasheet 标号有出入，不过，因为地址的唯一性，这并不会影响我们使用 IS62WV51216，因此，该原理图对这两个芯片都是可以正常使用的。



SRAM

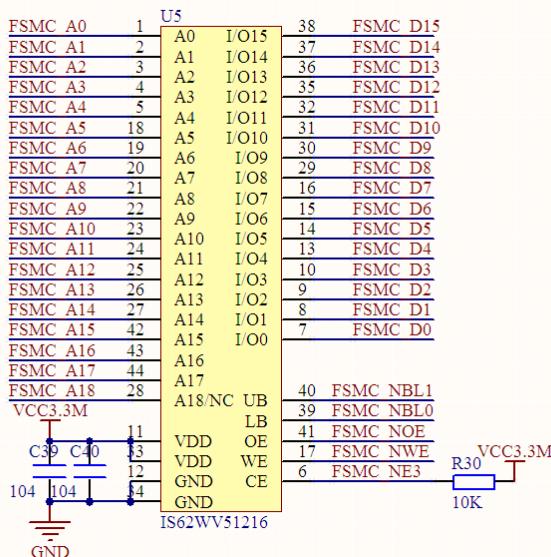


图 2.1.5.1 外扩 SRAM

图中 U6 为外扩的 SRAM 芯片,型号为:IS62WV51216,容量为 1M 字节,该芯片挂在 STM32 的 FSMC 上。这样大大扩展了 STM32 的内存(芯片本身只有 64K 字节),从而在需要大内存的时候,战舰 STM32 开发板也可以胜任。

2.1.6 LCD/OLED 模块接口

ALIENTEK 战舰 STM32 开发板板载的 LCD/OLED 模块接口电路如图 2.1.6.1 所示:

LCD/OLED

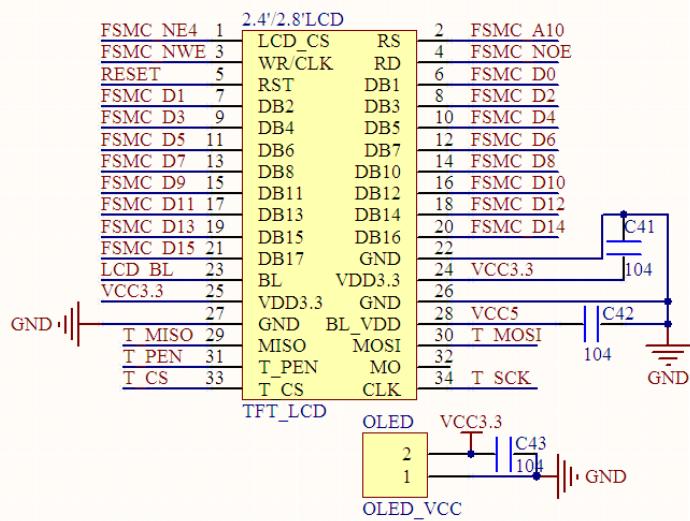


图 2.1.6.1 LCD/OLED 模块接口

图中 TFT_LCD 是一个通用的液晶模块接口,而 OLED 是一个给 OLED 显示模块供电的接口,它和 TFT_LCD 拼接在一起,组合成一个组合接口。当使用 2.4 寸/2.8 寸/3.5 寸的 LCD 时,我们接到 TFT_LCD 上就可以了(靠右插),而当我们使用 ALIENTEK 的 OLED 模块时,则接



OLED 排针做电源，同时会连接到 TFT_LCD 上的部分管脚（靠左插），从而实现 OLED 与 MCU 的连接。TFTLCD 模块也是接在 STM32F103ZET6 的 FSMC 上的，相比战舰 STM32 开发板，这样可以显著提高 LCD 刷屏速度。

图中的 T_MISO/T莫斯I/T_PEN/T_CS/T_CS 用来实现对液晶触摸屏的控制。LCD_BL 则控制 LCD 的背光。液晶复位信号 RESET 则是直接连接在开发板的复位按钮上，和 MCU 共用一个复位电路。

2.1.7 复位电路

ALIENTEK 战舰 STM32 开发板的复位电路如图 2.1.7.1 所示：

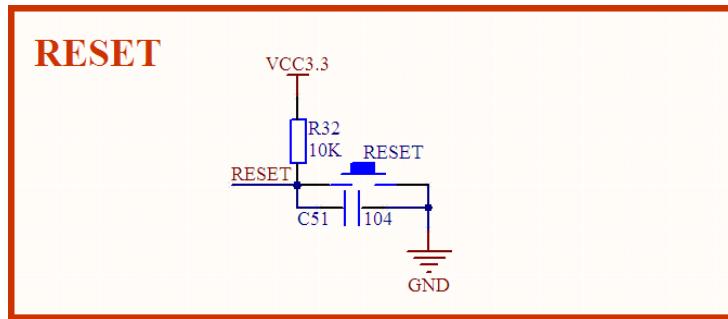


图 2.1.7.1 复位电路

因为 STM32 是低电平复位的，所以我们设计的电路也是低电平复位的，这里的 R32 和 C51 构成了上电复位电路。同时，开发板把 TFT_LCD 的复位引脚也接在 RESET 上，这样这个复位按钮不仅可以用来复位 MCU，还可以复位 LCD。

2.1.8 启动模式设置接口

ALIENTEK 战舰 STM32 开发板的启动模式设置端口电路如图 2.8.1.1 所示：

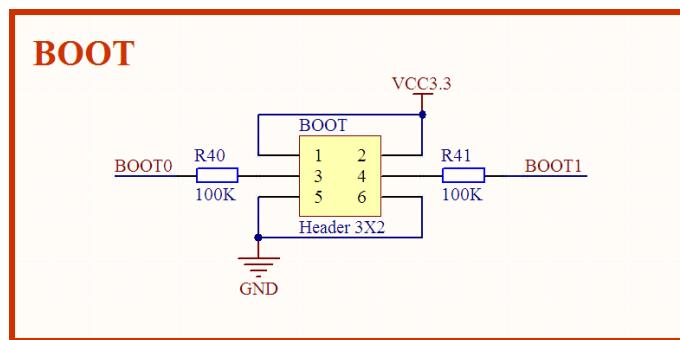


图 2.8.1.1 启动模式设置接口

上图的 BOOT0 和 BOOT1 用于设置 STM32 的启动方式，其对应启动模式如表 2.1.8.1 所示：

BOOT0	BOOT1	启动模式	说明
0	X	用户闪存存储器	用户闪存存储器，也就是FLASH启动
1	0	系统存储器	系统存储器启动，用于串口下载
1	1	SRAM启动	SRAM启动，用于在SRAM中调试代码

表 2.8.1.1 BOOT0、BOOT1 启动模式表

按照表 2.8.1.1，一般情况下如果我们想用串口下载代码，则必须配置 BOOT0 为 1，BOOT1 为 0，而如果想让 STM32 一按复位键就开始跑代码，则需要配置 BOOT0 为 0，BOOT1 随便设



置都可以。这里 ALIENTEK 战舰 STM32 开发板专门设计了一键下载电路，通过串口的 DTR 和 RTS 信号，来自动配置 BOOT0 和 RST 信号，因此不需要用户来手动切换他们的状态，直接串口下载软件自动控制，可以非常方便的下载代码。

2.1.9 RS232 串口

ALIENTEK 战舰 STM32 开发板板载的 RS232 串口电路，如图 2.1.9.1 所示：

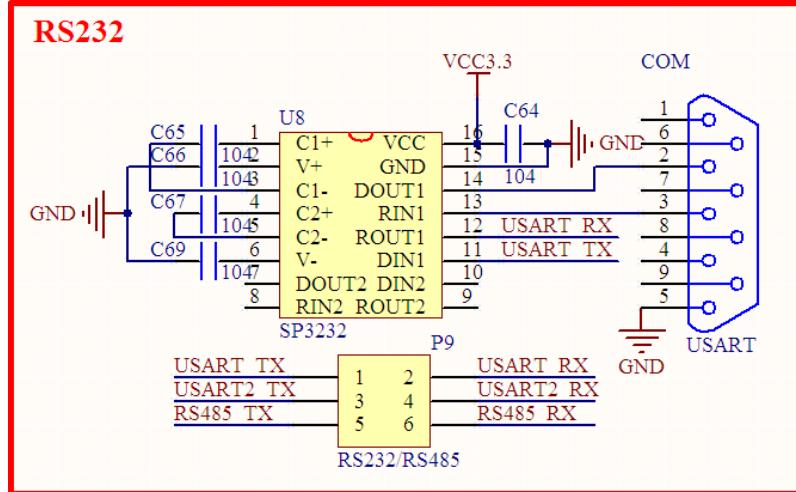


图 2.1.9.1 RS232 串口

因为 RS232 电平不能直接连接到 STM32，所以需要一个电平转换芯片。这里我们选择的是 SP3232（也可以用 MAX3232）来做电平转接，同时图中的 P9 用来实现 RS232/RS485 的选择，以满足不同实验的需要。

图中 USART2_TX/USART2_RX 连接在 MCU 的串口 2 上 (PA2/PA3)，所以这里的 RS232/RS485 都是通过串口 2 来实现的。图中 RS485_TX 和 RS485_RX 信号接在 SP3485 的 DI 和 RO 信号上。

因为 P9 的存在，其实还带来另外一个好处，就是我们可以把开发板变成一个 RS232 电平转换器，或者 RS485 电平转换器，比如你买的核心板，可能没有板载 RS485/RS232 接口，通过连接战舰 STM32 开发板的 P9 端口，就可以让你的核心板拥有 RS232/RS485 的功能。

2.1.10 RS485 接口

ALIENTEK 战舰 STM32 开发板板载的 RS485 接口电路如图 2.1.10.1 所示：

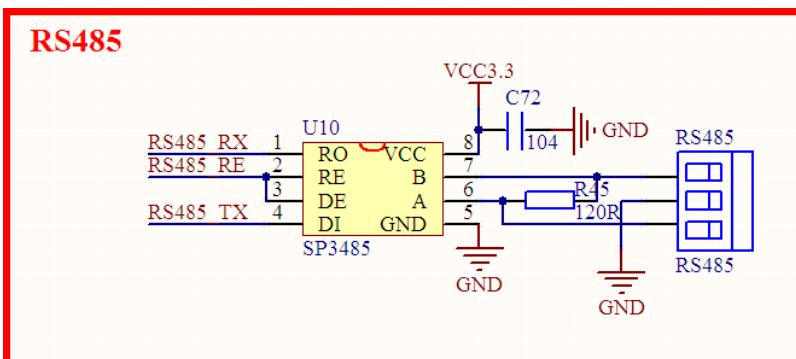


图 2.1.10.1 RS485 接口

RS485 电平也不能直接连接到 STM32，同样需要电平转换芯片。这里我们使用 SP3485 来



做 485 电平转换，其中 R48 为匹配电阻。

RS485_RX/RS485_TX 连接在 P9 上面，通过 P9 跳线来选择是否连接在 MCU 上面，RS485_RE 则是直接连接在 MCU 的 IO 口 (PG9) 上的，该信号用来控制 SP3485 的工作模式 (高电平为发送模式，低电平为接收模式)。

2.1.11 CAN/USB 接口

ALIENTEK 战舰 STM32 开发板板载的 CAN 接口电路以及 STM32 USB 接口电路如图 2.1.11.1 所示：

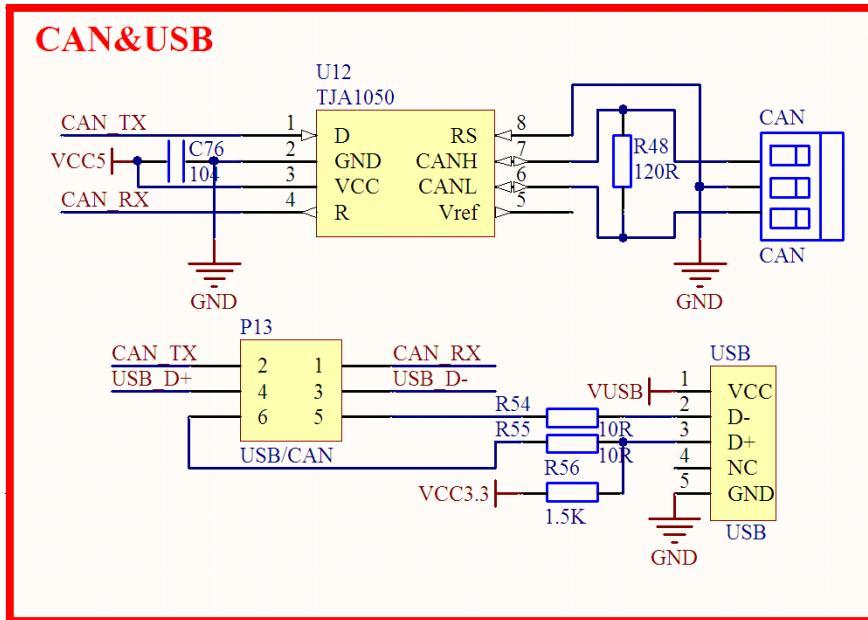


图 2.1.11.1 CAN/USB 接口

CAN 总线电平也不能直接连接到 STM32，同样需要电平转换芯片。这里我们使用 TJA1050 来做 CAN 电平转换，其中 R48 为匹配电阻。

USB_D+/USB_D- 连接在 MCU 的 USB 口 (PA12/PA11) 上，同时，因为 STM32 的 USB 和 CAN 共用这组信号，所以我们通过 P13 来选择使用 USB 还是 CAN。

图中的 USB 端子还具有供电功能，VUSB 为开发板的 USB 供电口，通过这个 USB 口，就可以给整个开发板供电了。

2.1.12 EEPROM

ALIENTEK 战舰 STM32 开发板板载的 EEPROM 电路如图 2.1.12.1 所示：

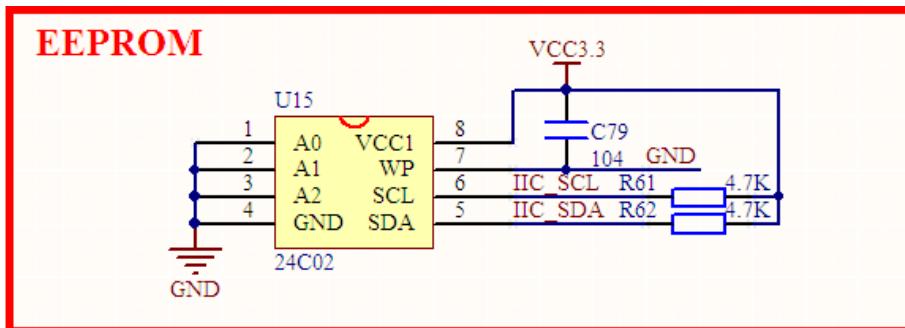


图 2.1.12.1 EEPROM



EEPROM 芯片我们使用的是 24C02，该芯片的容量为 2Kb，也就是 256 个字节，对于我们普通应用来说是足够的了。当然，你也可以选择换大的芯片，因为我们的电路在原理上是兼容 24C02~24C512 全系列 EEPROM 芯片的。

这里我们把 A0~A2 均接地，对 24C02 来说也就是把地址位设置成了 0 了，写程序的时候要注意这点。IIC_SCL 接在 MCU 的 PB10 上，IIC_SDA 接在 MCU 的 PB11 上，这里我们虽然接到 STM32 的硬件 IIC 上，但是我们并不提倡使用硬件 IIC，因为 STM32 的 IIC 是鸡肋！请谨慎使用。IIC_SCL/IIC_SDA 总线上总共挂了 3 个器件：24C02、ADXL345 和 RDA5820，后续我们将向大家介绍另外两个器件。

2.1.13 游戏手柄接口

ALIENTEK 战舰 STM32 开发板板载的游戏手柄接口电路如图 2.1.13.1 所示：

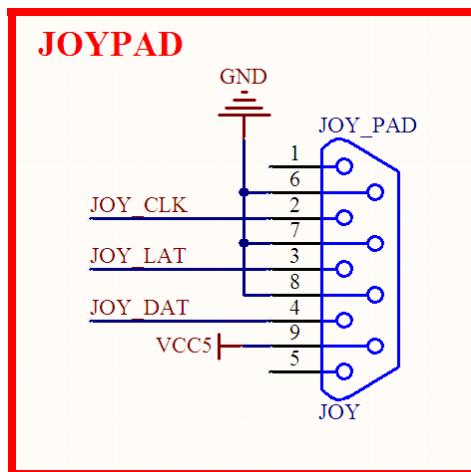


图 2.1.13.1 游戏手柄接口

因为很多 FC 游戏机(俗称红白机/小霸王游戏机)的手柄都是 9 针接口，刚好可以插到 9 针的串口公头里面。这里我们使用一个 DB9 公头来做 FC 游戏手柄接口。

JOY_CLK/JOY_LAT/JOY_DAT 分别连接在 MCU 的 PC12/PC8/PC9 上，这 3 个信号和 SDIO 的 SCK/D0/D1 共用，所以他们不能同时使用！这里特别提醒：因为这个 DB9 的 2,3 脚直接接在 STM32 的 IO 口，所以，这个口一定不要接 RS232 串口!! 否则可能直接把 STM32F103ZET6 给烧了。

2.1.14 SPI FLASH

ALIENTEK 战舰 STM32 开发板板载的 SPI FLASH 电路如图 2.1.14.1 所示：

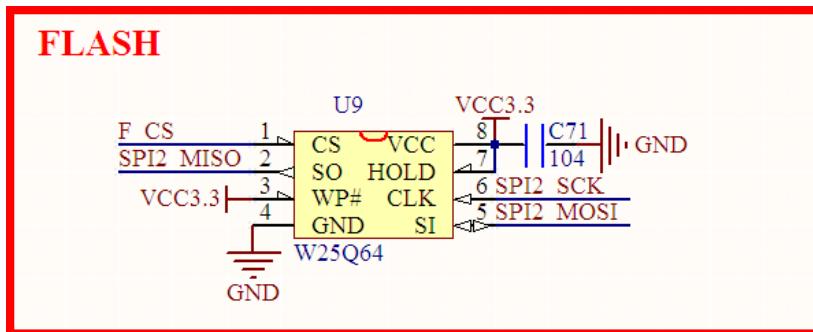


图 2.1.14.1 SPI FLASH 芯片



SPI FLASH 芯片型号为 W25Q64，该芯片的容量为 64Mb，也就是 8M 字节。该芯片和 SD 卡、NRF24L01 共用一个 SPI (SPI2)，通过片选来选择使用某个器件，在使用其中一个器件的时候，请务必禁止另外两个器件的片选信号。

图中 F_CS 连接在 MCU 的 PB12 上，SPI2_SCK/SPI2_MOSI/SPI2_MISO 则分别连接在 MCU 的 PB13/PB15/PB14 上。

2.1.15 3D 加速度传感器

ALIENTEK 战舰 STM32 开发板板载的 3D 加速度传感器电路如图 2.1.15.1 所示：

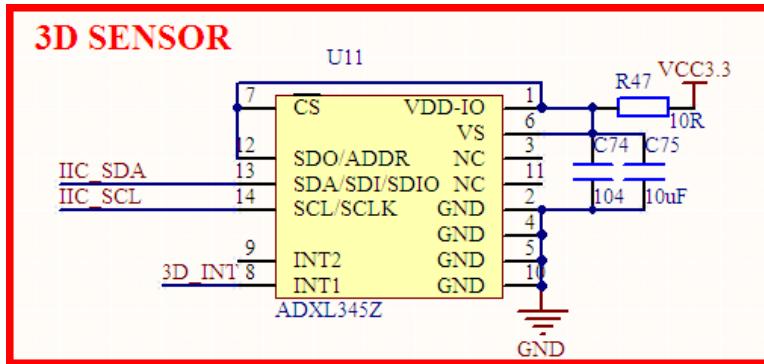


图 2.1.15.1 3D 加速度传感器

3D 加速度传感器芯片型号为 ADXL345，该芯片具有分辨率高（13 位），测量范围大（±16g）的特点，支持多种接口，这里我们使用 IIC 接口来访问。

同 24C02 一样，该芯片的 IIC_SCL 和 IIC_SDA 同样是挂在 PB10 和 PB11 上，他们共享一个 IIC 总线。

2.1.16 温湿度传感器接口

ALIENTEK 战舰 STM32 开发板板载的温湿度传感器接口电路如图 2.1.16.1 所示：

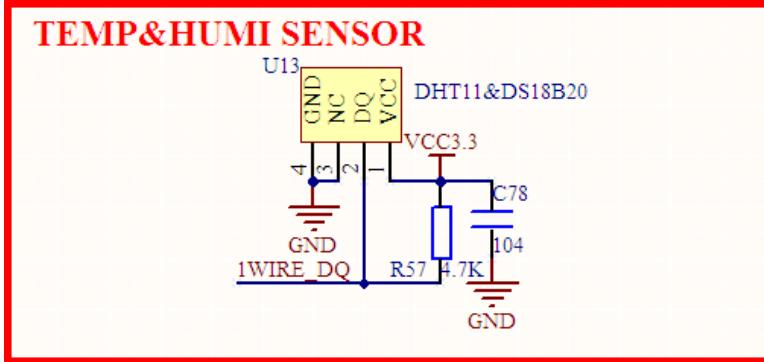


图 2.1.16.1 温湿度传感器接口

该接口支持 DS18B20/DS1820/DHT11 等单总线数字温湿度传感器。1WIRE_DQ 是传感器的数据线，该信号连接在 MCU 的 PG11 上。

2.1.17 红外接收头

ALIENTEK 战舰 STM32 开发板板载的红外接收头电路如图 2.1.17.1 所示：

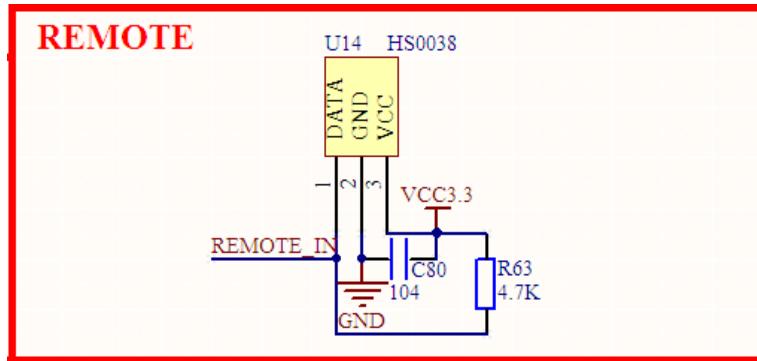


图 2.1.17.1 红外接收头

HS0038 是一个通用的红外接收头，几乎可以接收市面上所有红外遥控器的信号，有了它，就可以用红外遥控器来控制开发板了。REMOTE_IN 为红外接收头的输出信号，该信号连接在 MCU 的 PB9 上。

2.1.18 无线模块接口

ALIENTEK 战舰 STM32 开发板板载的无线模块接口电路如图 2.1.18.1 所示：

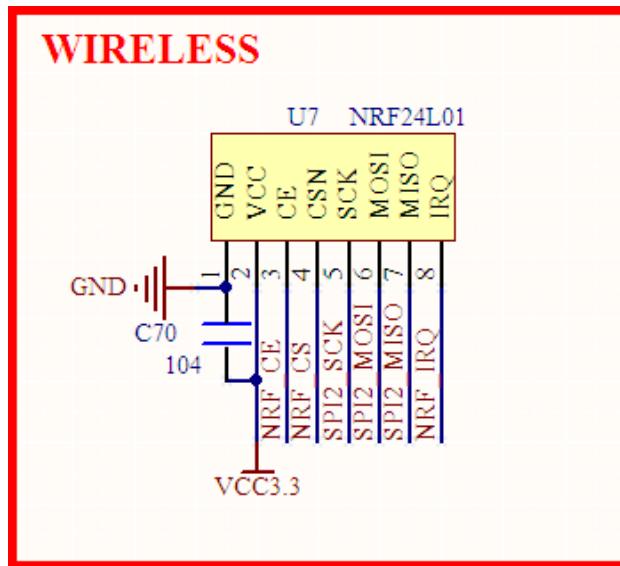


图 2.1.18.1 无线模块接口

该接口用来连接 NRF24L01 等 2.4G 无线模块，从而实现开发板与其他设备的无线数据传输（注意：NRF24L01 不能和蓝牙/WIFI 连接）。NRF24L01 无线模块的最大传输速度可以达到 2Mbps，传输距离最大可以到 30 米左右（空旷地，无干扰）。

NRF_CE/NRF_CS/NRF_IRQ 连接在 MCU 的 PG6/PG7/PG8 上，而另外 3 个 SPI 信号则和 SPI FLASH 共用。

2.1.19 LED

ALIENTEK 战舰 STM32 开发板板载总共有 3 个 LED，其原理图如图 2.1.19.1 所示：

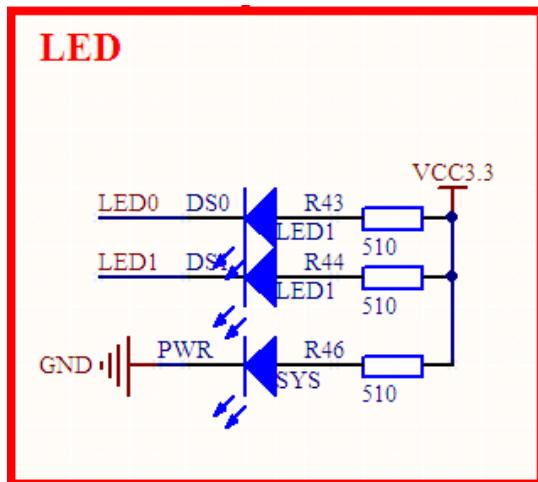


图 2.1.19.1 LED

其中 PWR 是系统电源指示灯，为蓝色。LED0 和 LED1 分别接在 PB5 和 PE5 上。为了方便大家判断，我们选择了 DS0 为红色的 LED，DS1 为绿色的 LED。

2.1.20 按键

ALIENTEK 战舰 STM32 开发板板载总共有 4 个输入按键，其原理图如图 2.1.20.1 所示：

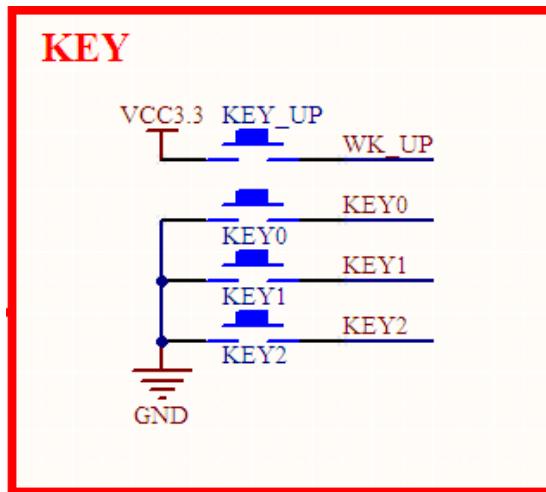


图 2.1.20.1 输入按键

KEY0、KEY1 和 KEY2 用作普通按键输入，分别连接在 PE4、PE3 和 PE2 上，这里并没有使用外部上拉电阻，但是 STM32 的 IO 作为输入的时候，可以设置上下拉电阻，所以我们使用 STM32 的内部上拉电阻来为按键提供上拉。

WK_UP 按键连接到 PA0(STM32 的 WKUP 引脚)，它除了可以用作普通输入按键外，还可以用作 STM32 的唤醒输入。这个按键是高电平触发的。

2.1.21 TPAD 电容触摸按键

ALIENTEK 战舰 STM32 开发板板载了一个电容触摸按键，其原理图如图 2.1.21.1 所示：

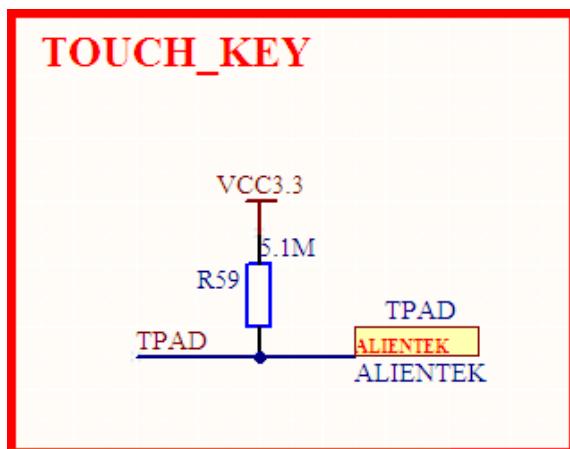


图 2.1.21.1 电容触摸按键

图中 5.1M 是电容充电电阻，TPAD 并没有直接连接在 MCU 上，而是连接在多功能端口（P14）上面，通过跳线帽来选择是否连接到 STM32。多功能端口，我们将在 2.1.25 节介绍。

电容触摸按键的原理我们将在后续的实战篇里面介绍。

2.1.22 PS/2 接口

ALIENTEK 战舰 STM32 开发板板载了一个 PS/2 接口，其原理图如图 2.1.22.1 所示：

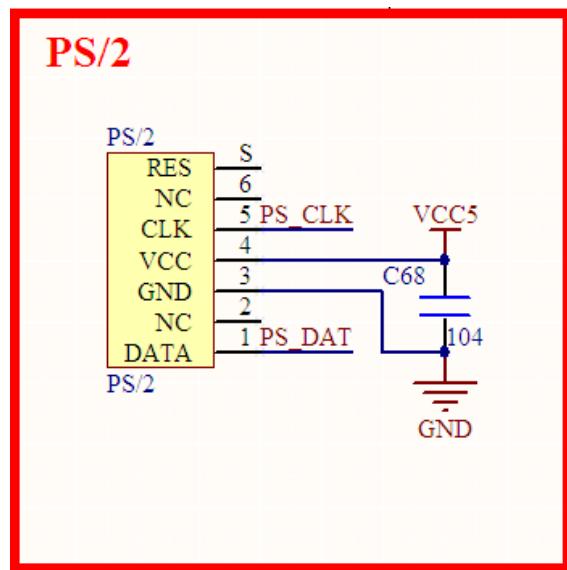


图 2.1.22.1 PS/2 接口

有了该接口，我们就可以用来连接外部标准的 PS/2 鼠标或键盘等设备了，也就大大的扩展了 ALIENTEK 战舰 STM32 开发板的输入。PS_CLK 和 PS_DAT 分别接 PC11 和 PC10，PS/2 的信号线的上拉电阻我们还是选择 STM32 内部的上拉电阻来实现。注意 PS/2 接口和 SDIO_D2 和 SDIO_D3 公用了 IO 口，所以他们不能同时工作。

2.1.23 OLED/摄像头模块接口

ALIENTEK 战舰 STM32 开发板板载了一个 OLED/摄像头模块接口，其原理图如图 2.1.23.1 所示：

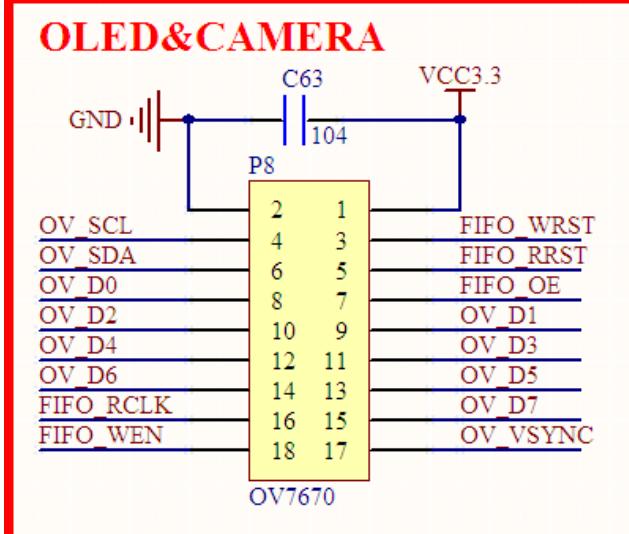


图 2.1.23.1 OLED/摄像头模块接口

图中 P8 是接口可以用来连接 ALIENTEK OLED 模块或者 ALIENTEK 摄像头模块。如果是 OLED 模块，则 FIFO_WEN 和 OV_VSYNC 不需要接（在板上靠左插即可），如果是摄像头模块，则需要用到全部引脚。

其中，OV_SCL/OV_SDA/FIFO_WRST/FIFO_RRST/FIFO_OE 这 5 个信号是分别连接在 MCU 的 PD3/PG13/PD6/PG14/PG15 上面，OV_D0~OV_D7 则连接在 PC0~7 上面（放在连续的 IO 上，可以提高读写效率），FIFO_RCLK/FIFO_WEN/OV_VSYNC 这 3 个信号是分别连接在 MCU 的 PB4/PB3/PA8 上面。其中 PB3 和 PB4 又是 JTAG 的 JTRST/JTDO 信号，所以在使用 OV7670 的时候，不要用 JTAG 仿真，要选择 SWD 模式（所以我们建议大家直接用 SWD 模式来连接我们的开发板，这样所有的实验都可以仿真！）。

2.1.24 有源蜂鸣器

ALIENTEK 战舰 STM32 开发板板载了一个有源蜂鸣器，其原理图如图 2.1.24.1 所示：

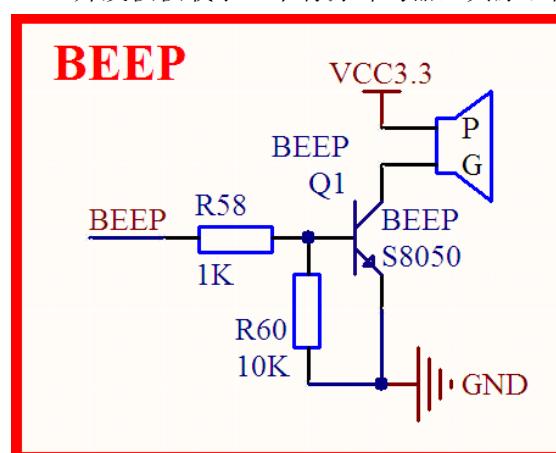


图 2.1.24.1 有源蜂鸣器

有源蜂鸣器是指自带了震荡电路的蜂鸣器，这种蜂鸣器一接上电就会自己震荡发声。而如果是无源蜂鸣器，则需要外加一定频率（2~5Khz）的驱动信号，才会发声。这里我们选择使用有源蜂鸣器，方便大家使用。



图中 Q1 是用来扩流，R60 则是一个下拉电阻，避免 MCU 复位的时候，蜂鸣器可能发声的现象。BEEP 信号直接连接在 MCU 的 PB8 上面，PB8 可以做 PWM 输出，所以大家如果想玩高级点（如：控制蜂鸣器“唱歌”），就可以使用 PWM 来控制蜂鸣器。

2.1.25 SD 卡/以太网模块接口

ALIENTEK 战舰 STM32 开发板板载了一个 SD 卡（大卡/相机卡）接口，但是并没有板载以太网，不过我们板载了以太网模块接口，通过外部模块扩展以太网，其原理图如图 2.1.25.1 所示：

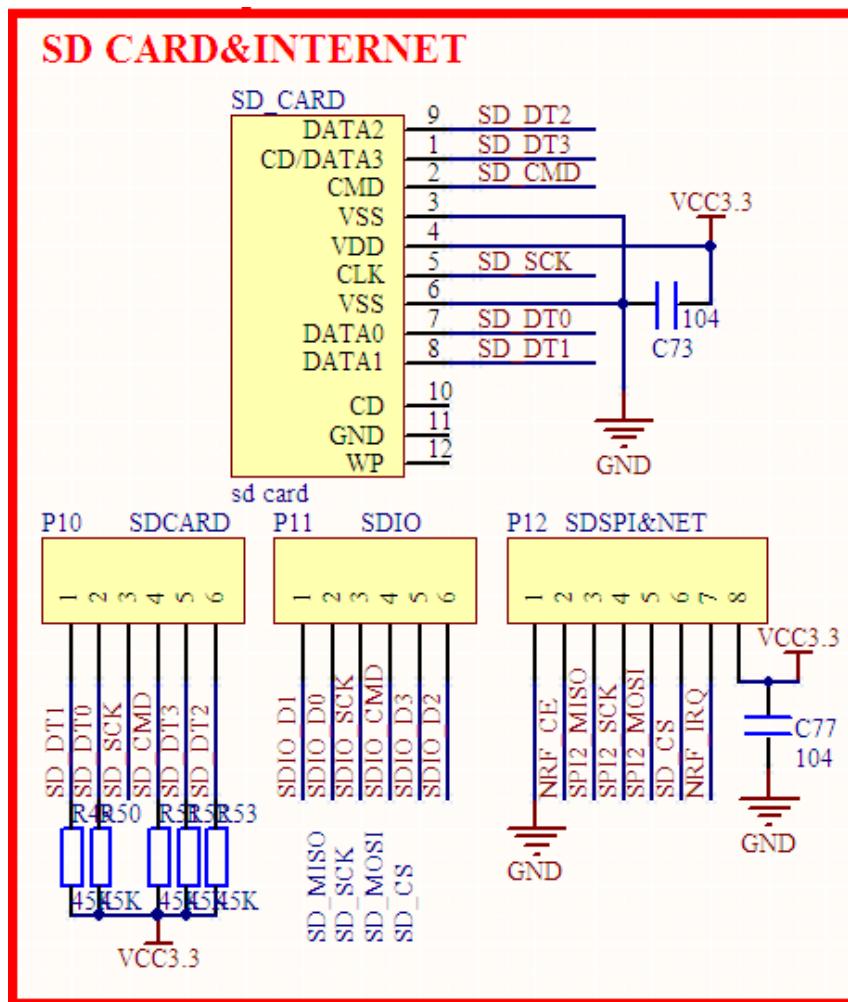


图 2.1.25.1 SD 卡/以太网接口

图中 SD_CARD 为 SD 卡接口，该接口在开发板的底面，这也是战舰 STM32 开发板底面唯一的元器件。

在开发板的 PCB 上 P10/P11/P12 组合在一起，构成一个 SD 卡接口方式选择接口，可以用来设置 SD 卡是工作在 SDIO 模式，还是工作在 SPI 模式。同时 P12 兼具以太网模块接口功能（因为 ALIENTEK 网络模块接口和 P12 接口一模一样，我们只需要拿一组排线把他们对接即可实现以太网与开发板的连接），这里需要注意以太网模块除 SD_CS 信号外，其余信号都是使用无线模块的。使用以太网模块的时候，SD 卡就只能工作在 SDIO 模式了，同时无线模块也将无法使用。



SD_DT0~SD_DT3 分别连接在 PC8~PC11 上面，他们和游戏手柄和 PS/2 接口信号共用 IO，所以在使用 SDIO 模式的时候，手柄和 PS/2 设备将不能使用。SD_SCK 和 SD_CMD 则分别连接在 PC12 和 PD2 上，而 SD_CS 和 SD_CMD 一样，也是连接在 PD2 上的，而 SD_CS 则是网络模块的 INT 信号，所以当不适用 INT 信号的时候，网络模块和 SD 卡可以同时工作，而当要用 INT 的时候，SD 卡将不能和网络模块一起使用，这点大家在使用上要稍加注意。P12 的其余信号都是和无线模块共用，前面已经有介绍了，这里我们就不再介绍了。

2.1.26 多功能端口

ALIENTEK 战舰 STM32 开发板板载的多功能端口，是由 P14 和 P3 构成的一个 6PIN 端口，其原理图如图 2.1.26.1 所示：

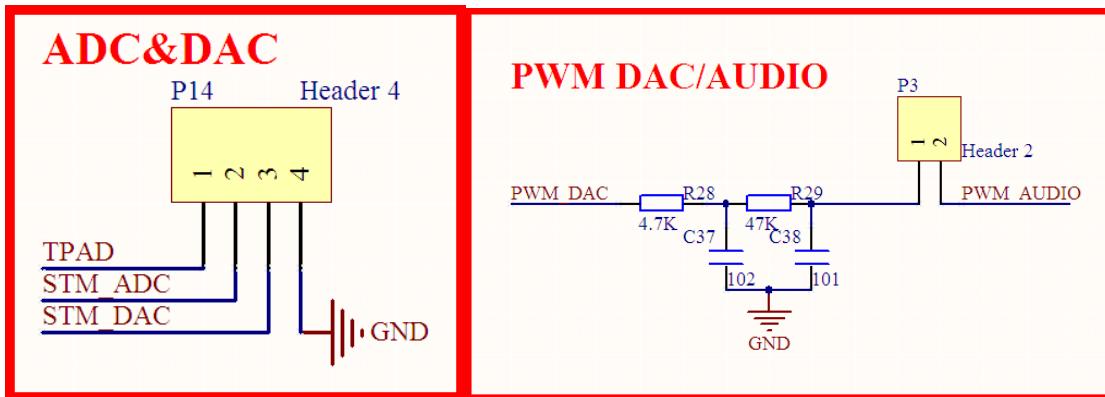


图 2.1.26.1 多功能端口

从上图，大家可能还看不出这个多功能端口的全部功能，别担心，下面我们会详细介绍。

首先介绍左侧的 P14，其中 TPAD 为电容触摸按键信号，连接在电容触摸按键上。STM_ADC 和 STM_DAC 则分别连接在 PA1 和 PA4 上，用于 ADC 采集或 DAC 输出。当需要电容触摸按键的时候，我们通过跳线帽短接 TPAD 和 STM_ADC，就可以实现电容触摸按键（利用定时器的输入捕获）。STM_DAC 信号则既可以用作 DAC 输出，也可以用作 ADC 输入，因为 STM32 的该管脚同时具有这两个复用功能。

我们再来看看 P3，PWM_DAC 连接在 MCU 的 PB6，是定时器 4 的通道 1 输出，后面跟一个二阶 RC 滤波电路，其截止频率为 33.8Khz。经过这个滤波电路，MCU 输出的方波就变为直流信号了。PWM_AUDIO 是一个音频输入通道，它连接到 74HC4052，再进入到 TDA1308 进行输出缓冲，最终输出到耳机。

单独介绍完了 P3 和 P14，我们再来看看他们组合在一起的多功能端口，如图 2.1.26.2 所示：

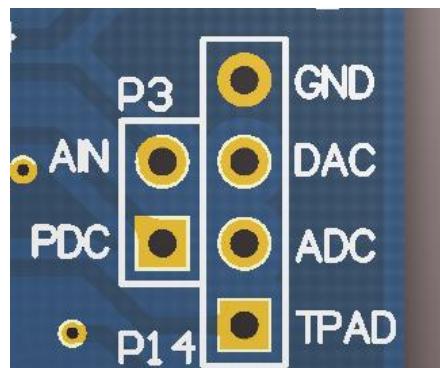


图 2.1.26.2 组合后的多功能端口



图中 AIN 是 PWM_AUDIO, PDC 是滤波后的 PWM_DAC 信号。下面我们来看看通过 1 个跳线帽，这个多功能接口可以实现哪些功能。

当不用跳线帽的时候：1, AIN 和 GND 组成一个音频输入通道。2, PDC 和 GND 组成一个 PWM_DAC 输出；3, DAC 和 GND 组成一个 DAC 输出/ADC 输入（因为 DAC 脚也刚好也可以做 ADC 输入）；4, ADC 和 GND 组成一组 ADC 输入；5, TPAD 和 GND 组成一个触摸按键接口，可以连接其他板子实现触摸按键。

当使用 1 个跳线帽的时候：1, AIN 和 PDC 组成一个 MCU 的音频输出通道，实现 PWM DAC 播放音乐。2, AIN 和 DAC 同样可以组成一个 MCU 的音频输出通道，也可以用来播放音乐。3, DAC 和 ADC 组成一个自输出测试，用 MCU 的 ADC 来测试 MCU 的 DAC 输出。4, PDC 和 ADC，组成另外一个子输出测试，用 MCU 的 ADC 来测试 MCU 的 PWM DAC 输出。5, ADC 和 TPAD，组成一个触摸按键输入通道，实现 MCU 的触摸按键功能。

从上面的分析，可以看出，这个多功能端口可以实现 10 个功能，所以，只要设计合理，1+1 是大于 2 的。

2.1.27 音频选择

ALIENTEK 战舰 STM32 开发板板载了多个音频输出设备，所以需要一个音频选择电路，来实现不同音频的切换，这里我们使用 74HC4052 模拟开关来实现音频切换，其原理图如图 2.1.27.1 所示：

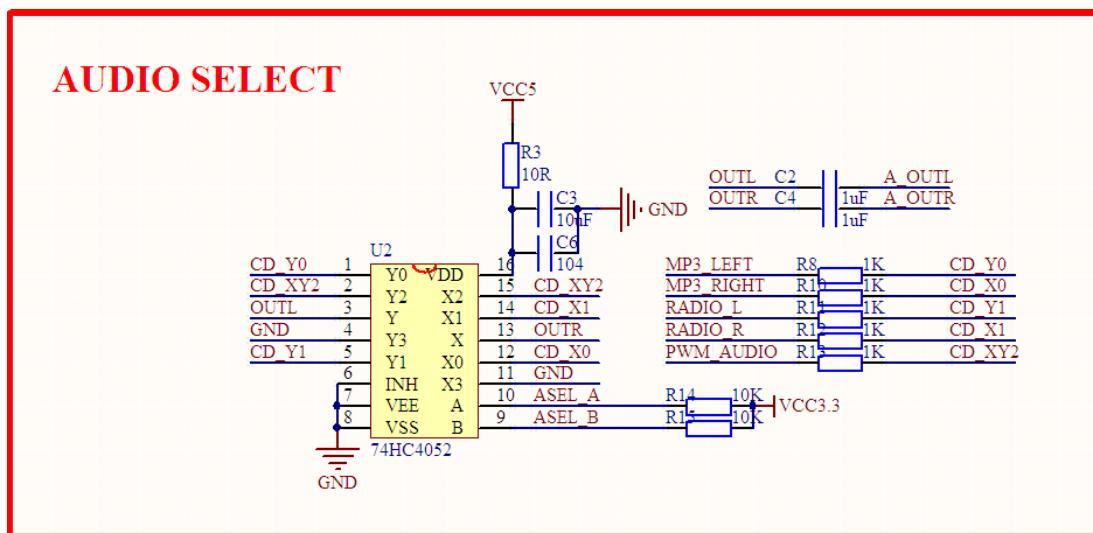


图 2.1.27.1 音频选择电路

74HC4052 是一个双 4 路模拟开关，工作电压可以低至 2V，这里我们选择该模拟开关来做音频切换。

图中 MP3_LEFT/MP3_RIGHT 连接在 VS1053 的音频输出端。RADIO_L 和 RADIO_R 是 RDA5820 的音频输出端。A_OUTR 和 A_OUTL 连接到 TDA1308 的输入端，最终输出到耳机。而 OUTL 和 OUTR 则还连接到了 RDA5820 的音频输入端，所以开发板的所有声音，其实都可以通过 FM 发射出去，大家可以在收音机里面听到来自开发板的声音。PWM_AUDIO 则是来自外部音源输入（我们提供的 USB 声卡实验，就需要用到这个通道）。ASEL_A 和 ASEL_B 直接连接在 MCU 的 PD7 和 PB7 上面，用来控制 74HC4052 的通道选择。



2.1.28 FM 收发

ALIENTEK 战舰 STM32 开发板板载了一颗 FM 收发芯片 RDA5820, 其原理图如图 2.1.28.1 所示:

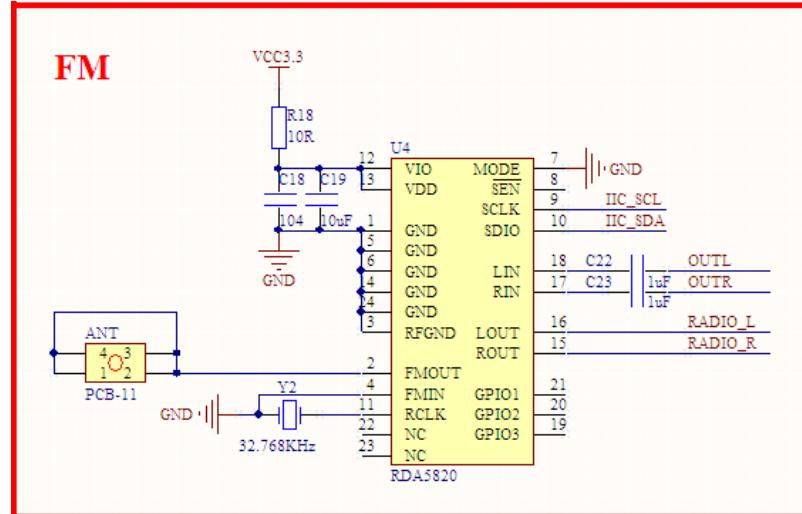


图 2.1.28.1 FM 收发电路

RDA5820 是一颗立体声 FM 收发芯片, 该芯片通过 IIC 接口控制, 可以实现 65~108MHz 的全球 FM 频段接收, 同时可以作为 FM 发射。RDA5820 接收与发送天线共用, 仅需要极少的外围器件即可正常工作。

图中 OUTL 和 OUTR 为 FM 发射的音频输入信号, RADIO_L 和 RADIO_R 是 FM 接收的音频输出信号, 连接到 74HC4052。同 24C02 一样, 该芯片的 IIC_SCL 和 IIC_SDA 同样是挂在 PB10 和 PB11 上, 他们共享一个 IIC 总线。

2.1.29 音频输出

ALIENTEK 战舰 STM32 开发板板载的音频输出电路, 其原理图如图 2.1.29.1 所示:

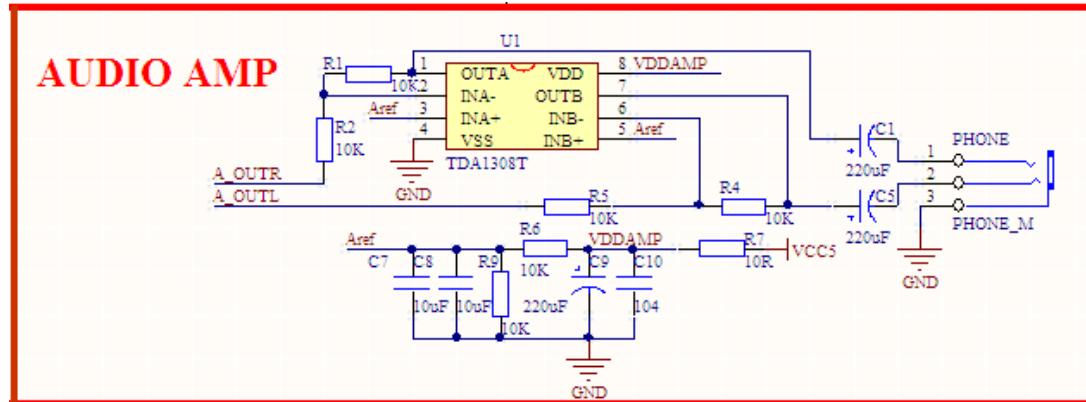


图 2.1.29.1 音频输出电路

图中 PHONE 为立体声音频输出插座, 可以直接插 3.5mm 的耳机。A_OUTR 和 A_OUTL 是来自 74HC4052 的音频输出信号, 直接输入到 TDA1308。

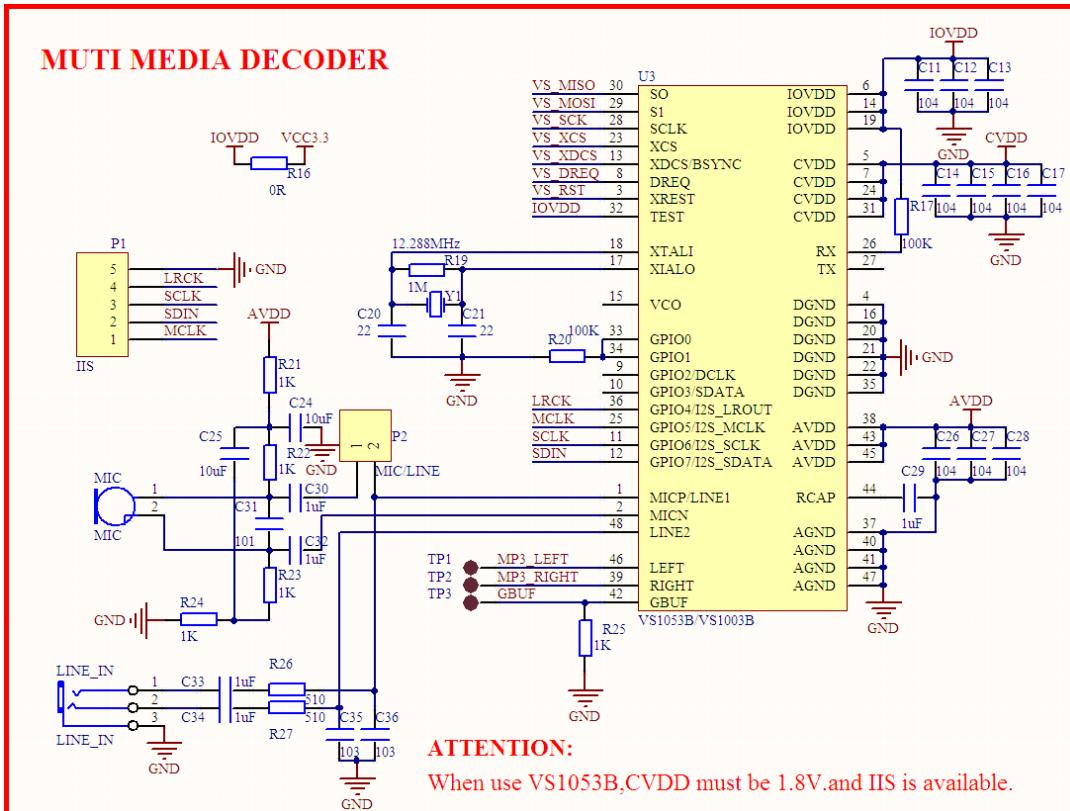
图中的 TDA1308 是 AB 类的数字音频(CD)专用耳机功放 IC。其具有低电压、低失真、高速率、强输出等优异的性能是以往的 TDA2822、TDA7050、LM386 等“经典”功放望尘莫及的。同时战舰 STM32 开发板搭载了效果一流的 VS1053 编解码芯片, 所以, 战舰 STM32 开发



板播放 MP3 的音质是非常不错的，胜过市面上很多中低端 MP3 的音质。

2.1.30 音频编解码

ALIENTEK 战舰 STM32 开发板板载 VS1053 音频编解码芯片，其原理图如图 2.1.30.1 所示：



2.1.30.1 音频编解码芯片

VS1053 是一颗单片 OGG/MP3/AAC/WMA/MIDI 音频解码器，通过 plugin 可以实现 FLAC 的解码，同时该芯片可以支持 IMA ADPCM 编码，通过 plugin 可以实现 OGG 编码。相比它的前辈：VS1003，VS1053 性能提升了不少，比如支持 OGG 编解码，支持 FLAC 解码，同时音质上也有比较大的提升，还支持空间效果设置。VS1053 还支持 IIS 输出，我们在开发板上引出了 IIS 接口（P1），通过这个接口，大家可以在外部接自己的 DAC，以达到更好的音质。

图中 P2 是 MIC 录音选择接口，这个接口主要在大家选择使用 LINE_IN 录音的时候，需要用到，断开 P2，就可以排除 MIC 对 LINE_IN 录音的干扰，从而达到更好的效果。默认我们是用跳线帽短接 P2 的。

图中 MP3_LEFT/MP3_RIGHT 这两个信号是连接到 74HC4052 的，通过模拟开关选择是否输出 MP3 音源。TP1/TP2/TP3 是 3 个测试点，用于测试。VS1053 通过 7 根线连接到 MCU，VS_MISO/VS_MOSI/VS_SCK/VS_XCS/VS_XDCS/VS_DREQ/VS_RST 这 7 根线分别连接到 MCU 的 PA6/PA7/PA5/PF7/PF6/PC13/PE6 上，VS1053 通过 STM32 的 SPI1 访问。

2.1.31 电源

ALIENTEK 战舰 STM32 开发板板载的电源供电部分，其原理图如图 2.1.31.1 所示：

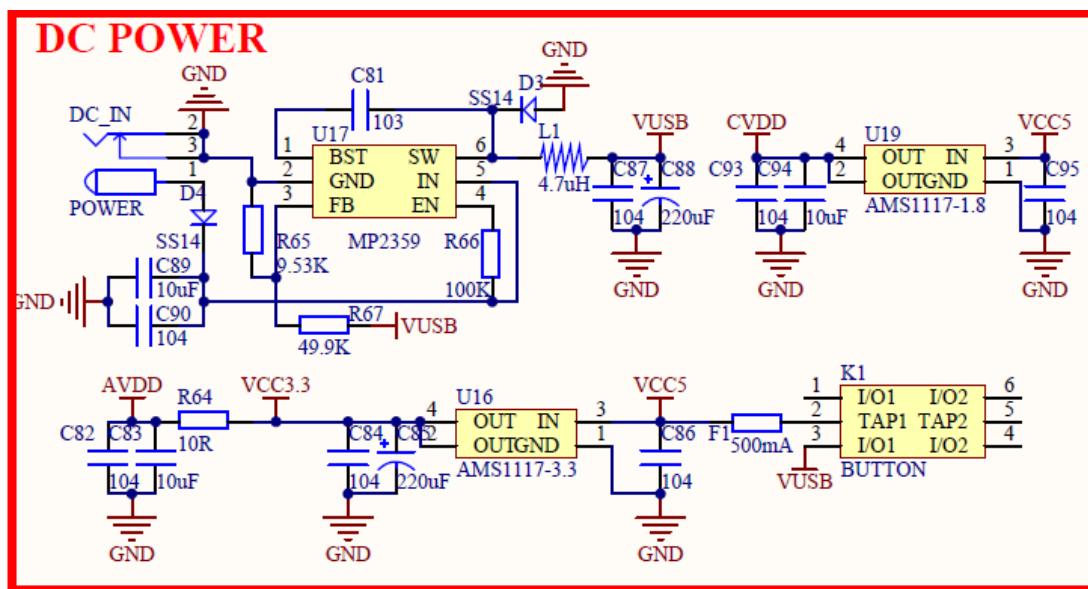


图 2.1.31.1 电源

图中，总共有3个稳压芯片：U16/U17/U19，DC_IN用于外部直流电源输入，经过U17 DC-DC芯片转换为5V电源输出，其中D4是防反接二极管，避免外部直流电源极性搞错的时候，烧坏开发板。K1为开发板的总电源开关，F1为500ma自恢复保险丝，用于保护电源。U16为3.3V稳压芯片，给开发板提供3.3V电源，而U19则是1.8V稳压芯片，供VS1053的CVDD使用。

这里还有 USB 供电部分没有列出来，其中 VUSB 就是来自 USB 供电部分，我们将在相应章节进行介绍。

2.1.32 电源输入输出接口

ALIENTEK 战舰 STM32 开发板板载了两组简单电源输入输出接口，其原理图如图 2.1.32.1 所示：

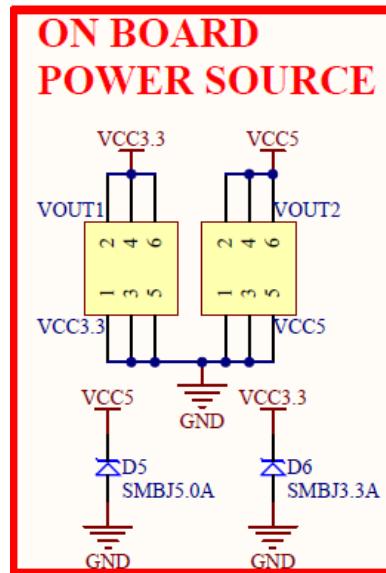


图 2.1.32.1 电源

图中，VOUT1 和 VOUT2 分别是 3.3V 和 5V 的电源输入输出接口，有了这 2 组接口，我们可以通过开发板给外部提供 3.3V 和 5V 电源了，虽然功率不大（最大 500ma），但是一般情况都够用了，大家在调试自己的小电路板的时候，有这两组电源还是比较方便的。同时这两组端



口，也可以用来由外部给开发板供电。

2.1.33 USB 串口

ALIENTEK 战舰 STM32 开发板板载了一个 USB 串口，其原理图如图 2.1.33.1 所示：

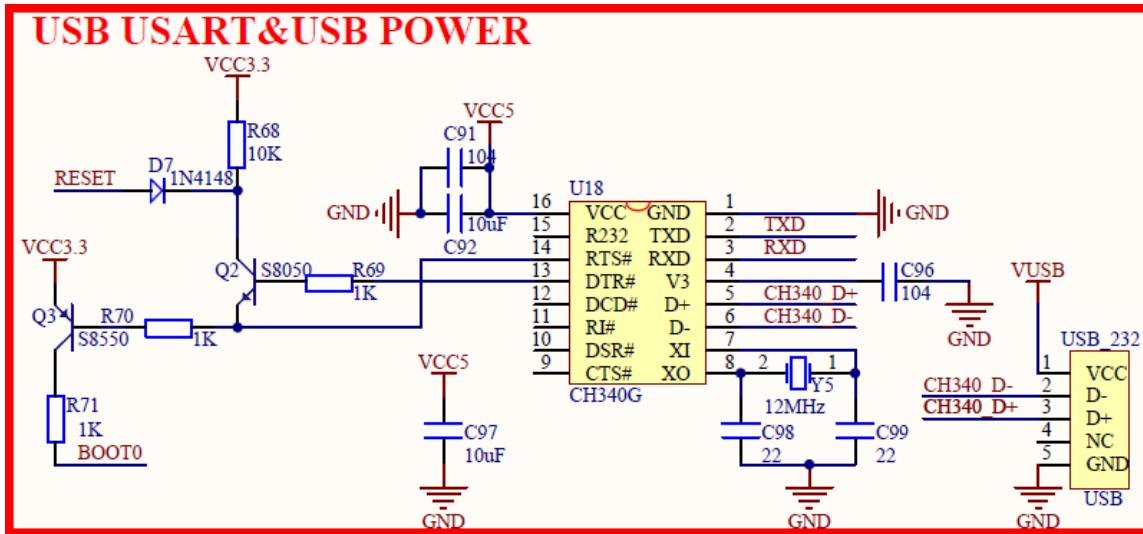


图 2.1.33.1 USB 串口

USB 转串口，我们选择的是 CH340G，我们的 Mini 板之前用的是 PL2303HX（后面会改为 CH340G），但是问题比较多，这次我们直接使用南京沁恒的 CH340G，稳定性经测试还不错，所以还是多支持下国产。

图中 Q2 和 Q3 的组合构成了我们开发板的一键下载电路，只需要在 mcuisp 软件设置：DTR 的低电平复位，RTS 高电平进 BootLoader。就可以一键下载代码了，而不需要手动设置 B0 和按复位了。其中，RESET 是开发板的复位信号，BOOT0 则是启动模式的 B0 信号。USB_232 是一个 MiniUSB 座，提供 CH340G 和电脑通信的接口，同时可以给开发板供电，VUSB 就是来自电脑 USB 的电源，USB_232 是本开发板的主要供电口。

2.2 开发板使用注意事项

为了让大家更好的使用 ALIENTEK 战舰 STM32 开发板，我们在这里总结该开发板使用的时候尤其要注意的一些问题，希望大家在使用的时候多多注意，以减少不必要的问题。

1. 开发板一般情况是由 USB_232 口供电，在第一次上电的时候由于 CH340G 在和电脑建立连接的过程中，导致 DTR/RTS 信号不稳定，会引起 STM32 复位 2~3 次左右，这个现象是正常的，后续按复位键就不会出现这种问题了。
2. 虽说开发板有 500mA 自恢复保险丝，但是由于自恢复保险丝是慢动作器件，所以在给外部供电的时候，还是请大家小心一点，不要超过这个限额，以免引起不必要的问题。
3. SPI2 被多个 SPI 器件共用 (SD 卡/无线模块/网络模块/W25Q64)，在使用的时候，必须保证同一时刻只有 1 个 SPI 器件是被选中的 (CS 为低)，其他器件必须设置为非选中 (CS 为高)，以免互相干扰。
4. JTAG 接口有几个信号 (JTDO/JTRST) 被摄像头模块/OLED 模块占用了，所以在调试这两个模块的时候，请大家选择 SWD 模式，其实最好就是一直用 SWD 模式。
5. 当你想使用某个 IO 口用作其他用处的时候，请先看看开发板的原理图，该 IO 口是否



有连接在开发板的某个外设上，如果有，该外设的这个信号是否会对你的使用造成干扰，先确定无干扰，再使用这个 IO。比如 PB8 就不怎么适合再用做其他输出，因为他接了蜂鸣器，如果你输出高电平就会听到蜂鸣器的叫声了。

- 6, 开发板上的跳线帽比较多，大家在使用某个功能的时候，要先查查这个是否需要设置跳线帽，以免浪费时间。
- 7, 当液晶显示白屏的时候，请先检查液晶模块是否插好（拔下来重新插试试），如果还不行，可以通过串口看看 LCD ID 是否正常，再做进一步的分析。
- 8, 开发板有 2 个 DB9 接口，但是请特别注意，只有 COM 这个接口（PS/2 右侧的）可以用来连接外部串口，而 JOY_PAD 这个接口（PS/2 左侧）是用来接手柄的，不能接串口，否则可能把 STM32 给烧了！请大家一定要注意这个问题!!

至此，本指南的实验平台（ALIENTEK 战舰 STM32 开发板）的硬件部分就介绍完了，了解了整个硬件对我们后面的学习会有很大帮助，有助于理解后面的代码，在编写软件的时候，可以事半功倍，希望大家细读！另外 ALIENTEK 开发板的其他资料及教程更新，都可以在技术论坛 www.openedv.com 下载到，大家可以经常去这个论坛获取更新的信息。

第二篇 软件篇

上一篇，我们介绍了本指南的实验平台，本篇我们将详细介绍 STM32 的开发软件：RVMDK。通过该篇的学习，你将了解到：1、如何在 RVMDK 下新建 STM32 工程；2、工程的编译；3、RVMDK 的一些使用技巧；4、软件仿真；5、程序下载；6、在线调试；以上几个环节概括了了一个完整的 STM32 开发流程。本篇将图文并茂的向大家介绍以上几个方面，通过本篇的学习，希望大家能掌握 STM32 的开发流程，并能独立开始 STM32 的编程和学习。

在学习 STM32 库函数之前，请大家准备如下资料，这些资料在我们光盘中都有：

1. STM32 固件库 V3.5 文件包： STM32F10x_StdPeriph_Lib_V3.5.0
光盘目录： 软件资料\STM32 固件库使用参考资料\
 2. STM32F10x_StdPeriph_Driver_3.5.0(中文版).chm： 中文版的固件库使用手册。
光盘目录： 软件资料\STM32 固件库使用参考资料\
 3. 《STM32 中文参考手册 V10》： 这个资料很重要，很多概念都在这个资料中。
光盘目录： STM32 参考资料/ STM32 中文参考手册_V10.pdf
 4. 《Cortex-M3 权威指南 Cn》这个讲解了很多 CM3 底层的东西。
光盘目录： STM32 参考资料/Cortex-M3 权威指南(中文).pdf
 5. MDK3.8a 软件： 这是编程编译软件。
光盘目录： 软件资料\软件\MDK3.80A
- 本篇将分为如下 3 个章节：
- 2.1, RVMDK 软件入门；
 - 2.2, STM32 开发基础知识入门；
 - 2.3, SYSTEM 文件介绍；



第三章 RVMDK 软件入门

本章将向大家介绍 RVMDK 软件的使用，ST 官方固件库介绍，同时还介绍怎样建一个基于 STM32 官方固件库 V3.5 的工程模板。通过本章的学习，我们最终将建立一个自己的 RVMDK 工程，最后本章还将向大家介绍 RVMDK 软件的一些使用技巧，希望大家在本章之后，能够对 RVMDK 这个软件有个比较全面的了解。

本章分为如下个小结：

- 3.1,STM32 官方固件库简介；
- 3.2,RVMDK3.80A 简介；
- 3.3,新建基于固件库的 RVMDK 工程模板；
- 3.4,MDK 下的程序下载与调试；
- 3.5,MDK 使用技巧；

3.1 STM32 官方固件库简介

ST(意法半导体)为了方便用户开发程序，提供了一套丰富的 STM32 固件库。到底什么是固件库？它与直接操作寄存器开发有什么区别和联系？很多初学用户很是费解，这一节，我们将讲解 STM32 固件库相关的基础知识，希望能够让大家对 STM32 固件库有一个初步的了解，至于固件库的详细使用方法，我们会在后面的章节一一介绍。这章节有一些图片是截图的权威手册。这一节的知识可以参考《STM32 固件库使用手册中文翻译版》P32，固件库手册讲解更加详细，这里只是提到一下，希望大家谅解。

官方包的地址：软件资料\STM32 固件库使用参考资料\ STM32F10x_StdPeriph_Lib_V3.5.0

3.1.1 库开发与寄存器开发的关系

很多用户都是从学 51 单片机开发转而想进一步学习 STM32 开发，他们习惯了 51 单片机的寄存器开发方式，突然一个 ST 官方库摆在面前会一头雾水，不知道从何下手。下面我们将通过一个简单的例子来告诉 STM32 固件库到底是什么，和寄存器开发有什么关系？其实一句话就可以概括：固件库就是函数的集合，固件库函数的作用是向下负责与寄存器直接打交道，向上提供用户函数调用的接口（API）。

在 51 的开发中我们常常的作法是直接操作寄存器，比如要控制某些 IO 口的状态，我们直接操作寄存器：

```
P0=0x11;
```

而在 STM32 的开发中，我们同样可以操作寄存器：

```
GPIOx->BRR = 0x0011;
```

这种方法当然可以，但是这种方法的劣势是你需要去掌握每个寄存器的用法，你才能正确使用 STM32，而对于 STM32 这种级别的 MCU，数百个寄存器记起来又是谈何容易。于是 ST(意法半导体)推出了官方固件库，固件库将这些寄存器底层操作都封装起来，提供一整套接口（API）供开发者调用，大多数场合下，你不需要去知道操作的是哪个寄存器，你只需要知道调用哪些函数即可。

比如上面的控制 BRR 寄存器实现电平控制，官方库封装了一个函数：

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

```
{
```



```

GPIOx->BRR = GPIO_Pin;
}

```

这个时候你不需要再直接去操作 BRR 寄存器了，你只需要知道怎么使用 GPIO_ResetBits()这个函数就可以了。在你对外设的工作原理有一定的了解之后，你再去看固件库函数，基本上函数名字能告诉你这个函数的功能是什么，该怎么使用，这样是不是开发会方便很多？

任何处理器，不管它有多么的高级，归根结底都是要对处理器的寄存器进行操作。但是固件库不是万能的，您如果想要把 STM32 学透，光读 STM32 固件库是远远不够的。你还是要了解一下 STM32 的原理，而这些原理了解了，你在进行固件库开发过程中才可能得心应手游刃有余。

3.1.2 STM32 固件库与 CMSIS 标准讲解

前一节我们讲到，STM32 固件库就是函数的集合，那么对这些函数有什么要求呢？？这里就涉及到一个 CMSIS 标准的基础知识，这部分知识可以从《Cortex-M3 权威指南》中了解到，我们这里只是对权威指南的讲解做个概括性的介绍。经常有人问到 STM32 和 ARM 以及 ARM7 是什么关系这样的问题，其实 ARM 是一个做芯片标准的公司，它负责的是芯片内核的架构设计，而 TI, ST 这样的公司，他们并不做标准，他们是芯片公司，他们是根据 ARM 公司提供的芯片内核标准设计自己的芯片。所以，任何一个做 Cortex-M3 芯片，他们的内核结构都是一样的，不同的是他们的存储器容量，片上外设，IO 以及其他模块的区别。所以你会发现，不同公司设计的 Cortex-M3 芯片他们的端口数量，串口数量，控制方法这些都是有区别的，这些资源他们可以根据自己的需求理念来设计。同一家公司设计的多种 Cortex-m3 内核芯片的片上外设也会有很大的区别，比如 STM32F103RBT 和 STM32F103ZET，他们的片上外设就有很大的区别。我们可以通过《Cortex-M3 权威指南》中的一个图来了解一下：

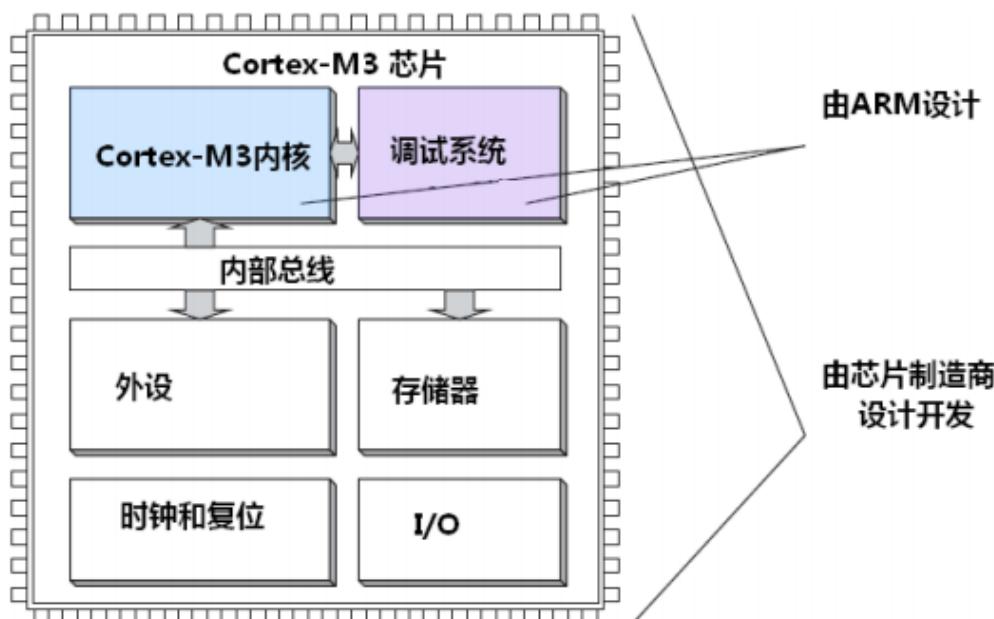


图 3.1.2.1 Cortex-m3 芯片结构

从上图可以看出，芯片虽然是芯片公司设计，但是内核却要服从 ARM 公司提出的 Cortex-M3 内核标准了，理所当然，芯片公司每卖出一片芯片，需要向 ARM 公司交一定的专利费。



既然大家都使用的是 Cortex-M3 核，也就是说，本质上大家都是一样的，这样 ARM 公司为了能让不同的芯片公司生产的 Cortex-M3 芯片能在软件上基本兼容，和芯片生产商共同提出了一套标准 CMSIS 标准(Cortex Microcontroller Software Interface Standard)，翻译过来是“ARM Cortex™ 微控制器软件接口标准”。ST 官方库就是根据这套标准设计的。这里我们又要引用参考资料里面的图片来看看基于 CMSIS 应用程序基本结构：

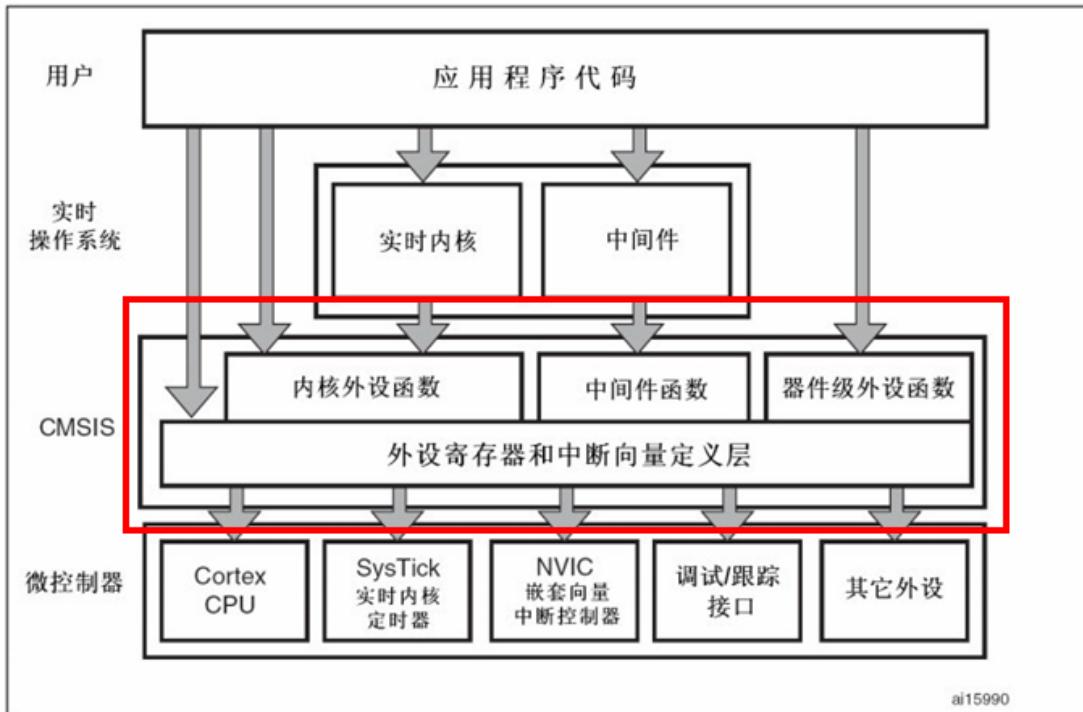


图 3.1.2.2 基于 CMSIS 应用程序基本结构

CMSIS 分为 3 个基本功能层：

- 1) 核内外设访问层：ARM 公司提供的访问，定义处理器内部寄存器地址以及功能函数。
- 2) 中间件访问层：定义访问中间件的通用 API，也是 ARM 公司提供。
- 3) 外设访问层：定义硬件寄存器的地址以及外设的访问函数。

从图中可以看出，CMSIS 层在整个系统中是处于中间层，向下负责与内核和各个外设直接打交道，向上提供实时操作系统用户程序调用的函数接口。如果没有 CMSIS 标准，那么各个芯片公司就会设计自己喜欢的风格的库函数，而 CMSIS 标准就是要强制规定，芯片生产公司设计的库函数必须按照 CMSIS 这套规范来设计。

其实不用这么讲这么复杂的，一个简单的例子，我们在使用 STM32 芯片的时候首先要进行系统初始化，CMSIS 规范就规定，系统初始化函数名字必须为 SystemInit，所以各个芯片公司写自己的库函数的时候就必须用 SystemInit 对系统进行初始化。CMSIS 还对各个外设驱动文件的文件名字规范化，以及函数名字规范化等一系列规定。上一节讲的函数 GPIO_ResetBits 这个函数名字也是不能随便定义的，是要遵循 CMSIS 规范的。

至于 CMSIS 的具体内容就不必多讲了，需要了解详细的朋友可以到网上搜索资料，相关资料可谓满天飞。

3.1.3 STM32 官方库包介绍

这一节内容主要讲解 ST 官方提供的 STM32 固件库包的结构。ST 官方提供的固件库完整包

可以在官方下载，我们光盘也会提供。固件库是不断完善升级的，所以有不同的版本，我们使用的是 V3.5 版本的固件库

大家可以看到光盘目录：软件资料\STM32 固件库使用参考资料\

STM32F10x_StdPeriph_Lib_V3.5.0 下面查看，这在我们论坛有下载。下面看看官方库包的目录结构：

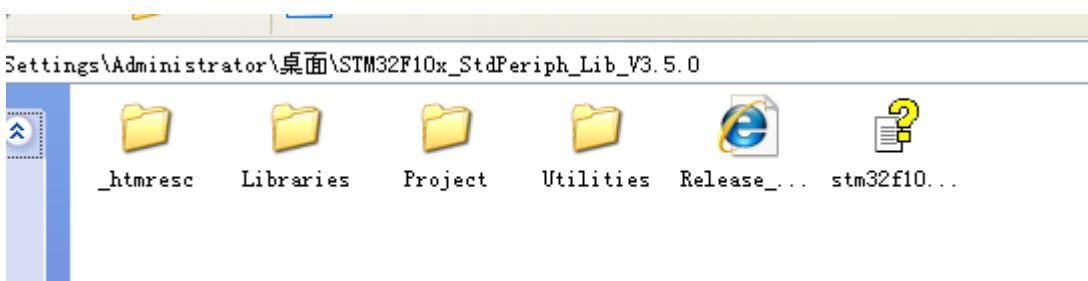


图 3.1.2.3 官方库包根目录

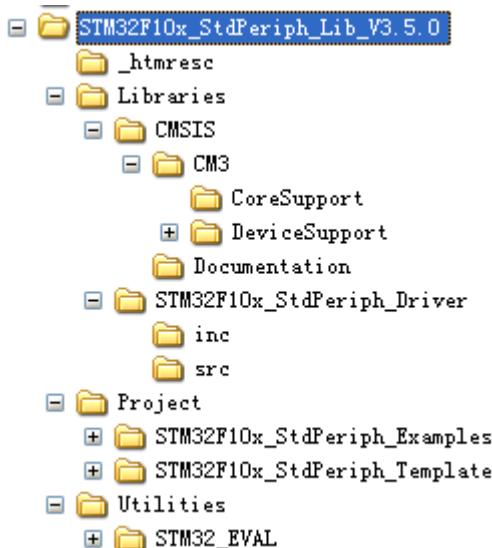


图 3.1.2.4 官方库目录列表

3.1.3.1 文件夹介绍：

Libraries 文件夹下面有 CMSIS 和 STM32F10x_StdPeriph_Driver 两个目录，这两个目录包含固件库核心的所有子文件夹和文件。其中 CMSIS 目录下面是启动文件，STM32F10x_StdPeriph_Driver 放的是 STM32 固件库源码文件。源文件目录下面的 inc 目录存放的是 stm32f10x_xxx.h 头文件，无需改动。src 目录下面放的是 stm32f10x_xxx.c 格式的固件库源码文件。每一个.c 文件和一个相应的.h 文件对应。这里的文件也是固件库的核心文件，每个外设对应一组文件。

Libraries 文件夹里面的文件在我们建立工程的时候都会使用到。

Project 文件夹下面有两个文件夹。顾名思义，STM32F10x_StdPeriph_Examples 文件夹下面存放的 ST 官方提供的固件实例源码，在以后的开发过程中，可以参考修改这个官方提供的实例来快速驱动自己的外设，很多开发板的实例都参考了官方提供的例程源码，这些源码对以后的学习非常重要。STM32F10x_StdPeriph_Template 文件夹下面存放的是工程模板。

Utilities 文件下就是官方评估板的一些对应源码，这个可以忽略不看。



根目录中还有一个 `stm32f10x_stdperiph_lib_um.chm` 文件，直接打开可以知道，这是一个固件库的帮助文档，这个文档非常有用，只可惜是英文的，在开发过程中，这个文档会经常被使用到。

3.1.3.2 关键文件介绍：

下面我们要着重介绍 Libraries 目录下面几个重要的文件。

`core_cm3.c` 和 `core_cm3.h` 文件位于 `\Libraries\CMSIS\CM3\CoreSupport` 目录下面的，这个就是 CMSIS 核心文件，提供进入 M3 内核接口，这是 ARM 公司提供，对所有 CM3 内核的芯片都一样。你永远都不需要修改这个文件，所以这里我们就点到为止。

和 CoreSupport 同一级还有一个 DeviceSupport 文件夹。`DeviceSupport\ST\STM32F10xt` 文件夹下面主要存放一些启动文件以及比较基础的寄存器定义以及中断向量定义的文件。

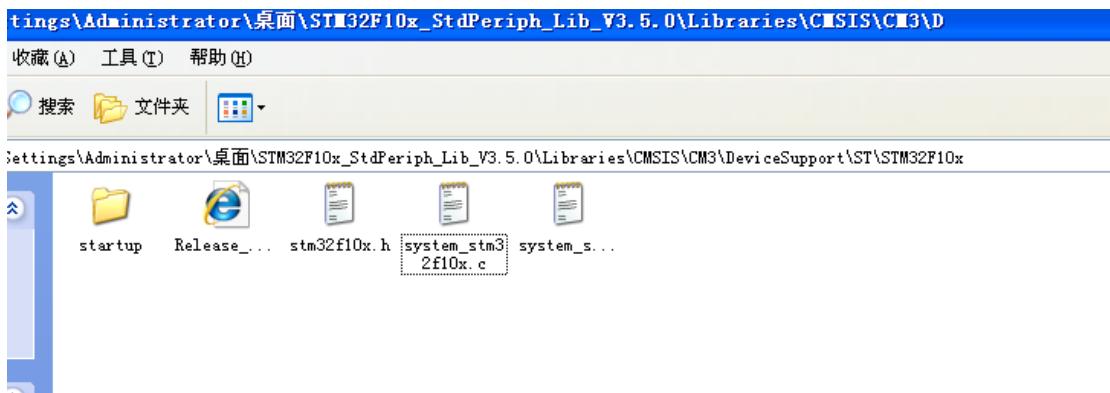


图 3.1.2.5 DeviceSupport\ST\STM32F10x 目录结构

这个目录下面有三个文件：`system_stm32f10x.c`, `system_stm32f10x.h` 以及 `stm32f10x.h` 文件。其中 `system_stm32f10x.c` 和对应的头文件 `system_stm32f10x.h` 文件的功能是设置系统以及总线时钟，这个里面有一个非常重要的 `SystemInit()` 函数，这个函数在我们系统启动的时候都会调用，用来设置系统的整个时钟系统。

`stm32f10x.h` 这个文件就相当重要了，只要你做 STM32 开发，你几乎时刻都要查看这个文件相关的定义。这个文件打开可以看到，里面非常多的结构体以及宏定义。这个文件里面主要是系统寄存器定义申明以及包装内存操作，对于这里是怎样申明以及怎样将内存操作封装起来的，我们在后面的章节“MDK 中寄存器地址名称映射分析”中会讲到。

在 `DeviceSupport\ST\STM32F10x` 同一级还有一个 `startup` 文件夹，这个文件夹里面放的文件顾名思义是启动文件。在 `\startup\arm` 目录下，我们可以看到 8 个 `startup` 开头的.s 文件。

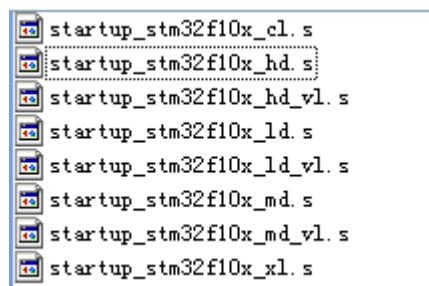




图 3.1.2.6 startup 文件

这里之所以有 8 个启动文件，是因为对于不同容量的芯片启动文件不一样。对于 103 系列，主要是用其中 3 个启动文件：

startup_stm32f10x_ld.s：适用于小容量产品
startup_stm32f10x_md.s：适用于中等容量产品
startup_stm32f10x_hd.s：适用于大容量产品

这里的容量是指 FLASH 的大小.判断方法如下：

小容量：FLASH≤32K
中容量：64K≤FLASH≤128K
大容量：256K≤FLASH

我们 ALIENTEK STM32 战舰版采用的 103ZET6 是属于大容量产品，所以我们的启动文件选择 startup_stm32f10x_hd.s, 而我们的 mini 板子采用的 103RBT6 是中等容量芯片，所以选择 startup_stm32f10x_md.s 启动文件。

启动文件到底什么作用，其实我们可以打开启动文件进去看看。启动文件主要是进行堆栈之类的初始化，中断向量表以及中断函数定义。启动文件要引导进入 main 函数。Reset_Handler 中断函数是唯一实现了的中断处理函数，其他的中断函数基本都是死循环。Reset_handler 在我们系统启动的时候会调用，下面让我们看看 Reset_handler 这段代码：

```
; Reset handler
Reset_Handler    PROC
    EXPORT  Reset_Handler          [WEAK]
    IMPORT  __main
    IMPORT  SystemInit
    LDR    R0, =SystemInit
    BLX    R0
    LDR    R0,=__main
    BX    R0
ENDP
```

这段代码我也看不懂，反正就知道，这里面要引导进入 main 函数，同时在进入 main 函数之前，首先要调用 SystemInit 系统初始化函数。

还有其他几个文件 stm32f10x_it.c,stm32f10x_it.h 以及 stm32f10x_conf.h 等文件，这里就不一一介绍。stm32f10x_it.c 里面是用来编写中断服务函数，中断服务函数也可以随意编写在工程里面的任意一个文件里面，个人觉得这个文件没太大意义。

stm32f10x_conf.h 文件打开可以看到一堆的#include, 这里你建立工程的时候，可以注释掉一些你不用的外设头文件。这里相信大家一看就明白。

这一节我们就简要介绍到这里，后面我们会介绍怎样建立基于 V3.5 版本固件库的工程模块。

3.2 RVMDK3.80A 简介

RVMDK 源自德国的 KEIL 公司，是 RealView MDK 的简称。在全球 RVMDK 被超过 10 万的嵌入式开发工程师使用，RealView MDK 集成了业内最领先的技术，包括 μ Vision3 集成开发环境与 RealView 编译器。支持 ARM7、ARM9 和最新的 Cortex-M3 核处理器，自动配置启动代码，集成 Flash 烧写模块，强大的 Simulation 设备模拟，性能分析等功能。与 ARM 之前的工

具包 ADS1.2 相比, RealView 编译器具有代更小、性能更高的优点, RealView 编译器与 ADS.2 的比较:

代码密度: 比 ADS1.2 编译的代码尺寸小 10%;

代码性能: 比 ADS1.2 编译的代码性能提高 20%;

目前 RVMDK 的最新版本是 RVMDK4.6, 4.0 以上的版本的 RVMDK 对 IDE 界面进行了很大改变, 并且支持 Cortex-M0 内核的处理器。作者曾用过 RVMDK3.24/3.80A/4.10 等几个版本, 并对他们进行了一些简单的对比, 从对比情况来看: 3.24 和 3.80a 在各方面的比较都差不多, 当然有人说稳定性 3.80A 要好一些, 这个我不做评论。但是 4.10 确实界面是好了, 支持的器件也多了, 但编译效率没有 3.24/3.80A 高, 尤其在编译后的代码执行速度 (FFT 运算), 4.10 要对速度进行-O2 优化才能和 3.24/3.80A 的普通级别相比。另外, 国内大都数单片机工程师都接触和使用过 KEIL, 相信大家都知道 KEIL 的使用是非常简单的, 而且很容易上手。RVMDK3.80A 的编译器界面和 KEIL 十分相似, 对于使用过 KEIL 的朋友来说, 更容易上手。基于以上几点, 本书将选择 RVMDK3.80A 版本的编译器作为学习 STM32 的软件。当然大家也可以根据自己的喜好换用 4.10 或以上版本的软件 (注意最新版本的 MDK 可能将你的山寨 JLINK 刷成砖头, 请慎重考虑)。

3.3 新建基于固件库的 RVMDK 工程模板

在前面的章节我们介绍了 STM32 官方库包的一些知识, 这些我们将着重讲解建立基于固件库的工程模板的详细步骤。在此之前, 首先我们要准备如下资料:

1) V3.5 固件库包: STM32F10x_StdPeriph_Lib_V3.5.0 这是 ST 官网下载的固件库完整版, 我们光盘目录:

软件资料\STM32 固件库使用参考资料\STM32F10x_StdPeriph_Lib_V3.5.0

我们官方论坛下载地址: <http://opendev.com/posts/list/6054.htm>

2) MDK3.8a 开发环境(我们的板子的开发环境目前是使用这个版本)。这在我们光盘的软件目录下面有安装包: 软件资料\软件\MDK3.80A

3) MDK 注册机, 这在我们光盘的 MDK 同一目录下面有。

光盘目录: 软件资料\软件\MDK3.80A\注册.exe

3.3.1 MDK3.8a 安装步骤

1) 找到 MDK 的安装文件并点击图标  , 这是 MDK 的安装文件, 和安装其他软件一样, 相信大家都会明白, 一直点击 Next 直到出现下面界面后, 随意填写好您的信息, 这些信息其实没啥要求, 可以随意填写, 然后点击 Next。

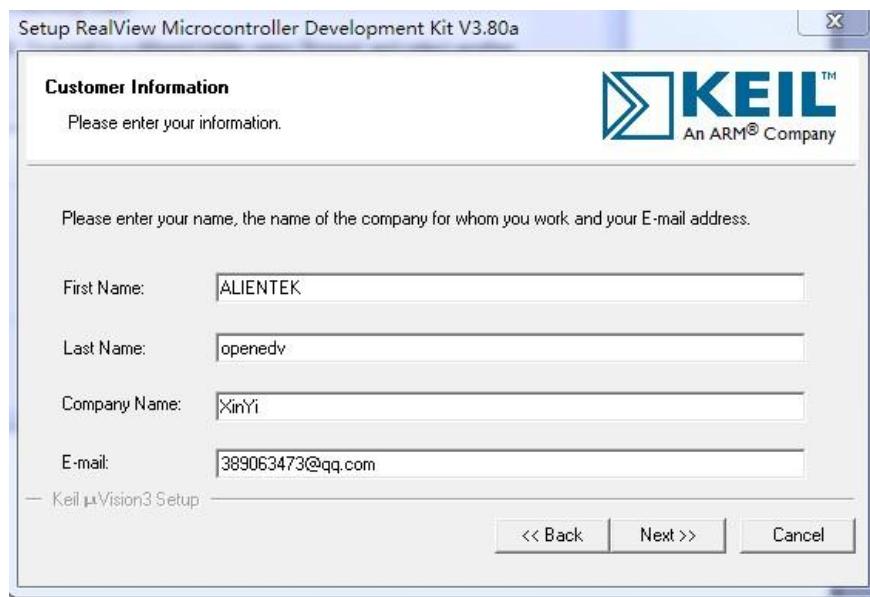


图 3.3.1.1MDK 填写用户信息

2) 接着出现下面的界面，按图中所示选择之后点击“Finish”之后，MDK 便安装完成。

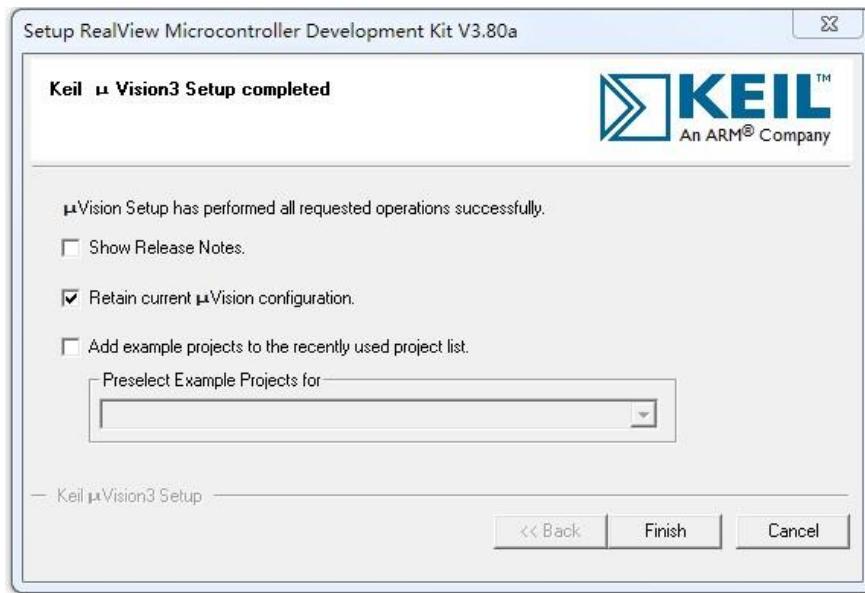


图 3.3.1.2 安装完成

3.3.2 添加 License Key

MDK 针对每台机会有一个 CID，copy 这个 CID 到注册机处生成 License Key，然后再将这个 License Key 添加到 MDK 里面去注册。详细步骤后面会一一介绍。

- 1) 打开运行 MDK。这里要注意，有些版本的 windows 系统(如 Vista)需要右键点击快捷方式选择“以管理员身份运行”，因为注册 license 需要管理员权限。打开 MDK 后有一个名字叫“LPC2129 simulator”的默认 Project，暂时我们可以不用理会。
- 2) 点击：File→License Management，弹出一个 License Management 界面，copy 界面中的(CID)：

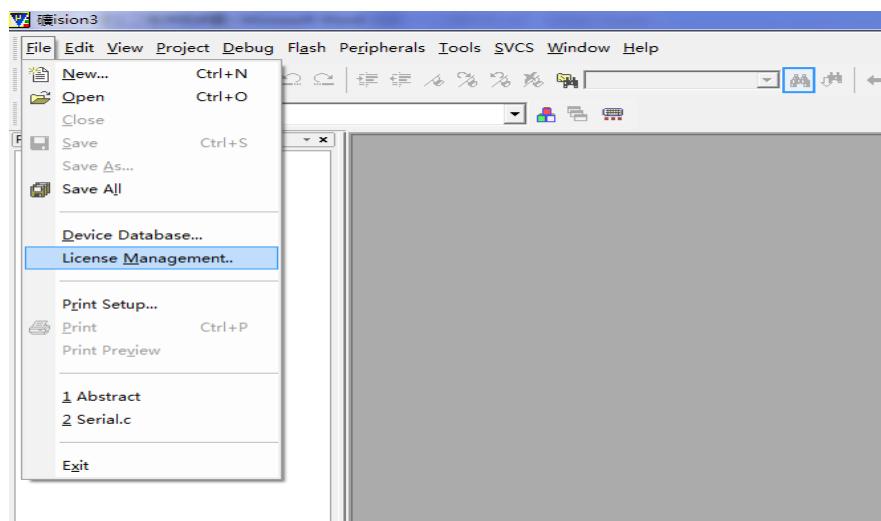


图 3.3.2.1 License Management 选择

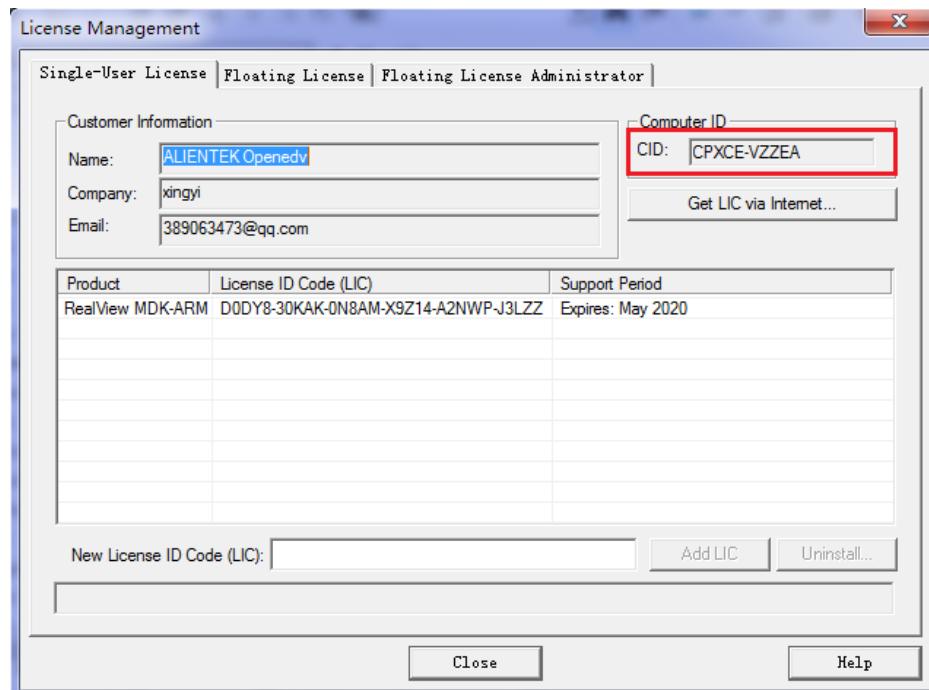


图 3.3.2.2 获取 CID

- 3) 打开光盘(软件资料\软件\MDK3.80A\注册.exe)下面的注册机 ，注册机我们会跟 MDK 安装包放在同一目录下面。
- 4) 接着会出现注册界面，黏贴刚才复制的 CID 到 CID 输入框，然后 Target 选择 ARM 之后，点击“Generate”，30 位的 License Key 会在下图红色圈出的部分生成。License Key 的格式：DODY8-30KAK-ON8AM-X9Z14-A2NWP-J3LZZ。

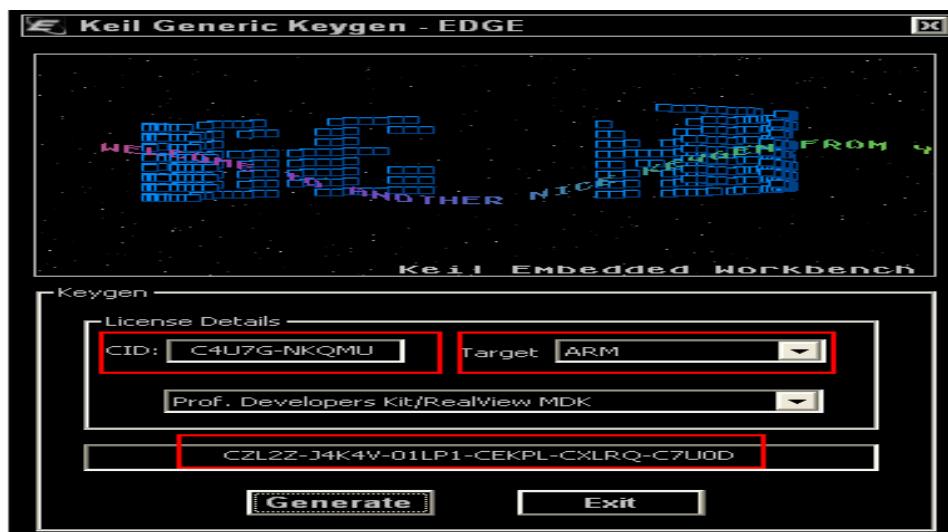


图 3.3.2.3 生成 License Key

- 5) 将这个 License Key 黏贴到 Keil 的 License Management 界面的 New License Id Code 一栏，然后点击“Add LIC”，添加成功后会出现成功提示。然后点击 Close 关闭这个界面即可。到此 License Key 便添加完成。

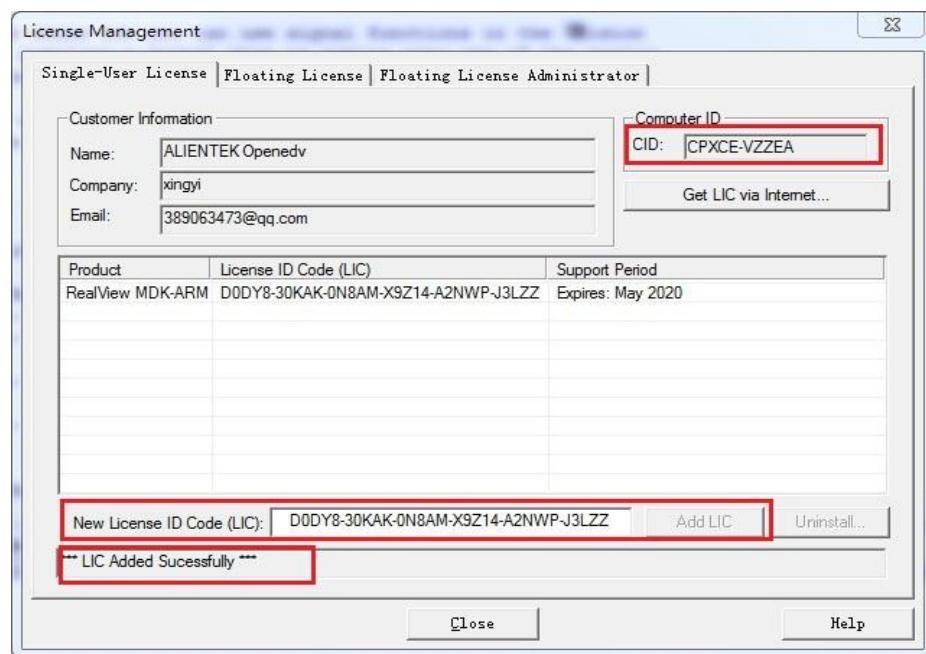


图 3.3.2.4 添加 License Key 成功

3.3.3 新建工程模板

- 回到 MDK 主界面，可以看到工程中有一个默认的工程，点击这个工程名字，然后选择菜单 Project->Close Project，就关闭掉这个工程了！这样整个 MDK 就是一个空的了，接下来我们将建立我们的工程模版。
- 在建立工程之前，我们建议用户在电脑的某个目录下面建立一个文件夹，后面所建立的工



程都可以放在这个文件夹下面，这里我们建立一个文件夹为 Template。

- 3) 点击 Keil 的菜单: Project ->New Uvision Project , 然后将目录定位到刚才建立的文件夹 Template 之下，在这个目录下面建立子文件夹 USER(我们的代码工程文件都是放在 USER 目录，很多人喜欢新建“Project”目录放在下面，这也是可以的，这个就看个人喜好了), 然后定位到 USER 目录下面，我们的工程文件就都保存到 USER 文件夹下面。工程命名为 Template,点击保存。

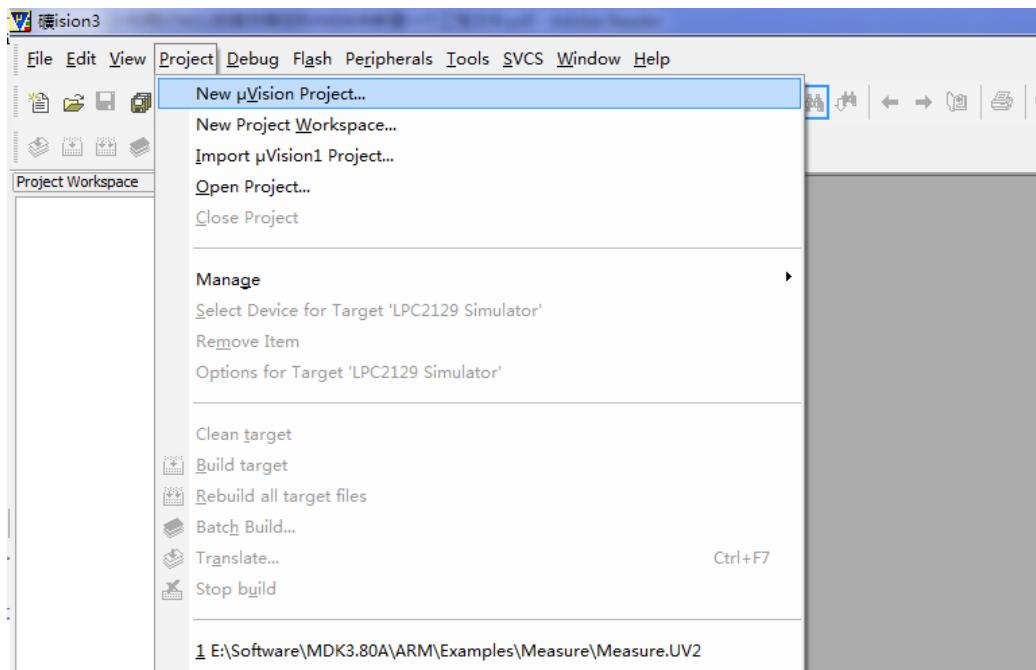


图 3.3.3.1 新建工程

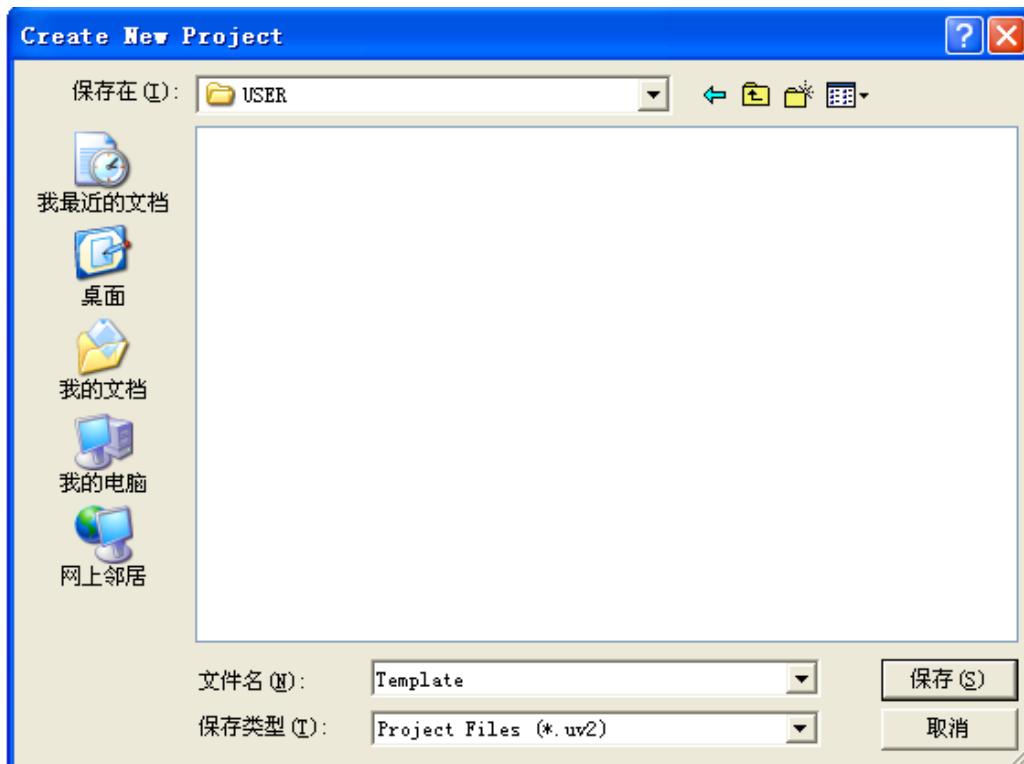


图 3.3.3.2 定义工程名称

- 4) 接下来会出现一个选择 Device 的界面，就是选择我们的芯片型号，这里我们定位到 STMicroelectronics 下面的 STM32F103ZE(针对我们的战舰板子是这个型号，如果是其他芯片，请选择对应的型号即可)。

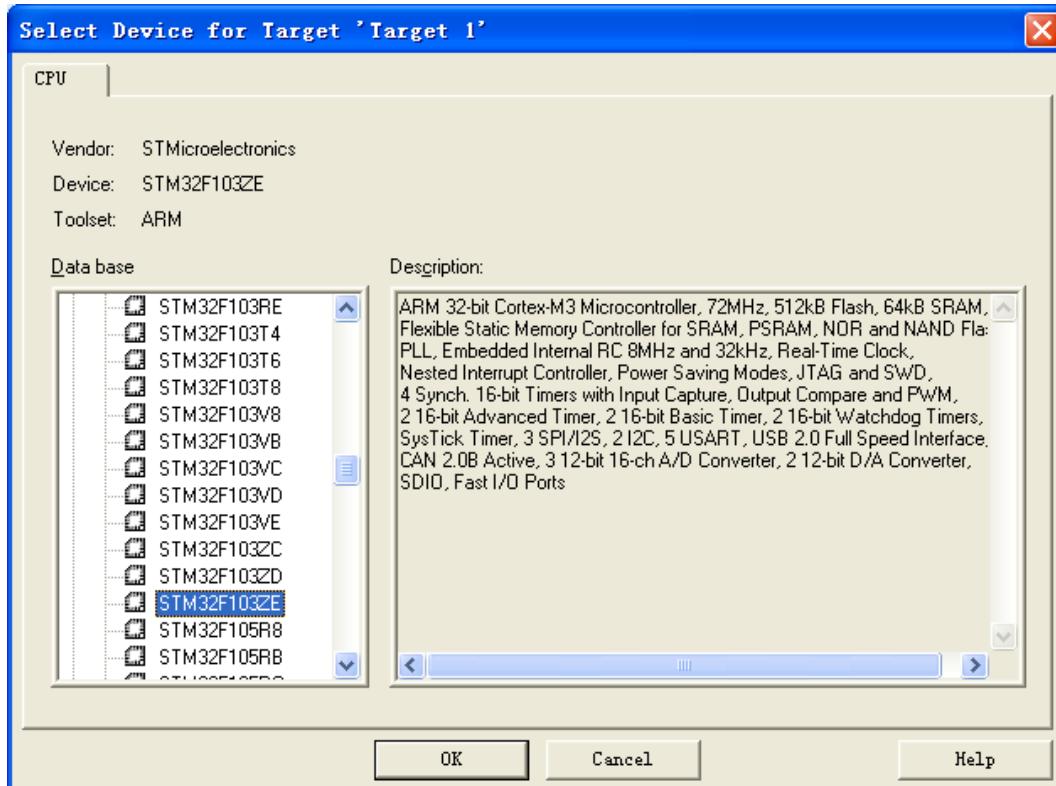




图 3.3.3.3 选择芯片型号

- 5) 弹出对话框“Copy STM32 Startup Code to project”，询问是否添加启动代码到我们的工程中，这里我们选择“否”，因为我们使用的 ST 固件库文件已经包含了启动文件。
- 6) 现在我们看看 USER 目录下面包含三个文件：

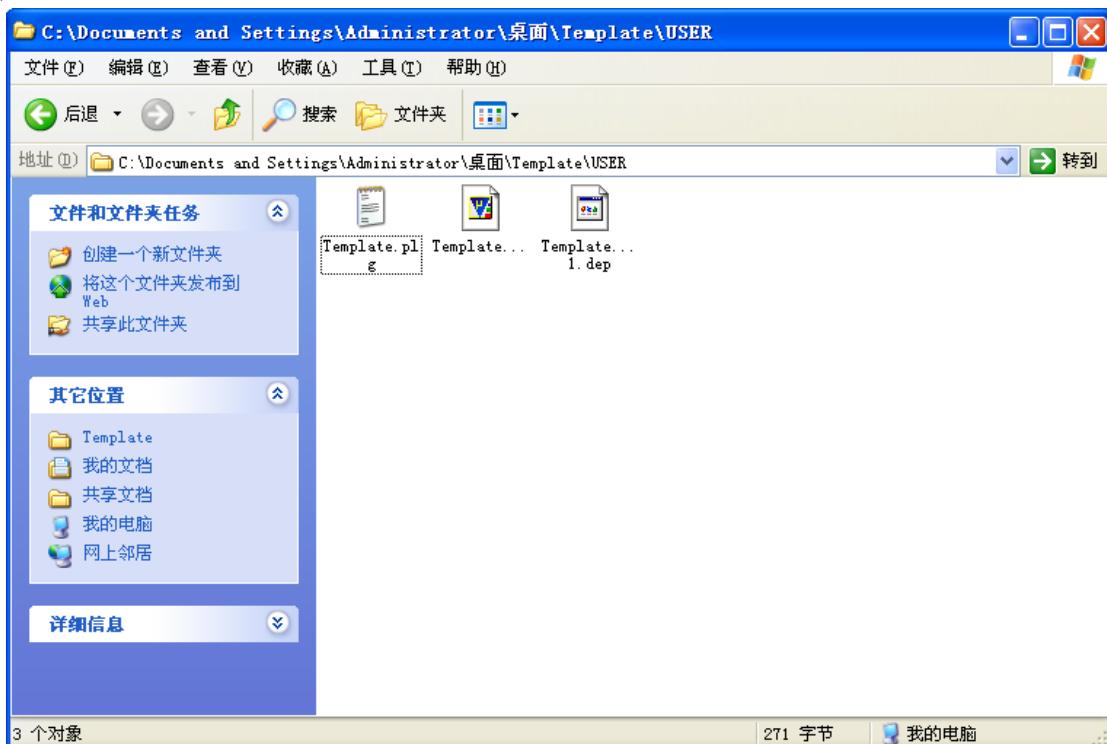


图 3.3.3.4 工程 USER 目录文件

- 7) 接下来，我们在 Template 工程目录下面，新建 3 个文件夹 CORE, OBJ 以及 STM32F10x_FWLib。CORE 用来存放核心文件和启动文件，OBJ 是用来存放编译过程文件以及 hex 文件，STM32F10x_FWLib 文件夹顾名思义用来存放 ST 官方提供的库函数源码文件。已有的 USER 目录除了用来放工程文件外，还用来存放主函数文件 main.c,以及其他包括 system_stm32f10x.c 等等。

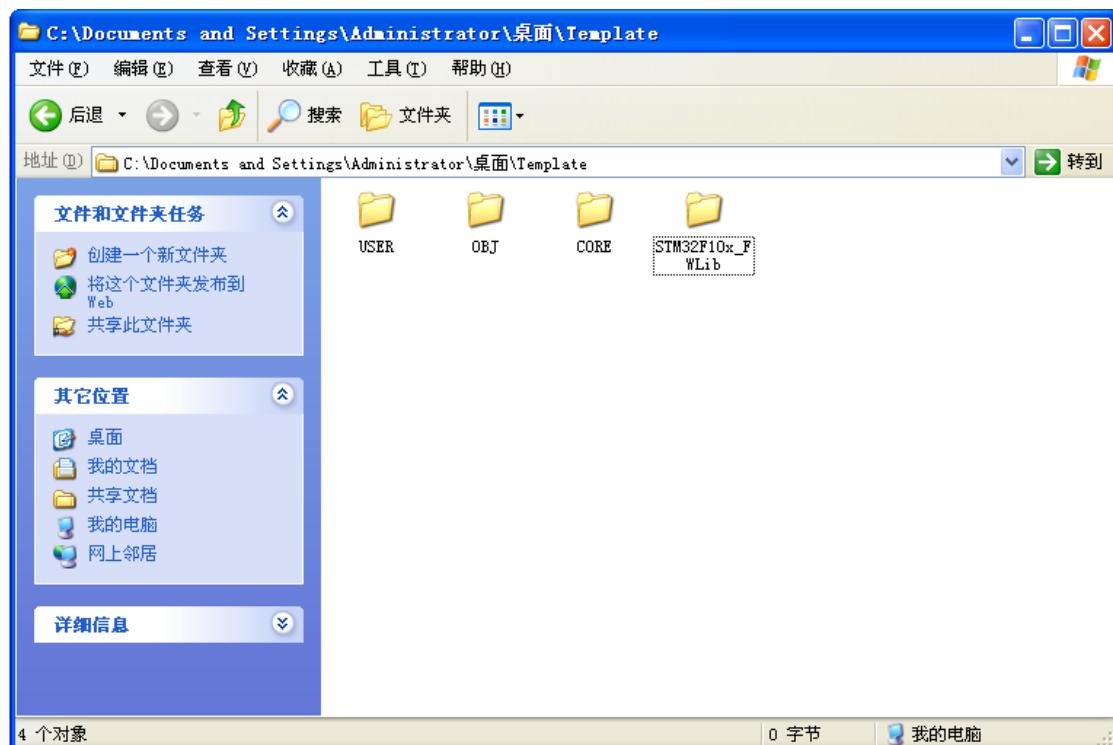


图 3.3.3.5 工程目录预览

- 8) 下面我们要将官方的固件库包里的源码文件复制到我们的工程目录文件夹下面。
打开官方固件库包，定位到我们之前准备好的固件库包的目录
STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\STM32F10x_StdPeriph_Driver 下面，
将目录下面的 src,inc 文件夹 copy 到我们刚才建立的 STM32F10x_FWLib 文件夹下面。
src 存放的是固件库的.c 文件， inc 存放的是对应的.h 文件，您不妨打开这两个文件目录过目一下里面的文件，每个外设对应一个.c 文件和一个.h 头文件。



图 3.3.3.6 官方库源码文件夹

- 9) 下面我们要将固件库包里面相关的启动文件复制到我们的工程目录 CORE 之下。
打开官方固件库包，定位到目录
STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\CoreSupport 下面，将文件 core_cm3.c 和文件 core_cm3.h 复制到 CORE 下面去。然后定位到目录
STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\aa

rm 下面，将里面 startup_stm32f10x_hd.s 文件复制到 CORE 下面。这里我们之前已经解释了不同容量的芯片使用不同的启动文件，我们的芯片 STM32F103ZET6 是大容量芯片，所以选择这个启动文件。

现在看看我们的 CORE 文件夹下面的文件：

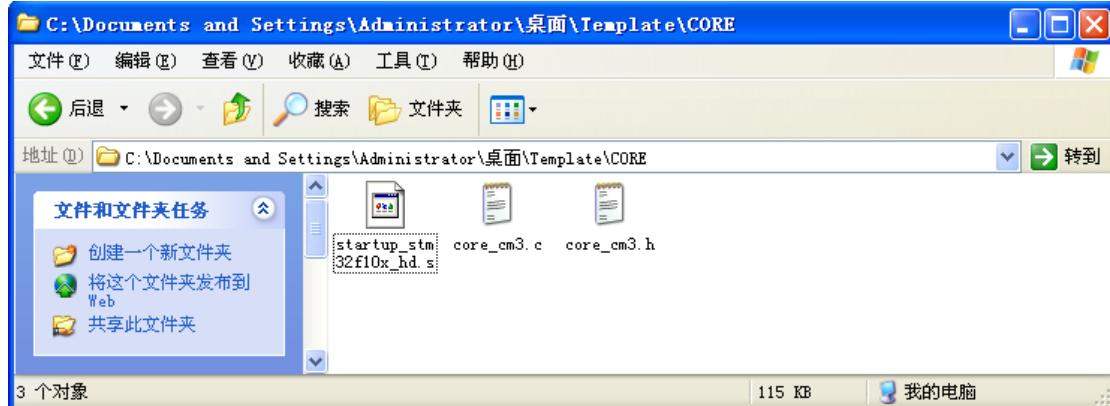


图 3.3.3.7 启动文件夹

10) 定位到目录：

STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x 下面将里面的三个文件 stm32f10x.h, system_stm32f10x.c, system_stm32f10x.h, 复制到我们的 USER 目录之下。然后将

STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template 下面的 4 个文件 main.c, stm32f10x_conf.h, stm32f10x_it.c, stm32f10x_it.h 复制到 USER 目录下面。

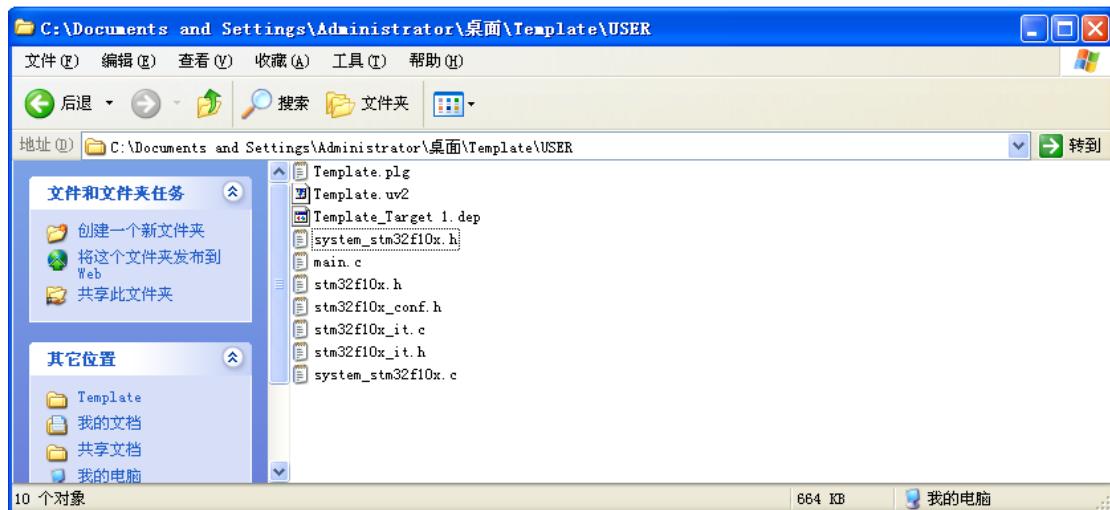


图 3.3.3.8 USER 目录文件浏览

11) 前面 10 个步骤，我们将需要的固件库相关文件复制到了我们的工程目录下面，下面我们将这些文件加入我们的工程中去。右键点击 Target1，选择 Manage Components

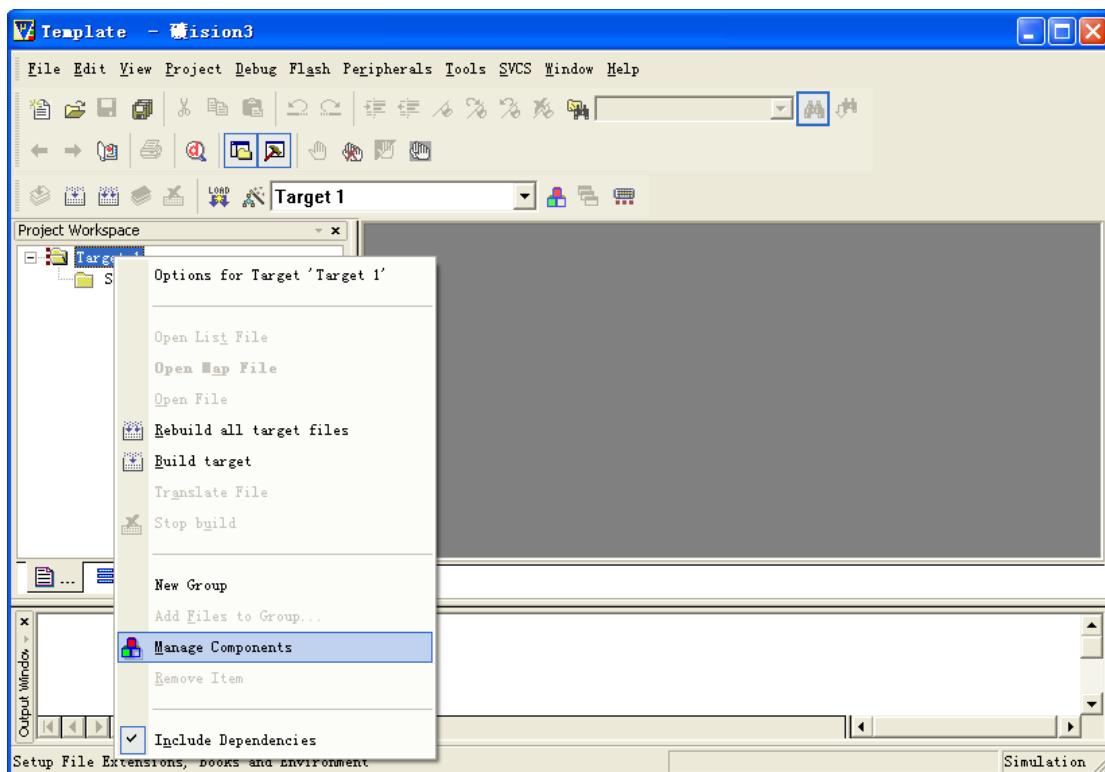


图 3.3.3.9 点击 Management Components

- 12) Project Targets 一栏, 我们将 Target 名字修改为 Template, 然后在 Groups 一栏删掉一个 Source Group1, 建立三个 Groups: USER,CORE,FWLIB。然后点击 OK, 可以看到我们的 Target 名字以及 Groups 情况。

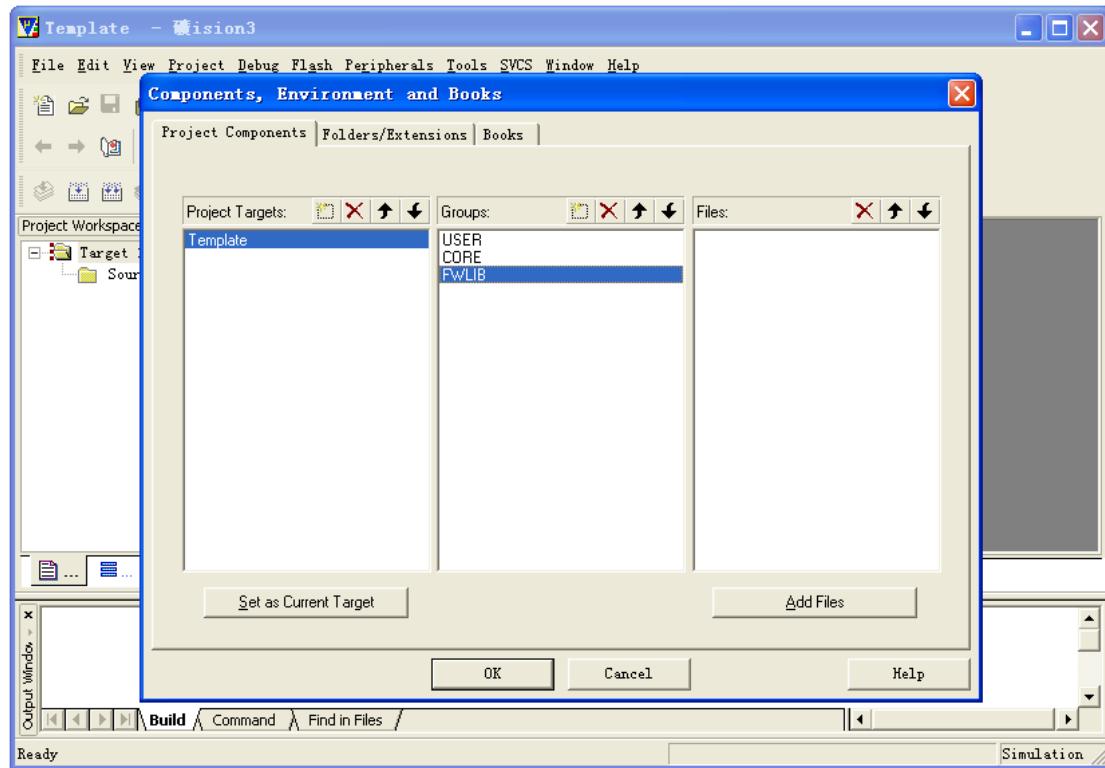




图 3.3.3.10

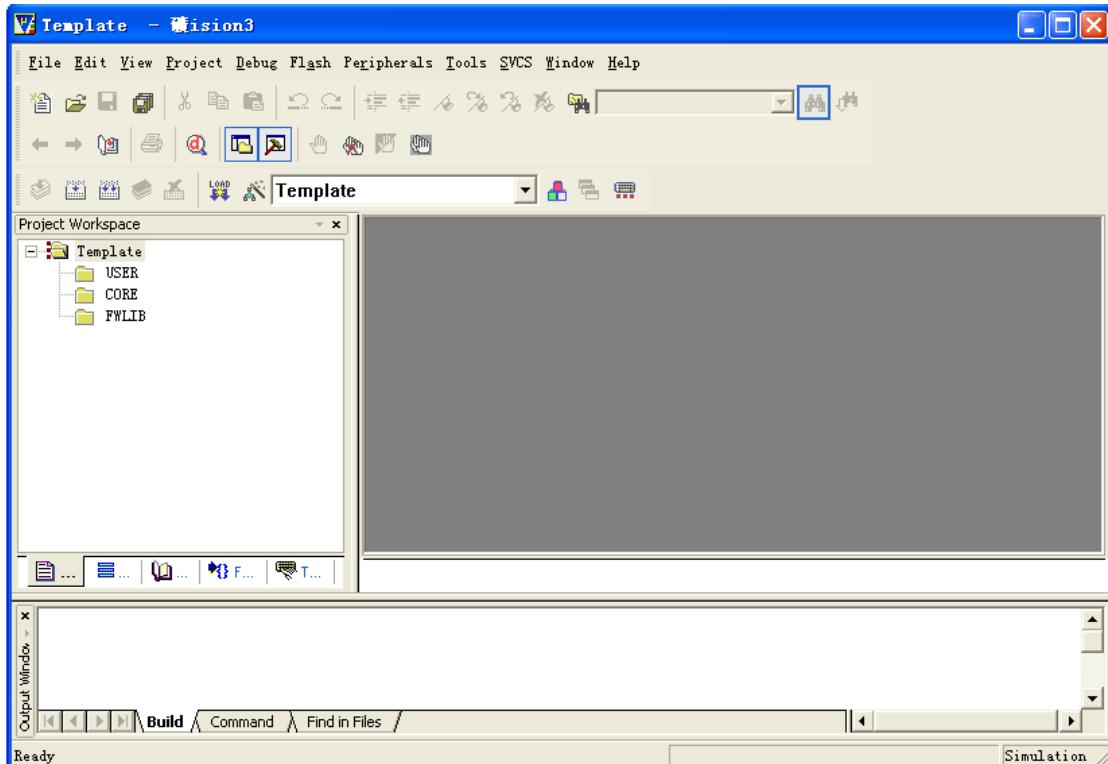


图 3.3.3.11

13) 下面我们往 Group 里面添加我们需要的文件。我们按照步骤 12 的方法，右键点击点击 Tempate，选择选择 Manage Components.然后选择需要添加文件的 Group，这里第一步我们选择 FWLIB，然后点击右边的 Add Files, 定位到我们刚才建立的目录 STM32F10x_FWLib/src 下面，将里面所有的文件选中(Ctrl+A)，然后点击 Add，然后 Close. 可以看到 Files 列表下面包含我们添加的文件。

这里需要说明一下，对于我们写代码，如果我们只用到了其中的某个外设，我们就可以不用添加没有用到的外设的库文件。例如我只用 GPIO，我可以只用添加 stm32f10x_gpio.c 而其他的可以不用添加。这里我们全部添加进来是为了后面方便，不用每次添加，当然这样的坏处是工程太大，编译起来速度慢，用户可以自行选择。

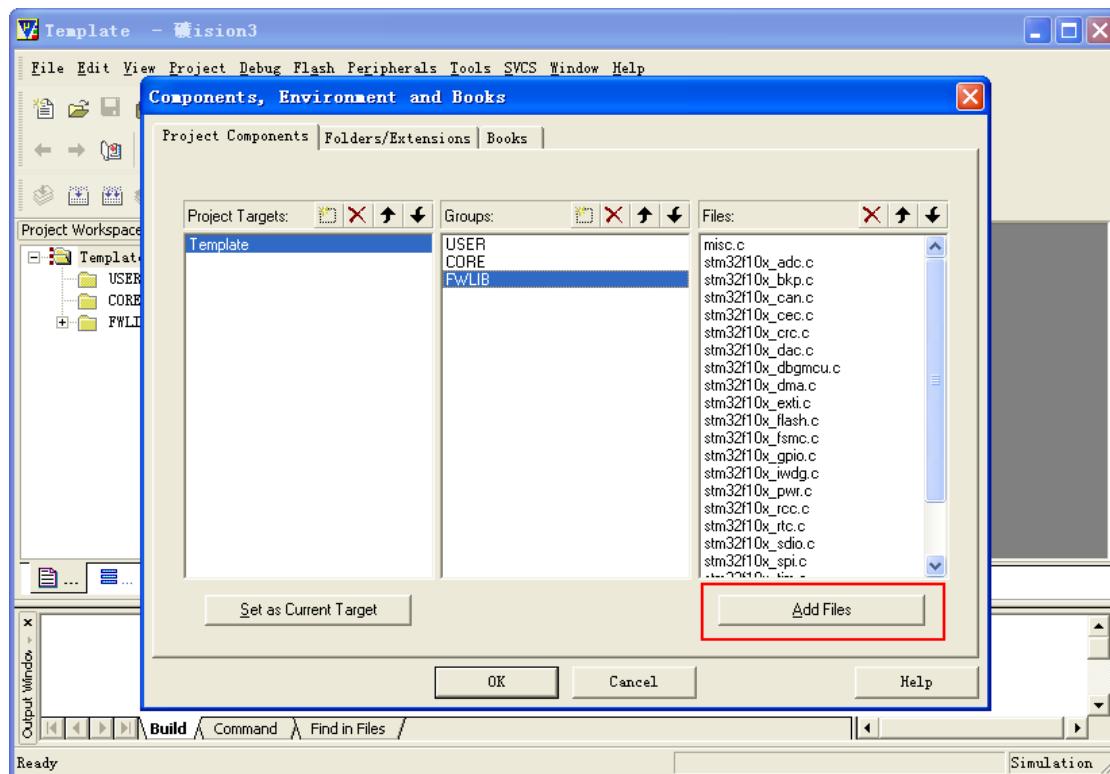


图 3.3.3.12

14) 用同样的方法，将 Groups 定位到 CORE 和 USER 下面，添加需要的文件。这里我们的 CORE 下面需要添加的文件为 core_cm3.c, startup_stm32f10x_hd.s, USER 目录下面需要添加的文件为 main.c, stm32f10x_it.c, system_stm32f10x.c.
这样我们需要添加的文件已经添加到我们的工程中去了，最后点击 OK，回到工程主界面。

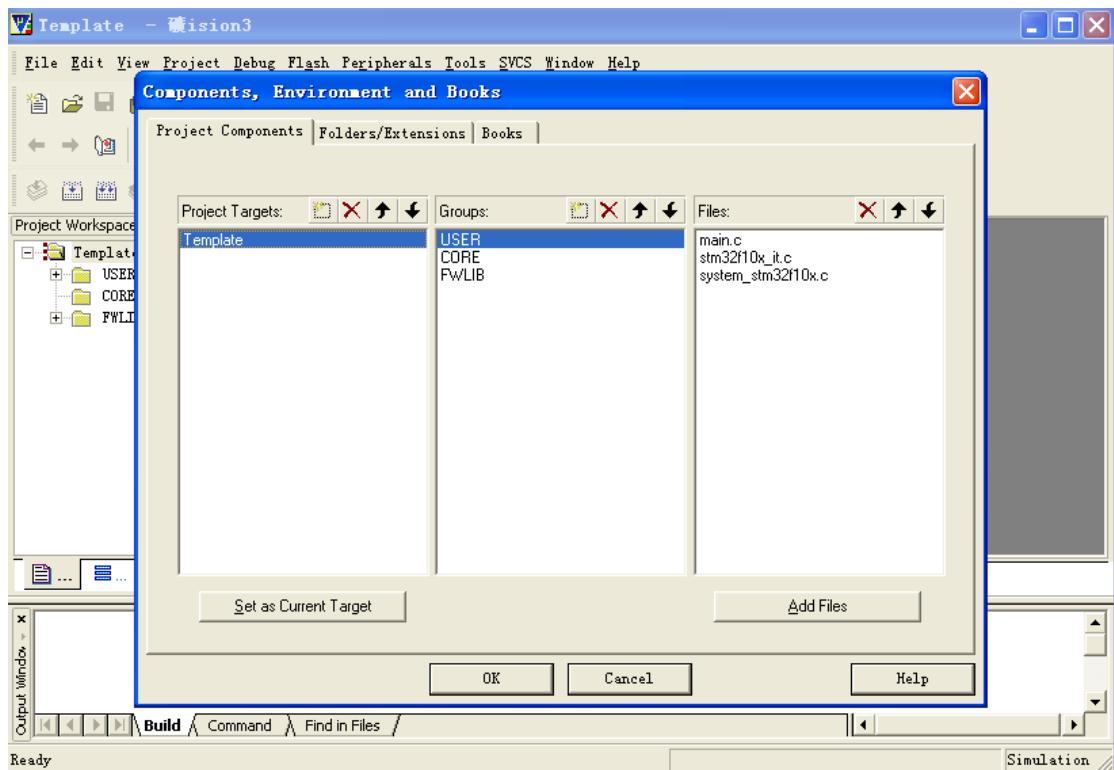


图 3.3.3.13

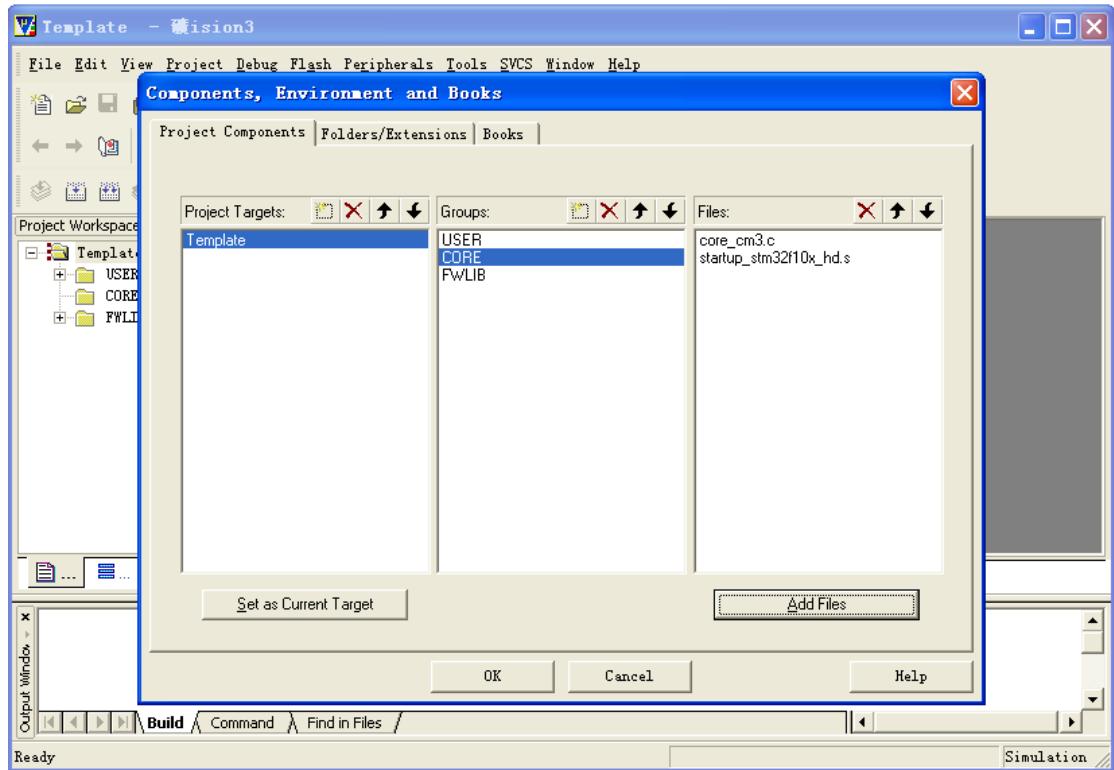


图 3.3.3.14

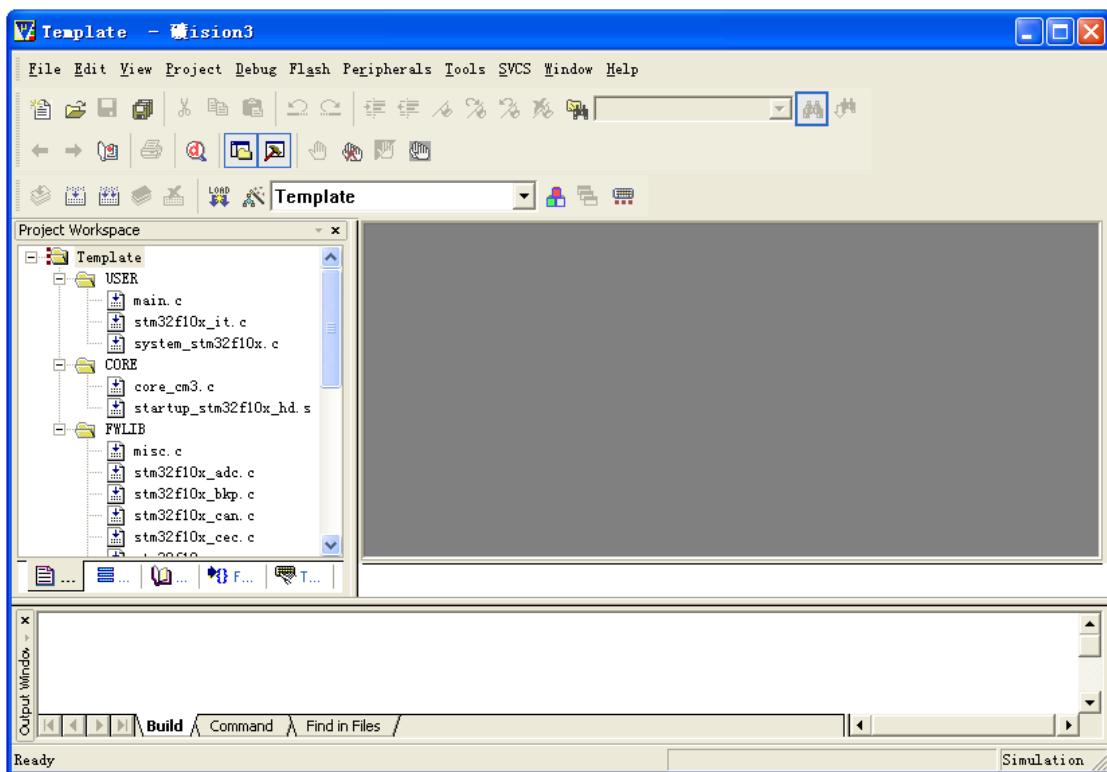


图 3.3.3.15

- 15) 接下来我们要编译工程，在编译之前我们首先要选择编译中间文件存放目录。方法是点击魔术棒，然后选择“Output”选项下面的“Select folder for objects...”，然后选择目录为我们上面新建的 OBJ 目录。

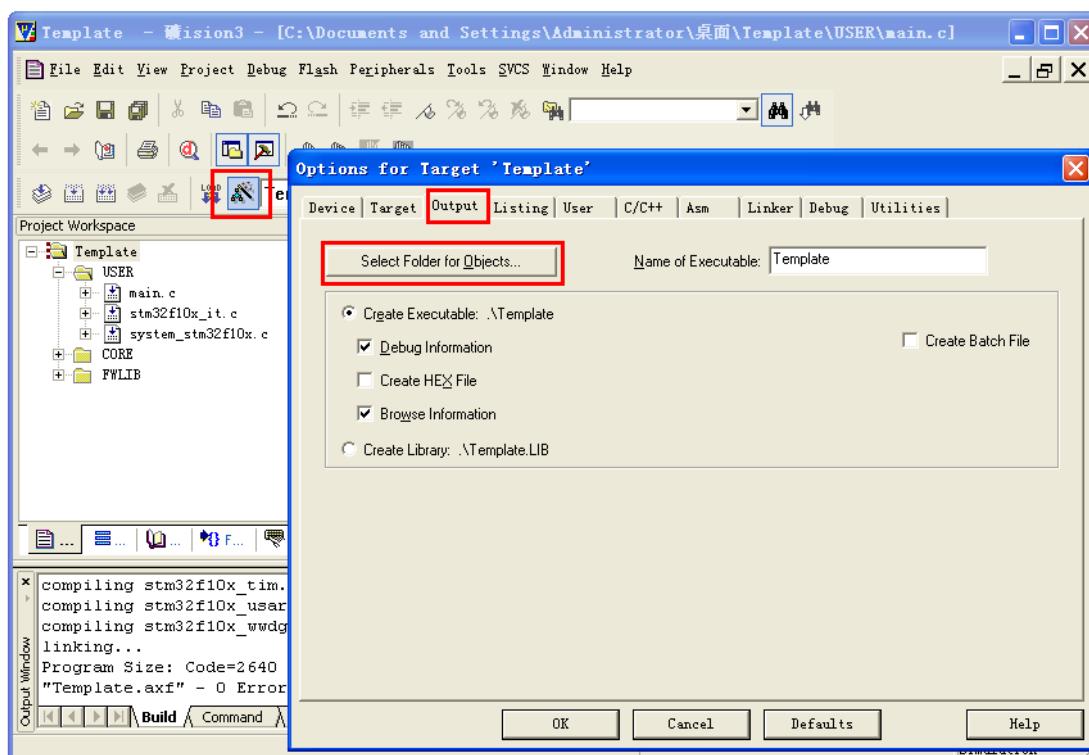


图 3.3.3.16



16) 下面我们点击编译按钮 编译工程，可以看到很多报错，因为找不到库文件。

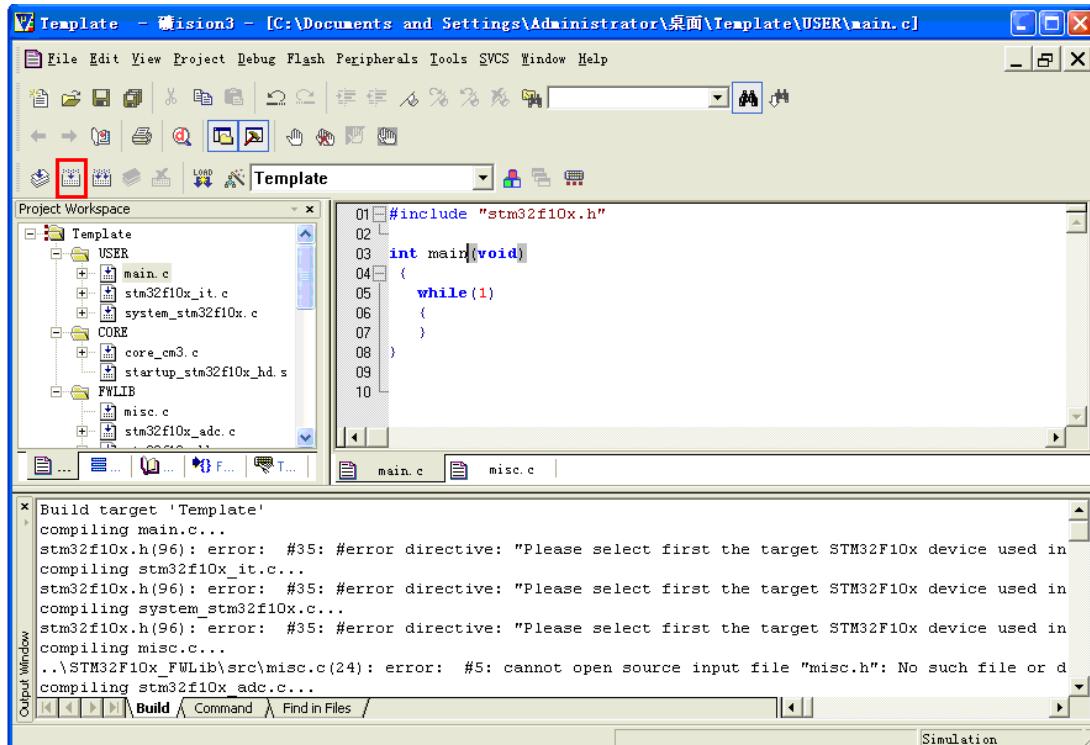


图 3.3.3.17

17) 下面我们要告诉 MDK，在哪些路径之下搜索需要的头文件，也就是头文件目录。回到工

程主菜单，点击魔术棒 ，出来一个菜单，然后点击 c/c++选项.然后点击 Include Paths 右边的按钮。弹出一个添加 path 的对话框，然后我们将图上面的 3 个目录添加进去。记住，keil 只会在一级目录查找，所以如果你的目录下面还有子目录，记得 path 一定要定位到最后一级子目录。然后点击 OK.

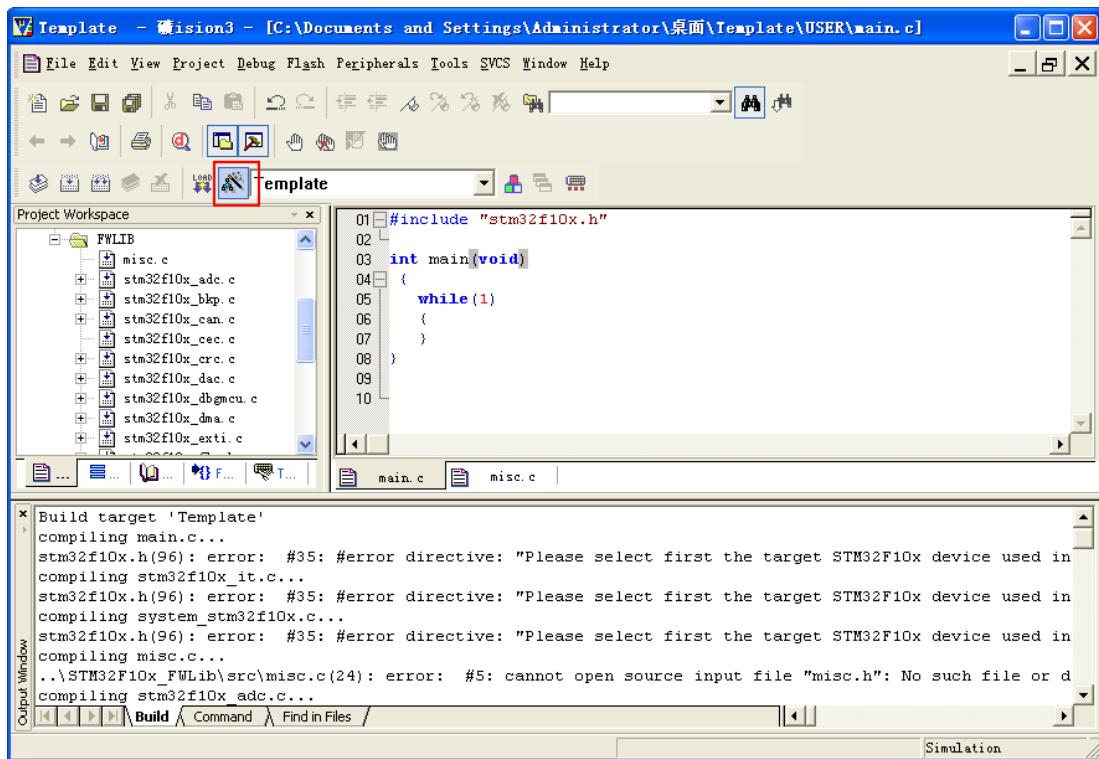


图 3.3.3.18

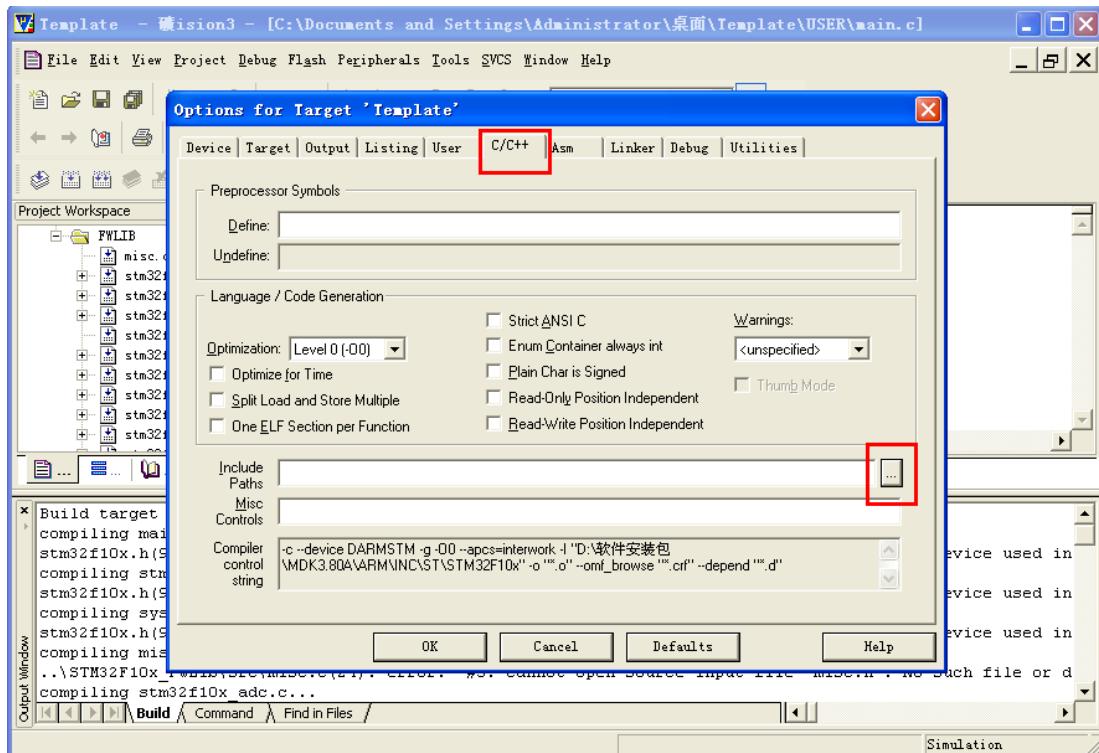


图 3.3.3.219

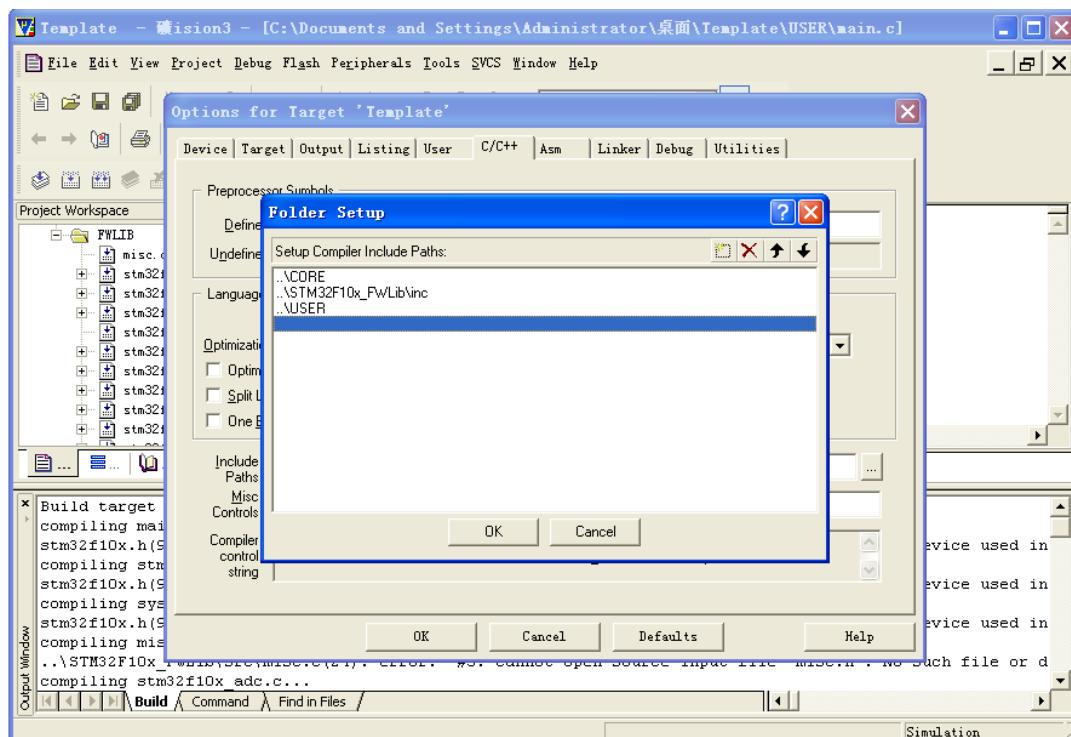


图 3.3.3.20

18) 接下来，我们再来编译工程，可以看到又报了很多同样的错误。为什么呢??

我们可以双击错误，然后会自动定位到文件 `stm32f10x.h` 中出错的地方，可以看到代码：

```
#if !defined (STM32F10X_LD) && !defined (STM32F10X_LD_VL) && !defined (STM32F10X_MD)
&& !defined (STM32F10X_MD_VL) && !defined (STM32F10X_HD) && !defined (STM32F10X_HD_VL)
&& !defined (STM32F10X_XL) && !defined (STM32F10X_CL)           #error "Please select first the
target STM32F10x device used in your application (in stm32f10x.h file)"#endif
```

这是因为 3.5 版本的库函数在配置和选择外设的时候通过宏定义来选择的，所以我们需要配置一个全局的宏定义变量。按照步骤 16，定位到 `c/c++` 界面，然后填写

“`STM32F10X_HD,USE_STDPERIPH_DRIVER`”到 `Define` 输入框里面。

这里解释一下，如果你用的是中容量那么 `STM32F10X_HD` 修改为 `STM32F10X_MD`，小容量修改为 `STM32F10X_LD`。然后点击 `OK`。

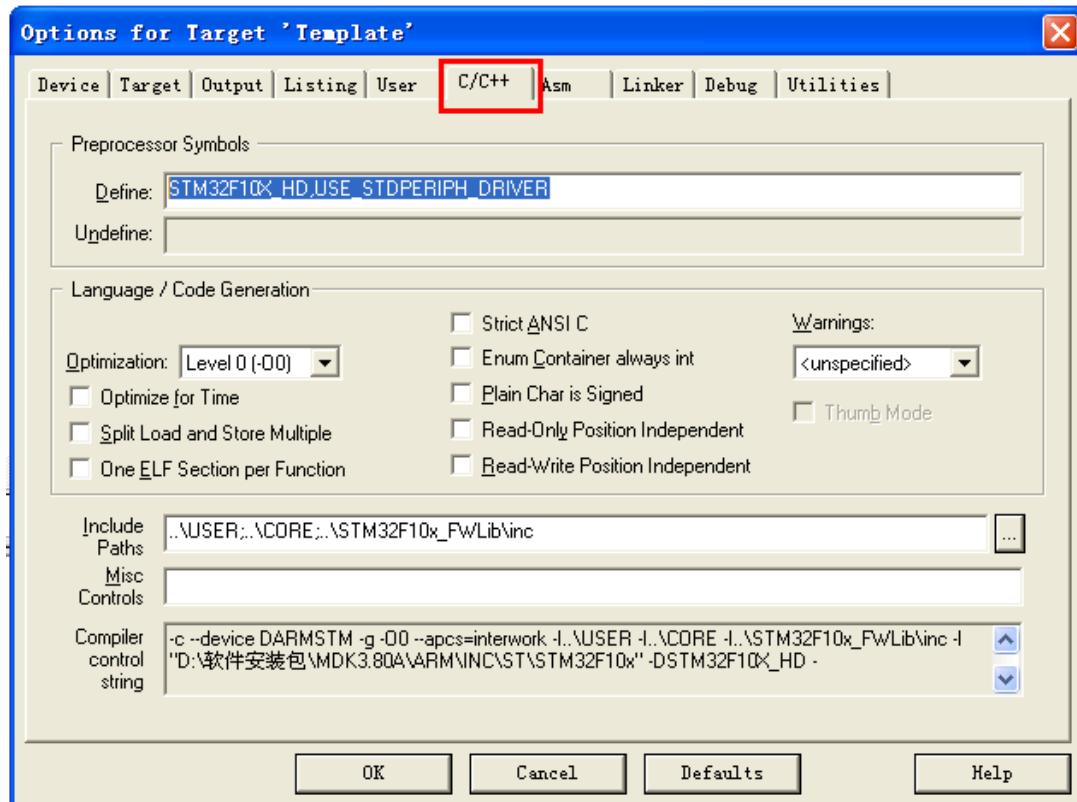


图 3.3.3.21

- 19) 这次在编译之前，我们记得打开工程 USUR 下面的 main.c，复制下面代码到 main.c 覆盖已有代码，然后进行编译。（记得在代码的最后面加上一个回车，否则会有警告），可以看到，这次编译已经成功了。

```
#include "stm32f10x.h"
void Delay(u32 count)
{
    u32 i=0;
    for(;i<count;i++);
}
int main(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|
                           RCC_APB2Periph_GPIOE, ENABLE); //使能 PB,PE 端口时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //LED0-->PB.5 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //IO 口速度为 50MHz
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化 GPIOB.5
    GPIO_SetBits(GPIOB,GPIO_Pin_5); //PB.5 输出高
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //LED1-->PE.5 推挽输出
    GPIO_Init(GPIOE, &GPIO_InitStructure); //初始化 GPIO
}
```



```

GPIO_SetBits(GPIOE,GPIO_Pin_5);           //PE.5 输出高
while(1)
{
    GPIO_ResetBits(GPIOB,GPIO_Pin_5);
    GPIO_SetBits(GPIOE,GPIO_Pin_5);
    Delay(3000000);
    GPIO_SetBits(GPIOB,GPIO_Pin_5);
    GPIO_ResetBits(GPIOE,GPIO_Pin_5);
    Delay(3000000);
}
}

```

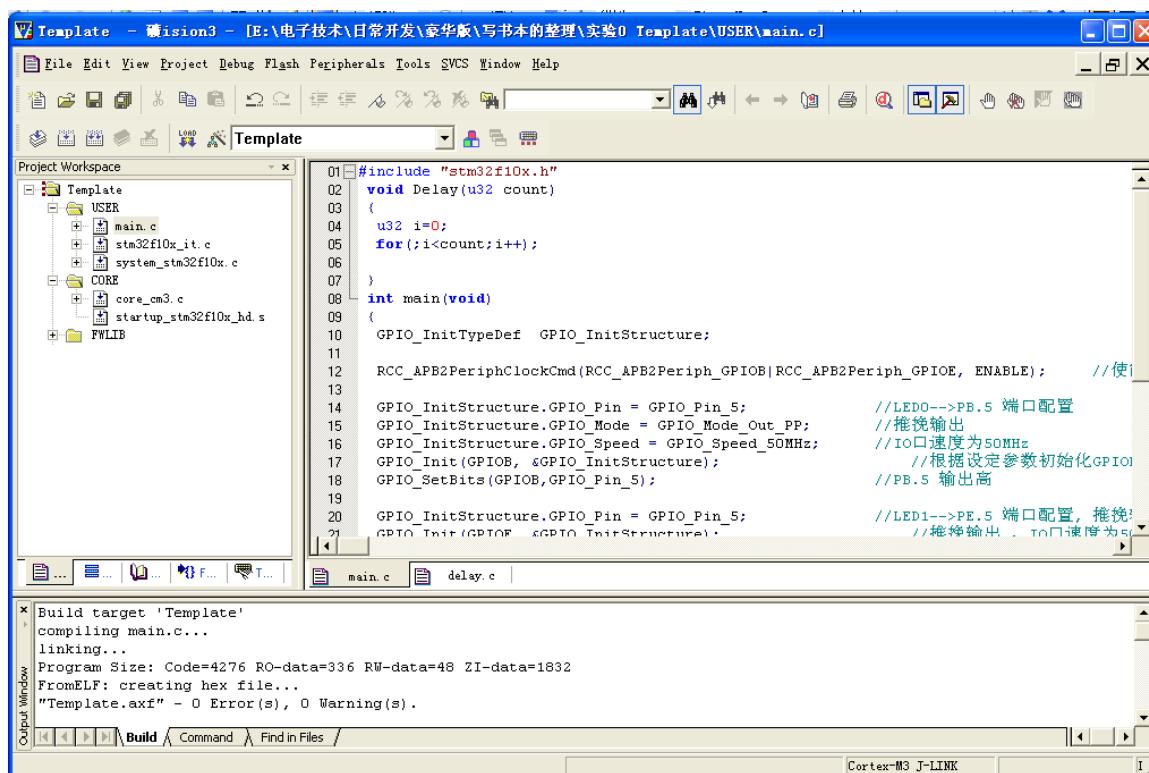


图 3.3.3.22

- 20) 这样一个工程模版建立完毕。下面还需要配置，让编译之后能够生成 hex 文件。同样点击魔术棒，进入配置菜单，选择 Output。然后勾上下三个选项。其中 Create HEX file 是编译生成 hex 文件，Browser Information 是可以查看变量和函数定义。还有就是我们要选择生产的 hex 文件和项目中间文件放在哪个目录，点击“Select folder for Objects...”定位目录，我们的选择定位到上面建立的 OBJ 目录下面。

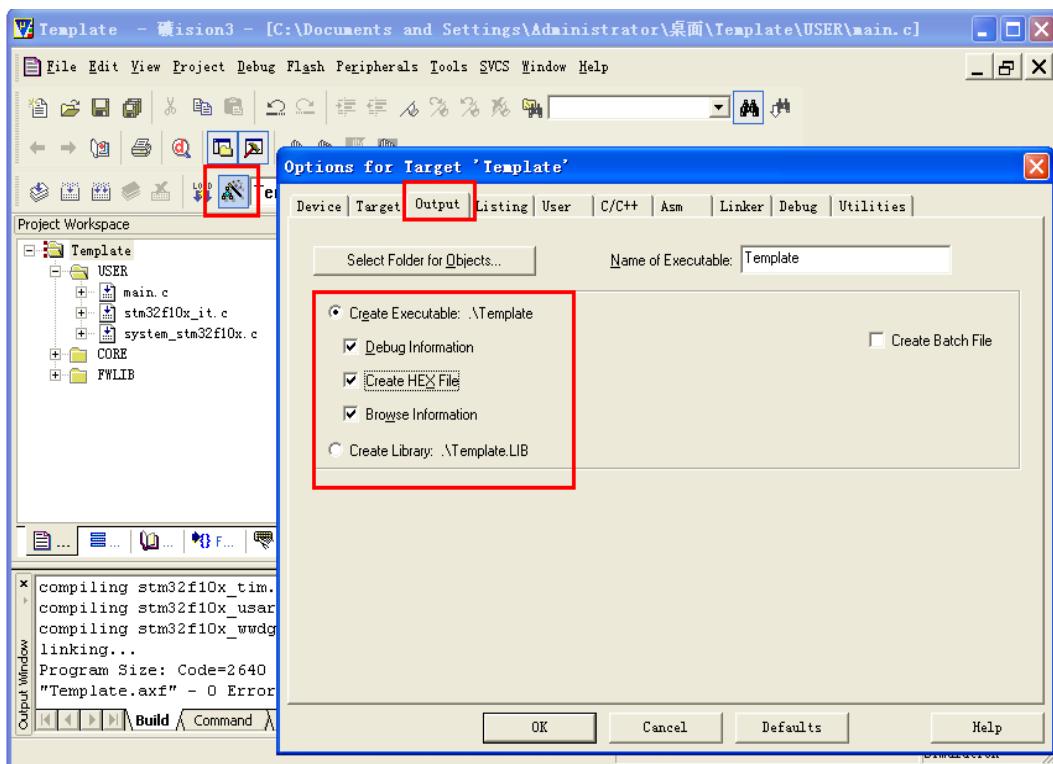


图 3.3.3.23

- 21) 重新编译代码，可以看到生成了 hex 文件在 OBJ 目录下面，这个文件我们用 mcuisp 下载到 mcu 即可。
到这里，一个基于固件库 V3.5 的工程模板就建立了。
- 22) 实际上经过前面 21 个步骤，我们的工程模板已经建立完成。但是在我们 ALIENTEK 提供的实验中，每个实验都有一个 SYSTEM 文件夹，下面有 3 个子目录分别为 sys,uart,delay，存放的是我们每个实验都要使用到的共用代码，该代码是由我们 ALIENTEK 编写，该代码的原理在我们第五章会有详细的讲解，我们这里只是引入到工程中，方便后面的实验建立工程。
首先，找到我们实验光盘，打开任何一个固件库的实验，可以看到下面有一个 SYSTEM 文件夹，比如我们打开实验 1 的工程目录如下：

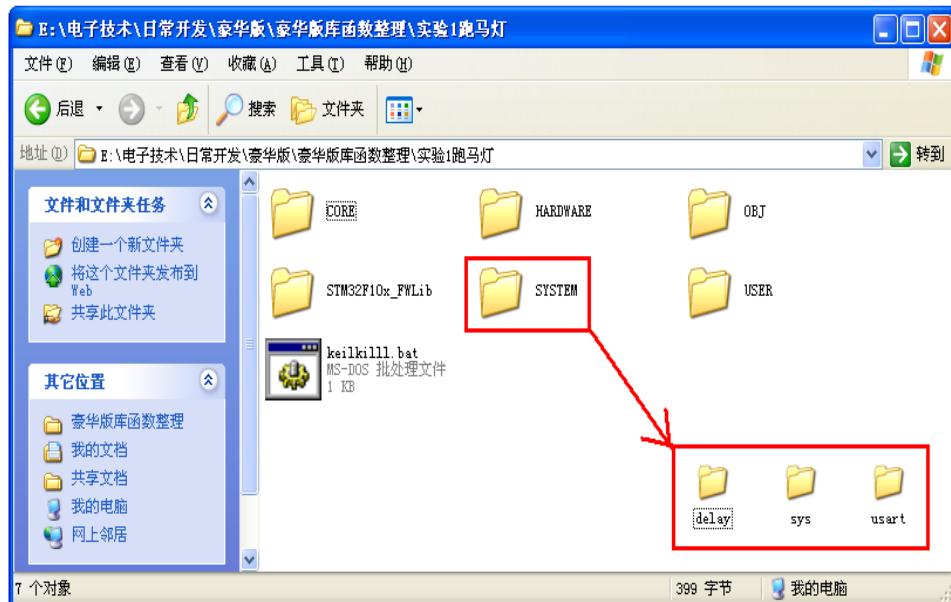


图 3.3.3.24

可以看到有一个 **SYSTEM** 文件夹，进入 **SYSTEM** 文件夹，里面有三个子文件夹分别为 **delay**, **sys**, **uart**，每个子文件夹下面都有相应的.c 文件和.h 文件。我们接下来要将这三个目录下面的代码加入到我们工程中去。

用我们之前讲解步骤 13 的办法，在工程中新建一个组，命名为 **SYSTEM**，然后加入这三个文件夹下面的.c 文件分别为 **sys.c**, **delay.c**, **uart.c**，如下图：

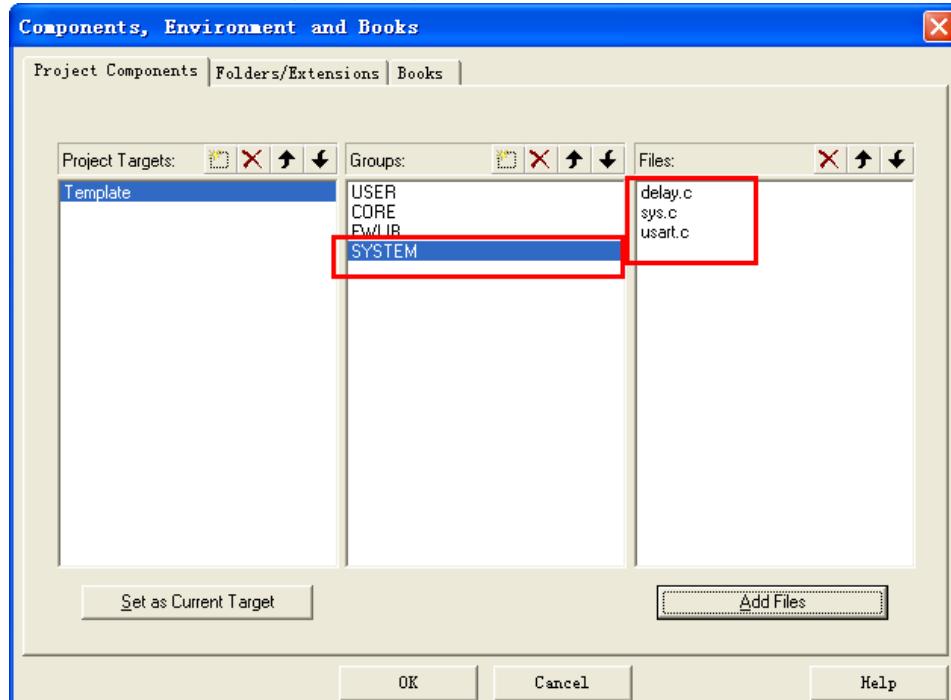


图 3.3.3.25

然后点击“OK”之后可以看到工程中多了一个 **SYSTEM** 组，下面有 3 个.c 文件。

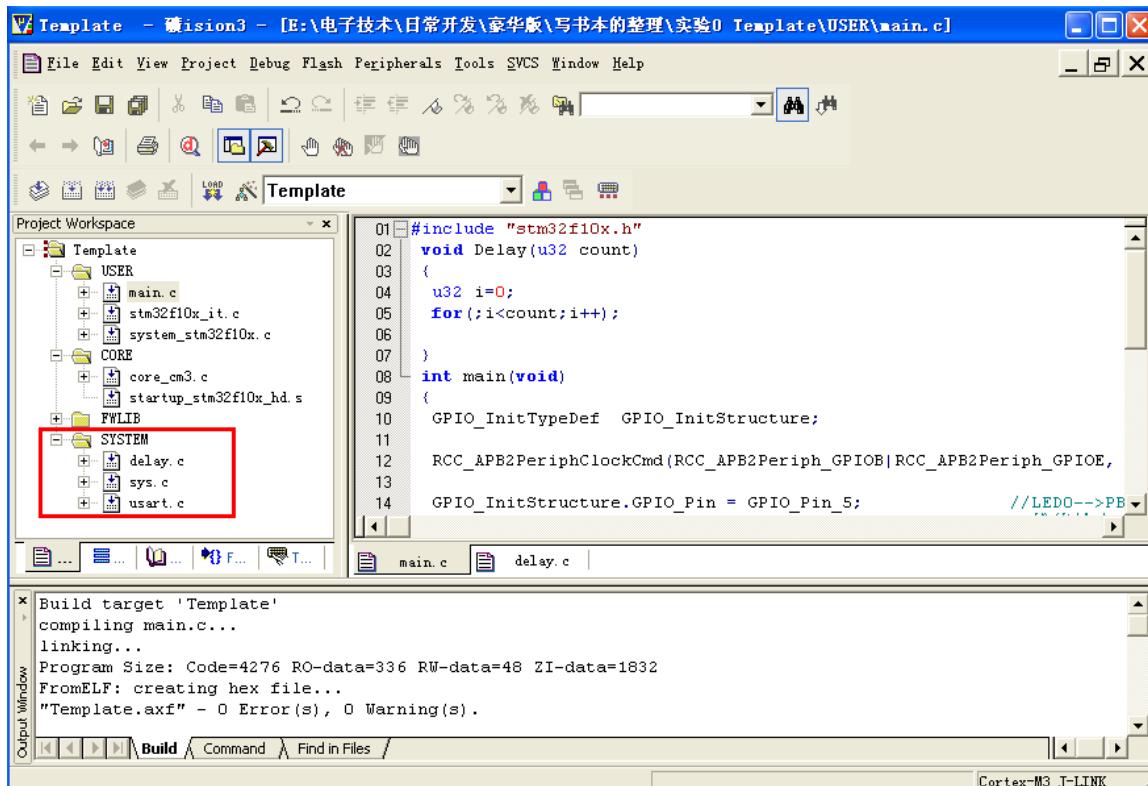


图 3.3.3.26

接下来我们将对应的三个目录（sys,uart,delay）加入到 PATH 中去，因为每个目录下面都有相应的.h 头文件，这请参考步骤 17 即可，加入后的截图是：

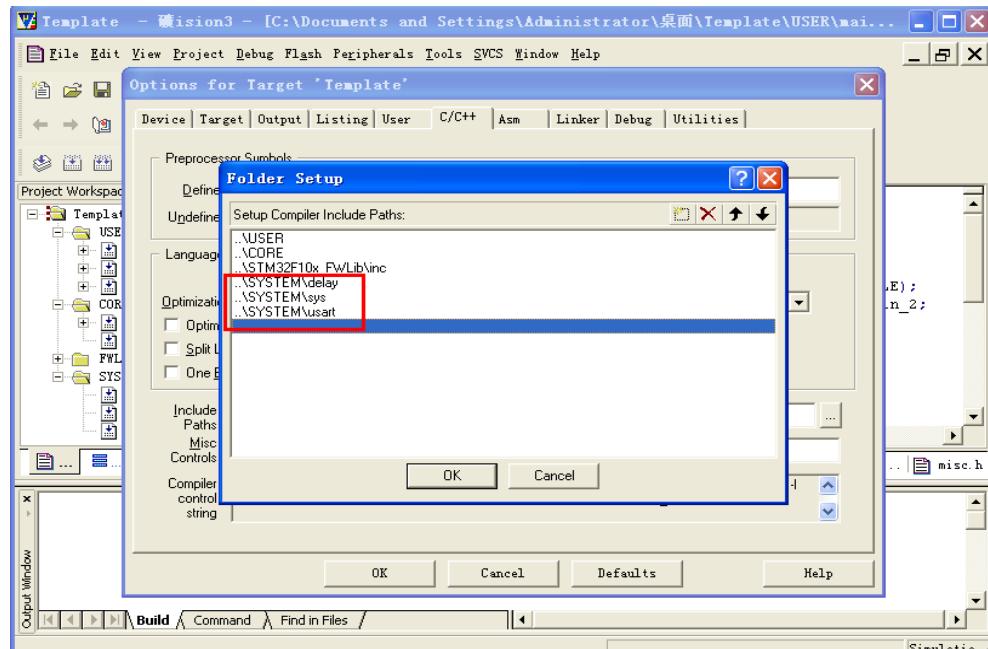


图 3.3.3.27

最后点击 OK。这样我们的工程模板就彻底完成了。我们建立好的工程模板在我们光盘的实验目录里面有，名字为“实验 0 Template 工程模板”大家可以打开对照一下。



3.4 MDK 下的程序下载与调试

上一节，我们学会了如何在 MDK 下创建 STM32 工程。本节，我们将向读者介绍 STM32 的代码下载以及调试。这里的调试包括了软件仿真和硬件调试（在线调试）。通过本章的学习，你将了解到：1、STM32 的程序下载；2、STM32 在 MDK 下的软件仿真；3、利用 JLINK 对 STM32 进行在线调试。

3.4.1 STM32 软件仿真

MDK 的一个强大的功能就是提供软件仿真，通过软件仿真，我们可以发现很多将要出现的问题，避免了下载到 STM32 里面来查这些错误，这样最大的好处是能很方便的检查程序存在的问题，因为在 MDK 的仿真下面，你可以查看很多硬件相关的寄存器，通过观察这些寄存器，你可以知道代码是不是真正有效。另外一个优点是不必频繁的刷机，从而延长了 STM32 的 FLASH 寿命（STM32 的 FLASH 寿命 $\geq 1W$ 次）。当然，软件仿真不是万能的，很多问题还是要到在线调试才能发现。废话不多说了，接下来我们开始进行软件仿真。

上一章，我们创立了一个工程模板，本节我们将教大家如何在 MDK3.80A 的软件环境下仿真这个工程，以验证我们代码的正确性。首先工程模板中 main.c 中代码修如下：

```
#include "delay.h"
#include "usart.h"
int main(void)
{
    u8 t=0;
    delay_init();
    NVIC_Configuration();
    uart_init(9600);
    while(1)
    {
        printf("t:%d\n",t);
        delay_ms(500);
        t++;
    }
}
```

在开始软件仿真之前，先检查一下配置是不是正确，在 IDE 里面点击 ，确定 Target 选项卡内容如图 3.4.1.1 所示（主要检查芯片型号和晶振频率，其他的一般默认就可以）：

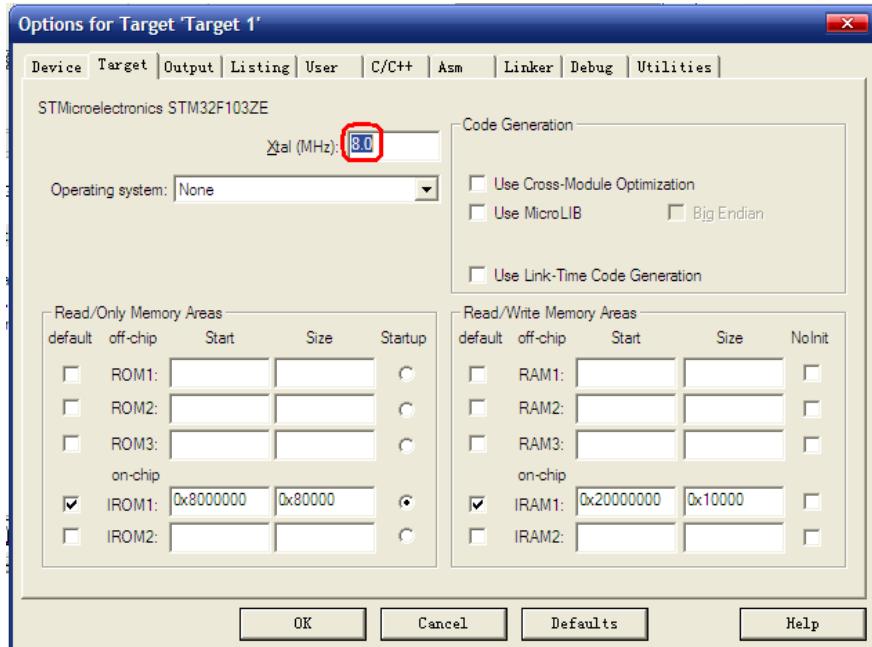


图 3.4.1.1 Target 选项卡

确认了芯片以及外部晶振频率 (8.0Mhz) 之后，基本上就确定了 MDK3.80A 软件仿真的硬件环境了，接下来，我们再点击 Debug 选项卡，设置为如图 3.4.1.2 所示：

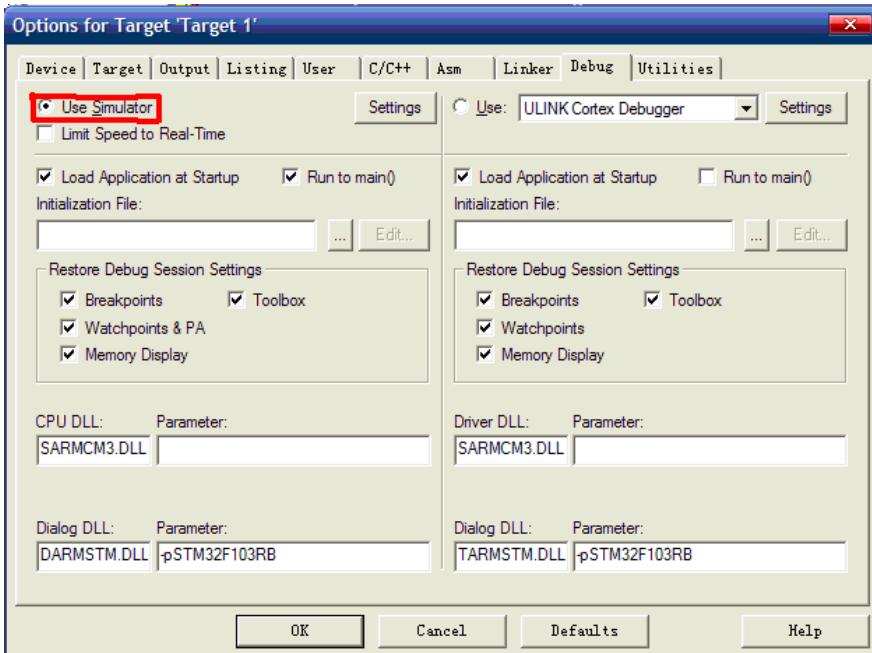


图 3.4.1.2 Debug 选项卡

在图 3.4.22 中，我们主要确认的是 Use Simulator 是否选择（因为如果选择右边的 Use，那就是用 ULINK 进行硬件 Debug 了，这个将在下面介绍），其他的采用默认的就可以。确认了这项之后，我们便可以选择 OK，退出 Options for Target 对话框了。

接下来，我们点击 (开始/停止仿真按钮)，开始仿真，出现如图 3.4.1.3 所示界面：

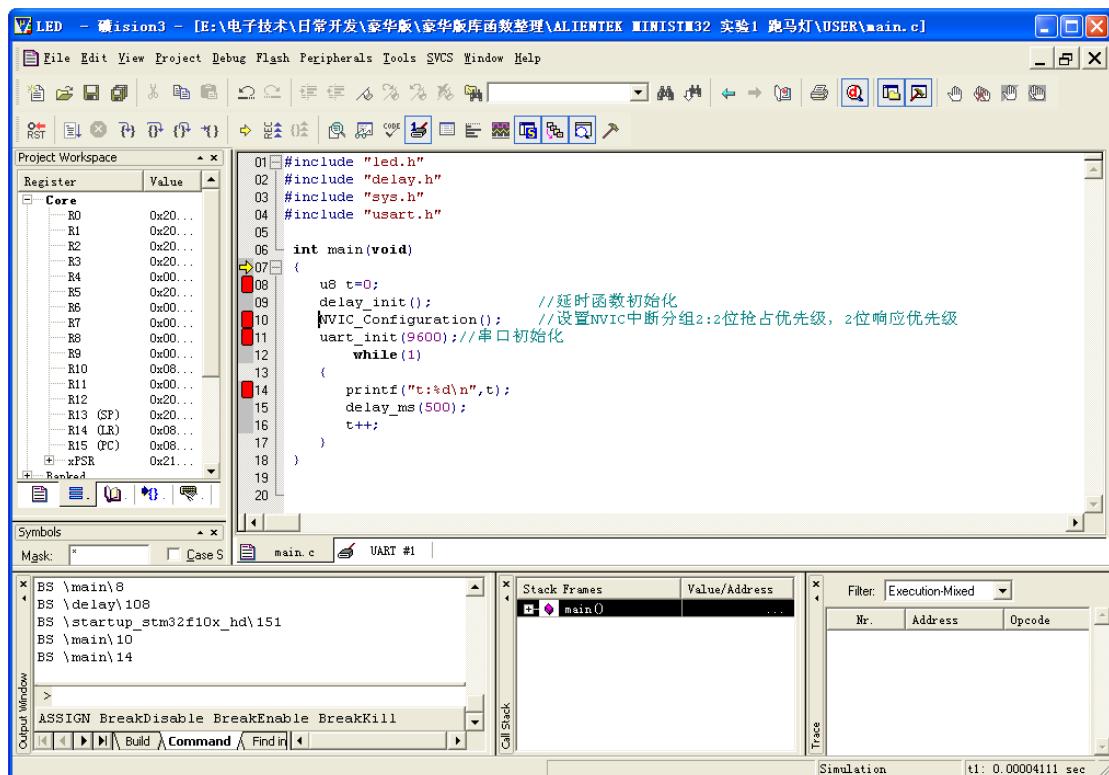


图 3.4.1.3 开始仿真

可以发现，多出了一个工具条，这就是 Debug 工具条，这个工具条在我们仿真的时候是非常有用的，下面简单介绍一下 Debug 工具条相关按钮的功能。Debug 工具条部分按钮的功能如图 3.4.1.4 所示：

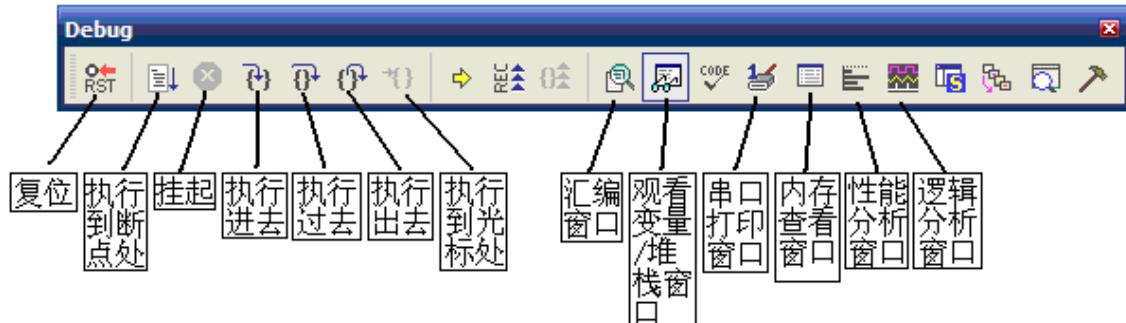


图 3.4.1.4 Debug 工具条

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

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

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

执行进去：该按钮用来实现执行到某个函数里面去的功能，在没有函数的情况下，是等同于执行过去按钮的。

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



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

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

汇编窗口：通过该按钮，就可以查看汇编代码，这对分析程序很有用。

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

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

内存查看窗口：该按钮按下，会弹出一个内存查看窗口，可以在里面输入你要查看的内存地址，然后观察这一片内存的变化情况。是很常用的一个调试窗口。

性能分析窗口：按下该按钮，会弹出一个观看各个函数执行时间和所占百分比的窗口，用来分析函数的性能是比较有用的。

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

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

这样，我们在上面的仿真界面里面选内存查看窗口、串口打印窗口。然后调节一下这两个窗口的位置，如图 3.4.1.5 所示：

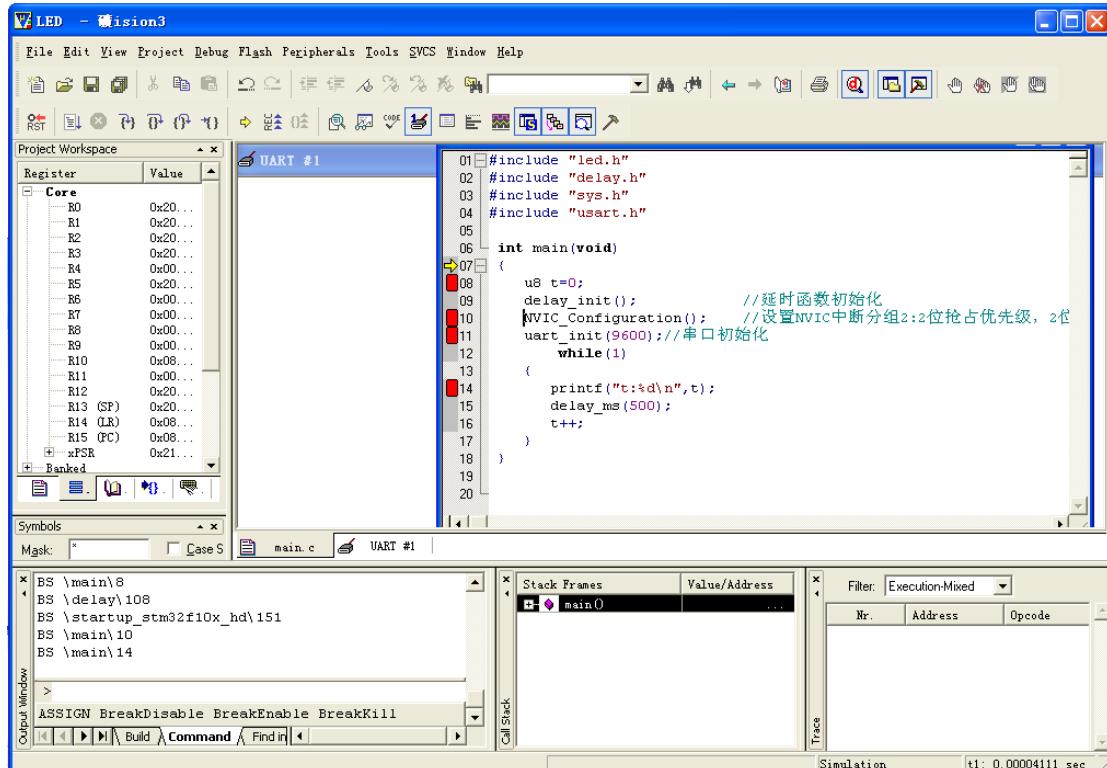


图 3.4.1.5 调出仿真串口打印窗口

我们把光标放到 main.c 的 09 行的空白处，然后双击鼠标左键，可以看到在 09 行的左边出现了一个红框，即表示设置了一个断点（也可以通过鼠标右键弹出菜单来加入），再次双击则取



消)。然后我们点击 ，执行到该断点处，如图 3.4.1.6 所示：

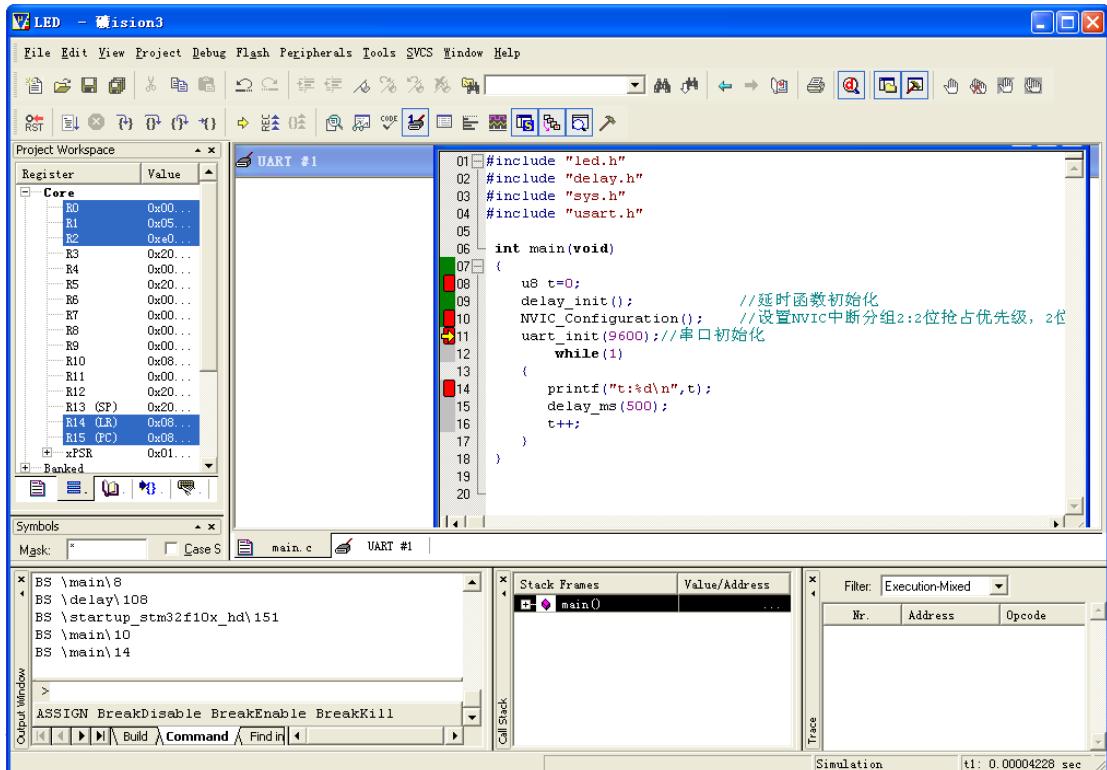


图 3.4.1.6 执行到断点处

我们现在先不忙着往下执行，点击菜单栏的 Peripherals->USARTs->USART 1。可以看到，有很多外设可以查看，这里我们查看的是串口 1 的情况。如图 3.4.1.7 所示：

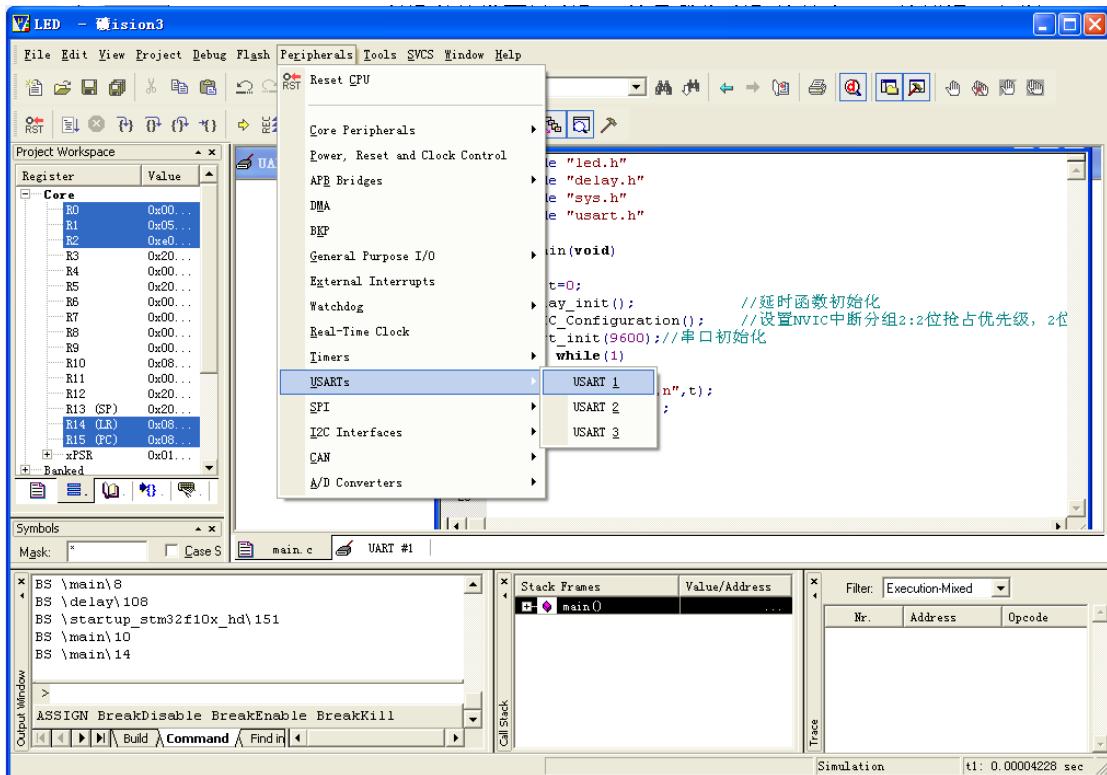




图 3.4.1.7 查看串口 1 相关寄存器

单击 USART1 后会在 IDE 之外出现一个如图 3.4.1.8 (a) 所示的界面：

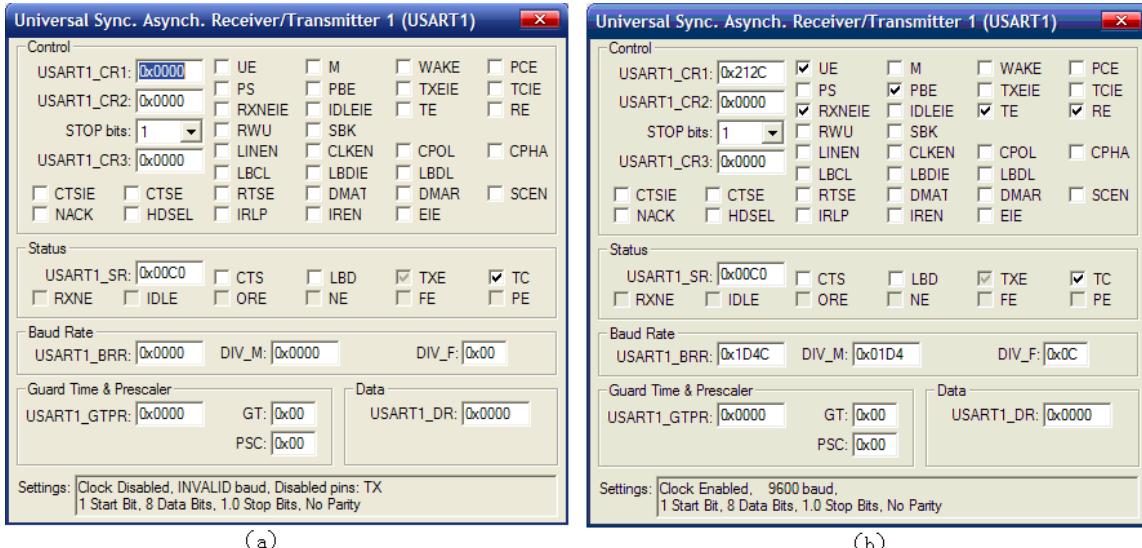


图 3.4.1.8 串口 1 各寄存器初始化前后对比

图 3.4.8 (a) 是 STM32 的串口 1 的默认设置状态，从中可以看到所有与串口相关的寄存器

全部在这上面表示出来了，而且有当前串口的波特率等信息的显示。我们接着单击一下 ，执行完串口初始化函数，得到了如图 3.4.8 (b) 所示的串口信息。大家可以对比一下这两个图的区别，就知道在 `uart_init(9600);` 这个函数里面大概执行了哪些操作。

通过图 3.4.8 (b)，我们可以查看串口 1 的各个寄存器设置状态，从而判断我们写的代码是否有问题，只有这里的设置正确了之后，才有可能在硬件上正确的执行。同样这样的方法也可以适用于很多其他外设，这个读者慢慢体会吧！这一方法不论是在排错还是在编写代码的时候，都是非常有用的。

然后我们继续单击 按钮，一步步执行，最后就会看到在 USART #1 中打印出相关的信息，如图 3.4.1.9 所示：

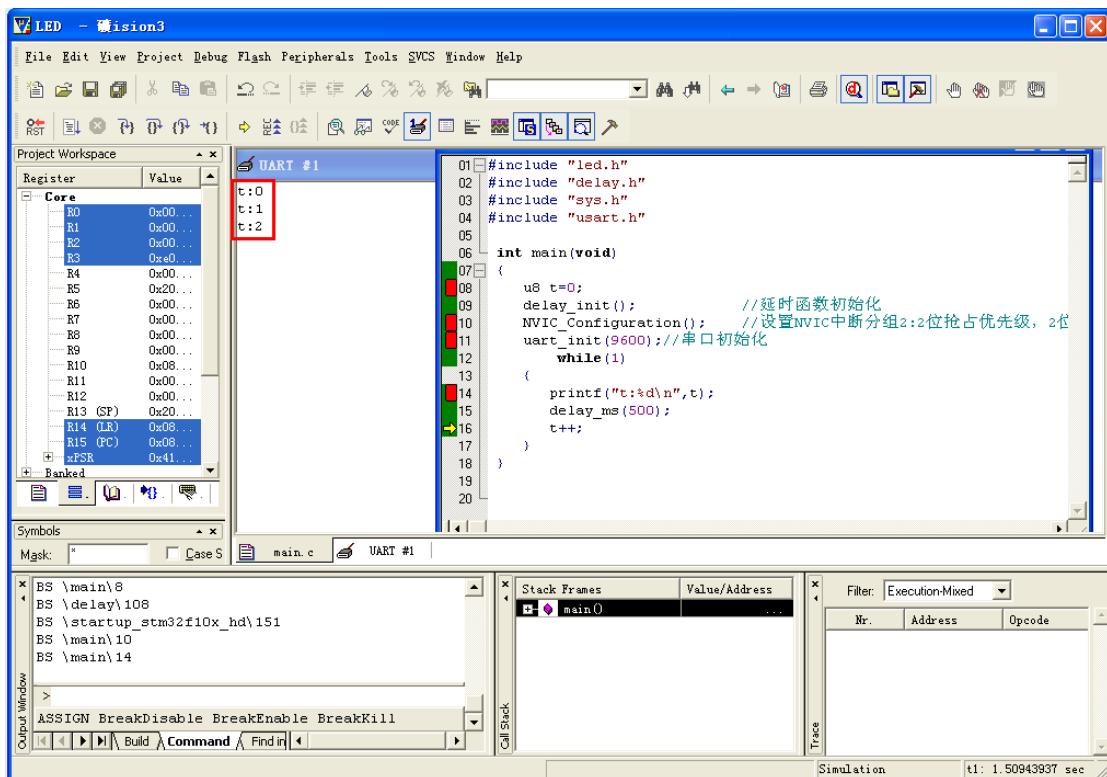


图 3.4.1.9 串口 1 输出信息

图中红色方框内的数据是串口 1 打印出来的，证明我们的仿真是通过的，代码运行时会在串口 1 不停的输出 t 的值，每 0.5s 执行一次。软件仿真时间可以在 IDE 的最下面（右下角）

观看，如图 3.4.1.10 所示。并且 t 自增，与我们预期的一致。再次按下 结束仿真。



图 3.4.1.10 仿真持续时间

至此，我们软件仿真就结束了，通过软件仿真，我们在 MDK3.80A 中验证了代码的正确性，接下来我们下载代码到硬件上来真正验证一下我们的代码是否在硬件上也是可行的。

3.4.2 STM32 程序下载

STM32 的程序下载有多种方法：USB、串口、JTAG、SWD 等，这几种方式，都可以用来给 STM32 下载代码。不过，我们最常用的，最经济的，就是通过串口给 STM32 下载代码。本节，我们将向大家介绍，如何利用串口给 STM32 下载代码。

STM32 的串口下载一般是通过串口 1 下载的，本指南的实验平台 ALIENTEK 战舰 STM32 开发板，不是通过 RS232 串口下载的，而是通过自带的 USB 串口来下载。看起来像是 USB 下载（只需一根 USB 线，并不需要串口线）的，实际上，是通过 USB 转成串口，然后再下载的。

下面，我们就一步步教大家如何在实验平台上利用 USB 串口来下载代码。

首先要在板子上设置一下，在板子上把 RXD 和 PA9(STM32 的 TXD), TXD 和 PA10(STM32 的 RXD)通过跳线帽连接起来，这样我们就把 CH340G 和 MCU 的串口 1 连接上了。这里由于 ALIENTEK 这款开发板自带了一键下载电路，所以我们并不需要去关心 BOOT0 和 BOOT1 的状态，但是为了让下下载完后可以按复位执行程序，我们建议大家把 BOOT1 和 BOOT0 都设置



为 0。设置完成如下图所示：

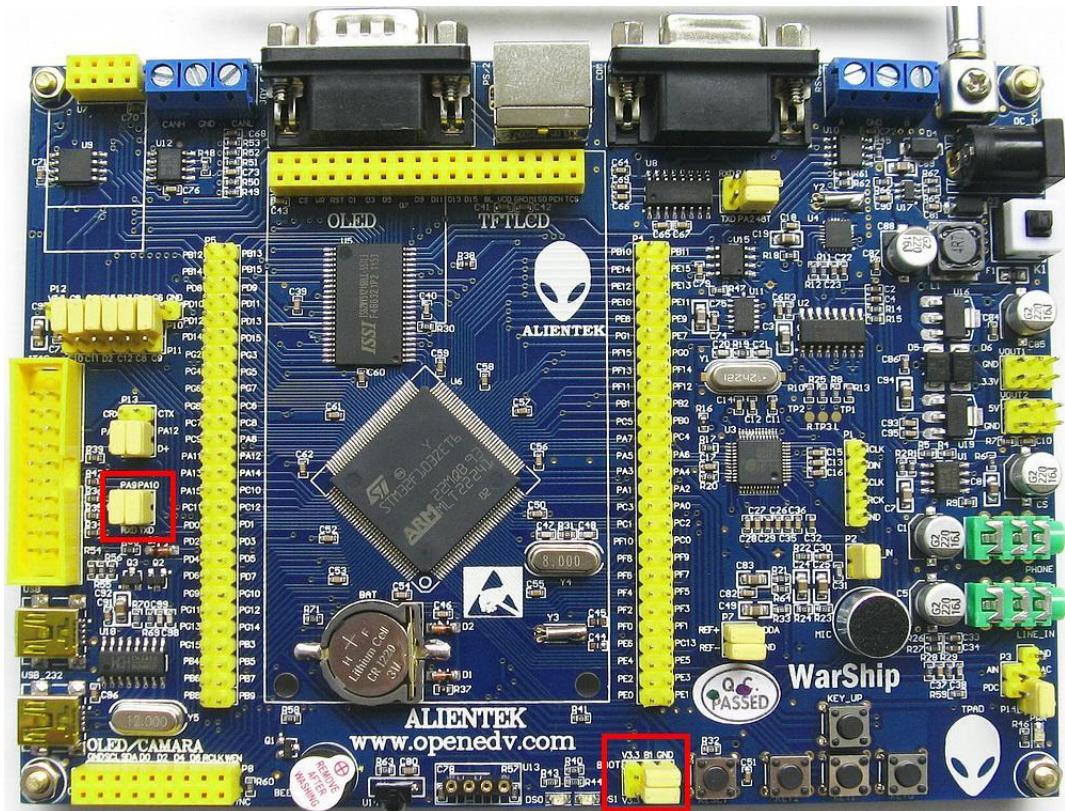


图 3.4.2.1 开发板串口下载跳线设置

这里简单说明一下一键下载电路的原理，我们知道，STM32 串口下载的标准方法是 2 个步骤：

- 1, 把 B0 接 V3.3（保持 B1 接 GND）。
- 2, 按一下复位按键。

通过这两个步骤，我们就可以通过串口下载代码了，下载完成之后，如果没有设置从 0X08000000 开始运行，则代码不会立即运行，此时，你还需要把 B0 接回 GND，然后再按一次复位，才会开始运行你刚刚下载的代码。所以整个过程，你得跳动 2 次跳线帽，还得按 2 次复位，比较繁琐。而我们的一键下载电路，则利用串口的 DTR 和 RTS 信号，分别控制 STM32 的复位和 B0，配合上位机软件 (mcuisp)，设置：DTR 的低电平复位，RTS 高电平进 BootLoader，这样，B0 和 STM32 的复位，完全可以由下载软件自动控制，从而实现一键下载。

接着我们在 USB_232 处插入 USB 线，并接上电脑，如果之前没有安装 CH340G 的驱动（如果已经安装过了驱动，则应该能在设备管理器里面看到 USB 串口，如果不能则要先卸载之前的驱动，卸载完后重启电脑，再重新安装我们提供的驱动），则电脑会提示找到新硬件，如图 3.4.2.2 所示：

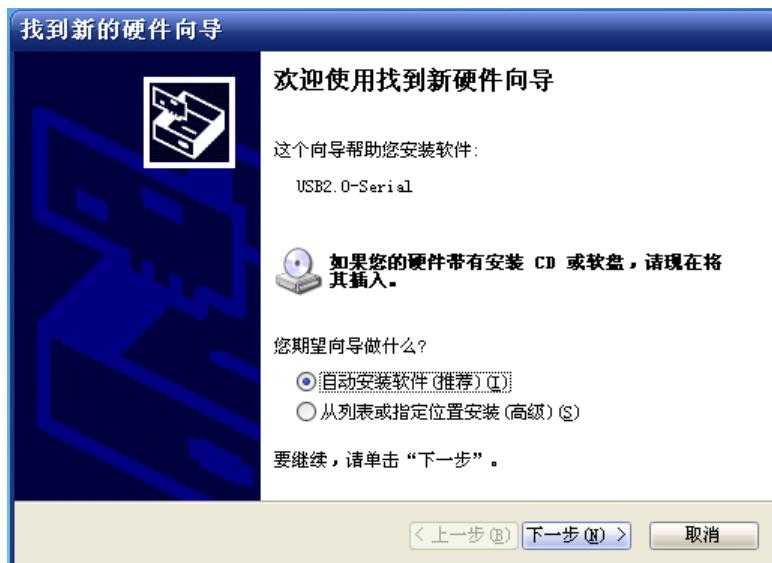


图 3.4.2.2 找到新硬件

我们不理会这个提示，直接找到光盘→软件资料→软件 文件夹下的 CH340 驱动，安装该驱动，如图 3.4.2.3 所示：



图 3.4.2.3 CH340 驱动安装

在驱动安装成功之后，拔掉 USB 线，然后重新插入电脑，此时电脑就会自动给其安装驱动了。在安装完成之后，可以在电脑的设备管理器里面找到 USB 串口（如果找不到，则重启下电脑），如图 3.4.2.4 所示：

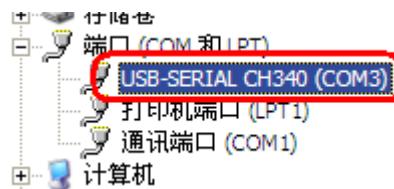


图 3.4.2.4 USB 串口

在图 4.2.4 中可以看到，我们的 USB 串口被识别为 COM3，这里需要注意的是：不同电脑可能不一样，你的可能是 COM4、COM5 等，但是 USB-SERIAL CH340，这个一定是一样的。如果没找到 USB 串口，则有可能是你安装有误，或者系统不兼容。

在安装了 USB 串口驱动之后，我们就可以开始串口下载代码了，这里我们的串口下载软件选择的是 mcuisp，该软件属于第三方软件，由单片机在线编程网提供，大家可以去 www.mcuisp.com 免费下载，本指南的光盘也附带了这个软件，版本为 V0.993。该软件启动界面如图 3.4.2.5 所示：



图 3.4.2.5 mcuisp 启动界面

然后我们选择要下载的 Hex 文件，以前面我们新建的工程为例，因为我们前面在工程建立的时候，就已经设置了生成 Hex 文件，所以编译的时候已经生成了 Hex 文件，我们只需要找到这个 Hex 文件下载即可。

用 mcuisp 软件打开 OBJ 文件夹，找到 TEST.hex，打开并进行相应设置后，如图 3.4.2.6 所示：



图 3.4.2.6 mcuisp 设置

图 4.2.6 中圈中的设置，是我们建议的设置。编程后执行，这个选项在无一键下载功能的条件下是很有用的，当选中该选项之后，可以在下载完程序之后自动运行代码。否则，还需要按复位键，才能开始运行刚刚下载的代码。

编程前重装文件，该选项也比较有用，当选中该选项之后，mcuisp 会在每次编程之前，将 hex 文件重新装载一遍，这对于代码调试的时候是比较有用的。特别提醒：不要选择使用 RamIsp，否则，可能没法正常下载。

最后，我们选择的 DTR 的低电平复位，RTS 高电平进 BootLoader，这个选择项选中，mcuisp



就会通过 DTR 和 RTS 信号来控制板载的一键下载功能电路，以实现一键下载功能。如果不选择，则无法实现一键下载功能。这个是必要的选项（在 BOOT0 接 GND 的条件下）。

在装载了 hex 文件之后，我们要下载代码还需要选择串口，这里 mcuisp 有智能串口搜索功能。每次打开 mcuisp 软件，软件会自动去搜索当前电脑上可用的串口，然后选中一个作为默认的串口（一般是你最后一次关闭时所选则的串口）。也可以通过点击菜单栏的搜索串口，来实现自动搜索当前可用串口。串口波特率则可以通过 bps 那里设置，对于 STM32，该波特率最大为 230400bps，这里我们一般选择最高的波特率：460800，让 mcuisp 自动去同步。找到 CH340 虚拟的串口，如图 3.4.2.7 所示：



图 3.4.2.7 CH340 虚拟串口

从之前 USB 串口的安装可知，开发板的 USB 串口被识别为 COM3 了（如果你的电脑是被识别为其他的串口，则选择相应的串口即可），所以我们选择 COM3。选择了相应串口之后，我们就可以通过按开始编程 (P) 这个按钮，一键下载代码到 STM32 上，下载成功后如图 3.4.2.8 所示：



图 3.4.2.8 下载完成



图 4.2.8 中，我们用圈圈圈出了 mcuisp 对一键下载电路的控制过程，其实就是控制 DTR 和 RTS 电平的变化，控制 BOOT0 和 RESET，从而实现自动下载。另外，界面提示已经下载完成（如果老提示：开始连接…，需要检查一下，开发板的设置是否正确，是否有其他因素干扰等），并且从 0X80000000 处开始运行了，我们打开串口调试助手选择 COM3，会发现从 ALIENTEK 战舰 STM32 开发板发回来的信息，如图 3.4.2.9 所示：



图 3.4.2.9 程序开始运行了

接收到的数据和我们仿真的是一样的，证明程序没有问题。至此，说明我们下载代码成功了，并且也从硬件上验证了我们代码的正确性。

3.4.3 STM32 硬件调试

上一节，我们介绍了如何通过利用串口给 STM32 下载代码，并在 ALIENTEK 战舰 STM32 开发板上验证了我们程序的正确性。这个代码比较简单，所以不需要硬件调试，我们直接就一次成功了。可是，如果你的代码工程比较大，难免存在一些 bug，这时，就有必要通过硬件调试来解决问题了。

串口只能下载代码，并不能实时跟踪调试，而利用调试工具，比如 JLINK、ULINK、STLINK 等就可以实时跟踪程序，从而找到你程序中的 bug，使你的开发事半功倍。这里我们以 JLINK V8 为例，说说如何在线调试 STM32。

JLINK V8 支持 JTAG 和 SWD，同时 STM32 也支持 JTAG 和 SWD。所以，我们有 2 种方式可以用来调试，JTAG 调试的时候，占用的 IO 线比较多，而 SWD 调试的时候占用的 IO 线很少，只需要两根即可。

JLINK V8 的驱动安装比较简单，我们在这里就不说了。在安装了 JLINK V8 的驱动之后，我们接上 JLINK V8，并把 JTAG 口插到 ALIENTEK 战舰 STM32 开发板上，打开之前 3.2 节新建的工程，点击 ，打开 Options for Target 选项卡，在 Debug 栏选择仿真工具为 Cortex-M3 J-LINK，如图 3.4.3.1 所示：

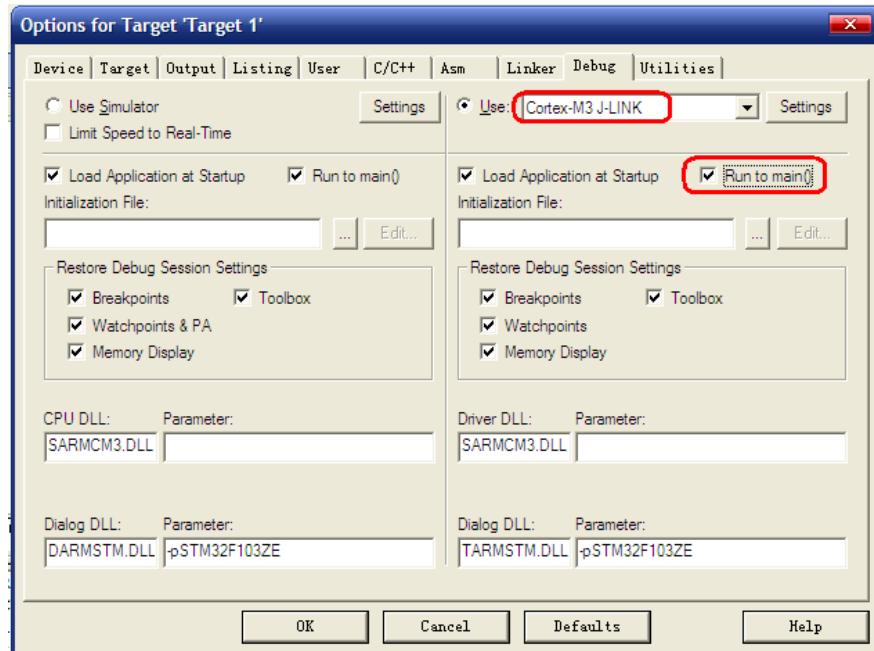


图 3.4.3.1 Debug 选项卡设置

上图中我们还勾选了 Run to main(), 该选项选中后, 只要点击仿真就会直接运行到 main 函数, 如果没选择这个选项, 则会先执行 startup_stm32f10x_hd.s 文件的 Reset_Handler, 再跳到 main 函数。

然后我们点击 Settings, 设置 J-LINK 的一些参数, 如图 3.4.3.2 所示:

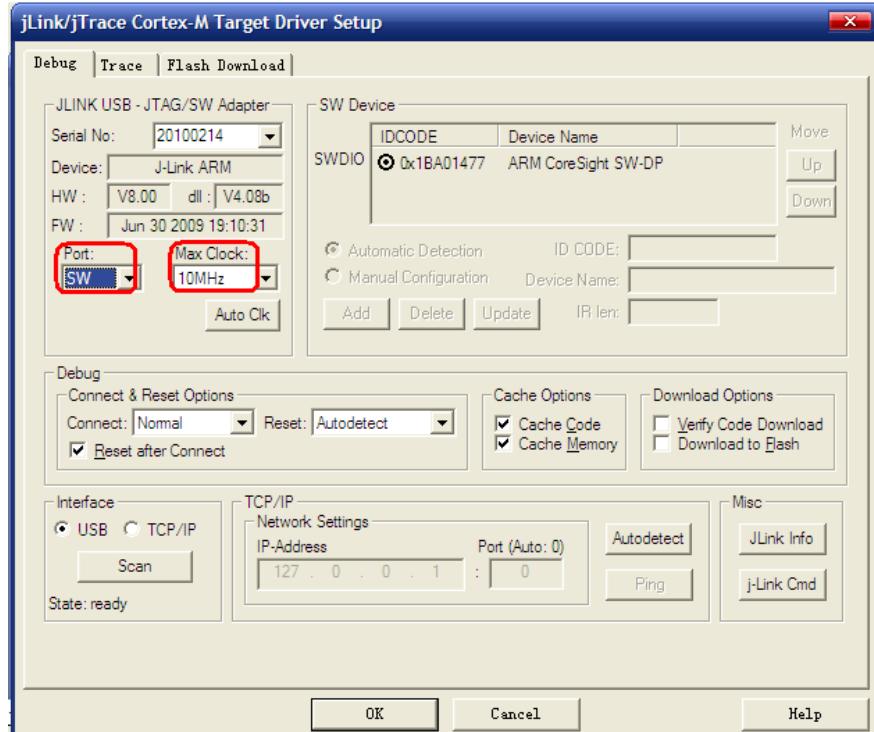


图 3.4.3.2 J-LINK 模式设置

图 4.3.2 中, 我们使用 J-LINK V8 的 SW 模式调试, 因为我们 JTAG 需要占用比 SW 模式多很多的 IO 口, 而在 ALIENTEK 战舰 STM32 开发板上这些 IO 口可能被其他外设用到, 可能造成部分外设无法使用。所以, 我们建议大家在调试的时候, 一定要选择 SW 模式。Max Clock,



可以点击 Auto Clk 来自动设置，图 4.3.2 中我们设置 SWD 的调试速度为 10Mhz，这里，如果你的 USB 数据线比较差，那么可能会出问题，此时，你可以通过降低这里的速率来试试。

单击 OK，完成此部分设置，接下来我们还需要在 Utilities 选项卡里面设置下载时的目标编程器，如图 3.4.3.3 所示：

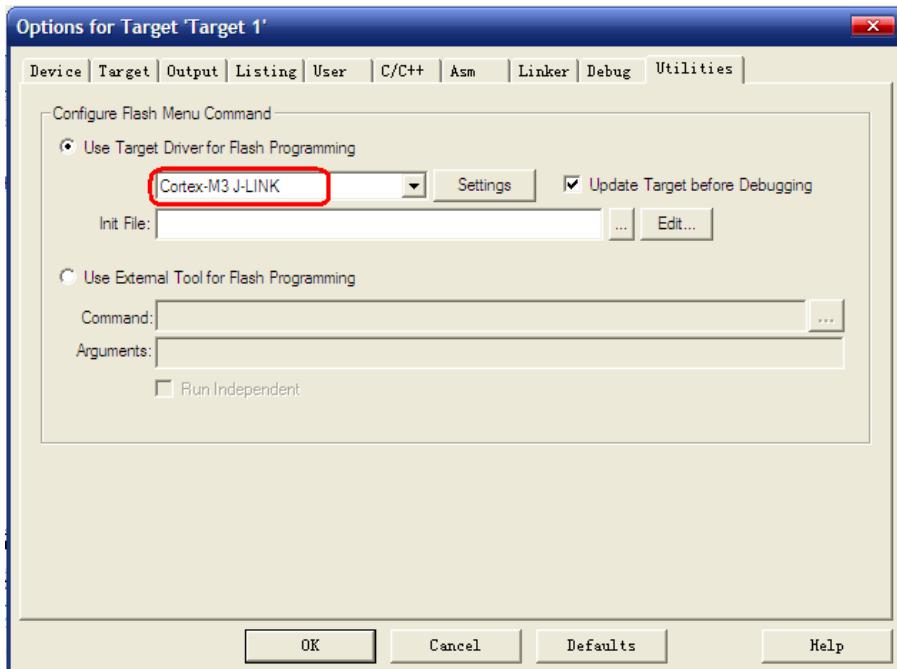


图 3.4.3.3 FLASH 编程器选择

图 3.4.3.3 中，我们选择 J-LINK 来调试 Cortex M3，然后点击 Settings，设置如图 3.4.3.4 所示：

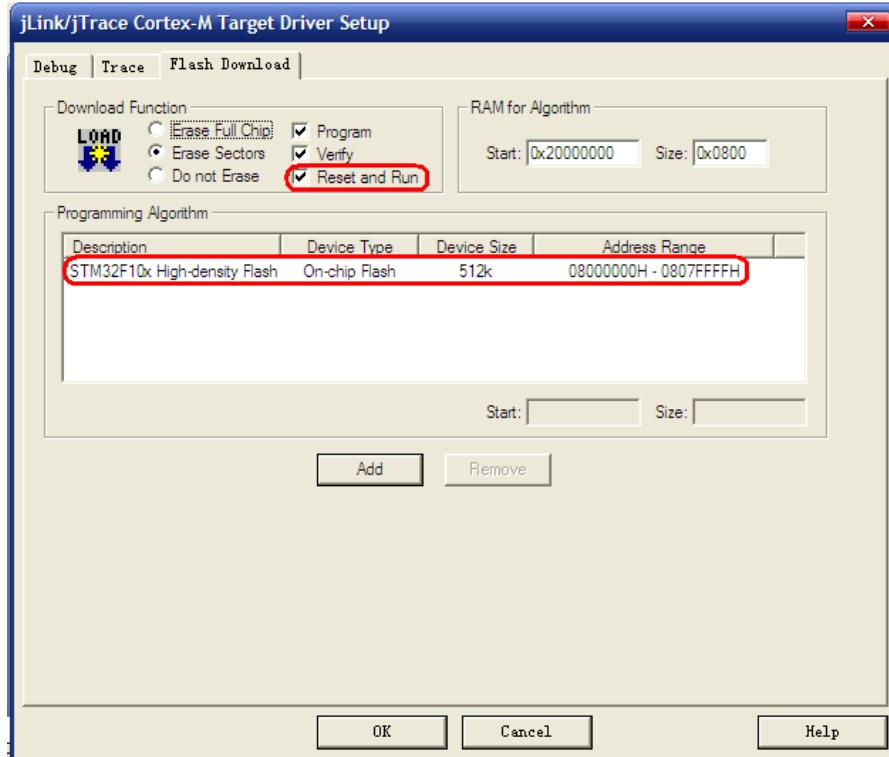




图 3.4.3.4 编程设置

这里要根据不同的 MCU 选择 FLASH 的大小, 因为我们开发板使用的是 STM32F103ZET6, 其 FLASH 大小为 512KB, 所以我们点击 Add, 并在 Programming Algorithm 里面选择 512K 型号的 STM32。然后选中 Reset and Run 选项, 以实现在编程后自动启动, 其他默认设置即可。设置完成之后, 如图 3.4.3.4 所示。

在设置完之后, 点击 OK, 然后再点击 OK, 回到 IDE 界面, 编译一下工程。再点击 ,

开始仿真 (如果开发板的代码没被更新过, 则会先更新代码, 再仿真, 你也可以通过按 , 只下载代码, 而不进入仿真。特别注意: 开发板上的 B0 和 B1 都要设置到 GND, 否则代码下载后不会自动运行的!), 如图 3.4.3.5 所示:

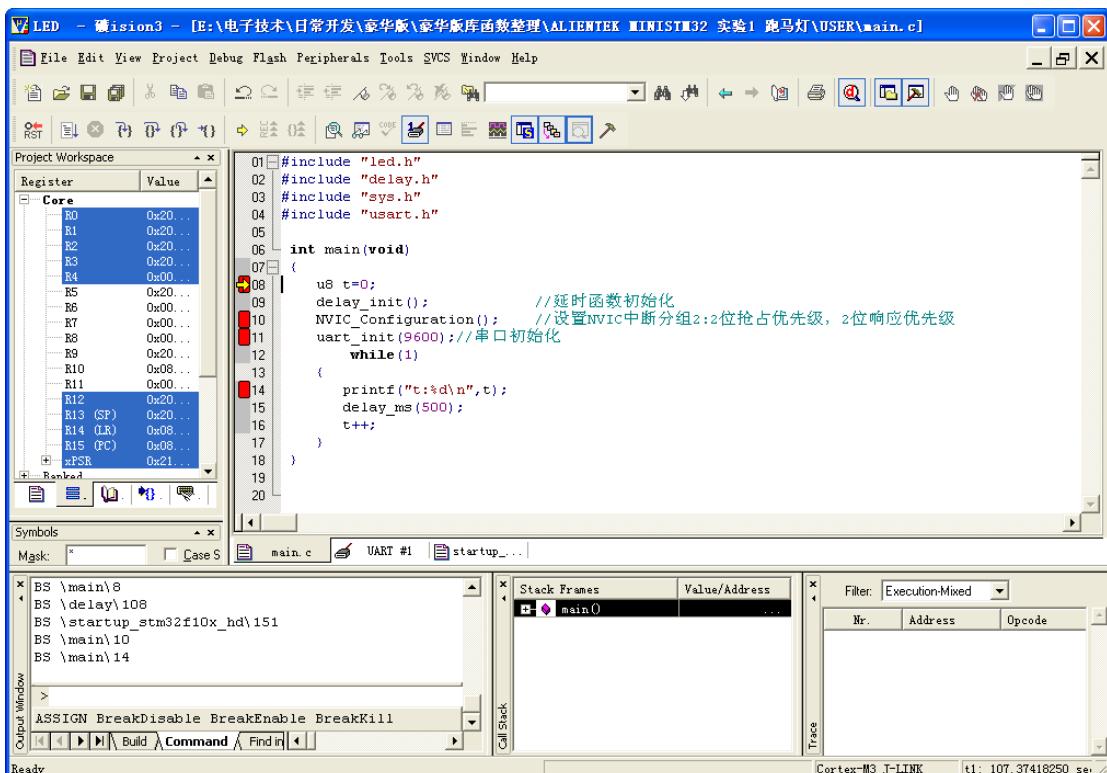


图 3.4.3.5 开始仿真

因为我们之前勾选了 Run to main() 选项, 所以, 程序直接就运行到了 main 函数的入口处, 我们在 delay_init() 处设置了一个断点, 点击 , 程序将会快速执行到该处。如图 3.4.3.6 所示:

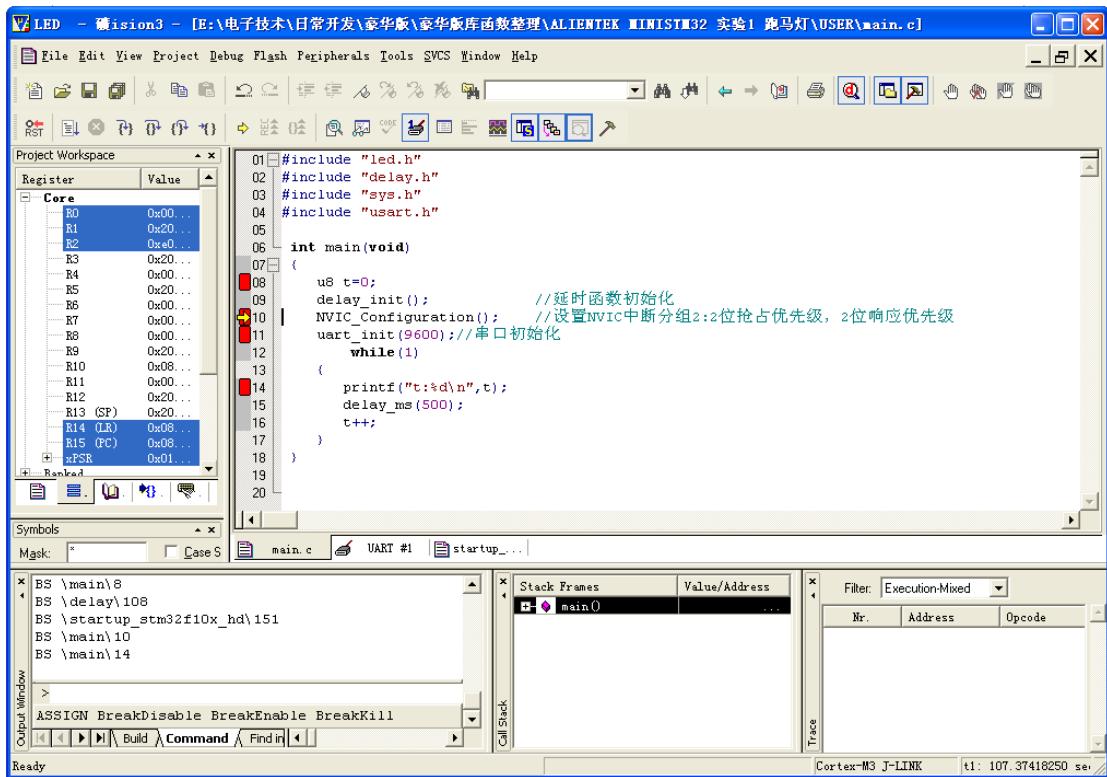


图 3.4.3.6 程序运行到断点处

接下来，我们就可以和软件仿真一样的开始操作了，不过这是真正的在硬件上的运行，而不是软件仿真！其结果更可信。硬件调试就给大家介绍到这里。

3.5 RVMDK 使用技巧

通过前面的学习，我们已经了解了如何在 RVMDK 里面建立属于自己的工程。下面，我们将向大家介绍 RVMDK 软件的一些使用技巧，这些技巧在代码编辑和编写方面会非常有用，希望大家好好掌握，最好实际操作一下，加深印象。

3.5.1 文本美化

文本美化，主要是设置一些关键字、注释、数字等的颜色和字体。前面我们在介绍 RVMDK 新建工程的时候看到界面如图 3.5.1.1 所示：

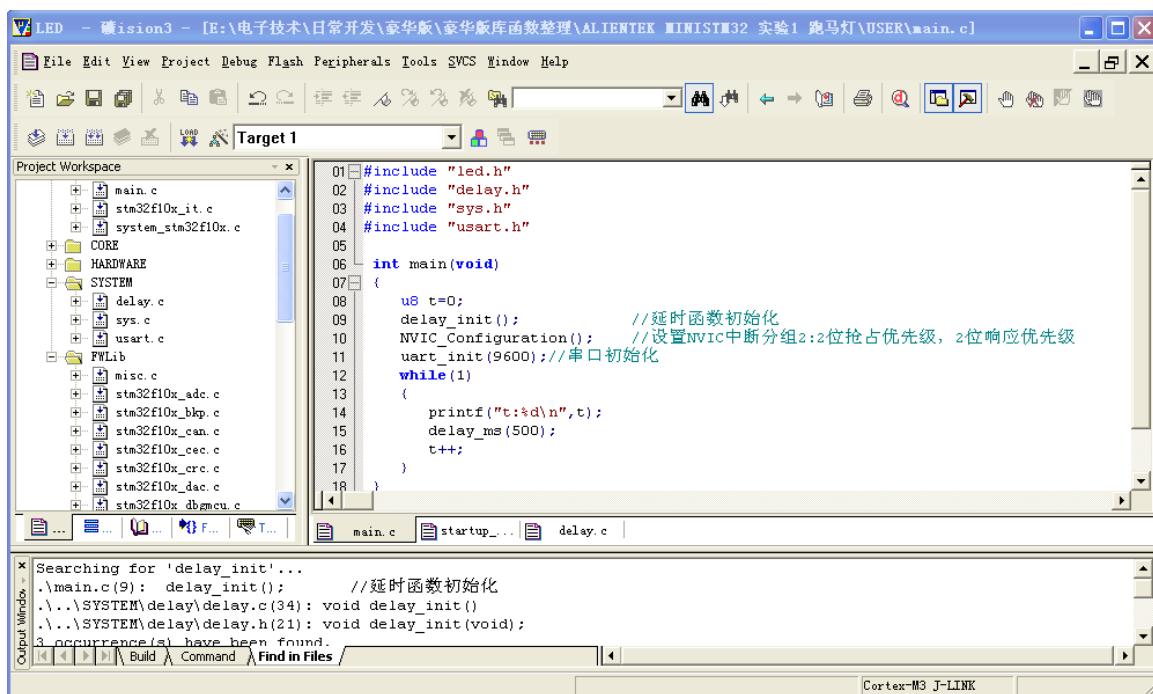


图 3.5.1.1 MDK 新建工程界面

这是 MDK 默认的设置，可以看到其中的关键字和注释等字体的颜色不是很漂亮，而 MDK 提供了我们自定义字体颜色的功能。我们可以在工具条上点击 (编辑配置对话框) 弹出如图 3.5.1.2 所示界面：

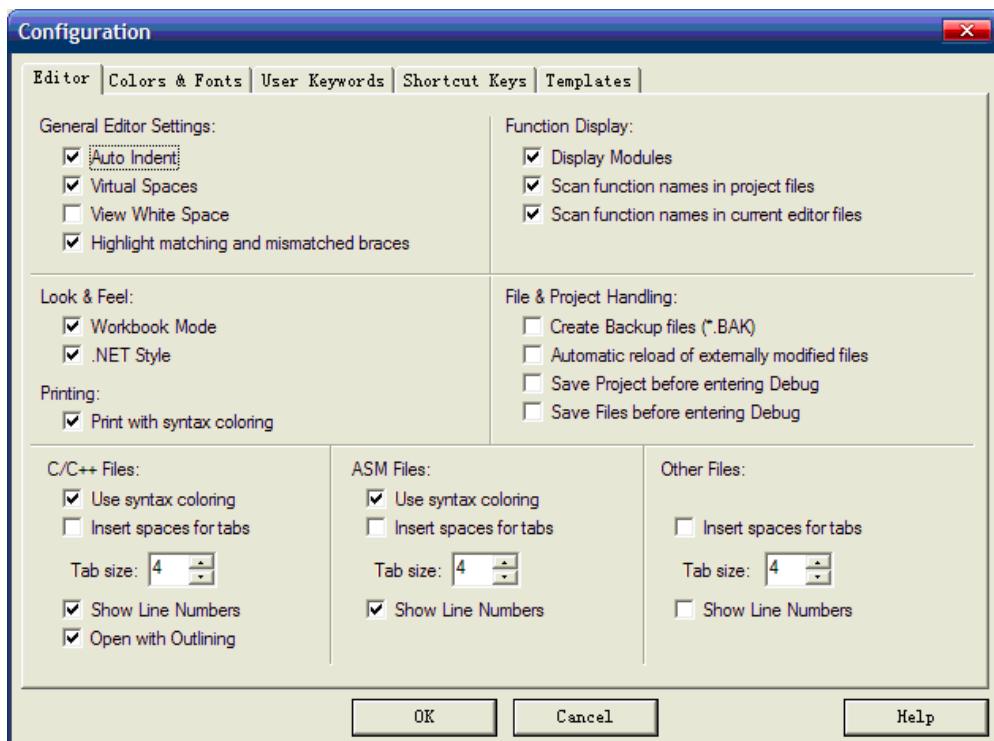


图 3.5.1.2 编辑配置对话框

在该对话框中我们选择 Colors&Fonts 选项卡，在该选项卡内，我们就可以设置自己的代码的子体和颜色了。由于我们使用的是 C 语言，故在 Text File Types 下面选择 ARM: Editor C Files



在右边就可以看到相应的元素了。如图 3.5.1.3 所示：

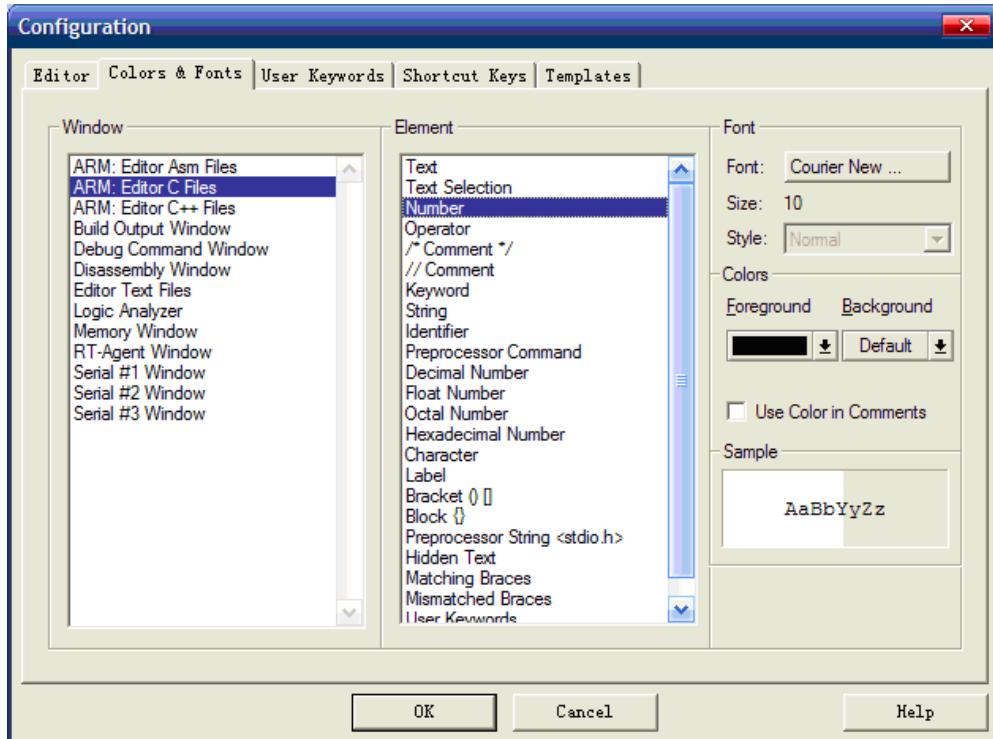


图 3.5.1.3 Colors&Fonts 选项卡

然后点击各个元素修改为你喜欢的颜色，当然也可以在 Font 栏设置你字体的类型，以及字体的大小等。设置成之后，点击 OK，就可以在主界面看到你所修改后的结果，例如我修改后的代码显示效果如图 3.5.1.4 所示：

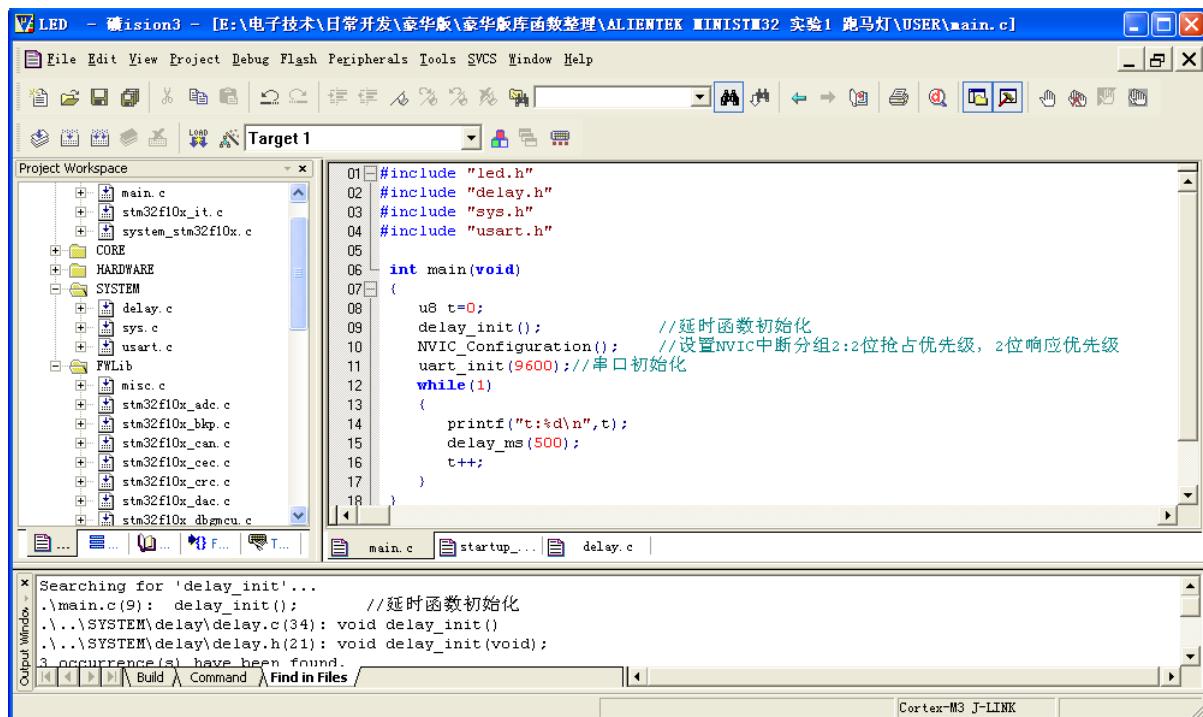


图 3.5.1.4 设置完后显示效果

这就比开始的效果好看一些了。当然你觉得字体小了可以在刚刚的对话框 Font 栏设置大一



点，觉得大了也可以设置小一点，总之设置到你认为可以为止。

细心的读者可能会发现，上面的代码里面有一个 u8，还是黑色的，这是一个用户自定义的关键字，为什么不显示蓝色（假定刚刚已经设置了用户自定义关键字颜色为蓝色）呢？这就又要回到我们刚刚的配置对话框了，单这次我们要选择 User Keywords 选项卡，同样选择 ARM: Editor C Files，在右边的 User Keywords 对话框下面输入你自己定义的关键字，如图 3.5.1.5 所示：

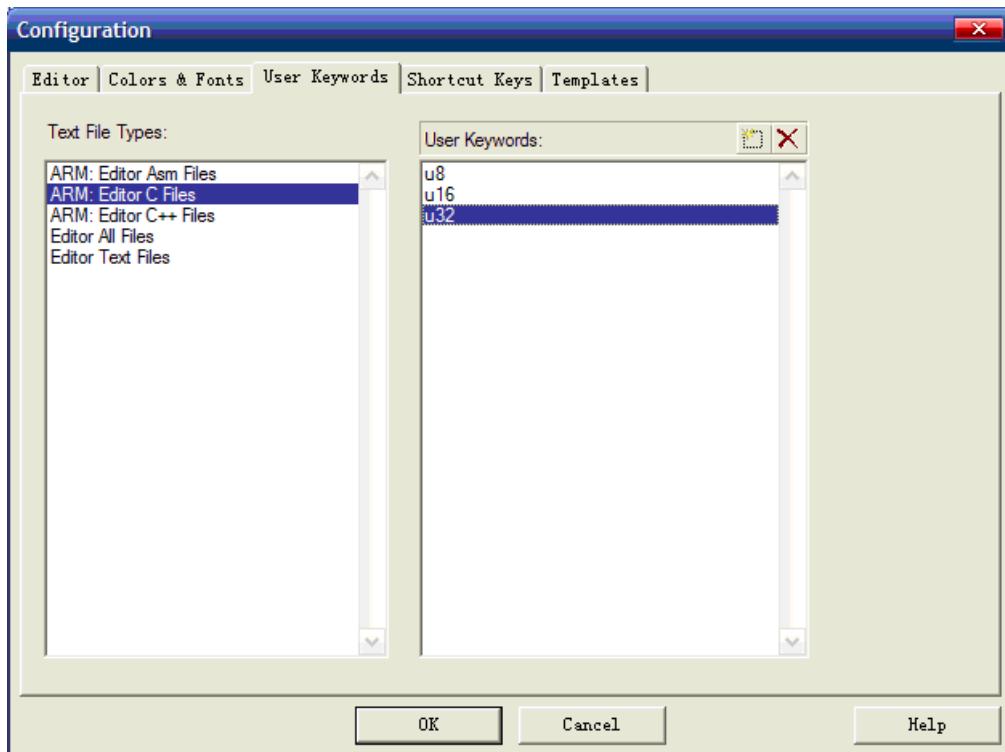


图 3.5.1.5 用户自定义关键字

图 3.5.5 中我定义了 u8、u16、u32 等 3 个关键字，这样在以后的代码编辑里面只要出现这三个关键字，肯定就会变成蓝色。点击 OK，再回到主界面，可以看到 u8 变成了蓝色了，如图 3.5.1.6 所示：

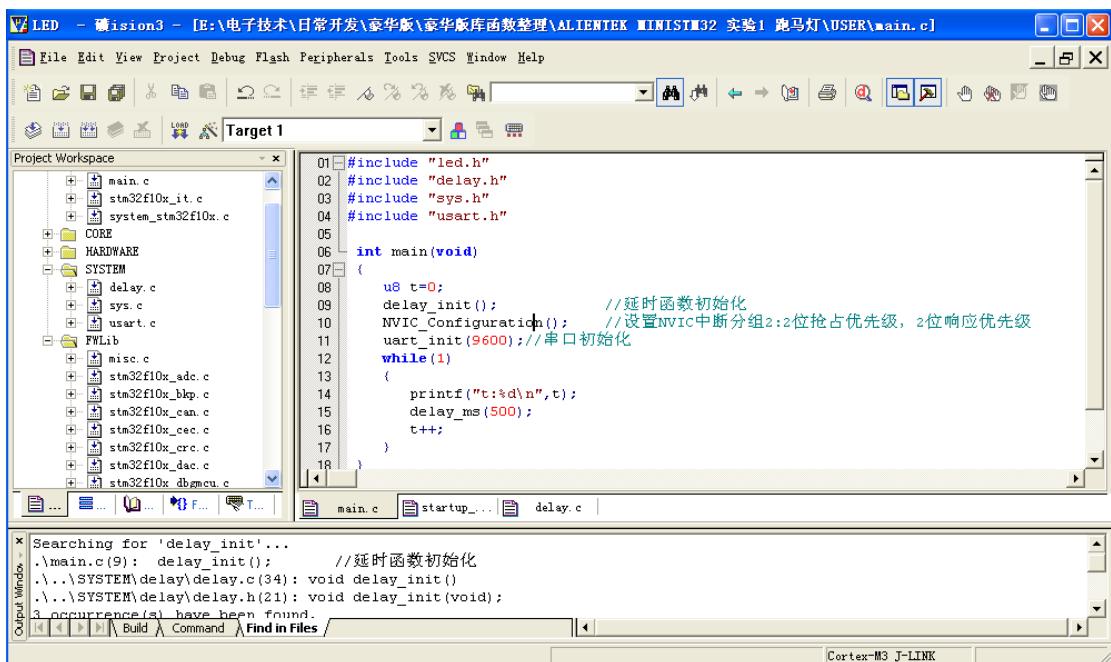


图 3.5.1.6 设置完后显示效果

其实这个编辑配置对话框里面，还可以对其他很多功能进行设置，比如按 TAB 键右移多少位，快捷键修改等，有兴趣的大家可以自己摸索一下。文本美化的技巧就为大家介绍到这里，接下来我们为大家介绍 RVMDK 的代码编辑技巧。

3.5.2 代码编辑技巧

这里给大家介绍几个我常用的技巧，这些小技巧能给我们的代码编辑带来很大的方便，相信对你的代码编写一定会有所帮助。

1) TAB 键的妙用

首先要介绍的就是 TAB 键的使用，这个键在很多编译器里面都是用来空位的，每按一下移空几个位。如果你是经常编写程序的对这个键一定再熟悉不过了。但是 MDK 的 TAB 键和一般编译器的 TAB 键有不同的地方，和 C++ 的 TAB 键差不多。MDK 的 TAB 键支持块操作。也就是可以让一片代码整体右移固定的几个位，也可以通过 SHIFT+TAB 键整体左移固定的几个位。

假设我们前面的串口 1 中断响应函数如图 3.5.2.1 所示：



```

070 void USART1_IRQHandler(void)
071 {
072     u8 res;
073     #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
074     OSIntEnter();
075     #endif
076     if(USART1->SR&(1<<5))//接收到数据
077     {
078         res=USART1->DR;
079         if((USART_RX_STA&0x8000)==0)//接收未完成
080         {
081             if(USART_RX_STA&0x4000)//接收到了0xd
082             {
083                 if(res!=0xa) USART_RX_STA=0;//接收错误,重新开始
084                 else USART_RX_STA|=0x8000; //接收完成了
085             }else //还没收到0xd
086             {
087                 if(res==0xd) USART_RX_STA|=0x4000;
088             else
089             {
090                 USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
091                 USART_RX_STA++;
092                 if(USART_RX_STA>(USART_REC_LEN-1)) USART_RX_STA=0;//接收数据错误,重新开始接收
093             }
094         }
095     }
096 }
097 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
098 OSIntExit();
099 #endif
100 }
```

图 3.5.2.1 头大的代码

图 3.5.2.1 中这样的代码大家肯定不会喜欢, 这还只是短短的 30 来行代码, 如果你的代码有几千行, 全部是这个样子, 不头大才怪。看到这样的代码我们就可以通过 TAB 键的妙用来快速修改为比较规范的代码格式。

选中一块然后按 TAB 键, 你可以看到整块代码都跟着右移了一定距离, 如图 3.5.2.2 所示:

```

070 void USART1_IRQHandler(void)
071 {
072     u8 res;
073     #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
074     OSIntEnter();
075     #endif
076     if(USART1->SR&(1<<5))//接收到数据
077     {
078         res=USART1->DR;
079         if((USART_RX_STA&0x8000)==0)//接收未完成
080         {
081             if(USART_RX_STA&0x4000)//接收到了0xd
082             {
083                 if(res!=0xa) USART_RX_STA=0;//接收错误,重新开始
084                 else USART_RX_STA|=0x8000; //接收完成了
085             }else //还没收到0xd
086             {
087                 if(res==0xd) USART_RX_STA|=0x4000;
088             else
089             {
090                 USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
091                 USART_RX_STA++;
092                 if(USART_RX_STA>(USART_REC_LEN-1)) USART_RX_STA=0;//接收数据错误,重新开始接收
093             }
094         }
095     }
096 }
097 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
098 OSIntExit();
099 #endif
100 }
```

图 3.5.2.2 代码整体偏移

接下来我们就是要多选几次, 然后多按几次 TAB 键就可以达到迅速使代码规范化的目的,



最终效果如图 3.5.2.3 所示

```

070 void USART1_IRQHandler(void)
071 {
072     u8 res;
073 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
074     OSIntEnter();
075 #endif
076     if(USART1->SR&(1<<5))//接收到数据
077     {
078         res=USART1->DR;
079         if((USART_RX_STA&0x8000)==0)//接收未完成
080         {
081             if(USART_RX_STA&0x4000)//接收到了0xd
082             {
083                 if(res!=0xa)USART_RX_STA=0;//接收错误,重新开始
084                 else USART_RX_STA|=0x8000; //接收完成了
085             }else //还没收到0xd
086             {
087                 if(res==0xd)USART_RX_STA|=0x4000;
088                 else
089                 {
090                     USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
091                     USART_RX_STA++;
092                     if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
093                 }
094             }
095         }
096     }
097 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
098     OSIntExit();
099 #endif
100 }
```

图 3.5.2.3 修改后的代码

图 3.5.2.3 中的代码相对于图 3.5.2.1 中的要好看多了，经过这样的整理之后，整个代码一下就变得有条理多了，看起来很舒服。

2) 快速定位函数/变量被定义的地方

上一节，我们介绍了 TAB 键的功能，接下来我们介绍一下如何快速查看一个函数或者变量所定义的地方。

大家在调试代码或编写代码的时候，一定有想看看某个函数是在那个地方定义的，具体里面的内容是怎么样的，也可能想看看某个变量或数组是在哪个地方定义的等。尤其在调试代码或者看别人代码的时候，如果编译器没有快速定位的功能的时候，你只能慢慢的自己找，代码量比较少还好，如果代码量一大，那就郁闷了，有时候要花很久的时间来找这个函数到底在哪里。型号 MDK 提供了这样的快速定位的功能（顺便说一下 CAVR 的 2.0 以后的版本也有这个功能）。只要你把光标放到这个函数/变量（xxx）的上面（xxx 为你要查看的函数或变量的名字），然后右键，弹出如图 3.3.2.4 所示的菜单栏：

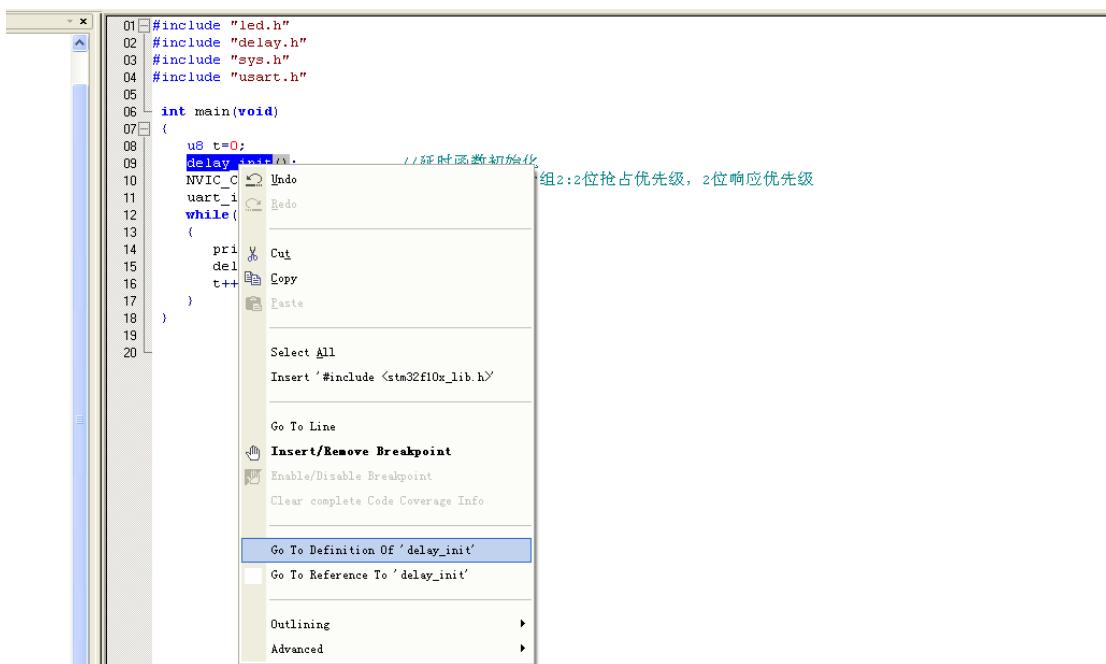


图 3.3.2.4 快速定位

在图 3.3.2.4 中，我们找到 Go to Definition Of ‘delay_init’ 这个地方，然后单击左键就可以快速跳到 delay_init 函数的定义处（注意要先在 Options for Target 的 Output 选项卡里面勾选 Browse Information 选项，再编译，再定位，否则无法定位！）。如图 3.3.2.5 所示：

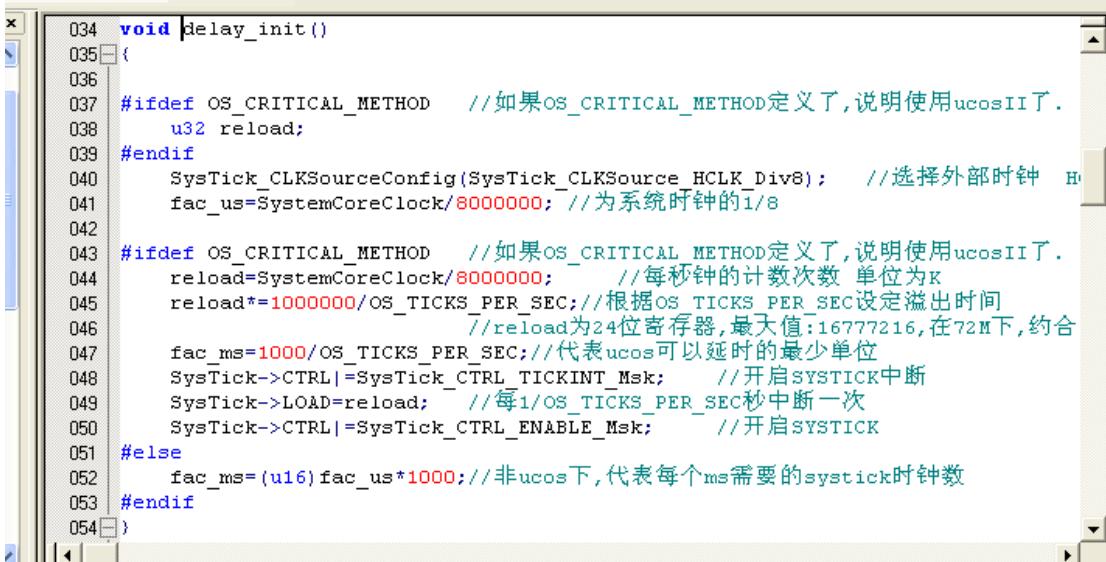


图 3.3.2.5 定位结果

对于变量，我们也可以按这样的操作快速来定位这个变量被定义的地方，大大缩短了你查找代码的时间。细心的大家会发现上面还有一个类似的选项，就是 Go to Reference To ‘delay_init’，这个是快速跳到该函数被声明的地方，有时候也会用到，但不如前者使用得多。

很多时候，我们利用 Go to Definition/ Reference 看完函数/变量的定义/申明后，又想返回之

前的代码继续看，此时我们可以通过 IDE 上的 按钮（Back to previous position）快速的返回之前的位置，这个按钮非常好用！



3) 快速注释与快速消注释

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

以 delay_init 函数为例，比如我要注释掉下图中所选中区域的代码，如图 3.5.2.6 所示：

```

034 void delay_init()
035 {
036
037 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
038     u32 reload;
039 #endif
040     SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8); //选择外部时钟 H
041     fac_us=SystemCoreClock/8000000; //为系统时钟的1/8
042
043 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
044     reload=SystemCoreClock/8000000; //每秒钟的计数次数 单位为K
045     reload*=1000000/OS_TICKS_PER_SEC;//根据OS TICKS PER SEC设定溢出时间
046 //reload为24位寄存器,最大值:16777216,在72M下,约合
047     fac_ms=1000/OS_TICKS_PER_SEC;//代表ucos可以延时的最少单位
048     SysTick->CTRL|=SysTick_CTRL_TICKINT_Msk; //开启SYSTICK中断
049     SysTick->LOAD=reload; //每1/OS_TICKS_PER_SEC秒中断一次
050     SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk; //开启SYSTICK
051 #else
052     fac_ms=(u16)fac_us*1000;//非ucos下,代表每个ms需要的systick时钟数
053 #endif
054 }

```

图 3.5.2.6 选中要注释的区域

我们只要在选中了之后，选择右键，再选择 Advanced->Comment Selection 就可以把这段代码注释掉了。执行这个操作以后的结果如图 3.5.2.7 所示：

```

034 void delay_init()
035 {
036
037 // #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII
038 //     u32 reload;
039 // #endif
040 //     SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8); //选择外部时钟 H
041 //     fac_us=SystemCoreClock/8000000; //为系统时钟的1/8
042 //
043 // #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII
044 //     reload=SystemCoreClock/8000000; //每秒钟的计数次数 单位为K
045 //     reload*=1000000/OS_TICKS_PER_SEC;//根据OS TICKS PER SEC设定溢出时间
046 // //reload为24位寄存器,最大值:16777216,在72M下,约合
047 //     fac_ms=1000/OS_TICKS_PER_SEC;//代表ucos可以延时的最少单位
048 //     SysTick->CTRL|=SysTick_CTRL_TICKINT_Msk; //开启SYSTICK中断
049 //     SysTick->LOAD=reload; //每1/OS_TICKS_PER_SEC秒中断一次
050 //     SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk; //开启SYSTICK
051 // #else
052 //     fac_ms=(u16)fac_us*1000;//非ucos下,代表每个ms需要的systick时钟数
053 // #endif
054 }

```

图 3.5.2.7 注释完毕

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



3.5.3 其他小技巧

除了前面介绍的几个比较常用的技巧，这里还介绍几个其他的小技巧，希望能让你的代码编写如虎添翼。

第一个是快速打开头文件。在将光标放到要打开的引用头文件上，然后右键选择 Open Document “XXX”，就可以快速打开这个文件了（XXX 是你要打开的头文件名字）。如图 3.5.3.1 所示：

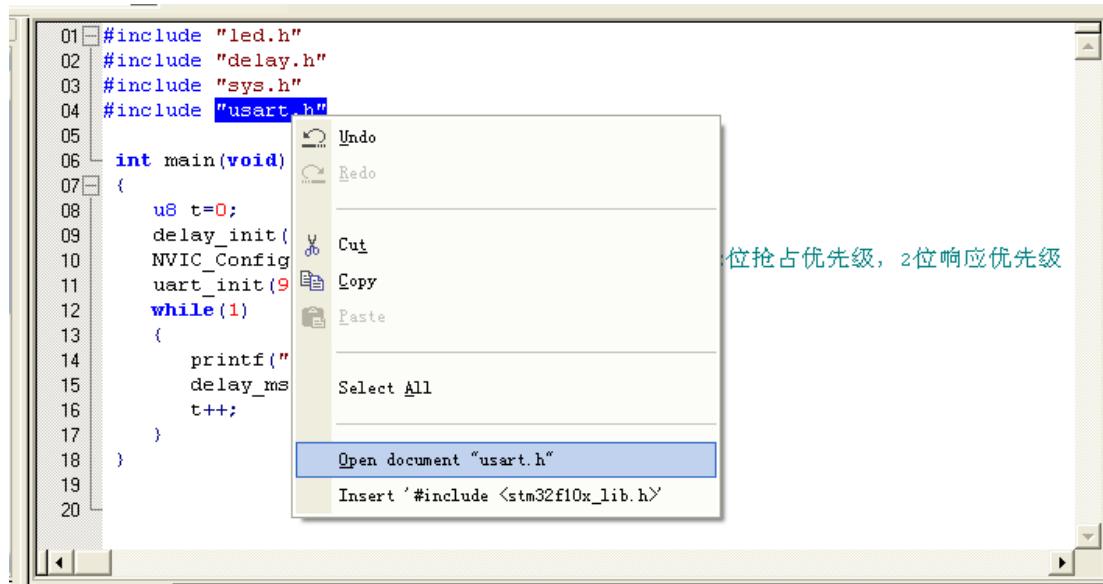


图 3.5.3.1 快速打开头文件

第二个小技巧是查找替换功能。这个和 WORD 等很多文档操作的替换功能是差不多的，在 MDK 里面查找替换的快捷键是“CTRL+H”，只要你按下该按钮就会调出如图 3.3.2 所示界面：

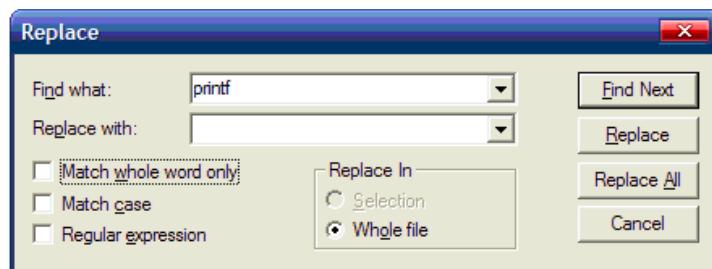


图 3.5.3.2 替换文本

这个替换的功能在有的时候是很有用的，它的用法与其他编辑工具或编译器的差不多，相信各位都不陌生了，这里就不在啰唆了。

第三个小技巧是跨文件查找功能，先双击你要找的函数/变量名（这里我们还是以系统时钟初始化函数：delay_init 为例），然后再点击 IDE 上面的 ，弹出如图 3.5.3.3 所示对话框：

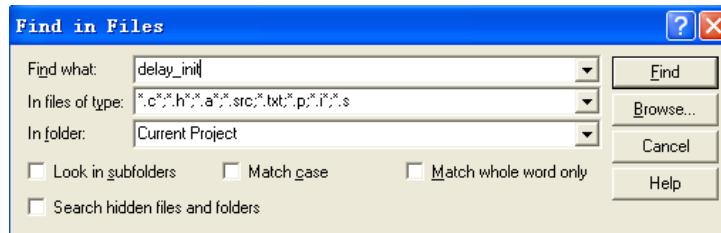


图 3.5.3.3 跨文件查找

点击 Find, MDK 就会帮你找出所有含有 delay_init 字段的文件并列出其所在位置, 如图 3.5.3.4 所示:

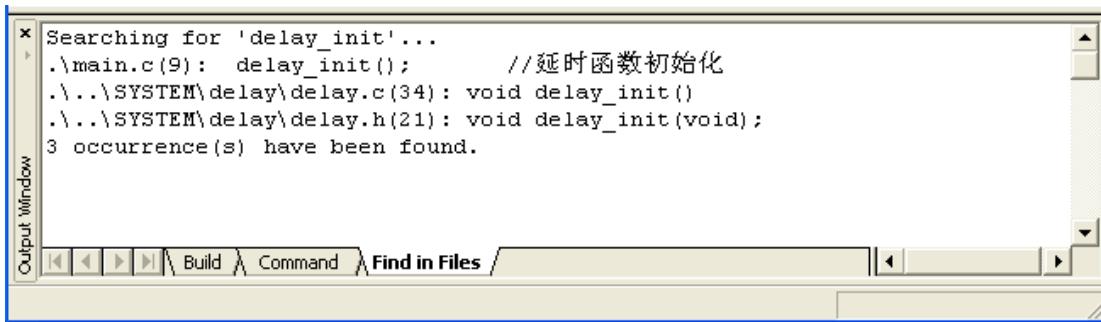


图 3.5.3.4 查找结果

该方法可以很方便的查找各种函数/变量, 而且可以限定搜索范围(比如只查找.c 文件和.h 文件等), 是非常实用的一个技巧。

3.5.4 调试技巧

MDK 自带了很多基础例程, 在这里我们要介绍的是如何利用 MDK 自带的这些例程来快速编写代码。当我开始接触 STM32 的时候, 基本上没有直接操作寄存器的例子, 网上的所有例子几乎都是使用 ST 自带的例程。也就是 MDK 自带的使用库函数的版本。所以在开始的时候, 我就根据 MDK 提供的例子, 对照数据手册, 再结合 MDK 提供的查看寄存器的功能, 一步步把带库函数的例子改成了直接操作寄存器的。一开始会比较难, 但是当你掌握规律了之后, 就可以很快的把 MDK 的使用库函数的代码改为直接操作寄存器的代码。这里总结几个要点:

1, 一款实用的开发板。

这个是实验的基础, 有时候软件仿真通过了, 在板上并不一定能跑起来, 而且有个开发板在手, 什么东西都可以直观的看到, 效果不是仿真能比的。但开发板不宜多, 多了的话连自己都不知道该学哪个了, 觉得这个也还可以, 那个也不错, 那就这个学半天, 那个学半天, 结果学个四不像。倒不如从一而终, 学完一个在学另外一个。

2, 两本参考资料, 即《STM32 参考手册》和《Cortex-M3 权威指南》。

《STM32 参考手册》是 ST 出的官方资料, 有 STM32 的详细介绍, 包括了 STM32 的各种寄存器定义以及功能等, 是学习 STM32 的必备资料之一。而《Cortex-M3 权威指南》则是对《STM32 参考手册》的补充, 后者一般认为使用 STM32 的人都对 CM3 有了较深的了解, 所以 Cortex-M3 的很多东西它只是一笔带过, 但前者对 Cortex-M3 有非常详细的说明, 这样两者搭配, 你就基本上任何问题都能得到解决了。

3, 多做实验, 多做笔记。

一个初学者, 一开始对 STM32 一般是没有概念的, 所以首先要做的就是多做实验,



一定要相信实践出真知，结合上面 2 本手册，你很快就会熟悉 STM32，进而随心所欲。其次要多做笔记，在你不知道的时候，找 MDK 的例子，找第二点中的两本手册，当你碰到新的知识点的时候，把它记下来，俗话说：好记性不如烂笔头。将你刚学到的东西用笔记下了，对以后没有坏处。

只要以上三点做好了，学习 STM32 基本上就不会有什么问题了。当你有需要用的东西，自己写代码写不出来了，就可以在 MDK 自带的例子中找找，看看是否有相关的例程。对于 STM32 的外设，MDK 基本都是带有例程的，所以一般你的问题，可以在 MDK 自带的例程中找到答案。

MDK 的例子分为 2 部分，一部分是与 USB 无关的，这部分代码存放在：D:\KEIL3.80A\ARM\Examples\ST\STM32F10xFWLib\Examples 目录下，而另外一部分与 USB 相关的例子则存放在：D:\KEIL3.80A\ARM\Examples\ST\STM32F10xUSBLib\Examples 目录下（D 盘是我 MDK3.80A 的安装盘，所以这里路径是这样的，如果你安装在其他位置，修改为相应的目录即可以）。

接下来我们用一个实例，来说明如何参考 MDK 的例子为自己所用。希望能起到抛砖引玉的作用。这里以一个 IO 口翻转为例，其实就是 LED 的闪烁，看看如何借用 MDK 的代码。首先打开 D:\KEIL3.80A\ARM\Examples\ST\STM32F10xFWLib\Examples 目录，可以看到很多例子，如图 3.5.4.1 所示：

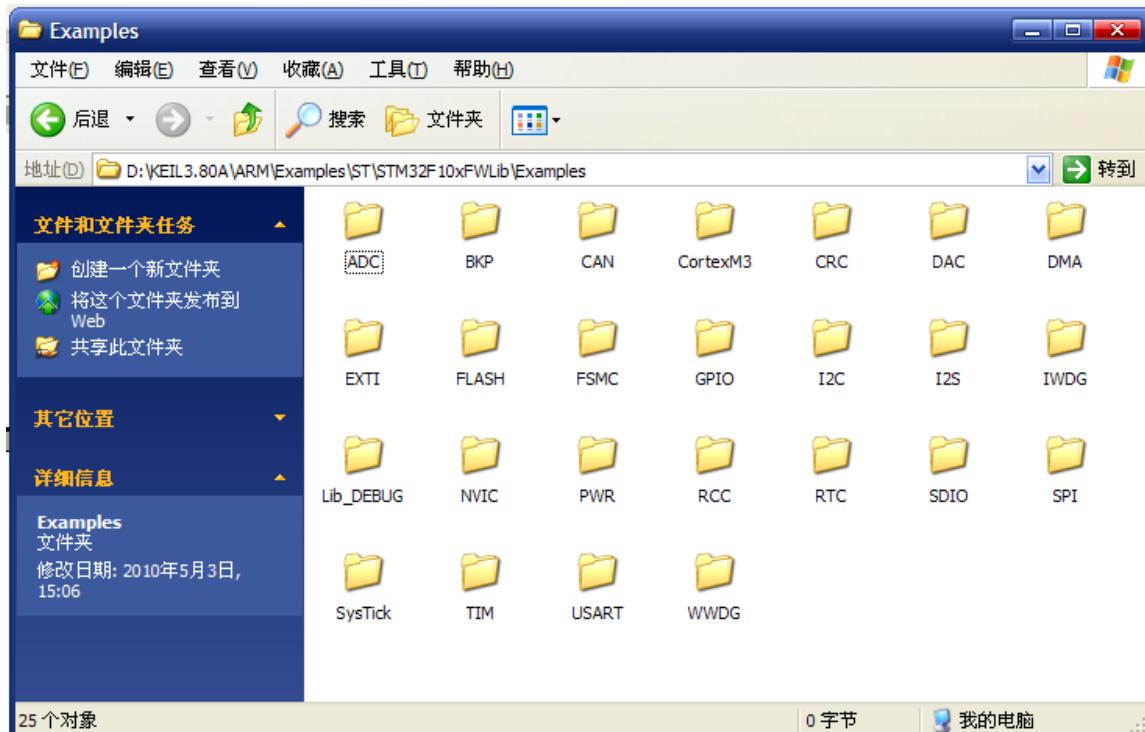


图 3.5.4.1 ST 提供的例程

IO 口翻转的例子在 GPIO 目录下的 IOToggle 下，我们将这个目录下面的所有文件拷贝到 D:\KEIL3.80A\ARM\Examples\ST\STM32F10xFWLib\Project 里面，这里会提示如图 3.5.4.2 所示的信息：



图 3.5.4.2 替换原有文件

我们选择全部就可以了。然后单击 Project.Uv2，打开工程，如图 3.5.4.3 所示：

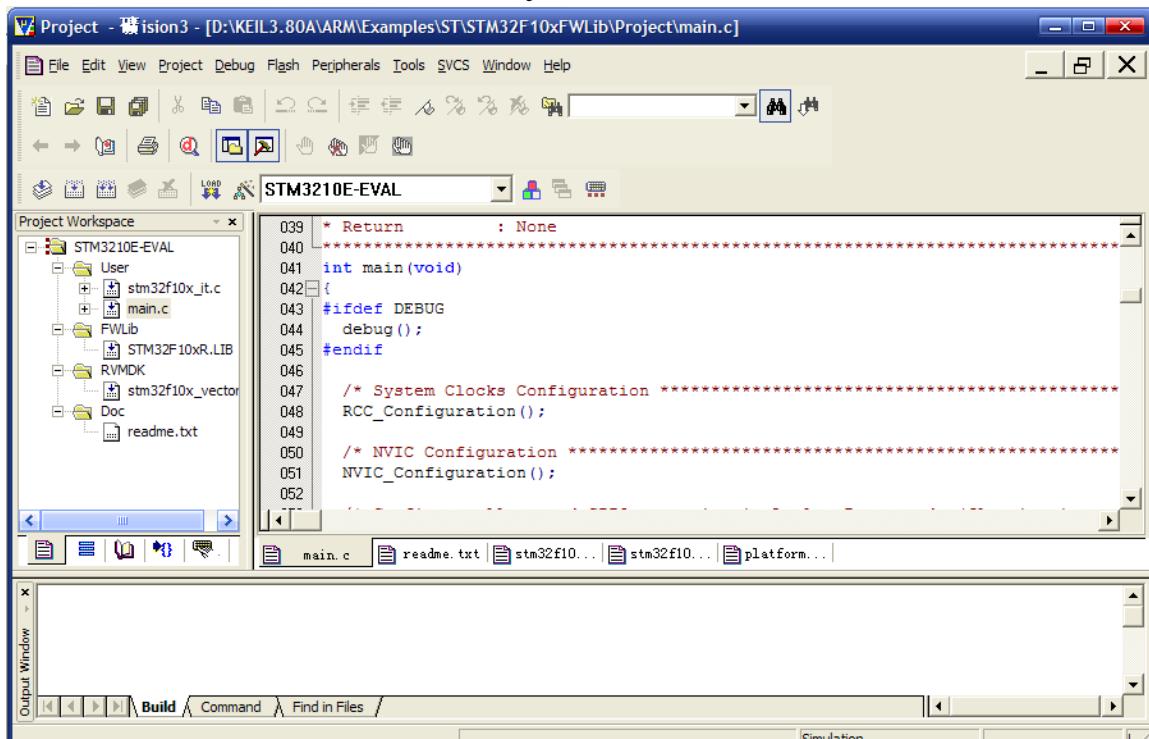


图 3.5.4.3 替换后的工程

然后点击 ，编译一遍。可以看到如图 3.3.4.4 所示的编译结果：

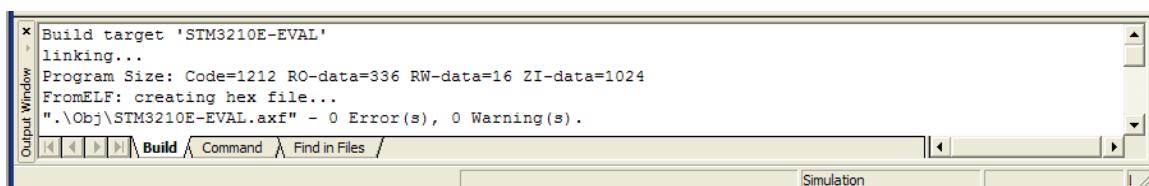


图 3.5.4.4 编译新工程

提示没有错误，没有警告。说明这个工程是可以用的。关于这个工程是如何使用的，在 readme.txt 里面是有详细说明的，在使用之前最好先看看这个说明。重点看看硬件环境的说明，如图 3.5.4.5 所示：

```
Hardware environment
=====
This example runs on STMicroelectronics STM3210B-EVAL and STM3210E-EVAL evaluation
boards and can be easily tailored to any other hardware.
To select the STMicroelectronics evaluation board used to run the example, uncomment
the corresponding line in platform_config.h file.

+ STM3210B-EVAL
  - Use LD1, LD2, LD3 and LD4 leds connected respectively to PC.06, PC.07, PC.08
    and PC.09 pins

+ STM3210E-EVAL
  - Use LD1, LD2, LD3 and LD4 leds connected respectively to PF.06, PF.07, PF.08
    and PF.09 pins
```

图 3.5.4.5 查看工程说明

从图 3.5.4.5 的说明可以知道，这个 LED 的翻转程序，对两款板子（STM3210B-EVAL 和 STM3210E-EVAL）分别是连在哪几个 IO 口上的，我们这个是在 USE_STM3210E_EVAL 板上运行的，所以使用的是 PF.6~9。

接下来我们要做的就是一步步跟踪代码，然后针对你的疑问点，打开 Peripherals 里面的相关外设，查看寄存器，看看 MDK 的示例代码是如何一步步修改里面的寄存器来实现的。对于外设的配置，MDK 一般都是调用库函数实现的，无法直接查看，这就需要你对照手册，慢慢摸索了，根据从寄存器看到的结果，大概也就能推出 MDK 是如何实现这样的操作了。其次一个重要的方法是通过查看汇编代码，来看到底是如何操作的，由于作者对汇编不熟悉，这里就不废话了，免得误导大家。

这样对照着 MDK 的例子，看看自己的代码在哪些地方和它有不一样的地方，如果出了问题，很可能就在这些不同的地方，只要根据 MDK 的示例来修改，一般你的问题就能得到解决。当然，这过程中需要多多查看手册，看看手册里怎么说的，MDK 又是怎么做的。



第四章 STM32 开发基础知识入门

这一章，我们将着重 STM32 开发的一些基础知识，让大家对 STM32 开发有一个初步的了解，为后面 STM32 的学习做一个铺垫，方便后面的学习。这一章的内容大家第一次看的时候可以只了解一个大概，后面需要用到这方面的知识的时候再回过头来仔细看看。这章我们分 7 个小结，

- 4.1 MDK 下 C 语言基础复习
- 4.2 STM32 系统架构
- 4.3 STM32 时钟系统
- 4.4 端口复用和重映射
- 4.5 STM32 NVIC 中断管理
- 4.6 MDK 中寄存器地址名称映射分析
- 4.7 MDK 固件库快速开发技巧

4.1 MDK 下 C 语言基础复习

这一节我们主要讲解一下 C 语言基础知识。C 语言知识博大精深，也不是我们三言两语能讲解清楚，同时我们相信学 STM32 这种级别 MCU 的用户，C 语言基础应该都是没问题的。我们这里主要是简单的复习一下几个 C 语言基础知识点，引导那些 C 语言基础知识不是很扎实的用户能够快速开发 STM32 程序。同时希望这些用户能够多去复习一下 C 语言基础知识，C 语言毕竟是单片机开发中的必备基础知识。对于 C 语言基础比较扎实的用户，这部分知识可以忽略不看。

4.1.1 位操作

C 语言位操作相信学过 C 语言的人都不陌生了，简而言之，就是对基本类型变量可以在位级别进行操作。这节的内容很多朋友都应该很熟练了，我这里也就点到为止，不深入探讨。下面我们先讲解几种位操作符，然后讲解位操作使用技巧。

C 语言支持如下 6 中位操作

运算符	含义	运算符	含义
&	按位与	~	取反
	按位或	<<	左移
^	按位异或	>>	右移

表 4.1.1.16 种位操作

这些与或非，取反，异或，右移，左移这些到底怎么回事，这里我们就不多做详细，相信大家学 C 语言的时候都学习过了。如果不懂的话，可以百度一下，非常多的知识讲解这些操作符。下面我们就着重讲解位操作在单片机开发中的一些实用技巧。

- 1) 不改变其他位的值的状况下，对某几个位进行设值。

这个场景单片机开发中经常使用，方法就是先对需要设置的位用&操作符进行清零操作，然后用|操作符设值。比如我要改变 GPIOA 的状态，可以先对寄存器的值进行&清零操作

```
GPIOA->CRL&=0XFFFFFF0F; //将第 4-7 位清 0
```

然后再与需要设置的值进行|或运算

```
GPIOA->CRL|=0X00000040; //设置相应位的值，不改变其他位的值
```

- 2) 移位操作提高代码的可读性。

移位操作在单片机开发中也非常重要，下面让我们看看固件库的 GPIO 初始化的函数里面的一行代码

```
GPIOx->BSRR = (((uint32_t)0x01) << pinpos);
```

这个操作就是将 BSRR 寄存器的第 pinpos 位设置为 1，为什么要通过左移而不是直接设置一个固定的值呢？其实，这是为了提高代码的可读性以及可重用性。这行代码可以很直观明了的知道，是将第 pinpos 位设置为 1。如果你写成

```
GPIOx->BSRR = 0x0030;
```

这样的代码就不好看也不好重用了。

类似这样的代码很多：

```
GPIOA->ODR|=1<<5; //PA.5 输出高,不改变其他位
```

这样我们一目了然，5 告诉我们是第 5 位也就是第 6 个端口，1 告诉我们是设置为 1 了。

3) ~取反操作使用技巧

SR 寄存器的每一位都代表一个状态，某个时刻我们希望去设置某一位的值为 0，同时其他位都保留为 1，简单的作法是直接给寄存器设置一个值：

```
TIMx->SR=0xFFFF7;
```

这样的作法设置第 3 位为 0，但是这样的作法同样不好看，并且可读性很差。看看库函数代码中怎样使用的：

```
TIMx->SR = (uint16_t)~TIM_FLAG;
```

而 TIM_FLAG 是通过宏定义定义的值：

#define TIM_FLAG_Update	((uint16_t)0x0001)
#define TIM_FLAG_CC1	((uint16_t)0x0002)

看这个应该很容易明白，可以直接从宏定义中看出 TIM_FLAG_Update 就是设置的第 0 位了，可读性非常强。

4.1.2 define 宏定义

define 是 C 语言中的预处理命令，它用于宏定义，可以提高源代码的可读性，为编程提供方便。常见的格式：

```
#define 标识符 字符串
```

“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。例如：

```
#define SYSCLK_FREQ_72MHz 72000000
```

定义标识符 SYSCLK_FREQ_72MHz 的值为 72000000。

至于 define 宏定义的其他一些知识，比如宏定义带参数这里我们就不多讲解。

4.1.3 ifdef 条件编译

单片机程序开发过程中，经常会遇到一种情况，当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。条件编译命令最常见的形式为：

```
#ifdef 标识符
程序段 1
#else
程序段 2
#endif
```

它的作用是：当标识符已经被定义过(一般是用#define 命令定义)，则对程序段 1 进行编译，否则编译程序段 2。 其中#else 部分也可以没有，即：

```
#ifdef
    程序段 1
#endif
```

这个条件编译在 MDK 里面是用得很多的，在 stm32f10x.h 这个头文件中经常会看到这样的语句：

```
#ifdef STM32F10X_HD
    大容量芯片需要的一些变量定义
#endif
```

而 STM32F10X_HD 则是我们通过#define 来定义的。条件编译也是 c 语言的基础知识，这里也就点到为止吧。

4.1.4 extern 变量申明

C 语言中 extern 可以置于变量或者函数前，以表示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。这里面要注意，对于 extern 申明变量可以多次，但定义只有一次。在我们的代码中你会看到看到这样的语句：

```
extern u16 USART_RX_STA;
```

这个语句是申明 USART_RX_STA 变量在其他文件中已经定义了，在这里要使用到。所以，你肯定可以找到在某个地方有变量定义的语句：

```
u16 USART_RX_STA;
```

的出现。下面通过一个例子说明一下使用方法。

在 Main.c 定义的全局变量 id，id 的初始化都是在 Main.c 里面进行的。

Main.c 文件

```
u8 id;//定义只允许一次
main()
{
    id=1;
    printf("d%",id);//id=1
    test();
    printf("d%",id);//id=2
}
```

但是我们希望在 test.c 的 changeId(void) 函数中使用变量 id，这个时候我们就需要在 test.c 里面去申明变量 id 是外部定义的了，因为如果不申明，变量 id 的作用域是到不了 test.c 文件中。看下面 test.c 中的代码：

```
extern u8 id;//申明变量 id 是在外部定义的，申明可以在很多个文件中进行
void test(void){
    id=2;
}
```

在 test.c 中申明变量 id 在外部定义，然后在 test.c 中就可以使用变量 id 了。

对于 extern 申明函数在外部定义的应用，这里我们就不多讲解了。

4.1.5 `typedef` 类型别名

`typedef` 用于为现有类型创建一个新的名字，或称为类型别名，用来简化变量的定义。
`typedef` 在 MDK 用得最多的就是定义结构体的类型别名和枚举类型了。

```
struct _GPIO
{
    __IO uint32_t CRL;
    __IO uint32_t CRH;
    ...
};
```

定义了一个结构体 GPIO，这样我们定义变量的方式为：

```
struct _GPIO GPIOA;//定义结构体变量 GPIOA
```

但是这样很繁琐，MDK 中有很多这样的结构体变量需要定义。这里我们可以为结体定义一个别名 `GPIO_TypeDef`，这样我们就可以在其他地方通过别名 `GPIO_TypeDef` 来定义结构体变量了。方法如下：

```
typedef struct
{
    __IO uint32_t CRL;
    __IO uint32_t CRH;
    ...
} GPIO_TypeDef;
```

`TypeDef` 为结构体定义一个别名 `GPIO_TypeDef`，这样我们可以通过 `GPIO_TypeDef` 来定义结构体变量：

```
GPIO_TypeDef GPIOA,_GPIOB;
```

这里的 `GPIO_TypeDef` 就跟 `struct _GPIO` 是等同的作用了。这样是不是方便很多？

4.1.6 结构体

经常很多用户提到，他们对结构体使用不是很熟悉，但是 MDK 中太多地方使用结构体以及结构体指针，这让他们一下子摸不着头脑，学习 STM32 的积极性大大降低，其实结构体并不是那么复杂，这里我们稍微提一下结构体的一些知识，还有一些知识我们会在下一节的“寄存器地址名称映射分析”中讲到一些。

声明结构体类型：

```
Struct 结构体名{
    成员列表;
    }变量名列表;
```

例如：

```
Struct U_TYPE {
    Int BaudRate
    Int WordLength;
}uart1,uart2;
```

在结构体申明的时候可以定义变量，也可以申明之后定义，方法是：

```
Struct 结构体名字 结构体变量列表；
```

例如：`struct U_TYPE usart1,usart2;`



结构体成员变量的引用方法是：

结构体变量名字.成员名

比如要引用 usart1 的成员 BaudRate，方法是：usart1.BaudRate；

结构体指针变量定义也是一样的，跟其他变量没有啥区别。

例如： struct U_TYPE *usart3; //定义结构体指针变量 usart1;

结构体指针成员变量引用方法是通过“->”符号实现，比如要访问 usart3 结构体指针指向的结构体的成员变量 BaudRate，方法是：

Usart3->BaudRate;

上面讲解了结构体和结构体指针的一些知识，其他的什么初始化这里就不多讲解了。讲到这里，有人会问，结构体到底有什么作用呢？为什么要使用结构体呢？下面我们将简单的通过一个实例回答一下这个问题。

在我们单片机程序开发过程中，经常会遇到要初始化一个外设比如串口，它的初始化状态是由几个属性来决定的，比如串口号，波特率，极性，以及模式。对于这种情况，在我们没有学习结构体的时候，我们一般的方法是：

void USART_Init(u8 usartx,u32 u32 BaudRate,u8 parity,u8 mode);

这种方式是有效的同时在一定场合是可取的。但是试想，如果有一天，我们希望往这个函数里面再传入一个参数，那么势必我们需要修改这个函数的定义，重新加入字长这个入口参数。于是我们的定义被修改为：

void USART_Init (u8 usartx,u32 BaudRate, u8 parity,u8 mode,u8 wordlength);

但是如果我们将这个函数的入口参数是随着开发不断的增多，那么是不是我们就要不断的修改函数的定义呢？这是不是给我们开发带来很多的麻烦？那又怎样解决这种情况呢？

这样如果我们使用到结构体就能解决这个问题了。我们可以在不改变入口参数的情况下，只需要改变结构体的成员变量，就可以达到上面改变入口参数的目的。

结构体就是将多个变量组合为一个有机的整体。上面的函数，BaudRate, wordlength, Parity, mode, wordlength 这些参数，他们对于串口而言，是一个有机整体，都是来设置串口参数的，所以我们可以将他们通过定义一个结构体来组合在一个。MDK 中是这样定义的：

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

于是，我们在初始化串口的时候入口参数就可以是 USART_InitTypeDef 类型的变量或者指针变量了，MDK 中是这样做的：

void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct);

这样，任何时候，我们只需要修改结构体成员变量，往结构体中间加入新的成员变量，而不需要修改函数定义就可以达到修改入口参数同样的目的了。这样的好处是不用修改任何函数定义就可以达到增加变量的目的。



理解了结构体在这个例子中间的作用吗？在以后的开发过程中，如果你的变量定义过多，如果某几个变量是用来描述某一个对象，你可以考虑将这些变量定义在结构体中，这样也许可以提高你的代码的可读性。

使用结构体组合参数，可以提高代码的可读性，不会觉得变量定义混乱。当然结构体的作用就远远不止这个了，同时，MDK 中用结构体来定义外设也不仅仅只是这个作用，这里我们只是举一个例子，通过最常用的场景，让大家理解结构体的一个作用而已。后面一节我们还会讲解结构体的一些其他知识。

4.2 STM32 系统架构

STM32 的系统架构比 51 单片机就要强大很多了。STM32 系统架构的知识可以在《STM32 中文参考手册 V10》的 P25~28 有讲解，这里我们也把这一部分知识抽取出来讲解，是为了大家在学习 STM32 之前对系统架构有一个初步的了解。这里的內容基本也是从中文参考手册中参考过来的，让大家能通过我们手册也了解到，免除了到处找资料的麻烦吧。如果需要详细了解 STM32 的系统架构，还需要在网上搜索其他资料学习学习。

我们这里所讲的 STM32 系统架构主要针对的 STM32F103 这些非互联型芯片。首先我们看看 STM32 的系统架构图：

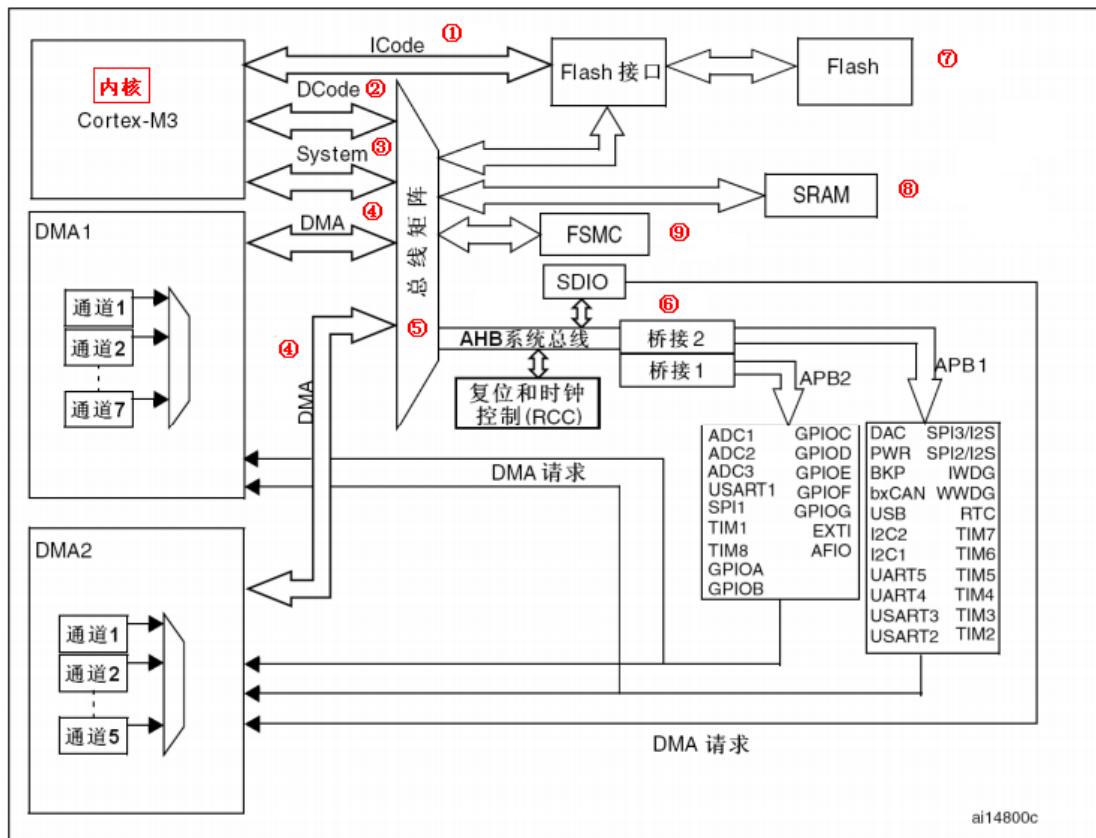


图 4.2.1STM32 系统架构图

STM32 主系统主要由四个驱动单元和四个被动单元构成。

四个驱动单元是：

内核 DCode 总线；



系统总线;

通用 DMA1;

通用 DMA2;

四被动单元是：

AHB 到 APB 的桥：连接所有的 APB 设备；

内部 FLASH 闪存；

内部 SRAM；

FSMC；

下面我们具体讲解一下图中几个总线的知识：

- ① ICode 总线：该总线将 M3 内核指令总线和闪存指令接口相连，指令的预取在该总线上面完成。
- ② DCode 总线：该总线将 M3 内核的 DCode 总线与闪存存储器的数据接口相连接，常量加载和调试访问在该总线上面完成。
- ③ 系统总线：该总线连接 M3 内核的系统总线到总线矩阵，总线矩阵协调内核和 DMA 间访问。
- ④ DMA 总线：该总线将 DMA 的 AHB 主控接口与总线矩阵相连，总线矩阵协调 CPU 的 DCode 和 DMA 到 SRAM, 闪存和外设的访问。
- ⑤ 总线矩阵：总线矩阵协调内核系统总线和 DMA 主控总线之间的访问仲裁，仲裁利用轮换算法。
- ⑥ AHB/APB 桥：这两个桥在 AHB 和 2 个 APB 总线间提供同步连接，APB1 操作速度限于 36MHz, APB2 操作速度全速。

对于系统架构的知识，在刚开始学习 STM32 的时候只需要一个大概的了解，大致知道是个什么情况即可。对于寻址之类的知识，这里就不做深入的讲解，中文参考手册都有很详细的讲解。

4.3 STM32 时钟系统

STM32 时钟系统的知识在《STM32 中文参考手册 V10》中的 P55~P73 有非常详细的讲解，网上关于时钟系统的讲解也基本都是参考的这里，讲不出啥特色，不过作为一个完整的参考手册，我们必然要提到时钟系统的知识。这些知识也不是什么原创，纯粹的是看网友发的帖子和手册来总结的，有一些直接是 copy 过来的，望大家谅解。

众所周知，时钟系统是 CPU 的脉搏，就像人的心跳一样。所以时钟系统的重要性就不言而喻了。STM32 的时钟系统比较复杂，不像简单的 51 单片机一个系统时钟就可以解决一切。于是有人要问，采用一个系统时钟不是很简单吗？为什么 STM32 要有多个时钟源呢？因为首先 STM32 本身非常复杂，外设非常的多，但是并不是所有外设都需要系统时钟这么高的频率，比如看门狗以及 RTC 只需要几十 k 的时钟即可。同一个电路，时钟越快功耗越大，同时抗电磁干扰能力也会越弱，所以对于较为复杂的 MCU 一般都是采取多时钟源的方法来解决这些问题。

首先让我们来看看 STM32 的时钟系统图吧：

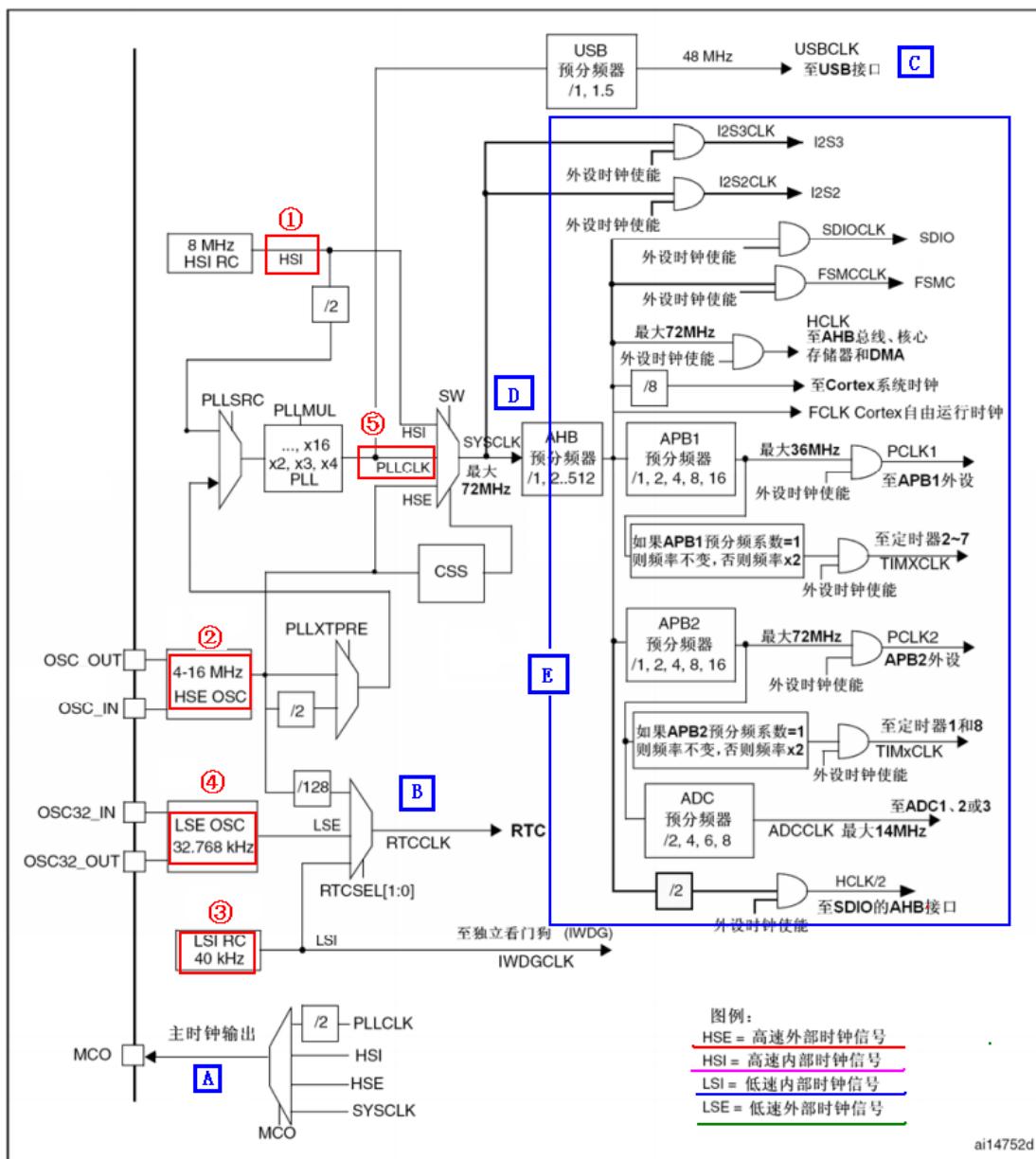


图 4.3.1 STM32 时钟系统图

在 STM32 中，有五个时钟源，为 HSI、HSE、LSI、LSE、PLL。从时钟频率来分可以分为高速时钟源和低速时钟源，在这 5 个中 HIS, HSE 以及 PLL 是高速时钟，LSI 和 LSE 是低速时钟。从来源可分为外部时钟源和内部时钟源，外部时钟源就是从外部通过接晶振的方式获取时钟源，其中 HSE 和 LSE 是外部时钟源，其他的是内部时钟源。下面我们看看 STM32 的 5 个时钟源，我们讲解顺序是按图中红圈标示的顺序：

- ①、HSI 是高速内部时钟，RC 振荡器，频率为 8MHz。
- ②、HSE 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz~16MHz。我们的开发板接的是 8M 的晶振。
- ③、LSI 是低速内部时钟，RC 振荡器，频率为 40kHz。独立看门狗的时钟源只能是 LSI，同时 LSI 还可以作为 RTC 的时钟源。
- ④、LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体。这个主要是 RTC 的时钟源。
- ⑤、PLL 为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HSE 或者 HSE/2。倍频可选择为



2~16 倍，但是其输出频率最大不得超过 72MHz。

上面我们简要概括了 STM32 的时钟源，那么这 5 个时钟源是怎么给各个外设以及系统提供时钟的呢？这里我们将一一讲解。我们还是从图的下方讲解起吧，因为下方比较简单。图中我们用 A~E 标示我们要讲解的地方。

- A. MCO 是 STM32 的一个时钟输出 IO(PA8)，它可以选择一个时钟信号输出，可以选择为 PLL 输出的 2 分频、HSI、HSE、或者系统时钟。这个时钟可以用来给外部其他系统提供时钟源。
- B. 这里是 RTC 时钟源，从图上可以看出，RTC 的时钟源可以选择 LSI，LSE，以及 HSE 的 128 分频。
- C. 从图中可以看出 C 处 USB 的时钟是来自 PLL 时钟源。STM32 中有一个全速功能的 USB 模块，其串行接口引擎需要一个频率为 48MHz 的时钟源。该时钟源只能从 PLL 输出端获取，可以选择为 1.5 分频或者 1 分频，也就是，当需要使用 USB 模块时，PLL 必须使能，并且时钟频率配置为 48MHz 或 72MHz。
- D. D 处就是 STM32 的系统时钟 SYSCLK，它是供 STM32 中绝大部分部件工作的时钟源。系统时钟可选择为 PLL 输出、HSI 或者 HSE。系统时钟最大频率为 72MHz，当然你也可以超频，不过一般情况为了系统稳定性是没有必要冒风险去超频的。
- E. 这里的 E 处是指其他所有外设了。从时钟图上可以看出，其他所有外设的时钟最终来源都是 SYSCLK。SYSCLK 通过 AHB 分频器分频后送给各模块使用。这些模块包括：
 - ①、AHB 总线、内核、内存和 DMA 使用的 HCLK 时钟。
 - ②、通过 8 分频后送给 Cortex 的系统定时器时钟，也就是 systick 了。
 - ③、直接送给 Cortex 的空闲运行时钟 FCLK。
 - ④、送给 APB1 分频器。APB1 分频器输出一路供 APB1 外设使用(PCLK1，最大频率 36MHz)，另一路送给定时器(Timer)2、3、4 倍频器使用。
 - ⑤、送给 APB2 分频器。APB2 分频器分频输出一路供 APB2 外设使用(PCLK2，最大频率 72MHz)，另一路送给定时器(Timer)1 倍频器使用。

其中需要理解的是 APB1 和 APB2 的区别，APB1 上面连接的是低速外设，包括电源接口、备份接口、CAN、USB、I2C1、I2C2、UART2、UART3 等等，APB2 上面连接的是高速外设包括 UART1、SPI1、Timer1、ADC1、ADC2、所有普通 IO 口(PA~PE)、第二功能 IO 口等。居宁老师的《稀里糊涂玩 STM32》资料里面教大家的记忆方法是 2>1，APB2 下面所挂的外设的时钟要比 APB1 的高。

在以上的时钟输出中，有很多是带使能控制的，例如 AHB 总线时钟、内核时钟、各种 APB1 外设、APB2 外设等等。当需要使用某模块时，记得一定要先使能对应的时钟。后面我们讲解实例的时候会讲解到时钟使能的方法。

STM32 时钟系统的配置除了初始化的时候在 system_stm32f10x.c 中的 SystemInit() 函数中外，其他的配置主要在 stm32f10x_rcc.c 文件中，里面有很多时钟设置函数，大家可以打开这个文件浏览一下，基本上看看函数的名称就知道这个函数的作用。在大家设置时钟的时候，一定要仔细参考 STM32 的时钟图，做到心中有数。这里需要指明一下，对于系统时钟，默认情况下是在 SystemInit 函数的 SetSysClock() 函数中间判断的，而设置是通过宏定义设置的。我们可以看看 SetSysClock() 函数体：

```
static void SetSysClock(void)
{
#ifndef SYSCLK_FREQ_HSE
```



```
SetSysClockToHSE();
#elif defined SYSCLK_FREQ_24MHz
    SetSysClockTo24();
#elif defined SYSCLK_FREQ_36MHz
    SetSysClockTo36();
#elif defined SYSCLK_FREQ_48MHz
    SetSysClockTo48();
#elif defined SYSCLK_FREQ_56MHz
    SetSysClockTo56();
#elif defined SYSCLK_FREQ_72MHz
    SetSysClockTo72();
#endif
}
```

这段代码非常简单，就是判断系统宏定义的时钟是多少，然后设置相应值。我们系统默认宏定义是 72MHz：

```
#define SYSCLK_FREQ_72MHz 72000000
```

如果你要设置为 36MHz，只需要注释掉上面代码，然后加入下面代码即可：

```
#define SYSCLK_FREQ_36MHz 36000000
```

同时还要注意的是，当我们设置好系统时钟后，可以通过变量 SystemCoreClock 获取系统时钟值，如果系统是 72M 时钟，那么 SystemCoreClock=72000000。这是在 system_stm32f10x.c 文件中设置的：

```
#ifdef SYSCLK_FREQ_HSE
    uint32_t SystemCoreClock      = SYSCLK_FREQ_HSE;
#elif defined SYSCLK_FREQ_36MHz
    uint32_t SystemCoreClock      = SYSCLK_FREQ_36MHz;
#elif defined SYSCLK_FREQ_48MHz
    uint32_t SystemCoreClock      = SYSCLK_FREQ_48MHz;
#elif defined SYSCLK_FREQ_56MHz
    uint32_t SystemCoreClock      = SYSCLK_FREQ_56MHz;
#elif defined SYSCLK_FREQ_72MHz
    uint32_t SystemCoreClock      = SYSCLK_FREQ_72MHz;
#else
    uint32_t SystemCoreClock      = HSI_VALUE;
#endif
```

这里总结一下 SystemInit() 函数中设置的系统时钟大小：

SYSCLK (系统时钟)	=72MHz
AHB 总线时钟(使用 SYSCLK)	=72MHz
APB1 总线时钟(PCLK1)	=36MHz
APB2 总线时钟(PCLK2)	=72MHz
PLL 时钟	=72MHz



4.4 端口复用和重映射

4.4.1 端口复用功能

STM32有很多的内置外设，这些外设的外部引脚都是与GPIO复用的。也就是说，一个GPIO如果可以复用为内置外设的功能引脚，那么当这个GPIO作为内置外设使用的时候，就叫做复用。这部分知识在《STM32中文参考手册V10》的P109, P116~P121有详细的讲解哪些GPIO管脚是可以复用为哪些内置外设的。这里我们就不一一讲解。

大家都知道，MCU都有串口，STM32有好几个串口。比如说STM32F103ZET6有5个串口，我们可以查手册知道，串口1的引脚对应的IO为PA9, PA10. PA9, PA10默认功能是GPIO，所以当PA9, PA10引脚作为串口1的TX, RX引脚使用的时候，那就是端口复用。

USART1_TX	PA9
USART1_RX	PA10

图 4.4.1.1 串口1复用管脚

复用端口初始化有几个步骤：

- 1) GPIO端口时钟使能。要使用到端口复用，当然要使能端口的时钟了。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
```
- 2) 复用的外设时钟使能。比如你要将端口PA9, PA10复用为串口，所以要使能串口时钟。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```
- 3) 端口模式配置。在IO复用位内设置外设功能引脚的时候，必须设置GPIO端口的模式，至于在复用功能下GPIO的模式是怎么对应的，这个可以查看手册《STM32中文参考手册V10》P110的表格“8.1.11 外设的GPIO配置”。这里我们拿Usart1举例：

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

图 4.4.1.2 串口复用GPIO配置

从表格中可以看出，我们要配置全双工的串口1，那么TX管脚需要配置为推挽复用输出，RX管脚配置为浮空输入或者带上拉输入。

```
//USART1_TX PA.9 复用推挽输出
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //PA.9
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
GPIO_Init(GPIOA, &GPIO_InitStructure);
//USART1_RX PA.10 浮空输入
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;//PA10
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;//浮空输入
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

上面代码的含义在我们的第一个实验学完之后大家自然会了解，这里只是做个概括。所以，我们在使用复用功能的是时候，最少要使能2个时钟：

- 1) GPIO时钟使能



2) 复用的外设时钟使能
同时要初始化 GPIO 以及复用外设功能

4.4.2 端口重映射

为了使不同器件封装的外设 IO 功能数量达到最优，可以把一些复用功能重新映射到其他一些引脚上。STM32 中有很多内置外设的输入输出引脚都具有重映射(remap)的功能。我们知道每个内置外设都有若干个输入输出引脚，一般这些引脚的输出端口都是固定不变的，为了让设计工程师可以更好地安排引脚的走向和功能，在 STM32 中引入了外设引脚重映射的概念，即一个外设的引脚除了具有默认的端口外，还可以通过设置重映射寄存器的方式，把这个外设的引脚映射到其它的端口。

简单的讲就是把管脚的外设功能映射到另一个管脚，但不是可以随便映射的，具体对应关系《STM32 中文参考手册 V10》的 P116 页“8.3 复用功能和调试配置”有讲解。这里我们同样拿串口 1 为例来讲解。

表47 USART1重映像

复用功能	USART1_REMAP = 0	USART1_REMAP = 1
USART1_TX	PA9	PB6
USART1_RX	PA10	PB7

图 4.4.2.1 串口重映射管脚表

上图是截取的中文参考手册中的重映射表，从表中可以看出，默认情况下，串口 1 复用的时候的引脚位 PA9, PA10，同时我们可以将 TX 和 RX 重新映射到管脚 PB6 和 PB7 上面去。

所以重映射我们同样要使能复用功能的时候讲解的 2 个时钟外，还要使能 AFIO 功能时钟，然后要调用重映射函数。详细步骤为：

1) 使能 GPIOB 时钟：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
```

2) 使能串口 1 时钟：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```

3) 使能 AFIO 时钟：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
```

4) 开启重映射：

```
GPIO_PinRemapConfig(GPIO_Remap_USART1, ENABLE);
```

这样就将串口的 TX 和 RX 重映射到管脚 PB6 和 PB7 上面了。至于有哪些功能可以重映射，大家除了查看中文参考手册之外，还可以从 GPIO_PinRemapConfig 函数入手查看第一个入口参数的取值范围可以得知。在 stm32f10x_gpio.h 文件中定义了取值范围为下面宏定义的标识符，这里我们贴一小部分：

```
#define GPIO_Remap_SPI1      ((uint32_t)0x00000001)
#define GPIO_Remap_I2C1       ((uint32_t)0x00000002)
#define GPIO_Remap_USART1     ((uint32_t)0x00000004)
#define GPIO_Remap_USART2     ((uint32_t)0x00000008)
#define GPIO_PartialRemap_USART3 ((uint32_t)0x00140010)
```



```
#define GPIO_FullRemap_USART3 ((uint32_t)0x00140030)
```

从上面可以看出，USART1 只有一种重映射，而对于 USART3，存在部分重映射和完全重映射。所谓部分重映射就是部分管脚和默认的是一样的，而部分管脚是重新映射到其他管脚。而完全重映射就是所有管脚都重新映射到其他管脚。看看手册中的 USART3 重映射表：

表45 USART3重映像

复用功能	USART3_REMAP[1:0] = 00 (没有重映像)	USART3_REMAP[1:0] = 01 (部分重映像) ⁽¹⁾	USART3_REMAP[1:0] = 11 (完全重映像) ⁽²⁾
USART3_TX	PB10	PC10	PD8
USART3_RX	PB11	PC11	PD9
USART3_CK	PB12	PC12	PD10
USART3_CTS	PB13		PD11
USART3_RTS	PB14		PD12

图 4.4.2.2 USART3 重映射管脚对应表

部分重映射就是 PB10, PB11, PB12 重映射到 PC10, PC11, PC12 上。而 PB13 和 PB14 和没有重映射情况是一样的，都是 USART3_CTS 和 USART3_RTS 对应管脚。完全重映射就是将这两个脚重新映射到 PD11 和 PD12 上去。我们要使用 USART3 的部分重映射，我们调用函数方法为：

```
GPIO_PinRemapConfig(GPIO_PartialRemap_USART3, ENABLE);
```

这些知识我们后面在使用的过程中间还会讲解，这里只是对重映射概念做个简要的描述。

4.5 STM32 NVIC 中断优先级管理

这节我们将对 STM32 的重要只是中断管理做个详细的介绍。

CM3 内核支持 256 个中断，其中包含了 16 个内核中断和 240 个外部中断，并且具有 256 级的可编程中断设置。但 STM32 并没有使用 CM3 内核的全部东西，而是只用了它的一部分。STM32 有 84 个中断，包括 16 个内核中断和 68 个可屏蔽中断，具有 16 级可编程的中断优先级。而我们常用的就是这 68 个可屏蔽中断，但是 STM32 的 68 个可屏蔽中断，在 STM32F103 系列上面，又只有 60 个（在 107 系列才有 68 个）。因为我们开发板选择的芯片是 STM32F103 系列的所以我们就只针对 STM32F103 系列这 60 个可屏蔽中断进行介绍。

在 MDK 内，与 NVIC 相关的寄存器，MDK 为其定义了如下的结构体：

```
typedef struct
{
    vu32 ISER[2];
    u32 RESERVED0[30];
    vu32 ICER[2];
    u32 RSERVED1[30];
    vu32 ISPR[2];
    u32 RESERVED2[30];
    vu32 ICPR[2];
    u32 RESERVED3[30];
    vu32 IABR[2];
    u32 RESERVED4[62];
    vu32 IPR[15];
}
```



} NVIC_TypeDef;

STM32 的中断在这些寄存器的控制下有序的执行的。只有了解这些中断寄存器，才能了解 STM32 的中断。下面简要介绍这几个寄存器：

ISER[2]: ISER 全称是：Interrupt Set-Enable Registers，这是一个中断使能寄存器组。上面说了 STM32F103 的可屏蔽中断只有 60 个，这里用了 2 个 32 位的寄存器，总共可以表示 64 个中断。而 STM32F103 只用了其中的前 60 位。ISER[0]的 bit0~bit31 分别对应中断 0~31。ISER[1] 的 bit0~27 对应中断 32~59；这样总共 60 个中断就分别对应上了。你要使能某个中断，必须设置相应的 ISER 位为 1，使该中断被使能(这里仅仅是使能，还要配合中断分组、屏蔽、IO 口映射等设置才算是一个完整的中断设置)。具体每一位对应哪个中断，请参考 stm32f10x_nvic.h 里面的第 36 行处。

ICER[2]: 全称是：Interrupt Clear-Enable Registers，是一个中断除能寄存器组。该寄存器组与 ISER 的作用恰好相反，是用来清除某个中断的使能的。其对应位的功能，也和 ICER 一样。这里要专门设置一个 ICER 来清除中断位，而不是向 ISER 写 0 来清除，是因为 NVIC 的这些寄存器都是写 1 有效的，写 0 是无效的。具体为什么这么设计，请看《CM3 权威指南》第 125 页，NVIC 概览一章。

ISPR[2]: 全称是：Interrupt Set-Pending Registers，是一个中断挂起控制寄存器组。每个位对应的中断和 ISER 是一样的。通过置 1，可以将正在进行的中断挂起，而执行同级或更高级别的中断。写 0 是无效的。

ICPR[2]: 全称是：Interrupt Clear-Pending Registers，是一个中断解挂控制寄存器组。其作用与 ISPR 相反，对应位也和 ISER 是一样的。通过设置 1，可以将挂起的中断接挂。写 0 无效。

IABR[2]: 全称是：Interrupt Active Bit Registers，是一个中断激活标志位寄存器组。这是一个只读寄存器，通过它可以知道当前在执行的中断是哪一个。在中断执行完了由硬件自动清零。对应位所代表的中断和 ISER 一样，如果为 1，则表示该位所对应的中断正在被执行。

IPR[15]: 全称是：Interrupt Priority Registers，是一个中断优先级控制的寄存器组。这个寄存器组相当重要！STM32 的中断分组与这个寄存器组密切相关。因为 STM32 的中断多达 60 多个，所以 STM32 采用中断分组的办法来确定中断的优先级。IPR 寄存器组由 15 个 32bit 的寄存器组成，每个可屏蔽中断占用 8bit，这样总共可以表示 $15 \times 4 = 60$ 个可屏蔽中断。刚好和 STM32 的可屏蔽中断数相等。IPR[0]的[31~24], [23~16], [15~8], [7~0]分别对应中断 3~0，依次类推，总共对应 60 个外部中断。而每个可屏蔽中断占用的 8bit 并没有全部使用，而是 只用了高 4 位。这 4 位，又分为抢占优先级和子优先级。抢占优先级在前，子优先级在后。而这两个优先级各占几个位又要根据 SCB->AIRCR 中的中断分组设置来决定。

这里简单介绍一下 STM32 的中断分组：STM32 将中断分为 5 个组，组 0~4。该分组的设置是由 SCB->AIRCR 寄存器的 bit10~8 来定义的。具体的分配关系如表 4.5.1 所示：

组	AIRCR[10: 8]	bit[7: 4]分配情况	分配结果
0	111	0: 4	0 位抢占优先级, 4 位响应优先级
1	110	1: 3	1 位抢占优先级, 3 位响应优先级
2	101	2: 2	2 位抢占优先级, 2 位响应优先级
3	100	3: 1	3 位抢占优先级, 1 位响应优先级
4	011	4: 0	4 位抢占优先级, 0 位响应优先级

表 4.5.1 AIRCR 中断分组设置表

通过这个表，我们就可以清楚的看到组 0~4 对应的配置关系，例如组设置为 3，那么此时所有的 60 个中断，每个中断的中断优先寄存器的高四位中的最高 3 位是抢占优先级，低 1 位是



响应优先级。每个中断，你可以设置抢占优先级为 0~7，响应优先级为 1 或 0。抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高。

这里需要注意两点：第一，如果两个中断的抢占优先级和响应优先级都是一样的话，则看哪个中断先发生就先执行；第二，高优先级的抢占优先级是可以打断正在进行的低抢占优先级中断的。而抢占优先级相同的中断，高优先级的响应优先级不可以打断低响应优先级的中断。

结合实例说明一下：假定设置中断优先级组为 2，然后设置中断 3(RTC 中断)的抢占优先级为 2，响应优先级为 1。中断 6(外部中断 0)的抢占优先级为 3，响应优先级为 0。中断 7(外部中断 1)的抢占优先级为 2，响应优先级为 0。那么这 3 个中断的优先级顺序为：中断 7>中断 3>中断 6。

上面例子中的中断 3 和中断 7 都可以打断中断 6 的中断。而中断 7 和中断 3 却不可以相互打断！

通过以上介绍，我们熟悉了 STM32 中断设置的大致过程。接下来我们介绍如何使用库函数实现以上中断分组设置以及中断优先级管理，使得我们以后的中断设置简单化。NVIC 中断管理函数主要在 misc.c 文件里面。

首先要讲解的是中断优先级分组函数 NVIC_PriorityGroupConfig，其函数申明如下：

```
void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup);
```

这个函数的作用是对中断的优先级进行分组，这个函数在系统中只能被调用一次，一旦分组确定就最好不要更改。这个函数我们可以找到其实现：

```
void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)
{
    assert_param(IS_NVIC_PRIORITY_GROUP(NVIC_PriorityGroup));
    SCB->AIRCR = AIRCR_VECTKEY_MASK | NVIC_PriorityGroup;
}
```

从函数体可以看出，这个函数唯一目的就是通过设置 SCB->AIRCR 寄存器来设置中断优先级分组，这在前面寄存器讲解的过程中已经讲到。而其入口参数通过双击选中函数体里面的“IS_NVIC_PRIORITY_GROUP”然后右键“Go to defition of …”可以查看到为：

```
#define IS_NVIC_PRIORITY_GROUP(GROUP)
(((GROUP) == NVIC_PriorityGroup_0) ||
 ((GROUP) == NVIC_PriorityGroup_1) ||
 ((GROUP) == NVIC_PriorityGroup_2) ||
 ((GROUP) == NVIC_PriorityGroup_3) ||
 ((GROUP) == NVIC_PriorityGroup_4))
```

这也是我们上面表 4.5.1 讲解的，分组范围为 0~4。比如我们设置整个系统的中断优先级分组值为 2，那么方法是：

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
```

这样就确定了一共为“2 位抢占优先级，2 位响应优先级”。

设置好了系统中断分组，那么对于每个中断我们又怎么确定他的抢占优先级和响应优先级呢？下面我们讲解一个重要的函数为中断初始化函数 NVIC_Init，其函数申明为：

```
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)
```

其中 NVIC_InitTypeDef 是一个结构体，我们可以看看结构体的成员变量：

```
typedef struct
{
    uint8_t NVIC_IRQChannel;
```



```

    uint8_t NVIC_IRQChannelPreemptionPriority;
    uint8_t NVIC_IRQChannelSubPriority;
    FunctionalState NVIC_IRQChannelCmd;
} NVIC_InitTypeDef;

```

NVIC_InitTypeDef 结构体中间有三个成员变量，这三个成员变量的作用是：

NVIC_IRQChannel：定义初始化的是哪个中断，这个我们可以在 stm32f10x.h 中找到每个中断对应的名字。例如 USART1_IRQn。

NVIC_IRQChannelPreemptionPriority：定义这个中断的抢占优先级别。

NVIC_IRQChannelSubPriority：定义这个中断的子优先级别。

NVIC_IRQChannelCmd：该中断是否使能。

比如我们要使能串口 1 的中断，同时设置抢占优先级为 1，子优先级位 2，初始化的方法是：

```

USART_InitTypeDef USART_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;//串口 1 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=1 ;// 抢占优先级为 1
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;// 子优先级位 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;           //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure); //根据上面指定的参数初始化 NVIC 寄存器

```

这里我们讲解了中断的分组的概念以及设定优先级值的方法，至于每种优先级还有一些关于清除中断，查看中断状态，这在后面我们讲解每个中断的时候会详细讲解到。最后我们总结一下中断优先级设置的步骤：

1. 系统运行开始的时候设置中断分组。确定组号，也就是确定抢占优先级和子优先级的分配位数。调用函数为 NVIC_PriorityGroupConfig();
2. 设置所用到的中断的中断优先级别。对每个中断调用函数为 NVIC_Init();

4.6 MDK 中寄存器地址名称映射分析

之所以要讲解这部分知识，是因为经常会遇到客户提到不明白 MDK 中那些结构体是怎么与寄存器地址对应起来的。这里我们就做一个简要的分析吧。

首先我们看看 51 中是怎么做的。51 单片机开发中经常会引用一个 reg51.h 的头文件，下面看看他是怎么把名字和寄存器联系起来的：

```
sfr P0 =0x80;
```

sfr 也是一种扩充数据类型，点用一个内存单元，值域为 0~255。利用它可以访问 51 单片机内部的所有特殊功能寄存器。如用 sfr P1 =0x90 这一句定义 P1 为 P1 端口在片内的寄存器。然后我们往地址为 0x80 的寄存器设值的方法是：P0=value；

那么在 STM32 中，是否也可以这样做呢？？答案是肯定的。肯定也可以通过同样的方式来做，但是 STM32 因为寄存器太多太多，如果一一以这样的方式列出来，那要好大的篇幅，既不方便开发，也显得太杂乱无序的感觉。所以 MDK 采用的方式是通过结构体来将寄存器组织在一起。下面我们就讲解 MDK 是怎么把结构体和地址对应起来的，为什么我们修改结构体成员变量的值就可以达到操作对应寄存器的值。这些事情都是在 stm32f10x.h 文件中完成的。我们通过 GPIOA 的几个寄存器的地址来讲解吧。



首先我们可以查看《STM32 中文参考手册 V10》中的寄存器地址映射表(P159):

表52 GPIO寄存器地址映像和复位值

偏移	寄存器	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000h	GPIOx_CRL	CNF7 [1:0]	MODE7 [1:0]	CNF6 [1:0]	MODE6 [1:0]	CNF5 [1:0]	MODE5 [1:0]	CNF4 [1:0]	MODE4 [1:0]	CNF3 [1:0]	MODE3 [1:0]	CNF2 [1:0]	MODE2 [1:0]	CNF1 [1:0]	MODE1 [1:0]	CNF0 [1:0]	MODE0 [1:0]																
		0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1			
004h	GPIOx_CRH	CNF15 [1:0]	MODE15 [1:0]	CNF14 [1:0]	MODE14 [1:0]	CNF13 [1:0]	MODE13 [1:0]	CNF12 [1:0]	MODE12 [1:0]	CNF11 [1:0]	MODE11 [1:0]	CNF10 [1:0]	MODE10 [1:0]	CNF9 [1:0]	MODE9 [1:0]	CNF8 [1:0]	MODE8 [1:0]																
		0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1			
008h	GPIOx_IDR	保留															IDR[15:0]																
		复位值															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																
00Ch	GPIOx_ODR	保留															ODR[15:0]																
		复位值															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																
010h	GPIOx_BSRR	BR[15:0]															BSR[15:0]																
		复位值															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																
014h	GPIOx_BRR	保留															BR[15:0]																
		复位值															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																
018h	GPIOx_LCKR	保留															LCKK 0	LCK[15:0]															
		复位值																															

图 4.6.1 GPIO 寄存器地址映像

从这个表我们可以看出，GPIOA 的 7 个寄存器都是 32 位的，所以每个寄存器占有 4 个地址，一共占用 28 个地址，地址偏移范围为 (000h~01Bh)。这个地址偏移是相对 GPIOA 的基址而言的。GPIOA 的基址是怎么算出来的呢？因为 GPIO 都是挂载在 APB2 总线之上，所以它的基址是由 APB2 总线的基址+GPIOA 在 APB2 总线上的偏移地址决定的。同理依次类推，我们便可以算出 GPIOA 基址了。这里设计到总线的一些知识，我们在后面会讲到。下面我们打开 stm32f10x.h 定位到 GPIO_TypeDef 定义处：

```
typedef struct
{
    __IO uint32_t CRL;
    __IO uint32_t CRH;
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint32_t BSRR;
    __IO uint32_t BRR;
    __IO uint32_t LCKR;
} GPIO_TypeDef;
```

然后定位到：

```
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
```

可以看出，GPIOA 是将 GPIOA_BASE 强制转换为 GPIO_TypeDef 指针，这句话的意思是，GPIOA 指向地址 GPIOA_BASE，GPIOA_BASE 存放的数据类型为 GPIO_TypeDef。然后双击“GPIOA_BASE”选中之后右键选中“Go to definition of”，便可一查看 GPIOA_BASE 的宏定义：

```
#define GPIOA_BASE (APB2PERIPH_BASE + 0x0800)
```

依次类推，可以找到最顶层：

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

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



所以我们便可以算出 GPIOA 的基址位：

$$\text{GPIOA_BASE} = 0x40000000 + 0x10000 + 0x0800 = 0x40010800$$

下面我们再跟《STM32 中文参考手册 V10》比较一下看看 GPIOA 的基址是不是 0x40010800。截图 P28 存储器映射表我们可以看到，GPIOA 的起始地址也就是基址确实是 0x40010800：

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

图 4.6.2 GPIO 存储器地址映射表

同样的道理，我们可以推算出其他外设的基址。

上面我们已经知道 GPIOA 的基址，那么那些 GPIOA 的 7 个寄存器的地址又是怎么算出来的呢？在上面我们讲过 GPIOA 的各个寄存器对于 GPIOA 基址的偏移地址，所以我们自然可以算出来每个寄存器的地址。

GPIOA 的寄存器的地址=GPIOA 基地址+寄存器相对 GPIOA 基地址的偏移值
这个偏移值在上面的寄存器地址映像表中可以查到。

那么在结构体里面这些寄存器又是怎么与地址一一对应的呢？这里就涉及到结构体的一个特征，那就是结构体存储的成员他们的地址是连续的。上面讲到 GPIOA 是指向 GPIO_TypeDef 类型的指针，又由于 GPIO_TypeDef 是结构体，所以自然而然我们就可以算出 GPIOA 指向的结构体成员变量对应地址了。

寄存器	偏移地址	实际地址=基地址+偏移地址
GPIOA->CRL	0x00	0x40010800+0x00
GPIOA->CRH;	0x04	0x40010800+0x04
GPIOA->IDR;	0x08	0x40010800+0x08
GPIOA->ODR	0x0c	0x40010800+0x0c
GPIOA->BSRR	0x10	0x40010800+0x10
GPIOA->BRR	0x14	0x40010800+0x14
GPIOA->LCKR	0x18	0x40010800+0x18

表 4.6.3 GPIOA 各寄存器实际地址表

我们可以把 GPIO_TypeDef 的定义中的成员变量的顺序和 GPIOx 寄存器地址映像对比可以发现，他们的顺序是一致的，如果不一致，就会导致地址混乱了。

这就是为什么固件库里面：GPIOA->BRR=value;就是设置地址为 0x40010800 +0x014(BRR 偏移量)=0x40010814 的寄存器 BRR 的值了。它和 51 里面 P0=value 是设置地址为 0x80 的 P0 寄存器的值是一样的道理。

看到这里你是否会学起来踏实一点呢？STM32 使用的方式虽然跟 51 单片机不一样，但是原理都是一致的。

4.7 MDK 固件库快速组织代码技巧

这一节主要讲解在使用 MDK 固件库开发的时候的一些小技巧，仅供初学者参考。这节的



知识大家可以在学习第一个跑马灯实验的时候参考一下，对初学者应该很有帮助。我们就用最简单的 GPIO 初始化函数为例。

现在我们要初始化某个 GPIO 端口，我们要怎样快速操作呢？在头文件 `stm32f10x_gpio.h` 头文件中，定义 GPIO 初始化函数为：

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
```

现在我们想写初始化函数，那么我们在不参考其他代码的前提下，怎么组织代码呢？

首先，我们可以看出，函数的入口参数是 `GPIO_TypeDef` 类型指针和 `GPIO_InitTypeDef` 类型指针，因为 `GPIO_TypeDef` 入口参数比较简单，所以我们通过第二个入口参数 `GPIO_InitTypeDef` 类型指针来讲解。双击 `GPIO_InitTypeDef` 后右键选择“Go to definition...”，如下图 4.7.1：

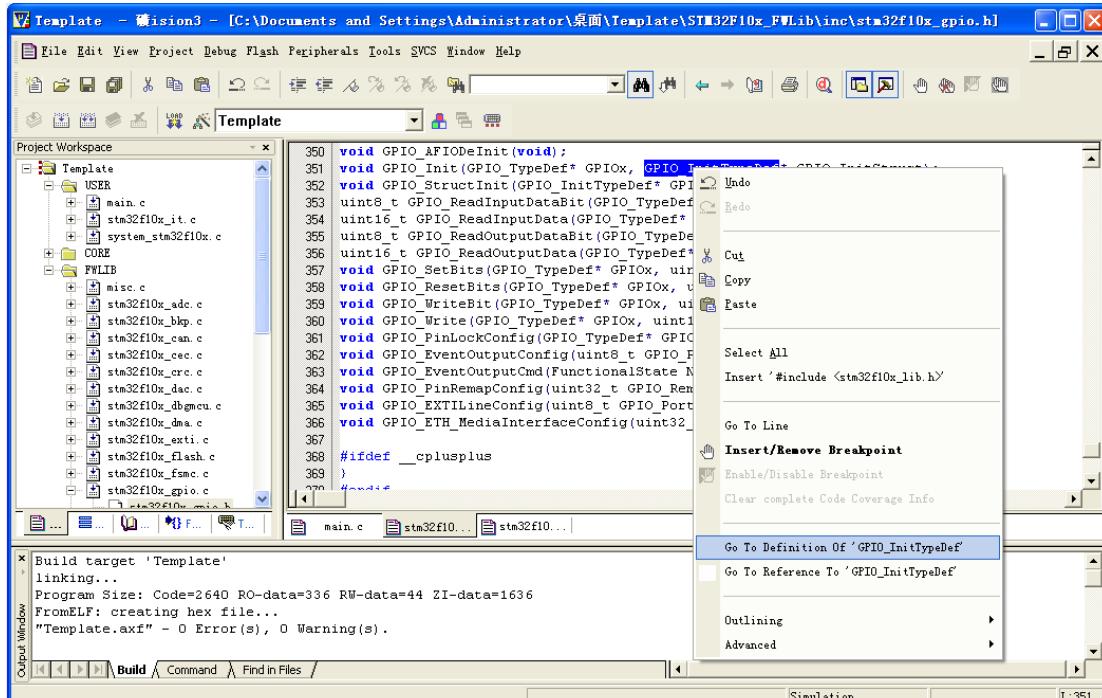


图 4.7.1 查看类型定义方法

于是定位到 `stm32f10x_gpio.h` 中 `GPIO_InitTypeDef` 的定义处：

```
typedef struct
{
    uint16_t GPIO_Pin;
    GPIO_Speed_TypeDef GPIO_Speed;
    GPIO_Mode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

可以看到这个结构体有 3 个成员变量，这也告诉我们一个信息，一个 GPIO 口的状态是由速度 (Speed) 和模式 (Mode) 来决定的。我们首先要定义一个结构体变量，下面我们定义：

```
GPIO_InitTypeDef GPIO_InitStructure;
```

接着我们要初始化结构体变量 `GPIO_InitStructure`。首先我们要初始化成员变量 `GPIO_Pin`，这个时候我们就有点迷糊了，这个变量到底可以设置哪些值呢？这些值的范围有什么规定吗？

这里我们就要找到 `GPIO_Init()` 函数的实现处，同样，双击 `GPIO_Init`，右键点击“Go to definition of ...”，这样光标定位到 `stm32f10x_gpio.c` 文件中的 `GPIO_Init` 函数体开始处，我们可以看到在函数的开始处有如下几行：

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
```



```
{
    .....
    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
    assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
    .....
    assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
    .....
}
```

顾名思义，assert_param 函数式对入口参数的有效性进行判断，所以我们可以从这个函数入手，确定我们的入口参数的范围。第一行是对第一个参数 GPIOx 进行有效性判断，双击 “IS_GPIO_ALL_PERIPH” 右键点击 “go to defition of...” 定位到了下面的定义：

```
#define IS_GPIO_ALL_PERIPH(PERIPH) (((PERIPH) == GPIOA) || \
    ((PERIPH) == GPIOB) || \
    ((PERIPH) == GPIOC) || \
    ((PERIPH) == GPIOD) || \
    ((PERIPH) == GPIOE) || \
    ((PERIPH) == GPIOF) || \
    ((PERIPH) == GPIOG))
```

很明显可以看出，GPIOx 的取值规定只允许是 GPIOA~GPIOG。

同样的办法，我们双击 “IS_GPIO_MODE” 右键点击 “go to defition of...” ,定位到下面的定义：

```
typedef enum
{
    GPIO_Mode_AIN = 0x0,
    GPIO_Mode_IN_FLOATING = 0x04,
    GPIO_Mode_IPD = 0x28,
    GPIO_Mode_IPU = 0x48,
    GPIO_Mode_Out_OD = 0x14,
    GPIO_Mode_Out_PP = 0x10,
    GPIO_Mode_AF_OD = 0x1C,
    GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;

#define IS_GPIO_MODE(MODE) (((MODE) == GPIO_Mode_AIN) || \
    ((MODE) == GPIO_Mode_IN_FLOATING) || \
    ((MODE) == GPIO_Mode_IPD) || \
    ((MODE) == GPIO_Mode_IPU) || \
    ((MODE) == GPIO_Mode_Out_OD) || \
    ((MODE) == GPIO_Mode_Out_PP) || \
    ((MODE) == GPIO_Mode_AF_OD) || \
    ((MODE) == GPIO_Mode_AF_PP))
```

所以 GPIO_InitStruct->GPIO_Mode 成员的取值范围只能是上面定义的 8 种。这 8 中模式是通过

一个枚举类型组织在一起的。

同样的方法可以找出 GPIO_Speed 的参数限制:

```
typedef enum
{
    GPIO_Speed_10MHz = 1,
    GPIO_Speed_2MHz,
    GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;

#define IS_GPIO_SPEED(SPEED) (((SPEED) == GPIO_Speed_10MHz) || \
                           ((SPEED) == GPIO_Speed_2MHz) || \
                           ((SPEED) == GPIO_Speed_50MHz))
```

同样的方法我们双击 “IS_GPIO_PIN” 右键点击 “go to defition of...” ,定位到下面的定义:

```
#define IS_GPIO_PIN(PIN) (((PIN) & (uint16_t)0x00) == 0x00) && ((PIN) != (uint16_t)0x00)
```

可以看出, GPIO_Pin 成员变量的取值范围为 0x0000 到 0xffff, 那么是不是我们写代码初始化就是直接给一个 16 位的数字呢? 这也是可以的, 但是大多数情况下, MDK 不会让你直接在入口参数处设置一个简单的数字, 因为这样代码的可读性太差, MDK 会将这些数字的意思通过宏定义定义出来, 这样可读性大大增强。我们可以看到在 IS_GPIO_PIN(PIN) 宏定义的上面还有数行宏定义:

```
#define GPIO_Pin_0          ((uint16_t)0x0001) /*!< Pin 0 selected */
#define GPIO_Pin_1          ((uint16_t)0x0002) /*!< Pin 1 selected */
#define GPIO_Pin_2          ((uint16_t)0x0004) /*!< Pin 2 selected */
#define GPIO_Pin_3          ((uint16_t)0x0008) /*!< Pin 3 selected */
#define GPIO_Pin_4          ((uint16_t)0x0010) /*!< Pin 4 selected */
.....
#define GPIO_Pin_14         ((uint16_t)0x4000) /*!< Pin 14 selected */
#define GPIO_Pin_15         ((uint16_t)0x8000) /*!< Pin 15 selected */
#define GPIO_Pin_All        ((uint16_t)0xFFFF) /*!< All pins selected */

#define IS_GPIO_PIN(PIN) (((PIN) & (uint16_t)0x00) == 0x00) && ((PIN) != (uint16_t)0x00))
```

这些宏定义 GPIO_Pin_0~GPIO_Pin_All 就是 MDK 事先定义好的, 我们写代码的时候初始化 GPIO_Pin 的时候入口参数可以是这些宏定义。对于这种情况, MDK 一般把取值范围的宏定义放在判断有效性语句的上方, 这样是为了方便大家查找。

讲到这里, 我们基本对 GPIO_Init 的入口参数有比较详细的了解了。于是我们可以组织起来下面的代码:

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

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

接着又有一个问题会被提出来，这个初始化函数一次只能初始化一个 IO 口吗？我要同时初始化很多个 IO 口，是不是要复制很多次这样的初始化代码呢？

这里又有一个小技巧了。从上面的 GPIO_Pin_x 的宏定义我们可以看出，这些值是 0,1,2,4 这样的数字，所以每个 IO 口选定都是对应着一个位，16 位的数据一共对应 16 个 IO 口。这个位为 0 那么这个对应的 IO 口不选定，这个位为 1 对应的 IO 口选定。如果多个 IO 口，他们都是对应同一个 GPIOx，那么我们可以通过|（或）的方式同时初始化多个 IO 口。这样操作的前提是，他们的 Mode 和 Speed 参数相同，因为 Mode 和 Speed 参数并不能一次定义多种。所以初始化多个 IO 口的方式可以是如下：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7; //指定端口
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;           //端口模式：推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;          //速度
GPIO_Init(GPIOB, &GPIO_InitStructure);                   //初始化
```

对于那些参数可以通过|(或)的方式连接，这既有章可循，同时也靠大家在开发过程中不断积累。

有客户经常问到，我每次使能时钟的时候都要去查看时钟树看那些外设是挂载在那个总线之下的，这好麻烦。学到这里我相信大家就可以很快速的解决这个问题了。

在 stm32f10x.h 文件里面我们可以看到如下的宏定义：

```
#define RCC_APB2Periph_GPIOA ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB ((uint32_t)0x00000008)
#define RCC_APB2Periph_GPIOC ((uint32_t)0x00000010)

#define RCC_APB1Periph_TIM2 ((uint32_t)0x00000001)
#define RCC_APB1Periph_TIM3 ((uint32_t)0x00000002)
#define RCC_APB1Periph_TIM4 ((uint32_t)0x00000004)

#define RCC_AHBPeriph_DMA1 ((uint32_t)0x00000001)
#define RCC_AHBPeriph_DMA2 ((uint32_t)0x00000002)
```

从上图可以很明显的看出 GPIOA~GPIOC 是挂载在 APB2 下面，TIM2~TIM4 是挂载在 APB1 下面，DMA 是挂载在 AHB 下面。所以在使能 DMA 的时候记住要调用的是 RCC_AHBPeriphClock() 函数使能，在使能 GPIO 的时候调用的是 RCC_APB2PeriphResetCmd() 函数使能。

大家会觉得上面讲解有点麻烦，每次要去查找 assert_param() 这个函数去寻找，那么有没有更好的办法呢？大家可以打开 GPIO_InitTypeDef 结构体定义：

```
typedef struct
{
    uint16_t GPIO_Pin;           /*!< Specifies the GPIO pins to be configured.
                                    This parameter can be any value of
                                    @ref GPIO_pins_define */
    GPIOSpeed_TypeDef GPIO_Speed; /*!< Specifies the speed for the selected pins.
                                    This parameter can be a value of
```



```
    @ref GPIOSpeed_TypeDef */  
    GPIOMode_TypeDef  GPIO_Mode; /*!< Specifies the operating mode for the selected  
                                 pins.This parameter can be a value of  
    @ref GPIOMode_TypeDef */  
}  
}GPIO_InitTypeDef;
```

从上图的结构体成员后面的注释我们可以看出 GPIO_Mode 的意思是

“Specifies the operating mode for the selected pins.This parameter can be a value of @ref GPIOMode_TypeDef”。从这段注释可以看出 GPIO_Mode 的取值为 GPIOMode_TypeDef 枚举类型的枚举值，大家同样可以用之前讲解的方法右键双击“GPIOMode_TypeDef”选择“Go to definition of ...”即可查看其取值范围。如果要确定详细的信息呢我们就得去查看手册了。对于去查看手册的哪个地方，你可以在函数 GPIO_Init()的函数体中搜索 GPIO_Mode 关键字，然后查看库函数设置 GPIO_Mode 是设置的哪个寄存器的那个位，然后去中文参考手册查看该寄存器相应位的定义以及前后文的描述。

这一节我们就讲解到这里，希望能对大家的开发有帮助。



第五章 SYSTEM 文件夹介绍

前面章节，我们介绍了如何在 MDK3.80A 下建立 STM32 工程，在这个新建的工程之中，我们用到了一个 SYSTEM 文件夹里面的代码，此文件夹里面的代码由 ALIENTEK 提供，包含了几乎每个实验都可能用到的延时函数，位带操作，串口打印代码等。这里我们组织在 SYSTEM 文件夹下面，目的也就是让这些常用的代码能随用随调。

SYSTEM 文件夹下包含了 delay、sys、usart 等三个文件夹。分别包含了 delay.c、sys.c、usart.c 及其头文件 delay.h,sys.h,usart.h。

本章，我们将向大家介绍这些代码，通过这章的学习，大家将了解到这些代码的由来，也希望大家可以灵活使用 SYSTEM 文件夹提供的函数，实际应用到自己的项目中去。

本章包括如下 3 个小结：

- 5.1, delay 文件夹代码介绍;
- 5.2, sys 文件夹代码介绍;
- 5.3, usart 文件夹代码介绍;

5.1 delay 文件夹代码介绍

delay 文件夹内包含了 delay.c 和 delay.h 两个文件，这两个文件用来实现系统的延时功能，其中包含 3 个函数（这里我们不讲 SysTick_Handler 函数，该函数在讲 ucos 的时候再介绍）：

```
void delay_init(u8 SYSCLK);
void delay_ms(u16 nms);
void delay_us(u32 nus);
```

下面分别介绍这三个函数，在介绍之前，我们先了解一下编程思想：CM3 内核的处理器，内部包含了一个 SysTick 定时器，SysTick 是一个 24 位的倒计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick 在《STM32 的参考手册》（这里是指 V10.0 版本，下同）里面介绍的很简单，其详细介绍，请参阅《Cortex-M3 权威指南》第 133 页。我们就是利用 STM32 的内部 SysTick 来实现延时的，这样既不占用中断，也不占用系统定时器。

这里我们将介绍的是 ALIENTEK 提供的最新版本的延时函数，该版本的延时函数支持在 ucos 下面使用，它可以和 ucos 共用 systick 定时器。首先我们简单介绍下 ucos 的时钟：ucos 运行需要一个系统时钟节拍（类似“心跳”），而这个节拍是固定的（由 OS_TICKS_PER_SEC 设置），比如 5ms（设置：OS_TICKS_PER_SEC=200 即可），在 STM32 下面，一般是由 systick 来提供这个节拍，也就是 systick 要设置为 5ms 中断一次，为 ucos 提供时钟节拍，而且这个时钟一般是不能被打断的（否则就不准了）。

因为在 ucos 下 systick 不能再被随意更改，如果我们还想利用 systick 来做 delay_us 或者 delay_ms 的延时，就必须想点办法了，这里我们利用的是时钟摘取法。以 delay_us 为例，比如 delay_us (50)，在刚进入 delay_us 的时候先计算好这段延时需要等待的 systick 计数次数，这里为 50*9（假设系统时钟为 72Mhz，那么 systick 每增加 1，就是 1/9us），然后我们就一直统计 systick 的计数变化，直到这个值变化了 50*9，一旦检测到变化达到或者超过这个值，就说明延时 50us 时间到了。

下面我们开始介绍这几个函数。

5.1.1 delay_init 函数

该函数用来初始化 2 个重要参数：fac_us 以及 fac_ms；同时把 SysTick 的时钟源选择为外部时钟，如果使用了 ucos，那么还会根据 OS_TICKS_PER_SEC 的配置情况，来配置 SysTick 的中断时间，并开启 SysTick 中断。具体代码如下：

```
//初始化延迟函数
//SYSTICK 的时钟固定为 HCLK 时钟的 1/8
void delay_init()
{
//如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
#ifndef OS_CRITICAL_METHOD
    u32 reload;
#endif

SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8); //选择外部时钟 HCLK/8
fac_us=SystemCoreClock/8000000; //为系统时钟的 1/8
//如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
#ifndef OS_CRITICAL_METHOD
    reload=SystemCoreClock/8000000; //每秒钟的计数次数 单位为 K
    reload*=1000000/OS_TICKS_PER_SEC; //根据 OS_TICKS_PER_SEC 设定溢出时间
        //reload 为 24 位寄存器,最大值:16777216,在 72M 下,
        //约 1.86s 左右
    fac_ms=1000/OS_TICKS_PER_SEC; //代表 ucos 可以延时的最少单位
    SysTick->CTRL|=SysTick_CTRL_TICKINT_Msk; //开启 SYSTICK 中断
    SysTick->LOAD=reload; //每 1/OS_TICKS_PER_SEC 秒中断一次
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk; //开启 SYSTICK
#else
    fac_ms=(u16)fac_us*1000; //非 ucos 下,代表每个 ms 需要的 systick 时钟数
#endif
}
```

可以看到，delay_init 函数使用了条件编译，来选择不同的初始化过程，如果不使用 ucos 的时候，就和《不完全手册》介绍的方法是一样的，而如果使用 ucos 的时候，则会进行一些不同的配置，这里的条件编译是根据 OS_CRITICAL_METHOD 这个宏来确定的，因为只要使用了 ucos，就一定会定义 OS_CRITICAL_METHOD 这个宏。

SysTick 是 MDK 定义了一个结构体（在 stm32f10x_map.h 里面），里面包含 CTRL、LOAD、VAL、CALIB 等 4 个寄存器，

SysTick->CTRL 的各位定义如图 5.1.1.1 所示：



位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后, SysTick 已经数到了 0, 则该位为 1。如果读取该位, 该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=Systick 倒数到 0 时产生 Systick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	Systick 定时器的使能位

图 5.1.1.1 Systick->CTRL 寄存器各位定义

Systick-> LOAD 的定义如图 5.1.1.2 所示:

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时, 将被重装载的值

图 5.1.1.2 Systick->LOAD 寄存器各位定义

Systick-> VAL 的定义如图 5.1.1.3 所示:

位段	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取时返回当前倒计数的值, 写它则使之清零, 同时还会清除在 Systick 控制及状态寄存器中的 COUNTFLAG 标志

图 5.1.1.3 Systick->VAL 寄存器各位定义

Systick-> CALIB 不常用, 在这里我们也用不到, 故不介绍了。

Systick_CLKSourceConfig(Systick_CLKSource_HCLK_Div8);这一句把 Systick 的时钟选择外部时钟, 这里需要注意的是: Systick 的时钟源自 HCLK 的 8 分频, 假设我们外部晶振为 8M, 然后倍频到 72M, 那么 Systick 的时钟即为 9Mhz, 也就是 Systick 的计数器 VAL 每减 1, 就代表时间过了 1/9us。所以 fac_us=SystemCoreClock/8000000;这句话就是计算在 SystemCoreClock 时钟频率下延时 1us 需要多少个 Systick 时钟周期。同理, fac_ms=(u16)fac_us*1000;就是计算延时 1ms 需要多少个 Systick 时钟周期, 它自然是 1us 的 1000 倍。初始化将计算出 fac_us 和 fac_ms 的值。

在不使用 ucos 的时候: fac_us, 为 us 延时的基数, 也就是延时 1us, Systick->LOAD 所应设置的值。fac_ms 为 ms 延时的基数, 也就是延时 1ms, Systick->LOAD 所应设置的值。fac_us 为 8 位整形数据, fac_ms 为 16 位整形数据。正因为如此, 系统时钟如果不是 8 的倍数, 则会导致延时函数不准确, 这也是我们推荐外部时钟选择 8M 的原因。这点大家要特别留意。

当使用 ucos 的时候, fac_us, 还是 us 延时的基数, 不过这个值不会被写到 Systick->LOAD 寄存器来实现延时, 而是通过时钟摘取的办法实现的(后面会介绍)。而 fac_ms 则代表 ucos 自带的延时函数所能实现的最小延时时间(如 OS_TICKS_PER_SEC=200, 那么 fac_ms 就是 5ms)。

5.1.2 delay_us 函数

该函数用来延时指定的 us, 其参数 nus 为要延时的微秒数。该函数有使用 ucos 和不使



用 ucos 两个版本，这里我们分别介绍，首先是不使用 ucos 的时候，实现函数如下：

```
//延时 nus
//nus 为要延时的 us 数.
void delay_us(u32 nus)
{
    u32 temp;
    SysTick->LOAD=nus*fac_us; //时间加载
    SysTick->VAL=0x00;         //清空计数器
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk ;           //开始倒数
    do
    {
        temp=SysTick->CTRL;
    }
    while(temp&0x01&&!(temp&(1<<16))); //等待时间到达
    SysTick->CTRL&=~SysTick_CTRL_ENABLE_Msk;           //关闭计数器
    SysTick->VAL =0X00;          //清空计数器
}
```

有了上面对 SysTick 寄存器的描述，这段代码不难理解。其实就是先把要延时的 us 数换算成 SysTick 的时钟数，然后写入 LOAD 寄存器。然后清空当前寄存器 VAL 的内容，再开启倒数功能。等到倒数结束，即延时了 nus。最后关闭 SysTick，清空 VAL 的值。实现一次延时 nus 的操作，但是这里要注意 nus 的值，不能太大，必须保证 $nus \leq (2^{24})/fac_us$ ，否则将导致延时时间不准确。这里特别说明一下：`temp&0x01`，这一句是用来判断 systick 定时器是否还处于开启状态，可以防止 systick 被意外关闭导致的死循环。这里面有一行开启 Systick 开始倒数代码需要解释一下：

```
SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk ;
```

其中 `SysTick_CTRL_ENABLE_Msk` 是 MDK 宏定义的一个变量，它的值就是 `0x01`，这行代码的意思就是设置 `SysTick->CTRL` 的第一位为 1，使能定时器。

再来看看使用 ucos 的时候，`delay_us` 的实现函数如下：

```
//延时 nus
//nus 为要延时的 us 数.
void delay_us(u32 nus)
{
    u32 ticks;
    u32 told,tnow,tcnt=0;
    u32 reload=SysTick->LOAD;      //LOAD 的值
    ticks=nus*fac_us;             //需要的节拍数
    tcnt=0;
    OSSchedLock();                //阻止 ucos 调度，防止打断 us 延时
    told=SysTick->VAL;            //刚进入时的计数器值
    while(1)
    {
        tnow=SysTick->VAL;
        if(tnow!=told)
```



```

    {
        if(tnow<told)tcnt+=told-tnow;//这里注意：SYSTICK 是一个递减的计数器就.
        else tcnt+=reload-tnow+told;
        told=tnow;
        if(tcnt>=ticks)break;//时间超过/等于要延迟的时间,则退出.
    }
};

OSSchedUnlock();           //开启 ucos 调度
}

```

这里就正是利用了我们前面提到的时钟摘取法， ticks 是延时 nus 需要等待的 SysTick 计数次数（也就是延时时间）， told 用于记录最近一次的 SysTick->VAL 值，然后 tnow 则是当前的 SysTick->VAL 值，通过他们的对比累加，实现 SysTick 计数次数的统计，统计值存放在 tcnt 里面，然后通过对比 tcnt 和 ticks，来判断延时是否到达，从而达到不修改 SysTick 实现 nus 的延时，从而可以和 ucos 共用一个 SysTick。

上面的 OSSchedLock 和 OSSchedUnlock 是 ucos 提供的两个函数，用于调度上锁和解锁，这里为了防止 ucos 在 delay_us 的时候打断延时，可能导致的延时不准，所以我们利用这两个函数来实现免打断，从而保证延时精度！同时，此时的 delay_us，可以实现最长 2^{32}us 的延时，大概是 4294 秒。

5.1.3 delay_ms 函数

该函数用来延时指定的 ms，其参数 nms 为要延时的微秒数。该函数同样有使用 ucos 和不使用 ucos 两个版本，这里我们分别介绍，首先是不使用 ucos 的时候，实现函数如下：

```

//延时 nms
//注意 nms 的范围
//SysTick->LOAD 为 24 位寄存器,所以,最大延时为:
//nms<=0xfffff*8*1000/SYSCLK
//SYSCLK 单位为 Hz,nms 单位为 ms
//对 72M 条件下,nms<=1864
void delay_ms(u16 nms)
{
    u32 temp;
    SysTick->LOAD=(u32)nms*fac_ms;//时间加载(SysTick->LOAD 为 24bit)
    SysTick->VAL =0x00;           //清空计数器
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk ;           //开始倒数
    do
    {
        temp=SysTick->CTRL;
    }
    while(temp&0x01&&!(temp&(1<<16)));//等待时间到达
    SysTick->CTRL&=~SysTick_CTRL_ENABLE_Msk;          //关闭计数器
    SysTick->VAL =0X00;           //清空计数器
}

```



此部分代码和 5.1.2 节的 delay_us (非 ucos 版本) 大致一样，但是要注意因为 LOAD 仅仅是一个 24bit 的寄存器，延时的 ms 数不能太长。否则超出了 LOAD 的范围，高位会被舍去，导致延时不准。最大延迟 ms 数可以通过公式: nms<=0xfffff*8*1000/SYSCLK 计算。SYSCLK 单位为 Hz, nms 的单位为 ms。如果时钟为 72M, 那么 nms 的最大值为 1864ms。超过这个值，建议通过多次调用 delay_ms 实现，否则就会导致延时不准确。

再来看看使用 ucos 的时候，delay_ms 的实现函数如下：

```
//延时 nms
//nms:要延时的 ms 数
void delay_ms(u16 nms)
{
    if(OSRunning==TRUE)//如果 os 已经在跑了
    {
        if(nms>=fac_ms)//延时的时间大于 ucos 的最少时间周期
        {
            OSTimeDly(nms/fac_ms);//ucos 延时
        }
        nms%=fac_ms;//ucos 已经无法提供这么小的延时了,采用普通方式延时
    }
    delay_us((u32)(nms*1000)); //普通方式延时
}
```

该函数中，OSRunning 是 ucos 正在运行的一个标志，OSTimeDly 是 ucos 提供的一个基于 ucos 时钟节拍的延时函数，其参数代表延时的时钟节拍数（假设 OS_TICKS_PER_SEC=200，那么 OSTimeDly(1)，就代表延时 5ms）。

当 ucos 还未运行的时候，我们的 delay_ms 就是直接由 delay_us 实现的，ucos 下的 delay_us 可以实现很长的延时而不溢出！，所以放心的使用 delay_us 来实现 delay_ms，不过由于 delay_us 的时候，任务调度被上锁了，所以还是建议不要用 delay_us 来延时很长的时间，否则影响整个系统的性能。

当 ucos 运行的时候，我们的 delay_ms 函数将先判断延时时长是否大于等于 1 个 ucos 时钟节拍 (fac_ms)，当大于这个值的时候，我们就通过调用 ucos 的延时函数来实现（此时任务可以调度），不足 1 个时钟节拍的时候，直接调用 delay_us 函数实现（此时任务无法调度）。

5.2 sys 文件夹代码介绍

sys 文件夹内包含了 sys.c 和 sys.h 两个文件。在 sys.h 里面定义了 STM32 的 IO 口输入读取宏定义和输出宏定义。sys.c 里面只定义了一个中断分组函数。下面我们将分别向大家介绍。

5.2.1 IO 口的位操作实现

该部分代码在 sys.h 文件中，实现对 STM32 各个 IO 口的位操作，包括读入和输出。当然在这些函数调用之前，必须先进行 IO 口时钟的使能和 IO 口功能定义。此部分仅仅对 IO 口进行输入输出读取和控制。



位带操作简单的说，就是把每个比特膨胀为一个 32 位的字，当访问这些字的时候就达到了访问比特的目的，比如说 BSRR 寄存器有 32 个位，那么可以映射到 32 个地址上，我们去访问这 32 个地址就达到访问 32 个比特的目的。这样我们往某个地址写 1 就达到往对应比特位写 1 的目的，同样往某个地址写 0 就达到往对应的比特位写 0 的目的。

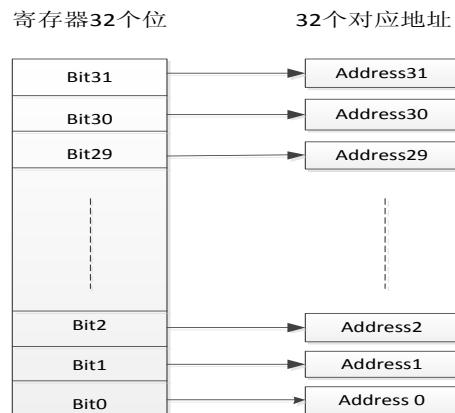


图 5.2.2.1 位带映射图

对于上图，我们往 Address0 地址写入 1，那么就可以达到往寄存器的第 0 位 Bit0 赋值 1 的目的。这里我们不想讲得过于复杂，因为位带操作在实际开发中可能只是用来 IO 口的输入输出还比较方便，其他操作在日常开发中也基本很少用。下面我们看看 sys.h 中位带操作的定义。

代码如下：

```
#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr & 0xFFFF)<<5)+(bitnum<<2))
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
#define BIT_ADDR(addr, bitnum) MEM_ADDR(BITBAND(addr, bitnum))

//IO 口地址映射
#define GPIOA_ODR_Addr      (GPIOA_BASE+12)//0x4001080C
#define GPIOB_ODR_Addr      (GPIOB_BASE+12)//0x40010C0C
#define GPIOC_ODR_Addr      (GPIOC_BASE+12)//0x4001100C
#define GPIOD_ODR_Addr      (GPIOD_BASE+12)//0x4001140C
#define GPIOE_ODR_Addr      (GPIOE_BASE+12)//0x4001180C
#define GPIOF_ODR_Addr      (GPIOF_BASE+12)//0x40011A0C
#define GPIOG_ODR_Addr      (GPIOG_BASE+12)//0x40011E0C

#define GPIOA_IDR_Addr      (GPIOA_BASE+8)//0x40010808
#define GPIOB_IDR_Addr      (GPIOB_BASE+8)//0x40010C08
#define GPIOC_IDR_Addr      (GPIOC_BASE+8)//0x40011008
#define GPIOD_IDR_Addr      (GPIOD_BASE+8)//0x40011408
#define GPIOE_IDR_Addr      (GPIOE_BASE+8)//0x40011808
#define GPIOF_IDR_Addr      (GPIOF_BASE+8)//0x40011A08
#define GPIOG_IDR_Addr      (GPIOG_BASE+8)//0x40011E08

//IO 口操作，只对单一的 IO 口！
//确保 n 的值小于 16!
```



```
#define PAout(n)    BIT_ADDR(GPIOA_ODR_Addr, n) //输出
#define PAin(n)      BIT_ADDR(GPIOA_IDR_Addr, n) //输入
#define PBout(n)     BIT_ADDR(GPIOB_ODR_Addr, n) //输出
#define PBin(n)      BIT_ADDR(GPIOB_IDR_Addr, n) //输入
#define PCout(n)     BIT_ADDR(GPIOC_ODR_Addr, n) //输出
#define PCin(n)      BIT_ADDR(GPIOC_IDR_Addr, n) //输入
#define PDout(n)     BIT_ADDR(GPIOD_ODR_Addr, n) //输出
#define PDin(n)      BIT_ADDR(GPIOD_IDR_Addr, n) //输入
#define PEout(n)     BIT_ADDR(GPIOE_ODR_Addr, n) //输出
#define PEin(n)      BIT_ADDR(GPIOE_IDR_Addr, n) //输入
#define PFout(n)     BIT_ADDR(GPIOF_ODR_Addr, n) //输出
#define PFin(n)      BIT_ADDR(GPIOF_IDR_Addr, n) //输入
#define PGout(n)     BIT_ADDR(GPIOG_ODR_Addr, n) //输出
#define PGin(n)      BIT_ADDR(GPIOG_IDR_Addr, n) //输入
```

以上代码的便是 GPIO 位带操作的具体实现，位带操作的详细说明，在权威指南中有详细讲解，请参考<<CM3 权威指南>>第五章(87 页~92 页)。比如说，我们调用 PAout(1)=1 是设置了 GPIOA 的第一个管脚 GPIOA.1 为 1，实际是设置了寄存器的某个位，但是我们的定义中可以跟踪过去看到却是通过计算访问了一个地址。上面一系列公式也就是计算 GPIO 的某个 io 口对应的位带区的地址了。

有了上面的代码，我们就可以像 51/AVR 一样操作 STM32 的 IO 口了。比如，我要 PORTA 的第七个 IO 口输出 1，则可以使用 PAout (6) =1；即可实现。我要判断 PORTA 的第 15 位是否等于 1，则可以使用 if (PAin (14) ==1) ...；就可以了。

这里顺便说一下，在 sys.h 中的还有个全局宏定义：

```
//0,不支持 ucos
//1,支持 ucos
#define SYSTEM_SUPPORT_UCOS      0          //定义系统文件夹是否支持 UCOS
```

SYSTEM_SUPPORT_UCOS，这个宏定义用来定义 SYSTEM 文件夹是否支持 ucos，如果在 ucos 下面使用 SYSTEM 文件夹，那么设置这个值为 1 即可，否则设置为 0（默认）。

5.2.2 中断分组设置函数

在 sys.c 里面只有一个函数就是 void NVIC_Configuration()中断配置函数，在这个函数里面我们只调用了固件库的中断分组配置函数，这只整个系统的中断分组为组别 2.这个函数在系统初始化的时候调用即可，并且永远只需要调用一次。

```
void NVIC_Configuration(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    //设置 NVIC 中断分组 2:2 位抢占//优先级，2 位响应优先级
}
```

5.3 usart 文件夹介绍

uart 文件夹内包含了 usart.c 和 usart.h 两个文件。这两个文件用于串口的初始化和中

断接收。这里只是针对串口 1，比如你要用串口 2 或者其他的串口，只要对代码稍作修改就可以了。uart.c 里面包含了 2 个函数一个是 void USART1_IRQHandler(void);另外一个是 void uart_init(u32 bound);里面还有一段对串口 printf 的支持代码，如果去掉，则会导致 printf 无法使用，虽然软件编译不会报错，但是硬件上 STM32 是无法启动的，这段代码不要去修改。

5.3.1 printf 函数支持

这段引入 printf 函数支持的代码在 usart.h 头文件的最上方，这段代码加入之后便可以通过 printf 函数向串口发送我们需要的内容，方便开发过程中查看代码执行情况以及一些变量值。这段代码不需要修改，引入到 usart.h 即可。

这段代码为：

```
//加入以下代码,支持 printf 函数,而不需要选择 use MicroLIB
#ifndef __USE_NO_SEMIHOSTING
#define __USE_NO_SEMIHOSTING
//标准库需要的支持函数
struct __FILE
{
    int handle;
};

FILE __stdout;
//定义_sys_exit()以避免使用半主机模式
_sys_exit(int x)
{
    x = x;
}

//重定义 fputc 函数
int fputc(int ch, FILE *f)
{
    while(USART_GetFlagStatus(USART1,USART_FLAG_TC)==RESET);
    USART_SendData(USART1,(uint8_t)ch);
    return ch;
}
#endif
```

5.3.2 uart_init 函数

void uart_init(u32 pclk2, u32 bound)函数是串口 1 初始化函数。该函数有 1 个参数为波特率，波特率这个参数对于大家来说应该不陌生，这里就不多说了。

```
void uart_init(u32 bound){
    //GPIO 端口设置
    GPIO_InitTypeDef GPIO_InitStructure;
```



```
USART_InitTypeDef USART_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;

RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1|RCC_APB2Periph_GPIOA
|RCC_APB2Periph_AFIO, ENABLE); //使能 USART1, GPIOA 时钟
//以及复用功能时钟

//USART1_TX PA.9
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //PA.9 复用推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 GPIOA.0 发送端

//USART1_RX PA.10 浮空输入
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;//浮空输入
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 GPIOA.10 接收端

//Usart1 NVIC 中断配置 配置
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn; //对应中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=3;//抢占优先级 3
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //子优先级 3
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure); //中断优先级配置

//USART 初始化设置
USART_InitStructureUSART_BaudRate = bound;//波特率设置;
USART_InitStructureUSART_WordLength = USART_WordLength_8b;//字长为 8 位
USART_InitStructureUSART_StopBits = USART_StopBits_1;//一个停止位
USART_InitStructureUSART_Parity = USART_Parity_No; //无奇偶校验位
USART_InitStructureUSART_HardwareFlowControl=
USART_HardwareFlowControl_None;//无硬件数据流控制
USART_InitStructureUSART_Mode = USART_Mode_Rx |USART_Mode_Tx;//收发模式
USART_Init(USART1, &USART_InitStructure); //初始化串口

USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);//开启中断
USART_Cmd(USART1, ENABLE); //使能串口
}
```

下面我们一一分析一下这段初始化代码。首先是一行时钟使能代码：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1
RCC_APB2Periph_GPIOA, ENABLE); //使能 USART1, GPIOA 时钟
```

这个时钟使能我们在端口复用的时候已经讲解过，大家可以翻到端口复用那一章节，有详细的讲解。在使用一个内置外设的时候，我们首先要使能相应的 GPIO 时钟，然后使能复用功能时钟和内置外设时钟。



接下来我们要初始化相应的 GPIO 端口为特定的状态，我们在复用内置外设的时候到底 GPIO 要设置成什么模式呢？这个在我们的端口复用一节也有讲解，那就是在《STM32 中文参考手册 V10》的 P110 “8.1.11 外设的 GPIO 配置”中有讲解，我们就继续截图下来：

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

所以接下来的两段代码就是将 TX(PA9)设置为推挽复用输出模式，将 RX(PA10)设置为浮空输入模式：

```
//USART1_TX PA.9
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //PA.9 复用推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
GPIO_Init(GPIOA, &GPIO_InitStructure);

//USART1_RX PA.10 浮空输入
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;//浮空输入
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

对于 GPIO 的知识我们在跑马灯实例会讲解到，这里暂时不做深入的讲解。

紧接着，我们要进行 usart1 的中断初始化，设置抢占优先级值和子优先级的值：

```
//Usart1 NVIC 中断配置 配置
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=3;//抢占优先级 3
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //子优先级 3
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化 VIC 寄存器
```

这段代码在我们的中断管理函数章节 4.5 有讲解中断管理相关的知识，大家可以翻阅一下。

在设置完中断优先级之后，接下来我们要设置串口 1 的初始化参数：

```
//USART 初始化设置
USART_InitStructureUSART_BaudRate = bound;//一般设置为 9600;
USART_InitStructureUSART_WordLength = USART_WordLength_8b;//字长为 8 位
USART_InitStructureUSART_StopBits = USART_StopBits_1;//一个停止位
USART_InitStructureUSART_Parity = USART_Parity_No; //无奇偶校验位
USART_InitStructureUSART_HardwareFlowControl=
    USART_HardwareFlowControl_None;//无硬件数据流控制
USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx;//收发
USART_Init(USART1, &USART_InitStructure); //初始化串口
```

从上面的源码我们可以看出，串口的初始化是通过调用 USART_Init()函数实现，而这个函数重要的参数就是就是结构体指针变量 USART_InitStructure，下面我们看看结构体定义：

```
typedef struct
```

```
{
```



```

    uint32_t USART_BaudRate;
    uint16_t USART_WordLength;
    uint16_t USART_StopBits;
    uint16_t USART_Parity;
    uint16_t USART_Mode;
    uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;

```

这个结构体有 6 个成员变量，所以我们有 6 个参数需要初始化。

第一个参数 `USART_BaudRate` 为串口波特率，波特率可以说是串口最重要的参数了，我们这里通过初始化传入参数 `baund` 来设定。第二个参数 `USART_WordLength` 为字长，这里我们设置为 8 位字长数据格式。第三个参数 `USART_StopBits` 为停止位设置，我们设置为 1 位停止位。第四个参数 `USART_Parity` 设定是否需要奇偶校验，我们设定为无奇偶校验位。第五个参数 `USART_Mode` 为串口模式，我们设置为全双工收发模式。第六个参数为是否支持硬件流控制，我们设置为无硬件流控制。

在设置完成串口中断优先级以及串口初始化之后，接下来就是开启串口中断以及使能串口了：

```

USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //开启中断
USART_Cmd(USART1, ENABLE); //使能串口

```

在开启串口中断和使能串口之后接下来就是写中断处理函数了，下一节我们将着重讲解中断处理函数。

5.3.3 USART1_IRQHandler 函数

`void USART1_IRQHandler(void)` 函数是串口 1 的中断响应函数，当串口 1 发生了相应的中断后，就会跳到该函数执行。中断相应函数的名字是不能随便定义的，一般我们都遵循 MDK 定义的函数名。这些函数名字在启动文件 `startup_stm32f10x_hd.s` 文件中可以找到。函数体里面通过函数：

```

if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    判断是否接受中断，如果是串口接受中断，则读取串口接收到的数据：
    Res = USART_ReceiveData(USART1); // (USART1->DR); //读取接收到的数据

```

读到数据后接下来就对数据进行分析。

这里我们设计了一个小小的接收协议：通过这个函数，配合一个数组 `USART_RX_BUF[]`，一个接收状态寄存器 `USART_RX_STA`（此寄存器其实就是一个全局变量，由作者自行添加。由于它起到类似寄存器的功能，这里暂且称之为寄存器）实现对串口数据的接收管理。`USART_RX_BUF` 的大小由 `USART_REC_LEN` 定义，也就是一次接收的数据最大不能超过 `USART_REC_LEN` 个字节。`USART_RX_STA` 是一个接收状态寄存器其各的定义如表 5.3.1.1 所示：

USART_RX_STA		
bit15	bit14	bit13~0
接收完成 标志	接收到 0X0D 标志	接收到的有效数据个数

表 5.3.1.1 接收状态寄存器位定义表

设计思路如下：



当接收到从电脑发过来的数据，把接收到的数据保存在 USART_RX_BUF 中，同时在接收状态寄存器（USART_RX_STA）中计数接收到的有效数据个数，当收到回车（回车的表示由 2 个字节组成：0X0D 和 0X0A）的第一个字节 0X0D 时，计数器将不再增加，等待 0X0A 的到来，而如果 0X0A 没有来到，则认为这次接收失败，重新开始下一次接收。如果顺利接收到 0X0A，则标记 USART_RX_STA 的第 15 位，这样完成一次接收，并等待该位被其他程序清除，从而开始下一次的接收，而如果迟迟没有收到 0X0D，那么在接收数据超过 USART_REC_LEN 的时候，则会丢弃前面的数据，重新接收。中断相应函数代码如下：

```
void USART1_IRQHandler(void) //串口 1 中断服务程序
{
    u8 Res;
#ifdef OS_TICKS_PER_SEC //如果时钟节拍数定义了,说明要使用 ucosII 了.
    OSIntEnter();
#endif
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
        //接收中断(接收到的数据必须是 0xd 0xa 结尾)
    {
        Res =USART_ReceiveData(USART1);//(USART1->DR); //读取接收到的数据

        if((USART_RX_STA&0x8000)==0)//接收未完成
        {
            if(USART_RX_STA&0x4000)//接收到了 0xd
            {
                if(Res!=0xa)USART_RX_STA=0;//接收错误,重新开始
                else USART_RX_STA|=0x8000; //接收完成了
            }
            else //还没收到 0X0D
            {
                if(Res==0xd)USART_RX_STA|=0x4000;
                else
                {
                    USART_RX_BUF[USART_RX_STA&0X3FFF]=Res ;
                    USART_RX_STA++;
                    if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;
                    //接收数据错误,重新开始接收
                }
            }
        }
    }
#ifdef OS_TICKS_PER_SEC //如果时钟节拍数定义了,说明要使用 ucosII 了.
    OSIntExit();
#endif
}
```

EN_USART1_RX 和 USART_REC_LEN 都是在 usart.h 文件里面定义的，当需要使用串口接收的时候，我们只要在 usart.h 里面设置 EN_USART1_RX 为 1 就可以了。不使用的时候，设置，EN_USART1_RX 为 0 即可，这样可以省出部分 sram 和 flash，我们默认是设置 EN_USART1_RX 为 1，也就是开启串口接收的。

OS_CRITICAL_METHOD，则是用来判断是否使用 ucos，如果使用了 ucos，则调用 OSIntEnter 和 OSIntExit 函数，如果没有使用 ucos，则不调用这两个函数（这两个函数用于实现中断嵌套处理，这里我们先不理会）。

第三篇 实战篇

经过前两篇的学习，我们对 STM32 开发的软件和硬件平台都有了个比较深入的了解了，接下来我们将通过实例，由浅入深，带大家一步步的学习 STM32。

STM32 的内部资源非常丰富，对于初学者来说，一般不知道从何开始。本篇将从 STM32 最简单的外设说起，然后一步步深入。每一个实例都配有详细的代码及解释，手把手教你如何入手 STM32 的各种外设，通过本篇的学习，希望大家能学会 STM32 绝大部分外设的使用。

本篇总共分为 56 章，每一章即一个实例，下面就让我们开始精彩的 STM32 之旅。

我们固件库版本源码在光盘目录：程序源码\标准例程-V3.5 库函数版本\ 之下。

第六章 跑马灯实验

STM32 最简单的外设莫过于 IO 口的高低电平控制了，本章将通过一个经典的跑马灯程序，带大家开启 STM32 之旅，通过本章的学习，你将了解到 STM32 的 IO 口作为输出使用的方法。在本章中，我们将通过代码控制 ALIENTEK 战舰 STM32 开发板上的两个 LED：DS0 和 DS1 交替闪烁，实现类似跑马灯的效果。本章分为如下四个小节：

- 6.1, STM32 IO 口简介
- 6.2, 硬件设计
- 6.3, 软件设计
- 6.4, 仿真与下载



6.1 STM32 IO 简介

本章将要实现的是控制 ALIENTEK 战舰 STM32 开发板上的两个 LED 实现一个类似跑马灯的效果，该实验的关键在于如何控制 STM32 的 IO 口输出。了解了 STM32 的 IO 口如何输出的，就可以实现跑马灯了。通过这一章的学习，你将初步掌握 STM32 基本 IO 口的使用，而这是迈向 STM32 的第一步。

这一章节因为是第一个实验章节，所以我们在这一章将讲解一些知识为后面的实验做铺垫。为了小节标号与后面实验章节一样，这里我们不另起一节来讲。

在讲解 STM32 的 GPIO 之前，首先打开我们光盘的第一个固件库版本实验工程跑马灯实验工程（光盘目录为：“4，程序源码\标准例程-V3.5 库函数版本\实验 1 跑马灯/USER/LED.Uv2”），可以看到我们的实验工程目录：

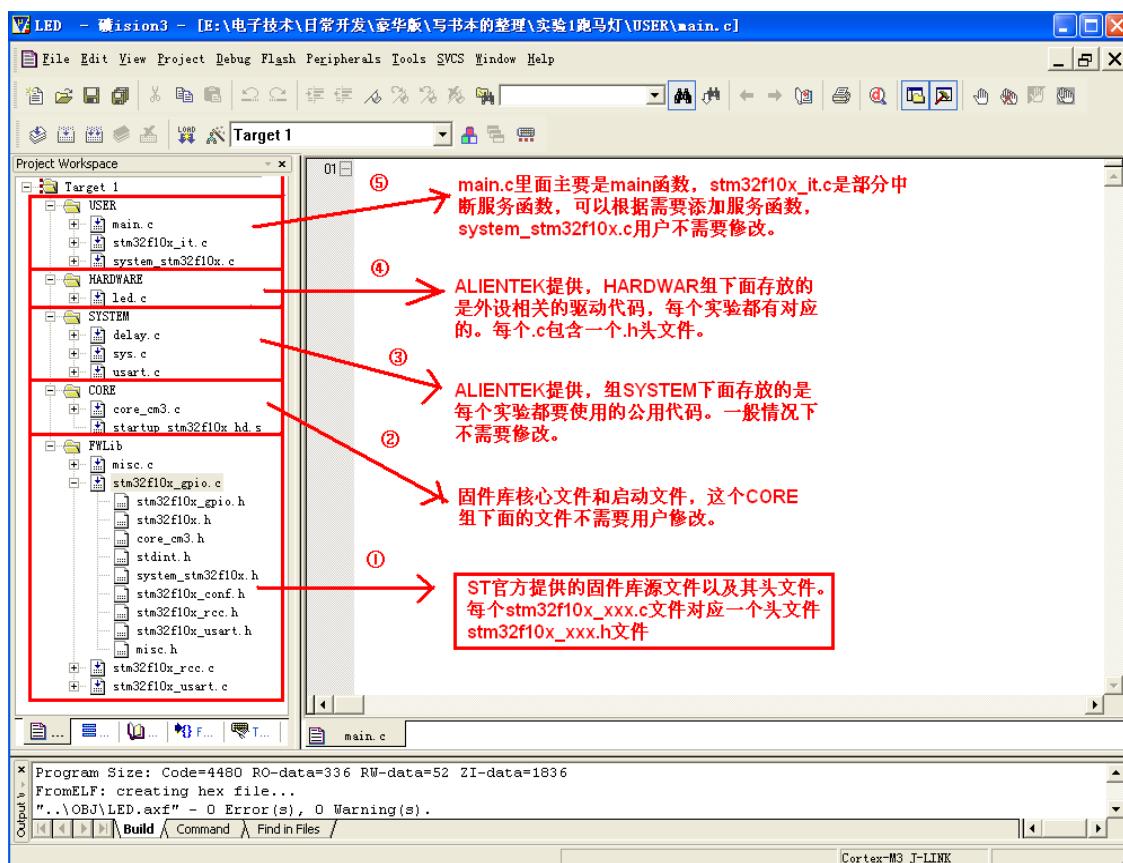


图 6.1.1 跑马灯实验目录结构

接下来我们逐一讲解一下我们的工程目录下面的组以及重要文件。

- ① 组 FWLib 下面存放的是 ST 官方提供的固件库函数，里面的函数我们可以根据需要添加和删除，但是一定要注意在头文件 `stm32f10x_conf.h` 文件中注释掉删除的源文件对应的头文件，这里面的文件内容用户不需要修改。
- ② 组 CORE 下面存放的是固件库必须的核心文件和启动文件。这里面的文件用户不需要修改。
- ③ 组 SYSTEM 是 ALIENTEK 提供的共用代码，这些代码的作用和讲解在第五章都有讲解，大家可以翻过去看下。



- ④ 组 HARDWARE 下面存放的是每个实验的外设驱动代码，他的实现是通过调用 FWLib 下面的固件库文件实现的，比如 led.c 里面调用 stm32f10x_gpio.c 里面的函数对 led 进行初始化，这里面的函数是讲解的重点。后面的实验中可以看到会引入多个源文件。
- ⑤ 组 USER 下面存放的主要是用户代码。但是 system_stm32f10x.c 文件用户不需要修改，同时 stm32f10x_it.c 里面存放的是中断服务函数，这两个文件的作用在 3.1 节有讲解，大家可以翻过去看看。Main.c 函数主要存放的是主函数了，这个大家应该很清楚。

针对第①步中怎么随意添加和删除固件库文件，这里我们稍微讲解一下。

首先从上面的图中可以看到，stm32f10x_gpio.c 源文件下面 include 了好几个头文件，其中有一个 stm32f10x_conf.h，这个文件会被每个固件库源文件引用。我们可以打开看看里面的内容：

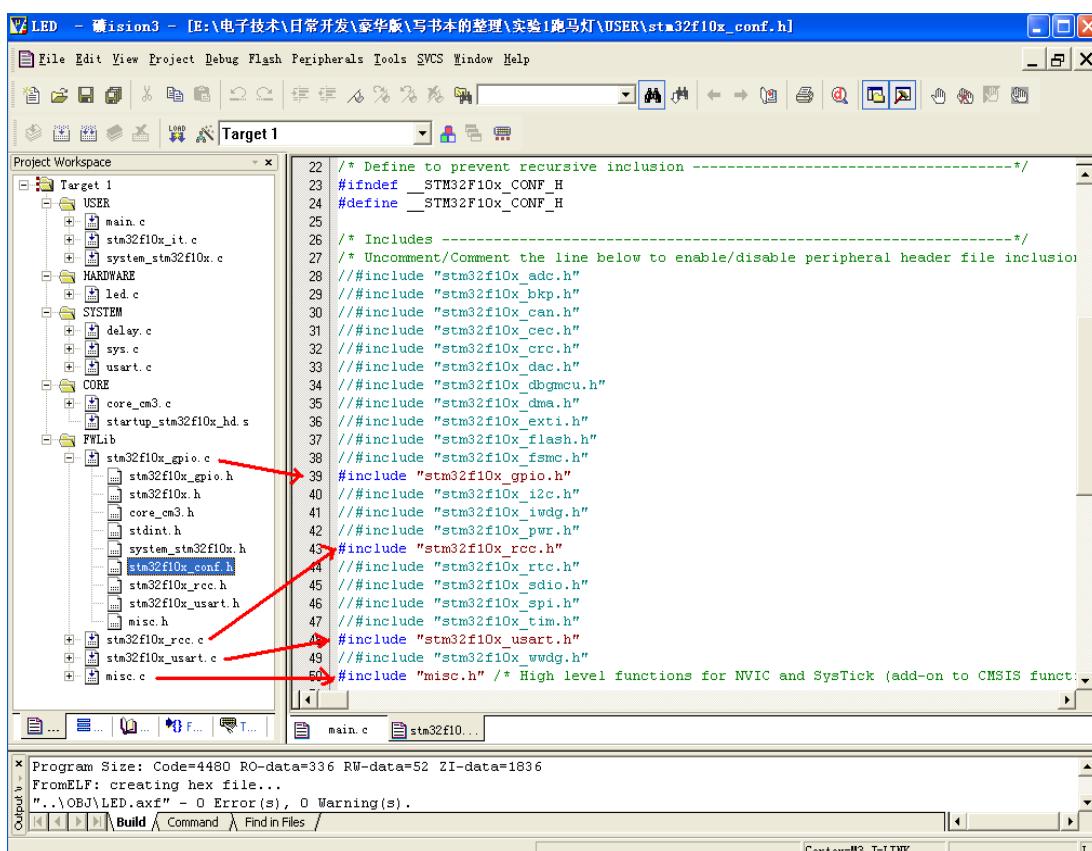


图 6.1.2 stm32f10x_conf 文件内容

从图中可以看出，在头文件 stm32f10x_conf.h 文件中，我们包含了四个.h 头文件，那是因为我们的 FWLib 组下面引入了相应的 4 个.c 源文件。同时大家记住，后面三个源文件 stm32f10x_rcc.c,stm32f10x_usart.c 以及 misc.c 在每个实验基本都需要添加。在这个实验中，因为 LED 是关系到 STM32 的 GPIO，所以我们增加了 stm32f10x_gpio.c 和头文件 stm32f10x_gpio.h 的引入。添加和删除固件库源文件的步骤是：

1. 在 stm32f10x_conf.h 文件引入需要的.h 头文件。这些头文件在每个实验的目录 \STM32F10x_FWLib\inc 下面都有存放。
2. 在 FWLib 下面加入步骤一中引入的.h 头文件对应的源文件。记住最好一一对应，否则就有可能会报错。这些源文件在每个实验的\STM32F10x_FWLib\src 目录下面都有存放。添加方法请参考 3.3 节的内容。

最后我们讲解一下这些组之间的层次结构：

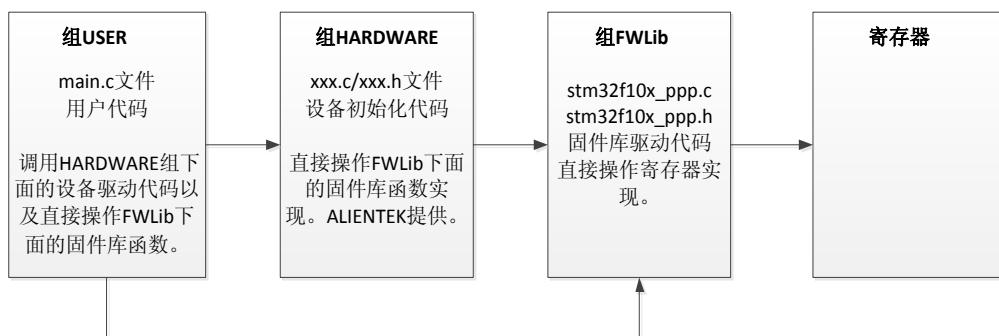


图 6.1.3 代码层次结构图

从层次图中可以看出，我们的用户代码和 HARDWARE 下面的外设驱动代码再不需要直接操作寄存器，而是直接或间接操作官方提供的固件库函数。但是后面我们的为了让大家更全面方便的了解外设，我们会增加重要的外设寄存器的讲解，这样对底层知识更加了解，方便我们深入学习固件库。

准备内容我们就讲解到这里，接下来我们就要进入我们跑马灯实验的讲解部分了。这里需要说明一下，我们在讲解固件库之前会首先对重要寄存器进行一个讲解，这样是为了大家对寄存器有个初步的了解。大家学习固件库，并不需要记住每个寄存器的作用，而只是通过了解寄存器来对外设一些功能有个大致的了解，这样对以后的学习也很有帮助。

首先要提一下，在固件库中，GPIO 端口操作对应的库函数函数以及相关定义在文件 `stm32f10x_gpio.h` 和 `stm32f10x_gpio.c` 中。

STM32 的 IO 口相比 51 而言要复杂得多，所以使用起来也困难很多。首先 STM32 的 IO 口可以由软件配置成如下 8 种模式：

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

每个 IO 口可以自由编程，但 IO 口寄存器必须要按 32 位字被访问。STM32 的很多 IO 口都是 5V 兼容的，这些 IO 口在与 5V 电平的外设连接的时候很有优势，具体哪些 IO 口是 5V 兼容的，可以从该芯片的数据手册管脚描述章节查到（I/O Level 标 FT 的就是 5V 电平兼容的）。

STM32 的每个 IO 端口都有 7 个寄存器来控制。他们分别是：配置模式的 2 个 32 位的端口配置寄存器 CRL 和 CRH；2 个 32 位的数据寄存器 IDR 和 ODR；1 个 32 位的置位/复位寄存器 BSRR；一个 16 位的复位寄存器 BRR；1 个 32 位的锁存寄存器 LCKR。大家如果想要了解每个寄存器的详细使用方法，可以参考《STM32 中文参考手册 V10》P105~P129。

CRL 和 CRH 控制着每个 IO 口的模式及输出速率。

STM32 的 IO 口位配置表如表 6.1.4 所示：



配置模式		CNF1	CNF0	MODE1	MODE0	PxODR寄存器
通用输出	推挽式(Push-Pull)	0	0	01 10 11 见表3.1.2	0或1 0或1 不使用 不使用	
	开漏(Open-Drain)		1			
复用功能输出	推挽式(Push-Pull)	1	0			
	开漏(Open-Drain)		1			
输入	模拟输入	0	0	00	不使用 不使用 0 1	
	浮空输入		1			
	下拉输入	1				
	上拉输入		0			

表 6.1.4 STM32 的 IO 口位配置表

STM32 输出模式配置如表 6.1.5 所示：

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

表 6.1.5 STM32 输出模式配置表

接下来我们看看端口低配置寄存器 CRL 的描述，如图 6.1.6 所示：

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

位31:30	CNFy[1:0]: 端口x配置位(y = 0...7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式
位29:28	MODEy[1:0]: 端口x的模式位(y = 0...7) 软件通过这些位配置相应的I/O端口，请参考表15端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式, 最大速度10MHz 10: 输出模式, 最大速度2MHz 11: 输出模式, 最大速度50MHz

图 6.1.6 端口低配置寄存器 CRL 各位描述

该寄存器的复位值为 0X4444 4444，从图 6.1.4 可以看到，复位值其实就是配置端口为浮空



输入模式。从上图还可以得出：STM32 的 CRL 控制着每组 IO 端口（A~G）的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式（ADC 用）、0X3 表示推挽输出模式（做输出口用，50M 速率）、0X8 表示上/下拉输入模式（做输入口用）、0XB 表示复用输出（使用 IO 口的第二功能，50M 速率）。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口。这里我们对 CRH 就不做详细介绍了。下面我们讲解一下怎样通过固件库设置 GPIO 的相关参数和输出。

GPIO 相关的函数和定义分布在固件库文件 stm32f10x_gpio.c 和头文件 stm32f10x_gpio.h 文件中。

在固件库开发中，操作寄存器 CRH 和 CRL 来配置 IO 口的模式和速度是通过 GPIO 初始化函数完成：

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
```

这个函数有两个参数，第一个参数是用来指定 GPIO，取值范围为 GPIOA~GPIOG。

第二个参数为初始化参数结构体指针，结构体类型为 GPIO_InitTypeDef。下面我们看看这个结构体的定义。首先我们打开我们光盘的跑马灯实验，然后找到 FWLib 组下面的 stm32f10x_gpio.c 文件，定位到 GPIO_Init 函数体处，双击入口参数类型 GPIO_InitTypeDef 后右键选择“Go to definition of ...”可以查看结构体的定义：

```
typedef struct
{
    uint16_t GPIO_Pin;
    GPIO_Speed_TypeDef GPIO_Speed;
    GPIO_Mode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

下面我们通过一个 GPIO 初始化实例来讲解这个结构体的成员变量的含义。

通过初始化结构体初始化 GPIO 的常用格式是：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;           //LED0-->PB.5 端口配置
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;     //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
GPIO_Init(GPIOB, &GPIO_InitStructure); //根据设定参数配置 GPIO
```

上面代码的意思是设置 GPIOB 的第 5 个端口为推挽输出模式，同时速度为 50M。从上面初始化代码可以看出，结构体 GPIO_InitStructure 的第一个成员变量 GPIO_Pin 用来设置是要初始化哪个或者哪些 IO 口；第二个成员变量 GPIO_Mode 是用来设置对应 IO 端口的输出输入模式，这些模式是上面我们讲解的 8 个模式，在 MDK 中是通过一个枚举类型定义的：

```
typedef enum
{
    GPIO_Mode_AIN = 0x0,           //模拟输入
    GPIO_Mode_IN_FLOATING = 0x04, //浮空输入
    GPIO_Mode_IPD = 0x28,          //下拉输入
    GPIO_Mode_IPU = 0x48,          //上拉输入
    GPIO_Mode_Out_OD = 0x14,       //开漏输出
    GPIO_Mode_Out_PP = 0x10,        //通用推挽输出
    GPIO_Mode_AF_OD = 0x1C,        //复用开漏输出
    GPIO_Mode_AF_PP = 0x18         //复用推挽
}
```



```
}GPIOMode_TypeDef;
```

第三个参数是 IO 口速度设置，有三个可选值，在 MDK 中同样通过枚举类型定义：

```
typedef enum
{
    GPIO_Speed_10MHz = 1,
    GPIO_Speed_2MHz,
    GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;
```

这些入口参数的取值范围怎么定位，怎么快速定位到这些入口参数取值范围的枚举类型，在我们上面章节 4.7 的“快速组织代码”章节有讲解，不明白的朋友可以翻回去看一下，这里我们就不重复讲解，在后面的实验中，我们也不再去重复讲解怎么定位每个参数的取值范围的方法。

IDR 是一个端口输入数据寄存器，只用了低 16 位。该寄存器为只读寄存器，并且只能以 16 位的形式读出。该寄存器各位的描述如图 6.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
位31:16		保留，始终读为0。													
位15:0		IDRy[15:0] : 端口输入数据(y = 0...15) 这些位为只读并只能以字(16位)的形式读出。读出的值为对应I/O口的状态。													

图 6.1.7 端口输入数据寄存器 IDR 各位描述

要想知道某个 IO 口的电平状态，你只要读这个寄存器，再看某个位的状态就可以了。使用起来是比较简单的。

在固件库中操作 IDR 寄存器读取 IO 端口数据是通过 GPIO_ReadInputDataBit 函数实现的：

```
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

比如我要读 GPIOA.5 的电平状态，那么方法是：

```
GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_5);
```

返回值是 1(Bit_SET)或者 0(Bit_RESET);

ODR 是一个端口输出数据寄存器，也只用了低 16 位。该寄存器为可读写，从该寄存器读出来的数据可以用于判断当前 IO 口的输出状态。而向该寄存器写数据，则可以控制某个 IO 口的输出电平。该寄存器的各位描述如图 6.1.8 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0

rW rW

位31:16	保留, 始终读为0。
--------	------------

图 6.1.8 端口输出数据寄存器 ODR 各位描述

在固件库中设置 ODR 寄存器的值来控制 IO 口的输出状态是通过函数 GPIO_Write 来实现的:

```
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
```

该函数一般用来往一次性一个 GPIO 的多个端口设值。

BSRR 寄存器是端口位设置/清除寄存器。该寄存器和 ODR 寄存器具有类似的作用，都可以用来设置 GPIO 端口的输出位是 1 还是 0。下面我们看看该寄存器的描述如下图:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
位31:16	BRy: 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。														
位15:0	BSy: 设置端口x的位y (y = 0...15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 设置对应的ODRy位为1														

图 6.1.9 端口位设置/清除寄存器 BSRR 各位描述

该寄存器通过举例子可以很清楚了解它的使用方法。例如你要设置 GPIOA 的第 1 个端口值为 1，那么你只需要往寄存器 BSRR 的低 16 位对应位写 1 即可:

```
GPIOA->BSRR=1<<1;
```

如果你要设置 GPIOA 的第 1 个端口值为 0，你只需要往寄存器高 16 位对应位写 1 即可:

```
GPIOA->BSRR=1<<(16+1)
```

该寄存器往相应位写 0 是无影响的，所以我们要设置某些位，我们不用管其他位的值。

BRR 寄存器是端口位清除寄存器。该寄存器的作用跟 BSRR 的高 16 位雷同，这里就不做详细讲解。在 STM32 固件库中，通过 BSRR 和 BRR 寄存器设置 GPIO 端口输出是通过函数 GPIO_SetBits() 和函数 GPIO_ResetBits() 来完成的。

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

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

在多数情况下，我们都是采用这两个函数来设置 GPIO 端口的输入和输出状态。比如我们要设



置 GPIOB.5 输出 1，那么方法为：

```
GPIO_SetBits(GPIOB, GPIO_Pin_5);
```

反之如果要设置 GPIOB.5 输出位 0，方法为：

```
GPIO_ResetBits (GPIOB, GPIO_Pin_5);
```

GPIO 相关的函数我们先讲解到这里。虽然 IO 操作步骤很简单，这里我们还是做个概括性的总结，操作步骤为：

- 1) 使能 IO 口时钟。调用函数为 RCC_APB2PeriphClockCmd()。
- 2) 初始化 IO 参数。调用函数 GPIO_Init();
- 3) 操作 IO。操作 IO 的方法就是上面我们讲解的方法。

上面我们讲解了 STM32 IO 口的基本知识以及固件库操作 GPIO 的一些函数方法，下面我们将讲解我们的跑马灯实验的硬件和软件设计。

6.2 硬件设计

本章用到的硬件只有 LED (DS0 和 DS1)。其电路在 ALIENTEK 战舰 STM32 开发板上默认是已经连接好了的。DS0 接 PB5，DS1 接 PE5。所以在硬件上不需要动任何东西。其连接原理图如图 6.2.1 下：

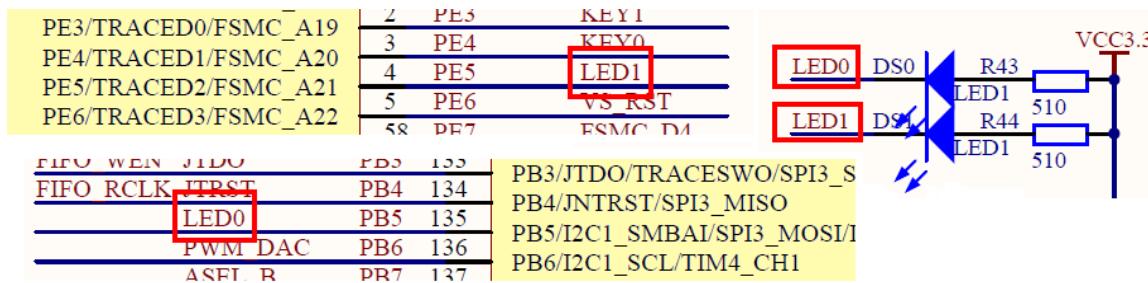


图 6.2.1 LED 与 STM32 连接原理图

6.3 软件设计

跑马灯实验我们主要用到的固件库文件是：

```
stm32f10x_gpio.c /stm32f10x_gpio.h
```

```
stm32f10x_rcc.c /stm32f10x_rcc.h
```

```
misc.c / misc.h
```

```
stm32f10x_usart /stm32f10x_usart.h
```

其中 `stm32f10x_rcc.h` 头文件在每个实验中都要引入，因为系统时钟配置函数以及相关的外设时钟使能函数都在这个源文件 `stm32f10x_rcc.c` 中。`stm32f10x_usart.h` 和 `misc.h` 头文件在我们 `SYSTEM` 文件夹中都需要使用到，所以每个实验都会引用。

在 `stm32f10x_conf.h` 文件里面，我们注释掉其他不用的头文件，只引入以下头文件：

```
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_usart.h"
#include "misc.h"
```

首先，找到之前 3.3 节新建的 Template 工程，在该文件夹下面新建一个 `HARDWARE` 的文件夹，用来存储以后与硬件相关的代码，然后在 `HARDWARE` 文件夹下新建一个 `LED` 文件夹，用来存放与 LED 相关的代码。如图 6.3.1 所示：



图 6.3.1 新建 HARDWARE 文件夹

然后我们打开 USER 文件夹下的 LED.Uv2 工程(如果是使用的上面新建的工程模板, 那么就是 Template.Uv2, 大家可以将其重命名为 LED.Uv2), 按 按钮新建一个文件, 然后保存在 HARDWARE->LED 文件夹下面, 保存为 led.c。在该文件中输入如下代码:

```
#include "led.h"
//初始化 PB5 和 PE5 为输出口.并使能这两个口的时钟
//LED IO 初始化
void LED_Init(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|
    RCC_APB2Periph_GPIOE, ENABLE);           //使能 PB,PE 端口时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;   //LED0-->PB.5 推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_SetBits(GPIOB,GPIO_Pin_5);           //PB.5 输出高
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;   //LED1-->PE.5 推挽输出
    GPIO_Init(GPIOE, &GPIO_InitStructure);
    GPIO_SetBits(GPIOE,GPIO_Pin_5);           //PE.5 输出高
}
```

该代码里面就包含了一个函数 void LED_Init(void), 该函数的功能就是用来实现配置 PB5 和 PE5 为推挽输出。这里需要注意的是: 在配置 STM32 外设的时候, 任何时候都要先使能该外设的时钟! GPIO 是挂载在 APB2 总线上的外设, 在固件库中对挂载在 APB2 总线上的外设时钟使能是通过函数 RCC_APB2PeriphClockCmd() 来实现的。对于这个入口参数设置, 在我们前面的“快速组织代码”章节已经讲解很清楚了。看看我们的代码:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|
```



```
RCC_APB2Periph_GPIOE, ENABLE); //使能 GPIOB,GPIOE 端口时钟
```

这行代码的作用是使能 APB2 总线上的 GPIOB 和 GPIOE 的时钟。

在配置完时钟之后，LED_Init 配置了 GPIOB.5 和 GPIOE.5 的模式为推挽输出，并且默认输出 1。这样就完成了对这两个 IO 口的初始化。函数代码是：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //LED0-->GPIOB.5 端口配置
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //IO 口速度为 50MHz
GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化 GPIOB.5
GPIO_SetBits(GPIOB, GPIO_Pin_5); //GPIOB.5 输出高

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //LED1-->GPIOE.5 推挽输出
GPIO_Init(GPIOE, &GPIO_InitStructure); //初始化 GPIOE.5
GPIO_SetBits(GPIOE, GPIO_Pin_5); //GPIOE.5 输出高
```

这里需要说明的是，因为 GPIOB 和 GPIOE 的 IO 口的初始化参数都是设置在结构体变量 GPIO_InitStructure 中，因为两个 IO 口的模式和速度都一样，所以我们只用初始化一次，在 GPIOE.5 的初始化的时候就不需要再重复初始化速度和模式了。最后一行代码：

```
GPIO_SetBits(GPIOE, GPIO_Pin_5);
```

的作用是在初始化中将 GPIOE.5 输出设置为高。

保存 led.c 代码，然后我们按同样的方法，新建一个 led.h 文件，也保存在 LED 文件夹下面。在 led.h 中输入如下代码：

```
#ifndef __LED_H
#define __LED_H
#include "sys.h"
//LED 端口定义
#define LED0 PBout(5)// DS0
#define LED1 PEout(5)// DS1
void LED_Init(void);//初始化
#endif
```

这段代码里面最关键就是 2 个宏定义：

```
#define LED0 PBout(5)// DS0
#define LED1 PEout(5)// DS1
```

这里使用的是位带操作来实现操作某个 IO 口的 1 个位的，关于位带操作前面第五章 5.2.1 已经有介绍，这里不再多说。需要说明的是，这里同样可以使用固件库操作来实现 IO 口操作。如下：

```
GPIO_SetBits(GPIOB, GPIO_Pin_5); //设置 GPIOB.5 输出 1,等同 LED0=1;
GPIO_ResetBits (GPIOB, GPIO_Pin_5); //设置 GPIOB.5 输出 0,等同 LED0=0;
```

有兴趣的朋友不妨修改我们的位带操作为库函数直接操作，这样也有利于学习。

将 led.h 也保存一下。接着，我们在 Manage Components 管理里面新建一个 HARDWARE 的组，并把 led.c 加入到这个组里面，如图 6.3.2 所示：

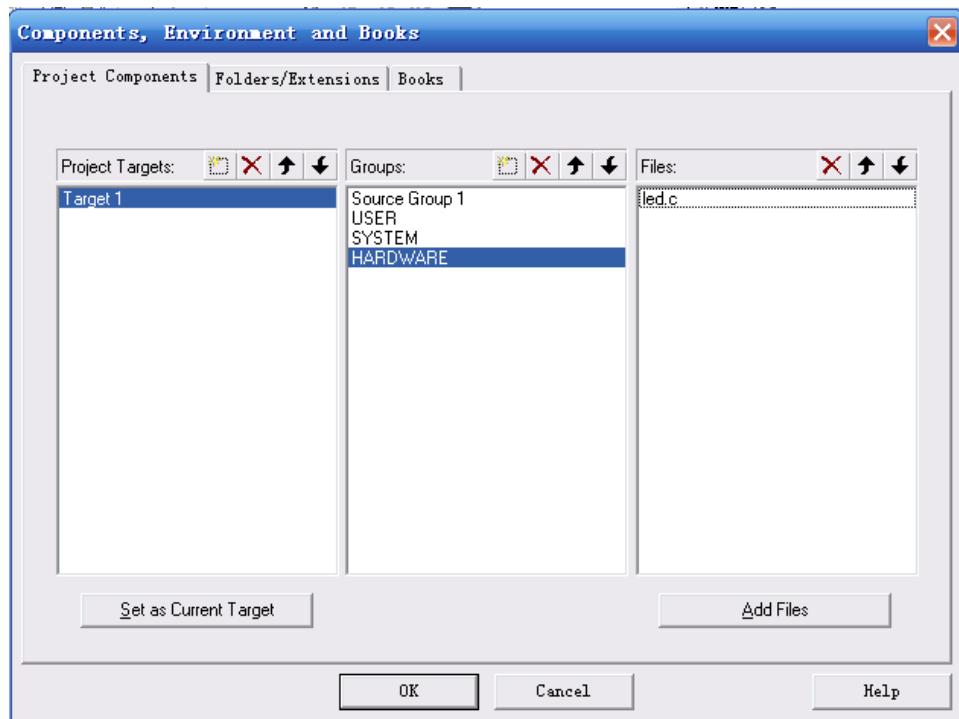


图 6.3.2 给工程新增 HARDWARE 组

单击 OK，回到工程，然后你会发现在 Project Workspace 里面多了一个 HARDWARE 的组，在改组下面有一个 led.c 的文件。如图 6.3.3 所示：

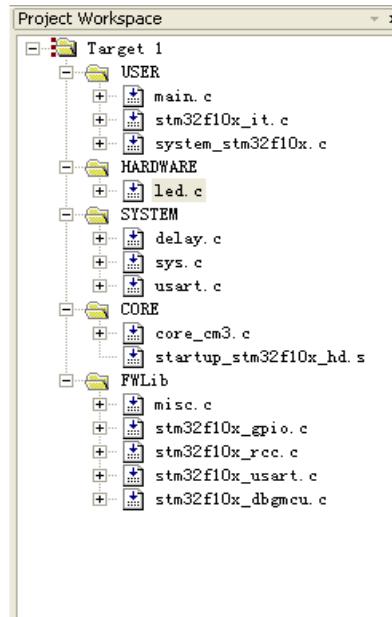


图 6.3.3 新增 HARDWARE 组

然后用之前介绍的方法（在 3.3.3 节介绍的）将 led.h 头文件的路径加入到工程里面。回到主界面，在 main 函数里面编写如下代码：

```
#include "led.h"
#include "delay.h"
#include "sys.h"
//ALIENTEK 战舰 STM32 开发板实验 1
```



```
//跑马灯实验
int main(void)
{
    delay_init();           //延时函数初始化
    LED_Init();             //初始化与 LED 连接的硬件接口
    while(1)
    {
        LED0=0;              //LED0=0;
        LED1=1;              //LED1=1;
        delay_ms(300);        //延时 300ms
        LED0=1;              //LED0=1;
        LED1=0;              //LED1=0;
        delay_ms(300);        //延时 300ms
    }
}
```

代码包含了#include "led.h"这句，使得 LED0、LED1、LED_Init 等能在 main()函数里被调用。这里我们需要重申的是，在固件库 V3.5 中，系统在启动的时候会调用 system_stm32f10x.c 中的函数 SystemInit()对系统时钟进行初始化，在时钟初始化完毕之后会调用 main()函数。所以我们不需要再在 main()函数中调用 SystemInit()函数。当然如果有需要重新设置时钟系统，可以写自己的时钟设置代码，SystemInit()只是将时钟系统初始化为默认状态。

main()函数非常简单，先调用 delay_init()初始化延时，接着就是调用 LED_Init()来初始化 GPIOB.5 和 GPIOE.5 为输出。最后在死循环里面实现 LED0 和 LED1 交替闪烁，间隔为 300ms。

上面是通过位带操作实现的 IO 操作，我们也可以修改 main()函数，直接通过库函数来操作 IO 达到同样的效果，大家不妨试试。

```
int main(void)
{
    delay_init();           //延时函数初始化
    LED_Init();             //初始化与 LED 连接的硬件接口
    while(1)
    {
        GPIO_ResetBits(GPIOB,GPIO_Pin_5);      // PB5 输出低,LED0=0;
        GPIO_SetBits(GPIOE,GPIO_Pin_5);          // PE5 输出高,LED1=1;
        delay_ms(300); //延时 300ms
        GPIO_SetBits(GPIOB,GPIO_Pin_5);          //PB5 输出高,LED0=1;
        GPIO_ResetBits(GPIOE,GPIO_Pin_5);         // PE5 输出低,LED1=0;
        delay_ms(300); //延时 300ms
    }
}
```

将主函数替换为上面代码，然后重新执行，可以看到，结果跟用位带操作一样的效果。大家可以对比一下。这个代码在我们光盘的实验代码“实验 1 跑马灯-库函数操作”中。

然后按 ，编译工程，得到结果如图 6.3.4 所示：



```

compiling stm32f10x_usart.c...
compiling stm32f10x_dbgmcu.c...
linking...
Program Size: Code=4472 RO-data=336 RW-data=52 ZI-data=1836
FromELF: creating hex file...
..\OBJ\LED.axf" - 0 Error(s), 0 Warning(s).

```

图 6.3.4 编译结果

可以看到没有错误，也没有警告。接下来，我们就先进行软件仿真，验证一下是否有错误的地方，然后下载到 Mini STM32 看看实际运行的结果。

6.4 仿真与下载

此代码，我们先进行软件仿真，看看结果对不对，根据软件仿真的结果，然后再下载到 ALIENTEK 战舰 STM32 板子上面看运行是否正确。

首先，我们进行软件仿真(请先确保 Options for Target→ Debug 选项卡里面已经设置为 Use Simulator)。先按 开始仿真，接着按 ，显示逻辑分析窗口，点击 Setup，新建两个信号 PORTB.5 和 PORTE.5，如图 6.4.1 所示：

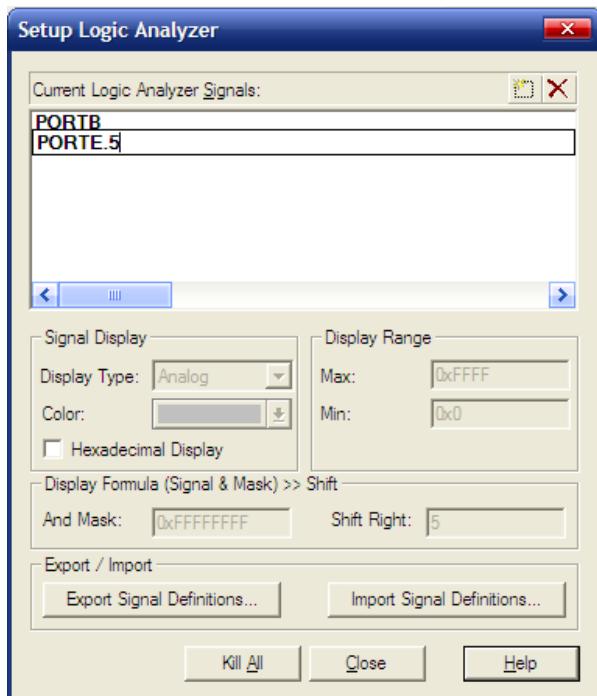


图 6.4.1 逻辑分析设置

Display Type 选择 bit，然后单击 Close 关闭该对话框，可以看到逻辑分析窗口出来了两个信号，如图 6.4.2 所示：

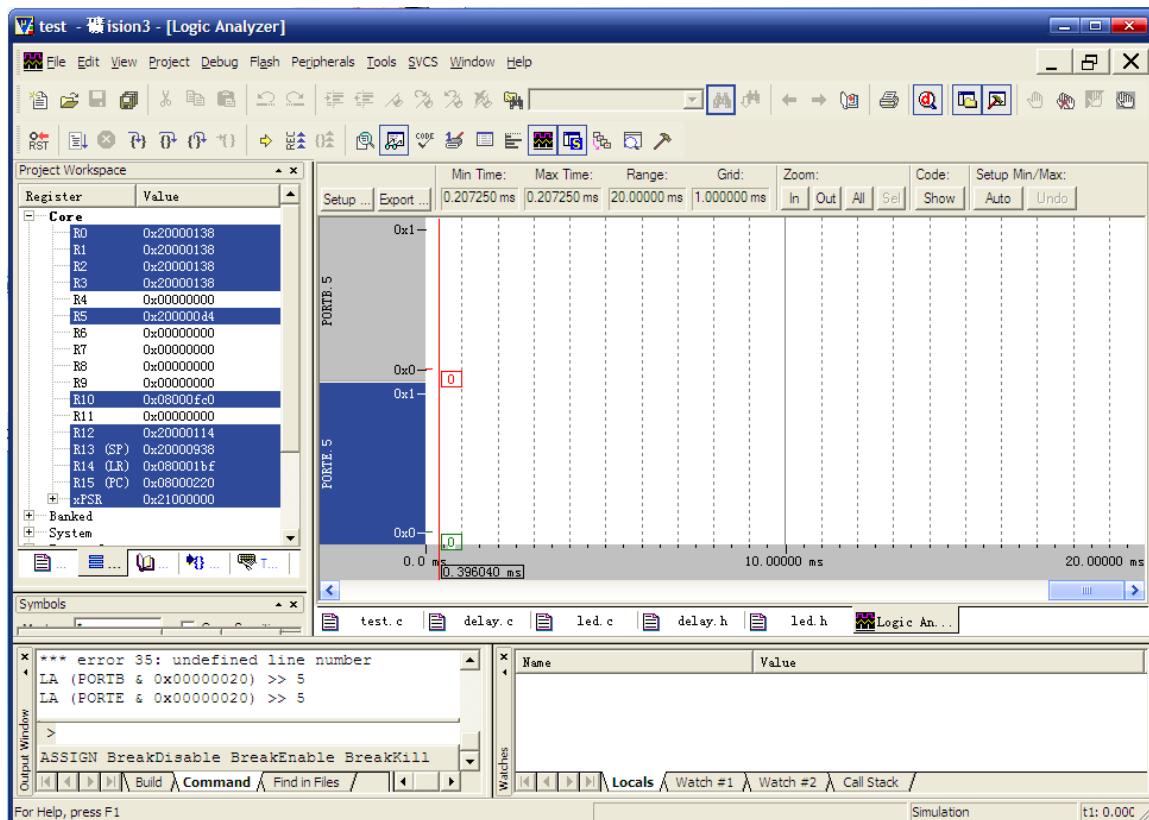


图 6.4.2 设置后的逻辑分析窗口

接着，点击 ，开始运行。运行一段时间之后，按 按钮，暂停仿真回到逻辑分析窗口，可以看到如图 6.4.3 所示的波形：

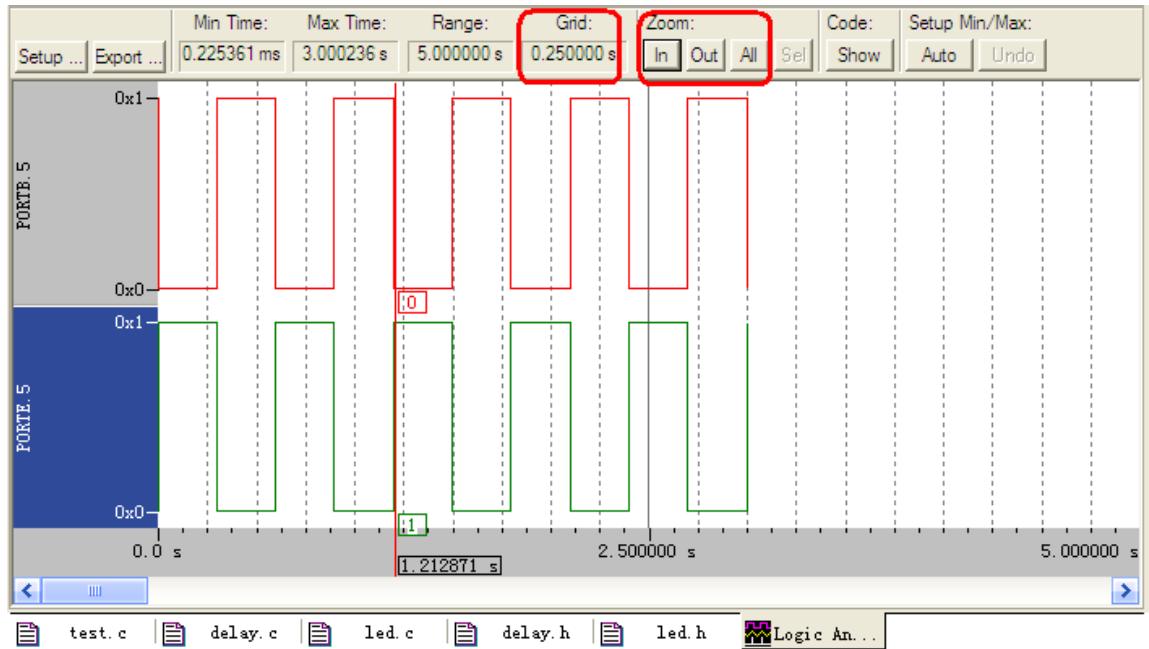


图 6.4.3 仿真波形

这里注意 Grid 要调节到 0.25s 左右比较合适，可以通过 Zoom 里面的 In 按钮来放大波形，通过 Out 按钮来缩小波形，或者按 All 显示全部波形。从上图中可以看到 PORTB.5 和 PORTE.5



交替输出，周期可以通过中间那根红线来测量。至此，我们的软件仿真已经顺利通过。

在软件仿真没有问题了之后，我们就可以把代码下载到开发板上，看看运行结果是否与我们仿真的一致。运行结果如图 6.4.4 所示：



图 6.4.4 执行结果

至此，我们的第一章的学习就结束了，本章作为 STM32 的入门第一个例子，详细介绍了 STM32 的 IO 口操作，同时巩固了前面的学习，并进一步介绍了 MDK 的软件仿真功能。希望大家好好理解一下。

第七章 蜂鸣器实验

上一章，我们介绍了 STM32 的 IO 口作为输出的使用，这一章，我们将通过另外一个例子讲述 STM32 的 IO 口作为输出的使用。在本章中，我们将利用一个 IO 口来控制板载的有源蜂鸣器，实现蜂鸣器控制。通过本章的学习，你将进一步了解 STM32 的 IO 口作为输出口使用的方法。本章分为如下几个小节：

- 7.1 蜂鸣器简介
- 7.2 硬件设计
- 7.3 软件设计
- 7.4 仿真与下载



7.1 蜂鸣器简介

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型。

战舰 STM32 开发板板载的蜂鸣器是电磁式的有源蜂鸣器，如图 7.1.1 所示：



图 7.1.1 有源蜂鸣器

这里的有源不是指电源的“源”，而是指有没有自带震荡电路，有源蜂鸣器自带了震荡电路，一通电就会发声；无源蜂鸣器则没有自带震荡电路，必须外部提供 2~5Khz 左右的方波驱动，才能发声。

前面我们已经对 STM32 的 IO 做了简单介绍，上一章，我们就是利用 STM32 的 IO 口直接驱动 LED 的，本章的蜂鸣器，我们能否直接用 STM32 的 IO 口驱动呢？让我们来分析下：STM32 的单个 IO 最大可以提供 25mA 电流（来自数据手册），而蜂鸣器的驱动电流是 30mA 左右，两者十分相近，但是全盘考虑，STM32 整个芯片的电流，最大也就 150mA，如果用 IO 口直接驱动蜂鸣器，其他地方用电就得省着点了…所以，我们不用 STM32 的 IO 直接驱动蜂鸣器，而是通过三极管扩流后再驱动蜂鸣器，这样 STM32 的 IO 只需要提供不到 1mA 的电流就足够了。

IO 口使用虽然简单，但是和外部电路的匹配设计，还是要十分讲究的，考虑越多，设计就越可靠，可能出现的问题也就越少。

本章将要实现的是控制 ALIENTEK 战舰 STM32 开发板上的蜂鸣器发出：“嘀”…“ 嘀”…的间隔声，进一步熟悉 STM32 IO 口的使用。

7.2 硬件设计

本章需要用到的硬件有：

- 1) 指示灯 DS0
- 2) 蜂鸣器

DS0 在上一章已有介绍，而蜂鸣器在硬件上也是直接连接好了的，不需要经过任何设置，直接编写代码就可以了。蜂鸣器的驱动信号连接在 STM32 的 PB8 上。如图 7.2.1 所示：

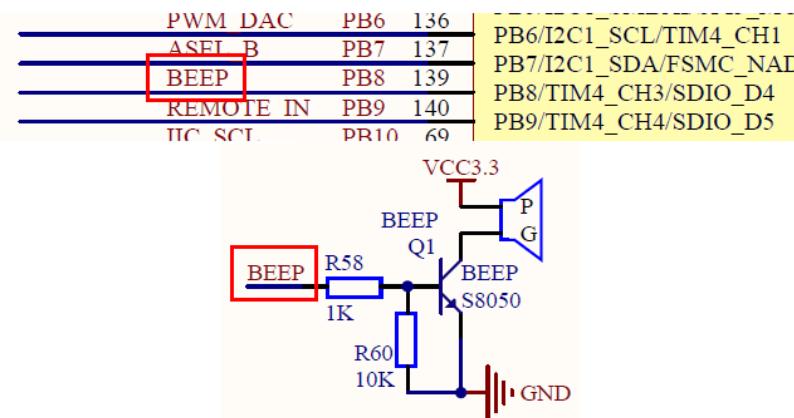


图 7.2.1 蜂鸣器与 STM32 连接原理图

图中我们用到一个 NPN 三极管（S8050）来驱动蜂鸣器，R60 主要用于防止蜂鸣器的误发声。当 PB.8 输出高电平的时候，蜂鸣器将发声，当 PB.8 输出低电平的时候，蜂鸣器停止发声。

7.3 软件设计

大家可以直接打开本实验工程。也可以按下面的步骤在实验 1 的基础上新建蜂鸣器实验工程。复制上一章的 LED 实验工程，然后打开 USER 目录，把目录下面工程 LED.Uv2 重命名为 BEEP.Uv2。, 然后在 HARDWARE 文件夹下新建一个 BEEP 文件夹，用来存放与蜂鸣器相关的代码。如图 7.3.1 所示：

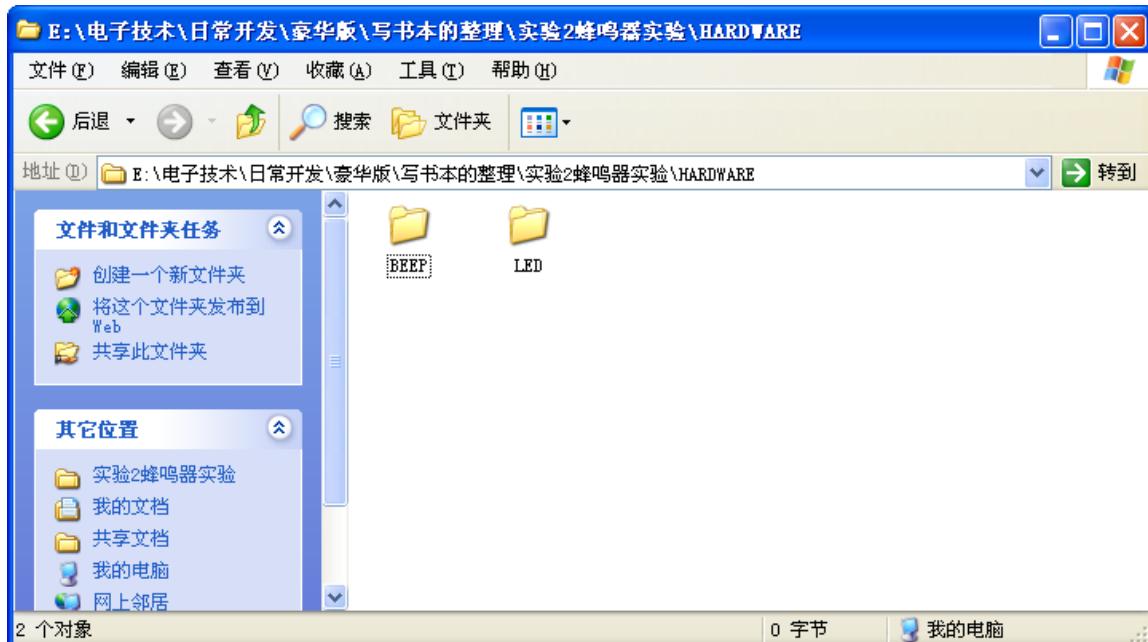


图 7.3.1 在 HARDWARE 下新增 BEEP 文件夹

然后我们打开 USER 文件夹下的 BEEP.Uv2 工程，按 按钮新建一个文件，然后保存在 HARDWARE→BEEP 文件夹下面，保存为 beep.c。在该文件中输入如下代码：

```
#include "beep.h"
```



```
//初始化 PB8 为输出口.并使能这个口的时钟
//LED IO 初始化
void BEEP_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //使能 GPIOB 端口时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8; //BEEP-->GPIOB.8 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度为 50MHz
    GPIO_Init(GPIOB, &GPIO_InitStructure); //根据参数初始化 GPIOB.8
    GPIO_ResetBits(GPIOB,GPIO_Pin_8); //输出 0, 关闭蜂鸣器输出
}
```

这段代码 仅包含 1 个函数: void BEEP_Init(void), 该函数的作用就是使能 PORTB 的时钟, 同时配置 PB8 为推挽输出。这里的初始化内容跟实验 6 跑马灯实验几乎是一样的, 这里就不做深入的讲解。

保存 beep.c 代码, 然后我们按同样的方法, 新建一个 beep.h 文件, 也保存在 BEEP 文件夹下面。在 beep.h 中输入如下代码:

```
#ifndef __BEEP_H
#define __BEEP_H
#include "sys.h"
//蜂鸣器端口定义
#define BEEP_PBout(8) // BEEP,蜂鸣器接口
void BEEP_Init(void); //初始化
#endif
```

和上一章一样, 我们这里还是通过位带操作来实现某个 IO 口的输出控制, BEEP 就直接代表了 PB8 的输出状态。我们只需要令 BEEP=1, 就可以让蜂鸣器发声。

将 beep.h 也保存。接着, 我们把 beep.c 加入到 HARDWARE 这个组里面, 这一次我们通过双击的方式来增加新的.c 文件, 双击 HARDWARE, 找到 beep.c, 加入到 HARDWARE 里面, 如图 7.3.2 所示:

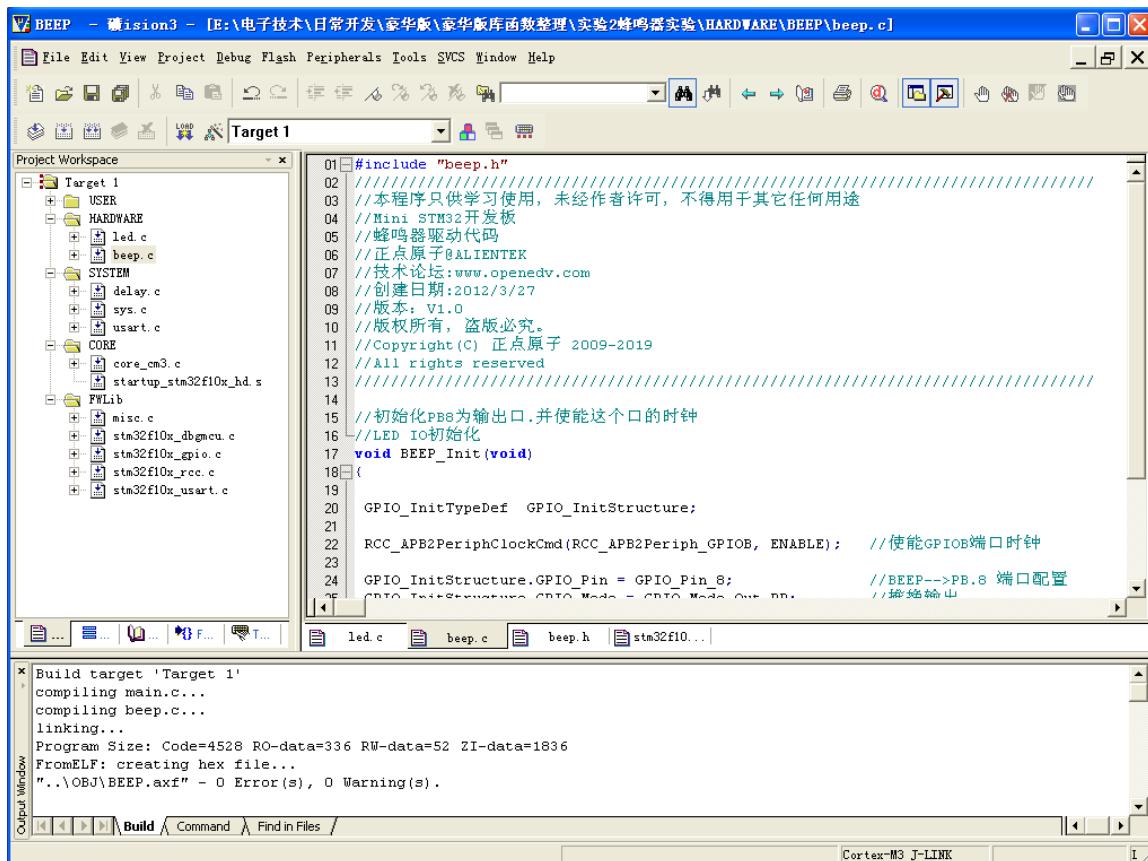


图 7.3.2 将 beep.c 加入 HARDWARE 组下

可以看到 HARDWARE 文件夹里面多了一个 beep.c 的文件，然后还是用老办法把 beep.h 头文件所在的路径加入到工程里面。回到主界面，在 main.c 里面编写如下代码：

```
#include "sys.h"
#include "delay.h"
#include "led.h"
#include "beep.h"

//ALIENTEK 战舰 STM32 开发板实验 2
//蜂鸣器实验

int main(void)
{
    delay_init();           //延时函数初始化
    LED_Init();             //初始化与 LED 连接的硬件接口
    BEEP_Init();            //初始化蜂鸣器端口

    while(1)
    {
        LED0=0;
        BEEP=0;
        delay_ms(300);
        LED0=1;
        BEEP=1;
        delay_ms(300);
    }
}
```



}

注意要将 BEEP 文件夹加入头文件包含路径，不能少，否则编译的时候会报错。这段代码就实现前面 7.1 节所阐述的功能，同时加入了 DS0 (LED0) 的闪烁来提示程序运行（后面的代码，我们基本都会加入这个），整个代码比较简单。

然后按 ，编译工程，得到结果如图 7.3.3 所示：

```
Build target 'Target 1'  
compiling main.c...  
compiling beep.c...  
linking...  
Program Size: Code=4528 RO-data=336 RW-data=52 ZI-data=1836  
FromELF: creating hex file...  
"..\OBJ\BEEP.axf" - 0 Error(s), 0 Warning(s).
```

图 7.3.3 编译结果

可以看到没有错误，也没有警告。从编译信息可以看出，我们的代码占用 FLASH 大小为：4864 字节 (4528+336)，所用的 SRAM 大小为：1888 个字节 (52+1836)。

这里我们解释一下，编译结果里面的几个数据的意义：

Code: 表示程序所占用 FLASH 的大小 (FLASH)。

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

RW-data: 即 Read Write-data，表示已被初始化的全局变量 (SRAM)

ZI-data: 即 Zero Init-data，表示未被初始化的全局变量(SRAM)

有了这个就可以知道你当前使用的 flash 和 sram 大小了，所以，一定要注意的是程序的大小不是.hex 文件的大小，而是编译后的 Code 和 RO-data 之和。

接下来，我们还是先进行软件仿真，验证一下是否有错误的地方，然后才下载到战舰 STM32 开发板看看实际运行的结果。

7.4 仿真与下载

我们可以先用软件仿真，看看结果对不对，根据软件仿真的结果，然后再下载到战舰 STM32 开发板上面看运行是否正确。

首先，我们进行软件仿真。先按 开始仿真，接着按 ，显示逻辑分析窗口，点击 Setup，新建 2 个信号：PORTB.5 和 PORTB.8，如图 7.4.1 所示：

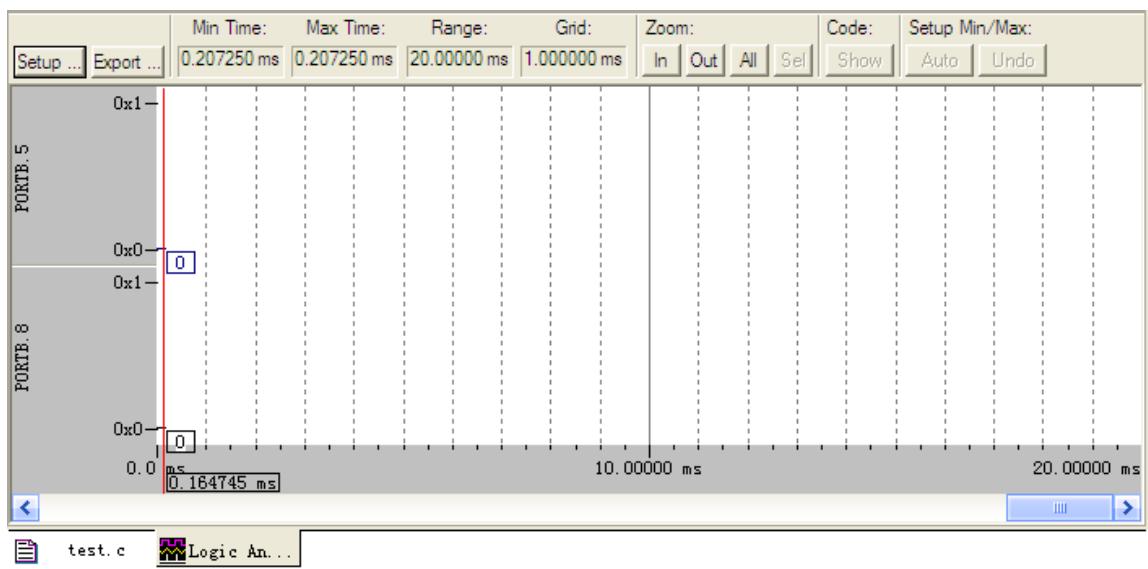


图 7.4.1 新建仿真信号

接着，点击 ，开始运行。运行一段时间之后，按 按钮，暂停仿真回到逻辑分析窗口，可以看到如图 7.4.2 所示的波形：

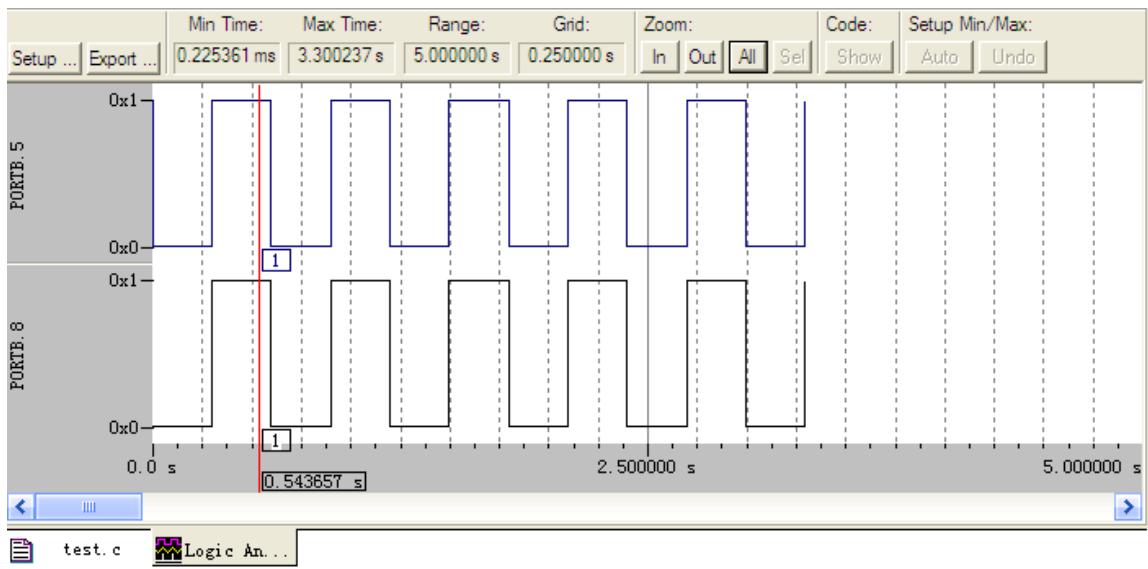


图 7.4.2 仿真波形

从图中我们可以看出，PB.5 和 PB.8 同时输出高和低电平，和我们预计的效果是一样的，周期是 0.6s，符合预期设计。接下来可以把代码下载到战舰 STM32 开发板上看看运行结果是否正确。

下载完代码，可以看到 DS0 亮的时候蜂鸣器不叫，而 DS0 灭的时候，蜂鸣器叫（因为他们的有效信号相反）。间隔为 0.3 秒左右，符合预期设计。

至此，我们的本章的学习就结束了。本章，作为 STM32 的入门第二个例子，进一步介绍了 STM32 的 IO 作为输出口的使用方法，同时巩固了前面的学习。希望大家在开发板上实际验证一下，从而加深印象。

第八章 按键输入实验

上两章，我们介绍了 STM32 的 IO 口作为输出的使用，这一章，我们将向大家介绍如何使用 STM32 的 IO 口作为输入用。在本章中，我们将利用板载的 4 个按键，来控制板载的两个 LED 的亮灭。通过本章的学习，你将了解到 STM32 的 IO 口作为输入口的使用方法。本章分为如下几个小节：

- 8.1 STM32 IO 口简介
- 8.2 硬件设计
- 8.3 软件设计
- 8.4 仿真与下载



8.1 STM32 IO 口简介

STM32 的 IO 口在上一章已经有了比较详细的介绍，这里我们不再多说。STM32 的 IO 口做输入使用的时候，是通过调用函数 GPIO_ReadInputDataBit() 来读取 IO 口的状态的。了解了这点，就可以开始我们的代码编写了。

这一章，我们将通过 ALIENTEK 战舰 STM32 开发板上载有的 4 个按钮（WK_UP、KEY0、KEY1 和 KEY2），来控制板上的 2 个 LED（DS0 和 DS1）和蜂鸣器，其中 WK_UP 控制蜂鸣器，按一次叫，再按一次停；KEY2 控制 DS0，按一次亮，再按一次灭；KEY1 控制 DS1，效果同 KEY2；KEY0 则同时控制 DS0 和 DS1，按一次，他们的状态就翻转一次。

8.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0、DS1
- 2) 蜂鸣器
- 3) 4 个按键：KEY0、KEY1、KEY2、和 KEY_UP。

DS0、DS1 以及蜂鸣器和 STM32 的连接在上两章都已经分别介绍了，在战舰 STM32 开发板上的按键 KEY0 连接在 PE4 上、KEY1 连接在 PE3 上、KEY2 连接在 PE2 上、WK_UP 连接在 PA0 上。如图 8.2.1 所示：

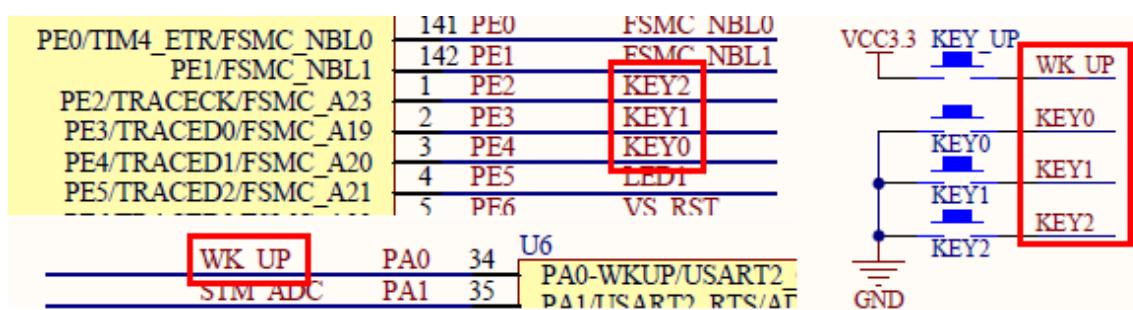


图 8.2.1 按键与 STM32 连接原理图

这里需要注意的是：KEY0、KEY1 和 KEY2 是低电平有效的，而 WK_UP 是高电平有效的，并且外部都没有上下拉电阻，所以，需要在 STM32 内部设置上下拉。

8.3 软件设计

从这章开始，我们的软件设计主要是通过直接打开我们光盘的实验工程，而不再讲解怎么加入文件和头文件目录。

打开我们的按键实验工程可以看到，我们引入了 key.c 文件以及头文件 key.h。下面我们首先打开 key.c 文件，代码如下：

```
#include "key.h"
#include "sys.h"
#include "delay.h"
//按键初始化函数
void KEY_Init(void) //IO 初始化
{
    GPIO_InitTypeDef GPIO_InitStructure;
```



```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|  
RCC_APB2Periph_GPIOE,ENABLE);           //使能 PORTA,PORTE 时钟  
  
GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4;//GPIOE.2~4  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;    //设置成上拉输入  
GPIO_Init(GPIOE, &GPIO_InitStructure);          //初始化 GPIOE2,3,4  
  
GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_0;        //初始化 WK_UP-->GPIOA.0  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;    //PA0 设置成输入，下拉  
GPIO_Init(GPIOA, &GPIO_InitStructure);          //初始化 GPIOA.0  
}  
//按键处理函数  
//返回按键值  
//mode:0,不支持连续按;1,支持连续按;  
//0,没有任何按键按下;1, KEY0 按下;2, KEY1 按下;3, KEY2 按下 ;4, KEY3 按下 WK_UP  
//注意此函数有响应优先级,KEY0>KEY1>KEY2>KEY3!!  
u8 KEY_Scan(u8 mode)  
{  
    static u8 key_up=1;                      //按键按松开标志  
    if(mode)key_up=1;                        //支持连接  
    if(key_up&&(KEY0==0||KEY1==0||KEY2==0||KEY3==1))  
    {  
        delay_ms(10);                      //去抖动  
        key_up=0;  
        if(KEY0==0)return KEY_RIGHT;  
        else if(KEY1==0)return KEY_DOWN;  
        else if(KEY2==0)return KEY_LEFT;  
        else if(KEY3==1)return KEY_UP;  
    }else if(KEY0==1&&KEY1==1&&KEY2==1&&KEY3==0)key_up=1;  
    return 0;                                // 无按键按下  
}
```

这段代码包含 2 个函数，void KEY_Init(void) 和 u8 KEY_Scan(u8 mode)，KEY_Init() 是用来初始化按键输入的 IO 口的。首先使能 GPIOA 和 GPIOE 时钟，然后实现 PA0、PE2~4 的输入设置，这里和第六章的输出配置差不多，只是这里用来设置成的是输入而第六章是输出。

KEY_Scan() 函数，则是用来扫描这 4 个 IO 口是否有按键按下。KEY_Scan() 函数，支持两种扫描方式，通过 mode 参数来设置。

当 mode 为 0 的时候，KEY_Scan() 函数将不支持连续按，扫描某个按键，该按键按下之后必须要松开，才能第二次触发，否则不会再响应这个按键，这样的好处就是可以防止按一次多次触发，而坏处就是在需要长按的时候比较不合适。

当 mode 为 1 的时候，KEY_Scan() 函数将支持连续按，如果某个按键一直按下，则会一直返回这个按键的键值，这样可以方便的实现长按检测。

有了 mode 这个参数，大家就可以根据自己的需要，选择不同的方式。这里要提醒大家，因为该函数里面有 static 变量，所以该函数不是一个可重入函数，在有 OS 的情况下，这个大家



要留意下。同时还有一点要注意的就是，该函数的按键扫描是有优先级的，最优先的是 KEY0，第二优先的是 KEY1，接着 KEY2，最后是 KEY3 (KEY3 对应 WK_UP 按键)。该函数有返回值，如果有按键按下，则返回非 0 值，如果没有或者按键不正确，则返回 0。

接下来我们看看头文件 key.h 里面的代码：

```
#ifndef __KEY_H
#define __KEY_H
#include "sys.h"

#define KEY0  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_4)    //读取按键 0
#define KEY1  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3)    //读取按键 1
#define KEY2  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2)    //读取按键 2
#define KEY3  GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)    //读取按键 3(WK_UP)

#define KEY_UP      4
#define KEY_LEFT    3
#define KEY_DOWN   2
#define KEY_RIGHT  1
void KEY_Init(void);                                //IO 初始化
u8 KEY_Scan(u8);                                    //按键扫描函数
#endif
```

这段代码里面最关键就是 4 个宏定义：

```
#define KEY0  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_4)    //读取按键 0
#define KEY1  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3)    //读取按键 1
#define KEY2  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2)    //读取按键 2
#define KEY3  GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)    //读取按键 3(WK_UP)
```

前面两个实验用的是位带操作实现设定某个 IO 口的位。这里我们采取的是库函数的读取 IO 口的值。当然，上面的功能也同样可以通过位带操作来简单的实现：

```
#define KEY0 PEin(4)    //PE4
#define KEY1 PEin(3)    //PE3
#define KEY2 PEin(2)    //PE2
#define KEY3 PAin(0)    //PA0  WK_UP
```

用库函数实现的好处是在各个 STM32 芯片上面的移植性非常好，不需要修改任何代码。用位带操作的好处是简洁，至于使用哪种方法，看各位的爱好了。

在 key.h 中，我们还定义了 KEY_UP/KEY_DOWN /KEY_LEFT /KEY_RIGHT 等 4 个宏定义，分别对应开发板上下左右 (WK_UP/KEY1/KEY2/KEY0) 按键按下时 KEY_Scan() 返回的值。这些宏定义的方向直接和开发板的按键排列方式相同，方便大家使用。

最后，我们看看 main.c 里面编写的主函数代码如下：

```
#include "led.h"
#include "delay.h"
#include "key.h"
#include "sys.h"
#include "beep.h"
//ALIENTEK 战舰 STM32 开发板实验 3
//按键输入实验
int main(void)
```



```
{  
    u8 t;  
    delay_init();           //延时函数初始化  
    LED_Init();            //LED 端口初始化  
    KEY_Init();            //初始化与按键连接的硬件接口  
    BEEP_Init();           //初始化蜂鸣器端口  
    LED0=0;                //先点亮红灯  
    while(1)  
    {  
        t=KEY_Scan(0);      //得到键值  
        if(t)  
        {  
            switch(t)  
            {  
                case KEY_UP:          //控制蜂鸣器  
                    BEEP=!BEEP;break;  
                case KEY_LEFT:         //控制 LED0 翻转  
                    LED0=!LED0;break;  
                case KEY_DOWN:         //控制 LED1 翻转  
                    LED1=!LED1;break;  
                case KEY_RIGHT:        //同时控制 LED0,LED1 翻转  
                    LED0=!LED0;LED1=!LED1;break;  
            }  
        }else delay_ms(10);  
    }  
}
```

主函数代码比较简单，先进行一系列的初始化操作，然后在死循环中调用按键扫描函数 KEY_Scan()扫描按键值，最后根据按键值控制 LED 和蜂鸣器的翻转。

8.4 仿真与下载

我们可以先用软件仿真，看看结果对不对，根据软件仿真的结果，然后再下载到战舰 STM32 板子上面看运行是否正确。

首先，我们进行软件仿真。先按 开始仿真，接着按 ，显示逻辑分析窗口，点击 Setup，新建 7 个信号 PORTB.5、PORTE.5、PORTB.8、PORTA.0、PORTE.2、PORTE.3、PORTE.4，如图 8.4.1 所示：

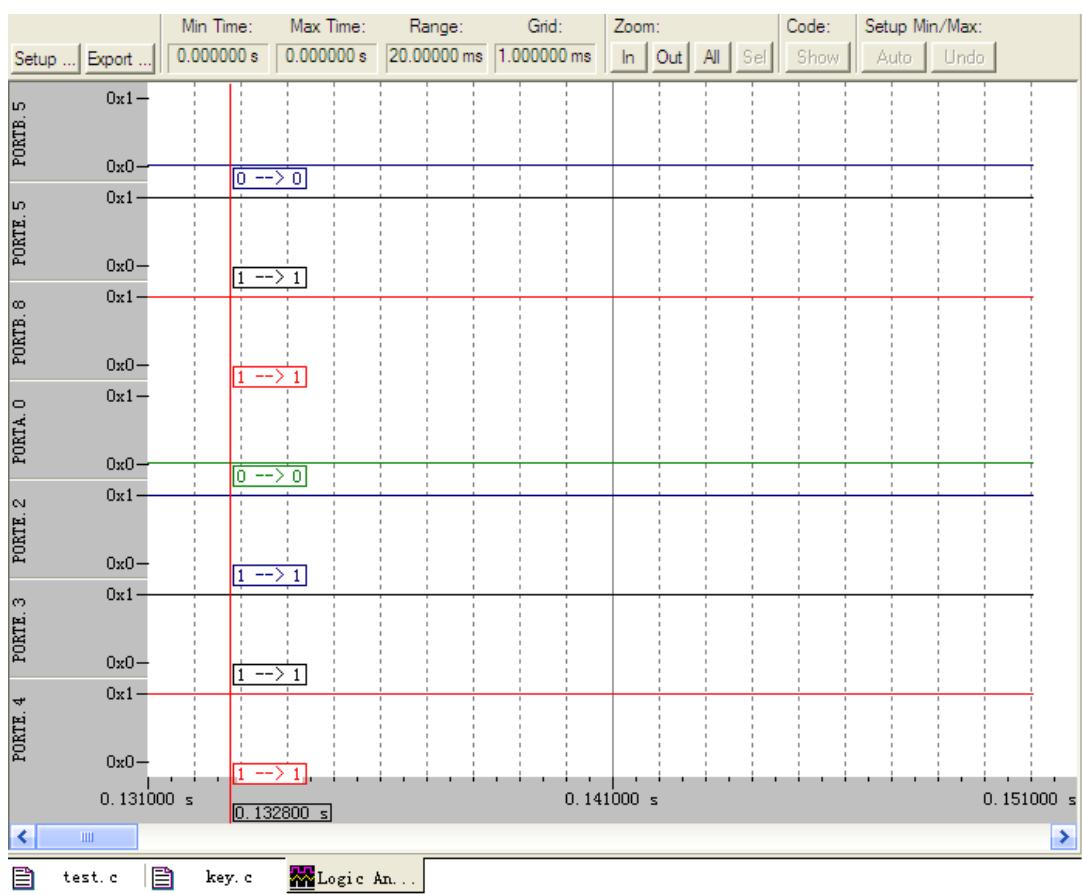


图 8.4.1 新建仿真信号

然后再点击 Peripherals → General Purpose I/O → GPIOE，弹出 GPIOE 的查看窗口，如图 8.4.2 所示：

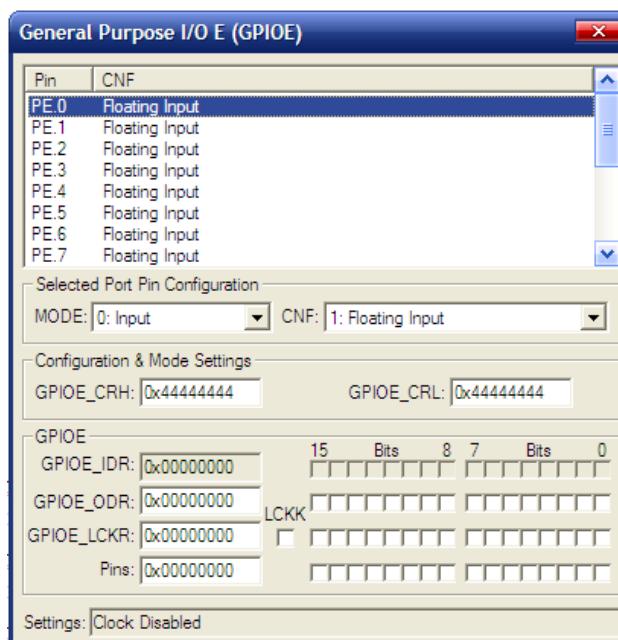


图 8.4.2 查看 GPIOE 寄存器



然后在 `t=KEY_Scan();` 这里设置一个断点，按 直接执行到这里，然后在 General Purpose I/O E 窗口内的 Pins 里面勾选 2、3、4 位，这是虽然我们已经设置了这几个 IO 口为上拉输入，但是 MDK 不会考虑 STM32 自带的上拉和下拉，所以我们得自己手动设置一下，来使得其初始状态和外部硬件的状态一摸一样。如图 8.4.3 所示：

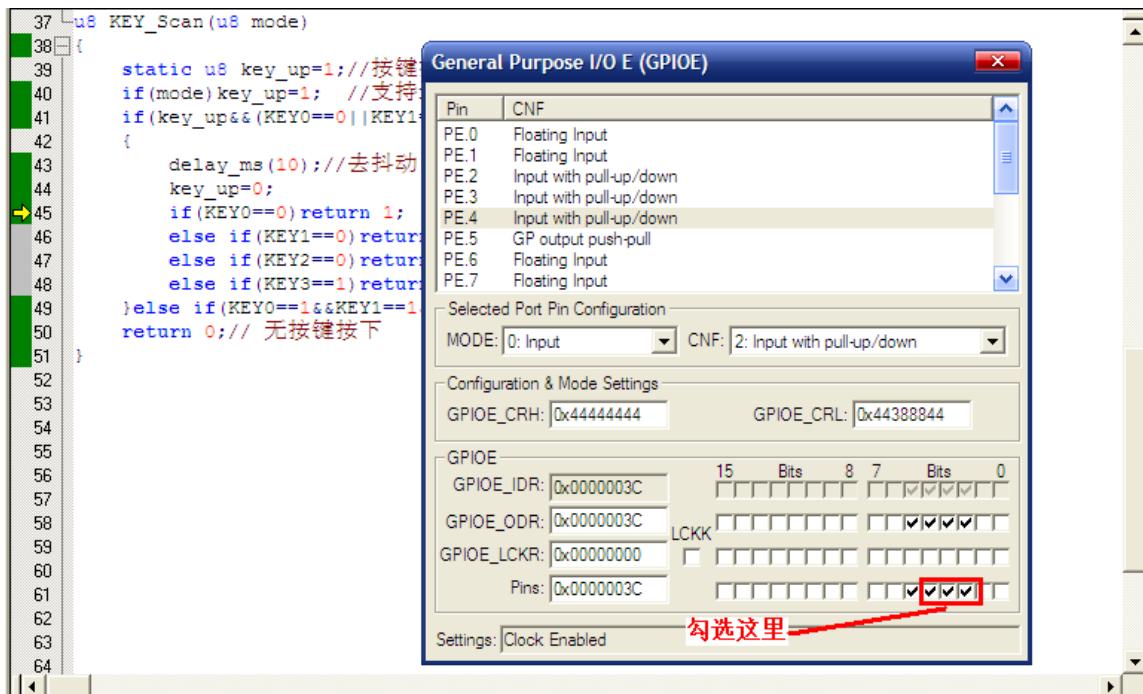


图 8.4.3 执行到断点处

本来我们还需要设置 PORTA.0 的，但是 GPIOA.0 是高电平有效，刚好默认的就满足要求，不需要再去勾选 PORTA.0 了。所以这里我们可以省略一个 GPIOA.0 的设置。接着我们执行过这句，可以看到 `t` 的值依旧为 0，也就是没有任何按键按下。接着我们再按 ，再次执行到 `t=KEY_Scan();`；我们此次把 Pins 的 PE2 取消勾选，再次执行过这句，得到 `t` 的值为 3，如图 8.4.4 所示：

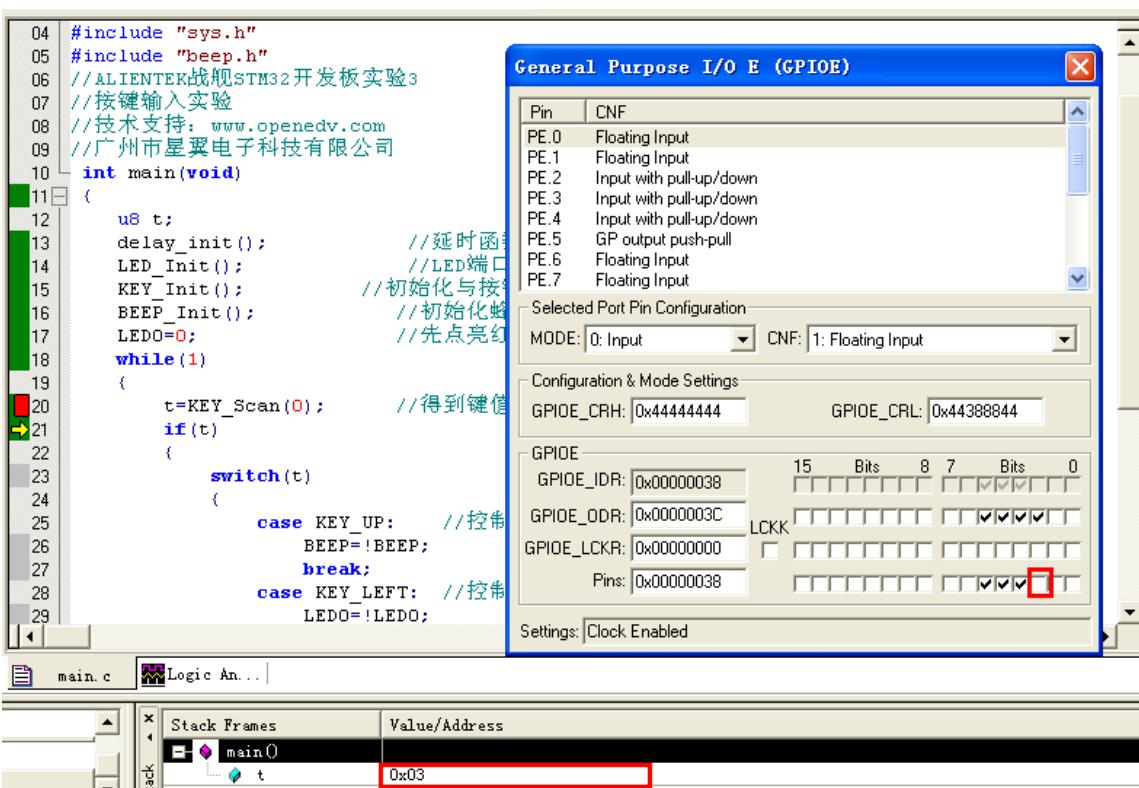


图 8.4.4 按键扫描结果

然后按相似的方法，分别取消勾选 PE3 和 PE4，以及勾选 PA0，然后再把它们还原，可以看到逻辑分析窗口的波形如图 8.4.5 所示：

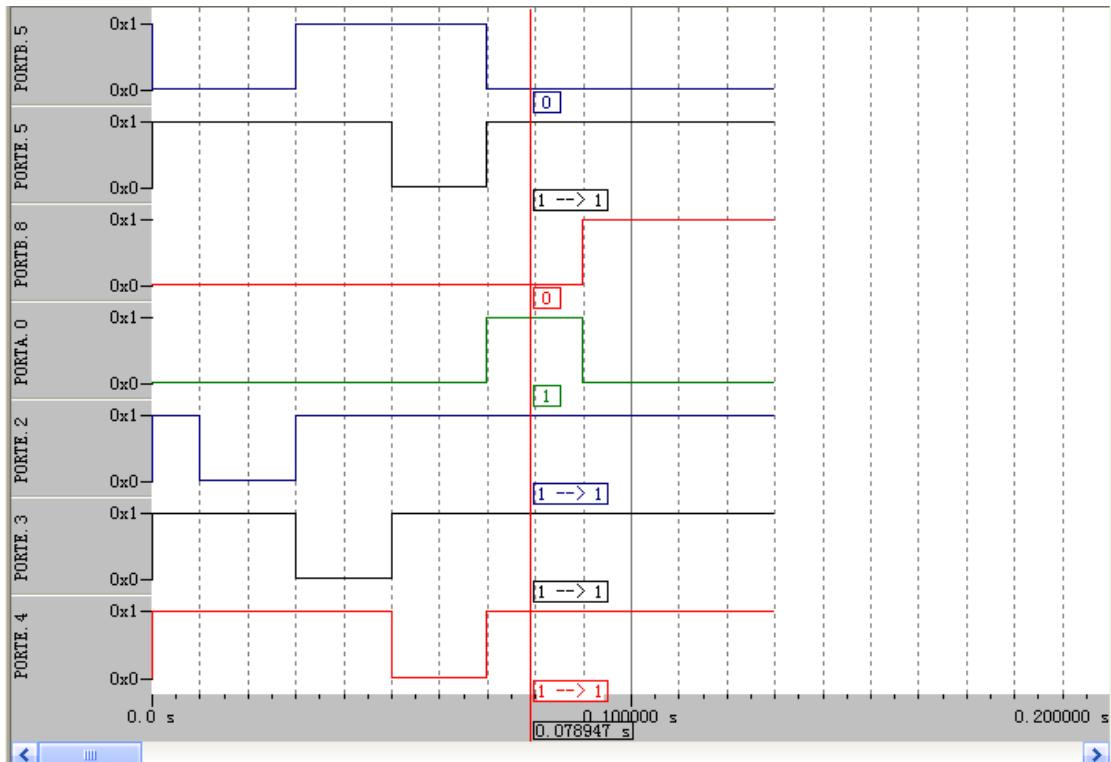


图 8.4.5 仿真波形

从图 8.4.5 可以看出，当 PE2 按下的时候 PB5 翻转，PE3 按下的时候 PE5 翻转，PE4 按下

的时候 PB5 和 PE5 一起翻转，PA0 按下的时候 PB8 翻转，使我们想要得到的结果。因此，可以确定软件仿真基本没有问题了。接下来可以把代码下载到 STM32 开发板上看看运行结果是否正确。

在下载完之后，我们可以按 KEY0、KEY1、KEY2 和 WK_UP 来看看 DS0 和 DS1 以及蜂鸣器的变化，是否和我们仿真的结果一致（结果肯定是一致的）。

至此，我们的本章的学习就结束了。本章，作为 STM32 的入门第三个例子，介绍了 STM32 的 IO 作为输入的使用方法，同时巩固了前面的学习。希望大家在开发板上实际验证一下，从而加深印象。

第九章 串口实验

前面两章介绍了 STM32 的 IO 口操作。这一章我们将学习 STM32 的串口，教大家如何使用 STM32 的串口来发送和接收数据。本章将实现如下功能：STM32 通过串口和上位机的对话，STM32 在收到上位机发过来的字符串后，原原本本的返回给上位机。本章分为如下几个小节：

9.1 STM32 串口简介

9.2 硬件设计

9.3 软件设计

9.4 下载验证



9.1 STM32 串口简介

串口作为 MCU 的重要外部接口，同时也是软件开发重要的调试手段，其重要性不言而喻。现在基本上所有的 MCU 都会带有串口，STM32 自然也不例外。

STM32 的串口资源相当丰富的，功能也相当强劲。ALIENTEK 战舰 STM32 开发板所使用的 STM32F103ZET6 最多可提供 5 路串口，有分数波特率发生器、支持同步单线通信和半双工单线通讯、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范、具有 DMA 等。

5.3 节对串口有过简单的介绍，大家看这个实验的时候记得翻过去看看。接下来我们将主要从库函数操作层面结合寄存器的描述，告诉你如何设置串口，以达到我们最基本的通信功能。本章，我们将实现利用串口 1 不停的打印信息到电脑上，同时接收从串口发过来的数据，把发送过来的数据直接送回给电脑。战舰 STM32 开发板板载了 1 个 USB 串口和 1 个 RS232 串口，我们本章介绍的是通过 USB 串口和电脑通信。

在 4.4.1 章节端口复用功能已经讲解过，对于复用功能的 IO，我们首先要使能 GPIO 时钟，然后使能复用功能时钟，同时要把 GPIO 模式设置为复用功能对应的模式（这个可以查看手册《STM32 中文参考手册 V10》P110 的表格“8. 1. 11 外设的 GPIO 配置”）。这些准备工作做完之后，剩下的当然是串口参数的初始化设置，包括波特率，停止位等等参数。在设置完成只能接下来就是使能串口，这很容易理解。同时，如果我们开启了串口的中断，当然要初始化 NVIC 设置中断优先级别，最后编写中断服务函数。

串口设置的一般步骤可以总结为如下几个步骤：

- 1) 串口时钟使能，GPIO 时钟使能
- 2) 串口复位
- 3) GPIO 端口模式设置
- 4) 串口参数初始化
- 5) 开启中断并且初始化 NVIC（如果需要开启中断才需要这个步骤）
- 6) 使能串口
- 7) 编写中断处理函数

下面，我们就简单介绍下这几个与串口基本配置直接相关的几个固件库函数。这些函数和定义主要分布在 `stm32f10x_usart.h` 和 `stm32f10x_usart.c` 文件中。

1.串口时钟使能。串口是挂载在 APB2 下面的外设，所以使能函数为：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1);
```

2.串口复位。当外设出现异常的时候可以通过复位设置，实现该外设的复位，然后重新配置这个外设达到让其重新工作的目的。一般在系统刚开始配置外设的时候，都会先执行复位该外设的操作。复位的是在函数 `USART_DeInit()` 中完成：

```
void USART_DeInit(USART_TypeDef* USARTx); //串口复位
```

比如我们要复位串口 1，方法为：

```
USART_DeInit(USART1); //复位串口 1
```

3.串口参数初始化。串口初始化是通过 `USART_Init()` 函数实现的，

```
void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct);
```

这个函数的第一个入口参数是指定初始化的串口标号，这里选择 `USART1`。

第二个入口参数是一个 `USART_InitTypeDef` 类型的结构体指针，这个结构体指针的成员变量用来设置串口的一些参数。一般的实现格式为：

```
USART_InitStruct USART_BaudRate = bound; //一般设置为 9600;
```



```

USART_InitStructure USART_WordLength = USART_WordLength_8b; //字长为8位数据格式
USART_InitStructure USART_StopBits = USART_StopBits_1;           //一个停止位
USART_InitStructure USART_Parity = USART_Parity_No;             //无奇偶校验位
USART_InitStructure USART_HardwareFlowControl
    = USART_HardwareFlowControl_None;                         //无硬件数据流控制
USART_InitStructure USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式
USART_Init(USART1, &USART_InitStructure);                      //初始化串口

```

从上面的初始化格式可以看出初始化需要设置的参数为：波特率，字长，停止位，奇偶校验位，硬件数据流控制，模式（收，发）。我们可以根据需要设置这些参数。

4.数据发送与接收。 STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和 RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到数据的时候，也是存在该寄存器内。

STM32 库函数操作 USART_DR 寄存器发送数据的函数是：

```
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);
```

通过该函数向串口寄存器 USART_DR 写入一个数据。

STM32 库函数操作 USART_DR 寄存器读取串口接收到的数据的函数是：

```
uint16_t USART_ReceiveData(USART_TypeDef* USARTx);
```

通过该函数可以读取串口接受到的数据。

5.串口状态。 串口的状态可以通过状态寄存器 USART_SR 读取。USART_SR 的各位描述如图 9.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留					CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE	
rc w0					r	rc w0	rc w0	r	rc w0	r	r	r	r	r	r

图 9.1.1 USART_SR 寄存器各位描述

这里我们关注一下两个位，第 5、6 位 RXNE 和 TC。

RXNE（读数据寄存器非空），当该位被置 1 的时候，就是提示已经有数据被接收到了，并且可以读出来了。这时候我们要做的就是尽快去读取 USART_DR，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。

TC（发送完成），当该位被置位的时候，表示 USART_DR 内的数据已经被发送完成了。如果设置了这个位的中断，则会产生中断。该位也有两种清零方式：1）读 USART_SR，写 USART_DR。2）直接向该位写 0。

状态寄存器的其他位我们这里就不做过多讲解，大家需要可以查看中文参考手册。

在我们固件库函数里面，读取串口状态的函数是：

```
FlagStatus USART_GetFlagStatus(USART_TypeDef* USARTx, uint16_t USART_FLAG);
```

这个函数的第二个入口参数非常关键，它是标示我们要查看串口的哪种状态，比如上面讲解的 RXNE(读数据寄存器非空)以及 TC(发送完成)。例如我们要判断读寄存器是否非空(RXNE)，操作库函数的方法是：

```
USART_GetFlagStatus(USART1, USART_FLAG_RXNE);
```

我们要判断发送是否完成(TC)，操作库函数的方法是：

```
USART_GetFlagStatus(USART1, USART_FLAG_TC);
```



这些标识号在 MDK 里面是通过宏定义定义的：

#define USART_IT_PE	((uint16_t)0x0028)
#define USART_IT_TXE	((uint16_t)0x0727)
#define USART_IT_TC	((uint16_t)0x0626)
#define USART_IT_RXNE	((uint16_t)0x0525)
#define USART_IT_IDLE	((uint16_t)0x0424)
#define USART_IT_LBD	((uint16_t)0x0846)
#define USART_IT_CTS	((uint16_t)0x096A)
#define USART_IT_ERR	((uint16_t)0x0060)
#define USART_IT_ORE	((uint16_t)0x0360)
#define USART_IT_NE	((uint16_t)0x0260)
#define USART_IT_FE	((uint16_t)0x0160)

6.串口使能。串口使能是通过函数 USART_Cmd()来实现的，这个很容易理解，使用方法是：

```
USART_Cmd(USART1, ENABLE); //使能串口
```

7.开启串口响应中断。有些时候当我们还需要开启串口中断，那么我们还需要使能串口中断，使能串口中断的函数是：

```
void USART_ITConfig(USART_TypeDef* USARTx, uint16_t USART_IT,
FunctionalState NewState)
```

这个函数的第二个入口参数是标示使能串口的类型，也就是使能哪种中断，因为串口的中断类型有很多种。比如在接收到数据的时候 (RXNE 读数据寄存器非空)，我们要产生中断，那么我们开启中断的方法是：

USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //开启中断，接收到数据中断
我们在发送数据结束的时候 (TC，发送完成) 要产生中断，那么方法是：

```
USART_ITConfig(USART1, USART_IT_TC, ENABLE);
```

8.获取相应中断状态。当我们使能了某个中断的时候，当该中断发生了，就会设置状态寄存器中的某个标志位。经常我们在中断处理函数中，要判断该中断是哪种中断，使用的函数是：

```
ITStatus USART_GetITStatus(USART_TypeDef* USARTx, uint16_t USART_IT)
```

比如我们使能了串口发送完成中断，那么当中断发生了，我们便可以在中断处理函数中调用这个函数来判断到底是否是串口发送完成中断，方法是：

```
USART_GetITStatus(USART1, USART_IT_TC)
```

返回值是 SET，说明是串口发送完成中断发生。

通过以上的讲解，我们就可以达到串口最基本的配置了，关于串口更详细的介绍，请参考《STM32 参考手册》第 516 页至 548 页，通用同步异步收发器一章。

9.2 硬件设计

本实验需要用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 串口 1

串口 1 之前还没有介绍过，本实验用到的串口 1 与 USB 串口并没有在 PCB 上连接在一起，需要通过跳线帽来连接一下。这里我们把 P6 的 RXD 和 TXD 用跳线帽与 PA9 和 PA10 连接起来。如图 9.2.1 所示：

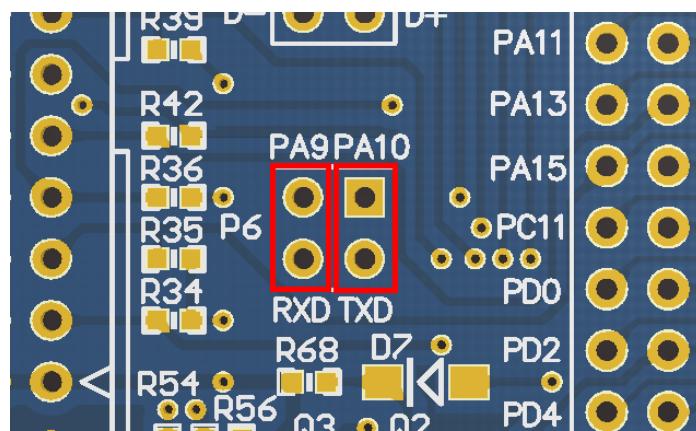


图 9.2.1 硬件连接图示意图

连接上这里之后，我们在硬件上就设置完成了，可以开始软件设计了。

9.3 软件设计

本章的代码设计，比前两章简单很多，因为我们的串口初始化代码和接收代码就是用我们之前介绍的 SYSTEM 文件夹下的串口部分的内容。这里我们对代码部分稍作讲解。

打开串口实验工程，然后在 SYSTEM 组下双击 usart.c，我们就可以看到该文件里面的代码，先介绍 uart_init 函数，该函数代码如下：

```
//初始化 IO 串口 1
//bound:波特率
void uart_init(u32 bound)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    //①串口时钟使能，GPIO 时钟使能，复用时钟使能
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1|
                           RCC_APB2Periph_GPIOA, ENABLE); //使能 USART1,GPIOA 时钟
    //②串口复位
    USART_DeInit(USART1); //复位串口 1
    //③GPIO 端口模式设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9; //USART1_TX PA.9
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 GPIOA.9

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10; //USART1_RX PA.10
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //浮空输入
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 GPIOA.10
    //④串口参数初始化
    USART_InitStructure.USART_BaudRate = bound; //波特率设置
```



```

USART_InitStructure USART_WordLength = USART_WordLength_8b; //字长为 8 位
USART_InitStructure USART_StopBits = USART_StopBits_1;           //一个停止位
USART_InitStructure USART_Parity = USART_Parity_No;             //无奇偶校验位
USART_InitStructure USART_HardwareFlowControl
    = USART_HardwareFlowControl_None;      //无硬件数据流控制
USART_InitStructure USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式
USART_Init(USART1, &USART_InitStructure);                      //初始化串口
#if EN_USART1_RX                                         //如果使能了接收
//⑤ 初始化 NVIC
NVIC_InitStructure NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure NVIC_IRQChannelPreemptionPriority=3;        //抢占优先级 3
NVIC_InitStructure NVIC_IRQChannelSubPriority = 3;            //子优先级 3
NVIC_InitStructure NVIC_IRQChannelCmd = ENABLE;                //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure);                                //中断优先级初始化
//⑥ 开启中断
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);               //开启中断
#endif
//⑦ 使能串口
USART_Cmd(USART1, ENABLE);                                    //使能串口
}

```

从该代码可以看出，其初始化串口的过程，和我们前面介绍的一致。我们用标号①~⑥标示了顺序：

- ① 串口时钟使能，GPIO 时钟使能
- ② 串口复位
- ③ GPIO 端口模式设置
- ④ 串口参数初始化
- ⑤ 初始化 NVIC 并且开启中断
- ⑥ 使能串口

这里需要重申的是，对于复用功能下的 GPIO 模式怎么判定，这个需要查看《中文参考手册 V10》P110 的表格“8.1.11 外设的 GPIO 配置”，这个我们在前面的端口复用章节有提到，这里 还是拿出来再讲解一下吧。查看手册得知，配置全双工的串口 1，那么 TX(PA9) 管脚需要配置为推挽复用输出，RX(PA10)管脚配置为浮空输入或者带上拉输入。模式配置参考下面表格：

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

表 9.3.1 串口 GPIO 模式配置表

对于 NVIC 中断优先级管理，我们在前面的章节（4.5 中断优先级管理）也有讲解，这里不做重复讲解了。

这里需要注意一点，因为我们使用到了串口的中断接收，必须在 usart.h 里面设置 EN_USART1_RX 为 1（默认设置就是 1 的）。该函数才会配置中断使能，以及开启串口 1 的



NVIC 中断。这里我们把串口 1 中断放在组 2，优先级设置为组 2 里面的最低。

接下来，根据之前讲解的步骤 7，还要编写中断服务函数。串口 1 的中断服务函数 USART1_IRQHandler，在 5.3.3 已经有详细介绍过了，这里我们就不再介绍了，大家可以翻过去看看。

介绍完了这两个函数，我们回到 main.c，在 main.c 里面编写如下代码：

```
#include "led.h"
#include "delay.h"
#include "key.h"
#include "sys.h"
#include "usart.h"

int main(void)
{
    u8 t;
    u8 len;
    u16 times=0;
    delay_init();                                //延时函数初始化
    NVIC_Configuration();                        //设置 NVIC 中断分组 2
    uart_init(9600);                            //串口初始化波特率为 9600
    LED_Init();                                 //LED 端口初始化
    KEY_Init();                                 //初始化与按键连接的硬件接口
    while(1)
    {
        if(USART_RX_STA&0x8000)
        {   len=USART_RX_STA&0x3f;                //得到此次接收到的数据长度
            printf("\r\n 您发送的消息为:\r\n\r\n");
            for(t=0;t<len;t++)
            {   USART_SendData(USART1, USART_RX_BUF[t]); //向串口 1 发送数据
                while(USART_GetFlagStatus(USART1,USART_FLAG_TC)!=SET);
                    //等待发送结束
            }
            printf("\r\n\r\n"); //插入换行
            USART_RX_STA=0;
        }else
        {   times++;
            if(times%5000==0)
            {   printf("\r\n 战舰 STM32 开发板 串口实验\r\n");
                printf("正点原子@ALIENTEK\r\n\r\n");
            }
            if(times%200==0)printf("请输入数据,以回车键结束\r\n");
            if(times%30==0)LED0=!LED0;           //闪烁 LED,提示系统正在运行.
            delay_ms(10);
        }
    }
}
```



```
}
```

这段代码比较简单，首先我们看看 NVIC_Configuration()函数，该函数是设置中断分组号为2，也就是2位抢占优先级和2位子优先级。

现在重点看下以下两句：

```
USART_SendData(USART1, USART_RX_BUF[t]); //向串口1发送数据
while(USART_GetFlagStatus(USART1, USART_FLAG_TC)!=SET);
```

第一句，其实这就是发送一个字节到串口。第二句呢，就是我们在我们发送一个数据到串口之后，要检测这个数据是否已经被发送完成了。USART_FLAG_TC 是宏定义的数据发送完成标识符。

其他的代码比较简单，我们执行编译之后看看有没有错误，没有错误就可以开始仿真与调试了。整个工程的编译结果如图 9.3.2 所示：

```
Build target 'Target 1'
linking...
Program Size: Code=4420 RO-data=336 RW-data=16 ZI-data=2344
From ELF: creating hex file...
"..\OBJ\test.axf" - 0 Error(s), 0 Warning(s).
```

图 9.3.2 编译结果

可以看到，编译没有任何错误和警告，下面我们可以开始下载验证了。

9.4 下载验证

前面 2 章实例，我们均介绍了软件仿真，仿真的基本技巧也差不多介绍完了，接下来我们将淡化这部分，因为代码都是经过作者检验，并且全部在 ALIENTEK 战舰 STM32 开发板上验证了的，有兴趣的朋友可以自己仿真看看。但是这里要说明几点：

1，IO 口复用的，信号在逻辑分析窗口是不能显示出来的（看不到波形），这一点请大家注意。比如串口的输出，SPI，USB，CAN 等。你在仿真的时候在该窗口看不到任何信息。遇到这样的情况，你就不得不准备一个逻辑分析仪，外加一个 ULINK 或者 JLINK 来做在线调试。但一般情况，这些都是有现成的例子，不用这几个东西一般也能编出来。

2，仿真并不能代表实际情况。只能从某些方面给你一些启示，告诉你大方向，不能尽信仿真，当然也不能完全没有仿真。比如上面 IO 口的输出，仿真的时候，其翻转速度可以达到很快，但是实际上 STM32 的 IO 输出就达不到这个速度。

总之，我们要合理的利用仿真，也不能过于依赖仿真。当仿真解决不了了，可以试试在线调试，在线调试一般都可以知道问题在哪个地方，但是问题要怎么解决还是得各位自己动脑筋、找资料了。

我们把程序下载到战舰 STM32 开发板，可以看到板子上的 DS0 开始闪烁，说明程序已经在跑了。串口调试助手，这里我们选择的是 SSCOM3.3，是由大虾网版主丁丁（聂小孟）编写，是一个非常好用的串口调试助手。它的优点有：1，绿色，免安装；2，体积小，整个软件才 700 多 KB；3，支持自动搜索串口；4，支持中文字符收发；5，具有扩展发送功能。以上是一些主要的优点，当然这个软件也有些小 bug，比如串口在连接的情况下，如果突然断开，那么程序就死机了，而且无法正常关闭（必须在任务管理器里面强制结束），还有只要该软件打开着，你的电脑就别想关机。不过瑕不掩瑜，该软件总的来说，还是非常适合各位使用的。

接着我们打 SSCOM3.3，设置串口为 CH340 的 USB 串口（根据你自己的电脑选择，我的电脑是 COM3），可以看到如图 9.4.1 所示信息：

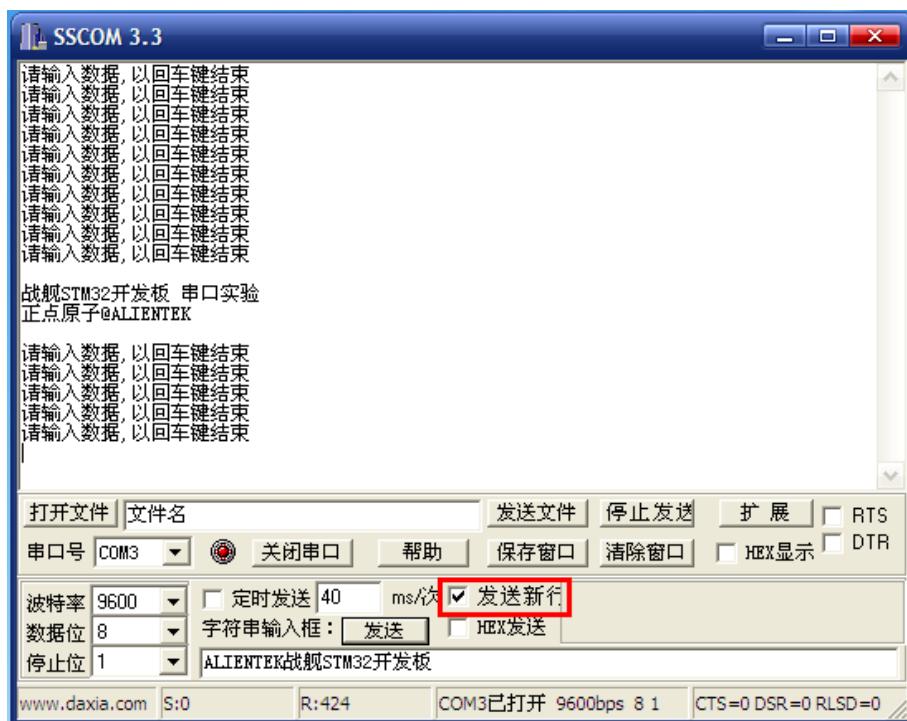


图 9.4.1 串口调试助手收到的信息

从图 9.4.1 可以看出, STM32 的串口数据发送是没问题的了。但是, 因为我们在程序上面设置了必须输入回车, 串口才认可接收到的数据, 所以必须在发送数据后再发送一个回车符, 这里 SSCOM3.3 提供的发送方法是通过勾选发送新行实现, 如图 9.4.1, 只要勾选了这个选项, 每次发送数据后, SSCOM3.3 都会自动多发一个回车。设置好了发送新行, 我们再在发送区输入你想要发送的文字, 然后单击发送, 可以得到如图 9.4.2 所示结果:

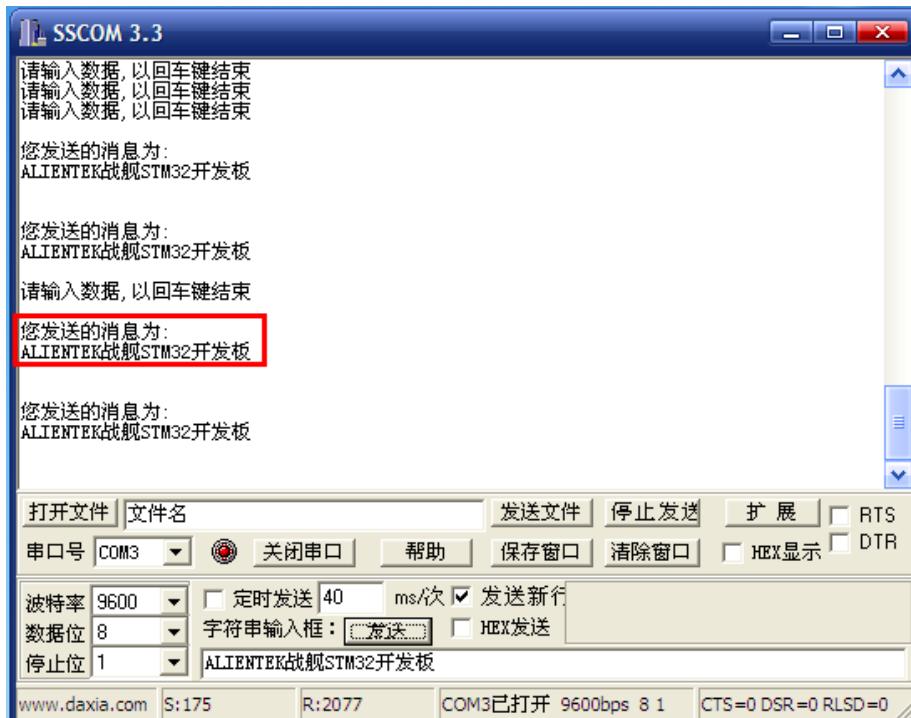


图 9.4.2 发送数据后收到的数据



可以看到，我们发送的消息被发送回来了（图中圈圈内）。各位可以试试，如果不发送回车（取消发送新行），在输入内容之后，直接按发送是什么结果。

第十章 外部中断实验

这一章，我们将向大家介绍如何使用 STM32 的外部输入中断。在前面几章的学习中，我们掌握了 STM32 的 IO 口最基本的操作。本章我们将介绍如何将 STM32 的 IO 口作为外部中断输入，在本章中，我们将以中断的方式，实现我们在第八章所实现的功能。本章分为如下几个部分：

- 10.1 STM32 外部中断简介
- 10.2 硬件设计
- 10.3 软件设计
- 10.4 下载验证



10.1 STM32 外部中断简介

STM32 的 IO 口在第六章有详细介绍，而中断管理分组管理在前面也有详细的阐述。这里我们将介绍 STM32 外部 IO 口的中断功能，通过中断的功能，达到第八章实验的效果，即：通过板载的 4 个按键，控制板载的两个 LED 的亮灭以及蜂鸣器的发声。

这里的代码主要分布在固件库的 `stm32f10x_exti.h` 和 `stm32f10x_exti.c` 文件中。

这里我们首先介绍 STM32 IO 口中断的一些基础概念。STM32 的每个 IO 都可以作为外部中断的中断输入口，这点也是 STM32 的强大之处。STM32F103 的中断控制器支持 19 个外部中断/事件请求。每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置。STM32F103 的 19 个外部中断为：

线 0~15：对应外部 IO 口的输入中断。

线 16：连接到 PVD 输出。

线 17：连接到 RTC 闹钟事件。

线 18：连接到 USB 唤醒事件。

从上面可以看出，STM32 供 IO 口使用的中断线只有 16 个，但是 STM32 的 IO 口却远远不止 16 个，那么 STM32 是怎么把 16 个中断线和 IO 口一一对应起来的呢？于是 STM32 就这样设计，GPIO 的管教 GPIOx.0~GPIOx.15(x=A,B,C,D,E, F,G)分别对应中断线 0~15。这样每个中断线对应了最多 7 个 IO 口，以线 0 为例：它对应了 GPIOA.0、GPIOB.0、GPIOC.0、GPIOD.0、GPIOE.0、GPIOF.0、GPIOG.0。而中断线每次只能连接到 1 个 IO 口上，这样就需要通过配置来决定对应的中断线配置到哪个 GPIO 上了。下面我们看看 GPIO 跟中断线的映射关系图：

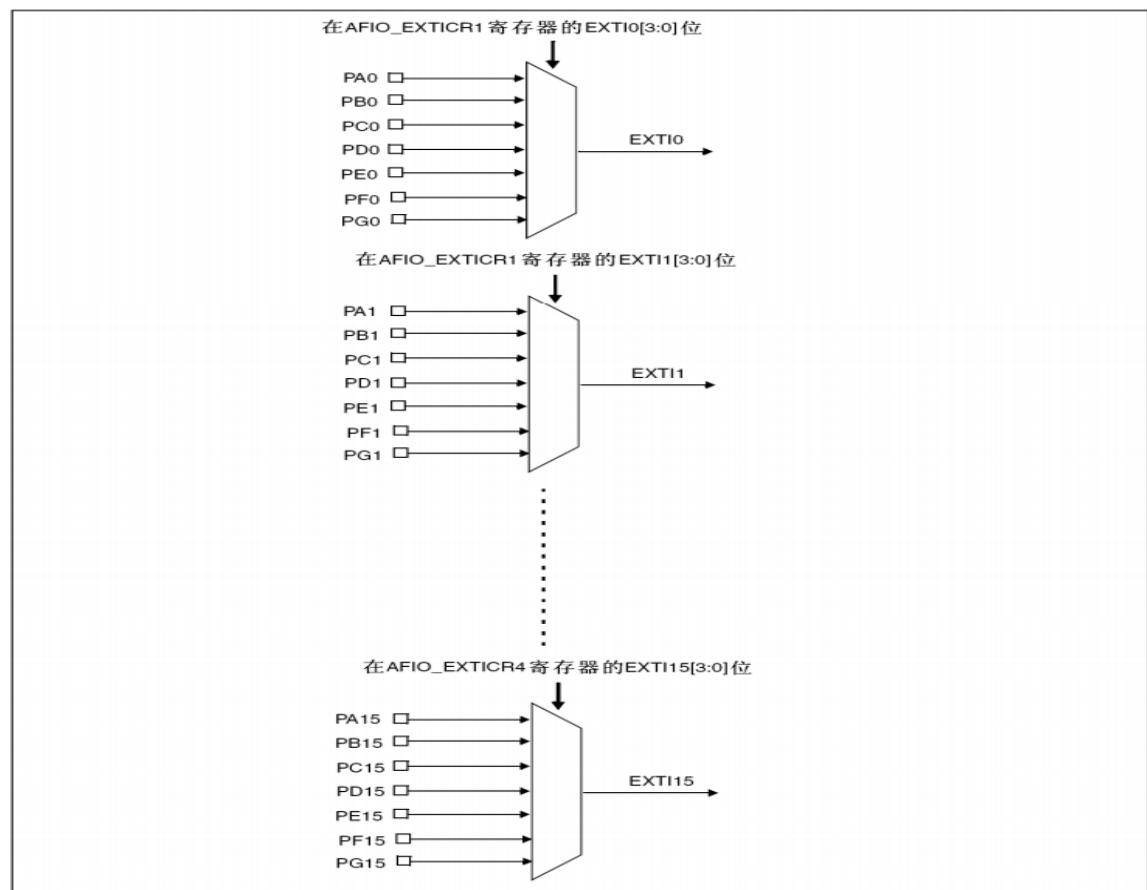


图 10.1.1 GPIO 和中断线的映射关系图



在库函数中，配置 GPIO 与中断线的映射关系的函数 GPIO_EXTILineConfig()来实现的：

```
void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource)
```

该函数将 GPIO 端口与中断线映射起来，使用范例是：

```
GPIO_EXTILineConfig(GPIO_PortSourceGPIOE,GPIO_PinSource2);
```

将中断线 2 与 GPIOE 映射起来，那么很显然是 GPIOE.2 与 EXTI2 中断线连接了。设置好中断线映射之后，那么到底来自这个 IO 口的中断是通过什么方式触发的呢？接下来我们就要设置该中断线上中断的初始化参数了。

中断线上中断的初始化是通过函数 EXTI_Init()实现的。EXTI_Init()函数的定义是：

```
void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct);
```

下面我们用一个使用范例来说明这个函数的使用：

```
EXTI_InitTypeDef  EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line=EXTI_Line4;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);      //根据 EXTI_InitStruct 中指定的
//参数初始化外设 EXTI 寄存器
```

上面的例子设置中断线 4 上的中断为下降沿触发。STM32 的外设的初始化都是通过结构体来设置初始值的，这里就不罗嗦结构体初始化的过程了。我们来看看结构体 EXTI_InitTypeDef 的成员变量：

```
typedef struct
{
    uint32_t EXTI_Line;
    EXTIMode_TypeDef EXTI_Mode;
    EXTITrigger_TypeDef EXTI_Trigger;
    FunctionalState EXTI_LineCmd;
}EXTI_InitTypeDef;
```

从定义可以看出，有 4 个参数需要设置。第一个参数是中断线的标号，取值范围为 EXTI_Line0~EXTI_Line15。这个在上面已经讲过中断线的概念。也就是说，这个函数配置的是某个中断线上的中断参数。第二个参数是中断模式，可选值为中断 EXTI_Mode_Interrupt 和事件 EXTI_Mode_Event。第三个参数是触发方式，可以是下降沿触发 EXTI_Trigger_Falling，上升沿触发 EXTI_Trigger_Rising，或者任意电平（上升沿和下降沿）触发 EXTI_Trigger_Rising_Falling，相信学过 51 的对这个不难理解。最后一个参数就是使能中断线了。

我们设置好中断线和 GPIO 映射关系，然后又设置好了中断的触发模式等初始化参数。既然是外部中断，涉及到中断我们当然还要设置 NVIC 中断优先级。这个在前面已经讲解过，这里我们就接着上面的范例， 设置中断线 2 的中断优先级。

```
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC IRQChannel = EXTI2_IRQn;           //使能按键外部中断通道
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x02; //抢占优先级 2,
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x02;        //子优先级 2
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;             //使能外部中断通道
```



```
NVIC_Init(&NVIC_InitStructure); //中断优先级分组初始化
```

上面这段代码相信大家都不陌生，我们在前面的串口实验的时候讲解过，这里不再讲解。

我们配置完中断优先级之后，接着我们要做的就是编写中断服务函数。中断服务函数的名字是在 MDK 中事先有定义的。这里需要说明一下，STM32 的 IO 口外部中断函数只有 6 个，分别为：

```
EXPORT EXTI0_IRQHandler  
EXPORT EXTI1_IRQHandler  
EXPORT EXTI2_IRQHandler  
EXPORT EXTI3_IRQHandler  
EXPORT EXTI4_IRQHandler  
EXPORT EXTI9_5_IRQHandler  
EXPORT EXTI15_10_IRQHandler
```

中断线 0-4 每个中断线对应一个中断函数，中断线 5-9 共用中断函数 EXTI9_5_IRQHandler，中断线 10-15 共用中断函数 EXTI15_10_IRQHandler。在编写中断服务函数的时候会经常使用到两个函数，第一个函数是判断某个中断线上的中断是否发生（标志位是否置位）：

```
ITStatus EXTI_GetITStatus(uint32_t EXTI_Line);
```

这个函数一般使用在中断服务函数的开头判断中断是否发生。另一个函数是清除某个中断线上的中断标志位：

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line);
```

这个函数一般应用在中断服务函数结束之前，清除中断标志位。

常用的中断服务函数格式为：

```
void EXTI2_IRQHandler(void)  
{  
    if(EXTI_GetITStatus(EXTI_Line3)!=RESET)//判断某个线上的中断是否发生  
    {  
        中断逻辑…  
        EXTI_ClearITPendingBit(EXTI_Line3); //清除 LINE 上的中断标志位  
    }  
}
```

在这里需要说明一下，固件库还提供了两个函数用来判断外部中断状态以及清除外部状态标志位的函数 EXTI_GetFlagStatus 和 EXTI_ClearFlag，他们的作用和前面两个函数的作用类似。只是在 EXTI_GetITStatus 函数中会先判断这种中断是否使能，使能了才去判断中断标志位，而 EXTI_GetFlagStatus 直接用来判断状态标志位。

讲到这里，相信大家对于 STM32 的 IO 口外部中断已经有了一定了了解。下面我们再总结一下使用 IO 口外部中断的一般步骤：

- 1) 初始化 IO 口为输入。
- 2) 开启 IO 口复用时钟，设置 IO 口与中断线的映射关系。
- 3) 初始化线上中断，设置触发条件等。
- 4) 配置中断分组 (NVIC)，并使能中断。
- 5) 编写中断服务函数。

通过以上几个步骤的设置，我们就可以正常使用外部中断了。

本章，我们要实现同第八章差不多的功能，但是这里我们使用的是中断来检测按键，还是 WK_UP 控制蜂鸣器，按一次叫，再按一次停；KEY2 控制 DS0，按一次亮，再按一次灭；KEY1



控制 DS1，效果同 KEY2；KEY0 则同时控制 DS0 和 DS1，按一次，他们的状态就翻转一次。

10.2 硬件设计

本实验用到的硬件资源和第八章实验的一模一样，不再多做介绍了。

10.3 软件设计

软件设计我们直接打开我们的光盘的实验 10 的工程，可以看到相比上一个工程，我们的 HARDWARE 目录下面增加了 exti.c 文件，同时固件库目录增加了 stm32f10x_exti.c 文件。

exit.c 文件总共包含 5 个函数。一个是外部中断初始化函数 void EXTIx_Init(void)，另外 4 个都是中断服务函数。

void EXTI0_IRQHandler(void) 是外部中断 0 的服务函数，负责 WK_UP 按键的中断检测；

void EXTI2_IRQHandler(void) 是外部中断 2 的服务函数，负责 KEY2 按键的中断检测；

void EXTI3_IRQHandler(void) 是外部中断 3 的服务函数，负责 KEY1 按键的中断检测；

void EXTI4_IRQHandler(void) 是外部中断 4 的服务函数，负责 KEY0 按键的中断检测；

因为 exit.c 里面的代码较多，而且对于每个中断线的配置几乎都是雷同，下面我们列出中断线 2 的相关配置代码：

```
#include "exti.h"
#include "led.h"
#include "key.h"
#include "delay.h"
#include "uart.h"
#include "beep.h"
//外部中断 0 服务程序
void EXTIx_Init(void)
{
    EXTI_InitTypeDef EXTI_InitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;
    KEY_Init();                                //按键端口初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE);
    //GPIOE.2 中断线以及中断初始化配置,下降沿触发
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOE,GPIO_PinSource2);
    EXTI_InitStruct.EXTI_Line=EXTI_Line2;
    EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿触发
    EXTI_InitStruct.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStruct);                //初始化中断线参数

    NVIC_InitStruct.NVIC_IRQChannel = EXTI2_IRQn;   //使能按键外部中断通道
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x02; //抢占优先级 2,
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x02; //子优先级 2
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;     //使能外部中断通道
```



```
NVIC_Init(&NVIC_InitStructure);  
}  
//外部中断 2 服务程序  
void EXTI2_IRQHandler(void)  
{  
    delay_ms(10);           //消抖  
    if(KEY2==0)             //按键 KEY2  
    {  
        LED0=!LED0;         //翻转 LED0 控制信号  
    }  
    EXTI_ClearITPendingBit(EXTI_Line2); //清除 LINE2 上的中断标志位  
}
```

外部中断初始化函数 void EXTIX_Init(void)，该函数严格按照我们之前的步骤来初始化外部中断，首先调用 KEY_Init()函数，利用第八章按键初始化函数，来初始化外部中断输入的 IO 口，接着调用 RCC_APB2PeriphClockCmd()函数来使能复用功能时钟。接着配置中断线和 GPIO 的映射关系，然后初始化中断线。需要说明的是因为我们的 WK_UP 按键是高电平有效的，而 KEY0、KEY1 和 KEY2 是低电平有效的，所以我们设置 WK_UP 为上升沿触发中断，而 KEY0、KEY1 和 KEY2 则设置为下降沿触发。这里我们把所有中断都分配到第二组，把按键的抢占优先级设置成一样，而子优先级不同，这四个按键，KEY0 的优先级最高。

接下来我们介绍各个按键的中断服务函数，一共 4 个。先看按键 KEY2 的中断服务函数 void EXTI2_IRQHandler (void)，该函数代码比较简单，先延时 10ms 以消抖，再检测 KEY2 是否还是为低电平，如果是，则执行此次操作（翻转 LED0 控制信号），如果不是，则直接跳过，在最后有一句 EXTI_ClearITPendingBit(EXTI_Line2);通过该句清除已经发生的中断请求。同样，我们可以发现 KEY0、KEY1 和 WK_UP 的中断服务函数和 KEY2 按键的十分相似，我们就不逐个介绍了。

接下来我们看看 main.c 里面里面的内容：

```
#include "led.h"  
#include "delay.h"  
#include "key.h"  
#include "sys.h"  
#include "uart.h"  
#include "exti.h"  
#include "beep.h"  
int main(void)  
{  
    delay_init();           //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级  
    uart_init(9600);        //串口初始化波特率为 9600  
    LED_Init();            //初始化与 LED 连接的硬件接口  
    BEEP_Init();            //初始化蜂鸣器端口  
    KEY_Init();             //初始化与按键连接的硬件接口  
    EXTIX_Init();           //外部中断初始化  
    LED0=0;                 //点亮 LED0
```



```
while(1)
{
    printf("OK\r\n");//打印 OK
    delay_ms(1000);
}
```

该部分代码很简单，在初始化完中断后，点亮 LED0，就进入死循环等待了，这里死循环里面通过一个 `printf` 函数来告诉我们系统正在运行，在中断发生后，就执行相应的处理，从而实现第八章类似的功能。

10.4 下载验证

在编译成功之后，我们就可以下载代码到战舰 STM32 开发板上，实际验证一下我们的程序是否正确。下载代码后，在串口调试助手里面可以看到如图 10.4.1 所示信息：



图 10.4.1 串口收到的数据

从图 10.4.1 可以看出，程序已经在运行了，此时可以通过按下 KEY0、KEY1、KEY2 和 WK_UP 来观察 DS0、DS1 以及蜂鸣器是否跟着按键的变化而变化。

第十一章 独立看门狗（IWDG）实验

这一章，我们将向大家介绍如何使用 STM32 的独立看门狗（以下简称 IWDG）。STM32 内部自带了 2 个看门狗：独立看门狗（IWDG）和窗口看门狗（WWDG）。这一章我们只介绍独立看门狗，窗口看门狗将在下一章介绍。在本章中，我们将通过按键 WK_UP 来喂狗，然后通过 DS0 提示复位状态。本章分为如下几个部分：

11.1 STM32 独立看门狗简介

11.2 硬件设计

11.3 软件设计

11.4 下载验证



11.1 STM32 独立看门狗简介

STM32 的独立看门狗由内部专门的 40Khz 低速时钟驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 40Khz，而是在 30~60Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 40Khz 的频率来计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

首先我们得讲解一下看门狗的原理。这个百度百科里面有很详细的解释。我们总结一下：单片机系统在外界的干扰下会出现程序跑飞的现象导致出现死循环，看门狗电路就是为了避免这种情况的发生。看门狗的作用就是在一定时间内（通过定时计数器实现）没有接收喂狗信号（表示 MCU 已经挂了），便实现处理器的自动复位重启（发送复位信号）。

下面我们在了解几个与独立看门狗相关联的寄存器之后讲解怎么通过库函数来实现配置。首先是键值寄存器 IWDG_KR，该寄存器的各位描述如图 11.1.1 所示：

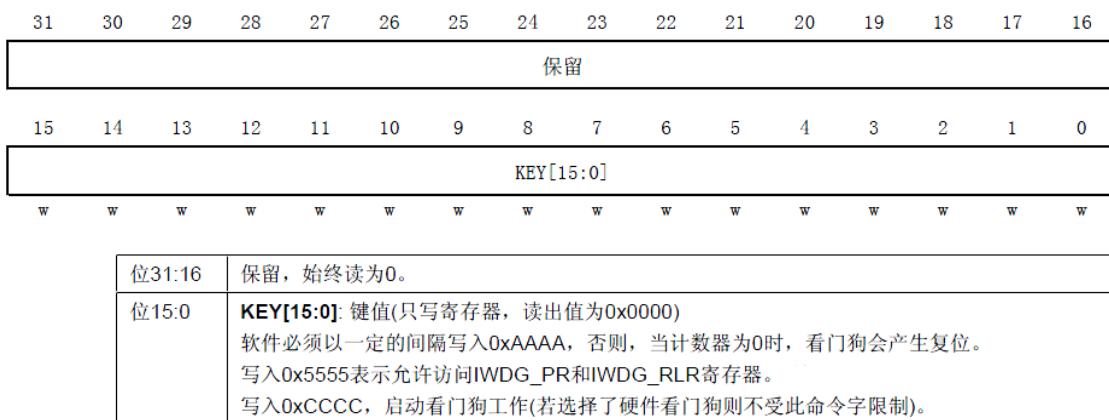


图 11.1.1 IWDG_KR 寄存器各位描述

在键值寄存器(IWDG_KR)中写入 0xCCCC，开始启用独立看门狗；此时计数器开始从其复位值 0xFFFF 递减计数。当计数器计数到末尾 0x000 时，会产生一个复位信号(IWDG_RESET)。无论何时，只要键寄存器 IWDG_KR 中被写入 0xAAAA，IWDG_RLR 中的值就会被重新加载到计数器中从而避免产生看门狗复位。

IWDG_PR 和 IWDG_RLR 寄存器具有写保护功能。要修改这两个寄存器的值，必须先向 IWDG_KR 寄存器中写入 0x5555。将其他值写入这个寄存器将会打乱操作顺序，寄存器将重新被保护。重装载操作(即写入 0xAAAA)也会启动写保护功能。

还有两个寄存器，一个预分频寄存器 (IWDG_PR)，该寄存器用来设置看门狗时钟的分频系数。另一个重装载寄存器。该寄存器用来保存重装载到计数器中的值。该寄存器也是一个 32 位寄存器，但是只有低 12 位是有效的。

只要对以上三个寄存器进行相应的设置，我们就可以启动 STM32 的独立看门狗，启动过程可以按如下步骤实现（独立看门狗相关的库函数和定义分布在文件 `stm32f10x_iwdg.h` 和 `stm32f10x_iwdg.c` 中）：

1) 取消寄存器写保护（向 IWDG_KR 写入 0X5555）

通过这步，我们取消 IWDG_PR 和 IWDG_RLR 的写保护，使后面可以操作这两个寄存器，设置 IWDG_PR 和 IWDG_RLR 的值。这在库函数中的实现函数是：

```
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
```

这个函数非常简单，顾名思义就是开启/取消写保护，也就是使能/失能写权限。

2) 设置独立看门狗的预分频系数和重装载值



设置看门狗的分频系数的函数是：

```
void IWDG_SetPrescaler(uint8_t IWDG_Prescaler); //设置 IWDG 预分频值
```

设置看门狗的重装载值的函数是：

```
void IWDG_SetReload(uint16_t Reload); //设置 IWDG 重装载值
```

设置好看门狗的分频系数 prer 和重装载值就可以知道看门狗的喂狗时间（也就是看门狗溢出时间），该时间的计算方式为：

$$Tout = ((4 \times 2^{prer}) \times rlr) / 40$$

其中 Tout 为看门狗溢出时间（单位为 ms）； prer 为看门狗时钟预分频值（IWDG_PR 值），范围为 0~7； rlr 为看门狗的重装载值（IWDG_RLR 的值）；

比如我们设定 prer 值为 4， rlr 值为 625，那么就可以得到 $Tout = 64 \times 625 / 40 = 1000\text{ms}$ ，这样，看门狗的溢出时间就是 1s，只要你在一秒钟之内，有一次写入 0XAAAA 到 IWDG_KR，就不会导致看门狗复位（当然写入多次也是可以的）。这里需要提醒大家的是，看门狗的时钟不是准确的 40Khz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

3) 重载计数值喂狗（向 IWDG_KR 写入 0XAAAA）

库函数里面重载计数值的函数是：

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

通过这句，将使 STM32 重新加载 IWDG_RLR 的值到看门狗计数器里面。即实现独立看门狗的喂狗操作。

4) 启动看门狗(向 IWDG_KR 写入 0XC000)

库函数里面启动独立看门狗的函数是：

```
IWDG_Enable(); //使能 IWDG
```

通过这句，来启动 STM32 的看门狗。注意 IWDG 在一旦启用，就不能再被关闭！想要关闭，只能重启，并且重启之后不能打开 IWDG，否则问题依旧，所以在这里提醒大家，如果不使用 IWDG 的话，就不要去打开它，免得麻烦。

通过上面 4 个步骤，我们就可以启动 STM32 的看门狗了，使能了看门狗，在程序里面就必须间隔一定时间喂狗，否则将导致程序复位。利用这一点，我们本章将通过一个 LED 灯来指示程序是否重启，来验证 STM32 的独立看门狗。

在配置看门狗后，DS0 将常亮，如果 WK_UP 按键按下，就喂狗，只要 WK_UP 不停的按，看门狗就一直不会产生复位，保持 DS0 的常亮，一旦超过看门狗定溢出时间（Tout）还没按，那么将会导致程序重启，这将导致 DS0 熄灭一次。

11.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) WK_UP 按键
- 3) 独立看门狗

前面两个在之前都有介绍，而独立看门狗实验的核心是在 STM32 内部进行，并不需要外部电路。但是考虑到指示当前状态和喂狗等操作，我们需要 2 个 IO 口，一个用来输入喂狗信号，另外一个用来指示程序是否重启。喂狗我们采用板上的 WK_UP 键来操作，而程序重启，则是通过 DS0 来指示的。



11.3 软件设计

我们直接打开光盘的独立看门狗实验工程，可以看到工程里面新增了 wdg.c，同时引入了头文件 wdg.h。同样的道理，我们要加入固件库看门狗支持文件 stm32f10x_iwdg.h 和 stm32f10x_iwdg.c 文件。

wdg.c 里面的代码如下：

```
#include "wdg.h"
//初始化独立看门狗
//prer:分频数:0~7(只有低 3 位有效!)
//分频因子=4*2^prer.但最大值只能是 256!
//rlr:重装载寄存器值:低 11 位有效.
//时间计算(大概):Tout=((4*2^prer)*rlr)/40 (ms).
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable); //①使能对寄存器 I 写操作
    IWDG_SetPrescaler(prer); //②设置 IWDG 预分频值:设置 IWDG 预分频值
    IWDG_SetReload(rlr); //②设置 IWDG 重装载值
    IWDG_ReloadCounter(); //③按照 IWDG 重装载寄存器的值重装载 IWDG 计数器
    IWDG_Enable(); //④使能 IWDG
}
//喂独立看门狗
void IWDG_Feed(void)
{
    IWDG_ReloadCounter(); //reload
}
```

该代码就 2 个函数，void IWDG_Init(u8 prer, u16 rlr)是独立看门狗初始化函数，就是按照上面介绍的步骤 1~4 来初始化独立看门狗的。该函数有 2 个参数，分别用来设置与预分频数与重装寄存器的值的。通过这两个参数，就可以大概知道看门狗复位的时间周期为多少了。其计算方式上面有详细的介绍，这里不再多说了。

void IWDG_Feed(void)函数，该函数用来喂狗，因为 STM32 的喂狗只需要向键值寄存器写入 0XAAAA 即可，也就是调用 IWDG_ReloadCounter()函数，所以，我们这个函数也是简单的很。

头文件 wdg.h 的源码如下大家可以看下，这里我们就不列出来了。

接下来我们看看主函数 main 的代码。在主程序里面我们先初始化一下系统代码，然后启动按键输入和看门狗，在看门狗开启后马上点亮 LED0 (DS0)，并进入死循环等待按键的输入，一旦 WK_UP 有按键，则喂狗，否则等待 IWDG 复位的到来。这段代码很容易理解，该部分代码如下：

```
int main(void)
{
    delay_init(); //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600); //串口初始化波特率为 9600
    LED_Init(); //初始化与 LED 连接的硬件接口
    KEY_Init(); //按键初始化
```



```
delay_ms(500);           //让人看得到灭
IWDG_Init(4,625);       //与分频数为 64,重载值为 625,溢出时间为 1s
LED0=0;                  //点亮 LED0
while(1)
{
    if(KEY_Scan(0)==KEY_UP)
    {
        IWDG_Feed();      //如果 WK_UP 按下,则喂狗
    }
    delay_ms(10);
}
}
```

上面的代码，鉴于篇幅考虑，我们没有把头文件给列出来（后续实例将会采用类同的方式处理），因为以后我们包含的头文件会越来越多，大家想看，可以直接打开光盘相关源码查看。至此，独立看门狗的实验代码，我们就全部编写完了，接着要做的就是下载验证了，看看我们的代码是否真的正确，当然在下载之前可以通过软件仿真看看是否可行。

11.4 下载验证

在编译成功之后，我们就可以下载代码到战舰 STM32 开发板上，实际验证一下，我们的程序是否正确。下载代码后，可以看到 DS0 不停的闪烁，证明程序在不停的复位，否则只会 DS0 常亮。这时我们试试不停的按 WK_UP 按键，可以看到 DS0 就常亮了，不会再闪烁。说明我们的实验是成功的。



第十二章 窗口门狗（WWDG）实验

这一章，我们将向大家介绍如何使用 STM32 的另外一个看门狗，窗口看门狗（以下简称 WWDG）。在本章中，我们将利用窗口看门狗的中断功能来喂狗，通过 DS0 和 DS1 提示程序的运行状态。本章分为如下几个部分：

12.1 STM32 窗口看门狗简介

12.2 硬件设计

12.3 软件设计

12.4 下载验证



12.1 STM32 窗口看门狗简介

窗口看门狗（WWWDG）通常被用来监测由外部干扰或不可预见的逻辑条件造成应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位(WWDG->CR 的第六位)变成 0 前被刷新，看门狗电路在达到预置的时间周期时，会产生一个 MCU 复位。在递减计数器达到窗口配置寄存器(WWDG->CFR)数值之前，如果 7 位的递减计数器数值(在控制寄存器中)被刷新，那么也将产生一个 MCU 复位。这表明递减计数器需要在一个有限的时间窗口中被刷新。他们的关系可以用图 12.1.1 来说明：

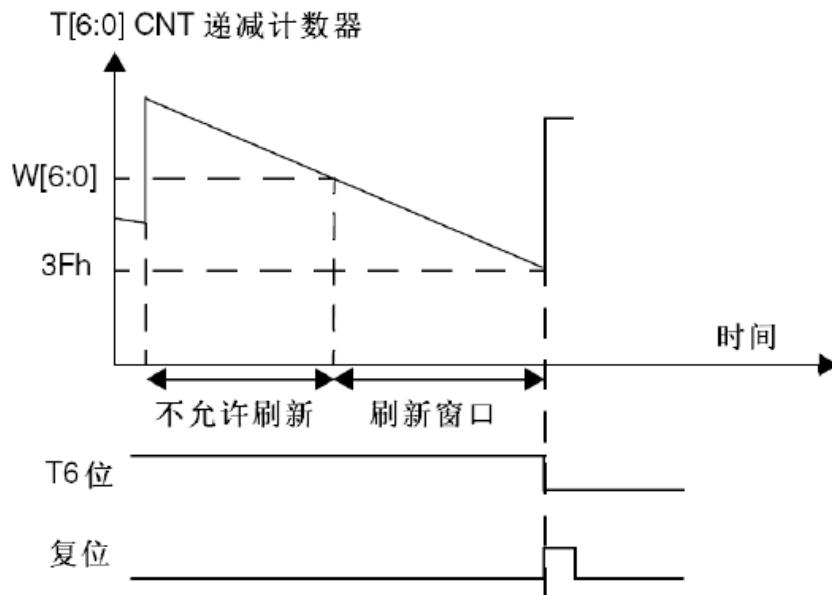


图 12.1.1 窗口看门狗工作示意图

图 12.1.1 中，T[6:0]就是 WWDG_CR 的低七位，W[6:0]即是 WWDG->CFR 的低七位。T[6:0]就是窗口看门狗的计数器，而 W[6:0]则是窗口看门狗的上窗口，下窗口值是固定的 (0X40)。当窗口看门狗的计数器在上窗口值之外被刷新，或者低于下窗口值都会产生复位。

上窗口值 (W[6:0]) 是由用户自己设定的，根据实际要求来设计窗口值，但是一定要确保窗口值大于 0X40，否则窗口就不存在了。

窗口看门狗的超时公式如下：

$$T_{wwdg} = (4096 \times 2^{WDGTB} \times (T[5:0]+1)) / F_{pclk1};$$

其中：

Twwdg：WWWDG 超时时间（单位为 ms）

Fpclk1：APB1 的时钟频率（单位为 KHz）

WDGTB：WWWDG 的预分频系数

T[5:0]：窗口看门狗的计数器低 6 位

根据上面的公式，假设 Fpclk1=36Mhz，那么可以得到最小-最大超时时间表如表 12.1.1 所示：



WDGTB	最小超时值	最大超时值
0	113 μ s	7.28ms
1	227 μ s	14.56ms
2	455 μ s	29.12ms
3	910 μ s	58.25ms

表 12.1.1 36M 时钟下窗口看门狗的最小最大超时表

接下来，我们介绍窗口看门狗的 3 个寄存器。首先介绍控制寄存器 (WWDG_CR)，该寄存器的各位描述如图 12.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								WDGA	T6	T5	T4	T3	T2	T1	T0
rs								rw	rw	rw	rw	rw	rw	rw	rw

图 12.1.2 WWDG_CR 寄存器各位描述

可以看出，这里我们的 WWDG_CR 只有低八位有效，T[6: 0]用来存储看门狗的计数器值，随时更新的，每个窗口看门狗计数周期 (4096×2^7 WDGTB) 减 1。当该计数器的值从 0X40 变为 0X3F 的时候，将产生看门狗复位。

WDGA 位则是看门狗的激活位，该位由软件置 1，以启动看门狗，并且一定要注意的是该位一旦设置，就只能在硬件复位后才能清零了。

窗口看门狗的第二个寄存器是配置寄存器 (WWDG_CFR)，该寄存器的各位及其描述如图 12.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16		
保留																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
保留								EWI	WDG TB1	WDG TB0	W6	W5	W4	W3	W2	W1	W0
rs								rw	rw	rw	rw	rw	rw	rw	rw	rw	

位31:8	保留。
位9	EWI: 提前唤醒中断 此位若置 1，则当计数器值达到 40h，即产生中断。 此中断只能由硬件在复位后清除。
位8:7	WDGTB[1:0]: 时基 预分频器的时基可根据如下修改： 00: CK计时器时钟(PCLK1除以4096)除以1 01: CK计时器时钟(PCLK1除以4096)除以2 10: CK计时器时钟(PCLK1除以4096)除以4 11: CK计时器时钟(PCLK1除以4096)除以8
位6:0	W[6:0]: 7位窗口值 这些位包含了用来与递减计数器进行比较用的窗口值。

图 12.1.3 WWDG_CFR 寄存器各位描述

该位中的 EWI 是提前唤醒中断，也就是在快要产生复位的前一段时间 (T[6:0]=0X40) 来提醒我们，需要进行喂狗了，否则将复位！因此，我们一般用该位来设置中断，当窗口看门狗的计数器值减到 0X40 的时候，如果该位设置，并开启了中断，则会产生中断，我们可以在中



断里面向 WWDG_CR 重新写入计数器的值，来达到喂狗的目的。注意这里在进入中断后，必须在不大于 1 个窗口看门狗计数周期的时间(在 PCLK1 频率为 36M 且 WDGTB 为 0 的条件下，该时间为 113us) 内重新写 WWDG_CR，否则，看门狗将产生复位！

最后我们要介绍的是状态寄存器 (WWDG_SR)，该寄存器用来记录当前是否有提前唤醒的标志。该寄存器仅有位 0 有效，其他都是保留位。当计数器值达到 40h 时，此位由硬件置 1。它必须通过软件写 0 来清除。对此位写 1 无效。即使中断未被使能，在计数器的值达到 0X40 的时候，此位也会被置 1。

在介绍完了窗口看门狗的寄存器之后，我们介绍要如何启用 STM32 的窗口看门狗。这里我们介绍库函数中用中断的方式来喂狗的方法，窗口看门狗库函数相关源码和定义分布在文件 stm32f10x_wwdg.c 文件和头文件 stm32f10x_wwdg.h 中。步骤如下：

1) 使能 WWDG 时钟

WWDG 不同于 IWDG，IWDG 有自己独立的 40Khz 时钟，不存在使能问题。而 WWDG 使用的是 PCLK1 的时钟，需要先使能时钟。方法是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE); // WWDG 时钟使能
```

2) 设置窗口值和分频数

设置窗口值的函数是：

```
void WWDG_SetWindowValue(uint8_t WindowValue);
```

这个函数就一个入口参数为窗口值，很容易理解。

设置分频数的函数是：

```
void WWDG_SetPrescaler(uint32_t WWDG_Prescaler);
```

这个函数同样只有一个入口参数就是分频值。

3) 开启 WWDG 中断并分组

开启 WWDG 中断的函数为：

```
WWDG_EnableIT(); //开启窗口看门狗中断
```

接下来是进行中断优先级配置，这里就不重复了，使用 NVIC_Init() 函数即可。

4) 设置计数器初始值并使能看门狗

这一步在库函数里面是通过一个函数实现的：

```
void WWDG_Enable(uint8_t Counter);
```

该函数既设置了计数器初始值，同时使能了窗口看门狗。

5) 编写中断服务函数

在最后，还是要编写窗口看门狗的中断服务函数，通过该函数来喂狗，喂狗要快，否则当窗口看门狗计数器值减到 0X3F 的时候，就会引起软复位了。在中断服务函数里面也要将状态寄存器的 EWIF 位清空。

完成了以上 5 个步骤之后，我们就可以使用 STM32 的窗口看门狗了。这一章的实验，我们将通过 DS0 来指示 STM32 是否被复位了，如果被复位了就会点亮 300ms。DS1 用来指示中断喂狗，每次中断喂狗翻转一次。

12.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0 和 DS1
- 2) 窗口看门狗

其中指示灯前面介绍过了，窗口看门狗属于 STM32 的内部资源，只需要软件设置好即可



正常工作。我们通过 DS0 和 DS1 来指示 STM32 的复位情况和窗口看门狗的喂狗情况。

12.3 软件设计

打开我们的窗口看门狗实验可以看到，相对于独立看门狗，我们只增加了窗口看门狗相关的库函数支持文件 `stm32f10x_wwdg.c/stm32f10x_wwdg.h`,然后在 `wdg.c` 加入如下代码（之前代码保留）：

```
//保存 WWDG 计数器的设置值,默认为最大.  
u8 WWDG_CNT=0x7f;  
//初始化窗口看门狗  
//tr :T[6:0],计数器值  
//wr :W[6:0],窗口值  
//fprer:分频系数 (WDGTB) ,仅最低 2 位有效  
//Fwwdg=PCLK1/(4096*2^fprer).  
void WWDG_Init(u8 tr,u8 wr,u32 fprer)  
{  
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE); // WWDG 时钟使能  
    WWDG_CNT=tr&WWDG_CNT;                                //初始化 WWDG_CNT.  
    WWDG_SetPrescaler(fprer);                            //设置 IWDG 预分频值  
    WWDG_SetWindowValue(wr);                            //设置窗口值  
    WWDG_Enable(WWDG_CNT);                            //使能看门狗,设置 counter  
    WWDG_ClearFlag();                                //清除提前唤醒中断标志位  
    WWDG_NVIC_Init();                                //初始化窗口看门狗 NVIC  
    WWDG_EnableIT();                                //开启窗口看门狗中断  
}  
//重设置 WWDG 计数器的值  
void WWDG_Set_Counter(u8 cnt)  
{  
    WWDG_Enable(cnt);                                //使能看门狗,设置 counter .  
}  
//窗口看门狗中断服务程序  
void WWDG_NVIC_Init()  
{  
    NVIC_InitTypeDef NVIC_InitStruct;  
    NVIC_InitStruct.NVIC_IRQChannel = WWDG_IRQn;      //WWDG 中断  
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 2; //抢占 2 子优先级 3 组 2  
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 3;       //抢占 2,子优先级 3,组 2  
    NVIC_Init(&NVIC_InitStruct);                      //NVIC 初始化  
}  
  
void WWDG_IRQHandler(void)  
{  
    WWDG_SetCounter(WWDG_CNT);                      //当禁掉此句后,窗口看门狗将产生复位
```



```
WWDG_ClearFlag();           //清除提前唤醒中断标志位
LED1=!LED1;                  //LED 状态翻转
}
```

新增的这四个函数都比较简单，第一个函数 void WWDG_Init(u8 tr, u8 wr, u8 fprer)用来设置 WWDG 的初始化值。包括看门狗计数器的值和看门狗比较值等。该函数就是按照我们上面的 4 个思路设计出来的代码。注意到这里有个全局变量 WWDG_CNT，该变量用来保存最初设置 WWDG_CR 计数器的值。在后续的中断服务函数里面，就又把该数值放回到 WWDG_CR 上。

WWDG_Set_Counter()函数比较简单，就是用来重设窗口看门狗的计数器值的。该函数很简单，我们就不多说了。

然后是中断分组函数，这个函数非常简单，之前有讲解，这里不重复。

最后在中断服务函数里面，先重设窗口看门狗的计数器值，然后清除提前唤醒中断标志。最后对 LED1 (DS1) 取反，来监测中断服务函数的执行了状况。我们再把这几个函数名加入到头文件里面去，以方便其他文件调用。

在完成了以上部分之后，我们就回到主函数，代码如下：

```
int main(void)
{
    delay_init();                      //延时函数初始化
    NVIC_Configuration();             //设置 NVIC 中断分组 2
    uart_init(9600);                  //串口初始化波特率为 9600
    LED_Init();                        //LED 初始化
    KEY_Init();                        //按键初始化
    LED0=0;
    delay_ms(300);
    WWDG_Init(0X7F,0X5F,WWDG_Prescaler_8); //计数器值为 7f,窗口寄存器为 5f,
                                              //分频数为 8
    while(1)
    {
        LED0=1;
    }
}
```

该函数通过 LED0(DS0)来指示是否正在初始化。而 LED1(DS1)用来指示是否发生了中断。我们先让 LED0 亮 300ms，然后关闭以用于判断是否有复位发生了。在初始化 WWDG 之后，我们回到死循环，关闭 LED1，并等待看门狗中断的触发/复位。

在编译完成之后，我们就可以下载这个程序到战舰 STM32 开发板上，看看结果是不是和我们设计的一样。

12.4 下载验证

将代码下载到战舰 STM32 后，可以看到 DS0 亮一下之后熄灭，紧接着 DS1 开始不停的闪烁。每秒钟闪烁 5 次左右，和我们预期的一致，说明我们的实验是成功的。



第十三章 定时器中断实验

这一章，我们将向大家介绍如何使用 STM32 的通用定时器，STM32 的定时器功能十分强大，有 TIME1 和 TIME8 等高级定时器，也有 TIME2~TIME5 等通用定时器，还有 TIME6 和 TIME7 等基本定时器。在《STM32 参考手册》里面，定时器的介绍占了 1/5 的篇幅，足见其重要性。在本章中，我们将利用 TIM3 的定时器中断来控制 DS1 的翻转，在主函数用 DS0 的翻转来提示程序正在运行。本章，我们选择难度适中的通用定时器来介绍，本章将分为如下几个部分：

- 13.1 STM32 通用定时器简介
- 13.2 硬件设计
- 13.3 软件设计
- 13.4 下载验证

13.1 STM32 通用定时器简介

STM32 的通用定时器是一个通过可编程预分频器（PSC）驱动的 16 位自动装载计数器（CNT）构成。STM32 的通用定时器可以被用于：测量输入信号的脉冲长度(输入捕获)或者产生输出波形(输出比较和 PWM)等。使用定时器预分频器和 RCC 时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。STM32 的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM3 的通用 TIMx (TIM2、TIM3、TIM4 和 TIM5) 定时器功能包括：

- 1) 16 位向上、向下、向上/向下自动装载计数器 (TIMx_CNT)。
- 2) 16 位可编程(可以实时修改)预分频器(TIMx_PSC)，计数器时钟频率的分频系数为 1~65535 之间的任意数值。
- 3) 4 个独立通道 (TIMx_CH1~4)，这些通道可以用来作为：
 - A. 输入捕获
 - B. 输出比较
 - C. PWM 生成(边缘或中间对齐模式)
 - D. 单脉冲模式输出
- 4) 可使用外部信号 (TIMx_ETR) 控制定时器和定时器互连 (可以用 1 个定时器控制另外一个定时器) 的同步电路。
- 5) 如下事件发生时产生中断/DMA：
 - A. 更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)
 - B. 触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)
 - C. 输入捕获
 - D. 输出比较
 - E. 支持针对定位的增量(正交)编码器和霍尔传感器电路
 - F. 触发输入作为外部时钟或者按周期的电流管理

由于 STM32 通用定时器比较复杂，这里我们不再多介绍，请大家直接参考《STM32 参考手册》第 253 页，通用定时器一章。为了深入了解 STM32 的通用寄存器，下面我们先介绍一下与我们这章的实验密切相关的几个通用定时器的寄存器。

首先是控制寄存器 1 (TIMx_CR1)，该寄存器的各位描述如图 13.1.1 所示：



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CKD[1:0]	ARPE	CMS[1:0]	DIR	OPM	URS	UDIS	CEN		

rw rw

位15:10	保留, 始终读为0。
位9:8	CKD[1:0]: 时钟分频因子 定义在定时器时钟(CK_INT)频率与数字滤波器(ETR,TIx)使用的采样频率之间的分频比例。 00: $t_{DTS} = t_{CK_INT}$ 01: $t_{DTS} = 2 \times t_{CK_INT}$ 10: $t_{DTS} = 4 \times t_{CK_INT}$ 11: 保留
位7	ARPE: 自动重装载预装载允许位 0: TIMx_ARR寄存器没有缓冲; 1: TIMx_ARR寄存器被装入缓冲器。
位6:5	CMS[1:0]: 选择中央对齐模式 00: 边沿对齐模式。计数器依据方向位(DIR)向上或向下计数。 01: 中央对齐模式1。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位, 只在计数器向下计数时被设置。 10: 中央对齐模式2。计数器交替地向上和向下计数。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位, 只在计数器向上计数时被设置。 11: 中央对齐模式3。计数器交替地向上和向下计数。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位, 在计数器向上和向下计数时均被设置。 注: 在计数器开启时(CEN=1), 不允许从边沿对齐模式转换到中央对齐模式。
位4	DIR: 方向 0: 计数器向上计数; 1: 计数器向下计数。 注: 当计数器配置为中央对齐模式或编码器模式时, 该位为只读。
位3	OPM: 单脉冲模式 0: 在发生更新事件时, 计数器不停止; 1: 在发生下一次更新事件(清除CEN位)时, 计数器停止。
位2	URS: 更新请求源 软件通过该位选择UEV事件的源 0: 如果允许产生更新中断或DMA请求, 则下述任一事件产生一个更新中断或DMA请求: <ul style="list-style-type: none"> - 计数器溢出/下溢 - 设置UG位 - 从模式控制器产生的更新 1: 如果允许产生更新中断或DMA请求, 则只有计数器溢出/下溢才产生一个更新中断或DMA请求。
位1	UDIS: 禁止更新 软件通过该位允许/禁止UEV事件的产生 0: 允许UEV。更新(UEV)事件由下述任一事件产生: <ul style="list-style-type: none"> - 计数器溢出/下溢 - 设置UG位 - 从模式控制器产生的更新 被缓存的寄存器被装入它们的预装载值。 1: 禁止UEV。不产生更新事件, 影子寄存器(ARR、PSC、CCRx)保持它们的值。如果设置了UG位或从模式控制器发出了一个硬件复位, 则计数器和预分频器被重新初始化。
位0	CEN: 使能计数器 0: 禁止计数器; 1: 使能计数器。 注: 在软件设置了CEN位后, 外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 在单脉冲模式下, 当发生更新事件时, CEN被自动清除。



图 13.1.1 TIMx_CR1 寄存器各位描述

首先我们来看看 TIMx_CR1 的最低位，也就是计数器使能位，该位必须置 1，才能让定时器开始计数。从第 4 位 DIR 可以看出默认的计数方式是向上计数，同时也可以向下计数，第 5,6 位是设置计数对齐方式的。从第 8 和第 9 位可以看出，我们还可以设置定时器的时钟分频因子为 1,2,4。接下来介绍第二个与我们这章密切相关的寄存器：DMA/中断使能寄存器（TIMx_DIER）。该寄存器是一个 16 位的寄存器，其各位描述如图 13.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	TDE	保留	CC4DE	CC3DE	CC2DE	CC1DE	UDE	保留	TIE	保留	CC4IE	CC3IE	CC2IE	CC1IE	UIE

RW RW

图 13.1.2 TIMx_DIER 寄存器各位描述

这里我们同样仅关心它的第 0 位，该位是更新中断允许位，本章用到的是定时器的更新中断，所以该位要设置为 1，来允许由于更新事件所产生的中断。

接下来我们看第三个与我们这章有关的寄存器：预分频寄存器（TIMx_PSC）。该寄存器用设置对时钟进行分频，然后提供给计数器，作为计数器的时钟。该寄存器的各位描述如图 13.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PSC[15:0]																
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	
位15:0		PSC[15:0]: 预分频器的值 计数器的时钟频率CK_CNT等于 $f_{CK_PSC}/(PSC[15:0]+1)$ 。 PSC包含了当更新事件产生时装入当前预分频器寄存器的值。														

图 13.1.3 TIMx_PSC 寄存器各位描述

这里，定时器的时钟来源有 4 个：

- 1) 内部时钟（CK_INT）
- 2) 外部时钟模式 1：外部输入脚（TIx）
- 3) 外部时钟模式 2：外部触发输入（ETR）
- 4) 内部触发输入（ITRx）：使用 A 定时器作为 B 定时器的预分频器（A 为 B 提供时钟）。

这些时钟，具体选择哪个可以通过 TIMx_SMCR 寄存器的相关位来设置。这里的 CK_INT 时钟是从 APB1 倍频的来的，除非 APB1 的时钟分频数设置为 1，否则通用定时器 TIMx 的时钟是 APB1 时钟的 2 倍，当 APB1 的时钟不分频的时候，通用定时器 TIMx 的时钟就等于 APB1 的时钟。这里还要注意的就是高级定时器的时钟不是来自 APB1，而是来自 APB2 的。

这里顺带介绍一下 TIMx_CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前定时器的计数值。

接着我们介绍自动重装载寄存器（TIMx_ARR），该寄存器在物理上实际对应着 2 个寄存器。一个是程序员可以直接操作的，另外一个是程序员看不到的，这个看不到的寄存器在《STM32 参考手册》里面被叫做影子寄存器。事实上真正起作用的是影子寄存器。根据 TIMx_CR1 寄存器中 APRE 位的设置：APRE=0 时，预装载寄存器的内容可以随时传送到影子寄存器，此时 2 者是连通的；而 APRE=1 时，在每一次更新事件（UEV）时，才把预装在寄存器的内容传送到影子寄存器。

自动重装载寄存器的各位描述如图 13.1.4 所示：



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ARR[15:0]																
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
位15:0		ARR[15:0]: 自动重装载的值 ARR包含了将要装载入实际的自动重装载寄存器的数值。 当自动重装载的值为空时，计数器不工作。														

图 13.1.4 TIMx_ARR 寄存器各位描述

最后，我们要介绍的寄存器是：状态寄存器（TIMx_SR）。该寄存器用来标记当前与定时器相关的各种事件/中断是否发生。该寄存器的各位描述如图 13.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	CC4OF	CC3OF	CC2OF	CC1OF	保留	TIF	保留	CC4IF	CC3IF	CC2IF	CC1IF	UIF			
	rc w0		rc w0												

图 13.1.5 TIMx_SR 寄存器各位描述

关于这些位的详细描述，请参考《STM32 参考手册》第 282 页。

只要对以上几个寄存器进行简单的设置，我们就可以使用通用定时器了，并且可以产生中断。

这一章，我们将使用定时器产生中断，然后在中断服务函数里面翻转 DS1 上的电平，来指示定时器中断的产生。接下来我们以通用定时器 TIM3 为实例，来说明要经过哪些步骤，才能达到这个要求，并产生中断。这里我们就对每个步骤通过库函数的实现方式来描述。首先要提到的是，定时器相关的库函数主要集中在固件库文件 stm32f10x_tim.h 和 stm32f10x_tim.c 文件中。

1) TIM3 时钟使能。

TIM3 是挂载在 APB1 之下，所以我们通过 APB1 总线下的使能使能函数来使能 TIM3。调用的函数是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //时钟使能
```

2) 初始化定时器参数, 设置自动重装值, 分频系数, 计数方式等。

在库函数中，定时器的初始化参数是通过初始化函数 TIM_TimeBaseInit 实现的：

```
voidTIM_TimeBaseInit(TIM_TypeDef*TIMx,
                      TIM_TimeBaseInitTypeDef*TIM_TimeBaseInitStruct);
```

第一个参数是确定是哪个定时器，这个比较容易理解。第二个参数是定时器初始化参数结构体指针，结构体类型为 TIM_TimeBaseInitTypeDef，下面我们看看这个结构体的定义：

```
typedef struct
{
    uint16_t TIM_Prescaler;
    uint16_t TIM_CounterMode;
    uint16_t TIM_Period;
    uint16_t TIM_ClockDivision;
    uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;
```

这个结构体一共有 5 个成员变量，要说明的是，对于通用定时器只有前面四个参数有用，最后一个参数 TIM_RepetitionCounter 是高级定时器才有的，这里不多解释。

第一个参数 TIM_Prescaler 是用来设置分频系数的，刚才上面有讲解。



第二个参数 `TIM_CounterMode` 是用来设置计数方式，上面讲解过，可以设置为向上计数，向下计数方式还有中央对齐计数方式，比较常用的是向上计数模式 `TIM_CounterMode_Up` 和向下计数模式 `TIM_CounterMode_Down`。

第三个参数是设置自动重载计数周期值，这在前面也已经讲解过。

第四个参数是用来设置时钟分频因子。

针对 `TIM3` 初始化范例代码格式：

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;  
TIM_TimeBaseStructure.TIM_Period = 5000;  
TIM_TimeBaseStructure.TIM_Prescaler = 7199;  
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;  
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;  
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
```

3) 设置 `TIM3_DIER` 允许更新中断。

因为我们要使用 `TIM3` 的更新中断，寄存器的相应位便可使能更新中断。在库函数里面定时器中断使能是通过 `TIM_ITConfig` 函数来实现的：

```
void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState);
```

第一个参数是选择定时器号，这个容易理解，取值为 `TIM1~TIM17`。

第二个参数非常关键，是用来指明我们使能的定时器中断的类型，定时器中断的类型有很多，包括更新中断 `TIM_IT_Update`，触发中断 `TIM_IT_Trigger`，以及输入捕获中断等等。

第三个参数就很简单了，就是失能还是使能。

例如我们要使能 `TIM3` 的更新中断，格式为：

```
TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);
```

4) `TIM3` 中断优先级设置。

在定时器中断使能之后，因为要产生中断，必不可少的要设置 `NVIC` 相关寄存器，设置中断优先级。之前多次讲解到用 `NVIC_Init` 函数实现中断优先级的设置，这里就不重复讲解。

5) 允许 `TIM3` 工作，也就是使能 `TIM3`。

光配置好定时器还不行，没有开启定时器，照样不能用。我们在配置完后要开启定时器，通过 `TIM3_CR1` 的 `CEN` 位来设置。在固件库里面使能定时器的函数是通过 `TIM_Cmd` 函数来实现的：

```
void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState)
```

这个函数非常简单，比如我们要使能定时器 3，方法为：

```
TIM_Cmd(TIM3, ENABLE); //使能 TIMx 外设
```

6) 编写中断服务函数。

在最后，还是要编写定时器中断服务函数，通过该函数来处理定时器产生的相关中断。在中断产生后，通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作，我们这里使用的是更新（溢出）中断，所以在状态寄存器 `SR` 的最低位。在处理完中断之后应该向 `TIM3_SR` 的最低位写 0，来清除该中断标志。

在固件库函数里面，用来读取中断状态寄存器的值判断中断类型的函数是：

```
ITStatus TIM_GetITStatus(TIM_TypeDef* TIMx, uint16_t)
```

该函数的作用是，判断定时器 `TIMx` 的中断类型 `TIM_IT` 是否发生中断。比如，我们要判断定时器 3 是否发生更新（溢出）中断，方法为：

```
if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET){}
```

固件库中清除中断标志位的函数是：



```
void TIM_ClearITPendingBit(TIM_TypeDef* TIMx, uint16_t TIM_IT)
```

该函数的作用是，清除定时器 TIMx 的中断 TIM_IT 标志位。使用起来非常简单，比如我们在 TIM3 的溢出中断发生后，我们要清除中断标志位，方法是：

```
TIM_ClearITPendingBit(TIM3, TIM_IT_Update);
```

这里需要说明一下，固件库还提供了两个函数用来判断定时器状态以及清除定时器状态标志位的函数 TIM_GetFlagStatus 和 TIM_ClearFlag，他们的作用和前面两个函数的作用类似。只是在 TIM_GetITStatus 函数中会先判断这种中断是否使能，使能了才去判断中断标志位，而 TIM_GetFlagStatus 直接用来判断状态标志位。

通过以上几个步骤，我们就可以达到我们的目的了，使用通用定时器的更新中断，来控制 DS1 的亮灭。

13.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0 和 DS1
- 2) 定时器 TIM3

本章将通过 TIM3 的中断来控制 DS1 的亮灭，DS1 是直接连接到 PE5 上的，这个前面已经有介绍了。而 TIM3 属于 STM32 的内部资源，只需要软件设置即可正常工作。

13.3 软件设计

软件设计我们直接打开我们光盘实验 8 定时器中断实验即可。我们可以看到我们的工程中的 HARDWARE 下面比以前多了一个 time.c 文件（包括头文件 time.h），这两个文件是我们自己编写。同时还引入了定时器相关的固件库函数文件 stm32f10x_tim.c 和头文件 stm32f10x_tim.h。下面我们来看看我们的 time.c 文件。

time.c 文件代码：

```
#include "timer.h"
#include "led.h"

//通用定时器 3 中断初始化
//这里时钟选择为 APB1 的 2 倍，而 APB1 为 36M
//arr：自动重装值。
//psc：时钟预分频数
//这里使用的是定时器 3！
void TIM3_Int_Init(u16 arr,u16 psc)
{
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //①时钟 TIM3 使能

    //定时器 TIM3 初始化
    TIM_TimeBaseStructure.TIM_Period = arr;      //设置自动重装载寄存器周期的值
    TIM_TimeBaseStructure.TIM_Prescaler = psc; //设置时钟频率除数的预分频值
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; //设置时钟分割
```

```

TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);           //②初始化 TIM3
TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE );                //③允许更新中断

//中断优先级 NVIC 设置
NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;          //TIM3 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //先占优先级 0 级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;        //从优先级 3 级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;           //IRQ 通道被使能
NVIC_Init(&NVIC_InitStructure);                         //④初始化 NVIC 寄存器

TIM_Cmd(TIM3, ENABLE);                                   //⑤使能 TIM3

}

//定时器 3 中断服务程序⑥
void TIM3_IRQHandler(void) //TIM3 中断
{
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) //检查 TIM3 更新中断发生与否
    {
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update ); //清除 TIM3 更新中断标志
        LED1=!LED1;
    }
}

```

该文件下包含一个中断服务函数和一个定时器 3 中断初始化函数，中断服务函数比较简单，在每次中断后，判断 TIM3 的中断类型，如果中断类型正确（溢出中断），则执行 LED1（DS1）的取反。

TIM3_Int_Init()函数就是执行我们上面 13.1 节介绍的那 6 个步骤，我们分别用标号①~⑥来标注，该函数的 2 个参数用来设置 TIM3 的溢出时间。在前面时钟系统部分我们讲解过，系统初始化的时候在默认的系统初始化函数 SystemInit 函数里面已经初始化 APB1 的时钟为 2 分频，所以 APB1 的时钟为 36M，而从 STM32 的内部时钟树图得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 的时钟频率将为 APB1 时钟的两倍。因此，TIM3 的时钟为 72M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

$$Tout = ((arr+1)*(psc+1))/Tclk;$$

其中：

Tclk：TIM3 的输入时钟频率（单位为 Mhz）。

Tout：TIM3 溢出时间（单位为 us）。

timer.h 文件的代码非常简单，一些函数申明，这里就不讲解。

最后，我们在主程序里面输入如下代码：

```

int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
}

```



```
uart_init(9600);          //串口初始化波特率为 9600
LED_Init();                //LED 端口初始化
TIM3_Int_Init(4999,7199); //10Khz 的计数频率, 计数到 5000 为 500ms
while(1)
{
    LED0=!LED0;
    delay_ms(200);
}
```

这里的代码和之前大同小异，此段代码对 TIM3 进行初始化之后，进入死循环等待 TIM3 溢出中断，当 TIM3_CNT 的值等于 TIM3_ARR 的值的时候，就会产生 TIM3 的更新中断，然后在中断里面取反 LED1，TIM3_CNT 再从 0 开始计数。根据上面的公式，我们可以算出中断溢出时间为 500ms。 $T_{out} = ((4999+1)*(7199+1))/72 = 500000\mu s = 500ms$ 。

13.4 下载验证

在完成软件设计之后，我们将编译好的文件下载到战舰 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停闪烁（每 400ms 闪烁一次），而 DS1 也是不停的闪烁，但是闪烁时间较 DS0 慢（1s 一次）。

第十四章 PWM 输出实验

上一章，我们介绍了 STM32 的通用定时器 TIM3，用该定时器的中断来控制 DS1 的闪烁，这一章，我们将向大家介绍如何使用 STM32 的 TIM3 来产生 PWM 输出。在本章中，我们将利用 TIM3 的通道 2，把通道 2 重映射到 PB5，产生 PWM 来控制 DS0 的亮度。本章分为如下几个部分：

- 14.1 PWM 简介
- 14.2 硬件设计
- 14.3 软件设计
- 14.4 下载验证



14.1 PWM 简介

脉冲宽度调制(PWM)，是英文“Pulse Width Modulation”的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。简单一点，就是对脉冲宽度的控制。

STM32 的定时器除了 TIM6 和 7。其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出，这样，STM32 最多可以同时产生 30 路 PWM 输出！这里我们仅利用 TIM3 的 CH2 产生一路 PWM 输出。如果要产生多路输出，大家可以根据我们的代码稍作修改即可。

同样，我们首先通过对 PWM 相关的寄存器进行讲解，大家了解了定时器 TIM3 的 PWM 原理之后，我们再讲解怎么使用库函数产生 PWM 输出。

要使 STM32 的通用定时器 TIMx 产生 PWM 输出，除了上一章介绍的寄存器外，我们还会用到 3 个寄存器，来控制 PWM 的。这三个寄存器分别是：捕获/比较模式寄存器 (TIMx_CCMR1/2)、捕获/比较使能寄存器 (TIMx_CCER)、捕获/比较寄存器 (TIMx_CCR1~4)。接下来我们简单介绍一下这三个寄存器。

首先是捕获/比较模式寄存器 (TIMx_CCMR1/2)，该寄存器总共有 2 个，TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 2，而 TIMx_CCMR2 控制 CH3 和 4。该寄存器的各位描述如图 14.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]	OC2PE	OC2FE	CC2S[1:0]	OC1CE	OC1M[2:0]	OC1PE	OC1FE	CC1S[1:0]						
IC2F[3:0]		IC2PSC[1:0]			IC1F[3:0]		IC1PSC[1:0]								

RW RW

图 14.1.1 TIMx_CCMR1 寄存器各位描述

该寄存器的有些位在不同模式下，功能不一样，所以在图 14.1.1 中，我们把寄存器分了 2 层，上面一层对应输出而下面的则对应输入。关于该寄存器的详细说明，请参考《STM32 参考手册》第 288 页，14.4.7 一节。这里我们需要说明的是模式设置位 OCxM，此部分由 3 位组成。总共可以配置成 7 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110/111。这两种 PWM 模式的区别就是输出电平的极性相反。

接下来，我们介绍捕获/比较使能寄存器 (TIMx_CCER)，该寄存器控制着各个输入输出通道的开关。该寄存器的各位描述如图 14.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	CC4P	CC4E	保留	CC3P	CC3E	保留	CC2P	CC2E	保留	CC1P	CC1E				

RW RW

图 14.1.2 TIMx_CCER 寄存器各位描述

该寄存器比较简单，我们这里只用到了 CC2E 位，该位是输入/捕获 2 输出使能位，要想 PWM 从 IO 口输出，这个位必须设置为 1，所以我们需要设置该位为 1。该寄存器更详细的介绍了，请参考《STM32 参考手册》第 292 页，14.4.9 这一节。

最后，我们介绍一下捕获/比较寄存器 (TIMx_CCR1~4)，该寄存器总共有 4 个，对应 4 个输出通道 CH1~4。因为这 4 个寄存器都差不多，我们仅以 TIMx_CCR1 为例介绍，该寄存器的各位描述如图 14.1.3 所示：



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位15:0		CCR1[15:0]: 捕获/比较1的值 若CC1通道配置为输出： CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载特性，写入的数值会立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC1端口上产生输出信号。 若CC1通道配置为输入： CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。													

图 14.1.3 寄存器 TIMx_CCR1 各位描述

在输出模式下，该寄存器的值与 CNT 的值比较，根据比较结果产生相应动作。利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的输出脉宽了。本章，我们使用的是 TIM3 的通道 2，所以我们需要修改 TIM3_CCR2 以实现脉宽控制 DS0 的亮度。

我们要利用 TIM3 的 CH2 输出 PWM 来控制 DS0 的亮度，但是 TIM3_CH2 默认是接在 PA7 上面的，而我们的 DS0 接在 PB5 上面，如果普通 MCU，可能就只能用飞线把 PA7 飞到 PB5 上来实现了，不过，我们用的是 STM32，它比较高级，可以通过重映射功能，把 TIM3_CH2 映射到 PB5 上。

STM32 的重映射控制是由复用重映射和调试 IO 配置寄存器 (AFIO_MAPR) 控制的，该寄存器的各位描述如图 14.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				SWJ_CFG[2:0]			保留			ADC2_E	ADC2_E	ADC1_E	ADC1_E	TIM5CH	
W	W	W					TRGREG	TRGINJ	TRGREG	TRGREG	TRGINJ	TRGREG	TRGINJ	4_IREM	AP
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PD01_REMAP	CAN_REMAP[1:0]	TIM4_REMAP	TIM3_REMAP[1:0]	TIM2_REMAP[1:0]	TIM1_REMAP[1:0]	USART3_REMAP[1:0]	USART2_REMAP[1:0]	USART1_REMAP[1:0]	I2C1_REMAP	SPI1_REMAP					
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

图 14.1.4 寄存器 AFIO_MAPR 各位描述

我们这里用到的是 TIM3 的重映射，从上图可以看出，TIM3_REMAP 是由[11:10]这 2 个位控制的。TIM3_REMAP[1:0]重映射控制表如表 14.1.1 所示：

复用功能	TIM3_REMAP[1:0] = 00 (没有重映像)	TIM3_REMAP[1:0] = 10 (部分重映像)	TIM3_REMAP[1:0] = 11 (完全重映像) ⁽¹⁾
TIM3_CH1	PA6	PB4	PC6
TIM3_CH2	PA7	PB5	PC7
TIM3_CH3	PB0		PC8
TIM3_CH4	PB1		PC9

表 14.1.1 TIM3_REMAP 重映射控制表

默认条件下，TIM3_REMAP[1:0]为 00，是没有重映射的，所以 TIM3_CH1~TIM3_CH4 分别是接在 PA6、PA7、PB0 和 PB1 上的，而我们想让 TIM3_CH2 映射到 PB5 上，则需要设置 TIM3_REMAP[1:0]=10，即部分重映射，这里需要注意，此时 TIM3_CH1 也被映射到 PB4 上了。

至此，我们把本章要用的几个相关寄存器都介绍完了，本章要实现通过重映射 TIM3_CH2 到 PB5 上，由 TIM3_CH2 输出 PWM 来控制 DS0 的亮度。下面我们介绍通过库函数来配置该功能的步骤。



首先要提到的是，PWM 相关的函数设置在库函数文件 `stm32f10x_tim.h` 和 `stm32f10x_tim.c` 文件中。

1) 开启 TIM3 时钟以及复用功能时钟，配置 PB5 为复用输出。

要使用 TIM3，我们必须先开启 TIM3 的时钟，这点相信大家看了这么多代码，应该明白了。这里我们还要配置 PB5 为复用输出，这是因为 TIM3_CH2 通道将重映射到 PB5 上，此时，PB5 属于复用功能输出。库函数使能 TIM3 时钟的方法是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //使能定时器3时钟
```

这在前面一章已经提到过。库函数设置 AFIO 时钟的方法是：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); //复用时钟使能
```

这两行代码很容易组织，这里不做过多重复的讲解。设置 PB5 为复用功能输出的方法在前面的几个实验都有类似的讲解，相信大家很明白，这里简单列出 GPIO 初始化的一行代码即可：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
```

2) 设置 TIM3_CH2 重映射到 PB5 上。

因为 TIM3_CH2 默认是接在 PA7 上的，所以我们需要设置 TIM3_REMAP 为部分重映射（通过 AFIO_MAPR 配置），让 TIM3_CH2 重映射到 PB5 上面。在库函数函数里面设置重映射的函数是：

```
void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
```

在前面 STM32 重映射章节 4.4.2 已经讲解过，STM32 重映射只能重映射到特定的端口。第一个入口参数可以理解为设置重映射的类型，比如 TIM3 部分重映射入口参数为

`GPIO_PartialRemap_TIM3`，这点可以顾名思义了。所以 TIM3 部分重映射的库函数实现方法是：

```
GPIO_PinRemapConfig(GPIO_PartialRemap_TIM3, ENABLE);
```

3) 初始化 TIM3，设置 TIM3 的 ARR 和 PSC。

在开启了 TIM3 的时钟之后，我们要设置 ARR 和 PSC 两个寄存器的值来控制输出 PWM 的周期。当 PWM 周期太慢（低于 50Hz）的时候，我们就会明显感觉到闪烁了。因此，PWM 周期在这里不宜设置的太小。这在库函数是通过 `TIM_TimeBaseInit` 函数实现的，在上一节定时器中断章节我们已经有讲解，这里就不详细讲解，调用的格式为：

```
TIM_TimeBaseStructure.TIM_Period = arr; //设置自动重装载值
```

```
TIM_TimeBaseStructure.TIM_Prescaler = psc; //设置预分频值
```

```
TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDTS = Tck_tim
```

```
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式
```

```
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //根据指定的参数初始化TIMx的
```

4) 设置 TIM3_CH2 的 PWM 模式，使能 TIM3 的 CH2 输出。

接下来，我们要设置 TIM3_CH2 为 PWM 模式（默认是冻结的），因为我们的 DS0 是低电平亮，而我们希望当 CCR2 的值小的时候，DS0 就暗，CCR2 值大的时候，DS0 就亮，所以我们要通过配置 TIM3_CCMR1 的相关位来控制 TIM3_CH2 的模式。在库函数中，PWM 通道设置是通过函数 `TIM_OC1Init()`~`TIM_OC4Init()` 来设置的，不同的通道的设置函数不一样，这里我们使用的是通道 2，所以使用的函数是 `TIM_OC2Init()`。

```
void TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCIInitTypeDef* TIM_OCIInitStruct);
```

这种初始化格式大家学到这里应该也熟悉了，所以我们直接来看看结构体 `TIM_OCIInitTypeDef` 的定义：

```
typedef struct
{
    uint16_t TIM_OCMode;
```



```

    uint16_t TIM_OutputState;
    uint16_t TIM_OutputNState; */
    uint16_t TIM_Pulse;
    uint16_t TIM_OCPolarity;
    uint16_t TIM_OCNPolarity;
    uint16_t TIM_OCIdleState;
    uint16_t TIM_OCNIdleState;
} TIM_OCInitTypeDef;

```

这里我们讲解一下与我们要求相关的几个成员变量：

参数 TIM_OCMODE 设置模式是 PWM 还是输出比较，这里我们是 PWM 模式。

参数 TIM_OutputState 用来设置比较输出使能，也就是使能 PWM 输出到端口。

参数 TIM_OCPolarity 用来设置极性是高还是低。

其他的参数 TIM_OutputNState, TIM_OCNPolarity, TIM_OCIdleState 和 TIM_OCNIdleState 是高级定时器 TIM1 和 TIM8 才用到的。

要实现我们上面提到的场景，方法是：

```

TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMODE = TIM_OCMODE_PWM2; //选择 PWM 模式 2
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; //输出极性高
TIM_OC2Init(TIM3, &TIM_OCInitStructure); //初始化 TIM3 OC2

```

5) 使能 TIM3。

在完成以上设置了之后，我们需要使能 TIM3。使能 TIM3 的方法前面已经讲解过：

```
TIM_Cmd(TIM3, ENABLE); //使能 TIM3
```

6) 修改 TIM3_CCR2 来控制占空比。

最后，在经过以上设置之后，PWM 其实已经开始输出了，只是其占空比和频率都是固定的，而我们通过修改 TIM3_CCR2 则可以控制 CH2 的输出占空比。继而控制 DS0 的亮度。

在库函数中，修改 TIM3_CCR2 占空比的函数是：

```
void TIM_SetCompare2(TIM_TypeDef* TIMx, uint16_t Compare2);
```

理所当然，对于其他通道，分别有一个函数名字，函数格式为 TIM_SetComparex(x=1,2,3,4)。

通过以上 6 个步骤，我们就可以控制 TIM3 的 CH2 输出 PWM 波了。

14.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 定时器 TIM3

这两个前面都有介绍，但是我们这里用到了 TIM3 的部分重映射功能，把 TIM3_CH2 直接映射到了 PB5 上，而通过前面的学习，我们知道 PB5 和 DS0 是直接连接的，所以电路上并没有任何变化。

14.3 软件设计

打开光盘里面的 PWM 输出实验代码可以看到相对上一章实验，我们在 timer.c 里面加入了如下代码：



```
//TIM3 PWM 部分初始化
//PWM 输出初始化
//arr: 自动重装值
//psc: 时钟预分频数
void TIM3_PWM_Init(u16 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //①使能定时器 3 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|
    RCC_APB2Periph_AFIO, ENABLE); //①使能 GPIO 和 AFIO 复用功能时钟

    GPIO_PinRemapConfig(GPIO_PartialRemap_TIM3, ENABLE); //②重映射 TIM3_CH2->PB5

    //设置该引脚为复用输出功能,输出 TIM3 CH2 的 PWM 脉冲波形  GPIOB.5
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //TIM_CH2
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure); //①初始化 GPIO

    //初始化 TIM3
    TIM_TimeBaseStructure.TIM_Period = arr; //设置在自动重装载周期值
    TIM_TimeBaseStructure.TIM_Prescaler = psc; //设置预分频值
    TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDTs = Tck_tim
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数模式
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //③初始化 TIMx

    //初始化 TIM3 Channel2 PWM 模式
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2; //选择 PWM 模式 2
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; //输出极性高
    TIM_OC2Init(TIM3, &TIM_OCInitStructure); //④初始化外设 TIM3 OC2

    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable); //使能预装载寄存器
    TIM_Cmd(TIM3, ENABLE); //⑤使能 TIM3
}
```

此部分代码包含了上面介绍的 PWM 输出设置的前 5 个步骤分别用标号①~⑤备注。这里我们关于 TIM3 的设置就不再说了，这里提醒下：在配置 AFIO 相关寄存器的时候，必须先开启辅助功能时钟。

头文件 timer.h 与上一章的不同是加入了 TIM3_PWM_Init 的声明,这个就没什么需要讲解

咯。

接下来，我们修改主程序里面的 main 函数如下：

```

int main(void)
{
    u16 led0pwmval=0;
    u8 dir=1;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    TIM3_PWM_Init(899,0);   //不分频,PWM 频率=72000/900=8Khz
    while(1)
    {
        delay_ms(10);
        if(dir)led0pwmval++;
        else led0pwmval--;
        if(led0pwmval>300)dir=0;
        if(led0pwmval==0)dir=1;
        TIM_SetCompare2(TIM3,led0pwmval);
    }
}

```

这里，我们从死循环函数可以看出，我们将 led0pwmval 这个值设置为 PWM 比较值，也就是通过 led0pwmval 来控制 PWM 的占空比，然后控制 led0pwmval 的值从 0 变到 300，然后又从 300 变到 0，如此循环，因此 DS0 的亮度也会跟着从暗变到亮，然后又从亮变到暗。至于这里的值，我们为什么取 300，是因为 PWM 的输出占空比达到这个值的时候，我们的 LED 亮度变化就不大了（虽然最大值可以设置到 899），因此设计过大的值在这里是没必要的。至此，我们的软件设计就完成了。

14.4 下载验证

在完成软件设计之后，将我们将编译好的文件下载到战舰 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停的由暗变到亮，然后又从亮变到暗。每个过程持续时间大概为 3 秒钟左右。

实际运行结果如下图 14.4.1 所示：



较暗



较亮

图 14.4.1 PWM 控制 DS0 亮度

第十五章 输入捕获实验

上一章，我们介绍了 STM32 的通用定时器作为 PWM 输出的使用方法，这一章，我们将向大家介绍通用定时器作为输入捕获的使用。在本章中，我们将用 TIM5 的通道 1 (PA0) 来做输入捕获，捕获 PA0 上高电平的脉宽（用 WK_UP 按键输入高电平），通过串口打印高电平脉宽时间，从本章分为如下几个部分：

- 15.1 输入捕获简介
- 15.2 硬件设计
- 15.3 软件设计
- 15.4 下载验证



15.1 输入捕获简介

输入捕获模式可以用来测量脉冲宽度或者测量频率。STM32 的定时器，除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。STM32 的输入捕获，简单的说就是通过检测 TIMx_CHx 上的边沿信号，在边沿信号发生跳变（比如上升沿/下降沿）的时候，将当前定时器的值（TIMx_CNT）存放到对应的通道的捕获/比较寄存器（TIMx_CCRx）里面，完成一次捕获。同时还可以配置捕获时是否触发中断/DMA 等。

本章我们用到 TIM5_CH1 来捕获高电平脉宽，也就是要先设置输入捕获为上升沿检测，记录发生上升沿的时候 TIM5_CNT 的值。然后配置捕获信号为下降沿捕获，当下降沿到来时，发生捕获，并记录此时的 TIM5_CNT 值。这样，前后两次 TIM5_CNT 之差，就是高电平的脉宽，同时 TIM5 的计数频率我们是知道的，从而可以计算出高电平脉宽的准确时间。

接下来，我们介绍我们本章需要用到的一些寄存器配置，需要用到的寄存器有：TIMx_ARR、TIMx_PSC、TIMx_CCMR1、TIMx_CCER、TIMx_DIER、TIMx_CR1、TIMx_CCR1 这些寄存器在前面 2 章全部都有提到（这里的 x=5），我们这里就不再全部罗列了，我们这里针对性的介绍这几个寄存器的配置。

首先 TIMx_ARR 和 TIMx_PSC，这两个寄存器用来设自动重装载值和 TIMx 的时钟分频，用法同前面介绍的，我们这里不再介绍。

再来看看捕获/比较模式寄存器 1：TIMx_CCMR1，这个寄存器在输入捕获的时候，非常有用，有必要重新介绍，该寄存器的各位描述如图 15.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]	OC2PE	OC2FE	CC2S[1:0]	OC1CE	OC1M[2:0]	OC1PE	OC1FE	CC1S[1:0]						
IC2F[3:0]	IC2PSC[1:0]				IC1F[3:0]	IC1PSC[1:0]									

RW RW

图 15.1.1 TIMx_CCMR1 寄存器各位描述

当在输入捕获模式下使用的时候，对应图 15.1.1 的第二行描述，从图中可以看出，TIMx_CCMR1 明显是针对 2 个通道的配置，低八位[7:0]用于捕获/比较通道 1 的控制，而高八位[15:8]则用于捕获/比较通道 2 的控制，因为 TIMx 还有 CCMR2 这个寄存器，所以可以知道 CCMR2 是用来控制通道 3 和通道 4（详见《STM32 参考手册》290 页，14.4.8 节）。

这里我们用到的是 TIM5 的捕获/比较通道 1，我们重点介绍 TIMx_CMMR1 的[7:0]位（其实高 8 位配置类似），TIMx_CMMR1 的[7:0]位详细描述见图 15.1.2 所示：



位7:4	IC1F[3:0]: 输入捕获1滤波器 (Input capture 1 filter) 这几位定义了TI1输入的采样频率及数字滤波器长度。数字滤波器由一个事件计数器组成，它记录到N个事件后会产生一个输出的跳变： <table border="0"> <tr><td>0000: 无滤波器, 以f_{DTS}采样</td><td>1000: 采样频率$f_{SAMPLING}=f_{DTS}/8$, N=6</td></tr> <tr><td>0001: 采样频率$f_{SAMPLING}=f_{CK_INT}$, N=2</td><td>1001: 采样频率$f_{SAMPLING}=f_{DTS}/8$, N=8</td></tr> <tr><td>0010: 采样频率$f_{SAMPLING}=f_{CK_INT}$, N=4</td><td>1010: 采样频率$f_{SAMPLING}=f_{DTS}/16$, N=5</td></tr> <tr><td>0011: 采样频率$f_{SAMPLING}=f_{CK_INT}$, N=8</td><td>1011: 采样频率$f_{SAMPLING}=f_{DTS}/16$, N=6</td></tr> <tr><td>0100: 采样频率$f_{SAMPLING}=f_{DTS}/2$, N=6</td><td>1100: 采样频率$f_{SAMPLING}=f_{DTS}/16$, N=8</td></tr> <tr><td>0101: 采样频率$f_{SAMPLING}=f_{DTS}/2$, N=8</td><td>1101: 采样频率$f_{SAMPLING}=f_{DTS}/32$, N=5</td></tr> <tr><td>0110: 采样频率$f_{SAMPLING}=f_{DTS}/4$, N=6</td><td>1110: 采样频率$f_{SAMPLING}=f_{DTS}/32$, N=6</td></tr> <tr><td>0111: 采样频率$f_{SAMPLING}=f_{DTS}/4$, N=8</td><td>1111: 采样频率$f_{SAMPLING}=f_{DTS}/32$, N=8</td></tr> </table> <p>注：在现在的芯片版本中，当ICxF[3:0]=1、2或3时，公式中的f_{DTS}由CK_INT替代。</p>	0000: 无滤波器, 以 f_{DTS} 采样	1000: 采样频率 $f_{SAMPLING}=f_{DTS}/8$, N=6	0001: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=2	1001: 采样频率 $f_{SAMPLING}=f_{DTS}/8$, N=8	0010: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=4	1010: 采样频率 $f_{SAMPLING}=f_{DTS}/16$, N=5	0011: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=8	1011: 采样频率 $f_{SAMPLING}=f_{DTS}/16$, N=6	0100: 采样频率 $f_{SAMPLING}=f_{DTS}/2$, N=6	1100: 采样频率 $f_{SAMPLING}=f_{DTS}/16$, N=8	0101: 采样频率 $f_{SAMPLING}=f_{DTS}/2$, N=8	1101: 采样频率 $f_{SAMPLING}=f_{DTS}/32$, N=5	0110: 采样频率 $f_{SAMPLING}=f_{DTS}/4$, N=6	1110: 采样频率 $f_{SAMPLING}=f_{DTS}/32$, N=6	0111: 采样频率 $f_{SAMPLING}=f_{DTS}/4$, N=8	1111: 采样频率 $f_{SAMPLING}=f_{DTS}/32$, N=8
0000: 无滤波器, 以 f_{DTS} 采样	1000: 采样频率 $f_{SAMPLING}=f_{DTS}/8$, N=6																
0001: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=2	1001: 采样频率 $f_{SAMPLING}=f_{DTS}/8$, N=8																
0010: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=4	1010: 采样频率 $f_{SAMPLING}=f_{DTS}/16$, N=5																
0011: 采样频率 $f_{SAMPLING}=f_{CK_INT}$, N=8	1011: 采样频率 $f_{SAMPLING}=f_{DTS}/16$, N=6																
0100: 采样频率 $f_{SAMPLING}=f_{DTS}/2$, N=6	1100: 采样频率 $f_{SAMPLING}=f_{DTS}/16$, N=8																
0101: 采样频率 $f_{SAMPLING}=f_{DTS}/2$, N=8	1101: 采样频率 $f_{SAMPLING}=f_{DTS}/32$, N=5																
0110: 采样频率 $f_{SAMPLING}=f_{DTS}/4$, N=6	1110: 采样频率 $f_{SAMPLING}=f_{DTS}/32$, N=6																
0111: 采样频率 $f_{SAMPLING}=f_{DTS}/4$, N=8	1111: 采样频率 $f_{SAMPLING}=f_{DTS}/32$, N=8																
位3:2	IC1PSC[1:0]: 输入/捕获1预分频器 (Input capture 1 prescaler) 这2位定义了CC1输入(IC1)的预分频系数。 一旦 $CC1E='0'$ (TIMx_CCER寄存器中)，则预分频器复位。 00: 无预分频器，捕获输入口上检测到的每一个边沿都触发一次捕获； 01: 每2个事件触发一次捕获； 10: 每4个事件触发一次捕获； 11: 每8个事件触发一次捕获。																
位1:0	CC1S[1:0]: 捕获/比较1选择 (Capture/Compare 1 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00: CC1通道被配置为输出； 01: CC1通道被配置为输入，IC1映射在TI1上； 10: CC1通道被配置为输入，IC1映射在TI2上； 11: CC1通道被配置为输入，IC1映射在TRC上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 <p>注：CC1S仅在通道关闭时(TIMx_CCER寄存器的$CC1E='0'$)才是可写的。</p>																

图 15.1.2 TIMx_CMMR1 [7:0]位详细描述

其中 CC1S[1:0]，这两个位用于 CCR1 的通道配置，这里我们设置 IC1S[1:0]=01，也就是配置 IC1 映射在 TI1 上（关于 IC1，TI1 不明白的，可以看《STM32 参考手册》14.2 节的图 98-通用定时器框图），即 CC1 对应 TIMx_CH1。

输入捕获 1 预分频器 IC1PSC[1:0]，这个比较好理解。我们是 1 次边沿就触发 1 次捕获，所以选择 00 就是了。

输入捕获 1 滤波器 IC1F[3:0]，这个用来设置输入采样频率和数字滤波器长度。其中， f_{CK_INT} 是定时器的输入频率 (TIMxCLK)，一般为 72Mhz，而 f_{DTS} 则是根据 TIMx_CR1 的 CKD[1:0] 的设置来确定的，如果 CKD[1:0] 设置为 00，那么 $f_{DTS}=f_{CK_INT}$ 。N 值就是滤波长度，举个简单的例子：假设 IC1F[3:0]=0011，并设置 IC1 映射到通道 1 上，且为上升沿触发，那么在捕获到上升沿的时候，再以 f_{CK_INT} 的频率，连续采样到 8 次通道 1 的电平，如果都是高电平，则说明却是一个有效的触发，就会触发输入捕获中断（如果开启了的话）。这样可以滤除那些高电平脉宽低于 8 个采样周期的脉冲信号，从而达到滤波的效果。这里，我们不做滤波处理，所以设置 IC1F[3:0]=0000，只要采集到上升沿，就触发捕获。



再来看看捕获/比较使能寄存器：TIMx_CCER，该寄存器的各位描述见图 14.1.2（在第 14 章）。本章我们要用到这个寄存器的最低 2 位，CC1E 和 CC1P 位。这两个位的描述如图 15.1.3 所示：

位1	CC1P: 输入/捕获1输出极性 (Capture/Compare 1 output polarity) CC1通道配置为输出: 0: OC1高电平有效 1: OC1低电平有效 CC1通道配置为输入: 该位选择是IC1还是IC1的反相信号作为触发或捕获信号。 0: 不反相: 捕获发生在IC1的上升沿; 当用作外部触发器时, IC1不反相。 1: 反相: 捕获发生在IC1的下降沿; 当用作外部触发器时, IC1反相。
位0	CC1E: 输入/捕获1输出使能 (Capture/Compare 1 output enable) CC1通道配置为输出: 0: 关闭— OC1禁止输出。 1: 开启— OC1信号输出到对应的输出引脚。 CC1通道配置为输入: 该位决定了计数器的值是否能捕获入TIMx_CCR1寄存器。 0: 捕获禁止; 1: 捕获使能。

图 15.1.3 TIMx_CCER 最低 2 位描述

所以，要使能输入捕获，必须设置 CC1E=0，而 CC1P 则根据自己的需要来配置。

接下来我们再看看 DMA/中断使能寄存器：TIMx_DIER，该寄存器的各位描述见图 13.1.2（在第 13 章），本章，我们需要用到中断来处理捕获数据，所以必须开启通道 1 的捕获比较中断，即 CC1IE 设置为 1。

控制寄存器：TIMx_CR1，我们只用到了它的最低位，也就是用来使能定时器的，这里前面两章都有介绍，请大家参考前面的章节。

最后再来看看捕获/比较寄存器 1：TIMx_CCR1，该寄存器用来存储捕获发生时，TIMx_CNT 的值，我们从 TIMx_CCR1 就可以读出通道 1 捕获发生时刻的 TIMx_CNT 值，通过两次捕获（一次上升沿捕获，一次下降沿捕获）的差值，就可以计算出高电平脉冲的宽度。

至此，我们把本章要用的几个相关寄存器都介绍完了，本章要实现通过输入捕获，来获取 TIM5_CH1(PA0) 上面的高电平脉冲宽度，并从串口打印捕获结果。下面我们介绍输入捕获的配置步骤：

1) 开启 TIM5 时钟和 GPIOA 时钟，配置 PA0 为下拉输入。

要使用 TIM5，我们必须先开启 TIM5 的时钟。这里我们还要配置 PA0 为下拉输入，因为我们要捕获 TIM5_CH1 上面的高电平脉宽，而 TIM5_CH1 是连接在 PA0 上面的。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE); //使能 TIM5 时钟
```

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
```

这两个函数的使用在前面多次提到，还有 GPIO 初始化，这里也不重复了。

2) 初始化 TIM5，设置 TIM5 的 ARR 和 PSC。

在开启了 TIM5 的时钟之后，我们要设置 ARR 和 PSC 两个寄存器的值来设置输入捕获的自动重装载值和计数频率。这在库函数中是通过 TIM_TimeBaseInit 函数实现的，在上面章节已经讲解过，这里不重复讲解。

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = arr; //设定计数器自动重装值
```



```

TIM_TimeBaseStructure.TIM_Prescaler =psc; //设置预分频值
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; // TDTS = Tck_tim
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数模式
TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure); //根据指定的参数初始化 Tim5

```

3) 设置 TIM5 的输入比较参数，开启输入捕获

输入比较参数的设置包括映射关系，滤波，分频以及捕获方式等。这里我们需要设置通道 1 为输入模式，且 IC1 映射到 TI1(通道 1)上面，并且不使用滤波（提高响应速度）器，上升沿捕获。库函数是通过 TIM_ICInit 函数来初始化输入比较参数的：

```
void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);
```

同样，我们来看看参数设置结构体 TIM_ICInitTypeDef 的定义：

```

typedef struct
{
    uint16_t TIM_Channel;
    uint16_t TIM_ICPolarity;
    uint16_t TIM_ICSelection;
    uint16_t TIM_ICPrescaler;
    uint16_t TIM_ICFilter;
} TIM_ICInitTypeDef;

```

参数 TIM_Channel 很好理解，用来设置通道。我们设置为通道 1，为 TIM_Channel_1。

参数 TIM_ICPolarity 是用来设置输入信号的有效捕获极性，这里我们设置为 TIM_ICPolarity_Rising，上升沿捕获。同时库函数还提供了单独设置通道 1 捕获极性的函数为：

```
TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Falling),
```

这表示通道 1 为上升沿捕获，我们后面会用到，同时对于其他三个通道也有一个类似的函数，使用的时候一定要分清楚使用的是哪个通道该调用哪个函数，格式为 TIM_OCxPolarityConfig()。

参数 TIM_ICSelection 是用来设置映射关系，我们配置 IC1 直接映射在 TI1 上，选择 TIM_ICSelection_DirectTI。

参数 TIM_ICPrescaler 用来设置输入捕获分频系数，我们这里不分频，所以选中 TIM_ICPSC_DIV1,还有 2,4,8 分频可选。

参数 TIM_ICFilter 设置滤波器长度，这里我们不使用滤波器，所以设置为 0。

这些参数的意义，在我们讲解寄存器的时候举例说明过，这里不做详细解释。

我们的配置代码是：

```

TIM_ICInitTypeDef TIM5_ICInitStructure;
TIM5_ICInitStructure.TIM_Channel = TIM_Channel_1; //选择输入端 IC1 映射到 TI1 上
TIM5_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; //上升沿捕获
TIM5_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //映射到 TI1 上
TIM5_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频
TIM5_ICInitStructure.TIM_ICFilter = 0x00; //IC1F=0000 配置输入滤波器 不滤波
TIM_ICInit(TIM5, &TIM5_ICInitStructure);

```

4) 使能捕获和更新中断（设置 TIM5 的 DIER 寄存器）

因为我们要捕获的是高电平信号的脉宽，所以，第一次捕获是上升沿，第二次捕获时下降沿，必须在捕获上升沿之后，设置捕获边沿为下降沿，同时，如果脉宽比较长，那么定时器就会溢出，对溢出必须做处理，否则结果就不准了。这两件事，我们都在中断里面做，所以必须开启捕获中断和更新中断。



这里我们使用定时器的开中断函数 TIM_ITConfig 即可使能捕获和更新中断：

```
TIM_ITConfig(TIM5,TIM_IT_Update|TIM_IT_CC1,ENABLE); //允许更新中断和捕获中断
```

5) 设置中断分组，编写中断服务函数

设置中断分组的方法前面多次提到这里我们不做讲解，主要是通过函数 NVIC_Init()来完成。分组完成后，我们还需要在中断函数里面完成数据处理和捕获设置等关键操作，从而实现高电平脉宽统计。在中断服务函数里面，跟以前的外部中断和定时器中断实验中一样，我们在中断开始的时候要进行中断类型判断，在中断结束的时候要清除中断标志位。使用到的函数在上面的实验已经讲解过，分别为 TIM_GetITStatus()函数和 TIM_ClearITPendingBit()函数。

```
if (TIM_GetITStatus(TIM5, TIM_IT_Update) != RESET){} //判断是否为更新中断
if (TIM_GetITStatus(TIM5, TIM_IT_CC1) != RESET){} //判断是否发生捕获事件
TIM_ClearITPendingBit(TIM5, TIM_IT_CC1|TIM_IT_Update); //清除中断和捕获标志位
```

6) 使能定时器（设置 TIM5 的 CR1 寄存器）

最后，必须打开定时器的计数器开关，启动 TIM5 的计数器，开始输入捕获。

```
TIM_Cmd(TIM5,ENABLE); //使能定时器 5
```

通过以上 6 步设置，定时器 5 的通道 1 就可以开始输入捕获了。

15.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) WK_UP 按键
- 3) 串口
- 4) 定时器 TIM3
- 5) 定时器 TIM5

前面 4 个，在之前的章节均有介绍。本节，我们将捕获 TIM5_CH1 (PA0) 上的高电平脉宽，通过 WK_UP 按键输入高电平，并从串口打印高电平脉宽。同时我们保留上节的 PWM 输出，大家也可以通过用杜邦线连接 PB5 和 PA0，来测量 PWM 输出的高电平脉宽。

15.3 软件设计

打开光盘的输入捕获实验，可以看到，我们的工程和上一个实验没有什么改动。因为我们的输入捕获代码是直接添加在 timer.c 和 timer.h 中。同时输入捕获相关的库函数还是在 stm32f10x_tim.c 和 stm32f10x_tim.h 文件中。

我们在 timer.c 里面加入如下代码：

```
//定时器 5 通道 1 输入捕获配置
TIM_ICInitTypeDef TIM5_ICInitStructure;
void TIM5_Cap_Init(u16 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE); //①使能 TIM5 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //①使能 GPIOA 时钟
```



```
//初始化 GPIOA.0 ①
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;           //PA0 设置
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;        //PA0 输入
GPIO_Init(GPIOA, &GPIO_InitStructure);             //初始化 GPIOA.0
GPIO_ResetBits(GPIOA,GPIO_Pin_0);                   //PA0 下拉

//②初始化 TIM5 参数
TIM_TimeBaseStructure.TIM_Period = arr;           //设定计数器自动重装值
TIM_TimeBaseStructure.TIM_Prescaler = psc;         //预分频器
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; // TDTs = Tck_tim
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数模式
TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure);    //初始化 TIMx

//③初始化 TIM5 输入捕获通道 1
TIM5_ICInitStructure.TIM_Channel = TIM_Channel_1; //选择输入端 IC1 映射到 TI1 上
TIM5_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; //上升沿捕获
TIM5_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //映射到 TI1 上
TIM5_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频
TIM5_ICInitStructure.TIM_ICFilter = 0x00;           //IC1F=0000 配置输入滤波器 不滤波
TIM_ICInit(TIM5, &TIM5_ICInitStructure);           //初始化 TIM5 输入捕获通道 1

//⑤初始化 NVIC 中断优先级分组
NVIC_InitStructure.NVIC_IRQChannel = TIM5_IRQn;    //TIM3 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; //先占优先级 2 级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;    //从优先级 0 级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;      //IRQ 通道被使能
NVIC_Init(&NVIC_InitStructure);                  //初始化 NVIC

TIM_ITConfig( TIM5,TIM_IT_Update|TIM_IT_CC1,ENABLE); //④允许更新中断捕获中断
TIM_Cmd(TIM5,ENABLE );                                //⑥使能定时器 5
}

u8 TIM5CH1_CAPTURE_STA=0; //输入捕获状态
u16 TIM5CH1_CAPTURE_VAL;//输入捕获值
//⑤定时器 5 中断服务程序
void TIM5_IRQHandler(void)
{
    if((TIM5CH1_CAPTURE_STA&0X80)==0)//还未成功捕获
    {
        if (TIM_GetITStatus(TIM5, TIM_IT_Update) != RESET)
    }
}
```



```

if(TIM5CH1_CAPTURE_STA&0X40)           //已经捕获到高电平了
{
    if((TIM5CH1_CAPTURE_STA&0X3F)==0X3F)//高电平太长了
    {
        TIM5CH1_CAPTURE_STA|=0X80;      //标记成功捕获了一次
        TIM5CH1_CAPTURE_VAL=0xFFFF;
    }else TIM5CH1_CAPTURE_STA++;
}
}

if (TIM_GetITStatus(TIM5, TIM_IT_CC1) != RESET)           //捕获 1 发生捕获事件
{
    if(TIM5CH1_CAPTURE_STA&0X40)           //捕获到一个下降沿
    {
        TIM5CH1_CAPTURE_STA|=0X80;          //标记成功捕获到一次上升沿
        TIM5CH1_CAPTURE_VAL=TIM_GetCapture1(TIM5);
        TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Rising); //设置为上升沿捕获
    }else                                //还未开始,第一次捕获上升沿
    {
        TIM5CH1_CAPTURE_STA=0;             //清空
        TIM5CH1_CAPTURE_VAL=0;
        TIM_SetCounter(TIM5,0);
        TIM5CH1_CAPTURE_STA|=0X40;         //标记捕获到了上升沿
        TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Falling); //设置为下降沿捕获
    }
}
}

TIM_ClearITPendingBit(TIM5, TIM_IT_CC1|TIM_IT_Update); //清除中断标志位
}

```

此部分代码包含 2 个函数，其中 TIM5_Cap_Init 函数用于 TIM5 通道 1 的输入捕获设置，其设置和我们上面讲的步骤是一样的，这里就不多说，我们通过标号①~⑥标注了前面讲解的步骤，大家可以对照看一下。重点来看看第二个函数。

TIM5_IRQHandler 是 TIM5 的中断服务函数，该函数用到了两个全局变量，用于辅助实现高电平捕获。其中 TIM5CH1_CAPTURE_STA，是用来记录捕获状态，该变量类似我们在 usart.c 里面自行定义的 USART_RX_STA 寄存器(其实就是一个变量，只是我们把它当成一个寄存器那样来使用)。TIM5CH1_CAPTURE_STA 各位描述如表 15.3.1 所示：

TIM5CH1_CAPTURE_STA		
bit7	bit6	bit5~0
捕获完成标志	捕获到高电平标志	捕获高电平后定时器溢出的次数

表 15.3.1 TIM5CH1_CAPTURE_STA 各位描述

另外一个变量 TIM5CH1_CAPTURE_VAL，则用来记录捕获到下降沿的时候，TIM5_CNT 的值。

现在我们来介绍一下，捕获高电平脉宽的思路：首先，设置 TIM5_CH1 捕获上升沿，这在



TIM5_Cap_Init 函数执行的时候就设置好了，然后等待上升沿中断到来，当捕获到上升沿中断，此时如果 TIM5CH1_CAPTURE_STA 的第 6 位为 0，则表示还没有捕获到新的上升沿，就先把 TIM5CH1_CAPTURE_STA、TIM5CH1_CAPTURE_VAL 和 TIM5->CNT 等清零，然后再设置 TIM5CH1_CAPTURE_STA 的第 6 位为 1，标记捕获到高电平，最后设置为下降沿捕获，等待下降沿到来。如果等待下降沿到来期间，定时器发生了溢出，就在 TIM5CH1_CAPTURE_STA 里面对溢出次数进行计数，当最大溢出次数来到的时候，就强制标记捕获完成（虽然此时还没有捕获到下降沿）。当下降沿到来的时候，先设置 TIM5CH1_CAPTURE_STA 的第 7 位为 1，标记成功捕获一次高电平，然后读取此时的定时器值到 TIM5CH1_CAPTURE_VAL 里面，最后设置为上升沿捕获，回到初始状态。

这样，我们就完成一次高电平捕获了，只要 TIM5CH1_CAPTURE_STA 的第 7 位一直为 1，那么就不会进行第二次捕获，我们在 main 函数处理完捕获数据后，将 TIM5CH1_CAPTURE_STA 置零，就可以开启第二次捕获。

这里我们还使用到一个函数 TIM_OC1PolarityConfig 来修改输入捕获通道 1 的极性的。相信这个不难理解：

```
void TIM_OC1PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity)
```

要设置为上升沿捕获，则为：

```
TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Rising); //设置为上升沿捕获
```

还有一个函数用来设置计数器寄存器值，这个同样很好理解：

```
TIM_SetCounter(TIM5,0);
```

上行代码的意思就是计数值清零。

头文件 timer.h 比较简单，这里我们就不多说了。

接下来，我们修改主程序里面的 main 函数如下：

```
extern u8  TIM5CH1_CAPTURE_STA;          //输入捕获状态
extern u16   TIM5CH1_CAPTURE_VAL;         //输入捕获值
int main(void)
{
    u32 temp=0;
    delay_init();                      //延时函数初始化
    NVIC_Configuration();             //设置 NVIC 中断分组 2
    uart_init(9600);                  //串口初始化波特率为 9600
    LED_Init();                       //LED 端口初始化
    TIM3_PWM_Init(899,0);            //不分频。 PWM 频率=72000/(899+1)=80Khz
    TIM5_Cap_Init(0xFFFF,72-1);      //以 1Mhz 的频率计数
    while(1)
    {
        delay_ms(10);
        TIM_SetCompare2(TIM3,TIM_GetCapture2(TIM3)+1);

        if(TIM_GetCapture2(TIM3)==300)
            TIM_SetCompare2(TIM3,0);
        if(TIM5CH1_CAPTURE_STA&0X80)//成功捕获到了一次上升沿
        {
            temp=TIM5CH1_CAPTURE_STA&0X3F;
```



```
temp*=65536;//溢出时间总和  
temp+=TIM5CH1_CAPTURE_VAL;//得到总的高电平时间  
printf("HIGH:%d us\r\n",temp); //打印总的高点平时间  
TIM5CH1_CAPTURE_STA=0; //开启下一次捕获  
}  
}  
}
```

该 main 函数是在 PWM 实验的基础上修改来的，我们保留了 PWM 输出，同时通过设置 TIM5_Cap_Init(0xFFFF,72-1)，将 TIM5_CH1 的捕获计数器设计为 1us 计数一次，并设置重装载值为最大，所以我们的捕获时间精度为 1us。

主函数通过 TIM5CH1_CAPTURE_STA 的第 7 位，来判断有没有成功捕获到一次高电平，如果成功捕获，则将高电平时间通过串口输出到电脑。

至此，我们的软件设计就完成了。

15.4 下载验证

在完成软件设计之后，将我们将编译好的文件下载到战舰 STM32 开发板上，可以看到 DS0 的状态和上一章差不多，由暗→亮的循环。说明程序已经正常在跑了，我们再打开串口调试助手，选择对应的串口，然后按 WK_UP 按键，可以看到串口打印的高电平持续时间，如图 15.4.1 所示：



图 15.4.1 PWM 控制 DS0 亮度

从上图可以看出，其中有 2 次高电平在 50us 以内的，这种就是按键按下时发生的抖动。这就是为什么我们按键输入的时候，一般都需要做防抖处理，防止类似的情况干扰正常输入。大家还可以用杜邦线连接 PA0 和 PB5，看看上一节中我们设置的 PWM 输出的高电平是如何变化

的。



第十六章 电容触摸按键实验

上一章，我们介绍了 STM32 的输入捕获功能及其使用。这一章，我们将向大家介绍如何通过输入捕获功能，来做一个电容触摸按键。在本章中，我们将用 TIM5 的通道 2 (PA1) 来做输入捕获，并实现一个简单的电容触摸按键，通过该按键控制 DS1 的亮灭。从本章分为如下几个部分：

- 16.1 电容触摸按键简介
- 16.2 硬件设计
- 16.3 软件设计
- 16.4 下载验证



16.1 电容触摸按键简介

触摸按键相对于传统的机械按键有寿命长、占用空间少、易于操作等诸多优点。大家看看如今的手机，触摸屏、触摸按键大行其道，而传统的机械按键，正在逐步从手机上面消失。本章，我们将给大家介绍一种简单的触摸按键：电容式触摸按键。

我们将利用战舰 STM32 开发板上的触摸按键（TPAD），来实现对 DS1 的亮灭控制。这里 TPAD 其实就是战舰 STM32 开发板上的一小块覆铜区域，实现原理如图 16.1.1 所示：

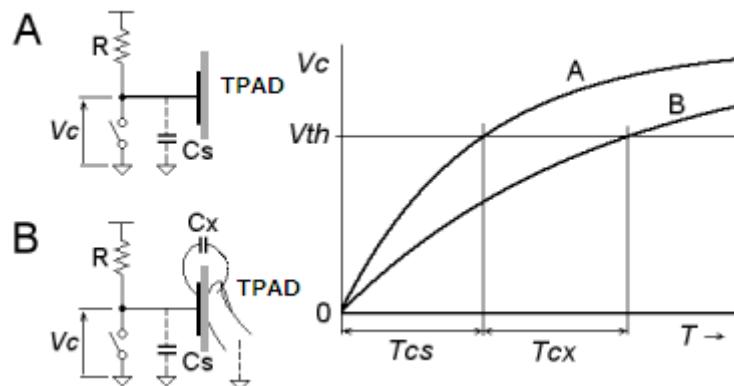


图 16.1.1 电容触摸按键原理

这里我们使用的是检测电容充放电时间的方法来判断是否有触摸，图中 R 是外接的电容充电电阻，Cs 是没有触摸按下时 TPAD 与 PCB 之间的杂散电容。而 Cx 则是有手指按下的时候，手指与 TPAD 之间形成的电容。图中的开关是电容放电开关（由实际使用时，由 STM32 的 IO 替代）。

先用开关将 Cs（或 Cs+Cx）上的电放尽，然后断开开关，让 R 给 Cs（或 Cs+Cx）充电，当没有手指触摸的时候，Cs 的充电曲线如图中的 A 曲线。而当有手指触摸的时候，手指和 TPAD 之间引入了新的电容 Cx，此时 Cs+Cx 的充电曲线如图中的 B 曲线。从上图可以看出，A、B 两种情况下，Vc 达到 Vth 的时间分别为 Tcs 和 Tcs+Tcx。

其中，除了 Cs 和 Cx 我们需要计算，其他都是已知的，根据电容充放电公式：

$$V_c = V_0 \cdot (1 - e^{-t/RC})$$

其中 V_c 为电容电压， V_0 为充电电压， R 为充电电阻， C 为电容容值， e 为自然底数， t 为充电时间。根据这个公式，我们就可以计算出 Cs 和 Cx。利用这个公式，我们还可以把战舰开发板作为一个简单的电容计，直接可以测电容容量了，有兴趣的朋友可以捣鼓下。

在本章中，其实我们只要能够区分 Tcs 和 Tcs+Tcx，就已经可以实现触摸检测了，当充电时间在 Tcs 附近，就可以认为没有触摸，而当充电时间大于 Tcs+Tx 时，就认为有触摸按下（Tx 为检测阀值）。

本章，我们使用 PA1(TIM5_CH2) 来检测 TPAD 是否有触摸，在每次检测之前，我们先配置 PA1 为推挽输出，将电容 Cs（或 Cs+Cx）放电，然后配置 PA1 为浮空输入，利用外部上拉电阻给电容 Cs(Cs+Cx) 充电，同时开启 TIM5_CH2 的输入捕获，检测上升沿，当检测到上升沿的时候，就认为电容充电完成了，完成一次捕获检测。

在 MCU 每次复位重启的时候，我们执行一次捕获检测（可以认为没触摸），记录此时的值，记为 tpad_default_val，作为判断的依据。在后续的捕获检测，我们就通过与 tpad_default_val 的



对比，来判断是不是有触摸发生。

关于输入捕获的配置，在上一章我们已经有详细介绍了，这里我们就不再介绍。至此，电容触摸按键的原理介绍完毕。

16.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0 和 DS1
- 2) 定时器 TIM5
- 3) 触摸按键 TPAD

前面两个之前均有介绍，我们需要通过 TIM5_CH2 (PA1) 采集 TPAD 的信号，所以本实验需要用跳线帽短接多功能端口(P14)的 TPAD 和 ADC，以实现 TPAD 连接到 PA1。如图 16.2.1 所示：

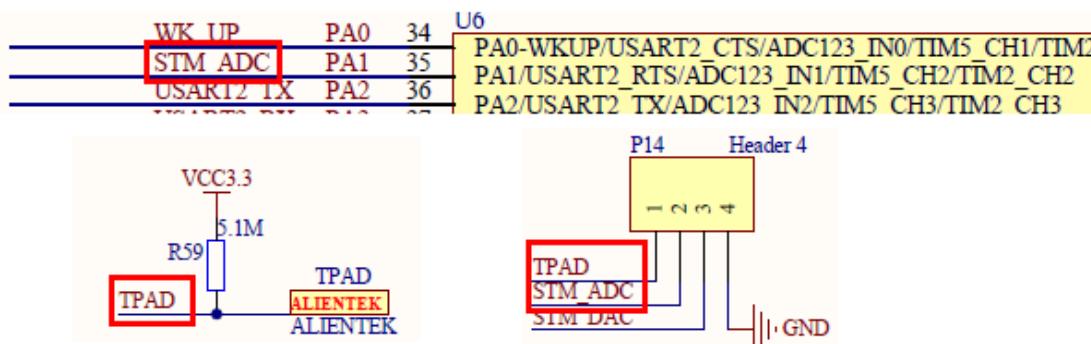


图 16.2.1 TPAD 与 STM32 连接原理图

硬件设置（用跳线帽短接多功能端口的 ADC 和 TPAD 即可）好之后，下面我们开始软件设计。

16.3 软件设计

前面讲解过，触摸按键我们是通过输入捕获实现的，所以使用的库函数依然是分布在 stm32f10x_tim.c 和 stm32f10x_tim.h 中。同时我们在 HARDWARE 组下面增加了 tpad.c 和 tpad.h 文件用来存放我们的触摸按键驱动代码。

打开 tpad.c 可以看到，我们在 tpad.c 里输入了如下代码：

```
#define TPAD_ARR_MAX_VAL      0xFFFF //最大的 ARR 值
vu16 tpad_default_val=0;//空载的时候(没有手按下),计数器需要的时间
//初始化触摸按键
//获得空载的时候触摸按键的取值.
//返回值:0,初始化成功;1,初始化失败
u8 TPad_Init()
{
    u16 buf[10];
    u16 temp;
    u8 j,i;
    //以 1Mhz 的频率计数
```



```
TIM5_CH2_Cap_Init(TPAD_ARR_MAX_VAL, SystemCoreClock/1000000-1);
for(i=0;i<10;i++)           //连续读取 10 次
{
    buf[i]=TPAD_Get_Val();
    delay_ms(10);
}
for(i=0;i<9;i++)           //排序
{  for(j=i+1;j<10;j++)
{   if(buf[i]>buf[j])      //升序排列
    {   temp=buf[i];
        buf[i]=buf[j];
        buf[j]=temp;
    }
}
temp=0;
for(i=2;i<8;i++)temp+=buf[i]; //取中间的 8 个数据进行平均
tpad_default_val=temp/6;
printf("tpad_default_val:%d\r\n",tpad_default_val);
//初始化遇到超过 TPAD_ARR_MAX_VAL/2 的数值,不正常!
if(tpad_default_val>TPAD_ARR_MAX_VAL/2) return 1;  return 0;
}
//复位一次
void TPAD_Reset(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能 PA 时钟
    //设置 GPIOA.1 为推挽使出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;                //PA1 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;          //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);                  //初始化 GPIOA.1
    GPIO_ResetBits(GPIOA,GPIO_Pin_1);                         //PA.1 输出 0,放电
    delay_ms(5);                                            //延时 5ms
    TIM_SetCounter(TIM5,0);                                  //归 0
    TIM_ClearITPendingBit(TIM5, TIM_IT_CC2|TIM_IT_Update); //清除中断标志
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IN_FLOATING;     //GPIOA.1 浮空输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
//得到定时器捕获值
//如果超时,则直接返回定时器的计数值.
u16 TPAD_Get_Val(void)
{
```



```
TPAD_Reset();
while(TIM_GetITStatus(TIM5, TIM_IT_CC2)== RESET) //等待溢出
{
    if(TIM_GetCounter(TIM5)>TPAD_ARR_MAX_VAL-500)
        return TIM_GetCounter(TIM5); //超时了,直接返回 CNT 的值
    };
    return TIM_GetCapture2(TIM5);
}

//读取 n 次,取最大值
u16 TPAD_Get_MaxVal(u8 n)
{
    u16 temp=0;
    u16 res=0;
    while(n--)
    {
        temp=TPAD_Get_Val(); //得到一次值
        if(temp>res)res=temp;
    };
    return res;
}

//扫描触摸按键
//mode:0,不支持连续触发(按下一次必须松开才能按下次);1,支持连续触发(可以一直按)
//返回值:0,没有按下;1,有按下;
#define TPAD_GATE_VAL 80 //触摸的门限值,也就是必须大于
//tpad_default_val+TPAD_GATE_VAL,才认为是有效触摸.
u8 TPAD_Scan(u8 mode)
{
    static u8 keyen=0; //0,可以开始检测;>0,还不能开始检测
    u8 res=0;
    u8 sample=3; //默认采样次数为 3 次
    u16 rval;
    if(mode)
    {
        sample=6; //支持连接的时候, 设置采样次数为 6 次
        keyen=0; //支持连接
    }
    rval=TPAD_Get_MaxVal(sample);
    if(rval>(tpad_default_val+TPAD_GATE_VAL))//大于
//tpad_default_val+TPAD_GATE_VAL,有效
    {
        rval=TPAD_Get_MaxVal(sample);
        if((keyen==0)&&(rval>(tpad_default_val+TPAD_GATE_VAL)))//大于
//tpad_default_val+TPAD_GATE_VAL,有效
```



```
{  
    res=1;  
}  
//printf("r:%d\r\n",rval);  
keyen=5;           //至少要再过 5 次之后才能按键有效  
}else if(keyen>2)keyen=2; //如果检测到按键松开,则直接将次数将为 2,以提高响应速度  
if(keyen)keyen--;  
return res;  
}  
//定时器 2 通道 2 输入捕获配置  
void TIM5_CH2_Cap_Init(u16 arr,u16 psc)  
{  
    GPIO_InitTypeDef  GPIO_InitStructure;  
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;  
    TIM_ICInitTypeDef  TIM5_ICInitStructure;  
  
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE); //使能 TIM5 时钟  
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能 PA 时钟  
    //设置 GPIOA.1 为浮空输入  
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;           //PA1 端口配置  
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;    //速度 50MHz  
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IN_FLOATING; //浮空输入  
    GPIO_Init(GPIOA, &GPIO_InitStructure);             //初始化 GPIOA.1  
  
    //初始化 TIM5  
    TIM_TimeBaseStructure.TIM_Period = arr;            //设定计数器自动重装值  
    TIM_TimeBaseStructure.TIM_Prescaler = psc;          //预分频器  
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; // TDTs = Tck_tim  
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式  
    TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure);      //根据参数初始化 TIMx  
    //初始化 TIM5 通道 2  
    TIM5_ICInitStructure.TIM_Channel = TIM_Channel_2; //选择输入端 IC2 映射到 TI5 上  
    TIM5_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; //上升沿捕获  
    TIM5_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;  
    TIM5_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频  
    TIM5_ICInitStructure.TIM_ICFilter = 0x03;//配置输入滤波器 8 个定时器时钟周期滤波  
    TIM_ICInit(TIM5, &TIM5_ICInitStructure);//初始化 I5 IC2  
  
    TIM_Cmd(TIM5,ENABLE );           //使能定时器 5  
}
```

此部分代码包含 6 个函数，我们将介绍其中 4 个比较重要的函数：TIM5_CH2_Cap_Init、TPAD_Get_Val、TPAD_Init 和 TPAD_Scan。

首先介绍 TIM5_CH2_Cap_Init 函数，该函数和上一章的输入捕获函数基本一样，不同的是，



这里我们设置的是 CH2 通道，并开启了输入滤波器。通过该函数的设置，我们将可以捕获 PA1 上的上升沿。关于配置的详细介绍大家可以看第 15 章输入捕获实验讲解。

我们再来看看 TPAD_Get_Val 函数，该函数用于得到定时器的一次捕获值。该函数先调用 TPAD_Reset，将电容放电，同时设置 TIM5_CNT 寄存器为 0，然后死循环等待发生上升沿捕获（或计数溢出），将捕获到的值（或溢出值）作为返回值返回。

接着我们介绍 TPAD_Init 函数，该函数用于初始化输入捕获，并获取默认的 TPAD 值。该函数有一个参数，用来传递系统时钟，其实是为了配置 TIM5_CH2_Cap_Init 为 1us 计数周期。在该函数中连续 10 次读取 TPAD 值，将这些值升序排列后取中间 6 个值再做平均（这样做的目的是尽量减少误差），并赋值给 tpad_default_val，用于后续触摸判断的标准。

最后，我们来看看 TPAD_Scan 函数，该函数用于扫描 TPAD 是否有触摸，该函数的参数 mode，用于设置是否支持连续触发。返回值如果是 0，说明没有触摸，如果是 1，则说明有触摸。该函数同样包含了一个静态变量，用于检测控制，类似第八章的 KEY_Scan 函数。所以该函数同样是不可重入的。在函数中，我们通过连续读取 3 次(不支持连续按的时候)TPAD 的值，取这他们的最大值，和 tpad_default_val+TPAD_GATE_VAL 比较，如果大于则说明有触摸，如果小于，则说明无触摸。其中 tpad_default_val 是我们在调用 TPAD_Init 函数的时候得到的值，而 TPAD_GATE_VAL 则是我们设定的一个门限值（这个大家可以通过实验数据得出，根据实际情况选择适合的值就好了），这里我们设置为 80。该函数，我们还做了一些其他的条件限制，让触摸按键有更好的效果，这个就请大家看代码自行参悟了。

在 tpad.h 文件里面，我们只是进行了一些函数申明。

接下来，我们看看主程序里面的 main 函数如下：

```
int main(void)
{
    u8 t=0;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    TPAD_Init();            //初始化触摸按键
    while(1)
    {
        if(TPAD_Scan(0))  //成功捕获到了一次上升沿(此函数执行时间至少 15ms)
        {
            LED1=!LED1; //LED1 取反
        }
        t++;
        if(t==15)
        {
            t=0;
            LED0=!LED0; //LED0 取反,提示程序正在运行
        }
        delay_ms(10);
    }
}
```

该 main 函数比较简单，TPAD_Init()函数执行之后，就开始触摸按键的扫描，当有触摸的



时候，对 DS1 取反，而 DS0 则有规律的间隔取反，提示程序正在运行。

这里还要提醒一下大家，不要把 `uart_init(9600);` 去掉，因为在 `TPAD_Init` 函数里面，我们有用到 `printf`，如果你去掉了 `uart_init`，就会导致 `printf` 无法执行，从而死机。

至此，我们的软件设计就完成了。

16.4 下载验证

在完成软件设计之后，将我们将编译好的文件下载到战舰 STM32 开发板上，可以看到 DS0 慢速闪烁，此时，我们用手指触摸 ALIENTEK 战舰 STM32 开发板上的 TPAD（右下角的 LOGO 标志），就可以控制 DS1 的亮灭了。不过你要确保 TPAD 和 ADC 的跳线帽连接上了哦！如图 16.4.1 所示：

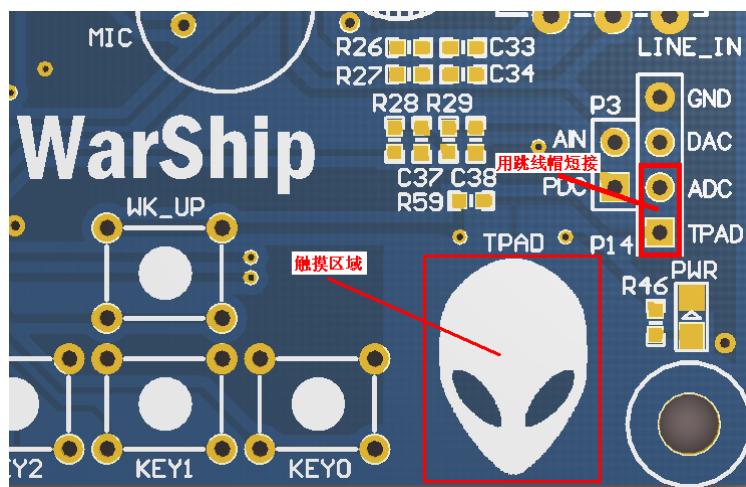


图 16.4.1 触摸区域和跳线帽短接方式

同时大家可以打开串口调试助手，每次复位的时候，会收到 `tpad_default_val` 的值，一般为 70 左右，根据 16.1 节提到的公式，我们可以计算出 C_s 的容值为 27pF 左右。



第十七章 OLED 显示实验

前面几章的实例，均没涉及到液晶显示，这一章，我们将向大家介绍 OLED 的使用。在本章中，我们将利用战舰 STM32 开发板上的 OLED 模块接口（与摄像头共用的这个），来点亮 OLED，并实现 ASCII 字符的显示。本章分为如下几个部分：

- 17.1 OLED 简介
- 17.2 硬件设计
- 17.3 软件设计
- 17.4 下载验证



17.1 OLED 简介

OLED，即有机发光二极管（Organic Light-Emitting Diode），又称为有机电激光显示（Organic Electroluminescence Display，OELD）。OLED 由于同时具备自发光，不需背光源、对比度高、厚度薄、视角广、反应速度快、可用于挠曲性面板、使用温度范围广、构造及制程较简单等优异之特性，被认为是下一代的平面显示器新兴应用技术。

LCD 都需要背光，而 OLED 不需要，因为它是自发光的。这样同样的显示，OLED 效果要来得好一些。以目前的技术，OLED 的尺寸还难以大型化，但是分辨率确可以做到很高。在本章中，我们使用的是 ALINETEK 的 OLED 显示模块，该模块有以下特点：

- 1) 模块有单色和双色两种可选，单色为纯蓝色，而双色则为黄蓝双色。
- 2) 尺寸小，显示尺寸为 0.96 寸，而模块的尺寸仅为 27mm*26mm 大小。
- 3) 高分辨率，该模块的分辨率为 128*64。
- 4) 多种接口方式，该模块提供了总共 5 种接口包括：6800、8080 两种并行接口方式、3 线或 4 线的穿行 SPI 接口方式、IIC 接口方式（只需要 2 根线就可以控制 OLED 了！）。
- 5) 不需要高压，直接接 3.3V 就可以工作了。

这里要提醒大家的是，该模块不和 5.0V 接口兼容，所以请大家在使用的时候一定要小心，别直接接到 5V 的系统上去，否则可能烧坏模块。以上 5 种模式通过模块的 BS0~2 设置，BS0~2 的设置与模块接口模式的关系如表 17.1.1 所示：

模块跳线口	IIC 接口	6800并行接口	8080并行接口	4线串行接口	3线串行接口
BS0	0	0	0	0	1
BS1	1	0	1	0	0
BS2	0	1	1	0	0

表 17.1.1 OLED 模块接口方式设置表

表 17.1.1 中：“1”代表接 VCC，而“0”代表接 GND。

该模块的外观图如图 17.1.1 所示：

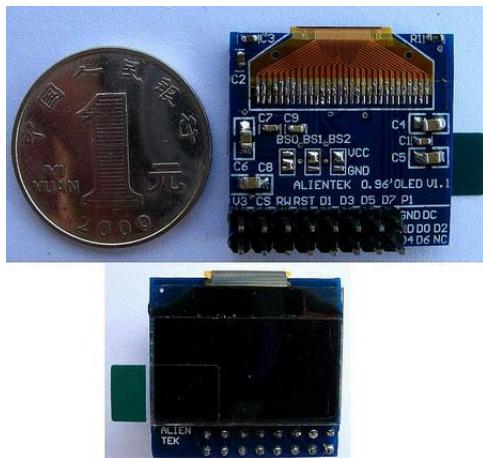


图 17.1.1 ALIENTEK OLED 模块外观图

ALIENTEK OLED 模块默认设置的是 BS0 接 GND，BS1 和 BS2 接 VCC，即使用 8080 并口方式，如果你想要设置为其他模式，则需要在 OLED 的背面，用烙铁修改 BS0~2 的设置。

模块的原理图如图 17.1.2 所示：

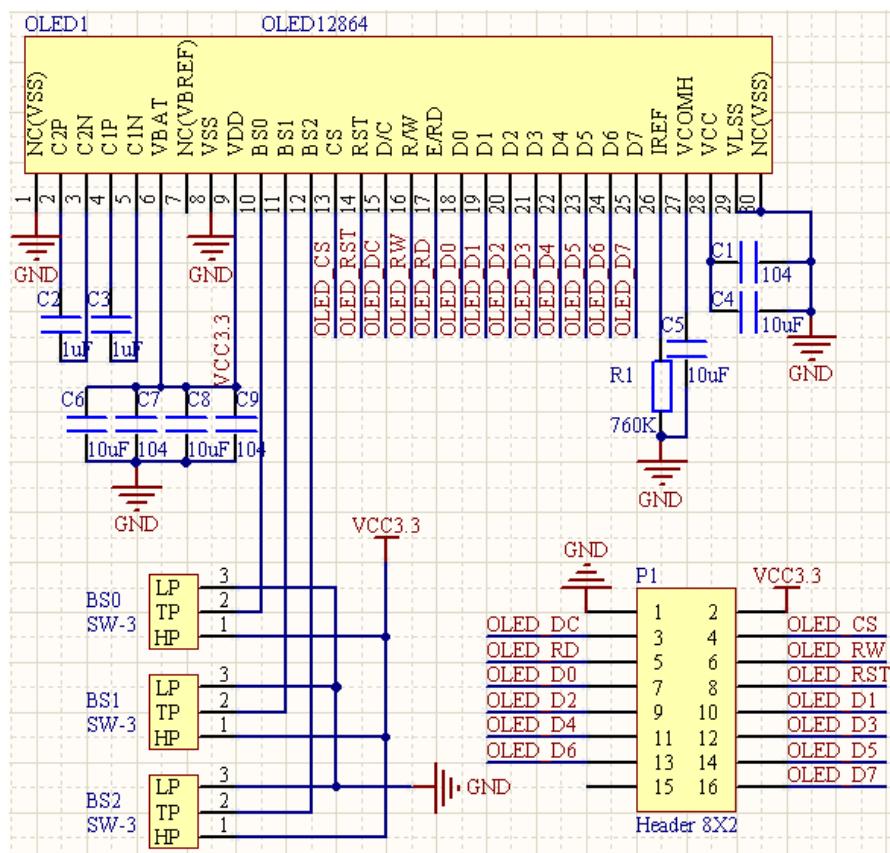


图 17.1.2 ALIENTEK OLED 模块原理图

该模块采用 8*2 的 2.54 排针与外部连接，总共有 16 个管脚，在 16 条线中，我们只用了 15 条，有一个是悬空的。15 条线中，电源和地线占了 2 条，还剩下 13 条信号线。在不同模式下，我们需要的信号线数量是不同的，在 8080 模式下，需要全部 13 条，而在 IIC 模式下，仅需要 2 条线就够了！这其中有一条是共同的，那就是复位线 RST (RES)，RST 上的低电平，将导致 OLED 复位，在每次初始化之前，都应该复位一下 OLED 模块。

ALIENTEK OLED 模块的控制器是 SSD1306，本章，我们将学习如何通过 STM32 来控制该模块显示字符和数字，本章的实例代码将可以支持 2 种方式与 OLED 模块连接，一种是 8080 的并口方式，另外一种是 4 线 SPI 方式。

首先我们介绍一下模块的 8080 并行接口，8080 并行接口的发明者是 INTEL，该总线也被广泛应用于各类液晶显示器，ALIENTEK OLED 模块也提供了这种接口，使得 MCU 可以快速的访问 OLED。ALIENTEK OLED 模块的 8080 接口方式需要如下一些信号线：

CS：OLED 片选信号。

WR：向 OLED 写入数据。

RD：从 OLED 读取数据。

D[7: 0]：8 位双向数据线。

RST(RES)：硬复位 OLED。

DC：命令/数据标志（0，读写命令；1，读写数据）。

模块的 8080 并口读/写的过程为：先根据要写入/读取的数据的类型，设置 DC 为高（数据）/低（命令），然后拉低片选，选中 SSD1306，接着我们根据是读数据，还是要写数据置 RD/WR 为低，然后：

在 RD 的上升沿，使数据锁存到数据线（D[7: 0]）上；



在 WR 的上升沿，使数据写入到 SSD1306 里面；
SSD1306 的 8080 并口写时序图如图 17.1.3 所示：

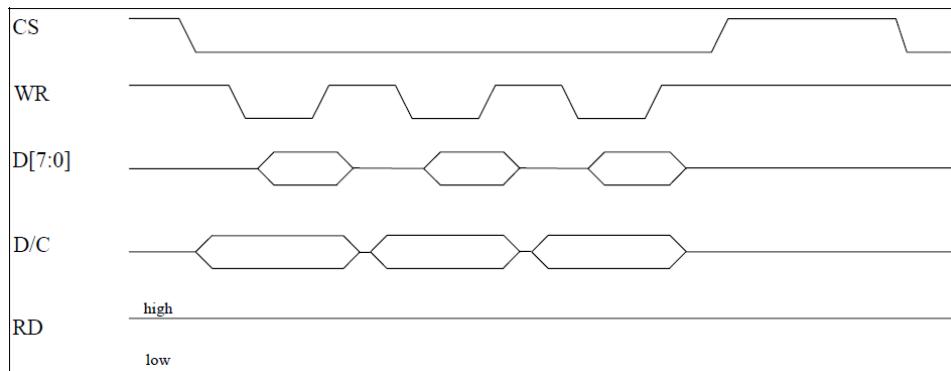


图 17.1.3 8080 并口写时序图

SSD1306 的 8080 并口读时序图如图 17.1.4 所示：

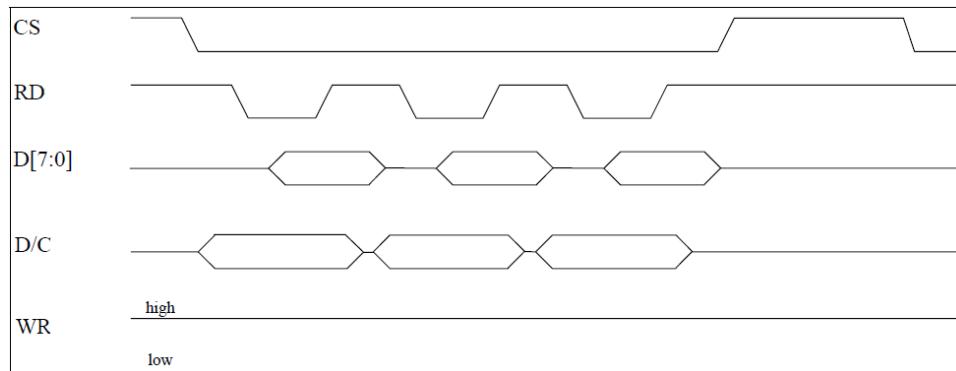


图 17.1.4 8080 并口读时序图

SSD1306 的 8080 接口方式下，控制脚的信号状态所对应的功能如表 17.1.2：

功能	RD	WR	CS	DC
写命令	H	↑	L	L
读状态	↑	H	L	L
写数据	H	↑	L	H
读数据	↑	H	L	H

表 17.1.2 控制脚信号状态功能表

在 8080 方式下读数据操作的时候，我们有时候（例如读显存的时候）需要一个假读命（Dummy Read），以使得微控制器的操作频率和显存的操作频率相匹配。在读取真正的数据之前，由一个的假读的过程。这里的假读，其实就是第一个读到的字节丢弃不要，从第二个开始，才是我们真正要读的数据。

一个典型的读显存的时序图，如图 17.1.5 所示：

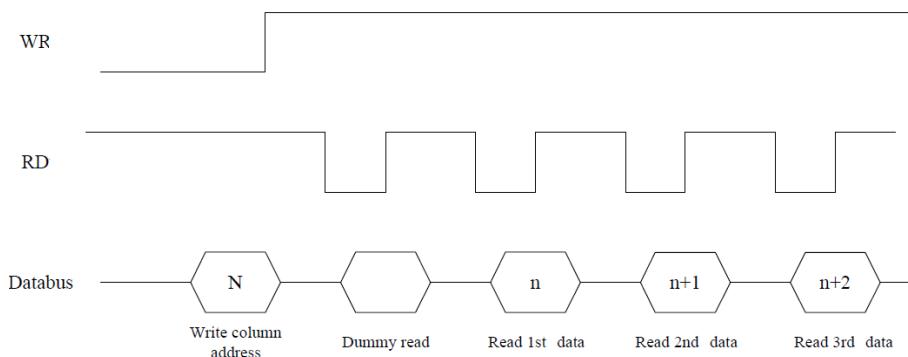


图 17.1.5 读显存时序图

可以看到，在发送了列地址之后，开始读数据，第一个是 Dummy Read，也就是假读，我们从第二个开始，才算是真正有效的数据。

并行接口模式就介绍到这里，我们接下来介绍一下 4 线串行 (SPI) 方式，4 先串口模式使用的信号线有如下几条：

CS：OLED 片选信号。

RST(RES)：硬复位 OLED。

DC：命令/数据标志 (0，读写命令；1，读写数据)。

SCLK：串行时钟线。在 4 线串行模式下，D0 信号线作为串行时钟线 SCLK。

SDIN：串行数据线。在 4 线串行模式下，D1 信号线作为串行数据线 SDIN。

模块的 D2 需要悬空，其他引脚可以接到 GND。在 4 线串行模式下，只能往模块写数据而不能读数据。

在 4 线 SPI 模式下，每个数据长度均为 8 位，在 SCLK 的上升沿，数据从 SDIN 移入到 SSD1306，并且是高位在前的。DC 线还是用作命令/数据的标志线。在 4 线 SPI 模式下，写操作的时序如图 17.1.6 所示：

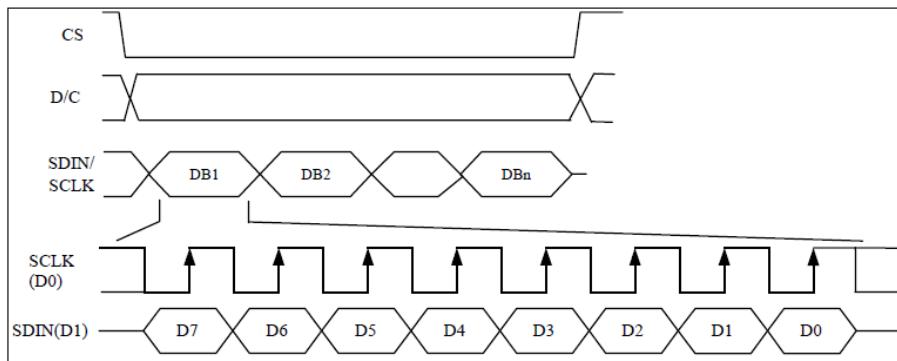


图 17.1.6 4 线 SPI 写操作时序图

4 线串行模式就为大家介绍到这里。其他还有几种模式，在 SSD1306 的数据手册上都有详细的介绍，如果要使用这些方式，请大家参考该手册。

接下来，我们介绍一下模块的显存，SSD1306 的显存总共为 128*64bit 大小，SSD1306 将这些显存分为了 8 页，其对应关系如表 17.1.3 所示：



列 (COM0~63)	行 (COLO~127)							
	SEG0	SEG1	SEG2	SEG125	SEG126	SEG127	
				PAGE0				
				PAGE1				
				PAGE2				
				PAGE3				
				PAGE4				
				PAGE5				
				PAGE6				
				PAGE7				

表 17.1.3 SSD1306 显存与屏幕对应关系表

可以看出，SSD1306 的每页包含了 128 个字节，总共 8 页，这样刚好是 128*64 的点阵大小。因为每次写入都是按字节写入的，这就存在一个问题，如果我们使用只写方式操作模块，那么，每次要写 8 个点，这样，我们在画点的时候，就必须把要设置的点所在的字节的每个位都搞清楚当前的状态（0/1？），否则写入的数据就会覆盖掉之前的状态，结果就是有些不需要显示的点，显示出来了，或者该显示的没有显示了。这个问题在能读的模式下，我们可以先读出来要写入的那个字节，得到当前状况，在修改了要改写的位之后再写进 GRAM，这样就不会影响到之前的状况了。但是这样需要能读 GRAM，对于 3 线或 4 线 SPI 模式，模块是不支持读的，而且读->改->写的方式速度也比较慢。

所以我们采用的办法是在 STM32 的内部建立一个 OLED 的 GRAM（共 128*8 个字节），在每次修改的时候，只是修改 STM32 上的 GRAM（实际上就是 SRAM），在修改完了之后，一次性把 STM32 上的 GRAM 写入到 OLED 的 GRAM。当然这个方法也有坏处，就是对于那些 SRAM 很小的单片机（比如 51 系列）就比较麻烦了。

SSD1306 的命令比较多，这里我们仅介绍几个比较常用的命令，这些命令如表 17.1.4 所示：

序号	指令	各位描述								命令	说明
		HEX	D7	D6	D5	D4	D3	D2	D1		
0	81	1	0	0	0	0	0	0	1	设置对比度	A 的值越大屏幕越亮， A 的范围从 0X00~0XFF
	A[7:0]	A7	A6	A5	A4	A3	A2	A1	A0		
1	AE/AF	1	0	1	0	1	1	1	X0	设置显示开关	X0=0，关闭显示； X0=1，开启显示；
2	8D	1	0	0	0	1	1	0	1	电荷泵设置	A2=0，关闭电荷泵 A2=1，开启电荷泵
	A[7:0]	*	*	0	1	0	A2	0	0		
3	B0~B7	1	0	1	1	0	X2	X1	X0	设置页地址	X[2:0]=0~7 对应页 0~7
4	00~0F	0	0	0	0	X3	X2	X1	X0	设置列地址	设置 8 位起始列地址的低四位
5	10~1F	0	0	0	0	X3	X2	X1	X0	设置列地址	设置 8 位起始列地址的高四位

表 17.1.4 SSD1306 常用命令表

第一个命令为 0X81，用于设置对比度的，这个命令包含了两个字节，第一个 0X81 为命令，随后发送的一个字节为要设置的对比度的值。这个值设置得越大屏幕就越亮。

第二个命令为 0XAE/0xAF。0XAE 为关闭显示命令；0xAF 为开启显示命令。

第三个命令为 0X8D，该指令也包含 2 个字节，第一个为命令字，第二个为设置值，第二个字节的 BIT2 表示电荷泵的开关状态，该位为 1，则开启电荷泵，为 0 则关闭。在模块初始化的时候，这个必须要开启，否则是看不到屏幕显示的。

第四个命令为 0XB0~B7，该命令用于设置页地址，其低三位的值对应着 GRAM 的页地址。



第五个指令为 0X00~0X0F，该指令用于设置显示时的起始列地址低四位。

第六个指令为 0X10~0X1F，该指令用于设置显示时的起始列地址高四位。

其他命令，我们就不在这里一一介绍了，大家可以参考 SSD1306 datasheet 的第 28 页。从这页开始，对 SSD1306 的指令有详细的介绍。

最后，我们再来介绍一下 OLED 模块的初始化过程，SSD1306 的典型初始化框图如图 17.1.7 所示：

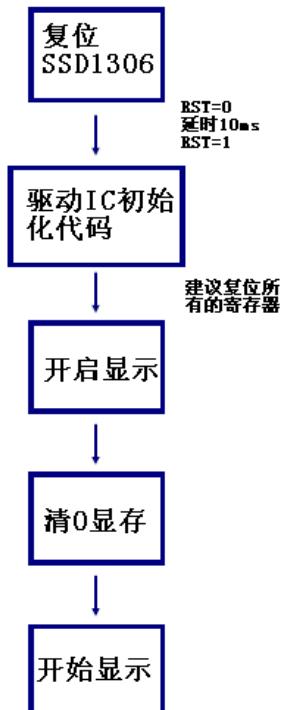


图 17.1.7 SSD1306 初始化框图

驱动 IC 的初始化代码，我们直接使用厂家推荐的设置就可以了，只要对细节部分进行一些修改，使其满足我们自己的要求即可，其他不需要变动。

OLED 的介绍就到此为止，我们重点向大家介绍了 ALIENTEK OLED 模块的相关知识，接下来我们将使用这个模块来显示字符和数字。通过以上介绍，我们可以得出 OLED 显示需要的相关设置步骤如下：

1) 设置 STM32 与 OLED 模块相连接的 IO。

这一步，先将我们与 OLED 模块相连的 IO 口设置为输出，具体使用哪些 IO 口，这里需要根据连接电路以及 OLED 模块所设置的通讯模式来确定。这些将在硬件设计部分向大家介绍。

2) 初始化 OLED 模块。

其实这里就是上面的初始化框图的内容，通过对 OLED 相关寄存器的初始化，来启动 OLED 的显示。为后续显示字符和数字做准备。

3) 通过函数将字符和数字显示到 OLED 模块上。

这里就是通过我们设计的程序，将要显示的字符送到 OLED 模块就可以了，这些函数将在软件设计部分向大家介绍。

通过以上三步，我们就可以使用 ALIENTEK OLED 模块来显示字符和数字了，在后面我们还将会给大家介绍显示汉字的方法。这一部分就先介绍到这里。



17.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) OLED 模块

OLED 模块的电路在 17.1 节已有详细说明了，这里我们介绍 OLED 模块与 ALIETEK 战舰 STM32 开发板的连接，战舰 STM32 开发板有两个地方可以接 OLED 模块，第一个是左下角的摄像头模块/OLED 模块共用接口，第二个是 LCD 模块和 OLED 模块的共用接口，不论哪个共用接口，OLED 都是靠左插的。这里我们选择摄像头模块/OLED 模块共用接口来接 OLED 模块，OLED 模块同战舰 STM32 开发板的连接图如图 17.2.1 所示：

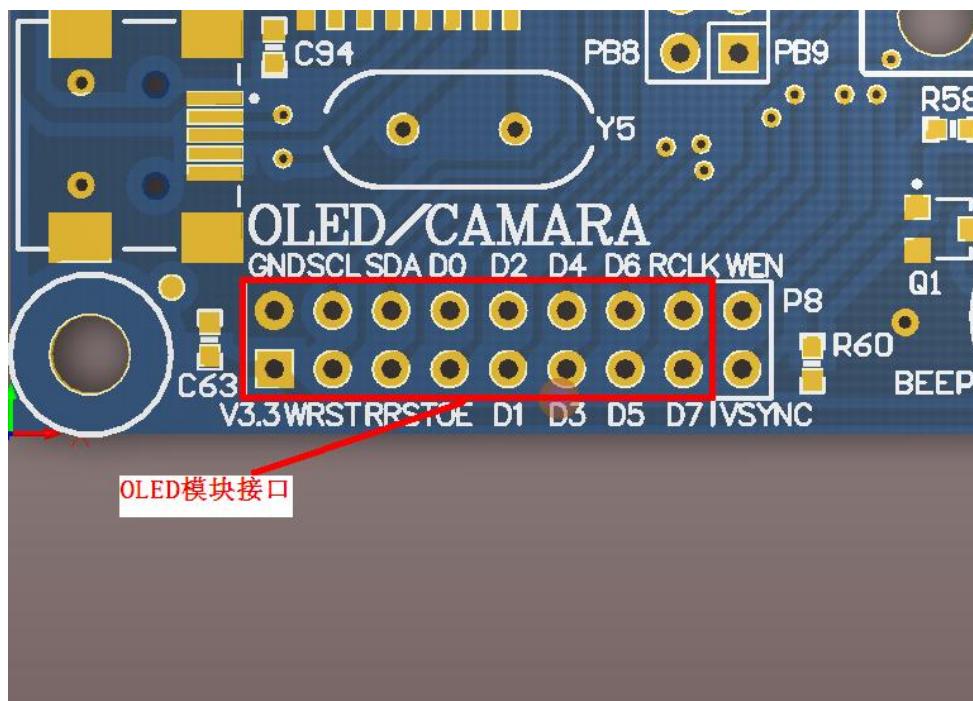


图 17.2.1 OLED 模块与开发板连接示意图

图中圈出来的部分就是连接 OLED 的接口，这里在硬件上，OLED 与战舰 STM32 开发板的 IO 口对应关系如下：

OLED_CS 对应 PD6;
OLED_RST 对应 PG15;
OLED_RS 对应 PD3;
OLED_WR 对应 PG14;
OLED_RD 对应 PG13;
OLED_D[7: 0]对应 PC[7: 0];

这些线的连接，战舰 STM32 的内部已经连接好了，我们只需要将 OLED 模块插上去就好了。实物连接如图 17.2.2 所示：



图 17.2.2 OLED 模块与开发板连接实物图

17.3 软件设计

我们直接打开 OLED 显示实验可以发现 HARDWARE 下面有一个 oled.c 文件，同时包含了头文件 oled.h。这里要说明一下，在我们寄存器版本的代码中，我们是采取的位带操作，这里我们采取的是库函数来进行 IO 操作。大家可以对照看一下。

oled.c 的代码，由于比较长，这里我们就不贴出来了，仅介绍几个比较重要的函数。首先是 OLED_Init 函数，该函数的结构比较简单，开始是对 IO 口的初始化，这里我们用了宏定义 OLED_MODE 来决定要设置的 IO 口，其他就是一些初始化序列了，我们按照厂家提供的资料来做就可以。最后要说明一点的是，因为 OLED 是无背光的，在初始化之后，我们把显存都清空了，所以在屏幕上是看不到任何内容的，跟没通电一个样，不要以为这就是初始化失败，要写入数据模块才会显示的。OLED_Init 函数代码如下：

```
//初始化 SSD1306
void OLED_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC|RCC_APB2Periph_GPIOD|
                           RCC_APB2Periph_GPIOG, ENABLE); //使能 PC,D,G 端口时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_6; //PD3,PD6 推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
    GPIO_Init(GPIOD, &GPIO_InitStructure); //初始化 GPIOD3,6
    GPIO_SetBits(GPIOD,GPIO_Pin_3|GPIO_Pin_6); //PD3,PD6 输出高

#if OLED_MODE==1
    GPIO_InitStructure.GPIO_Pin = 0xFF; //PC0~7 OUT 推挽输出
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    GPIO_SetBits(GPIOC,0xFF); //PC0~7 输出高
    //PG13,14,15 OUT 推挽输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
    GPIO_Init(GPIOG, &GPIO_InitStructure);
    //PG13,14,15 OUT 输出高
    GPIO_SetBits(GPIOG,GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15);
#else
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1; //PC0,1 OUT 推挽输出

```



```
GPIO_Init(GPIOC, &GPIO_InitStructure);
GPIO_SetBits(GPIOC,GPIO_Pin_0|GPIO_Pin_1);      //PC0,1 OUT  输出高
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;        //PG15 OUT 推挽输出 RST
GPIO_Init(GPIOG, &GPIO_InitStructure);
GPIO_SetBits(GPIOG,GPIO_Pin_15);                  //PG15 OUT  输出高
#endif
OLED_RST=0;
delay_ms(100);
OLED_RST=1;

OLED_WR_Byte(0xAE,OLED_CMD); //关闭显示
OLED_WR_Byte(0xD5,OLED_CMD); //设置时钟分频因子,震荡频率
OLED_WR_Byte(80,OLED_CMD);   //#[3:0],分频因子;[7:4],震荡频率
OLED_WR_Byte(0xA8,OLED_CMD); //设置驱动路数
OLED_WR_Byte(0X3F,OLED_CMD); //默认 0X3F(1/64)
OLED_WR_Byte(0xD3,OLED_CMD); //设置显示偏移
OLED_WR_Byte(0X00,OLED_CMD); //默认为 0
OLED_WR_Byte(0x40,OLED_CMD); //设置显示开始行 [5:0],行数.
OLED_WR_Byte(0x8D,OLED_CMD); //电荷泵设置
OLED_WR_Byte(0x14,OLED_CMD); //bit2, 开启/关闭
OLED_WR_Byte(0x20,OLED_CMD); //设置内存地址模式
OLED_WR_Byte(0x02,OLED_CMD); //#[1:0]00, 列地址模式;01,
                                //行地址模式;10,页地址模式;默认 10;
OLED_WR_Byte(0xA1,OLED_CMD); //段重定义设置,bit0:0,0->0;1,0->127;
OLED_WR_Byte(0xC0,OLED_CMD); //设置 COM 扫描方向;bit3:0,普通模式;1,
                                //重定义模式 COM[N-1]->COM0;N:驱动路数
OLED_WR_Byte(0xDA,OLED_CMD); //设置 COM 硬件引脚配置
OLED_WR_Byte(0x12,OLED_CMD); //#[5:4]配置
OLED_WR_Byte(0x81,OLED_CMD); //对比度设置
OLED_WR_Byte(0xEF,OLED_CMD); //1~255;默认 0X7F (亮度设置,越大越亮)
OLED_WR_Byte(0xD9,OLED_CMD); //设置预充电周期
OLED_WR_Byte(0xf1,OLED_CMD); //#[3:0],PHASE 1;[7:4],PHASE 2;
OLED_WR_Byte(0xDB,OLED_CMD); //设置 VCOMH 电压倍率
OLED_WR_Byte(0x30,OLED_CMD); //#[6:4] 000,0.65*vcc;001,0.77*vcc;011,0.83*vcc;
OLED_WR_Byte(0xA4,OLED_CMD); //全局显示开启;bit0:1,开启;0,关闭;(白屏/黑屏)
OLED_WR_Byte(0xA6,OLED_CMD); //设置显示方式;bit0:1,反相显示;0,正常显示
OLED_WR_Byte(0xAF,OLED_CMD); //开启显示
OLED_Clear(); //清屏
}
```

接着，要介绍的是 OLED_Refresh_Gram 函数。我们在 STM32 内部定义了一个块 GRAM:
u8 OLED_GRAM[128][8];此部分 GRAM 对应 OLED 模块上的 GRAM。在操作的时候，我们只要修改 STM32 内部的 GRAM 就可以了，然后通过 OLED_Refresh_Gram 函数把 GRAM 一次刷到 OLED 的 GRAM 上。该函数代码如下：



```
//更新显存到 LCD
void OLED_Refresh_Gram(void)
{
    u8 i,n;
    for(i=0;i<8;i++)
    {
        OLED_WR_Byte (0xb0+i,OLED_CMD);      //设置页地址 (0~7)
        OLED_WR_Byte (0x00,OLED_CMD);        //设置显示位置一列低地址
        OLED_WR_Byte (0x10,OLED_CMD);        //设置显示位置一列高地址
        for(n=0;n<128;n++)OLED_WR_Byte(OLED_GRAM[n][i],OLED_DATA);
    }
}
```

OLED_Refresh_Gram 函数先设置页地址，然后写入列地址（也就是纵坐标），然后从 0 开始写入 128 个字节，写满该页，最后循环把 8 页的内容都写入，就实现了整个从 STM32 显存到 OLED 显存的拷贝。

OLED_Refresh_Gram 函数还用到了一个外部函数，也就是我们接着要介绍的函数：OLED_WR_Byte，该函数直接和硬件相关，函数代码如下：

```
#if OLED_MODE==1
//向 SSD1306 写入一个字节。
//dat:要写入的数据/命令
//cmd:数据/命令标志 0,表示命令;1,表示数据;
void OLED_WR_Byte(u8 dat,u8 cmd)
{
    DATAOUT(dat);
    if(cmd)
        OLED_RS_Set();
    else
        OLED_RS_Clr();
    OLED_CS_Clr();
    OLED_WR_Clr();
    OLED_WR_Set();
    OLED_CS_Set();
    OLED_RS_Set();
}

#else
//向 SSD1306 写入一个字节。
//dat:要写入的数据/命令
//cmd:数据/命令标志 0,表示命令;1,表示数据;
void OLED_WR_Byte(u8 dat,u8 cmd)
{
    u8 i;
    if(cmd)
        OLED_RS_Set();
    else
```



```

    OLED_RS_Clr();
    OLED_CS_Clr();
    for(i=0;i<8;i++)
    {
        OLED_SCLK_Clr();
        if(dat&0x80)
            OLED_SDIN_Set();
        else
            OLED_SDIN_Clr();
        OLED_SCLK_Set();
        dat<<=1;
    }
    OLED_CS_Set();
    OLED_RS_Set();
}
#endif

```

这里有 2 个一样的函数，通过宏定义 `OLED_MODE` 来决定使用哪一个。如果 `OLED_MODE=1`，就定义为并口模式，选择第一个函数，而如果为 0，则为 4 线串口模式，选择第二个函数。这两个函数输入参数均为 2 个：`dat` 和 `cmd`，`dat` 为要写入的数据，`cmd` 则表明该数据是命令还是数据。这两个函数的时序操作就是根据上面我们对 8080 接口以及 4 线 SPI 接口的时序来编写的。

`OLED_GRAM[128][8]` 中的 128 代表列数（x 坐标），而 8 代表的是页，每页又包含 8 行，总共 64 行（y 坐标）。从高到低对应行数从小到大。比如，我们要在 `x=100, y=29` 这个点写入 1，则可以用这个句子实现：

```
OLED_GRAM[100][4]=1<<2;
```

一个通用的在点（x, y）置 1 表达式为：

```
OLED_GRAM[x][7-y/8]=1<<(7-y%8);
```

其中 x 的范围为：0~127；y 的范围为：0~63。

因此，我们可以得出下一个将要介绍的函数：画点函数，`void OLED_DrawPoint(u8 x, u8 y, u8 t)`；函数代码如下：

```

void OLED_DrawPoint(u8 x,u8 y,u8 t)
{
    u8 pos,bx,temp=0;
    if(x>127||y>63)return;//超出范围了.
    pos=7-y/8;
    bx=y%8;
    temp=1<<(7-bx);
    if(t)OLED_GRAM[x][pos]=temp;
    else OLED_GRAM[x][pos]&=~temp;
}

```

该函数有 3 个参数，前两个是坐标，第三个 `t` 为要写入 1 还是 0。该函数实现了我们在 OLED 模块上任意位置画点的功能。

在介绍完画点函数之后，我们介绍一下显示字符函数，`OLED_ShowChar`，在介绍之前，我们来介绍一下字符（ASCII 字符集）是怎么显示在 OLED 模块上去的。要显示字符，我们先要



有字符的点阵数据，ASCII 常用的字符集总共有 95 个，从空格符开始，分别为：!"#\$%&'()*+, -0123456789: ;<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~.

我们先要得到这个字符集的点阵数据，这里我们介绍一个款很好的字符提取软件：PCtoLCD2002 完美版。该软件可以提供各种字符，包括汉字（字体和大小都可以自己设置）阵提取，且取模方式可以设置好几种，常用的取模方式，该软件都支持。该软件还支持图形模式，也就是用户可以自己定义图片的大小，然后画图，根据所画的图形再生成点阵数据，这功能在制作图标或图片的时候很有用。

该软件的界面如图 17.3.1 所示：

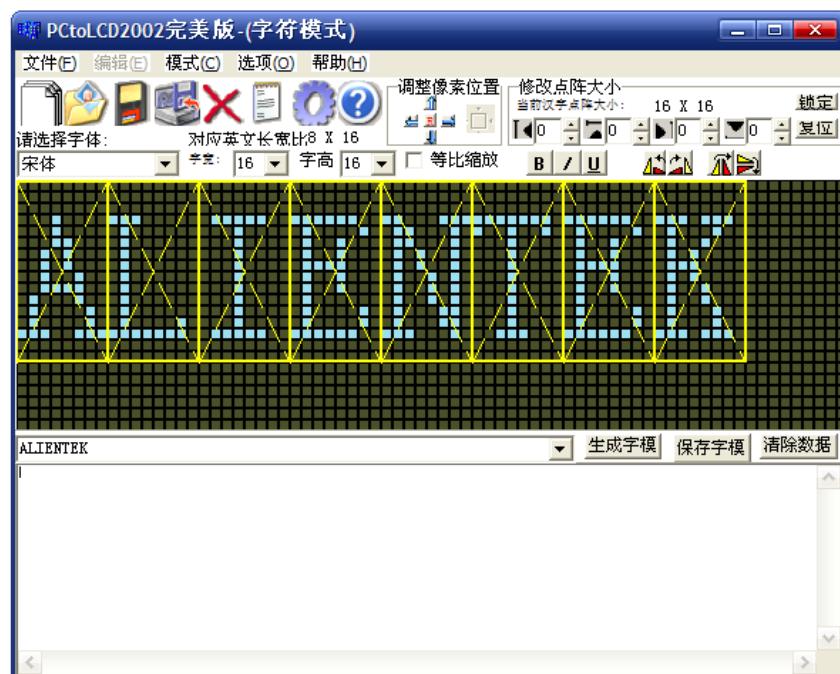


图 17.3.1 PCtoLCD2002 软件界面

然后我们选择设置，在设置里面设置取模方式如图 17.3.2 所示：

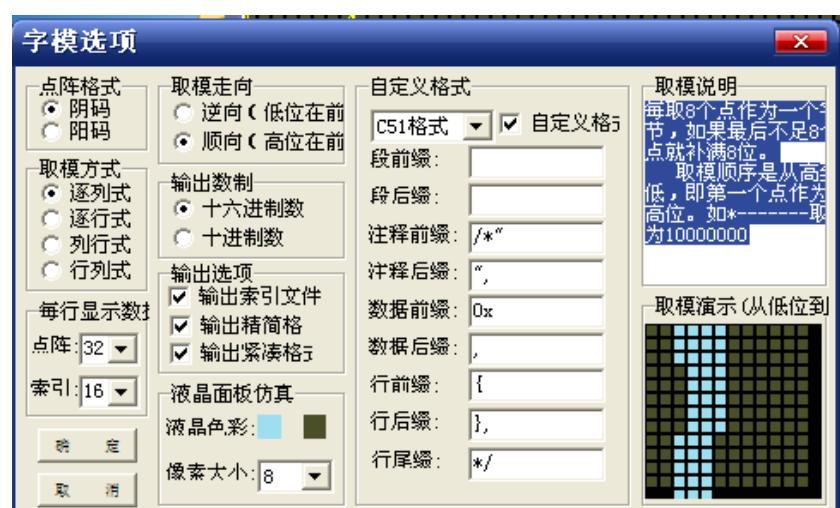


图 17.3.2 设置取模方式

上图设置的取模方式，在右上角的取模说明里面有，即：从第一列开始向下每取 8 个点作



为一个字节，如果最后不足 8 个点就补满 8 位。取模顺序是从高到低，即第一个点作为最高位。如*-----取为 10000000。其实就是按如图 17.3.3 所示的这种方式：

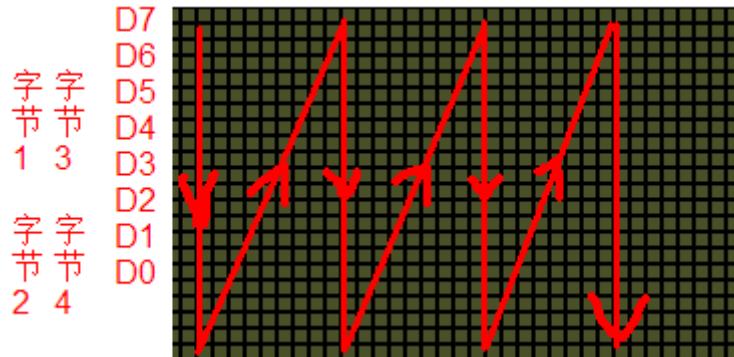


图 17.3.3 取模方式图解

从上到下，从左到右，高位在前。我们按这样的取模方式，然后把 ASCII 字符集按 12*6 大小和 16*0 大小取模出来（对应汉字大小为 12*12 和 16*16，字符的只有汉字的一半大！），保存在 oledfont.h 里面，每个 12*6 的字符占用 12 个字节，每个 16*8 的字符占用 16 个字节。具体见 oledfont.h 部分代码（该部分我们不再这里列出来了，请大家参考光盘里面的代码）。

在知道了取模方式之后，我们就可以根据取模的方式来编写显示字符的代码了，这里我们针对以上取模方式的显示字符代码如下：

```
void OLED_ShowChar(u8 x,u8 y,u8 chr,u8 size,u8 mode)
{
    u8 temp,t,t1;
    u8 y0=y;
    chr=chr-''; //得到偏移后的值
    for(t=0;t<size;t++)
    {
        if(size==12)temp=oled_asc2_1206[chr][t]; //调用 1206 字体
        else temp=oled_asc2_1608[chr][t]; //调用 1608 字体
        for(t1=0;t1<8;t1++)
        {
            if(temp&0x80)OLED_DrawPoint(x,y,mode);
            else OLED_DrawPoint(x,y,!mode);
            temp<<=1;
            y++;
            if((y-y0)==size)
            {
                y=y0;
                x++;
                break;
            }
        }
    }
}
```

该函数为字符以及字符串显示的核心部分，函数中 `chr=chr-''` 这句是要得到在字符点阵数



据里面的实际地址，因为我们的取模是从空格键开始的，例如 oled_asc2_1206[0][0]，代表的是空格符开始的点阵码。在接下来的代码，我们也是按照从上到小，从左到右的取模方式来编写的，先得到最高位，然后判断是写 1 还是 0，画点；接着读第二位，如此循环，直到一个字符的点阵全部取完为止。这其中涉及到列地址和行地址的自增，根据取模方式来理解，就不难了。

oled.c 的内容就为大家介绍到这里，下面看看 oled.h 的关键代码：

```
//OLED 模式设置
//0:4 线串行模式
//1:并行 8080 模式
#define OLED_MODE 1
//-----OLED 端口定义-----
#define OLED_CS_Clr() GPIO_ResetBits(GPIOB,GPIO_Pin_6)
#define OLED_CS_Set() GPIO_SetBits(GPIOB,GPIO_Pin_6)
#define OLED_RST_Clr() GPIO_ResetBits(GPIOB,GPIO_Pin_15)
#define OLED_RST_Set() GPIO_SetBits(GPIOB,GPIO_Pin_15)
#define OLED_RS_Clr() GPIO_ResetBits(GPIOB,GPIO_Pin_3)
#define OLED_RS_Set() GPIO_SetBits(GPIOB,GPIO_Pin_3)
#define OLED_WR_Clr() GPIO_ResetBits(GPIOB,GPIO_Pin_14)
#define OLED_WR_Set() GPIO_SetBits(GPIOB,GPIO_Pin_14)
#define OLED_RD_Clr() GPIO_ResetBits(GPIOB,GPIO_Pin_13)
#define OLED_RD_Set() GPIO_SetBits(GPIOB,GPIO_Pin_13)
//PC0~7,作为数据线
#define DATAOUT(x) GPIO_Write(GPIOC,x);//输出
//使用 4 线串行接口时使用
#define OLED_SCLK_Clr() GPIO_ResetBits(GPIOC,GPIO_Pin_0)
#define OLED_SCLK_Set() GPIO_SetBits(GPIOC,GPIO_Pin_0)
#define OLED_SDIN_Clr() GPIO_ResetBits(GPIOC,GPIO_Pin_1)
#define OLED_SDIN_Set() GPIO_SetBits(GPIOC,GPIO_Pin_1)
#define OLED_CMD 0 //写命令
#define OLED_DATA 1 //写数据
//OLED 控制用函数
void OLED_WR_Byte(u8 dat,u8 cmd);
void OLED_Display_On(void);
void OLED_Display_Off(void);
void OLED_Refresh_Gram(void);
void OLED_Init(void);
void OLED_Clear(void);
void OLED_DrawPoint(u8 x,u8 y,u8 t);
void OLED_Fill(u8 x1,u8 y1,u8 x2,u8 y2,u8 dot);
void OLED_ShowChar(u8 x,u8 y,u8 chr,u8 size,u8 mode);
void OLED_ShowNum(u8 x,u8 y,u32 num,u8 len,u8 size);
void OLED_ShowString(u8 x,u8 y,const u8 *p);
#endif
```

该部分比较简单，OLED_MODE 的定义也在这个文件里面，我们必须根据自己 OLED 模块 BS0~2 的设置（目前代码仅支持 8080 和 4 线 SPI）来确定 OLED_MODE 的值。这里的 IO 操作我们全部采用的是库函数而没有使用位带操作，目的是为了提高代码的可移植性。

最后我们来看看主函数源码：

```

int main(void)
{
    u8 t;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占 2 位响应优先级
    LED_Init();            //LED 端口初始化
    OLED_Init();           //初始化 OLED
    OLED_ShowString(0,0, "0.96' OLED TEST");
    OLED_ShowString(0,16,"ATOM@ALIENTEK");
    OLED_ShowString(0,32,"2010/06/3");
    OLED_ShowString(0,48,"ASCII:");
    OLED_ShowString(63,48,"CODE:");
    OLED_Refresh_Gram();
    t=' ';
    while(1)
    {
        OLED_ShowChar(48,48,t,16,1); //显示 ASCII 字符
        OLED_Refresh_Gram();
        t++;
        if(t> '~')t=' ';
        OLED_ShowNum(103,48,t,3,16);//显示 ASCII 字符的码值
        delay_ms(300);
        LED0=!LED0;
    }
}

```

该部分代码用于在 OLED 上显示一些字符，然后从空格键开始不停的循环显示 ASCII 字符集，并显示该字符的 ASCII 值。注意在 main.c 文件里面包含 oled.h 头文件，同时把 oled.c 文件加入到 HARDWARE 组下，然后我们编译此工程，直到编译成功为止。

17.4 下载验证

将代码下载到战舰 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 OLED 模块显示如图 17.4.1 所示：



图 17.4.1 OLED 显示效果

最后一行不停的显示 ASCII 字符以及其码值。通过这一章的学习，我们学会了 ALIENTEK OLED 模块的使用，在调试代码的时候，又多了一种显示信息的途径，在以后的程序编写中，大家可以好好利用。

第十八章 TFTLCD 显示实验

上一章我们介绍了 OLED 模块及其显示，但是该模块只能显示单色/双色，不能显示彩色，而且尺寸也较小。本章我们将介绍 ALIENTEK 2.8 寸 TFT LCD 模块，该模块采用 TFTLCD 面板，可以显示 16 位色的真彩图片。在本章中，我们将利用战舰 STM32 开发板上的 LCD 接口，来点亮 TFTLCD，并实现 ASCII 字符和彩色的显示等功能，并在串口打印 LCD 控制器 ID，同时在 LCD 上面显示。本章分为如下几个部分：

- 18.1 TFTLCD 简介
- 18.2 硬件设计
- 18.3 软件设计
- 18.4 下载验证



18.1 TFTLCD&FSMC 简介

本章我们将通过 STM32 的 FSMC 接口来控制 TFTLCD 的显示，所以本节分为两个部分，分别介绍 TFTLCD 和 FSMC。

18.1.1 TFTLCD 简介

TFT-LCD 即薄膜晶体管液晶显示器。其英文全称为：Thin Film Transistor-Liquid Crystal Display。TFT-LCD 与无源 TN-LCD、STN-LCD 的简单矩阵不同，它在液晶显示屏的每一个像素上都设置有一个薄膜晶体管（TFT），可有效地克服非选通时的串扰，使显示液晶屏的静态特性与扫描线数无关，因此大大提高了图像质量。TFT-LCD 也被叫做真彩液晶显示器。

上一章介绍了 OLED 模块，本章，我们给大家介绍 ALIENTEK TFTLCD 模块，该模块有如下特点：

- 1, 2.4' /2.8' /3.5' 3 种大小的屏幕可选。
- 2, 320×240 的分辨率 (3.5' 分辨率为:320*480)。
- 3, 16 位真彩显示。
- 4, 自带触摸屏，可以用来作为控制输入。

本章，我们以 2.8 寸的 ALIENTEK TFTLCD 模块为例介绍，该模块支持 65K 色显示，显示分辨率为 320×240，接口为 16 位的 80 并口，自带触摸屏。

该模块的外观图如图 18.1.1.1 所示：



图 18.1.1.1 ALIENTEK 2.8 寸 TFTLCD 外观图

模块原理图如图 18.1.1.2 所示：

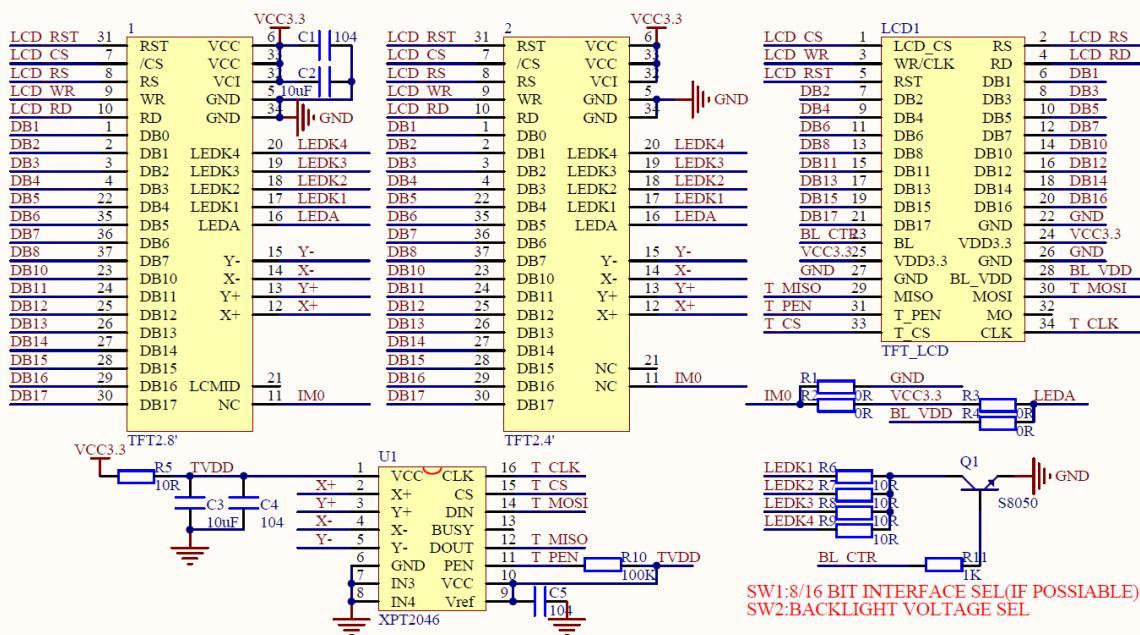


图 18.1.1.2 ALIENTEK 2.8寸 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，接口定义如图 18.1.1.3 所示：

LCD CS	1	LCD CS	RS	2	LCD RS
LCD WR	3	WR/CLK	RD	4	LCD RD
LCD RST	5	RST	DB1	6	DB1
DB2	7	DB2	DB3	8	DB3
DB4	9	DB4	DB5	10	DB5
DB6	11	DB6	DB7	12	DB7
DB8	13	DB8	DB10	14	DB10
DB11	15	DB11	DB12	16	DB12
DB13	17	DB13	DB14	18	DB14
DB15	19	DB15	DB16	20	DB16
DB16	21	DB17	GND	22	GND
DB17	23	BL_CTR3	VDD3.3	24	VCC3.3
		VCC3.325	GND	26	GND
		VDD3.3	BL_VDD	28	BL_VDD
		GND	MOSI	30	T_MOSI
		BL_VDD	MO	32	
		T_CS	CLK	34	T_CLK

图 18.1.1.3 ALIENTEK 2.8寸 TFTLCD 模块接口图

从图 18.1.1.3 可以看出，ALIENTEK TFTLCD 模块采用 16 位的并方式与外部连接，之所以不采用 8 位的方式，是因为彩屏的数据量比较大，尤其在显示图片的时候，如果用 8 位数据线，就会比 16 位方式慢一倍以上，我们当然希望速度越快越好，所以我们选择 16 位的接口。图 18.1.3 还列出了触摸屏芯片的接口，关于触摸屏本章我们不多介绍，后面的章节会有详细的介绍。该模块的 80 并口有如下一些信号线：

CS：TFTLCD 片选信号。

WR：向 TFTLCD 写入数据。

RD：从 TFTLCD 读取数据。

D[15: 0]：16 位双向数据线。



RST: 硬复位 TFTLCD。

RS: 命令/数据标志 (0, 读写命令; 1, 读写数据)。

80 并口在上一节我们已经有详细的介绍了, 这里我们就不再介绍, 需要说明的是, TFTLCD 模块的 RST 信号线是直接接到 STM32 的复位脚上, 并不由软件控制, 这样可以省下来一个 IO 口。另外我们还需要一个背光控制线来控制 TFTLCD 的背光。所以, 我们总共需要的 IO 口数目为 21 个。这里还需要注意, 我们标注的 DB1~DB8, DB10~DB17, 是相对于 LCD 控制 IC 标注的, 实际上大家可以把他们就等同于 D0~D15, 这样理解起来就比较简单一点。

ALIENTEK 提供的 2.8 寸 TFTLCD 模块, 其驱动芯片有很多种类型, 比如有: ILI9320/ILI9325 /ILI9328/ILI9341/SSD1289/LGDP4531/LGDP4535/R61505/SPFD5408/ RM68021 等(具体的型号, 大家可以通过下载本章实验代码, 通过串口或者 LCD 显示查看), 这里我们仅以 ILI9320 控制器为例进行介绍, 其他的控制基本都类似, 我们就不详细阐述了。

ILI9320 液晶控制器自带显存, 其显存总大小为 17280 (240*320*18/8), 即 18 位模式 (26 万色) 下的显存量。模块的 16 位数据线与显寸的对应关系如图 18.1.1.4 所示:

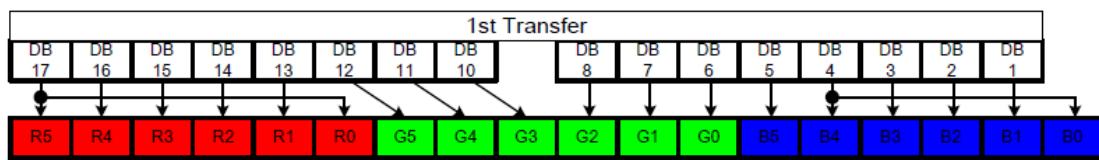


图 18.1.1.4 16 位数据与显存对应关系图

最低 5 位代表蓝色, 中间 6 位为绿色, 最高 5 位为红色。数值越大, 表示该颜色越深。

接下来, 我们介绍一下 ILI9320 的几个重要命令, 因为 ILI9320 的命令很多, 我们这里不可能一一介绍, 有兴趣的大家可以找到 ILI9320 的 datasheet 看看。里面对这些命令有详细的介绍。这里我们要介绍的命令列表如表 18.1.1.1 所示:

编号	指令	各位描述															命令	
		D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
R0	0X00	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	OSC	打开振荡器/读取控制器型号
		1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	
R3	0X03	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0	入口模式
R7	0X07	0	0	PTDE1	PTDE0	0	0	0	BASEE	0	0	GON	DTE	CL	0	D1	D0	显示控制
R32	0X20	0	0	0	0	0	0	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	行地址(X)设置	
R33	0X21	0	0	0	0	0	0	AD16	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	列地址(Y)设置	
R34	0X22	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	写数据到GRAM	
R80	0X50	0	0	0	0	0	0	0	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0	行起始地址(X)设置	
R81	0X51	0	0	0	0	0	0	0	HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0	行结束地址(X)设置	
R82	0X52	0	0	0	0	0	0	VSA8	VSA7	VSA6	VSA5	VSA4	VSA3	VSA2	VSA1	VSA0	列起始地址(Y)设置	
R83	0X53	0	0	0	0	0	0	VEA8	VEA7	VEA6	VEA5	VEA4	VEA3	VEA2	VEA1	VEA0	列结束地址(Y)设置	

表 18.1.1.1 ILI9320 常用命令表

R0, 这个命令, 有两个功能, 如果对它写, 则最低位为 OSC, 用于开启或关闭振荡器。而如果对它读操作, 则返回的是控制器的型号。这个命令最大的功能就是通过读它可以得到控制器的型号, 而我们代码在知道了控制器的型号之后, 可以针对不同型号的控制器, 进行不同的初始化。因为 93xx 系列的初始化, 其实都比较类似, 我们完全可以用一个代码兼容好几个控制器。

R3, 入口模式命令。我们重点关注的是 I/D0、I/D1、AM 这 3 个位, 因为这 3 个位控制了屏幕的显示方向。

AM: 控制 GRAM 更新方向。当 AM=0 的时候, 地址以行方向更新。当 AM=1 的时候, 地址以列方向更新。

I/D[1: 0]: 当更新了一个数据之后, 根据这两个位的设置来控制地址计数器自动增加/减少 1,

其关系如图 18.1.1.5 所示:



	I/D[1:0] = 00 行方向：减少 列方向：减少	I/D[1:0] = 01 行方向：增加 列方向：减少	I/D[1:0] = 10 行方向：减少 列方向：增加	I/D[1:0] = 11 行方向：增加 列方向：增加
AM = 0 行方向				
AM = 1 列方向				

图 18.1.1.5 GRAM 显示方向设置图

通过这几个位的设置，我们就可以控制屏幕的显示方向了，这种方法虽然简单，但是不是很通用，比如不同的液晶，可能这里差别就比较大，有的甚至无法通用！比如 9341 和 9320 就完全不通用。

R7，显示控制命令。该命令 CL 位用来控制是 8 位彩色，还是 26 万色。为 0 时 26 万色，为 1 时八位色。D1、D0、BASEE 这三个位用来控制显示开关与否的。当全部设置为 1 的时候开启显示，全 0 是关闭。我们一般通过该命令的设置来开启或关闭显示器，以降低功耗。

R32, R33, 设置 GRAM 的行地址和列地址。R32 用于设置列地址（X 坐标，0~239），R33 用于设置行地址（Y 坐标，0~319）。当我们要在某个指定点写入一个颜色的时候，先通过这两个命令设置到该点，然后写入颜色值就可以了。

R34，写数据到 GRAM 命令，当写入了这个命令之后，地址计数器才会自动的增加和减少。该命令是我们要介绍的这一组命令里面唯一的单个操作的命令，只需要写入该值就可以了，其他的都是要先写入命令编号，然后写入操作数。

R80~R83，行列 GRAM 地址位置设置。这几个命令用于设定你显示区域的大小，我们整个屏的大小为 240*320，但是有时候我们只需要在其中的一部分区域写入数据，如果用先写坐标，后写数据这样的方式来实现，则速度大打折扣。此时我们就可以通过这几个命令，在其中开辟一个区域，然后不停的丢数据，地址计数器就会根据 R3 的设置自动增加/减少，这样就不需要频繁的写地址了，大大提高了刷新的速度。

命令部分，我们就为大家介绍到这里，我们接下来看看要如何才能驱动 ALIENTEK TFTLCD 模块，这里 TFTLCD 模块的初始化和我们前面介绍的 OLED 模块的初始化框图是一样的，只是初始化代码部分不同。接下来我们也是将该模块用米来显示字符和数字。通过以上介绍，我们可以得出 TFTLCD 显示需要的相关设置步骤如下：

1) 设置 STM32 与 TFTLCD 模块相连接的 IO。

这一步，先将我们与 TFTLCD 模块相连的 IO 口进行初始化，以便驱动 LCD。这里我们用到的是 FSMC，FSMC 将在 18.1.2 节向大家详细介绍。

2) 初始化 TFTLCD 模块。

其实这里就是上和上面 OLED 模块的初始化过程差不多。通过向 TFTLCD 写入一系列的设置，来启动 TFTLCD 的显示。为后续显示字符和数字做准备。

3) 通过函数将字符和数字显示到 TFTLCD 模块上。



这里就是通过我们设计的程序，将要显示的字符送到 TFTLCD 模块就可以了，这些函数将在软件设计部分向大家介绍。通过以上三步，我们就可以使用 ALIENTEK TFTLCD 模块来显示字符和数字了，并且可以显示各种颜色的背景。

18.1.2 FSMC 简介

大容量，且引脚数在 100 脚以上的 STM32F103 芯片都带有 FSMC 接口，ALIENTEK 战舰 STM32 开发板的主芯片为 STM32F103ZET6，是带有 FSMC 接口的。

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡连接。STM32 的 FSMC 接口支持包括 SRAM、NAND FLASH、NOR FLASH 和 PSRAM 等存储器。FSMC 的框图如图 18.1.2.1 所示：

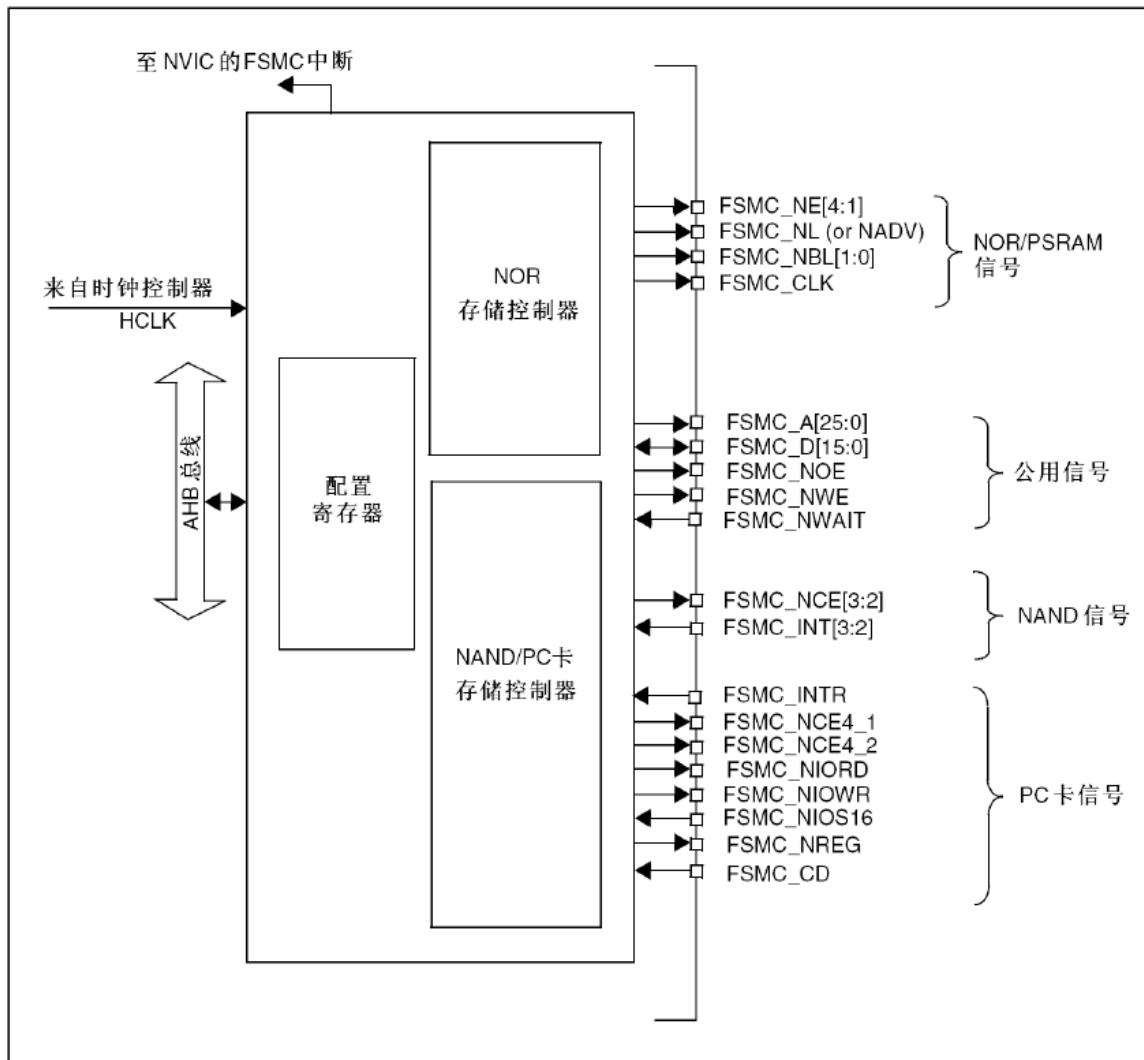


图 18.1.2.1 FSMC 框图

从上图我们可以看出，STM32 的 FSMC 将外部设备分为 3 类：NOR/PSRAM 设备、NAND 设备、PC 卡设备。他们共用地址数据总线等信号，他们具有不同的 CS 以区分不同的设备，比如本章我们用到的 TFTLCD 就是用的 FSMC_NE4 做片选，其实就是将 TFTLCD 当成 SRAM 来控制。

这里我们介绍下为什么可以把 TFTLCD 当成 SRAM 设备用：首先我们了解下外部 SRAM



的连接，外部 SRAM 的控制一般有：地址线（如 A0~A18）、数据线（如 D0~D15）、写信号（WE）、读信号（OE）、片选信号（CS），如果 SRAM 支持字节控制，那么还有 UB/LB 信号。而 TFTLCD 的信号我们在 18.1.1 节有介绍，包括：RS、D0~D15、WR、RD、CS、RST 和 BL 等，其中真正在操作 LCD 的时候需要用到的就只有：RS、D0~D15、WR、RD 和 CS。其操作时序和 SRAM 的控制完全类似，唯一不同就是 TFTLCD 有 RS 信号，但是没有地址信号。

TFTLCD 通过 RS 信号来决定传送的数据是数据还是命令，本质上可以理解为一个地址信号，比如我们把 RS 接在 A0 上面，那么当 FSMC 控制器写地址 0 的时候，会使得 A0 变为 0，对 TFTLCD 来说，就是写命令。而 FSMC 写地址 1 的时候，A0 将会变为 1，对 TFTLCD 来说，就是写数据了。这样，就把数据和命令区分开了，他们其实就是对应 SRAM 操作的两个连续地址。当然 RS 也可以接在其他地址线上，战舰 STM32 开发板是把 RS 连接在 A10 上面的。

STM32 的 FSMC 支持 8/16/32 位数据宽度，我们这里用到的 LCD 是 16 位宽度的，所以在设置的时候，选择 16 位宽就 OK 了。我们再来看看 FSMC 的外部设备地址映像，STM32 的 FSMC 将外部存储器划分为固定大小为 256M 字节的四个存储块，如图 18.1.2.2 所示：

地址	存储块	支持的存储器类型
6000 0000h 6FFF FFFFh	块 1 4 × 64 MB	NOR / PSRAM
7000 0000h 7FFF FFFFh	块 2 4 × 64 MB	NAND 闪存
8000 0000h 8FFF FFFFh	块 3 4 × 64 MB	
9000 0000h 9FFF FFFFh	块 4 4 × 64 MB	PC 卡

图 18.1.2.2 FSMC 存储块地址映像

从上图可以看出，FSMC 总共管理 1GB 空间，拥有 4 个存储块（Bank），本章，我们用到的是块 1，所以在本章我们仅讨论块 1 的相关配置，其他块的配置，请参考《STM32 参考手册》第 19 章（324 页）的相关介绍。

STM32 的 FSMC 存储块 1（Bank1）被分为 4 个区，每个区管理 64M 字节空间，每个区都有独立的寄存器对所连接的存储器进行配置。Bank1 的 256M 字节空间由 28 根地址线



(HADDR[27:0]) 寻址。

这里 HADDR 是内部 AHB 地址总线，其中 HADDR[25:0]来自外部存储器地址 FSMC_A[25:0]，而 HADDR[26:27]对 4 个区进行寻址。如表 18.1.2.1 所示：

Bank1 所选区	片选信号	地址范围	HADDR	
			[27:26]	[25:0]
第 1 区	FSMC_NE1	0X6000, 0000~63FF, FFFF	00	FSMC_A[25:0]
第 2 区	FSMC_NE2	0X6400, 0000~67FF, FFFF	01	
第 3 区	FSMC_NE3	0X6800, 0000~6BFF, FFFF	10	
第 4 区	FSMC_NE4	0X6C00, 0000~6FFF, FFFF	11	

表 18.1.2.1 Bank1 存储区选择表

表 18.1.2.1 中，我们要特别注意 HADDR[25:0]的对应关系：

当 Bank1 接的是 16 位宽度存储器的时候： HADDR[25:1] → FSMC[24:0]。

当 Bank1 接的是 8 位宽度存储器的时候： HADDR[25:0] → FSMC[25:0]。

不论外部接 8 位/16 位宽设备，FSMC_A[0]永远接在外部设备地址 A[0]。 这里，TFTLCD 使用的是 16 位数据宽度，所以 HADDR[0]并没有用到，只有 HADDR[25:1]是有效的，对应关系变为： HADDR[25:1] → FSMC[24:0]，相当于右移了一位，这里请大家特别留意。另外，HADDR[27:26]的设置，是不需要我们干预的，比如：当你选择使用 Bank1 的第三个区，即使用 FSMC_NE3 来连接外部设备的时候，即对应了 HADDR[27:26]=10，我们要做的就是配置对应第 3 区的寄存器组，来适应外部设备即可。STM32 的 FSMC 各 Bank 配置寄存器如表 18.1.2.2 所示：

内部控制器	存储块	管理的地址范围	支持的设备类型	配置寄存器
NOR FLASH 控制器	Bank1	0X6000, 0000~0X6FFF, FFFF	SRAM/ROM NOR FLASH PSRAM	FSMC_BCR1/2/3/4 FSMC_BTR1/2/2/3 FSMC_BWTR1/2/3/4
NAND FLASH /PC CARD 控制器	Bank2	0X7000, 0000~0X7FFF, FFFF	NAND FLASH	FSMC_PCR2/3/4 FSMC_SR2/3/4
	Bank3	0X8000, 0000~0X8FFF, FFFF		FSMC_PMEM2/3/4 FSMC_PATT2/3/4
	Bank4	0X9000, 0000~0X9FFF, FFFF	PC Card	FSMC_PI04

表 18.1.2.2 FSMC 各 Bank 配置寄存器表

对于 NOR FLASH 控制器，主要是通过 FSMC_BCRx、FSMC_BTRx 和 FSMC_BWTRx 寄存器设置（其中 x=1~4，对应 4 个区）。通过这 3 个寄存器，可以设置 FSMC 访问外部存储器的时序参数，拓宽了可选用的外部存储器的速度范围。FSMC 的 NOR FLASH 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要的设置的时间参数有 2 个：

- 1, HCLK 与 FSMC_CLK 的分频系数(CLKDIV)，可以为 2~16 分频；
- 2, 同步突发访问中获得第 1 个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。FSMC 综合了 SRAM / ROM、PSRAM 和 NOR Flash 产品的信号特点，定义了 4 种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数，如表 18.1.2.3 所列：



时序模型		简单描述	时间参数
异步	Mode1	SRAM/CRAM 时序	DATAST、ADDSET
	ModeA	SRAM/CRAM OE 选通型时序	DATAST、ADDSET
	Mode2/B	NOR FLASH 时序	DATAST、ADDSET
	ModeC	NOR FLASH OE 选通型时序	DATAST、ADDSET
	ModeD	延长地址保持时间的异步时序	DATAST、ADDSET、ADDHLK
同步突发		根据同步时钟 FSMC_CK 读取多个顺序单元的数据	CLKDIV、DATLAT

表 18.1.2.3 NOR FLASH 控制器支持的时序模型

在实际扩展时，根据选用存储器的特征确定时序模型，从而确定各时间参数与存储器读 / 写周期参数指标之间的计算关系；利用该计算关系和存储芯片数据手册中给定的参数指标，可计算出 FSMC 所需要的各时间参数，从而对时间参数寄存器进行合理的配置。

本章，我们使用异步模式 A（ModeA）方式来控制 TFTLCD，模式 A 的读操作时序如图 18.1.2.3 所示：

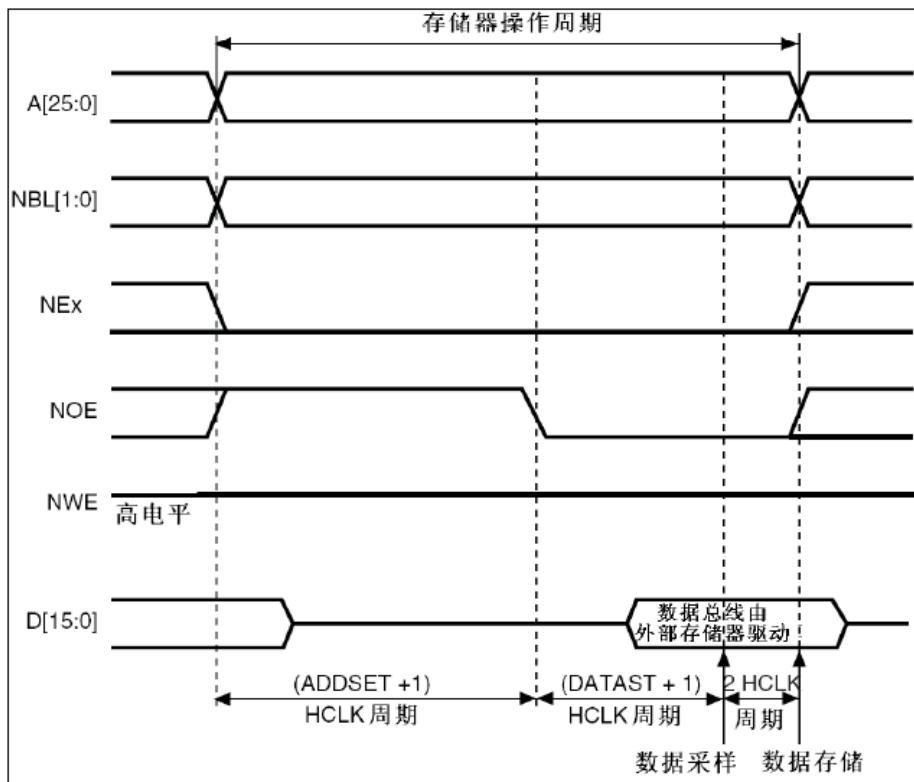


图 18.1.2.3 模式 A 读操作时序图

模式 A 支持独立的读写时序控制，这个对我们驱动 TFTLCD 来说非常有用，因为 TFTLCD 在读的时候，一般比较慢，而在写的时候可以比较快，如果读写用一样的时序，那么只能以读的时序为基准，从而导致写的速度变慢，或者在读数据的时候，重新配置 FSMC 的延时，在读操作完成的时候，再配置回写的时序，这样虽然也不会降低写的速度，但是频繁配置，比较麻烦。而如果有独立的读写时序控制，那么我们只要初始化的时候配置好，之后就不用再配置，既可以满足速度要求，又不需要频繁改配置。

模式 A 的写操作时序如图 18.1.2.4 所示：

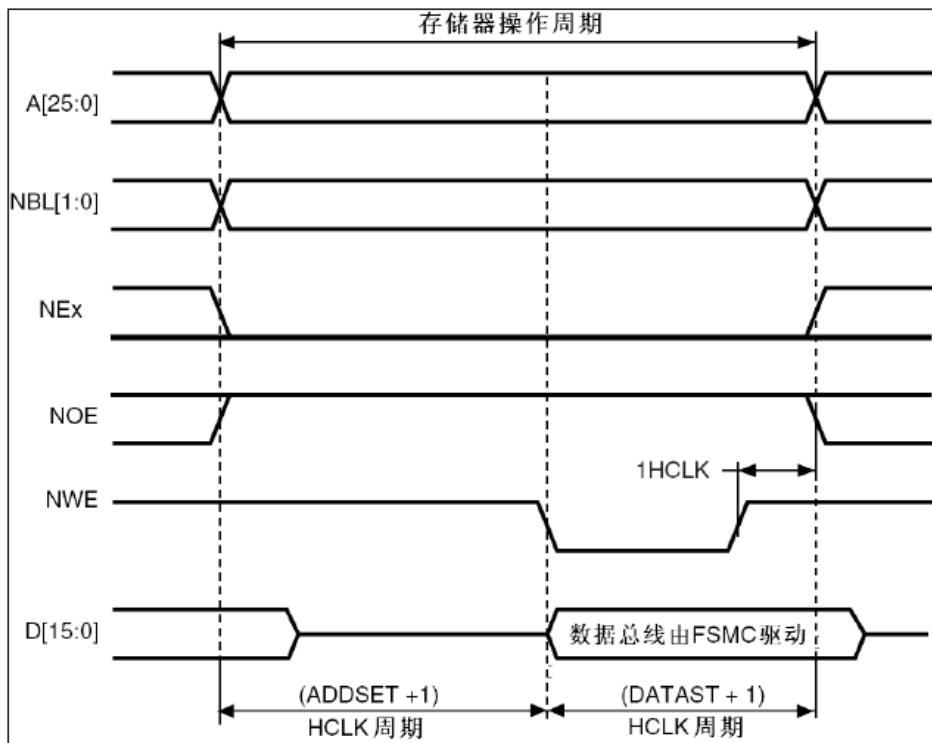


图 18.1.2.4 模式 A 写操作时序

从模式 A 的读写时序图，我们可以看出，读操作还存在额外的 2 个 HCLK 周期，用于数据存储，所以同样的配置读操作一般比写操作会慢一点。图 18.1.2.3 和图 18.1.2.4 中的 ADDSET 与 DATAST，是通过不同的寄存器设置的，接下来我们讲解一下 Bank1 的几个控制寄存器

首先，我们介绍 SRAM/NOR 闪存片选控制寄存器：FSMC_BCRx (x=1~4)，该寄存器各位描述如图 18.1.2.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留																															

res CBURSTRW 保留 EXTMOD WAITEN WREN WAITCFG WRAPMOD WAITPOL BURSTEN 保留 FACCE MWID MTYP MUXEN MBKEN

18.1.2.5 FSMC_BCRx 寄存器各位描述

该寄存器我们在本章用到的设置有：EXTMOD、WREN、MWID、MTYP 和 MBKEN 这几个设置，我们将逐个介绍。

EXTMOD: 扩展模式使能位，也就是是否允许读写不同的时序，很明显，我们本章需要读写不同的时序，故该位需要设置为 1。

WREN: 写使能位。我们需要向 TFTLCD 写数据，故该位必须设置为 1。

MWID[1:0]: 存储器数据总线宽度。00，表示 8 位数据模式；01 表示 16 位数据模式；10 和 11 保留。我们的 TFTLCD 是 16 位数据线，所以设置 WMID[1:0]=01。

MTYP[1:0]: 存储器类型。00 表示 SRAM、ROM；01 表示 PSRAM；10 表示 NOR FLASH；11 保留。前面提到，我们把 TFTLCD 当成 SRAM 用，所以需要设置 MTYP[1:0]=00。

MBKEN: 存储块使能位。这个容易理解，我们需要用到该存储块控制 TFTLCD，当然要使能这个存储块了。

接下来，我们看看 SRAM/NOR 闪存片选时序寄存器：FSMC_BTRx (x=1~4)，该寄存器各位描述如图 18.1.2.6 所示：

图 18.1.2.6 FSMC BTRx 寄存器各位描述

这个寄存器包含了每个存储器块的控制信息，可以用于 SRAM、ROM 和 NOR 闪存存储器。如果 FSMC_BCRx 寄存器中设置了 EXTMOD 位，则有两个时序寄存器分别对应读(本寄存器)和写操作(FSMC_BWTRx 寄存器)。因为我们要求读写分开时序控制，所以 EXTMOD 是使能了的，也就是本寄存器是读操作时序寄存器，控制读操作的相关时序。本章我们要用到的设置有：ACCMOD、DATAST 和 ADDSET 这三个设置。

ACCMOD[1:0]: 访问模式。00 表示访问模式 A; 01 表示访问模式 B; 10 表示访问模式 C; 11 表示访问模式 D, 本章我们用到模式 A, 故设置为 00。

DATAST[7:0]: 数据保持时间。0 为保留设置，其他设置则代表保持时间为: (DATAST +1) 个 HCLK 时钟周期，最大为 256 个 HCLK 周期。对 ILI9320 来说，其实就是 RD 低电平持续时间，一般为 150ns。而一个 HCLK 时钟周期为 13.8ns 左右 (1/72Mhz)，为了兼容其他屏，我们这里设置 DATAST 为 15，也就是 16 个 HCLK 周期，时间大约是 234ns (未计算数据存储的 2 个 HCLK 时间)。

ADDSET[3:0]: 地址建立时间。其建立时间为: (ADDSET+1) 个 HCLK 周期, 最大为 16 个 HCLK 周期。对 ILI9320 来说, 这里相当于 RD 高电平持续时间, 本来这里我们应该设置和 DATAST 一样, 但是由于 CS 切换延时的存在, 我们这里可以设置 ADDSET 为较小的值, 本章我们设置 ADDSET 为 1, 即 2 个 HCLK 周期, 同样可以正常使用。

最后，我们再来看看 SRAM/NOR 闪写时序寄存器：FSMC_BWTRx (x=1~4)，该寄存器各位描述如图 18.1.2.7 所示：

图 18.1.2.7 FSMC_BWTRx 寄存器各位描述

该寄存器在本章用作写操作时序控制寄存器，需要用到的设置同样是：ACCMOD、DATAST 和 ADDSET 这三个设置。这三个设置的方法同 FSMC_BTRx 一模一样，只是这里对应的是写操作的时序，ACCMOD 设置同 FSMC_BTRx 一模一样，同样是选择模式 A，另外 DATAST 和 ADDSET 则对应低电平和高电平持续时间，对 ILI9320 来说，这两个时间只需要 50ns 就够了，比读操作快得多。所以我们这里设置 DATAST 为 3，即 4 个 HCLK 周期，时间约为 55ns。同样由于 CS 切换延时的存在，我们可以设置 ADDSET 为 0，即 1 个 HCLK 周期。

至此，我们对 STM32 的 FSMC 介绍就差不多了，通过以上两个小节的了解，我们可以开始写 LCD 的驱动代码了。不过，这里还要给大家做下科普，在 MDK 的寄存器定义里面，并没有定义 FSMC_BCRx、FSMC_BTRx、FSMC_BWTRx 等这个单独的寄存器，而是将他们进行了一些组合。

FSMC BCR_x 和 FSMC BTR_x，组合成 BTCSR[8]寄存器组，他们的对应关系如下：

BTCR[0]对应 FSMC BCR1, BTCR[1]对应 FSMC BTR1

BTCR[2]对应FSMC BCR2, BTCR[3]对应FSMC BTR2



BTCR[4]对应 FSMC_BCR3, BTCR[5]对应 FSMC_BTR3

BTCR[6]对应 FSMC_BCR4, BTCR[7]对应 FSMC_BTR4

FSMC_BWTRx 则组合成 BWTR[7], 他们的对应关系如下:

BWTR[0]对应 FSMC_BWTR1, BWTR[2]对应 FSMC_BWTR2,

BWTR[4]对应 FSMC_BWTR3, BWTR[6]对应 FSMC_BWTR4,

BWTR[1]、BWTR[3]和 BWTR[5]保留, 没有用到。

通过上面的讲解, 通过对 FSMC 相关的寄存器的描述, 大家对 FSMC 的原理有了一个初步的认识, 如果还不熟悉的朋友, 请一定要搜索网络资料理解 FSMC 的原理。只有理解了原理, 使用库函数才可以得心应手。那么在库函数中是怎么实现 FSMC 的配置的呢? FSMC_BCRx, FSMC_BTRx 寄存器在库函数是通过什么函数来配置的呢? 下面我们来讲解一下 FSMC 相关的库函数:

1.FSMC 初始化函数

根据前面的讲解, 初始化 FSMC 主要是初始化三个寄存器 FSMC_BCRx, FSMC_BTRx, FSMC_BWTRx, 那么在固件库中是怎么初始化这三个参数的呢?

固件库提供了 3 个 FSMC 初始化函数分别为

```
FSMC_NORSRAMInit();
FSMC_NANDInit();
FSMC_PCCARDInit();
```

这三个函数分别用来初始化 4 种类型存储器。这里根据名字就很好判断对应关系。用来初始化 NOR 和 SRAM 使用同一个函数 FSMC_NORSRAMInit()。所以我们之后使用的 FSMC 初始化函数为 FSMC_NORSRAMInit()。下面我们看看函数定义:

```
void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct);
```

这个函数只有一个入口参数, 也就是 FSMC_NORSRAMInitTypeDef 类型指针变量, 这个结构体的成员变量非常多, 因为 FSMC 相关的配置项非常多。

```
typedef struct
{
    uint32_t FSMC_Bank;
    uint32_t FSMC_DataAddressMux;
    uint32_t FSMC_MemoryType;
    uint32_t FSMC_MemoryDataWidth;
    uint32_t FSMC_BurstAccessMode;
    uint32_t FSMC_AsynchronousWait;
    uint32_t FSMC_WaitSignalPolarity;
    uint32_t FSMC_WrapMode;
    uint32_t FSMC_WaitSignalActive;
    uint32_t FSMC_WriteOperation;
    uint32_t FSMC_WaitSignal;
    uint32_t FSMC_ExtendedMode;
    uint32_t FSMC_WriteBurst;
    FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;
    FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;
}FSMC_NORSRAMInitTypeDef;
```

从这个结构体我们可以看出, 前面有 13 个基本类型 (unit32_t) 的成员变量, 这 13 个参数是用



来配置片选控制寄存器 `FSMC_BCRx`。最后面还有两个

`FSMC_NORSRAMTimingInitTypeDef` 指针类型的成员变量。前面我们讲到，`FSMC` 有读时序和写时序之分，所以这里就是用来设置读时序和写时序的参数了，也就是说，这两个参数是用来配置寄存器 `FSMC_BTRx` 和 `FSMC_BWTRx`，后面我们会讲解到。下面我们主要来看看模式 A 下的相关配置参数：

参数 `FSMC_Bank` 用来设置使用到的存储块标号和区号，前面讲过，我们是使用的存储块 1 区号 4，所以选择值为 `FSMC_Bank1_NORSRAM4`。

参数 `FSMC_MemoryType` 用来设置存储器类型，我们这里是 SRAM，所以选择值为 `FSMC_MemoryType_SRAM`。

参数 `FSMC_MemoryDataWidth` 用来设置数据宽度，可选 8 位还是 16 位，这里我们是 16 位数据宽度，所以选择值为 `FSMC_MemoryDataWidth_16b`。

参数 `FSMC_WriteOperation` 用来设置写使能，毫无疑问，我们前面讲解过我们要向 TFT 写数据，所以要写使能，这里我们选择 `FSMC_WriteOperation_Enable`。

参数 `FSMC_ExtendedMode` 是设置扩展模式使能位，也就是是否允许读写不同的时序，这里我们采取的读写不同时序，所以设置值为 `FSMC_ExtendedMode_Enable`。

上面的这些参数是与模式 A 相关的，下面我们也来稍微了解一下其他几个参数的意义吧：

参数 `FSMC_DataAddressMux` 用来设置地址/数据复用使能，若设置为使能，那么地址的低 16 位和数据将共用数据总线，仅对 NOR 和 PSRAM 有效，所以我们设置为默认值不复用，值 `FSMC_DataAddressMux_Disable`。

参数 `FSMC_BurstAccessMode`，`FSMC_AsynchronousWait`，`FSMC_WaitSignalPolarity`，`FSMC_WaitSignalActive`，`FSMC_WrapMode`，`FSMC_WaitSignal`，`FSMC_WriteBurst` 和 `FSMC_WaitSignal` 这些参数在成组模式同步模式才需要设置，大家可以参考中文参考手册了解相关参数的意思。

接下来我们看看设置读写时序参数的两个变量 `FSMC_ReadWriteTimingStruct` 和 `FSMC_WriteTimingStruct`，他们都是 `FSMC_NORSRAMTimingInitTypeDef` 结构体指针类型，这两个参数在初始化的时候分别用来初始化片选控制寄存器 `FSMC_BTRx` 和写操作时序控制寄存器 `FSMC_BWTRx`。下面我们看看 `FSMC_NORSRAMTimingInitTypeDef` 类型的定义：

```
typedef struct
{
    uint32_t FSMC_AddressSetupTime;
    uint32_t FSMC_AddressHoldTime;
    uint32_t FSMC_DataSetupTime;
    uint32_t FSMC_BusTurnAroundDuration;
    uint32_t FSMC_CLKDivision;
    uint32_t FSMC_DataLatency;
    uint32_t FSMC_AccessMode;
}FSMC_NORSRAMTimingInitTypeDef;
```

这个结构体有 7 个参数用来设置 `FSMC` 读写时序。其实这些参数的意思我们前面在讲解 `FSMC` 的时序的时候有提到，主要是设计地址建立保持时间，数据建立时间等等配置，对于我们的实验中，读写时序不一样，读写速度要求不一样，所以对于参数 `FSMC_DataSetupTime` 设置了不同的值，大家可以对照理解一下。记住，这些参数的意义在前面讲解 `FSMC_BTRx` 和 `FSMC_BWTRx` 寄存器的时候都有提到，大家可以翻过去看看。



2.FSMC 使能函数

FSMC 对不同的存储器类型同样提供了不同的使能函数：

```
void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);
void FSMC_NANDCmd(uint32_t FSMC_Bank, FunctionalState NewState);
void FSMC_PCCARDCmd(FunctionalState NewState);
```

这个就比较好理解，我们这里不讲解，我们是 SRAM，所以使用的第一函数。

18.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) TFTLCD 模块

TFTLCD 模块的电路见图 18.1.1.2，这里我们介绍 TFTLCD 模块与 ALIETEK 战舰 STM32 开发板的连接，战舰 STM32 开发板底板的 LCD 接口和 ALIETEK TFTLCD 模块直接可以对插，连接关系如图 18.2.1 所示：

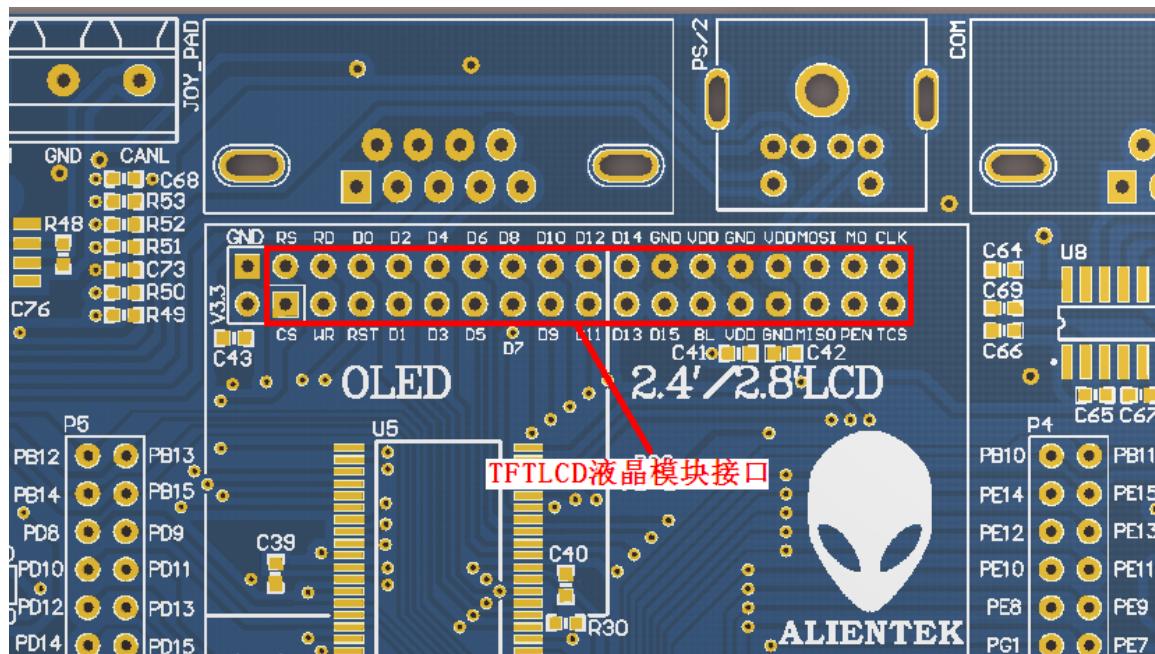


图 18.2.1 TFTLCD 与开发板连接示意图

图 18.2.1 中圈出来的部分就是连接 TFTLCD 模块的接口，板上的接口比液晶模块的插针要多 2 个口，液晶模块在这里是靠右插的。多出的 2 个口是给 OLED 用的，所以 OLED 模块在接这里的的时候，是靠左插的，这个请大家注意一下。

在硬件上，TFTLCD 模块与战舰 STM32 开发板的 IO 口对应关系如下：

- LCD_BL(背光控制)对应 PB0;
- LCD_CS 对应 PG12 即 FSMC_NE4;
- LCD_RS 对应 PG0 即 FSMC_A10;
- LCD_WR 对应 PD5 即 FSMC_NWE;
- LCD_RD 对应 PD4 即 FSMC_NOE;
- LCD_D[15:0]则直接连接在 FSMC_D15~FSMC_D0;

这些线的连接，战舰 STM32 开发板的内部已经连接好了，我们只需要将 TFTLCD 模块插



上去就好了。实物连接如图 18.2.2 所示：

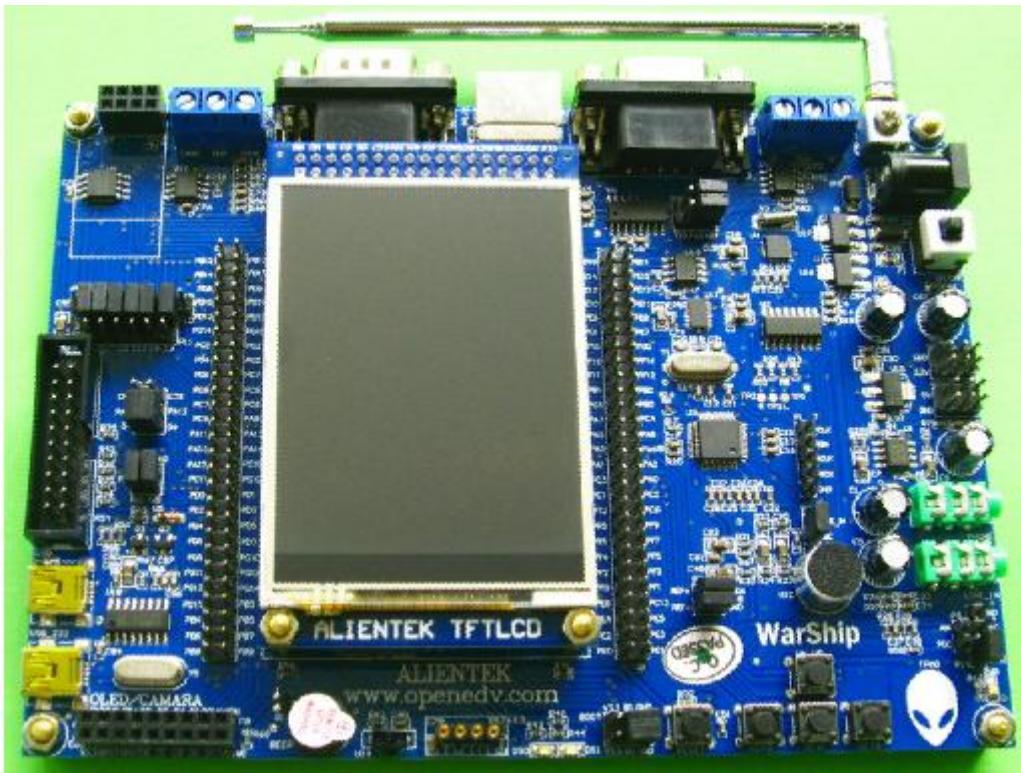


图 18.2.2 TFTLCD 与开发板连接实物图

18.3 软件设计

打开我们光盘的 TFT LCD 显示实验工程可以看到我们添加了两个文件 lcd.c 和头文件 lcd.h。同时，FSMC 相关的库函数分布在 stm32f10x_fsmc.c 文件和头文件 stm32f10x_fsmc.h 中。

在 lcd.c 里面代码比较多，我们这里就不贴出来了，只针对几个重要的函数进行讲解。完整版的代码见光盘→4，程序源码→标准例程-V3.5 库函数版本→实验 13 TFTLCD 显示实验的 lcd.c 文件。

本实验，我们用到 FSMC 驱动 LCD，通过前面的介绍，我们知道 TFTLCD 的 RS 接在 FSMC 的 A10 上面，CS 接在 FSMC_NE4 上，并且是 16 位数据总线。即我们使用的是 FSMC 存储器 1 的第 4 区，我们定义如下 LCD 操作结构体（在 lcd.h 里面定义）：

```
//LCD 操作结构体
typedef struct
{
    u16 LCD_REG;
    u16 LCD_RAM;
} LCD_TypeDef;

//使用 NOR/SRAM 的 Bank1.sector4,地址位 HADDR[27,26]=11 A10 作为数据命令区分线
//注意 16 位数据总线时，STM32 内部地址会右移一位对齐!
#define LCD_BASE      ((u32)(0x6C000000 | 0x0000007FE))
#define LCD          ((LCD_TypeDef *) LCD_BASE)
```

其中 LCD_BASE，必须根据我们外部电路的连接来确定，我们使用 Bank1.sector4 就是从地址 0X6C000000 开始，而 0X0000007FE，则是 A10 的偏移量。我们将这个地址强制转换为



LCD_TypeDef 结构体地址，那么可以得到 LCD->LCD_REG 的地址就是 0X6C00,07FE，对应 A10 的状态为 0(即 RS=0)，而 LCD->LCD_RAM 的地址就是 0X6C00,0800（结构体地址自增），对应 A10 的状态为 1（即 RS=1）。

所以，有了这个定义，当我们要往 LCD 写命令/数据的时候，可以这样写：

```
LCD->LCD_REG=CMD; //写命令  
LCD->LCD_RAM=DATA; //写数据
```

而读的时候反过来操作就可以了，如下所示：

```
CMD= LCD->LCD_REG; //读 LCD 寄存器  
DATA = LCD->LCD_RAM; //读 LCD 数据
```

这其中，CS、WR、RD 和 IO 口方向都是由 FSMC 控制，不需要我们手动设置了。接下来，我们先介绍一下 lcd.h 里面的另一个重要结构体：

```
//LCD 重要参数集  
typedef struct  
{  
    u16 width;           //LCD 宽度  
    u16 height;          //LCD 高度  
    u16 id;              //LCD ID  
    u8 dir;              //横屏还是竖屏控制：0，竖屏；1，横屏。  
    u8 wramcmd;          //开始写 gram 指令  
    u8 setxcmd;           //设置 x 坐标指令  
    u8 setycmd;           //设置 y 坐标指令  
}_lcd_dev;  
//LCD 参数  
extern _lcd_dev lcddev; //管理 LCD 重要参数
```

该结构体用于保存一些 LCD 重要参数信息，比如 LCD 的长宽、LCD ID（驱动 IC 型号）、LCD 横竖屏状态等，这个结构体虽然占用了 10 个字节的内存，但是却可以让我们的驱动函数支持不同尺寸的 LCD，同时可以实现 LCD 横竖屏切换等重要功能，所以还是利大于弊的。有了以上了解，下面我们开始介绍 lcd.c 里面的一些重要函数。

先看 6 个简单，但是很重要的函数：

```
//写寄存器函数  
//regval:寄存器值  
void LCD_WR_REG(u16 regval)  
{  
    LCD->LCD_REG=regval; //写入要写的寄存器序号  
}  
//写 LCD 数据  
//data:要写入的值  
void LCD_WR_DATA(u16 data)  
{  
    LCD->LCD_RAM=data;  
}  
//读 LCD 数据  
//返回值:读到的值
```



```
u16 LCD_RD_DATA(void)
{
    return LCD->LCD_RAM;
}

//写寄存器
//LCD_Reg:寄存器地址
//LCD_RegValue:要写入的数据
void LCD_WriteReg(u8 LCD_Reg, u16 LCD_RegValue)
{
    LCD->LCD_REG = LCD_Reg;           //写入要写的寄存器序号
    LCD->LCD_RAM = LCD_RegValue;     //写入数据
}

//读寄存器
//LCD_Reg:寄存器地址
//返回值:读到的数据
u16 LCD_ReadReg(u8 LCD_Reg)
{
    LCD_WR_REG(LCD_Reg);            //写入要读的寄存器序号
    delay_us(5);
    return LCD_RD_DATA();           //返回读到的值
}

//开始写 GRAM
void LCD_WriteRAM_Prepare(void)
{
    LCD->LCD_REG=lcddev.wramcmd;
}
```

因为 FSMC 自动控制了 WR/RD/CS 等这些信号，所以这 6 个函数实现起来都非常简单，我们就不多说，实现功能见函数前面的备注，通过这几个简单函数的组合，我们就可以对 LCD 进行各种操作了。

第七个要介绍的函数是坐标设置函数，该函数代码如下：

```
//设置光标位置
//Xpos:横坐标
//Ypos:纵坐标
void LCD_SetCursor(u16 Xpos, u16 Ypos)
{
    if(lcddev.id==0X9341)
    {
        LCD_WR_REG(lcddev.setxcmnd);
        LCD_WR_DATA(Xpos>>8);
        LCD_WR_DATA(Xpos&0xFF);
        LCD_WR_REG(lcddev.setycmd);
        LCD_WR_DATA(Ypos>>8);
        LCD_WR_DATA(Ypos&0xFF);
    }
}
```



```

if(lcddev.dir==1)Xpos=lcddev.width-1-Xpos; //横屏其实就是调转 x,y 坐标
LCD_WriteReg(lcddev.setxcmd, Xpos);
LCD_WriteReg(lcddev.setycmd, Ypos);
}
}

```

该函数实现将 LCD 的当前操作点设置到指定坐标(x,y)。因为 9341 的设置同其他屏有些不太一样，所以单独对 9341 进行了设置。

接下来我们介绍第八个函数：画点函数。该函数实现代码如下：

```

//画点
//x,y:坐标
//POINT_COLOR:此点的颜色
void LCD_DrawPoint(u16 x,u16 y)
{
    LCD_SetCursor(x,y);           //设置光标位置
    LCD_WriteRAM_Prepate();      //开始写入 GRAM
    LCD->LCD_RAM=POINT_COLOR;
}

```

该函数实现比较简单，就是先设置坐标，然后往坐标写颜色。其中 POINT_COLOR 是我们定义的一个全局变量，用于存放画笔颜色，顺带介绍一下另外一个全局变量：BACK_COLOR，该变量代表 LCD 的背景色。LCD_DrawPoint 函数虽然简单，但是至关重要，其他几乎所有上层函数，都是通过调用这个函数实现的。

有了画点，当然还需要有读点的函数，第九个介绍的函数就是读点函数，用于读取 LCD 的 GRAM，这里说明一下，为什么 OLED 模块没做读 GRAM 的函数，而这里做了。因为 OLED 模块是单色的，所需要全部 GRAM 也就 1K 个字节，而 TFTLCD 模块为彩色的，点数也比 OLED 模块多很多，以 16 位色计算，一款 320×240 的液晶，需要 $320 \times 240 \times 2$ 个字节来存储颜色值，也就是也需要 150K 字节，这对任何一款单片机来说，都不是一个小数目了。而且我们在图形叠加的时候，可以先读回原来的值，然后写入新的值，在完成叠加后，我们又恢复原来的值。这样在做一些简单菜单的时候，是很有用的。这里我们读取 TFTLCD 模块数据的函数为 LCD_ReadPoint，该函数直接返回读到的 GRAM 值。该函数使用之前要先设置读取的 GRAM 地址，通过 LCD_SetCursor 函数来实现。LCD_ReadPoint 的代码如下：

```

//读取个某点的颜色值
//x,y:坐标
//返回值:此点的颜色
u16 LCD_ReadPoint(u16 x,u16 y)
{
    u16 r=0,g=0,b=0;
    if(x>=lcddev.width||y>=lcddev.height) return 0; //超过了范围,直接返回
    LCD_SetCursor(x,y);
    if(lcddev.id==0X9341)LCD_WR_REG(0X2E);           //9341 发送读 GRAM 指令
    else LCD_WR_REG(R34);                            //其他 IC 发送读 GRAM 指令
    if(lcddev.id==0X9320)opt_delay(2);                //FOR 9320,延时 2us
    if(LCD->LCD_RAM)r=0;                           //dummy Read
    opt_delay(2);
}

```



```

r=LCD->LCD_RAM;           //实际坐标颜色
if(lcddev.id==0X9341)//9341 要分 2 次读出
{
    opt_delay(2);
    b=LCD->LCD_RAM;
    g=r&0xFF;      //对于 9341,第一次读取的是 RG 的值,R 在前,G 在后,各占 8 位
    g<<=8;
}
//这几种 IC 直接返回颜色值
if(lcddev.id==0X4535||lcddev.id==0X4531||lcddev.id==0X8989||lcddev.id==0XB505)return r;
else if(lcddev.id==0X9341)return (((r>>11)<<11)|((g>>10)<<5)|(b>>11));
//ILI9341 需要公式转换一下
else return LCD_BGR2RGB(r);           //其他 IC
}

```

在 LCD_ReadPoint 函数中，因为我们的代码不止支持一种 LCD 驱动器，所以，我们根据不同的 LCD 驱动器 ((lcddev.id) 型号，执行不同的操作，以实现对各个驱动器兼容，提高函数的通用性。

第十个要介绍的是字符显示函数 LCD_ShowChar，该函数同前面 OLED 模块的字符显示函数差不多，但是这里的字符显示函数多了 1 个功能，就是可以以叠加方式显示，或者以非叠加方式显示。叠加方式显示多用于在显示的图片上再显示字符。非叠加方式一般用于普通的显示。该函数实现代码如下：

```

//在指定位置显示一个字符
//x,y:起始坐标
//num:要显示的字符:" --->"~
//size:字体大小 12/16
//mode:叠加方式(1)还是非叠加方式(0)
void LCD_ShowChar(u16 x,u16 y,u8 num,u8 size,u8 mode)
{
    u8 temp,t1,t;
    u16 y0=y;
    u16 colortemp=POINT_COLOR;
    //设置窗口
    num=num-'';//得到偏移后的值
    if(!mode) //非叠加方式
    {
        for(t=0;t<size;t++)
        {
            if(size==12)temp=asc2_1206[num][t]; //调用 1206 字体
            else temp=asc2_1608[num][t];         //调用 1608 字体
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)POINT_COLOR=colortemp;
                else POINT_COLOR=BACK_COLOR;

```



```
LCD_DrawPoint(x,y);
temp<<=1;
y++;
if(x>=lcddev.height) return; //超区域了
if((y-y0)==size)
{
    y=y0;
    x++;
    if(x>=lcddev.width) return; //超区域了
    break;
}
}

}else//叠加方式
{
for(t=0;t<size;t++)
{
    if(size==12)temp=asc2_1206[num][t]; //调用 1206 字体
    else temp=asc2_1608[num][t]; //调用 1608 字体
    for(t1=0;t1<8;t1++)
    {
        if(temp&0x80)LCD_DrawPoint(x,y);
        temp<<=1;
        y++;
        if(x>=lcddev.height) return; //超区域了
        if((y-y0)==size)
        {
            y=y0;
            x++;
            if(x>=lcddev.width) return; //超区域了
            break;
        }
    }
}
POINT_COLOR=colortemp;
}
```

在 LCD_ShowChar 函数里面，我们采用画点函数来显示字符，虽然速度不如开辟窗口的方式，但是这样写可以有更好的兼容性，方便在不同 LCD 之间移植。该代码中我们用到了两个字符集点阵数据数组 asc2_1206 和 asc2_1608，这两个字符集的点阵数据的提取方式，同十七章介绍的提取方法是一模一样的。详细请参考第十七章。

最后，我们再介绍一下 TFTLCD 模块的初始化函数 LCD_Init，该函数先初始化 STM32 与 TFTLCD 连接的 IO 口，并配置 FSMC 控制器，然后读取 LCD 控制器的型号，根据控制 IC 的



型号执行不同的初始化代码，其简化代码如下：

```
//初始化 lcd
//该初始化函数可以初始化各种 ILI93XX 液晶,但是其他函数是基于 ILI9320 的!!!
//在其他型号的驱动芯片上没有测试!
void LCD_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    FSMC_NORSRAMInitTypeDef  FSMC_NSInitStructure;
    FSMC_NORSRAMTimingInitTypeDef  readWriteTiming;
    FSMC_NORSRAMTimingInitTypeDef  writeTiming;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC,ENABLE); //使能 FSMC 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|RCC_APB2Periph_GPIOD|
    RCC_APB2Periph_GPIOE|RCC_APB2Periph_GPIOG|
    RCC_APB2Periph_AFIO,ENABLE); // ①使能 GPIO 以及 AFIO 复用功能时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;           //PB0 推挽输出 背光
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;     //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);             //②初始化 PB0

    //PORTD 复用推挽输出
    GPIO_InitStructure.GPIO_Pin= GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_4|GPIO_Pin_5|
    GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_14|GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);              //②初始化 PORTD

    //PORTE 复用推挽输出
    GPIO_InitStructure.GPIO_Pin =GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|
    GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOE, &GPIO_InitStructure);              //②初始化 PORTE

    //PORTG12 复用推挽输出 A0
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_12; //PORTD 复用推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;         //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOG, &GPIO_InitStructure);                //②初始化 PORTG

    readWriteTiming.FSMC_AddressSetupTime = 0x01; //地址建立时间 2 个 HCLK 1
    readWriteTiming.FSMC_AddressHoldTime = 0x00; //地址保持时间模式 A 未用到
    readWriteTiming.FSMC_DataSetupTime = 0x0f; // 数据保存时间为 16 个 HCLK
```



```
readWriteTiming.FSMC_BusTurnAroundDuration = 0x00;
readWriteTiming.FSMC_CLKDivision = 0x00;
readWriteTiming.FSMC_DataLatency = 0x00;
readWriteTiming.FSMC_AccessMode = FSMC_AccessMode_A; //模式 A

writeTiming.FSMC_AddressSetupTime = 0x00; //地址建立时间为 1 个 HCLK
writeTiming.FSMC_AddressHoldTime = 0x00; //地址保持时间 (A
writeTiming.FSMC_DataSetupTime = 0x03; //数据保存时间为 4 个 HCLK
writeTiming.FSMC_BusTurnAroundDuration = 0x00;
writeTiming.FSMC_CLKDivision = 0x00;
writeTiming.FSMC_DataLatency = 0x00;
writeTiming.FSMC_AccessMode = FSMC_AccessMode_A; //模式 A

FSMC_NInitStruct.FSMC_Bank = FSMC_Bank1_NORSRAM4;//这里我们使
//用 NE4, 也就对应 BTCR[6],[7]。
FSMC_NInitStruct.FSMC_DataAddressMux =
    FSMC_DataAddressMux_Disable; //不复用数据地址
FSMC_NInitStruct.FSMC_MemoryType=FSMC_MemoryType_SRAM;// SRAM
FSMC_NInitStruct.FSMC_MemoryDataWidth= FSMC_MemoryDataWidth_16b;
//存储器数据宽度为 16bit
FSMC_NInitStruct.FSMC_BurstAccessMode=FSMC_BurstAccessMode_Disable;
FSMC_NInitStruct.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
FSMC_NInitStruct.FSMC_AsynchronousWait=
    FSMC_AsynchronousWait_Disable;
FSMC_NInitStruct.FSMC_WrapMode = FSMC_WrapMode_Disable;
FSMC_NInitStruct.FSMC_WaitSignalActive=
    FSMC_WaitSignalActive_BeforeWaitState;
FSMC_NInitStruct.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
//存储器写使能
FSMC_NInitStruct.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NInitStruct.FSMC_ExtendedMode = FSMC_ExtendedMode_Enable;
// 读写使用不同的时序
FSMC_NInitStruct.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
FSMC_NInitStruct.FSMC_ReadWriteTimingStruct = &readWriteTiming;
FSMC_NInitStruct.FSMC_WriteTimingStruct = &writeTiming; //写时序
FSMC_NORSRAMInit(&FSMC_NInitStruct); //③初始化 FSMC 配置

FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM4, ENABLE); // ④使能 BANK1
delay_ms(50); // delay 50 ms
LCD_WriteReg(0x0000,0x0001);
delay_ms(50); // delay 50 ms
lcddev.id = LCD_ReadReg(0x0000);
if(lcddev.id==0||lcddev.id==0xFFFF)//读到 ID 不正确 ⑤
```



```
{  
    //尝试 9341 的 ID 读取  
    LCD_WR_REG(0XD3);  
    LCD_RD_DATA();           //dummy read  
    LCD_RD_DATA();           //读到 0X00  
    lcddev.id=LCD_RD_DATA(); //读取 93  
    lcddev.id<<=8;  
    lcddev.id|=LCD_RD_DATA(); //读取 41  
    if(lcddev.id!=0X9341)      //非 9341,尝试是不是 6804  
    {  
        LCD_WR_REG(0XBF);  
        LCD_RD_DATA();           //dummy read  
        LCD_RD_DATA();           //读回 0X01  
        LCD_RD_DATA();           //读回 0XD0  
        lcddev.id=LCD_RD_DATA(); //这里读回 0X68  
        lcddev.id<<=8;  
        lcddev.id|=LCD_RD_DATA(); //这里读回 0X04  
    }  
    if(lcddev.id!=0X9341&&lcddev.id!=0X6804)  
        //不是 9341 也不是 6804, 尝试看看是不是 NT35310  
    {  
        LCD_WR_REG(0XD4);  
        LCD_RD_DATA();           //dummy read  
        LCD_RD_DATA();           //读回 0X01  
        lcddev.id=LCD_RD_DATA(); //读回 0X53  
        lcddev.id<<=8;  
        lcddev.id|=LCD_RD_DATA(); //这里读回 0X10  
    }  
}  
printf(" LCD ID:%x\r\n",lcddev.id);      //打印 LCD ID  
if(lcddev.id==0X9341)                  //9341 初始化  
{  
    .....//9341 初始化代码  
}else if(lcddev.id==0x9325)            //9325  
{  
    .....//9325 初始化代码  
}else if(lcddev.id==0x9328)            //ILI9328 OK  
{  
    .....//9328 初始化代码  
}else if(lcddev.id==0x9320||lcddev.id==0x9300)//未测试.  
{  
    .....//9300 初始化代码  
}else if(lcddev.id==0X9331)
```



```
{  
    .....//9331 初始化代码  
}else if(lcddev.id==0x5408)  
{  
    .....//5408 初始化代码  
}  
else if(lcddev.id==0x1505)//OK  
{  
    .....//1505 初始化代码  
}else if(lcddev.id==0xB505)  
{  
    .....//B505 初始化代码  
}else if(lcddev.id==0xC505)  
{  
    .....//C505 初始化代码  
}else if(lcddev.id==0x8989)  
{  
    .....//8989 初始化代码  
}else if(lcddev.id==0x4531)  
{  
    .....//4531 初始化代码  
}else if(lcddev.id==0x4535)  
{  
    .....//4535 初始化代码  
}  
LCD_Display_Dir(0);           //默认为竖屏显示  
LCD_LED=1;                   //点亮背光  
LCD_Clear(WHITE);  
}
```

从初始化代码可以看出，LCD 初始化步骤为①~⑤在代码中标注：

- ①GPIO,FSMC,AFIO 时钟使能。
- ②GPIO 初始化： GPIO_Init() 函数。
- ③FSMC 初始化： FSMC_NORSRAMInit() 函数。
- ④FSMC 使能： FSMC_NORSRAMCmd() 函数。

⑥ 同的 LCD 驱动器的初始化代码。

该函数先对 FSMC 相关 IO 进行初始化，然后是 FSMC 的初始化，这个我们在前面都有介绍，最后根据读到的 LCD ID，对不同的驱动器执行不同的初始化代码，从上面的代码可以看出，这个初始化函数可以针对十多款不同的驱动 IC 执行初始化操作，这样大大提高了整个程序的通用性。大家在以后的学习中应该多使用这样的方式，以提高程序的通用性、兼容性。

特别注意：本函数使用了 printf 来打印 LCD ID，所以，如果你在主函数里面没有初始化串口，那么将导致程序死在 printf 里面！！如果不想用 printf，那么请注释掉它。

大家可以打开 lcd.h 和 lcd.c 文件看里面的代码，了解各个函数功能。接下来，我们打开 main.c 内容如下：

```

int main(void)
{
    u8 x=0;
    u8 lcd_id[12];           //存放 LCD ID 字符串
    delay_init();             //延时函数初始化
    NVIC_Configuration();    //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);          //串口初始化波特率为 9600
    LED_Init();               //LED 端口初始化
    LCD_Init();
    POINT_COLOR=RED;
    sprintf((char*)lcd_id,"LCD ID:%04X",lcddev.id);//将 LCD ID 打印到 lcd_id 数组。
    while(1)
    {
        switch(x)
        {
            case 0:LCD_Clear(WHITE);break;
            case 1:LCD_Clear(BLACK);break;
            case 2:LCD_Clear(BLUE);break;
            case 3:LCD_Clear(RED);break;
            case 4:LCD_Clear(MAGENTA);break;
            case 5:LCD_Clear(GREEN);break;
            case 6:LCD_Clear(CYAN);break;
            case 7:LCD_Clear(YELLOW);break;
            case 8:LCD_Clear(BRRED);break;
            case 9:LCD_Clear(GRAY);break;
            case 10:LCD_Clear(LGRAY);break;
            case 11:LCD_Clear(BROWN);break;
        }
        POINT_COLOR=RED;
        LCD_ShowString(30,50,200,16,16,"WarShip STM32 ^_^");
        LCD_ShowString(30,70,200,16,16,"TFTLCD TEST");
        LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(30,110,200,16,16	lcd_id);      //显示 LCD ID
        LCD_ShowString(30,130,200,16,16,"2012/9/5");
        x++;
        if(x==12)x=0;
        LED0=!LED0;
        delay_ms(1000);
    }
}

```

该部分代码将显示一些固定的字符，同时显示 LCD 驱动 IC 的型号，然后不停的切换背景颜色，每 1s 切换一次。而 LED0 也会不停的闪烁，指示程序已经在运行了。其中我们用到一个 sprintf 的函数，该函数用法同 printf，只是 sprintf 把打印内容输出到指定的内存区间上，sprintf



的详细用法，请百度。

在编译通过之后，我们开始下载验证代码。

18.4 下载验证

将程序下载到战舰 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块的显示如图 18.4.1 所示：



图 18.4.1 TFTLCD 显示效果图

我们可以看到屏幕的背景是不停切换的，同时 DS0 不停的闪烁，证明我们的代码被正确的执行了，达到了我们预期的目的。

第十九章 USMART 调试组件实验

本章，我们将向大家介绍一个十分重要的辅助调试工具：USMART 调试组件。该组件由 ALIENTEK 开发提供，功能类似 linux 的 shell（RTT 的 finsh 也属于此类）。USMART 最主要的功能就是通过串口调用单片机里面的函数，并执行，对我们调试代码是很有帮助的。本章分为如下几个部分：

- 19.1 USMART 调试组件简介
- 19.2 硬件设计
- 19.3 软件设计
- 19.4 下载验证



19.1 USMART 调试组件简介

USMART 是由 ALIENTEK 开发的一个灵巧的串口调试互交组件，通过它你可以通过串口助手调用程序里面的任何函数，并执行。因此，你可以随意更改函数的输入参数(支持数字(10/16 进制)、字符串、函数入口地址等作为参数)，单个函数最多支持 10 个输入参数，并支持函数返回值显示，目前最新版本为 V3.1。

USMART 的特点如下：

- 1, 可以调用绝大部分用户直接编写的函数。
- 2, 资源占用极少 (最少情况: FLASH:4K; SRAM:72B)。
- 3, 支持参数类型多 (数字 (包含 10/16 进制)、字符串、函数指针等)。
- 4, 支持函数返回值显示。
- 5, 支持参数及返回值格式设置。
- 6, 支持函数执行时间计算 (V3.1 版本新特性)。
- 7, 使用方便。

有了 USMART，你可以轻易的修改函数参数、查看函数运行结果，从而快速解决问题。比如你调试一个摄像头模块，需要修改其中的几个参数来得到最佳的效果，普通的做法：写函数->修改参数->下载->看结果->不满意->修改参数->下载->看结果->不满意....不停的循环，直到满意为止。这样做很麻烦不说，单片机也是有寿命的啊，老这样不停的刷，很折寿的。而利用 USMART，则只需要在串口调试助手里面输入函数及参数，然后直接串口发送给单片机，就执行了一次参数调整，不满意的话，你在串口调试助手修改参数在发送就可以了，直到你满意为止。这样，修改参数十分方便，不需要编译、不需要下载、不会让单片机折寿。

USMART 支持的参数类型基本满足任何调试了，支持的类型有：10 或者 16 进制数字、字符串指针（如果该参数是用作参数返回的话，可能会有问题！）、函数指针等。因此绝大部分函数，可以直接被 USMART 调用，对于不能直接调用的，你只需要重写一个函数，把影响调用的参数去掉即可，这个重写后的函数，即可以被 USMART 调用了。

USMART 的实现流程简单概括就是：第一步，添加需要调用的函数（在 usmart_config.c 里面的 usmart_nametab 数组里面添加）；第二步，初始化串口；第三步，初始化 USMART（通过 usmart_init 函数实现）；第四步，轮询 usmart_scan 函数，处理串口数据。

经过以上简单介绍，我们对 USMART 有了个大概了解，接下来我们来简单介绍下 USMART 组件的移植。

USMART 组件总共包含 6 文件如图 19.1.1 所示：

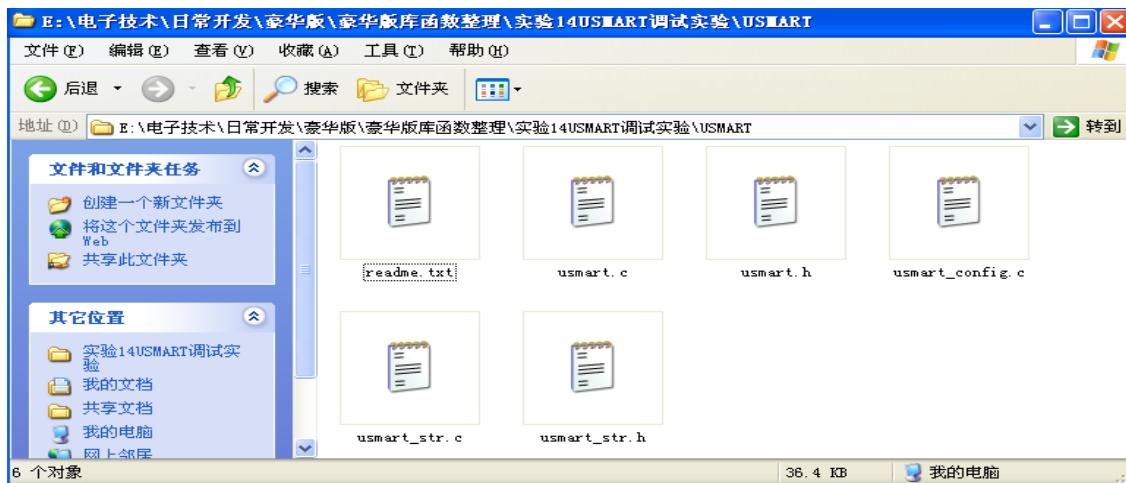




图 19.1.1 USMART 组件代码

其中 redeme.txt 是一个说明文件，不参与编译。其他五个文件，usmart.c 负责与外部互交等。usmat_str.c 主要负责命令和参数解析。usmart_config.c 主要由用户添加需要由 usmart 管理的函数。

usmart.h 和 usmart_str.h 是两个头文件，其中 usmart.h 里面含有几个用户配置宏定义，可以用来配置 usmart 的功能及总参数长度(直接和 SRAM 占用挂钩)、是否使能定时器扫描、是否使用读写函数等。

USMART 的移植，只需要实现 5 个函数。其中 4 个函数都在 usmart.c 里面，另外一个是串口接收函数，必须有由用户自己实现，用于接收串口发送过来的数据。

第一个函数，串口接收函数。该函数，我们是通过 SYSTEM 文件夹默认的串口接收来实现的，该函数在 5.3.1 节有介绍过，我们这里就不列出来了。SYSTEM 文件夹里面的串口接收函数，最大可以一次接收 200 字节，用于从串口接收函数名和参数等。大家如果在其他平台移植，请参考 SYSTEM 文件夹串口接收的实现方式进行移植。

第二个是 void usmart_init(void) 函数，该函数的实现代码如下：

```
//初始化串口控制器
//sysclk:系统时钟 (Mhz)
void usmart_init(u8 sysclk)
{
#if USMART_ENTIMX_SCAN==1
    Timer2_Init(1000,(u32)sysclk*100-1); //分频,时钟为 10K ,100ms 中断一次
    //注意,计数频率必须为 10Khz,以和 runtime 单位(0.1ms)同步.
#endif
    usmart_dev.sptype=1; //十六进制显示参数
}
```

该函数有一个参数 sysclk，就是用于定时器初始化。这里需要说明一下，为了让我们的库函数和寄存器实现函数一致，我们这里不直接通过 SystemCoreClock 来获取系统时钟，直接通过在外面设置的方式，当然你也可以去掉 sysclk 这个参数，这样函数体里面的 Timer2_Init 函数就可修改为 Timer2_Init(1000,(u32)SystemCoreClock/10000-1)。另外 USMART_ENTIMX_SCAN 是在 usmart.h 里面定义的一个是否使能定时器中断扫描的宏定义。如果为 1，就通过定时器初始化函数 Timer2_Init 初始化定时器 2 中断，每 100ms 中断一次，并在中断服务程序 TIM2_IRQHandler 里面调用 usmart_scan 函数进行扫描，这里我们就不列出代码，因为之前的实验对这方面讲解较多。如果为 0，那么需要用户需要自行间隔一定时间（100ms 左右为宜）调用一次 usmart_scan 函数，以实现串口数据处理。**注意：如果要使用函数执行时间统计功能 (runtime 1)，则必须设置 USMART_ENTIMX_SCAN 为 1。另外，为了让统计时间精确到 0.1ms，定时器的计数时钟频率必须设置为 10Khz，否则时间就不是 0.1ms 了。**

第三和第四个函数仅用于服务 USMART 的函数执行时间统计功能(串口指令：runtime 1)，分别是：usmart_reset_runtime 和 usmart_get_runtime，这两个函数代码如下：

```
//复位 runtime
//需要根据所移植到的 MCU 的定时器参数进行修改
void usmart_reset_runtime(void)
{
    TIM2->SR&=~(1<<0); //清除中断标志位
    TIM2->ARR=0xFFFF; //将重装载值设置到最大
```



```
TIM2->CNT=0;          //清空定时器的 CNT
usmart_dev.runtime=0;
}

//获得 runtime 时间
//返回值:执行时间,单位:0.1ms,最大延时时间为定时器 CNT 值的 2 倍*0.1ms
//需要根据所移植到的 MCU 的定时器参数进行修改
u32 usmart_get_runtime(void)
{
    if(TIM2->SR&0X0001)//在运行期间,产生了定时器溢出
    {
        usmart_dev.runtime+=0xFFFF;
    }
    usmart_dev.runtime+=TIM2->CNT;
    return usmart_dev.runtime;      //返回计数值
}
```

这里我们还是利用定时器 2 来做执行时间计算, usmart_reset_runtime 函数在每次 USMART 调用函数之前执行, 清除计数器, 然后在函数执行完之后, 调用 usmart_get_runtime 获取整个函数的运行时间。由于 usmart 调用的函数, 都是在中断里面执行的, 所以我们不太方便再用定时器的中断功能来实现定时器溢出统计, 因此, USMART 的函数执行时间统计功能, 最多可以统计定时器溢出 1 次的时间, 对 STM32 来说, 定时器是 16 位的, 最大计数是 65535, 而由于我们定时器设置的是 0.1ms 一个计时周期 (10Khz), 所以最长计时时间是: $65535*2*0.1\text{ms}=13.1$ 秒。也就是说, 如果函数执行时间超过 13.1 秒, 那么计时将不准确。

最后一个 usmart_scan 函数, 该函数用于执行 usmart 扫描, 该函数需要得到两个参量, 第一个是从串口接收到的数组(USART_RX_BUF), 第二个是串口接收状态(USART_RX_STA)。接收状态包括接收到的数据大小, 以及接收是否完成。该函数代码如下:

```
//usmart 扫描函数
//通过调用该函数,实现 usmart 的各个控制.该函数需要每隔一定时间被调用一次
//以及时执行从串口发过来的各个函数.
//本函数可以在中断里面调用,从而实现自动管理.
//如果非 ALIENTEK 用户,则 USART_RX_STA 和 USART_RX_BUF[]需要用户自己实现
void usmart_scan(void)
{
    u8 sta,len;
    if(USART_RX_STA&0x8000)//串口接收完成?
    {
        len=USART_RX_STA&0x3fff; //得到此次接收到的数据长度
        USART_RX_BUF[len]='\0'; //在末尾加入结束符.
        sta=usmart_dev.cmd_rec(USART_RX_BUF);//得到函数各个信息
        if(sta==0)usmart_dev.exe();//执行函数
        else
        {
            len=usmart_sys_cmd_exe(USART_RX_BUF);
            if(len!=USMART_FUNCERR)sta=len;
        }
    }
}
```



```
if(sta)
{
    switch(sta)
    {
        case USMART_FUNCERR:
            printf("函数错误!\r\n");
            break;
        case USMART_PARMERR:
            printf("参数错误!\r\n");
            break;
        case USMART_PARMOVER:
            printf("参数太多!\r\n");
            break;
        case USMART_NOFUNCIND:
            printf("未找到匹配的函数!\r\n");
            break;
    }
}
USART_RX_STA=0;//状态寄存器清空
}
```

该函数的执行过程：先判断串口接收是否完成（USART_RX_STA 的最高位是否为 1），如果完成，则取得串口接收到的数据长度（USART_RX_STA 的低 14 位），并在末尾增加结束符，再执行解析，解析完之后清空接收标记（USART_RX_STA 置零）。如果没执行完成，则直接跳过，不进行任何处理。

完成这几个函数的移植，你就可以使用 USMART 了。不过，需要注意的是，usmart 同外部的互交，一般是通过 usmart_dev 结构体实现，所以 usmart_init 和 usmart_scan 的调用分别是通过：usmart_dev.init 和 usmart_dev.scan 实现的。

下面，我们将在第十八章实验的基础上，移植 USMART，并通过 USMART 调用一些 TFTLCD 的内部函数，让大家初步了解 USMART 的使用。

19.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0 和 DS1
- 2) 串口
- 3) TFTLCD 模块

这三个硬件在前面章节均有介绍，本章不再介绍。

19.3 软件设计

这里我们在上一章的实验的基础上通过添加文件的方式讲解 USMART 的引入，当然大家也可以直接打开我们光盘的实例工程。打开上一章的工程，复制 USMART 文件夹（该文件夹



可以在光盘的本章实验例程里面找到) 到本工程文件夹下面, 如图 19.3.1 所示:

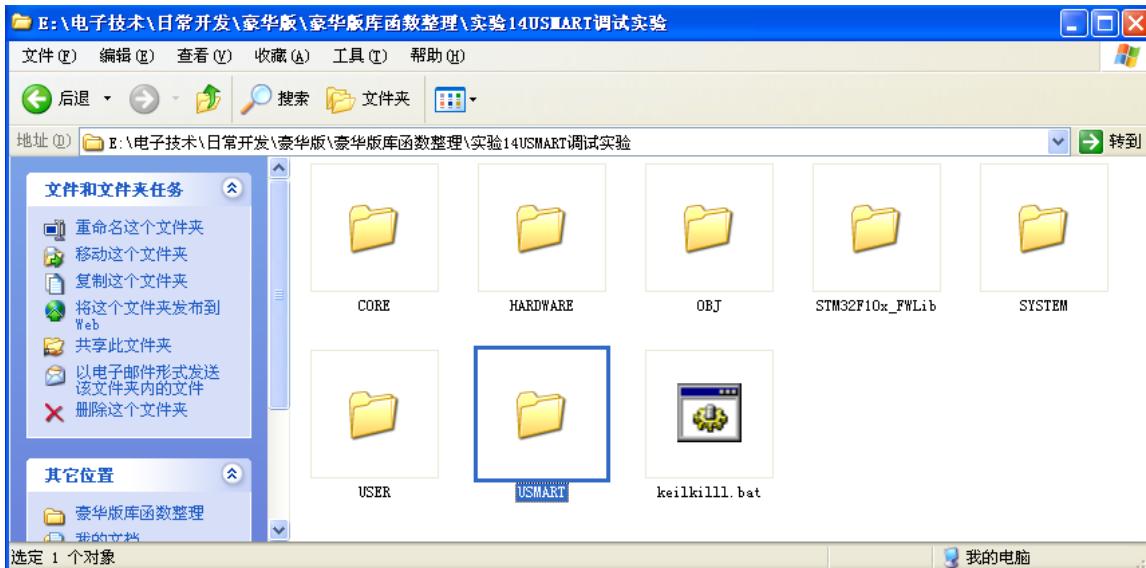


图 19.3.1 复制 USMART 文件夹到工程文件夹下

接着, 我们打开工程, 并新建 USMART 组, 添加 USMART 组件代码, 同时把 USMART 文件夹添加到头文件包含路径, 在主函数里面加入 include “usmart.h” 如图 19.3.2 所示:

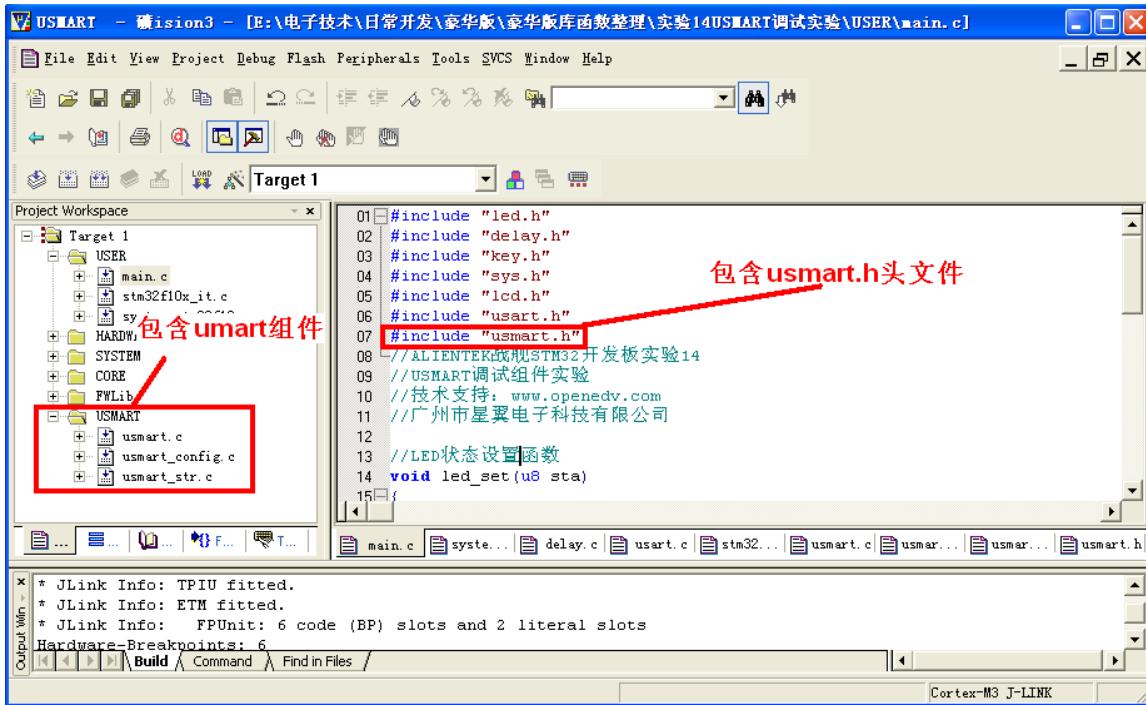


图 19.3.2 添加 USMART 组件代码

由于 USMART 默认提供了 STM32 的 TIM2 中断初始化设置代码, 我们只需要在 usmart.h 里面设置 USMART_ENTIMX_SCAN 为 1, 即可完成 TIM2 的设置, 通过 TIM2 的中断服务函数, 调用 usmart_dev.scan() (就是 usmart_scan 函数), 实现 usmart 的扫描。此部分代码我们就不列出来了, 请参考 usmart.c。

此时, 我们就可以使用 USMART 了, 不过在主程序里面还得执行 usmart 的初始化, 另外还需要针对你自己想要被 USMART 调用的函数在 usmart_config.c 里面进行添加。下面先介绍如何添加自己想要被 USMART 调用的函数, 打开 usmart_config.c, 如图 19.3.3 所示:



函数所在头文件
添加区

```

051 /////////////////////////////////////////////////////////////////// 用户配置区 ///////////////////////////////////////////////////////////////////
052 //这下面要包含所用到的函数所申明的头文件(用户自己添加)
053 #include "delay.h"
054 #include "uart.h"
055 #include "sys.h"
056
057 //函数名列表初始化(用户自己添加)
058 //用户直接在这里输入要执行的函数名及其查找串
059 struct _m_usmart_nametab usmart_nametab[]=
060 {
061
062 #if USMART_USE_WRFUNS==1 //如果使能了读写操作
063     (void*)read_addr,"void read_addr(u32 addr)",
064     (void*)write_addr,"void write_addr(u32 addr,u32 val)",
065 #endif
066     (void*)delay_ms,"void delay_ms(u16 nms)",
067     (void*)delay_us,"void delay_us(u32 nus)",
068 };
069 /////////////////////////////////////////////////////////////////// END ///////////////////////////////////////////////////////////////////

```

用户函数
添加区

test.c usmart.c usmart_c... usmart.h

图 19.3.3 添加需要被 USMART 调用的函数

这里的添加函数很简单，只要把函数所在头文件添加进来，并把函数名按上图所示的方式增加即可，默认我们添加了两个函数：delay_ms 和 delay_us。另外，read_addr 和 write_addr 属于 usmart 自带的函数，用于读写指定地址的数据，通过配置 USMART_USE_WRFUNS，可以使能或者禁止这两个函数。

这里我们根据自己的需要按上图的格式添加其他函数，添加完之后如图 19.3.4 所示：

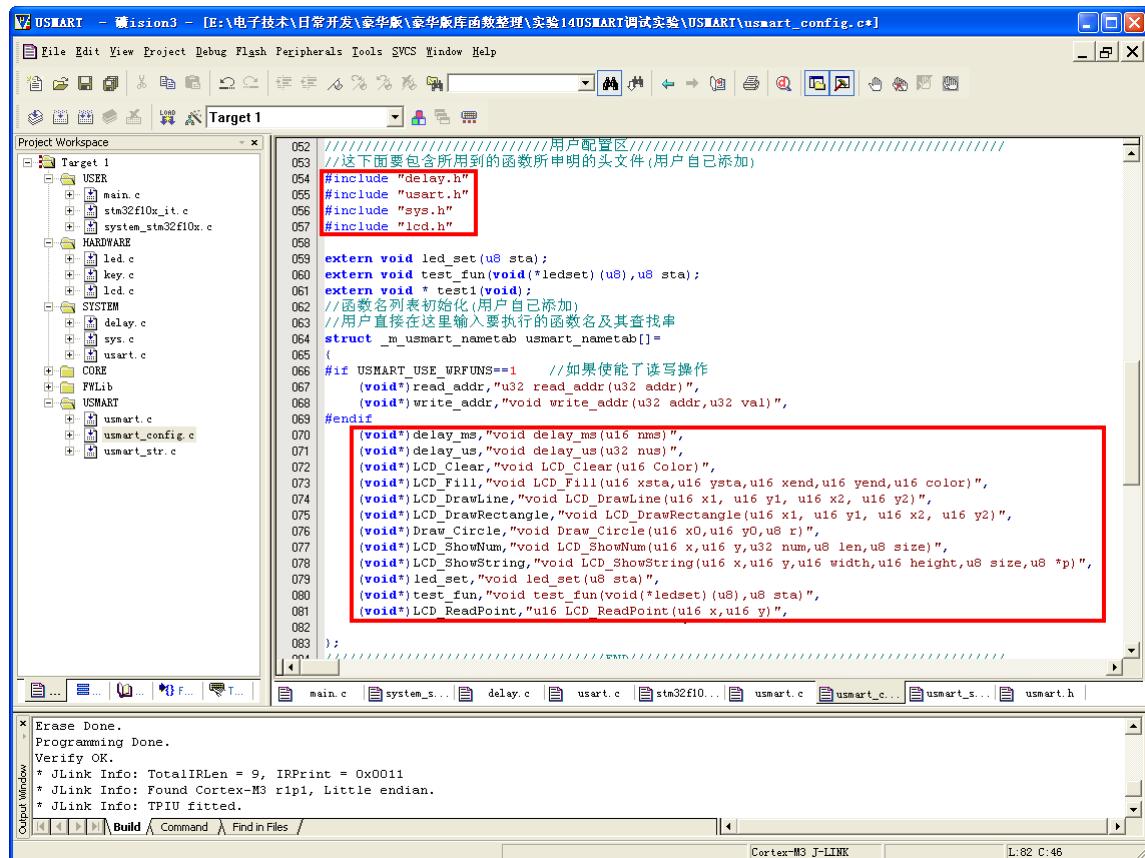


图 19.3.4 添加函数后

上图中，我们添加了 lcd.h，并添加了很多 LCD 函数，最后我们还添加了 led_set 和 test_fun 两个函数，这两个函数在 main.c 里面实现，代码如下：

```
//LED 状态设置函数
void led_set(u8 sta)
{
    LED1=sta;
}

//函数参数调用测试函数
void test_fun(void(*ledset)(u8),u8 sta)
{
    ledset(sta);
}
```

led_set 函数，用于设置 LED1 的状态，而第二个函数 test_fun 则是测试 USMART 对函数参数的支持的，test_fun 的第一个参数是函数，在 USMART 里面也是可以被调用的。

在添加完函数之后，我们修改 main 函数，如下：

```
int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
```



```
LCD_Init();
usmart_dev.init(SystemCoreClock/1000000); //初始化 USMART
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Mini STM32 ^_^");
LCD_ShowString(30,70,200,16,16,"USMART TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2011/6/18");
while(1)
{
    LED0=!LED0; delay_ms(500);
}
}
```

此代码显示简单的信息后，就是在死循环等待串口数据。

至此，整个 usmart 的移植就完成了。编译成功后，就可以下载程序到开发板，开始 USMART 的体验。

19.4 下载验证

将程序下载到战舰 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时，屏幕上显示了一些字符（就是主函数里面要显示的字符）。

我们打开串口调试助手，选择正确的串口号，并选择发送新行（即发送回车键）选项。如下图所示（点击扩展->隐藏，显示扩展界面）：

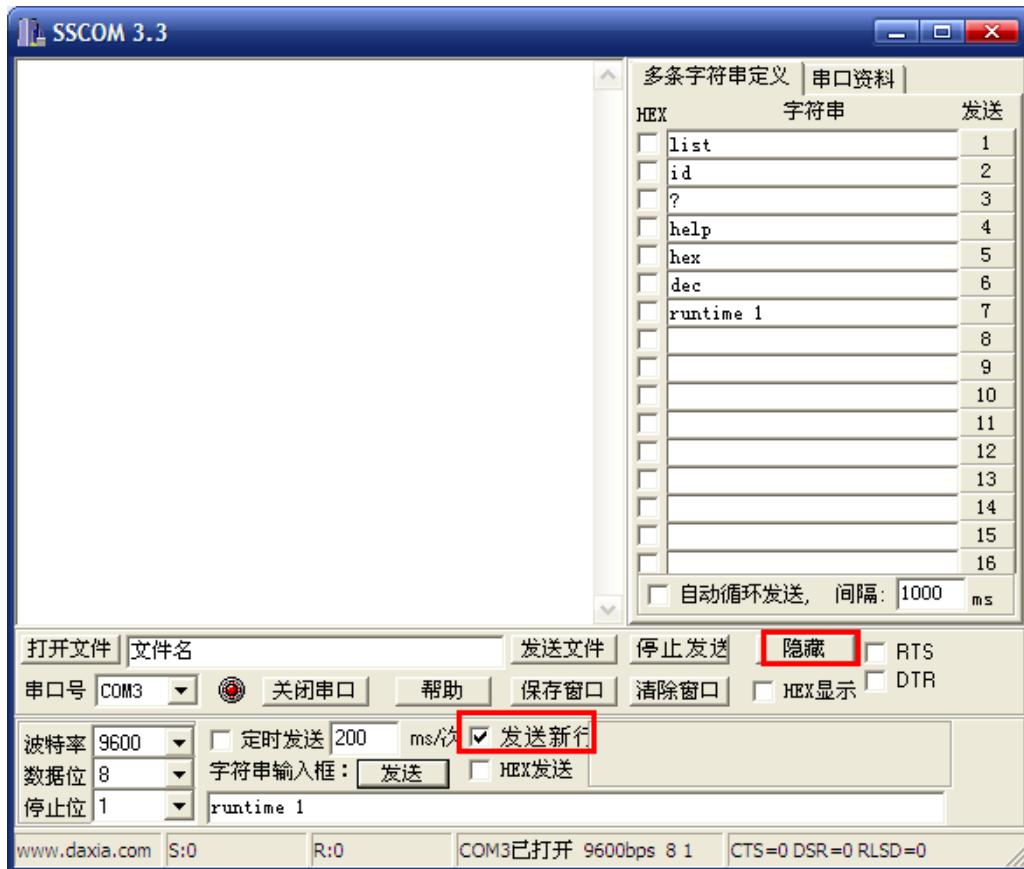




图 19.4.1 驱动串口调试助手

上图中 list、id、?、help、hex、dec 和 runtime 都属于 usmart 自带的系统命令。下面我们简单介绍下这几个命令：

list，该命令用于打印所有 usmart 可调用函数。发送该命令后，串口将受到所有能被 usmart 调用得到函数如图 19.4.2 所示：

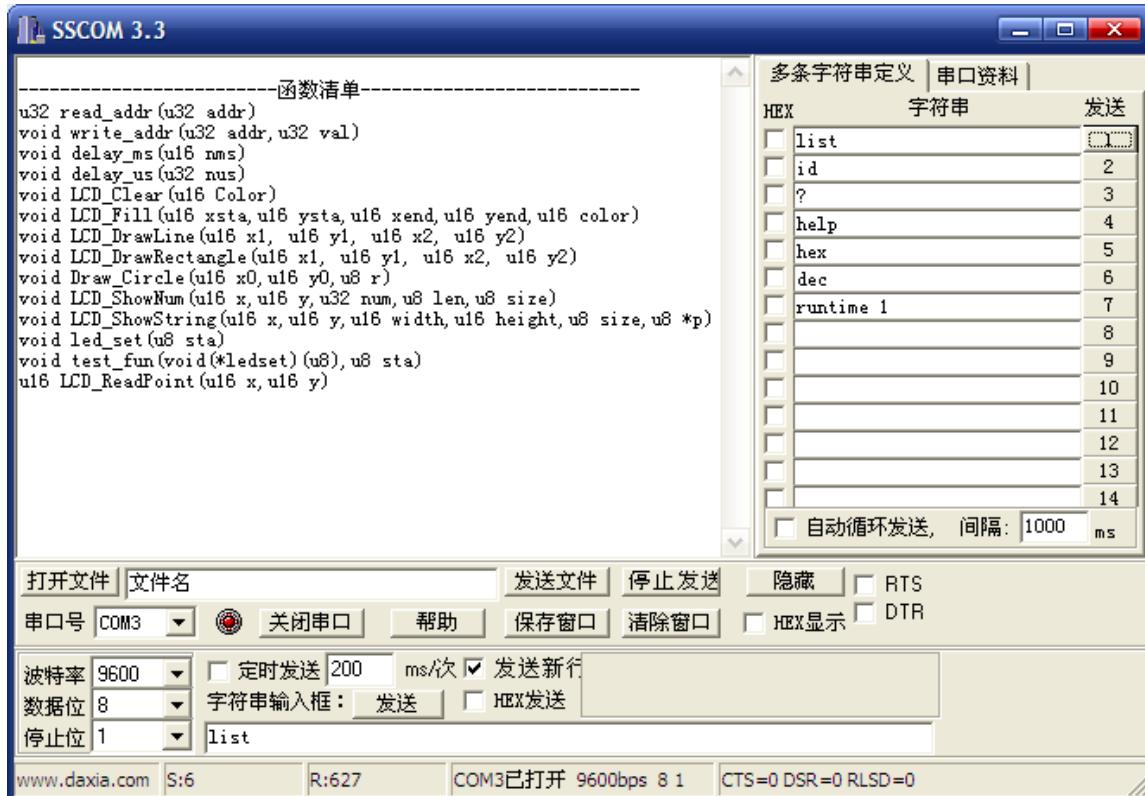


图 19.4.2 list 指令执行结果

id，该指令用于获取各个函数的入口地址。比如前面写的 test_fun 函数，就有一个函数参数，我们需要先通过 id 指令，获取 ledset 函数的 id（即入口地址），然后将这个 id 作为函数参数，传递给 test_fun。

? 和 help，这两个指令的功能是一样的。发送该指令后，串口将打印 usmart 使用的帮助信息。

hex 和 dec，这两个指令可以带参数，也可以不带参数。当不带参数的时候，hex 和 dec 分别用于设置串口显示数据格式为 16 进制/10 进制。当带参数的时候，hex 和 dec 就执行进制转换，比如输入：hex 1234，串口将打印：HEX:0X4D2，也就是将 1234 转换为 16 进制打印出来。又比如输入：dec 0X1234，串口将打印：DEC:4660，就是将 0X1234 转换为 10 进制打印出来。

runtime 指令，用于函数执行时间统计功能的开启和关闭，发送：runtime 1，可以开启函数执行时间统计功能；发送：runtime 0，可以关闭函数执行时间统计功能。函数执行时间统计功能，默认是关闭的。

大家可以亲自体验下这几个系统指令，不过要注意，所有的指令都是大小写敏感的，不要写错哦。

接下来，我们将介绍如何调用 list 所打印的这些函数，先来看一个简单的 delay_ms 的调用，我们分别输入 delay_ms(1000) 和 delay_ms(0x3E8)，如图 19.4.3 所示：

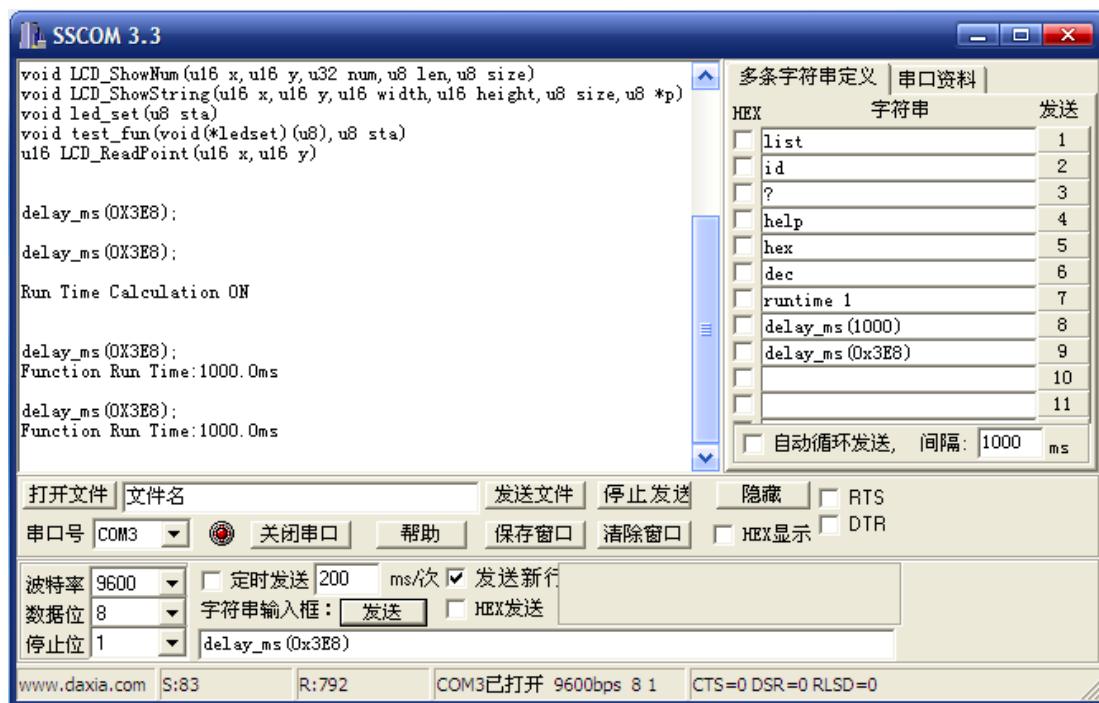


图 19.4.3 串口调用 delay_ms 函数

从上图可以看出, `delay_ms(1000)` 和 `delay_ms(0x3E8)` 的调用结果是一样的, 都是延时 1000ms, 因为 usmart 默认设置的是 hex 显示, 所以看到串口打印的参数都是 16 进制格式的, 大家可以通过发送 dec 指令切换为十进制显示。另外, 由于 USMART 对调用函数的参数大小写不敏感, 所以参数写成: `0X3E8` 或者 `0x3e8` 都是正确的。另外, 发送: `runtime 1`, 开启运行时间统计功能, 从测试结果看, USMART 的函数运行时间统计功能, 是相当准确的。

我们再看另外一个函数, `LCD_ShowString` 函数, 该函数用于显示字符串, 我们通过串口输入: `LCD_ShowString(20,200,200,100,16,"This is a test for usmart!!")`, 如图 19.4.4 所示:

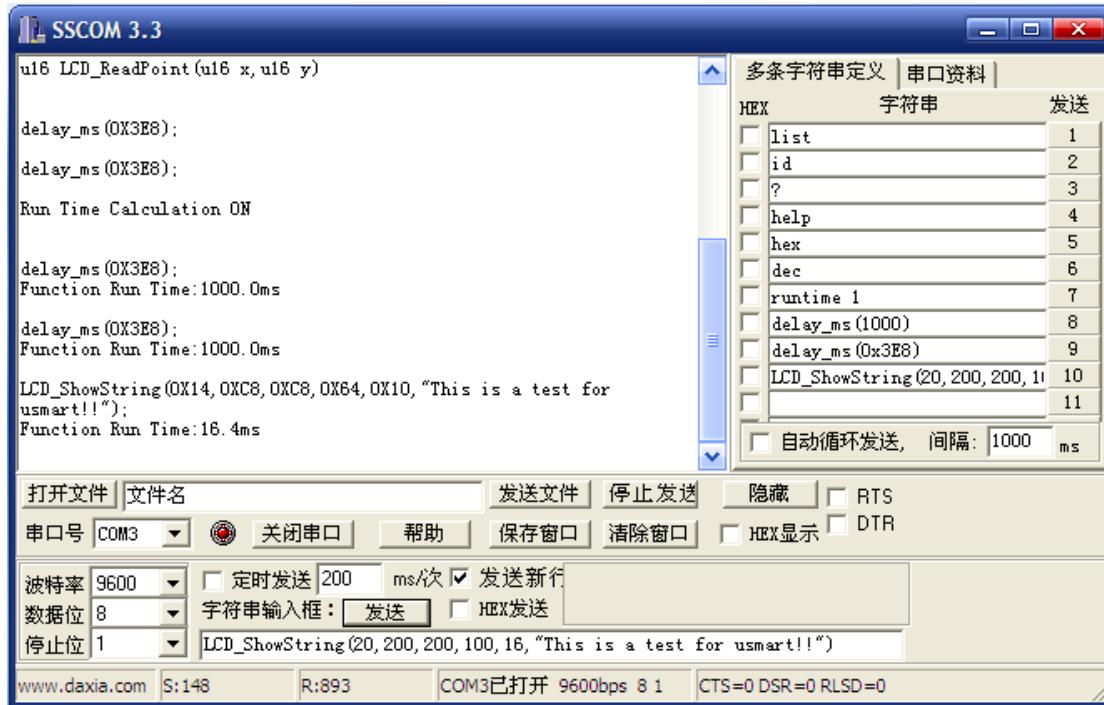


图 19.4.4 串口调用 LCD_ShowString 函数



该函数用于在指定区域，显示指定字符串，发送给开发板后，我们可以看到 LCD 在我们指定的地方显示了：This is a test for usmart!! 这个字符串。

其他函数的调用，也都是一样的方法，这里我们就不多介绍了，最后说一下带有函数参数的函数的调用。我们将 led_set 函数作为 test_fun 的参数，通过在 test_fun 里面调用 led_set 函数，实现对 DS1(LED1)的控制。前面说过，我们要调用带有函数参数的函数，就必须先得到函数参数的入口地址 (id)，通过输入 id 指令，我们可以得到 led_set 的函数入口地址是：0X0800022D，所以，我们在串口输入：test_fun(0X0800022D,0)，就可以控制 DS1 亮了。如图 19.4.5 所示：



图 19.4.5 串口调用 test_fun 函数

在开发板上，我们可以看到，收到串口发送的 test_fun(0X0800022D,0)后，开发板的 DS1 亮了，然后大家可以通过发送 test_fun(0X0800022D,1)，来关闭 DS1。说明我们成功的通过 test_fun 函数调用 led_set，实现了对 DS1 的控制。也就验证了 USMART 对函数参数的支持。

USMART 调试组件的使用，就为大家介绍到这里。USMART 是一个非常不错的调试组件，希望大家能学会使用，可以达到事半功倍的效果。

第二十章 RTC 实时时钟实验

前面我们介绍了两款液晶模块，这一章我们将介绍 STM32 的内部实时时钟（RTC）。在本章中，我们将利用 ALIENTEK 2.8 寸 TFTLCD 模块来显示日期和时间，实现一个简单的时钟。另外，本章将顺带向大家介绍 BKP 的使用。本章分为如下几个部分：

- 20.1 STM32 RTC 时钟简介
- 20.2 硬件设计
- 20.3 软件设计
- 20.4 下载验证



20.1 STM32 RTC 时钟简介

STM32 的实时时钟 (RTC) 是一个独立的定时器。STM32 的 RTC 模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。

RTC 模块和时钟配置系统(RCC_BDCR 寄存器)是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前，先要取消备份区域 (BKP) 写保护。

RTC 的简化框图，如图 20.1.1 所示：

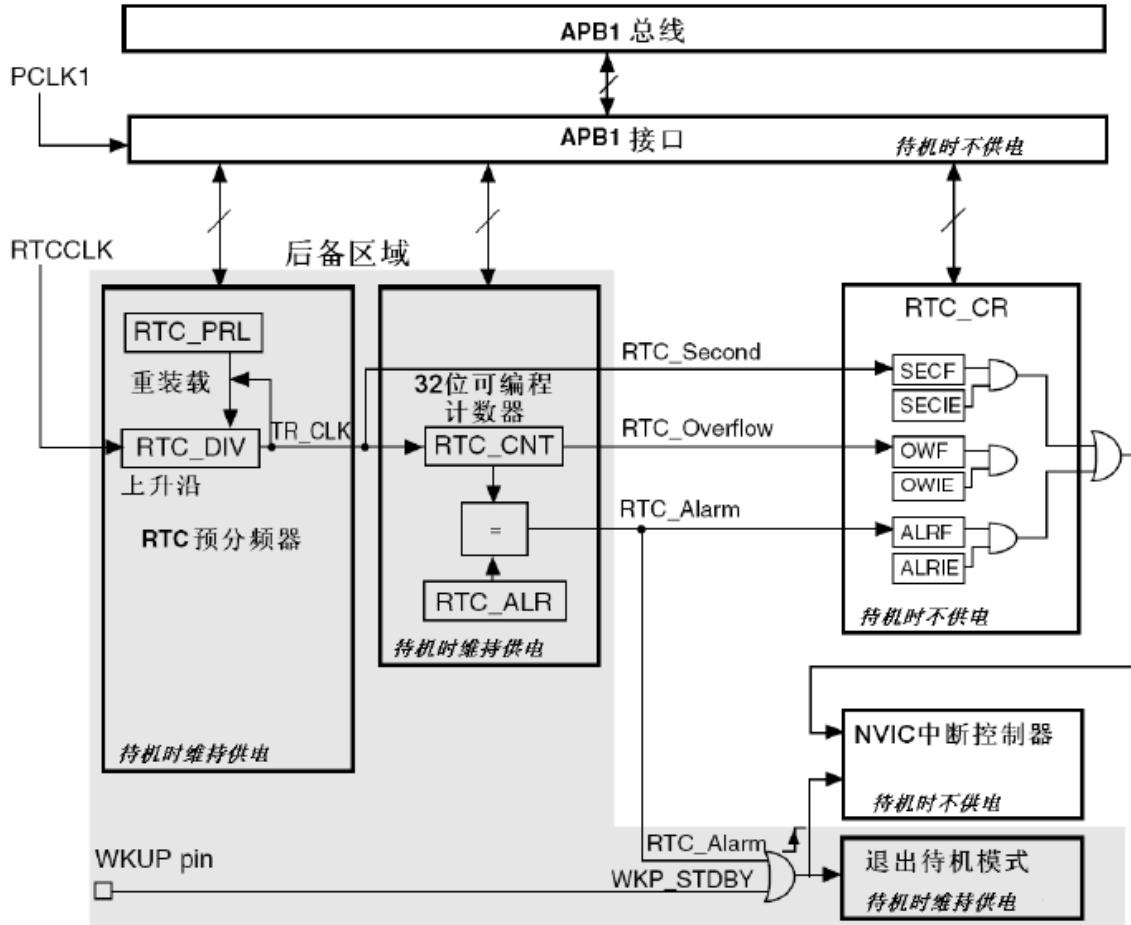


图 20.1.1 RTC 框图

RTC 由两个主要部分组成 (参见图 20.1.1)，第一部分(APB1 接口)用来和 APB1 总线相连。此单元还包含一组 16 位寄存器，可通过 APB1 总线对其进行读写操作。APB1 接口由 APB1 总线时钟驱动，用来与 APB1 总线连接。

另一部分(RTC 核心)由一组可编程计数器组成，分成两个主要模块。第一个模块是 RTC 的预分频模块，它可编程产生 1 秒的 RTC 时间基准 TR_CLK。RTC 的预分频模块包含了一个 20 位的可编程分频器(RTC 预分频器)。如果在 RTC_CR 寄存器中设置了相应的允许位，则在每个 TR_CLK 周期中 RTC 产生一个中断(秒中断)。第二个模块是一个 32 位的可编程计数器，可被初始化为当前的系统时间，一个 32 位的时钟计数器，按秒钟计算，可以记录 4294967296 秒，约合 136 年左右，作为一般应用，这已经是足够的了。



RTC 还有一个闹钟寄存器 RTC_ALR，用于产生闹钟。系统时间按 TR_CLK 周期累加并与存储在 RTC_ALR 寄存器中的可编程时间相比较，如果 RTC_CR 控制寄存器中设置了相应允许位，比较匹配时将产生一个闹钟中断。

RTC 内核完全独立于 RTC APB1 接口，而软件是通过 APB1 接口访问 RTC 的预分频值、计数器值和闹钟值的。但是相关可读寄存器只在 RTC APB1 时钟进行重新同步的 RTC 时钟的上升沿被更新，RTC 标志也是如此。这就意味着，如果 APB1 接口刚刚被开启之后，在第一次的内部寄存器更新之前，从 APB1 上读取的 RTC 寄存器值可能被破坏了（通常读到 0）。因此，若在读取 RTC 寄存器曾经被禁止的 RTC APB1 接口，软件首先必须等待 RTC_CRL 寄存器的 RSF 位（寄存器同步标志位，bit3）被硬件置 1。

要理解 RTC 原理，我们必须先通过对寄存器的讲解，让大家有一个全面的了解。接下来，我们介绍一下 RTC 相关的几个寄存器。首先要介绍的是 RTC 的控制寄存器，RTC 总共有 2 个控制寄存器 RTC_CRH 和 RTC_CRL，两个都是 16 位的。RTC_CRH 的各位描如图 20.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
保留														OWIE	ALRIE	SECIE
														RW	RW	RW
位15:3	保留，被硬件强制为0。															
位2	OWIE: 允许溢出中断位 0: 屏蔽(不允许)溢出中断 1: 允许溢出中断															
位1	ALRIE: 允许闹钟中断 0: 屏蔽(不允许)闹钟中断 1: 允许闹钟中断															
位0	SECIE: 允许秒中断 0: 屏蔽(不允许)秒中断 1: 允许秒中断															

图 20.1.2 RTC_CRH 寄存器各位描述

该寄存器用来控制中断的，我们本章将要用到秒钟中断，所以在该寄存器必须设置最低位为 1，以允许秒钟中断。我们再看看 RTC_CRL 寄存器。该寄存器各位描述如图 20.1.3 所示：



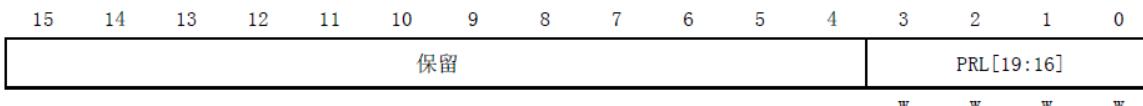
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								RTOFF	CNF	RSF	OWF	ALRF	SECF		
								r	rw	rc w0	rc w0	rc w0	rc w0		

位15:6	保留，被硬件强制为0。
位5	RTOFF: RTC操作关闭 RTC模块利用这位来指示对其寄存器进行的最后一次操作的状态，指示操作是否完成。若此位为0，则表示无法对任何的RTC寄存器进行写操作。此位为只读位。 0: 上一次对RTC寄存器的写操作仍在进行； 1: 上一次对RTC寄存器的写操作已经完成。
位4	CNF: 配置标志 此位必须由软件置'1'以进入配置模式，从而允许向RTC_CNT、RTC_ALR或RTC_PRL寄存器写入数据。只有当此位在被置'1'并重新由软件清'0'后，才会执行写操作。 0: 退出配置模式(开始更新RTC寄存器); 1: 进入配置模式。
位3	RSF: 寄存器同步标志 每当RTC_CNT寄存器和RTC_DIV寄存器由软件更新或清'0'时，此位由硬件置'1'。在APB1复位后，或APB1时钟停止后，此位必须由软件清'0'。要进行任何的读操作之前，用户程序必须等待这位被硬件置'1'，以确保RTC_CNT、RTC_ALR或RTC_PRL已经被同步。 0: 寄存器尚未被同步； 1: 寄存器已经被同步。
位2	OWF: 溢出标志 当32位可编程计数器溢出时，此位由硬件置'1'。如果RTC_CRH寄存器中OWIE=1，则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0: 无溢出； 1: 32位可编程计数器溢出。
位1	ALRF: 闹钟标志 当32位可编程计数器达到RTC_ALR寄存器所设置的预定值，此位由硬件置'1'。如果RTC_CRH寄存器中ALRIE=1，则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0: 无闹钟； 1: 有闹钟。
位0	SECF: 秒标志 当32位可编程预分频器溢出时，此位由硬件置'1'同时RTC计数器加1。因此，此标志为分辨率可编程的RTC计数器提供一个周期性的信号(通常为1秒)。如果RTC_CRH寄存器中SECIE=1，则产生中断。此位只能由软件清除。对此位写'1'是无效的。 0: 秒标志条件不成立； 1: 秒标志条件成立。

图 20.1.3 RTC_CRL 寄存器各位描述

本章我们用到的是该寄存器的 0、3~5 这几个位，第 0 位是秒钟标志位，我们在进入闹钟中断的时候，通过判断这位来决定是不是发生了秒钟中断。然后必须通过软件将该位清零（写 0）。第 3 位为寄存器同步标志位，我们在修改控制寄存器 RTC_CRH/CRL 之前，必须先判断该位，是否已经同步了，如果没有则等待同步，在没同步的情况下修改 RTC_CRH/CRL 的值是不行的。第 4 位为配置标位，在软件修改 RTC_CNT/RTC_ALR/RTC_PRL 的值的时候，必须先软件置位该位，以允许进入配置模式。第 5 位为 RTC 操作位，该位由硬件操作，软件只读。通过该位可以判断上次对 RTC 寄存器的操作是否完成，如果没有，我们必须等待上一次操作结束才能开始下一次操作。

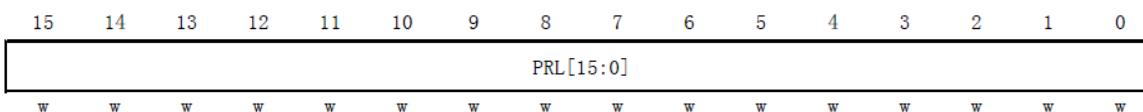
第二个要介绍的寄存器是 RTC 预分频装载寄存器，也有 2 个寄存器组成，RTC_PRLH 和 RTC_PRLL。这两个寄存器用来配置 RTC 时钟的分频数的，比如我们使用外部 32.768K 的晶振作为时钟的输入频率，那么我们要设置这两个寄存器的值为 32767，以得到一秒钟的计数频率。RTC_PRLH 的各位描述如图 20.1.4 所示：



位15:6	保留，被硬件强制为0。
位3:0	PRL[19:16]: RTC预分频装载值高位 根据以下公式，这些位用来定义计数器的时钟频率： $f_{TR_CLK} = f_{RTCCLK}/(PRL[19:0]+1)$ 注：不推荐使用0值，否则无法正确的产生RTC中断和标志位。

图 20.1.4 RTC_PRLH 寄存器各位描述

从图 20.1.4 可以看出，RTC_PRLH 只有低四位有效，用来存储 PRL 的 19~16 位。而 PRL 的前 16 位，存放在 RTC_PRLL 里面，寄存器 RTC_PRLL 的各位描述如图 20.1.5 所示：



位15:0	PRL[15:0]: RTC预分频装载值低位 根据以下公式，这些位用来定义计数器的时钟频率： $f_{TR_CLK} = f_{RTCCLK}/(PRL[19:0]+1)$
-------	--

图 20.1.5 RTC_PRLL 寄存器各位描述

在介绍完这两个寄存器之后，我们介绍 RTC 预分频器余数寄存器，该寄存器也有 2 个寄存器组成 RTC_DIVH 和 RTC_DIVL，这两个寄存器的作用就是用来获得比秒钟更为准确的时钟，比如可以得到 0.1 秒，或者 0.01 秒等。该寄存器的值自减的，用于保存还需要多少时钟周期获得一个秒信号。在一次秒钟更新后，由硬件重新装载。这两个寄存器和 RTC 预分频装载寄存器的各位是一样的，这里我们就不列出来了。

接着要介绍的是 RTC 最重要的寄存器，RTC 计数器寄存器 RTC_CNT。该寄存器由 2 个 16 位的寄存器组成 RTC_CNTH 和 RTC_CNTL，总共 32 位，用来记录秒钟值（一般情况下）。此两个计数器也比较简单，我们也不多说了。注意一点，在修改这个寄存器的时候要先进入配置模式。

最后我们介绍 RTC 部分的最后一个寄存器，RTC 闹钟寄存器，该寄存器也是由 2 个 16 位的寄存器组成 RTC_ALRH 和 RTC_ALRL。总共也是 32 位，用来标记闹钟产生的时钟（以秒为单位），如果 RTC_CNT 的值与 RTC_ALR 的值相等，并使能了中断的话，会产生一个闹钟中断。该寄存器的修改也要进入配置模式才能进行。

因为我们使用到备份寄存器来存储 RTC 的相关信息（我们这里主要用来标记时钟是否已经经过了配置），我们这里顺便介绍一下 STM32 的备份寄存器。

备份寄存器是 42 个 16 位的寄存器（战舰开发板就是大容量的），可用来存储 84 个字节的用户应用程序数据。他们处在备份域里，当 VDD 电源被切断，他们仍然由 VBAT 维持供电。即使系统在待机模式下被唤醒，或系统复位或电源复位时，他们也不会被复位。

此外，BKP 控制寄存器用来管理侵入检测和 RTC 校准功能，这里我们不作介绍。

复位后，对备份寄存器和 RTC 的访问被禁止，并且备份域被保护以防止可能存在的意外的操作。执行以下操作可以使能对备份寄存器和 RTC 的访问：

1) 通过设置寄存器 RCC_APB1ENR 的 PWREN 和 BKOPEN 位来打开电源和后备接口的时钟

2) 电源控制寄存器(PWR_CR)的 DBP 位来使能对后备寄存器和 RTC 的访问。

我们一般用 BKP 来存储 RTC 的校验值或者记录一些重要的数据，相当于一个 EEPROM，



不过这个 EEPROM 并不是真正的 EEPROM，而是需要电池来维持它的数据。关于 BKP 的详细介绍请看《STM32 参考手册》的第 47 页，5.1 一节。

最后，我们还要介绍一下备份区域控制寄存器 RCC_BDCR。该寄存器的个位描述如图 20.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留														BDRST	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTC EN	保留			RTCSEL[1:0]	保留			LSE BYP	LSE RDY	LSEON					
rw				rw	rw			rw	r	rw					
位31:17	保留，始终读为0。														
位16	BDRST : 备份域软件复位 由软件置'1'或清'0' 0: 复位未激活； 1: 复位整个备份域。														
位15	RTCEN : RTC时钟使能 由软件置'1'或清'0' 0: RTC时钟关闭； 1: RTC时钟开启。														
位14:10	保留，始终读为0。														
位9:8	RTCSEL[1:0] : RTC时钟源选择 由软件设置来选择RTC时钟源。一旦RTC时钟源被选定，直到下次后备域被复位，它不能在被改变。可通过设置 BDRST 位来清除。 00: 无时钟； 01: LSE振荡器作为RTC时钟； 10: LSI振荡器作为RTC时钟； 11: HSE振荡器在128分频后作为RTC时钟。														
位7:3	保留，始终读为0。														
位2	LSEBYP : 外部低速时钟振荡器旁路 在调试模式下由软件置'1'或清'0'来旁路LSE。只有在外部32kHz振荡器关闭时，才能写入该位 0: LSE时钟未被旁路； 1: LSE时钟被旁路。														
位1	LSERDY : 外部低速LSE就绪 由硬件置'1'或清'0'来指示是否外部32kHz振荡器就绪。在LSEON被清零后，该位需要6个外部低速振荡器的周期才被清零。 0: 外部32kHz振荡器未就绪； 1: 外部32kHz振荡器就绪。														
位0	LSEON : 外部低速振荡器使能 由软件置'1'或清'0' 0: 外部32kHz振荡器关闭； 1: 外部32kHz振荡器开启。														

图 20.1.6 RCC_BDCR 寄存器各位描述

RTC 的时钟源选择及使能设置都是通过这个寄存器来实现的，所以我们在 RTC 操作之前先要通过这个寄存器选择 RTC 的时钟源，然后才能开始其他的操作。

寄存器介绍就给大家介绍到这里了，我们下面来看看要经过哪几个步骤的配置才能使 RTC 正常工作，这里我们将对每个步骤通过库函数的实现方式来讲解。

RTC 相关的库函数在文件 `stm32f10x_rtc.c` 和 `stm32f10x_rtc.h` 文件中，BKP 相关的库函数在文件 `stm32f10x_bkp.c` 和文件 `stm32f10x_bkp.h` 文件中。



RTC 正常工作的一般配置步骤如下：

1) 使能电源时钟和备份区域时钟。

前面已经介绍了，我们要访问 RTC 和备份区域就必须先使能电源时钟和备份区域时钟。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);
```

2) 取消备份区写保护。

要向备份区域写入数据，就要先取消备份区域写保护（写保护在每次硬复位之后被使能），否则是无法向备份区域写入数据的。我们需要用到向备份区域写入一个字节，来标记时钟已经配置过了，这样避免每次复位之后重新配置时钟。取消备份区域写保护的库函数实现方法是：

```
PWR_BackupAccessCmd(ENABLE); //使能 RTC 和后备寄存器访问
```

3) 复位备份区域，开启外部低速振荡器。

在取消备份区域写保护之后，我们可以先对这个区域复位，以清除前面的设置，当然这个操作不要每次都执行，因为备份区域的复位将导致之前存在的数据丢失，所以要不要复位，要看情况而定。然后我们使能外部低速振荡器，注意这里一般要先判断 RCC_BDCR 的 LSERDY 位来确定低速振荡器已经就绪了才开始下面的操作。

备份区域复位的函数是：

```
BKP_DeInit();//复位备份区域
```

开启外部低速振荡器的函数是：

```
RCC_LSEConfig(RCC_LSE_ON); // 开启外部低速振荡器
```

4) 选择 RTC 时钟，并使能。

这里我们将通过 RCC_BDCR 的 RTCSEL 来选择选择外部 LSI 作为 RTC 的时钟。然后通过 RTCEN 位使能 RTC 时钟。

库函数中，选择 RTC 时钟的函数是：

```
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择 LSE 作为 RTC 时钟
```

对于 RTC 时钟的选择，还有 RCC_RTCCLKSource_LSI 和 RCC_RTCCLKSource_HSE_Div128 两个，顾名思义，前者为 LSI，后者为 HSE 的 128 分频，这在时钟系统章节有讲解过。

使能 RTC 时钟的函数是：

```
RCC_RTCCLKCmd(ENABLE); //使能 RTC 时钟
```

5) 设置 RTC 的分频，以及配置 RTC 时钟。

在开启了 RTC 时钟之后，我们要做的就是设置 RTC 时钟的分频数，通过 RTC_PRLH 和 RTC_PRLL 来设置，然后等待 RTC 寄存器操作完成，并同步之后，设置秒钟中断。然后设置 RTC 的允许配置位(RTC_CRH 的 CNF 位)，设置时间(其实就是设置 RTC_CNTH 和 RTC_CNTL 两个寄存器)。下面我们一一这些步骤用到的库函数：

在进行 RTC 配置之前首先要打开允许配置位(CNF)，库函数是：

```
RTC_EnterConfigMode(); // 允许配置
```

在配置完成之后，千万别忘记更新配置同时退出配置模式，函数是：

```
RTC_ExitConfigMode(); //退出配置模式，更新配置
```

设置 RTC 时钟分频数，库函数是：

```
void RTC_SetPrescaler(uint32_t PrescalerValue);
```

这个函数只有一个入口参数，就是 RTC 时钟的分频数，很好理解。

然后是设置秒中断允许，RTC 使能中断的函数是：

```
void RTC_ITConfig(uint16_t RTC_IT, FunctionalState NewState);
```

这个函数的第一个参数是设置秒中断类型，这些通过宏定义定义的。对于使能秒中断方法是：

```
RTC_ITConfig(RTC_IT_SEC, ENABLE); //使能 RTC 秒中断
```



下一步便是设置时间了，设置时间实际上就是设置 RTC 的计数值，时间与计数值之间是需要换算的。库函数中设置 RTC 计数值的方法是：

```
void RTC_SetCounter(uint32_t CounterValue)最后在配置完成之后
```

通过这个函数直接设置 RTC 计数值。

6) 更新配置，设置 RTC 中断分组。

在设置完时钟之后，我们将配置更新同时退出配置模式，这里还是通过 RTC_CRH 的 CNF 来实现。库函数的方法是：

```
RTC_ExitConfigMode();//退出配置模式，更新配置
```

在退出配置模式更新配置之后我们在备份区域 BKP_DR1 中写入 0X5050 代表我们已经初始化过时钟了，下次开机（或复位）的时候，先读取 BKP_DR1 的值，然后判断是否是 0X5050 来决定是不是要配置。接着我们配置 RTC 的秒钟中断，并进行分组。

往备份区域写用户数据的函数是：

```
void BKP_WriteBackupRegister(uint16_t BKP_DR, uint16_t Data);
```

这个函数的第一个参数就是寄存器的标号了，这个是通过宏定义定义的。比如我们要往 BKP_DR1 写入 0x5050，方法是：

```
BKP_WriteBackupRegister(BKP_DR1, 0X5050);
```

同时，有写便有读，读取备份区域指定寄存器的用户数据的函数是：

```
uint16_t BKP_ReadBackupRegister(uint16_t BKP_DR);
```

这个函数就很好理解了，这里不做过多讲解。

设置中断分组的方法之前已经详细讲解过，调用 NVIC_Init 函数即可，这里不做重复讲解。

7) 编写中断服务函数。

最后，我们要编写中断服务函数，在秒钟中断产生的时候，读取当前的时间值，并显示到 TFTLCD 模块上。

通过以上几个步骤，我们就完成了对 RTC 的配置，并通过秒钟中断来更新时间。接下来我们将进行下一步的工作。

20.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 串口
- 3) TFTLCD 模块
- 4) RTC

前面 3 个都介绍过了，而 RTC 属于 STM32 内部资源，其配置也是通过软件设置好就可以了。不过 RTC 不能断电，否则数据就丢失了，我们如果想让时间在断电后还可以继续走，那么必须确保开发板的电池有电（ALIENTEK 战舰 STM32 开发板标配是有电池的）。

20.3 软件设计

同样，打开我们光盘的 RTC 时钟实验，可以看到，我们的工程中加入了 rtc.c 源文件和 rtc.h 头文件，同时，引入了 stm32f10x_rtc.c 和 stm32f10x_bkp.c 库文件。

由于篇幅所限，rtc.c 中的代码，我们不全部贴出了，这里针对几个重要的函数，进行简要说明，首先是 RTC_Init，其代码如下：

```
//实时时钟配置
```



```
//初始化 RTC 时钟,同时检测时钟是否工作正常
//BKP->DR1 用于保存是否第一次配置的设置
//返回 0:正常
//其他:错误代码
u8 RTC_Init(void)
{
    u8 temp=0;
    //检查是不是第一次配置时钟
    if (BKP_ReadBackupRegister(BKP_DR1) != 0x5050)      //从指定的后备寄存器中
        //读出数据:读出了与写入的指定数据不相乎
    {
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR |
            RCC_APB1Periph_BKP, ENABLE);           //使能 PWR 和 BKP 外设时钟
        PWR_BackupAccessCmd(ENABLE);             //使能后备寄存器访问
        BKP_DeInit();                          //③复位备份区域
        RCC_LSEConfig(RCC_LSE_ON);              //设置外部低速晶振(LSE)
        while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET&&temp<250)
            //检查指定的 RCC 标志位设置与否,等待低速晶振就绪
            {
                temp++;
                delay_ms(10);
            }
        if(temp>=250) return 1;//初始化时钟失败,晶振有问题
        RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);    //设置 RTC 时钟
            //RTCCLK),选择 LSE 作为 RTC 时钟
        RCC_RTCCLKCmd(ENABLE);                   //使能 RTC 时钟
        RTC_WaitForLastTask();                  //等待最近一次对 RTC 寄存器的写操作完成
        RTC_WaitForSynchro();                  //等待 RTC 寄存器同步
        RTC_ITConfig(RTC_IT_SEC, ENABLE);       //使能 RTC 秒中断
        RTC_WaitForLastTask();                  //等待最近一次对 RTC 寄存器的写操作完成
        RTC_EnterConfigMode();                // 允许配置
        RTC_SetPrescaler(32767);               //设置 RTC 预分频的值
        RTC_WaitForLastTask();                  //等待最近一次对 RTC 寄存器的写操作完成
        RTC_Set(2009,12,2,10,0,55);           //设置时间
        RTC_ExitConfigMode();                 //退出配置模式
        BKP_WriteBackupRegister(BKP_DR1, 0X5050); //向指定的后备寄存器中
            //写入用户程序数据 0x5050
    }
    else//系统继续计时
    {
        RTC_WaitForSynchro();                //等待最近一次对 RTC 寄存器的写操作完成
        RTC_ITConfig(RTC_IT_SEC, ENABLE);   //使能 RTC 秒中断
        RTC_WaitForLastTask();              //等待最近一次对 RTC 寄存器的写操作完成
    }
}
```



```
    }
    RTC_NVIC_Config();           //RCT 中断分组设置
    RTC_Get();                  //更新时间
    return 0;                   //ok
}
```

该函数用来初始化 RTC 时钟，但是只在第一次的时候设置时间，以后如果重新上电/复位都不会再进行时间设置了（前提是备份电池有电），在第一次配置的时候，我们是按照上面介绍的 RTC 初始化步骤来做的，这里就不在多说了，这里我们设置时间是通过时间设置函数 RTC_Set(2012,9,7,13,16,55);来实现的，这里我们默认将时间设置为 2012 年 9 月 7 日 13 点 16 分 55 秒。在设置好时间之后，我们通过 BKP_WriteBackupRegister() 函数向 BKP->DR1 写入标志字 0X5050，用于标记时间已经被设置了。这样，再次发生复位的时候，该函数通过 BKP_ReadBackupRegister() 读取 BKP->DR1 的值，来判断决定是不是需要重新设置时间，如果不设置，则跳过时间设置，仅仅使能秒钟中断一下，就进行中断分组，然后返回了。这样不会重复设置时间，使得我们设置的时间不会因复位或者断电而丢失。

该函数还有返回值，返回值代表此次操作的成功与否，如果返回 0，则代表初始化 RTC 成功，如果返回值非零则代表错误代码了。

介绍完 RTC_Init，我们来介绍一下 RTC_Set 函数，该函数代码如下：

```
//设置时钟
//把输入的时钟转换为秒钟
//以 1970 年 1 月 1 日为基准
//1970~2099 年为合法年份
//返回值:0,成功;其他:错误代码.
//月份数据表
u8 const table_week[12]={0,3,3,6,1,4,6,2,5,0,3,5}; //月修正数据表
//平年的月份日期表
const u8 mon_table[12]={31,28,31,30,31,30,31,31,30,31,30,31};
u8 RTC_Set(u16 syear,u8 smon,u8 sday,u8 hour,u8 min,u8 sec)
{
    u16 t;
    u32 seccount=0;
    if(syear<1970||syear>2099)return 1;
    for(t=1970;t<syear;t++)           //把所有年份的秒钟相加
    {
        if(Is_Leap_Year(t))seccount+=31622400;//闰年的秒钟数
        else seccount+=31536000;             //平年的秒钟数
    }
    smon-=1;
    for(t=0;t<smon;t++)               //把前面月份的秒钟数相加
    {
        seccount+=(u32)mon_table[t]*86400; //月份秒钟数相加
        if(Is_Leap_Year(syear)&&t==1)seccount+=86400;//闰年 2 月份增加一天的秒钟数
    }
    seccount+=(u32)(sday-1)*86400;      //把前面日期的秒钟数相加
    seccount+=(u32)hour*3600;           //小时秒钟数
```



```

seccount+=(u32)min*60;           //分钟秒钟数
seccount+=sec;                  //最后的秒钟加上去
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR |
RCC_APB1Periph_BKP, ENABLE);    //使能 PWR 和 BKP 外设时钟
PWR_BackupAccessCmd(ENABLE);    //使能 RTC 和后备寄存器访问
RTC_SetCounter(seccount);       //设置 RTC 计数器的值
RTC_WaitForLastTask();          //等待最近一次对 RTC 寄存器的写操作完成
return 0;
}

```

该函数用于设置时间，把我们输入的时间，转换为以 1970 年 1 月 1 日 0 时 0 分 0 秒当做起始时间的秒钟信号，后续的计算都以这个时间为基准的，由于 STM32 的秒钟计数器可以保存 136 年的秒钟数据，这样我们可以计时到 2106 年。

接着，我们介绍一下 RTC_Get 函数，该函数用于获取时间和日期等数据，其代码如下：

```

//得到当前的时间，结果保存在 calendar 结构体里面
//返回值:0,成功;其他:错误代码.
u8 RTC_Get(void)
{
    static u16 daycnt=0;
    u32 timecount=0;
    u32 temp=0;
    u16 temp1=0;
    timecount=RTC->CNTH;           //得到计数器中的值(秒钟数)
    timecount<<=16;
    timecount+=RTC->CNTL;
    temp=timecount/86400;           //得到天数(秒钟数对应的)
    if(daycnt!=temp)               //超过一天了
    {
        daycnt=temp;
        temp1=1970;                 //从 1970 年开始
        while(temp>=365)
        {
            if(Is_Leap_Year(temp1)) //是闰年
            {
                if(temp>=366)temp-=366; //闰年的秒钟数
                else break;
            }
            else temp-=365;         //平年
            temp1++;
        }
        calendar.w_year=temp1;      //得到年份
        temp1=0;
        while(temp>=28)           //超过了一个月
        {
            if(Is_Leap_Year(calendar.w_year)&&temp1==1)//当年是不是闰年/2 月份

```



```
{  
    if(temp>=29)temp-=29;//闰年的秒钟数  
    else break;  
}  
else  
{    if(temp>=mon_table[temp1])temp-=mon_table[temp1];//平年  
    else break;  
}  
temp1++;  
}  
calendar.w_month=temp1+1;//得到月份  
calendar.w_date=temp+1; //得到日期  
}  
temp=timecount%86400; //得到秒钟数  
calendar.hour=temp/3600; //小时  
calendar.min=(temp%3600)/60; //分钟  
calendar.sec=(temp%3600)%60; //秒钟  
calendar.week=RTC_Get_Week(calendar.w_year,calendar.w_month,calendar.w_date);  
//获取星期  
return 0;  
}
```

函数其实就是在存储在秒寄存器 RTC->CNTH 和 RTC->CNTL 中的秒数据(通过函数 RTC_SetCounter 设置)转换为真正的时间和日期。该代码还用到了一个 calendar 的结构体, calendar 是我们在 rtc.h 里面将要定义的一个时间结构体, 用来存放时钟的年月日时分秒等信息。因为 STM32 的 RTC 只有秒计数器, 而年月日, 时分秒这些需要我们自己软件计算。我们把计算好的值保存在 calendar 里面, 方便其他程序调用。

最后, 我们介绍一下秒中断服务函数, 该函数代码如下:

```
//RTC 时钟中断  
//每秒触发一次  
void RTC_IRQHandler(void)  
{  
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET) //秒钟中断  
    {  
        RTC_Get(); //更新时间  
    }  
    if(RTC_GetITStatus(RTC_IT_ALR)!= RESET) //闹钟中断  
    {  
        RTC_ClearITPendingBit(RTC_IT_ALR); //清闹钟中断  
    }  
    RTC_ClearITPendingBit(RTC_IT_SEC|RTC_IT_OW); //清闹钟中断  
    RTC_WaitForLastTask();  
}
```

此部分代码比较简单, 我们通过 RTC_GetITStatus 来判断发生的是何种中断, 如果是秒钟



中断，则执行一次时间的计算，获得最新时间。从而，我们可以在 calendar 里面读到时间、日期等信息。

rtc.c 的其他程序，这里就不再介绍了，请大家直接看光盘的源码。接下来看看 rtc.h 代码，在 rtc.h 中，我们定义了一个结构体：

```
typedef struct
{
    vu8 hour;
    vu8 min;
    vu8 sec;
    //公历日月年周
    vu16 w_year;
    vu8 w_month;
    vu8 w_date;
    vu8 week;
} _calendar_obj;
```

从上面结构体定义可以看到 _calendar_obj 结构体所包含的成员变量是一个完整的公历信息，包括年、月、日、周、时、分、秒等 7 个元素。我们以后要知道当前时间，只需要通过 RTC_Get 函数，执行时钟转换，然后就可以从 calendar 里面读出当前的公历时间了。

最后看看 main.c 里面的代码如下：

```
int main(void)
{
    u8 t=0;
    delay_init();                      //延时函数初始化
    NVIC_Configuration();              //设置 NVIC 中断分组 2
    uart_init(9600);                  //串口初始化波特率为 9600
    LED_Init();                        //LED 端口初始化
    LCD_Init();                        //LCD 初始化
    usmart_dev.init(72);               //初始化 USMART
    POINT_COLOR=RED;                  //设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"RTC TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/7");
    while(RTC_Init())                //RTC 初始化，一定要初始化成功
    {
        LCD_ShowString(60,130,200,16,16,"RTC ERROR!      ");
        delay_ms(800);
        LCD_ShowString(60,130,200,16,16,"RTC Trying...");
    }
    //显示时间
    POINT_COLOR=BLUE;                 //设置字体为蓝色
    LCD_ShowString(60,130,200,16,16,"      - -      ");
    LCD_ShowString(60,162,200,16,16,"      : :      ");
    while(1)
```



```
{  
    if(t!=calendar.sec)  
    {  
        t=calendar.sec;  
        LCD_ShowNum(60,130,calendar.w_year,4,16);  
  
        LCD_ShowNum(100,130,calendar.w_month,2,16);  
  
        LCD_ShowNum(124,130,calendar.w_date,2,16);  
        switch(calendar.week)  
        {  
            case 0:LCD_ShowString(60,148,200,16,16,"Sunday      ");  
                break;  
            case 1:LCD_ShowString(60,148,200,16,16,"Monday      ");  
                break;  
            case 2:LCD_ShowString(60,148,200,16,16,"Tuesday     ");  
                break;  
            case 3:LCD_ShowString(60,148,200,16,16,"Wednesday");  
                break;  
            case 4:LCD_ShowString(60,148,200,16,16,"Thursday   ");  
                break;  
            case 5:LCD_ShowString(60,148,200,16,16,"Friday     ");  
                break;  
            case 6:LCD_ShowString(60,148,200,16,16,"Saturday   ");  
                break;  
        }  
        LCD_ShowNum(60,162,calendar.hour,2,16);  
        LCD_ShowNum(84,162,calendar.min,2,16);  
        LCD_ShowNum(108,162,calendar.sec,2,16);  
        LED0=!LED0;  
    }  
    delay_ms(10);  
};  
}
```

这部分代码就不再需要详细解释了，在包含了 `rtc.h` 之后，通过判断 `calendar.sec` 是否改变来决定要不要更新时间显示。同时我们设置 `LED0` 每 2 秒钟闪烁一次，用来提示程序已经开始跑了。

为了方便设置时间，我们在 `usmart_config.c` 里面，修改 `usmart_nametab` 如下：

```
struct _m_usmart_nametab usmart_nametab[]=  
{  
#if USMART_USE_WRFUNS==1           //如果使能了读写操作  
    (void*)read_addr,"u32 read_addr(u32 addr)",  
    (void*)write_addr,"void write_addr(u32 addr,u32 val)",  
};
```



```
#endif  
(void*)delay_ms,"void delay_ms(u16 nms)",  
(void*)delay_us,"void delay_us(u32 nus)",  
(void*)RTC_Set,"u8 RTC_Set(u16 syear,u8 smon,u8 sday,u8 hour,u8 min,u8 sec)",  
};
```

将 RTC_Set 加入了 usmart，同时去掉了上一章的设置（减少代码量），这样通过串口就可以直接设置 RTC 时间了。

至此，RTC 实时时钟的软件设计就完成了，接下来就让我们来检验一下，我们的程序是否正确了。

20.4 下载验证

将程序下载到战舰 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块开始显示时间，实际显示效果如图 20.4.1 所示：



图 20.4.1 RTC 实验效果图

如果时间不正确，大家可以用上一章介绍的方法，通过串口调用 RTC_Set 来设置一下当前时间。



第二十一章 待机唤醒实验

本章我们将向大家介绍 STM32 的待机唤醒功能。在本章中，我们将利用 WK_UP 按键来实现唤醒和进入待机模式的功能，然后利用 DS0 指示状态。本章将分为如下几个部分：

- 21.1 STM32 待机模式简介
- 21.2 硬件设计
- 21.3 软件设计
- 21.4 下载验证



21.1 STM32 待机模式简介

很多单片机都有低功耗模式，STM32也不例外。在系统或电源复位以后，微控制器处于运行状态。运行状态下的 HCLK 为 CPU 提供时钟，内核执行程序代码。当 CPU 不需继续运行时，可以利用多个低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗，最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。STM32 的 3 种低功耗模式我们在 5.2.4 节有粗略介绍，这里我们再回顾一下。

STM32 的低功耗模式有 3 种：

- 1) 睡眠模式（CM3 内核停止，外设仍然运行）
- 2) 停止模式（所有时钟都停止）
- 3) 待机模式（1.8V 内核电源关闭）

在运行模式下，我们也可以通过降低系统时钟关闭 APB 和 AHB 总线上未被使用的外设的时钟来降低功耗。三种低功耗模式一览表见表 21.1.1 所示：

模式	进入操作	唤醒	对1.8V区域时钟的影响	对VDD区域时钟的影响	电压调节器
睡眠 (SLEEP-NOW或 SLEEP-ON-EXIT)	WFI	任一中断	CPU 时钟关， 对其他时钟和 ADC 时钟无影 响	无	开
	WFE	唤醒事件			
停机	PDDS 和 LPDS 位 +SLEEPDEEP 位 +WFI 或 WFE	任一外部中断(在外 部中断寄存器中设 置)	所有使用 1.8V 的区域的时钟 都已关闭，HSI 和 HSE 的振荡 器关闭	在低功耗模式下可 进行开/关设置(依 据电源控制寄存器 (PWR_CR)的设定)	关
待机	PDDS 位 +SLEEPDEEP 位 +WFI 或 WFE	WKUP 引脚的上升 沿、RTC 警告事 件、NRST 引脚上的 外部复位、IWDG 复 位			

表 21.1.1 STM32 低功耗一览表

在这三种低功耗模式中，最低功耗的是待机模式，在此模式下，最低只需要 2uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 20uA 左右。最后就是睡眠模式了。用户可以根据自己的需求来决定使用哪种低功耗模式。

本章，我们仅对 STM32 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32 的最低功耗。该模式是在 CM3 深睡眠模式时关闭电压调节器。整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失。仅备份的寄存器和待机电路维持供电。那么我们如何进入待机模式呢？其实很简单，只要按图 21.1.1 所示的步骤执行就可以了：

待机模式	说明
进入	在以下条件下执行 WFI 或 WFE 指令： - 设置 Cortex™-M3 系统控制寄存器中的 SLEEPDEEP 位 - 设置电源控制寄存器 (PWR_CR) 中的 PDDS 位 - 清除电源控制/状态寄存器 (PWR_CSR) 中的 WUF 位
退出	WKUP 引脚的上升沿、RTC 阔钟、NRST 引脚上外部复位、IWDG 复位。
唤醒延时	复位阶段时电压调节器的启动。

图 21.1.1 STM32 进入及退出待机模式的条件

图 21.1.1 还列出了退出待机模式的操作，从图 21.1.1 可知，我们有 4 种方式可以退出待机模式，即当一个外部复位(NRST 引脚)、IWDG 复位、WKUP 引脚上的上升沿或 RTC 阔钟事件

发生时，微控制器从待机模式退出。从待机唤醒后，除了电源控制/状态寄存器(PWR_CSR)，所有寄存器被复位。

从待机模式唤醒后的代码执行等同于复位后的执行(采样启动模式引脚, 读取复位向量等)。电源控制/状态寄存器(PWR_CSR)将会指示内核由待机状态退出。

在进入待机模式后，除了复位引脚以及被设置为防入侵或校准输出时的 TAMPER 引脚和被使能的唤醒引脚（WK_UP 脚），其他的 IO 引脚都将处于高阻态。

图 21.1.1 已经清楚的说明了进入待机模式的通用步骤，其中涉及到 2 个寄存器，即电源控制寄存器 (PWR_CR) 和电源控制/状态寄存器 (PWR_CSR)。下面我们介绍一下这两个寄存器：

电源控制寄存器（PWR CR），该寄存器的各位描述如图 21.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS	
rw				rw		rw		rw		rc	wl	rc	wl	rw	

位 31:9	保留。始终读为0。								
位 8	DBP: 取消后备区域的写保护 在复位后，RTC和后备寄存器处于被保护状态以防意外写入。设置这位允许写入这些寄存器。 0: 禁止写入RTC和后备寄存器 1: 允许写入RTC和后备寄存器								
位 7:5	PLS[2:0]: PVD电平选择 这些位用于选择电源电压监测器的电压阀值 <table style="margin-left: 200px; margin-top: 10px;"> <tr><td>000: 2.2V</td><td>100: 2.6V</td></tr> <tr><td>001: 2.3V</td><td>101: 2.7V</td></tr> <tr><td>010: 2.4V</td><td>110: 2.8V</td></tr> <tr><td>011: 2.5V</td><td>111: 2.9V</td></tr> </table> <p>注：详细说明参见数据手册中的电气特性部分。</p>	000: 2.2V	100: 2.6V	001: 2.3V	101: 2.7V	010: 2.4V	110: 2.8V	011: 2.5V	111: 2.9V
000: 2.2V	100: 2.6V								
001: 2.3V	101: 2.7V								
010: 2.4V	110: 2.8V								
011: 2.5V	111: 2.9V								
位 4	PVDE: 电源电压监测器(PVD)使能 0: 禁止PVD 1: 开启PVD								
位 3	CSBF: 清除待机位 始终读出为0 0: 无功效 1: 清除SBF待机位(写)								
位 2	CWUF: 清除唤醒位 始终读出为0 0: 无功效 1: 2个系统时钟周期后清除WUF唤醒位(写)								
位 1	PDDS: 掉电深睡眠 与LPDS位协同操作 0: 当CPU进入深睡眠时进入停机模式，调压器的状态由LPDS位控制。 1: CPU进入深睡眠时进入待机模式。								
位 0	LPDS: 深睡眠下的低功耗 PDDS=0时，与PDDS位协同操作 0: 在停机模式下电压调压器开启 1: 在停机模式下电压调压器处于低功耗模式								



图 21.1.2 PWR_CR 寄存器各位描述

这里我们通过设置 PWR_CR 的 PDDS 位，使 CPU 进入深度睡眠时进入待机模式，同时我们通过 CWUF 位，清除之前的唤醒位。电源控制/状态寄存器(PWR_CSR)的各位描述如图 21.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留					EWUP	保留					PVDO	SBF	WUF		
rw								r	r	r					

位31:9	保留。始终读为0。
位 8	EWUP: 使能WKUP管脚 0: WKUP管脚为通用I/O。WKUP管脚上的事件不能将CPU从待机模式唤醒 1: WKUP管脚用于将CPU从待机模式唤醒，WKUP管脚被强置为输入下拉的配置(WKUP管脚上的上升沿将系统从待机模式唤醒) 注：在系统复位时清除这一位。
位 7:3	保留。始终读为0。
位 2	PVDO: PVD输出 当PVD被PVDE位使能后该位才有效 0: V _{DD} /V _{DDA} 高于由PLS[2:0]选定的PVD阈值 1: V _{DD} /V _{DDA} 低于由PLS[2:0]选定的PVD阈值 注：在待机模式下PVD被停止。因此，待机模式后或复位后，直到设置PVDE位之前，该位为0。
位 1	SBF: 待机标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CSBF位清除。 0: 系统不在待机模式 1: 系统进入待机模式
位 0	WUF: 唤醒标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CWUF位清除。 0: 没有发生唤醒事件 1: 在WKUP管脚上发生唤醒事件或出现RTC闹钟事件。 注：当WKUP管脚已经是高电平时，在(通过设置EWUP位)使能WKUP管脚时，会检测到一个额外的事件。

图 21.1.3 PWR_CSR 寄存器各位描述

这里，我们通过设置 PWR_CSR 的 EWUP 位，来使能 WKUP 引脚用于待机模式唤醒。我们还可以从 WUF 来检查是否发生了唤醒事件。不过本章我们并没有用到。

通过以上介绍，我们了解了进入待机模式的方法，以及设置 WK_UP 引脚用于把 STM32 从待机模式唤醒的方法。具体步骤如下：

1) 使能电源时钟。

因为要配置电源控制寄存器，所以必须先使能电源时钟。

在库函数中，使能电源时钟的方法是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); //使能 PWR 外设时钟
```

这个函数非常容易理解。

2) 设置 WK_UP 引脚作为唤醒源。

使能时钟之后后再设置 PWR_CSR 的 EWUP 位，使能 WK_UP 用于将 CPU 从待机模式唤



醒。在库函数中，设置使能 WK_UP 用于唤醒 CPU 待机模式的函数是：

```
PWR_WakeUpPinCmd(ENABLE); //使能唤醒管脚功能
```

3) 设置 SLEEPDEEP 位，设置 PDDS 位，执行 WFI 指令，进入待机模式。

进入待机模式，首先要设置 SLEEPDEEP 位（该位在系统控制寄存器（SCB_SCR）的第二位，详见《CM3 权威指南》，第 182 页表 13.1），接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。在库函数中，进行上面三个功能进入待机模式是在函数 PWR_EnterSTANDBYMode 中实现的：

```
void PWR_EnterSTANDBYMode(void);
```

4) 最后编写 WK_UP 中断函数。

因为我们通过 WK_UP 中断（PA0 中断）来唤醒 CPU，所以我们有必要设置一下该中断函数，同时我们也通过该函数里面进入待机模式。

通过以上几个步骤的设置，我们就可以使用 STM32 的待机模式了，并且可以通过 WK_UP 来唤醒 CPU，我们最终要实现这样一个功能：通过长按（3 秒）WK_UP 按键开机，并且通过 DS0 的闪烁指示程序已经开始运行，再次长按该键，则进入待机模式，DS0 关闭，程序停止运行。类似于手机的开关机。

21.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) WK_UP 按键

本章，我们使用了 WK_UP 按键用于唤醒和进入待机模式。然后通过 DS0 来指示程序是否在运行。这两个硬件的连接前面均有介绍。

21.3 软件设计

打开待机唤醒实验工程，我们可以发现工程中多了一个 wkup.c 和 wkup.h 文件，相关的用户代码写在这两个文件中。同时，对于待机唤醒功能，我们需要引入 stm32f10x_pwr.c 和 stm32f0x_pwr.h 文件。

打开 wkup.c，可以看到如下关键代码：

```
void Sys_Standby(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); //使能 PWR 外设时钟
    PWR_WakeUpPinCmd(ENABLE); //使能唤醒管脚功能
    PWR_EnterSTANDBYMode(); //进入待命（STANDBY）模式
}

//系统进入待机模式
void Sys_Enter_Standy(void)
{
    RCC_APB2PeriphResetCmd(0X01FC,DISABLE); //复位所有 IO 口
    Sys_Standby();
}

//检测 WKUP 脚的信号
//返回值 1:连续按下 3s 以上
//      0:错误的触发
```



```
u8 Check_WKUP(void)
{
    u8 t=0;
    u8 tx=0;                                //记录松开的次数
    LED0=0;                                  //亮灯 DS0
    while(1)
    {
        if(WKUP_KD)                         //已经按下了
        {
            t++;
            tx=0;
        }else
        {
            tx++;                            //超过 300ms 内没有 WKUP 信号
            if(tx>3)
            {
                LED0=1;
                return 0;                      //错误的按键,按下次数不够
            }
        }
        delay_ms(30);
        if(t>=100)                          //按下超过 3 秒钟
        {
            LED0=0;                          //点亮 DS0
            return 1;                        //按下 3s 以上了
        }
    }
}

//中断,检测到 PA0 脚的一个上升沿.
//中断线 0 线上的中断检测
void EXTI0_IRQHandler(void)
{
    EXTI_ClearITPendingBit(EXTI_Line0); // 清除 LINE10 上的中断标志位
    if(Check_WKUP())                  //关机?
    {
        Sys_Enter_Standby();
    }
}

//PA0 WKUP 唤醒初始化
void WKUP_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
    RCC_APB2Periph_AFIO, ENABLE);           //使能 GPIOA 和复用功能时钟
```



```

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;           //PA.0
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD; //上拉输入
GPIO_Init(GPIOA, &GPIO_InitStructure);           //初始化 IO
//使用外部中断方式
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);
//中断线 0 连接 GPIOA.0
EXTI_InitStructure.EXTI_Line = EXTI_Line0;      //设置按键所有的外部线路
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //外部中断模式
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; //上升沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure); // 初始化外部中断

NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn; //使能外部中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; //先占优先级 2 级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2; //从优先级 2 级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //外部中断通道使能
NVIC_Init(&NVIC_InitStructure); // 初始化 NVIC

if(Check_WKUP() == 0) Sys_Standby(); //不是开机,进入待机模式
}

```

注意，这个项目中不要同时引用 exit.c 文件，因为 exit.c 里面也有一个 void EXTI0_IRQHandler(void) 函数，如果不删除，MDK 就会报错。该部分代码比较简单，我们在这里说明两点：1，在 void Sys_Enter_Standby(void) 函数里面，我们要在进入待机模式前把所有开启的外设全部关闭，我们这里仅仅复位了所有的 IO 口，使得 IO 口全部为浮空输入。其他外设（比如 ADC 等），大家根据自己所开启的情况进行一一关闭即可，这样才能达到最低功耗！2，在 void WKUP_Init(void) 函数里面，我们要先判断 WK_UP 是否按下了 3 秒钟，来决定要不要开机，如果没有按下 3 秒钟，程序直接就进入了待机模式。所以在下载完代码的时候，是看不到任何反应的。我们必须先按 WK_UP 按键 3 秒钟以开机，才能看到 DS0 闪烁。

wkup.h 头文件的代码非常简单，这里我们就不列出来。最后我们看看 main.c 里面 main 函数代码如下：

```

int main(void)
{
    delay_init(); //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2
    uart_init(9600); //串口初始化波特率为 9600
    LED_Init(); //LED 端口初始化
    WKUP_Init(); //待机唤醒初始化
    LCD_Init(); //LCD 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Warship STM32");
    LCD_ShowString(30,70,200,16,16,"WKUP TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2012/9/7");
}

```



```
while(1)
{
    LED0=!LED0;
    delay_ms(250);
}
}
```

这里我们先初始化 LED 和 WK_UP 按键（通过 WKUP_Init() 函数初始化），在死循环里面等待 WK_UP 中断的到来，在得到中断后，在中断函数里面判断 WK_UP 按下的时间长短，来决定是否进入待机模式。在 WKUP_Init 函数里面，我们有检测 WK_UP 是否按下 3 秒来决定是否开机，这点在前面已经介绍了。大家在下载完代码的时候要注意一下。

21.4 下载与测试

在代码编译成功之后，下载代码到 ALIENTEK 战舰 STM32 开发板上，此时，看到开发板 DS0 亮了一下（Check_WKUP 函数执行了 LED0=0 的操作），就没有反应了。其实这是正常的，在程序下载完之后，开发板不能检测到 WK_UP 的持续按下，所以直接进入待机模式，看起来和没有下载代码一样。此时，我们长按 WK_UP 按键 3 秒钟左右，可以看到 DS0 开始闪烁。然后再长按 WK_UP，DS0 会灭掉，程序再次进入待机模式。



第二十二章 ADC 实验

本章我们将向大家介绍 STM32 的 ADC 功能。在本章中，我们将利用 STM32 的 ADC1 通道 0 来采样外部电压值，并在 TFTLCD 模块上显示出来。本章将分为如下几个部分：

- 22.1 STM32 ADC 简介
- 22.2 硬件设计
- 22.3 软件设计
- 22.4 下载验证



22.1 STM32 ADC 简介

STM32 拥有 1~3 个 ADC(STM32F101/102 系列只有 1 个 ADC)，这些 ADC 可以独立使用，也可以使用双重模式（提高采样率）。STM32 的 ADC 是 12 位逐次逼近型的模拟数字转换器。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。

STM32F103 系列最少都拥有 2 个 ADC，我们选择的 STM32F103ZET 包含有 3 个 ADC。STM32 的 ADC 最大的转换速率为 1Mhz，也就是转换时间为 1us (在 ADCCLK=14M,采样周期为 1.5 个 ADC 时钟下得到)，不要让 ADC 的时钟超过 14M，否则将导致结果准确度下降。

STM32 将 ADC 的转换分为 2 个通道组：规则通道组和注入通道组。规则通道相当于你正常运行的程序，而注入通道呢，就相当于中断。在你程序正常执行的时候，中断是可以打断你的执行的。同这个类似，注入通道的转换可以打断规则通道的转换，在注入通道被转换完成之后，规则通道才得以继续转换。

通过一个形象的例子可以说明：假如你在家里的院子内放了 5 个温度探头，室内放了 3 个温度探头；你需要时刻监视室外温度即可，但偶尔你想看看室内的温度；因此你可以使用规则通道组循环扫描室外的 5 个探头并显示 AD 转换结果，当你想看室内温度时，通过一个按钮启动注入转换组(3 个室内探头)并暂时显示室内温度，当你放开这个按钮后，系统又会回到规则通道组继续检测室外温度。从系统设计上，测量并显示室内温度的过程中断了测量并显示室外温度的过程，但程序设计上可以在初始化阶段分别设置好不同的转换组，系统运行中不必再变更循环转换的配置，从而达到两个任务互不干扰和快速切换的结果。可以设想一下，如果没有规则组和注入组的划分，当你按下按钮后，需要从新配置 AD 循环扫描的通道，然后在释放按钮后需再次配置 AD 循环扫描的通道。

上面的例子因为速度较慢，不能完全体现这样区分(规则通道组和注入通道组)的好处，但在工业应用领域中有很多检测和监视探头需要较快地处理，这样对 AD 转换的分组将简化事件处理的程序并提高事件处理的速度。

STM32 其 ADC 的规则通道组最多包含 16 个转换，而注入通道组最多包含 4 个通道。关于这两个通道组的详细介绍，请参考《STM32 参考手册的》第 155 页，第 11 章。

STM32 的 ADC 可以进行很多种不同的转换模式，这些模式在《STM32 参考手册》的第 11 章也都有详细介绍，我们这里就不一一列举了。我们本章仅介绍如何使用规则通道的单次转换模式。

STM32 的 ADC 在单次转换模式下，只执行一次转换，该模式可以通过 ADC_CR2 寄存器的 ADON 位（只适用于规则通道）启动，也可以通过外部触发启动（适用于规则通道和注入通道），这是 CONT 位为 0。

以规则通道为例，一旦所选择的通道转换完成，转换结果将被存在 ADC_DR 寄存器中，EOC（转换结束）标志将被置位，如果设置了 EOCIE，则会产生中断。然后 ADC 将停止，直到下次启动。

接下来，我们介绍一下我们执行规则通道的单次转换，需要用到的 ADC 寄存器。第一个要介绍的是 ADC 控制寄存器 (ADC_CR1 和 ADC_CR2)。ADC_CR1 的各位描述如图 22.1.1 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
保留								AWDEN	AWD ENJ	保留		DUALMOD[3:0]				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
DISCNUM[2:0]	DISC ENJ	DISC EN	JAUTO	AWD SGL	SCAN	JEOC IE	AWDIE	EOCIE	AWDCH[4:0]							
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

图 22.1.1 ADC_CR1 寄存器各位描述

这里我们不再详细介绍每个位，而是抽出几个我们本章要用到的位进行针对性的介绍，详细的说明及介绍，请参考《STM32 参考手册》第 11 章的相关章节。

ADC_CR1 的 SCAN 位，该位用于设置扫描模式，由软件设置和清除，如果设置为 1，则使用扫描模式，如果为 0，则关闭扫描模式。在扫描模式下，由 ADC_SQRx 或 ADC_JSQRx 寄存器选中的通道被转换。如果设置了 EOCIE 或 JEDECIE，只在最后一个通道转换完毕后才会产生 EOC 或 JEDEC 中断。

ADC_CR1[19: 16]用于设置 ADC 的操作模式，详细的对应关系如图 22.1.2 所示：

位19:16	DUALMOD[3:0]: 双模式选择 软件使用这些位选择操作模式。 0000: 独立模式 0001: 混合的同步规则+注入同步模式 0010: 混合的同步规则+交替触发模式 0011: 混合同步注入+快速交替模式 0100: 混合同步注入+慢速交替模式 0101: 注入同步模式 0110: 规则同步模式 0111: 快速交替模式 1000: 慢速交替模式 1001: 交替触发模式 注：在ADC2和ADC3中这些位为保留位 在双模式中，改变通道的配置会产生一个重新开始的条件，这将导致同步丢失。建议在进行任何配置改变前关闭双模式。
--------	---

图 22.1.2 ADC 操作模式

本章我们要使用的是独立模式，所以设置这几位为 0 就可以了。接着我们介绍 ADC_CR2，该寄存器的各位描述如图 22.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
保留								TS VREFE	SW START	SW STARTJ	EXT TRIG	EXTSEL[2:0]				保留
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
JEXT TRIG	JEXTSEL[2:0]		ALIGN	保留	DMA	保留				RST CAL	CAL	CONT	ADON	保留		
RW	RW	RW	RW	RW	RW	RW								RW		

图 22.1.3 ADC_CR2 寄存器操作模式

该寄存器我们也只针对性的介绍一些位：ADON 用于开关 AD 转换器。而 CONT 位用于设置是否进行连续转换，我们使用单次转换，所以 CONT 位必须为 0。CAL 和 RSTCAL 用于 AD 校准。ALIGN 用于设置数据对齐，我们使用右对齐，该位设置为 0。

EXTSEL[2: 0]用于选择启动规则转换组转换的外部事件，详细的设置关系如图 22.1.4 所示：



位19:17	EXTSEL[2:0]: 选择启动规则通道组转换的外部事件 这些位选择用于启动规则通道组转换的外部事件 ADC1和ADC2的触发配置如下 <table border="0" style="width: 100%;"><tr><td style="width: 50%;">000: 定时器1的CC1事件</td><td style="width: 50%;">100: 定时器3的TRGO事件</td></tr><tr><td>001: 定时器1的CC2事件</td><td>101: 定时器4的CC4事件</td></tr><tr><td>010: 定时器1的CC3事件</td><td>110: EXTI线11/ TIM8_TRGO, 仅大容量产品具有TIM8_TRGO功能</td></tr><tr><td>011: 定时器2的CC2事件</td><td>111: SWSTART</td></tr></table> ADC3的触发配置如下 <table border="0" style="width: 100%;"><tr><td style="width: 50%;">000: 定时器3的CC1事件</td><td style="width: 50%;">100: 定时器8的TRGO事件</td></tr><tr><td>001: 定时器2的CC3事件</td><td>101: 定时器5的CC1事件</td></tr><tr><td>010: 定时器1的CC3事件</td><td>110: 定时器5的CC3事件</td></tr><tr><td>011: 定时器8的CC1事件</td><td>111: SWSTART</td></tr></table>	000: 定时器1的CC1事件	100: 定时器3的TRGO事件	001: 定时器1的CC2事件	101: 定时器4的CC4事件	010: 定时器1的CC3事件	110: EXTI线11/ TIM8_TRGO, 仅大容量产品具有TIM8_TRGO功能	011: 定时器2的CC2事件	111: SWSTART	000: 定时器3的CC1事件	100: 定时器8的TRGO事件	001: 定时器2的CC3事件	101: 定时器5的CC1事件	010: 定时器1的CC3事件	110: 定时器5的CC3事件	011: 定时器8的CC1事件	111: SWSTART
000: 定时器1的CC1事件	100: 定时器3的TRGO事件																
001: 定时器1的CC2事件	101: 定时器4的CC4事件																
010: 定时器1的CC3事件	110: EXTI线11/ TIM8_TRGO, 仅大容量产品具有TIM8_TRGO功能																
011: 定时器2的CC2事件	111: SWSTART																
000: 定时器3的CC1事件	100: 定时器8的TRGO事件																
001: 定时器2的CC3事件	101: 定时器5的CC1事件																
010: 定时器1的CC3事件	110: 定时器5的CC3事件																
011: 定时器8的CC1事件	111: SWSTART																

图 22.1.4 ADC 选择启动规则转换事件设置

我们这里使用的是软件触发 (SWSTART)，所以设置这 3 个位为 111。ADC_CR2 的 SWSTART 位用于开始规则通道的转换，我们每次转换（单次转换模式下）都需要向该位写 1。AWDEN 为用于使能温度传感器和 Vrefint。STM32 内部的温度传感器我们将在下一节介绍。

第二个要介绍的是 ADC 采样事件寄存器 (ADC_SMPR1 和 ADC_SMPR2)，这两个寄存器用于设置通道 0~17 的采样时间，每个通道占用 3 个位。ADC_SMPR1 的各位描述如图 22.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16									
保留						SMP17[2:0]	SMP16[2:0]	SMP15[2:1]																
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
SMP 15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]											
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW									
位31:24		保留。必须保持为0。																						
位23:0		SMPx[2:0]: 选择通道x的采样时间 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 <table border="0" style="width: 100%;"><tr><td style="width: 50%;">000: 1.5周期</td><td style="width: 50%;">100: 41.5周期</td></tr><tr><td>001: 7.5周期</td><td>101: 55.5周期</td></tr><tr><td>010: 13.5周期</td><td>110: 71.5周期</td></tr><tr><td>011: 28.5周期</td><td>111: 239.5周期</td></tr></table> 注： - ADC1的模拟输入通道16和通道17在芯片内部分别连到了温度传感器和VREFINT。 - ADC2的模拟输入通道16和通道17在芯片内部连到了VSS。 - ADC3模拟输入通道14, 15, 16, 17与Vss相连															000: 1.5周期	100: 41.5周期	001: 7.5周期	101: 55.5周期	010: 13.5周期	110: 71.5周期	011: 28.5周期	111: 239.5周期
000: 1.5周期	100: 41.5周期																							
001: 7.5周期	101: 55.5周期																							
010: 13.5周期	110: 71.5周期																							
011: 28.5周期	111: 239.5周期																							

图 22.1.5 ADC_SMPR1 寄存器各位描述

ADC_SMPR2 的各位描述如下图 22.1.6 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP 5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位31:30		保留。必须保持为0。													
位29:0		SMPx[2:0]: 选择通道x的采样时间 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。													
		000: 1.5周期 100: 41.5周期 001: 7.5周期 101: 55.5周期 010: 13.5周期 110: 71.5周期 011: 28.5周期 111: 239.5周期													
		注：ADC3模拟输入通道9与Vss相连													

图 22.1.6 ADC_SMPR2 寄存器各位描述

对于每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率。ADC 的转换时间可以由以下公式计算：

$$T_{covn} = \text{采样时间} + 12.5 \text{ 个周期}$$

其中：Tcovn 为总转换时间，采样时间是根据每个通道的 SMP 位的设置来决定的。例如，当 ADCCLK=14Mhz 的时候，并设置 1.5 个周期的采样时间，则得到：Tcovn=1.5+12.5=14 个周期=1us。

第三个要介绍的是 ADC 规则序列寄存器 (ADC_SQR1~3) ,该寄存器总共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC_SQR1，该寄存器的各位描述如图 22.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留						L[3:0]			SQ16[4:1]						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16 0	SQ15[4:0]			SQ14[4:0]			SQ13[4:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位31:24		保留。必须保持为0。													
位23:20		L[3:0]: 规则通道序列长度 这些位定义了在规则通道转换序列中转换总数。 0000: 1个转换 0001: 2个转换 1111: 16个转换													
位19:15		SQ16[4:0]: 规则序列中的第16个转换 这些位定义了转换序列中的第16个转换通道的编号(0~17)。													
位14:10		SQ15[4:0]: 规则序列中的第15个转换													
位9:5		SQ14[4:0]: 规则序列中的第14个转换													
位4:0		SQ13[4:0]: 规则序列中的第13个转换													

图 22.1.7 ADC_SQR1 寄存器各位描述

L[3:0]用于存储规则序列的长度，我们这里只用了 1 个，所以设置这几个位的值为 0。其他的 SQ13~16 则存储了规则序列中第 13~16 个通道的编号 (0~17)。另外两个规则序列寄存器



同 ADC_SQR1 大同小异，我们这里就不再介绍了，要说明一点的是：我们选择的是单次转换，所以只有一个通道在规则序列里面，这个序列就是 SQ1，通过 ADC_SQR3 的最低 5 位（也就是 SQ1）设置。

第四个要介绍的是 ADC 规则数据寄存器(ADC_DR)。规则序列中的 AD 转化结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。ADC_DR 的各位描述如图 22.1.8：

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31:16		ADC2DATA[15:0]: ADC2转换的数据 - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见10.9：双ADC模式 - 在ADC2中：不用这些位。													
位15:0		DATA[15:0]: 规则转换的数据 这些位为只读，包含了规则通道的转换结果。数据是左或右对齐，如图25和图26所示。													

图 22.1.8 ADC_JDRx 寄存器各位描述

这里要提醒一点的是，该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

最后一个要介绍的 ADC 寄存器为 ADC 状态寄存器 (ADC_SR)，该寄存器保存了 ADC 转换时的各种状态。该寄存器的各位描述如图 22.1.9 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16											
保留																										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
保留								STRT	JSTRT	JEOC	EOC	AWD														
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">位31:15</td><td style="padding: 2px;">保留。必须保持为0。</td></tr> <tr> <td style="padding: 2px;">位4</td><td style="padding: 2px;">STRT: 规则通道开始位 该位由硬件在规则通道转换开始时设置，由软件清除。 0: 规则通道转换未开始 1: 规则通道转换已开始</td></tr> <tr> <td style="padding: 2px;">位3</td><td style="padding: 2px;">JSTRT: 注入通道开始位 该位由硬件在注入通道组转换开始时设置，由软件清除。 0: 注入通道转换未开始 1: 注入通道转换已开始</td></tr> <tr> <td style="padding: 2px;">位2</td><td style="padding: 2px;">JEOC: 注入通道转换结束位 该位由硬件在所有注入通道组转换结束时设置，由软件清除 0: 转换未完成 1: 转换完成</td></tr> <tr> <td style="padding: 2px;">位1</td><td style="padding: 2px;">EOC: 转换结束位 该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取ADC_DR时清除 0: 转换未完成 1: 转换完成</td></tr> <tr> <td style="padding: 2px;">位0</td><td style="padding: 2px;">AWD: 模拟看门狗标志位 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置，由软件清除 0: 没有发生模拟看门狗事件 1: 发生模拟看门狗事件</td></tr> </table>															位31:15	保留。必须保持为0。	位4	STRT: 规则通道开始位 该位由硬件在规则通道转换开始时设置，由软件清除。 0: 规则通道转换未开始 1: 规则通道转换已开始	位3	JSTRT: 注入通道开始位 该位由硬件在注入通道组转换开始时设置，由软件清除。 0: 注入通道转换未开始 1: 注入通道转换已开始	位2	JEOC: 注入通道转换结束位 该位由硬件在所有注入通道组转换结束时设置，由软件清除 0: 转换未完成 1: 转换完成	位1	EOC: 转换结束位 该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取ADC_DR时清除 0: 转换未完成 1: 转换完成	位0	AWD: 模拟看门狗标志位 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置，由软件清除 0: 没有发生模拟看门狗事件 1: 发生模拟看门狗事件
位31:15	保留。必须保持为0。																									
位4	STRT: 规则通道开始位 该位由硬件在规则通道转换开始时设置，由软件清除。 0: 规则通道转换未开始 1: 规则通道转换已开始																									
位3	JSTRT: 注入通道开始位 该位由硬件在注入通道组转换开始时设置，由软件清除。 0: 注入通道转换未开始 1: 注入通道转换已开始																									
位2	JEOC: 注入通道转换结束位 该位由硬件在所有注入通道组转换结束时设置，由软件清除 0: 转换未完成 1: 转换完成																									
位1	EOC: 转换结束位 该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取ADC_DR时清除 0: 转换未完成 1: 转换完成																									
位0	AWD: 模拟看门狗标志位 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置，由软件清除 0: 没有发生模拟看门狗事件 1: 发生模拟看门狗事件																									

图 22.1.9 ADC_SR 寄存器各位描述

这里我们要用到的是 EOC 位，我们通过判断该位来决定是否此次规则通道的 AD 转换已经完成，如果完成我们就从 ADC_DR 中读取转换结果，否则等待转换完成。

通过以上寄存器的介绍，我们了解了 STM32 的单次转换模式下的相关设置，下面我们介绍使用库函数的函数来设定使用 ADC1 的通道 1 进行 AD 转换。这里需要说明一下，使用到的库函数分布在 stm32f10x_adc.c 文件和 stm32f10x_adc.h 文件中。下面讲解其详细设置步骤：

1) 开启 PA 口时钟和 ADC1 时钟，设置 PA1 为模拟输入。

STM32F103ZET6 的 ADC 通道 1 在 PA1 上，所以，我们先要使能 PORTA 的时钟和 **ADC1** 时钟，然后设置 PA1 为模拟输入。使能 GPIOA 和 ADC 时钟用 RCC_APB2PeriphClockCmd 函数，设置 PA1 的输入方式，使用 GPIO_Init 函数即可。这里我们列出 STM32 的 ADC 通道与 GPIO 对应表：



	ADC1	ADC2	ADC3
通道0	PA0	PA0	PA0
通道1	PA1	PA1	PA1
通道2	PA2	PA2	PA2
通道3	PA3	PA3	PA3
通道4	PA4	PA4	PF6
通道5	PA5	PA5	PF7
通道6	PA6	PA6	PF8
通道7	PA7	PA7	PF9
通道8	PB0	PB0	PF10
通道9	PB1	PB1	
通道10	PC0	PC0	PC0
通道11	PC1	PC1	PC1
通道12	PC2	PC2	PC2
通道13	PC3	PC3	PC3
通道14	PC4	PC4	
通道15	PC5	PC5	
通道16	温度传感器		
通道17	内部参照电压		

图 22.1.20 ADC 通道与 GPIO 对应表

2) 复位 ADC1, 同时设置 ADC1 分频因子。

开启 ADC1 时钟之后, 我们要复位 ADC1, 将 ADC1 的全部寄存器重设为缺省值之后我们就可以通过 RCC_CFGR 设置 ADC1 的分频因子。分频因子要确保 ADC1 的时钟 (ADCCLK) 不要超过 14Mhz。这个我们设置分频因子位 6, 时钟为 $72/6=12MHz$, 库函数的实现方法是:

```
RCC_ADCCLKConfig(RCC_PCLK2_Div6);
```

ADC 时钟复位的方法是:

```
ADC_DeInit(ADC1);
```

这个函数非常容易理解, 就是复位指定的 ADC。

3) 初始化 ADC1 参数, 设置 ADC1 的工作模式以及规则序列的相关信息。

在设置完分频因子之后, 我们就可以开始 ADC1 的模式配置了, 设置单次转换模式、触发方式选择、数据对齐方式等都在这一步实现。同时, 我们还要设置 ADC1 规则序列的相关信息, 我们这里只有一个通道, 并且是单次转换的, 所以设置规则序列中通道数为 1。这些在库函数中是通过函数 ADC_Init 实现的, 下面我们看看其定义:

```
void ADC_Init(ADC_TypeDef* ADCx, ADC_ItfTypeDef* ADC_InitStruct);
```

从函数定义可以看出, 第一个参数是指定 ADC 号。这里我们来看看第二个参数, 跟其他外设初始化一样, 同样是通过设置结构体成员变量的值来设定参数。

```
typedef struct
{
    uint32_t ADC_Mode;
    FunctionalState ADC_ScanConvMode;
    FunctionalState ADC_ContinuousConvMode;
    uint32_t ADC_ExternalTrigConv;
    uint32_t ADC_DataAlign;
    uint8_t ADC_NbrOfChannel;
}ADC_ItfTypeDef;
```



参数 ADC_Mode 故名是以是用来设置 ADC 的模式。前面讲解过，ADC 的模式非常多，包括独立模式，注入同步模式等等，这里我们选择独立模式，所以参数为 ADC_Mode_Independent。

参数 ADC_ScanConvMode 用来设置是否开启扫描模式，因为是单次转换，这里我们选择不开启值 DISABLE 即可。

参数 ADC_ContinuousConvMode 用来设置是否开启连续转换模式，因为是单次转换模式，所以我们选择不开启连续转换模式，DISABLE 即可。

参数 ADC_ExternalTrigConv 是用来设置启动规则转换组转换的外部事件，这里我们选择软件触发，选择值为 ADC_ExternalTrigConv_None 即可。

参数 DataAlign 用来设置 ADC 数据对齐方式是左对齐还是右对齐，这里我们选择右对齐方式 ADC_DataAlign_Right。

参数 ADC_NbrOfChannel 用来设置规则序列的长度，这里我们是单次转换，所以值为 1 即可。

通过上面对每个参数的讲解，下面来看看我们的初始化范例：

```
ADC_InitTypeDef ADC_InitStructure;
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; //ADC 工作模式:独立模式
ADC_InitStructure.ADC_ScanConvMode = DISABLE; //AD 单通道模式
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; //AD 单次转换模式
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
//转换由软件而不是外部触发启动
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //ADC 数据右对齐
ADC_InitStructure.ADC_NbrOfChannel = 1; //顺序进行规则转换的 ADC 通道的数目 1
ADC_Init(ADC1, &ADC_InitStructure); //根据指定的参数初始化外设 ADCx
```

5) 使能 ADC 并校准。

在设置完了以上信息后，我们就使能 AD 转换器，执行复位校准和 AD 校准，注意这两步是必须的！不校准将导致结果很不准确。

使能指定的 ADC 的方法是：

```
ADC_Cmd(ADC1, ENABLE); //使能指定的 ADC1
```

执行复位校准的方法是：

```
ADC_ResetCalibration(ADC1);
```

执行 ADC 校准的方法是：

```
ADC_StartCalibration(ADC1); //开始指定 ADC1 的校准状态
```

记住，每次进行校准之后要等待校准结束。这里是通过获取校准状态来判断是否校准是否结束。

下面我们一一列出复位校准和 AD 校准的等待结束方法：

```
while(ADC_GetResetCalibrationStatus(ADC1)); //等待复位校准结束
while(ADC_GetCalibrationStatus(ADC1)); //等待校 AD 准结束
```

6) 读取 ADC 值。

在上面的校准完成之后，ADC 就算准备好了。接下来我们要做的就是设置规则序列 1 里面的通道，采样顺序，以及通道的采样周期，然后启动 ADC 转换。在转换结束后，读取 ADC 转换结果值就是了。这里设置规则序列通道以及采样周期的函数是：

```
void ADC-RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel,
    uint8_t Rank, uint8_t ADC_SampleTime);
```

我们这里是规则序列中的第 1 个转换，同时采样周期为 239.5，所以设置为：

```
ADC-RegularChannelConfig(ADC1, ch, 1, ADC_SampleTime_239Cycles5 );
```

软件开启 ADC 转换的方法是：



ADC_SoftwareStartConvCmd(ADC1, ENABLE); //使能指定的 ADC1 的软件转换启动功能
开启转换之后，就可以获取转换 ADC 转换结果数据，方法是：

```
ADC_GetConversionValue(ADC1);
```

同时在 AD 转换中，我们还要根据状态寄存器的标志位来获取 AD 转换的各个状态信息。库函数获取 AD 转换的状态信息的函数是：

```
FlagStatus ADC_GetFlagStatus(ADC_TypeDef* ADCx, uint8_t ADC_FLAG)
```

比如我们要判断 ADC1d 的转换是否结束，方法是：

```
while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC)); //等待转换结束
```

这里还需要说明一下 ADC 的参考电压，战舰 STM32 开发板使用的是 STM32F103ZET6，该芯片有外部参考电压：Vref-和 Vref+，其中 Vref-必须和 VSSA 连接在一起，而 Vref+的输入范围为：2.4~VDDA。战舰 STM32 开发板通过 P7 端口，设置 Vref-和 Vref+设置参考电压，默认的我们是通过跳线帽将 Vref-接到 GND，Vref+接到 VDDA，参考电压就是 3.3V。如果大家想自己设置其他参考电压，将你的参考电压接在 Vref-和 Vref+上就 OK 了。本章我们的参考电压设置的是 3.3V。

通过以上几个步骤的设置，我们就能正常的使用 STM32 的 ADC1 来执行 AD 转换操作了。

22.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) TFTLCD 模块
- 3) ADC
- 4) 杜邦线

前面 2 个均已介绍过，而 ADC 属于 STM32 内部资源，实际上我们只需要软件设置就可以正常工作，不过我们需要在外部连接其端口到被测电压上面。本章，我们通过 ADC1 的通道 1 (PA1) 来读取外部电压值，战舰 STM32 开发板没有设计参考电压源在上面，但是板上有几个可以提供测试的地方：1，3.3V 电源。2，GND。3，后备电池。注意：这里不能接到板上 5V 电源上去测试，这可能会烧坏 ADC!

因为要连接到其他地方测试电压，所以我们需要 1 跟杜邦线，或者自备的连接线也可以，一头插在多功能端口 P14 的 ADC 插针上 (与 PA1 连接)，另外一头就接你要测试的电压点 (确保该电压不大于 3.3V 即可)。

22.3 软件设计

打开我们的 ADC 转换实验，可以看到工程中多了一个 adc.c 文件和 adc.h 文件。同时 ADC 相关的库函数是在 stm32f10x_adc.c 文件和 stm32f10x_adc.h 文件中。

打开 adc.c，可以看到代码如下：

```
//初始化 ADC
//这里我们仅以规则通道为例
//我们默认将开启通道 0~3
void Adc_Init(void)
{
    ADC_InitTypeDef ADC_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
```



```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |  
RCC_APB2Periph_ADC1 , ENABLE ); //使能 ADC1 通道时钟  
RCC_ADCCLKConfig(RCC_PCLK2_Div6); //设置 ADC 分频因子 6  
//72M/6=12,ADC 最大时间不能超过 14M  
//PA1 作为模拟通道输入引脚  
GPIO_InitStructure.GPIO_Pin =GPIO_Pin_1;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;//模拟输入  
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 GPIOA.1  
  
ADC_DeInit(ADC1); //复位 ADC1,将外设 ADC1 的全部寄存器重设为缺省值  
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; //ADC 独立模式  
ADC_InitStructure.ADC_ScanConvMode = DISABLE; //单通道模式  
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; //单次转换模式  
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;//转换由  
//软件而不是外部触发启动  
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //ADC 数据右对齐  
ADC_InitStructure.ADC_NbrOfChannel = 1; //顺序进行规则转换的 ADC 通道的数目  
ADC_Init(ADC1, &ADC_InitStructure); //根据指定的参数初始化外设 ADCx  
ADC_Cmd(ADC1, ENABLE); //使能指定的 ADC1  
ADC_ResetCalibration(ADC1); //开启复位校准  
while(ADC_GetResetCalibrationStatus(ADC1)); //等待复位校准结束  
ADC_StartCalibration(ADC1); //开启 AD 校准  
while(ADC_GetCalibrationStatus(ADC1)); //等待校准结束  
}  
//获得 ADC 值  
//ch:通道值 0~3  
u16 Get_Adc(u8 ch)  
{  
    //设置指定 ADC 的规则组通道, 设置它们的转化顺序和采样时间  
    ADC-RegularChannelConfig(ADC1, ch, 1, ADC_SampleTime_239Cycles5 ); //通道 1  
    //规则采样顺序值为 1,采样时间为 239.5 周期  
    ADC_SoftwareStartConvCmd(ADC1, ENABLE); //使能指定的 ADC1 的软件转换功能  
    while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC ));//等待转换结束  
    return ADC_GetConversionValue(ADC1); //返回最近一次 ADC1 规则组的转换结果  
}  
u16 Get_Adc_Average(u8 ch,u8 times)  
{  
    u32 temp_val=0;  
    u8 t;  
    for(t=0;t<times;t++)  
    {    temp_val+=Get_Adc(ch);  
        delay_ms(5);  
    }  
}
```



```
    return temp_val/times;  
}
```

此部分代码就 3 个函数，Adc_Init 函数用于初始化 ADC1。这里基本上是按我们上面的步骤来初始化的，我们仅开通了 1 个通道，即通道 1。第二个函数 Get_Adc，用于读取某个通道的 ADC 值，例如我们读取通道 1 上的 ADC 值，就可以通过 Get_Adc(1) 得到。最后一个函数 Get_Adc_Average，用于多次获取 ADC 值，取平均，用来提高准确度。

接下来看看 main.c 的代码如下：

```
int main(void)  
{  
    u16 adcx;  
    float temp;  
    delay_init();           //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2  
    uart_init(9600);        //串口初始化波特率为 9600  
    LED_Init();            //LED 端口初始化  
    LCD_Init();             //LCD 初始化  
    Adc_Init();             //ADC 初始化  
    POINT_COLOR=RED;        //设置字体为红色  
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");  
    LCD_ShowString(60,70,200,16,16,"ADC TEST");  
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(60,110,200,16,16,"2012/9/7");  
    //显示提示信息  
    POINT_COLOR=BLUE;       //设置字体为蓝色  
    LCD_ShowString(60,130,200,16,16,"ADC_CH0_VAL:");  
    LCD_ShowString(60,150,200,16,16,"ADC_CH0_VOL:0.000V");  
    while(1)  
    {  
        adcx=Get_Adc_Average(ADC_CH1,10);  
        LCD_ShowxNum(156,130,adcx,4,16,0);//显示 ADC 的值  
        temp=(float)adcx*(3.3/4096);  
        adcx=temp;  
        LCD_ShowxNum(156,150,adcx,1,16,0);//显示电压值  
        temp-=adcx;  
        temp*=1000;  
        LCD_ShowxNum(172,150,temp,3,16,0X80);  
        LED0=!LED0;  
        delay_ms(250);  
    }  
}
```

此部分代码，我们在 TFTLCD 模块上显示一些提示信息后，将每隔 250ms 读取一次 ADC 通道 0 的值，并显示读到的 ADC 值(数字量)，以及其转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。



22.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 22.4.1 所示：

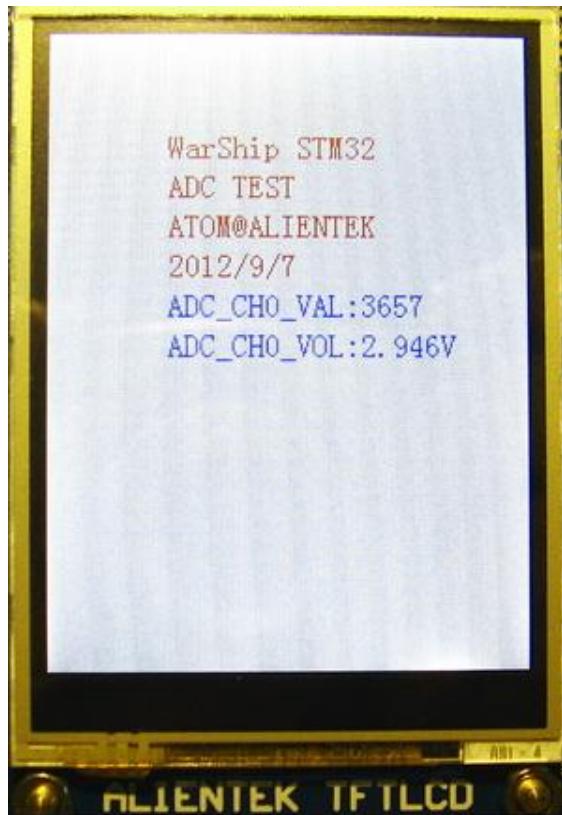


图 22.4.1 ADC 实验实际测试图

上图中，我们是将 ADC 和 TPAD 连接在一起，可以看到 TPAD 信号电平为 3V 左右，这是因为存在上拉电阻 R59 的缘故。

同时伴随 DS0 的不停闪烁，提示程序在运行。大家可以试试把杜邦线接到其他地方，看看电压值是否准确？但是一定别接到 5V 上面去，否则可能烧坏 ADC！

通过这一章的学习，我们了解了 STM32 ADC 的使用，但这仅仅是 STM32 强大的 ADC 功能的一小点应用。STM32 的 ADC 在很多地方都可以用到，其 ADC 的 DMA 功能是很不错的，建议有兴趣的大家深入研究下 STM32 的 ADC，相信会给你以后的开发带来方便。



第二十三章 内部温度传感器实验

本章我们将向大家介绍 STM32 的内部温度传感器。在本章中，我们将利用 STM32 的内部温度传感器来读取温度值，并在 TFTLCD 模块上显示出来。本章分为如下几个部分：

- 23.1 STM32 内部温度传感器简介
- 23.2 硬件设计
- 23.3 软件设计
- 23.4 下载验证



23.1 STM32 内部温度传感器简介

STM32 有一个内部的温度传感器，可以用来测量 CPU 及周围的温度(TA)。该温度传感器在内部和 ADCx_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值。温度传感器模拟输入推荐采样时间是 $17.1 \mu s$ 。STM32 的内部温度传感器支持的温度范围为：-40~125 度。精度比较差，为 $\pm 1.5^{\circ}C$ 左右。

STM32 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部通道就差不多了。关于 ADC 的设置，我们在第十八章已经进行了详细的介绍，这里就不再多说。接下来我们介绍一下和温度传感器设置相关的 2 个地方。

第一个地方，我们要使用 STM32 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CR2 的 AWDEN 位 (bit23) 设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32 的内部温度传感器固定的连接在 ADC 的通道 16 上，所以，我们在设置好 ADC 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以计算出当前温度。计算公式如下：

$$T ({}^{\circ}C) = \{ (V_{25} - V_{sense}) / Avg_Slope \} + 25$$

上式中：

$V_{25} = V_{sense}$ 在 25 度时的数值（典型值为：1.43）。

Avg_Slope =温度与 V_{sense} 曲线的平均斜率（单位为 $mv/{}^{\circ}C$ 或 $uv/{}^{\circ}C$ ）（典型值为 $4.3Mv/{}^{\circ}C$ ）。

利用以上公式，我们就可以方便的计算出当前温度传感器的温度了。

现在，我们就可以总结一下 STM32 内部温度传感器使用的步骤了，如下：

1) 设置 ADC，开启内部温度传感器。

关于如何设置 ADC，上一节已经介绍了，我们采用与上一节相似的设置。不同的是上一节温度传感器是读取外部通道的值，而内部温度传感器相当与把通道端口连接在内部温度传感器上。所以这里，我们要开启内部温度传感器功能：

```
ADC_TempSensorVrefintCmd(ENABLE);
```

2) 读取通道 16 的 AD 值，计算结果。

在设置完之后，我们就可以读取温度传感器的电压值了，得到该值就可以用上面的公式计算温度值。从上个实验的 ADC 通道与 GPIO 对应表（图 22.1.20 ADC 通道与 GPIO 对应表）可以知道，内部温度传感器是通过对应的是 ADC 的通道 16。其它的跟上一节的讲解是一样的。

23.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 D80
- 2) TFTLCD 模块
- 3) ADC
- 4) 内部温度传感器

前三个之前均有介绍，而内部温度传感器也是在 STM32 内部，不需要外部设置，我们只需要软件设置就 OK 了。



23.3 软件设计

打开内部温度传感器实验工程，可以看到文件中多了一个 tsensor.c 文件和 tsensor.h 文件。tsensor.c 文件中有三个函数分别为 T_Adc_Init, T_Get_Temp, T_Get_Advanced_Average.这三个函数的作用跟上章 ADC 实验基本是一样的。不同的是在 Adc_Init 函数中设置为开启内部温度传感器模式，代码如下：

```
void T_Adc_Init(void) //ADC 通道初始化
{
    ADC_InitTypeDef ADC_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
    RCC_APB2Periph_ADC1, ENABLE );           //使能 GPIOA,ADC1 通道时钟
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);        //分频因子 6 时钟为 72M/6=12MHz
    ADC_DeInit(ADC1);                      //将外设 ADC1 的全部寄存器重设为缺省值

    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; //ADC 独立模式
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;        //模数转换:单通道模式
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; //单次转换模式
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None; //软件触发
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //ADC 数据右对齐
    ADC_InitStructure.ADC_NbrOfChannel = 1; //顺序进行规则转换的 ADC 通道的数目
    ADC_Init(ADC1, &ADC_InitStructure); //根据指定的参数初始化 ADCx

    ADC_TempSensorVrefintCmd(ENABLE); //开启内部温度传感器
    ADC_Cmd(ADC1, ENABLE);          //使能指定的 ADC1
    ADC_ResetCalibration(ADC1);     //重置指定的 ADC1 的复位寄存器
    while(ADC_GetResetCalibrationStatus(ADC1)); //等待校准完成
    ADC_StartCalibration(ADC1);     //AD 校准
    while(ADC_GetCalibrationStatus(ADC1)); //等待校准完成
}
```

这部分代码与上一章的 Adc_Init 代码几乎一摸一样，去掉外部 IO 初始化，同时在函数里面增加了如下代码开启温度传感器：

```
ADC_TempSensorVrefintCmd(ENABLE); //开启内部温度传感器
```

接着打开 tsensor.h 可以看到，相比 ADC 实验，我们也只增加了一句，就是宏定义多增加了一个温度传感器通道 ADC_CH_TEMP。接下来我们就可以开始读取温度传感器的电压了。在 main.c 文件里面我们的 main 函数代码如下：

```
int main(void)
{
    u16 adcx;
    float temp;
    float temperate;
    delay_init(); //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
```



```
uart_init(9600);          //串口初始化波特率为 9600
LED_Init();                //LED 端口初始化
LCD_Init();                //LCD 初始化
T_Adc_Init();              //ADC 初始化

POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,200,16,16,"WarShip STM32");
LCD_ShowString(60,70,200,16,16,"ADC TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/7");
//显示提示信息
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(60,130,200,16,16,"TEMP_VAL:");
LCD_ShowString(60,150,200,16,16,"TEMP_VOL:0.000V");
LCD_ShowString(60,170,200,16,16,"TEMPERATE:00.00C");
while(1)
{
    adcx=T_Get_Adc_Average(ADC_CH_TEMP,10);
    LCD_ShowxNum(132,130,adcx,4,16,0);//显示 ADC 的值
    temp=(float)adcx*(3.3/4096);
    temperate=temp;                      //保存温度传感器的电压值
    adcx=temp;
    LCD_ShowxNum(132,150,adcx,1,16,0);      //显示电压值整数部分
    temp=(u8)temp;                        //减掉整数部分
    LCD_ShowxNum(148,150,temp*1000,3,16,0X80); //显示电压小数部分
    temperate=(1.43-temperate)/0.0043+25;   //计算出当前温度值
    LCD_ShowxNum(140,170,(u8)temperate,2,16,0); //显示温度整数部分
    temperate=(u8)temperate;
    LCD_ShowxNum(164,170,temperate*100,2,16,0X80); //显示温度小数部分
    LED0=!LED0;
    delay_ms(250);
}
}
```

这里同上一章的主函数也大同小异，上面的代码将温度传感器得到的电压值，换算成温度值。然后，我们在 TFTLCD 模块上显示出来。

代码设计部分就为大家讲解到这里，下面我们开始下载验证。

23.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 23.4.1 所示：



图 23.4.1 内部温度传感器实验测试图

伴随 DS0 的不停闪烁，提示程序在运行。大家可以看看你的温度值与实际是否相符合（因为芯片会发热，一般会比实际温度稍高一些）？



第二十四章 DAC 实验

上两章，我们介绍了 STM32 的 ADC 使用，本章我们将向大家介绍 STM32 的 DAC 功能。在本章中，我们将利用按键（或 USMART）控制 STM32 内部 DAC 模块的通道 1 来输出电压，通过 ADC1 的通道 1 采集 DAC 的输出电压，在 LCD 模块上面显示 ADC 获取到的电压值以及 DAC 的设定输出电压值等信息。本章将分为如下几个部分：

- 24.1 STM32 DAC 简介
- 24.2 硬件设计
- 24.3 软件设计
- 24.4 下载验证



24.1 STM32 DAC 简介

大容量的 STM32F103 具有内部 DAC，战舰 STM32 选择的是 STM32F103ZET6 属于大容量产品，所以是带有 DAC 模块的。

STM32 的 DAC 模块(数字/模拟转换模块)是 12 位数字输入，电压输出型的 DAC。DAC 可以配置为 8 位或 12 位模式，也可以与 DMA 控制器配合使用。DAC 工作在 12 位模式时，数据可以设置成左对齐或右对齐。DAC 模块有 2 个输出通道，每个通道都有单独的转换器。在双 DAC 模式下，2 个通道可以独立地进行转换，也可以同时进行转换并同步地更新 2 个通道的输出。DAC 可以通过引脚输入参考电压 VREF+ 以获得更精确的转换结果。

STM32 的 DAC 模块主要特点有：

- ① 2 个 DAC 转换器：每个转换器对应 1 个输出通道
- ② 8 位或者 12 位单调输出
- ③ 12 位模式下数据左对齐或者右对齐
- ④ 同步更新功能
- ⑤ 噪声波形生成
- ⑥ 三角波形生成
- ⑦ 双 DAC 通道同时或者分别转换
- ⑧ 每个通道都有 DMA 功能

单个 DAC 通道的框图如图 24.1.1 所示：

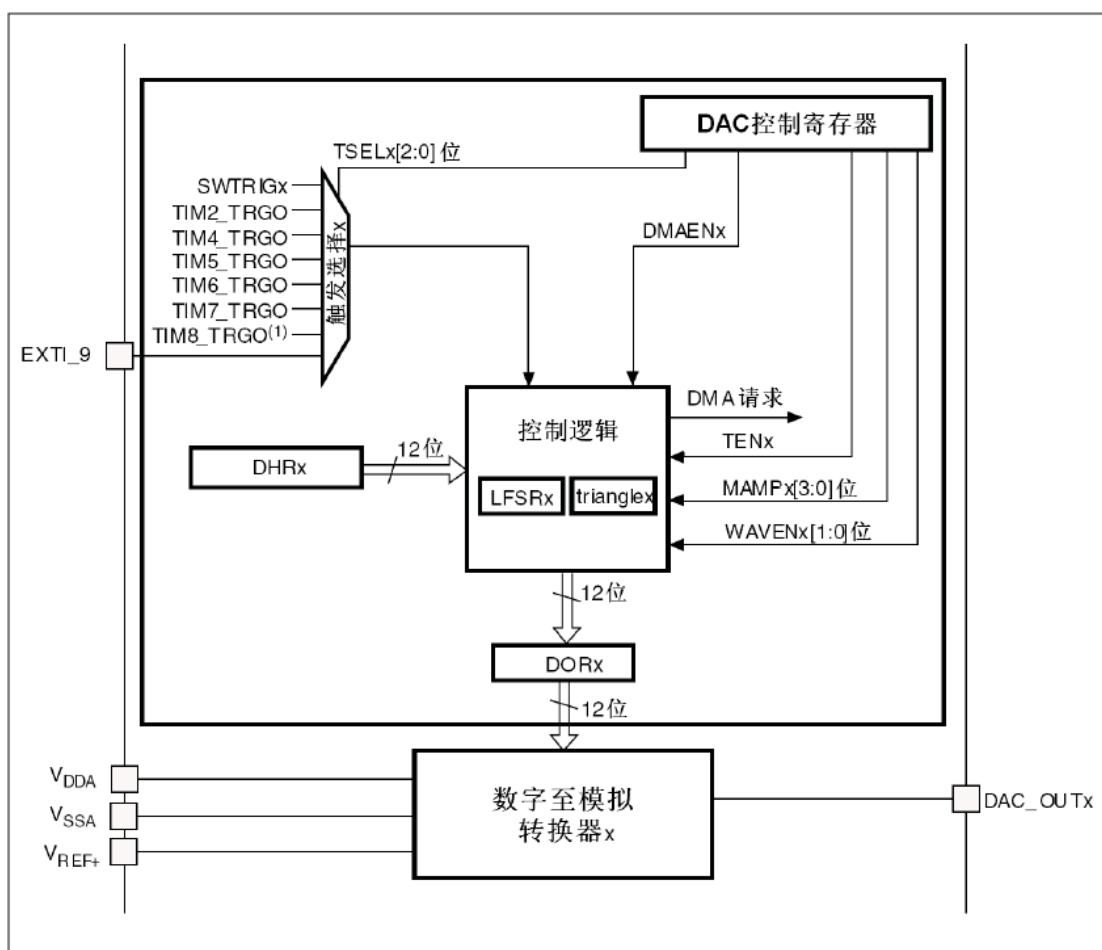


图 24.1.1 DAC 通道模块框图



图中 VDDA 和 VSSA 为 DAC 模块模拟部分的供电，而 Vref+则是 DAC 模块的参考电压。DAC_OUTx 就是 DAC 的输出通道了（对应 PA4 或者 PA5 引脚）。

从图 24.1.1 可以看出，DAC 输出是受 DORx 寄存器直接控制的，但是我们不能直接往 DORx 寄存器写入数据，而是通过 DHRx 间接的传给 DORx 寄存器，实现对 DAC 输出的控制。前面我们提到，STM32 的 DAC 支持 8/12 位模式，8 位模式的时候是固定的右对齐的，而 12 位模式又可以设置左对齐/右对齐。单 DAC 通道 x，总共有 3 种情况：

- ① 8 位数据右对齐：用户将数据写入 DAC_DHR8Rx[7:0]位（实际是存入 DHRx[11:4]位）。
- ② 12 位数据左对齐：用户将数据写入 DAC_DHR12Lx[15:4]位（实际是存入 DHRx[11:0]位）。
- ③ 12 位数据右对齐：用户将数据写入 DAC_DHR12Rx[11:0]位（实际是存入 DHRx[11:0]位）。

我们本章使用的就是单 DAC 通道 1，采用 12 位右对齐格式，所以采用第③种情况。

如果没有选中硬件触发(寄存器 DAC_CR1 的 TENx 位置'0')，存入寄存器 DAC_DHRx 的数据会在一个 APB1 时钟周期后自动传至寄存器 DAC_DORx。如果选中硬件触发(寄存器 DAC_CR1 的 TENx 位置'1')，数据传输在触发发生以后 3 个 APB1 时钟周期后完成。一旦数据从 DAC_DHRx 寄存器装入 DAC_DORx 寄存器，在经过时间 $t_{SETTLING}$ 之后，输出即有效，这段时间的长短依电源电压和模拟输出负载的不同会有所变化。我们可以从 STM32F103ZET6 的数据手册查到 $t_{SETTLING}$ 的典型值为 3us，最大是 4us。所以 DAC 的转换速度最快是 250K 左右。

本章我们将不使用硬件触发 (TEN=0)，其转换的时间框图如图 24.1.2 所示：

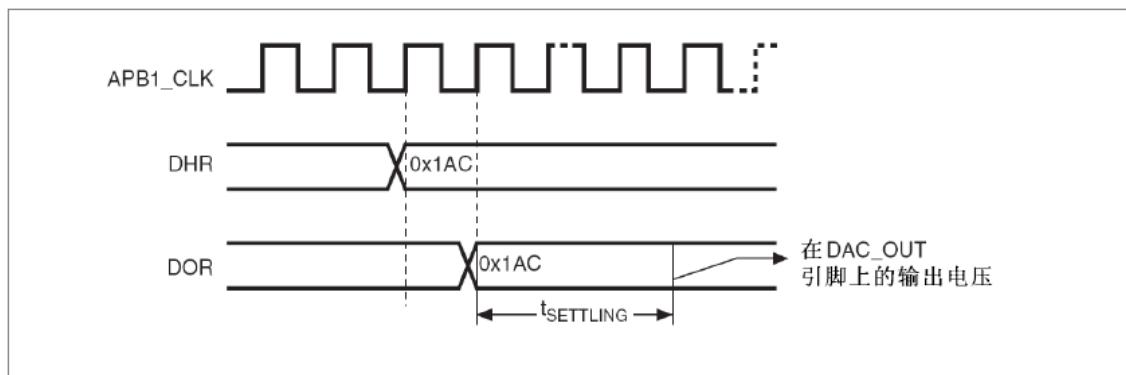


图 24.1.2 TEN=0 时 DAC 模块转换时间框图

当 DAC 的参考电压为 Vref+的时候，DAC 的输出电压是线性的从 0~Vref+，12 位模式下 DAC 输出电压与 Vref+以及 DORx 的计算公式如下：

$$DACx \text{ 输出电压} = Vref^* (DORx/4095)$$

接下来，我们介绍一下要实现 DAC 的通道 1 输出，需要用到的一些寄存器。首先是 DAC 控制寄存器 DAC_CR，该寄存器的各位描述如图 24.1.3 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留	DMAEN2	MAMP2[3:0]		WAVE2[2:0]	TSEL2[2:0]	TEN2	BOFF2	EN2							
	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
保留	DMAEN1	MAMP13:0]		WAVE1[2:0]	TSEL1[2:0]	TEN1	BOFF1	EN1							
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 24.1.3 寄存器 DAC_CR 各位描述

DAC_CR 的低 16 位用于控制通道 1，而高 16 位用于控制通道 2，我们这里仅列出比较重要的最低 8 位的详细描述，如图 24.1.4 所示：

位7:6	WAVE1[1:0]: DAC通道1噪声/三角波生成使能 (DAC channel1 noise/triangle wave generation enable) 该2位由软件设置和清除。 00: 关闭波形生成； 10: 使能噪声波形发生器； 1x: 使能三角波发生器。
位5:3	TSEL1[2:0]: DAC通道1触发选择 (DAC channel1 trigger selection) 该位用于选择DAC通道1的外部触发事件。 000: TIM6 TRGO事件； 001: 对于互联型产品是TIM3 TRGO事件，对于大容量产品是TIM8 TRGO事件； 010: TIM7 TRGO事件； 011: TIM5 TRGO事件； 100: TIM2 TRGO事件； 101: TIM4 TRGO事件； 110: 外部中断线9； 111: 软件触发。 注意：该位只能在TEN1= 1(DAC通道1触发使能)时设置。
位2	TEN1: DAC通道1触发使能 (DAC channel1 trigger enable) 该位由软件设置和清除，用来使能/关闭DAC通道1的触发。 0: 关闭DAC通道1触发，写入寄存器DAC_DHRx的数据在1个APB1时钟周期后传入寄存器DAC_DOR1； 1: 使能DAC通道1触发，写入寄存器DAC_DHRx的数据在3个APB1时钟周期后传入寄存器DAC_DOR1。 注意：如果选择软件触发，写入寄存器DAC_DHRx的数据只需要1个APB1时钟周期就可以传入寄存器DAC_DOR1。
位1	BOFF1: 关闭DAC通道1输出缓存 (DAC channel1 output buffer disable) 该位由软件设置和清除，用来使能/关闭DAC通道1的输出缓存。 0: 使能DAC通道1输出缓存； 1: 关闭DAC通道1输出缓存。
位0	EN1: DAC通道1使能 (DAC channel1 enable) 该位由软件设置和清除，用来使能/失能DAC通道1。 0: 关闭DAC通道1； 1: 使能DAC通道1。

图 24.1.4 寄存器 DAC_CR 低八位详细描述

首先，我们来看 DAC 通道 1 使能位(EN1)，该位用来控制 DAC 通道 1 使能的，本章我们就是用的 DAC 通道 1，所以该位设置为 1。

再看关闭 DAC 通道 1 输出缓存控制位 (BOFF1)，这里 STM32 的 DAC 输出缓存做的有些



不好，如果使能的话，虽然输出能力强一点，但是输出没法到 0，这是个很严重的问题。所以本章我们不使用输出缓存。即设置该位为 1。

DAC 通道 1 触发使能位 (TEN1)，该位用来控制是否使用触发，里我们不使用触发，所以设置该位为 0。

DAC 通道 1 触发选择位 (TSEL1[2:0])，这里我们没用到外部触发，所以设置这几个位为 0 就行了。

DAC 通道 1 噪声/三角波生成使能位 (WAVE1[1:0])，这里我们同样没用到波形发生器，故也设置为 0 即可。

DAC 通道 1 屏蔽/幅值选择器 (MAMP[3:0])，这些位仅在使用了波形发生器的时候有用，本章没有用到波形发生器，故设置为 0 就可以了。

最后是 DAC 通道 1 DMA 使能位 (DMAEN1)，本章我们没有用到 DMA 功能，故还是设置为 0。

通道 2 的情况和通道 1 一模一样，这里就不细说了。在 DAC_CR 设置好之后，DAC 就可以正常工作了，我们仅需要再设置 DAC 的数据保持寄存器的值，就可以在 DAC 输出通道得到你想要的电压了（对应 IO 口设置为模拟输入）。本章，我们用的是 DAC 通道 1 的 12 位右对齐数据保持寄存器：DAC_DHR12R1，该寄存器各位描述如图 24.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16												
保留																											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
保留				DACC1DHR[11:0]																							
位31:12	保留。																										
位11:0	DACC1DHR[11:0]: DAC通道1的12位右对齐数据 (DAC channel1 12-bit right-aligned data) 该位由软件写入，表示DAC通道1的12位数据。																										
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw												

图 24.1.5 寄存器 DAC_DHR12R1 各位描述

该寄存器用来设置 DAC 输出，通过写入 12 位数据到该寄存器，就可以在 DAC 输出通道 1 (PA4) 得到我们所要的结果。

通过以上介绍，我们了解了 STM32 实现 DAC 输出的相关设置，本章我们将使用库函数的方法来设置 DAC 模块的通道 1 来输出模拟电压，其详细设置步骤如下：

1) 开启 PA 口时钟，设置 PA4 为模拟输入。

STM32F103ZET6 的 DAC 通道 1 在 PA4 上，所以，我们先要使能 PORTA 的时钟，然后设置 PA4 为模拟输入。DAC 本身是输出，但是为什么端口要设置为模拟输入模式呢？因为一旦使能 DACx 通道之后，相应的 GPIO 引脚 (PA4 或者 PA5) 会自动与 DAC 的模拟输出相连，设置为输入，是为了避免额外的干扰。

使能 GPIOA 时钟：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能 PORTA 时钟
```

设置 PA1 为模拟输入只需要设置初始化参数即可：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //模拟输入
```

2) 使能 DAC1 时钟。

同其他外设一样，要想使用，必须先开启相应的时钟。STM32 的 DAC 模块时钟是由 APB1 提供的，所以我们调用函数 RCC_APB1PeriphClockCmd() 设置 DAC 模块的时钟使能。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE); //使能 DAC 通道时钟
```



3) 初始化 DAC, 设置 DAC 的工作模式。

该部分设置全部通过 DAC_CR 设置实现，包括：DAC 通道 1 使能、DAC 通道 1 输出缓存关闭、不使用触发、不使用波形发生器等设置。这里 DMA 初始化是通过函数 DAC_Init 完成的：

```
void DAC_Init(uint32_t DAC_Channel, DAC_InitTypeDef* DAC_InitStruct)
```

跟前面一样，首先我们来看看参数设置结构体类型 DAC_InitTypeDef 的定义：

```
typedef struct
{
    uint32_t DAC_Trigger;
    uint32_t DAC_WaveGeneration;
    uint32_t DAC_LFSRUnmask_TriangleAmplitude;
    uint32_t DAC_OutputBuffer;
}DAC_InitTypeDef;
```

这个结构体的定义还是比较简单的，只有四个成员变量，下面我们一一讲解。

第一个参数 DAC_Trigger 用来设置是否使用触发功能，前面已经讲解过这个的含义，这里我们不是用触发功能，所以值为 DAC_Trigger_None。

第二个参数 DAC_WaveGeneration 用来设置是否使用波形发生，这里我们前面同样讲解过不使用。所以值为 DAC_WaveGeneration_None。

第三个参数 DAC_LFSRUnmask_TriangleAmplitude 用来设置屏蔽/幅值选择器，这个变量只在使用波形发生器的时候才有用，这里我们设置为 0 即可，值为 DAC_LFSRUnmask_Bit0。

第四个参数 DAC_OutputBuffer 是用来设置输出缓存控制位，前面讲解过，我们不使用输出缓存，所以值为 DAC_OutputBuffer_Disable。到此四个参数设置完毕。看看我们的实例代码：

```
DAC_InitTypeDef DAC_InitType;
DAC_InitType.DAC_Trigger=DAC_Trigger_None; //不使用触发功能 TEN1=0
DAC_InitType.DAC_WaveGeneration=DAC_WaveGeneration_None;//不使用波形发生
DAC_InitType.DAC_LFSRUnmask_TriangleAmplitude=DAC_LFSRUnmask_Bit0;
DAC_InitType.DAC_OutputBuffer=DAC_OutputBuffer_Disable ; //DAC1 输出缓存关闭
DAC_Init(DAC_Channel_1,&DAC_InitType); //初始化 DAC 通道 1
```

4) 使能 DAC 转换通道

初始化 DAC 之后，理所当然要使能 DAC 转换通道，库函数方法是：

```
DAC_Cmd(DAC_Channel_1, ENABLE); //使能 DAC1
```

5) 设置 DAC 的输出值。

通过前面 4 个步骤的设置，DAC 就可以开始工作了，我们使用 12 位右对齐数据格式，所以我们通过设置 DHR12R1，就可以在 DAC 输出引脚（PA4）得到不同的电压值了。库函数的函数是：

```
DAC_SetChannel1Data(DAC_Align_12b_R, 0);
```

第一个参数设置对齐方式，可以为 12 位右对齐 DAC_Align_12b_R，12 位左对齐 DAC_Align_12b_L 以及 8 位右对齐 DAC_Align_8b_R 方式。

第二个参数就是 DAC 的输入值了，这个很好理解，初始化设置为 0。

这里，还可以读出 DAC 的数值，函数是：

```
DAC_GetDataOutputValue(DAC_Channel_1);
```

设置和读出一一对应很好理解，这里就不多讲解了。



最后，再提醒一下大家，本例程，我们使用的是 3.3V 的参考电压，即 Vref+连接 VDDA。
通过以上几个步骤的设置，我们就能正常的使用 STM32 的 DAC 通道 1 来输出不同的模拟电压了。

24.2 硬件设计

本章用到的硬件资源有：

- 1)指示灯 DS0
- 2) WK_UP 和 KEY1 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) ADC
- 6) DAC

本章，我们使用 DAC 通道 1 输出模拟电压，然后通过 ADC1 的通道 1 对该输出电压进行读取，并显示在 LCD 模块上面，DAC 的输出电压，我们通过按键（或 USMART）进行设置。

我们需要用到 ADC 采集 DAC 的输出电压，所以需要在硬件上把他们短接起来。ADC 和 DAC 的连接原理图如图 24.2.1 所示：

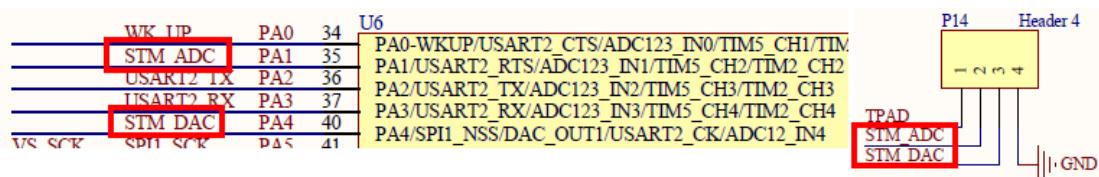


图 24.2.1 ADC、DAC 与 STM32 连接原理图

P14 是多功能端口，我们只需要通过跳线帽短接 P14 的 ADC 和 DAC，就可以开始做本章实验了。如图 24.2.2 所示：

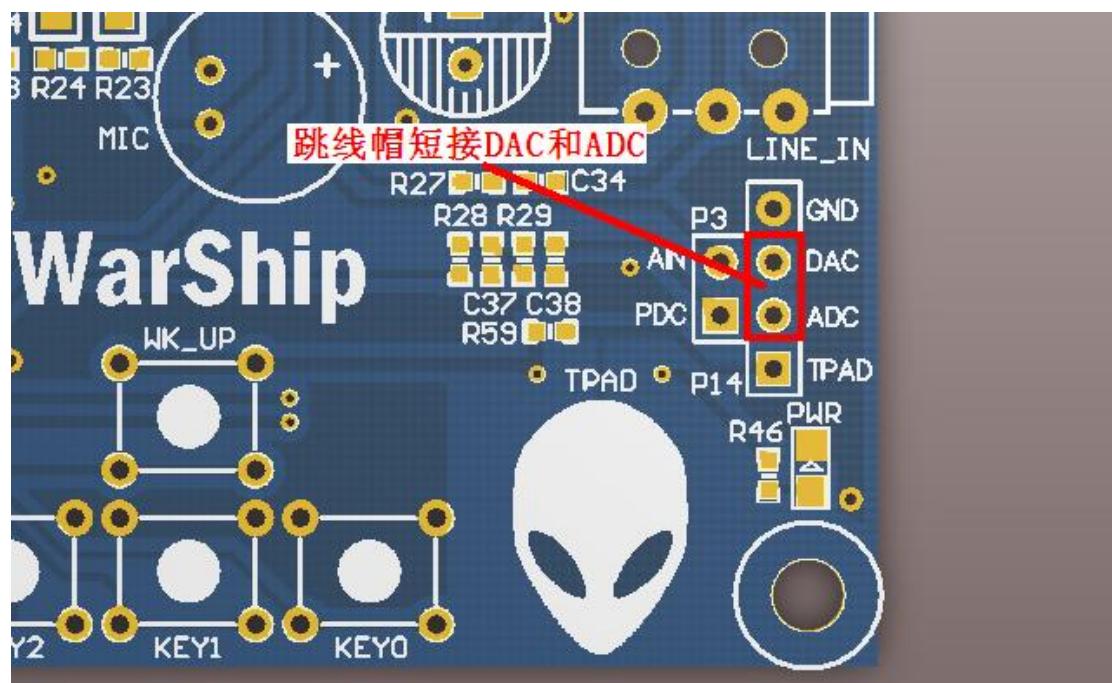


图 24.2.2 硬件连接示意图



24.3 软件设计

打开光盘的 DAC 实验可以看到，项目中添加了 dac.c 文件以及头文件 dac.h。同时，dac 相关的函数分布在固件库文件 stm32f10x_dac.c 文件和 stm32f10x_dac.h 头文件中。

打开 dac.c，代码如下：

```
#include "dac.h"
//DAC 通道 1 输出初始化
void Dac1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    DAC_InitTypeDef DAC_InitType;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE ); //①使能 PA 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE ); //②使能 DAC 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4; // 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //模拟输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure); //①初始化 GPIOA
    GPIO_SetBits(GPIOA,GPIO_Pin_4) ; //PA.4 输出高

    DAC_InitType.DAC_Trigger=DAC_Trigger_None; //不使用触发功能
    DAC_InitType.DAC_WaveGeneration=DAC_WaveGeneration_None;//不使用波形发生
    DAC_InitType.DAC_LFSRUnmask_TriangleAmplitude=DAC_LFSRUnmask_Bit0;
    DAC_InitType.DAC_OutputBuffer=DAC_OutputBuffer_Disable ; //DAC1 输出缓存关闭
    DAC_Init(DAC_Channel_1,&DAC_InitType); //③初始化 DAC 通道 1

    DAC_Cmd(DAC_Channel_1, ENABLE); //④使能 DAC1
    DAC_SetChannel1Data(DAC_Align_12b_R, 0); //⑤12 位右对齐,设置 DAC 初始值
}

//设置通道 1 输出电压
//vol:0~3300,代表 0~3.3V
void Dac1_Set_Vol(u16 vol)
{
    float temp=vol;
    temp/=1000;
    temp=temp*4096/3.3;
    DAC_SetChannel1Data(DAC_Align_12b_R,temp);// 12 位右对齐设置 DAC 值
}
```

此部分代码有 2 个函数，Dac1_Init 函数用于初始化 DAC 通道 1。步骤①~⑤基本上是按我们上面的步骤来初始化的，经过这个初始化之后，我们就可以正常使用 DAC 通道 1 了。第二个函数 Dac1_Set_Vol，用于设置 DAC 通道 1 的输出电压，通过 USMART 调用该函数，就可以



随意设置 DAC 通道 1 的输出电压了。

接下来我们看看 main 函数如下：

```
int main(void)
{
    u16 adcx;
    float temp;
    u8 t=0;
    u16 dacval=0;
    u8 key;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2
    uart_init(9600);       //串口初始化波特率为 9600
    KEY_Init();             //初始化按键程序
    LED_Init();             //LED 端口初始化
    LCD_Init();             //LCD 初始化
    usmart_dev.init(72);   //初始化 USMART
    Adc_Init();             //ADC 初始化
    Dac1_Init();            //DAC 初始化

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"Mini STM32");
    LCD_ShowString(60,70,200,16,16,"DAC TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/8/3");
    LCD_ShowString(60,130,200,16,16,"WKUP:+ KEY1:-");
    //显示提示信息
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(60,150,200,16,16,"DAC VAL:");
    LCD_ShowString(60,170,200,16,16,"DAC VOL:0.000V");
    LCD_ShowString(60,190,200,16,16,"ADC VOL:0.000V");

    DAC_SetChannel1Data(DAC_Align_12b_R, 0);      //初始值为 0
    while(1)
    {
        t++;
        key=KEY_Scan(0);
        if(key==4)
        {
            if(dacval<4000)dacval+=200;
            DAC_SetChannel1Data(DAC_Align_12b_R, dacval); //设置 DAC 值
        }else if(key==2)
        {
            if(dacval>200)dacval-=200;
        }
    }
}
```



```
    else dacval=0;
    DAC_SetChannel1Data(DAC_Align_12b_R, dacval); //设置 DAC 值
}
if(t==10||key==2||key==4)      //WKUP/KEY1 按下了,或者定时时间到了
{
    adcx=DAC_GetDataOutputValue(DAC_Channel_1); //读取前面设置 DAC 的值
    LCD_ShowxNum(124,150,adcx,4,16,0);          //显示 DAC 寄存器值
    temp=(float)adcx*(3.3/4096);                //得到 DAC 电压值
    adcx=temp;
    LCD_ShowxNum(124,170,temp,1,16,0);          //显示电压值整数部分
    temp-=adcx;
    temp*=1000;
    LCD_ShowxNum(140,170,temp,3,16,0X80);        //显示电压值的小数部分
    adcx=Get_Adc_Average(ADC_Channel_1,10);       //得到 ADC 转换值
    temp=(float)adcx*(3.3/4096);                //得到 ADC 电压值
    adcx=temp;
    LCD_ShowxNum(124,190,temp,1,16,0);          //显示电压值整数部分
    temp-=adcx;
    temp*=1000;
    LCD_ShowxNum(140,190,temp,3,16,0X80);        //显示电压值的小数部分
    LED0=!LED0;
    t=0;
}
delay_ms(10);
}
```

此部分代码，我们先对需要用到的模块进行初始化，然后显示一些提示信息，本章我们通过 WK_UP 和 KEY1（也就是上下键）来实现对 DAC 输出的幅值控制。按下 WK_UP 增加，按 KEY1 减小。同时在 LCD 上面显示 DHR12R1 寄存器的值、DAC 设计输出电压以及 ADC 采集到的 DAC 输出电压。

本章，我们还可以利用 USMART 来设置 DAC 的输出电压值，故需要将 Dac1_Set_Vol 函数加入 USMART 控制，方法前面已经有详细的介绍了，大家这里自行添加，或者直接查看我们光盘的源码。

从 main 函数代码可以看出，按键设置输出电压的时候，每次都是以 0.161V 递增或递减的，而通过 USMART 调用 Dac1_Set_Vol 函数，则可以实现任意电平输出控制（当然得在 DAC 可控范围内）。

24.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 24.4.1 所示：

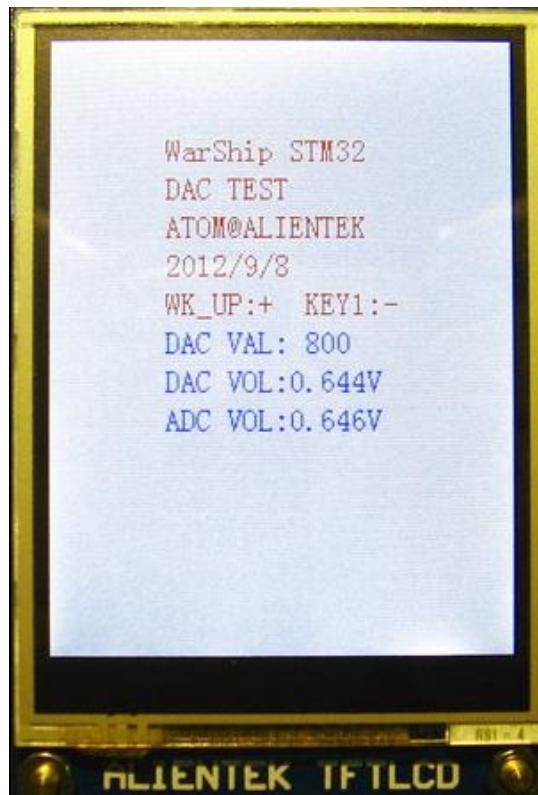


图 24.4.1 ADC 实验实际测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。此时，我们通过按 WK_UP 按键，可以看到输出电压增大，按 KEY1 则变小。

大家可以试试在 USMART 调用 Dac1_Set_Vol 函数，来设置 DAC 通道 1 的输出电压，如图 24.4.2 所示：



图 24.4.2 通过 usmart 设置 DAC 通道 1 的电压输出



第二十五章 PWM DAC 实验

上一章，我们介绍了 STM32 自带 DAC 模块的使用，但不是每个 STM32 都有 DAC 模块的，对于那些没有 DAC 模块的芯片，我们可以通过 PWM+RC 滤波来实现一个 PWM DAC。本章我们将向大家介绍如何利用 STM32 的 PWM 来设计一个 DAC。我们将利用按键（或 USMART）控制 STM32 的 PWM 输出，从而控制 PWM DAC 的输出电压，通过 ADC1 的通道 1 采集 PWM DAC 的输出电压，并在 LCD 模块上面显示 ADC 获取到的电压值以及 PWM DAC 的设定输出电压值等信息。本章将分为如下几个部分：

- 25.1 PWM DAC 简介
- 25.2 硬件设计
- 25.3 软件设计
- 25.4 下载验证



25.1 PWM DAC 简介

虽然大容量的 STM32F103 具有内部 DAC，但是更多的型号是没有 DAC 的，不过 STM32 所有的芯片都有 PWM 输出，因此，我们可以用 PWM+简单的 RC 滤波来实现 DAC 输出，从而节省成本。

PWM 本质上其实是一种周期一定，而高低电平占空比可调的方波。实际电路的典型 PWM 波形，如图 25.1.1 所示：

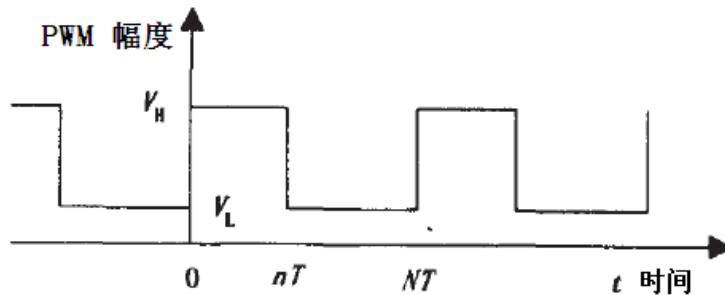


图 25.1.1 实际电路典型 PWM 波形

图 25.1.1 的 PWM 波形可以用分段函数表示为式①：

$$f(t) = \begin{cases} V_H & kNT \leq t \leq nT + kNT \\ V_L & kNT + nT \leq t \leq NT + kNT \end{cases} \quad ①$$

其中：T 是单片机中计数脉冲的基本周期，也就是 STM32 定时器的计数频率的倒数。N 是 PWM 波一个周期的计数脉冲个数，也就是 STM32 的 ARR-1 的值。n 是 PWM 波一个周期中高电平的计数脉冲个数，也就是 STM32 的 CCRx 的值。VH 和 VL 分别是 PWM 波的高低电平电压值，k 为谐波次数，t 为时间。我们将①式展开成傅里叶级数，得到公式②：

$$\begin{aligned} f(t) = & \left[\frac{n}{N} (V_H - V_L) + V_L \right] + \\ & 2 \frac{V_H - V_L}{\pi} \sin\left(\frac{n}{N}\pi\right) \cos\left(\frac{2\pi}{NT}t - \frac{n\pi}{N}k\right) + \quad ② \\ & \sum_{k=2}^{\infty} 2 \frac{V_H - V_L}{k\pi} \left| \sin\left(\frac{n\pi}{N}k\right) \right| \cos\left(\frac{2\pi}{NT}kt - \frac{n\pi}{N}k\right) \end{aligned}$$

从②式可以看出，式中第 1 个方括弧为直流分量，第 2 项为 1 次谐波分量，第 3 项为大于 1 次的高次谐波分量。式②中的直流分量与 n 成线性关系，并随着 n 从 0 到 N，直流分量从 VL 到 VL+VH 之间变化。这正是电压输出的 DAC 所需要的。因此，如果能把式②中除直流分量外的谐波过滤掉，则可以得到从 PWM 波到电压输出 DAC 的转换，即：PWM 波可以通过一个低通滤波器进行解调。式②中的第 2 项的幅度和相角与 n 有关，频率为 1/(NT)，其实就是 PWM 的输出频率。该频率是设计低通滤波器的依据。如果能把 1 次谐波很好过滤掉，则高次谐波就应该基本不存在了。

通过上面的了解，我们可以得到 PWM DAC 的分辨率，计算公式如下：

$$\text{分辨率} = \log_2(N)$$

这里假设 n 的最小变化为 1，当 N=256 的时候，分辨率就是 8 位。而 STM32 的定时器都是 16 位的，可以很容易得到更高的分辨率，分辨率越高，速度就越慢。不过我们在本章要设计的 DAC 分辨率为 8 位。

在 8 位分辨条件下，我们一般要求 1 次谐波对输出电压的影响不要超过 1 个位的精度，也就是 $3.3/256=0.01289V$ 。假设 VH 为 3.3V，VL 为 0V，那么一次谐波的最大值是 $2*3.3/\pi=2.1V$ ，这就要求我们的 RC 滤波电路提供至少 $-20\log(2.1/0.01289)=-44dB$ 的衰减。

STM32 的定时器最快的计数频率是 72Mhz，8 为分辨率的时候，PWM 频率为 $72M/256=281.25Khz$ 。如果是 1 阶 RC 滤波，则要求截止频率为 $1.77Khz$ ，如果为 2 阶 RC 滤波，则要求截止频率为 $22.34Khz$ 。

战舰 STM32 开发板的 PWM DAC 输出采用二阶 RC 滤波，该部分原理图如图 25.1.2 所示：

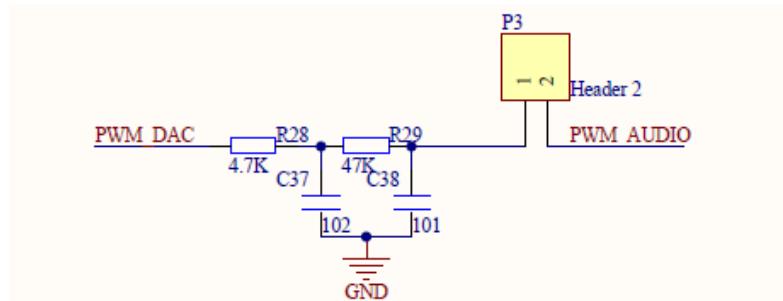


图 25.1.2 PWM DAC 二阶 RC 滤波原理图

二阶 RC 滤波截止频率计算公式为：

$$f = \frac{1}{2} \pi RC$$

以上公式要求 $R_1=R_2=R$, $C_2=C_1=C$ 。根据这个公式，我们计算出图 25.1.2 的截止频率为：33.8Khz 超过了 22.34Khz，这个和我们前面提到的要求有点出入，原因是该电路我们还需要用作 PWM DAC 音频输出，而音频信号带宽是 22.05Khz，为了让音频信号能够通过该低通滤波，同时为了标准化参数选取，所以确定了这样的参数。实测精度在 0.5LSB 以内。

PWM DAC 的原理部分，就为大家介绍到这里。

25.2 硬件设计

本章用到的硬件资源有：

- 1) 指示灯 DS0
- 2) WK_UP 和 KEY1 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) ADC
- 6) PWM DAC

本章，我们使用 STM32 的 TIM4_CH1(PB6)输出 PWM，经过二阶 RC 滤波后，转换为直流输出，实现 PWM DAC。同上一章一样，我们通过 ADC1 的通道 1(PA1)读取 PWM DAC 的输出，并在 LCD 模块上显示相关数值，通过按键和 USMART 控制 PWM DAC 的输出值。我们需要用到 ADC 采集 DAC 的输出电压，所以需要在硬件上将 PWM DAC 和 ADC 短接起来，PWM DAC 部分原理图如图 25.2.1 所示：

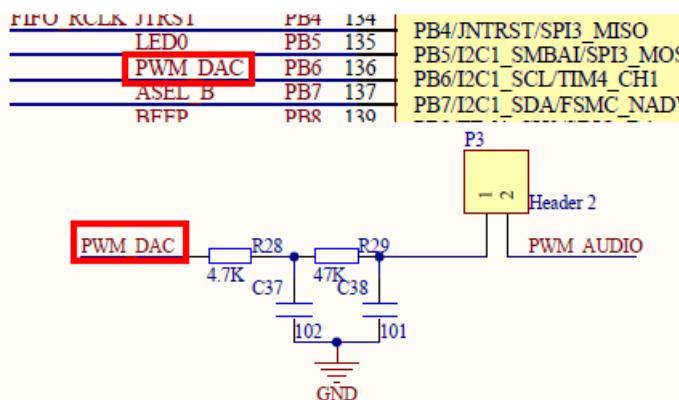


图 25.2.1 PWM DAC 原理图

在硬件上，我们还需要用跳线帽短接多功能端口的 PDC 和 ADC，如图 25.2.2 所示：

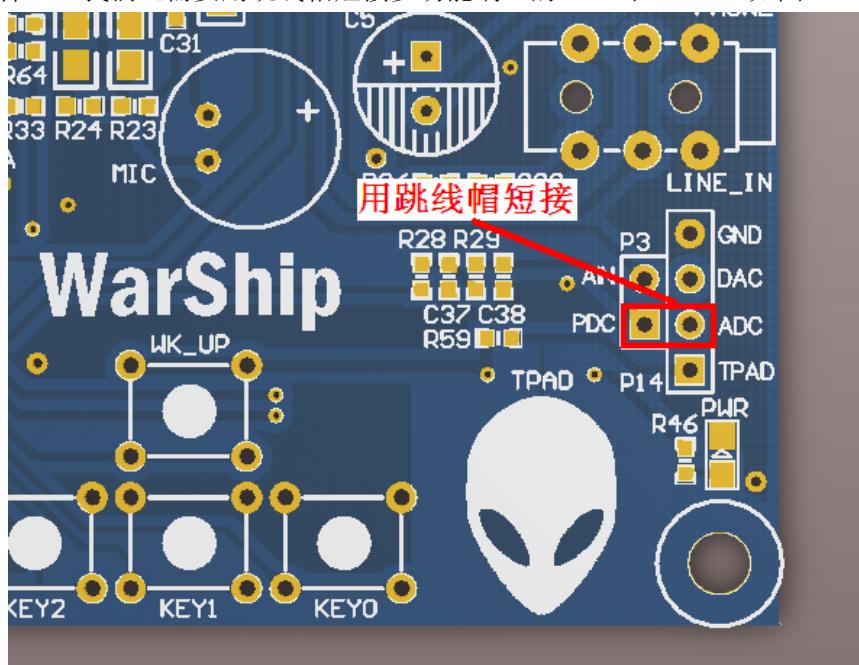


图 25.2.2 硬件连接示意图

25.3 软件设计

找到 PWM DAC 实验工程，打开 timer.c 文件，可以看到我们在文件的最后添加了一个新的函数：TIM4_PWM_Init，该函数代码如下：

```
//TIM4 CH1 PWM 输出设置
//PWM 输出初始化
//arr: 自动重装值
//psc: 时钟预分频数
void TIM4_PWM_Init(u16 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;
```



```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE); //使能 TIMx 外设
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //使能 PB 时钟

//设置该引脚为复用输出功能,输出 TIM4 CH1 的 PWM 脉冲波形
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6; //TIM_CH1
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用功能输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化 GPIO

TIM_TimeBaseStructure.TIM_Period = arr; //设置自动重装载周期值
TIM_TimeBaseStructure.TIM_Prescaler = psc; //设置预分频值 不分频
TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDTs = Tck_tim
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //T 向上计数
TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure); //初始化 TIMx

TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2; //CH1 PWM2 模式
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
TIM_OCInitStructure.TIM_Pulse = 0; //设置待装入捕获比较寄存器的脉冲值
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //OC1 低电平有效
TIM_OC1Init(TIM4, &TIM_OCInitStructure); //根据指定的参数初始化外设 TIMx

TIM_OC1PreloadConfig(TIM4, TIM_OCPreload_Enable); //CH1 预装载使能
TIM_ARRPreloadConfig(TIM4, ENABLE); //使能 TIMx 在 ARR 上的预装载寄存器
TIM_Cmd(TIM4, ENABLE); //使能 TIMx
```

}

该函数用来初始化 TIM4_CH1 的 PWM 输出 (PB6), 其原理同之前介绍的 PWM 输出一模一样, 只是换过一个定时器而已。这里就不细说了。

接下来我们打开 main.c 文件, 看到如下代码:

```
//设置输出电压
//vol:0~330,代表 0~3.3V
void PWM_DAC_Set(u16 vol)
{
    float temp=vol;
    temp/=100;
    temp=temp*256/3.3;
    TIM_SetCompare1(TIM4,temp);
}

int main(void)
{
    u16 adcx;
    float temp;
```



```
u8 t=0;
u16 pwmval=0;
u8 key;
delay_init();           //延时函数初始化
NVIC_Configuration(); //设置 NVIC 中断分组 2
uart_init(9600);       //串口初始化波特率为 9600
KEY_Init();             //KEY 初始化
LED_Init();             //LED 端口初始化
LCD_Init();             //LCD 初始化
Adc_Init();             //ADC 初始化
TIM4_PWM_Init(256,0);  //TIM4 PWM 初始化, Fpwm=72M/256=281.25Khz.
TIM_SetCompare1(TIM4,100); //初始值为 0

POINT_COLOR=RED;        //设置字体为红色
LCD_ShowString(60,50,200,16,16,"Mini STM32");
LCD_ShowString(60,70,200,16,16,"PWM DAC TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/8/3");
LCD_ShowString(60,130,200,16,16,"WKUP:+ KEY1:-");
//显示提示信息
POINT_COLOR=BLUE;       //设置字体为蓝色
LCD_ShowString(60,150,200,16,16,"PWM VAL:");
LCD_ShowString(60,170,200,16,16,"DAC VOL:0.000V");
LCD_ShowString(60,190,200,16,16,"ADC VOL:0.000V");
TIM_SetCompare1(TIM4,pwmval);//初始值
while(1)
{
    t++;
    key=KEY_Scan(0);
    if(key==4)
    {
        if(pwmval<250)pwmval+=10;
        TIM_SetCompare1(TIM4,pwmval);      //输出
    }else if(key==2)
    {
        if(pwmval>10)pwmval-=10;
        else pwmval=0;
        TIM_SetCompare1(TIM4,pwmval);      //输出
    }
    if(t==10||key==2||key==4) //WKUP/KEY1 按下了,或者定时时间到了
    {
        adcx=TIM_GetCapture1(TIM4);
        LCD_ShowxNum(124,150,adcx,4,16,0); //显示 DAC 寄存器值
        temp=(float)adcx*(3.3/256);          //得到 DAC 电压值
        adcx=temp;
        LCD_ShowxNum(124,170,temp,1,16,0); //显示电压值整数部分
    }
}
```



```
temp=adcx;
temp*=1000;
LCD_ShowxNum(140,170,temp,3,16,0x80); //显示电压值的小数部分
adcx=Get_Adc_Average(ADC_Channel_1,20); //得到 ADC 转换值
temp=(float)adcx*(3.3/4096); //得到 ADC 电压值
adcx=temp;
LCD_ShowxNum(124,190,temp,1,16,0); //显示电压值整数部分
temp=adcx;
temp*=1000;
LCD_ShowxNum(140,190,temp,3,16,0x80); //显示电压值的小数部分
t=0;
LED0=!LED0;
}
delay_ms(10);
}
```

此部分代码，同上一章的基本一样，先对需要用到的模块进行初始化，然后显示一些提示信息，本章我们通过 WK_UP 和 KEY1（也就是上下键）来实现对 PWM 脉宽的控制，经过 RC 滤波，最终实现对 DAC 输出幅值的控制。按下 WK_UP 增加，按 KEY1 减小。同时在 LCD 上面显示 TIM4_CCR1 寄存器的值、PWM DAC 设计输出电压以及 ADC 采集到的实际输出电压。同时 DS0 闪烁，提示程序运行状况。

不过此部分代码还有一个 PWM_DAC_Set 函数，用于 USMART 调用，从而通过串口控制 PWM DAC 的输出，所以还需要将 PWM_DAC_Set 函数加入 USMART 控制，方法前面已经有详细的介绍了，大家这里自行添加，或者直接查看我们光盘的源码。

25.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 25.4.1 所示：

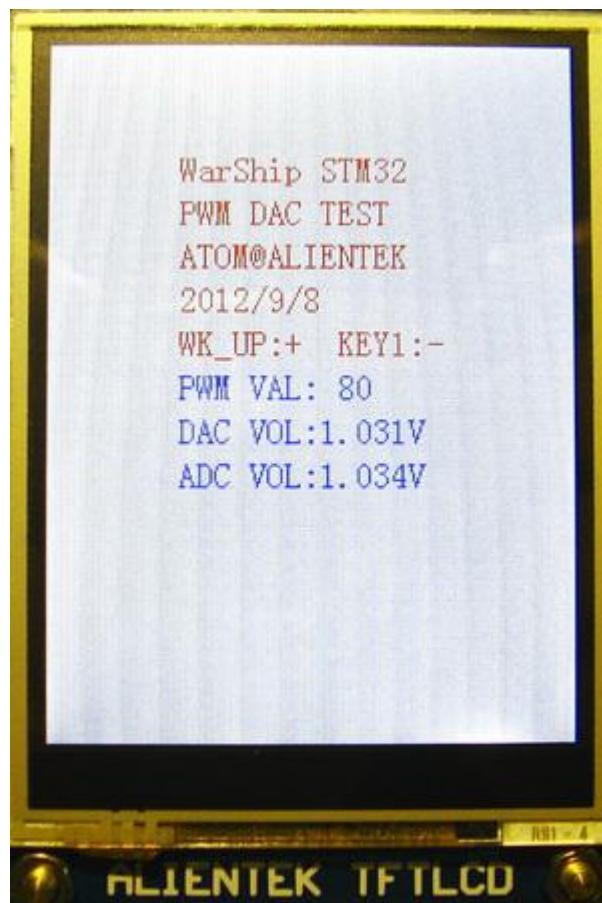


图 25.4.1 ADC 实验实际测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。此时，我们通过按 WK_UP 按键，可以看到输出电压增大，按 KEY1 则变小。

大家可以试试在 USMART 调用 PWM_DAC_Set 函数，来设置 PWM DAC 的输出电压，如图 25.4.2 所示：



图 25.4.2 通过 usmart 设置 PWM DAC 的电压输出

第二十六章 DMA 实验

本章我们将向大家介绍 STM32 的 DMA。在本章中，我们将利用 STM32 的 DMA 来实现串口数据传送，并在 TFTLCD 模块上显示当前的传送进度。本章分为如下几个部分：

26.1 STM32 DMA 简介

26.2 硬件设计

26.3 软件设计

26.4 下载验证



26.1 STM32 DMA 简介

DMA，全称为：Direct Memory Access，即直接存储器访问，DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器 来实行和完成。典型的例子就是移动一个外部内存的区块到芯片内部更快的内存区。像是这样的操作并没有让处理器工作拖延，反而可以被重新排程去处理其他的工作。DMA 传输对于高效能嵌入式系统算法和网络是很重要的。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，能使 CPU 的效率大为提高。

STM32 最多有 2 个 DMA 控制器 (DMA2 仅存在大容量产品中)，DMA1 有 7 个通道。DMA2 有 5 个通道。每个通道专门用来管理来自于一个或多个外设对存储器访问的请求。还有一个仲裁起来协调各个 DMA 请求的优先权。

STM32 的 DMA 有以下一些特性：

- 每个通道都直接连接专用的硬件 DMA 请求，每个通道都同样支持软件触发。这些功能通过软件来配置。
- 在七个请求间的优先权可以通过软件编程设置(共有四级：很高、高、中等和低)，假如在相等优先权时由硬件决定(请求 0 优先于请求 1，依此类推)。
- 独立的源和目标数据区的传输宽度(字节、半字、全字)，模拟打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。
- 支持循环的缓冲器管理
- 每个通道都有 3 个事件标志(DMA 半传输，DMA 传输完成和 DMA 传输出错)，这 3 个事件标志逻辑或成为一个单独的中断请求。
- 存储器和存储器间的传输
- 外设和存储器，存储器和外设的传输
- 闪存、SRAM、外设的 SRAM、APB1 APB2 和 AHB 外设均可作为访问的源和目标。
- 可编程的数据传输数目：最大为 65536

STM32F103ZET6 有两个 DMA 控制器，DMA1 和 DMA2，本章，我们仅针对 DMA1 进行介绍。

从外设 (TIMx、ADC、SPIx、I2Cx 和 USARTx) 产生的 DMA 请求，通过逻辑或输入到 DMA 控制器，这就意味着同时只能有一个请求有效。外设的 DMA 请求，可以通过设置相应的外设寄存器中的控制位，被独立地开启或关闭。

表 26.1.1 是 DMA1 各通道一览表：

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC	ADC1						
SPI		SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I ² C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP



表 26.1.1 DMA1 个通道一览表

这里解释一下上面说的逻辑或，例如通道 1 的几个 DMA1 请求(ADC1、TIM2_CH3、TIM4_CH1)，这几个是通过逻辑或到通道 1 的，这样我们在同一时间，就只能使用其中的一个。其他通道也是类似的。

这里我们要使用的是串口 1 的 DMA 传送，也就是要用到通道 4。接下来，我们介绍一下 DMA 设置相关的几个寄存器。

第一个是 DMA 中断状态寄存器 (DMA_ISR)。该寄存器的各位描述如图 26.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留			TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5	
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
TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
位31:28		保留，始终读为0。													
位27, 23, 19, 15, 11, 7, 3		TEIFx: 通道x的传输错误标志(x = 1 ... 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有传输错误(TE); 1: 在通道x发生了传输错误(TE)。													
位26, 22, 18, 14, 10, 6, 2		HTIFx: 通道x的半传输标志(x = 1 ... 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有半传输事件(HT); 1: 在通道x产生了半传输事件(HT)。													
位25, 21, 17, 13, 9, 5, 1		TCIFx: 通道x的传输完成标志(x = 1 ... 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有传输完成事件(TC); 1: 在通道x产生了传输完成事件(TC)。													
位24, 20, 16, 12, 8, 4, 0		GIFx: 通道x的全局中断标志(x = 1 ... 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有TE、HT或TC事件; 1: 在通道x产生了TE、HT或TC事件。													

图 26.1.1 DMA_ISR 寄存器各位描述

我们如果开启了 DMA_ISR 中这些中断，在达到条件后就会跳到中断服务函数里面去，即使没开启，我们也可以通过查询这些位来获得当前 DMA 传输的状态。这里我们常用的是 TCIFx，即通道 DMA 传输完成与否的标志。注意此寄存器为只读寄存器，所以在这些位被置位之后，只能通过其他的操作来清除。

第二个是 DMA 中断标志清除寄存器 (DMA_IFCR)。该寄存器的各位描述如图 26.1.2 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				CTEIF	CHTIF	CTCIF	CGIF	CTEIF	CHTIF	CTCIF	CGIF	CTEIF	CHTIF	CTCIF	CGIF
				7	7	7	7	6	6	6	6	5	5	5	5
				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
CTEIF	CHTIF	CTCIF	CGIF	CTEIF	CHTIF	CTCIF	CGIF	CTEIF	CHTIF	CTCIF	CGIF	CTEIF	CHTIF	CTCIF	CGIF
4	4	4	4	3	3	3	3	2	2	2	2	1	1	1	1
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位31:28		保留，始终读为0。													
位27, 23, 19, 15, 11, 7, 3		CTEIFx: 清除通道x的传输错误标志(x = 1 ... 7) 这些位由软件设置和清除。 0: 不起作用 1: 清除DMA_ISR寄存器中的对应TEIF标志。													
位26, 22, 18, 14, 10, 6, 2		CHTIFx: 清除通道x的半传输标志(x = 1 ... 7) 这些位由软件设置和清除。 0: 不起作用 0: 清除DMA_ISR寄存器中的对应HTIF标志。													
位25, 21, 17, 13, 9, 5, 1		CTCIFx: 清除通道x的传输完成标志(x = 1 ... 7) 这些位由软件设置和清除。 0: 不起作用 0: 清除DMA_ISR寄存器中的对应TCIF标志。													
位24, 20, 16, 12, 8, 4, 0		CGIFx: 清除通道x的全局中断标志(x = 1 ... 7) 这些位由软件设置和清除。 0: 不起作用 0: 清除DMA_ISR寄存器中的对应的GIF、TEIF、HTIF和TCIF标志。													

图 26.1.2 DMA_IFCR 寄存器各位描述

DMA_IFCR 的各位就是用来清除 DMA_ISR 的对应位的，通过写 0 清除。在 DMA_ISR 被置位后，我们必须通过向该位寄存器对应的位写入 0 来清除。

第三个是 DMA 通道 x 配置寄存器 (DMA_CCRx) (x=1~7, 下同)。该寄存器的我们在这里就不贴出来了，见《STM32 参考手册》第 150 页 10.4.3 一节。该寄存器控制着 DMA 的很多相关信息，包括数据宽度、外设及存储器的宽度、通道优先级、增量模式、传输方向、中断允许、使能等都是通过该寄存器来设置的。所以 DMA_CCRx 是 DMA 传输的核心控制寄存器。

第四个是 DMA 通道 x 传输数据量寄存器 (DMA_CNDTRx)。这个寄存器控制 DMA 通道 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值会随着传输的进行而减少，当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成了。所以可以通过这个寄存器的值来知道当前 DMA 传输的进度。

第五个是 DMA 通道 x 的外设地址寄存器 (DMA_CPARx)。该寄存器用来存储 STM32 外设的地址，比如我们使用串口 1，那么该寄存器必须写入 0x40013804 (其实就是&USART1_DR)。如果使用其他外设，就修改成相应外设的地址就行了。

最后一个也是 DMA 通道 x 的存储器地址寄存器 (DMA_CMARx)，该寄存器和 DMA_CPARx 差不多，但是是用来放存储器的地址的。比如我们使用 SendBuf[5200] 数组来做存储器，那么我们在 DMA_CMARx 中写入&SendBuf 就可以了。

DMA 相关寄存器就为大家介绍到这里，此节我们要用到串口 1 的发送，属于 DMA1 的通道 4 (表 26.1.1)，接下来我们就介绍库函数下 DMA1 通道 4 的配置步骤：

1) 使能 DMA 时钟

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //使能 DMA 时钟
```

2) 初始化 DMA 通道 4 参数



前面讲解过，DMA 通道配置参数种类比较繁多，包括内存地址，外设地址，传输数据长度，数据宽度，通道优先级等等。这些参数的配置在库函数中都是在函数 DMA_Init 中完成，下面我们就看看函数定义：

```
void DMA_Init(DMA_Channel_TypeDef* DMAy_Channelx,  
              DMA_InitTypeDef* DMA_InitStruct)
```

函数的第一个参数是指定初始化的 DMA 通道号，这个很容易理解，下面我们主要看看第二个参数。跟其他外设一样，同样是通过初始化结构体成员变量值来达到初始化的目的，下面我们来看看 DMA_InitTypeDef 结构体的定义：

```
typedef struct  
{  
    uint32_t DMA_PeripheralBaseAddr;  
    uint32_t DMA_MemoryBaseAddr;  
    uint32_t DMA_DIR;  
    uint32_t DMA_BufferSize;  
    uint32_t DMA_PeripheralInc;  
    uint32_t DMA_MemoryInc;  
    uint32_t DMA_PeripheralDataSize;  
    uint32_t DMA_MemoryDataSize;  
    uint32_t DMA_Mode;  
    uint32_t DMA_Priority;  
    uint32_t DMA_M2M;  
} DMA_InitTypeDef;
```

这个结构体的成员比较多，但是每个成员变量的意义我们在前面基本都已经讲解过，这里我们一一做个简要的介绍。

第一个参数 DMA_PeripheralBaseAddr 用来设置 DMA 传输的外设地址，比如要进行串口 DMA 传输，那么外设地址为串口接受发送数据存储器 USART1->DR 的地址，表示方法为 &USART1->DR。

第二个参数 DMA_MemoryBaseAddr 为内存地址，也就是我们存放 DMA 传输数据的内存地址。

第三个参数 DMA_DIR 设置数据传输方向，决定是从外设读取数据到内存还是从内存读取数据发送到外设，也就是外设是源地还是目的地，这里我们设置为从内存读取数据发送到串口，所以外设自然就是目的地了，所以选择值为 DMA_DIR_PeripheralDST。

第四个参数 DMA_BufferSize 设置一次传输数据量的大小，这个很容易理解。

第五个参数 DMA_PeripheralInc 设置传输数据的时候外设地址是不变还是递增。如果设置为递增，那么下一次传输的时候地址加 1，这里因为我们是一直往固定外设地址 &USART1->DR 发送数据，所以地址不递增，值为 DMA_PeripheralInc_Disable；

第六个参数 DMA_MemoryInc 设置传输数据时候内存地址是否递增。这个参数和 DMA_PeripheralInc 意思接近，只不过针对的是内存。这里我们的场景是将内存中连续存储单元的数据发送到串口，毫无疑问内存地址是需要递增的，所以值为 DMA_MemoryInc_Enable。

第七个参数 DMA_PeripheralDataSize 用来设置外设的数据长度是为字节传输(8bits)，半字传输(16bits)还是字传输(32bits)，这里我们是 8 位字节传输，所以值设置为 DMA_PeripheralDataSize_Byte。

第八个参数 DMA_MemoryDataSize 是用来设置内存的数据长度，和第七个参数意思接近，这里我们同样设置为字节传输 DMA_MemoryDataSize_Byte。



第九个参数 DMA_Mode 用来设置 DMA 模式是否循环采集，也就是说，比如我们要从内存中采集 64 个字节发送到串口，如果设置为重复采集，那么它会在 64 个字节采集完成之后继续从内存的第一个地址采集，如此循环。这里我们设置为一次连续采集完成之后不循环。所以设置值为 DMA_Mode_Normal。在我们下面的实验中，如果设置此参数为循环采集，那么你会看到串口不停的打印数据，不会中断，大家在实验中可以修改这个参数测试一下。

第十个参数是设置 DMA 通道的优先级，有低，中，高，超高等三种模式，这个在前面讲解过，这里我们设置优先级别为中级，所以值为 DMA_Priority_Medium。如果要开启多个通道，那么这个值就非常有意义。

第十一个参数 DMA_M2M 设置是否是存储器到存储器模式传输，这里我们选择 DMA_M2M_Disable。

这里我们给出上面场景的实例代码：

```
DMA_InitTypeDef DMA_InitStructure;
DMA_InitStructure.DMA_PeripheralBaseAddr = &USART1->DR; //DMA 外设 ADC 基址
DMA_InitStructure.DMA_MemoryBaseAddr = cmar; //DMA 内存基地址
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST; //从内存读取发送到外设
DMA_InitStructure.DMA_BufferSize = 64; //DMA 通道的 DMA 缓存的大小
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //外设地址不变
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; //内存地址递增
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte; //8 位
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte; //8 位
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal; //工作在正常缓存模式
DMA_InitStructure.DMA_Priority = DMA_Priority_Medium; //DMA 通道 x 拥有中优先级
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; //非内存到内存传输
DMA_Init(DMA_CHx, &DMA_InitStructure); //根据指定的参数初始化
```

3) 使能串口 DMA 发送

进行 DMA 配置之后，我们就要开启串口的 DMA 发送功能，使用的函数是：

```
USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);
```

如果是要使能串口 DMA 接受，那么第二个参数修改为 USART_DMAReq_Rx 即可。

4) 使能 DMA1 通道 4，启动传输。

使能串口 DMA 发送之后，我们接着就要使能 DMA 传输通道：

```
DMA_Cmd(DMA_CHx, ENABLE);
```

通过以上 3 步设置，我们就可以启动一次 USART1 的 DMA 传输了。

5) 查询 DMA 传输状态

在 DMA 传输过程中，我们要查询 DMA 传输通道的状态，使用的函数是：

```
FlagStatus DMA_GetFlagStatus(uint32_t DMAy_FLAG)
```

比如我们要查询 DMA 通道 4 传输是否完成，方法是：

```
DMA_GetFlagStatus(DMA2_FLAG_TC4);
```

这里还有一个比较重要的函数就是获取当前剩余数据量大小的函数：

```
uint16_t DMA_GetCurrDataCounter(DMA_Channel_TypeDef* DMAy_Channelx)
```

比如我们要获取 DMA 通道 4 还有多少个数据没有传输，方法是：

```
DMA_GetCurrDataCounter(DMA1_Channel4);
```

DMA 相关的库函数我们就讲解到这里，大家可以查看固件库中文手册详细了解。



26.2 硬件设计

所以本章用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY0 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) DMA

本章我们将利用外部按键 KEY0 来控制 DMA 的传送，每按一次 KEY0，DMA 就传送一次数据到 USART1，然后在 TFTLCD 模块上显示进度等信息。DS0 还是用来做为程序运行的指示灯。

本章实验需要注意 P6 口的 RXD 和 TXD 是否和 PA9 和 PA10 连接上，如果没有，请先连接。

26.3 软件设计

打开我们的 DMA 传输实验，可以发现，我们的实验中多了 dma.c 文件和其头文件 dma.h，同时我们要引入 dma 相关的库函数文件 stm32f10x_dma.c 和 stm32f10x_dma.h。

打开 dma.c 文件，代码如下：

```
#include "dma.h"  
  
DMA_InitTypeDef DMA_InitStructure;  
u16 DMA1_MEM_LEN;//保存 DMA 每次数据传送的长度  
//DMA1 的各通道配置  
//这里的传输形式是固定的,这点要根据不同的情况来修改  
//从存储器->外设模式/8 位数据宽度/存储器增量模式  
//DMA_CHx:DMA 通道 CHx  
//cpar:外设地址  
//cmar:存储器地址  
//cndtr:数据传输量  
void MYDMA_Config(DMA_Channel_TypeDef* DMA_CHx,u32 cpar,u32 cmr,u16 cndtr)  
{  
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //使能 DMA 时钟  
    DMA_DeInit(DMA_CHx); //将 DMA 的通道 1 寄存器重设为缺省值  
    DMA1_MEM_LEN=cndtr;  
    DMA_InitStructure.DMA_PeripheralBaseAddr = cpar; //DMA 外设 ADC 基地址  
    DMA_InitStructure.DMA_MemoryBaseAddr = cmr; //DMA 内存基地址  
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST; //数据传输方向内存到外设  
    DMA_InitStructure.DMA_BufferSize = cndtr; //DMA 通道的 DMA 缓存的大小  
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //外设地址不变  
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; //内存地址寄存器递增  
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte; //数据宽度为 8 位  
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte; //数据宽度  
                                //为 8 位  
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal; //工作在正常缓存模式  
    DMA_InitStructure.DMA_Priority = DMA_Priority_Medium; //DM 通道拥有中优先级
```



```
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;      //非内存到内存传输
DMA_Init(DMA_CHx, &DMA_InitStructure);           //初始化 DMA 的通道
}
//开启一次 DMA 传输
void MYDMA_Enable(DMA_Channel_TypeDef*DMA_CHx)
{
    DMA_Cmd(DMA_CHx, DISABLE ); //关闭 USART1 TX DMA1 所指示的通道
    DMA_SetCurrDataCounter(DMA1_Channel4,DMA1_MEM_LEN); //设置 DMA 缓存的大小
    DMA_Cmd(DMA_CHx, ENABLE); //使能 USART1 TX DMA1 所指示的通道
}
```

该部分代码仅仅 2 个函数，MYDMA_Config 函数，基本上就是按照我们上面介绍的步骤来初始化 DMA 的，该函数在外部只能修改通道、源地址、目标地址和传输数据量等几个参数，更多的其他设置只能在该函数内部修改。MYDMA_Enable 函数就是设置 DMA 缓存大小并且使能 DMA 通道。对照前面的配置步骤的详细讲解看看这部分代码即可。

最后我们看看 main 函数如下：

```
u8 SendBuff[5200];
const u8 TEXT_TO_SEND[]{"ALIENTEK Warship STM32 DMA 串口实验"};
int main(void)
{
    u16 i;
    u8 t=0;
    u8 j,mask=0;
    float pro=0;           //进度
    delay_init();          //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
    LCD_Init();            //初始化 LCD
    KEY_Init();            //按键初始化
    MYDMA_Config(DMA1_Channel4,(u32)&USART1->DR,(u32)SendBuff,5168);
                           //DMA1 通道 4,外设为串口 1,存储器为 SendBuff,长度 5168.
    POINT_COLOR=RED;       //设置字体为红色
    LCD_ShowString(60,50,200,16,16,"Mini STM32");
    LCD_ShowString(60,70,200,16,16,"DMA TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/8/3");
    LCD_ShowString(60,130,200,16,16,"KEY0:Start");
    //显示提示信息
    j=sizeof(TEXT_TO_SEND);
    for(i=0;i<5168;i++)
    {
        if(t>=j)           //填充 ASCII 字符集数据
        {
            }
```



```
if(mask)
{
    SendBuff[i]=0x0a;
    t=0;
}
else
{
    SendBuff[i]=0x0d;
    mask++;
}
}
else //复制 TEXT_TO_SEND 语句
{
    mask=0;
    SendBuff[i]=TEXT_TO_SEND[t];
    t++;
}
}
POINT_COLOR=BLUE; //设置字体为蓝色
i=0;
while(1)
{
    t=KEY_Scan(0);
    if(t==KEY_RIGHT) //KEY0 按下
    {
        LCD_ShowString(60,150,200,16,16,"Start Transimit....");
        LCD_ShowString(60,170,200,16,16,"%");//显示百分号
        printf("\r\nDMA DATA:\r\n");
        USART_DMACmd(USART1,USART_DMAReq_Tx,ENABLE);
        //使能串口 1 的 DMA 发送
        MYDMA_Enable(DMA1_Channel4); //开始一次 DMA 传输!
        //等待 DMA 传输完成, 此时我们来做另外一些事, 点灯
        //实际应用中, 传输数据期间, 可以执行另外的任务
        while(1)
        {
            if(DMA_GetFlagStatus(DMA1_FLAG_TC4)!=RESET) //判断传输完成
            {
                DMA_ClearFlag(DMA1_FLAG_TC4); //清除通道 4 传输完成标志
                break;
            }
            pro=DMA_GetCurrDataCounter(DMA1_Channel4);//得到当前剩余数据量
            pro=1-pro/5168; //得到百分比
            pro*=100; //扩大 100 倍
            LCD_ShowNum(60,170,pro,3,16);
        }
        LCD_ShowNum(60,170,100,3,16); //显示 100%
        LCD_ShowString(60,150,200,16,16,"Transimit Finished!"); //传送完成
    }
    i++;
    delay_ms(10);
    if(i==20)
```



```
{    LED0=!LED0;//提示系统正在运行
    i=0;
}
}

}
```

main 函数的流程大致是：先初始化内存 SendBuff 的值，然后通过 KEY0 开启串口 DMA 发送，在发送过程中，通过 DMA_GetCurrDataCounter() 函数获取当前还剩余的数据量来计算传输百分比，最后在传输结束之后清除相应标志位，提示已经传输完成。这里还有点要注意，因为是使用的串口 1 DMA 发送，所以代码中使用 USART_DMACmd 函数开启串口的 DMA 发送：

```
USART_DMACmd(USART1,USART_DMAReq_Tx,ENABLE); //使能串口 1 的 DMA 发送
至此，DMA 串口传输的软件设计就完成了。
```

26.4 下载验证

在代码编译成功之后，我们通过串口下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 26.4.1 所示：



图 26.4.1 DMA 串口实验实物测试图

伴随 DS0 的不停闪烁，提示程序在运行。我们打开串口调试助手，然后按 KEY0，可以看到串口显示如图 26.4.2 所示的内容：



图 26.4.2 串口收到的数据内容

上图中，接收到的数据，并没有进行换行，而实际上，我们的串口是发送了换行符的，这是SSCOM3.3串口调试助手的又一个小bug。大家可以换一个别的串口调试助手，一般都是可以收到整齐的数据的。

同时可以看到TFTLCD上显示了进度等信息，如图26.4.3所示：





26.4.3 DMA 串口数据传输中

至此，我们整个 DMA 串口实验就结束了，希望大家通过本章的学习，掌握 STM32 的 DMA 使用。DMA 是个非常好的功能，它不但能减轻 CPU 负担，还能提高数据传输速度，合理的应用 DMA，往往能让你的程序设计变得简单。



第二十七章 IIC 实验

本章我们将向大家介绍如何利用 STM32 的普通 IO 口模拟 IIC 时序，并实现和 24C02 之间的双向通信。在本章中，我们将利用 STM32 的普通 IO 口模拟 IIC 时序，来实现 24C02 的读写，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 27.1 IIC 简介
- 27.2 硬件设计
- 27.3 软件设计
- 27.4 下载验证



27.1 IIC 简介

IIC(Inter—Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，高速 IIC 总线一般可达 400kbps 以上。

I2C 总线在传送数据过程中共有三种类型信号，它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。

应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

这些信号中，起始信号是必需的，结束信号和应答信号，都可以不要。IIC 总线时序图如图 27.1.1 所示：

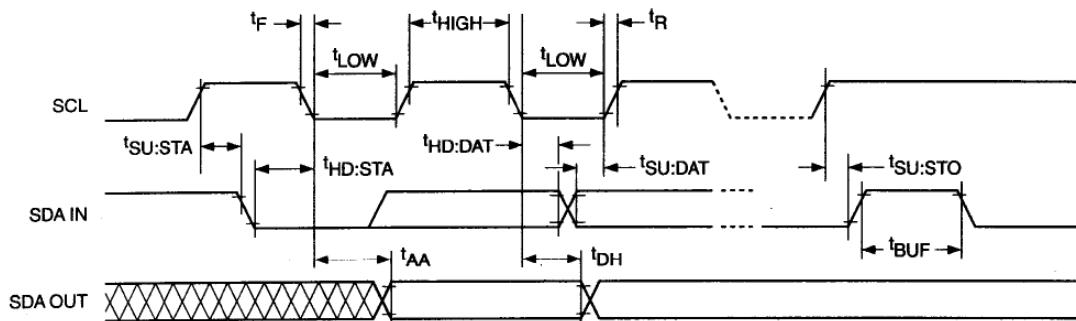


图 27.1.1 IIC 总线时序图

ALIENTEK 战舰 STM32 开发板板载的 EEPROM 芯片型号为 24C02。该芯片的总容量是 256 个字节，该芯片通过 IIC 总线与外部连接，我们本章就通过 STM32 来实现 24C02 的读写。

目前大部分 MCU 都带有 IIC 总线接口，STM32 也不例外。但是这里我们不使用 STM32 的硬件 IIC 来读写 24C02，而是通过软件模拟。STM32 的硬件 IIC 非常复杂，更重要的是不稳定，故不推荐使用。所以我们这里就通过模拟来实现了。有兴趣的读者可以研究一下 STM32 的硬件 IIC。

本章实验功能简介：开机的时候先检测 24C02 是否存在，然后在主循环里面用 1 个按键 (KEY0) 用来执行写入 24C02 的操作，另外一个按键 (WK_UP) 用来执行读出操作，在 TFTLCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

27.2 硬件设计

本章需要用到的硬件资源有：

- 1) 指示灯 DS0
- 2) WK_UP 和 KEY1 按键
- 3) 串口 (USMART 使用)
- 4) TFTLCD 模块
- 5) 24C02



前面 4 部分的资源，我们前面已经介绍了，请大家参考相关章节。这里只介绍 24C02 与 STM32 的连接，24C02 的 SCL 和 SDA 分别连在 STM32 的 PB10 和 PB11 上的，连接关系如图 27.2.1 所示：

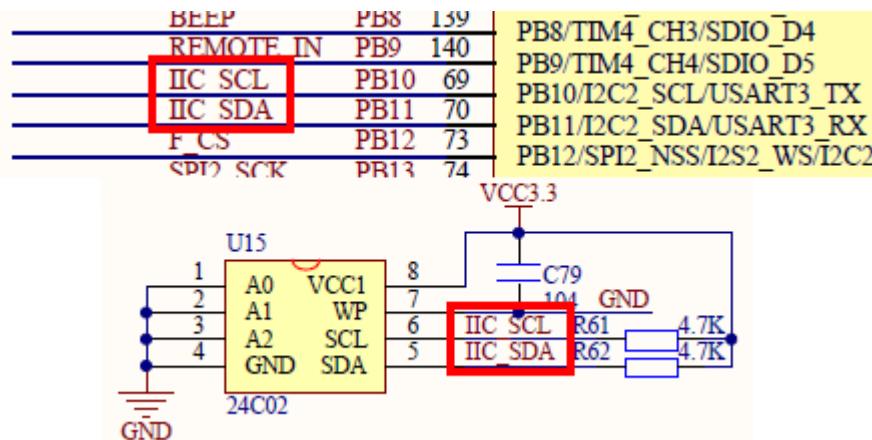


图 27.2.1 STM32 与 24C02 连接图

27.3 软件设计

打开 IIC 实验工程，我们可以看到工程中加入了两个源文件分别是 myiic.c 和 24cxx.c，myiic.c 文件存放 iic 驱动代码，24cxx.c 文件存放 24C02 驱动代码：

打开 myiic.c 文件，代码如下：

```
#include "myiic.h"
#include "delay.h"
//初始化 IIC
void IIC_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE );//PB 时钟使能
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10|GPIO_Pin_11;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;           //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);                  //初始化 GPIO
    GPIO_SetBits(GPIOB,GPIO_Pin_10|GPIO_Pin_11);            //PB10,PB11 输出高
}
//产生 IIC 起始信号
void IIC_Start(void)
{
    SDA_OUT();      //sda 线输出
    IIC_SDA=1;
    IIC_SCL=1;
    delay_us(4);
    IIC_SDA=0;     //START:when CLK is high,DATA change from high to low
    delay_us(4);
}
```



```
IIC_SCL=0; //锁住 I2C 总线，准备发送或接收数据
}

//产生 IIC 停止信号
void IIC_Stop(void)
{
    SDA_OUT(); //sda 线输出
    IIC_SCL=0;
    IIC_SDA=0; //STOP:when CLK is high DATA change form low to high
    delay_us(4);
    IIC_SCL=1;
    IIC_SDA=1; //发送 I2C 总线结束信号
    delay_us(4);
}

//等待应答信号到来
//返回值： 1， 接应收应答失败
//          0， 接收应答成功
u8 IIC_Wait_Ack(void)
{
    u8 ucErrTime=0;
    SDA_IN(); //SDA 设置为输入
    IIC_SDA=1;delay_us(1);
    IIC_SCL=1;delay_us(1);
    while(READ_SDA)
    {
        ucErrTime++;
        if(ucErrTime>250)
        {
            IIC_Stop();
            return 1;
        }
    }
    IIC_SCL=0; //时钟输出 0
    return 0;
}

//产生 ACK 应答
void IIC_Ack(void)
{
    IIC_SCL=0;
    SDA_OUT();
    IIC_SDA=0;
    delay_us(2);
    IIC_SCL=1;
    delay_us(2);
    IIC_SCL=0;
}

//不产生 ACK 应答
```



```
void IIC_NAck(void)
{
    IIC_SCL=0;
    SDA_OUT();
    IIC_SDA=1;
    delay_us(2);
    IIC_SCL=1;
    delay_us(2);
    IIC_SCL=0;
}

//IIC 发送一个字节
//返回从机有无应答
//1, 有应答
//0, 无应答
void IIC_Send_Byte(u8 txd)
{
    u8 t;
    SDA_OUT();
    IIC_SCL=0;//拉低时钟开始数据传输
    for(t=0;t<8;t++)
    {
        IIC_SDA=(txd&0x80)>>7;
        txd<<=1;
        delay_us(2); //对 TEA5767 这三个延时都是必须的
        IIC_SCL=1;
        delay_us(2);
        IIC_SCL=0;
        delay_us(2);
    }
}

//读 1 个字节, ack=1 时, 发送 ACK, ack=0, 发送 nACK
u8 IIC_Read_Byte(unsigned char ack)
{
    unsigned char i,receive=0;
    SDA_IN(); //SDA 设置为输入
    for(i=0;i<8;i++)
    {
        IIC_SCL=0;
        delay_us(2);
        IIC_SCL=1;
        receive<<=1;
        if(READ_SDA)receive++;
        delay_us(1);
    }
    if (!ack)
        IIC_NAck(); //发送 nACK
    else
        IIC_Ack(); //发送 ACK
}
```



```
    return receive;  
}
```

该部分为 IIC 驱动代码，实现包括 IIC 的初始化（IO 口）、IIC 开始、IIC 结束、ACK、IIC 读写等功能，在其他函数里面，只需要调用相关的 IIC 函数就可以和外部 IIC 器件通信了，这里并不局限于 24C02，该段代码可以用在任何 IIC 设备上。

下面我们看看头文件 myiic.h 的代码，里面有两行代码为直接通过寄存器操作设置 IO 口的模式为输入还是输出，代码如下：

```
#define SDA_IN() {GPIOB->CRH&=0xFFFF0FFF;GPIOB->CRH|=8<<12;}  
#define SDA_OUT() {GPIOB->CRH&=0xFFFF0FFF;GPIOB->CRH|=3<<12;}
```

其他部分都是一些函数申明之类的，这里不做过多解释。

接下来我们看看 24cxx.c 文件代码：

```
#include "24cxx.h"  
#include "delay.h"  
//初始化 IIC 接口  
void AT24CXX_Init(void)  
{  
    IIC_Init();  
}  
//在 AT24CXX 指定地址读出一个数据  
//ReadAddr:开始读数的地址  
//返回值 :读到的数据  
u8 AT24CXX_ReadOneByte(u16 ReadAddr)  
{  
    u8 temp=0;  
    IIC_Start();  
    if(EE_TYPE>AT24C16)  
    {  
        IIC_Send_Byte(0XA0);  
        //发送写命令  
        IIC_Wait_Ack();  
        IIC_Send_Byte(ReadAddr>>8);  
        //发送高地址  
    }else IIC_Send_Byte(0XA0+((ReadAddr/256)<<1));  
    //发送器件地址 0XA0,写数据  
    IIC_Wait_Ack();  
    IIC_Send_Byte(ReadAddr%256);  
    //发送低地址  
    IIC_Wait_Ack();  
    IIC_Start();  
    IIC_Send_Byte(0XA1);  
    //进入接收模式  
    IIC_Wait_Ack();  
    temp=IIC_Read_Byte(0);  
    IIC_Stop();  
    //产生一个停止条件  
    return temp;  
}  
//在 AT24CXX 指定地址写入一个数据  
//WriteAddr :写入数据的目的地址
```



```
//DataToWrite:要写入的数据
void AT24CXX_WriteOneByte(u16 WriteAddr,u8 DataToWrite)
{
    IIC_Start();
    if(EE_TYPE>AT24C16)
    {
        IIC_Send_Byte(0XA0);                      //发送写命令
        IIC_Wait_Ack();
        IIC_Send_Byte(WriteAddr>>8); //发送高地址
    }else IIC_Send_Byte(0XA0+((WriteAddr/256)<<1)); //发送器件地址 0XA0,写数据
    IIC_Wait_Ack();
    IIC_Send_Byte(WriteAddr%256);                //发送低地址
    IIC_Wait_Ack();
    IIC_Send_Byte(DataToWrite);                  //发送字节

    IIC_Wait_Ack();
    IIC_Stop();                                //产生一个停止条件
    delay_ms(10);
}

//在 AT24CXX 里面的指定地址开始写入长度为 Len 的数据
//该函数用于写入 16bit 或者 32bit 的数据.
//WriteAddr :开始写入的地址
//DataToWrite:数据数组首地址
//Len       :要写入数据的长度 2,4
void AT24CXX_WriteLenByte(u16 WriteAddr,u32 DataToWrite,u8 Len)
{
    u8 t;
    for(t=0;t<Len;t++)
    {
        AT24CXX_WriteOneByte(WriteAddr+t,(DataToWrite>>(8*t))&0xff);
    }
}

//在 AT24CXX 里面的指定地址开始读出长度为 Len 的数据
//该函数用于读出 16bit 或者 32bit 的数据.
//ReadAddr   :开始读出的地址
//返回值     :数据
//Len        :要读出数据的长度 2,4
u32 AT24CXX_ReadLenByte(u16 ReadAddr,u8 Len)
{
    u8 t;
    u32 temp=0;
    for(t=0;t<Len;t++)
    {
        temp<<=8;
        temp+=AT24CXX_ReadOneByte(ReadAddr+Len-t-1);
    }
    return temp;
}
```



```
}

//检查 AT24CXX 是否正常
//这里用了 24XX 的最后一个地址(255)来存储标志字.
//如果用其他 24C 系列,这个地址要修改
//返回 1:检测失败
//返回 0:检测成功
u8 AT24CXX_Check(void)
{
    u8 temp;
    temp=AT24CXX_ReadOneByte(255);      //避免每次开机都写 AT24CXX
    if(temp==0X55)return 0;
    else                                //排除第一次初始化的情况
    {
        AT24CXX_WriteOneByte(255,0X55);
        temp=AT24CXX_ReadOneByte(255);
        if(temp==0X55)return 0;
    }
    return 1;
}

//在 AT24CXX 里面的指定地址开始读出指定个数的数据
//ReadAddr :开始读出的地址 对 24c02 为 0~255
//pBuffer :数据数组首地址
//NumToRead:要读出数据的个数
void AT24CXX_Read(u16 ReadAddr,u8 *pBuffer,u16 NumToRead)
{
    while(NumToRead)
    {
        *pBuffer++=AT24CXX_ReadOneByte(ReadAddr++);
        NumToRead--;
    }
}

//在 AT24CXX 里面的指定地址开始写入指定个数的数据
//WriteAddr :开始写入的地址 对 24c02 为 0~255
//pBuffer :数据数组首地址
//NumToWrite:要写入数据的个数
void AT24CXX_Write(u16 WriteAddr,u8 *pBuffer,u16 NumToWrite)
{
    while(NumToWrite--)
    {
        AT24CXX_WriteOneByte(WriteAddr,*pBuffer);
        WriteAddr++;
        pBuffer++;
    }
}
```

这部分代码实际就是通过 IIC 接口来操作 24Cxx 芯片，理论上是可以支持 24Cxx 所有系列的芯片的（地址引脚必须都设置为 0），但是我们测试只测试了 24C02，其他器件有待测试。大家也可以验证一下，24CXX 的型号定义在 24cxx.h 文件里面，通过 EE_TYPE 设置。

最后，我们在 main 函数里面编写应用代码，main 函数如下：



```
const u8 TEXT_Buffer[]{"Warship STM32 IIC TEST "};  
#define SIZE sizeof(TEXT_Buffer)  
int main(void)  
{  
    u8 key;  
    u16 i=0;  
    u8 datatemp[SIZE];  
    delay_init();           //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级  
    uart_init(9600);        //串口初始化波特率为 9600  
    KEY_Init();  
    LED_Init();            //LED 端口初始化  
    LCD_Init();  
    AT24CXX_Init();        //IIC 初始化  
    POINT_COLOR=RED;//设置字体为红色  
    LCD_ShowString(60,50,200,16,16,"Mini STM32");  
    LCD_ShowString(60,70,200,16,16,"IIC TEST");  
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(60,110,200,16,16,"2012/8/5");  
    LCD_ShowString(60,130,200,16,16,"WKUP:Write KEY1:Read"); //显示提示信息  
    while(AT24CXX_Check()) //检测不到 24c02  
    {  
        LCD_ShowString(60,150,200,16,16,"24C02 Check Failed!");  
        delay_ms(500);  
        LCD_ShowString(60,150,200,16,16,"Please Check!");  
        delay_ms(500);  
        LED0=!LED0;          //DS0 闪烁  
    }  
    LCD_ShowString(60,150,200,16,16,"24C02 Ready!");  
    POINT_COLOR=BLUE;      //设置字体为蓝色  
    while(1)  
    {  
        key=KEY_Scan(0);  
        if(key==KEY_UP)    //KEY_UP 按下,写入 24C02  
        {  
            LCD_Fill(0,170,239,319,WHITE);           //清除半屏  
            LCD_ShowString(60,170,200,16,16,"Start Write 24C02....");  
            AT24CXX_Write(0,(u8*)TEXT_Buffer,SIZE);  
            LCD_ShowString(60,170,200,16,16,"24C02 Write Finished!"); //提示传送完成  
        }  
        if(key==KEY_DOWN)           //KEY_DOWN 按下,读取字符串并显示  
        {  
            LCD_ShowString(60,170,200,16,16,"Start Read 24C02....");  
            AT24CXX_Read(0,datatemp,SIZE);  
            LCD_ShowString(60,170,200,16,16,"The Data Readed Is: "); //提示传送完成  
            LCD_ShowString(60,190,200,16,16,datatemp); //显示读到的字符串  
        }  
    }  
}
```



```
    }
    i++;
    delay_ms(10);
    if(i==20)
    {
        LED0=!LED0; //提示系统正在运行
        i=0;
    }
}
```

该段代码，我们通过 KEY_UP (WK_UP) 按键来控制 24C02 的写入，通过另外一个按键 KEY1(KEY_DOWN)来控制 24C02 的读取。并在 LCD 模块上面显示相关信息。

至此，我们的软件设计部分就结束了。

27.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，通过先按 WK_UP 按键写入数据，然后按 KEY1 读取数据，得到如图 27.4.1 所示：

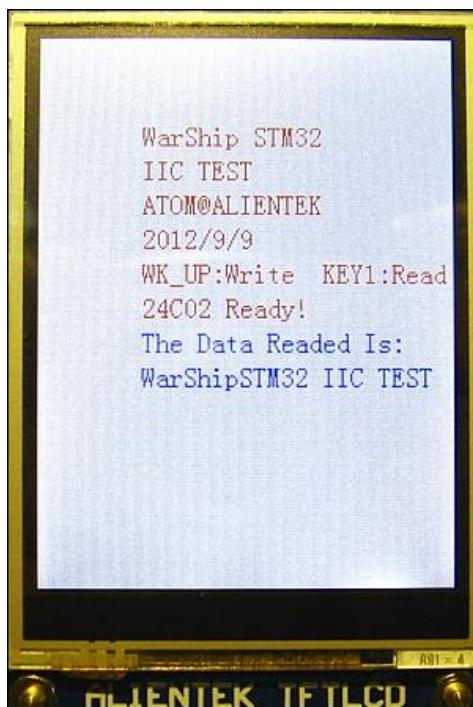


图 27.4.1 程序运行效果图

同时 DS0 会不停的闪烁，提示程序正在运行。程序在开机的时候会检测 24C02 是否存在，如果不存在则会在 TFTLCD 模块上显示错误信息，同时 DS0 慢闪。读者可以通过跳线帽把 PB10 和 PB11 短接就可以看到报错了。

我们通过在 USMART 里面加入 AT24cxx_WriteOneByte 和 AT24cxx_ReadOneByte 函数，就可以通过 USMART 读取和写入 24C02 的任何地址了。如图 27.4.2 所示：

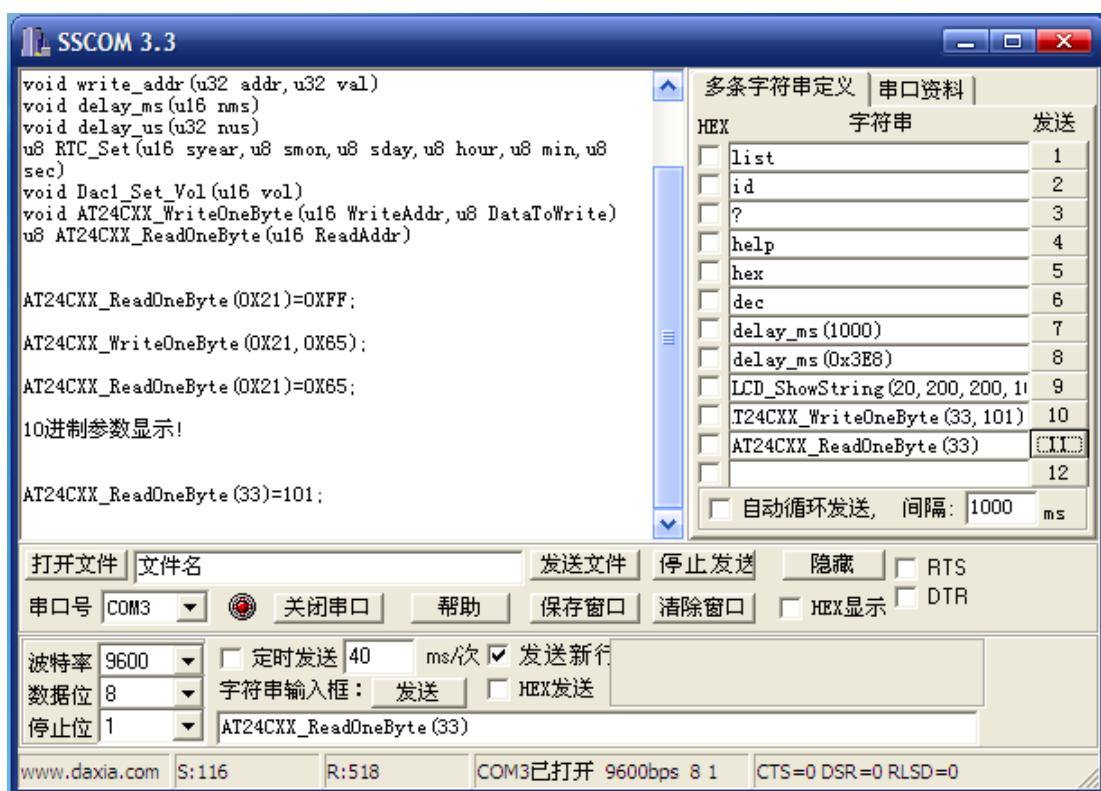


图 27.4.2 USMART 控制 24C02 读写

第二十八章 SPI 实验

本章我们将向大家介绍 STM32 的 SPI 功能。在本章中，我们将利用 STM32 自带的 SPI 来实现对外部 FLASH（W25Q64）的读写，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 28.1 SPI 简介
- 28.2 硬件设计
- 28.3 软件设计
- 28.4 下载验证



28.1 SPI 简介

SPI 是英语 Serial Peripheral interface 的缩写，顾名思义就是串行外围设备接口。是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。SPI 接口主要应用在 EEPROM, FLASH, 实时时钟, AD 转换器, 还有数字信号处理器和数字信号解码器之间。SPI, 是一种高速的, 全双工, 同步的通信总线, 并且在芯片的管脚上只占用四根线, 节约了芯片的管脚, 同时为 PCB 的布局上节省空间, 提供方便, 正是出于这种简单易用的特性, 现在越来越多的芯片集成了这种通信协议, STM32 也有 SPI 接口。下面我们看看 SPI 的内部简明图 (图 28.1.1):

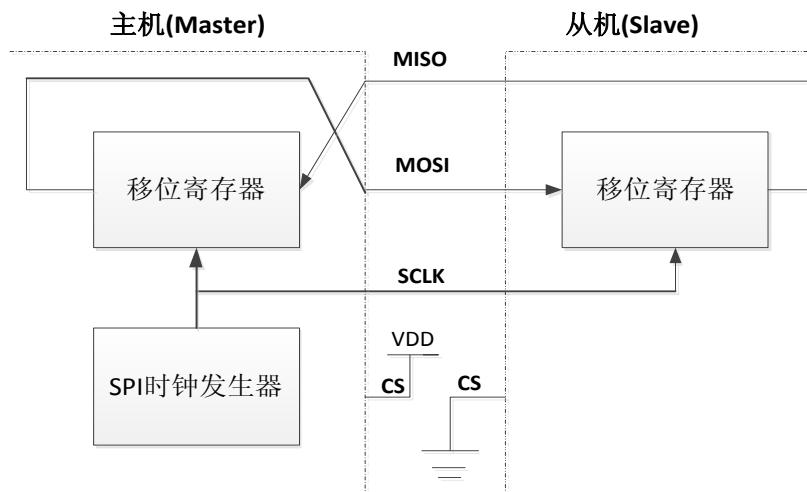


图 28.1.1 SPI 内部结构简明图

SPI 接口一般使用 4 条线通信:

MISO 主设备数据输入, 从设备数据输出。

MOSI 主设备数据输出, 从设备数据输入。

SCLK 时钟信号, 由主设备产生。

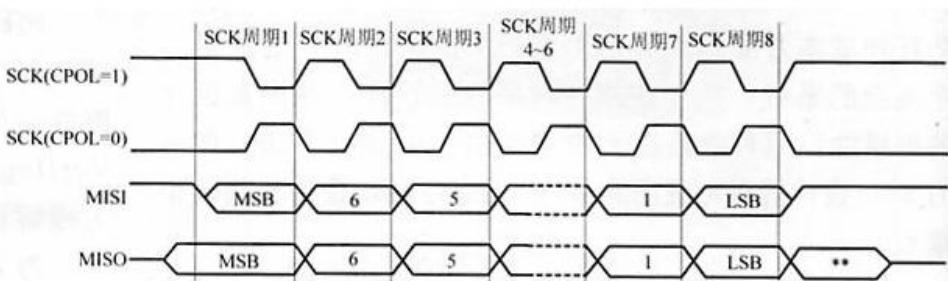
CS 从设备片选信号, 由主设备控制。

从图中可以看出, 主机和从机都有一个串行移位寄存器, 主机通过向它的 SPI 串行寄存器写入一个字节来发起一次传输。寄存器通过 MOSI 信号线将字节传送给从机, 从机也将自己的移位寄存器中的内容通过 MISO 信号线返回给主机。这样, 两个移位寄存器中的内容就被交换。外设的写操作和读操作是同步完成的。如果只进行写操作, 主机只需忽略接收到的字节; 反之, 若主机要读取从机的一个字节, 就必须发送一个空字节来引发从机的传输。

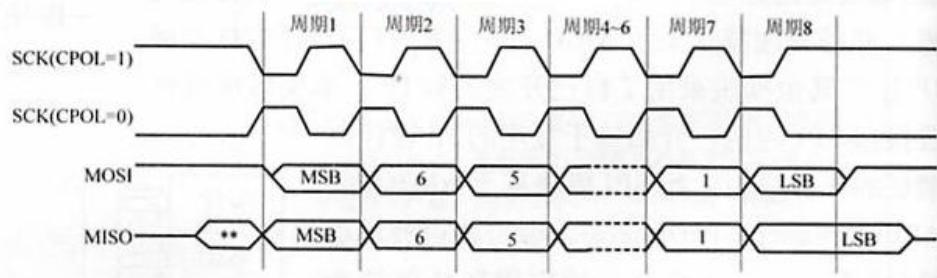
SPI 主要特点有: 可以同时发出和接收串行数据; 可以当作主机或从机工作; 提供频率可编程时钟; 发送结束中断标志; 写冲突保护; 总线竞争保护等。

SPI 总线四种工作方式 SPI 模块为了和外设进行数据交换, 根据外设工作要求, 其输出串行同步时钟极性和相位可以进行配置, 时钟极性 (CPOL) 对传输协议没有重大的影响。如果 CPOL=0, 串行同步时钟的空闲状态为低电平; 如果 CPOL=1, 串行同步时钟的空闲状态为高电平。时钟相位 (CPHA) 能够配置用于选择两种不同的传输协议之一进行数据传输。如果 CPHA=0, 在串行同步时钟的第一个跳变沿 (上升或下降) 数据被采样; 如果 CPHA=1, 在串行同步时钟的第二个跳变沿 (上升或下降) 数据被采样。SPI 主模块和与之通信的外设备时钟相位和极性应该一致。

不同时钟相位下的总线数据传输时序如图 28.1.2 所示:



CPHA=0 时 SPI 总线数据传输时序



CPHA=1 时 SPI 总线数据传输时序

图 28.1.2 不同时钟相位下的总线传输时序 (CPHA=0/1)

STM32 的 SPI 功能很强大，SPI 时钟最多可以到 18Mhz，支持 DMA，可以配置为 SPI 协议或者 I2S 协议（仅大容量型号支持，战舰 STM32 开发板是支持的）。

本章，我们将利用 STM32 的 SPI 来读取外部 SPI FLASH 芯片（W25Q64），实现类似上节的功能。这里对 SPI 我们只简单介绍一下 SPI 的使用，STM32 的 SPI 详细介绍请参考《STM32 参考手册》第 457 页，23 节。然后我们再介绍下 SPI FLASH 芯片。

这节，我们使用 STM32 的 SPI2 的主模式，下面就来看看 SPI2 部分的设置步骤吧。SPI 相关的库函数和定义分布在文件 `stm32f10x_spi.c` 以及头文件 `stm32f10x_spi.h` 中。STM32 的主模式配置步骤如下：

1) 配置相关引脚的复用功能，使能 SPI2 时钟

我们要用 SPI2，第一步就要使能 SPI2 的时钟。其次要设置 SPI2 的相关引脚为复用输出，这样才会连接到 SPI2 上否则这些 IO 口还是默认的状态，也就是标准输入输出口。这里我们使用的是 PB13、14、15 这 3 个（SCK.、MISO、MOSI，CS 使用软件管理方式），所以设置这三个为复用 IO。

```
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(    RCC_APB2Periph_GPIOB, ENABLE );//PORTB 时钟使能
RCC_APB1PeriphClockCmd(    RCC_APB1Periph_SPI2,   ENABLE );//SPI2 时钟使能
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //PB13/14/15 复用推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化 GPIOB
```

类似的时钟使能和 IO 初始化我们前面多次讲解到，这里不做详细介绍。

2) 初始化 SPI2, 设置 SPI2 工作模式

接下来我们要初始化 SPI2，设置 SPI2 为主机模式，设置数据格式为 8 位，然设置 SCK 时钟



极性及采样方式。并设置 SPI2 的时钟频率（最大 18Mhz），以及数据的格式（MSB 在前还是 LSB 在前）。这在库函数中是通过 SPI_Init 函数来实现的。

```
void SPI_Init(SPI_TypeDef* SPIx, SPI_InitTypeDef* SPI_InitStruct);
```

跟其他外设初始化一样，第一个参数是 SPI 标号，这里我们是使用的 SPI2。下面我们来看看第二个参数结构体类型 SPI_InitTypeDef 的定义：

```
typedef struct
{
    uint16_t SPI_Direction;
    uint16_t SPI_Mode;
    uint16_t SPI_DataSize;
    uint16_t SPI_CPOL;
    uint16_t SPI_CPHA;
    uint16_t SPI_NSS;
    uint16_t SPI_BaudRatePrescaler;
    uint16_t SPI_FirstBit;
    uint16_t SPI_CRCPolynomial;
}SPI_InitTypeDef;
```

结构体成员变量比较多，这里我们挑取几个重要的成员变量讲解一下：

第一个参数 SPI_Direction 是用来设置 SPI 的通信方式，可以选择为半双工，全双工，以及串行发和串行收方式，这里我们选择全双工模式 SPI_Direction_2Lines_FullDuplex。

第二个参数 SPI_Mode 用来设置 SPI 的主从模式，这里我们设置为主机模式 SPI_Mode_Master，当然有需要你也可以选择为从机模式 SPI_Mode_Slave。

第三个参数 SPI_DataSize 为 8 位还是 16 位帧格式选择项，这里我们是 8 位传输，选择 SPI_DataSize_8b。

第四个参数 SPI_CPOL 用来设置时钟极性，我们设置串行同步时钟的空闲状态为高电平所以我们选择 SPI_CPOL_High。

第五个参数 SPI_CPHA 用来设置时钟相位，也就是选择在串行同步时钟的第几个跳变沿（上升或下降）数据被采样，可以为第一个或者第二个条边沿采集，这里我们选择第二个跳变沿，所以选择 SPI_CPHA_2Edge

第六个参数 SPI_NSS 设置 NSS 信号由硬件（NSS 管脚）还是软件控制，这里我们通过软件控制 NSS 关键，而不是硬件自动控制，所以选择 SPI_NSS_Soft。

第七个参数 SPI_BaudRatePrescaler 很关键，就是设置 SPI 波特率预分频值也就是决定 SPI 的时钟的参数，从不分频道 256 分频 8 个可选值，初始化的时候我们选择 256 分频值 SPI_BaudRatePrescaler_256，传输速度为 $36M/256=140.625KHz$ 。

第八个参数 SPI_FirstBit 设置数据传输顺序是 MSB 位在前还是 LSB 位在前，，这里我们选择 SPI_FirstBit_MSB 高位在前。

第九个参数 SPI_CRCPolynomial 是用来设置 CRC 校验多项式，提高通信可靠性，大于 1 即可。设置好上面 9 个参数，我们就可以初始化 SPI 外设了。初始化的范例格式为：

```
SPI_InitTypeDef SPI_InitStructure;
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //双线双向全双工
SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //主 SPI
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; // SPI 发送接收 8 位帧结构
SPI_InitStructure.SPI_CPOL = SPI_CPOL_High; //串行同步时钟的空闲状态为高电平
```



```
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge; //第二个跳变沿数据被采样
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //NSS 信号由软件控制
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256; //预分频 256
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //数据传输从 MSB 位开始
SPI_InitStructure.SPI_CRCPolynomial = 7; //CRC 值计算的多项式
SPI_Init(SPI2, &SPI_InitStructure); //根据指定的参数初始化外设 SPIx 寄存器
```

3) 使能 SPI2

初始化完成之后接下来是要使能 SPI2 通信了，在使能 SPI2 之后，我们就可以开始 SPI 通讯了。使能 SPI2 的方法是：

```
SPI_Cmd(SPI2, ENABLE); //使能 SPI 外设
```

4) SPI 传输数据

通信接口当然需要有发送数据和接受数据的函数，固件库提供的发送数据函数原型为：

```
void SPI_I2S_SendData(SPI_TypeDef* SPIx, uint16_t Data);
```

这个函数很好理解，往 SPIx 数据寄存器写入数据 Data，从而实现发送。

固件库提供的接受数据函数原型为：

```
uint16_t SPI_I2S_ReceiveData(SPI_TypeDef* SPIx);
```

这个函数也不难理解，从 SPIx 数据寄存器读出接收到的数据。

5) 查看 SPI 传输状态

在 SPI 传输过程中，我们经常要判断数据是否传输完成，发送区是否为空等等状态，这是通过函数 SPI_I2S_GetFlagStatus 实现的，这个函数很简单就不详细讲解，判断发送是否完成的方法是：

```
SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_RXNE);
```

SPI2 的使用就介绍到这里，接下来介绍一下 W25Q64。W25Q64 是华邦公司推出的大容量 SPI FLASH 产品，W25Q64 的容量为 64Mb，该系列还有 W25Q80/16/32 等。ALIENTEK 所选择的 W25Q64 容量为 64Mb，也就是 8M 字节。

W25Q64 将 8M 的容量分为 128 个块 (Block)，每个块大小为 64K 字节，每个块又分为 16 个扇区 (Sector)，每个扇区 4K 个字节。W25Q64 的最少擦除单位为一个扇区，也就是每次必须擦除 4K 个字节。这样我们需要给 W25Q64 开辟一个至少 4K 的缓存区，这样对 SRAM 要求比较高，要求芯片必须有 4K 以上 SRAM 才能很好的操作。

W25Q64 的擦写周期多达 10W 次，具有 20 年的数据保存期限，支持电压为 2.7~3.6V，W25Q64 支持标准的 SPI，还支持双输出/四输出的 SPI，最大 SPI 时钟可以到 80Mhz (双输出时相当于 160Mhz，四输出时相当于 320M)，更多的 W25Q64 的介绍，请参考 W25Q64 的 DATASHEET。

28.2 硬件设计

本章实验功能简介：开机的时候先检测 W25Q64 是否存在，然后在主循环里面用 1 个按键 (KEY0) 用来执行写入 W25Q64 的操作，另外一个按键 (WK_UP) 用来执行读出操作，在 TFTLCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) WK_UP 和 KEY1 按键
- 3) TFTLCD 模块



4) SPI

5) W25Q64

这里只介绍 W25Q64 与 STM32 的连接，板上的 W25Q64 是直接连在 STM32 的 SPI2 上的，连接关系如图 28.2.1 所示：

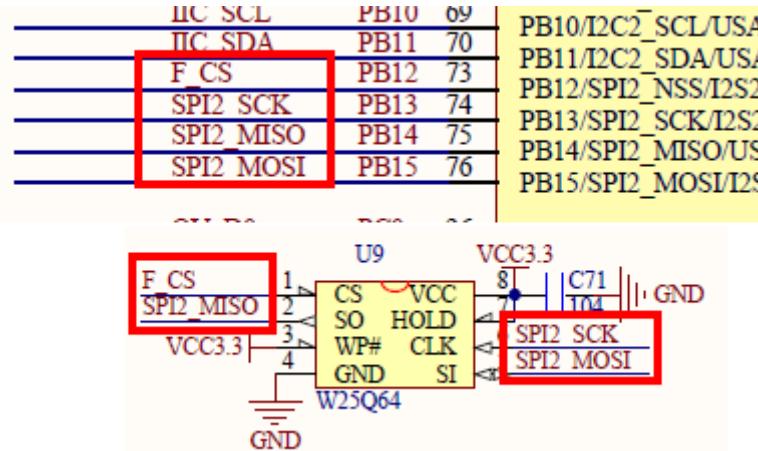


图 28.2.1 STM32 与 W25Q64 连接电路图

28.3 软件设计

打开我们光盘的 SPI 实验工程，可以看到我们加入了 spi.c,flash.c 文件以及头文件 spi.h 和 flash.h，同时引入了库函数文件 stm32f10x_spi.c 文件以及头文件 stm32f10x_spi.h。

打开 spi.c 文件，看到如下代码：

```
#include "spi.h"

//以下是 SPI 模块的初始化代码，配置成主机模式，访问 SD Card/W25Q64/NRF24L01
//SPI 口初始化
//这里针是对 SPI2 的初始化
void SPI2_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef SPI_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE );//PORTB 时钟使能
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE );//①SPI2 时钟使能

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //PB13/14/15 复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure); //①初始化 GPIOB
    GPIO_SetBits(GPIOB,GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15); //PB13/14/15 上拉

    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //设置 SPI 全双工
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //设置 SPI 工作模式:设置为主 SPI
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; // 8 位帧结构
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;//选择了串行时钟的稳态:时钟悬空高
```



```
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge; //数据捕获于第二个时钟沿
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //NSS 信号由硬件管理
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256; //预分频 256
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //数据传输从 MSB 位开始
SPI_InitStructure.SPI_CRCPolynomial = 7; //CRC 值计算的多项式
SPI_Init(SPI2, &SPI_InitStructure); //②根据指定的参数初始化外设 SPIx 寄存器

    SPI_Cmd(SPI2, ENABLE); //③使能 SPI 外设
    SPI2_ReadWriteByte(0xff); //④启动传输
}

//SPI 速度设置函数
//SpeedSet://SPI_BaudRatePrescaler_256 256 分频 (SPI 281.25K@sys 72M)
void SPI2_SetSpeed(u8 SPI_BaudRatePrescaler)
{
    assert_param(IS_SPI_BAUDRATE_PRESCALER(SPI_BaudRatePrescaler));
    SPI2->CR1&=0xFFC7;
    SPI2->CR1|=SPI_BaudRatePrescaler; //设置 SPI2 速度
    SPI_Cmd(SPI2,ENABLE);

}

//SPIx 读写一个字节
//TxData:要写入的字节
//返回值:读取到的字节
u8 SPI2_ReadWriteByte(u8 TxData)
{
    u8 retry=0;
    while (SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_TXE) == RESET) //等待发送区空
    {
        retry++;
        if(retry>200)return 0;
    }
    SPI_I2S_SendData(SPI2, TxData); //通过外设 SPIx 发送一个数据
    retry=0;
    while (SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_RXNE) == RESET) //等待接收
        //完一个 byte
    {
        retry++;
        if(retry>200)return 0;
    }
    return SPI_I2S_ReceiveData(SPI2); //返回通过 SPIx 最近接收的数据
}
```

此部分代码主要初始化 SPI，这里我们选择的是 SPI2，所以在 SPI2_Init 函数里面，其相关的操作都是针对 SPI2 的，其初始化步骤和我们上面介绍步骤 1-5 一样，我们在代码中也使用了①~⑤标注。在初始化之后，我们就可以开始使用 SPI2 了，在 SPI2_Init 函数里面，把 SPI2 的



波特率设置成了最低（36Mhz，256 分频为 140.625KHz）。在外部函数里面，我们通过 SPI2_SetSpeed 来设置 SPI2 的速度，而我们的数据发送和接收则是通过 SPI2_ReadWriteByte 函数来实现的。SPI2_SetSpeed 函数我们是通过寄存器设置方式来实现的，因为固件库并没有提供单独的设置分频系数的函数，当然，我们也可以勉强的调用 SPI_Init 初始化函数来实现分频系数修改。要读懂这段代码，可以直接查找中文参考手册中 SPI 章节的寄存器 CR1 的描述即可。

下面我们打开 flash.c，里面编写的是与 W25Q64 操作相关的代码，由于篇幅所限，详细代码，这里就不贴出了。我们仅介绍几个重要的函数，首先是 SPI_Flash_Read 函数，该函数用于从 W25Q64 的指定地址读出指定长度的数据。其代码如下：

```
//读取 SPI FLASH
//在指定地址开始读取指定长度的数据
// pBuffer:数据存储区
//ReadAddr:开始读取的地址(24bit)
//NumByteToRead:要读取的字节数(最大 65535)
void SPI_Flash_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead)
{
    u16 i;
    SPI_FLASH_CS=0;                      //使能器件
    SPI2_ReadWriteByte(W25X_ReadData);     //发送读取命令
    SPI2_ReadWriteByte((u8)((ReadAddr)>>16)); //发送 24bit 地址
    SPI2_ReadWriteByte((u8)((ReadAddr)>>8));
    SPI2_ReadWriteByte((u8)ReadAddr);
    for(i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPI2_ReadWriteByte(0xFF); //循环读数
    }
    SPI_FLASH_CS=1;
}
```

由于 W25Q64 支持以任意地址（但是不能超过 W25Q64 的地址范围）开始读取数据，所以，这个代码相对来说就比较简单了，在发送 24 位地址之后，程序就可以开始循环读数据了，其地址会自动增加的，不过要注意，不能读的数据超过了 W25Q64 的地址范围哦！否则读出来的数据，就不是你想要的数据了。

有读的函数，当然就有写的函数了，接下来，我们介绍 SPI_Flash_Write 这个函数，该函数的作用与 SPI_Flash_Read 的作用类似，不过是用来写数据到 W25Q64 里面的，其代码如下：

```
//写 SPI FLASH
//在指定地址开始写入指定长度的数据
//该函数带擦除操作!
// pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大 65535)
u8 SPI_FLASH_BUFFER[4096];
void SPI_Flash_Write(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
{
    u32 secpos;
```



```
u16 secoff;
u16 secremain;
u16 i;
u8 * SPI_FLASH_BUF;
SPI_FLASH_BUF=SPI_FLASH_BUFFER;
secpos=WriteAddr/4096;           //扇区地址 0~511 for w25x16
secoff=WriteAddr%4096;          //在扇区内的偏移
secremain=4096-secoff;          //扇区剩余空间大小
//printf("ad:%X,nb:%X\r\n",WriteAddr,NumByteToWrite); //测试用
if(NumByteToWrite<=secremain)secremain=NumByteToWrite;//不大于 4096 个字节
while(1)
{
    SPI_Flash_Read(SPI_FLASH_BUF,secpos*4096,4096); //读出整个扇区的内容
    for(i=0;i<secremain;i++)                      //校验数据
    {
        if(SPI_FLASH_BUF[secoff+i]!=0xFF)break;      //需要擦除
    }
    if(i<secremain)                                //需要擦除
    {
        SPI_Flash_Erase_Sector(secpos);             //擦除这个扇区
        for(i=0;i<secremain;i++)                  //复制
        {
            SPI_FLASH_BUF[i+secoff]=pBuffer[i];
        }
        SPI_Flash_Write_NoCheck(SPI_FLASH_BUF,secpos*4096,4096);
        //写入整个扇区
    }else SPI_Flash_Write_NoCheck(pBuffer,WriteAddr,secremain);
    //写已经擦除了的,直接写入扇区剩余区间.
    if(NumByteToWrite==secremain)break;//写入结束了
    else//写入未结束
    {
        secpos++;                     //扇区地址增 1
        secoff=0;                     //偏移位置为 0
        pBuffer+=secremain;           //指针偏移
        WriteAddr+=secremain;         //写地址偏移
        NumByteToWrite-=secremain;     //字节数递减
        if(NumByteToWrite>4096)secremain=4096; //下一个扇区还是写不完
        else secremain=NumByteToWrite;   //下一个扇区可以写完了
    }
};
}
```

该函数可以在 W25Q64 的任意地址开始写入任意长度（必须不超过 W25Q64 的容量）的数据。我们这里简单介绍一下思路：先获得首地址（WriteAddr）所在的扇区，并计算在扇区内的偏移，然后判断要写入的数据长度是否超过本扇区所剩下的长度，如果不超过，再先看看是否要擦除，如果不，直接写入数据即可，如果要，则读出整个扇区，在偏移处开始写入指定长度的数据，然后擦除这个扇区，再一次性写入。当所需要写入的数据长度超过一个扇区的长度



的时候，我们先按照前面的步骤把扇区剩余部分写完，再在新扇区内执行同样的操作，如此循环，直到写入结束。

其他的代码就比较简单了，我们这里不介绍了。接着打开 flahs.h 文件可以看到，这里面就定义了一些与 W25Q64 操作相关的命令（部分省略了），这些命令在 W25Q64 的数据手册上都有详细的介绍，感兴趣的读者可以参考该数据手册，其他的就没啥好说的了。。最后，我们看看 main.c 里面代码如下：

```
const u8 TEXT_Buffer[]{"Warship STM32 SPI TEST"};  
#define SIZE sizeof(TEXT_Buffer)  
int main(void)  
{  
    u8 key;  
    u16 i=0;  
    u8 datatemp[SIZE];  
    u32 FLASH_SIZE;  
    delay_init();           //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级  
    uart_init(9600);        //串口初始化波特率为 9600  
    LED_Init();            //LED 端口初始化  
    LCD_Init();             //初始化 LCD  
    KEY_Init();             //初始化 KEY  
    SPI_Flash_Init();       //SPI FLASH 初始化  
    POINT_COLOR=RED; //设置字体为红色  
    LCD_ShowString(60,50,200,16,16,"Mini STM32");  
    LCD_ShowString(60,70,200,16,16,"SPI TEST");  
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(60,110,200,16,16,"2012/8/5");  
    LCD_ShowString(60,130,200,16,16,"WKUP:Write KEY1:Read"); //显示提示信息  
    while(SPI_Flash_ReadID()!=W25Q64)                                //检测不到 W25Q64  
    {  
        LCD_ShowString(60,150,200,16,16,"25Q64 Check Failed!");  
        delay_ms(500);  
        LCD_ShowString(60,150,200,16,16,"Please Check!");  
        delay_ms(500);  
        LED0=!LED0;//DS0 闪烁  
    }  
    LCD_ShowString(60,150,200,16,16,"25Q64 Ready!");  
  
    FLASH_SIZE=8*1024*1024;//FLASH 大小为 8M 字节  
    POINT_COLOR=BLUE;//设置字体为蓝色  
    while(1)  
    {  
        key=KEY_Scan(0);  
        if(key==KEY_UP)          //KEY_UP 按下,写入 W25Q64
```



```
{    LCD_Fill(0,170,239,319,WHITE); //清除半屏
    LCD_ShowString(60,170,200,16,16,"Start Write W25Q64....");
    SPI_Flash_Write((u8*)TEXT_Buffer,FLASH_SIZE-100,SIZE);//从倒数第 100
    //个地址处开始,写入 SIZE 长度的数据
    LCD_ShowString(60,170,200,16,16,"W25Q64 Write Finished");//提示传送完成
}
if(key==KEY_DOWN) //KEY_DOWN 按下,读取字符串并显示
{
    LCD_ShowString(60,170,200,16,16,"Start Read W25Q64....");
    SPI_Flash_Read(datatemp,FLASH_SIZE-100,SIZE);//从倒数第 100 个地址处
    //开始,读出 SIZE 个字节
    LCD_ShowString(60,170,200,16,16,"The Data Readed Is: ");
    LCD_ShowString(60,190,200,16,16,datatemp);//显示读到的字符串
}
i++;
delay_ms(10);
if(i==20)
{
    LED0=!LED0; //提示系统正在运行
    i=0;
}
}
}
```

这部分代码和 IIC 实验那部分代码大同小异，我们就不多说了，实现的功能就和 IIC 差不多，不过此次写入和读出的是 SPI FLASH，而不是 EEPROM。

28.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，通过先按 WK_UP 按键写入数据，然后按 KEY1 读取数据，得到如图 28.4.1 所示：

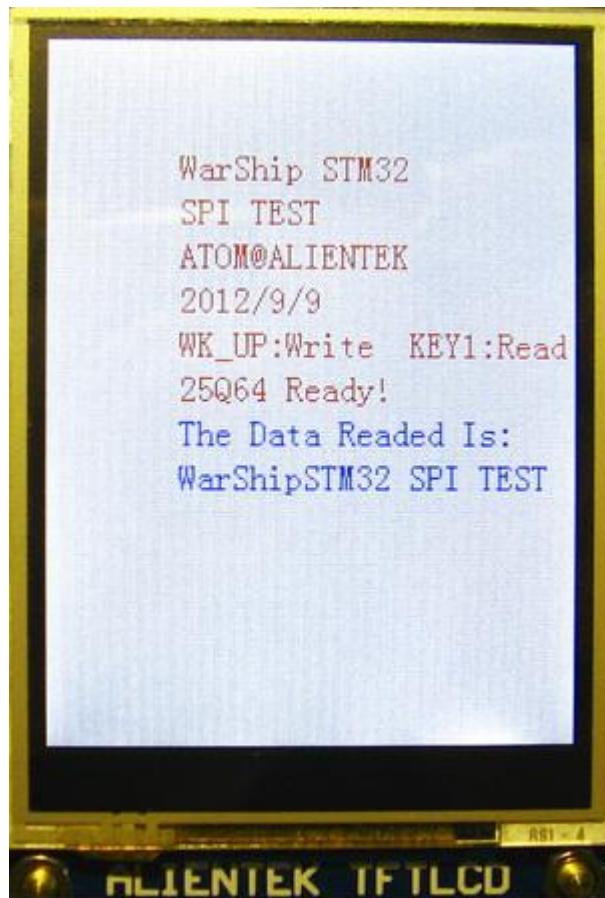


图 28.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。程序在开机的时候会检测 W25Q64 是否存在，如果不存在则会在 TFTLCD 模块上显示错误信息，同时 DS0 慢闪。大家可以通过跳线帽把 PB12 和 PB13 短接就可以看到报错了。

第二十九章 485 实验

本章我们将向大家介绍如何利用 STM32 的串口实现 485 通信（半双工）。在本章中，我们将利用 STM32 的串口 2 来实现两块开发板之间的 485 通信，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 29.1 485 简介
- 29.2 硬件设计
- 29.3 软件设计
- 29.4 下载验证



29.1 485 简介

485(一般称作 RS485/EIA-485)是隶属于 OSI 模型物理层的电气特性规定为 2 线，半双工，多点通信的标准。它的电气特性和 RS-232 大不一样。用缆线两端的电压差值来表示传递信号。RS485 仅仅规定了接受端和发送端的电气特性。它没有规定或推荐任何数据协议。

RS485 的特点包括：

- 1) 接口电平低，不易损坏芯片。RS485 的电气特性：逻辑“1”以两线间的电压差为+(2~6)V 表示；逻辑“0”以两线间的电压差为-(2~6)V 表示。接口信号电平比 RS232 降低了，不易损坏接口电路的芯片，且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。
- 2) 传输速率高。10 米时，RS485 的数据最高传输速率可达 35Mbps，在 1200m 时，传输速度可达 100Kbps。
- 3) 抗干扰能力强。RS485 接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。
- 4) 传输距离远，支持节点多。RS485 总线最长可以传输 1200m 以上（速率≤100Kbps）一般最大支持 32 个节点，如果使用特制的 485 芯片，可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

RS485 推荐使用在点对点网络中，线型，总线型，不能是星型，环型网络。理想情况下 RS485 需要 2 个匹配电阻，其阻值要求等于传输电缆的特性阻抗(一般为 120Ω)。没有特性阻抗的话，当所有的设备都静止或者没有能量的时候就会产生噪声，而且线移需要双端的电压差。没有终接电阻的话，会使得较快速的发送端产生多个数据信号的边缘，导致数据传输出错。485 推荐的连接方式如图 29.1.2 所示：

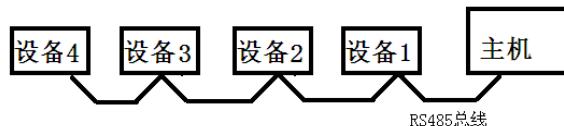


图 29.1.2 RS485 连接

在上面的连接中，如果需要添加匹配电阻，我们一般在总线的起止端加入，也就是主机和设备 4 上面各加一个 120Ω 的匹配电阻。

由于 RS485 具有传输距离远、传输速度快、支持节点多和抗干扰能力更强等特点，所以 RS485 有很广泛的应用。

战舰 STM32 开发板采用 SP3485 作为收发器，该芯片支持 3.3V 供电，最大传输速度可达 10Mbps，支持多达 32 个节点，并且有输出短路保护。该芯片的框图如图 29.1.2 所示：

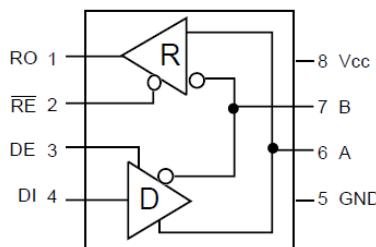


图 29.1.2 SP3485 框图

图中 A、B 总线接口，用于连接 485 总线。RO 是接收输出端，DI 是发送数据收入端，RE 是接收使能信号(低电平有效)，DE 是发送使能信号(高电平有效)。



本章，我们通过该芯片连接 STM32 的串口 2，实现两个开发板之间的 485 通信。本章将实现这样的功能：通过连接两个战舰 STM32 开发板的 RS485 接口，然后由 KEY0 控制发送，当按下一个开发板的 KEY0 的时候，就发送 5 个数据给另外一个开发板，并在两个开发板上分别显示发送的值和接收到的值。

本章，我们只需要配置好串口 2，就可以实现正常的 485 通信了，串口 2 的配置和串口 1 基本类似，只是串口的时钟来自 APB1，最大频率为 36Mhz。

29.2 硬件设计

本章要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) KEY0 按键
- 3) TFTLCD 模块
- 4) 串口 2
- 5) RS485 收发芯片

前面 3 个都有详细介绍，这里我们介绍 RS485 和串口 2 的连接关系，如图 29.2.1 所示：

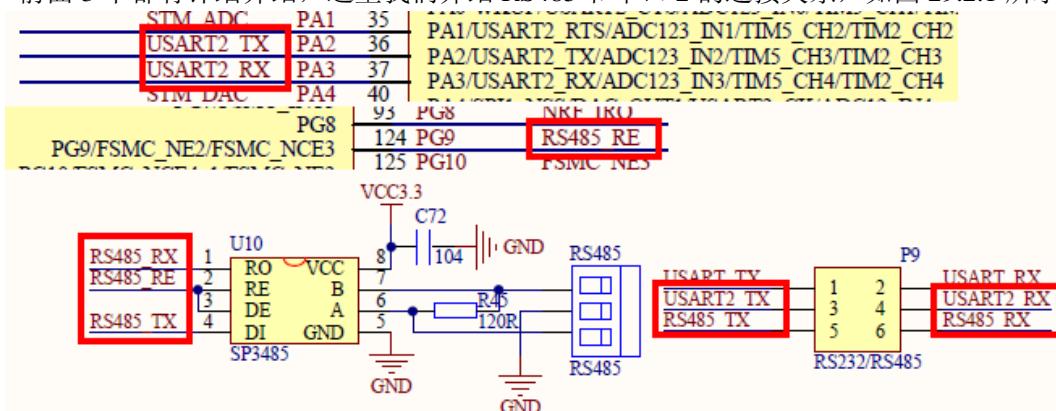


图 29.2.1 STM32 与 SP3485 连接电路图

从上图可以看出：STM32 的串口 2 通过 P9 端口设置，连接到 SP3485，通过 STM32 的 PG9 控制 SP3485 的收发，当 PG9=0 的时候，为接收模式；当 PG9=1 的时候，为发送模式。这里注意，我们要设置好开发板上 P9 排针的连接，通过跳线帽将 PA2 和 PA3 分别连接到 485T 和 485R 上面，如图 29.2.2 所示：

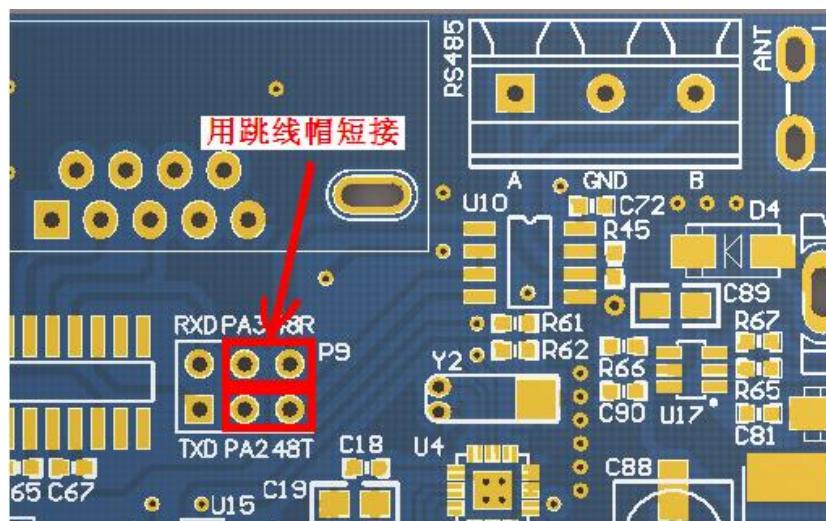




图 29.2.2 硬件连接示意图

最后，我们用 2 根导线将两个开发板 RS485 端子的 A 和 A, B 和 B 连接起来。这里注意不要接反了（A 接 B），接反了会导致通讯异常！！

29.3 软件设计

打开我们的 485 实验例程，可以发现项目中加入了一个 rs485.c 文件以及其头文件 rs485 头文件，同时 485 通信调用的库函数和定义分布在 stm32f10x_usart.c 文件和头文件 stm32f10x_usart.h 文件中。

打开 rs485.c 文件，代码如下：

```
#include "sys.h"
#include "rs485.h"
#include "delay.h"
#ifndef EN_USART2_RX      //如果使能了接收
//接收缓存区
u8 RS485_RX_BUF[64];    //接收缓冲,最大 64 个字节.
//接收到的数据长度
u8 RS485_RX_CNT=0;
void USART2_IRQHandler(void)
{
    u8 res;
    if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET) //接收到数据
    {
        res =USART_ReceiveData(USART2);                      //读取接收到的数据
        if(RS485_RX_CNT<64)
        {
            RS485_RX_BUF[RS485_RX_CNT]=res;                //记录接收到的值
            RS485_RX_CNT++;                                //接收数据增加 1
        }
    }
}
#endif
//初始化 IO 串口 2
//bound:波特率
void RS485_Init(u32 bound)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|
    RCC_APB2Periph_GPIOG, ENABLE);                  //使能 GPIOA,G 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE);//使能串口 2 时钟
```



```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;           //PG9 端口配置
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;     //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOG, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;            //PA2
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       //复用推挽
GPIO_Init(GPIOA, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;//PA3
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //浮空输入
GPIO_Init(GPIOA, &GPIO_InitStructure);

RCC_APB1PeriphResetCmd(RCC_APB1Periph_USART2,ENABLE); //复位串口 2
RCC_APB1PeriphResetCmd(RCC_APB1Periph_USART2,DISABLE);//停止复位

#endif EN_USART2_RX                                //如果使能了接收
USART_InitStructure USART_BaudRate = bound;        //一般设置为 9600;
USART_InitStructure USART_WordLength = USART_WordLength_8b;//8 位数据长度
USART_InitStructure USART_StopBits = USART_StopBits_1; //一个停止位
USART_InitStructure USART_Parity = USART_Parity_No; //奇偶校验位
USART_InitStructure USART_HardwareFlowControl=
          USART_HardwareFlowControl_None; //无硬件数据流控制
USART_InitStructure USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//收发
USART_Init(USART2, &USART_InitStructure);           //初始化串口

NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;   //使能串口 2 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3; //先占优先级 2 级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;      //从优先级 2 级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;        //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);                   //初始化 NVIC 寄存器

USART_ITConfig(USART2, USART_IT_RXNE, ENABLE); //开启中断
USART_Cmd(USART2, ENABLE);                      //使能串口

#endif
RS485_TX_EN=0;                                 //默认为接收模式
}

//RS485 发送 len 个字节.
//buf:发送区首地址
//len:发送的字节数(为了和本代码的接收匹配,这里建议不要超过 64 个字节)
void RS485_Send_Data(u8 *buf,u8 len)
{
    u8 t;
```



```
RS485_TX_EN=1;          //设置为发送模式
for(t=0;t<len;t++)
{
    while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
    USART_SendData(USART2,buf[t]);
}

while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
RS485_RX_CNT=0;
RS485_TX_EN=0;          //设置为接收模式
}

//RS485 查询接收到的数据
//buf:接收缓存首地址
//len:读到的数据长度
void RS485_Receive_Data(u8 *buf,u8 *len)
{
    u8 rxlen=RS485_RX_CNT;
    u8 i=0;
    *len=0;           //默认为 0
    delay_ms(10);    //等待 10ms,连续超过 10ms 没有接收到一个数据,则认为接收结束
    if(rxlen==RS485_RX_CNT&&rxlen)//接收到了数据,且接收完成了
    {
        for(i=0;i<rxlen;i++)
        {
            buf[i]=RS485_RX_BUF[i];
        }
        *len=RS485_RX_CNT;      //记录本次数据长度
        RS485_RX_CNT=0;        //清零
    }
}
```

此部分代码总共 4 个函数，其中 RS485_Init 函数为 485 通信初始化函数，其实基本上就是在配置串口 2，只是把 PG9 也顺带配置了，用于控制 SP3485 的收发。同时如果使能中断接收的话，会执行串口 2 的中断接收配置。USART2_IRQHandler 函数用于中断接收来自 485 总线的数据，将其存放在 RS485_RX_BUF 里面。最后 RS485_Send_Data 和 RS485_Receive_Data 这两个函数用来发送数据到 485 总线和读取从 485 总线收到的数据，都比较简单。

头文件 rs485.h 中代码比较简单，在其中我们开启了串口 2 的中断接收。最后，我们看看主函数 main 的内容如下：

```
int main(void)
{
    u8 key;
    u8 i=0,t=0;
    u8 cnt=0;
    u8 rs485buf[5];
```



```
delay_init();          //延时函数初始化
NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
uart_init(9600);      //串口初始化波特率为 9600
LED_Init();           //LED 端口初始化
LCD_Init();           //LCD 初始化
KEY_Init();           //KEY 初始化
RS485_Init(9600);    // RS485 初始化

POINT_COLOR=RED; //设置字体为红色
LCD_ShowString(60,50,200,16,16,"Mini STM32");
LCD_ShowString(60,70,200,16,16,"RS485 TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/8/5");
LCD_ShowString(60,130,200,16,16,"KEY0:Send");      //显示提示信息
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(60,150,200,16,16,"Count:");        //显示当前计数值
LCD_ShowString(60,170,200,16,16,"Send Data:");     //提示发送的数据
LCD_ShowString(60,210,200,16,16,"Receive Data:");   //提示接收到的数据
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY_RIGHT)                                //KEY0 按下,发送一次数据
    {
        for(i=0;i<5;i++)
        {
            rs485buf[i]=cnt+i;                      //填充发送缓冲区
            LCD_ShowxNum(60+i*32,190,rs485buf[i],3,16,0X80); //显示数据
        }
        RS485_Send_Data(rs485buf,5);                //发送 5 个字节
    }
    RS485_Receive_Data(rs485buf,&key);
    if(key)                                         //接收到有数据
    {
        if(key>5)key=5;                            //最大是 5 个数据.
        for(i=0;i<key;i++)
        LCD_ShowxNum(60+i*32,230,rs485buf[i],3,16,0X80); //显示数据
    }
    t++;
    delay_ms(10);
    if(t==20)
    {
        LED0=!LED0;                             //提示系统正在运行
        t=0;
        cnt++;
        LCD_ShowxNum(60+48,150,cnt,3,16,0X80); //显示数据
    }
}
```



```
    }  
}  
}
```

此部分代码，cnt是一个累加数，一旦KEY_RIGHT(KEY0)按下，就以这个数位基准连续发送5个数据。当485总线收到数据的时候，就将收到的数据直接显示在LCD屏幕上。

29.4 下载验证

在代码编译成功之后，我们通过下载代码到ALIENTEK战舰STM32开发板上（注意要2个开发板都下载这个代码哦），得到如图28.4.1所示：



图 28.4.1 程序运行效果图

伴随DS0的不停闪烁，提示程序在运行。此时，我们按下KEY0就可以在另外一个开发板上面收到这个开发板发送的数据了。如图28.4.2所示：

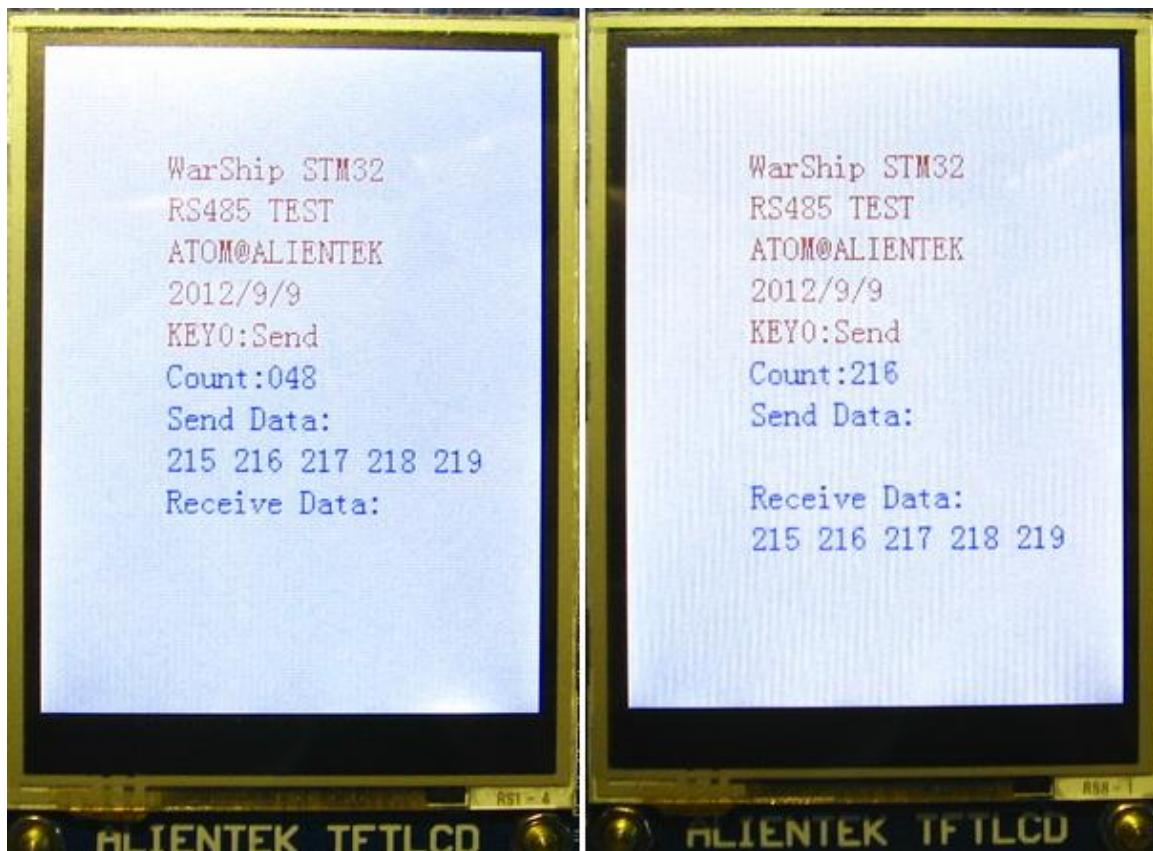


图 28.4.2 RS485 实验测试图片

上图中，左侧的图片来自开发板 A，发送了 5 个数据，右侧的图片来自开发板 B，接收到了来自开发板 A 的 5 个数据。

本章介绍的 485 总线时通过串口控制收发的，我们只需要将 P9 的跳线帽稍作改变，该实验就变成了一个 RS232 串口通信实验了，通过对接两个开发板的 RS232 接口，即可得到同样的实验现象，有兴趣的读者可以实验一下。

第三十章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32 自带的 CAN 控制器来实现两个开发板之间的 CAN 通讯，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 30.1 CAN 简介
- 30.2 硬件设计
- 30.3 软件设计
- 30.4 下载验证



30.1 CAN 简介

CAN 是 Controller Area Network 的缩写 (以下称为 CAN)，是 ISO 国际标准化的串行通信协议。在当前的汽车产业中，出于对安全性、舒适性、方便性、低公害、低成本的要求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN，进行大量数据的高速通信”的需要，1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化，现在在欧洲已是汽车网络的标准协议。

现在，CAN 的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。现场总线是当今自动化领域技术发展的热点之一，被誉为自动化领域的计算机局域网。它的出现为分布式控制系统实现各节点之间实时、可靠的数据通信提供了强有力的技术支持。

CAN 控制器根据两根线上的电位差来判断总线电平。总线电平分为显性电平和隐性电平，二者必居其一。发送方通过使总线电平发生变化，将消息发送给接收方。

CAN 协议具有一下特点：

- 1) **多主控制。**在总线空闲时，所有单元都可以发送消息（多主控制），而两个以上的单元同时开始发送消息时，根据标识符 (Identifier 以下简称 ID) 决定优先级。ID 并不是表示发送的目的地址，而是表示访问总线的消息的优先级。两个以上的单元同时开始发送消息时，对各消息 ID 的每个位进行逐个仲裁比较。仲裁获胜（被判定为优先级最高）的单元可继续发送消息，仲裁失利的单元则立刻停止发送而进行接收工作。
- 2) **系统的柔軟性。**与总线相连的单元没有类似于“地址”的信息。因此在总线上增加单元时，连接在总线上的其它单元的软硬件及应用层都不需要改变。
- 3) **通信速度较快，通信距离远。**最高 1Mbps (距离小于 40M)，最远可达 10KM (速率低于 5Kbps)。
- 4) **具有错误检测、错误通知和错误恢复功能。**所有单元都可以检测错误 (错误检测功能)，检测出错误的单元会立即同时通知其他所有单元 (错误通知功能)，正在发送消息的单元一旦检测出错误，会强制结束当前的发送。强制结束发送的单元会不断反复地重新发送此消息直到成功发送为止 (错误恢复功能)。
- 5) **故障封闭功能。**CAN 可以判断出错误的类型是总线上暂时的数据错误 (如外部噪声等) 还是持续的数据错误 (如单元内部故障、驱动器故障、断线等)。由此功能，当总线上发生持续数据错误时，可将引起此故障的单元从总线上隔离出去。
- 6) **连接节点多。**CAN 总线是可同时连接多个单元的总线。可连接的单元总数理论上是没有限制的。但实际上可连接的单元数受总线上的时间延迟及电气负载的限制。降低通信速度，可连接的单元数增加；提高通信速度，则可连接的单元数减少。

正是因为 CAN 协议的这些特点，使得 CAN 特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。

CAN 协议经过 ISO 标准化后有两个标准：ISO11898 标准和 ISO11519-2 标准。其中 ISO11898 是针对通信速率为 125Kbps~1Mbps 的高速通信标准，而 ISO11519-2 是针对通信速率为 125Kbps 以下的低速通信标准。

本章，我们使用的是 450Kbps 的通信速率，使用的是 ISO11898 标准，该标准的物理层特征如图 30.1.1 所示：

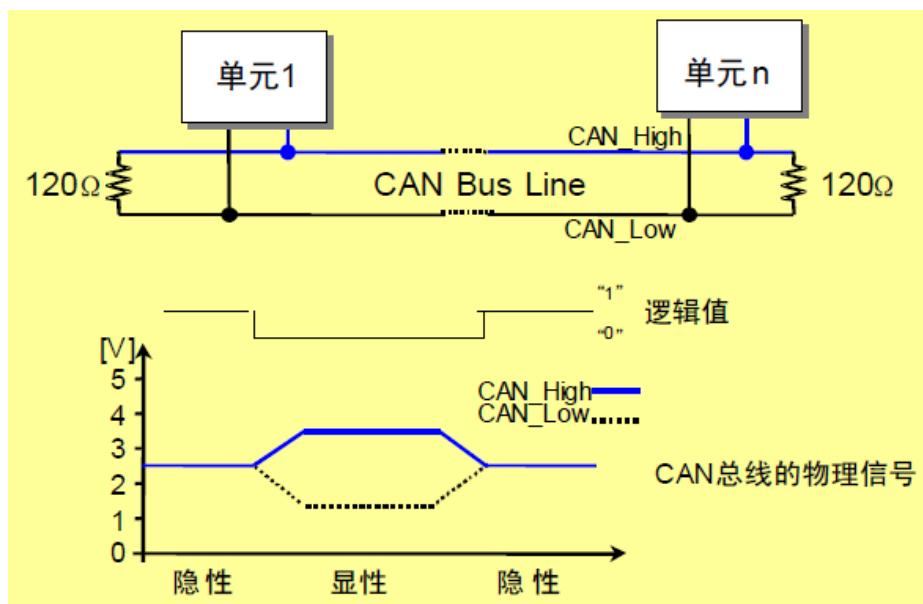


图 30.1.1 ISO11898 物理层特性

从该特性可以看出，显性电平对应逻辑 0，CAN_H 和 CAN_L 之差为 2.5V 左右。而隐形电平对应逻辑 1，CAN_H 和 CAN_L 之差为 0V。在总线上显性电平具有优先权，只要有一个单元输出显性电平，总线上即为显性电平。而隐形电平则具有包容的意味，只有所有的单元都输出隐形电平，总线上才为隐形电平（显性电平比隐形电平更强）。另外，在 CAN 总线的起止端都有一个 120Ω 的终端电阻，来做阻抗匹配，以减少回波反射。

CAN 协议是通过以下 5 种类型的帧进行的：

- 数据帧
- 要控帧
- 错误帧
- 过载帧
- 帧间隔

另外，数据帧和遥控帧有标准格式和扩展格式两种格式。标准格式有 11 个位的标识符 (ID)，扩展格式有 29 个位的 ID。各种帧的用途如表 30.1.1 所示：

帧类型	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它单元通知错误的帧
过载帧	用于接收单元通知其尚未做好接收准备的帧
间隔帧	用于将数据帧及遥控帧与前面的帧分离开来的帧

表 30.1.1 CAN 协议各种帧及其用途

由于篇幅所限，我们这里仅对数据帧进行详细介绍，数据帧一般由 7 个段构成，即：

- (1) 帧起始。表示数据帧开始的段。
- (2) 仲裁段。表示该帧优先级的段。
- (3) 控制段。表示数据的字节数及保留位的段。
- (4) 数据段。数据的内容，一帧可发送 0~8 个字节的数据。
- (5) CRC 段。检查帧的传输错误的段。



(6) ACK 段。表示确认正常接收的段。

(7) 帧结束。表示数据帧结束的段。

数据帧的构成如图 30.1.2 所示：

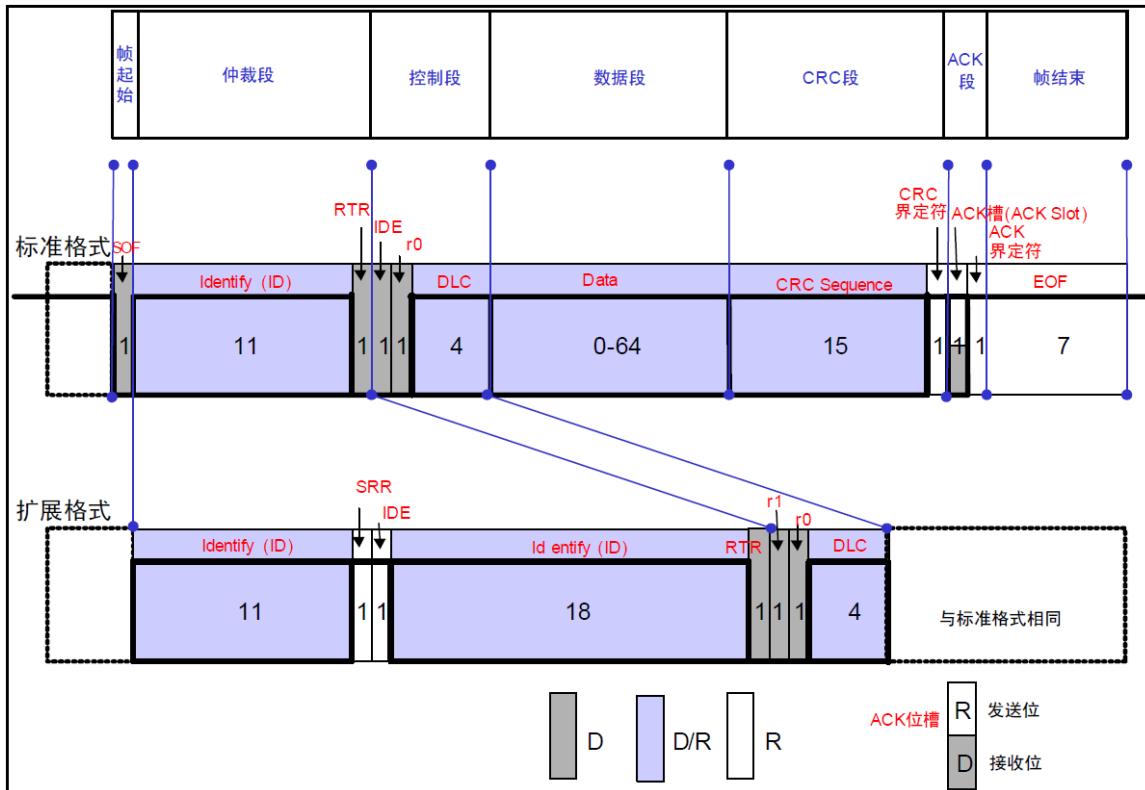


图 30.1.2 数据帧的构成

图中 D 表示显性电平，R 表示隐形电平（下同）。

帧起始，这个比较简单，标准帧和扩展帧都是由 1 个位的显性电平表示帧起始。

仲裁段，表示数据优先级的段，标准帧和扩展帧格式在本段有所区别，如图 30.1.3 所示：

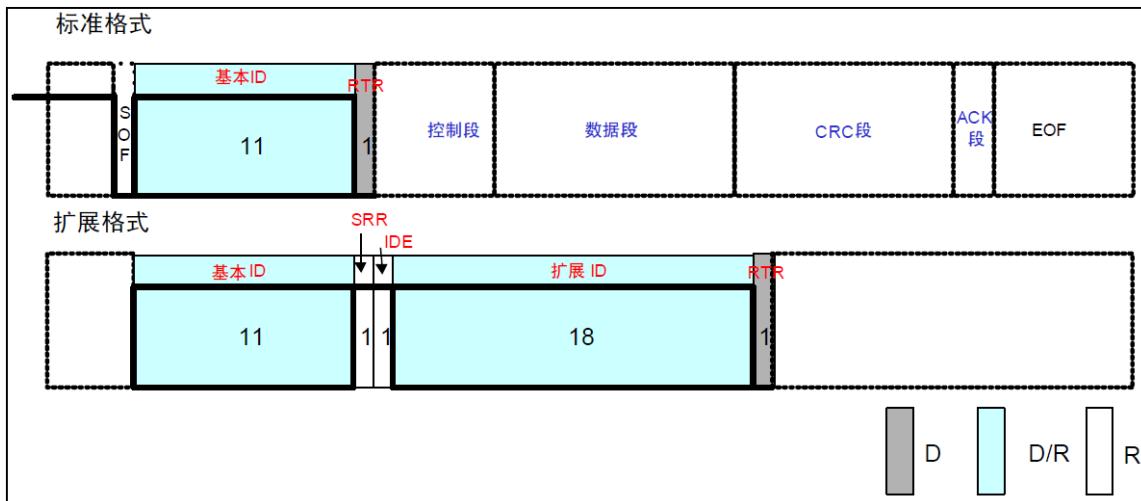


图 30.1.3 数据帧仲裁段构成

标准格式的 ID 有 11 个位。从 ID28 到 ID18 被依次发送。禁止高 7 位都为隐性（禁止设定：ID=1111111XXXX）。扩展格式的 ID 有 29 个位。基本 ID 从 ID28 到 ID18，扩展 ID 由



ID17 到 ID0 表示。基本 ID 和标准格式的 ID 相同。禁止高 7 位都为隐性（禁止设定：基本 ID=1111111XXXX）。

其中 RTR 位用于标识是否是远程帧（0，数据帧；1，远程帧），IDE 位为标识符选择位（0，使用标准标识符；1，使用扩展标识符），SRR 位为代替远程请求位，为隐性位，它代替了标准帧中的 RTR 位。

控制段，由 6 个位构成，表示数据段的字节数。标准帧和扩展帧的控制段稍有不同，如图 30.1.4 所示：

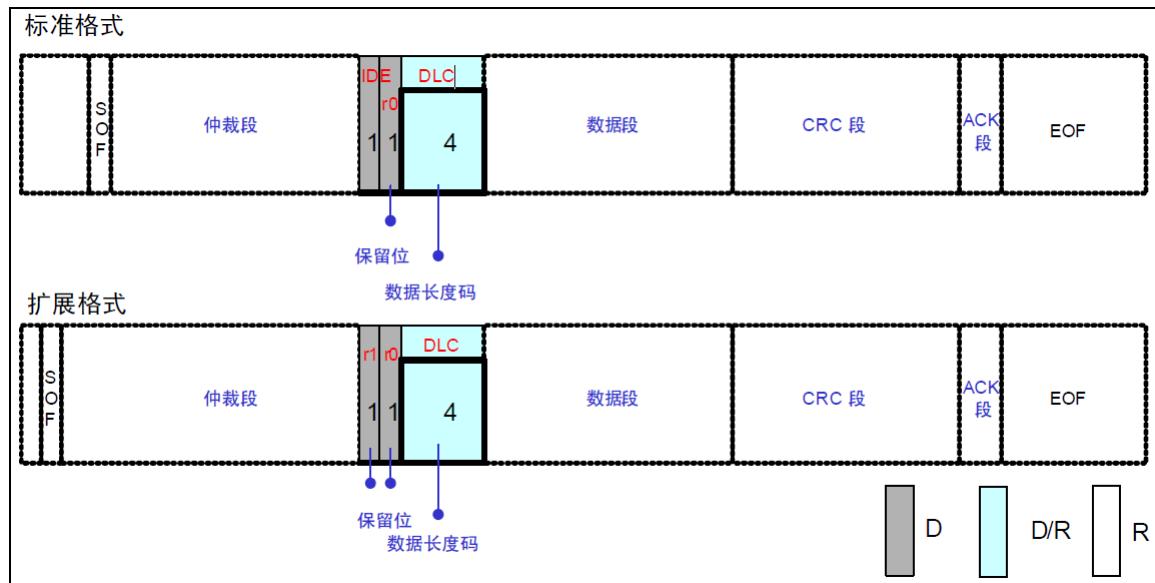


图 30.1.4 数据帧控制段构成

上图中，r0 和 r1 为保留位，必须全部以显性电平发送，但是接收端可以接收显性、隐性及任意组合的电平。DLC 段为数据长度表示段，高位在前，DLC 段有效值为 0~8，但是接收方接收到 9~15 的时候并不认为是错误。

数据段，该段可包含 0~8 个字节的数据。从最高位（MSB）开始输出，标准帧和扩展帧在这个段的定义都是一样的。如图 30.1.5 所示：

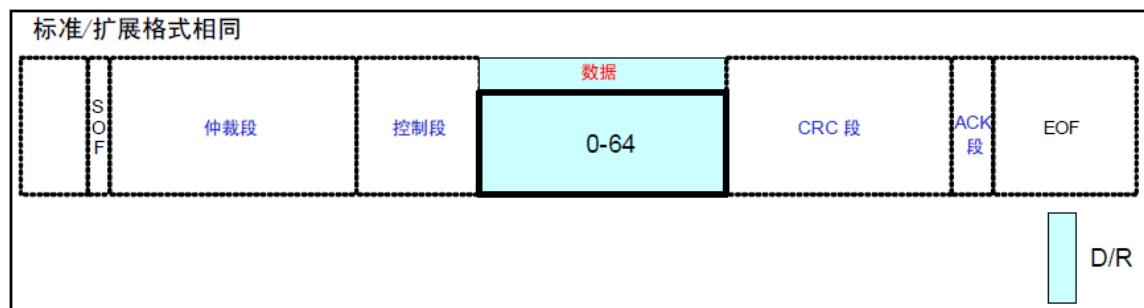


图 30.1.5 数据帧数据段构成

CRC 段，该段用于检查帧传输错误。由 15 个位的 CRC 顺序和 1 个位的 CRC 界定符（用于分隔的位）组成，标准帧和扩展帧在这个段的格式也是相同的。如图 30.1.6 所示：

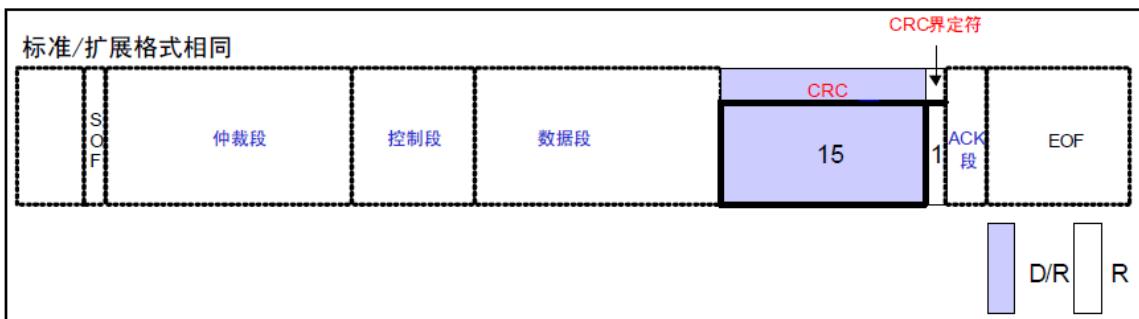


图 30.1.6 数据帧 CRC 段构成

此段 CRC 的值计算范围包括：帧起始、仲裁段、控制段、数据段。接收方以同样的算法计算 CRC 值并进行比较，不一致时会通报错误。

ACK 段，此段用来确认是否正常接收。由 ACK 槽(ACK Slot)和 ACK 界定符 2 个位组成。标准帧和扩展帧在这个段的格式也是相同的。如图 30.1.7 所示：

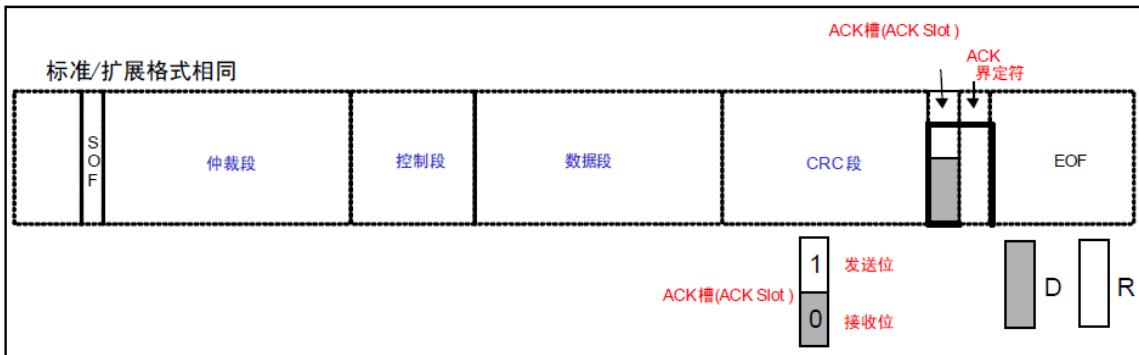


图 30.1.7 数据帧 CRC 段构成

发送单元的 ACK，发送 2 个位的隐性位，而接收到正确消息的单元在 ACK 槽 (ACK Slot) 发送显性位，通知发送单元正常接收结束，这个过程叫发送 ACK/返回 ACK。发送 ACK 的是在既不处于总线关闭态也不处于休眠态的所有接收单元中，接收到正常消息的单元（发送单元不发送 ACK）。所谓正常消息是指不含填充错误、格式错误、CRC 错误的消息。

帧结束，这个段也比较简单，标准帧和扩展帧在这个段格式一样，由 7 个位的隐性位组成。至此，数据帧的 7 个段就介绍完了，其他帧的介绍，请大家参考光盘的 CAN 入门书.pdf 相关章节。接下来，我们再来看看 CAN 的位时序。

由发送单元在非同步的情况下发送的每秒钟的位数称为位速率。一个位可分为 4 段。

- 同步段 (SS)
- 传播时间段 (PTS)
- 相位缓冲段 1 (PBS1)
- 相位缓冲段 2 (PBS2)

这些段又由可称为 Time Quantum (以下称为 Tq) 的最短时间单位构成。

1 位分为 4 个段，每个段又由若干个 Tq 构成，这称为位时序。

1 位由多少个 Tq 构成、每个段又由多少个 Tq 构成等，可以任意设定位时序。通过设定位时序，多个单元可同时采样，也可任意设定采样点。各段的作用和 Tq 数如表 30.1.2 所示：



段名称	段的作用	Tq 数	
同步段 (SS: Synchronization Segment)	多个连接在总线上的单元通过此段实现时序调整，同步进行接收和发送的工作。由隐性电平到显性电平的边沿或由显性电平到隐性电平边沿最好出现在此段中。	1Tq	8~25Tq
传播时间段 (PTS: Propagation Time Segment)	用于吸收网络上的物理延迟的段。 所谓的网络的物理延迟指发送单元的输出延迟、总线上信号的传播延迟、接收单元的输入延迟。 这个段的时间为以上各延迟时间的和的两倍。	1~8Tq	
相位缓冲段 1 (PBS1: Phase Buffer Segment 1)	当信号边沿不能被包含于 SS 段中时，可在此段进行补偿。	1~8Tq	
相位缓冲段 2 (PBS2: Phase Buffer Segment 2)	由于各单元以各自独立的时钟工作，细微的时钟误差会累积起来，PBS 段可用于吸收此误差。 通过对相位缓冲段加减 SJW 吸收误差。 SJW 加大后允许误差加大，但通信速度下降。	2~8Tq	
再同步补偿宽度 (SJW: reSynchronization Jump Width)	因时钟频率偏差、传送延迟等，各单元有同步误差。SJW 为补偿此误差的最大值。	1~4Tq	

表 30.1.2 一个位各段及其作用

1 个位的构成如图 30.1.8 所示：

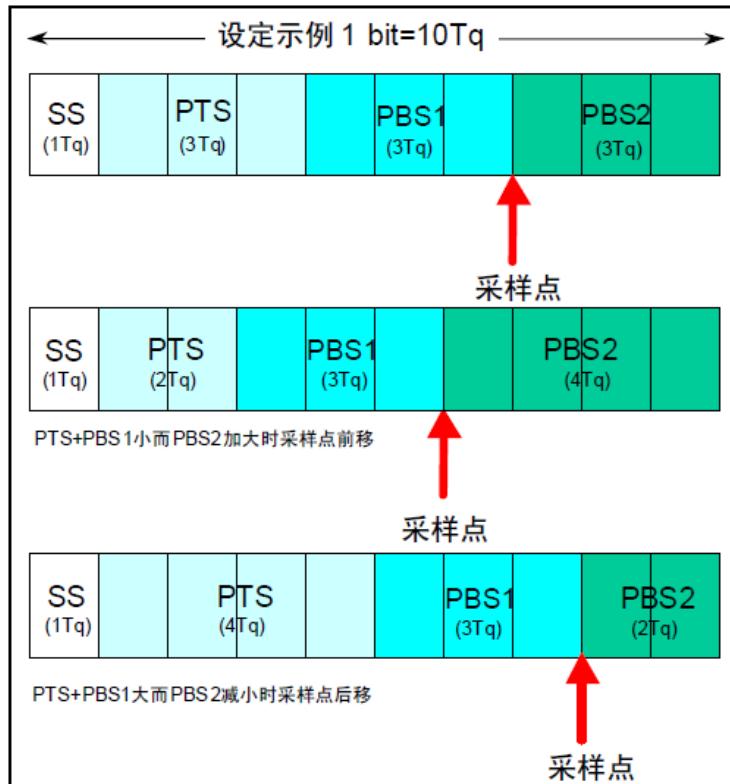


图 30.1.8 一个位的构成



上图的采样点，是指读取总线电平，并将读到的电平作为位值的点。位置在 PBS1 结束处。根据这个位时序，我们就可以计算 CAN 通信的波特率了。具体计算方法，我们等下再介绍，前面提到的 CAN 协议具有仲裁功能，下面我们来看看是如何实现的。

在总线空闲态，最先开始发送消息的单元获得发送权。

当多个单元同时开始发送时，各发送单元从仲裁段的第一位开始进行仲裁。连续输出显性电平最多的单元可继续发送。实现过程，如图 30.1.9 所示：

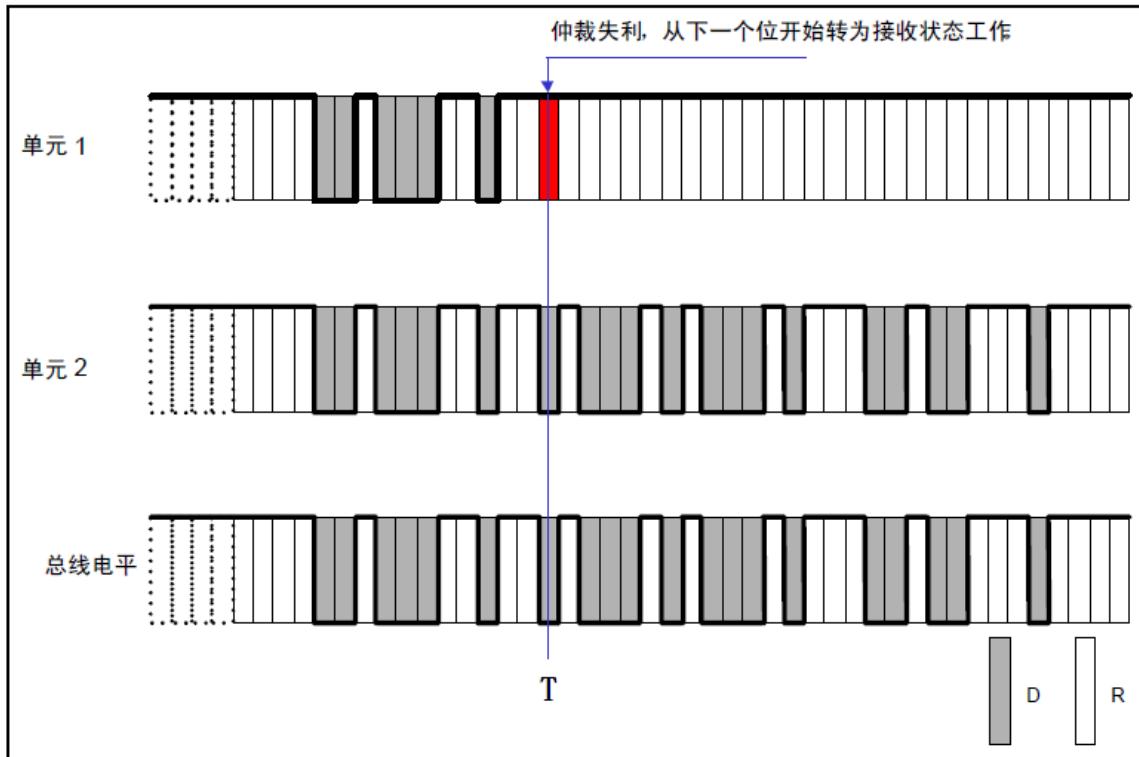


图 30.1.9 CAN 总线仲裁过程

上图中，单元 1 和单元 2 同时开始向总线发送数据，开始部分他们的数据格式是一样的，故无法区分优先级，直到 T 时刻，单元 1 输出隐性电平，而单元 2 输出显性电平，此时单元 1 仲裁失利，立刻转入接收状态工作，不再与单元 2 竞争，而单元 2 则顺利获得总线使用权，继续发送自己的数据。这就实现了仲裁，让连续发送显性电平多的单元获得总线使用权。

通过以上介绍，我们对 CAN 总线有了个大概了解（详细介绍参考光盘的：《CAN 入门书.pdf》），接下来我们介绍下 STM32 的 CAN 控制器。

STM32 自带的是 bxCAN，即基本扩展 CAN。它支持 CAN 协议 2.0A 和 2.0B。它的设计目标是，以最小的 CPU 负荷来高效处理大量收到的报文。它也支持报文发送的优先级要求(优先级特性可软件配置)。对于安全紧要的应用，bxCAN 提供所有支持时间触发通信模式所需的硬件功能。

STM32 的 bxCAN 的主要特点有：

- 支持 CAN 协议 2.0A 和 2.0B 主动模式
- 波特率最高达 1Mbps
- 支持时间触发通信
- 具有 3 个发送邮箱
- 具有 3 级深度的 2 个接收 FIFO
- 可变的过滤器组（最多 28 个）



在 STM32 互聯型產品中，帶有 2 個 CAN 控制器，而我們使用的 STM32F103ZET6 屬于增強型，不是互聯型，只有 1 個 CAN 控制器。雙 CAN 的框圖如圖 30.1.10 所示：

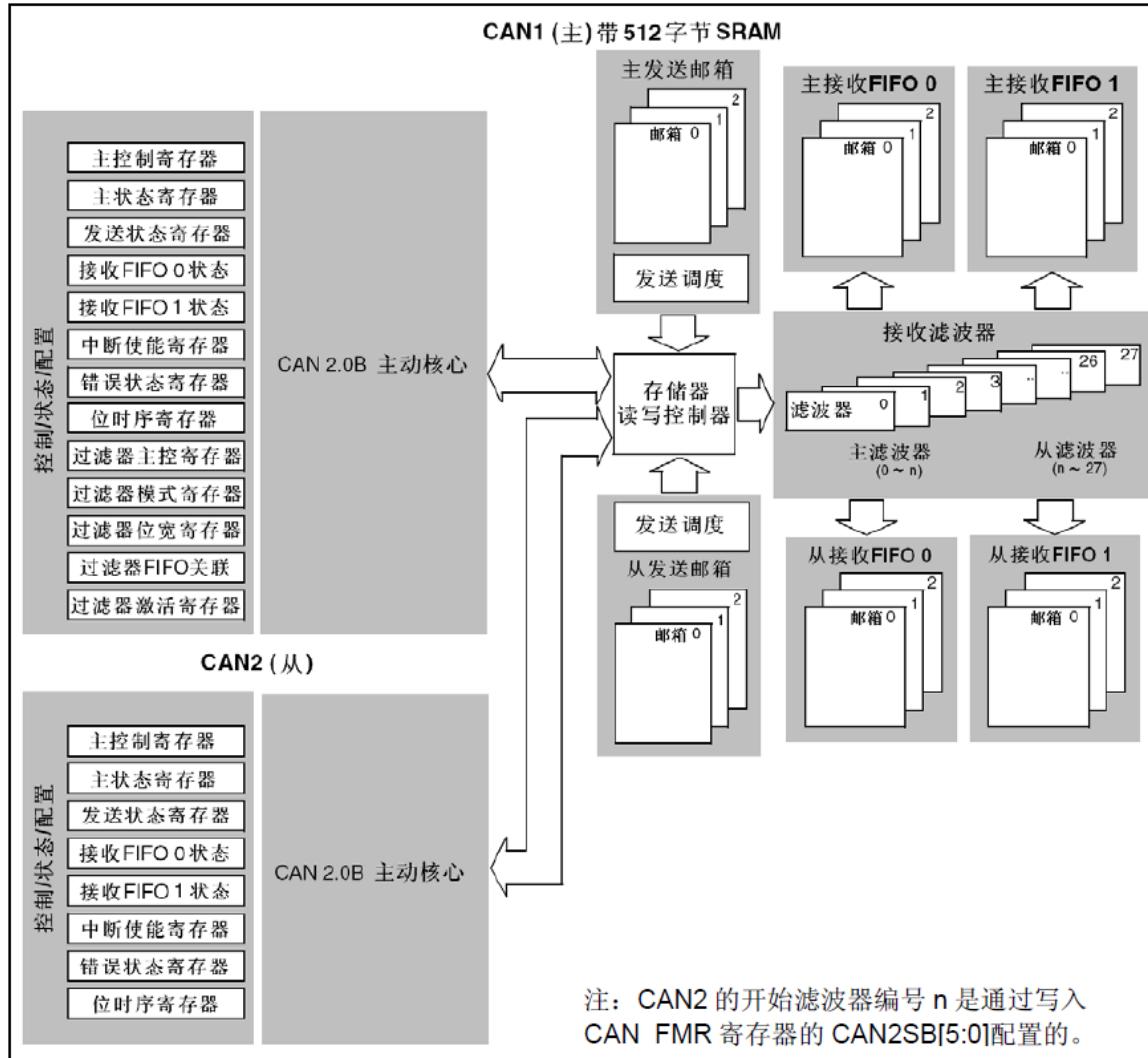


图 30.1.10 双 CAN 框图

从图中可以看出两个 CAN 都分别拥有自己的发送邮箱和接收 FIFO，但是他们共用 28 个滤波器。通过 CAN_FMR 寄存器的设置，可以设置滤波器的分配方式。

STM32 的标识符过滤是一个比较复杂的东东，它的存在减少了 CPU 处理 CAN 通信的开销。STM32 的过滤器组最多有 28 个（互聯型），但是 STM32F103ZET6 只有 14 个（增强型），每个过滤器组 x 由 2 个 32 位寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32 每个过滤器组的位宽都可以独立配置，以满足应用程序的不同需求。根据位宽的不同，每个过滤器组可提供：

- 1 个 32 位过滤器，包括：STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位
- 2 个 16 位过滤器，包括：STDID[10:0]、IDE、RTR 和 EXTID[17:15]位

此外过滤器可配置为，屏蔽位模式和标识符列表模式。

在屏蔽位模式下，标识符寄存器和屏蔽寄存器一起，指定报文标识符的任何一位，应该按照“必须匹配”或“不用关心”处理。

而在标识符列表模式下，屏蔽寄存器也被当作标识符寄存器用。因此，不是采用一个标识符加一个屏蔽位的方式，而是使用 2 个标识符寄存器。接收报文标识符的每一位都必须跟过滤



器标识符相同。

通过 CAN_FMR 寄存器，可以配置过滤器组的位宽和工作模式，如图 30.1.11 所示：

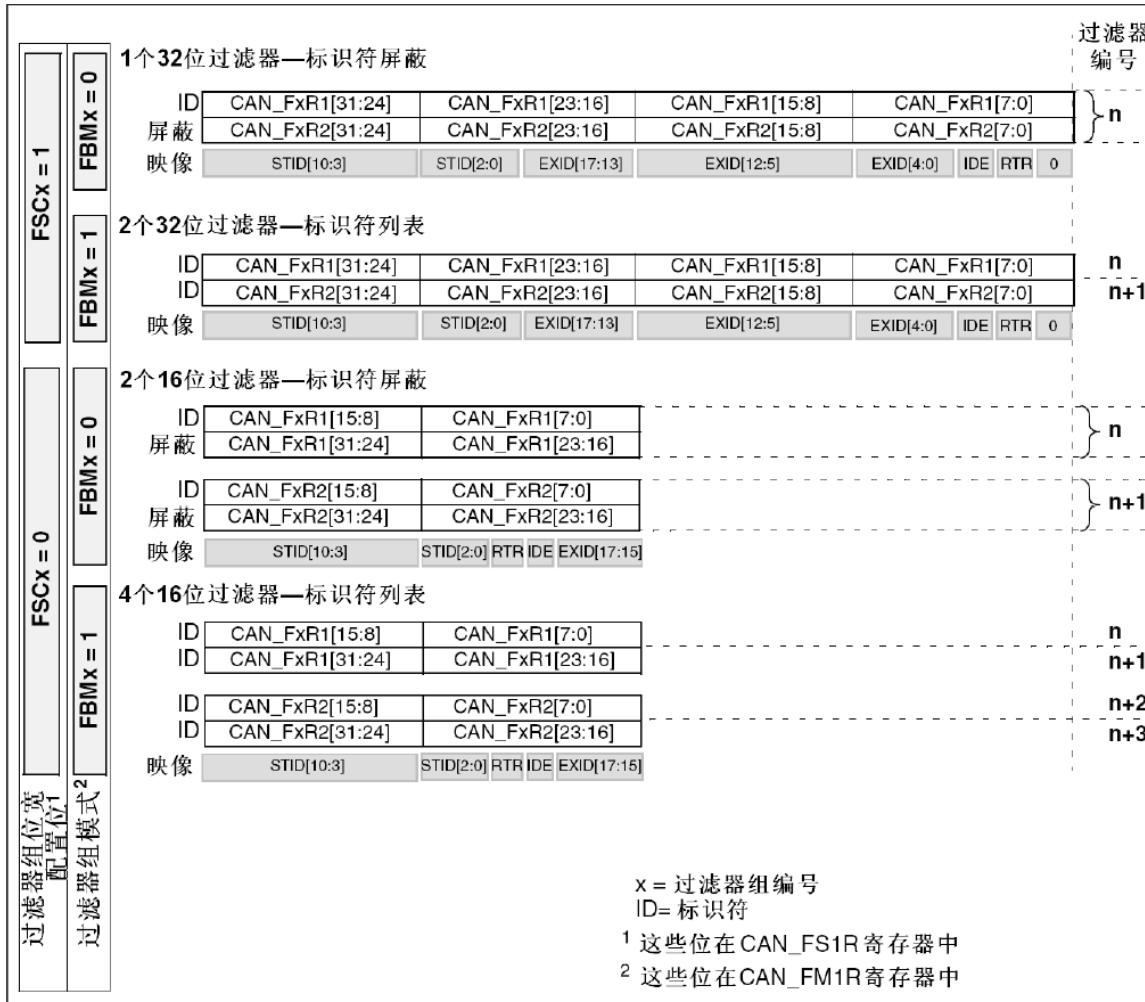


图 30.1.11 过滤器组位宽模式设置

为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。

为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。

应用程序不用的过滤器组，应该保持在禁用状态。

过滤器组中的每个过滤器，都被编号为(叫做过滤器号，图 30.1.11 中的 n)从 0 开始，到某个最大数值—取决于过滤器组的模式和位宽的设置。

举个简单的例子，我们设置过滤器组 0 工作在：1 个 32 位过滤器-标识符屏蔽模式，然后设置 CAN_FOR1=0xFFFF0000, CAN_FOR2=0xFF00FF00。其中存放到 CAN_FOR1 的值就是期望收到的 ID，即我们希望收到的映像 (STID+EXTID+IDE+RTR) 最好是：0xFFFF0000。而 0xFF00FF00 就是设置我们需要必须关心的 ID，表示收到的映像，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_FOR1 中对应的位一模一样，而另外的 16 个位则不关心，可以一样，也可以不一样，都认为是正确的 ID，即收到的映像必须是 0xFFxx00xx，才算是正确的 (x 表示不关心)。

关于标识符过滤的详细介绍，请参考《STM32 参考手册》的 22.7.4 节 (431 页)。接下来，我们看看 STM32 的 CAN 发送和接收的流程。



30.1.1 CAN 发送流程

CAN 发送流程为：程序选择 1 个空置的邮箱（TME=1）→设置标识符（ID），数据长度和发送数据→设置 CAN_TIxR 的 TXRQ 位为 1，请求发送→邮箱挂号（等待成为最高优先级）→预定发送（等待总线空闲）→发送→邮箱空置。整个流程如图 30.1.12 所示：

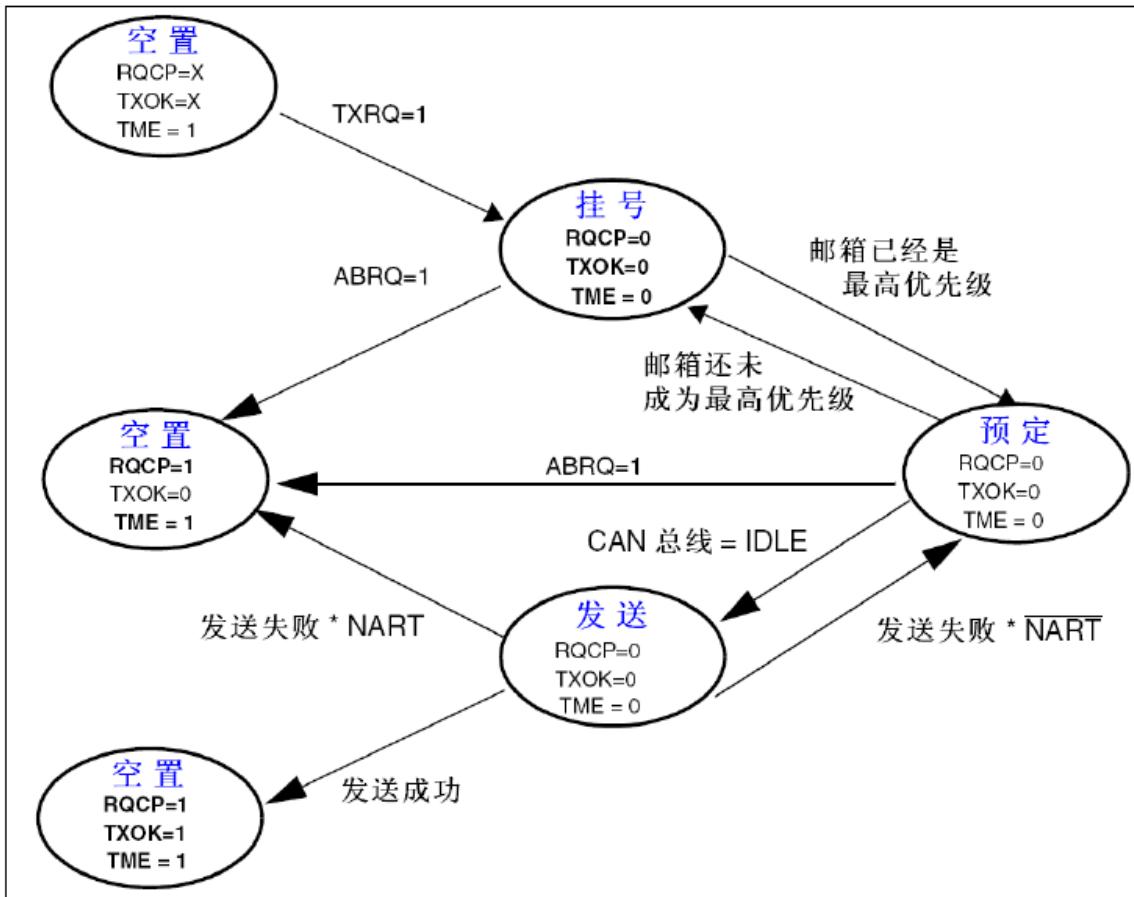


图 30.1.12 发送邮箱

上图中，还包含了很多其他处理，不强制退出发送（ABRQ=1）和发送失败处理等。通过这个流程图，我们大致了解了 CAN 的发送流程，后面的数据发送，我们基本就是按照此流程来走。接下来再看看 CAN 的接收流程。

30.1.2 CAN 接收流程

CAN 接收到的有效报文，被存储在 3 级邮箱深度的 FIFO 中。FIFO 完全由硬件来管理，从而节省了 CPU 的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取 FIFO 输出邮箱，来读取 FIFO 中最先收到的报文。这里的有效报文是指那些正确被接收的（直到 EOF 都没有错误）且通过了标识符过滤的报文。前面我们知道 CAN 的接收有 2 个 FIFO，我们每个滤波器组都可以设置其关联的 FIFO，通过 CAN_FFA1R 的设置，可以将滤波器组关联到 FIFO0/FIFO1。

CAN 接收流程为：FIFO 空→收到有效报文→挂号_1（存入 FIFO 的一个邮箱，这个由硬件控制，我们不需要理会）→收到有效报文→挂号_2→收到有效报文→挂号_3→收到有效报文→溢出。

这个流程里面，我们没有考虑从 FIFO 读出报文的情况，实际情况是：我们必须在 FIFO 溢



出之前，读出至少 1 个报文，否则下个报文到来，将导致 FIFO 溢出，从而出现报文丢失。每读出 1 个报文，相应的挂号就减 1，直到 FIFO 空。CAN 接收流程如图 30.1.13 所示：

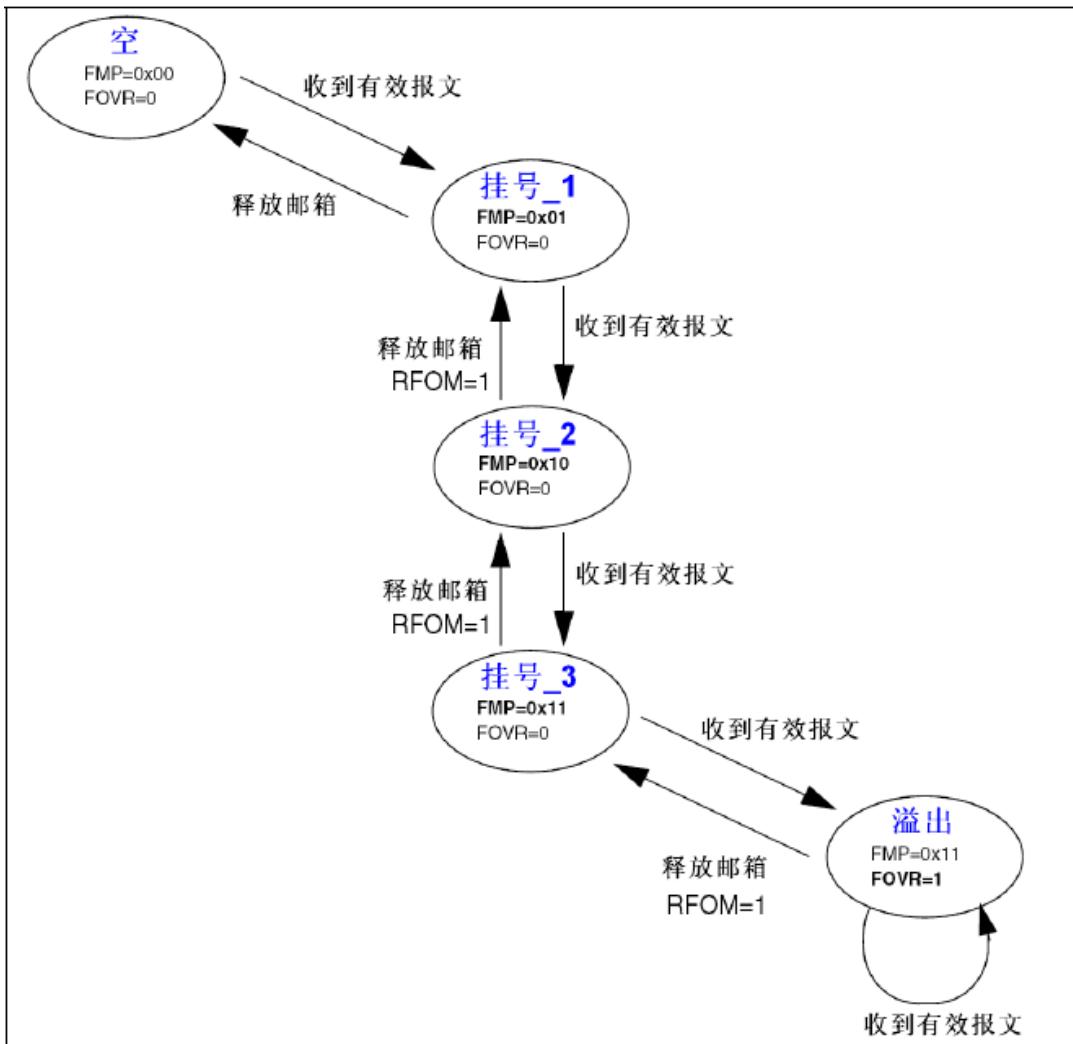


图 30.1.13 FIFO 接收报文

FIFO 接收到的报文数，我们可以通过查询 CAN_RXR 的 FMP 寄存器来得到，只要 FMP 不为 0，我们就可以从 FIFO 读出收到的报文。

接下来，我们简单看看 STM32 的 CAN 位时间特性，STM32 的 CAN 位时间特性和之前我们介绍的，稍有点区别。STM32 把传播时间段和相位缓冲段 1 (STM32 称之为时间段 1) 合并了，所以 STM32 的 CAN 一个位只有 3 段：同步段 (SYNC_SEG)、时间段 1 (BS1) 和时间段 2 (BS2)。STM32 的 BS1 段可以设置为 1~16 个时间单元，刚好等于我们上面介绍的传播时间段和相位缓冲段 1 之和。STM32 的 CAN 位时序如图 30.1.14 所示：

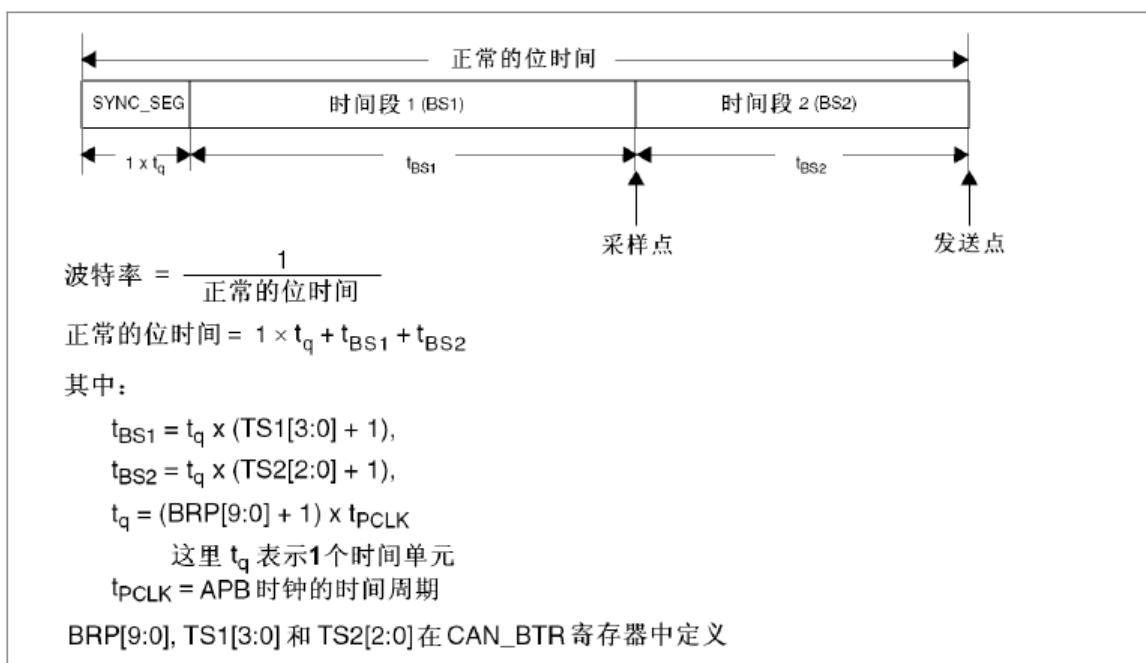


图 30.1.14 STM32 CAN 位时序

图中还给出了 CAN 波特率的计算公式，我们只需要知道 BS1 和 BS2 的设置，以及 APB1 的时钟频率（一般为 36Mhz），就可以方便的计算出波特率。比如设置 TS1=6、TS2=7 和 BRP=4，在 APB1 频率为 36Mhz 的条件下，即可得到 CAN 通信的波特率= $36000 / [(7+8+1)*5] = 450\text{Kbps}$ 。

接下来，我们介绍一下本章需要用到的一些比较重要的寄存器。首先，来看 CAN 的主控制寄存器（CAN_MCR），该寄存器各位描述如图 30.1.15：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留														DBF	
rw															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESET	保留														INRQ
rs	res														rW

图 30.1.15 寄存器 CAN_MCR 各位描述

该寄存器的详细描述，请参考《STM32 参考手册》22.9.2 节（439 页），这里我们仅介绍下 INRQ 位，该位用来控制初始化请求。

软件对该位清 0，可使 CAN 从初始化模式进入正常工作模式：当 CAN 在接收引脚检测到连续的 11 个隐性位后，CAN 就达到同步，并为接收和发送数据作好准备了。为此，硬件相应地对 CAN_MSR 寄存器的 INAK 位置‘0’。

软件对该位置 1 可使 CAN 从正常工作模式进入初始化模式：一旦当前的 CAN 活动(发送或接收)结束，CAN 就进入初始化模式。相应地，硬件对 CAN_MSR 寄存器的 INAK 位置‘1’。

所以我们在 CAN 初始化的时候，先要设置该位为 1，然后进行初始化（尤其是 CAN_BTR 的设置，该寄存器，必须在 CAN 正常工作之前设置），之后再设置该位为 0，让 CAN 进入正常工作模式。

第二个，我们介绍 CAN 位时序寄存器（CAN_BTR），该寄存器用于设置分频、Tbs1、Tbs2 以及 Tsjw 等非常重要的参数，直接决定了 CAN 的波特率。另外该寄存器还可以设置 CAN 的工作模式，该寄存器各位描述如图 30.1.16 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	保留				SJW[1:0]	保留	TS2[2:0]				TS1[3:0]			
rw	rw	res				rw	rw	res	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留				BRP[9:0]											
res				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
位31	SILM: 静默模式(用于调试) (Silent mode (debug)) 0: 正常状态; 1: 静默模式。														
位30	LBKM: 环回模式(用于调试) (Loop back mode (debug)) 0: 禁止环回模式; 1: 允许环回模式。														
位29:26	保留位, 硬件强制为0。														
位25:24	SJW[1:0]: 重新同步跳跃宽度 (Resynchronization jump width) 为了重新同步, 该位域定义了CAN硬件在每位中可以延长或缩短多少个时间单元的上限。 $t_{SJW} = t_{CAN} \times (SJW[1:0] + 1)$ 。														
位23	保留位, 硬件强制为0。														
位22:20	TS2[2:0]: 时间段2 (Time segment 2) 该位域定义了时间段2占用了多少个时间单元 $t_{BS2} = t_{CAN} \times (TS2[2:0] + 1)$ 。														
位19:16	TS1[3:0]: 时间段1 (Time segment 1) 该位域定义了时间段1占用了多少个时间单元 $t_{BS1} = t_{CAN} \times (TS1[3:0] + 1)$ 关于位时间特性的详细信息, 请参考22.7.7节位时间特性。														
位15:10	保留位, 硬件强制其值为0。														
位9:0	BRP[9:0]: 波特率分频器 (Baud rate prescaler) 该位域定义了时间单元(t_q)的时间长度 $t_q = (BRP[9:0]+1) \times t_{PCLK}$														

图 30.1.16 寄存器 CAN_BTR 各位描述

STM32 提供了两种测试模式, 环回模式和静默模式, 当然他们组合还可以组合成环回静默模式。这里我们简单介绍下环回模式。

在环回模式下, bxCAN 把发送的报文当作接收的报文并保存(如果可以通过接收过滤)在接收邮箱里。也就是环回模式是一个自发自收的模式, 如图 30.1.17 所示:

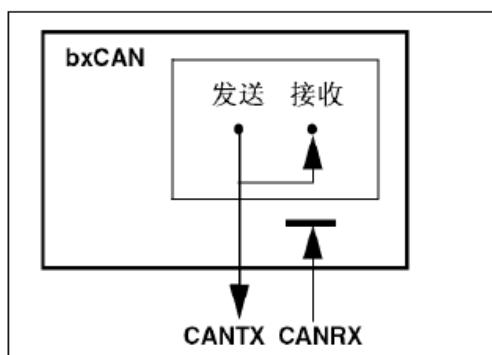


图 30.1.17 CAN 环回模式

环回模式可用于自测试。为了避免外部的影响, 在环回模式下 CAN 内核忽略确认错误(在数据/远程帧的确认位时刻, 不检测是否有显性位)。在环回模式下, bxCAN 在内部把 Tx 输出回馈到 Rx 输入上, 而完全忽略 CANRX 引脚的实际状态。发送的报文可以在 CANTX 引脚上检测到。



第三个，我们介绍 CAN 发送邮箱标识符寄存器 (CAN_TIxR) ($x=0\sim3$)，该寄存器各位描述如图 30.1.18 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STID[10:0]/EXID[28:18]												EXID[17:13]			
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
EXID[12:0]												IDE	RTR	TXRQ	
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位31:21	STID[10:0]/EXID[28:18]: 标准标识符或扩展标识符 (Standard identifier or extended identifier) 依据IDE位的内容，这些位或是标准标识符，或是扩展身份标识的高字节。														
	位20:3 EXID[17:0]: 扩展标识符 (Extended identifier) 扩展身份标识的低字节。														
位2	IDE: 标识符选择 (Identifier extension) 该位决定发送邮箱中报文使用的标识符类型 0: 使用标准标识符； 1: 使用扩展标识符。														
	位1 RTR: 远程发送请求 (Remote transmission request) 0: 数据帧； 1: 远程帧。														
位0	TXRQ: 发送数据请求 (Transmit mailbox request) 由软件对其置'1'，来请求发送邮箱的数据。当数据发送完成，邮箱为空时，硬件对其进行清'0'。														

图 30.1.18 寄存器 CAN_TIxR 各位描述

该寄存器主要用来设置标识符（包括扩展标识符），另外还可以设置帧类型，通过 TXRQ 值 1，来请求邮箱发送。因为有 3 个发送邮箱，所以寄存器 CAN_TIxR 有 3 个。

第四个，我们介绍 CAN 发送邮箱数据长度和时间戳寄存器 (CAN_TDTxR) ($x=0\sim2$)，该寄存器我们本章仅用来设置数据长度，即最低 4 个位。比较简单，这里就不详细介绍了。

第五个，我介绍的是 CAN 发送邮箱低字节数据寄存器 (CAN_TDLxR) ($x=0\sim2$)，该寄存器各位描述如图 30.1.19 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]												DATA2[7:0]			
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
DATA1[7:0]												DATA0[7:0]			
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位31:24	DATA3[7:0]: 数据字节3 (Data byte 3) 报文的数据字节3。														
	位23:16 DATA2[7:0]: 数据字节2 (Data byte 2) 报文的数据字节2。														
位15:8	DATA1[7:0]: 数据字节1 (Data byte 1) 报文的数据字节1。														
	位7:0 DATA0[7:0]: 数据字节0 (Data byte 0) 报文包含0到8个字节数据，且从字节0开始。														

图 30.1.19 寄存器 CAN_TDLxR 各位描述

该寄存器用来存储将要发送的数据，这里只能存储低 4 个字节，另外还有一个寄存器



CAN_TDHzR，该寄存器用来存储高 4 个字节，这样总共就可以存储 8 个字节。CAN_TDHzR 的各位描述同 CAN_TDLzR 类似，我们就不单独介绍了。

第六个，我们介绍 CAN 接收 FIFO 邮箱标识符寄存器 (CAN_RIxR) (x=0/1)，该寄存器各位描述同 CAN_TIxR 寄存器几乎一模一样，只是最低位为保留位，该寄存器用于保存接收到的报文标识符等信息，我们可以通过读该寄存器获取相关信息。

同样的，CAN 接收 FIFO 邮箱数据长度和时间戳寄存器 (CAN_RDTxR)、CAN 接收 FIFO 邮箱低字节数据寄存器 (CAN_RDLxR) 和 CAN 接收 FIFO 邮箱高字节数据寄存器 (CAN_RDHzR) 分别和发送邮箱的：CAN_TDTxR、CAN_TDLxR 以及 CAN_TDHzR 类似，这里我们就不单独一一介绍了。详细介绍，请参考《STM32 参考手册》22.9.3 节 (447 页)。

第七个，我们介绍 CAN 过滤器模式寄存器 (CAN_FM1R)，该寄存器各位描述如图 30.1.20 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FBM15	FBM14	FBM13	FBM12	FBM11	FBM10	FBM9	FBM8	FBM7	FBM6	FBM5	FBM4	FBM3	FBM2	FBM1	FBM0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位31:28		保留位，硬件强制为0													
位13:0		FBMx : 过滤器模式 (Filter mode) 过滤器组x的工作模式。 0: 过滤器组x的2个32位寄存器工作在标识符屏蔽位模式； 1: 过滤器组x的2个32位寄存器工作在标识符列表模式。 注：位27:14只出现在互联型产品中，其它产品为保留位。													

图 30.1.20 寄存器 CAN_FM1R 各位描述

该寄存器用于设置各滤波器组的工作模式，对 28 个滤波器组的工作模式，都可以通过该寄存器设置，不过该寄存器必须在过滤器处于初始化模式下 (CAN_FMR 的 FINIT 位=1)，才可以进行设置。对 STM32F103ZET6 来说，只有[13:0]这 14 个位有效。

第八个，我们介绍 CAN 过滤器位宽寄存器(CAN_FS1R)，该寄存器各位描述如图 30.1.21 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FSC15	FSC14	FSC13	FSC12	FSC11	FSC10	FSC9	FSC8	FSC7	FSC6	FSC5	FSC4	FSC3	FSC2	FSC1	FSC0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位31:28		保留位，硬件强制为0													
位13:0		FSCx : 过滤器位宽设置 (Filter scale configuration) 过滤器组x(13~0)的位宽。 0: 过滤器位宽为2个16位； 1: 过滤器位宽为单个32位。 注：位27:14只出现在互联型产品中，其它产品为保留位。													

图 30.1.21 寄存器 CAN_FS1R 各位描述

该寄存器用于设置各滤波器组的位宽，对 28 个滤波器组的位宽设置，都可以通过该寄存器实现。该寄存器也只能在过滤器处于初始化模式下进行设置。对 STM32F103ZET6 来说，同样



只有[13:0]这 14 个位有效。

第九个，我们介绍 CAN 过滤器 FIFO 关联寄存器 (CAN_FFA1R)，该寄存器各位描述如图 30.1.22 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FFA15	FFA14	FFA13	FFA12	FFA11	FFA10	FFA9	FFA8	FFA7	FFA6	FFA5	FFA4	FFA3	FFA2	FFA1	FFA0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
位31:14		保留位，硬件强制为0。													
位13:0		FFAx : 过滤器位宽设置 (Filter FIFO assignment for filter x) 报文在通过了某过滤器的过滤后，将被存放到其关联的FIFO中。 0: 过滤器被关联到FIFO0; 1: 过滤器被关联到FIFO1。 注：位27:14只出现在互联型产品中，其它产品为保留位。													

图 30.1.22 寄存器 CAN_FFA1R 各位描述

该寄存器设置报文通过滤波器组之后，被存入的 FIFO，如果对应位为 0，则存放到 FIFO0；如果为 1，则存放到 FIFO1。该寄存器也只能在过滤器处于初始化模式下配置。

第十个，我们介绍 CAN 过滤器激活寄存器 (CAN_FA1R)，该寄存器各位对应滤波器组和前面的几个寄存器类似，这里就不列出了，对对应位置 1，即开启对应的滤波器组；置 0 则关闭该滤波器组。

最后，我们介绍 CAN 的过滤器组 i 的寄存器 x (CAN_FiRx) (互联产品中 $i=0\sim27$ ，其它产品中 $i=0\sim13$ ； $x=1/2$)。该寄存器各位描述如图 30.1.23 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FB31	FB30	FB29	FB28	FB27	FB26	FB25	FB24	FB23	FB22	FB21	FB20	FB19	FB18	FB17	FB16
RW															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FB15	FB14	FB13	FB12	FB11	FB10	FB9	FB8	FB7	FB6	FB5	FB4	FB3	FB2	FB1	FB0
RW															

在所有的配置情况下：

位31:0	FB[31:0] : 过滤器位 (Filter bits) 标识符模式 寄存器的每位对应于所期望的标识符的相应位的电平。 0: 期望相应位为显性位； 1: 期望相应位为隐性位。 屏蔽位模式 寄存器的每位指示是否对应的标识符寄存器位一定要与期望的标识符的相应位一致。 0: 不关心，该位不用于比较； 1: 必须匹配，到来的标识符位必须与滤波器对应的标识符寄存器位相一致。
-------	---

图 30.1.23 寄存器 CAN_FiRx 各位描述

每个滤波器组的 CAN_FiRx 都由 2 个 32 位寄存器构成，即：CAN_FiR1 和 CAN_FiR2。根据过滤器位宽和模式的不同设置，这两个寄存器的功能也不尽相同。关于过滤器的映射，功能描述和屏蔽寄存器的关联，请参见图 30.1.11。

关于 CAN 的介绍，就到此结束了。接下来，我们看看本章我们将实现的功能，及 CAN 的配置步骤。

本章，我们通过 WK_UP 按键选择 CAN 的工作模式(正常模式/环回模式)，然后通过 KEY0



控制数据发送，并通过查询的办法，将接收到的数据显示在 LCD 模块上。如果是环回模式，我们不需要 2 个开发板。如果是正常模式，我们就需要 2 个战舰开发板，并且将他们的 CAN 接口对接起来，然后一个开发板发送数据，另外一个开发板将接收到的数据显示在 LCD 模块上。

最后，我们来看看本章的 CAN 的初始化配置步骤,CAN 相关的固件库函数和定义分布在文件 `stm32f10x_can.c` 和头文件 `stm32f10x_can.h` 文件中。

1) 配置相关引脚的复用功能，使能 CAN 时钟。

我们要用 CAN，第一步就要使能 CAN 的时钟。其次要设置 CAN 的相关引脚为复用输出，这里我们需要设置 PA11 为上拉输入（CAN_RX 引脚）PA12 为复用输出（CAN_TX 引脚），并使能 PA 口的时钟。使能 CAN1 时钟的函数是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); //使能 CAN1 时钟
```

对于 GPIO 时钟的使能以及初始化配置这里就不多讲解，前面都有提到，很容易组织代码。

2) 设置 CAN 工作模式及波特率等。

这一步通过先设置 CAN_MCR 寄存器的 INRQ 位，让 CAN 进入初始化模式，然后设置 CAN_MCR 的其他相关控制位。再通过 CAN_BTR 设置波特率和工作模式(正常模式/环回模式)等信息。最后设置 INRQ 为 0，退出初始化模式。

在库函数中，提供了函数 `CAN_Init()`用来初始化 CAN 的工作模式以及波特率，`CAN_Init()` 函数体中，在初始化之前，会设置 CAN_MCR 寄存器的 INRQ 为 1 让其进入初始化模式，然后初始化 CAN_MCR 寄存器和 CRN_BTR 寄存器之后，会设置 CAN_MCR 寄存器的 INRQ 为 0 让其退出初始化模式。所以我们在调用这个函数的前后不需要再进行初始化模式设置。下面我们来看看 `CAN_Init()` 函数的定义：

```
uint8_t CAN_Init(CAN_TypeDef* CANx, CAN_InitTypeDef* CAN_InitStruct);
```

第一个参数就是 CAN 标号，这里我们的芯片只有一个 CAN，所以就是 CAN1。

第二个参数是 CAN 初始化结构体指针，结构体类型是 `CAN_InitTypeDef`，下面我们来看看这个结构体的定义：

```
typedef struct
{
    uint16_t CAN_Prescaler;
    uint8_t CAN_Mode;
    uint8_t CAN_SJW;
    uint8_t CAN_BS1;
    uint8_t CAN_BS2;
    FunctionalState CAN_TTCM;
    FunctionalState CAN_ABOM;
    FunctionalState CAN_AWUM;
    FunctionalState CAN_NART;
    FunctionalState CAN_RFLM;
    FunctionalState CAN_TXFP;
} CAN_InitTypeDef;
```

这个结构体看起来成员变量比较多，实际上参数可以分为两类。前面 5 个参数是用来设置寄存器 `CAN_BTR`，用来设置模式以及波特率相关的参数，这在前面有讲解过，设置模式的参数是 `CAN_Mode`，我们实验中用到回环模式 `CAN_Mode_LoopBack` 和常规模式 `CAN_Mode_Normal`，大家还可以选择静默模式以及静默回环模式测试。其他设置波特率相关的参数 `CAN_Prescaler`, `CAN_SJW`, `CAN_BS1` 和 `CAN_BS2` 分别用来设置波特率分频器，重新同步跳跃宽度以及时间



段 1 和时间段 2 占用的时间单元数。后面 6 个成员变量用来设置寄存器 CAN_MCR，也就是设置 CAN 通信相关的控制位。大家可以去翻翻中文参考手册中这两个寄存器的描述，非常详细，我们在前面也有讲解到。初始化实例为：

```

CAN_InitStructure.CAN_TTCM=DISABLE;           //非时间触发通信模式
CAN_InitStructure.CAN_ABOM=DISABLE;           //软件自动离线管理
CAN_InitStructure.CAN_AWUM=DISABLE;           //睡眠模式通过软件唤醒
CAN_InitStructure.CAN_NART=ENABLE;            //禁止报文自动传送
CAN_InitStructure.CAN_RFLM=DISABLE;           //报文不锁定,新的覆盖旧的
CAN_InitStructure.CAN_TXFP=DISABLE;           //优先级由报文标识符决定
CAN_InitStructure.CAN_Mode= CAN_Mode_LoopBack; //模式设置： 1,回环模式;
//设置波特率
CAN_InitStructure.CAN_SJW=CAN_SJW_1tq;//重新同步跳跃宽度为个时间单位
CAN_InitStructure.CAN_BS1=CAN_BS1_8tq; //时间段 1 占用 8 个时间单位
CAN_InitStructure.CAN_BS2=CAN_BS2_7tq;//时间段 2 占用 7 个时间单位
CAN_InitStructure.CAN_Prescaler=5; //分频系数(Fdiv)
CAN_Init(CAN1, &CAN_InitStructure);          // 初始化 CAN1

```

3) 设置滤波器。

本章，我们将使用滤波器组 0，并工作在 32 位标识符屏蔽位模式下。先设置 CAN_FMR 的 FINIT 位，让过滤器组工作在初始化模式下，然后设置滤波器组 0 的工作模式以及标识符 ID 和屏蔽位。最后激活滤波器，并退出滤波器初始化模式。

在库函数中，提供了函数 CAN_FilterInit()用来初始化 CAN 的滤波器相关参数，CAN_Init()函数体中，在初始化之前，会设置 CAN_FMR 寄存器的 INRQ 为 INIT 让其进入初始化模式，然后初始化 CAN 滤波器相关的寄存器之后，会设置 CAN_FMR 寄存器的 FINIT 为 0 让其退出初始化模式。所以我们在调用这个函数的前后不需要再进行初始化模式设置。下面我们来看看 CAN_FilterInit()函数的定义：

```
void CAN_FilterInit(CAN_FilterInitTypeDef* CAN_FilterInitStruct);
```

这个函数只有一个入口参数就是 CAN 滤波器初始化结构体指针，结构体类型为 CAN_FilterInitTypeDef，下面我们看看类型定义：

```

typedef struct
{
    uint16_t CAN_FilterIdHigh;
    uint16_t CAN_FilterIdLow;
    uint16_t CAN_FilterMaskIdHigh;
    uint16_t CAN_FilterMaskIdLow;
    uint16_t CAN_FilterFIFOAssignment;
    uint8_t CAN_FilterNumber;
    uint8_t CAN_FilterMode;
    uint8_t CAN_FilterScale;
    FunctionalState CAN_FilterActivation;
} CAN_FilterInitTypeDef;

```

结构体一共有 9 个成员变量，第 1 个至第 4 个是用来设置过滤器的 32 位 id 以及 32 位 mask id，分别通过 2 个 16 位来组合的，这个在前面有讲解过它们的意义。

第 5 个成员变量 CAN_FilterFIFOAssignment 用来设置 FIFO 和过滤器的关联关系，我们的实验



是关联的过滤器 0 到 FIFO0，值为 CAN_Filter_FIFO0。

第 6 个成员变量 CAN_FilterNumber 用来设置初始化的过滤器组，取值范围为 0~13。

第 7 个成员变量 FilterMode 用来设置过滤器组的模式，取值为标识符列表模式

CAN_FilterMode_IdList 和标识符屏蔽位模式 CAN_FilterMode_IdMask。

第 8 个成员变量 FilterScale 用来设置过滤器的位宽为 2 个 16 位 CAN_FilterScale_16bit 还是 1 个 32 位 CAN_FilterScale_32bit。

第 9 个成员变量 CAN_FilterActivation 就很明了了，用来激活该过滤器。

过滤器初始化参考实例代码：

```
CAN_FilterInitStructure.CAN_FilterNumber=0; //过滤器 0
CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit; //32 位
CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000;///32 位 ID
CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000;//32 位 MASK
CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;
CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO0;// FIFO0
CAN_FilterInitStructure.CAN_FilterActivation=ENABLE; //激活过滤器 0
CAN_FilterInit(&CAN_FilterInitStructure); //滤波器初始化
```

至此，CAN 就可以开始正常工作了。如果用到中断，就还需要进行中断相关的配置，本章因为没用到中断，所以就不作介绍了。

4) 发送接受消息

在初始化 CAN 相关参数以及过滤器之后，接下来就是发送和接收消息了。库函数中提供了发送和接受消息的函数。发送消息的函数是：

```
uint8_t CAN_Transmit(CAN_TypeDef* CANx, CanTxMsg* TxMessage);
```

这个函数比较好理解，第一个参数是 CAN 标号，我们使用 CAN1。第二个参数是相关消息结构体 CanTxMsg 指针类型，CanTxMsg 结构体的成员变量用来设置标准标识符，扩展标识符，消息类型和消息帧长度等信息。

接受消息的函数是：

```
void CAN_Receive(CAN_TypeDef* CANx, uint8_t FIFOIndex, CanRxMsg* RxMessage);
```

前面两个参数也比较好理解，CAN 标号和 FIFO 号。第二个参数 RxMessage 是用来存放接收到的消息信息。

结构体 CanRxMsg 和结构体 CanTxMsg 比较接近，分别用来定义发送消息和描述接受消息，大家可以对照看一下，也比较好理解。

5) CAN 状态获取

对于 CAN 发送消息的状态，挂起消息数目等等之类的传输状态信息，库函数提供了一些列的函数，包括 CAN_TransmitStatus() 函数，CAN_MessagePending() 函数，CAN_GetFlagStatus() 函数等等，大家可以根据需要来调用。



30.2 硬件设计

本章要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) KEY0 和 WK_UP 按键
- 3) TFTLCD 模块
- 4) CAN
- 5) CAN 收发芯片 JTA1050

前面 3 个之前都已经详细介绍过了，这里我们介绍 STM32 与 TJA1050 连接关系，如图 30.2.1 所示：

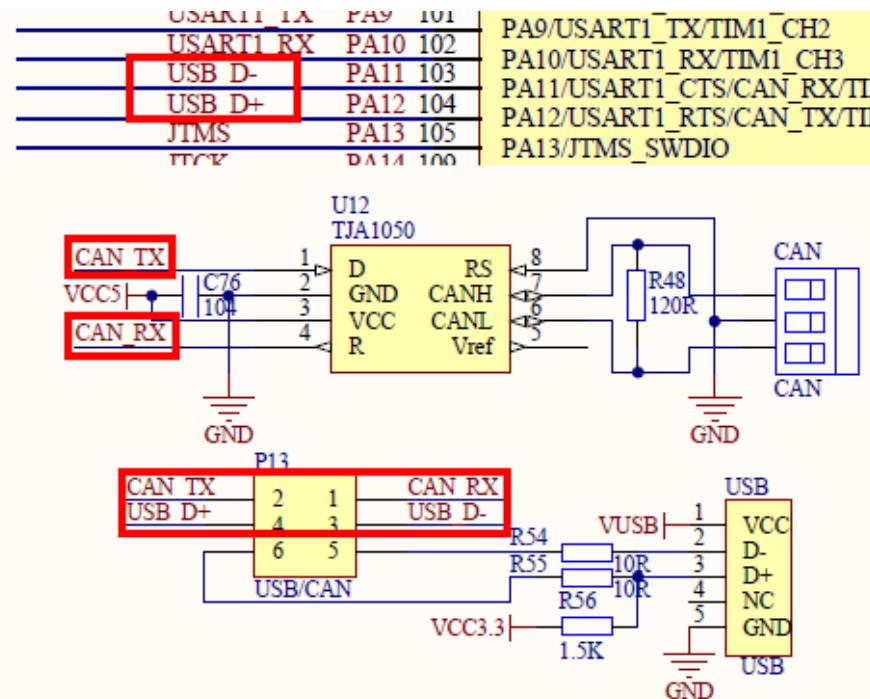


图 30.2.1 STM32 与 TJA1050 连接电路图

从上图可以看出：STM32 的 CAN 通过 P13 的设置，连接到 TJA1050 收发芯片，然后通过接线端子（CAN）同外部的 CAN 总线连接。图中可以看出，在战舰 STM32 开发板上面是带有 120Ω 的终端电阻的，如果我们的开发板不是作为 CAN 的终端的话，需要把这个电阻去掉，以免影响通信。

这里还要注意，我们要设置好开发板上 P13 排针的连接，通过跳线帽将 PA11 和 PA12 分别连接到 CRX (CAN_RX) 和 CTX (CAN_TX) 上面，如图 30.2.2 所示：

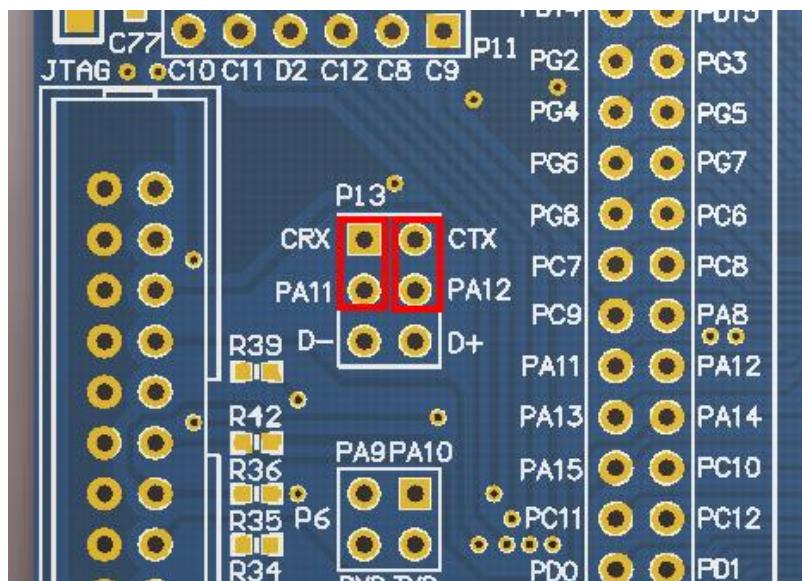


图 30.2.2 硬件连接示意图

最后，我们用 2 根导线将两个开发板 CAN 端子的 CAN_L 和 CAN_L, CAN_H 和 CAN_H 连接起来。这里注意不要接反了 (CAN_L 接 CAN_H)，接反了会导致通讯异常！！

30.3 软件设计

打开 CAN 通信实验的工程可以看到，我们增加了文件 can.c 以及头文件 can.h，同时 CAN 相关的固件库函数和定义分布在文件 stm32f10x_can.c 和头文件 stm32f10x_can.h 中。

打开 can.c 文件，代码如下：

```
#include "can.h"
#include "led.h"
#include "delay.h"
#include "usart.h"

//CAN 初始化
//tsjw:重新同步跳跃时间单元. CAN_SJW_1tq~CAN_SJW_4tq
//tbs2:时间段 2 的时间单元.
//tbs1:时间段 1 的时间单元 CAN_BS1_1tq ~CAN_BS1_16tq
//brp :波特率分频器.范围:1~1024;(实际要加 1,也就是 1~1024) tq=(brp)*tpclk1
//注意以上参数任何一个都不能设为 0,否则会乱.
//波特率=Fpclk1/((tsjw+tbs1+tbs2)*brp);
//mode:0,普通模式;1,回环模式;
//Fpclk1 的时钟在初始化的时候设置为 36M,如果设置 CAN_Normal_Init(1,8,7,5,1);
//则波特率为:36M/((1+8+7)*5)=450Kbps
//返回值:0,初始化 OK;
// 其他,初始化失败;
u8 CAN_Mode_Init(u8 tsjw,u8 tbs2,u8 tbs1,u16 brp,u8 mode)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    CAN_InitTypeDef      CAN_InitStructure;
```



```
CAN_FilterInitTypeDef CAN_FilterInitStructure;
#if CAN_RX0_INT_ENABLE
    NVIC_InitTypeDef NVIC_InitStructure;
#endif
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能 PORTA 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); //使能 CAN1 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 IO

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 IO

//CAN 单元设置
    CAN_InitStructure.CAN_TTCM=DISABLE; //非时间触发通信模式
    CAN_InitStructure.CAN_ABOM=DISABLE; //软件自动离线管理
    CAN_InitStructure.CAN_AWUM=DISABLE; //睡眠模式通过软件唤醒
    CAN_InitStructure.CAN_NART=ENABLE; //禁止报文自动传送
    CAN_InitStructure.CAN_RFLM=DISABLE; //报文不锁定,新的覆盖旧的
    CAN_InitStructure.CAN_TXFP=DISABLE; //优先级由报文标识符决定
    CAN_InitStructure.CAN_Mode= mode; //模式设置: 0,普通模式;1,回环模式;
//设置波特率
    CAN_InitStructure.CAN_SJW=tsjw; //重新同步跳跃宽度(Tsjw)
    CAN_InitStructure.CAN_BS1=tbs1; //时间段 1 占用时间单位
    CAN_InitStructure.CAN_BS2=tbs2; //时间段 3 占用时间单位
    CAN_InitStructure.CAN_Prescaler=brp; //分频系数(Fdiv)为 brp+1
    CAN_Init(CAN1, &CAN_InitStructure); // 初始化 CAN1

    CAN_FilterInitStructure.CAN_FilterNumber=0; //过滤器 0
    CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit; //32 位
    CAN_FilterInitStructure.CAN_FilterIdHigh=0x0000; //32 位 ID
    CAN_FilterInitStructure.CAN_FilterIdLow=0x0000;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh=0x0000; //32 位 MASK
    CAN_FilterInitStructure.CAN_FilterMaskIdLow=0x0000;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_FIFO0; // FIFO0
    CAN_FilterInitStructure.CAN_FilterActivation=ENABLE; //激活过滤器 0
    CAN_FilterInit(&CAN_FilterInitStructure); //滤波器初始化

#endif CAN_RX0_INT_ENABLE
```



```
CAN_ITConfig(CAN1,CAN_IT_FMP0,ENABLE);           //FIFO0 消息挂号中断允许.
NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0 IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;      // 主优先级为 1
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;            // 次优先级为 0
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

#endif
return 0;
}

#if CAN_RX0_INT_ENABLE    //使能 RX0 中断
//中断服务函数
void USB_LP_CAN1_RX0_IRQHandler(void)
{
    CanRxMsg RxMessage;
    int i=0;
    CAN_Receive(CAN1, 0, &RxMessage);
    for(i=0;i<8;i++)
        printf("rxbuf[%d]:%d\r\n",i,RxMessage.Data[i]);
}
#endif

//can 发送一组数据(固定格式:ID 为 0X12,标准帧,数据帧)
//len:数据长度(最大为 8)
//msg:数据指针,最大为 8 个字节.
//返回值:0,成功;
//        其他,失败;
u8 Can_Send_Msg(u8* msg,u8 len)
{
    u8 mbox;
    u16 i=0;
    CanTxMsg TxMessage;
    TxMessage.StdId=0x12;                      // 标准标识符为 0
    TxMessage.ExtId=0x12;                      // 设置扩展标示符 (29 位)
    TxMessage.IDE=0;                           // 使用扩展标识符
    TxMessage.RTR=0;                          // 消息类型为数据帧, 一帧 8 位
    TxMessage.DLC=len;                         // 发送两帧信息
    for(i=0;i<len;i++)                        // 第一帧信息
        TxMessage.Data[i]=msg[i];
    mbox= CAN_Transmit(CAN1, &TxMessage);
    i=0;
    while((CAN_TransmitStatus(CAN1, mbox)!= CAN_TxStatus_Ok)&&(i<0xffff))i++; //等待
                                                //发送结束
    if(i>=0xffff)return 1;
    return 0;
}
```



```
}

//can 口接收数据查询
//buf:数据缓存区;
//返回值:0,无数据被收到;
//          其他,接收的数据长度;
u8 Can_Receive_Msg(u8 *buf)
{
    u32 i;
    CanRxMsg RxMessage;
    if( CAN_MessagePending(CAN1,CAN_FIFO0)==0) return 0;//没有接收到数据,直接退出
    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);           //读取数据
    for(i=0;i<8;i++)
        buf[i]=RxMessage.Data[i];
    return RxMessage.DLC;
}
```

此部分代码总共 3 个函数，首先是 CAN_Mode_Init 函数。该函数用于 CAN 的初始化，该函数带有 5 个参数，可以设置 CAN 通信的波特率和工作模式等，在该函数中，我们就是按 30.1 节末尾的介绍来初始化的，本章中，我们设计滤波器组 0 工作在 32 位标识符屏蔽模式，从设计值可以看出，该滤波器是不会对任何标识符进行过滤的，因为所有的标识符位都被设置成不需要关心，这样设计，主要是方便大家实验。

第二个函数，Can_Send_Msg 函数。该函数用于 CAN 报文的发送，主要是设置标识符 ID 等信息，写入数据长度和数据，并请求发送，实现一次报文的发送。

第三个函数，Can_Receive_Msg 函数。用来接受数据并且将接受到的数据存放到 buf 中。

can.c 里面，还包含了中断接收的配置，通过 can.h 的 CAN_RX0_INT_ENABLE 宏定义，来配置是否使能中断接收，本章我们不开启中断接收的。其他函数我们就不一一介绍了，都比较简单，大家自行理解即可。

最后，我们来看看 main.c 文件的内容：

```
int main(void)
{
    u8 key;
    u8 i=0,t=0;
    u8 cnt=0;
    u8 canbuf[8];
    u8 res;
    u8 mode=CAN_Mode_LoopBack;//CAN 工作模式;CAN_Mode_Normal(0):
                                //普通模式, CAN_Mode_LoopBack(1): 环回模式
    delay_init();             //延时函数初始化
    NVIC_Configuration();     //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);          //串口初始化波特率为 9600
    LED_Init();                //初始化与 LED 连接的硬件接口
    LCD_Init();                //初始化 LCD
```



```
KEY_Init();           //按键初始化

CAN_Mode_Init(CAN_SJW_1tq,CAN_BS2_8tq,CAN_BS1_7tq,5,
              CAN_Mode_LoopBack); //CAN 初始化环回模式,波特率 450Kbps
POINT_COLOR=RED;      //设置字体为红色
LCD_ShowString(60,50,200,16,16,"WarShip STM32");
LCD_ShowString(60,70,200,16,16,"CAN TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/11");
LCD_ShowString(60,130,200,16,16,"LoopBack Mode");
LCD_ShowString(60,150,200,16,16,"KEY0:Send WK_UP:Mode"); //显示提示信息
POINT_COLOR=BLUE;          //设置字体为蓝色
LCD_ShowString(60,170,200,16,16,"Count:");           //显示当前计数值
LCD_ShowString(60,190,200,16,16,"Send Data:");        //提示发送的数据
LCD_ShowString(60,250,200,16,16,"Receive Data:");     //提示接收到的数据
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY_RIGHT) //KEY0 按下,发送一次数据
    {
        for(i=0;i<8;i++)
        {
            canbuf[i]=cnt+i; //填充发送缓冲区
            if(i<4)LCD_ShowxNum(60+i*32,210,canbuf[i],3,16,0X80); //显示数据
            else LCD_ShowxNum(60+(i-4)*32,230,canbuf[i],3,16,0X80); //显示数据
        }
        res=Can_Send_Msg(canbuf,8); //发送 8 个字节
        if(res)LCD_ShowString(60+80,190,200,16,16,"Failed"); //提示发送失败
        else LCD_ShowString(60+80,190,200,16,16,"OK "); //提示发送成功
    }
    }else if(key==KEY_UP) //WK_UP 按下, 改变 CAN 的工作模式
    {
        mode=!mode;
        CAN_Mode_Init(CAN_SJW_1tq,CAN_BS2_8tq,CAN_BS1_7tq,5,mode);
                    //CAN 普通模式初始化, 波特率 450Kbps
        POINT_COLOR=RED; //设置字体为红色
        if(mode==0) //普通模式, 需要 2 个开发板
        {
            LCD_ShowString(60,130,200,16,16,"Normal Mode ");
        }else //回环模式,一个开发板就可以测试了.
        {
            LCD_ShowString(60,130,200,16,16,"LoopBack Mode");
        }
    }
}
```



```
POINT_COLOR=BLUE;           //设置字体为蓝色
}
key=Can_Receive_Msg(canbuf);
if(key)                     //接收到有数据
{
    LCD_Fill(60,270,130,310,WHITE); //清除之前的显示
    for(i=0;i<key;i++)
    {
        if(i<4)LCD_ShowxNum(60+i*32,270,canbuf[i],3,16,0X80); //显示数据
        else LCD_ShowxNum(60+(i-4)*32,290,canbuf[i],3,16,0X80); //显示数据
    }
    t++;
    delay_ms(10);
    if(t==20)
    {
        LED0=!LED0;           //提示系统正在运行
        t=0;
        cnt++;
        LCD_ShowxNum(60+48,170,cnt,3,16,0X80); //显示数据
    }
}
}
```

此部分代码，我们主要关注下

CAN_Mode_Init(CAN_SJW_1tq,CAN_BS1_8tq,CAN_BS2_7tq,5,CAN_Mode_LoopBack); 该函数用于设置波特率和 CAN 的模式，根据前面的波特率计算公式，我们知道这里的波特率被初始化为 450Kbps。mode 参数用于设置 CAN 的工作模式（正常模式/环回模式），通过 WK_UP 按键，可以随时切换模式。cnt 是一个累加数，一旦 KEY_RIGHT (KEY0) 按下，就以这个数位基准连续发送 5 个数据。当 CAN 总线收到数据的时候，就将收到的数据直接显示在 LCD 屏幕上。

30.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 30.4.1 所示：

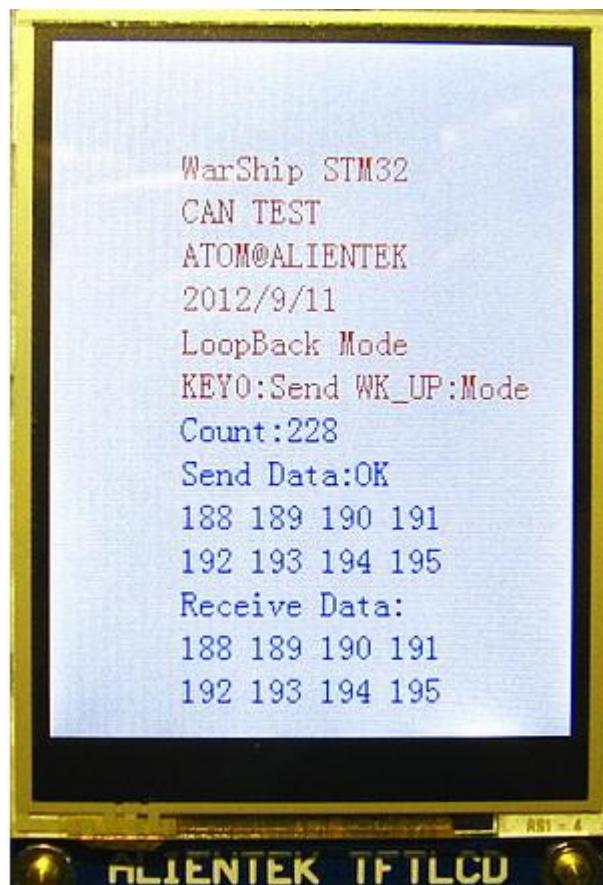


图 30.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。默认我们是设置的环回模式，此时，我们按下 KEY0 就可以在 LCD 模块上面看到自发自收的数据（如上图所示），如果我们选择正常模式（通过 WK_UP 按键切换），就必须连接两个开发板的 CAN 接口，然后就可以互发数据了。如图 30.4.2 所示：



图 30.4.2 CAN 普通模式数据收发测试

上图中，左侧的图片来自开发板 A，发送了 8 个数据，右侧来自开发板 B，收到了来自开发板 A 的 8 个数据。

第三十一章 触摸屏实验

本章，我们将介绍如何使用 STM32 来驱动触摸屏，ALIENTEK 战舰 STM32 开发板本身并没有触摸屏控制器，但是它支持触摸屏，可以通过外接带触摸屏的 LCD 模块（比如 ALIENTEK TFTLCD 模块），来实现触摸屏控制。在本章中，我们将向大家介绍 STM32 控制 ALIENTEK TFTLCD 模块，使用软件模拟 SPI 来实现对 TFTLCD 模块的触摸屏控制，最终实现一个手写板的功能。本章分为如下几个部分：

- 31.1 触摸屏简介
- 31.2 硬件设计
- 31.3 软件设计
- 31.4 下载验证

31.1 触摸屏简介

我们一般液晶所用的触摸屏，最多的就是电阻式触摸屏了（多点触摸属于电容式触摸屏，比如几乎所有智能机都支持多点触摸，它们所用的屏就是电容式的触摸屏），ALIENTEK TFTLCD 自带的触摸屏属于电阻式触摸屏，下面简单介绍下电阻式触摸屏的原理。

电阻式触摸屏利用压力感应进行控制。电阻触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，它以一层玻璃或硬塑料平板作为基层，表面涂有一层透明氧化金属（透明的导电电阻）导电层，上面再盖有一层外表面硬化处理、光滑防擦的塑料层、它的内表面也涂有一层涂层、在他们之间有许多细小的（小于 1/1000 英寸）的透明隔离点把两层导电层隔开绝缘。当手指触摸屏幕时，两层导电层在触摸点位置就有了接触，电阻发生变化，在 X 和 Y 两个方向上产生信号，然后送触摸屏控制器。控制器侦测到这一接触并计算出 (X, Y) 的位置，再根据获得的位置模拟鼠标的方式运作。这就是电阻技术触摸屏的最基本的原理。

电阻屏的特点有：

- 1) 是一种对外界完全隔离的工作环境，不怕灰尘、水汽和油污。
- 2) 可以用任何物体来触摸，可以用来写字画画，这是它们比较大的优势。
- 3) 电阻触摸屏的精度只取决于 A/D 转换的精度，因此都能轻松达到 4096*4096。

从以上介绍可知，触摸屏都需要一个 AD 转换器，一般来说是需要一个控制器的。ALIENTEK TFTLCD 模块选择的是四线电阻式触摸屏，这种触摸屏的控制芯片有很多，包括：ADS7843、ADS7846、TSC2046、XPT2046 和 AK4182 等。这几款芯片的驱动基本上是一样的，也就是你只要写出了 ADS7843 的驱动，这个驱动对其他几个芯片也是有效的。而且封装也有一样的，完全 PIN TO PIN 兼容。所以在替换起来，很方便。

ALIENTEK TFTLCD 模块自带的触摸屏控制芯片为 XPT2046。XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置，除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。XPT2046 采用微小的封装形式：TSSOP-16,QFN-16(0.75mm 厚度)和 VFBGA-48。工作温度范围为 -40°C ~ +85°C。

该芯片完全是兼容 ADS7843 和 ADS7846 的，关于这个芯片的详细使用，可以参考这两个芯片的 datasheet。



31.2 硬件设计

本章实验功能简介：开机的时候先通过 24C02 的数据判断触摸屏是否已经校准过，如果没有校准，则执行校准程序，校准过后再进入手写程序。如果已经校准了，就直接进入手写程序，此时可以通过按动屏幕来实现手写输入。屏幕上会有一个清空的操作区域（RST），点击这个地方就会将输入全部清除，恢复白板状态。程序会设置一个强制校准，就是通过按 KEY0 来实现，只要按下 KEY0 就会进入强制校准程序。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) KEY0 按键
- 3) TFTLCD 模块（带触摸屏）
- 4) 24C02

所有这些资源与 STM32 的连接图，在前面都已经介绍了，这里我们只针对 TFTLCD 模块与 STM32 的连接端口再说明一下，TFTLCD 模块的触摸屏总共有 5 跟线与 STM32 连接，连接电路图如图 31.2.1 所示：

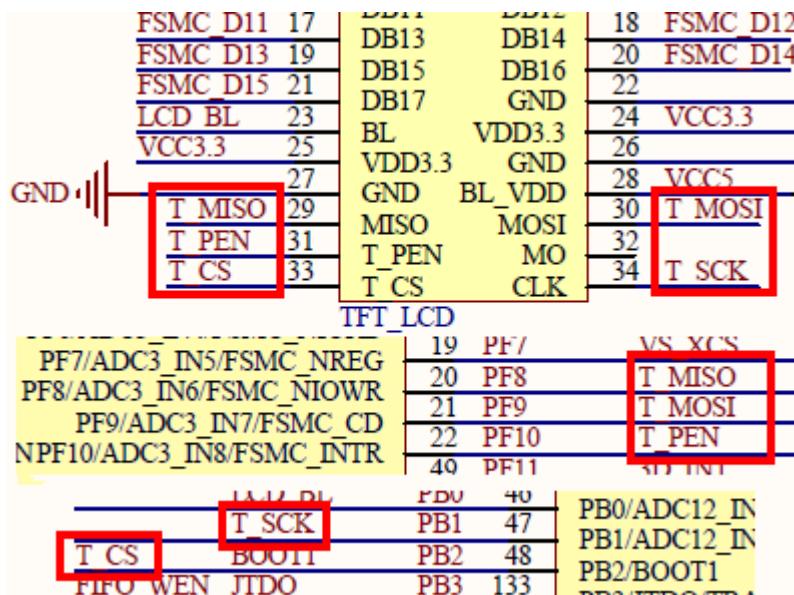


图 31.2.1 触摸屏与 STM32 的连接图

从图中可以看出，T_MISO、T_PEN、T_CS、T_MOSI 和 T_SCK 分别连接在 STM32 的：PF8、PF10、PB2、PF9 和 PB1 上。

31.3 软件设计

打开上一章的工程，首先在 HARDWARE 文件夹下新建一个 TOUCH 文件夹。然后新建一个 touch.c 和 touch.h 的文件保存在 TOUCH 文件夹下，并将这个文件夹加入头文件包含路径。

打开 touch.c 文件，在里面输入与触摸屏相关的代码，这里我们也不全部贴出来了，仅介绍几个重要的函数。

首先我们要介绍的是 TP_Read_XY2 这个函数，该函数专门用于从触摸屏控制 IC 读取坐标值（0~4095），TP_Read_XY2 的代码如下：

```
//连续 2 次读取触摸屏 IC,且这两次的偏差不能超过
//ERR_RANGE,满足条件,则认为读数正确,否则读数错误.
```



```
//该函数能大大提高准确度
//x,y:读取到的坐标值
//返回值:0,失败;1,成功。
#define ERR_RANGE 50 //误差范围
u8 TP_Read_XY2(u16 *x,u16 *y)
{
    u16 x1,y1;
    u16 x2,y2;
    u8 flag;
    flag=TP_Read_XY(&x1,&y1);
    if(flag==0)return(0);
    flag=TP_Read_XY(&x2,&y2);
    if(flag==0)return(0);
    if(((x2<=x1&&x1<=x2+ERR_RANGE)||((x1<=x2&&x2<=x1+ERR_RANGE))
        //前后两次采样在+- ERR_RANGE 内
        &&((y2<=y1&&y1<=y2+ERR_RANGE)||((y1<=y2&&y2<=y1+ERR_RANGE)))
    {
        *x=(x1+x2)/2;
        *y=(y1+y2)/2;
        return 1;
    }else return 0;
}
```

该函数采用了一个非常好的办法来读取屏幕坐标值，就是连续读两次，两次读取的值之差不能超过一个特定的值（ERR_RANGE），通过这种方式，我们可以大大提高触摸屏的准确度。另外该函数调用的 TP_Read_XY 函数，用于单次读取坐标值。TP_Read_XY 也采用了一些软件滤波算法，具体见光盘的源码。接下来，我们介绍另外一个函数 TP_Adjust，该函数源码如下：

```
//触摸屏校准代码
//得到四个校准参数
void TP_Adjust(void)
{
    u16 pos_temp[4][2];//坐标缓存值
    u8 cnt=0;
    u16 d1,d2;
    u32 tem1,tem2;
    float fac;
    u16 outtime=0;
    cnt=0;
    POINT_COLOR=BLUE;
    BACK_COLOR =WHITE;
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=RED;//红色
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=BLACK;
```



```
LCD_ShowString(40,40,160,100,16,(u8*)TP_REMIND_MSG_TBL); //显示提示信息
TP_Drow_Touch_Point(20,20,RED); //画点 1
tp_dev.sta=0; //消除触发信号
tp_dev.xfac=0; //xfac 用来标记是否校准过,所以校准之前必须清掉!以免错误
while(1) //如果连续 10 秒钟没有按下,则自动退出
{
    tp_dev.scan(1); //扫描物理坐标
    if((tp_dev.sta&0xc0)==TP_CATH_PRES) //按键按下了一次(此时按键松开了.)
    {
        outtime=0;
        tp_dev.sta&=~(1<<6); //标记按键已经被处理过了.
        pos_temp[cnt][0]=tp_dev.x;
        pos_temp[cnt][1]=tp_dev.y;
        cnt++;
        switch(cnt)
        {
            case 1:
                TP_Drow_Touch_Point(20,20,WHITE); //清除点 1
                TP_Drow_Touch_Point(lcddev.width-20,20,RED); //画点 2
                break;
            case 2:
                TP_Drow_Touch_Point(lcddev.width-20,20,WHITE); //清除点 2
                TP_Drow_Touch_Point(20,lcddev.height-20,RED); //画点 3
                break;
            case 3:
                TP_Drow_Touch_Point(20,lcddev.height-20,WHITE); //清除点 3
                TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,RED); //画点 4
                break;
            case 4://全部四个点已经得到
                //对边相等
                tem1=abs(pos_temp[0][0]-pos_temp[1][0]); //x1-x2
                tem2=abs(pos_temp[0][1]-pos_temp[1][1]); //y1-y2
                tem1*=tem1;
                tem2*=tem2;
                d1=sqrt(tem1+tem2); //得到 1,2 的距离
                tem1=abs(pos_temp[2][0]-pos_temp[3][0]); //x3-x4
                tem2=abs(pos_temp[2][1]-pos_temp[3][1]); //y3-y4
                tem1*=tem1;
                tem2*=tem2;
                d2=sqrt(tem1+tem2); //得到 3,4 的距离
                fac=(float)d1/d2;
                if(fac<0.95||fac>1.05||d1==0||d2==0)//不合格

```



```
{  
    cnt=0;  
    TP_Drow_Touch_Point(lcddev.width-20, lcddev.height-20, WHITE);  
    //清除点 4  
    TP_Drow_Touch_Point(20,20,RED); //画点 1  
    TP_Adj_Info_Show(pos_temp[0][0],pos_temp[0][1],pos_temp[1]  
    [0],pos_temp[1][1],pos_temp[2][0],pos_temp[2][1],pos_temp[3]  
    [0],pos_temp[3][1],fac*100); //显示数据  
    continue;  
}  
tem1=abs(pos_temp[0][0]-pos_temp[2][0]); //x1-x3  
tem2=abs(pos_temp[0][1]-pos_temp[2][1]); //y1-y3  
tem1*=tem1;  
tem2*=tem2;  
d1=sqrt(tem1+tem2); //得到 1,3 的距离  
tem1=abs(pos_temp[1][0]-pos_temp[3][0]); //x2-x4  
tem2=abs(pos_temp[1][1]-pos_temp[3][1]); //y2-y4  
tem1*=tem1;  
tem2*=tem2;  
d2=sqrt(tem1+tem2); //得到 2,4 的距离  
fac=(float)d1/d2;  
if(fac<0.95||fac>1.05) //不合格  
{  
    cnt=0;  
    TP_Drow_Touch_Point(lcddev.width-20, lcddev.height-20,  
    WHITE); //清除点 4  
    TP_Drow_Touch_Point(20,20,RED); //画点 1  
    TP_Adj_Info_Show(pos_temp[0][0],pos_temp[0][1],pos_temp[1]  
    [0],pos_temp[1][1],pos_temp[2][0],pos_temp[2][1],pos_temp[3]  
    [0],pos_temp[3][1],fac*100); //显示数据  
    continue;  
} //正确了  
//对角线相等  
tem1=abs(pos_temp[1][0]-pos_temp[2][0]); //x1-x3  
tem2=abs(pos_temp[1][1]-pos_temp[2][1]); //y1-y3  
tem1*=tem1;  
tem2*=tem2;  
d1=sqrt(tem1+tem2); //得到 1,4 的距离  
tem1=abs(pos_temp[0][0]-pos_temp[3][0]); //x2-x4  
tem2=abs(pos_temp[0][1]-pos_temp[3][1]); //y2-y4  
tem1*=tem1;  
tem2*=tem2;  
d2=sqrt(tem1+tem2); //得到 2,3 的距离
```



```
fac=(float)d1/d2;
if(fac<0.95||fac>1.05)//不合格
{
    cnt=0;
    TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,
    WHITE);//清除点 4
    TP_Drow_Touch_Point(20,20,RED);//画点 1
    TP_Adj_Info_Show(pos_temp[0][0],pos_temp[0][1],pos_temp[1]
    [0],pos_temp[1][1],pos_temp[2][0],pos_temp[2][1],pos_temp[3]
    [0],pos_temp[3][1],fac*100);//显示数据
    continue;
}//正确了
//计算结果
tp_dev.xfac=(float)(lcddev.width-40)/(pos_temp[1][0]-pos_temp[0][0]);
//得到 xfac
tp_dev.xoff=(lcddev.width-tp_dev.xfac*(pos_temp[1][0]+pos_temp[0]
[0]))/2;//得到 xoff
tp_dev.yfac=(float)(lcddev.height-40)/(pos_temp[2][1]-pos_temp[0][1]
); //得到 yfac
tp_dev.yoff=(lcddev.height-tp_dev.yfac*(pos_temp[2][1]+pos_temp[0]
[1]))/2;//得到 yoff
if(abs(tp_dev.xfac)>2||abs(tp_dev.yfac)>2)//触屏和预设的相反了.
{
    cnt=0;
    TP_Drow_Touch_Point(lcddev.width-20,lcddev.height-20,WHITE
    );//清除点 4
    TP_Drow_Touch_Point(20,20,RED); //画点 1
    LCD_ShowString(40,26,lcddev.width,lcddev.height,16,"TP Need
    readjust!");
    tp_dev.touchtype=!tp_dev.touchtype;//修改触屏类型.
    if(tp_dev.touchtype)//X,Y 方向与屏幕相反
    {
        CMD_RDX=0X90;
        CMD_RDY=0XD0;
    }
    else //X,Y 方向与屏幕相同
    {
        CMD_RDX=0XD0;
        CMD_RDY=0X90;
    }
    continue;
}
POINT_COLOR=BLUE;
LCD_Clear(WHITE);//清屏
```



```
LCD_ShowString(35,110,lcddev.width,lcddev.height,16,"Touch Screen
Adjust OK!"); //校正完成
delay_ms(1000);
TP_Save_Adjdata();
LCD_Clear(WHITE); //清屏
return; //校正完成
}
}
delay_ms(10);
outtime++;
if(outtime>1000)
{
    TP_Get_Adjdata();
    break;
}
}
```

TP_Adjust 是此部分最核心的代码，在这里，给大家介绍一下我们这里所使用的触摸屏校正原理：我们传统的鼠标是一种相对定位系统，只和前一次鼠标的位置坐标有关。而触摸屏则是一种绝对坐标系统，要选哪就直接点哪，与相对定位系统有着本质的区别。绝对坐标系统的特点是每一次定位坐标与上一次定位坐标没有关系，每次触摸的数据通过校准转为屏幕上的坐标，不管在什么情况下，触摸屏这套坐标在同一点的输出数据是稳定的。不过由于技术原理的原因，并不能保证同一点触摸每一次采样数据相同，不能保证绝对坐标定位，点不准，这就是触摸屏最怕出现的问题：漂移。对于性能质量好的触摸屏来说，漂移的情况出现并不是很严重。所以很多应用触摸屏的系统启动后，进入应用程序前，先要执行校准程序。通常应用程序中使用的 LCD 坐标是以像素为单位的。比如说：左上角的坐标是一组非 0 的数值，比如 (20, 20)，而右下角的坐标为 (220, 300)。这些点的坐标都是以像素为单位的，而从触摸屏中读出的是点的物理坐标，其坐标轴的方向、XY 值的比例因子、偏移量都与 LCD 坐标不同，所以，需要在程序中把物理坐标首先转换为像素坐标，然后再赋给 POS 结构，达到坐标转换的目的。

校正思路：在了解了校正原理之后，我们可以得出下面的一个从物理坐标到像素坐标的转换关系式：

$$LCDx = xfac * Px + xoff;$$

$$LCDy = yfac * Py + yoff;$$

其中(LCDx,LCDy)是在 LCD 上的像素坐标，(Px,Py)是从触摸屏读到的物理坐标。xfac, yfac 分别是 X 轴方向和 Y 轴方向的比例因子，而 xoff 和 yoff 则是这两个方向的偏移量。

这样我们只要事先在屏幕上面显示 4 个点（这四个点的坐标是已知的），分别按这四个点就可以从触摸屏读到 4 个物理坐标，这样就可以通过待定系数法求出 xfac、yfac、xoff、yoff 这四个参数。我们保存好这四个参数，在以后的使用中，我们把所有得到的物理坐标都按照这个关系式来计算，得到的就是准确的屏幕坐标。达到了触摸屏校准的目的。

TP_Adjust 就是根据上面的原理设计的校准函数，注意该函数里面多次使用了 lcddev.width 和 lcddev.height，用于坐标设置，主要是为了兼容不同尺寸的 LCD（比如 320*480 和 320*240 的屏都可以兼容）。其他的函数我们这里就不多介绍了，保存 touch.c 文件，并把该文件加入到 HARDWARE 组下。接下来打开 touch.h 文件，在该文件里面输入如下代码：



```
#ifndef __TOUCH_H__
#define __TOUCH_H__
#include "sys.h"
#define TP_PRES_DOWN 0x80 //触屏被按下
#define TP_CATH_PRES 0x40 //有按键按下了
//触摸屏控制器
typedef struct
{
    u8 (*init)(void);           //初始化触摸屏控制器
    u8 (*scan)(u8);            //扫描触摸屏.0,屏幕扫描;1,物理坐标;
    void (*adjust)(void);       //触摸屏校准
    u16 x0;                    //原始坐标(第一次按下时的坐标)
    u16 y0;                    //当前坐标(此次扫描时,触屏的坐标)
    u16 x;                     //当前坐标(此次扫描时,触屏的坐标)
    u16 y;                     //笔的状态
    u8  sta;                   //b7:按下 1/松开 0;
                                //b6:0,没有按键按下;1,有按键按下.

////////////////////////////触摸屏校准参数///////////////////////
float xfac;
float yfac;
short xoff;
short yoff;
//新增的参数,当触摸屏的左右上下完全颠倒时需要用到.
//touchtype=0 的时候,适合左右为 X 坐标,上下为 Y 坐标的 TP.
//touchtype=1 的时候,适合左右为 Y 坐标,上下为 X 坐标的 TP.
u8 touchtype;
}_m_tp_dev;
extern _m_tp_dev tp_dev;          //触屏控制器在 touch.c 里面定义
//与触摸屏芯片连接引脚
#define PEN      PFin(10)     //PF10 INT
#define DOUT     PFin(8)      //PF8  MISO
#define TDIN     PFout(9)     //PF9  MOSI
#define TCLK     PBout(1)     //PB1  SCLK
#define TCS      PBout(2)     //PB2  CS
void TP_Write_Byte(u8 num);        //向控制芯片写入一个数据
u16 TP_Read_AD(u8 CMD);          //读取 AD 转换值
u16 TP_Read_XOY(u8 xy);          //带滤波的坐标读取(X/Y)
u8 TP_Read_XY(u16 *x,u16 *y);    //双方向读取(X+Y)
u8 TP_Read_XY2(u16 *x,u16 *y);   //带加强滤波的双方向坐标读取
void TP_Drow_Touch_Point(u16 x,u16 y,u16 color); //画一个坐标校准点
void TP_Draw_Big_Point(u16 x,u16 y,u16 color);   //画一个大点
u8 TP_Scan(u8 tp);               //扫描
```



```
void TP_Save_Adjdata(void);           //保存校准参数
u8 TP_Get_Adjdata(void);             //读取校准参数
void TP_Adjust(void);                //触摸屏校准
u8 TP_Init(void);                  //初始化
void TP_Adj_Info_Show(u16 x0,u16 y0,u16 x1,u16 y1,u16 x2,u16 y2,u16 x3,u16 y3,u16 fac);
//显示校准信息
#endif
```

上述代码，我们定义了 `_m_tp_dev` 结构体，用于管理和记录触摸屏相关信息，在外部调用的时候，我们一般直接调用 `tp_dev` 的相关成员函数/变量即可达到需要的效果。这样种设计简化了接口，另外管理和维护也比较方便，大家可以效仿一下。其他部分我们不做多的介绍，最后我们打开 `main.c`，修改代码如下：

```
//清屏，重新装载对话界面
void Load_Drow_Dialog(void)
{
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(lcddev.width-24,0,200,16,16,"RST");//显示清屏区域
    POINT_COLOR=RED;//设置画笔蓝色
}
int main(void)
{
    u8 key;
    u8 i=0;
    delay_init();          //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
    LCD_Init();            //LCD 初始化
    KEY_Init();            //按键初始化
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"TOUCH TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/11");
    LCD_ShowString(60,130,200,16,16,"Press KEY0 to Adjust");
    tp_dev.init();
    delay_ms(1500);
    Load_Drow_Dialog();
    while(1)
    {
        key=KEY_Scan(0);
        tp_dev.scan(0);
        if(tp_dev.sta&TP_PRES_DOWN)           //触摸屏被按下
```



```
{  
    if(tp_dev.x<lcddev.width&&tp_dev.y<lcddev.height)  
    {  
        if(tp_dev.x>(lcddev.width-24)&&tp_dev.y<16)Load_Drow_Dialog();//清除  
        else TP_Draw_Big_Point(tp_dev.x,tp_dev.y,RED); //画图  
    }  
    }else delay_ms(10); //没有按键按下的时候  
    if(key==KEY_RIGHT) //KEY_RIGHT 按下,则执行校准程序  
    {  
        LCD_Clear(WHITE);//清屏  
        TP_Adjust(); //屏幕校准  
        TP_Save_Adjdata();  
        Load_Drow_Dialog();  
    }  
    i++;  
    if(i==20)  
    {  
        i=0;  
        LED0=!LED0;  
    }  
}
```

此函数就实现了我们上面介绍的本章所要实现的功能。当然这里还用到我们之前写的24CXX 的代码，用来保存和调用触摸屏的校准信息（在触摸屏校准函数和初始化函数里面）。

31.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 31.4.1 所示：

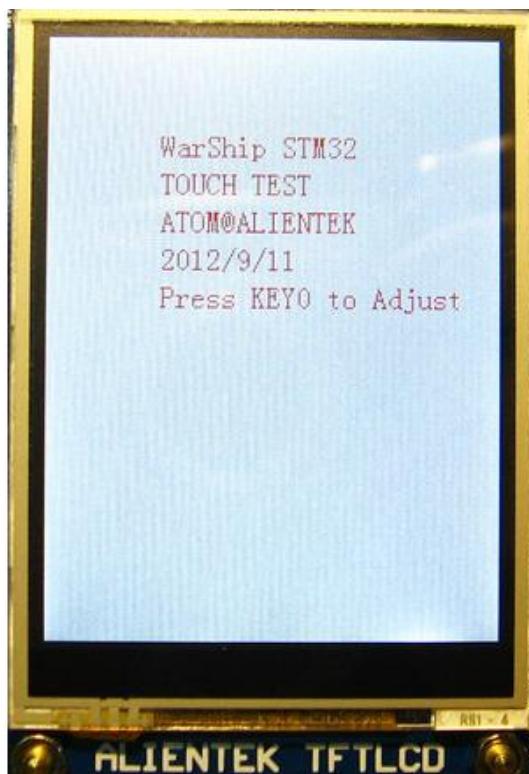


图 31.4.1 程序运行效果

如果已经校准过了，则在等待 1.5s 之后进入手写界面，同时 DS0 开始闪烁，界面如图 31.4.2 所示：



图 31.4.2 手写界面

此时，我们就可以在该界面下用笔或者手指输入信息了。如果没有校准过，则会自动进入校准程序（当你发现精度不行的时候，也可以通过按 KEY0 进入校准程序），如图 31.4.3 所示，

在校准完成之后自动进入手写界面。

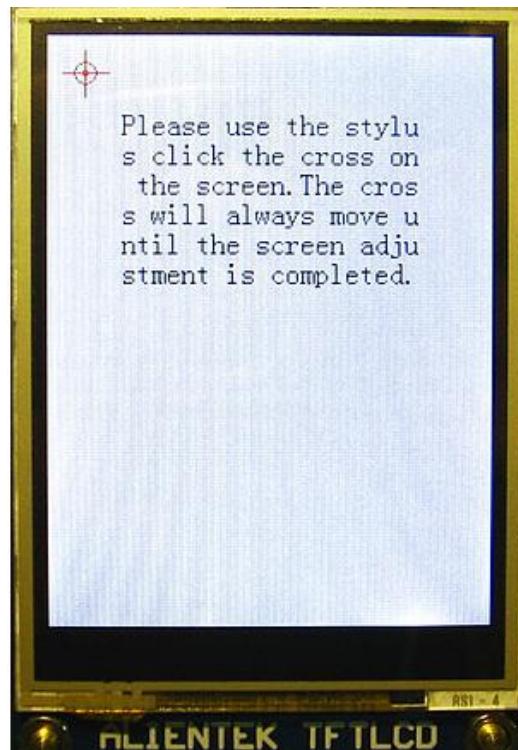


图 31.4.3 校准界面

第三十二章 红外遥控实验

本章，我们将向大家介绍如何通过 STM32 来解码红外遥控器的信号。ALIENTK 战舰 STM32 开发板标配了红外接收头和一个很小巧的红外遥控器。在本章中，我们将利用 STM32 的输入捕获功能，解码开发板标配的这个红外遥控器的编码信号，并将解码后的键值 TFTLCD 模块上显示出来。本章分为如下几个部分：

- 32.1 红外遥控简介
- 32.2 硬件设计
- 32.3 软件设计
- 32.4 下载验证



32.1 红外遥控简介

红外遥控是一种无线、非接触控制技术，具有抗干扰能力强，信息传输可靠，功耗低，成本低，易实现等显著优点，被诸多电子设备特别是家用电器广泛采用，并越来越多的应用到计算机系统中。

由于红外线遥控不具有像无线电遥控那样穿过障碍物去控制被控对象的能力，所以，在设计红外线遥控器时，不必要像无线电遥控器那样，每套(发射器和接收器)要有不同的遥控频率或编码(否则，就会隔墙控制或干扰邻居的家用电器)，所以同类产品的红外线遥控器，可以有相同的遥控频率或编码，而不会出现遥控信号“串门”的情况。这对于大批量生产以及在家用电器上普及红外线遥控提供了极大的方面。由于红外线为不可见光，因此对环境影响很小，再由红外光波动波长远小于无线电波的波长，所以红外线遥控不会影响其他家用电器，也不会影响临近的无线电设备。

红外遥控的编码目前广泛使用的是：NEC Protocol 的 PWM(脉冲宽度调制)和 Philips

RC-5 Protocol 的 PPM(脉冲位置调制)。ALIENTEK 战舰 STM32 开发板配套的遥控器使用的是 NEC 协议，其特征如下：

- 1、8 位地址和 8 位指令长度；
- 2、地址和命令 2 次传输（确保可靠性）
- 3、PWM 脉冲位置调制，以发射红外载波的占空比代表“0”和“1”；
- 4、载波频率为 38Khz；
- 5、位时间为 1.125ms 或 2.25ms；

NEC 码的位定义：一个脉冲对应 560us 的连续载波，一个逻辑 1 传输需要 2.25ms (560us 脉冲+1680us 低电平)，一个逻辑 0 的传输需要 1.125ms (560us 脉冲+560us 低电平)。而遥控接收头在收到脉冲的时候为低电平，在没有脉冲的时候为高电平，这样，我们在接收头端收到的信号为：逻辑 1 应该是 560us 低+1680us 高，逻辑 0 应该是 560us 低+560us 高。

NEC 遥控指令的数据格式为：同步码、地址码、地址反码、控制码、控制反码。同步码由一个 9ms 的低电平和一个 4.5ms 的高电平组成，地址码、地址反码、控制码、控制反码均是 8 位数据格式。按照低位在前，高位在后的顺序发送。采用反码是为了增加传输的可靠性（可用于校验）。

我们遥控器的按键 2 按下时，从红外接收头端收到的波形如图 32.1.1 所示：

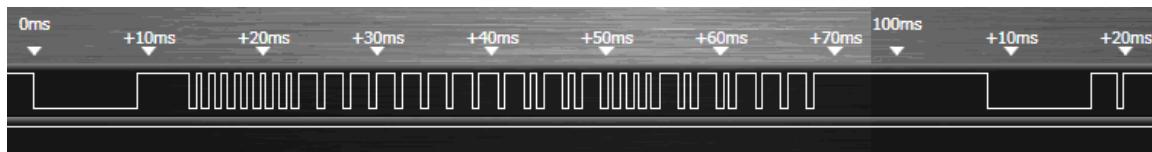


图 32.1.1 按键 2 所对应的红外波形

从图 32.1.1 中可以看到，其地址码为 0，控制码为 168。可以看到在 100ms 之后，我们还收到了几个脉冲，这是 NEC 码规定的连发码(由 9ms 低电平+2.5m 高电平+0.56ms 低电平+97.94ms 高电平组成)，如果在一帧数据发送完毕之后，按键仍然没有放开，则发射重复码，即连发码，可以通过统计连发码的次数来标记按键按下的长短/次数。

第十五章我们曾经介绍过利用输入捕获来测量高电平的脉宽，本章解码红外遥控信号，刚好可以利用输入捕获的这个功能来实现遥控解码。关于输入捕获的介绍，请参考第十五章的内容。



32.2 硬件设计

本实验采用定时器的输入捕获功能实现红外解码，本章实验功能简介：开机在 LCD 上显示一些信息之后，即进入等待红外触发，如过接收到正确的红外信号，则解码，并在 LCD 上显示键值和所代表的意义，以及按键次数等信息。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) TFTLCD 模块（带触摸屏）
- 3) 红外接收头
- 4) 红外遥控器

前两个，在之前的实例已经介绍过了，遥控器属于外部器件，遥控接收头在板子上，与 MCU 的连接原理图如 32.2.1 所示：

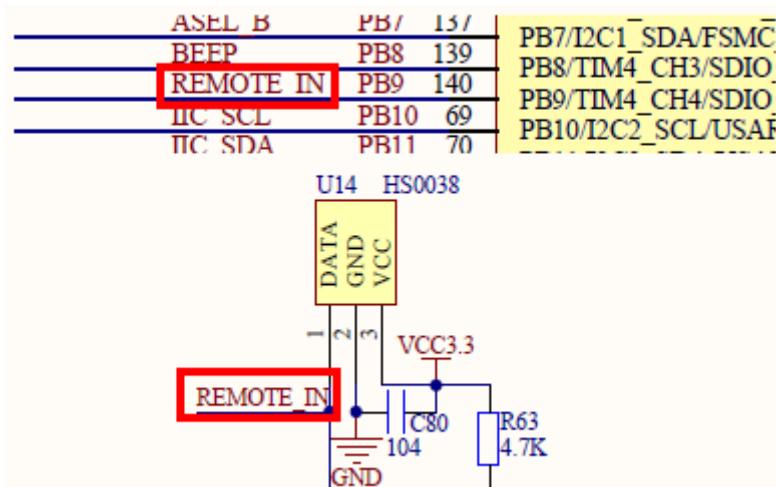


图 32.2.1 红外遥控接收头与 STM32 的连接电路图

红外遥控接收头连接在 STM32 的 PB9 (TIM4_CH4) 上。硬件上不需要变动，只要程序将 TIM4_CH4 设计为输入捕获，然后将收到的脉冲信号解码就可以了。开发板配套的红外遥控器外观如图 32.2.2 所示：



图 32.2.2 红外遥控器



32.3 软件设计

打开我们光盘的红外遥控器实验工程，可以看到我们添加了 remote.c 和 remote.h 两个文件，同时因为我们使用的是输入捕获，所以还用到库函数 stm32f10x_tim.c 和头文件 stm32f10x_tim.h。

打开 remote.c 文件，代码如下：

```
#include "remote.h"
#include "delay.h"
#include "usart.h"
//红外遥控初始化
//设置 IO 以及定时器 4 的输入捕获
void Remote_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_ICInitTypeDef  TIM_ICInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE); //使能 PORTB 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE); //TIM4 时钟使能

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;           //PB9 输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;        //上拉输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);             //初始化 GPIOB.9
    GPIO_SetBits(GPIOB,GPIO_Pin_9);                     //GPIOB.9 输出高

    TIM_TimeBaseStructure.TIM_Period = 10000; //设定计数器自动重装值 最大 10ms 溢出
    TIM_TimeBaseStructure.TIM_Prescaler =(72-1); //预分频器,1M 的计数频率,1us
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; //设置时钟分割
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
                                         //TIM 向上计数模式
    TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure); //根据指定的参数初始化 TIMx

    TIM_ICInitStructure.TIM_Channel = TIM_Channel_4; // 选择输入 IC4 映射到 TI4 上
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;//上升沿捕获
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频
    TIM_ICInitStructure.TIM_ICFilter = 0x03;//IC4F=0011 8 个定时器时钟周期滤波
    TIM_ICInit(TIM4, &TIM_ICInitStructure);//初始化定时器输入捕获通道

    TIM_Cmd(TIM4,ENABLE );                      //使能定时器 4
```



```
NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;           //TIM3 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;   //先占优先级 0 级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;          //从优先级 3 级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;            //IRQ 通道被使能
NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化外设 NVIC 寄存器

TIM_ITConfig( TIM4,TIM_IT_Update|TIM_IT_CC4,ENABLE); //允许更新中断 ,
//允许 CC4IE 捕获中断
}
//遥控器接收状态
//[7]:收到了引导码标志
//[6]:得到了一个按键的所有信息
//[5]:保留
//[4]:标记上升沿是否已经被捕获
//[3:0]:溢出计时器
u8 RmtSta=0;
u16 Dval; //下降沿时计数器的值
u32 RmtRec=0; //红外接收到的数据
u8 RmtCnt=0; //按键按下的次数
//定时器 2 中断服务程序
void TIM4_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM4,TIM_IT_Update)!=RESET)
    {
        if(RmtSta&0x80)//上次有数据被接收到了
        {
            RmtSta&=~0X10; //取消上升沿已经被捕获标记
            if((RmtSta&0X0F)==0X00)RmtSta|=1<<6; //标记已经完成一次键值信息采集
            if((RmtSta&0X0F)<14)RmtSta++;
            else
            {
                RmtSta&=~(1<<7); //清空引导标识
                RmtSta&=0XF0; //清空计数器
            }
        }
    }
    if(TIM_GetITStatus(TIM4,TIM_IT_CC4)!=RESET)
    {
        if(RDATA)//上升沿捕获
        {
            TIM_OC4PolarityConfig(TIM4,TIM_ICPolarity_Falling); //下降沿捕获
            TIM_SetCounter(TIM4,0); //清空定时器值
            RmtSta|=0X10; //标记上升沿已经被捕获
        }
        else //下降沿捕获
        {
    }
```



```
Dval=TIM_GetCapture4(TIM4); //读取 CCR1 也可以清 CC1IF 标志位
TIM_OC4PolarityConfig(TIM4,TIM_ICPolarity_Rising); //上升沿捕获

if(RmtSta&0X10) //完成一次高电平捕获
{
    if(RmtSta&0X80)//接收到了引导码
    {

        if(Dval>300&&Dval<800) //560 为标准值,560us
        {
            RmtRec<<=1; //左移一位.
            RmtRec|=0; //接收到 0
        }else if(Dval>1400&&Dval<1800) //1680 为标准值,1680us
        {
            RmtRec<<=1; //左移一位.
            RmtRec|=1; //接收到 1
        }else if(Dval>2200&&Dval<2600) //得到按键键值增加的信息
        //2500 为标准值 2.5ms
        {
            RmtCnt++; //按键次数增加 1 次
            RmtSta&=0XF0; //清空计时器
        }
        }else if(Dval>4200&&Dval<4700) //4500 为标准值 4.5ms
        {
            RmtSta|=1<<7; //标记成功接收到了引导码
            RmtCnt=0; //清除按键次数计数器
        }
    }
    RmtSta&=~(1<<4);
}
}

TIM_ClearFlag(TIM4,TIM_IT_Update|TIM_IT_CC4);
}

//处理红外键盘
//返回值:
//    0,没有任何按键按下
//其他,按下的按键键值.
u8 Remote_Scan(void)
{
    u8 sta=0;
    u8 t1,t2;
    if(RmtSta&(1<<6))//得到一个按键的所有信息了
```



```

{
    t1=RmtRec>>24;           //得到地址码
    t2=(RmtRec>>16)&0xff;    //得到地址反码
    if((t1==(u8)~t2)&&t1==REMOTE_ID)//检验遥控识别码(ID)及地址
    {
        t1=RmtRec>>8;
        t2=RmtRec;
        if(t1==(u8)~t2)sta=t1;//键值正确
    }
    if((sta==0)||((RmtSta&0X80)==0))//按键数据错误/遥控已经没有按下了
    {
        RmtSta&=~(1<<6); //清除接收到有效按键标识
        RmtCnt=0;           //清除按键次数计数器
    }
}
return sta;
}

```

该部分代码包含 3 个函数，首先是 `Remote_Init` 函数，该函数用于初始化 IO 口，并配置 `TIM4_CH4` 为输入捕获，并设置其相关参数，这里的配置跟输入捕获实验的配置基本接近，大家可以参考一下输入捕获实验的讲解。`TIM4_IRQHandler` 函数是 `TIM4` 的中断服务函数，在该函数里面，实现对红外信号的高电平脉冲的捕获，同时根据我们之前简介的协议内容来解码，该函数用到几个全局变量，用于辅助解码，并存储解码结果。最后是 `Remote_Scan` 函数，该函数用来扫描解码结果，相当于我们的按键扫描，输入捕获解码的红外数据，通过该函数传送给其他程序。

接下来打开 `remote.h`，该文件代码比较简单，宏定义的标识符 `REMOTE_ID` 就是我们开发板配套的遥控器的识别码，对于其他遥控器可能不一样，只要修改这个为你所使用的遥控器的一致就可以了。其他是一些函数的声明。下面我们看看 `main.c` 里面主函数代码：

```

int main(void)
{
    u8 key;
    u8 t=0;
    u8 *str=0;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //LCD 初始化
    KEY_Init();              //按键端口初始化
    Remote_Init();           //红外接收初始化

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"REMOTE TEST");
}

```



```
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/12");
LCD_ShowString(60,130,200,16,16,"KEYVAL:");
LCD_ShowString(60,150,200,16,16,"KEYCNT:");
LCD_ShowString(60,170,200,16,16,"SYMBOL:");
while(1)
{
    key=Remote_Scan();
    if(key)
    {
        LCD_ShowNum(116,130,key,3,16);          //显示键值
        LCD_ShowNum(116,150,RmtCnt,3,16);      //显示按键次数
        switch(key)
        {
            case 0:str="ERROR";break;
            case 162:str="POWER";break;
            case 98:str="UP";break;
            case 2:str="PLAY";break;
            case 226:str="ALIENTEK";break;
            case 194:str="RIGHT";break;
            case 34:str="LEFT";break;
            case 224:str="VOL-";break;
            case 168:str="DOWN";break;
            case 144:str="VOL+";break;
            case 104:str="1";break;
            case 152:str="2";break;
            case 176:str="3";break;
            case 48:str="4";break;
            case 24:str="5";break;
            case 122:str="6";break;
            case 16:str="7";break;
            case 56:str="8";break;
            case 90:str="9";break;
            case 66:str="0";break;
            case 82:str="DELETE";break;
        }
        LCD_Fill(116,170,116+8*8,170+16,WHITE); //清楚之前的显示
        LCD_ShowString(116,170,200,16,16,str);   //显示 SYMBOL
    }else delay_ms(10);
    t++;
    if(t==20)
    {
        t=0;
        LED0=!LED0;
    }
}
```



```
}
```

主函数 main 代码比较简单，进行一系列初始化后，根据扫描到的按键值来显示。至此，我们的软件设计部分就结束了。

32.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 32.4.1 所示的内容：



图 32.4.1 程序运行时 LCD 显示内容

此时我们通过遥控器按下不同的按键，则可以看到 LCD 上显示了不同按键的键值以及按键次数和对应的遥控器上的符号。如图 32.4.2 所示：



图 32.4.2 解码成功

第三十三章 游戏手柄实验

相信 80 后小时候都有玩过 FC 游戏机（又称：红白机/小霸王游戏机），那是一代经典，给我们的童年带了无限乐趣。本章，我们将向大家介绍如何通过 STM32 来驱动 FC 游戏机手柄，将 FC 游戏机的手柄作为战舰 STM32 开发板的输入设备（综合实验可以直接通过这个手柄来玩 FC 游戏）。

在本章中，我们将使用 STM32 驱动 FC 手柄，将手柄的按键键值等信息通过 TFTLCD 模块显示出来。本章分为如下几个部分：

- 33.1 游戏手柄简介
- 33.2 硬件设计
- 33.3 软件设计
- 33.4 下载验证



33.1 游戏手柄简介

FC 游戏机曾今是一统天下（现在也还是很多人玩），红极一时，那时任天堂单是 FC 机的主机的发售收入就超过全美国的电视台的收入的总和。本章，我们将使用 STM32 来驱动 FC 手柄，实现手柄控制信号的读取，我们先来了解一下 FC 手柄。

FC 手柄，大致可分为两种：一种手柄插口是 11 针的，一种是 9 针的。但 11 针的现在市面上很少了（因为 11 针手柄是早期 FC 组装兼容机最主要的周边），现在几乎都是 9 针 FC 组装手柄的天下，所以我们本章使用的是 9 针 FC 手柄，该手柄还有一个特点，就是可以直接和 DR9 的串口头对插！这样同开发板的连接就简单了。FC 手柄的外观如图 33.1.1 所示：



图 33.1.1 FC 手柄外观图

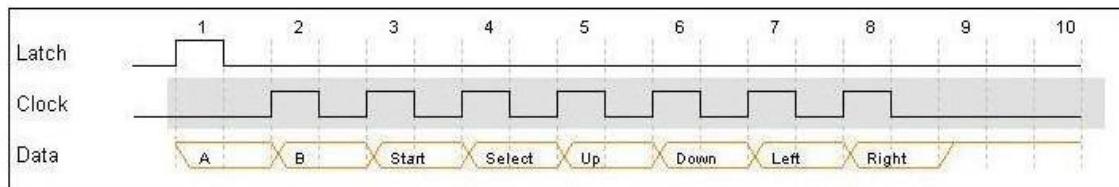
这种手柄一般有 10 个按键（实际是 8 个键值）：上、下、左、右、Start、Select、A、B、A 连发、B 连发。这里的 A 和 A 连发是一个键值，而 B 和 B 连发也是一个键值，只是连发按键当你一直按下的时候，会不停的发送（方便快速按键，比如发炮弹之类的功能）。

FC 手柄的控制电路，由 1 个 8 位并入串出的移位寄存器（CD4021），外加一个时基集成电路（NE555，用于连发）构成。不过现在的手柄，为了节约成本，直接就在 PCB 上做绑定了，所以你拆开手柄，一般是看不到里面有四四方方的 IC，而只有一个黑色的小点，所有电路都集成到这个里面了，但是他们的控制和读取方法还是一样的。

9 针手柄的读取时序和接线图如图 33.1.2 所示：



读取时序图



插头接线图

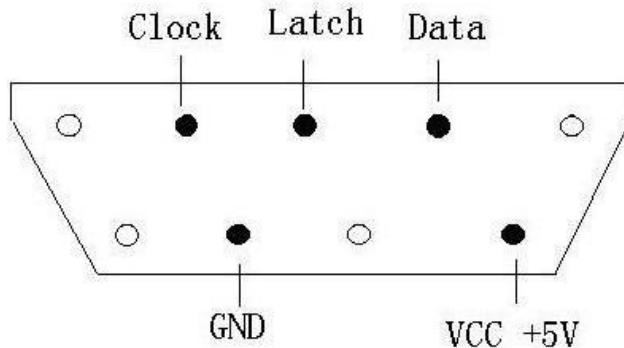


图 33.1.2 FC 手柄读取时序和接线图

从上图可看出，读取手柄按键值的信息十分简单：先 **Latch**（锁存键值），然后就得到了第一个按键值（A），之后在 **Clock** 的作用下，依次读取其他按键的键值，总共 8 个按键键值。

有了以上了解，我们就可以通过 STM32 的 IO 来驱动 FC 手柄了。

33.2 硬件设计

本实验采用 STM32 的 3 个普通 IO 连接 FC 手柄的 Clock、Data 和 Latch 信号，本章实验功能简介：在主函数不停的查询手柄输入，一旦检测到输入信号，则在 LCD 模块上面显示键值和对应的按键符号。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) TFTLCD 模块
- 3) FC 手柄接口 (JOYPAD)
- 4) FC 手柄

前两个，在之前的实例已经介绍过了，FC 手柄属于外部器件。战舰 STM32 开发板板载了一个 FC 手柄接口（就是一个 DR9 接头），该接口与 MCU 的连接原理图如 33.2.1 所示：

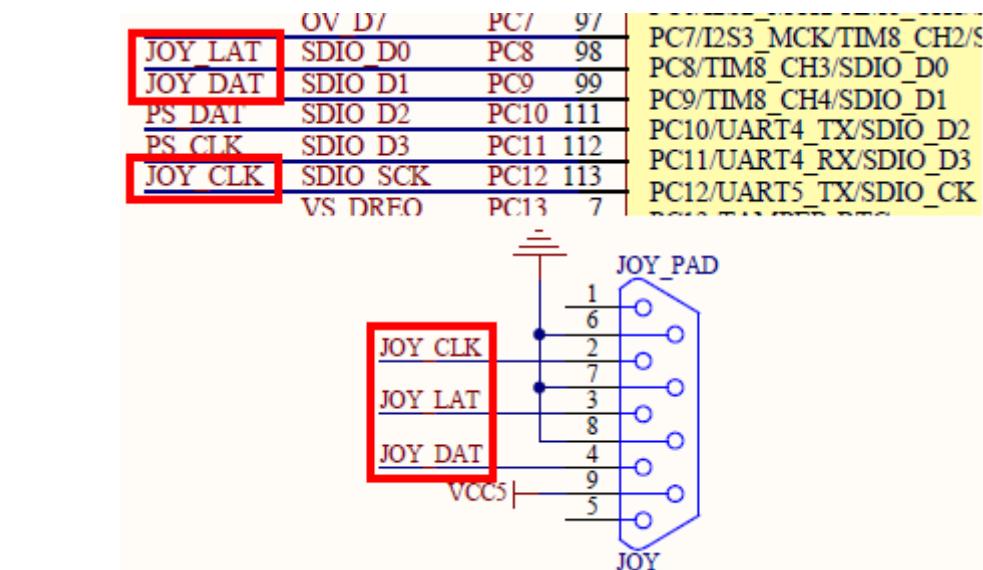


图 33.2.1 FC 手柄接头与 STM32 的连接电路图

图中，JOY_PAD 就是用来连接 FC 手柄的，该接头采用标准的 DR9 座，战舰 STM32 开发板上有 2 个 DR9 座，一个用来接 FC 手柄（有 JOY_PAD 字样，LCD 左上），另外一个用来接 RS232 串口（有 COM 字样，LCD 右上），这两个头千万不要接错！否则可能烧坏手柄或者烧坏 STM32。

从上图我们知道，手柄的 CLK (Clock)、LAT (Latch) 和 DAT (Data) 分别连接在 STM32 的 PC12、PC8 和 PC9 上面，这里与 SDIO 部分信号线共用了，所以当使用 SDIO 的时候，就不能使用 FC 手柄了。因为信号线都是直连的，所以我们在开发板上不需要做配置，只需要将 FC 手柄插入 JOY_PAD 插口即可。

开发板配套的手柄，见图 33.1.1。



33.3 软件设计

打开我们的游戏手柄实验工程，可以看到我们的工程中添加了 joypad.c 文件以及其头文件 joypad.h 文件。

打开 joypad.c 文件，代码如下：

```
#include "joypad.h"
//初始化手柄接口.
void JOYPAD_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); //使能 PC 端口时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_12;           //端口
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;                //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);                         //初始化 GPIO
    GPIO_SetBits(GPIOC,GPIO_Pin_8|GPIO_Pin_12);                   //输出高

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;                        //上拉输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;                   //初始化 GPIO
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    GPIO_SetBits(GPIOC,GPIO_Pin_9);                                //输出高
}

//读取手柄按键值.
//FC 手柄数据输出格式:
//每给一个脉冲,输出一位数据,输出顺序:
//A->B->SELECT->START->UP->DOWN->LEFT->RIGHT.
//总共 8 位,对于有 C 按钮的手柄,按下 C 其实就等于 A+B 同时按下.
//按下是 0,松开是 1.
//返回值:
//[0]:右 [1]:左 [2]:下 [3]:上 [1]:Start [5]:Select [6]:B [7]:A
u8 JOYPAD_Read(void)
{
    u8 temp=0;
    u8 t;
    JOYPAD_LAT=1;                                              //锁存当前状态
    JOYPAD_LAT=0;
    for(t=0;t<8;t++)
    {
        temp<<=1;
        if(JOYPAD_DAT)temp|=0x01;    //LOAD 之后, 就得到第一个数据
        JOYPAD_CLK=1;               //每给一次脉冲, 收到一个数据
    }
}
```



```
    JOYPAD_CLK=0;  
}  
return temp;  
}
```

该部分代码仅 2 个函数，都比较简单，JOYPAD_Init 函数用于初始化 IO，即把 PC8、PC9 和 PC12 设置为正确的状态，以便同 FC 手柄通信。另外一个函数 JOYPAD_Read 就是按照图 33.1.2 所示的时序读取 FC 手柄，该函数的返回值就是手柄的状态。

接下来打开 joypad.h 可以看到该文件里代码主要是定义位带操作实现 IO 控制，当然，你也可以跟 LE 试验一样通过库函数设置。

最后我们看看 main.c 主函数代码：

```
const u8*JOYPAD_SYMBOL_TBL[8]=  
{ "Right","Left","Down","Up","Start","Select","A","B"};//手柄按键符号定义  
int main(void)  
{  
    u8 key;  
    u8 t=0,i=0;  
    delay_init();          //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级  
    uart_init(9600);       //串口初始化波特率为 9600  
    LED_Init();           //LED 端口初始化  
    LCD_Init();           //LCD 初始化  
    JOYPAD_Init();        //手柄初始化  
    POINT_COLOR=RED;//设置字体为红色  
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");  
    LCD_ShowString(60,70,200,16,16,"JOYPAD TEST");  
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(60,110,200,16,16,"2012/9/12");  
    LCD_ShowString(60,130,200,16,16,"KEYVAL:");  
    LCD_ShowString(60,150,200,16,16,"SYMBOL:");  
    POINT_COLOR=BLUE;//设置字体为红色  
    while(1)  
    {  
        key=JOYPAD_Read();  
        if(key!=0XFF)  
        {  
            LCD_ShowNum(116,130,key,3,16);//显示键值  
            for(i=0;i<8;i++)  
            {  
                if((key&(1<<i))==0)  
                {  
                    LCD_Fill(60+56,150,60+56+48,150+16,WHITE);//清除之前的显示  
                    LCD_ShowString(60+56,150,200,16,16,(u8*)JOYPAD_SYMBOL_TBL[i]);//显示符号  
                }  
            }  
        }  
    }  
}
```



```
        }
    }
delay_ms(10);
t++;
if(t==20)
{
    t=0;
    LED0=!LED0;
}
}
}
```

此部分代码也比较简单，初始化 JOYPAD 之后，就一直扫描 FC 手柄（通过 JOYPAD_Read 函数实现），然后只要接收到手柄的有效信号，就在 LCD 模块上面显示出来。

至此，我们的软件设计部分就结束了。

33.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 33.4.1 所示的内容：



图 33.4.1 程序运行时 LCD 显示内容

此时我们按下 FC 手柄的按键，则可以看到 LCD 上显示了对应按键的键值以及对应的符号。如图 33.4.2 所示：



图 33.4.2 解码成功



第三十四章 三轴加速度传感器实验

自从有了 Iphone，各种新技术的普及程度越来越快，人们喜欢的不再是摔不坏的诺基亚，而是用户体验极佳的 Iphone。

本章，我们介绍一种当今智能手机普遍具有的传感器：加速度传感器。在手机上，这个功能可以用来：自动切换横竖屏、玩游戏和唱歌等。ALIENTEK 战舰 STM32 开发板自带了加速度传感器：ADXL345。本章我们将使用 STM32 来驱动 ADXL345，读取 3 个方向的重力加速度值，并转换为角度，显示在 TFTLCD 模块上。本章分为以下几个部分：

- 34.1 ADXL345 简介
- 34.2 硬件设计
- 34.3 软件设计
- 34.4 下载验证



34.1 ADXL345 简介

ADXL345 是 ADI 公司的一款 3 轴、数字输出的加速度传感器。ADXL345 是 ADI 公司推出的基于 iMEMS 技术的 3 轴、数字输出加速度传感器。该加速度传感器的特点有：

- 分辨率高。最高 13 位分辨率。
- 量程可变。具有 $+/2g$, $+/4g$, $+/8g$, $+/16g$ 可变的测量范围。
- 灵敏度高。最高达 $3.9mg/LSB$, 能测量不到 1.0° 的倾斜角度变化。
- 功耗低。 $40\sim145\mu A$ 的超低功耗, 待机模式只有 $0.1\mu A$ 。
- 尺寸小。整个 IC 尺寸只有 $3mm \times 5mm \times 1mm$, LGA 封装。

ADXL 支持标准的 I2C 或 SPI 数字接口, 自带 32 级 FIFO 存储, 并且内部有多种运动状态检测和灵活的中断方式等特性。ADXL345 传感器的检测轴如图 34.1.1 所示:

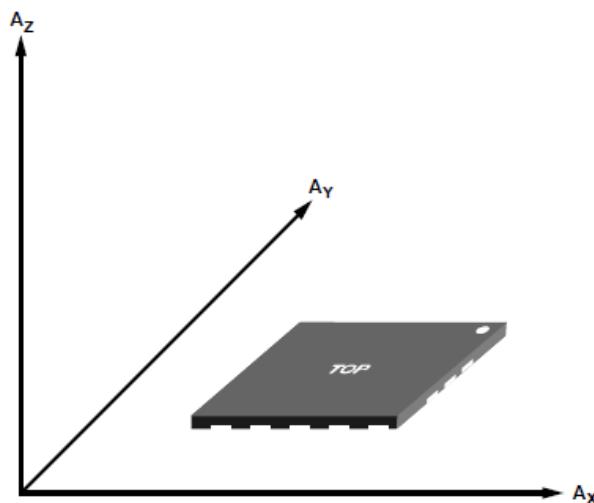


图 34.1.1 ADXL345 的三个检测轴

当 ADXL345 沿检测轴正向加速时, 它对正加速度进行检测。在检测重力时用户需要注意, 当检测轴的方向与重力的方向相反时检测到的是正加速度。图 34.1.2 所示为输出对重力的响应。

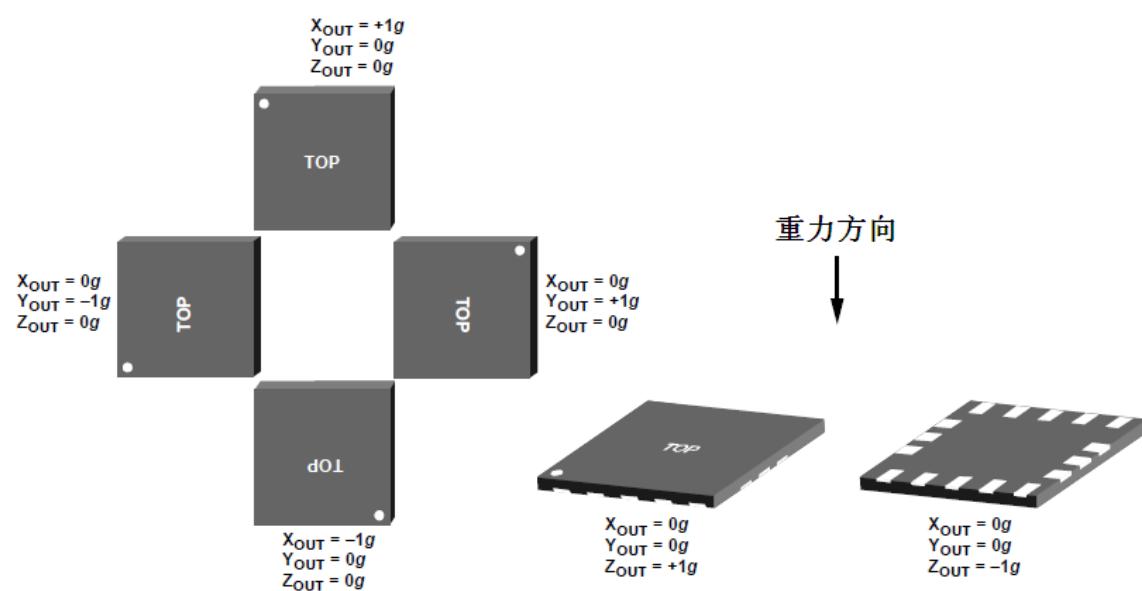


图 34.1.2 ADXL345 输出对重力的响应



图 34.1.2 列出了 ADXL345 在不同摆放方式时的输出，以便后续分析。接下来我们看看 ADXL345 的引脚图，如图 34.1.3 所示：

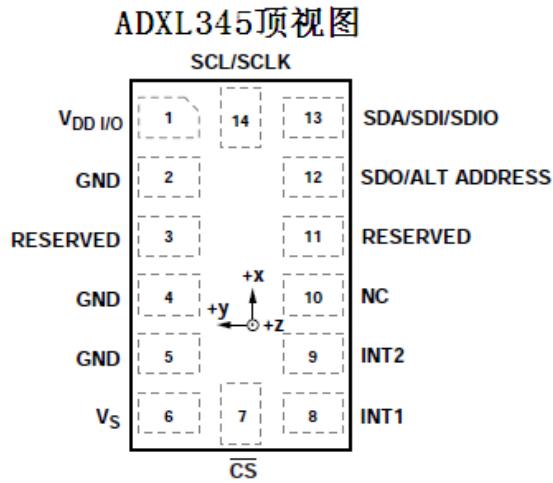


图 34.1.3 ADXL345 引脚图

ADXL345 支持 SPI 和 IIC 两种通信方式，为了节省 IO 口，战舰 STM32 开发板采用的是 IIC 方式连接，官方推荐的 IIC 连接电路如图 34.1.4 所示：

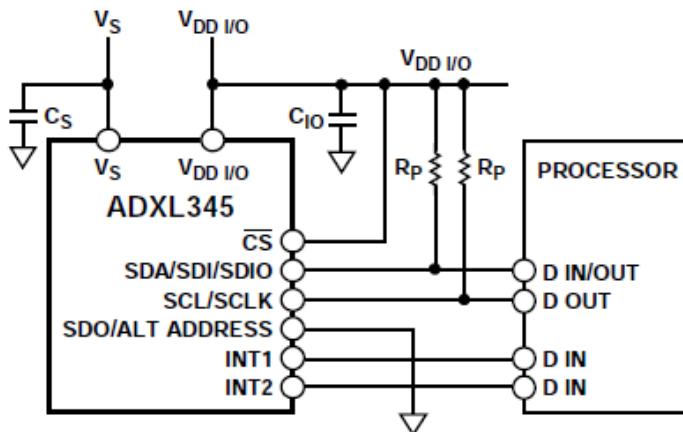


图 34.1.4 ADXL345 IIC 模式连接电路

从上图可看出，ADXL345 的连接十分简单，外围需要的器件也极少（就 2 个电容），如上连接（SDO/ALT ADDRESS 接地），则 ADXL345 的地址为 0X53（不含最低位），如果 SDO/ALT ADDRESS 接高，那么 ADXL345 的地址将变为 0X1D（不含最低位）。IIC 通信的时序我们在之前已经介绍过（第二十七章，IIC 实验），这里就不再细说了。

最后，我们介绍一下 ADXL345 的初始化步骤。ADXL345 的初始化步骤如下：

- 1) 上电
- 2) 等待 1.1ms
- 3) 初始化命令序列
- 4) 结束

其中上电这个动作发生在开发板第一次上电的时候，在上电之后，等待 1.1ms 左右，就可以开始发送初始化序列了，初始化序列一结束，ADXL345 就开始正常工作了。这里的初始化序列，最简单的只需要配置 3 个寄存器，如表 34.1.1 所示：



步骤	寄存器地址	寄存器名字	寄存器值	功能描述
1	0X31	DATA_FORMAT	0X0B	±16g, 13位模式
2	0X2D	POWER_CTL	0X08	测量模式
3	0X2E	INT_ENABLE	0X80	使能 DATA_READY 中断

表 34.1.1 ADXL345 最简单的初始化命令序列

发送以上序列给 ADXL345 以后，ADXL345 即开始正常工作。

ADXL345 我们就介绍到这里，详细的介绍，请参考 ADXL345 的数据手册。

34.2 硬件设计

本实验采用 STM32 的 3 个普通 IO 连接 ADXL345，本章实验功能简介：主函数不停的查询 ADXL345 的转换结果，得到 x、y 和 z 三个方向的加速度值（读数值），然后将其转换为与自然系坐标的角度，并将结果在 LCD 模块上显示出来。DS0 来指示程序正在运行，通过按下 WK_UP 按键，可以进行 ADXL345 的自动校准（DS1 用于提示正在校准）。

所要用到的硬件资源如下：

- 1) 指示灯 DS0、DS1
- 2) WK_UP 按键
- 3) TFTLCD 模块
- 4) ADXL345

前 3 个，在之前的实例已经介绍过了，这里我们仅介绍 ADXL345 与战舰 STM32 开发板的连接。该接口与 MCU 的连接原理图如 34.2.1 所示：

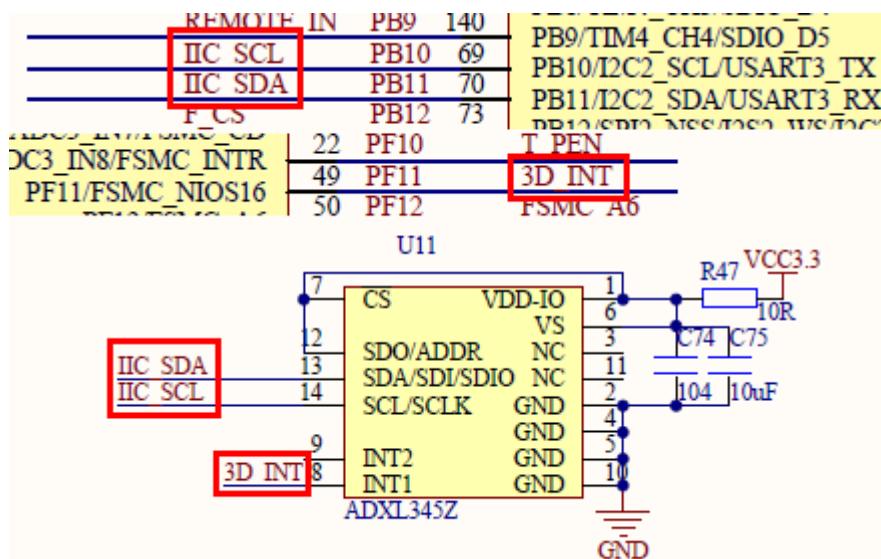


图 34.2.1 ADXL345 与 STM32 的连接电路图

从上图可以看出，ADXL345 通过三根线与 STM32 开发板连接，其中 IIC 总线时和 24C02 以及 RDA5820 共用，接在 PB10 和 PB11 上面。ADXL345 的两个中断输出，这里我们只用了一个，连接在 STM32 的 PF11 脚，另外这里的地址线是接 3.3V，所以 ADXL345 的地址是 0X1D，转换为 0X3A 写入，0X3B 读取。



34.3 软件设计

打开我们的三轴加速度传感器实验工程可以看到，工程中添加了 adxl345.c 文件和 adxl345.h 头文件，传感器相关的初始化代码以及驱动代码都分布在这个文件中。同时，我们还引入了 myiic.c 和头文件 myiic.h，因为传感器是通过 IIC 接口控制的。

打开 adxl345.c 文件，代码如下：

```
#include "adxl345.h"
#include "sys.h"
#include "delay.h"
#include "math.h"
//初始化 ADXL345.
//返回值:0,初始化成功;1,初始化失败.
u8 ADXL345_Init(void)
{
    IIC_Init();                                //初始化 IIC 总线
    if(ADXL345_RD_Reg(DEVICE_ID)==0XE5)        //读取器件 ID
    {
        ADXL345_WR_Reg(DATA_FORMAT,0X2B);       //低电平中断输出,13 位全分辨率,输出数据右对齐,16g 量程
        ADXL345_WR_Reg(BW_RATE,0x0A);           //数据输出速度为 100Hz
        ADXL345_WR_Reg(POWER_CTL,0x28);          //链接使能,测量模式
        ADXL345_WR_Reg(INT_ENABLE,0x00);         //不使用中断
        ADXL345_WR_Reg(OFSX,0x00);
        ADXL345_WR_Reg(OFSY,0x00);
        ADXL345_WR_Reg(OFSZ,0x00);
        return 0;
    }
    return 1;
}
//写 ADXL345 寄存器
//addr:寄存器地址
//val:要写入的值
//返回值:无
void ADXL345_WR_Reg(u8 addr,u8 val)
{
    IIC_Start();
    IIC_Send_Byt(ADXL_WRITE);      //发送写器件指令
    IIC_Wait_Ack();
    IIC_Send_Byt(addr);           //发送寄存器地址
    IIC_Wait_Ack();
    IIC_Send_Byt(val);            //发送值
    IIC_Wait_Ack();
}
```



```
IIC_Stop(); //产生一个停止条件
}

//读 ADXL345 寄存器
//addr:寄存器地址
//返回值:读到的值
u8 ADXL345_RD_Reg(u8 addr)
{
    u8 temp=0;
    IIC_Start();
    IIC_Send_Byte(ADXL_WRITE); //发送写器件指令
    temp=IIC_Wait_Ack();
    IIC_Send_Byte(addr); //发送寄存器地址
    temp=IIC_Wait_Ack();
    IIC_Start(); //重新启动
    IIC_Send_Byte(ADXL_READ); //发送读器件指令
    temp=IIC_Wait_Ack();
    temp=IIC_Read_Byte(0); //读取一个字节,不继续再读,发送 NAK
    IIC_Stop(); //产生一个停止条件
    return temp; //返回读到的值
}

//读取 ADXL 的平均值
//x,y,z:读取 10 次后取平均值
void ADXL345_RD_Avval(short *x,short *y,short *z)
{
    short tx=0,ty=0,tz=0;
    u8 i;
    for(i=0;i<10;i++)
    {
        ADXL345_RD_XYZ(x,y,z);
        delay_ms(10);
        tx+=(short)*x; ty+=(short)*y; tz+=(short)*z;
    }
    *x=tx/10; *y=ty/10; *z=tz/10;
}

//自动校准
//xval,yval,zval:x,y,z 轴的校准值
void ADXL345_AUTO_Adjust(char *xval,char *yval,char *zval)
{
    short tx,ty,tz;
    u8 i;
    short offx=0,offy=0,offz=0;
    ADXL345_WR_Reg(POWER_CTL,0x00); //先进入休眠模式.
    delay_ms(100);
```



```
ADXL345_WR_Reg(DATA_FORMAT,0X2B);
//低电平中断输出,13位全分辨率,输出数据右对齐,16g 量程
ADXL345_WR_Reg(BW_RATE,0x0A);      //数据输出速度为 100Hz
ADXL345_WR_Reg(POWER_CTL,0x28);     //链接使能,测量模式
ADXL345_WR_Reg(INT_ENABLE,0x00);    //不使用中断
ADXL345_WR_Reg(OFSX,0x00);
ADXL345_WR_Reg(OFSY,0x00);
ADXL345_WR_Reg(OFSZ,0x00);
delay_ms(12);
for(i=0;i<10;i++)
{
    ADXL345_RD_Avval(&tx,&ty,&tz);
    offx+=tx; offy+=ty; offz+=tz;
}
offx/=10; offy/=10; offz/=10;
*xval=-offx/4; *yval=-offy/4; *zval=-(offz-256)/4;
ADXL345_WR_Reg(OFSX,*xval);
ADXL345_WR_Reg(OFSY,*yval);
ADXL345_WR_Reg(OFSZ,*zval);
}

//读取 3 个轴的数据
//x,y,z:读取到的数据
void ADXL345_RD_XYZ(short *x,short *y,short *z)
{
    u8 buf[6],i;
    IIC_Start();
    IIC_Send_Byte(ADXL_WRITE); //发送写器件指令
    IIC_Wait_Ack();
    IIC_Send_Byte(0x32);      //发送寄存器地址(数据缓存的起始地址为 0X32)
    IIC_Wait_Ack();
    IIC_Start();              //重新启动
    IIC_Send_Byte(ADXL_READ); //发送读器件指令
    IIC_Wait_Ack();
    for(i=0;i<6;i++)
    {
        if(i==5)buf[i]=IIC_Read_Byte(0); //读取一个字节,不继续再读,发送 NACK
        else buf[i]=IIC_Read_Byte(1);   //读取一个字节,继续读,发送 ACK
    }
    IIC_Stop();                //产生一个停止条件
    *x=(short)((u16)buf[1]<<8)+buf[0];
    *y=(short)((u16)buf[3]<<8)+buf[2];
    *z=(short)((u16)buf[5]<<8)+buf[4];
}
```



```
//读取 ADXL345 的数据 times 次,再取平均
//x,y,z:读到的数据
//times:读取多少次
void ADXL345_Read_Average(short *x,short *y,short *z,u8 times)
{
    u8 i;
    short tx,ty,tz;
    *x=0; *y=0; *z=0;
    if(times)//读取次数不为 0
    {
        for(i=0;i<times;i++)//连续读取 times 次
        {
            ADXL345_RD_XYZ(&tx,&ty,&tz);
            *x+=tx; *y+=ty; *z+=tz;
            delay_ms(5);
        }
        *x/=times; *y/=times; *z/=times;
    }
}
//得到角度
//x,y,z:x,y,z 方向的重力加速度分量(不需要单位,直接数值即可)
//dir:要获得的角度.0,与 Z 轴的角度;1,与 X 轴的角度;2,与 Y 轴的角度.
//返回值:角度值.单位 0.1° .
short ADXL345_Get_Angle(float x,float y,float z,u8 dir)
{
    float temp,res=0;
    switch(dir)
    {
        case 0://与自然 Z 轴的角度
            temp=sqrt((x*x+y*y))/z;
            res=atan(temp);
            break;
        case 1://与自然 X 轴的角度
            temp=x/sqrt((y*y+z*z));
            res=atan(temp);
            break;
        case 2://与自然 Y 轴的角度
            temp=y/sqrt((x*x+z*z));
            res=atan(temp);
            break;
    }
    return res*1800/3.14;
}
```



该部分代码总共有 8 个函数，这里我们仅介绍其中 4 个。首先是 ADXL345_Init 函数，该函数用来初始化 ADXL345，和前面我们提到的步骤差不多，不过本章我们而是采用查询的方式来读取数据的，所以在这里并没有开启中断。另外 3 个偏移寄存器，都默认设置为 0。

其次，我们介绍 ADXL345_RD_XYZ 函数，该函数用于从 ADXL345 读取数据，通过该函数可以读取 ADXL345 的转换结果，得到三个轴的加速度值（仅是数值，并没有转换单位）。

接着，我们介绍 ADXL345_AUTO_Adjust 函数，该函数用于 ADXL345 的校准，ADXL345 有偏移校准的功能，该功能的详细介绍请参考 ADXL345 数据手册的第 29 页，偏移校准部分。这里我们就不细说了，如果不进行校准的话，ADXL345 的读数可能会有些偏差，通过校准，我们可以讲这个偏差减少甚至消除。

最后，我们看看 ADXL345_Get_Angle 函数，该函数根据 ADXL345 的读值，转换为与自然坐标系的角度。计算公式如下：

$$\text{加速度传感器 Z 轴与自然坐标系 Z 轴夹角: } \angle 1 = \tan^{-1}\left(\frac{\sqrt{A_x^2 + A_y^2}}{A_z}\right);$$

$$\text{加速度传感器 X 轴与自然坐标系 X 轴夹角: } \angle 2 = \tan^{-1}\left(\frac{A_x}{\sqrt{A_y^2 + A_z^2}}\right);$$

$$\text{加速度传感器 Y 轴与自然坐标系 Y 轴夹角: } \angle 3 = \tan^{-1}\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right);$$

其中 Ax, Ay, Az 分别代表从 ADXL345 读到的 X, Y, Z 方向的加速度值。通过该函数，我们只需要知道三个方向的加速度值，就可以将其转换为对应的弧度值，再通过弧度角度转换，就可以得到角度值了。

其他函数，我们就不介绍了，也比较简单。接下来打开 adxl345.h 可以看到里面主要是一些宏定义标识符以及函数申明，这里就不多讲解了。

最后看看 main.c 里面代码如下：

```
void Adxl_Show_Num(u16 x,u16 y,short num,u8 mode)
{
    if(mode==0) //显示加速度值
    {
        if(num<0)
        {
            LCD_ShowChar(x,y,'-',16,0); //显示负号
            num=-num; //转为正数
        }
        else LCD_ShowChar(x,y,' ',16,0); //去掉负号
        LCD_ShowNum(x+8,y,num,4,16); //显示值
    }
    else //显示角度值
    {
        if(num<0)
        {
            LCD_ShowChar(x,y,'-',16,0); //显示负号
            num=-num; //转为正数
        }
        else LCD_ShowChar(x,y,' ',16,0); //去掉负号
    }
}
```



```
LCD_ShowNum(x+8,y,num/10,2,16);      //显示整数部分
LCD_ShowChar(x+24,y,'.',16,0);        //显示小数点
LCD_ShowNum(x+32,y,num%10,1,16);      //显示小数部分
}

}

int main(void)
{
    u8 key;
    u8 t=0;
    short x,y,z;
    short angx,angy,angz;
    delay_init();           //延时函数初始化
    NVIC_Configuration();  //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //初始化 LCD
    KEY_Init();              //初始化 KEY

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"3D TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/12");
    LCD_ShowString(60,130,200,16,16,"KEY0:Auto Adjust");
    while(ADXL345_Init()) //3D 加速度传感器初始化
    {
        LCD_ShowString(60,150,200,16,16,"ADXL345 Error");
        delay_ms(200);
        LCD_Fill(60,150,239,150+16,WHITE);
        delay_ms(200);
    }
    LCD_ShowString(60,150,200,16,16,"ADXL345 OK");
    LCD_ShowString(60,170,200,16,16,"X VAL:");
    LCD_ShowString(60,190,200,16,16,"Y VAL:");
    LCD_ShowString(60,210,200,16,16,"Z VAL:");
    LCD_ShowString(60,230,200,16,16,"X ANG:");
    LCD_ShowString(60,250,200,16,16,"Y ANG:");
    LCD_ShowString(60,270,200,16,16,"Z ANG:");
    POINT_COLOR=BLUE;//设置字体为红色
    while(1)
    {
        if(t%10==0)//每 100ms 读取一次
        {
```



```
//得到 X,Y,Z 轴的加速度值(原始值)
ADXL345_Read_Average(&x,&y,&z,10); //读取 X,Y,Z 三个方向的加速度值
Adxl_Show_Num(60+48,170,x,0); //显示加速度原始值
Adxl_Show_Num(60+48,190,y,0);
Adxl_Show_Num(60+48,210,z,0);
//得到角度值,并显示
angx=ADXL345_Get_Angle(x,y,z,1);
angy=ADXL345_Get_Angle(x,y,z,2);
angz=ADXL345_Get_Angle(x,y,z,0);
Adxl_Show_Num(60+48,230,angx,1); //显示角度值
Adxl_Show_Num(60+48,250,angy,1);
Adxl_Show_Num(60+48,270,angz,1);
}
key=KEY_Scan(0);
if(key==KEY_UP)
{
    LED1=0;//绿灯亮,提示校准中
    ADXL345_AUTO_Adjust((char*)&x,(char*)&y,(char*)&z);//自动校准
    LED1=1;//绿灯灭,提示校准完成
}
delay_ms(10);
t++;
if(t==20)
{
    t=0;
    LED0=!LED0;
}
}
}
```

此部分代码除了 main 函数，还有一个 Adxl_Show_Num 函数，该函数用于数据显示，因为在 lcd.c 里面，没有提供可以显示小数和负数的函数，所以我们这里编写了该函数，来实现小数和负数的显示，以满足本章要求。

其他部分，我们就不多说了。至此，我们的软件设计部分就结束了。

34.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 34.4.1 所示的内容：



图 34.4.1 程序运行时 LCD 显示内容

可以看到，X 方向和 Z 方向的角度有些大（最佳值是 0），所以我们按下 WK_UP 键，进行一次校准（注意校准时保持开发板水平，并且稳定），校准后如图 34.4.2 所示：

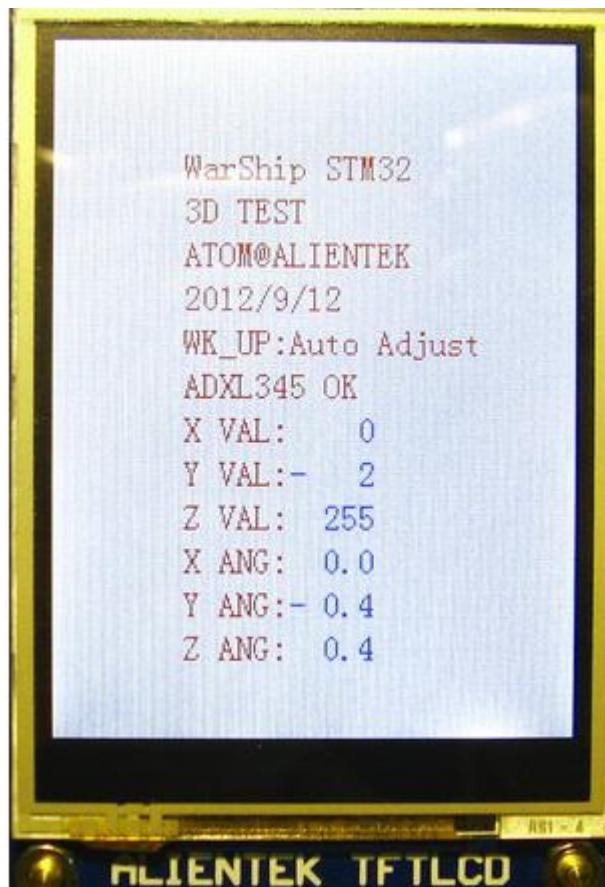


图 34.4.2 校准后

可以看到，校准后，比未校准前好了很多，此时我们移动开发板到不同角度，可以看到 X、Y、Z 的数值和角度也跟着变化。



第三十五章 DS18B20 数字温度传感器实验

STM32 虽然内部自带了温度传感器，但是因为芯片温升较大等问题，与实际温度差别较大，所以，本章我们将向大家介绍如何通过 STM32 来读取外部数字温度传感器的温度，来得到较为准确的环境温度。在本章中，我们将学习使用单总线技术，通过它来实现 STM32 和外部温度传感器（DS18B20）的通信，并把从温度传感器得到的温度显示在 TFTLCD 模块上。本章分为如下几个部分：

- 35.1 DS18B20 简介
- 35.2 硬件设计
- 35.3 软件设计
- 35.4 下载验证



35.1 DS18B20 简介

DS18B20 是由 DALLAS 半导体公司推出的一种的“一线总线”接口的温度传感器。与传统的热敏电阻等测温元件相比，它是一种新型的体积小、适用电压宽、与微处理器接口简单的数字化温度传感器。一线总线结构具有简洁且经济的特点，可使用户轻松地组建传感器网络，从而为测量系统的构建引入全新概念，测量温度范围为 -55~+125°C，精度为 ±0.5°C。现场温度直接以“一线总线”的数字方式传输，大大提高了系统的抗干扰性。它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~12 位的数字值读数方式。它工作在 3~5.5V 的电压范围，采用多种封装形式，从而使系统设计灵活、方便，设定分辨率及用户设定的报警温度存储在 EEPROM 中，掉电后依然保存。其内部结构如图 35.1.1 所示：

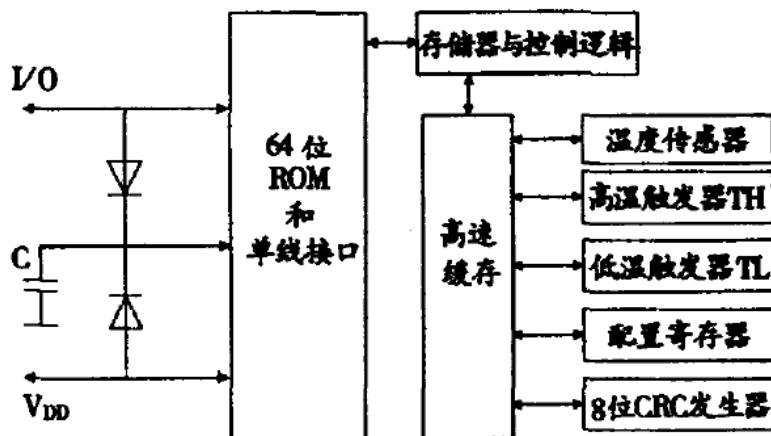


图 35.1.1 DS18B20 内部结构图

ROM 中的 64 位序列号是出厂前被光记好的，它可以看作是该 DS18B20 的地址序列码，每 DS18B20 的 64 位序列号均不相同。64 位 ROM 的排列是：前 8 位是产品家族码，接着 48 位是 DS18B20 的序列号，最后 8 位是前面 56 位的循环冗余校验码(CRC=X8+X5 +X4 +1)。ROM 作用是使每一个 DS18B20 都各不相同，这样就可实现一根总线上挂接多个。

所有的单总线器件要求采用严格的信号时序，以保证数据的完整性。DS18B20 共有 6 种信号类型：复位脉冲、应答脉冲、写 0、写 1、读 0 和读 1。所有这些信号，除了应答脉冲以外，都由主机发出同步信号。并且发送所有的命令和数据都是字节的低位在前。这里我们简单介绍这几个信号的时序：

1) 复位脉冲和应答脉冲

单总线上的所有通信都是以初始化序列开始。主机输出低电平，保持低电平时间至少 480 μs，以产生复位脉冲。接着主机释放总线，4.7K 的上拉电阻将单总线拉高，延时 15~60 μs，并进入接收模式(Rx)。接着 DS18B20 拉低总线 60~240 μs，以产生低电平应答脉冲，若为低电平，再延时 480 μs。

2) 写时序

写时序包括写 0 时序和写 1 时序。所有写时序至少需要 60μs，且在 2 次独立的写时序之间至少需要 1μs 的恢复时间，两种写时序均起始于主机拉低总线。写 1 时序：主机输出低电平，延时 2μs，然后释放总线，延时 60μs。写 0 时序：主机输出低电平，延时 60μs，然后释放总线，延时 2μs。

3) 读时序

单总线器件仅在主机发出读时序时，才向主机传输数据，所以，在主机发出读数据命令后，



必须马上产生读时序，以便从机能够传输数据。所有读时序至少需要 60us，且在 2 次独立的读时序之间至少需要 1us 的恢复时间。每个读时序都由主机发起，至少拉低总线 1us。主机在读时序期间必须释放总线，并且在时序起始后的 15us 之内采样总线状态。典型的读时序过程为：主机输出低电平延时 2us，然后主机转入输入模式延时 12us，然后读取单总线当前的电平，然后延时 50us。

在了解了单总线时序之后，我们来看看 DS18B20 的典型温度读取过程，DS18B20 的典型温度读取过程为：复位 → 发 SKIP ROM 命令（0XCC）→发开始转换命令（0X44）→延时 → 复位 → 发送 SKIP ROM 命令（0XCC）→发读存储器命令（0XBEB）→连续读出两个字节数据(即温度)→结束。

DS18B20 的介绍就到这里，更详细的介绍，请大家参考 DS18B20 的技术手册。

35.2 硬件设计

由于开发板上标准配置是没有 DS18B20 这个传感器的，只有接口，所以要做本章的实验，大家必须找一个 DS18B20 插在预留的 18B20 接口上。

本章实验功能简介：开机的时候先检测是否有 DS18B20 存在，如果没有，则提示错误。只有在检测到 DS18B20 之后才开始读取温度并显示在 LCD 上，如果发现了 DS18B20，则程序每隔 100ms 左右读取一次数据，并把温度显示在 LCD 上。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) TFTLCD 模块
- 3) DS18B20 接口
- 4) DS18B20 温度传感器

前两部分，在之前的实例已经介绍过了，而 DS18B20 温度传感器属于外部器件（板上没有直接焊接），这里也不介绍。本章，我们仅介绍 DS18B20 接口和 STM32 的连接电路，如图 35.2.1 所示：

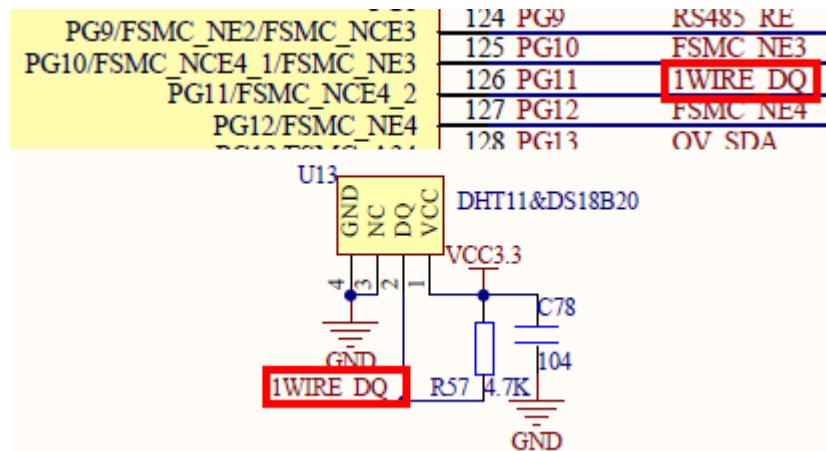


图 35.2.1 DS18B20 接口与 STM32 的连接电路图

从上图可以看出，我们使用的是 STM32 的 PG11 来连接 U13 的 DQ 引脚，图中 U13 为 DHT11（数字温湿度传感器）和 DS18B20 共用的一个接口，DHT11 我们将在下一章介绍。DS18B20 只用到其中的 3 个引脚（U13 的 1、2 和 3 脚），将 DS18B20 传感器插入到这个上面就可以通过 STM32 来读取 DS18B20 的温度了。连接示意图如图 35.2.2 所示：

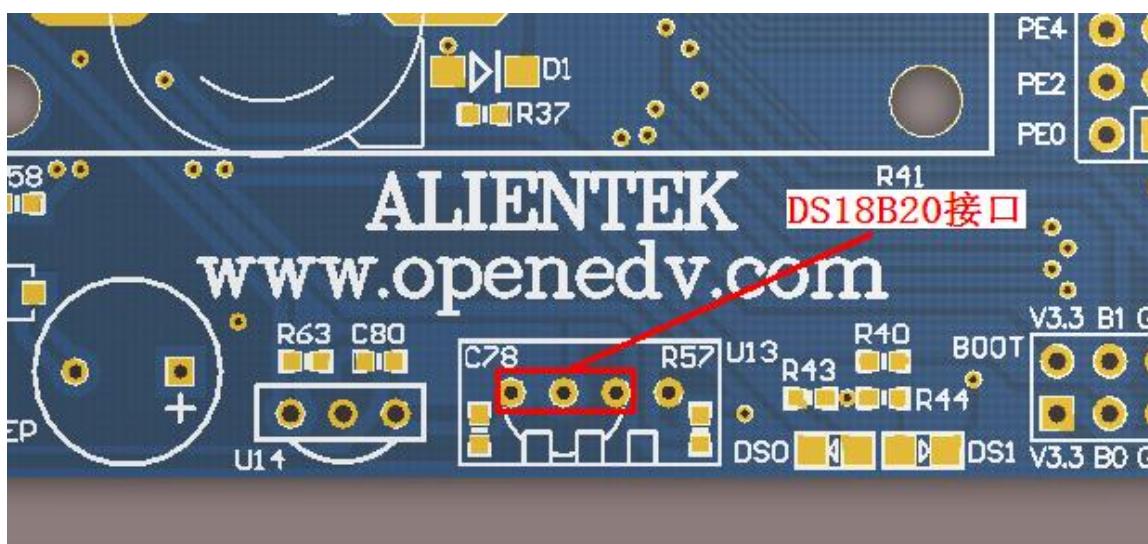


图 35.2.2 DS18B20 连接示意图

从上图可以看出，DS18B20 的平面部分（有字的那面）应该朝内，而曲面部分朝外。然后插入如图所示的三个孔内。

35.3 软件设计

打开我们的 DS18B20 数字温度传感器实验工程可以看到我们添加了 ds18b20.c 文件以及其头文件 ds18b20.h 文件，所有 ds18b20 驱动代码和相关定义都分布在这两个文件中。

打开 ds18b20.c，该文件代码如下：

```
#include "ds18b20.h"
#include "delay.h"

//复位 DS18B20
void DS18B20_Rst(void)
{
    DS18B20_IO_OUT();      //SET PA0 OUTPUT
    DS18B20_DQ_OUT=0;     //拉低 DQ
    delay_us(750);        //拉低 750us
    DS18B20_DQ_OUT=1;     //DQ=1
    delay_us(15);         //15US
}

//等待 DS18B20 的回应
//返回 1:未检测到 DS18B20 的存在
//返回 0:存在
u8 DS18B20_Check(void)
{
    u8 retry=0;
    DS18B20_IO_IN();//SET PA0 INPUT
    while (DS18B20_DQ_IN&&retry<200)
    {
        retry++;
    }
}
```



```
delay_us(1);
};

if(retry>=200) return 1;
else retry=0;
while (!DS18B20_DQ_IN&&&retry<240)
{
    retry++;
    delay_us(1);
};
if(retry>=240) return 1;
return 0;
}

//从 DS18B20 读取一个位
//返回值: 1/0
u8 DS18B20_Read_Bit(void)           // read one bit
{
    u8 data;
    DS18B20_IO_OUT(); //SET PA0 OUTPUT
    DS18B20_DQ_OUT=0;
    delay_us(2);
    DS18B20_DQ_OUT=1;
    DS18B20_IO_IN(); //SET PA0 INPUT
    delay_us(12);
    if(DS18B20_DQ_IN) data=1;
    else data=0;
    delay_us(50);
    return data;
}

//从 DS18B20 读取一个字节
//返回值: 读到的数据
u8 DS18B20_Read_Byte(void)          // read one byte
{
    u8 i,j,dat;
    dat=0;
    for (i=1;i<=8;i++)
    {
        j=DS18B20_Read_Bit();
        dat=(j<<7)|(dat>>1);
    }
    return dat;
}

//写一个字节到 DS18B20
//dat: 要写入的字节
```



```
void DS18B20_Write_Byte(u8 dat)
{
    u8 j;
    u8 testb;
    DS18B20_IO_OUT();//SET PA0 OUTPUT;
    for (j=1;j<=8;j++)
    {
        testb=dat&0x01;
        dat=dat>>1;
        if (testb)
        {   DS18B20_DQ_OUT=0;// Write 1
            delay_us(2);
            DS18B20_DQ_OUT=1;
            delay_us(60);
        }
        else
        {   DS18B20_DQ_OUT=0;// Write 0
            delay_us(60);
            DS18B20_DQ_OUT=1;
            delay_us(2);
        }
    }
}

//开始温度转换
void DS18B20_Start(void)// ds1820 start convert
{
    DS18B20_Rst();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc);// skip rom
    DS18B20_Write_Byte(0x44);// convert
}

//初始化 DS18B20 的 IO 口 DQ 同时检测 DS 的存在
//返回 1:不存在
//返回 0:存在
u8 DS18B20_Init(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOG, ENABLE); //使能 PG 口时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;           //PORTG.11 推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOG, &GPIO_InitStructure);           //初始化 GPIO
```



```
GPIO_SetBits(GPIOG,GPIO_Pin_11);           //输出 1
DS18B20_Rst();
return DS18B20_Check();
} //从 ds18b20 得到温度值
//精度: 0.1C
//返回值: 温度值 (-550~1250)
short DS18B20_Get_Temp(void)
{
    u8 temp;
    u8 TL,TH;
    short tem;
    DS18B20_Start();                      // ds1820 start convert
    DS18B20_Rst();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc); // skip rom
    DS18B20_Write_Byte(0xbe); // convert
    TL=DS18B20_Read_Byte(); // LSB
    TH=DS18B20_Read_Byte(); // MSB
    if(TH>7)
    {
        TH=~TH;
        TL=~TL;
        temp=0; //温度为负
    }else temp=1; //温度为正
    tem=TH; //获得高八位
    tem<<=8;
    tem+=TL; //获得底八位
    tem=(float)tem*0.625; //转换
    if(temp) return tem; //返回温度值
    else return -tem;
}
```

该部分代码就是根据我们前面介绍的单总线操作时序来读取 DS18B20 的温度值的, DS18B20 的温度通过 DS18B20_Get_Temp 函数读取, 该函数的返回值为带符号的短整形数据, 返回值的范围为-550~1250, 其实就是温度值扩大了 10 倍。

然后我们打开 ds18b20.h, 该文件下面主要是一些 IO 口位带操作定义以及函数申明, 没有什么特别需要讲解的地方。最后打开 main.c, 该文件代码如下:

```
int main(void)
{
    u8 t=0;
    short temperature;
    delay_init(); //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600); //串口初始化波特率为 9600
```



```
LED_Init();           //LED 端口初始化
LCD_Init();          //LCD 初始化
KEY_Init();          //KEY 初始化

POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,200,16,16,"WarShip STM32");
LCD_ShowString(60,70,200,16,16,"DS18B20 TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/12");
while(DS18B20_Init()) //DS18B20 初始化
{
    LCD_ShowString(60,130,200,16,16,"DS18B20 Error");
    delay_ms(200);
    LCD_Fill(60,130,239,130+16,WHITE);
    delay_ms(200);
}
LCD_ShowString(60,130,200,16,16,"DS18B20 OK");
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(60,150,200,16,16,"Temp:   . C");
while(1)
{
    if(t%10==0)//每 100ms 读取一次
    {
        temperature=DS18B20_Get_Temp();
        if(temperature<0)
        {
            LCD_ShowChar(60+40,150,'-',16,0);           //显示负号
            temperature=-temperature;                   //转为正数
        }else LCD_ShowChar(60+40,150,' ',16,0);        //去掉负号
        LCD_ShowNum(60+40+8,150,temperature/10,2,16);  //显示正数部分
        LCD_ShowNum(60+40+32,150,temperature%10,1,16); //显示小数部分
    }
    delay_ms(10);
    t++;
    if(t==20)
    {
        t=0;
        LED0=!LED0;
    }
}
```

主函数代码很简单，一系列初始化之后，就是每 100ms 读取一次 18B20 的值，然后转化为



温度后显示在 LCD 上。至此，我们本章的软件设计就结束了。

35.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示开始显示当前的温度值（假定 DS18B20 已经接上去了），如图 35.4.1 所示：



图 35.4.1 DS18B20 读取到的温度值

该程序还可以读取并显示负温度值的，只是由于本人在广州，是没办法看到了（除非放到冰箱），具备条件的读者可以测试一下。



第三十六章 DHT11 数字温湿度传感器实验

上一章，我们介绍了数字温度传感器 DS18B20 的使用，本章我们将介绍数字温湿度传感器 DHT11 的使用，该传感器不但能测温度，还能测湿度。本章我们将向大家介绍如何使用 STM32 来读取 DHT11 数字温湿度传感器，从而得到环境温度和湿度等信息，并把从温湿度值显示在 TFTLCD 模块上。本章分为如下几个部分：

- 36.1 DHT11 简介
- 36.2 硬件设计
- 36.3 软件设计
- 36.4 下载验证



36.1 DHT11 简介

DHT11 是一款湿温度一体化的数字传感器。该传感器包括一个电阻式测湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。通过单片机等微处理器简单的电路连接就能够实时的采集本地湿度和温度。DHT11 与单片机之间能采用简单的单总线进行通信，仅仅需要一个 I/O 口。传感器内部湿度和温度数据 40Bit 的数据一次性传给单片机，数据采用校验和方式进行校验，有效的保证数据传输的准确性。DHT11 功耗很低，5V 电源电压下，工作平均最大电流 0.5mA。

DHT11 的技术参数如下：

- 工作电压范围：3.3V-5.5V
- 工作电流：平均 0.5mA
- 输出：单总线数字信号
- 测量范围：湿度 20~90%RH，温度 0~50°C
- 精度：湿度±5%，温度±2°C
- 分辨率：湿度 1%，温度 1°C

DHT11 的管脚排列如图 36.1.1 所示：

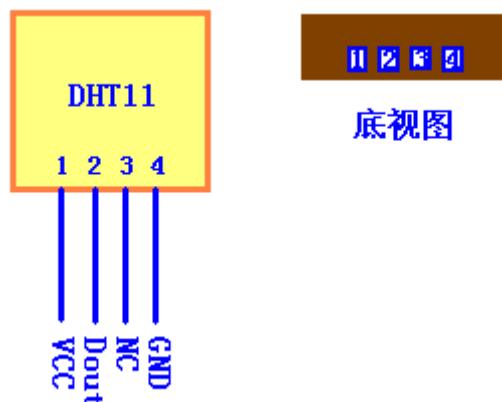


图 36.1.1 DHT11 管脚排列图

虽然 DHT11 与 DS18B20 类似，都是单总线访问，但是 DHT11 的访问，相对 DS18B20 来说要简单很多。下面我们先来看看 DHT11 的数据结构。

DHT11 数字湿温度传感器采用单总线数据格式。即，单个数据引脚端口完成输入输出双向传输。其数据包由 5Byte (40Bit) 组成。数据分小数部分和整数部分，一次完整的数据传输为 40bit，高位先出。DHT11 的数据格式为：8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数数据+8bit 校验和。其中校验和数据为前四个字节相加。

传感器数据输出的是未编码的二进制数据。数据(湿度、温度、整数、小数)之间应该分开处理。例如，某次从 DHT11 读到的数据如图 36.1.2 所示：

byte4	byte3	byte2	byte1	byte0
00101101	00000000	00011100	00000000	01001001
整数	小数	整数	小数	校验和
湿度		温度		校验和

图 36.1.2 某次读取到 DHT11 的数据

由以上数据就可得到湿度和温度的值，计算方法：



湿度 = byte4 . byte3=45.0 (%RH)

温度 = byte2 . byte1=28.0 (°C)

校验= byte4+ byte3+ byte2+ byte1=73(=湿度+温度)(校验正确)

可以看出, DHT11 的数据格式是十分简单的,DHT11 和 MCU 的一次通信最大为 3ms 左右,建议主机连续读取时间间隔不要小于 100ms。

下面, 我们介绍一下 DHT11 的传输时序。DHT11 的数据发送流程如图 36.1.3 所示:

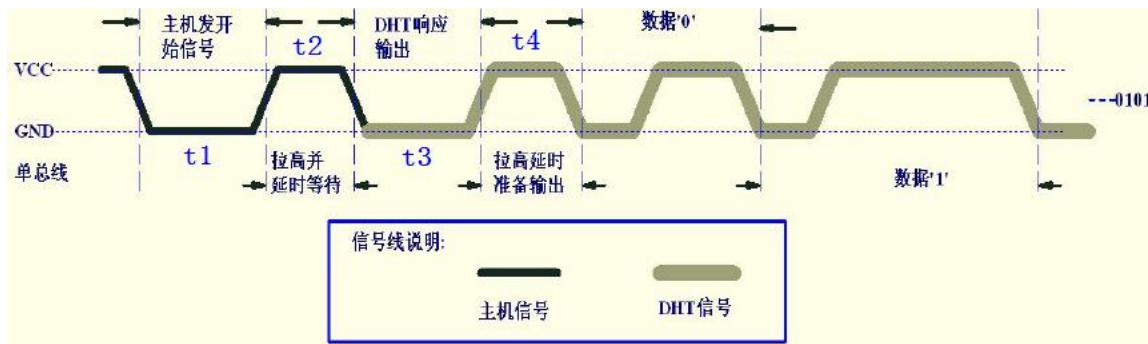


图 36.1.3 DHT11 数据发送流程

首先主机发送开始信号, 即: 拉低数据线, 保持 t_1 (至少 18ms) 时间, 然后拉高数据线 t_2 (20~40us) 时间, 然后读取 DHT11 的响应, 正常的话, DHT11 会拉低数据线, 保持 t_3 (40~50us) 时间, 作为响应信号, 然后 DHT11 拉高数据线, 保持 t_4 (40~50us) 时间后, 开始输出数据。

DHT11 输出数字 ‘0’ 的时序如图 36.1.4 所示:

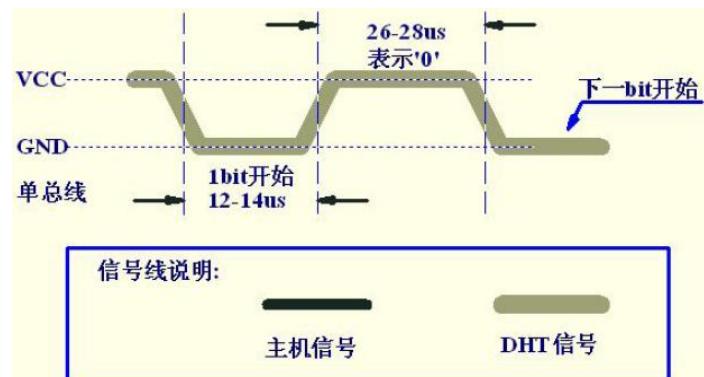


图 36.1.4 DHT11 数字 ‘0’ 时序

DHT11 输出数字 ‘1’ 的时序如图 36.1.5 所示:

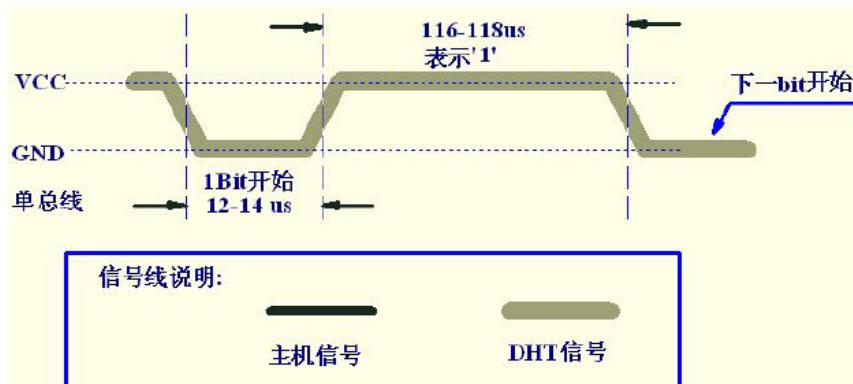


图 36.1.5 DHT11 数字 ‘1’ 时序



通过以上了解，我们就可以通过 STM32 来实现对 DHT11 的读取了。DHT11 的介绍就到这里，更详细的介绍，请参考 DHT11 的数据手册。

36.2 硬件设计

由于开发板上标准配置是没有 DHT11 这个传感器的，只有接口，所以要做本章的实验，大家必须找一个 DHT11 插在预留的 DHT11 接口上。

本章实验功能简介：开机的时候先检测是否有 DHT11 存在，如果没有，则提示错误。只有在检测到 DHT11 之后才开始读取温湿度值，并显示在 LCD 上，如果发现了 DHT11，则程序每隔 100ms 左右读取一次数据，并把温湿度显示在 LCD 上。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) TFTLCD 模块
- 3) DHT11 接口
- 4) DHT11 温湿度传感器

这些我们都已经介绍过了，DHT11 的接口和 DS18B20 的接口是共用一个的，不过 DHT11 有 4 条腿，需要把 U13 的 4 个接口都用上，将 DHT11 传感器插入到这个上面就可以通过 STM32 来读取温湿度值了。连接示意图如图 36.2.1 所示：

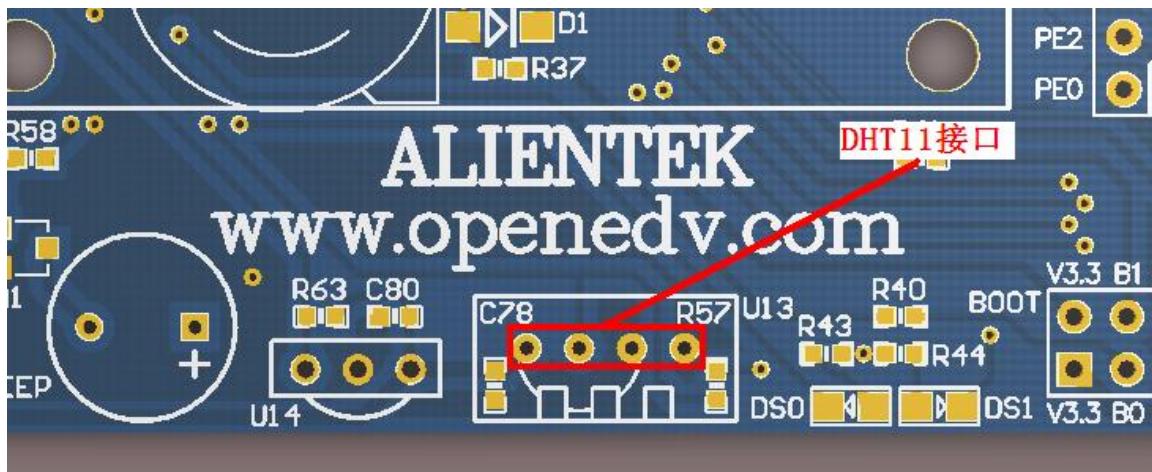


图 36.2.1 DHT11 连接示意图

这里要注意，将 DHT11 贴有字的一面朝内，而有很多孔的一面朝外，然后插入如图所示的四个孔内就可以了。

36.3 软件设计

打开 DHT11 数字温湿度传感器实验工程可以发现，我们在工程中添加了 dht11.c 文件和 dht11.h 文件，所有 DHT11 相关的驱动代码和定义都在这两个文件中。

打开 dht11.c 代码如下：

```
#include "dht11.h"
#include "delay.h"
//复位 DHT11
void DHT11_Rst(void)
```



```
{  
    DHT11_IO_OUT(); //SET OUTPUT  
    DHT11_DQ_OUT=0; //拉低 DQ  
    delay_ms(20); //拉低至少 18ms  
    DHT11_DQ_OUT=1; //DQ=1  
    delay_us(30); //主机拉高 20~40us  
}  
//等待 DHT11 的回应  
//返回 1:未检测到 DHT11 的存在  
//返回 0:存在  
u8 DHT11_Check(void)  
{  
    u8 retry=0;  
    DHT11_IO_IN();//SET INPUT  
    while (DHT11_DQ_IN&&retry<100) //DHT11 会拉低 40~80us  
    {  
        retry++;  
        delay_us(1);  
    };  
    if(retry>=100)return 1;  
    else retry=0;  
    while (!DHT11_DQ_IN&&retry<100)//DHT11 拉低后会再次拉高 40~80us  
    {  
        retry++;  
        delay_us(1);  
    };  
    if(retry>=100)return 1;  
    return 0;  
}  
//从 DHT11 读取一个位  
//返回值: 1/0  
u8 DHT11_Read_Bit(void)  
{  
    u8 retry=0;  
    while(DHT11_DQ_IN&&retry<100)//等待变为低电平  
    {  
        retry++;  
        delay_us(1);  
    }  
    retry=0;  
    while(!DHT11_DQ_IN&&retry<100)//等待变高电平  
    {  
        retry++;  
    }  
}
```



```
    delay_us(1);
}
delay_us(40);//等待 40us
if(DHT11_DQ_IN) return 1;
else return 0;
}

//从 DHT11 读取一个字节
//返回值：读到的数据
u8 DHT11_Read_Byte(void)
{
    u8 i,dat;
    dat=0;
    for (i=0;i<8;i++)
    {
        dat<<=1;
        dat|=DHT11_Read_Bit();
    }
    return dat;
}

//从 DHT11 读取一次数据
//temp:温度值(范围:0~50° )
//humi:湿度值(范围:20%~90%)
//返回值：0,正常;1,读取失败
u8 DHT11_Read_Data(u8 *temp,u8 *humi)
{
    u8 buf[5];
    u8 i;
    DHT11_Rst();
    if(DHT11_Check()==0)
    {
        for(i=0;i<5;i++)//读取 40 位数据
        {
            buf[i]=DHT11_Read_Byte();
        }
        if((buf[0]+buf[1]+buf[2]+buf[3]==buf[4])
        {
            *humi=buf[0];
            *temp=buf[2];
        }
    }
    else return 1;
    return 0;
}

//初始化 DHT11 的 IO 口 DQ 同时检测 DHT11 的存在
```



```
//返回 1:不存在
//返回 0:存在
u8 DHT11_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOG, ENABLE); //使能 PG 端口时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;           //PG11 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;      //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOG, &GPIO_InitStructure);             //初始化 IO 口
    GPIO_SetBits(GPIOG, GPIO_Pin_11);                   //PG11 输出高

    DHT11_Rst();           //复位 DHT11
    return DHT11_Check(); //等待 DHT11 的回应
}
```

该部分代码首先是通过函数 DHT11_Init 初始化传感器，然后根据我们前面介绍的单总线操作时序来读取 DHT11 的温湿度值的，DHT11 的温湿度值通过 DHT11_Read_Data 函数读取，如果返回 0，则说明读取成功，返回 1，则说明读取失败。同样我们打开 dht11.h 可以看到，头文件中主要是一些端口配置以及函数申明，代码比较简单。接下来我们打开 main.c，该文件代码如下：

```
int main(void)
{
    u8 t=0;
    u8 temperature;
    u8 humidity;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //初始化 LCD
    KEY_Init();              //初始化 KEY

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"DS18B20 TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/12");
    while(DHT11_Init()) //DHT11 初始化
    {
        LCD_ShowString(60,130,200,16,16,"DHT11 Error");
        delay_ms(200);
        LCD_Fill(60,130,239,130+16,WHITE);
```



```
delay_ms(200);
}

LCD_ShowString(60,130,200,16,16,"DHT11 OK");
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(60,150,200,16,16,"Temp:  C");
LCD_ShowString(60,170,200,16,16,"Humi:  % ");
while(1)
{
    if(t%10==0)//每 100ms 读取一次
    {
        DHT11_Read_Data(&temperature,&humidity);      //读取温湿度值

        LCD_ShowNum(60+40,150,temperature,2,16);      //显示温度
        LCD_ShowNum(60+40,170,humidity,2,16);          //显示湿度
    }
    delay_ms(10);
    t++;
    if(t==20)
    {
        t=0;
        LED0=!LED0;
    }
}
```

主函数比较简单，进行一系列初始化后，如果 DHT11 初始化成功，那么每隔 100ms 读取一次转换数据并显示在液晶上。至此，我们本章的软件设计就结束了。

36.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示开始显示当前的温度值（假定 DHT11 已经接上去了），如图 36.4.1 所示：

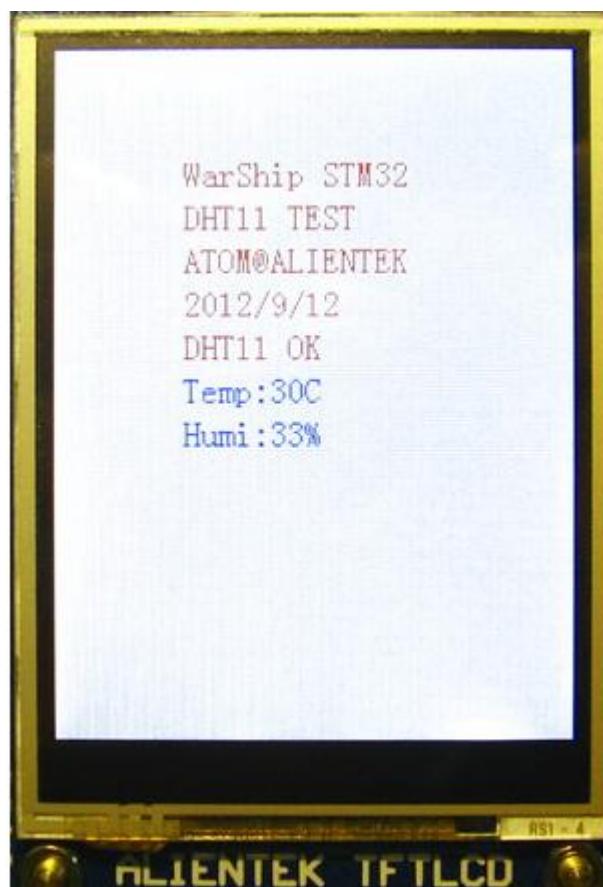


图 36.4.1 DHT11 读取到的温湿度值

至此，本章实验结束。大家可以将本章通过 DHT11 读取到的温度值，和前一章的通过 DS18B20 读取到的温度值对比一下，看看哪个更准确？

第三十七章 无线通信实验

ALIENTKE 战舰 STM32 开发板带有一个 2.4G 无线模块（NRF24L01 模块）通信接口，采用 8 脚插针方式与开发板连接。本章我们将以 NRF24L01 模块为例向大家介绍如何在 ALIENTEK 战舰 STM32 开发板上实现无线通信。在本章中，我们将使用两块战舰 STM32 开发板，一块用于发送收据，另外一块用于接收，从而实现无线数据传输。本章分为如下几个部分：

37.1 NRF24L01 无线模块简介

37.2 硬件设计

37.3 软件设计

37.4 下载验证



37.1 NRF24L01 无线模块简介

NRF24L01 无线模块，采用的芯片是 NRF24L01，该芯片的主要特点如下：

- 1) 2.4G 全球开放的 ISM 频段，免许可证使用。
- 2) 最高工作速率 2Mbps，高校的 GFSK 调制，抗干扰能力强。
- 3) 125 个可选的频道，满足多点通信和调频通信的需要。
- 4) 内置 CRC 检错和点对多点的通信地址控制。
- 5) 低工作电压 (1.9~3.6V)。
- 6) 可设置自动应答，确保数据可靠传输。

该芯片通过 SPI 与外部 MCU 通信，最大的 SPI 速度可以达到 10Mhz。本章我们用到的模块是深圳云佳科技生产的 NRF24L01，该模块已经被很多公司大量使用，成熟度和稳定性都是相当不错的。该模块的外形和引脚图如图 37.1.1 所示：

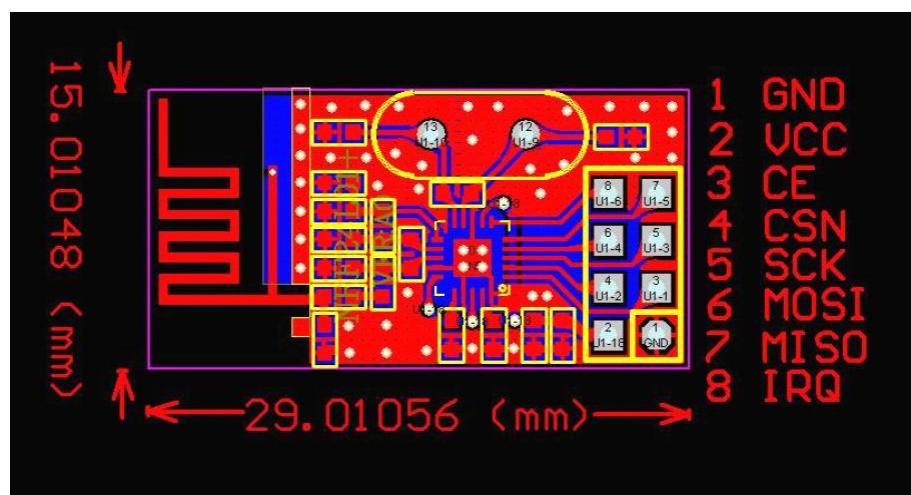


图 37.1.1 NRF24L01 无线模块外观引脚图

模块 VCC 脚的电压范围为 1.9~3.6V，建议不要超过 3.6V，否则可能烧坏模块，一般用 3.3V 电压比较合适。除了 VCC 和 GND 脚，其他引脚都可以和 5V 单片机的 IO 口直连，正是因为其兼容 5V 单片机的 IO，故使用上具有很大优势。

关于 NRF24L01 的详细介绍，请参考 NRF24L01 的技术手册。

37.2 硬件设计

本章实验功能简介：开机的时候先检测 NRF24L01 模块是否存在，在检测到 NRF24L01 模块之后，根据 KEY0 和 KEY1 的设置来决定模块的工作模式，在设定好工作模式之后，就会不停的发送/接收数据，同样用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) KEY0 和 KEY1 按键
- 3) TFTLCD 模块
- 4) NRF24L01 接口
- 5) NRF24L01 模块

NRF24L01 模块属于外部模块，这里我们仅介绍 NRF24L01 接口和 STM32 的连接情况，他们的连接关系如图 37.2.1 所示：

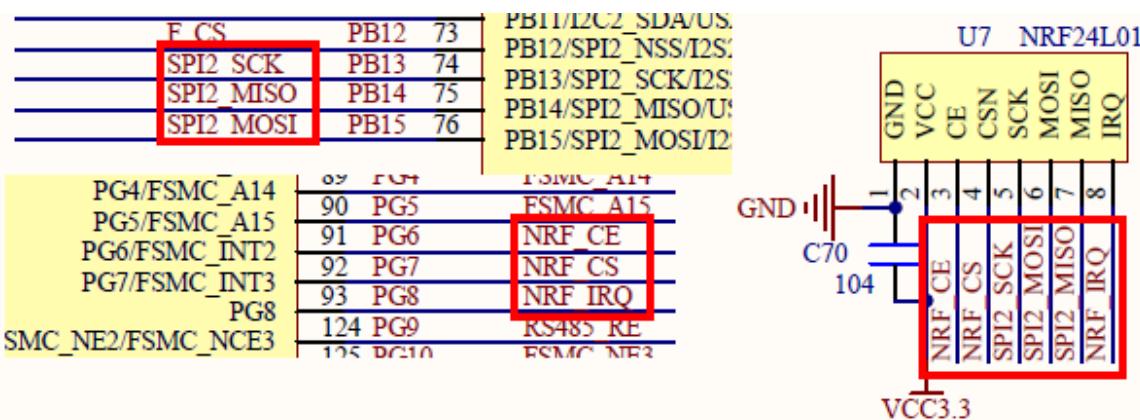


图 37.2.1 NRF24L01 模块接口与 STM32 连接原理图

这里NRF24L01也是使用的SPI2，和W25Q64以及SD卡等共用一个SPI接口，所以在使用的时候，他们分时复用SPI2。本章我们需要把SD卡和W25Q64的片选信号置高，以防止这两个器件对NRF24L01的通信造成干扰。

由于无线通信实验是双向的，所以至少要有两个模块同时能工作，这里我们使用2套ALIENTEK战舰STM32开发板来向大家演示。

37.3 软件设计

打开我们的无线通信实验项目工程，可以看到我们加入了 24l01.c 文件和 24l01.h 头文件，所有 24L01 相关的驱动代码和定义都在这两个文件中实现。同时，我们还加入了之前的 spi 驱动文件 spi.c 和 spi.h 头文件，因为 24L01 是通过 SPI 接口通信的。

打开 24l01.c 文件，代码如下：

```
#include "24l01.h"
#include "lcd.h"
#include "delay.h"
#include "spi.h"
#include "usart.h"

const u8 TX_ADDRESS[TX_ADR_WIDTH]={0x34,0x43,0x10,0x10,0x01}; //发送地址
const u8 RX_ADDRESS[RX_ADR_WIDTH]={0x34,0x43,0x10,0x10,0x01}; //发送地址
//初始化 24L01 的 IO 口
void NRF24L01_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef SPI_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|
                           RCC_APB2Periph_GPIOD|
                           RCC_APB2Periph_GPIOG, ENABLE); //使能 PB,D,G 端口时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12; //PB12 上拉 防止 W25X 的干扰
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化指定 IO
    GPIO_SetBits(GPIOB,GPIO_Pin_12); //上拉
}
```



```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;           //PD2 上拉 禁止 SD 卡的干扰
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
GPIO_SetBits(GPIOD,GPIO_Pin_2);                  //初始化指定 IO

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7; //PG6 7 推挽
GPIO_Init(GPIOG, &GPIO_InitStructure);           //初始化指定 IO

GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_8;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;      //PG8 输入
GPIO_Init(GPIOG, &GPIO_InitStructure);           //初始化指定 IO

GPIO_ResetBits(GPIOG,GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8); //PG6,7,8 上拉

SPI2_Init();          //初始化 SPI
SPI_Cmd(SPI2, DISABLE); // SPI 外设不使能

SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //设置
//SPI 单向或者双向的数据模式:SPI 设置为双线双向全双工
SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //设置 SPI 工作模式:设置为主 SPI
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; // 8 位帧结构
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; //时钟悬空低
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge; //数据捕获于第 1 个时钟沿
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //NSS 信号由软件控制
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_16;
//定义波特率预分频的值:波特率预分频值为 16
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; // MSB 位开始
SPI_InitStructure.SPI_CRCPolynomial = 7; //CRC 值计算的多项式
SPI_Init(SPI2, &SPI_InitStructure); //根据指定的参数初始化外设 SPIx

SPI_Cmd(SPI2, ENABLE); //使能 SPI 外设
NRF24L01_CE=0;        //使能 24L01
NRF24L01_CSN=1;       //SPI 片选取消
}

//检测 24L01 是否存在
//返回值:0, 成功;1, 失败
u8 NRF24L01_Check(void)
{
    u8 buf[5]={0XA5,0XA5,0XA5,0XA5,0XA5};
    u8 i;
    SPI2_SetSpeed(SPI_BaudRatePrescaler_4); //spi 速度为 9Mhz
    NRF24L01_Write_Buf(WRITE_REG_NRF+TX_ADDR,buf,5); //写入 5 个字节的地址.
    NRF24L01_Read_Buf(TX_ADDR,buf,5); //读出写入的地址
```



```
for(i=0;i<5;i++)if(buf[i]!=0XA5)break;
if(i!=5)return 1; //检测 24L01 错误
return 0;          //检测到 24L01
}

//SPI 写寄存器
//reg:指定寄存器地址
//value:写入的值
u8 NRF24L01_Write_Reg(u8 reg,u8 value)
{
    u8 status;
    NRF24L01_CSN=0;           //使能 SPI 传输
    status =SPI2_ReadWriteByte(reg); //发送寄存器号
    SPI2_ReadWriteByte(value);   //写入寄存器的值
    NRF24L01_CSN=1;           //禁止 SPI 传输
    return(status);            //返回状态值
}

//读取 SPI 寄存器值
//reg:要读的寄存器
u8 NRF24L01_Read_Reg(u8 reg)
{
    u8 reg_val;
    NRF24L01_CSN = 0;         //使能 SPI 传输
    SPI2_ReadWriteByte(reg);  //发送寄存器号
    reg_val=SPI2_ReadWriteByte(0XFF); //读取寄存器内容
    NRF24L01_CSN = 1;         //禁止 SPI 传输
    return(reg_val);          //返回状态值
}

//在指定位置读出指定长度的数据
//reg:寄存器(位置)
//*pBuf:数据指针
//len:数据长度
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Read_Buf(u8 reg,u8 *pBuf,u8 len)
{
    u8 status,u8_ctr;
    NRF24L01_CSN = 0;           //使能 SPI 传输
    status=SPI2_ReadWriteByte(reg); //发送寄存器值(位置),并读取状态值
    for(u8_ctr=0;u8_ctr<len;u8_ctr++)pBuf[u8_ctr]=SPI2_ReadWriteByte(0XFF);
    NRF24L01_CSN=1;             //关闭 SPI 传输
    return status;               //返回读到的状态值
}

//在指定位置写指定长度的数据
//reg:寄存器(位置)
```



```
/*pBuf:数据指针
//len:数据长度
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Write_Buf(u8 reg, u8 *pBuf, u8 len)
{
    u8 status,u8_ctr;
    NRF24L01_CSN = 0;           //使能 SPI 传输
    status = SPI2_ReadWriteByte(reg); //发送寄存器值(位置),并读取状态值
    for(u8_ctr=0; u8_ctr<len; u8_ctr++)SPI2_ReadWriteByte(*pBuf++); //写入数据
    NRF24L01_CSN = 1;           //关闭 SPI 传输
    return status;              //返回读到的状态值
}

//启动 NRF24L01 发送一次数据
//txbuf:待发送数据首地址
//返回值:发送完成状况
u8 NRF24L01_TxPacket(u8 *txbuf)
{
    u8 sta;
    SPI2_SetSpeed(SPI_BaudRatePrescaler_8); //spi 速度为 9Mhz  NRF24L01_CE=0;
    NRF24L01_Write_Buf(WR_TX_PLOAD,txbuf,TX_PLOAD_WIDTH); //写数据到
//TX BUF 32 个字节
    NRF24L01_CE=1;           //启动发送
    while(NRF24L01_IRQ!=0);   //等待发送完成
    sta=NRF24L01_Read_Reg(STATUS); //读取状态寄存器的值
    NRF24L01_Write_Reg(WRITE_REG_NRF+STATUS,sta); //清除 TX_DS 或
//MAX_RT 中断标志
    if(sta&MAX_TX)           //达到最大重发次数
    {
        NRF24L01_Write_Reg(FLUSH_RX,0xff); //清除 RX FIFO 寄存器
        return MAX_TX;
    }
    if(sta&TX_OK)//发送完成
    {
        return TX_OK;
    }
    return 0xff;//其他原因发送失败
}

//启动 NRF24L01 发送一次数据
//txbuf:待发送数据首地址
//返回值:0, 接收完成; 其他, 错误代码
u8 NRF24L01_RxPacket(u8 *rdbuf)
{
    u8 sta;
```



```
SPI2_SetSpeed(SPI_BaudRatePrescaler_8); // 9Mhz
sta=NRF24L01_Read_Reg(STATUS); //读取状态寄存器的值
NRF24L01_Write_Reg(WRITE_REG_NRF+STATUS,sta); //清除 TX_DS 或
//MAX_RT 中断标志
if(sta&RX_OK)//接收到数据
{
    NRF24L01_Read_Buf(RD_RX_PLOAD,rxbuf,RX_PLOAD_WIDTH); //读取数据
    NRF24L01_Write_Reg(FLUSH_RX,0xff); //清除 RX FIFO 寄存器
    return 0;
}
return 1;//没收到任何数据
}

//该函数初始化 NRF24L01 到 RX 模式
//设置 RX 地址,写 RX 数据宽度,选择 RF 频道,波特率和 LNA HCURR
//当 CE 变高后,即进入 RX 模式,并可以接收数据了
void NRF24L01_RX_Mode(void)
{
    NRF24L01_CE=0;
    NRF24L01_Write_Buf(WRITE_REG_NRF+RX_ADDR_P0,
                        (u8*)RX_ADDRESS,RX_ADR_WIDTH);
    NRF24L01_Write_Reg(WRITE_REG_NRF+EN_AA,0x01); //使能通道 0 的自动应答
    NRF24L01_Write_Reg(WRITE_REG_NRF+EN_RXADDR,0x01); //使能通道 0 的接收地址

    NRF24L01_Write_Reg(WRITE_REG_NRF+RF_CH,40); //设置 RF 通信频率

    NRF24L01_Write_Reg(WRITE_REG_NRF+RX_PW_P0,RX_PLOAD_WIDTH); //选择通道
    //0 的有效数据宽度
    NRF24L01_Write_Reg(WRITE_REG_NRF+RF_SETUP,0x0f); //设置 TX 发射参数,
    //0db 增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(WRITE_REG_NRF+CONFIG, 0x0f); //配置基本工作模式的参
    //数;PWR_UP,EN_CRC,16BIT_CRC,接收模式
    NRF24L01_CE = 1; //CE 为高,进入接收模式
}

//该函数初始化 NRF24L01 到 TX 模式
//设置 TX 地址,写 TX 数据宽度,设置 RX 自动应答的地址,
//填充 TX 发送数据,选择 RF 频道,波特率和 LNA HCURR
//PWR_UP,CRC 使能
//当 CE 变高后,即进入 RX 模式,并可以接收数据了
//CE 为高大于 10us,则启动发送.
void NRF24L01_TX_Mode(void)
{
    NRF24L01_CE=0;
    NRF24L01_Write_Buf(WRITE_REG_NRF+TX_ADDR,
```



```
(u8*)TX_ADDRESS,TX_ADR_WIDTH);
NRF24L01_Write_Buf(WRITE_REG_NRF+RX_ADDR_P0,
(u8*)RX_ADDRESS,RX_ADR_WIDTH); //设置 TX 节点地址,主要为了使能 ACK
    NRF24L01_Write_Reg(WRITE_REG_NRF+EN_AA,0x01); //使能通道 0 的自动应答
    NRF24L01_Write_Reg(WRITE_REG_NRF+EN_RXADDR,0x01); //使能通道 0 的接收地址
    NRF24L01_Write_Reg(WRITE_REG_NRF+SETUP_RETR,0x1a); //设置自动重发间隔时
                                                //间:500us + 86us;最大自动重发次数:10 次
    NRF24L01_Write_Reg(WRITE_REG_NRF+RF_CH,40); //设置 RF 通道为 40
    NRF24L01_Write_Reg(WRITE_REG_NRF+RF_SETUP,0x0f); //设置 TX 发射参数,0db
                                                //增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(WRITE_REG_NRF+CONFIG,0xe0); //配置基本工作模式的参
                                                //数;PWR_UP,EN_CRC,16BIT_CRC,接收模式,开启所有中断
    NRF24L01_CE=1;//CE 为高,10us 后启动发送
}
```

此部分代码我们不多介绍，在这里强调一个要注意的地方，在 NRF24L01_Init 函数里面，我们调用了 SPI2_Init() 函数，该函数我们在第二十八章曾有提到，在第二十八章的设置里面，SCK 空闲时为高，但是 NRF24L01 的 SPI 通信时序如图 37.3.1 所示：

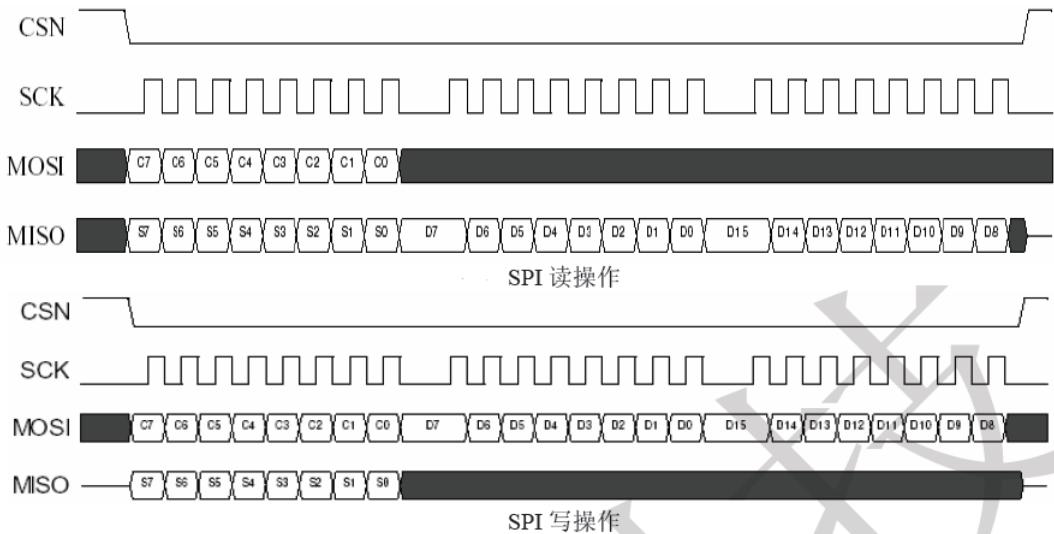


图 37.3.1 NRF24L01 读写操作时序

上图中 Cn 代表指令位，Sn 代表状态寄存器位，Dn 代表数据位。从图中可以看出，SCK 空闲的时候是低电平的，而数据在 SCK 的上升沿被读写。所以，我们需要设置 SPI 的 CPOL 和 CPHA 均为 0，来满足 NRF24L01 对 SPI 操作的要求。所以，我们在 NRF24L01_Init 函数里面又单独添加了将 CPOL 和 CPHA 设置为 0 的代码。

接下来我们看看 24101.h 代码，该头文件主要定义了一些 24L01 的命令字以及函数声明，这里还通过 TX_PLOAD_WIDTH 和 RX_PLOAD_WIDTH 决定了发射和接收的数据宽度，也就是我们每次发射和接受的有效字节数。NRF24L01 每次最多传输 32 个字节，再多的字节传输则需要多次传送。

最后我们看看主函数代码。打开 main.c 文件代码如下：

```
int main(void)
{
    u8 key,mode;
```



```
u16 t=0;
u8 tmp_buf[33];
delay_init();           //延时函数初始化
NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
uart_init(9600);        //串口初始化波特率为 9600
LED_Init();            //LED 端口初始化
LCD_Init();            //初始化 LCD
KEY_Init();            //按键初始化
NRF24L01_Init();       //初始化 NRF24L01

POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,200,16,16,"WarShip STM32");
LCD_ShowString(60,70,200,16,16,"NRF24L01 TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/13");
while(NRF24L01_Check()) //检查 NRF24L01 是否在位.
{
    LCD_ShowString(60,130,200,16,16,"NRF24L01 Error");
    delay_ms(200);
    LCD_Fill(60,130,239,130+16,WHITE);
    delay_ms(200);
}
LCD_ShowString(60,130,200,16,16,"NRF24L01 OK");
while(1)//在该部分确定进入哪个模式!
{
    key=KEY_Scan(0);
    if(key==KEY_RIGHT)
    {
        mode=0;
        break;
    }
    else if(key==KEY_DOWN)
    {
        mode=1;
        break;
    }
    t++;
}
if(t==100)LCD_ShowString(10,150,230,16,16,"KEY0:RX_Mode
                                         KEY1:TX_Mode"); //闪烁显示提示信息
if(t==200)
{
    LCD_Fill(10,150,230,150+16,WHITE);
    t=0;
}
delay_ms(5);
}
LCD_Fill(10,150,240,166,WHITE); //清空上面的显示
POINT_COLOR=BLUE;           //设置字体为蓝色
```



```
if(mode==0)          //RX 模式
{
    LCD_ShowString(60,150,200,16,16,"NRF24L01 RX_Mode");
    LCD_ShowString(60,170,200,16,16,"Received DATA:");
    NRF24L01_RX_Mode();
    while(1)
    {
        if(NRF24L01_RxPacket(tmp_buf)==0)//一旦接收到信息,则显示出来.
        {   tmp_buf[32]=0;           //加入字符串结束符
            LCD_ShowString(0,190,239,32,16,tmp_buf);
        }else delay_us(100);
        t++;
        if(t==10000)             //大约 1s 钟改变一次状态
        {   t=0;
            LED0=!LED0;
        }
    };
}
else//TX 模式
{
    LCD_ShowString(60,150,200,16,16,"NRF24L01 TX_Mode");
    NRF24L01_TX_Mode();
    mode=' ';//从空格键开始
    while(1)
    {
        if(NRF24L01_TxPacket(tmp_buf)==TX_OK)
        {   LCD_ShowString(60,170,239,32,16,"Sended DATA:");
            LCD_ShowString(0,190,239,32,16,tmp_buf);
            key=mode;
            for(t=0;t<32;t++)
            {
                key++;
                if(key>('~'))key=' ';
                tmp_buf[t]=key;
            }
            mode++;
            if(mode>'~')mode=' ';
            tmp_buf[32]=0;//加入结束符
        }else
        {   LCD_ShowString(60,170,239,32,16,"Send Failed ");
            LCD_Fill(0,188,240,218,WHITE);//清空上面的显示
        };
        LED0=!LED0;
        delay_ms(1500);
    };
}
```



```
}
```

主函数显示进行一系列外设初始化，然后 check 无线模块，检测通过后，通过按键扫描检测按键来决定是进入发送还是接受模式。至此，我们整个实验的软件设计就完成了。

37.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 37.4.1 所示的内容（默认 NRF24L01 已经接上了）：

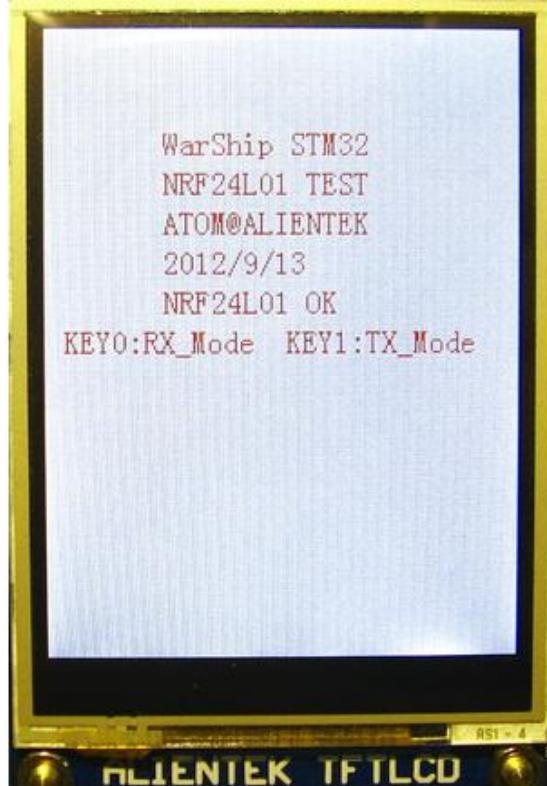


图 37.4.1 选择工作模式界面

通过 KEY0 和 KEY1 来选择 NRF24L01 模块所要进入的工作模式，我们两个开发板一个选择发送，一个选择接收就可以了。

设置好后通信界面如图 37.4.2 所示：

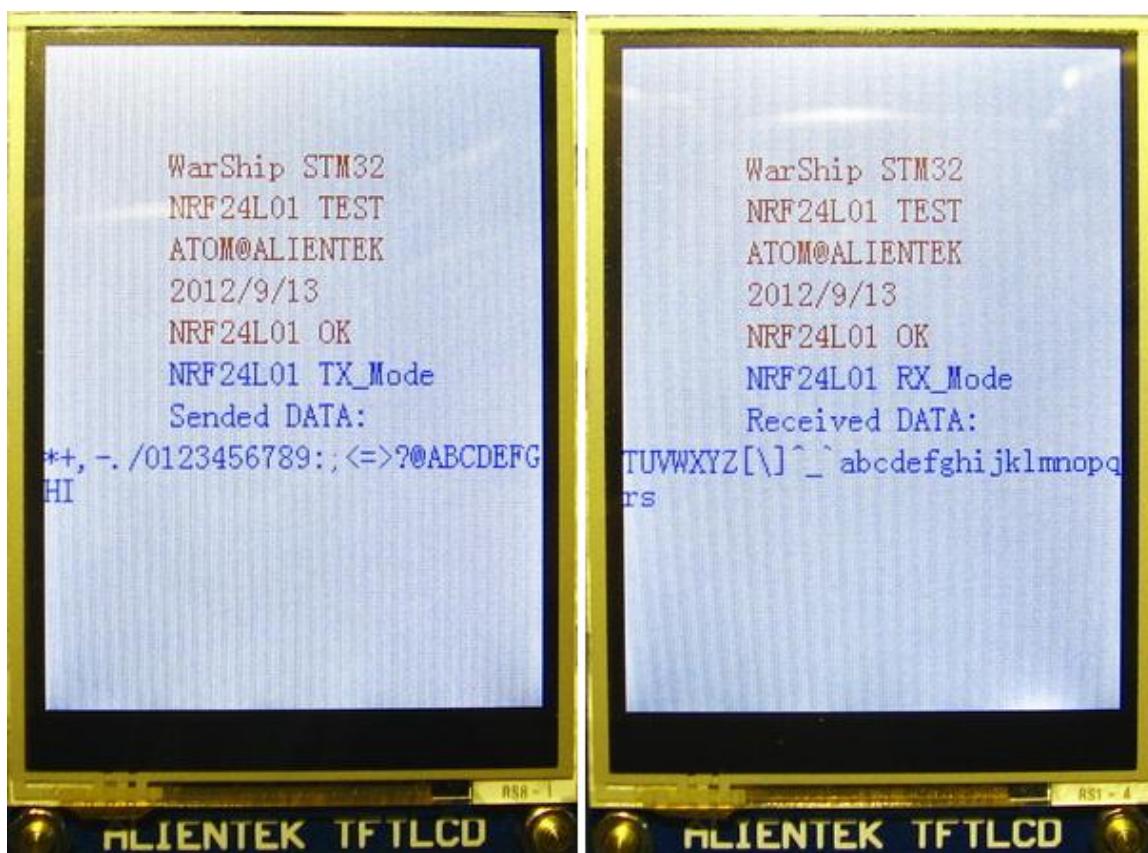


图 37.4.2 通信界面

第三十八章 PS2 鼠标实验

PS/2 作为电脑的标准输入接口，用于鼠标键盘等设备。PS/2 只需要一个简单的接口（2个 IO 口），就可以外扩鼠标、键盘等，是单片机理想的输入外扩方式。

ALIENTEK 战舰 STM32 开发板也自带了一个 PS/2 接口，可以用来驱动标准的鼠标、键盘等外设，也可以用来驱动一些 PS/2 接口的小键盘，条码扫描枪等。在本章中，我们将向大家介绍，如何在 ALIENTEK 战舰 STM32 开发板上，通过 PS/2 接口来驱动电脑鼠标。本章分为如下几个部分：

- 38.1 PS/2 简介
- 38.2 硬件设计
- 38.3 软件设计
- 38.4 下载验证



38.1 PS/2 简介

PS/2 是电脑上常见的接口之一，用于鼠标、键盘等设备。一般情况下，PS/2 接口的鼠标为绿色，键盘为紫色。

PS/2 接口是输入装置接口，而不是传输接口。所以 PS2 口根本没有传输速率的概念，只有扫描速率。在 Windows 环境下，ps/2 鼠标的采样率默认为 60 次/秒，USB 鼠标的采样率为 120 次/秒。较高的采样率理论上可以提高鼠标的移动精度。

物理上的 PS/2 端口可有 2 种，一种是 5 脚的，一种是六脚的。下面给出这两种 PS/2 接口的引脚定义图，如图 38.1.1 所示：

Male 公的	Female 母的	5-pin DIN (AT/XT):	5 脚 DIN(AT/XT)
		1 - Clock 2 - Data 3 - Not Implemented 4 - Ground 5 - +5v	1-时钟 2-数据 3-未实现，保留 4-电源地 5-电源+5V
(Plug) 插头	(Socket) 插座		

Male 公的	Female 母的	6-pin Mini-DIN (PS/2):	6 脚 Mini-DIN(PS/2)
		1 - Data 2 - Not Implemented 3 - Ground 4 - +5v 5 - Clock 6 - Not Implemented	1-数据 2-未实现，保留 3-电源地 4-电源+5V 5-时钟 6-未实现，保留
(Plug) 插头	(Socket) 插座		

图 38.1.1 PS/2 引脚定义图

从图 38.1.1 可以看出，不管是 5 脚还是 6 脚的 PS/2 接头，都是有 4 根有用的线连接：时钟线、数据线、电源线、地线。PS/2 设备的电源是 5V 的，而数据线和时钟线均是集电极开路的，这两根信号线都需要接一个上拉电阻（开发板上使用的是 10K）。

PS/2 鼠标和键盘遵循一种双向同步串行协议，换句话说每次数据线上发送一位数据并且每在时钟线上发一个脉冲就被读入。键盘/鼠标可以发送数据到主机，而主机也可以发送数据到设备，但主机总是在总线上有优先权，它可以在任何时候抑制来自于键盘/鼠标的通讯，只要把时钟拉低即可。

从设备到主机的数据在时钟信号的下降沿被主机读取，而从主机到设备的数据在时钟信号的上升沿被设备读取。不论通信方向如何，时钟总是由设备产生的，最大的时钟频率为 33Khz，大多数设备工作在 10~20Khz。

鼠标键盘，采用的是一种每帧包含 11/12 位的串行协议，这些位的含义如表 38.1.1 所示：

bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
应答位	停止位	校验位									起始位
仅在主机 到设备通 信中存在	总为1	奇校验	MSB						LSB	总为0	

表 38.1.2 鼠标/键盘帧数据格式

表 38.1.2 中校验位的含义是：如果数据位中包含偶数个 1，则校验位为 1；如果数据位中包含奇数个 1，则校验位为 0。数据位中的 1 的个数加上校验位总为奇数（奇校验），用于数据侦错。当主机发送数据给键盘/鼠标的时候，设备会发送一个握手信号来应答数据已经被收到了，



该位不会出现在设备到主机的通信中。

设备到主机的通信过程：

正常情况下数据线和时钟线都是高电平，当键盘/鼠标有数据要发送时，它先检测时钟线，确认时钟线是高电平。如果不是，则是主机抑制了通信，设备必须缓冲任何要发送的数据，直到重新获得总线的控制权（键盘有 16 字节的缓冲区而鼠标的缓冲区仅存储最后一个要发送的数据包）。如果时钟线是高电平，设备就可以开始传送数据了。

设备到主机的数据在时钟线的下降沿被主机读入，如图 38.1.2 所示：

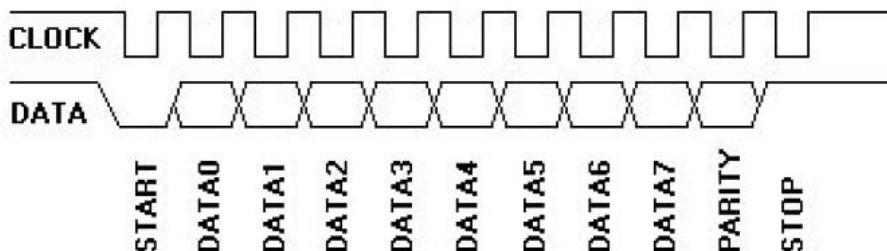


图 38.1.2 设备到主机通信时序图

主机可以在设备发送数据的时候拉低时钟线来放弃当前数据的传送。

主机到设备的通信过程：

主机到设备的通信与设备到主机的通信有点不同，因为 PS/2 的时钟总是由设备产生的，如果主机要发送数据，则它必须首先把时钟线和数据线设置为请求发送状态。请求发送状态通过如下过程实现：

1. 拉低时钟线至少 100us 以抑制通信。
2. 拉低数据线，以应用“请求发送”，然后释放时钟线。

设备在不超过 10ms 的时间内就会检测这个状态，当设备检测到这个状态后，它将开始产生时钟信号，并且在设备提供的时钟脉冲驱动下输入八个数据位和一个停止位。主机仅当时钟线为低的时候改变数据线，而数据在时钟脉冲的上升沿被锁存，这与发生在设备到主机通讯的过程中正好相反。

主机到设备的通信时序图如图 38.1.3 所示：

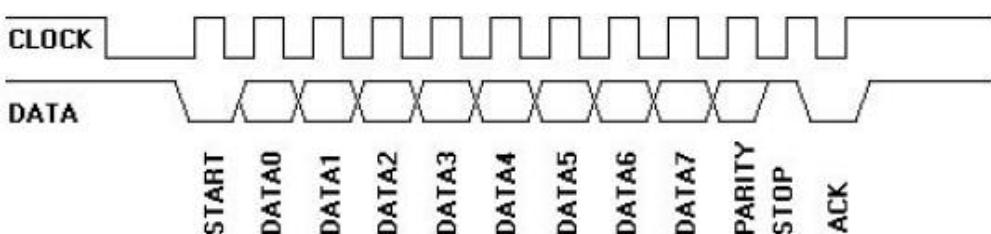


图 38.1.3 主机到设备通信时序图

以上简单介绍了 PS/2 协议的通信过程，更多的介绍请参考《PS/2 技术参考》一文。本章我们要驱动一个 PS/2 鼠标，所以接下来简单介绍一下 PS/2 鼠标的相关信息。

标准的 PS/2 鼠标支持下面的输入：X（左右）位移、Y（上下）位移、左键、中键和右键。但是我们目前用到鼠标大都还有滚轮，有的还有更多的按键，这就是所谓的 Intellimouse。它支持 5 个鼠标按键和三个位移轴（左右、上下和滚轮）。

标准的鼠标有两个计数器保持位移的跟踪：X 位移计数器和 Y 位移计数器。可存放 9 位的 2 进制补码，并且每个计数器都有相关的溢出标志。它们的内容连同三个鼠标按钮的状态一起以三字节移动数据包的形式发送给主机，位移计数器表示从最后一次位移数据包被送往主机后所发生的位移量。

标准 PS/2 鼠标发送唯一和按键信息以 3 字节的数据包格式发给主机，三个数据包的意义如图 38.1.4 所示：

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2					X Movement			
Byte 3					Y Movement			

图 38.1.4 标准鼠标位移数据包格式

位移计数器是一个 9 位 2 的补码整数，其最高位作为符号位出现在位移数据包的第一个字节里。这些计数器在鼠标读取输入发现有位移时被更新。这些值是自从最后一次发送位移数据包给主机后位移的累计量（即最后一次包发给主机后位移计数器被复位位移计数器可表示的值的范围是-255 到+255）。如果超过了范围，相应的溢出位就会被置位，并在复位之前，计数器不会再增减。

而所谓的 Intellimouse，因为多了 2 个按键和一个滚轮，所以 Intellimouse 的一个位移数据包由 4 个字节组成，如图 38.1.5 所示：

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2					X Movement			
Byte 3					Y Movement			
Byte 4	Always 0	Always 0	5th Btn	4th Btn	Z3	Z2	Z1	Z0

图 38.1.5 Intellimouse 鼠标位移数据包格式

Z0-Z3 是 2 的补码，用于表示从上次数据报告以来滚轮的位移量。有效范围从-8 到+7，第四键如果按下，则 4th Btn 位被置位，如果没有按下，则 4th Btn 位为 0。第五键也与此类似。

鼠标的介绍我们就简单的介绍到这里，详细的说明请参考光盘《PS/2 技术参考》第三章 PS/2 鼠标接口（第 36 页）。

38.2 硬件设计

本章实验功能简介：开机的时候先检测是否有鼠标接入，如果没有/检测错误，则提示错误代码。只有在检测到 PS/2 鼠标之后才开始后续操作，当检测到鼠标之后，就在 LCD 上显示鼠标位移数据包的内容，并转换为坐标值，在 LCD 上显示，如果有按键按下，则会提示按下的哪个按键。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) TFTLCD 模块
- 3) PS/2 鼠标/键盘接口
- 4) PS/2 鼠标

本章需要用到一个PS/2接口的鼠标，大家得自备一个。下面我来看一看开发板PS/2接口与 STM32 的连接电路，如图38.2.1所示：

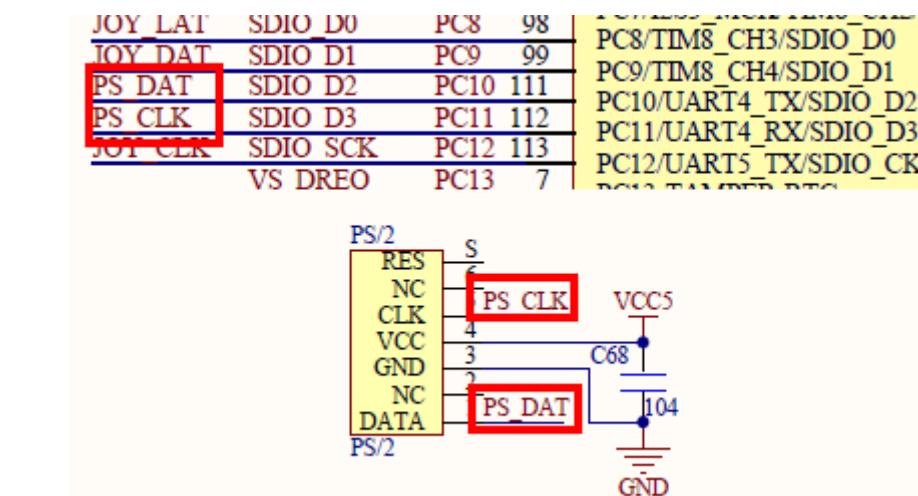


图 38.2.1 PS/2 接口与 STM32 的连接电路图

可以看到，PS/2 接口与 STM32 的连接仅仅 2 个 IO 口，其中 PS_CLK 连接在 PC11 上面，而 PS_DAT 则连接在 PC10 上面，这两个口和 SDIO_D2 和 SDIO_D3 复用了，所以在 SDIO 使用的时候，就不能使用 PS/2 设备了，这个在使用的时候大家要注意一下。

38.3 软件设计

打开 PS2 鼠标实验的工程，可以看到我们在工程中新建了 ps2.c 和头文件 ps2.h 以及 mouse.c 和 mouse.h 头文件。

打开 ps2.c，代码如下：

```
#include "ps2.h"
#include "uart.h"

//PS2 产生的时钟频率在 10~20Khz(最大 33K)
//高/低电平的持续时间为 25~50us 之间.
//PS2_Status 当前状态标志
//[7]:接收到一次数据;[6]:校验错误;[5:4]:当前工作的模式;[3:0]:收到的数据长度;
u8 PS2_Status=CMDMODE; //默认为命令模式
u8 PS2_DATA_BUF[16]; //ps2 数据缓存区
//位计数器
u8 BIT_Count=0;
//中断 15~10 处理函数
//每 11 个 bit,为接收 1 个字节
//每接收完一个包(11 位)后,设备至少会等待 50ms 再发送下一个包
//只做了鼠标部分,键盘部分暂时未加入
//CHECK OK 2010/5/2
EXTI_InitTypeDef EXTI_InitStructure;
void EXTI15_10_IRQHandler(void)
{
    static u8 tempdata=0;
    static u8 parity=0;
    if(EXTI_GetITStatus(EXTI_Line11)==SET) //中断 11 产生了相应的中断
        {
            if((tempdata>=0x80)&&(tempdata<=0x9F)) //校验通过
                parity=0;
            else
                parity=1;
            if(parity==0)
                {
                    if(tempdata>=0x01)&&(tempdata<=0x0A))
                        PS2_Status=CMDMODE;
                    else if(tempdata>=0x0B)&&(tempdata<=0x0F))
                        PS2_Status=DATA_MODE;
                    else if(tempdata>=0x10)&&(tempdata<=0x1F))
                        PS2_Status=KEYBOARD_MODE;
                    else if(tempdata>=0x20)&&(tempdata<=0x2F))
                        PS2_Status=MOUSE_MODE;
                    else
                        PS2_Status=UNKNOWN_MODE;
                }
            else
                PS2_Status=ERR_MODE;
        }
}
```



```
{  
    EXTI_ClearITPendingBit(EXTI_Line11); //清除 LINE11 上的中断标志位  
    if(BIT_Count==0)  
    {  
        parity=0;  
        tempdata=0;  
    }  
    BIT_Count++;  
    if(BIT_Count>1&&BIT_Count<10)//这里获得数据  
    {  
        tempdata>>=1;  
        if(PS2_SDA)  
        {  
            tempdata|=0x80;  
            parity++;//记录 1 的个数  
        }  
    }  
}else if(BIT_Count==10)//得到校验位  
{  
    if(PS2_SDA)parity|=0x80;//校验位为 1  
}  
if(BIT_Count==11)//接收到 1 个字节的数据了  
{  
    BIT_Count=parity&0x7f;//取得 1 的个数  
  
    if(((BIT_Count%2==0)&&(parity&0x80))|((BIT_Count%2==1)&&(parity&0x80)==0))  
    {  
        //PS2_Status|=1<<7;//标记得到数据  
        BIT_Count=PS2_Status&0x0f;  
        PS2_DATA_BUF[BIT_Count]=tempdata;//保存数据  
        if(BIT_Count<15)PS2_Status++; //数据长度加 1  
        BIT_Count=PS2_Status&0x30; //得到模式  
        switch(BIT_Count)  
        {  
            case CMDMODE://命令模式下,每收到一个字节都会产生接收完成  
                PS2_Dis_Data_Report();//禁止数据传输  
                PS2_Status|=1<<7; //标记得到数据  
                break;  
            case KEYBOARD:  
                break;  
            case MOUSE:  
                if(MOUSE_ID==0)//标准鼠标,3 个字节  
                {  
                    if((PS2_Status&0x0f)==3)  
                    {  
                        PS2_Status|=1<<7; //标记得到数据  
                    }  
                }  
        }  
    }  
}
```



```
        PS2_Dis_Data_Report();//禁止数据传输
    }
}else if(MOUSE_ID==3)//扩展鼠标,4个字节
{
    if((PS2_Status&0x0f)==4)
    {
        PS2_Status|=1<<7;      //标记得到数据
        PS2_Dis_Data_Report();//禁止数据传输
    }
}
break;
}
}
else
{
    PS2_Status|=1<<6;    //标记校验错误
    PS2_Status&=0xf0;    //清除接收数据计数器
}
BIT_Count=0;
}
}
}

//禁止数据传输
//把时钟线拉低,禁止数据传输
//CHECK OK 2010/5/2
void PS2_Dis_Data_Report(void)
{
    PS2_Set_Int(0);          //关闭中断
    PS2_SET_SCL_OUT(); //设置 SCL 为输出
    PS2_SCL_OUT=0;          //抑制传输
}
//使能数据传输
//释放时钟线
//CHECK OK 2010/5/2
void PS2_En_Data_Report(void)
{
    PS2_SET_SCL_IN(); //设置 SCL 为输入
    PS2_SET_SDA_IN(); //SDA IN
    PS2_SCL_OUT=1;      //上拉
    PS2_SDA_OUT=1;
    PS2_Set_Int(1);      //开启中断
}

//PS2 中断屏蔽设置
```



```
//en:1, 开启;0, 关闭;
//CHECK OK 2010/5/2
void PS2_Set_Int(u8 en)
{
    EXTI_ClearITPendingBit(EXTI_Line11); //清除 EXTI11 线路挂起位

    if(en)
    {
        EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    }else{
        EXTI_InitStructure.EXTI_LineCmd = DISABLE;
    }
    EXTI_Init(&EXTI_InitStructure); //根据指定的参数初始化 EXTI

}

//等待 PS2 时钟线 sta 状态改变
//sta:1, 等待变为 1;0, 等待变为 0;
//返回值:0, 时钟线变成了 sta;1, 超时溢出;
//CHECK OK 2010/5/2
u8 Wait_PS2_Scl(u8 sta)
{
    u16 t=0;
    sta=!sta;
    while(PS2_SCL==sta)
    {
        delay_us(1);
        t++;
        if(t>16000) return 1;//时间溢出 (设备会在 10ms 内检测这个状态)
    }
    return 0;//被拉低了
}
//在发送命令/数据之后,等待设备应答,该函数用来获取应答
//返回得到的值
//返回 0, 且 PS2_Status.6=1, 则产生了错误
//CHECK OK 2010/5/2
u8 PS2_Get_Byte(void)
{
    u16 t=0;
    u8 temp=0;
    while(1)//最大等待 55ms
    {
        t++;
        delay_us(10);
```



```
if(PS2_Status&0x80)      //得到了一次数据
{
    temp=PS2_DATA_BUF[PS2_Status&0x0f-1];
    PS2_Status&=0x70;//清除计数器，接收到数据标记
    break;
} else if(t>5500||PS2_Status&0x40)break;//超时溢出/接收错误
}
PS2_En_Data_Report(); //使能数据传输
return temp;
}

//发送一个命令到 PS2.
//返回值:0, 无错误, 其他, 错误代码
u8 PS2_Send_Cmd(u8 cmd)
{
    u8 i;
    u8 high=0;           //记录 1 的个数
    PS2_Set_Int(0);      //屏蔽中断
    PS2_SET_SCL_OUT();   //设置 SCL 为输出
    PS2_SET_SDA_OUT();   //SDA OUT
    PS2_SCL_OUT=0;       //拉低时钟线
    delay_us(120);       //保持至少 100us
    PS2_SDA_OUT=0;       //拉低数据线
    delay_us(10);
    PS2_SET_SCL_IN();    //释放时钟线, 这里 PS2 设备得到第一个位, 开始位
    PS2_SCL_OUT=1;
    if(Wait_PS2_Scl(0)==0) //等待时钟拉低
    {
        for(i=0;i<8;i++)
        {
            if(cmd&0x01)
            {
                PS2_SDA_OUT=1;
                high++;
            } else PS2_SDA_OUT=0;
            cmd>>=1;
            //这些地方没有检测错误, 因为这些地方不会产生死循环
            Wait_PS2_Scl(1); //等待时钟拉高 发送 8 个位
            Wait_PS2_Scl(0); //等待时钟拉低
        }
        if((high%2)==0)PS2_SDA_OUT=1;//发送校验位 10
        else PS2_SDA_OUT=0;
        Wait_PS2_Scl(1); //等待时钟拉高 10 位
        Wait_PS2_Scl(0); //等待时钟拉低
    }
}
```



```
PS2_SDA_OUT=1; //发送停止位 11
Wait_PS2_Scl(1); //等待时钟拉高 11 位
PS2_SET_SDA_IN(); //SDA in
Wait_PS2_Scl(0); //等待时钟拉低
if(PS2_SDA==0)Wait_PS2_Scl(1);//等待时钟拉高 12 位
else
{
    PS2_En_Data_Report();
    return 1;//发送失败
}
}else
{
    PS2_En_Data_Report();
    return 2;//发送失败
}
PS2_En_Data_Report();
return 0; //发送成功
}

//PS2 初始化
//CHECK OK 2010/5/2
void PS2_Init(void)
{

    NVIC_InitTypeDef NVIC_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); //使能 PC 端口时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10|GPIO_Pin_11; //PC10,11
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure); //初始化指定端口

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOC,GPIO_PinSource11); //中断线 11
    //中断线初始化
    EXTI_InitStructureEXTI_Line=EXTI_Line11;
    EXTI_InitStructureEXTI_Mode = EXTI_Mode_Interrupt; //中断
    EXTI_InitStructureEXTI_Trigger = EXTI_Trigger_Rising; //上升沿出发
    EXTI_InitStructureEXTI_LineCmd = ENABLE; //使能中断线
    EXTI_Init(&EXTI_InitStructure); //根据指定的参数初始化 EXTI

    //中断分组初始化
    NVIC_InitStructure.NVIC IRQChannel = EXTI15_10_IRQn; //使能按键所在的中断通道
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 1; //抢占优先级 1
```



```
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;      //子优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;        //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);    //根据指定的参数初始化外设 NVIC 寄存器

}
```

该部分为底层的 PS/2 协议驱动程序，采用中断接收 PS/2 设备产生的时钟信号，然后解析。PS2_Init 函数主要是进行一些 GPIO 初始化，中断线映射以及中断分组设置。这里相信大家已经很熟悉这段代码了。头文件 ps2.h 中的代码很简单，相信大家都看得懂，这里不拿出来介绍了。然后打开 mouse.c，代码如下：

```
#include "mouse.h"
#include "uart.h"
#include "lcd.h"
u8 MOUSE_ID;//用来标记鼠标 ID
PS2_Mouse MouseX;
//处理 MOUSE 的数据
void Mouse_Data_Pro(void)
{
    MouseX.x_pos+=(signed char)PS2_DATA_BUF[1];
    MouseX.y_pos+=(signed char)PS2_DATA_BUF[2];
    MouseX.z_pos+=(signed char)PS2_DATA_BUF[3];
    MouseX.bt_mask=PS2_DATA_BUF[0]&0X07;//取出掩码
}
//初始化鼠标
//返回:0,初始化成功
//其他:错误代码
//CHECK OK 2010/5/2
u8 Init_Mouse(void)
{
    u8 t;
    PS2_Init();
    delay_ms(800);          //等待上电复位完成
    PS2_Status=CMDMODE;     //进入命令模式
    t=PS2_Send_Cmd(PS_RESET); //复位鼠标
    if(t!=0)return 1;
    t=PS2_Get_Byte();
    if(t!=0XAFA)return 2;
    t=0;
    while((PS2_Status&0x80)==0)//等待复位完毕
    {
        t++;
        delay_ms(10);
        if(t>50)return 3;
    }
}
```



```
PS2_Get_Byte(); //得到 0XAA
PS2_Get_Byte(); //得到 ID 0X00
//进入滚轮模式的特殊初始化序列
PS2_Send_Cmd(SET_SAMPLE_RATE); //进入设置采样率
if(PS2_Get_Byte() != 0XFA) return 4; //传输失败
PS2_Send_Cmd(0XC8); //采样率 200
if(PS2_Get_Byte() != 0XFA) return 5; //传输失败
PS2_Send_Cmd(SET_SAMPLE_RATE); //进入设置采样率
if(PS2_Get_Byte() != 0XFA) return 6; //传输失败
PS2_Send_Cmd(0X64); //采样率 100
if(PS2_Get_Byte() != 0XFA) return 7; //传输失败
PS2_Send_Cmd(SET_SAMPLE_RATE); //进入设置采样率
if(PS2_Get_Byte() != 0XFA) return 8; //传输失败
PS2_Send_Cmd(0X50); //采样率 80
if(PS2_Get_Byte() != 0XFA) return 9; //传输失败
//序列完成
PS2_Send_Cmd(GET_DEVICE_ID); //读取 ID
if(PS2_Get_Byte() != 0XFA) return 10; //传输失败
MOUSE_ID = PS2_Get_Byte(); //得到 MOUSE ID

PS2_Send_Cmd(SET_SAMPLE_RATE); //再次进入设置采样率
if(PS2_Get_Byte() != 0XFA) return 11; //传输失败
PS2_Send_Cmd(0X0A); //采样率 10
if(PS2_Get_Byte() != 0XFA) return 12; //传输失败
PS2_Send_Cmd(GET_DEVICE_ID); //读取 ID
if(PS2_Get_Byte() != 0XFA) return 13; //传输失败
MOUSE_ID = PS2_Get_Byte(); //得到 MOUSE ID

PS2_Send_Cmd(SET_RESOLUTION); //设置分辨率
if(PS2_Get_Byte() != 0XFA) return 14; //传输失败
PS2_Send_Cmd(0X03); //8 点/mm
if(PS2_Get_Byte() != 0XFA) return 15; //传输失败
PS2_Send_Cmd(SET_SCALING11); //设置缩放比率为 1:1
if(PS2_Get_Byte() != 0XFA) return 16; //传输失败

PS2_Send_Cmd(SET_SAMPLE_RATE); //设置采样率
if(PS2_Get_Byte() != 0XFA) return 17; //传输失败
PS2_Send_Cmd(0X28); //40
if(PS2_Get_Byte() != 0XFA) return 18; //传输失败

PS2_Send_Cmd(EN_DATA_REPORT); //使能数据报告
if(PS2_Get_Byte() != 0XFA) return 19; //传输失败
```



```
PS2_Status=MOUSE; //进入鼠标模式
return 0;           //无错误,初始化成功
}
```

该部分仅 2 个函数, Init_Mouse 用于初始化鼠标, 让鼠标进入 Intellimouse 模式, 里面的初始化序列完全按照《PS/2 技术参考》里面介绍的来设计。另外一个函数就是将收到的数据简单处理一下。 接下来打开 mouse.h 可以看到我们定义了一个鼠标结构体:

```
typedef struct
{
    short x_pos;    //横坐标
    short y_pos;    //纵坐标
    short z_pos;    //滚轮坐标
    u8  bt_mask;   //按键标识,bit2 中间键;bit1,右键;bit0,左键
} PS2_Mouse;
```

该鼠标结构体用于存放鼠标相关的数据, 并对鼠标的相关命令进行了宏定义(部分被省略)。最后, 打开 main.c 文件, 代码如下:

```
//显示鼠标的坐标值
//x,y:在 LCD 上显示的坐标位置
//pos:坐标值
void Mouse_Show_Pos(u16 x,u16 y,short pos)
{
    if(pos<0)
    {
        LCD_ShowChar(x,y,'-',16,0); //显示负号
        pos=-pos;                //转为正数
    }else LCD_ShowChar(x,y,',',16,0); //去掉负号
    LCD_ShowNum(x+8,y,pos,5,16); //显示值
}

int main(void)
{
    u8 t;
    u8 errcnt=0;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //初始化 LCD
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"Mouse TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/13");
```



```
while(Init_Mouse()) //检查鼠标是否在位.  
{  
    LCD_ShowString(60,130,200,16,16,"Mouse Error");  
    delay_ms(400);  
    LCD_Fill(60,130,239,130+16,WHITE);  
    delay_ms(100);  
}  
LCD_ShowString(60,130,200,16,16,"Mouse OK");  
LCD_ShowString(60,150,200,16,16,"Mouse ID:");  
LCD_ShowNum(132,150,MOUSE_ID,3,16);//填充模式  
POINT_COLOR=BLUE;  
LCD_ShowString(30,170,200,16,16,"BUF[0]:");  
LCD_ShowString(30,186,200,16,16,"BUF[1]:");  
LCD_ShowString(30,202,200,16,16,"BUF[2]:");  
if(MOUSE_ID==3)LCD_ShowString(30,218,200,16,16,"BUF[3]:");  
LCD_ShowString(90+30,170,200,16,16,"X POS:");  
LCD_ShowString(90+30,186,200,16,16,"Y POS:");  
LCD_ShowString(90+30,202,200,16,16,"Z POS:");  
if(MOUSE_ID==3)LCD_ShowString(90+30,218,200,16,16,"BUTTON:");  
t=0;  
while(1)  
{  
    if(PS2_Status&0x80)//得到了一次数据  
    {  
        LCD_ShowNum(56+30,170,PS2_DATA_BUF[0],3,16);//填充模式  
        LCD_ShowNum(56+30,186,PS2_DATA_BUF[1],3,16);//填充模式  
        LCD_ShowNum(56+30,202,PS2_DATA_BUF[2],3,16);//填充模式  
        if(MOUSE_ID==3)LCD_ShowNum(56+30,218,PS2_DATA_BUF[3],3,16);  
        Mouse_Data_Pro();//处理数据  
        Mouse_Show_Pos(146+30,170,MouseX.x_pos); //X 坐标  
        Mouse_Show_Pos(146+30,186,MouseX.y_pos); //Y 坐标  
        if(MOUSE_ID==3)Mouse_Show_Pos(146+30,202,MouseX.z_pos); //滚轮位置  
        if(MouseX.bt_mask&0x01)LCD_ShowString(146+30,218,200,16,16,"LEFT");  
        else LCD_ShowString(146+30,218,200,16,16,"      ");  
        if(MouseX.bt_mask&0x02)LCD_ShowString(146+30,234,200,16,16,"RIGHT");  
        else LCD_ShowString(146+30,234,200,16,16,"      ");  
        if(MouseX.bt_mask&0x04)LCD_ShowString(146+30,250,200,16,16,"MIDDLE");  
        else LCD_ShowString(146+30,250,200,16,16,"      ");  
        PS2_Status=MOUSE;  
        PS2_En_Data_Report();//使能数据报告  
    }else if(PS2_Status&0x40)  
    {  
        errcnt++;  
        PS2_Status=MOUSE;
```

```
        LCD_ShowNum(86+30,234,errcnt,3,16);//填充模式  
    }  
    t++;  
    delay_ms(1);  
    if(t==200)  
    {  
        t=0;  
        LED0=!LED0;  
    }  
}
```

此部分，除了 main 函数，我们还编写了 Mouse_Show_Pos 函数，用于在指定位置显示鼠标坐标值，并支持负数显示，通过该函数，可以方便我们显示鼠标坐标数据。至此，PS/2 鼠标实验的软件设计部分就结束了。

38.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 38.4.1 所示内容（假定 PS/2 鼠标已经接上，并且初始化成功）：

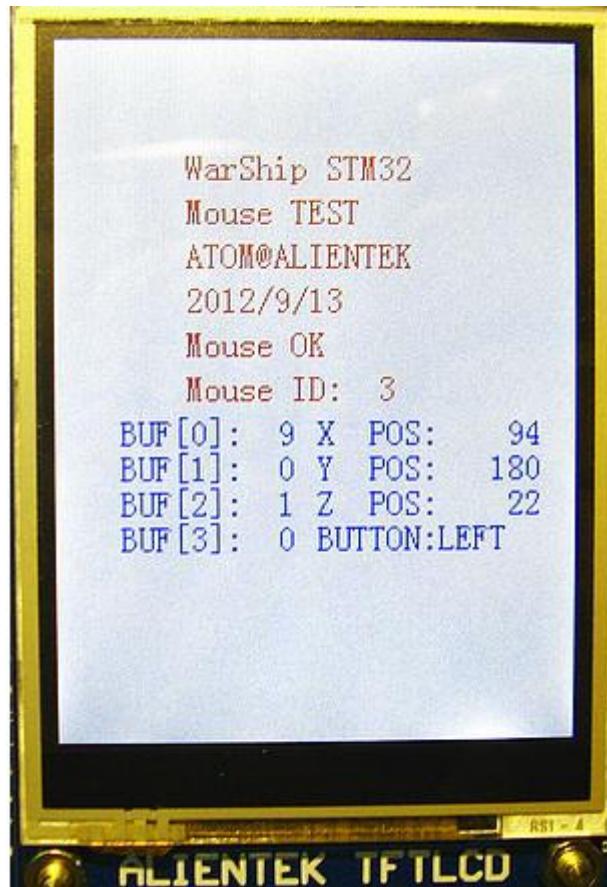


图 38.4.1 PS/2 鼠标实验显示结果

移动鼠标，或者按动按键，就可以看到上面的数据不断变化，证明我们的鼠标已经成功被驱动了，接下来我们就可以使用鼠标来控制 STM32 了。



第三十九章 FLASH 模拟 EEPROM 实验

STM32 本身没有自带 EEPROM，但是 STM32 具有 IAP（在应用编程）功能，所以我们可以把它的 FLASH 当成 EEPROM 来使用。本章，我们将利用 STM32 内部的 FLASH 来实现第二十八章类似的效果，不过这次我们是将数据直接存放在 STM32 内部，而不是存放在 W25Q64。本章分为如下几个部分：

- 39.1 STM32 FLASH 简介
- 39.2 硬件设计
- 39.3 软件设计
- 39.4 下载验证



39.1 STM32 FLASH 简介

不同型号的 STM32，其 FLASH 容量也有所不同，最小的只有 16K 字节，最大的则达到了 1024K 字节。战舰 STM32 开发板选择的 STM32F103ZET6 的 FLASH 容量为 512K 字节，属于大容量产品（另外还有中容量和小容量产品），大容量产品的闪存模块组织如图 39.1.1 所示：

块	名称	地址范围	长度(字节)
主存储器	页0	0x0800 0000 – 0x0800 07FF	2K
	页1	0x0800 0800 – 0x0800 0FFF	2K
	页2	0x0800 1000 – 0x0801 17FF	2K
	页3	0x0800 1800 – 0x0801 FFFF	2K
	.	.	.
	.	.	.
	页255	0x0807 F800 – 0x0807 FFFF	2K
信息块	启动程序代码	0x1FFF F000 – 0x1FFF F7FF	2K
	用户选择字节	0x1FFF F800 – 0x1FFF F80F	16
闪存存储器 接口寄存器	FLASH_ACR	0x4002 2000 – 0x4002 2003	4
	FLASH_KEYR	0x4002 2004 – 0x4002 2007	4
	FLASH_OPTKEYR	0x4002 2008 – 0x4002 200B	4
	FLASH_SR	0x4002 200C – 0x4002 200F	4
	FLASH_CR	0x4002 2010 – 0x4002 2013	4
	FLASH_AR	0x4002 2014 – 0x4002 2017	4
	保留	0x4002 2018 – 0x4002 201B	4
	FLASH_OBR	0x4002 201C – 0x4002 201F	4
	FLASH_WRPR	0x4002 2020 – 0x4002 2023	4

图 39.1.1 大容量产品闪存模块组织

STM32 的闪存模块由：主存储器、信息块和闪存存储器接口寄存器等 3 部分组成。

主存储器，该部分用来存放代码和数据常数（如 const 类型的数据）。对于大容量产品，其被划分为 256 页，每页 2K 字节。注意，小容量和中容量产品则每页只有 1K 字节。从上图可以看出主存储器的起始地址就是 0X08000000，B0、B1 都接 GND 的时候，就是从 0X08000000 开始运行代码的。

信息块，该部分分为 2 个小部分，其中启动程序代码，是用来存储 ST 自带的启动程序，用于串口下载代码，当 B0 接 V3.3，B1 接 GND 的时候，运行的就是这部分代码。用户选择字节，则一般用于配置写保护、读保护等功能，本章不作介绍。

闪存存储器接口寄存器，该部分用于控制闪存读写等，是整个闪存模块的控制机构。

对主存储器和信息块的写入由内嵌的闪存编程/擦除控制器(FPEC)管理；编程与擦除的高电压由内部产生。

在执行闪存写操作时，任何对闪存的读操作都会锁住总线，在写操作完成后读操作才能正确地进行；既在进行写或擦除操作时，不能进行代码或数据的读取操作。

闪存的读取

内置闪存模块可以在通用地址空间直接寻址，任何 32 位数据的读操作都能访问闪存模块的内容并得到相应的数据。读接口在闪存端包含一个读控制器，还包含一个 AHB 接口与 CPU 衔接。这个接口的主要工作是产生读闪存的控制信号并预取 CPU 要求的指令块，预取指令块仅用

于在 I-Code 总线上的取指操作，数据常量是通过 D-Code 总线访问的。这两条总线的访问目标是相同的闪存模块，访问 D-Code 将比预取指令优先级高。

这里要特别留意一个闪存等待时间，因为 CPU 运行速度比 FLASH 快得多，STM32F103 的 FLASH 最快访问速度 $\leq 24\text{Mhz}$ ，如果 CPU 频率超过这个速度，那么必须加入等待时间，比如我们一般使用 72Mhz 的主频，那么 FLASH 等待周期就必须设置为 2，该设置通过 FLASH_ACR 寄存器设置。

例如，我们要从地址 `addr`，读取一个半字（半字为 16 位，字为 32 位），可以通过如下的语句读取：

```
data=*(vu16*)addr;
```

将 `addr` 强制转换为 `vu16` 指针，然后取该指针所指向的地址的值，即得到了 `addr` 地址的值。类似的，将上面的 `vu16` 改为 `vu8`，即可读取指定地址的一个字节。相对 FLASH 读取来说，STM32 FLASH 的写就复杂一点了，下面我们介绍 STM32 闪存的编程和擦除。

闪存的编程和擦除

STM32 的闪存编程是由 FPEC（闪存编程和擦除控制器）模块处理的，这个模块包含 7 个 32 位寄存器，他们分别是：

- FPEC 键寄存器(FLASH_KEYR)
- 选择字节键寄存器(FLASH_OPTKEYR)
- 闪存控制寄存器(FLASH_CR)
- 闪存状态寄存器(FLASH_SR)
- 闪存地址寄存器(FLASH_AR)
- 选择字节寄存器(FLASH_OBR)
- 写保护寄存器(FLASH_WRPTR)

其中 FPEC 键寄存器总共有 3 个键值：

RDPRT 键=0X000000A5

KEY1=0X45670123

KEY2=0XCDEF89AB

STM32 复位后，FPEC 模块是被保护的，不能写入 FLASH_CR 寄存器；通过写入特定的序列到 FLASH_KEYR 寄存器可以打开 FPEC 模块（即写入 KEY1 和 KEY2），只有在写保护被解除后，我们才能操作相关寄存器。

STM32 闪存的编程每次必须写入 16 位（不能单纯的写入 8 位数据哦！），当 FLASH_CR 寄存器的 PG 位为‘1’时，在一个闪存地址写入一个半字将启动一次编程；写入任何非半字的数据，FPEC 都会产生总线错误。在编程过程中(BSY 位为‘1’)，任何读写闪存的操作都会使 CPU 暂停，直到此次闪存编程结束。

同样，STM32 的 FLASH 在编程的时候，也必须要求其写入地址的 FLASH 是被擦除了的（也就是其值必须是 0xFFFF），否则无法写入，在 FLASH_SR 寄存器的 PGERR 位将得到一个警告。

STM32 的 FLASH 编程过程如图 39.1.2 所示：

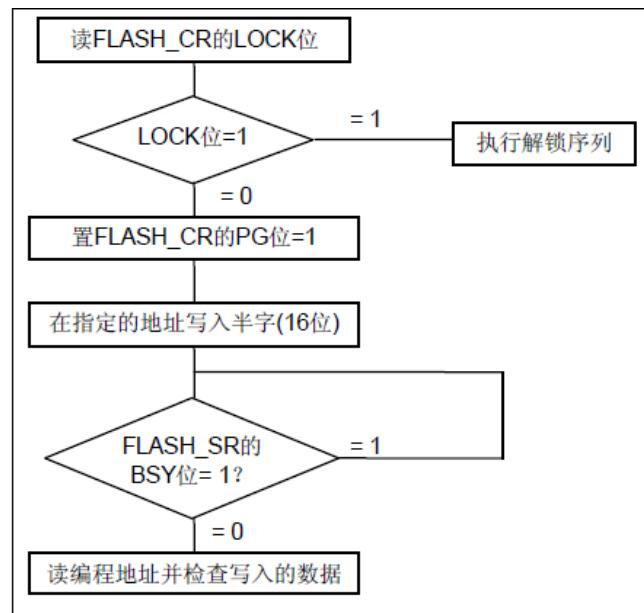


图 39.1.2 STM32 闪存编程过程

从上图可以得到闪存的编程顺序如下：

- 检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁
- 检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的编程操作
- 设置 FLASH_CR 寄存器的 PG 位为‘1’
- 在指定的地址写入要编程的半字
- 等待 BSY 位变为‘0’
- 读出写入的地址并验证数据

前面提到，我们在 STM32 的 FLASH 编程的时候，要先判断缩写地址是否被擦除了，所以，我们有必要再介绍一下 STM32 的闪存擦除，STM32 的闪存擦除分为两种：页擦除和整片擦除。页擦除过程如图 39.1.3 所示

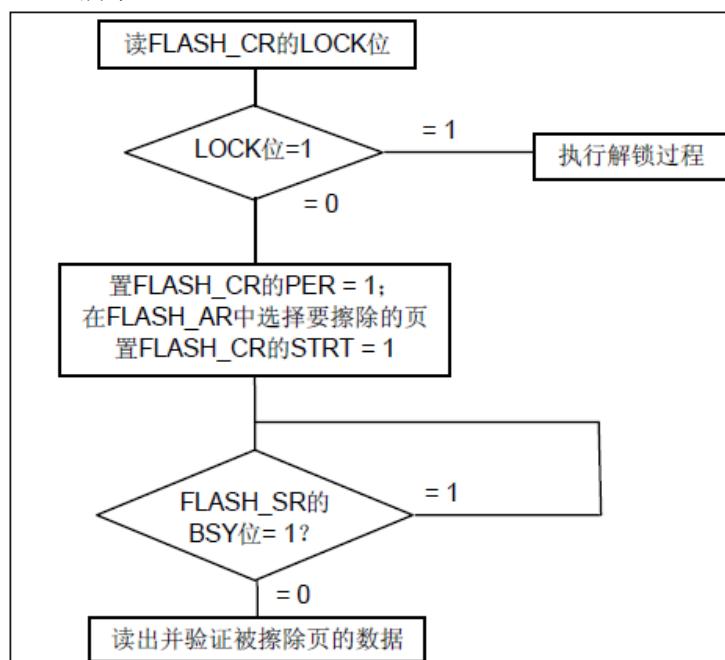


图 39.1.3 STM32 闪存页擦除过程



从上图可以看出，STM32 的页擦除顺序为：

- 检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁
- 检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的闪存操作
- 设置 FLASH_CR 寄存器的 PER 位为'1'
- 用 FLASH_AR 寄存器选择要擦除的页
- 设置 FLASH_CR 寄存器的 STRT 位为'1'
- 等待 BSY 位变为'0'
- 读出被擦除的页并做验证

本章，我们只用到了 STM32 的页擦除功能，整片擦除功能我们在这里就不介绍了。通过以上了解，我们基本上知道了 STM32 闪存的读写所要执行的步骤了，接下来，我们看看与读写相关的寄存器说明。

第一个介绍的是 FPEC 键寄存器：FLASH_KEYR。该寄存器各位描述如图 39.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FKEYR[31:16]															
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
FKEYR[15:0]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

注：所有这些位是只写的，读出时返回0。

位31~0	FKEYR : FPEC键 这些位用于输入FPEC的解锁键。
-------	--

图 39.1.4 寄存器 FLASH_KEYR 各位描述

该寄存器主要用来解锁 FPEC，必须在该寄存器写入特定的序列（KEY1 和 KEY2）解锁后，才能对 FLASH_CR 寄存器进行写操作。

第二个要介绍的是闪存控制寄存器：FLASH_CR。该寄存器的各位描述如图 39.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
res															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	EOPIE	保留	ERRIE	OPTWRE	保留	LOCK	STRT	OPTER	OPTPG	保留	MER	PER	PG		
res	rw	res	rw	rw	res	rw	rw	rw	rw	res	rw	rw	rw	rw	rw

图 39.1.5 寄存器 FLASH_CR 各位描述

该寄存器我们本章只用到了它的 LOCK、STRT、PER 和 PG 等 4 个位。

LOCK 位，该位用于指示 FLASH_CR 寄存器是否被锁住，该位在检测到正确的解锁序列后，硬件将其清零。在一次不成功的解锁操作后，在下次系统复位之前，该位将不再改变。

STRT 位，该位用于开始一次擦除操作。在该位写入 1，将执行一次擦除操作。

PER 位，该位用于选择页擦除操作，在页擦除的时候，需要将该位置 1。

PG 位，该位用于选择编程操作，在往 FLASH 写数据的时候，该位需要置 1。

FLASH_CR 的其他位，我们就不在这里介绍了，请大家参考《STM32F10xxx 闪存编程参考手册》第 18 页。

第三个要介绍的是闪存状态寄存器：FLASH_SR。该寄存器各位描述如图 39.1.6 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
res															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留															
res															

位31~6	保留。必须保持为清除状态'0'
位5	EOP: 操作结束 当闪存操作(编程/擦除)完成时，硬件设置这位为'1'，写入'1'可以清除这位状态。 注：每次成功的编程或擦除都会设置EOP状态。
位4	WRPRTERR: 写保护错误 试图对写保护的闪存地址编程时，硬件设置这位为'1'，写入'1'可以清除这位状态。
位3	保留。必须保持为清除状态'0'
位2	PGERR: 编程错误 试图对内容不是'0xFFFF'的地址编程时，硬件设置这位为'1'，写入'1'可以清除这位状态。 注：进行编程操作之前，必须先清除FLASH_CR寄存器的STRT位。
位1	保留。必须保持为清除状态'0'
位0	BSY: 忙 该位指示闪存操作正在进行。在闪存操作开始时，该位被设置为'1'；在操作结束或发生错误时该位被清除为'0'。

图 39.1.6 寄存器 FLASH_SR 各位描述

该寄存器主要用来指示当前 FPEC 的操作编程状态。

最后，我们再来看看闪存地址寄存器：FLASH_AR。该寄存器各位描述如图 39.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FAR[31:16]															
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
FAR[15:0]															
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

这些位由硬件修改为当前/最后使用的地址。在页擦除操作中，软件必须修改这个寄存器以指定要擦除的页。

位31~0	FAR: 闪存地址 当进行编程时选择要编程的地址，当进行页擦除时选择要擦除的页。 注意：当FLASH_SR中的BSY位为'1'时，不能写这个寄存器。
-------	---

图 39.1.7 寄存器 FLASH_AR 各位描述

该寄存器在本章，我们主要用来设置要擦除的页。

关于 STM32 FLASH 的基础知识介绍，我们就介绍到这。更详细的介绍，请参考《STM32F10xxx 闪存编程参考手册》。下面我们讲解使用 STM32 的官方固件库操作 FLASH 的几个常用函数。这些函数和定义分布在文件 stm32f10x_flash.c 以及 stm32f10x_flash.h 文件中。

1. 锁定解锁函数

上面讲解到在对 FLASH 进行写操作前必须先解锁，解锁操作也就是必须在 FLASH_KEYR 寄



存器写入特定的序列（KEY1 和 KEY2），固件库函数实现很简单：

```
void FLASH_Unlock(void);
```

同样的道理，在对 FLASH 写操作完成之后，我们要锁定 FLASH，使用的库函数是：

```
void FLASH_Lock(void);
```

2. 写操作函数

固件库提供了三个 FLASH 写函数：

```
FLASH_Status FLASH_ProgramWord(uint32_t Address, uint32_t Data);
FLASH_Status FLASH_ProgramHalfWord(uint32_t Address, uint16_t Data);
FLASH_Status FLASH_ProgramOptionByteData(uint32_t Address, uint8_t Data);
```

顾名思义分别为：FLASH_ProgramWord 为 32 位字写入函数，其他分别为 16 位半字写入和用户选择字节写入函数。这里需要说明，32 位字节写入实际上是写入的两次 16 位数据，写完第一次后地址+2，这与我们前面讲解的 STM32 闪存的编程每次必须写入 16 位并不矛盾。写入 8 位实际也是占用的两个地址了，跟写入 16 位基本上没啥区别。

3. 擦除函数

固件库提供三个 FLASH 擦除函数：

```
FLASH_Status FLASH_ErasePage(uint32_t Page_Address);
FLASH_Status FLASH_EraseAllPages(void);
FLASH_Status FLASH_EraseOptionBytes(void);
```

这三个函数可以顾名思义了，非常简单。

4. 获取 FLASH 状态

主要是用的函数是：

```
FLASH_Status FLASH_GetStatus(void);
```

返回值是通过枚举类型定义的：

```
typedef enum
{
    FLASH_BUSY = 1, //忙
    FLASH_ERROR_PG, //编程错误
    FLASH_ERROR_WRP, //写保护错误
    FLASH_COMPLETE, //操作完成
    FLASH_TIMEOUT //操作超时
}FLASH_Status;
```

从这里面我们可以看到 FLASH 操作的 5 个状态，每个代表的意思我们在后面注释了。

5. 等待操作完成函数

在执行闪存写操作时，任何对闪存的读操作都会锁住总线，在写操作完成后读操作才能正确地进行；既在进行写或擦除操作时，不能进行代码或数据的读取操作。

所以在每次操作之前，我们都要等待上一次操作完成这次操作才能开始。使用的函数是：

```
FLASH_Status FLASH_WaitForLastOperation(uint32_t Timeout)
```

入口参数为等待时间，返回值是 FLASH 的状态，这个很容易理解，这个函数本身我们在固件库中使用得不多，但是在固件库函数体中间可以多次看到。

6. 读 FLASH 特定地址数据函数

有写就必定有读，而读取 FLASH 指定地址的半字的函数固件库并没有给出来，这里我们自己写的一个函数：

```
u16 STMFLASH_ReadHalfWord(u32 faddr)
```



```
{  
    return *(vu16*)faddr;  
}
```

39.2 硬件设计

本章实验功能简介：开机的时候先显示一些提示信息，然后在主循环里面检测两个按键，其中1个按键（WK_UP）用来执行写入FLASH的操作，另外一个按键（KEY1）用来执行读出操作，在TFTLCD模块上显示相关信息。同时用DS0提示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) WK_UP 和 KEY1 按键
- 3) TFTLCD 模块
- 4) STM32 内部 FLASH

本章需要用到的资源和电路连接，在之前已经全部有介绍过了，接下来我们直接开始软件设计。

39.3 软件设计

打开我们的FLASH模拟EEPROM实验工程，可以看到我们添加了两个文件stmflash.c和stm32flash.h。同时我们还引入了固件库flash操作文件stm32f10x_flash.c和头文件stm32f10x_flash.h。

打开stmflash.c文件，代码如下：

```
#include "stmflash.h"  
#include "delay.h"  
#include "uart.h"  
//读取指定地址的半字(16位数据)  
//faddr:读地址(此地址必须为2的倍数!!)  
//返回值:对应数据.  
u16 STMFLASH_ReadHalfWord(u32 faddr)  
{  
    return *(vu16*)faddr;  
}  
#if STM32_FLASH_WREN //如果使能了写  
//不检查的写入  
//WriteAddr:起始地址  
//pBuffer:数据指针  
//NumToWrite:半字(16位)数  
void STMFLASH_Write_NoCheck(u32 WriteAddr,u16 *pBuffer,u16 NumToWrite)  
{  
    u16 i;  
    for(i=0;i<NumToWrite;i++)  
    {
```



```
FLASH_ProgramHalfWord(WriteAddr,pBuffer[i]);
WriteAddr+=2;//地址增加 2.
}
}

//从指定地址开始写入指定长度的数据
//WriteAddr:起始地址(此地址必须为 2 的倍数!!)
//pBuffer:数据指针
//NumToWrite:半字(16 位)数(就是要写入的 16 位数据的个数.)
#if STM32_FLASH_SIZE<256
#define STM_SECTOR_SIZE 1024 //字节
#else
#define STM_SECTOR_SIZE 2048
#endif
u16 STMFLASH_BUF[STM_SECTOR_SIZE/2];//最多是 2K 字节
void STMFLASH_Write(u32 WriteAddr,u16 *pBuffer,u16 NumToWrite)
{
    u32 secpos;          //扇区地址
    u16 secoff;          //扇区内偏移地址(16 位字计算)
    u16 secremain;        //扇区内剩余地址(16 位字计算)
    u16 i;
    u32 offaddr;         //去掉 0X08000000 后的地址
    if(WriteAddr<STM32_FLASH_BASE||(WriteAddr>=
(STM32_FLASH_BASE+1024*STM32_FLASH_SIZE)))return;//非法地址
    FLASH_Unlock();           //解锁
    offaddr=WriteAddr-STM32_FLASH_BASE;           //实际偏移地址.
    secpos=offaddr/STM_SECTOR_SIZE;
    secoff=(offaddr%STM_SECTOR_SIZE)/2; //在扇区内的偏移(2 个字节为基本单位.)
    secremain=STM_SECTOR_SIZE/2-secoff; //扇区剩余空间大小
    if(NumToWrite<=secremain)secremain=NumToWrite;//不大于该扇区范围
    while(1)
    {
        STMFLASH_Read(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE,
                      STMFLASH_BUF,STM_SECTOR_SIZE/2);    //读出整个扇区的内容
        for(i=0;i<secremain;i++)           //校验数据
        {
            if(STMFLASH_BUF[secoff+i]!=0xFFFF)break;//需要擦除
        }
        if(i<secremain) //需要擦除
        {
            //擦除这个扇区
            FLASH_ErasePage(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE);
            for(i=0;i<secremain;i++)//复制
            {
                STMFLASH_BUF[i+secoff]=pBuffer[i];
            }
        }
    }
}
```



```
        }

STMFLASH_Write_NoCheck(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE,
STMFLASH_BUFS,STM_SECTOR_SIZE/2); //写入整个扇区

} else STMFLASH_Write_NoCheck(WriteAddr,pBuffer,secremain); //写已经擦除
//了的,直接写入扇区剩余区间.

if(NumToWrite==secremain)break; //写入结束了
else //写入未结束
{
    secpos++; //扇区地址增 1
    secoff=0; //偏移位置为 0
    pBuffer+=secremain; //指针偏移
    WriteAddr+=secremain; //写地址偏移
    NumToWrite-=secremain; //字节(16 位)数递减
    //下一个扇区还是写不完
    if(NumToWrite>(STM_SECTOR_SIZE/2))secremain=STM_SECTOR_SIZE/2;
    else secremain=NumToWrite; //下一个扇区可以写完了
}
};

FLASH_Lock(); //上锁
}

#endif

//从指定地址开始读出指定长度的数据
//ReadAddr:起始地址
//pBuffer:数据指针
//NumToRead:半字(16 位)数
void STMFLASH_Read(u32 ReadAddr,u16 *pBuffer,u16 NumToRead)
{
    u16 i;
    for(i=0;i<NumToRead;i++)
    {
        pBuffer[i]=STMFLASH_ReadHalfWord(ReadAddr); //读取 2 个字节.
        ReadAddr+=2; //偏移 2 个字节.
    }
}

///////////////////////////////
//WriteAddr:起始地址
//WriteData:要写入的数据
void Test_Write(u32 WriteAddr,u16 WriteData)
{
    STMFLASH_Write(WriteAddr,&WriteData,1); //写入一个字
}
```



STMFLASH_ReadHalfWord()函数的实现原理我们在前面已经详细讲解了。下面我们重点介绍一下 STMFLASH_Write 函数，该函数用于在 STM32 的指定地址写入指定长度的数据，该函数的实现基本类似第 28 章的 SPI_Flash_Write 函数，不过该函数对写入地址是有要求的，必须保证以下两点：

- 1, 该地址必须是用户代码区以外的地址。
- 2, 该地址必须是 2 的倍数。

条件 1 比较好理解，如果把用户代码给卡擦了，可想而知你运行的程序可能就被废了，从而很可能出现死机的情况。条件 2 则是 STM32 FLASH 的要求，每次必须写入 16 位，如果你写的地址不是 2 的倍数，那么写入的数据，可能就不是写在你要写的地址了。

另外，该函数的 STMFLASH_BUF 数组，也是根据所用 STM32 的 FLASH 容量来确定的，战舰 STM32 开发板的 FLASH 是 512K 字节，所以 STM_SECTOR_SIZE 的值为 512，故该数组大小为 2K 字节。其他函数我们就不做介绍了。

然后打开 stmflash.h，该文件代码非常简单，我们就不做介绍了。

最后，打开 main.c 文件，main 函数如下：

```
const u8 TEXT_Buffer[]={"WarshipSTM32 FLASH TEST"};  
#define SIZE sizeof(TEXT_Buffer) //数组长度  
#define FLASH_SAVE_ADDR 0X08070000 //设置 FLASH 保存地址(必须为偶数)  
  
int main(void)  
{  
    u8 key;  
    u16 i=0;  
    u8 datatemp[SIZE];  
    delay_init(); //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级  
    uart_init(9600); //串口初始化波特率为 9600  
    LED_Init(); //LED 端口初始化  
    LCD_Init(); //初始化 LCD  
    KEY_Init(); //初始化 KEY  
  
    POINT_COLOR=RED;//设置字体为红色  
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");  
    LCD_ShowString(60,70,200,16,16,"FLASH EEPROM TEST");  
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(60,110,200,16,16,"2012/9/13");  
    LCD_ShowString(60,130,200,16,16,"WK_UP:Write KEY1:Read");  
    POINT_COLOR=BLUE;  
    //显示提示信息  
    POINT_COLOR=BLUE;//设置字体为蓝色  
    //FLASH_SetLatency(FLASH_ACR_LATENCY_2);  
    while(1)  
    {  
        key=KEY_Scan(0);
```



```
if(key==KEY_UP)//WK_UP 按下,写入 STM32 FLASH
{
    LCD_Fill(0,150,239,319,WHITE);//清除半屏
    LCD_ShowString(60,150,200,16,16,"Start Write FLASH....");
    STMFLASH_Write(FLASH_SAVE_ADDR,(u16*)TEXT_Buffer,SIZE);
    LCD_ShowString(60,150,200,16,16,"FLASH Write Finished");//提示传送完成
}
if(key==KEY_DOWN)//KEY1 按下,读取字符串并显示
{
    LCD_ShowString(60,150,200,16,16,"Start Read FLASH.... ");
    STMFLASH_Read(FLASH_SAVE_ADDR,(u16*)datatemp,SIZE);
    LCD_ShowString(60,150,200,16,16,"The Data Readed Is: ");
    LCD_ShowString(60,170,200,16,16,datatemp);//显示读到的字符串
}
i++;
delay_ms(10);
if(i==20)
{
    LED0=!LED0;//提示系统正在运行
    i=0;
}
}
```

主函数部分代码非常简单，首先进行按键扫描，然后分别进行按键的写操作和读操作。至此，我们的软件设计部分就结束了。

39.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，通过先按 WK_UP 按键写入数据，然后按 KEY1 读取数据，得到如图 39.4.1 所示：



图 39.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。本章的测试，我们还可以借助 USMART，在 USMART 里面添加 STMFLASH_ReadHalfWord 函数，既可以读取任意地址的数据。当然，你也可以将 STMFLASH_Write 稍微改造下，这样就可以在 USMART 里面验证 STM32 FLASH 的读写了。

第四十章 FM 收发实验

ALIENTEK 战舰 STM32 开发板上板载了一颗 FM 收发芯片：RDA5820。该芯片不但可以用来做 FM 接收，实现收音机功能，还可以用来做 FM 发射，实现电台的功能。

本章，我们将使用 STM32 驱动 RDA5820，实现 FM 接收和 FM 发射两个功能。本章分为如下几个部分：

- 40.1 RDA5820 简介
- 40.2 硬件设计
- 40.3 软件设计
- 40.4 下载验证



40.1 RDA5820 简介

RDA5820 是北京锐迪科推出的一款集成度非常高的立体声 FM 收发芯片。该芯片具有以下特点：

- FM 发射和接收一体
- 支持 65Mhz~115Mhz 的全球 FM 接收频段，收发天线共用。
- 支持 IIC/SPI 接口
- 支持 32.768K 晶振
- 数字音量及自动 AGC 控制
- 支持立体声/单声道切换，带软件静音功能
- 支持 I2S 接口（输入/输出）
- 内置 LDO，使用电压范围宽（2.7~5.5V）
- 高功率 32 欧负载音频输出、可直接驱动耳机
- 集成度高、功耗低、尺寸小（4mm*4mm QFN 封装）、应用简单

RDA5820 应用范围很宽，在很多手机、MP3、MP4 甚至平板电脑上都有应用。RDA5820 的引脚图如图 40.1.1 所示：

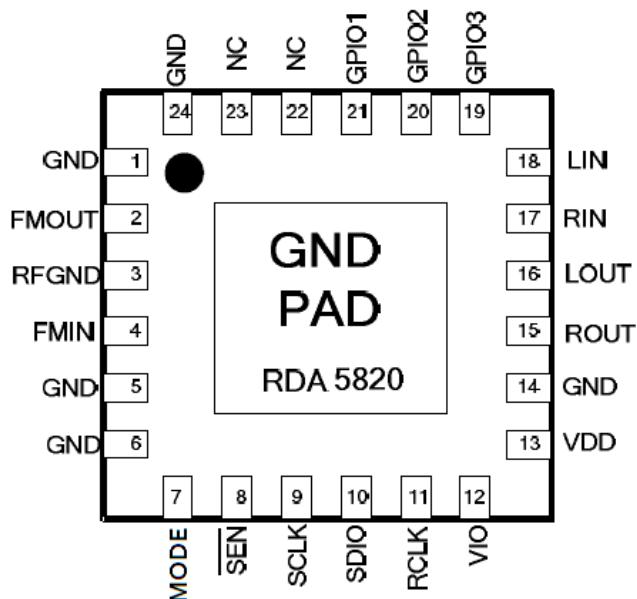


图 40.1.1 RDA5820 引脚图

RDA5820 支持 2 种通信模式，SPI 和 IIC，在战舰 STM32 开发板上面，使用的是 RDA5820 的 IIC 模式。通过将图 40.1.1 的 MODE 脚接 GND，RDA5820 即进入 IIC 模式，此时 SCLK 充当 IIC 的 SCL，SDIO 充当 IIC 的 SDA。RDA5820 的 IIC 地址为 0X11（不包含最低位），对应读为 0X23，写为 0X20。

模式设置

RDA5820 的模式设置通过 40H（寄存器地址 0X40）寄存器的 CHIP_FUNC[3:0]位来设置，RDA5820 可以工作在 RX 模式、TX 模式、PA 模式和 DAC 模式等，本章我们只介绍 RX 模式和 TX 模式。

通过设置 CHIP_FUNC[3:0]=0 即可定义当前工作模式为 FM 接收模式。在该模式下，我们即可实现 FM 收音机功能。

通过设置 CHIP_FUNC[3:0]=1 即可定义当前工作模式为 FM 发送模式。在该模式下，我们



即可实现 FM 电台的功能。

频点设置

软件可以通过配置 03H (寄存器地址 0X30) 寄存器来选择 FM 频道。搜台 (Seek) 的步进长度 (100KHz、200KHz 或 50KHz) 由 SPACE[1:0] 来选择，频道由 CHAN[9:0] 来选择，频率范围 (76MHz~91MHz、87MHz~108MHz 或 76MHz~108MHz 或 用户自定义 65MHz~115MHz 范围内频段) 由 BAND[1:0] 来选择。自定义的频段由寄存器 53H (chan_bottom) 和 54H (chan_top) 来设置，单位为 100KHz，即定义 65MHz~76MHz，可设置 BAND[1:0]=3 (用户自定义频段)，并且设置 chan_bottom=0x028A, chan_top=0x02f8。

频点计算方法如下 (该公式也适用于 FM 频点的读取):

$$\text{FMfreq} = \text{SPACE} * \text{CHAN} + \text{FMBTM}$$

其中 FMfreq 即我们需要的 FM 频率 (Mhz), SPACE 为我们设置的步进长度 (Khz), CHAN 是我们设置的频点值, FMBTM 则是我们在 BAND 里面所选频段的最低频率, 当 BAND=0 的实惠, FMBTM=87Mhz ; BAND=1 的时候, FMBTM=76Mhz ; BAND=2 的时候, FMBTM=CHAN_BOTTOM*0.1Mhz。

例如, 我们要设置 FM 频率为 93.0Mhz, 假设 BAND=0, SPACE=100Khz。那么我们只需要设置 CHAN=60 即可。

频点设置部分, FM 接收和 FM 发送是共用的, 对两者都适用。

关于 RDA5820, 我们就介绍到这, 详细的使用说明, 请大家参考《RDA5820 编程指南》和 RDA5820 的数据手册。

40.2 硬件设计

本章实验功能简介: 开机默认设置 FM 为接收模式, 设置接收频率为 93.6Mhz, 我们插上耳机就能听到该频率的电台。通过 KEY0 和 KEY2, 我们可以调节 FM 频率 (发送/接收), 在 FM 接收模式下, KEY1 用于自动搜台, 在 FM 发送模式下, KEY1 无效。通过 WK_UP 按键切换 RDA5820 的工作模式 (FM 发送/FM 接收)。在 FM 发送模式下, 我们可以通过在多功能端口 (P3) 的 AIN 排针上面输入音频信号, 就可以在收音机里面听到该音频。同时用 DS0 提示程序正在运行。

所要用到的硬件资源如下:

- 1) 指示灯 DS0
- 2) KEY0、KEY1、KEY2 和 WK_UP 等四个按键
- 3) TFTLCD 模块
- 4) 串口 (USMART 使用)
- 5) RDA5820
- 6) 74HC4052
- 7) TDA1308

前面 4 个我们就不介绍了, RDA5820 与 STM32 的连接电路如图 40.2.1 所示:

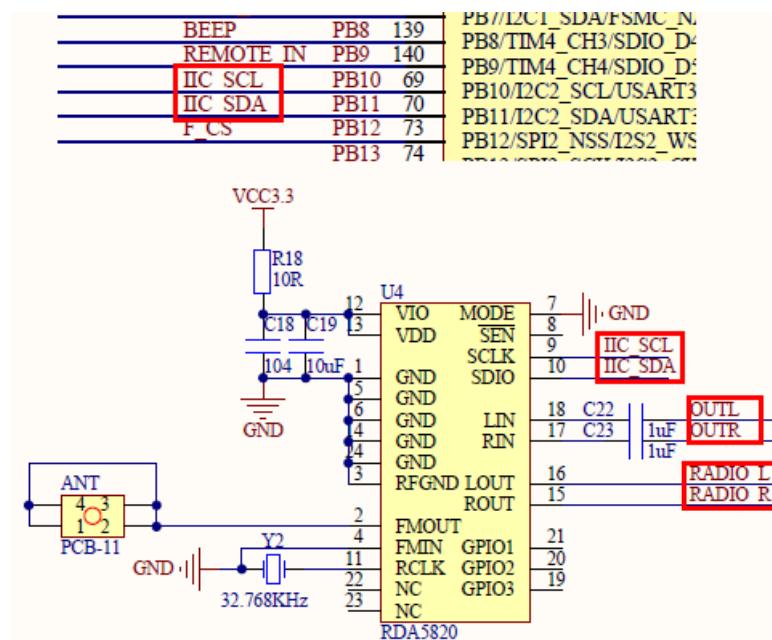
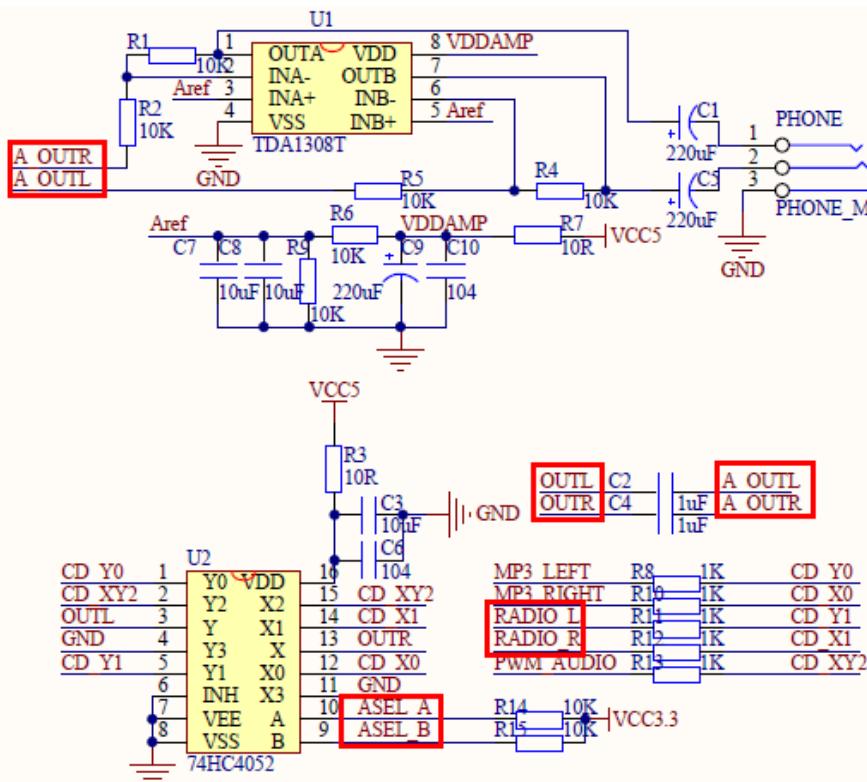


图 40.2.1 RDA5820 与 STM32 的连接图

这里 RDA5820 和之前介绍的 24C02 共用 IIC 总线，他们都是接在 STM32 的 PB10 和 PB11 两个脚上，上图中 OUTL 和 OUTR 接在 RDA5820 的 LIN 和 RIN，OUTR 和 OUTL 是来自音频选择器（74HC4052）的输出端，作为 FM 发送时的音源输入。另外 RADIO_L 和 RADIO_R 则是 FM 收音的音频输出，他们接在音频选择器的一对输入上面。

音频选择器（74HC4052）和耳机驱动（TDA1308）的连接电路如图 40.2.2 所示：





可以实现 4 组立体声音源的切换，战舰 STM32 开发板有 3 路音源输出：FM 收音机、MP3 (VS1053) 输出、PWM DAC 输出，我们通过 74HC4052 实现这 3 路音源的切换。图中 ASEL_A 和 ASEL_B 是其控制信号，分别连接在 STM32 的 PD7 和 PB7 上，用于控制音源切换。OUTL 和 OUTR 是 74HC4052 的输出端，被分为两路，一路接到 FM 发射的音频输入端，另外一路接到耳机驱动的输入端 (A_OUTR 和 A_OUTL)。

TDA1308 是一款性能十分优异 (秒杀 TDA2822、TDA7050 等) 的 AB 类数字音频专用耳放 IC，战舰 STM32 搭载这颗耳机驱动，其 MP3 播放的音质可以打败市面上很多中低端 MP3 播放器。A_OUTR 和 A_OUTL 是来自 74HC4052 的输出信号，作为 TDA1308 的输入端，经过 TDA1308 驱动后，输出到耳机插座。

硬件上，我们不需要做其他变动，只需找个耳机插到开发板的耳机插口，将开发板板载的天线拉出来，然后下载本章的实验例程，就可以听广播了。

40.3 软件设计

打开我们的 FM 收发实验工程，可以看到我们工程中新添加了 rda5820.c 文件和头文件 rda5820.h 以及 audiosel.c 文件和头文件 audiosel.h。

打 rda5820.c，代码如下：

```
#include "rda5820.h"
#include "delay.h"
//初始化
//0,初始化成功;
//其他,初始化失败.
u8 RDA5820_Init(void)
{
    u16 id;
    IIC_Init(); //初始化 IIC 口
    id=RDA5820_RD_Reg(RDA5820_R00); //读取 ID =0X5805
    if(id==0X5805) //读取 ID 正确
    {
        RDA5820_WR_Reg(RDA5820_R02,0x0002); //软复位
        delay_ms(50);
        RDA5820_WR_Reg(RDA5820_R02,0xC001); //立体声,上电
        delay_ms(600); //等待时钟稳定
        RDA5820_WR_Reg(RDA5820_R05,0X884F); //搜索强度 8,LNAN,1.8mA,VOL 最大
        RDA5820_WR_Reg(0X07,0X7800);
        RDA5820_WR_Reg(0X13,0X0008);
        RDA5820_WR_Reg(0X15,0x1420); //VCO 设置 0x17A0/0x1420
        RDA5820_WR_Reg(0X16,0XC000);
        RDA5820_WR_Reg(0X1C,0X3126);
        RDA5820_WR_Reg(0X22,0X9C24); //fm_true
        RDA5820_WR_Reg(0X47,0XF660); //tx rds
    }else return 1;//初始化失败
```



```
    return 0;
}

//写 RDA5820 寄存器
void RDA5820_WR_Reg(u8 addr,u16 val)
{
    IIC_Start();
    IIC_Send_Byte(RDA5820_WRITE); //发送写命令
    IIC_Wait_Ack();
    IIC_Send_Byte(addr);          //发送地址
    IIC_Wait_Ack();
    IIC_Send_Byte(val>>8);      //发送高字节
    IIC_Wait_Ack();
    IIC_Send_Byte(val&0XFF);     //发送低字节
    IIC_Wait_Ack();
    IIC_Stop();                  //产生一个停止条件
}

//读 RDA5820 寄存器
u16 RDA5820_RD_Reg(u8 addr)
{
    u16 res;
    IIC_Start();
    IIC_Send_Byte(RDA5820_WRITE); //发送写命令
    IIC_Wait_Ack();
    IIC_Send_Byte(addr);          //发送地址
    IIC_Wait_Ack();
    IIC_Start();
    IIC_Send_Byte(RDA5820_READ); //发送读命令
    IIC_Wait_Ack();
    res=IIC_Read_Byte(1);         //读高字节,发送 ACK
    res<<=8;
    res|=IIC_Read_Byte(0);        //读低字节,发送 NACK

    IIC_Stop();                  //产生一个停止条件
    return res;                  //返回读到的数据
}

//设置 RDA5820 为 RX 模式
void RDA5820_RX_Mode(void)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X40);   //读取 0X40 的内容
    temp&=0xffff0;               //RX 模式
    RDA5820_WR_Reg(0X40,temp);  //FM RX 模式
}
```



```
//设置 RDA5820 为 TX 模式
void RDA5820_TX_Mode(void)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X40);      //读取 0X40 的内容
    temp&=0xffff0;
    temp|=0x0001;                  //TX 模式
    RDA5820_WR_Reg(0X40,temp);    //FM TM 模式
}

//得到信号强度
//返回值范围:0~127
u8 RDA5820_Rssi_Get(void)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X0B);      //读取 0X0B 的内容
    return temp>>9;                //返回信号强度
}

//设置音量 ok
//vol:0~15;
void RDA5820_Vol_Set(u8 vol)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X05);      //读取 0X05 的内容
    temp&=0XFFF0;
    temp|=vol&0X0F;
    RDA5820_WR_Reg(0X05,temp);    //设置音量
}

//静音设置
//mute:0,不静音;1,静音
void RDA5820_Mute_Set(u8 mute)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X02);      //读取 0X02 的内容
    if(mute)temp|=1<<14;
    else temp&=~(1<<14);
    RDA5820_WR_Reg(0X02,temp);    //设置 MUTE
}

//设置灵敏度
//rss:0~127;
void RDA5820_Rssi_Set(u8 rssi)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X05);      //读取 0X05 的内容
```



```
temp&=0X80FF;
temp|=(u16)rssi<<8;
RDA5820_WR_Reg(0X05,temp); //设置 RSSI
}

//设置 TX 发送功率
//gain:0~63
void RDA5820_TxPAG_Set(u8 gain)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X42); //读取 0X42 的内容
    temp&=0XFFC0;
    temp|=gain; //GAIN
    RDA5820_WR_Reg(0X42,temp); //设置 PA 的功率
}

//设置 TX 输入信号增益
//gain:0~7
void RDA5820_TxPGA_Set(u8 gain)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X42); //读取 0X42 的内容
    temp&=0XF8FF;
    temp|=gain<<8; //GAIN
    RDA5820_WR_Reg(0X42,temp); //设置 PGA
}

//设置 RDA5820 的工作频段
//band:0,87~108Mhz;1,76~91Mhz;2,76~108Mhz;3,用户自定义(53H~54H)
void RDA5820_Band_Set(u8 band)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X03); //读取 0X03 的内容
    temp&=0XFFF3;
    temp|=band<<2;
    RDA5820_WR_Reg(0X03,temp); //设置 BAND
}

//设置 RDA5820 的步进频率
//band:0,100Khz;1,200Khz;3,50Khz;3,保留
void RDA5820_Space_Set(u8 spc)
{
    u16 temp;
    temp=RDA5820_RD_Reg(0X03); //读取 0X03 的内容
    temp&=0XFFFC;
    temp|=spc;
    RDA5820_WR_Reg(0X03,temp); //设置 BAND
```



```
}

//设置 RDA5820 的频率
//freq:频率值(单位为 10Khz),比如 10805,表示 108.05Mhz
void RDA5820_Freq_Set(u16 freq)
{
    u16 temp;
    u8 spc=0,band=0;
    u16 fbtm,chan;
    temp=RDA5820_RD_Reg(0X03); //读取 0X03 的内容
    temp&=0X001F;
    band=(temp>>2)&0x03;      //得到频带
    spc=temp&0x03;            //得到分辨率
    if(spc==0)spc=10;
    else if(spc==1)spc=20;
    else spc=5;
    if(band==0)fbtm=8700;
    else if(band==1||band==2)fbtm=7600;
    else
    {
        fbtm=RDA5820_RD_Reg(0X53); //得到 bottom 频率
        fbtm*=10;
    }
    if(freq<fbtm)return;
    chan=(freq-fbtm)/spc;        //得到 CHAN 应该写入的值
    chan&=0X3FF;                //取低 10 位
    temp|=chan<<6;
    temp|=1<<4;                //TONE ENABLE
    RDA5820_WR_Reg(0X03,temp); //设置频率
    delay_ms(20);               //等待 20ms
    while((RDA5820_RD_Reg(0X0B)&(1<<7))==0); //等待 FM_READY

}

//得到当前频率
//返回值:频率值(单位 10Khz)
u16 RDA5820_Freq_Get(void)
{
    u16 temp;
    u8 spc=0,band=0;
    u16 fbtm,chan;
    temp=RDA5820_RD_Reg(0X03); //读取 0X03 的内容
    chan=temp>>6;
    band=(temp>>2)&0x03;      //得到频带
    spc=temp&0x03;            //得到分辨率
```



```
if(spc==0)spc=10;
else if(spc==1)spc=20;
else spc=5;
if(band==0)fbtm=8700;
else if(band==1||band==2)fbtm=7600;
else
{
    fbtm=RDA5820_RD_Reg(0X53);//得到 bottom 频率
    fbtm*=10;
}
temp=fbtm+chan*spc;
return temp;//返回频率值
}
```

本部分代码我们就不细说了，都是一些寄存器配置，比较简单。头文件 rda5820.h 的代码也比较简单，这里不详细说明了。接下来我们看看 audiosel.c 文件代码如下：

```
#include "audiosel.h"
//声音初始化
void Audiosel_Init(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|
                           RCC_APB2Periph_GPIOD, ENABLE);      //使能 PB PD 端口时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;           // PB.6 推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;    //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;           //PD.7 推挽输出
    GPIO_Init(GPIOD, &GPIO_InitStructure);

}

//设置 4052 的选择通道
//声音通道选择
//0 //MP3 通道
//1 //收音机通道
//2 //PWM 音频通道
//3 //无声
void Audiosel_Set(u8 ch)
{
    AUDIO_SELA=ch&0X01;
    AUDIO_SELB=(ch>>1)&0X01;
```



}

此部分代码很简单，用来控制 74HC4052。同样，头文件 audiosel.h 我们这里也不列出来，大家可以打开看看。

最后，打开 main.c 文件，代码如下：

```
void RDA5820_Show_Msg(void)
{
    u8 rssi;
    u16 freq;
    freq=RDA5820_Freq_Get();           //读取设置到的频率值
    LCD_ShowNum(100,210,freq/100,3,16); //显示频率整数部分
    LCD_ShowNum(132,210,(freq%100)/10,1,16); //显示频率小数部分
    rssi=RDA5820_Rssi_Get();           //得到信号强度
    LCD_ShowNum(100,230,rssi,2,16);     //显示信号强度
}

int main(void)
{
    u8 key,rssi;
    u16 freqset=8700;//默认为 87Mhz
    u8 i=0;
    u8 mode=0; //接收模式
    delay_init();          //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    KEY_Init();            //按键初始化
    LED_Init();             //LED 端口初始化
    LCD_Init();             //LCD 初始化
    Audiosel_Init();
    Audiosel_Set(AUDIO_RADIO);
    RDA5820_Init();
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"RDA5820 TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/14");
    LCD_ShowString(60,130,200,16,16,"KEY0:Freq+ KEY2:Freq-");
    LCD_ShowString(60,150,200,16,16,"KEY1:Auto Search(RX)");
    LCD_ShowString(60,170,200,16,16,"KEY_UP:Mode Set");
    POINT_COLOR=BLUE;
    //显示提示信息
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(60,190,200,16,16,"Mode:FM RX");
    LCD_ShowString(60,210,200,16,16,"Freq: 93.6Mhz");
```



```
LCD_ShowString(60,230,200,16,16,"Rssi:");

RDA5820_Band_Set(0);      //设置频段为 87~108Mhz
RDA5820_Space_Set(0);    //设置步进为 100Khz
RDA5820_TxPGA_Set(3);   //信号增益设置为 3
RDA5820_TxPAG_Set(63); //发射功率为最大.
RDA5820_RX_Mode();      //设置为接收模式
freqset=9360;            //默认为 93.6Mhz
RDA5820_Freq_Set(freqset);//设置频率
while(1)
{
    key=KEY_Scan(0);//不支持连接
    switch(key)
    {
        case 0://无任何按键按下
        break;
        case KEY_UP://切换模式
        mode=!mode;
        if(mode)
        {
            Audiosel_Set(AUDIO_PWM); //设置到 PWM 音频通道
            RDA5820_TX_Mode();     //发送模式
            RDA5820_Freq_Set(freqset); //设置频率
            LCD_ShowString(100,190,200,16,16,"FM TX");
        }else
        {
            Audiosel_Set(AUDIO_RADIO); //设置到收音机声道
            RDA5820_RX_Mode();       //接收模式
            RDA5820_Freq_Set(freqset); //设置频率
            LCD_ShowString(100,190,200,16,16,"FM RX");
        }
        break;
    case KEY_DOWN://自动搜索下一个电台.
        if(mode==0)//仅在接收模式有效
        {
            while(1)
            {
                if(freqset<10800)freqset+=10; //频率增加 100Khz
                else freqset=8700;          //回到起点
                RDA5820_Freq_Set(freqset);  //设置频率
                delay_ms(10);              //等待调频信号稳定
                if(RDA5820_RD_Reg(0X0B)&(1<<8))//是一个有效电台.
                {

```



```
RDA5820_Show_Msg();           //显示信息
break;
}
RDA5820_Show_Msg();           //显示信息
//在搜台期间有按键按下,则跳出搜台.
key=KEY_Scan(0); //不支持连接
if(key)break;
}
}
break;
case KEY_LEFT://频率减
if(freqset>8700)freqset-=10;   //频率减少 100Khz
else freqset=10800;           //越界处理
RDA5820_Freq_Set(freqset);   //设置频率
RDA5820_Show_Msg(); //显示信息
break;
case KEY_RIGHT://频率增
if(freqset<10800)freqset+=10;   //频率增加 100Khz
else freqset=8700;           //越界处理
RDA5820_Freq_Set(freqset);   //设置频率
RDA5820_Show_Msg(); //显示信息
break;
}
i++;
delay_ms(10);
if(i==200)//每两秒左右检测一次信号强度等信息.
{
    i=0;
    rssi=RDA5820_Rssi_Get();           //得到信号强度
    LCD_ShowNum(100,230,rssi,2,16);   //显示信号强度
}
if((i%20)==0)LED0=!LED0;//DS0 闪烁, 提示程序运行
}
}
```

此部分代码除了 main 函数,还有一个 RDA5820_Show_Msg 函数,该函数用于显示当前 FM 频率和信号强度等信息。main 函数开始设置默认频率,然后根据按键来设置 FM 的收发模式以及频率。本章,我面可以利用 USMART 来设置 RDA5820 的各项参数,方便大家快速掌握。在 usmart_nametab 里面,我面加入如下五个函数:

```
(void*)RDA5820_Rssi_Set,"void RDA5820_Rssi_Set(u8 rssi)",
(void*)RDA5820_Band_Set,"void RDA5820_Band_Set(u8 band)",
(void*)RDA5820_Freq_Set,"void RDA5820_Freq_Set(u16 freq)",
(void*)RDA5820_Vol_Set,"void RDA5820_Vol_Set(u8 vol)",
(void*)RDA5820_TxPGA_Set,"void RDA5820_TxPGA_Set(u8 gain)",
```



这样，我面就可以在串口调用这些函数，从而修改 RDA5820 的配置。至此，我们的软件设计部分就结束了。

40.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 40.4.1 所示界面：



图 40.4.1 程序运行效果图

此时，我们就可以在耳机上面听到广播了（注意，将开发板自带的天线拉出来，可以提高接收能力），通过 KEY0 和 KEY1，调节频率，也可以通过按 KEY1 自动搜索下一个电台（PS：如果收不到台，说明你住的地方信号不好，跑到窗户边或者室外，一般就可以收到电台了）。

通过 WK_UP 按键，可以切换工作模式，例如：切换到 FM TX 模式，就可以通过单独的收音机（或者另外一块战舰 STM32 开发板）接收到开发板发出的 FM 信号了，此时在多功能接口 P3 的 AIN 端输入音频信号，就可以在收音机端接收这个音频信号了。

本章还可以通过 USMART 设置 RDA5820 的相关参数，感兴趣的朋友可以动手试试。

第四十一章 摄像头实验

ALIENTEK 战舰 STM32 开发板板载了一个摄像头接口（P8），该接口可以用来连接 ALIENTEK OV7670 摄像头模块。本章，我们将使用 STM32 驱动 ALIENTEK OV7670 摄像头模块，实现摄像头功能。本章分为如下几个部分：

- 41.1 OV7670 简介
- 41.2 硬件设计
- 41.3 软件设计
- 41.4 下载验证



41.1 OV7670 简介

OV7670 是 OV (OmniVision) 公司生产的一颗 1/6 寸的 CMOS VGA 图像传感器。该传感器体积小、工作电压低，提供单片 VGA 摄像头和影像处理器的所有功能。通过 SCCB 总线控制，可以输出整帧、子采样、取窗口等方式的各种分辨率 8 位影像数据。该产品 VGA 图像最高达到 30 帧/秒。用户可以完全控制图像质量、数据格式和传输方式。所有图像处理功能过程包括伽玛曲线、白平衡、度、色度等都可以通过 SCCB 接口编程。OmniVision 图像传感器应用独有的传感器技术，通过减少或消除光学或电子缺陷如固定图案噪声、托尾、浮散等，提高图像质量，得到清晰的稳定的彩色图像。

OV7670 的特点有：

- 高灵敏度、低电压适合嵌入式应用
- 标准的 SCCB 接口，兼容 IIC 接口
- 支持 RawRGB、RGB(GBR4:2:2, RGB565/RGB555/RGB444), YUV(4:2:2) 和 YCbCr (4:2:2) 输出格式
- 支持 VGA、CIF，和从 CIF 到 40*30 的各种尺寸输出
- 支持自动曝光控制、自动增益控制、自动白平衡、自动消除灯光条纹、自动黑电平校准等自动控制功能。同时支持色饱和度、色相、伽马、锐度等设置。
- 支持闪光灯
- 支持图像缩放

OV7670 的功能框图如图 41.1.1 所示：

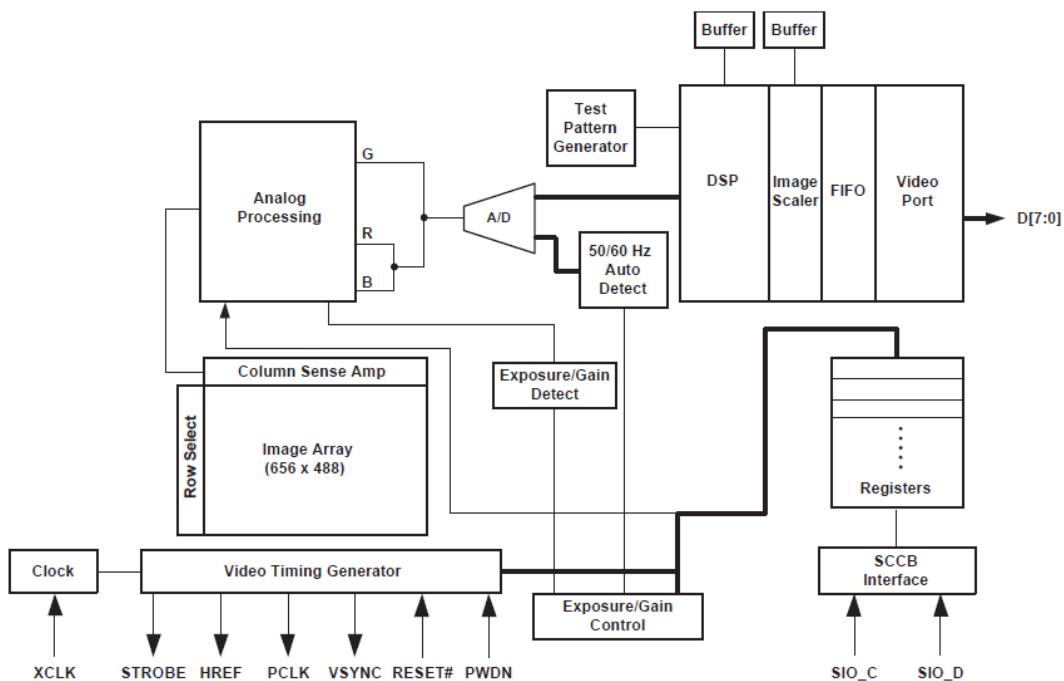


图 41.1.1 OV7670 功能框图

OV7670 传感器包括如下一些功能模块。

1. 感光整列 (Image Array)

OV7670 总共有 656*488 个像素，其中 640*480 个有效（即有效像素为 30W）。

2. 时序发生器 (Video Timing Generator)

时序发生器具有的功能包括：整列控制和帧率发生（7 种不同格式输出）、内部信号发生器



和分布、帧率时序、自动曝光控制、输出外部时序（VSYNC、HREF/HSYNC 和 PCLK）。

3. 模拟信号处理（Analog Processing）

模拟信号处理所有模拟功能，并包括：自动增益（AGC）和自动白平衡（AWB）。

4.A/D 转换（A/D）

原始的信号经过模拟处理器模块之后，分 G 和 BR 两路进入一个 10 位的 A/D 转换器，A/D 转换器工作在 12M 频率，与像素频率完全同步（转换的频率和帧率有关）。

除 A/D 转换器外，该模块还有以下三个功能：

- 黑电平校正（BLC）
- U/V 通道延迟
- A/D 范围控制

A/D 范围乘积和 A/D 的范围控制共同设置 A/D 的范围和最大值，允许用户根据应用调整图片的亮度。

5. 测试图案发生器（Test Pattern Generator）

测试图案发生器功能包括：八色彩色条图案、渐变至黑白彩色条图案和输出脚移位“1”。

6. 数字处理器（DSP）

这个部分控制由原始信号插值到 RGB 信号的过程，并控制一些图像质量：

- 边缘锐化（二维高通滤波器）
- 颜色空间转换（原始信号到 RGB 或者 YUV/YCbYCr）
- RGB 色彩矩阵以消除串扰
- 色相和饱和度的控制
- 黑/白点补偿
- 降噪
- 镜头补偿
- 可编程的伽玛
- 十位到八位数据转换

7. 缩放功能（Image Scaler）

这个模块按照预先设置的要求输出数据格式，能将 YUV/RGB 信号从 VGA 缩小到 CIF 以下的任何尺寸。

8. 数字视频接口（Digital Video Port）

通过寄存器 COM2[1:0]，调节 IOL/IOH 的驱动电流，以适应用户的负载。

9. SCCB 接口（SCCB Interface）

SCCB 接口控制图像传感器芯片的运行，详细使用方法参照光盘的《OmniVision Technologies Serial Camera Control Bus(SCCB) Specification》这个文档

10. LED 和闪光灯的输出控制（LED and Strobe Flash Control Output）

OV7670 有闪光灯模式，可以控制外接闪光灯或闪光 LED 的工作。

OV7670 的寄存器通过 SCCB 时序访问并设置，SCCB 时序和 IIC 时序十分类似，在本章我们不做介绍，请大家参考光盘的相关文档。

接下来我们介绍一下 OV7670 的图像数据输出格式。首先我们简单介绍几个定义：

VGA，即分辨率为 640*480 的输出模式；

QVGA，即分辨率为 320*240 的输出格式，也就是本章我们需要用到的格式；

QQVGA，即分辨率为 160*120 的输出格式；

PCLK，即像素时钟，一个 PCLK 时钟，输出一个像素(或半个像素)。

VSYNC，即帧同步信号。



HREF /HSYNC，即行同步信号。

OV7670 的图像数据输出（通过 D[7:0]）就是在 PCLK，VSYNC 和 HREF/ HSYNC 的控制下进行的。首先看看行输出时序，如图 41.1.2 所示：

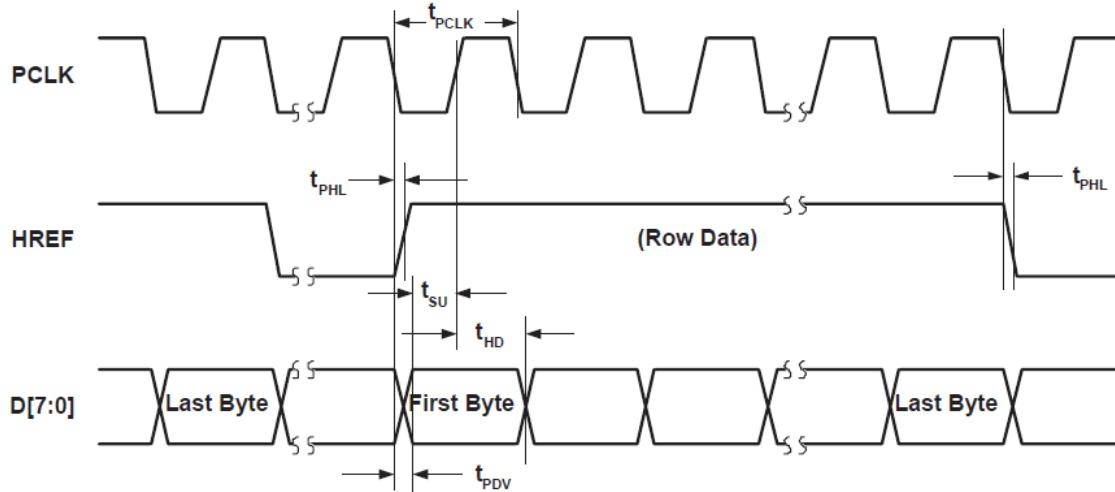


图 41.1.2 OV7670 行输出时序

从上图可以看出，图像数据在 HREF 为高的时候输出，当 HREF 变高后，每一个 PCLK 时钟，输出一个字节数据。比如我们采用 VGA 时序，RGB565 格式输出，每 2 个字节组成一个像素的颜色（高字节在前，低字节在后），这样每行输出总共有 640×2 个 PCLK 周期，输出 640×2 个字节。

再来看看帧时序（VGA 模式），如图 41.1.3 所示：

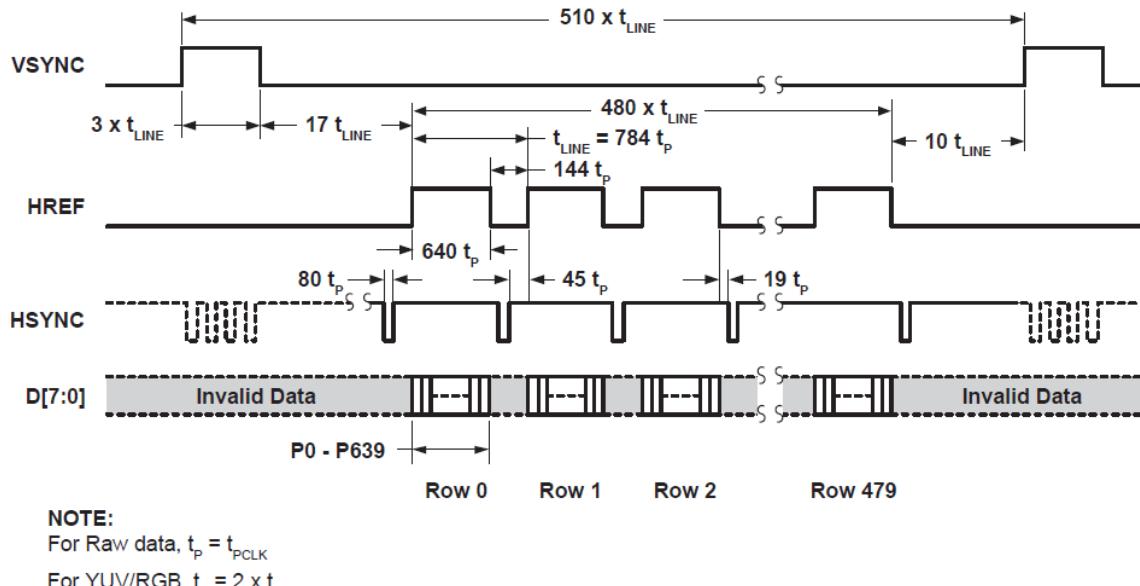


图 41.1.3 OV7670 帧时序

上图清楚的表示了 OV7670 在 VGA 模式下的数据输出，注意，图中的 HSYNC 和 HREF 其实是同一个引脚产生的信号，只是在不同场合下面，使用不同的信号方式，我们本章用到的是 HREF。

因为 OV7670 的像素时钟（PCLK）最高可达 24Mhz，我们用 STM32F103ZET6 的 IO 口直接抓取，是非常困难的，也十分占耗 CPU（可以通过降低 PCLK 输出频率，来实现 IO 口抓取，



但是不推荐)。所以,本章我们并不是采取直接抓取来自 OV7670 的数据,而是通过 FIFO 读取,ALIENTEK OV7670 摄像头模块自带了一个 FIFO 芯片,用于暂存图像数据,有了这个芯片,我们就可以很方便的获取图像数据了,而不再需要单片机具有高速 IO,也不会耗费多少 CPU,可以说,只要是个单片机,都可以通过 ALIENTEK OV7670 摄像头模块实现拍照的功能。

接下来我们介绍一下 ALIENTEK OV7670 摄像头模块。该模块的外观如图 41.1.4:

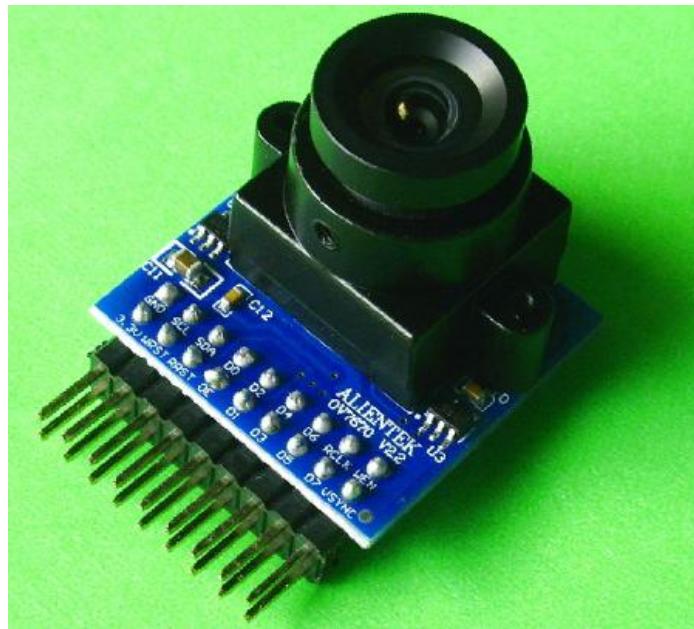


图 41.1.4 ALIENTEK OV7670 摄像头模块外观图

模块原理图如图 41.1.5 所示:

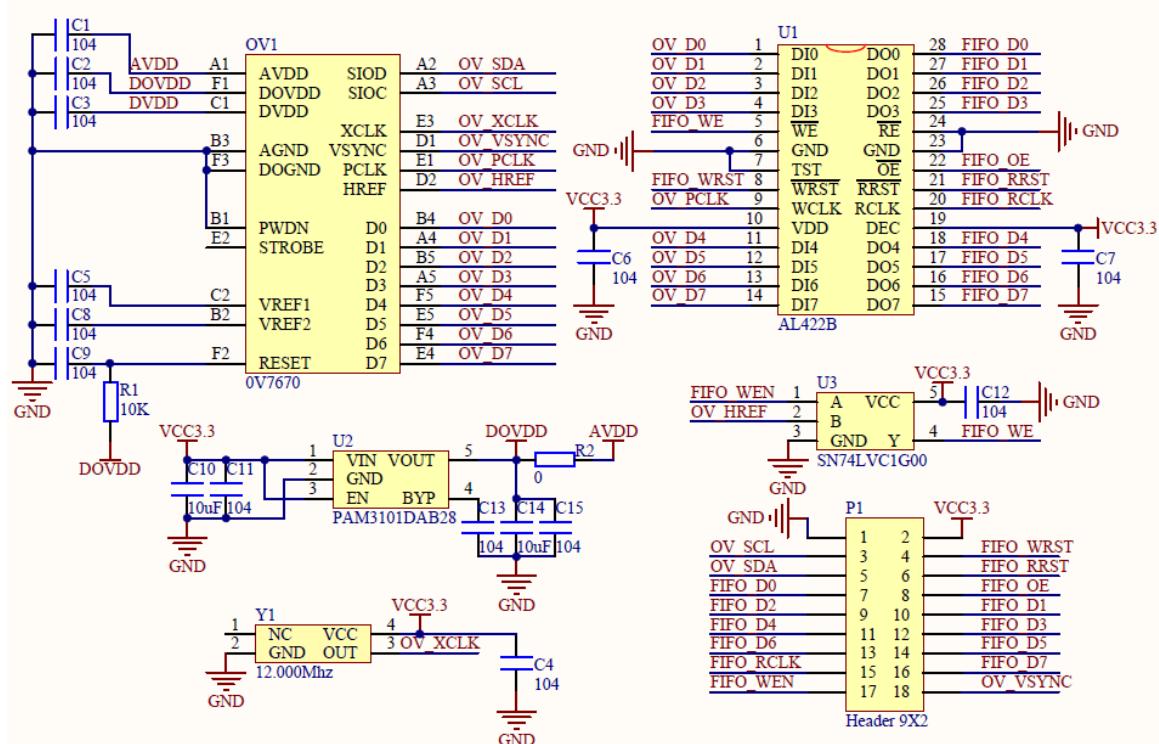


图 41.1.5 ALIENTEK OV7670 摄像头模块原理图



从上图可以看出，ALIENTEK OV7670 摄像头模块自带了有源晶振，用于产生 12M 时钟作为 OV7670 的 XCLK 输入。同时自带了稳压芯片，用于提供 OV7670 稳定的 2.8V 工作电压，并带有一个 FIFO 芯片（AL422B），该 FIFO 芯片的容量是 384K 字节，足够存储 2 帧 QVGA 的图像数据。模块通过一个 2*9 的双排排针（P1）与外部通信，与外部的通信信号如表 41.1.1 所示：

信号	作用描述	信号	作用描述
VCC3.3	模块供电脚，接 3.3V 电源	FIFO_WEN	FIFO 写使能
GND	模块地线	FIFO_WRST	FIFO 写指针复位
OV_SCL	SCCB 通信时钟信号	FIFO_RRST	FIFO 读指针复位
OV_SDA	SCCB 通信数据信号	FIFO_OE	FIFO 输出使能（片选）
FIFO_D[7:0]	FIFO 输出数据（8 位）	OV_VSYNC	OV7670 帧同步信号
FIFO_RCLK	读 FIFO 时钟		

表 41.1.1 OV7670 模块信号及其作用描述

下面我们来看看如何使用 ALIENTEK OV7670 摄像头模块（以 QVGA 模式，RGB565 格式为例）。对于该模块，我们只关心两点：1，如何存储图像数据；2，如何读取图像数据。

首先，我们来看如何存储图像数据。

ALIENTEK OV7670 摄像头模块存储图像数据的过程为：等待 OV7670 同步信号 → FIFO 写指针复位 → FIFO 写使能 → 等待第二个 OV7670 同步信号 → FIFO 写禁止。通过以上 5 个步骤，我们就完成了 1 帧图像数据的存储。

接下来，我们来看看如何读取图像数据。

在存储完一帧图像以后，我们就可以开始读取图像数据了。读取过程为：FIFO 读指针复位 → 给 FIFO 读时钟（FIFO_RCLK）→ 读取第一个像素高字节 → 给 FIFO 读时钟 → 读取第一个像素低字节 → 给 FIFO 读时钟 → 读取第二个像素高字节 → 循环读取剩余像素 → 结束。

可以看出，ALIENTEK OV7670 摄像头模块数据的读取也是十分简单，比如 QVGA 模式，RGB565 格式，我们总共循环读取 320*240*2 次，就可以读取 1 帧图像数据，把这些数据写入 LCD 模块，我们就可以看到摄像头捕捉到的画面了。

OV7670 还可以对输出图像进行各种设置，详见光盘《OV7670 中文数据手册 1.01》和《OV7670 software application note》这两个文档，对 AL422B 的操作时序，请大家参考 AL422B 的数据手册。

了解了 OV7670 模块的数据存储和读取，我们就可以开始设计代码了，本章，我们用一个外部中断，来捕捉帧同步信号（VSYNC），然后在中断里面启动 OV7670 模块的图像数据存储，等待下一次 VSHNC 信号到来，我们就关闭数据存储，然后一帧数据就存储完成了，在主函数里面就可以慢慢的将这一帧数据读出来，放到 LCD 即可显示了，同时开始第二帧数据的存储，如此循环，实现摄像头功能。

本章，我们将使用摄像头模块的 QVGA 输出（320*240），刚好和战舰 STM32 开发板使用的 LCD 模块分辨率一样，一帧输出就是一屏数据，提高速度的同时也不浪费资源。注意：ALIENTEK OV7670 摄像头模块自带的 FIFO 是没办法缓存一帧的 VGA 图像的，如果使用 VGA 输出，那么你必须在 FIFO 写满之前开始读 FIFO 数据，保证数据不被覆盖。

41.2 硬件设计

本章实验功能简介：开机后，初始化摄像头模块（OV7670），如果初始化成功，则在 LCD



模块上面显示摄像头模块所拍摄到的内容。我们可以通过 KEY0 设置光照模式（5 种模式）、通过 KEY1 设置色饱和度，通过 KEY2 设置亮度，通过 WK_UP 设置对比度，通过 TPAD 设置特效（总共 7 种特效）。通过串口，我们可以查看当前的帧率（这里是指 LCD 显示的帧率，而不是指 OV7670 的输出帧率），同时可以借助 USMART 设置 OV7670 的寄存器，方便大家调试。DS0 指示程序运行状态。

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 5 个按键（包括 TPAD 触摸按键）
- 3) 串口
- 4) TFTLCD 模块
- 5) 摄像头模块接口
- 6) 摄像头模块

ALIENTEK OV7670 摄像头模块在 41.1 节已经有详细介绍过，这里我们主要介绍该模块与 ALIETEK 战舰 STM32 开发板的连接。

在开发板的左下角的 2*9 的 P8 排座，是摄像头模块/OLED 模块共用接口，在第十七章，我们曾简单介绍过这个接口。本章，我们只需要将 ALIENTEK OV7670 摄像头模块插入这个接口即可，该接口与 STM32 的连接关系如图 41.2.1 所示：

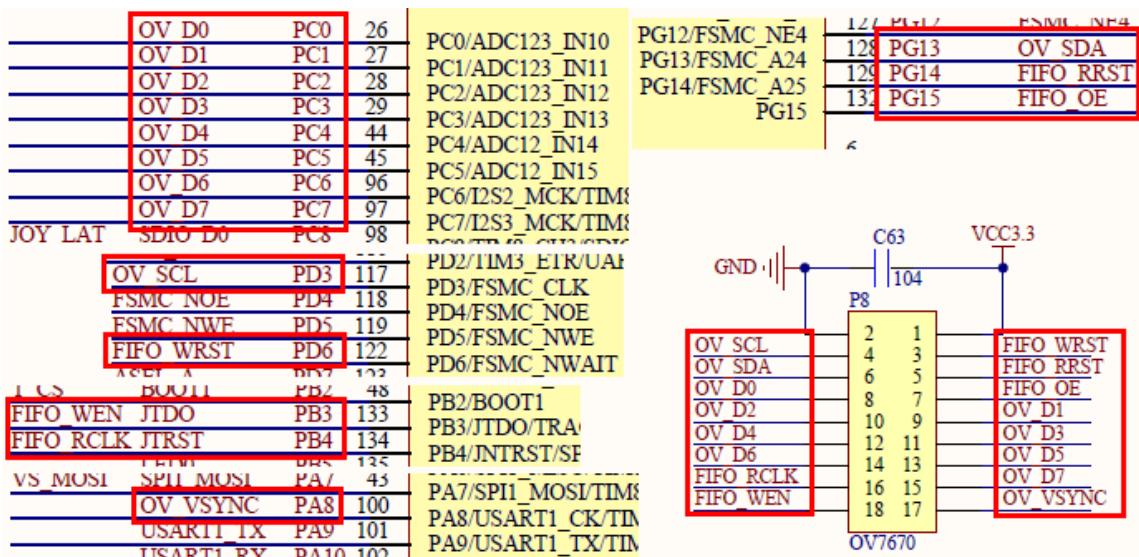


图 41.2.1 摄像头模块接口与 STM32 连接图

从上图可以看出，OV7670 摄像头模块的各信号脚与 STM32 的连接关系为：

- OV_SDA 接 PG13;
- OV_SCL 接 PD3;
- FIFO_RCLK 接 PB4;
- FIFO_WEN 接 PB3;
- FIFO_WRST 接 PD6;
- FIFO_RRST 接 PG14;
- FIFO_OE 接 PG15;
- OV_VSYNC 接 PA8;
- OV_D[7:0]接 PC[7:0];

这些线的连接，战舰 STM32 的内部已经连接好了，我们只需要将 OV7670 摄像头模块插上



去就好了。实物连接如图 41.2.2 所示：

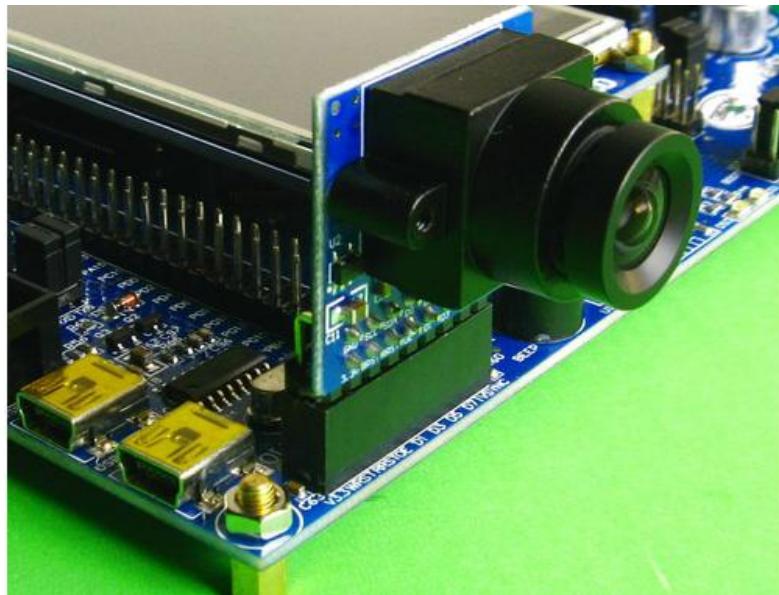


图 41.2.2 OV7670 摄像头模块与开发板连接实物图

41.3 软件设计

打开我们摄像头实验的工程，可以看到我们的工程中多了 ov7670.c 和 sccb.c 源文件，以及头文件 ov7670.h、sccb.h 和 ov7670cfg.h 等 5 个文件。

本章总共新增了 5 个文件，代码比较多，我们就不一一列出了，仅挑两个重要的地方进行讲解。首先，我们来看 ov7670.c 里面的 OV7670_Init 函数，该函数代码如下：

```
u8 OV7670_Init(void)
{
    u8 temp;
    u16 i=0;
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB|
    RCC_APB2Periph_GPIOC|RCC_APB2Periph_GPIOD|
    RCC_APB2Periph_GPIOG, ENABLE);      //使能相关端口时钟

    GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_8;      //PA8 输入 上拉
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);          //初始化 GPIOA.8

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4;    // 端口配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;        //推挽输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_SetBits(GPIOB,GPIO_Pin_3|GPIO_Pin_4);    //初始化 GPIO
```



```
GPIO_InitStructure.GPIO_Pin = 0xff; //PC0~7 输入 上拉
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
GPIO_Init(GPIOC, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOD, &GPIO_InitStructure);
GPIO_SetBits(GPIOD,GPIO_Pin_6);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14|GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOG, &GPIO_InitStructure);
GPIO_SetBits(GPIOG,GPIO_Pin_14|GPIO_Pin_15);

GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable,ENABLE); //SWD
SCCB_Init(); //初始化 SCCB 的 IO 口
if(SCCB_WR_Reg(0x12,0x80))return 1; //复位 SCCB
delay_ms(50);
//读取产品型号
temp=SCCB_RD_Reg(0x0b);
if(temp!=0x73)return 2;
temp=SCCB_RD_Reg(0x0a);
if(temp!=0x76)return 2;
//初始化序列
for(i=0;i<sizeof.ov7670_init_reg_tbl/sizeof.ov7670_init_reg_tbl[0])/2;i++)
{
    SCCB_WR_Reg.ov7670_init_reg_tbl[i][0],ov7670_init_reg_tbl[i][1]);
    delay_ms(2);
}
return 0x00; //ok
}
```

此部分代码先初始化 OV7670 相关的 IO 口(包括 SCCB_Init)，然后最主要的是完成 OV7670 的寄存器序列初始化。OV7670 的寄存器特多(百几十个)，配置特麻烦，幸好厂家有提供参考配置序列(详见《OV7670 software application note》)，本章我们用到的配置序列，存放在 ov7670_init_reg_tbl 这个数组里面，该数组是一个 2 维数组，存储初始化序列寄存器及其对应的值，该数组存放在 ov7670cfg.h 里面。

接下来，我们看看 ov7670cfg.h 里面 ov7670_init_reg_tbl 的内容，ov7670cfg.h 文件的代码如下：

```
//初始化寄存器序列及其对应的值
const u8 ov7670_init_reg_tbl[][2]=
{
    /*以下为 OV7670 QVGA RGB565 参数 */
    {0x3a, 0x04},//
```



```
{0x40, 0x10},  
{0x12, 0x14}//QVGA,RGB 输出  
.....省略部分设置  
{0x6e, 0x11}//100  
{0x6f, 0x9f}//0x9e for advance AWB  
{0x55, 0x00}//亮度  
{0x56, 0x40}//对比度  
{0x57, 0x80}//0x40, change according to Jim's request  
};
```

以上代码，我们省略了很多（全部贴出来太长了），我们大概了解下结构，每个条目的第一个字节为寄存器号（也就是寄存器地址），第二个字节为要设置的值，比如{0x3a, 0x04}，就表示在 0X03 地址，写入 0X04 这个值。

通过这么一长串（110 多个）寄存器的配置，我们就完成了 OV7670 的初始化，本章我们配置 OV7670 工作在 QVGA 模式，RGB565 格式输出。在完成初始化之后，我们既可以开始读取 OV7670 的数据了。

OV7670 文件夹里面的其他代码我们就不逐个介绍了，请大家参考光盘该例程源码。

因为本章我们还用到了帧率（LCD 显示的帧率）统计和中断处理，所以我们还需要修改 timer.c、timer.h、exti.c 及 exti.h 这几个文件。

在 timer.c 里面，我们新增 TIM6_Int_Init 和 TIM6_IRQHandler 两个函数，用于统计帧率，增加代码如下：

```
u8 ov_frame; //统计帧数  
//定时器 6 中断服务程序  
void TIM6_IRQHandler(void)  
{    if (TIM_GetITStatus(TIM6, TIM_IT_Update) != RESET) //更新中断发生  
    {  
        LED1=!LED1;  
        printf("frame:%dfps\r\n",ov_frame); //打印帧率  
        ov_frame=0;  
    }  
    TIM_ClearITPendingBit(TIM6, TIM_IT_Update ); //清中断标志位  
  
}  
//基本定时器 6 中断初始化  
//这里时钟选择为 APB1 的 2 倍，而 APB1 为 36M  
//arr：自动重装值。  
//psc：时钟预分频数  
//这里使用的是定时器 3！  
void TIM6_Int_Init(u16 arr,u16 psc)  
{  
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;  
    NVIC_InitTypeDef NVIC_InitStructure;  
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE); //时钟使能
```



```

TIM_TimeBaseStructure.TIM_Period = arr; //自动重装载周期值
TIM_TimeBaseStructure.TIM_Prescaler = psc; //预分频值
TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDTS = Tck_tim
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式
TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure); //根据指定的参数初始化 TIMx

TIM_ITConfig( TIM6,TIM_IT_Update|TIM_IT_Trigger,ENABLE); //使能更新触发中断
TIM_Cmd(TIM6, ENABLE); //使能 TIMx 外设
NVIC_InitStructure.NVIC_IRQChannel = TIM6_IRQn; //TIM3 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; //先占优先级 0 级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //从优先级 3 级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道被使能
NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化外设 NVIC 寄存器

}

```

这里，我们用到基本定时器 TIM6 来统计帧率，也就是 1 秒钟中断一次，打印 ov_frame 的值，ov_frame 用于统计 LCD 帧率。

再在 timer.h 里面添加 TIM6_Int_Init 函数的定义，就完成对 timer.c 和 timer.h 的修改了。

在 exti.c 里面添加 EXTI8_Init 和 EXTI9_5_IRQHandler 函数，用于 OV7670 模块的 FIFO 写控制，exti.c 文件新增部分代码如下：

```

//ov_sta:0,开始一帧数据采集
u8 ov_stas;
//外部中断 5~9 服务程序
void EXTI9_5_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line8)==SET)//是 8 线的中断
    {
        if(ov_stas<2)
        {
            if(ov_stas==0)
            {
                OV7670_WRST=0; //复位写指针
                OV7670_WRST=1;
                OV7670_WREN=1; //允许写入 FIFO
            }
            else
            {
                OV7670_WREN=0; //禁止写入 FIFO
                OV7670_WRST=0; //复位写指针
                OV7670_WRST=1;
            }
            ov_stas++;
        }
    }
    EXTI_ClearITPendingBit(EXTI_Line8); //清除 EXTI8 线路挂起位
}

```



```

}

//外部中断 8 初始化
void EXTI8_Init(void)
{
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA,GPIO_PinSource8);//PA8 对中断线 8

    EXTI_InitStructure.EXTI_Line=EXTI_Line8;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure); //根据指定的参数初始化外设 EXTI 寄存器

    NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQHandler; //使能外部中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //抢占优先级 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; //子优先级 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //使能外部中断通道
    NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化外设 NVIC 寄存器
}

```

因为 OV7670 的帧同步信号 (OV_VSYNC) 接在 PA8 上面, 所以我们这里配置 PA8 作为中端输入, 因为 STM32 的外部中断 5~9 共用一个中端服务函数 (EXTI9_5_IRQHandler), 所以在该函数里面, 我们需要先判断中断是不是来自中断线 8 的, 然后再做处理。

中断处理部分很简单, 通过一个 ov_sta 来控制 OV7670 模块的 FIFO 写操作。当 ov_sta=0 的时候, 表示 FIFO 存储的数据已经被成功读取了 (ov_sta 在读完 FIFO 数据的时候被清零), 然后只要 OV_VSYNC 信号到来, 我们就先复位一下写指针, 然后 ov_sta=1, 标志着写指针已经复位, 目前正在往 FIFO 里面写数据。再等下一个 OV_VSYNC 到来, 也就表明一帧数据已经存储完毕了, 此时我们设置 OV7670_WREN 为 0, 禁止再往 OV7670 写入数据, 此时 ov_sta 自增为 2。其他程序, 只要读到 ov_sta 为 2, 就表示一帧数据已经准备好了, 可以读出, 在读完数据之后, 程序设置 ov_sta 为 0, 则开启下一轮 FIFO 数据存储。

再在 exti.h 里面添加 EXTI8_Init 函数的定义, 就完成对 exti.c 和 exti.h 的修改了。

最后, 打开 main.c 文件, 代码如下:

```

const u8*LMODE_TBL[5]={ "Auto","Sunny","Cloudy","Office","Home"};/5 种光照模式
const u8*EFFECTS_TBL[7]={ "Normal","Negative","B&W","Redish","Greenish","Bluish",
                           "Antique"}; //7 种特效

extern u8 ov_sta; //在 exti.c 里面定义
extern u8 ov_frame; //在 timer.c 里面定义
//更新 LCD 显示
void camera_refresh(void)
{
    u32 j;
    u16 color;
    if(ov_sta==2)

```



```
{  
    LCD_Scan_Dir(U2D_L2R);      //从上到下,从左到右  
    LCD_SetCursor(0x00,0x0000);  //设置光标位置  
    LCD_WriteRAM_Prepate();    //开始写入 GRAM  
    OV7670_RRST=0;             //开始复位读指针  
    OV7670_RCK=0;  
    OV7670_RCK=1;  
    OV7670_RCK=0;  
    OV7670_RRST=1;             //复位读指针结束  
    OV7670_RCK=1;  
    for(j=0;j<76800;j++)  
    {  
        OV7670_RCK=0;  
        color=GPIOC->IDR&0XFF; //读数据  
        OV7670_RCK=1;  
        color<<=8;  
        OV7670_RCK=0;  
        color|=GPIOC->IDR&0XFF; //读数据  
        OV7670_RCK=1;  
        LCD->LCD_RAM=color;  
    }  
    EXTI_ClearITPendingBit(EXTI_Line8); //清除 LINE8 上的中断标志位  
    ov_sto=0;                         //开始下一次采集  
    ov_frame++;  
    LCD_Scan_Dir(DFT_SCAN_DIR);       //恢复默认扫描方向  
}  
}  
int main(void)  
{  
    u8 key;  
    u8 lightmode=0,saturation=2,brightness=2,contrast=2;  
    u8 effect=0;  
    u8 i=0;  
    u8 msgbuf[15];//消息缓存区  
    u8 tm=0;  
    delay_init();           //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级  
    uart_init(9600);       //串口初始化波特率为 9600  
    LED_Init();            //LED 端口初始化  
    LCD_Init();            //LCD 初始化  
    KEY_Init();            //KEY 初始化  
    TPAD_Init(72);         //触摸按键初始化
```



```
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,200,16,16,"WarShip STM32");
LCD_ShowString(60,70,200,16,16,"OV7670 TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/14");
LCD_ShowString(60,130,200,16,16,"KEY0:Light Mode");
LCD_ShowString(60,150,200,16,16,"KEY1:Saturation");
LCD_ShowString(60,170,200,16,16,"KEY2:Brightness");
LCD_ShowString(60,190,200,16,16,"KEY_UP:Contrast");
LCD_ShowString(60,210,200,16,16,"TPAD:Effects");
LCD_ShowString(60,230,200,16,16,"OV7670 Init...");

while(OV7670_Init())//初始化 OV7670
{
    LCD_ShowString(60,230,200,16,16,"OV7670 Error!!!");
    delay_ms(200);
    LCD_Fill(60,230,239,246,WHITE);
    delay_ms(200);
}

LCD_ShowString(60,230,200,16,16,"OV7670 Init OK");
delay_ms(1500);

OV7670_Light_Mode(lightmode);
OV7670_Color_Saturation(saturation);
OV7670_Brightness(brightness);
OV7670_Contrast(contrast);
OV7670_Special_Effects(effect);

TIM6_Int_Init(10000,7199);           //10Khz 计数频率,1 秒钟中断
EXTI8_Init();                      //使能定时器捕获
OV7670_Window_Set(10,174,240,320); //设置窗口
OV7670_CS=0;

while(1)
{
    key=KEY_Scan(0);//不支持连接
    if(key)
    {
        tm=20;
        switch(key)
        {
            case KEY_RIGHT: //灯光模式 Light Mode
                lightmode++;
                if(lightmode>4)lightmode=0;
                OV7670_Light_Mode(lightmode);
                sprintf((char*)msgbuf,"%s",LMODE_TBL[lightmode]);
                break;
        }
    }
}
```



```
case KEY_DOWN://饱和度 Saturation
    saturation++;
    if(saturation>4)saturation=0;
    OV7670_Color_Saturation(saturation);
    sprintf((char*)msgbuf,"Saturation:%d",(signed char)saturation-2);
    break;
case KEY_LEFT: //亮度 Brightness
    brightness++;
    if(brightness>4)brightness=0;
    OV7670_Brightness(brightness);
    sprintf((char*)msgbuf,"Brightness:%d",(signed char)brightness-2);
    break;
case KEY_UP: //对比度 Contrast
    contrast++;
    if(contrast>4)contrast=0;
    OV7670_Contrast(contrast);
    sprintf((char*)msgbuf,"Contrast:%d",(signed char)contrast-2);
    break;
}
}

if(TPAD_Scan(0))//检测到触摸按键
{
    effect++;
    if(effect>6)effect=0;
    OV7670_Special_Effects(effect);//设置特效
    sprintf((char*)msgbuf,"%s",EFFECTS_TBL[effect]);
    tm=20;
}

camera_refresh();//更新显示
if(tm)
{
    LCD_ShowString(60,60,200,16,16,msgbuf);
    tm--;
}
i++;
if(i==15)//DS0 闪烁.
{
    i=0;
    LED0=!LED0;
}
}
```

此部分代码除了 main 函数，还有一个 camera_refresh 函数，该函数用于将摄像头模块 FIFO



的数据读出，并显示在 LCD 上面。main 函数则比较简单，我们就不细说了。

前面提到，我们要用 USMART 来设置摄像头的参数，我们只需要在 usmart_nametab 里面添加 SCCB_WR_Reg 和 SCCB_RD_Reg 这两个函数，就可以轻松调试摄像头了。

41.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 41.4.1 所示界面：

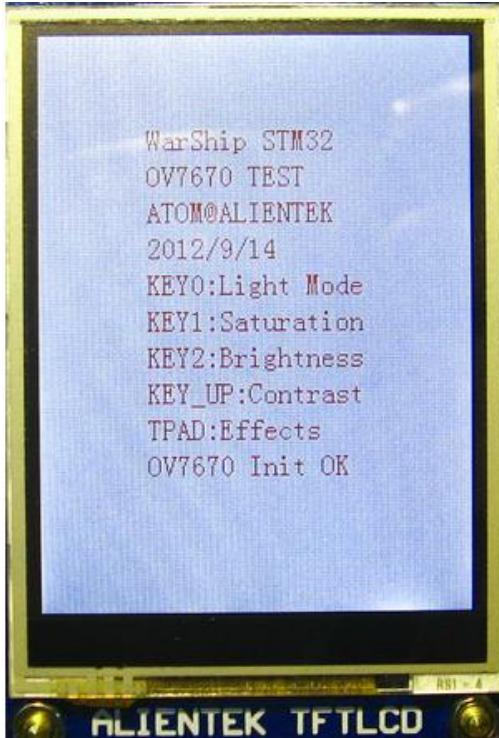


图 41.4.1 程序运行效果图

此时，我们可以按不同的按键（KEY0~KEY2、WK_UP、TPAD 等），来设置摄像头的相关参数和模式，得到不同的成像效果。同时，你还可以在串口，通过 USMART 调用 SCCB_WR_Reg 函数，来设置 OV7670 的各寄存器，达到调试测试 OV7670 的目的，如图 41.4.2 所示：



图 41.4.2 USMART 调试 OV7670

从上图还可以看出，LCD 显示帧率为 8 帧左右，则可以推断 OV7670 的输出帧率则至少是 $3 \times 8 = 24$ 帧以上（实际是 30 帧）。

第四十二章 外部 SRAM 实验

STM32F103ZET6 自带了 64K 字节的 SRAM，对一般应用来说，已经足够了，不过在一些对内存要求高的场合，STM32 自带的这些内存就不够用了。比如跑算法或者跑 GUI 等，就可能不太够用，所以战舰 STM32 开发板板载了一颗 1M 字节容量的 SRAM 芯片：IS62WV51216，满足大内存使用的需求。

本章，我们将使用 STM32 来驱动 IS62WV51216，实现对 IS62WV51216 的访问控制，并测试其容量。本章分为如下几个部分：

42.1 IS62WV51216 简介

42.2 硬件设计

42.3 软件设计

42.4 下载验证



42.1 IS62WV51216 简介

IS62WV51216 是 ISSI (Integrated Silicon Solution, Inc) 公司生产的一颗 16 位宽 512K (512*16, 即 1M 字节) 容量的 CMOS 静态内存芯片。该芯片具有如下几个特点：

- 高速。具有 45ns/55ns 访问速度。
- 低功耗。
- TTL 电平兼容。
- 全静态操作。不需要刷新和时钟电路。
- 三态输出。
- 字节控制功能。支持高/低字节控制。

IS62WV51216 的功能框图如图 42.1.1 所示：

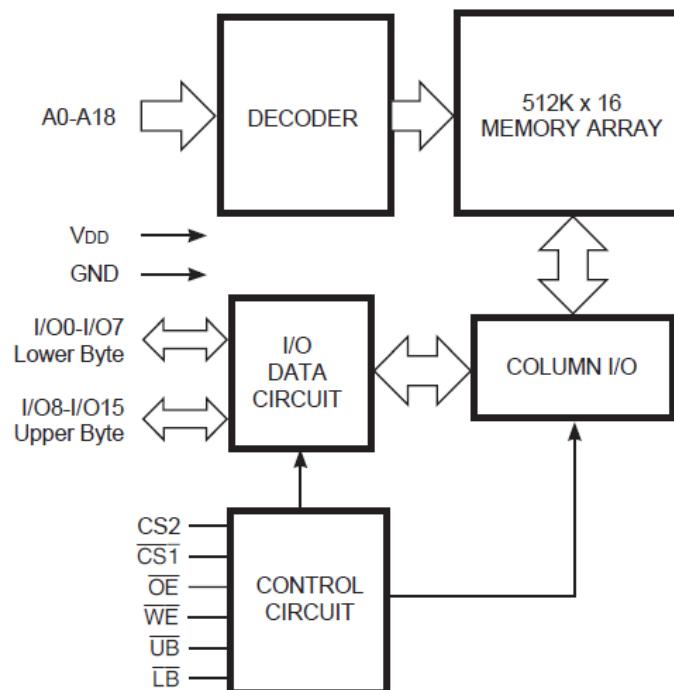


图 42.1.1 IS62WV51216 功能框图

图中 A0~18 为地址线，总共 19 根地址线（即 $2^{19}=512K$, $1K=1024$ ）；IO0~15 为数据线，总共 16 根数据线。CS2 和 CS1 都是片选信号，不过 CS2 是高电平有效 CS1 是低电平有效；OE 是输出使能信号（读信号）；WE 为写使能信号；UB 和 LB 分别是高字节控制和低字节控制信号；

战舰 STM32 开发板使用的是 TSOP44 封装的 IS62WV51216 芯片，该芯片直接接在 STM32 的 FSMC 上，IS62WV51216 原理图如图 42.1.2 所示：

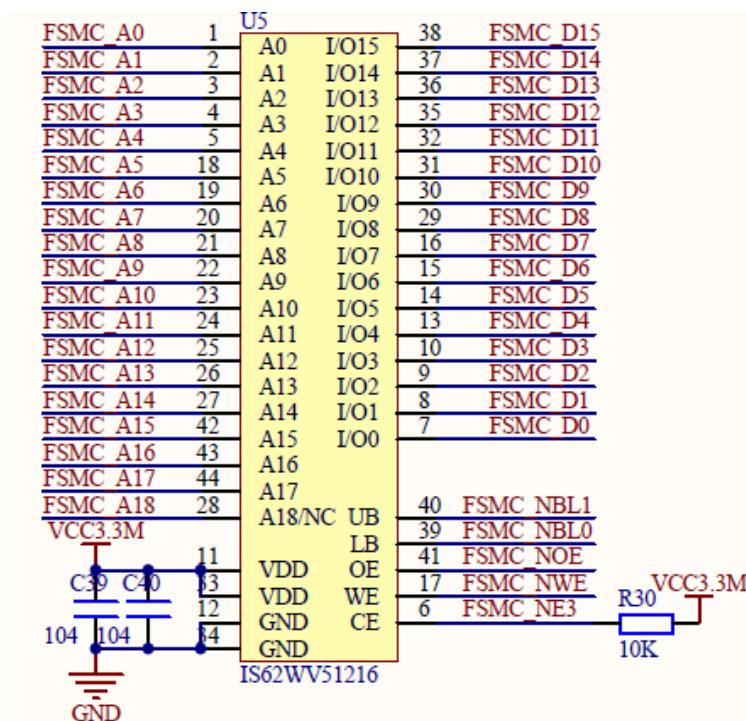


图 42.1.2 IS62WV51216 原理图

从原理图可以看出，IS62WV51216 同 STM32 的连接关系：

A[0:18]接 FSMC_A[0:18]

D[0:15]接 FSMC_D[0:15]

UB 接 FSMC_NBL1

LB 接 FSMC_NBL0

OE 接 FSMC_OE

WE 接 FSMC_WE

CS 接 FSMC_NE3

本章，我们使用 FSMC 的 BANK1 区域 3 来控制 IS62WV51216，关于 FSMC 的详细介绍，我们在第十八章已经介绍过，在第十八章，我们采用的是读写不同的时序来操作 TFTLCD 模块（因为 TFTLCD 模块读的速度比写的速度慢很多），但是在本章，因为 IS62WV51216 的读写时间基本一致，所以，我们设置读写相同的时序来访问 FSMC。关于 FSMC 的详细介绍，请大家看第十八章和《STM32 参考手册》。

IS62WV51216 就介绍到这，最后，我们来看看实现 IS62WV51216 的访问，需要对 FSMC 进行哪些配置。FSMC 的详细配置介绍在之前的 LCD 实验章节已经有详细讲解，这里就做一个概括性的讲解。步骤如下：

1) 使能 FSMC 时钟，并配置 FSMC 相关的 IO 及其时钟使能。

要使用 FSMC，当然首先得开启其时钟。然后需要把 FSMC_D0~15, FSMCA0~18 等相关 IO 口，全部配置为复用输出，并使能各 IO 组的时钟。

使能 FSMC 时钟的方法：

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC,ENABLE);
```

对于其他 IO 口设置的方法前面讲解很详细，这里不做过多的讲解。

2) 设置 FSMC BANK1 区域 3。

此部分包括设置区域 3 的存储器的工作模式、位宽和读写时序等。本章我们使用模式 A、



16位宽，读写共用一个时序寄存器。使用的函数是：

```
void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct)
```

这个函数的讲解在前面 LCD 实验的时候已经讲解很详细，所以大家可以回过头看看相关的讲解。具体的设置方法请参考我们的 sram.c 文件中的 `FSMC_SRAM_Init()` 函数。

3) 使能 BANK1 区域 3。

使能 BANK 的方法跟前面 LCD 实验也是一样的，这里也不做详细讲解，函数是：

```
void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);
```

通过以上几个步骤，我们就完成了 FSMC 的配置，可以访问 IS62WV51216 了，这里还需要注意，因为我们使用的是 BANK1 的区域 3，所以 HADDR[27:26]=10，故外部内存的首地址为 0X68000000。

42.2 硬件设计

本章实验功能简介：开机后，显示提示信息，然后按下 **KEY1** 按键，即测试外部 SRAM 容量大小并显示在 LCD 上。按下 **WK_UP** 按键，即显示预存在外部 SRAM 的数据。DS0 指示程序运行状态。

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY1 和 WK_UP 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) IS62WV51216

这些我们都已经介绍过 (IS62WV51216 与 STM32 的各 IO 对应关系，请参考光盘原理图)，接下来我们开始软件设计。

42.3 软件设计

打开外部 SRAM 实验工程，可以看到，我们增加了 sram.c 文件以及头文件 sram.h，FSMC 初始化相关配置和定义都在这两个文件中。同时还引入了 FSMC 固件库文件 `stm32f10x_fsmc.c` 和 `stm32f10x_fsmc.h` 文件。

打开 sram.c 文件，代码如下：

```
#include "sram.h"  
#include "usart.h"  
  
//使用 NOR/SRAM 的 Bank1.sector3,地址位 HADDR[27,26]=10  
//对 IS61LV25616/IS62WV25616,地址线范围为 A0~A17  
//对 IS61LV51216/IS62WV51216,地址线范围为 A0~A18  
#define Bank1_SRAM3_ADDR ((u32)(0x68000000))  
  
//初始化外部 SRAM  
void FSMC_SRAM_Init(void)  
{  
    FSMC_NORSRAMInitTypeDef  FSMC_NSInitStructure;  
    FSMC_NORSRAMTimingInitTypeDef  readWriteTiming;  
    GPIO_InitTypeDef  GPIO_InitStructure;
```



```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD|RCC_APB2Periph_GPIOE|
RCC_APB2Periph_GPIOF|RCC_APB2Periph_GPIOG,ENABLE);
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC,ENABLE);

GPIO_InitStructure.GPIO_Pin = 0xFF33;           //PORTD 复用推挽输出
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOD, &GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = 0xFF83;           //PORTE 复用推挽输出
GPIO_Init(GPIOE, &GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = 0xF03F;           //PORTD 复用推挽输出
GPIO_Init(GPIOF, &GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = 0x043F;           //PORTD 复用推挽输出
GPIO_Init(GPIOG, &GPIO_InitStructure);

readWriteTiming.FSMC_AddressSetupTime = 0x00;   //地址建立时间为 1 个 HCLK
readWriteTiming.FSMC_AddressHoldTime = 0x00;    //地址保持时间模式 A 未用到
readWriteTiming.FSMC_DataSetupTime = 0x03;       //数据保持时间为 3 个 HCLK
readWriteTiming.FSMC_BusTurnAroundDuration = 0x00;
readWriteTiming.FSMC_CLKDivision = 0x00;
readWriteTiming.FSMC_DataLatency = 0x00;
readWriteTiming.FSMC_AccessMode = FSMC_AccessMode_A; //模式 A

FSMC_NInitStruct.FSMC_Bank = FSMC_Bank1_NORSRAM3;// BTCR[4],[5]。
FSMC_NInitStruct.FSMC_DataAddressMux= FSMC_DataAddressMux_Disable;
FSMC_NInitStruct.FSMC_MemoryType =FSMC_MemoryType_SRAM //SRAM
FSMC_NInitStruct.FSMC_MemoryDataWidth= FSMC_MemoryDataWidth_16b;
                           //存储器数据宽度为 16bit
FSMC_NInitStruct.FSMC_BurstAccessMode=FSMC_BurstAccessMode_Disable;
FSMC_NInitStruct.FSMC_WaitSignalPolarity=FSMC_WaitSignalPolarity_Low;
FSMC_NInitStruct.FSMC_AsynchronousWait=FSMC_AsynchronousWait_Disable;
FSMC_NInitStruct.FSMC_WrapMode = FSMC_WrapMode_Disable;
FSMC_NInitStruct.FSMC_WaitSignalActive=
                           FSMC_WaitSignalActive_BeforeWaitState;
FSMC_NInitStruct.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
                           //存储器写使能
FSMC_NInitStruct.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NInitStruct.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable;
                           // 读写使用相同的时序
FSMC_NInitStruct.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
FSMC_NInitStruct.FSMC_ReadWriteTimingStruct = &readWriteTiming;
FSMC_NInitStruct.FSMC_WriteTimingStruct = &readWriteTiming;
FSMC_NORSRAMInit(&FSMC_NInitStruct); //初始化 FSMC 配置
```

```

        FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM3, ENABLE); // 使能 BANK3
    }
    //在指定地址开始,连续写入 n 个字节.
    // pBuffer:字节指针
    //WriteAddr:要写入的地址
    //n:要写入的字节数
    void FSMC_SRAM_WriteBuffer(u8* pBuffer,u32 WriteAddr,u32 n)
    {
        for(;n!=0;n--)
        {
            *(vu8*)(Bank1_SRAM3_ADDR+WriteAddr)=*pBuffer;
            WriteAddr+=2;//这里需要加 2, 是因为 STM32 的 FSMC
                           //地址右移一位对齐.加 2 相当于加 1.
            pBuffer++;
        }
    }
    //在指定地址开始,连续读出 n 个字节.
    // pBuffer:字节指针
    //ReadAddr:要读出的起始地址
    //n:要写入的字节数
    void FSMC_SRAM_ReadBuffer(u8* pBuffer,u32 ReadAddr,u32 n)
    {
        for(;n!=0;n--)
        {
            *pBuffer++=*(vu8*)(Bank1_SRAM3_ADDR+ReadAddr);
            ReadAddr+=2;//这里需要加 2, 是因为 STM32 的 FSMC 地
                         //址右移一位对齐.加 2 相当于加 1.
        }
    }
}

```

此部分代码包含 3 个函数，FSMC_SRAM_Init 函数用于初始化，包括 FSMC 相关 IO 口的初始化以及 FSMC 配置。另外，FSMC_SRAM_WriteBuffer 和 FSMC_SRAM_ReadBuffer 这两个函数分别用于在外部 SRAM 的指定地址写入和读取指定长度的数据（以字节为单位）。这里需要注意的是：FSMC 当位宽为 16 位的时候，HADDR 右移一位同地址对齐，但是 ReadAddr 我们这里却没有加 2，而是加 1，是因为我们这里用的数据为宽是 8 位，通过 UB 和 LB 来控制高低字节位，所以地址在这里是可以只加 1 的。另外，因为我们使用的是 BANK1，区域 3，所以外部 SRAM 的基址为：0x68000000。

sram.h 文件的内容就很简单，这里我们就不列出来了。

下面我们打开 main.c 文件，内容如下：

```

u32 testram[250000] __attribute__((at(0X68000000)));//测试用数组
//外部内存测试(最大支持 1M 字节内存测试)
void fsmc_sram_test(u16 x,u16 y)
{
    u32 i=0;

```



```
u8 temp=0;
u8 sval=0;    //在地址 0 读到的数据
LCD_ShowString(x,y,239,y+16,16,"Ex Memory Test: 0KB");
//每隔 4K 字节,写入一个数据,总共写入 256 个数据,刚好是 1M 字节
for(i=0;i<1024*1024;i+=4096)
{
    FSMC_SRAM_WriteBuffer(&temp,i,1);
    temp++;
}
//依次读出之前写入的数据,进行校验
for(i=0;i<1024*1024;i+=4096)
{
    FSMC_SRAM_ReadBuffer(&temp,i,1);
    if(i==0)sval=temp;
    else if(temp<=sval)break;//后面读出的数据一定要比第一次读到的数据大.
    LCD_ShowxNum(x+15*8,y,(u16)(temp-sval+1)*4,4,16,0); //显示内存容量
}
}

int main(void)
{
    u8 key;
    u8 i=0;
    u32 ts=0;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //LCD 初始化
    KEY_Init();              //KEY 初始化
    FSMC_SRAM_Init();       //初始化外部 SRAM
    POINT_COLOR=RED;         //设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"SRAM TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/16");
    LCD_ShowString(60,130,200,16,16,"KEY1:Test Sram");
    LCD_ShowString(60,150,200,16,16,"WK_UP:TEST Data");
    POINT_COLOR=BLUE;//设置字体为蓝色
    for(ts=0;ts<250000;ts++)testsram[ts]=ts;//预存测试数据
    while(1)
    {
        key=KEY_Scan(0);//不支持连接
    }
}
```



```
if(key==KEY_DOWN)fsmc_sram_test(60,170);//测试 SRAM 容量
else if(key==KEY_UP)//打印预存测试数据
{
    for(ts=0;ts<250000;ts++)LCD_ShowxNum(60,190,testsram[ts],6,16,0);
        //显示测试数据
}else delay_ms(10);
i++;
if(i==20)//DS0 闪烁.
{
    i=0;
    LED0=!LED0;
}
}
```

此部分代码除了 main 函数，还有一个 fsmc_sram_test 函数，该函数用于测试外部 SRAM 的容量大小，并显示其容量。main 函数则比较简单，我们就不细说了。

此段代码，我们定义了一个超大数组 testsram，我们指定该数组定义在外部 sram 起始地址（`__attribute__((at(0X68000000)))`），该数组用来测试外部 SRAM 数据的读写。注意该数组的定义方法，是我们推荐的使用外部 SRAM 的方法。如果想用 MDK 自动分配，那么需要用到分散加载还需要添加汇编的 FSMC 初始化代码，相对来说比较麻烦。而且外部 SRAM 访问速度又远不如内部 SRAM，如果将一些需要快速访问的 SRAM 定义到了外部 SRAM，将会严重拖慢程序运行速度。而如果以我们推荐的方式来分配外部 SRAM，那么就可以控制 SRAM 的分配，可以针对性的选择放外部还是放内部，有利于提高程序运行速度，使用起来也比较方便。

42.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 42.4.1 所示界面：



图 42.4.1 程序运行效果图

此时，我们按下 KEY1，就可以在 LCD 上看到内存测试的画面，同样，按下 WK_UP，就可以看到 LCD 显示存放在数组 testsram 里面的测试数据，如图 42.4.2 所示：



图 42.4.2 外部 SRAM 测试界面

该实验我们还可以借助 USMART 来测试，只需要在 usmart_nametab 里面添加读写 SRAM 的两个函数，就可以用 USMART 来测试外部 SRAM 了。



第四十三章 内存管理实验

上一节，我们学会了使用 STM32 驱动外部 SRAM，以扩展 STM32 的内存，加上 STM32 本身自带的 64K 字节内存，我们可供使用的内存还是比较多的。如果我们所用的内存都像上一节的 `testsram` 那样，定义一个数组来使用，显然不是一个好办法。

本章，我们将学习内存管理，实现对内存的动态管理。本章分为如下几个部分：

- 43.1 内存管理简介
- 43.2 硬件设计
- 43.3 软件设计
- 43.4 下载验证



43.1 内存管理简介

内存管理，是指软件运行时对计算机内存资源的分配和使用的技术。其最主要的目的是如何高效，快速的分配，并且在适当的时候释放和回收内存资源。内存管理的实现方法有很多种，他们其实最终都是要实现 2 个函数：malloc 和 free；malloc 函数用于内存申请，free 函数用于内存释放。

本章，我们介绍一种比较简单的办法来实现：分块式内存管理。下面我们介绍一下该方法的实现原理，如图 43.1.1 所示：

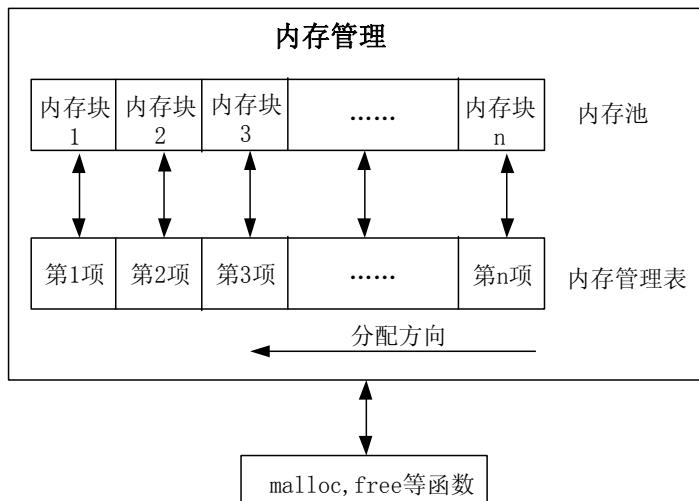


图 43.1.1 分块式内存管理原理

从上图可以看出，分块式内存管理由内存池和内存管理表两部分组成。内存池被等分为 n 块，对应的内存管理表，大小也为 n ，内存管理表的每一个项对应内存池的一块内存。

内存管理表的项值代表的意义为：当该项值为 0 的时候，代表对应的内存块未被占用，当该项值非零的时候，代表该项对应的内存块已经被占用，其数值则代表被连续占用的内存块数。比如某项值为 10，那么说明包括本项对应的内存块在内，总共分配了 10 个内存块给外部的某个指针。

内存分配方向如图所示，是从顶→底的分配方向。即首先从最末端开始找空内存。当内存管理刚初始化的时候，内存表全部清零，表示没有任何内存块被占用。

分配原理

当指针 p 调用 malloc 申请内存的时候，先判断 p 要分配的内存块数 (m)，然后从第 n 项开始，向下查找，直到找到 m 块连续的空内存块（即对应内存管理表项为 0），然后将这 m 个内存管理表项的值都设置为 m （标记被占用），最后，把最后的这个空内存块的地址返回指针 p ，完成一次分配。注意，如果当内存不够的时候（找到最后也没找到连续的 m 块空闲内存），则返回 NULL 给 p ，表示分配失败。

释放原理

当 p 申请的内存用完，需要释放的时候，调用 free 函数实现。free 函数先判断 p 指向的内存地址所对应的内存块，然后找到对应的内存管理表项目，得到 p 所占用的内存块数目 m （内存管理表项目的值就是所分配内存块的数目），将这 m 个内存管理表项目的值都清零，标记释放，完成一次内存释放。

关于分块式内存管理的原理，我们就介绍到这里。

43.2 硬件设计

本章实验功能简介：开机后，显示提示信息，等待外部输入。KEY0 用于申请内存，每次申请 2K 字节内存。KEY1 用于写数据到申请到的内存里面。KEY2 用于释放内存。WK_UP 用于切换操作内存区（内部内存/外部内存）。DS0 用于指示程序运行状态。本章我们还可以通过 USMART 调试，测试内存管理函数。

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 四个按键
- 3) 串口
- 4) TFTLCD 模块
- 5) IS62WV51216

这些我们都已经介绍过，接下来我们开始软件设计。

43.3 软件设计

本章，我们将内存管理部分单独做一个分组，在工程目录下新建一个 MALLOC 的文件夹，然后新建 malloc.c 和 malloc.h 两个文件，将他们保存在 MALLOC 文件夹下。

在 MDK 新建一个 MALLOC 的组，然后将 malloc.c 文件加入到该组，并将 MALLOC 文件夹添加到头文件包含路径。

打开 malloc.c 文件，输入如下代码：

```
#include "malloc.h"
//内存池(4字节对齐)
__align(4) u8 mem1base[MEM1_MAX_SIZE]; //内部 SRAM 内存池
__align(4) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0X68000000)));
//外部 SRAM 内存池
//内存管理表
u16 mem1mapbase[MEM1_ALLOC_TABLE_SIZE]; //内部 SRAM 内存池 MAP
u16 mem2mapbase[MEM2_ALLOC_TABLE_SIZE] __attribute__((at(0X68000000+
MEM2_MAX_SIZE))); //外部 SRAM 内存池 MAP
//内存管理参数
const u32 memtblsize[2]={MEM1_ALLOC_TABLE_SIZE, MEM2_ALLOC_TABLE_SIZE};
//内存表大小
const u32 memblksize[2]={MEM1_BLOCK_SIZE, MEM2_BLOCK_SIZE};
//内存分块大小
const u32 memsize[2]={MEM1_MAX_SIZE, MEM2_MAX_SIZE};
//内存总大小
//内存管理控制器
struct _m_malloc_dev malloc_dev=
{
    mem_init,                      //内存初始化
    mem_perused,                   //内存使用率
    mem1base,mem2base,             //内存池
    mem1mapbase,mem2mapbase,        //内存管理状态表
}
```



```
0,0,           //内存管理未就绪
};

//复制内存
//*des:目的地址
//*src:源地址
//n:需要复制的内存长度(字节为单位)
void mymemcpy(void *des,void *src,u32 n)
{
    u8 *xdes=des;
    u8 *xsrc=src;
    while(n--)*xdes++=*xsrc++;
}

//设置内存
//*s:内存首地址
//c :要设置的值
//count:需要设置的内存大小(字节为单位)
void mymemset(void *s,u8 c,u32 count)
{
    u8 *xs = s;
    while(count--)*xs++=c;
}

//内存管理初始化
//memx:所属内存块
void mem_init(u8 memx)
{
    mymemset(mallco_dev.memmap[memx], 0,memtblsize[memx]*2); //内存状态表数据清零
    mymemset(mallco_dev.membase[memx], 0,memsize[memx]); //内存池所有数据清零
    mallco_dev.memrdy[memx]=1; //内存管理初始化 OK
}

//获取内存使用率
//memx:所属内存块
//返回值:使用率(0~100)
u8 mem_perused(u8 memx)
{
    u32 used=0;
    u32 i;
    for(i=0;i<memtblsize[memx];i++) if(mallco_dev.memmap[memx][i])used++;
    return (used*100)/(memtblsize[memx]);
}

//内存分配(内部调用)
//memx:所属内存块
//size:要分配的内存大小(字节)
//返回值:0xFFFFFFFF代表错误;其他,内存偏移地址
```



```
u32 mem_malloc(u8 memx,u32 size)
{
    signed long offset=0;
    u16 nmemb; //需要的内存块数
    u16 cmemb=0;//连续空内存块数
    u32 i;
    if(!mallco_dev.memrdy[memx])mallco_dev.init(memx);//未初始化,先执行初始化
    if(size==0)return 0xFFFFFFFF; //不需要分配
    nmemb=size/memblksize[memx]; //获取需要分配的连续内存块数
    if(size%memblksize[memx])nmemb++;
    for(offset=memtblsize[memx]-1;offset>=0;offset--)//搜索整个内存控制区
    {
        if(!mallco_dev.memmap[memx][offset])cmemb++; //连续空内存块数增加
        else cmemb=0; //连续内存块清零
        if(cmemb==nmemb) //找到了连续 nmemb 个空内存块
        {
            for(i=0;i<nmemb;i++) //标注内存块非空
            {
                mallco_dev.memmap[memx][offset+i]=nmemb;
            }
            return (offset*memblksize[memx]); //返回偏移地址
        }
    }
    return 0xFFFFFFFF; //未找到符合分配条件的内存块
}
//释放内存(内部调用)
//memx:所属内存块
//offset:内存地址偏移
//返回值:0,释放成功;1,释放失败;
u8 mem_free(u8 memx,u32 offset)
{
    int i;
    if(!mallco_dev.memrdy[memx])//未初始化,先执行初始化
    {
        mallco_dev.init(memx); return 1;//未初始化
    }
    if(offset<memsize[memx])//偏移在内存池内.
    {
        int index=offset/memblksize[memx]; //偏移所在内存块号码
        int nmemb=mallco_dev.memmap[memx][index]; //内存块数量
        for(i=0;i<nmemb;i++) //内存块清零
        {
            mallco_dev.memmap[memx][index+i]=0;
        }
    }
}
```



```
    }
    return 0;
}else return 2;//偏移超区了.
}

//释放内存(外部调用)
//memx:所属内存块
//ptr:内存首地址
void myfree(u8 memx,void *ptr)
{
    u32 offset;
    if(ptr==NULL) return;//地址为 0.
    offset=(u32)ptr-(u32)mallco_dev.membase[memx];
    mem_free(memx,offset);//释放内存
}

//分配内存(外部调用)
//memx:所属内存块
//size:内存大小(字节)
//返回值:分配到的内存首地址.
void *mymalloc(u8 memx,u32 size)
{
    u32 offset;
    offset=mem_malloc(memx,size);
    if(offset==0xFFFFFFFF) return NULL;
    else return (void*)((u32)mallco_dev.membase[memx]+offset);
}

//重新分配内存(外部调用)
//memx:所属内存块
//*ptr:旧内存首地址
//size:要分配的内存大小(字节)
//返回值:新分配到的内存首地址.
void *myrealloc(u8 memx,void *ptr,u32 size)
{
    u32 offset;
    offset=mem_malloc(memx,size);
    if(offset==0xFFFFFFFF) return NULL;
    else
    {
        mymemcpy((void*)((u32)mallco_dev.membase[memx]+offset),ptr,size);
        //拷贝旧内存内容到新内存
        myfree(memx,ptr); //释放旧内存
        return (void*)((u32)mallco_dev.membase[memx]+offset); //返回新内存首地址
    }
}
```



这里，我们通过内存管理控制器 mallco_dev 结构体（mallco_dev 结构体见 malloc.h），实现对两个内存池的管理控制。一个是外部内存池，定义为：

```
_align(4) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0X68000000)));
```

另一个是内部内存池，定义为：

```
_align(4) u8 mem1base[MEM1_MAX_SIZE];
```

其中，MEM1_MAX_SIZE 和 MEM2_MAX_SIZE 为在 malloc.h 里面定义的内存池大小，外部内存池指定地址为 0X68000000，也就是从外部 SRAM 的首地址开始的，内部内存则由编译器自动分配。`_align(4)` 定义内存池为 4 字节对齐，这个非常重要！如果不加这个限制，在某些情况下（比如分配内存给结构体指针），可能出现错误，所以一定要加上这个。

此部分代码的核心函数为：mem_malloc 和 mem_free，分别用于内存申请和内存释放。思路就是我们在 43.1 接所介绍的那样分配和释放内存，不过这两个函数只是内部调用，外部调用我们使用的是 mymalloc 和 myfree 两个函数。其他函数我们就不多介绍了，保存 malloc.c，然后，打开 malloc.h，在该文件里面输入如下代码：

```
#ifndef __MALLOC_H
#define __MALLOC_H

typedef unsigned long  u32;
typedef unsigned short u16;
typedef unsigned char   u8;

#ifndef NULL
#define NULL 0
#endif

#define SRAMIN  0 //内部内存池
#define SRAMEX  1 //外部内存池

//mem1 内存参数设定.mem1 完全处于内部 SRAM 里面
#define MEM1_BLOCK_SIZE          32      //内存块大小为 32 字节
#define MEM1_MAX_SIZE            40*1024 //最大管理内存 40K
#define MEM1_ALLOC_TABLE_SIZE    MEM1_MAX_SIZE/MEM1_BLOCK_SIZE
//内存表大小

//mem2 内存参数设定.mem2 的内存池处于外部 SRAM 里面,其他的处于内部 SRAM 里面
#define MEM2_BLOCK_SIZE          32      //内存块大小为 32 字节
#define MEM2_MAX_SIZE            200*1024 //最大管理内存 200K
#define MEM2_ALLOC_TABLE_SIZE    MEM2_MAX_SIZE/MEM2_BLOCK_SIZE
//内存表大小

//内存管理控制器
struct _m_mallco_dev
{
    void (*init)(u8);           //初始化
    u8 (*perused)(u8);         //内存使用率
    u8 *membase[2];             //内存池 管理 2 个区域的内存
    u16 *memmap[2];             //内存管理状态表
    u8  memrdy[2];              //内存管理是否就绪
};

extern struct _m_mallco_dev mallco_dev; //在 mallco.c 里面定义
```



```

void mymemset(void *s,u8 c,u32 count);      //设置内存
void mymemcpy(void *des,void *src,u32 n) ;   //复制内存
void mem_init(u8 memx);                      //内存管理初始化函数(外/内部调用)
u32 mem_malloc(u8 memx,u32 size);            //内存分配(内部调用)
u8 mem_free(u8 memx,u32 offset);             //内存释放(内部调用)
u8 mem_perused(u8 memx);                     //获得内存使用率(外/内部调用)

///////////////////////////////
//用户调用函数
void myfree(u8 memx,void *ptr);              //内存释放(外部调用)
void *mymalloc(u8 memx,u32 size);             //内存分配(外部调用)
void *myrealloc(u8 memx,void *ptr,u32 size);  //重新分配内存(外部调用)
#endif

```

这部分代码，定义了很多关键数据，比如内存块大小的定义：MEM1_BLOCK_SIZE 和 MEM2_BLOCK_SIZE，都是 32 字节。内存池总大小，内部为 40K，外部为 200K（最大支持到近 1M 字节，不过为了方便演示，这里只管理 200K 内存）。MEM1_ALLOC_TABLE_SIZE 和 MEM2_ALLOC_TABLE_SIZE，则分别代表内存池 1 和 2 的内存管理表大小。

从这里可以看出，如果内存分块越小，那么内存管理表就越大，当分块为 2 字节 1 个块的时候，内存管理表就和内存池一样大了（管理表的每项都是 u16 类型）。显然是不合适的，我们这里取 32 字节，比例为 1:16，内存管理表相对就比较小了。

其他就不多说了，大家自行看代码理解就好。保存此部分代码。最后，打开 main.c 文件，修改代码如下：

```

int main(void)
{
    u8 key;
    u8 i=0;
    u8 *p=0;
    u8 *tp=0;
    u8 paddr[18];           //存放 P Addr:+p 地址的 ASCII 值
    u8 sramx=0;              //默认为内部 sram
    delay_init();            //延时函数初始化
    NVIC_Configuration();   //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);          //串口初始化波特率为 9600
    LED_Init();                //LED 端口初始化
    LCD_Init();                //LCD 初始化
    KEY_Init();                //KEY 初始化
    usmart_dev.init(72);       //初始化 USMART
    KEY_Init();                //按键初始化
    FSMC_SRAM_Init();          //初始化外部 SRAM
    mem_init(SRAMIN);          //初始化内部内存池
    mem_init(SRAMEX);          //初始化外部内存池
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"MALLOC TEST");
}

```



```
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/16");
LCD_ShowString(60,130,200,16,16,"KEY0:Malloc KEY2:Free");
LCD_ShowString(60,150,200,16,16,"KEY_UP:SRAMx KEY1:Read");
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(60,170,200,16,16,"SRAMIN");
LCD_ShowString(60,190,200,16,16,"SRAMIN USED: %");
LCD_ShowString(60,210,200,16,16,"SRAMEX USED: %");
while(1)
{
    key=KEY_Scan(0);//不支持连接
    switch(key)
    {
        case 0://没有按键按下
            break;
        case KEY_RIGHT://KEY0 按下
            p=mymalloc(sramx,2048);//申请 2K 字节
            if(p!=NULL)sprintf((char*)p,"Memory Malloc Test%03d",i);
            //向 p 写入一些内容
            break;
        case KEY_DOWN://KEY1 按下
            if(p!=NULL)
            {
                sprintf((char*)p,"Memory Malloc Test%03d",i);//更新显示内容
                LCD_ShowString(60,250,200,16,16,p);          //显示 P 的内容
            }
            break;
        case KEY_LEFT://KEY2 按下
            myfree(sramx,p);//释放内存
            p=0;           //指向空地址
            break;
        case KEY_UP://KEY UP 按下
            sramx=!sramx; //切换当前 malloc/free 操作对象
            if(sramx)LCD_ShowString(60,170,200,16,16,"SRAMEX");
            else LCD_ShowString(60,170,200,16,16,"SRAMIN");
            break;
    }
    if(tp!=p)
    {
        tp=p;
        sprintf((char*)paddr,"P Addr:0X%08X",(u32)tp);
        LCD_ShowString(60,230,200,16,16,paddr);      //显示 p 的地址
        if(p)LCD_ShowString(60,250,200,16,16,p);      //显示 P 的内容
    }
}
```



```
    else LCD_Fill(60,250,239,266,WHITE);      //p=0,清除显示
}
delay_ms(10);
i++;
if((i%20)==0)//DS0 闪烁.
{
    LCD_ShowNum(60+96,190,mem_perused(SRAMIN),3,16);
    //显示内部内存使用率
    LCD_ShowNum(60+96,210,mem_perused(SRAMEX),3,16);
    //显示外部内存使用率
    LED0=!LED0;
}
}
```

该部分代码比较简单，主要是对 mymalloc 和 myfree 的应用。不过这里提醒大家，如果对一个指针进行多次内存申请，而之前的申请又没释放，那么将造成“内存泄露”，这是内存管理所不希望发生的，久而久之，可能导致无内存可用的情况！所以，在使用的时候，请大家一定记得，申请的内存在用完以后，一定要释放。

另外，本章希望利用 USMART 调试内存管理，所以在 USMART 里面添加了 mymalloc 和 myfree 两个函数，用于测试内存分配和内存释放。大家可以通过 USMART 自行测试。

43.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 43.4.1 所示界面：

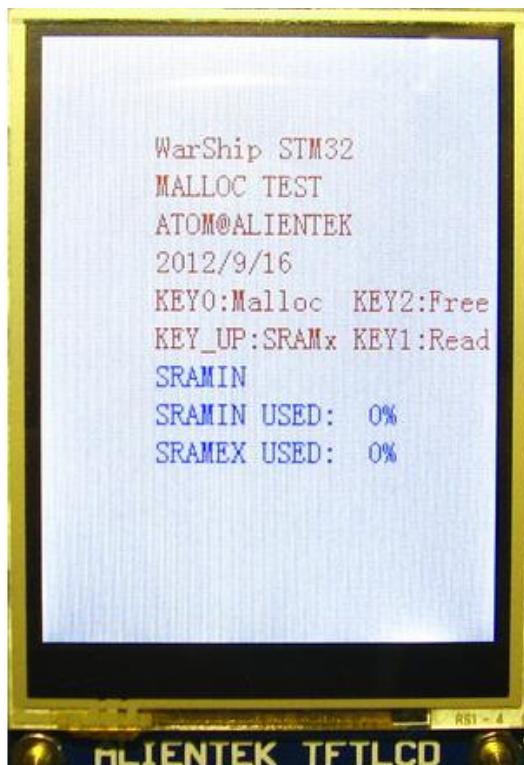




图 43.4.1 程序运行效果图

可以看到，内外内存的使用率均为 0%，说明还没有任何内存被使用，此时我们按下 KEY0，就可以看到内部内存被使用 5% 了，同时看到下面提示了指针 p 所指向的地址（其实就是被分配到的内存地址）和内容。多按几次 KEY0，可以看到内存使用率持续上升（注意对比 p 的值，可以发现是递减的，说明是从顶部开始分配内存！），此时如果按下 KEY2，可以发现内存使用率降低了 5%，但是再按 KEY2 将不再降低，说明“内存泄露”了。这就是前面提到的对一个指针多次申请内存，而之前申请的内存又没释放，导致的“内存泄露”。

按 KEY_UP 按键，可以切换当前操作内存（内部内存/外部内存），KEY1 键用于更新 p 的内容，更新后的内容将重新显示在 LCD 模块上面。

本章，我们还可以借助 USMART，测试内存的分配和释放，有兴趣的朋友可以动手试试。如图 43.4.2 所示：

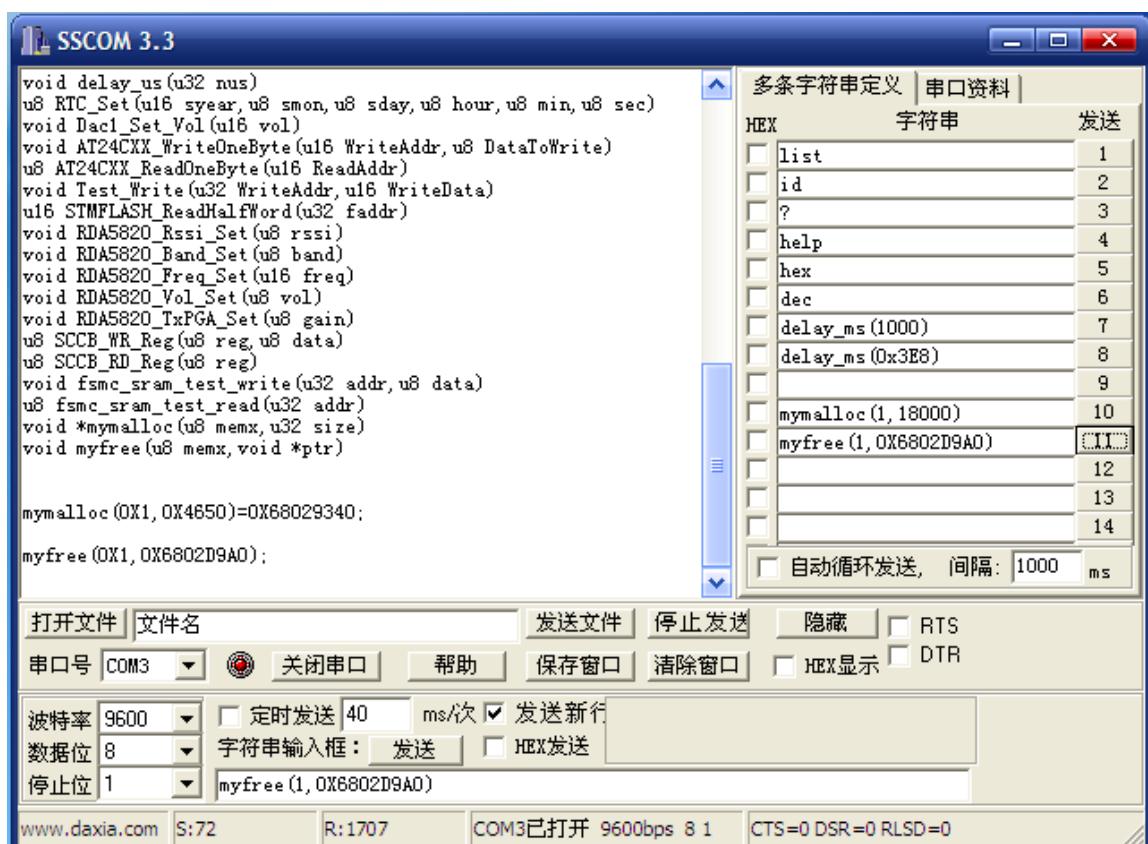


图 43.4.2 USMART 测试内存管理函数



第四十四章 SD 卡实验

很多单片机系统都需要大容量存储设备，以存储数据。目前常用的有 U 盘，FLASH 芯片，SD 卡等。他们各有优点，综合比较，最适合单片机系统的莫过于 SD 卡了，它不仅容量可以做到很大（32Gb 以上），而且支持 SPI 接口，方便移动，并且有几种体积的尺寸可供选择（标准的 SD 卡尺寸，以及 TF 卡尺寸等），能满足不同应用的要求。

只需要 4 个 IO 口即可外扩一个最大达 32GB 以上的外部存储器，容量从几十 M 到几十 G 选择尺度很大，更换也很方便，编程也简单，是单片机大容量外部存储器的首选。

ALIENTKE 战舰 STM32 开发板自带了标准的 SD 卡接口，可使用 STM32 自带的 SPI/SDIO 接口驱动（通过跳线帽选择驱动方式），本章我们使用 SPI 驱动，最高通信速度可达 18Mbps，每秒可传输数据 2M 字节以上，对于一般应用足够了。在本章中，我们将向大家介绍，如何在 ALIENTEK 战舰 STM32 开发板上实现 SD 卡的读取。本章分为如下几个部分：

- 44.1 SD 卡简介
- 44.2 硬件设计
- 44.3 软件设计
- 44.4 下载验证



44.1 SD 卡简介

SD 卡 (Secure Digital Memory Card) 中文翻译为安全数码卡，它是在 MMC 的基础上发展而来，是一种基于半导体快闪记忆器的新一代记忆设备，它被广泛地于便携式装置上使用，例如数码相机、个人数码助理(PDA)和多媒体播放器等。SD 卡由日本松下、东芝及美国 SanDisk 公司于 1999 年 8 月共同开发研制。大小犹如一张邮票的 SD 记忆卡，重量只有 2 克，但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。按容量分类，可以将 SD 卡分为 3 类：SD 卡、SDHC 卡、SDXC 卡。如表 44.1.1 所示：

容量	命名	简称
0~2G	Standard Capacity SD Memory Card	SDSC 或 SD
2G~32G	High Capacity SD Memory Card	SDHC
32G~2T	Extended Capacity SD Memory Card	SDXC

表 44.1.1 SD 卡按容量分类

SD 卡和 SDHC 卡协议基本兼容，但是 SDXC 卡，同这两者区别就比较大了，本章我们讨论的主要是 SD/SDHC 卡（简称 SD 卡）。

SD 卡一般支持 2 种操作模式：

- 1, SD 卡模式（通过 SDIO 通信）；
- 2, SPI 模式；

主机可以选择以上任意一种模式同 SD 卡通信，SD 卡模式允许 4 线的高速数据传输。SPI 模式允许简单的通过 SPI 接口来和 SD 卡通信，这种模式同 SD 卡模式相比就是丧失了速度。

SD 卡的引脚排序如下图 44.1.1 所示：

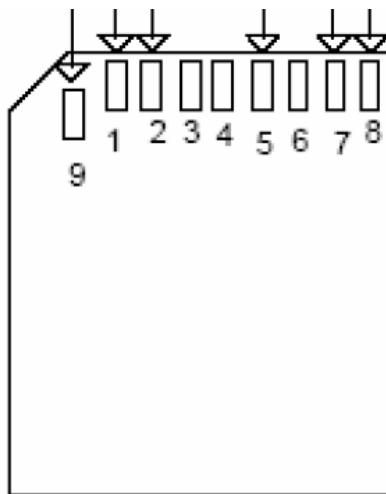


图 44.1.1 SD 卡引脚排序图

SD 卡引脚功能描述如表 45.1.2 所示：

针脚	1	2	3	4	5	6	7	8	9
SD卡模式	CD/DAT3	CMD	VSS	VCC	CLK	VSS	DATO	DAT1	DAT2
SPI模式	CS	MOSI	VSS	VCC	CLK	VSS	MISO	NC	NC

表 45.1.2 SD 卡引脚功能表

SD 卡只能使用 3.3V 的 IO 电平，所以，MCU 一定要能够支持 3.3V 的 IO 端口输出。注意：在 SPI 模式下，CS/MOSI/MISO/CLK 都需要加 10~100K 左右的上拉电阻。



SD 卡有 5 个寄存器，如表 45.1.3 所示：

名称	宽度	描述
CID	128	卡标识寄存器
RCA	16	相对卡地址 (Relative card address) 寄存器：本地系统中卡的地址，动态变化，在主机初始化的时候确定 *SPI 模式中没有
CSD	128	卡描述数据：卡操作条件相关的信息数据
SCR	64	SD 配置寄存器：SD 卡特定信息数据
OCR	32	操作条件寄存器

表 45.1.3 SD 卡相关寄存器

关于这些寄存器的详细描述，请参考光盘相关 SD 卡资料。我们在这里就不描述了。接下来，我们看看 SD 卡的命令格式，如表 45.1.4 所示：

字节 1			字节 2--5		字节 6		
7	6	5 0	31	0	7	1	0
0	1	command	命令参数		CRC		1

表 45.1.4 SD 卡命令格式

SD 卡的指令由 6 个字节组成，字节 1 的最高 2 位固定为 01，低 6 位为命令号（比如 CMD16，为 10000B 即 16 进制的 0X10，完整的 CMD16，第一个字节为 01010000，即 0X10+0X40）。

字节 2~5 为命令参数，有些命令是没有参数的。

字节 6 的高七位为 CRC 值，最低位恒定为 1。

SD 卡的命令总共有 12 类，分为 Class0~Class11，本章，我们仅介绍几个比较重要的命令，如表 45.1.5 所示：

命令	参数	回应	描述
CMD0 (0X00)	NONE	R1	复位 SD 卡
CMD8 (0X08)	VHS+Check pattern	R7	发送接口状态命令
CMD9 (0X09)	NONE	R1	读取卡特定数据寄存器
CMD10 (0X0A)	NONE	R1	读取卡标志数据寄存器
CMD16 (0X10)	块大小	R1	设置块大小（字节数）
CMD17 (0X11)	地址	R1	读取一个块的数据
CMD24 (0X18)	地址	R1	写入一个块的数据
CMD41 (0X29)	NONE	R3	发送给主机容量支持信息和激活卡初始化过程
CMD55 (0X37)	NONE	R1	告诉 SD 卡，下一个特定应用命令
CMD58 (0X3A)	NONE	R3	读取 OCR 寄存器

表 45.1.5 SD 卡部分命令

上表中，大部分的命令是初始化的时候用的。表中的 R1、R3 和 R7 等是 SD 卡的回应，SD 卡和单片机的通信采用发送应答机制，如图 45.1.2 所示：

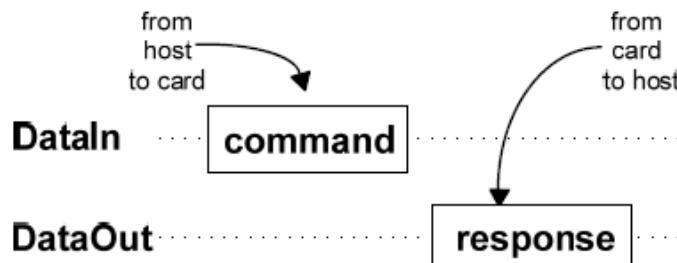


图 45.1.2 SD 卡命令传输过程

每发送一个命令，SD 卡都会给出一个应答，以告知主机该命令的执行情况，或者返回主机需要获取的数据。SPI 模式下，SD 卡针对不同的命令，应答可以是 R1~R7，R1 的应答，各位描述如表 45.1.6 所示：

R1 响应格式								
位	7	6	5	4	3	2	1	0
含义	开始位 始终为 0	参数 错误	地址 错误	擦除序列 错误	CRC 错误	非法 命令	擦除 复位	闲置 状态

表 45.1.6 R1 响应各位描述

R2~R7 的响应，我们就不介绍了，请的大家参考 SD 卡 2.0 协议。接下来，我们看看 SD 卡初始化过程。因为我们使用的是 SPI 模式，所以先得让 SD 卡进入 SPI 模式。方法如下：在 SD 卡收到复位命令 (CMD0) 时，CS 为有效电平 (低电平) 则 SPI 模式被启用。不过在发送 CMD0 之前，要发送 >74 个时钟，这是因为 SD 卡内部有个供电电压上升时间，大概为 64 个 CLK，剩下的 10 个 CLK 用于 SD 卡同步，之后才能开始 CMD0 的操作，在卡初始化的时候，CLK 时钟最大不能超过 400Khz！。

接着我们看看 SD 卡的初始化，SD 卡的典型初始化过程如下：

- 1、初始化与 SD 卡连接的硬件条件 (MCU 的 SPI 配置，IO 口配置);
- 2、上电延时 (>74 个 CLK);
- 3、复位卡 (CMD0)，进入 IDLE 状态；
- 4、发送 CMD8，检查是否支持 2.0 协议；
- 5、根据不同协议检查 SD 卡 (命令包括：CMD55、CMD41、CMD58 和 CMD1 等)；
- 6、取消片选，发多 8 个 CLK，结束初始化

这样我们就完成了对 SD 卡的初始化，注意末尾发送的 8 个 CLK 是提供 SD 卡额外的时钟，完成某些操作。通过 SD 卡初始化，我们可以知道 SD 卡的类型 (V1、V2、V2HC 或者 MMC)，在完成了初始化之后，就可以开始读写数据了。

SD 卡读取数据，这里通过 CMD17 来实现，具体过程如下：

- 1、发送 CMD17；
- 2、接收卡响应 R1；
- 3、接收数据起始令牌 0XFE；
- 4、接收数据；
- 5、接收 2 个字节的 CRC，如果不使用 CRC，这两个字节在读取后可以丢掉。
- 6、禁止片选之后，发多 8 个 CLK；

以上就是一个典型的读取 SD 卡数据过程，SD 卡的写于读数据差不多，写数据通过 CMD24 来实现，具体过程如下：



- 1、发送 CMD24;
- 2、接收卡响应 R1;
- 3、发送写数据起始令牌 0XFE;
- 4、发送数据;
- 5、发送 2 字节的伪 CRC;
- 6、禁止片选之后，发多 8 个 CLK;

以上就是一个典型的写 SD 卡过程。关于 SD 卡的介绍，我们就介绍到这里，更详细的介绍请参考光盘 SD 卡的参考资料（SD 卡 2.0 协议）。

44.2 硬件设计

本章实验功能简介：开机的时候先初始化 SD 卡，如果 SD 卡初始化完成，则提示 LCD 初始化成功。按下 KEY0，读取 SD 卡扇区 0 的数据，然后通过串口发送到电脑。如果没初始化通过，则在 LCD 上提示初始化失败。同样用 DS0 来指示程序正在运行。

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY0 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡接口
- 6) SD 卡

前面四部分，在之前的实例已经介绍过了，这里我们介绍一下 SD 卡接口和 STM32 的连接关系，如图 44.2.1 所示：

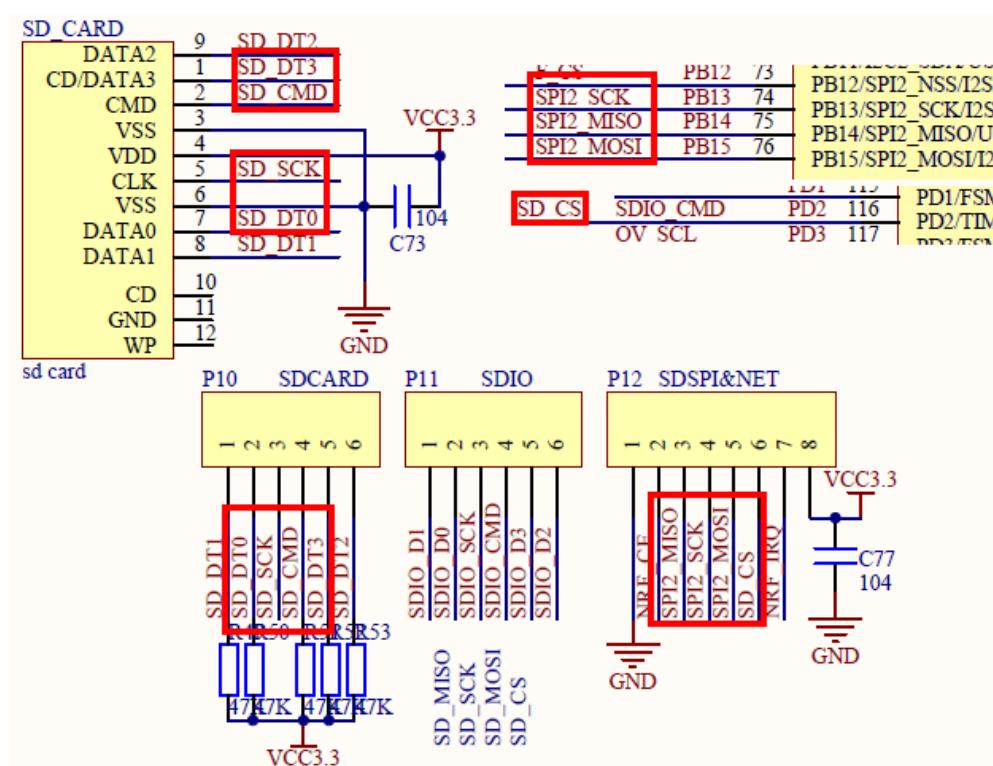


图44.2.1 SD卡接口与STM32连接原理图



我们用跳线帽将P10的SD_DT3、SD_CMD、SD_SCK、SD_DT0分别同P12的SD_CS、SPI2_MOSI、SPI2_SCK、SPI2_MISO连接起来，即实现SD卡的SPI模式连接。硬件连接示意图如图44.2.2所示：

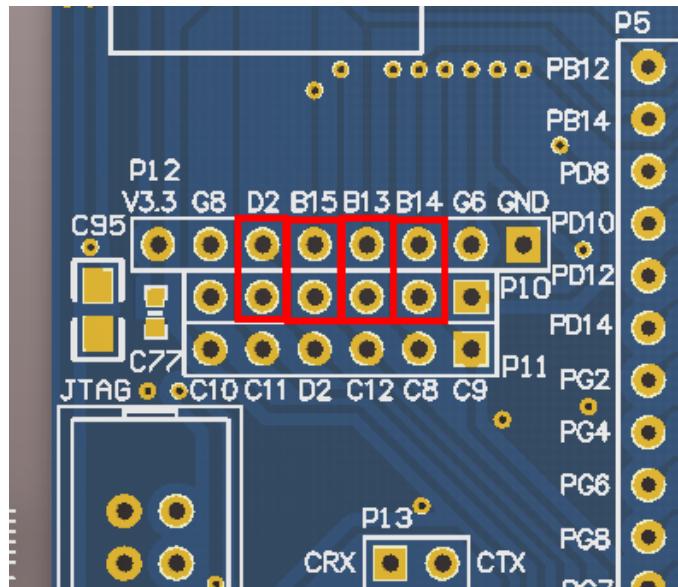


图44.2.2 SD卡SPI方式硬件连接示意图

将图中所示的4处，用跳线帽短接，接口实现SD卡与STM32的SPI连接。最后，你还得自备一个SD卡，将其插入板子下面的SD卡接口。

44.3 软件设计

打开SD卡实验工程，可以看到我们新建了MMC_SD.C文件和MMC_SD.h，所有SD卡相关的驱动代码和定义都在这两个文件中。

打开MMC_SD.C文件，在该文件里面，我们输入与SD卡相关的操作代码，这里由于篇幅限制，我们不贴出所有代码，仅介绍两个最重要的函数，第一个是SD_Initialize函数，该函数源码如下：

```
//初始化SD卡
u8 SD_Initialize(void)
{
    u8 r1; //存放SD卡的返回值
    u16 retry; //用来进行超时计数
    u8 buf[4];
    u16 i;
    SD_SPI_Init(); //初始化IO
    SD_SPI_SpeedLow(); //设置到低速模式
    for(i=0;i<10;i++)SD_SPI_ReadWriteByte(0xFF); //发送最少74个脉冲
    retry=20;
    do
    {
        r1=SD_SendCmd(CMD0,0,0x95); //进入IDLE状态
    }while((r1!=0X01) && retry--);
```



```
SD_Type=0;//默认无卡
if(r1==0X01)
{
    if(SD_SendCmd(CMD8,0x1AA,0x87)==1)//SD V2.0
    {
        for(i=0;i<4;i++)buf[i]=SD_SPI_ReadWriteByte(0XFF);
                    //Get trailing return value of R7 resp
        if(buf[2]==0X01&&buf[3]==0XAA)//卡是否支持 2.7~3.6V
        {
            retry=0XFFF;
            do
            {
                SD_SendCmd(CMD55,0,0X01); //发送 CMD55
                r1=SD_SendCmd(CMD41,0x40000000,0X01);//发送 CMD41
            }while(r1&&retry--);
            if(retry&&SD_SendCmd(CMD58,0,0X01)==0)//鉴别 SD2.0 卡版本开始
            {
                for(i=0;i<4;i++)buf[i]=SD_SPI_ReadWriteByte(0XFF); //得到 OCR 值
                if(buf[0]&0x40)SD_Type=SD_TYPE_V2HC;      //检查 CCS
                else SD_Type=SD_TYPE_V2;
            }
        }
    }
}else//SD V1.x/ MMC  V3
{
    SD_SendCmd(CMD55,0,0X01);      //发送 CMD55
    r1=SD_SendCmd(CMD41,0,0X01);   //发送 CMD41
    if(r1<=1)
    {
        SD_Type=SD_TYPE_V1;
        retry=0XFFF;
        do //等待退出 IDLE 模式
        {
            SD_SendCmd(CMD55,0,0X01); //发送 CMD55
            r1=SD_SendCmd(CMD41,0,0X01);//发送 CMD41
        }while(r1&&retry--);
    }
    else
    {
        SD_Type=SD_TYPE_MMC;//MMC V3
        retry=0XFFF;
        do //等待退出 IDLE 模式
        {
            r1=SD_SendCmd(CMD1,0,0X01);//发送 CMD1
        }while(r1&&retry--);
    }
}
```



```
        }
        if(retry==0||SD_SendCmd(CMD16,512,0X01)!=0)SD_Type=SD_TYPE_ERR;
                                //错误的卡
    }
}

SD_DisSelect();      //取消片选
SD_SPI_SpeedHigh(); //高速
if(SD_Type) return 0;
else if(r1) return r1;
return 0xaa;         //其他错误
}
```

该函数先设置与 SD 相关的 I/O 口及 SPI 初始化，然后发送 CMD0，进入 IDLE 状态，并设置 SD 卡为 SPI 模式通信，然后判断 SD 卡类型，完成 SD 卡的初始化，注意该函数调用的 SD_SPI_Init 等函数，实际是对 SPI2 的相关函数进行了一层封装，方便移植。另外一个要介绍的函数是 SD_ReadDisk，该函数用于从 SD 卡读取一个扇区的数据（这里一般为 512 字节），该函数代码如下：

```
//读 SD 卡
//buf:数据缓存区
//sector:扇区
//cnt:扇区数
//返回值:0,ok;其他,失败.
u8 SD_ReadDisk(u8*buf,u32 sector,u8 cnt)
{
    u8 r1;
    if(SD_Type!=SD_TYPE_V2HC)sector <<= 9; //转换为字节地址
    if(cnt==1)
    {
        r1=SD_SendCmd(CMD17,sector,0X01); //读命令
        if(r1==0)                         //指令发送成功
        {
            r1=SD_RecvData(buf,512);       //接收 512 个字节
        }
    }
    else
    {
        r1=SD_SendCmd(CMD18,sector,0X01); //连续读命令
        do
        {
            r1=SD_RecvData(buf,512);       //接收 512 个字节
            buf+=512;
        }while(--cnt && r1==0);
        SD_SendCmd(CMD12,0,0X01);        //发送停止命令
    }
    SD_DisSelect();                  //取消片选
}
```



```
    return r1;
}
```

此函数先发送 CMD17 命令，然后读取一个扇区的数据，详细见代码，这里我们就不多介绍了。然后打开 **MMC_SD.H**，该文件主要是一些命令的宏定义以及函数声明，在这里我们设定了 SD 卡的 CS 管脚为 PD2。接下开我们看看主函数里面编写的应用代码，打开 **main.c** 文件，代码如下：

```
int main(void)
{
    u8 key;
    u32 sd_size;
    u8 t=0;
    u8 *buf;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
    LCD_Init();            //初始化 LCD
    KEY_Init();            //初始化按键
    FSMC_SRAM_Init();     //初始化外部 SRAM
    mem_init(SRAMIN);     //初始化内部内存池

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"MALLOC TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/17");
    LCD_ShowString(60,130,200,16,16,"KEY0:Read Sector 0");
    while(SD_Initialize())//检测不到 SD 卡
    {
        LCD_ShowString(60,150,200,16,16,"SD Card Error!");
        delay_ms(500);
        LCD_ShowString(60,150,200,16,16,"Please Check! ");
        delay_ms(500);
        LED0=!LED0;//DS0 闪烁
    }
    POINT_COLOR=BLUE;//设置字体为蓝色
    //检测 SD 卡成功
    LCD_ShowString(60,150,200,16,16,"SD Card OK      ");
    LCD_ShowString(60,170,200,16,16,"SD Card Size:      MB");
    sd_size=SD_GetSectorCount();//得到扇区数
    LCD_ShowNum(164,170,sd_size>>11,5,16);//显示 SD 卡容量
    while(1)
    {
```



```
key=KEY_Scan(0);
if(key==KEY_RIGHT)//KEY0 按下了
{
    buf=mymalloc(0,512);      //申请内存
    if(SD_ReadDisk(buf,0,1)==0) //读取 0 扇区的内容
    {
        LCD_ShowString(60,190,200,16,16,"USART1 Sending Data... ");
        printf("SECTOR 0 DATA:\r\n");
        for(sd_size=0;sd_size<512;sd_size++)printf("%x ",buf[sd_size]);//打印
                                         0 扇区数据
        printf("\r\nDATA ENDED\r\n");
        LCD_ShowString(60,190,200,16,16,"USART1 Send Data Over!");
    }
    myfree(0,buf);//释放内存
}
t++;
delay_ms(10);
if(t==20)
{
    LED0=!LED0;
    t=0;
}
}
```

这里我们通过 SD_GetSectorCount 函数来得到 SD 卡的扇区数，间接得到 SD 卡容量，然后在液晶上显示出来，接着我们通过按键 KEY0 控制读取 SD 卡的扇区 0，然后把读到的数据通过串口打印出来。这里，我们对上一章学过的内存管理小试牛刀，稍微用了下，以后我们会尽量使用内存管理来设计。

44.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 44.4.1 所示的内容（默认 SD 卡已经接上了）：



图 44.4.1 程序运行效果图

打开串口调试助手，按下 KEY0 就可以看到从开发板发回来的数据了，如图 44.4.2 所示：

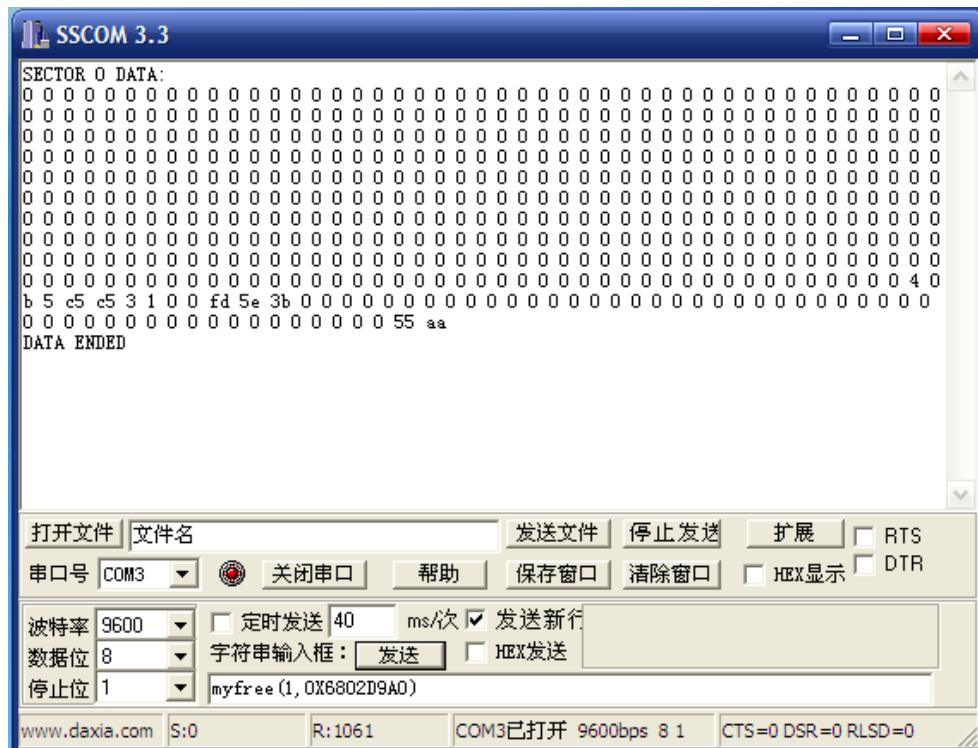


图 44.4.2 串口收到的 SD 卡扇区 0 内容

这里请大家注意，不同的 SD 卡，读出来的扇区 0 是不尽相同的，所以不要因为你读出来的数据和图 44.4.2 不同而感到惊讶。



第四十五章 FATFS 实验

上一章，我们学习了 SD 卡的使用，不过仅仅是简单的实现读扇区而已，真正要好好应用 SD 卡，必须使用文件系统管理，本章，我们将使用 FATFS 来管理 SD 卡，实现 SD 卡文件的读写等基本功能。本章分为如下几个部分：

45.1 FATFS 简介

45.2 硬件设计

45.3 软件设计

45.4 下载验证



45.1 FATFS 简介

FATFS 是一个完全免费开源的 FAT 文件系统模块，专门为小型的嵌入式系统而设计。它完全用标准 C 语言编写，所以具有良好的硬件平台独立性，可以移植到 8051、PIC、AVR、SH、Z80、H8、ARM 等系列单片机上而只需做简单的修改。它支持 FAT12、FAT16 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读 / 写，并特别对 8 位单片机和 16 位单片机做了优化。

FATFS 的特点有：

- Windows 兼容的 FAT 文件系统（支持 FAT12/FAT16/FAT32）
- 与平台无关，移植简单
- 代码量少、效率高
- 多种配置选项
 - ◆ 支持多卷（物理驱动器或分区，最多 10 个卷）
 - ◆ 多个 ANSI/OEM 代码页包括 DBCS
 - ◆ 支持长文件名、ANSI/OEM 或 Unicode
 - ◆ 支持 RTOS
 - ◆ 支持多种扇区大小
 - ◆ 只读、最小化的 API 和 I/O 缓冲区等

FATFS 的这些特点，加上免费、开源的原则，使得 FATFS 应用非常广泛。FATFS 模块的层次结构如图 45.1.1 所示：

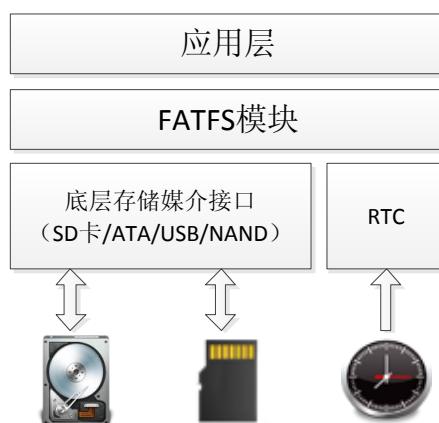


图 45.1.1 FATFS 层次结构图

最顶层是应用层，使用者无需理会 FATFS 的内部结构和复杂的 FAT 协议，只需要调用 FATFS 模块提供给用户的一系列应用接口函数，如 `f_open`, `f_read`, `f_write` 和 `f_close` 等，就可以像在 PC 上读 / 写文件那样简单。

中间层 FATFS 模块，实现了 FAT 文件读 / 写协议。FATFS 模块提供的是 `ff.c` 和 `ff.h`。除非有必要，使用者一般不用修改，使用时将头文件直接包含进去即可。

需要我们编写移植代码的是 FATFS 模块提供的底层接口，它包括存储媒介读 / 写接口(`disk I/O`) 和供给文件创建修改时间的实时时钟。

FATFS 的源码，大家可以在：http://elm-chan.org/fsw/ff/00index_e.html 这个网站下载到，目前最新版本为 R0.09a。本章我们就使用最新版本的 FATFS 作为介绍，下载最新版本的 FATFS 软件包，解压后可以得到两个文件夹：`doc` 和 `src`。`doc` 里面主要是对 FATFS 的介绍，而 `src` 里



面才是我们需要的源码。

其中，与平台无关的是：

ffconf.h	FATFS 模块配置文件
ff.h	FATFS 和应用模块公用的包含文件
ff.c	FATFS 模块
diskio.h	FATFS 和 disk I/O 模块公用的包含文件
interger.h	数据类型定义
option	可选的外部功能（比如支持中文等）

与平台相关的代码（需要用户提供）是：

diskio.c	FATFS 和 disk I/O 模块接口层文件
----------	--------------------------

FATFS 模块在移植的时候，我们一般只需要修改 2 个文件，即 ffconf.h 和 diskio.c。FATFS 模块的所有配置项都是存放在 ffconf.h 里面，我们可以通过配置里面的一些选项，来满足自己的需求。接下来我们介绍几个重要的配置选项。

1) _FS_TINY。这个选项在 R0.07 版本中开始出现，之前的版本都是以独立的 C 文件出现（FATFS 和 Tiny FATFS），有了这个选项之后，两者整合在一起了，使用起来更方便。我们使用 FATFS，所以把这个选项定义为 0 即可。

2) _FS_READONLY。这个用来配置是不是只读，本章我们需要读写都用，所以这里设置为 0 即可。

3) _USE_STRFUNC。这个用来设置是否支持字符串类操作，比如 f_putc, f_puts 等，本章我们需要用到，故设置这里为 1。

4) _USE_MKFS。这个用来定时是否使能格式化，本章需要用到，所以设置这里为 1。

5) _USE_FASTSEEK。这个用来使能快速定位，我们设置为 1，使能快速定位。

6) _CODE_PAGE。这个用于设置语言类型，包括很多选项（见 FATFS 官网说明），我们这里设置为 936，即简体中文（GBK 码，需要 c936.c 文件支持，该文件在 option 文件夹）。

7) _USE_LFN。该选项用于设置是否支持长文件名（还需要_CODE_PAGE 支持），取值范围为 0~3。0，表示不支持长文件名，1~3 是支持长文件名，但是存储地方不一样，我们选择使用 3，通过 ff_malloc 函数来动态分配长文件名的存储区域。

8) _VOLUMES。用于设置 FATFS 支持的逻辑设备数目，我们设置为 2，即支持 2 个设备。

9) _MAX_SS。扇区缓冲的最大值，一般设置为 512。

其他配置项，我们这里就不一一介绍了，FATFS 的说明文档里面有很详细的介绍，大家自己阅读即可。下面我们来讲讲 FATFS 的移植，FATFS 的移植主要分为 3 步：

① 数据类型：在 interger.h 里面去定义好数据的类型。这里需要了解你用的编译器的数据类型，并根据编译器定义好数据类型。

② 配置：通过 ffconf.h 配置 FATFS 的相关功能，以满足你的需要。

③ 函数编写：打开 diskio.c，进行底层驱动编写，一般需要编写 6 个接口函数，如图 45.1.2 所示：

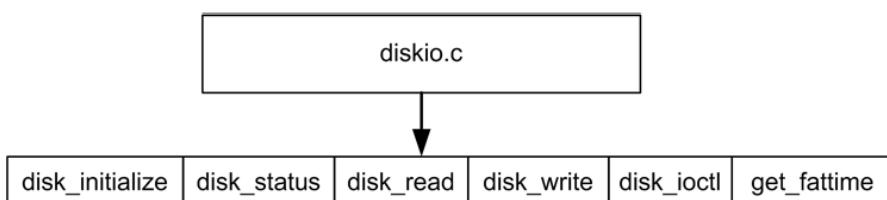


图 45.1.2 diskio 需要实现的函数



通过以上三步，我们即可完成对 FATFS 的移植。

第一步，我们使用的是 MDK3.80a 编译器，器数据类型和 integer.h 里面定义的一致，所以此步，我们不需要做任何改动。

第二步，关于 ffconf.h 里面的相关配置，我们在前面已经有介绍（之前介绍的 9 个配置），我们将对应配置修改为我们介绍时候的值即可，其他的配置用默认配置。

第三步，因为 FATFS 模块完全与磁盘 I/O 层分开，因此需要下面的函数来实现底层物理磁盘的读写与获取当前时间。底层磁盘 I/O 模块并不是 FATFS 的一部分，并且必须由用户提供。这些函数一般有 6 个，在 diskio.c 里面。

首先是 disk_initialize 函数，该函数介绍如图 45.1.3 所示：

函数名称	disk_initialize	
函数原型	DSTATUS disk_initialize(BYTE Drive)	
功能描述	初始化磁盘驱动器	
函数参数	Drive：指定要初始化的逻辑驱动器号，即盘符，应当取值 0~9	
返回值	函数返回一个磁盘状态作为结果，对于磁盘状态的细节信息，请参考 disk_status 函数	
所在文件	ff.c	
示例	disk_initialize(0);	/* 初始化驱动器 0 */
注意事项	disk_initialize 函数初始化一个逻辑驱动器为读/写做准备，函数成功时，返回值的 STA_NOINIT 标志被清零； 应用程序不应调用此函数，否则卷上的 FAT 结构可能会损坏； 如果需要重新初始化文件系统，可使用 f_mount 函数； 在 FatFs 模块上卷注册处理时调用该函数可控制设备的改变； 此函数在 FatFs 挂在卷时调用，应用程序不应该在 FatFs 活动时使用此函数	

图 45.1.3 disk_initialize 函数介绍

第二个函数是 disk_status 函数，该函数介绍如图 45.1.4 所示：

函数名称	disk_status	
函数原型	DSTATUS disk_status(BYTE Drive)	
功能描述	返回当前磁盘驱动器的状态	
函数参数	Drive：指定要确认的逻辑驱动器号，即盘符，应当取值 0~9	
返回值	磁盘状态返回下列标志的组合，FatFs 只使用 STA_NOINIT 和 STA_PROTECTED STA_NOINIT： 表明磁盘驱动未初始化，下面列出了产生该标志置位或清零的原因： 置位：系统复位，磁盘被移除和磁盘初始化函数失败； 清零：磁盘初始化函数成功 STA_NODISK： 表明驱动器中没有设备，安装磁盘驱动器后总为 0 STA_PROTECTED： 表明设备被写保护，不支持写保护的设备总为 0，当 STA_NODISK 置位时非法	
所在文件	ff.c	
示例	disk_status(0);	/* 获取驱动器 0 的状态 */

图 45.1.4 disk_status 函数介绍

第三个函数是 disk_read 函数，该函数介绍如图 45.1.5 所示：



函数名称	disk_read
函数原型	DRESULT disk_read(BYTE Drive, BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount)
功能描述	从磁盘驱动器上读取扇区
函数参数	<p>Drive: 指定逻辑驱动器号, 即盘符, 应当取值 0~9 Buffer: 指向存储读取数据字节数组的指针, 需要为所读取字节数的大小, 扇区统计的扇区大小是需要的 注: FatFs 指定的内存地址并不总是字对齐的, 如果硬件不支持不对齐的数据传输, 函数里需要进行处理 SectorNumber: 指定起始扇区的逻辑块 (LBA) 上的地址 SectorCount: 指定要读取的扇区数, 取值 1~128</p>
返回值	RES_OK(0): 函数成功 RES_ERROR: 读操作期间产生了任何错误且不能恢复它 RES_PARERR: 非法参数 RES_NOTRDY: 磁盘驱动器没有初始化
所在文件	ff.c

图 45.1.5 disk_read 函数介绍

第四个函数是 disk_write 函数, 该函数介绍如图 45.1.6 所示:

函数名称	disk_write
函数原型	DRESULT disk_write(BYTE Drive, const BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount)
功能描述	向磁盘写入一个或多个扇区
函数参数	<p>Drive: 指定逻辑驱动器号, 即盘符, 应当取值 0~9 Buffer: 指向要写入字节数组的指针, 注: FatFs 指定的内存地址并不总是字对齐的, 如果硬件不支持不对齐的数据传输, 函数里需要进行处理 SectorNumber: 指定起始扇区的逻辑块 (LBA) 上的地址 SectorNumber: 指定要写入的扇区数, 取值 1~128</p>
返回值	RES_OK(0): 函数成功 RES_ERROR: 读操作期间产生了任何错误且不能恢复它 RES_WRPRT: 媒体被写保护 RES_PARERR: 非法参数 RES_NOTRDY: 磁盘驱动器没有初始化
所在文件	ff.c
注意事项	只读配置中不需要此函数

图 45.1.6 disk_write 函数介绍

第五个函数是 disk_ioctl 函数, 该函数介绍如图 45.1.7 所示:



函数名称	disk_ioctl
函数原型	DRESULT disk_ioctl(BYTE Drive, BYTE Command, void* Buffer)
功能描述	控制设备指定特性和除了读/写外的杂项功能
函数参数	<p>Drive: 指定逻辑驱动器号, 即盘符, 应当取值 0~9 Command: 指定命令代码 Buffer: 指向参数缓冲区的指针, 取决于命令代码, 不使用时, 指定一个 NULL 指针</p>
返回值	<p>RES_OK(0): 函数成功 RES_ERROR: 读操作期间产生了任何错误且不能恢复它 RES_PARERR: 非法参数 RES_NOTRDY: 磁盘驱动器没有初始化</p>
所在文件	ff.c
注意事项	<p>CTRL_SYNC: 确保磁盘驱动器已经完成了写处理, 当磁盘 I/O 有一个写回缓存, 立即刷新原扇区, 只读配置下不适用此命令 GET_SECTOR_SIZE: 返回磁盘的扇区大小, 只用于 f_mkfs() GET_SECTOR_COUNT: 返回可利用的扇区数, _MAX_SS >= 1024 时可用 GET_BLOCK_SIZE: 获取擦除块大小, 只用于 f_mkfs() CTRL_ERASE_SECTOR: 强制擦除一块的扇区, _USE_ERASE >0 时可用</p>

图 45.1.7 disk_ioctl 函数介绍

最后一个函数是 get_fattime 函数, 该函数介绍如图 45.1.8 所示:

函数名称	get_fattime
函数原型	DWORD get_fattime()
功能描述	获取当前时间
函数参数	无
返回值	<p>当前时间以双字值封装返回, 位域如下: bit31:25 年 (0~127) (从 1980 开始) bit24:21 月 (1~12) bit20:16 日 (1~31) bit15:11 小时 (0~23) bit10:5 分钟 (0~59) bit4:0 秒 (0~29)</p>
所在文件	ff.c
注意事项	<p>get_fattime 函数必须返回一个合法的时间即使系统不支持实时时钟, 如果返回 0, 文件没有一个合法的时间; 只读配置下无需此函数</p>

图 45.1.8 get_fattime 函数介绍

以上六个函数, 我们将在软件设计部分一一实现。通过以上 3 个步骤, 我们就完成了对 FATFS 的移植, 就可以在我们的代码里面使用 FATFS 了。

FATFS 提供了很多 API 函数, 这些函数 FATFS 的自带介绍文件里面都有详细的介绍(包括参考代码), 我们这里就不多说了。这里需要注意的是, 在使用 FATFS 的时候, 必须先通过 f_mount 函数注册一个工作区, 才能开始后续 API 的使用, 关于 FATFS 的介绍, 我们就介绍到这里。大家可以通过 FATFS 自带的介绍文件进一步了解和熟悉 FATFS 的使用。

45.2 硬件设计

本章实验功能简介：开机的时候先初始化 SD 卡，初始化成功之后，注册两个工作区（一个给 SD 卡用，一个给 SPI FLASH 用），然后获取 SD 卡的容量和剩余空间，并显示在 LCD 模块上，最后等待 USMART 输入指令进行各项测试。本实验通过 DS0 指示程序运行状态。

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 串口
- 3) TFTLCD 模块
- 4) SD 卡接口
- 5) SD 卡
- 6) SPI FLASH

这些，我们在之前都已经介绍过，如有不清楚，请参考之前内容。

45.3 软件设计

打开我们的 FATFS 实验工程可以看到，我们将 FATFS 部分单独做一个分组，在工程目录下新建一个 FATFS 的文件夹，然后将 FATFS R0.09a 程序包解压到该文件夹下。同时，我们在 FATFS 文件夹里面新建一个 exfun 的文件夹，用于存放我们针对 FATFS 做的一些扩展代码。设计完如图 45.3.1 所示：

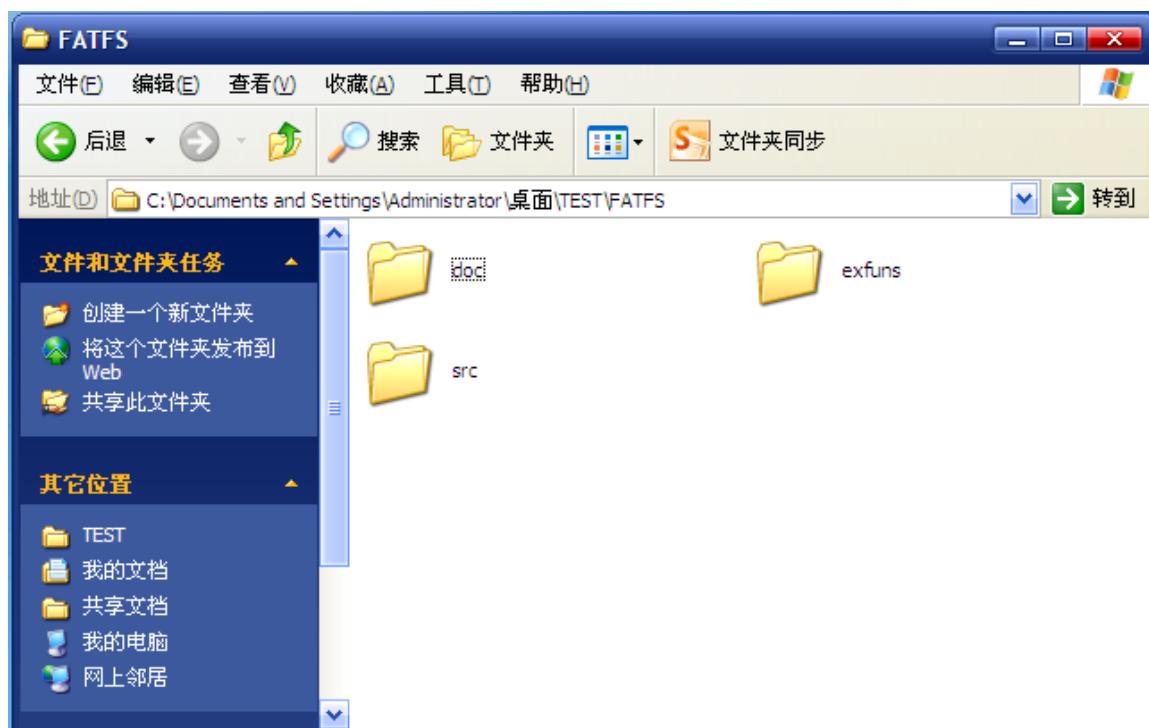


图 45.3.1 FATFS 文件夹子目录

下面我们将对工程中 FATFS 部分的代码进行讲解。

打开 diskio.c，代码如下：

```
#include "mmc_sd.h"
#include "diskio.h"
#include "flash.h"
```



```
#include "malloc.h"
#define SD_CARD 0 //SD 卡,卷标为 0
#define EX_FLASH 1 //外部 flash,卷标为 1
#define FLASH_SECTOR_SIZE 512
//对于 W25Q64
//前 6M 字节给 fatfs 用,6M 字节后~6M+500K 给用户用,6M+500K 以后,用于存放字库,
//字库占用 1.5M.
u16 FLASH_SECTOR_COUNT=2048*6; //6M 字节,默认为 W25Q64
#define FLASH_BLOCK_SIZE 8 //每个 BLOCK 有 8 个扇区
//初始化磁盘
DSTATUS disk_initialize (
    BYTE drv /* Physical drive nmuber (0..) */
)
{
    u8 res=0;
    switch(drv)
    {
        case SD_CARD://SD 卡
            res = SD_Initialize(); //SD_Initialize()
            if(res)//STM32 SPI 的 bug,在 sd 卡操作失败的时候如果不执行下面的语句,
            { //可能导致 SPI 读写异常
                SD_SPI_SpeedLow();
                SD_SPI_ReadWriteByte(0xff); //提供额外的 8 个时钟
                SD_SPI_SpeedHigh();
            }
            break;
        case EX_FLASH://外部 flash
            SPI_Flash_Init();
            if(SPI_FLASH_TYPE==W25Q64)FLASH_SECTOR_COUNT=2048*6;
            else FLASH_SECTOR_COUNT=2048*2; //其他
            break;
        default:
            res=1;
    }
    if(res) return STA_NOINIT;
    else return 0; //初始化成功
}
//获得磁盘状态
DSTATUS disk_status (
    BYTE drv /* Physical drive nmuber (0..) */
)
{
    return 0;
```



```
{  
    //读扇区  
    //drv:磁盘编号 0~9  
    //*buff:数据接收缓冲首地址  
    //sector:扇区地址  
    //count:需要读取的扇区数  
    DRESULT disk_read (  
        BYTE drv,          /* Physical drive nmuber (0..) */  
        BYTE *buff,         /* Data buffer to store read data */  
        DWORD sector,      /* Sector address (LBA) */  
        BYTE count         /* Number of sectors to read (1..255) */  
    )  
    {  
        u8 res=0;  
        if (!count) return RES_PARERR; //count 不能等于 0, 否则返回参数错误  
        switch(drv)  
        {  
            case SD_CARD://SD 卡  
                res=SD_ReadDisk(buff,sector,count);  
                if(res) //STM32 SPI 的 bug,在 sd 卡操作失败的时候如果不执行下面的语句,  
                { //可能导致 SPI 读写异常  
                    SD_SPI_SpeedLow();  
                    SD_SPI_ReadWriteByte(0xff); //提供额外的 8 个时钟  
                    SD_SPI_SpeedHigh();  
                }  
                break;  
            case EX_FLASH://外部 flash  
                for(;count>0;count--)  
                {  
                    SPI_Flash_Read(buff,sector*FLASH_SECTOR_SIZE,FLASH_SECTOR_SIZE);  
                    sector++; buff+=FLASH_SECTOR_SIZE;  
                }  
                res=0;  
                break;  
            default:  
                res=1;  
        }  
        //处理返回值, 将 SPI_SD_driver.c 的返回值转成 ff.c 的返回值  
        if(res==0x00) return RES_OK;  
        else return RES_ERROR;  
    }  
    //写扇区  
    //drv:磁盘编号 0~9
```



```
/*buff:发送数据首地址
//sector:扇区地址
//count:需要写入的扇区数
#if _READONLY == 0
DRESULT disk_write (
    BYTE drv,           /* Physical drive nmuber (0..) */
    const BYTE *buff,    /* Data to be written */
    DWORD sector,       /* Sector address (LBA) */
    BYTE count          /* Number of sectors to write (1..255) */

)
{
    u8 res=0;
    if (!count) return RES_PARERR; //count 不能等于 0, 否则返回参数错误
    switch(drv)
    {
        case SD_CARD://SD 卡
            res=SD_WriteDisk((u8*)buff,sector,count);
            break;
        case EX_FLASH://外部 flash
            for(;count>0;count--)
            {
                SPI_Flash_Write((u8*)buff,sector*FLASH_SECTOR_SIZE,
                    FLASH_SECTOR_SIZE);
                sector++;
                buff+=FLASH_SECTOR_SIZE;
            }
            res=0;
            break;
        default:
            res=1;
    }
    //处理返回值, 将 SPI_SD_driver.c 的返回值转成 ff.c 的返回值
    if(res == 0x00) return RES_OK;
    else return RES_ERROR;
}
#endif /* _READONLY */
//其他表参数的获得
//drv:磁盘编号 0~9
//ctrl:控制代码
/*buff:发送/接收缓冲区指针
DRESULT disk_ioctl (
    BYTE drv,           /* Physical drive nmuber (0..) */
    BYTE ctrl,          /* Control code */
```



```
void *buff      /* Buffer to send/receive control data */  
)  
{  
    DRESULT res;  
    if(drv==SD_CARD)//SD 卡  
    {  
        switch(ctrl)  
        {  
            case CTRL_SYNC:  
                SD_CS=0;  
                if(SD_WaitReady()==0)res = RES_OK;  
                else res = RES_ERROR;  
                SD_CS=1;  
                break;  
            case GET_SECTOR_SIZE:  
                *(WORD*)buff = 512; res = RES_OK;  
                break;  
            case GET_BLOCK_SIZE:  
                *(WORD*)buff = 8; res = RES_OK;  
                break;  
            case GET_SECTOR_COUNT:  
                *(DWORD*)buff = SD_GetSectorCount();res = RES_OK;  
                break;  
            default:  
                res = RES_PARERR;  
                break;  
        }  
    }else if(drv==EX_FLASH) //外部 FLASH  
    {  
        switch(ctrl)  
        {  
            case CTRL_SYNC:  
                res = RES_OK;  
                break;  
            case GET_SECTOR_SIZE:  
                *(WORD*)buff = FLASH_SECTOR_SIZE;  
                res = RES_OK;  
                break;  
            case GET_BLOCK_SIZE:  
                *(WORD*)buff = FLASH_BLOCK_SIZE;  
                res = RES_OK;  
                break;  
            case GET_SECTOR_COUNT:
```



```
*(DWORD*)buff = FLASH_SECTOR_COUNT;
res = RES_OK;
break;
default:
    res = RES_PARERR;
    break;
}
}else res=RES_ERROR;//其他的不支持
return res;
}
//获得时间
//User defined function to give a current time to fatfs module      */
//31-25: Year(0-127 org.1980), 24-21: Month(1-12), 20-16: Day(1-31) */
//15-11: Hour(0-23), 10-5: Minute(0-59), 4-0: Second(0-29 *2) */
DWORD get_fattime (void)
{
    return 0;
}
//动态分配内存
void *ff_malloc (UINT size)
{
    return (void*)mymalloc(SRAMIN,size);
}
//释放内存
void ff_free (void* mf)
{
    myfree(SRAMIN,mf);
}
```

该函数实现了我们 45.1 节提到的 6 个函数，同时因为在 ffconf.h 里面设置对长文件名的支持为方法 3，所以必须实现 ff_malloc 和 ff_free 这两个函数。本章，我们用 FATFS 管理了 2 个磁盘：SD 卡和 SPI FLASH。SD 卡比较好说，但是 SPI FLASH，因为其扇区是 4K 字节大小，我们为了方便设计，强制将其扇区定义为 512 字节，这样带来的好处就是设计使用相对简单，坏处就是擦除次数大增，所以不要随便往 SPI FLASH 里面写数据，非必要最好别写，如果频繁写的话，很容易将 SPI FLASH 写坏。

保存 diskio.c，然后打开 ffconf.h，修改相关配置，并保存，此部分就不贴代码了，请大家参考光盘源码。

前面提到，我们在 FATFS 文件夹下还新建了一个 exfun 的文件夹，该文件夹用于保存一些 FATFS 一些针对 FATFS 的扩展代码，本章，我们编写了 4 个文件，分别是：exfun.c、exfun.h、fattester.c 和 fattester.h。其中 exfun.c 主要定义了一些全局变量，方便 FATFS 的使用，同时实现了磁盘容量获取等函数。而 fattester.c 文件则主要是为了测试 FATFS 用，因为 FATFS 的很多函数无法直接通过 USMART 调用，所以我们在 fattester.c 里面对这些函数进行了一次再封装，使得可以通过 USMART 调用。这几个文件的代码，我们就不贴出来了，请大家参考光盘源



码，我们将 exfun. c 和 fattester. c 加入 FATFS 组下，同时将 exfun. c 文件夹加入头文件包含路径。

然后，我们打开 main. c，修改 main 函数如下：

```
int main(void)
{
    u32 total,free;
    u8 t=0;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //初始化 LCD
    KEY_Init();              //初始化按键
    exfun. init();           //为 fatfs 相关变量申请内存
    usmart_dev.init(72);
    mem_init(SRAMIN);      //初始化内部内存池
    POINT_COLOR=RED; //设置字体为红色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"FATFS TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"Use USMART for test");
    LCD_ShowString(60,130,200,16,16,"2012/9/18");
    while(SD_Initialize())           //检测 SD 卡
    {
        LCD_ShowString(60,150,200,16,16,"SD Card Error!");
        delay_ms(200);
        LCD_Fill(60,150,240,150+16,WHITE); //清除显示
        delay_ms(200);
        LED0=!LED0; //DS0 闪烁
    }
    exfun. init();           //为 fatfs 相关变量申请内存
    f_mount(0,fs[0]);        //挂载 SD 卡
    f_mount(1,fs[1]);        //挂载 FLASH.
    while(exf_getfree("0",&total,&free)) //得到 SD 卡的总容量和剩余容量
    {
        LCD_ShowString(60,150,200,16,16,"Fatfs Error!");
        delay_ms(200);
        LCD_Fill(60,150,240,150+16,WHITE); //清除显示
        delay_ms(200);
        LED0=!LED0; //DS0 闪烁
    }
    POINT_COLOR=BLUE; //设置字体为蓝色
    LCD_ShowString(60,150,200,16,16,"FATFS OK!");
```



```
LCD_ShowString(60,170,200,16,16,"SD Total Size:      MB");  
LCD_ShowString(60,190,200,16,16,"SD  Free Size:      MB");  
LCD_ShowNum(172,170,total>>10,5,16);      //显示 SD 卡总容量 MB  
LCD_ShowNum(172,190,free>>10,5,16);      //显示 SD 卡剩余容量 MB  
while(1)  
{  
    t++;  
    delay_ms(200);  
    LED0=!LED0;  
}  
}
```

在 main 函数里面，我们为 SD 卡和 FLASH 都注册了工作区（挂载），在初始化 SD 卡并显示其容量信息后，进入死循环，等待 USMART 测试。

最后，我们在 usmart_config.c 里面的 usmart_nametab 数组添加如下内容：

```
(void*)mf_mount,"u8 mf_mount(u8 drv)",  
(void*)mf_open,"u8 mf_open(u8*path,u8 mode)",  
(void*)mf_close,"u8 mf_close(void)",  
(void*)mf_read,"u8 mf_read(u16 len)",  
(void*)mf_write,"u8 mf_write(u8*dat,u16 len)",  
(void*)mf_opendir,"u8 mf_opendir(u8* path)",  
(void*)mf_readdir,"u8 mf_readdir(void)",  
(void*)mf_scan_files,"u8 mf_scan_files(u8 * path)",  
(void*)mf_showfree,"u32 mf_showfree(u8 *drv)",  
(void*)mf_lseek,"u8 mf_lseek(u32 offset)",  
(void*)mf_tell,"u32 mf_tell(void)",  
(void*)mf_size,"u32 mf_size(void)",  
(void*)mf_mkdir,"u8 mf_mkdir(u8*pname)",  
(void*)mf_fmkfs,"u8 mf_fmkfs(u8 drv,u8 mode,u16 au)",  
(void*)mf_unlink,"u8 mf_unlink(u8 *pname)",  
(void*)mf_rename,"u8 mf_rename(u8 *oldname,u8* newname)",  
(void*)mf_gets,"void mf_gets(u16 size)",  
(void*)mf_putc,"u8 mf_putc(u8 c)",  
(void*)mf_puts,"u8 mf_puts(u8*c)",
```

这些函数均是在 fattester.c 里面实现，通过调用这些函数，即可实现对 FATFS 对应 API 函数的测试。至此，软件设计部分就结束了。

45.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 显示如图 45.4.1 所示的内容（默认 SD 卡已经接上了）：



图 45.4.1 程序运行效果图

打开串口调试助手，我们就可以串口调用前面添加的各种 FATFS 测试函数了，比如我们输入 `mf_scan_files("0:")`，即可扫描 SD 卡根目录的所有文件，如图 45.4.2 所示：

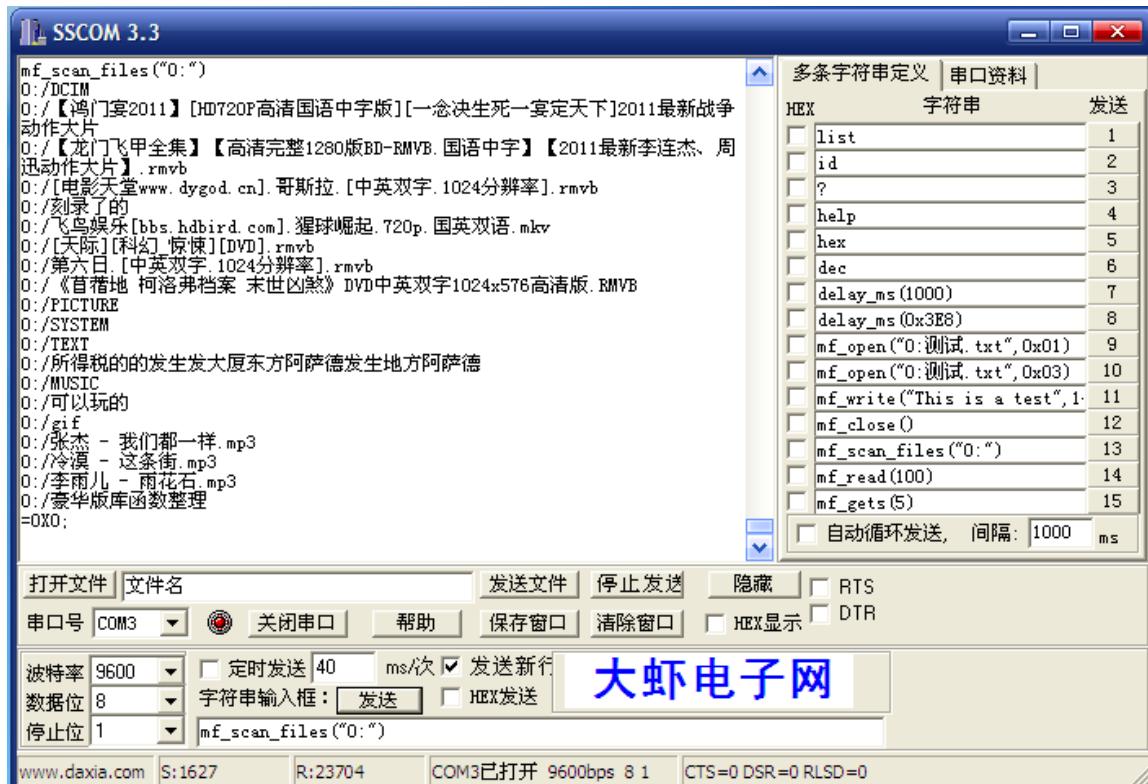


图 45.4.2 扫描 SD 卡根目录所有文件

其他函数的测试，用类似的办法即可实现。注意这里 0 代表 SD 卡，1 代表 SPI FLASH。另外，提醒大家，`mf_unlink` 函数，在删除文件夹的时候，必须保证文件夹是空的，才可以正常删除，否则不能删除。



第四十六章 汉字显示实验

汉字显示在很多单片机系统都需要用到，少则几个字，多则整个汉字库的支持，更有甚者还要支持多国字库，那就更麻烦了。本章，我们将向大家介绍，如何用 STM32 控制 LCD 显示汉字。在本章中，我们将使用外部 FLASH 来存储字库，并可以通过 SD 卡更新字库。STM32 读取存在 FLASH 里面的字库，然后将汉字显示在 LCD 上面。本章分为如下几个部分：

- 46.1 汉字显示原理简介
- 46.2 硬件设计
- 46.3 软件设计
- 46.4 下载验证



46.1 汉字显示原理简介

常用的汉字内码系统有 GB2312, GB13000, GBK, BIG5（繁体）等几种，其中 GB2312 支持的汉字仅有几千个，很多时候不够用，而 GBK 内码不仅完全兼容 GB2312，还支持了繁体字，总汉字数有 2 万多个，完全能满足我们一般应用的要求。

本实例我们将制作一个 GBK 字库，制作好的字库放在 SD 卡里面，然后通过 SD 卡，将字库文件复制到外部 FLASH 芯片 W25Q64 里，这样，W25Q64 就相当于一个汉字字库芯片了。

汉字在液晶上的显示原理与前面显示字符的是一样的。汉字在液晶上的显示其实就是一些点的显示与不显示，这就相当于我们的笔一样，有笔经过的地方就画出来，没经过的地方就不画。所以要显示汉字，我们首先要知道汉字的点阵数据，这些数据可以由专门的软件来生成。只要知道了一个汉字点阵的生成方法，那么我们在程序里面就可以把这个点阵数据解析成一个汉字。

知道显示了一个汉字，就可以推及整个汉字库了。汉字在各种文件里面的存储不是以点阵数据的形式存储的（否则那占用的空间就太大了），而是以内码的形式存储的，就是 GB2312/GBK/BIG5 等这几种的一种，每个汉字对应着一个内码，在知道了内码之后再去字库里面查找这个汉字的点阵数据，然后在液晶上显示出来。这个过程我们是看不到，但是计算机是要去执行的。

单片机要显示汉字也与此类似：汉字内码（GBK/GB2312）→查找点阵库→解析→显示。

所以只要我们有了整个汉字库的点阵，就可以把电脑上的文本信息在单片机上显示出来了。这里我们要解决的最大问题就是制作一个与汉字内码对得上号的汉字点阵库。而且要方便单片机的查找。每个 GBK 码由 2 个字节组成，第一个字节为 0X81~0XFE，第二个字节分为两部分，一是 0X40~0X7E，二是 0X80~0XFE。其中与 GB2312 相同的区域，字完全相同。

我们把第一个字节代表的意义称为区，那么 GBK 里面总共有 126 个区（0XFE-0X81+1），每个区内有 190 个汉字（0XFE-0X80+0X7E-0X40+2），总共就有 $126 \times 190 = 23940$ 个汉字。我们的点阵库只要按照这个编码规则从 0X8140 开始，逐一建立，每个区的点阵大小为每个汉字所用的字节数*190。这样，我们就可以得到在这个字库里面定位汉字的方法：

当 $\text{GBKL} < 0X7F$ 时： $\text{Hp} = ((\text{GBKH}-0x81) * 190 + \text{GBKL}-0X40) * (\text{size} * 2)$;

当 $\text{GBKL} > 0X80$ 时： $\text{Hp} = ((\text{GBKH}-0x81) * 190 + \text{GBKL}-0X41) * (\text{size} * 2)$;

其中 GBKH、GBKL 分别代表 GBK 的第一个字节和第二个字节（也就是高位和低位），size 代表汉字字体的大小（比如 16 字体，12 字体等），Hp 则为对应汉字点阵数据在字库里面的起始地址（假设是从 0 开始存放）。

这样我们只要得到了汉字的 GBK 码，就可以显示这个汉字了。从而实现汉字在液晶上的显示。

上一章，我们提到要用 cc936.c，以支持长文件名，但是 cc936.c 文件里面的两个数组太大了（172KB），直接刷在单片机里面，太占用 flash 了，所以我们必须把这两个数组存放在外部 flash。cc936 里面包含的两个数组 oem2uni 和 uni2oem 存放 unicode 和 gbk 的互相转换对照表，这两个数组很大，这里我们利用 ALIENTEK 提供的一个 C 语言数组转 BIN（二进制）的软件：C2B 转换助手 V1.1.exe，将这两个数组转为 BIN 文件，我们将这两个数组拷贝出来存放为一个新的文本文件，假设为 UNIGBK.TXT，然后用 C2B 转换助手打开这个文本文件，如图 46.1.1 所示：



图 46.1.1 C2B 转换助手

然后点击转换，就可以在当前目录下（文本文件所在目录下）得到一个 UNIGBK.bin 的文件。这样就完成将 C 语言数组转换为.bin 文件，然后只需要将 UNIGBK.bin 保存到外部 FLASH 就实现了该数组的转移。

在 cc936.c 里面，主要是通过 ff_convert 调用这两个数组，实现 UNICODE 和 GBK 的互转，该函数原代码如下：

```

WCHAR ff_convert ( /* Converted code, 0 means conversion error */
    WCHAR src,      /* Character code to be converted */
    UINT   dir       /* 0: Unicode to OEMCP, 1: OEMCP to Unicode */
)
{
    const WCHAR *p;
    WCHAR c;
    int i, n, li, hi;
    if (src < 0x80) { /* ASCII */
        c = src;
    } else {
        if (dir) { /* OEMCP to unicode */
            p = oem2uni;
            hi = sizeof(oem2uni) / 4 - 1;
        } else { /* Unicode to OEMCP */
            p = uni2oem;
            hi = sizeof(uni2oem) / 4 - 1;
        }
        li = 0;
        for (n = 16; n; n--) {
            i = li + (hi - li) / 2;
            if (src == p[i * 2]) break;
        }
    }
}

```



```
    if (src > p[i * 2]) li = i;
    else hi = i;
}
c = n ? p[i * 2 + 1] : 0;
}
return c;
}
```

此段代码，通过二分法（16 阶）在数组里面查找 UNICODE（或 GBK）码对应的 GBK（或 UNICODE）码。当我们将数组存放在外部 flash 的时候，将该函数修改为：

```
WCHAR ff_convert ( /* Converted code, 0 means conversion error */
    WCHAR src,      /* Character code to be converted */
    UINT     dir      /* 0: Unicode to OEMCP, 1: OEMCP to Unicode */
)
{
    WCHAR t[2];
    WCHAR c;
    u32 i, li, hi;
    u16 n;
    u32 gbk2uni_offset=0;
    if (src < 0x80)c = src;//ASCII,直接不用转换.
    else
    {
        if(dir) gbk2uni_offset=ftinfo.ugbksize/2; //GBK 2 UNICODE
        else gbk2uni_offset=0; //UNICODE 2 GBK
        /* Unicode to OEMCP */
        hi=ftinfo.ugbksize/2;//对半开.
        hi =hi / 4 - 1;
        li = 0;
        for (n = 16; n; n--)
        {
            i = li + (hi - li) / 2;
            SPI_Flash_Read((u8*)&t,ftinfo.ugbkaddr+i*4+gbk2uni_offset,4); //读出 4 个字节
            if (src == t[0]) break;
            if (src > t[0])li = i;
            else hi = i;
        }
        c = n ? t[1] : 0;
    }
    return c;
}
```

代码中的 ftinfo.ugbksize 为我们刚刚生成的 UNIGBK.bin 的大小，而 ftinfo.ugbkaddr 是我们存放 UNIGBK.bin 文件的首地址。这里同样采用的是二分法查找，关于 cc936.c 的修改，我们就介绍到这。



字库的生成，我们要用到一款软件，由易木雨软件工作室设计的点阵字库生成器 V3.8。该软件可以在 WINDOWS 系统下生成任意点阵大小的 ASCII、GB2312(简体中文)、GBK(简体中文)、BIG5(繁体中文)、HANGUL(韩文)、SJIS(日文)、Unicode 以及泰文，越南文、俄文、乌克兰文，拉丁文，8859 系列等共二十几种编码的字库，不但支持生成二进制文件格式的文件，也可以生成 BDF 文件，还支持生成图片功能，并支持横向，纵向等多种扫描方式，且扫描方式可以根据用户的需求进行增加。该软件的界面如图 46.1.1 所示：



图 46.1.2 点阵字库生成器默认界面

比如我们要生成 16*16 的 GBK 字库，则选择：936 中文 PRC GBK，字宽和高均选择 16，字体大小选择 12（比较适合），然后模式选择纵向取模方式二（字节高位在前，低位在后），最后点击创建，就可以开始生成我们需要的字库了。具体设置如图 46.1.3 所示：

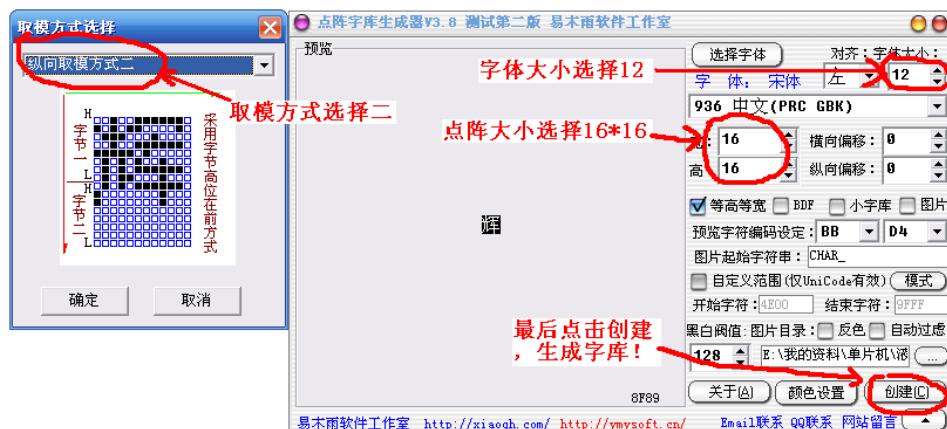


图 46.1.3 生成 GBK16*16 字库的设置方法

这里注意，软件里面的字体大小并不是我们生成点阵的大小，12 字体是 XP 的叫法，我们字体的大小以宽和高的大小来决定！可以简单的这么认为：XP 的 12 字体，基本上就等于 16*16 大小。该软件还可以生成其他很多字库，字体也可选，详细的介绍请看软件自带的《点阵字库生成器说明书》。

本章，我们生成两个字库文件 GBK12.DZK 和 GBK16.DZK，并将后缀名改为.fon（方便识别^_^），备用。关于汉字显示原理，我们就介绍到这。



46.2 硬件设计

本章实验功能简介：开机的时候先检测 W25Q64 中是否已经存在字库，如果存在，则按次序显示汉字(两种字体都显示)。如果没有，则检测 SD 卡和文件系统，并查找 SYSTEM 文件夹下的 FONT 文件夹，在该文件夹内查找 UNIGBK.BIN、GBK12.FON 和 GBK16.FON (这几个文件的由来，我们前面已经介绍了)。在检测到这些文件之后，就开始更新字库，更新完毕才开始显示汉字。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) 串口
- 3) TFTLCD 模块
- 4) SD 卡接口
- 5) SD 卡
- 6) SPI FLASH

这几部分分，在之前的实例中都介绍过了，我们在此就不介绍了。

46.3 软件设计

打开汉字显示实验可以看到，我们在 TEXT 文件夹下新建 fontupd.c、fontupd.h、text.c、text.h 这 4 个文件。并将该文件夹加入了头文件包含路径。

打开 fontupd.c，该文件代码如下：

```
#include "fontupd.h"
#include "ff.h"
#include "flash.h"
#include "lcd.h"
#include "malloc.h"

u32 FONTINFOADDR=(1024*6+500)*1024;//默认是 6M+500K 后开始
//字库信息结构体。
//用来保存字库基本信息，地址，大小等
_font_info ftinfo;
//在 sd 卡中的路径
const u8 *GBK16_SDPATH="0:/SYSTEM/FONT/GBK16.FON"; //GBK16 存放位置
const u8 *GBK12_SDPATH="0:/SYSTEM/FONT/GBK12.FON"; //GBK12 存放位置
const u8 *UNIGBK_SDPATH="0:/SYSTEM/FONT/UNIGBK.BIN";//UNIGBK.BIN 存放位置
//在 25Qxx 中的路径
const u8 *GBK16_25QPATH="1:/SYSTEM/FONT/GBK16.FON"; //GBK12 的存放位置
const u8 *GBK12_25QPATH="1:/SYSTEM/FONT/GBK12.FON"; //GBK12 的存放位置
const u8 *UNIGBK_25QPATH="1:/SYSTEM/FONT/UNIGBK.BIN";//UNIGBK.BIN 存放位置
//显示当前字体更新进度
//x,y:坐标
//size:字体大小
//fsize:整个文件大小
//pos:当前文件指针位置
u32 fupd_prog(u16 x,u16 y,u8 size,u32 fsize,u32 pos)
```



```
{  
    float prog;  
    u8 t=0xFF;  
    prog=(float)pos/fsize;  
    prog*=100;  
    if(t!=prog)  
    {  
        LCD_ShowString(x+3*size/2,y,240,320,size,"%");  
        t=prog;  
        if(t>100)t=100;  
        LCD_ShowNum(x,y,t,3,size); //显示数值  
    }  
    return 0;  
}  
//更新某一个  
//x,y:坐标  
//size:字体大小  
//fxpath:路径  
//fx:更新的内容 0,unibk;1,gbk12;2,gbk16;  
//返回值:0,成功;其他,失败.  
u8 updata_fontx(u16 x,u16 y,u8 size,u8 *fxpath,u8 fx)  
{  
    u32 flashaddr=0;  
    FIL * fftemp;  
    u8 *tempbuf;  
    u8 res;  
    u16 bread;  
    u32 offx=0;  
    u8 rval=0;  
    fftemp=(FIL*)mymalloc(SRAMIN,sizeof(FIL)); //分配内存  
    if(fftemp==NULL)rval=1;  
    tempbuf=mymalloc(SRAMIN,4096); //分配 4096 个字节空间  
    if(tempbuf==NULL)rval=1;  
    res=f_open(fftemp,(const TCHAR*)fxpath,FA_READ);  
    if(res)rval=2;//打开文件失败  
    if(rval==0)  
    {  
        if(fx==0) //更新 UNIBK.BIN  
        {  
            ftinfo.ugbkaddr=FONTINFOADDR+sizeof(ftinfo);  
            //信息头之后, 紧跟 UNIBK 转换码表  
            ftinfo.ugbksize=fftemp->fsize; //UNIBK 大小  
            flashaddr=ftinfo.ugbkaddr;  
        }  
    }  
}
```



```
 }else if(fx==1) //GBK12
 {
 ftinfo.f12addr=ftinfo.ugbkaddr+ftinfo.ugbksize; //UNIGBK 后, 跟 GBK12 字库
 ftinfo.gbk12size=fftemp->fsize; //GBK12 字库大小
 flashaddr=ftinfo.f12addr; //GBK12 的起始地址
 }else //GBK16
 {
 ftinfo.f16addr=ftinfo.f12addr+ftinfo.gbk12size; //GBK12 后, 跟 GBK16 字库
 ftinfo.gkb16size=fftemp->fsize; //GBK16 字库大小
 flashaddr=ftinfo.f16addr; //GBK16 的起始地址
 }
 while(res==FR_OK)//死循环执行
 {
 res=f_read(fftemp,tempbuf,4096,(UINT *)&bread);//读取数据
 if(res!=FR_OK)break; //执行错误
 SPI_Flash_Write(tempbuf,offx+flashaddr,4096); //从 0 开始写入 4096 个数据
 offx+=bread;
 fupd_prog(x,y,size,fftemp->fsize,offx); //进度显示
 if(bread!=4096)break; //读完了.
 }
 f_close(fftemp);
 }
 myfree(SRAMIN,fftemp); //释放内存
 myfree(SRAMIN,tempbuf); //释放内存
 return res;
 }
 //更新字体文件,UNIGBK,GBK12,GBK16 一起更新
 //x,y:提示信息的显示地址
 //size:字体大小
 //提示信息字体大小
 //src:0,从 SD 卡更新.
 // 1,从 25QXX 更新
 //返回值:0,更新成功;
 // 其他,错误代码.
 u8 update_font(u16 x,u16 y,u8 size,u8 src)
 {
 u8 *gbk16_path;
 u8 *gbk12_path;
 u8 *unibk_path;
 u8 res;
 if(src)//从 25qxx 更新
 {
 unibk_path=(u8*)UNIGBK_25QPATH;
```



```
gbk12_path=(u8*)GBK12_25QPATH;
gbk16_path=(u8*)GBK16_25QPATH;
}else//从 sd 卡更新
{
    unibk_path=(u8*)UNIGBK_SDPATH;
    gbk12_path=(u8*)GBK12_SDPATH;
    gbk16_path=(u8*)GBK16_SDPATH;
}
res=0xFF;
ftinfo.fontok=0xFF;
SPI_Flash_Write((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo));
//清除之前字库成功的标志.防止更新到一半重启,导致的字库部分数据丢失.
SPI_Flash_Read((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo));//读出 ftinfo 结构体数据
LCD_ShowString(x,y,240,320,size,"Updating UNIGBK.BIN");
res=updata_fontx(x+20*size/2,y,size,unibk_path,0);           //更新 UNIGBK.BIN
if(res)return 1;
LCD_ShowString(x,y,240,320,size,"Updating GBK12.BIN   ");
res=updata_fontx(x+20*size/2,y,size,gbk12_path,1);           //更新 GBK12.FON
if(res)return 2;
LCD_ShowString(x,y,240,320,size,"Updating GBK16.BIN   ");
res=updata_fontx(x+20*size/2,y,size,gbk16_path,2);           //更新 GBK16.FON
if(res)return 3;
//全部更新好了
ftinfo.fontok=0XAA;
SPI_Flash_Write((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo)); //保存字库信息
return 0;//无错误.
}
//初始化字体
//返回值:0,字库完好.
//      其他,字库丢失
u8 font_init(void)
{
    SPI_Flash_Init();
    FONTINFOADDR=(1024*6+500)*1024; //W25Q64,6M 以后
    ftinfo.ugbkaddr=FONTINFOADDR+25; //UNICODEGBK 表存放首地址固定地址
    SPI_Flash_Read((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo));//读出 ftinfo 结构体数据
    if(ftinfo.fontok!=0XAA)return 1;           //字库错误.
    return 0;
}
```

此部分代码主要用于字库的更新操作（包含 UNIGBK 的转换码表更新），其中 ftinfo 是我们在 fontupd.h 里面定义的一个结构体，用于记录字库首地址及字库大小等信息。因为我们将 W25Q64 的前 6M 字节给 FATFS 管理（用做本地磁盘），然后又预留了 500K 字节给用户自己使用，最后的 1.5M 字节（W25Q64 总共 8M 字节），才是 UNIGBK 码表和字库的存储空间，所以，



我们的存储地址是从 $(1024*6+500)*1024$ 处开始的。最开始的 25 个字节给 ftinfo 用，用于保存 ftinfo 结构体数据。之后开始的是 UNIGBK.bin 的存放地址，之后再是 GBK12.fon 的存放地址，最后存放 GBK16.fon。

接下来我们看看头文件 fontupd.h 的内容：

```
#ifndef __FONTUPD_H__
#define __FONTUPD_H__
#include <stm32f10x_map.h>
//前面 6M 被 fatfs 占用了.
//6M 以后紧跟的 500K 字节,用户可以随便用.
//6M+500K 字节以后的字节,被字库占用了,不能动!
//字体信息保存地址,占 25 个字节,第 1 个字节用于标记字库是否存在.后续每 8 个字节一组,
//分别保存起始地址和文件大小
extern u32 FONTINFOADDR;
//字库信息结构体定义
//用来保存字库基本信息, 地址, 大小等
__packed typedef struct
{
    u8 fontok;           //字库存在标志, 0XAA, 字库正常; 其他, 字库不存在
    u32 ubkbaddr;        //unigbk 的地址
    u32 ubksize;         //unigbk 的大小
    u32 f12addr;         //gbk12 地址
    u32 gbk12size;       //gbk12 的大小
    u32 f16addr;         //gbk16 地址
    u32 gkb16size;       //gbk16 的大小
}__font_info;
extern __font_info ftinfo; //字库信息结构体
u32 fupd_prog(u16 x,u16 y,u8 size,u32 fsize,u32 pos); //显示更新进度
u8 updata_fontx(u16 x,u16 y,u8 size,u8 *xpath,u8 fx); //更新指定字库
u8 update_font(u16 x,u16 y,u8 size,u8 src); //更新全部字库
u8 font_init(void); //初始化字库
#endif
```

这里，我们可以看到 ftinfo 的结构体定义，总共占用 25 个字节，第一个字节用来标识字库是否 OK，其他的用来记录地址和文件大小。打开 text.c 文件，该文件代码如下：

```
#include "sys.h"
#include "fontupd.h"
#include "flash.h"
#include "lcd.h"
#include "text.h"
#include "string.h"
//code 字符指针开始
//从字库中查找出字模
//code 字符串的开始地址, GBK 码
//mat 数据存放地址 size*2 bytes 大小
```



```
void Get_HzMat(unsigned char *code,unsigned char *mat,u8 size)
{
    unsigned char qh,ql;
    unsigned char i;
    unsigned long foffset;
    qh=*code;
    ql=*(++code);
    if(qh<0x81||ql<0x40||ql==0xff||qh==0xff)//非 常用汉字
    {
        for(i=0;i<(size*2);i++)*mat++=0x00;//填充满格
        return; //结束访问
    }
    if(ql<0x7f)ql-=0x40;//注意!
    else ql-=0x41;
    qh-=0x81;
    foffset=((unsigned long)190*qh+ql)*(size*2);//得到字库中的字节偏移量
    if(size==16)SPI_Flash_Read(mat,foffset+ftinfo.f16addr,32);
    else SPI_Flash_Read(mat,foffset+ftinfo.f12addr,24);
}

//显示一个指定大小的汉字
//x,y :汉字的坐标
//font:汉字 GBK 码
//size:字体大小
//mode:0,正常显示,1,叠加显示
void Show_Font(u16 x,u16 y,u8 *font,u8 size,u8 mode)
{
    u8 temp,t,t1;
    u16 y0=y;
    u8 dzk[32];
    u16 tempcolor;
    if(size!=12&&size!=16)return;//不支持的 size
    Get_HzMat(font,dzk,size);//得到相应大小的点阵数据
    if(mode==0)//正常显示
    {
        for(t=0;t<size*2;t++)
        {
            temp=dzk[t];//得到 12 数据
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)LCD_DrawPoint(x,y);
                else
                {
                    tempcolor=POINT_COLOR;
                }
            }
        }
    }
}
```



```
POINT_COLOR=BACK_COLOR;
LCD_DrawPoint(x,y);
POINT_COLOR=tempcolor;//还原
}
temp<=1; y++;
if((y-y0)==size) { y=y0; x++; break; }
}
}
}else//叠加显示
{
for(t=0;t<size*2;t++)
{
temp=dzk[t];//得到 12 数据
for(t1=0;t1<8;t1++)
{
if(temp&0x80)LCD_DrawPoint(x,y);
temp<=1; y++;
if((y-y0)==size) { y=y0; x++; break; }
}
}
}
//在指定位置开始显示一个字符串
//支持自动换行
//(x,y):起始坐标
//width,height:区域
//str :字符串
//size :字体大小
//mode:0,非叠加方式;1,叠加方式
void Show_Str(u16 x,u16 y,u16 width,u16 height,u8*str,u8 size,u8 mode)
{
.....此处代码省略
}
//在指定宽度的中间显示字符串
//如果字符长度超过了 len,则用 Show_Str 显示
//len:指定要显示的宽度
void Show_Str_Mid(u16 x,u16 y,u8*str,u8 size,u8 len)
{
.....//此处代码省略
}
```

此部分代码总共有 4 个函数，我们省略了两个函数（Show_Str_Mid 和 Show_Str）的代码，另外两个函数，Get_HzMat 函数用于获取 GBK 码对应的汉字字库，通过我们 46.1 节介绍的办法，在外部 flash 查找字库，然后返回对应的字库点阵。Show_Font 函数用于在指定地址显示一



个指定大小的汉字，采用的方法和 LCD_ShowChar 所采用的方法一样，都是画点显示，这里就不细说了。保存此部分代码，并把 text.c 文件加入 TEXT 组下。text.h 里面都是一些函数申明，这里我们就不贴出来了，详见光盘本例程源码。

前面提到我们队 cc936.c 文件做了修改，我们将其命名为 mycc936.c，并保存在 exfun 文件夹下，将工程 FATFS 组下的 cc936.c 删除，然后重新添加 mycc936.c 到 FATFS 组下，mycc936.c 的源码就不贴出来了，其实就是在 cc936.c 的基础上去掉了两个大数组，然后对 ff_convert 进行了修改，详见光盘本例程源码。

最后，我们看看文件 main.c 里面 main 函数代码如下：

```
int main(void)
{
    u32 fontcnt;
    u8 i,j;
    u8 fontx[2];//gbk 码
    u8 key,t;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();              //初始化液晶
    KEY_Init();              //按键初始化
    usmart_dev.init(72);     //usmart 初始化
    mem_init(SRAMIN);       //初始化内部内存池

    exfun_init();           //为 fatfs 相关变量申请内存
    f_mount(0,fs[0]);        //挂载 SD 卡
    f_mount(1,fs[1]);        //挂载 FLASH.
    key=KEY_Scan(0);         //按键扫描
    while(font_init()||key==KEY_UP) //检查字库
    {
        UPD:
        LCD_Clear(WHITE);          //清屏
        POINT_COLOR=RED;           //设置字体为红色
        LCD_ShowString(60,50,200,16,16,"Warship STM32");
        while(SD_Initialize())      //检测 SD 卡
        {
            LCD_ShowString(60,70,200,16,16,"SD Card Failed!");
            delay_ms(200);
            LCD_Fill(60,70,200+60,70+16,WHITE);
            delay_ms(200);
        }
        LCD_ShowString(60,70,200,16,16,"SD Card OK");
        LCD_ShowString(60,90,200,16,16,"Font Updating...");  

        key=update_font(20,110,16,0);//从 SD 卡更新
    }
}
```



```
while(key)//更新失败
{
    LCD_ShowString(60,110,200,16,16,"Font Update Failed!");
    delay_ms(200);
    LCD_Fill(20,110,200+20,110+16,WHITE);
    delay_ms(200);
}
LCD_ShowString(60,110,200,16,16,"Font Update Success!");
delay_ms(1500);
LCD_Clear(WHITE);//清屏
}

POINT_COLOR=RED;
Show_Str(60,50,200,16,"战舰 STM32 开发板",16,0);
Show_Str(60,70,200,16,"GBK 字库测试程序",16,0);
Show_Str(60,90,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,110,200,16,"2012 年 9 月 18 日",16,0);
Show_Str(60,130,200,16,"按 KEY0,更新字库",16,0);
POINT_COLOR=BLUE;
Show_Str(60,150,200,16,"内码高字节:",16,0);
Show_Str(60,170,200,16,"内码低字节:",16,0);
Show_Str(60,190,200,16,"对应汉字(16*16)为:",16,0);
Show_Str(60,212,200,12,"对应汉字(12*12)为:",12,0);
Show_Str(60,230,200,16,"汉字计数器:",16,0);
LCD_Fill(60,130,200+60,130+16,WHITE);
while(1)
{
    fontcnt=0;
    for(i=0x81;i<0xff;i++)
    {
        fontx[0]=i;
        LCD_ShowNum(148,150,i,3,16);//显示内码高字节
        for(j=0x40;j<0xfe;j++)
        {
            if(j==0x7f)continue;
            fontcnt++;
            LCD_ShowNum(148,170,j,3,16);//显示内码低字节
            LCD_ShowNum(148,230,fontcnt,5,16);//显示内码低字节
            fontx[1]=j;
            Show_Font(204,190,fontx,16,0);
            Show_Font(168,212,fontx,12,0);
            t=200;
            while(t--)//延时,同时扫描按键
            {

```



```
    delay_ms(1);
    key=KEY_Scan(0);
    if(key==KEY_UP)goto UPD;
}
LED0=!LED0;
}
}
}
```

此部分代码就实现了我们在硬件描述部分所描述的功能，至此整个软件设计就完成了。这节有太多的代码，而且工程也增加了不少，我们来看看工程的截图吧，整个工程截图如图 46.3.1 所示：

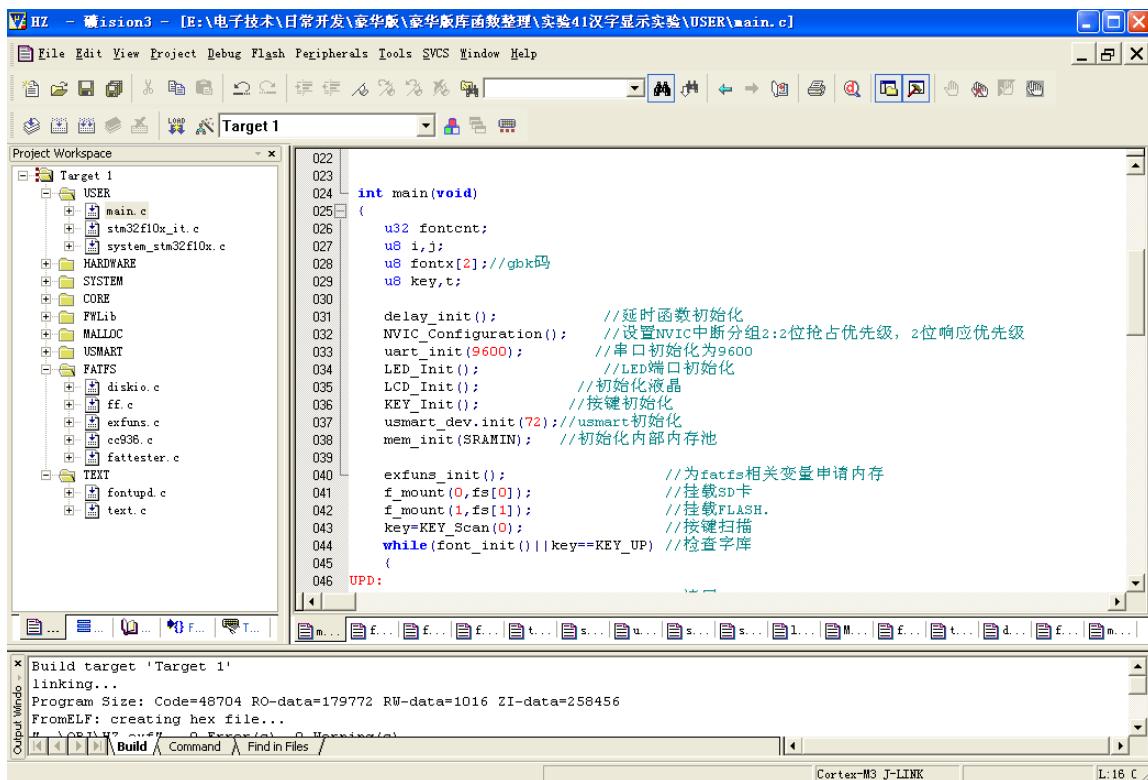


图 46.3.1 工程建截图

46.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 开始显示汉字及汉字内码，如图 46.4.1 所示：

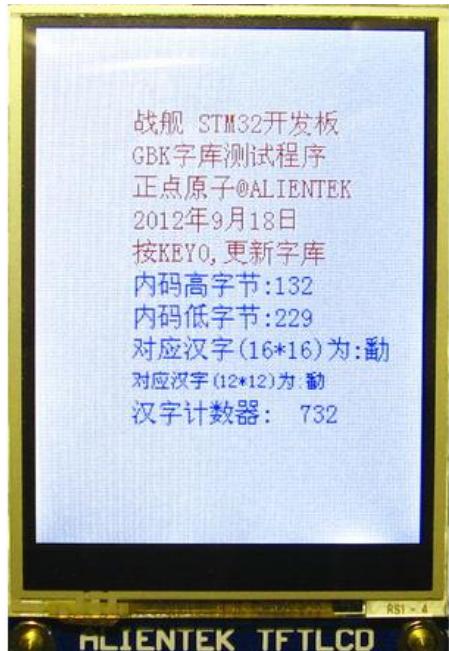


图 46.4.1 汉字显示实验显示效果

一开始就显示汉字，是因为 ALIENTEK 战舰 STM32 开发板在出厂的时候都是测试过的，里面刷了综合测试程序，已经把字库写入到了 W25Q64 里面，所以并不会提示更新字库。如果你想要更新字库，那么则必须先找一张 SD 卡，把我们提供的 SYSTEM 文件夹 COPY 到 SD 卡根目录下，重启程序。然后，在显示汉字的时候，按下 KEY0，就可以开始更新字库了。

字库更新界面如图 46.4.2 所示：



图 46.4.2 汉字字库更新界面

第四十七章 图片显示实验

数码相框日渐流行，数码相框显示的图片一般为 BMP/JPG/JPEG/GIF 等格式，其实用我们的战舰 STM32 开发板也可以显示图片。在本章中，我们将向大家介绍如何通过 STM32 来解码 BMP/JPG/JPEG/GIF 等图片，并在 LCD 上显示出来。本章分为如下几个部分：

- 47.1 图片格式简介
- 47.2 硬件设计
- 47.3 软件设计
- 47.4 下载验证



47.1 图片格式简介

我们常用的图片格式有很多，一般最常用的有三种：JPEG（或 JPG）、BMP 和 GIF。其中 JPEG（或 JPG）和 BMP 是静态图片，而 GIF 则是可以实现动态图片。下面，我们简单介绍一下这三种图片格式。

首先，我们来看看 BMP 图片格式。BMP（全称 Bitmap）是 Window 操作系统中的标准图像文件格式，文件后缀名为“.bmp”，使用非常广。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此，BMP 文件所占用的空间很大，但是没有失真。BMP 文件的图像深度可选 1bit、4bit、8bit、16bit、24bit 及 32bit。BMP 文件存储数据时，图像的扫描方式是按从左到右、从下到上的顺序。

典型的 BMP 图像文件由四部分组成：

- 1, 位图头文件数据结构，它包含 BMP 图像文件的类型、显示内容等信息；
- 2, 位图信息数据结构，它包含有 BMP 图像的宽、高、压缩方法，以及定义颜色等信息
- 3, 调色板，这个部分是可选的，有些位图需要调色板，有些位图，比如真彩色图（24 位的 BMP）就不需要调色板；
- 4, 位图数据，这部分的内容根据 BMP 位图使用的位数不同而不同，在 24 位图中直接使用 RGB，而其他的小于 24 位的使用调色板中颜色索引值。

关于 BMP 的详细介绍，请参考光盘的《BMP 图片文件详解.pdf》。接下来我们看看 JPEG 文件格式。

JPEG 是 Joint Photographic Experts Group(联合图像专家组)的缩写，文件后缀名为“.jpg”或“.jpeg”，是最常用的图像文件格式，由一个软件开发联合会组织制定，同 BMP 格式不同，JPEG 是一种有损压缩格式，能够将图像压缩在很小的储存空间，图像中重复或不重要的资料会被丢失，因此容易造成图像数据的损伤（BMP 不会，但是 BMP 占用空间大）。尤其是使用过高的压缩比例，将使最终解压缩后恢复的图像质量明显降低，如果追求高品质图像，不宜采用过高压缩比例。但是 JPEG 压缩技术十分先进，它用有损压缩方式去除冗余的图像数据，在获得极高的压缩率的同时能展现十分丰富生动的图像，换句话说，就是可以用最少的磁盘空间得到较好的图像品质。而且 JPEG 是一种很灵活的格式，具有调节图像质量的功能，允许用不同的压缩比例对文件进行压缩，支持多种压缩级别，压缩比率通常在 10:1 到 40:1 之间，压缩比越大，品质就越低；相反地，压缩比越小，品质就越好。比如可以把 1. 37Mb 的 BMP 位图文件压缩至 20. 3KB。当然也可以在图像质量和文件尺寸之间找到平衡点。JPEG 格式压缩的主要是高频信息，对色彩的信息保留较好，适合应用于互联网，可减少图像的传输时间，可以支持 24bit 真彩色，也普遍应用于需要连续色调的图像。

JPEG/JPG 的解码过程可以简单的概述为如下几个部分：

1、从文件头读出文件的相关信息。

JPEG 文件数据分为文件头和图像数据两大部分，其中文件头记录了图像的版本、长宽、采样因子、量化表、哈夫曼表等重要信息。所以解码前必须将文件头信息读出，以备

图像数据解码过程之用。

2、从图像数据流读取一个最小编码单元(MCU)，并提取出里边的各个颜色分量单元。

3、将颜色分量单元从数据流恢复成矩阵数据。

使用文件头给出的哈夫曼表，对分割出来的颜色分量单元进行解码，将其恢复成 8 × 8 的数据矩阵。



4、 8×8 的数据矩阵进一步解码。

此部分解码工作以 8×8 的数据矩阵为单位，其中包括相邻矩阵的直流系数差分解码、使用文件头给出的量化表反量化数据、反 Zig-zag 编码、隔行正负纠正、反向离散余弦变换等 5 个步骤，最终输出仍然是一个 8×8 的数据矩阵。

5、颜色系统 YCrCb 向 RGB 转换。

将一个 MCU 的各个颜色分量单元解码结果整合起来，将图像颜色系统从 YCrCb 向 RGB 转换。

6、排列整合各个 MCU 的解码数据。

不断读取数据流中的 MCU 并对其解码，直至读完所有 MCU 为止，将各 MCU 解码后的数据正确排列成完整的图像。

JPEG 的解码本身是比较复杂的，这里 FATFS 的作者，提供了一个轻量级的 JPG/JPEG 解码库：TjpgDec，最少仅需 3KB 的 RAM 和 3.5KB 的 FLASH 即可实现 JPG/JPEG 解码，本例程采用 TjpgDec 作为 JPG/JPEG 的解码库，关于 TjpgDec 的详细使用，请参考光盘：6，软件资料\图片解码\TjpgDec 技术手册 这个文档。

BMP 和 JPEG 这两种图片格式均不支持动态效果，而 GIF 则是可以支持动态效果。最后，我们来看看 GIF 图片格式。

GIF(Graphics Interchange Format)是 CompuServe 公司开发的图像文件存储格式，1987 年开发的 GIF 文件格式版本号是 GIF87a，1989 年进行了扩充，扩充后的版本号定义为 GIF89a。

GIF 图像文件以数据块(block)为单位来存储图像的相关信息。一个 GIF 文件由表示图形/图像的数据块、数据子块以及显示图形/图像的控制信息块组成，称为 GIF 数据流(Data Stream)。数据流中的所有控制信息块和数据块都必须在文件头(Header)和文件结束块(Trailer)之间。

GIF 文件格式采用了 LZW(Lempel-Ziv Walch)压缩算法来存储图像数据，定义了允许用户为图像设置背景的透明(transparency)属性。此外，GIF 文件格式可在同一个文件中存放多幅彩色图形/图像。如果在 GIF 文件中存放有多幅图，它们可以像演幻灯片那样显示或者像动画那样演示。

一个 GIF 文件的结构可分为文件头(File Header)、GIF 数据流(GIF Data Stream)和文件终结器(Trailer)三个部分。文件头包含 GIF 文件署名(Signature)和版本号(Version)；GIF 数据流由控制标识符、图象块(Image Block)和其他的一些扩展块组成；文件终结器只有一个值为 0x3B 的字符(';) 表示文件结束。

关于 GIF 的详细介绍，请参考光盘 GIF 解码相关资料。图片格式简介，我们就介绍到这里。

47.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY1 可以快速浏览下一张和上一张。如果未找到 PICTURE 文件夹/任何图片文件，则提示错误。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) 串口



3) TFTLCD 模块

4) SD 卡

5) SPI FLASH

这几部分，在之前的实例中都介绍过了，我们在此就不介绍了。需要注意的是，我们在 SD 卡根目录下要建一个 PICTURE 的文件夹，用来存放 JPEG、JPG、BMP 或 GIF 等图片（由于解码程序的问题，不是所有的 JPEG、JPG 图片都能打开，如果不能打开，则用 XP 自带的画图工具保存一下，再放到 PICTURE 文件夹下就可以打开了）。

47.3 软件设计

打开我们图片显示实验工程可以发现，我们的工程中多了一个 PICTURE 的组，组下面有四个.c 文件 bmp.c,tjpdg.c,gif.c 和 piclib.c。同时这四个源文件还引入了它们相应的头文件 bmp.h,tjpdg.h,gif.h 和 piclib.h。

其中 bmp.c 和 bmp.h 用于实现对 bmp 文件的解码；tjpdg.c 和 tjpdg.h 用于实现对 jpeg/jpg 文件的解码；gif.c 和 gif.h 用于实现对 gif 文件的解码；这几个代码太长了，所以我们在那里不贴出来，请大家参考光盘本例程的源码，我们打开 piclib.c，代码如下：

```
#include "piclib.h"
#include "lcd.h"

_pic_info picinfo;      //图片信息
_pic_phy pic_phy;      //图片显示物理接口
//lcd.h 没有提供划横线函数,需要自己实现
void piclib_draw_hline(u16 x0,u16 y0,u16 len,u16 color)
{
    if((len==0)||(x0>lcddev.width)||((y0>lcddev.height))return;
    LCD_Fill(x0,y0,x0+len-1,y0,color);
}

//画图初始化,在画图之前,必须先调用此函数
//指定画点/读点
void piclib_init(void)
{
    pic_phy.read_point=LCD_ReadPoint;      //读点函数实现
    pic_phy.draw_point=LCD_Fast_DrawPoint; //画点函数实现
    pic_phy.fill=LCD_Fill;                //填充函数实现
    pic_phy.draw_hline=piclib_draw_hline;   //画线函数实现
    pic_phy.set_window=LCD_Set_Window;    //开窗函数
    pic_phy.fill_prepare=LCD_WriteRAM_Prepare;//开始填充
    pic_phy.fill_point=LCD_WriteRAM;       //填充点
    picinfo.lcdwidth=lcddev.width;        //得到 LCD 的宽度像素
    picinfo.lcdheight=lcddev.height;       //得到 LCD 的高度像素
    picinfo.ImgWidth=0;                  //初始化宽度为 0
    picinfo.ImgHeight=0;                 //初始化高度为 0
    picinfo.Div_Fac=0;                  //初始化缩放系数为 0
    picinfo.S_Height=0;                 //初始化设定的高度为 0
    picinfo.S_Width=0;                  //初始化设定的宽度为 0
```



```
picinfo.S_XOFF=0;      //初始化 x 轴的偏移量为 0
picinfo.S_YOFF=0;      //初始化 y 轴的偏移量为 0
picinfo.staticx=0;      //初始化当前显示到的 x 坐标为 0
picinfo.staticy=0;      //初始化当前显示到的 y 坐标为 0
}
//快速 ALPHA BLENDING 算法.
//src:源颜色
//dst:目标颜色
//alpha:透明程度(0~32)
//返回值:混合后的颜色.
u16 piclib_alpha_blend(u16 src,u16 dst,u8 alpha)
{
    u32 src2;
    u32 dst2;
    src2=((src<<16)|src)&0x07E0F81F;
    dst2=((dst<<16)|dst)&0x07E0F81F;
    dst2=(((dst2-src2)*alpha)>>5)+src2)&0x07E0F81F;
    return (dst2>>16)|dst2;
}
//初始化智能画点
//内部调用
void ai_draw_init(void)
{
    float temp,temp1;
    temp=(float)picinfo.S_Width/picinfo.ImgWidth;
    temp1=(float)picinfo.S_Height/picinfo.ImgHeight;
    if(temp<temp1)temp1=temp;//取较小的那个
    if(temp1>1)temp1=1;
    //使图片处于所给区域的中间
    picinfo.S_XOFF+=(picinfo.S_Width-temp1*pinfo.ImgWidth)/2;
    picinfo.S_YOFF+=(picinfo.S_Height-temp1*pinfo.ImgHeight)/2;
    temp1*=8192;//扩大 8192 倍
    pinfo.Div_Fac=temp1;
    picinfo.staticx=0xffff;
    picinfo.staticy=0xffff;//放到一个不可能的值上面
}
//判断这个像素是否可以显示
//(x,y) :像素原始坐标
//chg   :功能变量.
//返回值:0,不需要显示.1,需要显示
u8 is_element_ok(u16 x,u16 y,u8 chg)
{
    if(x!=picinfo.staticx||y!=picinfo.staticy)
```



```
{  
    if(chg==1){picinfo.staticx=x; picinfo.staticy=y;}  
    return 1;  
}else return 0;  
}  
//智能画图  
//FileName:要显示的图片文件 BMP/JPG/JPEG/GIF  
//x,y,width,height:坐标及显示区域尺寸  
//fast:使能 jpeg/jpg 小图片(图片尺寸小于等于液晶分辨率)快速解码,0,不使能;1,使能.  
//acolor :alphablend 的颜色(仅对不大于 320*240 的 32 位 bmp 有效!)  
//abdnum :alphablend 的值(0~32 有效,其余值表示不使用 alphablend,  
//仅对不大于 320*240 的 32 位 bmp 有效!)  
//图片在开始和结束的坐标点范围内显示  
u8 ai_load_picfile(const u8 *filename,u16 x,u16 y,u16 width,u16 height,u8 fast)  
{  
    u8 res;//返回值  
    u8 temp;  
    if((x+width)>lcddev.width)return PIC_WINDOW_ERR; //x 坐标超范围了.  
    if((y+height)>lcddev.height)return PIC_WINDOW_ERR; //y 坐标超范围了.  
    //得到显示方框大小  
    if(width==0||height==0)return PIC_WINDOW_ERR; //窗口设定错误  
    picinfo.S_Height=height;  
    picinfo.S_Width=width;  
    //显示区域无效  
    if(picinfo.S_Height==0||picinfo.S_Width==0)  
    {  
        picinfo.S_Height=lcddev.height;  
        picinfo.S_Width=lcddev.width;  
        return FALSE;  
    }  
    //显示的开始坐标点  
    picinfo.S_YOFF=y; picinfo.S_XOFF=x;  
    //文件名传递  
    temp=f_typetell((u8*)filename); //得到文件的类型  
    switch(temp)  
    {  
        case T_BMP:  
            res=stdbmp_decode(filename); break; //解码 bmp  
        case T_JPG:  
        case T_JPEG:  
            res=jpg_decode(filename,fast); break; //解码 JPG/JPEG  
        case T_GIF:  
            res=gif_decode(filename,x,y,width,height); break; //解码 gif
```



```

default:
    res=PIC_FORMAT_ERR;break;           //非图片格式!!!
}
return res;
}

```

此段代码总共 6 个函数，其中，`piclib_draw_hline` 函数用于快速画横线，由 gif 解码程序使用。

`piclib_init` 函数，该函数用于初始化图片解码的相关信息，其中 `_pic_phy` 是我们在 `piclib.h` 里面定义的一个结构体，用于管理底层 LCD 接口函数，这些函数必须由用户在外部实现。`_pic_info` 则是另外一个结构体，用于图片缩放处理。

`piclib_alpha_blend` 函数，该函数用于实现半透明效果，在小格式（分辨率小于 240*320）`bmp` 解码的时候，可能被用到。

`ai_draw_init` 函数，该函数用于实现图片在显示区域的居中显示初始化，其实就是根据图片大小选择缩放比例和坐标偏移值。

`is_element_ok` 函数，该函数用于判断一个点是不是应该显示出来，在图片缩放的时候该函数是必须用到的。

`ai_load_picfile` 函数，该函数是整个图片显示的对外接口，外部程序，通过调用该函数，可以实现 `bmp`、`jpg/jpeg` 和 `gif` 的显示，该函数根据输入文件的后缀名，判断文件格式，然后交给相应的解码程序（`bmp` 解码/`jpeg` 解码/`gif` 解码），执行解码，完成图片显示。注意，这里我们用到一个 `f_typetell` 的函数，来判断文件的后缀名，`f_typetell` 函数在 `exfun.c` 里面实现，具体请参考光盘源码。

下面我们看看 `piclib.h` 头文件源码：

```

#ifndef __PICLIB_H
#define __PICLIB_H
#include "sys.h"
#include "lcd.h"
#include "malloc.h"
#include "ff.h"
#include "exfun.h"
#include "bmp.h"
#include "tjpd.h"
#include "gif.h"
#define PIC_FORMAT_ERR      0x27      //格式错误
#define PIC_SIZE_ERR        0x28      //图片尺寸错误
#define PIC_WINDOW_ERR      0x29      //窗口设定错误
#define PIC_MEM_ERR         0x11      //内存错误
//图片显示物理层接口
//在移植的时候,必须由用户自己实现这几个函数
typedef struct
{
    u16(*read_point)(u16,u16); //u16 read_point(u16 x,u16 y) 读点函数
    void(*draw_point)(u16,u16,u16); //void draw_point(u16 x,u16 y,u16 color) 画点函数
    void(*fill)(u16,u16,u16,u16,u16);
}

```



```

//void fill(u16 sx,u16 sy,u16 ex,u16 ey,u16 color)           单色填充函数
void(*draw_hline)(u16,u16,u16,u16);                           画水平线函数
//void draw_hline(u16 x0,u16 y0,u16 len,u16 color)
void(*set_window)(u16,u16,u16,u16);                           设置显示窗口函数
//void set_window(u16 sx,u16 sy,u16 width,u16 height)
void(*fill_prepare)(void); //void draw_prepare(void)          准备填充
void(*fill_point)(u16); //void fill_point(u16 color)        填充一个像素点
} _pic_phy;
extern _pic_phy pic_phy;
//图像信息
typedef struct
{
    u16 lcdwidth; //LCD 的宽度
    u16 lcdheight; //LCD 的高度
    u32 ImgWidth; //图像的实际宽度和高度
    u32 ImgHeight;
    u32 Div_Fac; //缩放系数 (扩大了 8192 倍的)
    u32 S_Height; //设定的高度和宽度
    u32 S_Width;
    u32 S_XOFF; //x 轴和 y 轴的偏移量
    u32 S_YOFF; u32 staticx; //当前显示到的 x y 坐标
    u32 staticy;
} _pic_info;
extern _pic_info picinfo;//图像信息
void piclib_init(void); //初始化画图
u16 piclib_alpha_blend(u16 src,u16 dst,u8 alpha); //alphablend 处理
void ai_draw_init(void); //初始化智能画图
u8 is_element_ok(u16 x,u16 y,u8 chg); //判断像素是否有效
u8 ai_load_picfile(const u8 *filename,u16 x,u16 y,u16 width,u16 height); //智能画图
#endif

```

这里基本就是我们前面提到的两个结构体的定义以及一些函数的申明。最后我们看看 main.c 里面的源码：

```

//得到 path 路径下,目标文件的总个数
//path:路径
//返回值:总有效文件数
u16 pic_get_tnum(u8 *path)
{
    u8 res;
    u16 rval=0;
    DIR tdir; //临时目录
    FILINFO tfileinfo; //临时文件信息
    u8 *fn;
    res=f_opendir(&tdir,(const TCHAR*)path); //打开目录

```



```
tfileinfo.lfsize=_MAX_LFN*2+1;           //长文件名最大长度
tfileinfo.lfname=mymalloc(SRAMIN,tfileinfo.lfsize); //为长文件缓存区分配内存
if(res==FR_OK&&tfileinfo.lfname!=NULL)
{
    while(1)//查询总的有效的文件数
    {
        res=f_readdir(&tdir,&tfileinfo);           //读取目录下的一个文件
        if(res!=FR_OK||tfileinfo.fname[0]==0)break; //错误了/到末尾了,退出
        fn=(u8*)(*tfileinfo.lfname?tfileinfo.lfname:tfileinfo.fname);
        res=f_typetell(fn);
        if((res&0XF0)==0X50) rval++;//图片文件? 则有效文件加 1
    }
}
return rval;
}

int main(void)
{
    u8 res;
    DIR picdir;           //图片目录
    FILINFO picfileinfo; //文件信息
    u8 *fn;               //长文件名
    u8 *pname;             //带路径的文件名
    u16 totpicnum;        //图片文件总数
    u16 curindex;          //图片当前索引
    u8 key;                //键值
    u8 pause=0;             //暂停标记
    u8 t;
    u16 temp;
    u16 *picindextbl;      //图片索引表
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();              //LED 端口初始化
    LCD_Init();              //LCD 初始化
    KEY_Init();              //KEY 初始化
    usmart_dev.init(72);     //usmart 初始化
    mem_init(SRAMIN);       //初始化内部内存池
    exfun_init();            //为 fatfs 相关变量申请内存
    f_mount(0,fs[0]);        //挂载 SD 卡
    f_mount(1,fs[1]);        //挂载 FLASH.
    POINT_COLOR=RED;
    while(font_init())        //检查字库
    {
```



```
LCD_ShowString(60,50,200,16,16,"Font Error!");
delay_ms(200);
LCD_Fill(60,50,240,66,WHITE);//清除显示
}

Show_Str(60,50,200,16,"战舰 STM32 开发板",16,0);
Show_Str(60,70,200,16,"图片显示程序",16,0);
Show_Str(60,90,200,16,"KEY0:NEXT KEY1:PREV",16,0);

Show_Str(60,110,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,130,200,16,"2012 年 9 月 19 日",16,0);
while(f_opendir(&picdir,"0:/PICTURE"))//打开图片文件夹
{
    Show_Str(60,150,240,16,"PICTURE 文件夹错误!",16,0);
    delay_ms(200);
    LCD_Fill(60,150,240,146,WHITE);//清除显示
    delay_ms(200);
}
totpicnum=pic_get_tnum("0:/PICTURE"); //得到总有效文件数
while(totpicnum==NULL)//图片文件为 0
{
    Show_Str(60,150,240,16,"没有图片文件!",16,0);
    delay_ms(200);
    LCD_Fill(60,150,240,146,WHITE);//清除显示
    delay_ms(200);
}
picfileinfo.lfsize=_MAX_LFN*2+1;      //长文件名最大长度
picfileinfo.lfname=mymalloc(SRAMIN,picfileinfo.lfsize); //为长文件缓存区分配内存
pname=mymalloc(SRAMIN,picfileinfo.lfsize); //为带路径的文件名分配内存
picindextbl=mymalloc(SRAMIN,2*totpicnum); //申请 2*totpicnum 个字节的内存,
//用于存放图片索引
while(picfileinfo.lfname==NULL||pname==NULL||picindextbl==NULL)//内存分配出错
{
    Show_Str(60,150,240,16,"内存分配失败!",16,0);
    delay_ms(200);
    LCD_Fill(60,150,240,146,WHITE);//清除显示
    delay_ms(200);
}
//记录索引
res=f_opendir(&picdir,"0:/PICTURE"); //打开目录
if(res==FR_OK)
{
    curindex=0;//当前索引为 0
    while(1)//全部查询一遍
```



```
{  
    temp=picdir.index;                      //记录当前 index  
    res=f_readdir(&picdir,&picfileinfo);      //读取目录下的一个文件  
    if(res!=FR_OK||picfileinfo.fname[0]==0)break; //错误了/到末尾了,退出  
    fn=(u8*)(*picfileinfo.lfname?picfileinfo.lfname:picfileinfo.fname);  
    res=f_t typetell(fn);  
    if((res&0XF0)==0X50)//取高四位,看看是不是图片文件  
    {  
        picindextbl[curindex]=temp;//记录索引  
        curindex++;  
    }  
}  
}  
Show_Str(60,150,240,16,"开始显示...",16,0);  
delay_ms(1500);  
piclib_init();                         //初始化画图  
curindex=0;                            //从 0 开始显示  
res=f_opendir(&picdir,(const TCHAR*)"0:/PICTURE"); //打开目录  
while(res==FR_OK)//打开成功  
{  
    dir_sdi(&picdir,picindextbl[curindex]);      //改变当前目录索引  
    res=f_readdir(&picdir,&picfileinfo);          //读取目录下的一个文件  
    if(res!=FR_OK||picfileinfo.fname[0]==0)break; //错误了/到末尾了,退出  
    fn=(u8*)(*picfileinfo.lfname?picfileinfo.lfname:picfileinfo.fname);  
    strcpy((char*)pname,"0:/PICTURE/");           //复制路径(目录)  
    strcat((char*)pname,(const char*)fn);          //将文件名接在后面  
    LCD_Clear(BLACK);  
    ai_load_picfile(pname,0,0,lcddev.width,lcddev.height,1);//显示图片  
    Show_Str(2,2,240,16,pname,16,1);                //显示图片名字  
    t=0;  
    while(1)  
    {  
        key=KEY_Scan(0);                          //扫描按键  
        if(t>250)key=KEY_RIGHT; //模拟一次按下右键  
        if(key==KEY_LEFT)           //上一张  
        {  
            if(curindex)curindex--;  
            else curindex=totpicnum-1;  
            break;  
        }else if(key==KEY_RIGHT)//下一张  
        {  
            curindex++;  
            if(curindex>=totpicnum)curindex=0;//到末尾的时候,自动从头开始  
        }  
    }  
}
```



```
        break;
    }else if(key==KEY_UP)
    {
        pause=!pause;
        LED1=!pause;      //暂停的时候 LED1 亮.
    }
    if(pause==0)t++;
    delay_ms(10);
}
res=0;
}
myfree(SRAMIN,picfileinfo.lfname);      //释放内存
myfree(SRAMIN,pname);                  //释放内存
myfree(SRAMIN,picindextbl);            //释放内存
}
```

此部分除了 main 函数，还有一个 pic_get_tnum 的函数，用来得到 path 路径下，所有有效文件（图片文件）的个数。在 main 函数里面我们通过索引（图片文件在 PICTURE 文件夹下的编号），来查找上一个/下一个图片文件，这里我们需要用到 fatfs 自带的一个函数：dir_sdi，来设置当前目录的索引（因为 f_readdir 只能沿着索引一直往下找，不能往上找），方便定位到任何一个文件。dir_sdi 在 FATFS 下面被定义为 static 函数，所以我们必须在 ff.c 里面将该函数的 static 修饰词去掉，然后在 ff.h 里面添加该函数的申明，以便 main 函数使用。

其他部分就比较简单了，至此，整个图片显示实验的软件设计部分就结束了。该程序将实现浏览 PICTURE 文件夹下的所有图片，并显示其名字，每隔 3s 左右切换一幅图片。

47.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，可以看到 LCD 开始显示图片（假设 SD 卡及文件都准备好了），如图 47.4.1 所示：



图 37.4.1 图片显示实验显示效果

按 KEY0 和 KEY2 可以快速切换到下一张或上一张。同时，由于我们的代码支持 gif 格式的图片显示（注意尺寸不能超过 LCD 屏幕尺寸），所以可以放一些 gif 图片到 PICTURE 文件夹，来看动画了。

本章，同样可以通过 USMART 来测试该实验，将 ai_load_picfile 函数加入 USMART 控制（方法前面已经讲了很多次了），就可以通过串口调用该函数，在屏幕上任何区域显示任何你想要显示的图片了！

第四十八章 照相机实验

上一章，我们学习了图片解码，本章我们将学习 bmp 编码，结合前面的摄像头实验，实现一个简单的照相机。本章分为如下几个部分：

- 48.1 BMP 编码简介
- 48.2 硬件设计
- 48.3 软件设计
- 48.4 下载验证



48.1 BMP 编码简介

上一章，我们学习了各种图片格式的解码。本章，我们介绍最简单的图片编码方法：BMP 图片编码。通过前面的了解，我们知道 BMP 文件是由文件头、位图信息头、颜色信息和图形数据等四部分组成。我们先来了解下这几个部分。

1、BMP 文件头（14 字节）：BMP 文件头数据结构含有 BMP 文件的类型、文件大小和位图起始位置等信息。

```
//BMP 文件头
typedef __packed struct
{
    u16  bfType ;          //文件标志.只对'BM',用来识别 BMP 位图类型
    u32  bfSize ;          //文件大小,占四个字节
    u16  bfReserved1 ;     //保留
    u16  bfReserved2 ;     //保留
    u32  bfOffBits ;       //从文件开始到位图数据(bitmap data)开始之间的偏移量
}BITMAPFILEHEADER ;
```

2、位图信息头（40 字节）：BMP 位图信息头数据用于说明位图的尺寸等信息。

```
typedef __packed struct
{
    u32 biSize ;           //说明 BITMAPINFOHEADER 结构所需要的字数。
    long  biWidth ;         //说明图象的宽度,以象素为单位
    long  biHeight ;        //说明图象的高度,以象素为单位
    u16  biPlanes ;        //为目标设备说明位面数,其值将总是被设为 1
    u16  biBitCount ;       //说明比特数/象素,其值为 1、4、8、16、24、或 32
    u32 biCompression ;    //说明图象数据压缩的类型。其值可以是下述值之一:
    //BI_RGB: 没有压缩;
    //BI_RLE8: 每个象素 8 比特的 RLE 压缩编码,压缩格式由 2 字节组成
    //BI_RLE4: 每个象素 4 比特的 RLE 压缩编码,压缩格式由 2 字节组成
    //BI_BITFIELDS: 每个象素的比特由指定的掩码决定。
    u32 biSizeImage ; //说明图象的大小,以字节为单位。当用 BI_RGB 格式时,可设置为 0
    long  biXPelsPerMeter ;//说明水平分辨率,用象素/米表示
    long  biYPelsPerMeter ;//说明垂直分辨率,用象素/米表示
    u32 biClrUsed ;        //说明位图实际使用的彩色表中的颜色索引数
    u32 biClrImportant ;   //说明对图象显示有重要影响的颜色索引的数目,
                           //如果是 0, 表示都重要。
}BITMAPINFOHEADER ;
```

3、颜色表：颜色表用于说明位图中的颜色，它有若干个表项，每一个表项是一个 RGBQUAD 类型的结构，定义一种颜色。

```
typedef __packed struct
{
    u8 rgbBlue ;           //指定蓝色强度
    u8 rgbGreen ;          //指定绿色强度
```



```
u8 rgbRed ;      //指定红色强度
u8 rgbReserved ; //保留，设置为 0
}RGBQUAD ;
```

颜色表中 RGBQUAD 结构数据的个数由 biBitCount 来确定：当 biBitCount=1、4、8 时，分别有 2、16、256 个表项；当 biBitCount 大于 8 时，没有颜色表项。

BMP 文件头、位图信息头和颜色表组成位图信息（我们将 BMP 文件头也加进来，方便处理），BITMAPINFO 结构定义如下：

```
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1];
}BITMAPINFO;
```

4、位图数据：位图数据记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右，扫描行之间是从下到上。位图的一个像素值所占的字节数：

当 biBitCount=1 时，8 个像素占 1 个字节；
当 biBitCount=4 时，2 个像素占 1 个字节；
当 biBitCount=8 时，1 个像素占 1 个字节；
当 biBitCount=16 时，1 个像素占 2 个字节；
当 biBitCount=24 时，1 个像素占 3 个字节；
当 biBitCount=32 时，1 个像素占 4 个字节；

biBitCount=1 表示位图最多有两种颜色，缺省情况下是黑色和白色，你也可以自己定义这两种颜色。图像信息头装调色板中将有两个调色板项，称为索引 0 和索引 1。图象数据阵列中的每一位表示一个象素。如果一个位是 0，显示时就使用索引 0 的 RGB 值，如果位是 1，则使用索引 1 的 RGB 值。

biBitCount=16 表示位图最多有 65536 种颜色。每个色素用 16 位（2 个字节）表示。这种格式叫作高彩色，或叫增强型 16 位色，或 64K 色。它的情况比较复杂，当 biCompression 成员的值是 BI_RGB 时，它没有调色板。16 位中，最低的 5 位表示蓝色分量，中间的 5 位表示绿色分量，高的 5 位表示红色分量，一共占用了 15 位，最高的一位保留，设为 0。这种格式也被称作 555 16 位位图。如果 biCompression 成员的值是 BI_BITFIELDS，那么情况就复杂了，首先是原来调色板的位置被三个 DWORD 变量占据，称为红、绿、蓝掩码。分别用于描述红、绿、蓝分量在 16 位中所占的位置。在 Windows 95（或 98）中，系统可接受两种格式的位域：555 和 565，在 555 格式下，红、绿、蓝的掩码分别是：0x7C00、0x03E0、0x001F，而在 565 格式下，它们则分别为：0xF800、0x07E0、0x001F。你在读取一个像素之后，可以分别用掩码“与”上像素值，从而提取出想要的颜色分量（当然还要再经过适当的左右移操作）。在 NT 系统中，则没有格式限制，只不过要求掩码之间不能有重叠。（注：这种格式的图像使用起来是比较麻烦的，不过因为它的显示效果接近于真彩，而图像数据又比真彩图像小的多，所以，它更多的被用于游戏软件）。

biBitCount=32 表示位图最多有 4294967296(2 的 32 次方)种颜色。这种位图的结构与 16 位位图结构非常类似，当 biCompression 成员的值是 BI_RGB 时，它也没有调色板，32 位中有 24 位用于存放 RGB 值，顺序是：最高位—保留，红 8 位、绿 8 位、蓝 8 位。这种格式也被成为 888 32 位图。如果 biCompression 成员的值是 BI_BITFIELDS 时，原来调色板的位置将被三个 DWORD 变量占据，成为红、绿、蓝掩码，分别用于描述红、绿、蓝分量在 32 位中所占的位



置。在 Windows 95(or 98)中，系统只接受 888 格式，也就是说三个掩码的值将只能是：0xFF0000、0xFF00、0xFF。而在 NT 系统中，你只要注意使掩码之间不产生重叠就行。（注：这种图像格式比较规整，因为它是 DWORD 对齐的，所以在内存中进行图像处理时可进行汇编级的代码优化（简单））。

通过以上了解，我们对 BMP 有了一个比较深入的了解，本章，我们采用 16 位 BMP 编码（因为我们的 LCD 就是 16 位色的，而且 16 位 BMP 编码比 24 位 BMP 编码更省空间），故我们需要设置 biBitCount 的值为 16，这样得到新的位图信息（BITMAPINFO）结构体：

```
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    u32 RGB_MASK[3];           //调色板用于存放 RGB 掩码.
}BITMAPINFO;
```

其实就是颜色表由 3 个 RGB 掩码代替。最后，我们来看看将 LCD 的显存保存为 BMP 格式的图片文件的步骤：

1) 创建 BMP 位图信息，并初始化各个相关信息

这里，我们要设置 BMP 图片的分辨率为 LCD 分辨率（240*320）、BMP 图片的大小（整个 BMP 文件大小）、BMP 的像素位数（16 位）和掩码等信息。

2) 创建新 BMP 文件，写入 BMP 位图信息

我们要保存 BMP，当然要存放在某个地方（文件），所以需要先创建文件，同时先保存 BMP 位图信息，之后才开始 BMP 数据的写入。

3) 保存位图数据。

这里就比较简单了，只需要从 LCD 的 GRAM 里面读取各点的颜色值，依次写入第二步创建的 BMP 文件即可。注意：保存顺序（即读 GRAM 顺序）是从左到右，从下到上。

4) 关闭文件。

使用 FATFS，在文件创建之后，必须调用 f_close，文件才会真正体现在文件系统里面，否则是不会写入的！这个要特别注意，写完之后，一定要调用 f_close。

BMP 编码就介绍到这里。

48.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后检测 SD 卡根目录是否存在 PHOTO 文件夹，如果不存在则创建，如果创建失败，则报错（提示拍照功能不可用）。在找到 SD 卡的 PHOTO 文件夹后，开始初始化 OV7670，在初始化成功之后，就一直在屏幕显示 OV7670 拍到的内容。当按下 WK_UP 按键的时候，即进行拍照，此时 DS1 亮，拍照保存成功之后，蜂鸣器会发出“滴”的一声，提示拍照成功，同时 DS1 灭。DS0 还是用于指示程序运行状态。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 和 DS1
- 2) WK_UP 按键
- 3) 蜂鸣器
- 4) 串口
- 5) TFTLCD 模块
- 6) SD 卡



7) SPI FLASH

8) 摄像头模块

这几部分，在之前的实例中都介绍过了，我们在此就不介绍了。

48.3 软件设计

打开照相机实验工程，然后打开 PICTURE 组下的 bmp.c 文件，可以看到我们在该文件里面添加 bmp 编码函数 bmp_encode，该函数代码如下：

```
//BMP 编码函数
//将当前 LCD 屏幕的指定区域截图,存为 16 位格式的 BMP 文件 RGB565 格式.
//保存为 rgb565 则需要掩码,利用原来的调色板位置增加掩码.这里我们已经增加了掩码.
//保存为 rgb555 格式则需要颜色转换,耗时间比较久,所以保存为 565 是最快速的办法.
//filename:存放路径
//x,y:在屏幕上的起始坐标
//mode:模式.0,仅仅创建新文件的方式编码;1,如果之前存在文件,则覆盖之前的文件.
//如果没有,则创建新的文件.
//返回值:0,成功;其他,错误码.
u8 bmp_encode(u8 *filename,u16 x,u16 y,u16 width,u16 height,u8 mode)
{
    FIL* f_bmp;
    u16 bmpheadsize;          //bmp 头大小
    BITMAPINFO hbmp;          //bmp 头
    u8 res=0;
    u16 tx,ty;                //图像尺寸
    u16 *databuf;             //数据缓存区地址
    u16 pixcnt;                //像素计数器
    u16 bi4width;              //水平像素字节数
    if(width==0||height==0) return PIC_WINDOW_ERR;      //区域错误
    if((x+width-1)>lcddev.width) return PIC_WINDOW_ERR;  //区域错误
    if((y+height-1)>lcddev.height) return PIC_WINDOW_ERR; //区域错误
#if BMP_USE_MALLOC == 1 //使用 malloc
    databuf=(u16*)mymalloc(SRAMIN,1024); //开辟至少 bi4width 大小的字节的内存区域
    //对 240 宽的屏,480 个字节就够了.
    if(databuf==NULL) return PIC_MEM_ERR; //内存申请失败.
    f_bmp=(FIL *)mymalloc(SRAMIN,sizeof(FIL)); //开辟 FIL 字节的内存区域
    if(f_bmp==NULL) return PIC_MEM_ERR; //内存申请失败.
{
    myfree(SRAMIN,databuf);
    return PIC_MEM_ERR;
}
#else
    databuf=(u16*)bmpreadbuf;
    f_bmp=&f_bfile;
}
```



```
#endif

    bmpheadsize=sizeof(hbmp);           //得到 bmp 文件头的大小
    mymemset((u8*)&hbmp,0,sizeof(hbmp));//置零空申请到的内存.
    hbmp.bmiHeader.biSize=sizeof(BITMAPINFOHEADER);//信息头大小
    hbmp.bmiHeader.biWidth=width;        //bmp 的宽度
    hbmp.bmiHeader.biHeight=height;      //bmp 的高度
    hbmp.bmiHeader.biPlanes=1;          //恒为 1
    hbmp.bmiHeader.biBitCount=16;        //bmp 为 16 位色 bmp
    hbmp.bmiHeader.biCompression=BI_BITFIELDS;//每个象素的比特由指定的掩码决定。
    hbmp.bmiHeader.biSizeImage=hbmp.bmiHeader.biHeight*hbmp.bmiHeader.biWidth*
    hbmp.bmiHeader.biBitCount/8;//bmp 数据区大小
    hbmp.bmfHeader.bfType=((u16)'M'<<8)+'B';//BM 格式标志
    hbmp.bmfHeader.bfSize=bmpheadsize+hbmp.bmiHeader.biSizeImage;//整个 bmp 的大小
    hbmp.bmfHeader.bfOffBits=bmpheadsize;//到数据区的偏移
    hbmp.RGB_MASK[0]=0X00F800;          //红色掩码
    hbmp.RGB_MASK[1]=0X0007E0;          //绿色掩码
    hbmp.RGB_MASK[2]=0X00001F;          //蓝色掩码
    if(mode==1)res=f_open(f_bmp,(const TCHAR*)filename,FA_READ|FA_WRITE);
    //尝试打开之前的文件
    if(mode==0||res==0x04)res=f_open(f_bmp,(const TCHAR*)filename,FA_WRITE|
    FA_CREATE_NEW);//模式 0,或者尝试打开失败,则创建新文件
    if((hbmp.bmiHeader.biWidth*2)%4)//水平像素(字节)不为 4 的倍数
    {
        bi4width=((hbmp.bmiHeader.biWidth*2)/4+1)*4;
        //实际要写入的宽度像素,必须为 4 的倍数.
    }else bi4width=hbmp.bmiHeader.biWidth*2;           //刚好为 4 的倍数
    if(res==FR_OK)//创建成功
    {
        res=f_write(f_bmp,(u8*)&hbmp,bmpheadsize,&bw);//写入 BMP 首部
        for(ty=y+height-1;hbmp.bmiHeader.biHeight;ty--)
        {
            pixcnt=0;
            for(tx=x;pixcnt!=(bi4width/2);)
            {
                if(pixcnt<hbmp.bmiHeader.biWidth)databuf[pixcnt]=LCD_ReadPoint(tx,ty);
                //读取坐标点的值
                else databuf[pixcnt]=0Xffff;//补充白色的像素.
                pixcnt++; tx++;
            }
            hbmp.bmiHeader.biHeight--;
            res=f_write(f_bmp,(u8*)databuf,bi4width,&bw);//写入数据
        }
        f_close(f_bmp);
    }
```



```
    }
#endif BMP_USE_MALLOC == 1 //使用 malloc
myfree(SRAMIN,databuf);
myfree(SRAMIN,f_bmp);
#endif
return res;
}
```

该函数实现了对 LCD 屏幕的任意指定区域进行截屏保存，用到的方法就是 48.1 节我们所介绍的方法，该函数实现了将 LCD 任意指定区域的内容，保存为 16 位 BMP 格式，存放在指定位置（由 filename 决定）。注意，代码中的 BMP_USE_MALLOC 是在 bmp.h 定义的一个宏，用于设置是否使用 malloc，本章我们选择使用 malloc。

接着打开 bmp.h，可以发现我们在 bmp.h 里面添加 bmp_encode 函数的申明。

接下来，我们看看主函数，打开 main.c，修改该文件代码如下：

```
extern u8 ov_sto;      //在 exit.c 里面定义
extern u8 ov_frame;   //在 timer.c 里面定义
//更新 LCD 显示
void camera_refresh(void)
{
    u32 j;
    u16 color;
    if(ov_sto==2)
    {
        LCD_Scan_Dir(U2D_L2R);      //从上到下,从左到右
        LCD_SetCursor(0x00,0x0000); //设置光标位置
        LCD_WriteRAM_Prepae();     //开始写入 GRAM
        OV7670_RRST=0;             //开始复位读指针
        OV7670_RCK=0;
        OV7670_RCK=1;
        OV7670_RCK=0;
        OV7670_RRST=1;             //复位读指针结束
        OV7670_RCK=1;
        for(j=0;j<76800;j++)
        {
            OV7670_RCK=0;
            color=GPIOC->IDR&0XFF; //读数据
            OV7670_RCK=1;
            color<<=8;
            OV7670_RCK=0;
            color|=GPIOC->IDR&0XFF; //读数据
            OV7670_RCK=1;
            LCD->LCD_RAM=color;
        }
        EXTI_ClearITPendingBit(EXTI_Line8); //清除 EXTI8 线路挂起位
    }
}
```



```
ov_sta=0;           //开始下一次采集
ov_frame++;
LCD_Scan_Dir(DFT_SCAN_DIR); //恢复默认扫描方向
}

}

//文件名自增（避免覆盖）
//组合成:形如"0:PHOTO/PIC13141.bmp"的文件名
void camera_new.pathname(u8 *pname)
{
    u8 res;
    u16 index=0;
    while(index<0xFFFF)
    {
        sprintf((char*)pname,"0:PHOTO/PIC%05d.bmp",index);
        res=f_open(ftemp,(const TCHAR*)pname,FA_READ);//尝试打开这个文件
        if(res==FR_NO_FILE)break; //该文件名不存在=正是我们需要的.
        index++;
    }
}
int main(void)
{
    u8 res;
    u8 *pname;          //带路径的文件名
    u8 key;            //键值
    u8 i;
    u8 sd_ok=1;         //0,SD卡不正常;1,SD卡正常.
    delay_init();       //延时函数初始化
    NVIC_Configuration(); //设置NVIC中断分组2:2位抢占优先级,2位响应优先级
    uart_init(9600);   //串口初始化波特率为9600
    LED_Init();         //LED端口初始化
    LCD_Init();         //LCD初始化
    BEEP_Init();        //蜂鸣器初始化
    KEY_Init();         //KEY初始化
    mem_init(SRAMIN);  //初始化内部内存池

    exfun_init();       //为fatfs相关变量申请内存
    f_mount(0,fs[0]);  //挂载SD卡
    f_mount(1,fs[1]);  //挂载FLASH.
    piclib_init();      //初始化画图

    POINT_COLOR=RED;
    while(font_init()) //检查字库
    {
```



```
LCD_ShowString(60,50,200,16,16,"Font Error!");
delay_ms(200);
LCD_Fill(60,50,240,66,WHITE);//清除显示
}

Show_Str(60,50,200,16,"战舰 STM32 开发板",16,0);
Show_Str(60,70,200,16,"照相机实验",16,0);
Show_Str(60,90,200,16,"WK_UP:拍照",16,0);
Show_Str(60,110,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,130,200,16,"2012 年 9 月 19 日",16,0);
res=f_mkdir("0:/PHOTO");      //创建 PHOTO 文件夹
if(res!=FR_EXIST&&res!=FR_OK) //发生了错误
{
    Show_Str(60,150,240,16,"SD 卡错误!",16,0);
    delay_ms(200);
    Show_Str(60,170,240,16,"拍照功能将不可用!",16,0);
    sd_ok=0;
}
else
{
    Show_Str(60,150,240,16,"SD 卡正常!",16,0);
    delay_ms(200);
    Show_Str(60,170,240,16,"KEY_UP:拍照",16,0);
    sd_ok=1;
}
pname=mymalloc(SRAMIN,30); //为带路径的文件名分配 30 个字节的内存
while(pname==NULL)          //内存分配出错
{
    Show_Str(60,190,240,16,"内存分配失败!",16,0);
    delay_ms(200);
    LCD_Fill(60,190,240,146,WHITE);//清除显示
    delay_ms(200);
}
while(OV7670_Init())//初始化 OV7670
{
    Show_Str(60,190,240,16,"OV7670 错误!",16,0);
    delay_ms(200);
    LCD_Fill(60,190,239,186,WHITE);
    delay_ms(200);
}
Show_Str(60,190,200,16,"OV7670 正常",16,0);
delay_ms(1500);
TIM6_Int_Init(10000,7199);      //10Khz 计数频率,1 秒钟中断

EXTI8_Init();                  //使能定时器捕获
```



```
OV7670_Window_Set(10,174,240,320); //设置窗口
OV7670_CS=0;
while(1)
{
    key=KEY_Scan(0);//不支持连接
    if(key==KEY_UP)
    {
        if(sd_ok)
        {
            LED1=0; //点亮 DS1,提示正在拍照
            camera_new.pathname(pname);//得到文件名
            if(bmp_encode(pname,0,0,240,320,0))//拍照有误
            {
                Show_Str(40,130,240,12,"写入文件错误!",12,0);
            }else
            {
                Show_Str(40,130,240,12,"拍照成功!",12,0);
                Show_Str(40,150,240,12,"保存为:",12,0);
                Show_Str(40+42,150,240,12,pname,12,0);
                BEEP=1;//蜂鸣器短叫, 提示拍照完成
                delay_ms(100);
            }
        }else //提示 SD 卡错误
        {
            Show_Str(40,130,240,12,"SD 卡错误!",12,0);
            Show_Str(40,150,240,12,"拍照功能不可用!",12,0);
        }
        BEEP=0;//关闭蜂鸣器
        LED1=1;//关闭 DS1
        delay_ms(1800);//等待 1.8 秒钟
    }else delay_ms(10);
    camera_refresh();//更新显示
    i++;
    if(i==20)//DS0 闪烁.
    {
        i=0;
        LED0=!LED0;
    }
}
```

此部分代码，和第四十一章的代码有点类似，只是这里我们多了一个 `camera_new.pathname` 函数，用于获取新的 `bmp` 文件名字（不覆盖旧的）。在 `main` 函数里面，我们通过 `WK_UP` 按键

控制拍照（调用 `bmp_encode` 函数实现），其他部分我们就不多介绍了，至此照相机实验代码编写完成。

最后，本实验可以通过 USMART 来测试 BMP 编码函数，将 `bmp_encode` 函数添加到 USMART 管理，即可通过串口自行控制拍照，方便测试。

48.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 战舰 STM32 开发板上，得到如图 48.4.1 所示界面：



图 48.4.1 程序运行效果图

随后，进入监控界面。此时，我们可以按下 `WK_UP` 即可进行拍照。拍照得到的照片效果如图 48.4.2 所示：



图 48.4.2 拍照样图

最后，我们还可以通过 USMART 调用 `bmp_encode` 函数，实现串口控制拍照，还可以拍成各种尺寸哦（不过必须小于 240*320）！

第四十九章 音乐播放器实验

前几年，MP3 曾经风行一时，几乎人手一个。如果能自己做一个 MP3，我想对于很多朋友来说，是十分骄傲的事情。ALIENTEK 战舰 STM32 开发板自带了一颗非常强劲的 MP3 解码芯片：VS1053，利用该芯片，我们可以实现 MP3/OGG/WMA/WAV/FLAC/AAC/MIDI 等各种音频文件的播放。本章，我们将利用战舰 STM32 开发板实现一个简单的 MP3 播放器。本章分为如下几个部：

- 49.1 MP3 简介
- 49.2 硬件设计
- 49.3 软件设计
- 49.4 下载验证



49.1 VS1053 简介

VS1053 是继 VS1003 后荷兰 VLSI 公司出品的又一款高性能解码芯片。该芯片可以实现对 MP3/OGG/WMA/FLAC/WAV/AAC/MIDI 等音频格式的解码，同时还可以支持 ADPCM/OGG 等格式的编码，性能相对以往的 VS1003 提升不少。VS1053 拥有一个高性能的 DSP 处理器核 VS_DSP，16K 的指令 RAM，0.5K 的数据 RAM，通过 SPI 控制，具有 8 个可用的通用 IO 口和一个串口，芯片内部还带了一个可变采样率的立体声 ADC（支持咪头/咪头+线路/2 线路）、一个高性能立体声 DAC 及音频耳机放大器。

VS1053 的特性如下：

- 支持众多音频格式解码，包括OGG/MP3/WMA/WAV/FLAC（需要加载patch）/MIDI/AAC等。
- 对话筒输入或线路输入的音频信号进行OGG（需要加载patch）/IMA ADPCM编码
- 高低音控制
- 带有EarSpeaker空间效果（用耳机虚拟现场空间效果）
- 单时钟操作12..13MHz
- 内部PLL锁相环时钟倍频器
- 低功耗
- 内含高性能片上立体声DAC，两声道间无相位差
- 过零交差侦测和平滑的音量调整
- 内含能驱动30 欧负载的耳机驱动器
- 模拟，数字，I/O 单独供电
- 为用户代码和数据准备的16KB片上RAM
- 可扩展外部DAC的I2S接口
- 用于控制和数据的串行接口（SPI）
- 可被用作微处理器的从机
- 特殊应用的SPI Flash引导
- 供调试用途的UART接口
- 新功能可以通过软件和 8 GPIO 添加

VS1053 相对于它的前辈 VS1003，增加了编解码格式的支持（比如支持 OGG/FLAC，还支持 OGG 编码，VS1003 不支持）、增加了 GPIO 数量到 8 个（VS1003 只有 4 个）、增加了内部指令 RAM 容量到 16KiB（VS1003 只有 5.5KiB）、增加了 I2S 接口（VS1003 没有）、支持 EarSpeaker 空间效果（VS1003 不支持）等。同时 VS1053 的 DAC 相对于 VS1003 有不少提高，同样的歌曲，用 VS1053 播放，听起来比 1003 效果好很多。

VS1053 的封装引脚和 VS1003 完全兼容，所以如果你以前用的是 VS1003，则只需要把 VS1003 换成 VS1053，就可以实现硬件更新，电路板完全不用修改。不过需要注意的是 VS1003 的 CVDD 是 2.5V，而 VS1053 的 CVDD 是 1.8V，所以你还需要把稳压芯片也变一下，其他都可以照旧了。

VS1053 通过 SPI 接口来接受输入的音频数据流，它可以是一个系统的从机，也可以作为独立的主机。这里我们只把它当成从机使用。我们通过 SPI 口向 VS1053 不停的输入音频数据，它就会自动帮我们解码了，然后从输出通道输出音乐，这时我们接上耳机就能听到所播放的歌曲了。

ALIENTEK 战舰 STM32 开发板，自带了一颗 VS1053 音频编解码芯片，所以，我们直接可以通过开发板来播放各种音频格式，实现一个音乐播放器。战舰 STM32 开发板自带的 VS1053 解码芯片电路原理图，如图 49.1.1 所示：

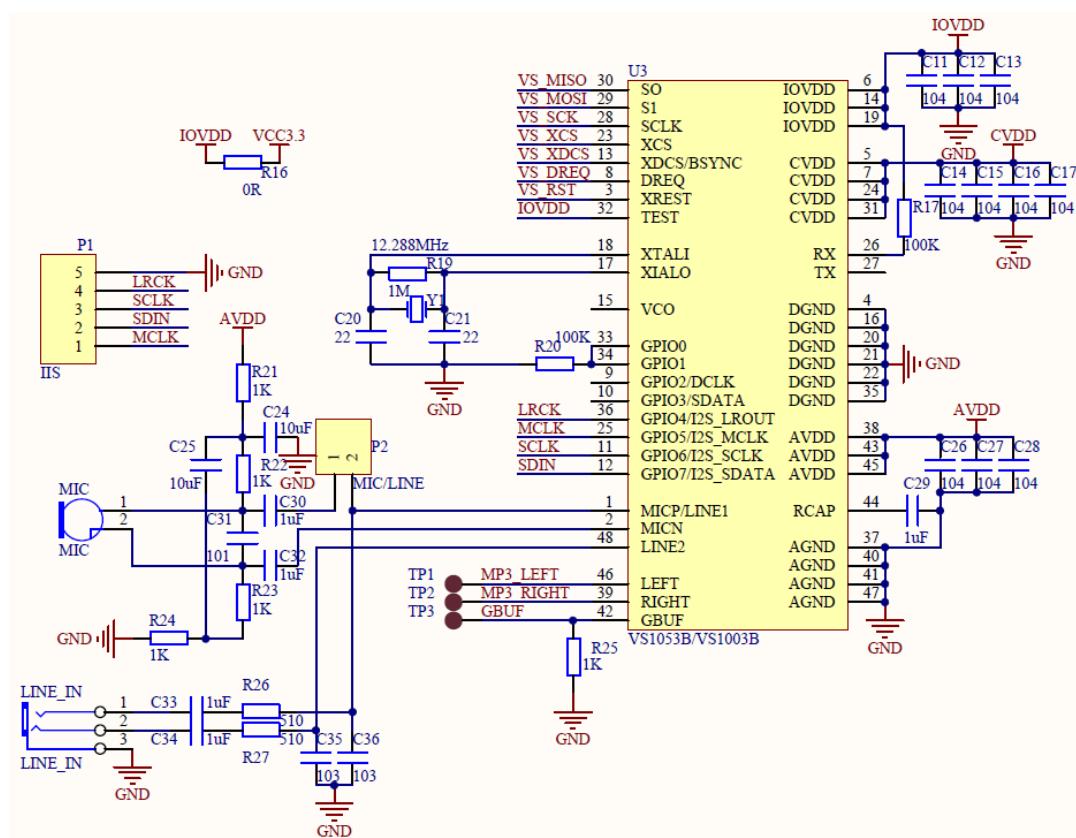


图 49.1.1 ALIENTEK 音频解码模块原理图

VS1053 通过 7 根线同 STM32 连接，他们是：VS_MISO、VS_MOSI、VS_SCK、VS_XCS、VS_XDCS、VS_DREQ 和 VS_RST。这 7 根线同 STM32 的连接关系如表 49.1.1 所示：

芯片	信号线							
VS1053	VS_MISO	VS_MOSI	VS_SCK	VS_XCS	VS_XDCS	VS_DREQ	VS_RST	
STM32F103ZET6	PA6	PA7	PA5	PF7	PF6	PC13	PE6	

表 49.1.1 VS1053 各信号线与 STM32 连接关系

其中 VS_RST 是 VS1053 的复位信号线，低电平有效。VS_DREQ 是一个数据请求信号，用来通知主机，VS1053 可以接收数据与否。VS_MISO、VS_MOSI 和 VS_SCK 则是 VS1053 的 SPI 接口他们在 VS_XCS 和 VS_XDCS 下面来执行不同的操作。从上表可以看出，VS1053 的 SPI 是接在 STM32 的 SPI1 上面的。

VS1053 的 SPI 支持两种模式：1，VS1002 有效模式（即新模式）。2，VS1001 兼容模式。这里我们仅介绍 VS1002 有效模式（此模式也是 VS1053 的默认模式）。表 49.1.2 是在新模式下 VS1053 的 SPI 信号线功能描述：

SDI 管脚	SCI 管脚	描述
XDCS	XCS	低电平有效片选输入，高电平强制使串行接口进入 standby 模式，结束当前操作。高电平也强制使串行输出 SO 变成高阻态。如果 SM_SDISHARE 为 1，不使用 XDCS，但是此信号在 XCS 中产生。
SCK		串行时钟输入。串行时钟也使用内部的寄存器接口主时钟。SCK 可以被门控或是连续的。对任一情况，在 XCS 变为低电平后，SCK 上的第一个上升沿标志着第一位数据被写入。
SI		串行输入，如果片选有效，SI 就在 SCK 的上升沿处采样。
-	SO	串行输出，在读操作时，数据在 SCK 的下降沿处从此脚移出，在写操作时为高阻态。

表 49.1.2 VS1053 新模式下 SPI 口信号线功能



VS1053 的 SPI 数据传送，分为 SDI 和 SCI，分别用来传输数据/命令。SDI 和前面介绍的 SPI 协议一样的，不过 VS1053 的数据传输是通过 DREQ 控制的，主机在判断 DREQ 有效（高电平）之后，直接发送即可（一次可以发送 32 个字节）。

这里我们重点介绍一下 SCI。SCI 串行总线命令接口包含了一个指令字节、一个地址字节和一个 16 位的数据字节。读写操作可以读写单个寄存器，在 SCK 的上升沿读出数据位，所以主机必须在下降沿刷新数据。SCI 的字节数据总是高位在前低位在后的。第一个字节指令字节，只有 2 个指令，也就是读和写，读为 0X03，写为 0X02。

一个典型的 SCI 读时序如图 49.1.2 所示：

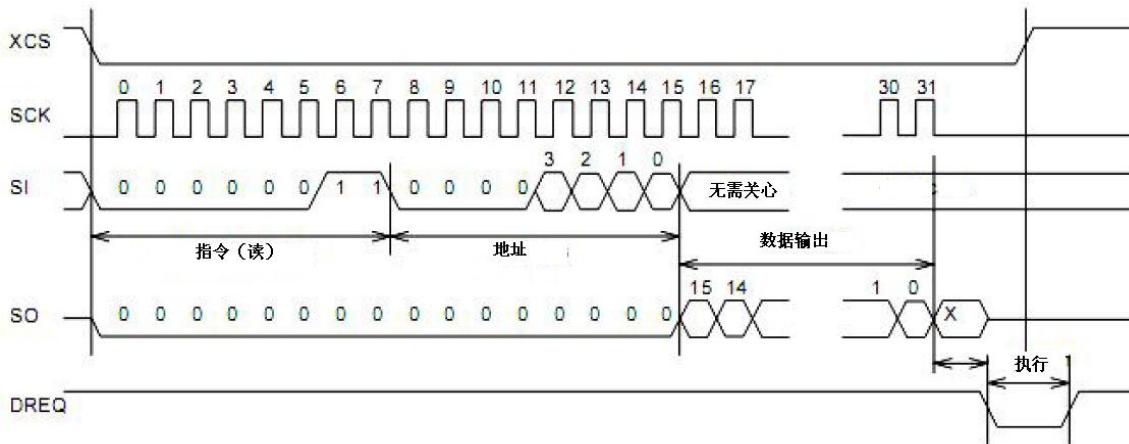


图 49.1.2 SCI 读时序

从图 49.1.2 可以看出，向 VS1053 读取数据，通过先拉低 XCS (VS_XCS)，然后发送读指令 (0X03)，再发送一个地址，最后，我们在 SO 线 (VS_MISO) 上就可以读到输出的数据了。而同时 SI (VS_MOSI) 上的数据将被忽略。

看完了 SCI 的读，我们再来看看 SCI 的写时序，如图 49.1.3 所示：

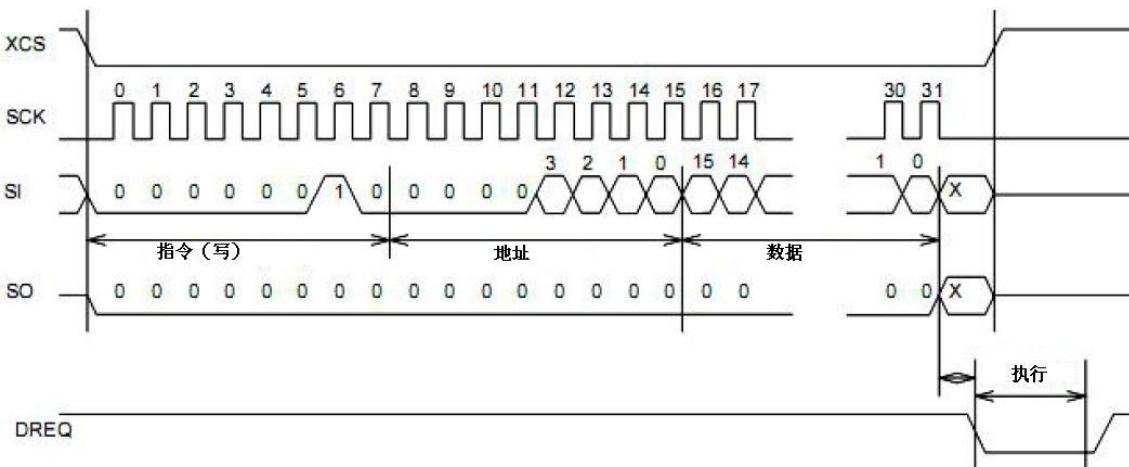


图 49.1.3 SCI 写时序

图 49.1.3 中，其时序和图 49.1.2 基本类似，都是先发指令，再发地址。不过写时序中，我们的指令是写指令 (0X02)，并且数据是通过 SI 写入 VS1053 的，SO 则一直维持低电平。细心的读者可能发现了，在这两个图中，DREQ 信号上都产生了一个短暂的低脉冲，也就是执行时间。这个不难理解，我们在写入和读出 VS1053 的数据之后，它需要一些时间来处理内部的事情，这段时间，是不允许外部打断的，所以，我们在 SCI 操作之前，最好判断一下 DREQ 是



否为高电平，如果不是，则等待 DREQ 变为高。

了解了 VS1053 的 SPI 读写，我们再来看看 VS1053 的 SCI 寄存器，VS1053 的所有 SCI 寄存器如表 49.1.3 所示：

SCI 寄存器				
寄存器	类型	复位值	缩写	描述
0X00	RW	0X0800	MODE	模式控制
0X01	RW	0X000C	STATUS	VS1053 状态
0X02	RW	0X0000	BASS	内置低音/高音控制
0X03	RW	0X0000	CLOCKF	时钟频率+倍频数
0X04	RW	0X0000	DECODE_TIME	解码时间长度(秒)
0X05	RW	0X0000	AUDATA	各种音频数据
0X06	RW	0X0000	WRAM	RAM 写/读
0X07	RW	0X0000	WRAMADDR	RAM 写/读的基址
0X08	R	0X0000	HDATA0	流的数据标头 0
0X09	R	0X0000	HDATA1	流的数据标头 1
0X0A	RW	0X0000	AIADDR	应用程序起始地址
0X0B	RW	0X0000	VOL	音量控制
0X0C	RW	0X0000	AICTRL0	应用控制寄存器 0
0X0D	RW	0X0000	AICTRL1	应用控制寄存器 1
0X0E	RW	0X0000	AICTRL2	应用控制寄存器 2
0X0F	RW	0X0000	AICTRL3	应用控制寄存器 3

表 49.1.3 SCI 寄存器

VS1053 总共有 16 个 SCI 寄存器，这里我们不介绍全部的寄存器，仅仅介绍几个我们在本章需要用到的寄存器。

首先是 MODE 寄存器，该寄存器用于控制 VS1053 的操作，是最关键的寄存器之一，该寄存器的复位值为 0x0800，其实就是默认设置为新模式。表 49.1.4 是 MODE 寄存器的各位描述：

位	0	1	2	3	4	5	6	7
名称	SM_DIFF	SM_LAYER12	SM_RESET	SM_CANCEL	SM_EARSPEAKER_LO	SM_TEST	SM_STREAM	SM_EARSPEAKER_HI
功能	差分	允许MPEG I&II	软件复位	取消当前文件的解码	EarSpeaker 低设定	允许SDI 测试	流模式	EarSpeaker 高设定
描述	0, 正常的同相音频 1, 左通道反相	0, 不允许 1, 允许	0, 不复位 1, 复位	0, 不取消 1, 取消	0, 关闭 1, 激活	0, 禁止 1, 允许	0, 不是 1, 是	0, 关闭 1, 激活
位	8	9	10	11	12	13	14	15
名称	SM_DACT	SM_SDIOORD	SM_SDISHARE	SM_SDINEW	SM_ADPCM	-	SM_LINE1	SM_CLK_RANGE
功能	DCLK的有效边沿	SDI位顺序	共享SPI片选	VS1002本地SPI 模式	ADPCM激活	-	咪/线路1 选择	输入时钟范围
描述	0, 上升沿 1, 下降沿	0, MSB在前 1, MSB在后	0, 不共享 1, 共享	0, 非本地模式 1, 本地模式	0, 不激活 1, 激活	-	0, MICP 1, LINE1	0, 12..13Mhz 1, 24..26Mhz

表 49.1.4 MODE 寄存器各位描述

这个寄存器，我们这里只介绍一下第 2 和第 11 位，也就是 SM_RESET 和 SM_SDINEW。其他位，我们用默认的即可。这里 SM_RESET，可以提供一次软复位，建议在每播放一首歌曲之后，软复位一次。SM_SDINEW 为模式设置位，这里我们选择的是 VS1002 新模式(本地模式)，



所以设置该位为 1（默认的设置）。其他位的详细介绍，请参考 VS1053 的数据手册。

接着我们看看 BASS 寄存器，该寄存器可以用于设置 VS1053 的高低音效。该寄存器的各位描述如表 49.1.5 所示：

名称	位	描述
ST_AMPLITUDE	15: 12	高音控制, 1.5dB 步进 (-8..7 ,为 0 表示关闭)
ST_FREQLIMIT	11: 8	最低频限 1000Hz 步进 (0..15)
SB_AMPLITUDE	7: 4	低音加重, 1dB 步进 (0..15 ,为 0 表示关闭)
SB_FREQLIMIT	3: 0	最低频限 10Hz 步进 (2..15)

表 49.1.5 BASS 寄存器各位描述

通过这个寄存器以上位的一些设置，我们可以随意配置自己喜欢的音效（其实就是高低音的调节）。VS1053 的 EarSpeaker 效果则由 MODE 寄存器控制，请参考表 49.1.4。

接下来，我们介绍一下 CLOCKF 寄存器，这个寄存器用来设置时钟频率、倍频等相关信息，该寄存器的各位描述如表 49.1.6 所示：

CLOCKF 寄存器			
位	15:13	12:11	10:0
名称	SC_MULT	SC_ADD	SC_FREQ
描述	时钟倍频数	允许倍频	时钟频率
说明	CLKI=XTALI × (SC_MULT×0.5+1)	倍频增量 =SC_ADD*0.5	当时钟频率不为12.288M 时，外部时钟的频率。 外部时钟为12.288时，此部分设置为0即可

表 49.1.6 CLOCKF 寄存器各位描述

此寄存器，重点说明 SC_FREQ，SC_FREQ 是以 4Khz 为步进的一个时钟寄存器，当外部时钟不是 12.288M 的时候，其计算公式为：

$$\text{SC_FREQ}=(\text{XTALI}-8000000)/4000$$

式中为 XTALI 的单位为 Hz。表 49.1.6 中 CLKI 是内部时钟频率，XTALI 是外部晶振的时钟频率。由于我们使用的是 12.288M 的晶振，在这里设置此寄存器的值为 0X9800，也就是设置内部时钟频率为输入时钟频率的 3 倍，倍频增量为 1.0 倍。

接下来，我们看看 DECODE_TIME 这个寄存器。该寄存器是一个存放解码时间的寄存器，以秒钟为单位，我们通过读取该寄存器的值，就可以得到解码时间了。不过它是一个累计时间，所以我们需要在每首歌播放之前把它清空一下，以得到这首歌的准确解码时间。

HDATA0 和 HDTA1 是两个数据流头寄存器，不同的音频文件，读出来的值意义不一样，我们可以通过这两个寄存器来获取音频文件的码率，从而可以计算音频文件的总长度。这两个寄存器的详细介绍，请参考 VS1053 的数据手册。

最后我们介绍一下 VOL 这个寄存器，该寄存器用于控制 VS1053 的输出音量，该寄存器可以分别控制左右声道的音量，每个声道的控制范围为 0~254，每个增量代表 0.5db 的衰减，所以该值越小，代表音量越大。比如设置为 0X0000 则音量最大，而设置为 0XFEFE 则音量最小。注意：如果设置 VOL 的值为 0xFFFF，将使芯片进入掉电模式！

关于 VS1053 的介绍，我们就介绍到这里，更详细的介绍请看 VS1053 的数据手册。接下来我们说说如何控制通过最简单的步骤，来控制 VS1053 播放一首歌曲。

1) 复位 VS1053

这里包括了硬复位和软复位，是为了让 VS1053 的状态回到原始状态，准备解码下一首歌



曲。这里建议大家在每首歌曲播放之前都执行一次硬件复位和软件复位，以便更好的播放音乐。

2) 配置 VS1053 的相关寄存器

这里我们配置的寄存器包括 VS1053 的模式寄存器（MODE）、时钟寄存器（CLOCKF）、音调寄存器（BASS）、音量寄存器（VOL）等。

3) 发送音频数据

当经过以上两步配置以后，我们剩下来要做的事情，就是往 VS1053 里面扔音频数据了，只要是 VS1053 支持的音频格式，直接往里面丢就可以了，VS1053 会自动识别，并进行播放。不过发送数据要在 DREQ 信号的控制下有序的进行，不能乱发。这个规则很简单：只要 DREQ 变高，就向 VS1053 发送 32 个字节。然后继续等待 DREQ 变高，直到音频数据发送完。

经过以上三步，我们就可以播放音乐了。这一部分就先介绍到这里。

49.2 硬件设计

本章实验功能简介：开机先检测字库是否存在，如果检测无问题，则对 VS1053 进行 RAM 测试和正弦测试，测试完后开始循环播放 SD 卡 MUSIC 文件夹里面的歌曲（必须在 SD 卡根目录建立一个 MUSIC 文件夹，并存放歌曲在里面），在 TFTLCD 上显示歌曲名字、播放时间、歌曲总时间、歌曲总数目、当前歌曲的编号等信息。KEY0 用于选择下一曲，KEY2 用于选择上一曲，WK_UP 和 KEY1 用来调节音量。DS0 还是用于指示程序运行状态，DS1 用于指示 VS1053 正在初始化。

本实验用到的资源如下：

- 1) 指示灯 DS0
- 2) 四个按键（WK_UP/KEY0/KEY1/KEY2）
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡
- 6) SPI FLASH
- 7) VS1053
- 8) 74HC4052
- 9) TDA1308

这些硬件我们都已经介绍过了，其中 74HC4052 和 TDA1308 分别是用作音频选择和耳机驱动，在第四十章，我们已经介绍过。本章，我们使用的 VS1053 同 STM32 的连接关系详见表 49.1.1。

本实验，大家需要准备 1 个 SD 卡（在里面新建一个 MUSIC 文件夹，并存放一些歌曲在 MUSIC 文件夹下）和一个耳机，分别插入 SD 卡接口和耳机接口，然后下载本实验就可以通过耳机来听歌了。

49.3 软件设计

打开 MP3 播放器工程，可以看到我们在工程中新建了一个组 APP，下面加入了 mp3 解码文件 mp3player.c 以及头文件 mp3player.h。同时我们还在 HARDWARE 组中加入了文件 VS10XX.c，以及头文件 VS10XX.h 和 flac.h。

首先打开 VS10XX.c，里面的代码我们不一一贴出了，这里挑几个重要的函数给大家介绍一下，首先要介绍的是 VS_Soft_Reset，该函数用于软复位 VS1003，其代码如下：

```
//软复位 VS10XX
```



```
void VS_Soft_Reset(void)
{
    u8 retry=0;
    while(VS_DQ==0); //等待软件复位结束
    VS_SPI_ReadWriteByte(0Xff); //启动传输
    retry=0;
    while(VS_RD_Reg(SPI_MODE)!=0x0800) // 软件复位,新模式
    {
        VS_WR_Cmd(SPI_MODE,0x0804); // 软件复位,新模式
        delay_ms(2); //等待至少 1.35ms
        if(retry++>100)break;
    }
    while(VS_DQ==0); //等待软件复位结束
    retry=0;
    while(VS_RD_Reg(SPI_CLOCKF)!=0X9800) //设置 VS10XX 的时钟,3 倍频 ,1.5xADD
    {
        VS_WR_Cmd(SPI_CLOCKF,0X9800); //设置 VS10XX 的时钟,3 倍频 ,1.5xADD
        if(retry++>100)break;
    }
    delay_ms(20);
}
```

该函数比较简单，先配置一下 VS1053 的模式顺便执行软复位操作，在软复位结束之后，再设置好时钟，完成一次软复位。接下来，我们介绍一下 VS_WR_Cmd 函数，该函数用于向 VS1003 写命令，代码如下：

```
//向 VS10XX 写命令
//address:命令地址
//data:命令数据
void VS_WR_Cmd(u8 address,u16 data)
{
    while(VS_DQ==0); //等待空闲
    VS_SPI_SpeedLow(); //低速
    VS_XDCS=1; ;
    VS_XCS=0;
    VS_SPI_ReadWriteByte(VS_WRITE_COMMAND); //发送 VS10XX 的写命令
    VS_SPI_ReadWriteByte(address); //地址
    VS_SPI_ReadWriteByte(data>>8); //发送高八位
    VS_SPI_ReadWriteByte(data); //第八位
    VS_XCS=1;
    VS_SPI_SpeedHigh(); //高速
}
```

该函数用于向 VS1053 发送命令，这里要注意 VS1053 的写操作比读操作快（写 1/4 CLKI，读 1/7 CLKI），虽然说写寄存器最快可以到 1/4CLKI，但是经实测在 1/4CLKI 的时候会出错，所以在写寄存器的时候最好把 SPI 速度调慢点，然后在发送音频数据的时候，就可以 1/4CLKI



的速度了。有写命令的函数，当然也有读命令的函数了。VS_RD_Reg 用于读取 VS1053 的寄存器的内容。该函数代码如下：

```
//读 VS10XX 的寄存器
//address: 寄存器地址
//返回值: 读到的值
//注意不要用倍速读取,会出错
u16 VS_RD_Reg(u8 address)
{
    u16 temp=0;
    while(VS_DQ==0);//非等待空闲状态
    VS_SPI_SpeedLow();//低速
    VS_XDCS=1;
    VS_XCS=0;
    VS_SPI_ReadWriteByte(VS_READ_COMMAND); //发送 VS10XX 的读命令
    VS_SPI_ReadWriteByte(address);           //地址
    temp=VS_SPI_ReadWriteByte(0xff);          //读取高字节
    temp=temp<<8;
    temp+=VS_SPI_ReadWriteByte(0xff);          //读取低字节
    VS_XCS=1;
    VS_SPI_SpeedHigh();//高速
    return temp;
}
```

该函数的作用和 VS_WR_Cmd 的作用基本相反，用于读取寄存器的值。VS10XX.c 的剩余代码、VS10XX.h 以及 flac.h 的代码，这里就不贴出来了，其中 flac.h 仅仅用来存储播放 flac 格式所需要的 patch 文件，以支持 flac 解码。大家可以去光盘查看他们的详细源码。保存 VS10XX.c、VS10XX.h 和 flac.h 三个文件。把 VS10XX.c 加入到 HARDWARE 组下。然后我们打开 mp3player.c，该文件我们仅介绍一个函数，其他代码请看光盘的源码。这里要介绍的是 mp3_play_song 函数，该函数代码如下：

```
//播放一曲指定的歌曲
//返回值:0,正常播放完成; 1,下一曲; 2,上一曲; 0XFF,出现错误了;
u8 mp3_play_song(u8 *pname)
{
    FIL* fmp3;
    u16 br;
    u8 res,rval;
    u8 *databuf;
    u16 i=0;
    u8 key;
    rval=0;
    fmp3=(FIL*)mymalloc(SRAMIN,sizeof(FIL)); //申请内存
    databuf=(u8*)mymalloc(SRAMIN,4096);        //开辟 4096 字节的内存区域
    if(databuf==NULL||fmp3==NULL)rval=0XFF; //内存申请失败.
    if(rval==0)
```



```
{  
    VS_Restart_Play();           //重启播放  
    VS_Set_All();                //设置音量等信息  
    VS_Reset_DecodeTime();       //复位解码时间  
    res=f_typedell(pname);        //得到文件后缀  
    if(res==0x4c)//如果是 flac,加载 patch  
    {  
        VS_Load_Patch((u16*)vs1053b_patch,VS1053B_PATCHLEN);  
    }  
    res=f_open(fmp3,(const TCHAR*)pname,FA_READ);//打开文件  
    if(res==0)//打开成功.  
    {  
        VS_SPI_SpeedHigh(); //高速  
        while(rval==0)  
        {  
            res=f_read(fmp3,databuf,4096,(UINT*)&br); //读出 4096 个字节  
            i=0;  
            do//主播放循环  
            {  
                if(VS_Send_MusicData(databuf+i)==0)//给 VS10XX 发送音频数据  
                {  
                    i+=32;  
                }else  
                {  
                    key=KEY_Scan(0);  
                    switch(key)  
                    {  
                        case KEY_RIGHT: rval=1; break; //下一曲  
                        case KEY_LEFT: rval=2; break; //上一曲  
                        case KEY_UP:    //音量增加  
                            if(vsset.mvol<250)  
                            {  
                                vsset.mvol+=5;  
                                VS_Set_Vol(vsset.mvol);  
                            }else vsset.mvol=250;  
                            mp3_vol_show((vsset.mvol-100)/5); //音量限制在: 100~  
                            //250,显示时,按照公式(vol-100)/5,显示,也就是 0~30  
                            break;  
                        case KEY_DOWN://音量减  
                            if(vsset.mvol>100)  
                            {  
                                vsset.mvol-=5;  
                                VS_Set_Vol(vsset.mvol);  
                            }  
                    }  
                }  
            }  
        }  
    }  
}
```



```
        }else vsset.mvol=100;
        mp3_vol_show((vsset.mvol-100)/5);
        break;
    }
    mp3_msg_show(fmp3->fsize);//显示信息
}
}while(i<4096);//循环发送 4096 个字节
if(br!=4096||res!=0) {rval=0; break;}//读完了.
}
f_close(fmp3);
}else rval=0XFF;//出现错误
}
myfree(SRAMIN,databuf);
myfree(SRAMIN,fmp3);
return rval;
}
```

该函数，就是我们解码 MP3 的核心函数了，该函数在初始化 VS1053 后，根据文件格式选择是否加载 patch（如果是 flac 格式，则需要加载 patch），最后在死循环里面等待 DREQ 信号的到来，每次 VS_DQ 变高，就通过 VS_Send_MusicData 函数向 VS1053 发送 32 个字节，直到整个文件读完。此段代码还包含了对按键的处理（音量调节、上一首、下一首）及当前播放的歌曲的一些状态（码率、播放时间、总时间）显示。

mp3player.c 的其他代码和 mp3player.h 在这里就不详细介绍了，请大家直接参考光盘源码。最后，我们看看 main.c 文件的内容：

```
int main(void)
{
    delay_init();          //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
    LCD_Init();            //LCD 初始化
    KEY_Init();            //KEY 初始化
    Audiosel_Init();       //初始化音源选择
    usmart_dev.init(72);   //usmart 初始化
    mem_init(SRAMIN);     //初始化内部内存池
    VS_Init();             //初始化 VS1053
    exfun_init();          //为 fatfs 相关变量申请内存
    f_mount(0,fs[0]);      //挂载 SD 卡
    f_mount(1,fs[1]);      //挂载 FLASH.
    POINT_COLOR=RED;
    while(font_init())      //检查字库
    {
        LCD_ShowString(60,50,200,16,16,"Font Error!");
        delay_ms(200);
    }
}
```



```
LCD_Fill(60,50,240,66,WHITE); //清除显示
}
Show_Str(60,50,200,16,"战舰 STM32 开发板",16,0);
Show_Str(60,70,200,16,"音乐播放器实验",16,0);
Show_Str(60,90,200,16,"广州星翼电子",16,0);
Show_Str(60,110,200,16,"2012 年 9 月 20 日",16,0);
Show_Str(60,130,200,16,"KEY0:NEXT KEY2:PREV",16,0);
Show_Str(60,150,200,16,"KEY_UP:VOL+ KEY1:VOL-",16,0);
while(1)
{
    Audiosel_Set(0); //音频通道选择 MP3 音源
    LED1=0;
    Show_Str(60,170,200,16,"存储器测试...",16,0);
    printf("Ram Test:0X%04X\r\n",VS_Ram_Test()); //打印 RAM 测试结果
    Show_Str(60,170,200,16,"正弦波测试...",16,0);
    VS_Sine_Test();
    Show_Str(60,170,200,16,"<<音乐播放器>>",16,0);
    LED1=1;
    mp3_play();
}
}
```

该函数先检测外部 flash 是否存在字库，然后选择音频通道为 MP3 音源，之后执行 VS1053 的 RAM 测试和正弦测试，这两个测试结束后，调用 mp3_play 函数开始播放 SD 卡 MUSIC 文件夹里面的音乐。软件部分就介绍到这里。

49.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，程序先执行字库监测，然后对 VS1053 进行 RAM 测试和正弦测试。

当检测到 SD 卡根目录的 MUSIC 文件夹有有效音频文件（VS1053 所支持的格式）的时候，就开始自动播放歌曲了，如图 49.4.1 所示：



图 49.4.1 MP3 播放中

从上图可以看出，当前正在播放第 4 首歌曲，总共 4 首歌曲，歌曲名、播放时间、总时长、码率、音量等信息等也都有显示。此时 DS0 会随着音乐的播放而闪烁，2 秒闪烁一次。

只要我们在开发板的 PHONE 端子插入耳机，就能听到歌曲的声音了。同时，我们可以通过按 KEY0 和 KEY2 来切换下一曲和上一曲，通过 WK_UP 按键来控制音量增加，通过 KEY1 控制音量减小。

本实验，我们还可以通过 USMART 来测试 VS1053 的其他功能，通过将 VS10XX.c 里面的部分函数加入 USMART 管理，我们可以很方便的设置/获取 VS1053 各种参数，达到验证测试的目的。有兴趣的朋友，可以实验测试一下。

至此，我们就完成了一个简单的 MP3 播放器了，在此基础上进一步完善，就可以做出一个比较实用的 MP3 了。大家可以自己发挥想象，做出一个你心仪的 MP3。



第五十章 录音机实验

上一章，我们实现了一个简单的音乐播放器，本章我们将在上一章的基础上，实现一个简单的录音机，实现 WAV 录音。本章分为如下几个部：

- 50.1 WAV 简介
- 50.2 硬件设计
- 50.3 软件设计
- 50.4 下载验证



50.1 WAV 简介

WAV 即 WAVE 文件，WAV 是计算机领域最常用的数字化声音文件格式之一，它是微软专门为 Windows 系统定义的波形文件格式（Waveform Audio），由于其扩展名为“*.wav”。它符合 RIFF(Resource Interchange File Format)文件规范，用于保存 Windows 平台的音频信息资源，被 Windows 平台及其应用程序所广泛支持，该格式也支持 MSADPCM, CCITT A LAW 等多种压缩运算法，支持多种音频数字，取样频率和声道，标准格式化的 WAV 文件和 CD 格式一样，也是 44.1K 的取样频率，16 位量化数字，因此在声音文件质量和 CD 相差无几！

ALIENTEK 战舰 STM32 开发板板载的 VS1053 支持 2 种格式的 WAV 录音：PCM 格式或者 IMA ADPCM 格式，其中 PCM（脉冲编码调制）是最基本的 WAVE 文件格式，这种文件直接存储采样的声音数据没有经过任何的压缩。而 IMA ADPCM 则是使用了压缩算法，压缩比率为 4:1。

本章，我们主要讨论 PCM，因为这个最简单。我们将利用 VS1053 实现 16 位，8Khz 采样率的单声道 WAV 录音(PCM 格式)。要想实现 WAV 录音得先了解一下 WAV 文件的格式，WAVE 文件是由若干个 Chunk 组成的。按照在文件中的出现位置包括：RIFF WAVE Chunk、Format Chunk、Fact Chunk(可选)和 Data Chunk。每个 Chunk 由块标识符、数据大小和数据三部分组成，如图 50.1.1 所示：



图 50.1.1 Chunk 结构示意图

其中块标识符由 4 个 ASCII 码构成，数据大小则标出紧跟其后的数据的长度(单位为字节)，注意这个长度不包含块标识符和数据大小的长度，即不包含最前面的 8 个字节。所以实际 Chunk 的大小为数据大小加 8。

首先，我们来看看 RIFF 块（RIFF WAVE Chunk），该块以“RIFF”作为标示，紧跟 wav 文件大小（该大小是 wav 文件的总大小-8），然后数据段为“WAVE”，表示是 wav 文件。RIFF 块的 Chunk 结构如下：

```
//RIFF 块
typedef __packed struct
{
    u32 ChunkID;           //chunk id;这里固定为"RIFF",即 0X46464952
    u32 ChunkSize ;        //集合大小;文件总大小-8
    u32 Format;           //格式;WAVE,即 0X45564157
}ChunkRIFF ;
```

接着，我们看看 Format 块（Format Chunk），该块以“fmt ”作为标示（注意有个空格！），一般情况下，该段的大小为 16 个字节，但是有些软件生成的 wav 格式，该部分可能有 18 个字节，含有 2 个字节的附加信息。Format 块的 Chunk 结构如下：

```
//fmt 块
typedef __packed struct
{
    u32 ChunkID;           //chunk id;这里固定为"fmt ",即 0X20746D66
    u32 ChunkSize ;        //子集合大小(不包括 ID 和 Size);这里为:20.
```



```

u16 AudioFormat;      //音频格式;0X10,表示线性 PCM;0X11 表示 IMA ADPCM
u16 NumOfChannels;   //通道数量;1,表示单声道;2,表示双声道;
u32 SampleRate;      //采样率;0X1F40,表示 8Khz
u32 ByteRate;        //字节速率;
u16 BlockAlign;      //块对齐(字节);
u16 BitsPerSample;   //单个采样数据大小;4 位 ADPCM,设置为 4

}ChunkFMT;

```

接下来，我们再看看 Fact 块（Fact Chunk），该块为可选块，以“fact”作为标示，不是每个 WAV 文件都有，在非 PCM 格式的文件中，一般会在 Format 结构后面加入一个 Fact 块，该块 Chunk 结构如下：

```

//fact 块
typedef __packed struct
{
    u32 ChunkID;          //chunk id;这里固定为"fact",即 0X74636166;
    u32 ChunkSize;        //子集合大小(不包括 ID 和 Size);这里为:4.
    u32 DataFactSize;     //数据转换为 PCM 格式后的大小

}ChunkFACT;

```

DataFactSize 是这个 Chunk 中最重要的数据，如果这是某种压缩格式的声音文件，那么从这里就可以知道他解压缩后的大小。对于解压时的计算会有很大的好处！不过本章我们使用的是 PCM 格式，所以不存在这个块。

最后，我们来看看数据块（Data Chunk），该块是真正保存 wav 数据的地方，以“data”作为该 Chunk 的标示。然后是数据的大小。紧接着就是 wav 数据。根据 Format Chunk 中的声道数以及采样 bit 数，wav 数据的 bit 位置可以分成如表 50.1.1 所示的几种形式：

单声道	取样 1	取样 2	取样 3	取样 4
8 位量化	声道 0	声道 0	声道 0	声道 0
双声道	取样 1		取样 2	
8 位量化	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)
单声道	取样 1		取样 2	
16 位量化	声道 0(低字节)	声道 0(高字节)	声道 0(低字节)	声道 0(高字节)
双声道	取样 1			
16 位量化	声道 0 (左, 低字节)	声道 0 (左, 高字节)	声道 1 (右, 低字节)	声道 1 (右, 高字节)

表 50.1.1 WAVE 文件数据采样格式

本章，我们采用的是 16 位，单声道，所以每个取样为 2 个字节，低字节在前，高字节在后。数据块的 Chunk 结构如下：

```

//data 块
typedef __packed struct
{
    u32 ChunkID;          //chunk id;这里固定为"data",即 0X61746164
    u32 ChunkSize;        //子集合大小(不包括 ID 和 Size);文件大小-60.

}ChunkDATA;

```

通过以上学习，我们对 WAVE 文件有了个大概了解。接下来，我们看看如何使用 VS1053

实现 WAV (PCM 格式) 录音。

激活 PCM 录音

VS1053 激活 PCM 录音需要设置的寄存器和相关位如表 50.1.2 所示:

寄存器	位域	说明
SCI_MODE	2, 12, 14	开始 ADPCM 模式, 选择: 咪/线路1
SCI_AICTRL0	15..0	采样率 8000..48000 Hz (在录音启动时读取的)
SCI_AICTRL1	15..0	录音增益 (1024 = 1x) 或 0 是自动增益控制 (AGC)
SCI_AICTRL2	15..0	自动增益放大器的最大值 (1024 = 1x, 65535 = 64x)
SCI_AICTRL3	1..0 2 15..3	0=联合立体声(共用 AGC), 1=双声道(各自的 AGC), 2=左通道, 3=右通道 0=IMA ADPCM 模式, 1=线性 PCM 模式 保留, 设置为 0

表 50.1.2 VS1053 激活 PCM 录音相关寄存器

通过设置 SCI_MODE 寄存器的 2、12、14 位, 来激活 PCM 录音, SCI_MODE 的各位描述见表 49.1.4(也可以参考 VS1053 的数据手册)。SCI_AICTRL0 寄存器用于设置采样率, 我们本章用的是 8K 的采样率, 所以设置这个值为 8000 即可。SCI_AICTRL1 寄存器用于设置 AGC, 1024 相当于数字增加 1, 这里建议大家设置 AGC 在 4 (4*1024) 左右比较合适。SCI_AICTRL2 用于设置自动 AGC 的时候的最大值, 当设置为 0 的时候表示最大 64(65536), 这个大家按自己的需要设置即可。最后, SCI_AICTRL3, 我们本章用到的是咪头线性 PCM 单声道录音, 所以设置该寄存器值为 6。

通过这几个寄存器的设置, 我们就激活 VS1053 的 PCM 录音了。不过, VS1053 的 PCM 录音有一个小 BUG, 必须通过加载 patch 才能解决, 如果不加载 patch, 那么 VS1053 是不输出 PCM 数据的, VLSI 提供了我们这个 patch, 只需要通过软件加载即可。

读取 PCM 数据

在激活了 PCM 录音之后, SCI_HDAT0 和 SCI_HDAT1 有了新的功能。VS1053 的 PCM 采样缓冲区由 1024 个 16 位数据组成, 如果 SCI_HDAT1 大于 0, 则说明可以从 SCI_HDAT0 读取至少 SCI_HDAT1 个 16 位数据, 如果数据没有被及时读取, 那么将溢出, 并返回空的状态。

注意, 如果 SCI_HDAT1 ≥ 896 , 最好等待缓冲区溢出, 以免数据混叠。所以, 对我们来说, 只需要判断 SCI_HDAT1 的值非零, 然后从 SCI_HDAT0 读取对应长度的数据, 即完成一次数据读取, 以此循环, 即可实现 PCM 数据的持续采集。

最后, 我们看看本章实现 WAV 录音需要经过哪些步骤:

1) 设置 VS1053 PCM 采样参数

这一步, 我们要设置 PCM 的格式 (线性 PCM)、采样率 (8K)、位数 (16 位)、通道数 (单声道) 等重要参数, 同时还要选择采样通道 (咪头), 还包括 AGC 设置等。可以说这里的设置直接决定了我们 wav 文件的性质。

2) 激活 VS1053 的 PCM 模式, 加载 patch

通过激活 VS1053 的 PCM 格式, 让其开始 PCM 数据采集, 同时, 由于 VS1053 的 BUG, 我们需要加载 patch, 以实现正常的 PCM 数据接收。

3) 创建 WAV 文件, 并保存 wav 头

在前两部设置成功之后, 我们即可正常的从 SCI_HDAT0 读取我们需要的 PCM 数据了, 不过在这之前, 我们需要先在创建一个新的文件, 并写入 wav 头, 然后才能开始写入我们的 PCM 数据。

4) 读取 PCM 数据



经过前面几步的处理，这一步就比较简单了，只需要不停的从 SCI_HDAT0 读取数据，然后存入 wav 文件即可，不过这里我们还需要做文件大小统计，在最后的时候写入 wav 头里面。

5) 计算整个文件大小，重新保存 wav 头并关闭文件

在结束录音的时候，我们必须知道本次录音的大小（数据大小和整个文件大小），然后更新 wav 头，重新写入文件，最后因为 FATFS，在文件创建之后，必须调用 f_close，文件才会真正体现在文件系统里面，否则是不会写入的！所以最后还需要调用 f_close，以保存文件。

50.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后初始化 VS1053，进行 RAM 测试和正弦测试，之后，检测 SD 卡根目录是否存在 RECORDER 文件夹，如果不存在则创建，如果创建失败，则报错。在找到 SD 卡的 RECORDER 文件夹后，即设置 VS1053 进入录音模式，此时可以在耳机听到 VS1053 采集的音频。KEY0 用于开始/暂停录音，KEY2 用于保存并停止录音，WK_UP 用于 AGC 增加、KEY1 用于 AGC 减小，TPAD 用于播放最近一次的录音。当我们按下 KEY0 的时候，可以在屏幕上看到录音文件的名字，以及录音时间，然后通过 KEY2 可以保存该文件，同时停止录音（文件名和时间也都将清零），在完成一个录音后，我们可以通过按 TPAD 按键，来试听刚刚的录音。DS0 用于提示程序正在运行，DS1 用于指示当前是否处于录音暂停状态。

本实验用到的资源如下：

- 1) 指示灯 DS0 和 DS1
- 2) 五个按键 (WK_UP/KEY0/KEY1/KEY2/TPAD)
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡
- 6) SPI FLASH
- 7) VS1053
- 8) 74HC4052
- 9) TDA1308

本章用到的硬件资源同上一章基本一样，就多了一个 TPAD 按键，用于播放最近一次录音。

本实验，大家需要准备 1 个 SD 卡和一个耳机，分别插入 SD 卡接口和耳机接口，然后下载本实验就可以实现一个简单的录音机了。

50.3 软件设计

打开录音机实验工程可以发现，我们在工程中添加了 recorder.c 文件，以及其头文件 recorder.h 文件。

因为 recorder.c 代码比较多，我们这里仅介绍其中的三个函数，首先是设置 VS1053 进入 PCM 模式的函数：recoder_enter_rec_mode，该函数代码如下：

```
//进入 PCM 录音模式
//agc:0,自动增益.1024 相当于 1 倍,512 相当于 0.5 倍,最大值 65535=64 倍
void recoder_enter_rec_mode(u16 agc)
{
    //如果是 IMA ADPCM,采样率计算公式如下:
```



```

//采样率=CLKI/256*d;
//假设 d=0,并 2 倍频,外部晶振为 12.288M.那么 Fc=(2*12288000)/256*6=16Khz
//如果是线性 PCM,采样率直接就写采样值
VS_WR_Cmd(SPI_BASS,0x0000);
VS_WR_Cmd(SPI_AICTRL0,8000); //设置采样率,设置为 8Khz
VS_WR_Cmd(SPI_AICTRL1,agc);
//设置增益,0,自动增益.1024 相当于 1 倍,512 相当于 0.5 倍,最大值 65535=64 倍
VS_WR_Cmd(SPI_AICTRL2,0); //设置增益最大值,0,代表最大值 65536=64X
VS_WR_Cmd(SPI_AICTRL3,6); //左通道(MIC 单声道输入)
VS_WR_Cmd(SPI_CLOCKF,0X2000);
//设置 VS10XX 的时钟,MULT:2 倍频;ADD:不允许;CLK:12.288Mhz
VS_WR_Cmd(SPI_MODE,0x1804); //MIC,录音激活
delay_ms(5); //等待至少 1.35ms
VS_Load_Patch((u16*)wav_plugin,40);//VS1053 的 WAV 录音需要 patch
}

```

该函数就是用我们前面介绍的方法,激活 VS1053 的 PCM 模式,本章,我们使用的是 8Khz 采样率,16 位单声道线性 PCM 模式,AGC 通过函数参数设置。最后加载 patch(用于修复 VS1053 录音 BUG)。

第二个函数是初始化 wav 头的函数: recoder_wav_init, 该函数代码如下:

```

//初始化 WAV 头.
void recoder_wav_init(__WaveHeader* wavhead) //初始化 WAV 头
{
    wavhead->riff.ChunkID=0X46464952; //RIFF"
    wavhead->riff.ChunkSize=0; //还未确定,最后需要计算
    wavhead->riff.Format=0X45564157; //WAVE"
    wavhead->fmt.ChunkID=0X20746D66; //fmt "
    wavhead->fmt.ChunkSize=16; //大小为 16 个字节
    wavhead->fmt.AudioFormat=0X01; //0X01,表示 PCM;0X01,表示 IMA ADPCM
    wavhead->fmt.NumOfChannels=1; //单声道
    wavhead->fmt.SampleRate=8000; //8Khz 采样率 采样速率
    wavhead->fmt.ByteRate=wavhead->fmt.SampleRate*2;//16 位,即 2 个字节
    wavhead->fmt.BlockAlign=2; //块大小,2 个字节为一个块
    wavhead->fmt.BitsPerSample=16; //16 位 PCM
    wavhead->data.ChunkID=0X61746164; //"data"
    wavhead->data.ChunkSize=0; //数据大小,还需要计算
}

```

该函数初始化 wav 头的绝大部分数据,这里我们设置了该 wav 文件为 8Khz 采样率,16 位线性 PCM 格式,另外由于录音还未真正开始,所以文件大小和数据大小都还是未知的,要等录音结束才能知道。该函数 __WaveHeader 结构体就是由前面介绍的三个 Chunk 组成,结构为:

```

//wav 头
typedef __packed struct
{
    ChunkRIFF riff; //riff 块

```



```
ChunkFMT fmt; //fmt 块
//ChunkFACT fact; //fact 块 线性 PCM,没有这个结构体
ChunkDATA data; //data 块
}__WaveHeader;
```

最后，我们介绍 recoder_play 函数，是录音机实现的主循环函数，该函数代码如下：

```
//录音机
//所有录音文件,均保存在 SD 卡 RECORDER 文件夹内.
u8 recoder_play(void)
{
    u8 res, key, rval=0;
    __WaveHeader *wavhead=0;
    u32 sectorsize=0;
    FIL* f_rec=0; //文件
    DIR recdir; //目录
    u8 *recbuf; //数据内存
    u16 w;
    u16 idx=0;
    u8 rec_sta=0; //录音状态
                    //#[7]:0,没有录音;1,有录音;
                    //#[6:1]:保留
                    //#[0]:0,正在录音;1,暂停录音;
    u8 *pname=0;
    u8 timecnt=0; //计时器
    u32 recsec=0; //录音时间
    u8 recagc=4; //默认增益为 4
    while(f_opendir(&recdir,"0:/RECORDER"))//打开录音文件夹
    {
        Show_Str(60,230,240,16,"RECORDER 文件夹错误!",16,0); delay_ms(200);
        LCD_Fill(60,230,240,246,WHITE); delay_ms(200); //清除显示
        f_mkdir("0:/RECORDER");//创建该目录
    }
    f_rec=(FIL *)mymalloc(SRAMIN,sizeof(FIL)); //开辟 FIL 字节的内存区域
    if(f_rec==NULL)rval=1; //申请失败
    wavhead=(__WaveHeader*)mymalloc(SRAMIN,sizeof(__WaveHeader));
    //开辟 __WaveHeader 字节的内存区域
    if(wavhead==NULL)rval=1;
    recbuf=mymalloc(SRAMIN,512);
    if(recbuf==NULL)rval=1;
    pname=mymalloc(SRAMIN,30);
    //申请 30 个字节内存,存放路径+名字, 类似"0:RECORDER/REC00001.wav"
    if(pname==NULL)rval=1;
    if(rval==0) //内存申请 OK
    {
```



```
recoder_enter_rec_mode(1024*recagc);
while(VS_RD_Reg(SPI_HDAT1)>>8);      //等到 buf 较为空闲再开始
recoder_show_time(recsec);             //显示时间
recoder_show_agc(recagc);              //显示 agc
pname[0]=0;                           //pname 没有任何文件名
while(rval==0)
{
    key=KEY_Scan(0);
    switch(key)
    {
        case KEY_LEFT: //STOP&SAVE
            if(rec_sta&0X80)//有录音
            {
                wavhead->riff.ChunkSize=sectorsize*512+36; //文件大小-8;
                wavhead->data.ChunkSize=sectorsize*512;      //数据大小
                f_lseek(f_rec,0);                            //偏移到文件头.
                f_write(f_rec,(const void*)wavhead,sizeof(__WaveHeader),
                         &bw);//写入头数据
                f_close(f_rec); sectorsize=0;
            }
            rec_sta=0; recsec=0; LED1=1; //关闭 DS1
            LCD_Fill(60,230,240,246,WHITE);
            //清除显示,清除之前显示的录音文件名
            recoder_show_time(recsec); //显示时间
            break;
        case KEY_RIGHT://REC/PAUSE
            if(rec_sta&0X01)//原来是暂停,继续录音
            {
                rec_sta&=0XFE;//取消暂停
            }else if(rec_sta&0X80)//已经在录音了,暂停
            {
                rec_sta|=0X01; //暂停
            }else           //还没开始录音
            {
                rec_sta|=0X80; //开始录音
                recoder_new.pathname(pname);           //得到新的名字
                Show_Str(60,230,240,16,pname+11,16,0); //显示录音文件名字
                recoder_wav_init(wavhead);           //初始化 wav 数据
                res=f_open(f_rec,(const TCHAR*)pname,FA_CREATE_ALWAYS
                           |FA_WRITE);
                if(res)//文件创建失败
                {
                    rec_sta=0; //创建文件失败,不能录音
                }
            }
        }
    }
}
```



```
rval=0XFE; //提示是否存在 SD 卡
}else res=f_write(f_rec,(const void*)wavhead,
sizeof(__WaveHeader),&bw);//写入头数据
}
if(rec_sta&0X01)LED1=0; //提示正在暂停
else LED1=1;
break;
case KEY_UP: //AGC+
case KEY_DOWN://AGC-
if(key==KEY_UP)recagc++;
else if(recagc)recagc--;
if(recagc>15)recagc=15;//范围限定为 0~15,自动 AGC.其他 AGC 倍数
recodec_show_agc(recagc);
VS_WR_Cmd(SPI_AICTRL1,1024*recagc);
//设置增益,0,自动增益.1024 相当于 1 倍,512 相当于 0.5 倍
break;
}
if(rec_sta==0X80)//已经在录音了
{
w=VS_RD_Reg(SPI_HDAT1);
if((w>=256)&&(w<896))
{
idx=0;
while(idx<512) //一次读取 512 字节
{
w=VS_RD_Reg(SPI_HDAT0);
recbuf[idx++]=w&0xFF; recbuf[idx++]=w>>8;
}
res=f_write(f_rec,recbuf,512,&bw);//写入文件
if(res) break;//写入出错.
sectorSize++; //扇区数增加 1,约为 32ms
}
}
}else//没有开始录音, 则检测 TPAD 按键
{
if(TPAD_Scan(0)&&pname[0])//如果触摸按键被按下,且 pname 不为空
{
Show_Str(60,230,240,16,"播放:",16,0);
Show_Str(60+40,230,240,16,pname+11,16,0);//显示播放的文件名字
rec_play_wav(pname); //播放 pname
LCD_Fill(60,230,240,246,WHITE); //清除之前显示的录音文件名
recodec_enter_rec_mode(1024*recagc);//重新进入录音模式
while(VS_RD_Reg(SPI_HDAT1)>>8); //等到 buf 较为空闲再开始
recodec_show_time(recsec); //显示时间
}
```



```
        recoder_show_agc(recagc);           //显示 agc
    }
    delay_ms(5); timecnt++;
    if((timecnt%20)==0)LED0=!LED0;//DS0 闪烁
}
if(recsec!=(sectorsize*4/125))//录音时间显示
{
    LED0=!LED0;//DS0 闪烁
    recsec=sectorsize*4/125;
    recoder_show_time(recsec);//显示时间
}
}
myfree(SRAMIN,wavhead); myfree(SRAMIN,recbuf);
myfree(SRAMIN,f_rec);myfree(SRAMIN,pname);
return rval;
}
```

该函数实现了我们在硬件设计时介绍的功能，我们就不详细介绍了。recorder.c 的其他代码和 recorder.h 的代码我们这里就不再贴出了，请大家参考光盘本实验的源码。最后我们看看我们的主函数：

```
int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
    LCD_Init();            //LCD 初始化
    KEY_Init();            //按键初始化
    Audiosel_Init();       //初始化音源选择
    mem_init(SRAMIN);     //初始化内部内存池
    VS_Init();             //VS1053 初始化
    exfuns_init();         //为 fatfs 相关变量申请内存
    f_mount(0,fs[0]);      //挂载 SD 卡
    f_mount(1,fs[1]);      //挂载 FLASH.
    POINT_COLOR=RED;
    while(font_init())     //检查字库
    {
        LCD_ShowString(60,50,200,16,16,"Font Error!");
        delay_ms(200);
        LCD_Fill(60,50,240,66,WHITE);//清除显示
    }
    Show_Str(60,50,200,16,"战舰 STM32 开发板",16,0);
    Show_Str(60,70,200,16,"WAV 录音机实验",16,0);
}
```



```
Show_Str(60,90,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,110,200,16,"2012 年 9 月 20 日",16,0);
Show_Str(60,130,200,16,"KEY0:REC/PAUSE",16,0);
Show_Str(60,150,200,16,"KEY2:STOP&SAVE",16,0);
Show_Str(60,170,200,16,"KEY_UP:AGC+ KEY1:AGC-",16,0);
Show_Str(60,190,200,16,"TPAD:Play The File",16,0);
while(1)
{
    Audiosel_Set(0); //MP3 通道
    LED1=0;
    Show_Str(60,210,200,16,"存储器测试...",16,0);
    VS_Ram_Test();
    Show_Str(60,210,200,16,"正弦波测试...",16,0);
    VS_Sine_Test();
    Show_Str(60,210,200,16,"<<WAV 录音机>>",16,0);
    LED1=1;
    recoder_play();
}
}
```

该函数代码同上一章的 main 函数代码几乎一样，只是我们这里增加了 TPAD 初始化，然后修改了一些显示内容，其他两者就都差不多了，我们就不再细说了。
至此，本实验的软件设计部分结束。

50.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，程序先检测字库，然后对 VS1053 进行 RAM 测试和正弦测试，之后检测 SD 卡的 RECORDER 文件夹，一切顺利通过之后，激活 VS1053 的 PCM 录音模式，得到，如图 50.4.1 所示：



图 50.4.1 录音机界面

此时，我们按下 KEY0 就开始录音了，此时看到屏幕显示录音文件的名字以及录音时长，如图 50.4.2 所示：



图 50.4.2 录音进行中

在录音的时候按下 KEY0 则执行暂停/继续录音的切换，通过 DS1 指示录音暂停，按 WK_UP 和 KEY1 可以调节 AGC，AGC 越大，越灵敏，不过不建议设置太大，因为这可能导致失真。通过按下 KEY2，可以停止当前录音，并保存录音文件。在完成一次录音文件保存之后，我们



可以通过按 TPAD 按键，来实现播放这个录音文件（即播放最近一次的录音文件），实现试听。

我们将开发板的录音文件放到电脑上面，可以通过属性查看录音文件的属性，如图 50.4.3 所示：



图 50.4.3 录音文件属性

这和我们预期的效果一样，通过电脑端的播放器（winamp/千千静听等）可以直接播放我们所录的音频。经实测，效果还是非常不错的。

第五十一章 手写识别实验

现在几乎所有带触摸屏的手机都能实现手写识别。本章，我们将利用 ALIENTEK 提供的手写识别库，在 ALIENTEK 战舰 STM32 开发板上实现一个简单的数字字母手写识别。本章分为如下几个部：

- 51.1 手写识别简介
- 51.2 硬件设计
- 51.3 软件设计
- 51.4 下载验证



51.1 手写识别简介

手写识别，是指对在手写设备上书写时产生的有序轨迹信息进行识别的过程，是人际交互最自然、最方便的手段之一。随着智能手机和平板电脑等移动设备的普及，手写识别的应用也被越来越多的设备采用。

手写识别能够使用户按照最自然、最方便的输入方式进行文字输入，易学易用，可取代键盘或者鼠标。用于手写输入的设备有许多种，比如电磁感应手写板、压感式手写板、触摸屏、触控板、超声波笔等。ALIENTEK 战舰 STM32 开发板自带了 2.8 寸触摸屏，可以用来作为手写识别的输入设备。接下来，我们将给大家简单介绍下手写识别的实现过程。

手写识别与其他识别系统如语音识别图像识别一样分为两个过程：训练学习过程；识别过程。如图 51.1.1 所示：

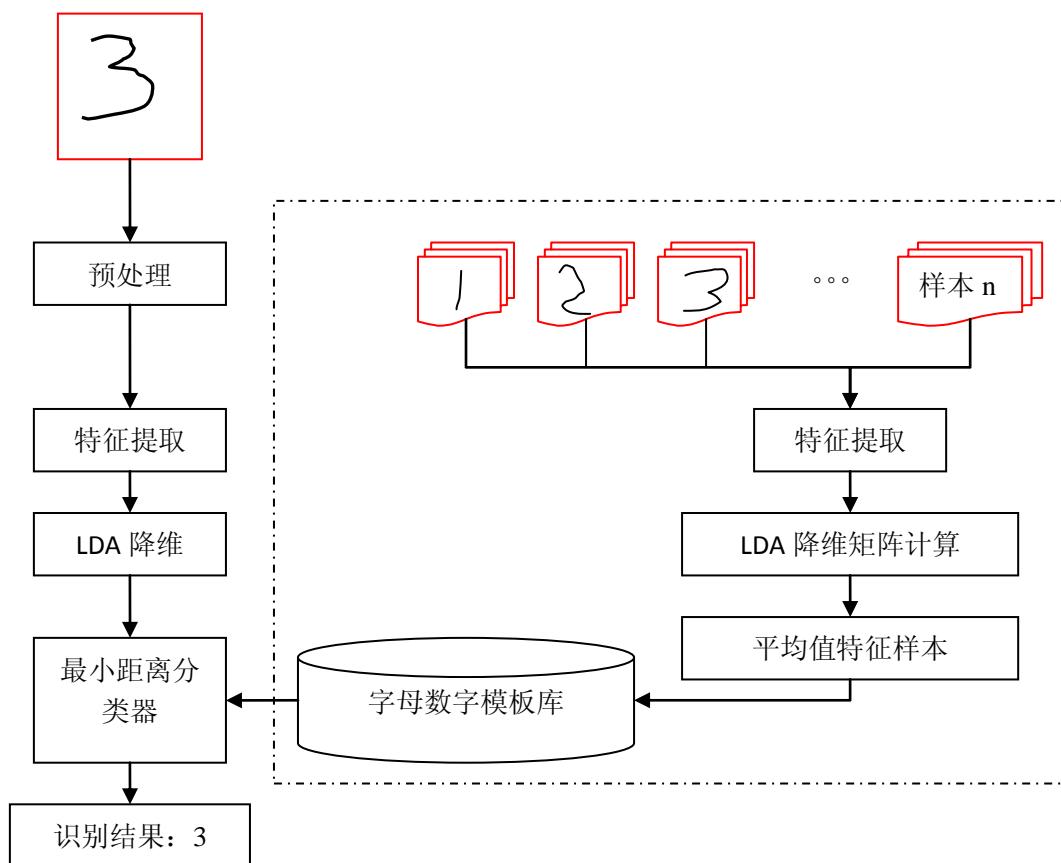


图 51.1.1 字母数字识别系统示意图。

上图中虚线部分为训练学习过程，该过程首先需要使用设备采集大量数据样本，样本类别数目为 0~9, a~z, A~Z 总共 62 类，每个类别 5~10 个样本不等（样本越多识别率就越高）。对这些样本进行传统的把方向特征提取，提取后特征维数为 512 维，这对 STM32 来讲计算量，合模板库的存储量来说都难以接受，所以需要运行一些方法进行降维，这里采用 LDA 线性判决策分析的方法进行降维，所谓线性判决分析，即是假设所有样服从高斯分布（正态分布）对样本进行低维投影，以达到各个样本间的距离最大化。关于 LDA 的更多知识可以阅读 (<http://wenku.baidu.com/view/f05c731452d380eb62946d39.html>) 等参考文档。这里将维度降到



64 维，然后针对各个样本类别进行平均计算得到该类别的样本模板。

而对于识别过程，首先得到触屏输入的有序轨迹，然后进行一些预处理，预处理主要包括重采样，归一化处理。重采样主要是因为不同的输入设备不同的输入处理方式产生的有序轨迹序列有所不同，为了达到更好的识别结果我们需要对训练样本和识别输入的样本进行重采样处理，这里主要应用隔点重采样的方法对输入的序列进行重采样；而归一化就是因为不同的书写风格采样分辨率的差异会导致字体太小不同，因此需要对输入轨迹进行归一化。这里把样本进行线性缩放的方法归一化为 64*64 像素。

接下来进行同样的八方向特征提取操作。所谓八方向特征就是首相将经过预处理后的 64*64 输入进行切分成 8*8 的小方格，每个方格 8*8 个像素；然后对每个 8*8 个小格进行各个方向的点数统计。如某个方格内一共有 10 个点，其中八个方向的点分别为：1、3、5、2、3、4、3、2 那么这个格子得到的八个特征向量为[0.1, 0.3, 0.5, 0.2, 0.3, 0.4, 0.3, 0.2]。总共有 64 个格子于是一个样本最终能得到 64*8=512 维特征，更多八方向特征提取可以参考一下两个文档：

1, <http://wenku.baidu.com/view/d37e5a49e518964bcf847ca5.html>;

2, <http://wenku.baidu.com/view/3e7506254b35eefdc8d333a1.html>;

由于训练过程进行了 LDA 降维计算，所以识别过程同样需要对应的 LDA 降维过程得到最终的 64 维特征。这个计算过程就是在训练模板的过程中可以运算得到一个 512*64 维的矩阵，那么我们通过矩阵乘运算可以得到 64 维的最终特征值。

$$\begin{bmatrix} d_1, d_2, \dots, d_{512} \end{bmatrix} \times \begin{bmatrix} l & \dots & l \\ \vdots & \ddots & \vdots \\ l & \dots & l \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_{64} \end{bmatrix}$$

最后将这 64 维特征分别与模板中的特征进行求距离运算。得到最小的距离为该输入的最佳识别结果输出。

$$output = \arg \min_{i \in [1, 62]} \{(f_1 - f_1^i)^2 + (f_2 - f_2^i)^2 + \dots + (f_{64} - f_{64}^i)^2\}$$

关于手写识别原理，我们就介绍到这里。如果想自己实现手写识别，那得花很多时间学习和研究，但是如果只是应用的话，那么就只需要知道怎么用就 OK 了，相对来说，简单的多。

ALIENTEK 提供了一个数字字母识别库，这样我们不需要关心手写识别是如何实现的，只需要知道这个库怎么用，就能实现手写识别。ALIENTEK 提供的手写识别库由 4 个文件组成：ATKNCR_M_V2.0.lib、ATKNCR_N_V2.0.lib、atk_ncr.c 和 atk_ncr.h。

ATKNCR_M_V2.0.lib 和 ATKNCR_N_V2.0.lib 是两个识别用的库文件（两个版本），使用的时候，选择其中之一即可。ATKNCR_M_V2.0.lib 用于使用内存管理的情况，用户必须自己实现 alientek_ncr_malloc 和 alientek_ncr_free 两个函数。而 ATKNCR_N_V2.0.lib 用于不使用内存管理的情况，通过全局变量来定义缓存区，缓存区需要提供至少 3K 左右的 RAM。大家根据自己的需要，选择不同的版本即可。ALIENTEK 手写识别库资源需求：FLASH:52K 左右，RAM: 6K 左右。

atk_ncr.c 代码如下：

```
#include "atk_ncr.h"
#include "malloc.h"
//内存设置函数
void alientek_ncr_memset(char *p,char c,unsigned long len)
{
```



```
mymemset((u8*)p,(u8)c,(u32)len);
}

//内存申请函数
void *alientek_ncr_malloc(unsigned int size)
{
    return mymalloc(SRAMIN,size);
}

//内存清空函数
void alientek_ncr_free(void *ptr)
{
    myfree(SRAMIN,ptr);
}
```

这里，主要实现了 alientek_ncr_malloc、alientek_ncr_free 和 alientek_ncr_memset 等三个函数。

atk_ncr.h 则是识别库文件同外部函数的接口函数声明

```
#ifndef __ATK_NCR_H
#define __ATK_NCR_H

//当使用 ATKNCR_M_Vx.x.lib 的时候,不需要理会 ATK_NCR_TRACEBUF1_SIZE 和
//ATK_NCR_TRACEBUF2_SIZE
//当使用 ATKNCR_N_Vx.x.lib 的时候,如果出现识别死机,请适当增加
//ATK_NCR_TRACEBUF1_SIZE 和 ATK_NCR_TRACEBUF2_SIZE 的值
#define ATK_NCR_TRACEBUF1_SIZE 500*4
//定义第一个 tracebuf 大小(单位为字节),如果出现死机,请把该数组适当改大
#define ATK_NCR_TRACEBUF2_SIZE 250*4
//定义第二个 tracebuf 大小(单位为字节),如果出现死机,请把该数组适当改大
//输入轨迹坐标类型
__packed typedef struct _atk_ncr_point
{
    short x; //x 轴坐标
    short y; //y 轴坐标
}atk_ncr_point;
//外部调用函数
//初始化识别器
//返回值:0,初始化成功
//      1,初始化失败
unsigned char alientek_ncr_init(void);
void alientek_ncr_stop(void); //停止识别器
//识别器识别
//track:输入点阵集合
//potnum:输入点阵的点数,就是 track 的大小
//charnum:期望输出的结果数,就是你希望输出多少个匹配结果
//mode:识别模式
//1,仅识别数字
```



```
//2,进识别大写字母
//3,仅识别小写字母
//4,混合识别(全部识别)
//result:结果缓存区(至少为:charnum+1 个字节)
void alientek_ncr(atk_ncr_point * track,int potnum,int charnum,unsigned char mode,char*result);
void alientek_ncr_memset(char *p,char c,unsigned long len); //内存设置函数
//动态申请内存,当使用 ATKNCR_M_Vx.x.lib 时,必须实现.
void *alientek_ncr_malloc(unsigned int size);
//动态释放内存,当使用 ATKNCR_M_Vx.x.lib 时,必须实现.
void alientek_ncr_free(void *ptr);
#endif
```

此段代码中，我们定义了一些外部接口函数以及一个轨迹结构体等。

`alientek_ncr_init`，该函数用与初始化识别器，该函数在.lib 文件实现，在识别开始之前，我们应该调用该函数。

`alientek_ncr_stop`，该函数用于停止识别器，在识别完成之后（不需要再识别），我们调用该函数，如果一直处于识别状态，则没必要调用。该函数也是在.lib 文件实现。

`alientek_ncr`，该函数就是识别函数了。它有 5 个参数，第一个参数 `track`，为输入轨迹点的坐标集（最好 200 以内）；第二个参数 `potnum`，为坐标集点坐标的个数；第三个参数 `charnum`，为期望输出的结果数，即希望输出多少个匹配结果，识别器按匹配程度排序输出（最佳匹配排第一）；第四个参数 `mode`，该函数用于设置模式，识别器总共支持 4 中模式：

- 1, 仅识别数字
- 2, 进识别大写字母
- 3, 仅识别小写字母
- 4, 混合识别(全部识别)

最后一个参数是 `result`，用来输出结果，注意这个结果是 ASCII 码格式的。

`alientek_ncr_memset`、`alientek_ncr_free` 和 `alientek_ncr_free` 这 3 个函数在 `atk_ncr.c` 里面实现，这里就不多说了。

最后，我们看看通过 ALIENTEK 提供的手写数字字母识别库实现数字字母识别的步骤：

1) 调用 `alientek_ncr_init` 函数, 初始化识别程序

该函数用来初始化识别器，在手写识别进行之前，必须调用该函数。

2) 获取输入的点阵数据

此步，我们通过触摸屏获取输入轨迹点阵坐标，然后存放到一个缓存区里面，注意至少要输入 2 个不同坐标的点阵数据，才能正常识别。注意输入点数不要太多，太多的话，需要更多的内存，我们推荐的输入点数范围：100~200 点。

3) 调用 `alientek_ncr` 函数, 得到识别结果.

通过调用 `alientek_ncr` 函数，我们可以得到输入点阵的识别结果，结果将保存在 `result` 参数里面，采用 ASCII 码格式存储

4) 调用 `alientek_ncr_stop` 函数, 终止识别.

如果不需要继续识别，则调用 `alientek_ncr_stop` 函数，终止识别器。如果还需要继续识别，重复步骤 2 和步骤 3 即可。

以上 4 个步骤，就是使用 ALIENTEK 手写识别库的方法，十分简单。



51.2 硬件设计

本章实验功能简介：开机的时候先初始化手写识别器，然后检测字库，之后进入等待输入状态。此时，我们在手写区写数字/字符，在每次写入结束后，自动进入识别状态，进行识别，然后将识别结果输出在 LCD 模块上面（同时打印到串口）。通过按 KEY0 可以进行模式切换（4 种模式都可以测试），通过按 KEY2，可以进入触摸屏校准（如果发现触摸屏不准，请执行此操作）。DS0 用于指示程序运行状态。

本实验用到的资源如下：

- 1) 指示灯 DS0
- 2) KEY0 和 KEY2 两个按键
- 3) 串口
- 4) TFTLCD 模块（含触摸屏）
- 5) SPI FLASH

这些用到的硬件，我们在之前都已经介绍过，这里就不再介绍了。

51.3 软件设计

打开我们光盘的手写识别实验，可以看到我们添加了 ATKNCR_M_V2.0.lib 和 atk_ncr.c 两个文件到工程中。关于 ATKNCR_M_V2.0.lib 和 atk_ncr.c 前面已有介绍，我们这里就不再多说，我们在 main.c 里面修改 main 函数如下：

```
//最大记录的轨迹点数
atk_ncr_point READ_BUF[200];
int main(void)
{
    u8 i=0;
    u8 tcnt=0;
    u8 res[10];
    u8 key;
    u16 pcnt=0;
    u8 mode=4;           //默认是混合模式
    delay_init();         //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);      //串口初始化波特率为 9600
    LED_Init();           //LED 端口初始化
    LCD_Init();           //LCD 初始化
    KEY_Init();           //按键初始化
    TP_Init();             //触摸屏初始化
    mem_init(SRAMIN);     //初始化内部内存池
    alientek_ncr_init();   //初始化手写识别

    POINT_COLOR=RED;
    while(font_init())           //检查字库
    {
        LCD_ShowString(60,50,200,16,16,"Font Error!");
    }
}
```



```
delay_ms(200);
LCD_Fill(60,50,240,66,WHITE); //清除显示
}

RESTART:
Show_Str(60,10,200,16,"战舰 STM32 开发板",16,0);
Show_Str(60,30,200,16,"手写识别实验",16,0);
Show_Str(60,50,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,70,200,16,"KEY0:MODE KEY2:Adjust",16,0);
Show_Str(60,90,200,16,"识别结果:",16,0);
LCD_DrawRectangle(19,114,220,315);
POINT_COLOR=BLUE;
Show_Str(96,207,200,16,"手写区",16,0);
tcnt=100;
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY_LEFT)
    {
        TP_Adjust(); //屏幕校准
        LCD_Clear(WHITE);
        goto RESTART; //重新加载界面
    }
    if(key==KEY_RIGHT)
    {
        LCD_Fill(20,115,219,314,WHITE); //清除当前显示
        mode++;
        if(mode>4)mode=1;
        switch(mode)
        {
            case 1:
                Show_Str(80,207,200,16,"仅识别数字",16,0);
                break;
            case 2:
                Show_Str(64,207,200,16,"仅识别大写字母",16,0);
                break;
            case 3:
                Show_Str(64,207,200,16,"仅识别小写字母",16,0);
                break;
            case 4:
                Show_Str(88,207,200,16,"全部识别",16,0);
                break;
        }
        tcnt=100;
    }
}
```



```
}

tp_dev.scan(0); //扫描
if(tp_dev.sta&TP_PRES_DOWN) //有按键被按下
{
    delay_ms(1); //必要的延时,否则老认为有按键按下.
    tcnt=0; //松开时的计数器清空
    if((tp_dev.x<220&&tp_dev.x>=20)&&(tp_dev.y<315&&tp_dev.y>=115))
    {
        TP_Draw_Big_Point(tp_dev.x,tp_dev.y,BLUE); //画图
        if(pcnt<200) //总点数少于 200
        {
            if(pcnt)
            {

if((READ_BUF[pcnt-1].y!=tp_dev.y)&&(READ_BUF[pcnt-1].x!=tp_dev.x)) //x,y 不相等
{
    READ_BUF[pcnt].x=tp_dev.x;
    READ_BUF[pcnt].y=tp_dev.y;
    pcnt++;
}
else
{
    READ_BUF[pcnt].x=tp_dev.x;
    READ_BUF[pcnt].y=tp_dev.y;
    pcnt++;
}
}
}
}
} //按键松开了
{
    tcnt++;
    delay_ms(10);
    //延时识别
    i++;
    if(tcnt==40)
    {
        if(pcnt) //有有效的输入
        {
            printf("总点数:%d\r\n",pcnt);
            alientek_ncr(READ_BUF,pcnt,6,mode,(char*)res);
            printf("识别结果:%s\r\n",res);
            pcnt=0;
            POINT_COLOR=BLUE; //设置画笔蓝色
        }
    }
}
```



```
LCD_ShowString(60+72,90,200,16,16,res);
}
LCD_Fill(20,115,219,314,WHITE);
}
}
if(i==30)
{
    i=0;
    LED0=!LED0;
}
}
```

该函数同触摸屏实验的 main 函数有点类似，不过加入了一些处理，以实现 51.1.2 节提到的功能。其中，READ_BUF 用来存储输入轨迹点阵，大小为 200，即最大输入不能超过 200 点，注意：这里我们采集的都是不用的点阵（即相邻的坐标不相等）。这样可以大大减少重复点阵的大小，而重复点阵对识别是没有帮助的。

至此，本实验的软件设计部分结束。

51.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，得到，如图 51.4.1 所示：

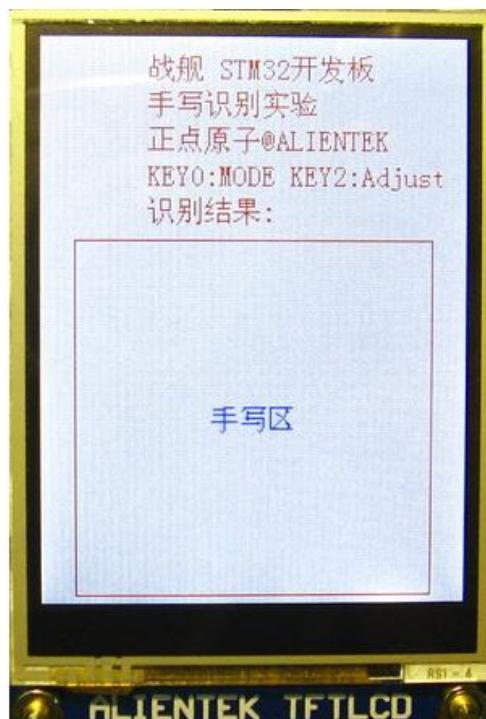


图 51.4.1 手写识别界面

此时，我们在识别区写数字/字母，即可得到识别结果，如图 51.4.2 所示：

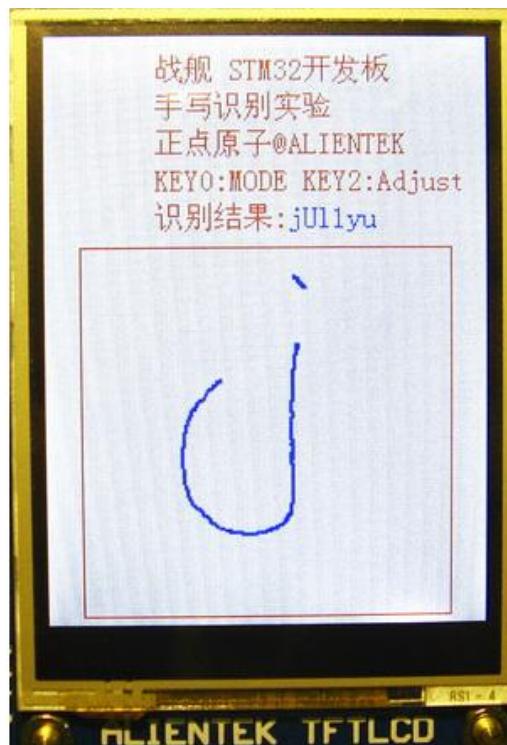


图 51.4.2 手写识别结果

按下 KEY0 可以切换识别模式，同时在识别区提示当前模式。按下 KEY2 可以进行屏幕校准。每次识别结束，会在串口打印本次识别的输入点数和识别结果，大家可以通过串口助手查看。



第五十二章 T9 拼音输入法实验

上一章，我们在 ALIENTEK 战舰 STM32 开发板上实现了手写识别输入，但是该方法只能输入数字或者字母，不能输入汉字。本章，我们将给大家介绍如何在 ALIENTEK 战舰 STM32 开发板上实现一个简单的 T9 中文拼音输入法。本章分为如下几个部：

52.1 拼音输入法简介

52.2 硬件设计

52.3 软件设计

52.4 下载验证

52.1 拼音输入法简介

在计算机上汉字的输入法有很多种，比如拼音输入法、五笔输入法、笔画输入法、区位输入法等。其中，又以拼音输入法用的最多。拼音输入法又可以分为很多类，比如全拼输入、双拼输入等。

而在手机上，用的最多的应该算是 T9 拼音输入法了，T9 输入法全名为智能输入法，字库容量九千多字，支持十多种语言。T9 输入法是由美国特捷通讯（Tegic Communications）软件公司开发的，该输入法解决了小型掌上设备的文字输入问题，已经成为全球手机文字输入的标准之一。

一般，手机拼音输入键盘如图 52.1.1 所示：



图 52.1.1 手机拼音输入键盘

在这个键盘上，我们对比下传统的输入法和 T9 输入法，输入“中国”两个字需要的按键次数。传统的方法，先按 4 次 9，输入字母 z，再按 2 次 4，输入字母 h，再按 3 次 6，输入字母 o，再按 2 次 6，输入字母 n，最后按 1 次 4，输入字母 g。这样，输入“中”字，要按键 12 次，接着同样的方法，输入“国”字，需要按 6 次，总共就是 18 次按键。

如果是 T9，我们输入“中”字，只需要输入：9、4、6、6、4，即可实现输入“中”字，在选择中字之后，T9 会联想出一系列同中字组合的次，如文、国、断、山等。这样输入“国”字，我们直接选择即可，所以输入“国”字按键 0 次，这样 T9 总共只需要 5 次按键。

这就是 T9 智能输入法的优越之处。正因为 T9 输入法高效便捷的输入方式得到了众多手机厂商的采用，以至于 T9 成为了使用频率最高知名度最大的手机输入法。

本章，我们实现的 T9 拼音输入法，没有真正的 T9 那么强大，我们这里仅实现输入部分，不支持词组联想。

本章，我们主要通过一个和数字串对应的拼音索引表来实现 T9 拼音输入，我们先将汉语拼音所有可能的组合全部列出来，如下所示：

```
const u8 PY_mb_space []={""};  
const u8 PY_mb_a      []={"啊阿唵𠮣钢屁嘎鋼呵唵"};  
const u8 PY_mb_ai     []={"爱埃挨哎唉哀𠵼癌𦨻矮艾碍隘𢺔𢺔唉𡥃𡥃暖暖破娘靄"};  
const u8 PY_mb_an     []={"安俺按暗岸案鞍氨谙胺埯𢺔犴桉铵鶴黯"};  
.....此处省略 N 多组合  
const u8 PY_mb_zu     []={"足租祖沮阻组卒族俎菹𦫸"};  
const u8 PY_mb_zuan   []={"钻攢纂饋躰"};  
const u8 PY_mb_zui    []={"最罪嘴醉蕞觜"};  
const u8 PY_mb_zun    []={"尊遵樽樽撙撙"};  
const u8 PY_mb_zuo    []={"左佐做作坐座昨撮咤柞昨琢嘬柞胙柞酢"};
```

这里我们只列出了部分组合，我们将这些组合称之为码表，然后将这些码表和其对应的数



字串对应起来，组成一个拼音索引表，如下所示：

```
const py_index py_index3[] =  
{  
    {"", "", (u8*)PY_mb_space},  
    {"2", "a", (u8*)PY_mb_a},  
    {"3", "e", (u8*)PY_mb_e},  
    {"6", "o", (u8*)PY_mb_o},  
    {"24", "ai", (u8*)PY_mb_ai},  
    {"26", "an", (u8*)PY_mb_an},  
    .....此处省略 N 多组合  
    {"94664", "zhong", (u8*)PY_mb_zhong},  
    {"94824", "zhuai", (u8*)PY_mb_zhuai},  
    {"94826", "zhuan", (u8*)PY_mb_zhuan},  
    {"248264", "chuang", (u8*)PY_mb_chuang},  
    {"748264", "shuang", (u8*)PY_mb_shuang},  
    {"948264", "zhuang", (u8*)PY_mb_zhuang},  
}
```

其中 py_index 是一个结构体，定义如下：

```
typedef struct  
{  
    u8 *py_input; //输入的字符串  
    u8 *py; //对应的拼音  
    u8 *pymb; //码表  
}py_index;
```

其中 py_input，即与拼音对应的数字串，比如“94824”。py，即与 py_input 数字串对应的拼音，如果 py_input=“94824”，那么 py 就是“zhuai”。最后 pymb，就是我们前面说到的码表。注意，一个数字串可以对应多个拼音，也可以对应多个码表。

在有了这个拼音索引表（py_index3）之后，我们只需要将输入的数字串和 py_index3 索引表里面所有成员的 py_input 对比，将所有完全匹配的情况记录下来，用户要输入的汉字就被确定了，然后由用户选择可能的拼音组成（假设有多个匹配的项目），再选择对应的汉字，即完成一次汉字输入。

当然还可能是找遍了索引表，也没有发现一个完全符合要求的成员，那么我们会统计匹配数最多的情况，作为最佳结果，反馈给用户。比如，用户输入“323”，找不到完全匹配的情况，那么我们就将能和“32”匹配的结果返回给用户。这样，用户还是可以得到输入结果，同时还可以知道输入有问题，提示用户需要检查输入是否正确。

以上，就是我们的 T9 拼音输入法原理，关于拼音输入法，我们就介绍到这里。

最后，我们看看一个完整的 T9 拼音输入步骤（过程）：

1) 输入拼音数字串

本章，我们用到的 T9 拼音输入法的核心思想就是对比用户输入的拼音数字串，所以必须先由用户输入拼音数字串。

2) 在拼音索引表里面查找和输入字符串匹配的项，并记录

在得到用户输入的拼音数字串之后，在拼音索引表里面查找所有匹配的项目，如果有完全匹配的项目，就全部记录下来，如果没有完全匹配的项目，则记录匹配情况最好的一



个项目。

3) 显示匹配清单里面所有可能的汉字，供用户选择.

将匹配项目的拼音和对应的汉字显示出来，供用户选择。如果有多个匹配项（一个数字串对应多个拼音的情况），则用户还可以选择拼音。

4) 用户选择匹配项，并选择对应的汉字.

用户对匹配的拼音和汉字进行选择，选中其真正想输入的拼音和汉字，实现一次拼音输入。

以上 4 个步骤，就可以实现一个简单的 T9 汉字拼音输入法。

52.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后显示提示信息和绘制拼音输入表，之后进入等待输入状态。此时用户可以通过屏幕上的拼音输入表输入拼音数字串（通过 **DEL** 可以实现退格），然后程序自动检测与之对应的拼音和汉字，并显示在屏幕上（同时输出到串口）。如果有多个匹配的拼音，则通过 **WK_UP** 和 **KEY1** 进行选择。按键 **KEY0** 用于清除一次输入，按键 **KEY2** 用于触摸屏校准。

本实验用到的资源如下：

- 1) 指示灯 **DS0**
- 2) 四个按键 (**KEY0/KEY1/KEY2/WK_UP**)
- 3) 串口
- 4) TFTLCD 模块（含触摸屏）
- 5) SPI FLASH

这些用到的硬件，我们在之前都已经介绍过，这里就不再介绍了。

52.3 软件设计

打开拼音输入法工程，可以看到我们首先在 **HARDWARE** 文件夹所在的文件夹下新建一个 **T9INPUT** 的文件夹。在该文件夹下面新建 **pyinput.c**、**pyinput.h** 和 **pymb.h** 三个文件，然后在工程里面新建了一个 **T9INPUT** 的组，将 **pyinput.c** 加入到该组下面。最后，将 **T9INPUT** 文件夹加入头文件包含路径。

打开 **pyinput.c**，该文件代码如下：

```
#include "sys.h"
#include "uart.h"
#include "pymb.h"
#include "pyinput.h"
#include "string.h"
//拼音输入法
pyinput t9=
{
    get_pymb,
    0,
};
//比较两个字符串的匹配情况
//返回值:0xff,表示完全匹配.
```



```
//      其他,匹配的字符数
u8 str_match(u8*str1,u8*str2)
{
    u8 i=0;
    while(1)
    {
        if(*str1!=*str2)break;          //部分匹配
        if(*str1=='\0'){i=0xFF;break;}//完全匹配
        i++; str1++; str2++;
    }
    return i;//两个字符串相等
}

//获取匹配的拼音码表
/*strin,输入的字符串,形如:"726"
/**matchlist,输出的匹配表.
//返回值:[7],0,表示完全匹配; 1, 表示部分匹配 (仅在没有完全匹配的时候才会出现)
//      [6:0],完全匹配的时候, 表示完全匹配的拼音个数
//      部分匹配的时候, 表示有效匹配的位数
u8 get_matched_pymb(u8 *strin,py_index **matchlist)
{
    py_index *bestmatch;//最佳匹配
    u16 pyindex_len;
    u16 i;
    u8 temp,mcnt=0,bmcnt=0;
    bestmatch=(py_index*)&py_index3[0];//默认为 a 的匹配
    pyindex_len=sizeof(py_index3)/sizeof(py_index3[0]);//得到 py 索引表的大小.
    for(i=0;i<pyindex_len;i++)
    {
        temp=str_match(strin,(u8*)py_index3[i].py_input);
        if(temp)
        {
            if(temp==0xFF)matchlist[mcnt++]=(py_index*)&py_index3[i];
            else if(temp>bmcnt)//找最佳匹配
            {
                bmcnt=temp;
                bestmatch=(py_index*)&py_index3[i];//最好的匹配.
            }
        }
    }
    if(mcnt==0&&bmcnt)//没有完全匹配的结果,但是有部分匹配的结果
    {
        matchlist[0]=bestmatch;
```



```
mcnt=bmcnt|0X80;      //返回部分匹配的有效位数
}
return mcnt;//返回匹配的个数
}
//得到拼音码表.
//str:输入字符串
//返回值:匹配个数.
u8 get_pymb(u8* str)
{
    return get_matched_pymb(str,t9.pymb);
}
//串口测试用
void test_py(u8 *inputstr)
{
    .....代码省略
}
```

这里总共就 4 个函数，其中 `get_matched_pymb`，是核心，该函数实现将用户输入拼音数字串同拼音索引表里面的各个项对比，找出匹配结果，并将完全匹配的项目存放在 `matchlist` 里面，同时记录匹配数。对于那些没有完全匹配的输入串，则查找与其最佳匹配的项目，并将匹配的长度返回。函数 `test_py`（代码省略）用于给 `usmart` 调用，实现串口测试，该函数可有可无，只是在串口测试的时候才用到，如果不使用的话，可以去掉，本章，我们将其加入 `usmart` 控制，大家可以通过该函数实现串口调试拼音输入法。

其他两个函数，也比较简单了，我们这里就不细说了。打开 `pyinput.h`，下代码如下：

```
#ifndef __PYINPUT_H
#define __PYINPUT_H
#include "sys.h"
//拼音码表与拼音的对应表
typedef struct
{
    u8 *py_input;//输入的字符串
    u8 *py;        //对应的拼音
    u8 *pymb;     //码表
}py_index;
#define MAX_MATCH_PYMB 10 //最大匹配数
//拼音输入法
typedef struct
{
    u8(*getpymb)(u8 *instr);           //字符串到码表获取函数
    py_index *pymb[MAX_MATCH_PYMB];    //码表存放位置
}pyinput;
extern pyinput t9;
u8 str_match(u8*str1,u8*str2);
u8 get_matched_pymb(u8 *strin,py_index **matchlist);
```



```
u8 get_pymb(u8* str);
void test_py(u8 *inputstr);
#endif
```

pymb.h 里面完全就是我们前面介绍的拼音码表，该文件很大，里面存储了所有我们可以输入的汉字，此部分代码就不贴出来了，请大家参考光盘本例程的源码。

最后，看看我们的 main.c 函数，输入代码如下：

```
const u8* kbd_tbl[9]={ "<","2","3","4","5","6","7","8","9",};//数字表
const u8* kbs_tbl[9]={ "DEL","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz",};//字符表
//加载键盘界面
//x,y:界面起始坐标
void py_load_ui(u16 x,u16 y)
{
    u16 i;
    POINT_COLOR=RED;
    LCD_DrawRectangle(x,y,x+180,y+120);
    LCD_DrawRectangle(x+60,y,x+120,y+120);
    LCD_DrawRectangle(x,y+40,x+180,y+80);
    POINT_COLOR=BLUE;
    for(i=0;i<9;i++)
    {
        Show_Str_Mid(x+(i%3)*60,y+4+40*(i/3),(u8*)kbd_tbl[i],16,60);
        Show_Str_Mid(x+(i%3)*60,y+20+40*(i/3),(u8*)kbs_tbl[i],16,60);
    }
}
//按键状态设置
//x,y:键盘坐标
//key:键值 (0~8)
//sta:状态， 0， 松开； 1， 按下；
void py_key_staset(u16 x,u16 y,u8 keyx,u8 sta)
{
    u16 i=keyx/3,j=keyx%3;
    if(keyx>8)return;
    if(sta)LCD_Fill(x+j*60+1,y+i*40+1,x+j*60+59,y+i*40+39,WHITE);
    else LCD_Fill(x+j*60+1,y+i*40+1,x+j*60+59,y+i*40+39,WHITE);
    Show_Str_Mid(x+j*60,y+4+40*i,(u8*)kbd_tbl[keyx],16,60);
    Show_Str_Mid(x+j*60,y+20+40*i,(u8*)kbs_tbl[keyx],16,60);
}
//得到触摸屏的输入
//x,y:键盘坐标
//返回值：按键键值 (1~9 有效； 0,无效)
u8 py_get_keynum(u16 x,u16 y)
{
    u16 i,j;
```



```
static u8 key_x=0;//0,没有任何按键按下; 1~9, 1~9号按键按下
u8 key=0;
tp_dev.scan(0);
if(tp_dev.sta&TP_PRES_DOWN)           //触摸屏被按下
{
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            if(tp_dev.x<(x+j*60+60)&&tp_dev.x>(x+j*60)&&tp_dev.y<(y+i*40+40)
                &&tp_dev.y>(y+i*40)) { key=i*3+j+1; break; }
        }
        if(key)
        {
            if(key_x==key)key=0;
            else
            {
                py_key_staset(x,y,key_x-1,0);
                key_x=key;
                py_key_staset(x,y,key_x-1,1);
            }
            break;
        }
    }
}else if(key_x) {py_key_staset(x,y,key_x-1,0); key_x=0; }
return key;
}
//显示结果.
//index:0,表示没有一个匹配的结果.清空之前的显示
// 其他,索引号
void py_show_result(u8 index)
{
    LCD_ShowNum(30+144,125,index,1,16);      //显示当前的索引
    LCD_Fill(30+40,125,30+40+48,130+16,WHITE); //清除之前的显示
    LCD_Fill(30+40,145,30+200,145+48,WHITE); //清除之前的显示
    if(index)
    {
        Show_Str(30+40,125,200,16,t9.pymb[index-1]->py,16,0); //显示拼音
        Show_Str(30+40,145,160,48,t9.pymb[index-1]->pymb,16,0); //显示对应的汉字
        printf("\r\n 拼音:%s\r\n",t9.pymb[index-1]->py); //串口输出拼音
        printf("结果:%s\r\n",t9.pymb[index-1]->pymb); //串口输出结果
    }
}
```



```
int main(void)
{
    u8 i=0;
    u8 result_num;
    u8 cur_index;
    u8 key;
    u8 inputstr[7];           //最大输入 6 个字符+结束符
    u8 inputlen;             //输入长度
    delay_init();            //延时函数初始化
    NVIC_Configuration();   //设置 NVIC 中断分组 2:2 位抢占优先级，2 位响应优先级
    uart_init(9600);         //串口初始化波特率为 9600
    LCD_Init();              //初始化液晶
    LED_Init();              //LED 初始化
    KEY_Init();              //按键初始化
    TP_Init();               //触摸屏初始化
    usmart_dev.init(72);     //usmart 初始化
    mem_init(SRAMIN);        //初始化内部内存池

    RESTART:
    POINT_COLOR=RED;
    while(font_init())        //检查字库
    {
        LCD_ShowString(60,50,200,16,16,"Font Error!");
        delay_ms(200);
        LCD_Fill(60,50,240,66,WHITE); //清除显示
    }
    Show_Str(60,5,200,16,"战舰 STM32 开发板",16,0);
    Show_Str(60,25,200,16,"拼音输入法实验",16,0);
    Show_Str(60,45,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,65,200,16," KEY2:校准 KEY0:清除",16,0);
    Show_Str(30,85,200,16,"KEY_UP:上翻 KEY1:下翻",16,0);
    Show_Str(30,105,200,16,"输入:          匹配:  ",16,0);
    Show_Str(30,125,200,16,"拼音:          当前:  ",16,0);
    Show_Str(30,145,210,32,"结果:",16,0);
    py_load_ui(30,195);
    memset(inputstr,0,7);   //全部清零
    inputlen=0;              //输入长度为 0
    result_num=0;             //总匹配数清零
    cur_index=0;
    while(1)
    {
        i++;
        delay_ms(10);
        key=py_get_keynum(30,195);
```



```
if(key)
{
    if(key==1)//删除
    {
        if(inputlen)inputlen--;
        inputstr[inputlen]='\0';//添加结束符
    }else
    {
        inputstr[inputlen]=key+'0';//输入字符
        if(inputlen<7)inputlen++;
    }
    if(inputstr[0]!=NULL)
    {
        key=t9.getpymb(inputstr); //得到匹配的结果数
        if(key)//有部分匹配/完全匹配的结果
        {
            result_num=key;           //总匹配结果
            cur_index=1;             //当前为第一个索引
            if(key&0X80)           //是部分匹配
            {
                inputlen=key&0X7F;   //有效匹配位数
                inputstr[inputlen]='\0';//不匹配的位数去掉
                if(inputlen>1)result_num=t9.getpymb(inputstr);//重新获取完全匹
配字符数
            }
        }else                         //没有任何匹配
        {
            inputlen--;
            inputstr[inputlen]='\0';
        }
    }else
    {
        cur_index=0;
        result_num=0;
    }
    LCD_Fill(30+40,105,30+40+48,110+16,WHITE); //清除之前的显示
    LCD_ShowNum(30+144,105,result_num,1,16); //显示匹配的结果数
    Show_Str(30+40,105,200,16,inputstr,16,0); //显示有效的数字串
    py_show_result(cur_index);                  //显示第 cur_index 的匹配结果
}
if(result_num)//存在匹配的结果
{
    key=KEY_Scan(0);
```



```
switch(key)
{
    case KEY_UP://上翻
        if(cur_index<result_num)cur_index++;
        else cur_index=1;
        py_show_result(cur_index); //显示第 cur_index 的匹配结果
        break;
    case KEY_DOWN://下翻
        if(cur_index>1)cur_index--;
        else cur_index=result_num;
        py_show_result(cur_index); //显示第 cur_index 的匹配结果
        break;
    case KEY_RIGHT://清除输入
        LCD_Fill(30+40,145,30+200,145+48,WHITE); //清除之前的显示
        goto RESTART;
    case KEY_LEFT://重新校准
        tp_dev.adjust();
        LCD_Clear(WHITE);
        goto RESTART;
    }
}
if(i==30)
{
    i=0;
    LED0=!LED0;
}
}
```

此部分代码除 main 函数外还有 4 个函数。首先，py_load_ui，该函数用于加载输入键盘，在 LCD 上面显示我们输入拼音数字串的虚拟键盘。py_key_staset，该函数用与设置虚拟键盘某个按键的状态（按下/松开）。py_get_keynum，该函数用于得到触摸屏当前按下的按键键值，通过该函数实现拼音数字串的获取。最后，py_show_result，该函数用于显示输入串的匹配结果，并将结果打印到串口。

在 main 函数里面，实现了我们在 52.2 节所说的功能，这里我们并没有实现汉字选择功能，但是有本例程作为基础，再实现汉字选择功能就比较简单了，大家自行实现即可。

最后，我们将 test_py 函数加入 USMART 控制，以便大家串口调试。

至此，本实验的软件设计部分结束。

52.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，得到，如图 52.4.1 所示：



图 52.4.1 汉字输入法界面

此时，我们在虚拟键盘上输入拼音数字串，即可实现拼音输入，如图 52.4.2 所示：



图 52.4.2 实现拼音输入

如果发现输入错了，可以通过屏幕上的 DEL 按钮，来退格。如果有多个匹配的情况（匹配值大于 1），则可以通过 WK_UP 和 KEY1 来选择拼音。通过按下 KEY0，可以清楚当前输入，通过按下 KEY2，可以实现触摸屏校准。

我们还可以通过 USMART 调用 test_py 来实现输入法调试，如图 52.4.3 所示：

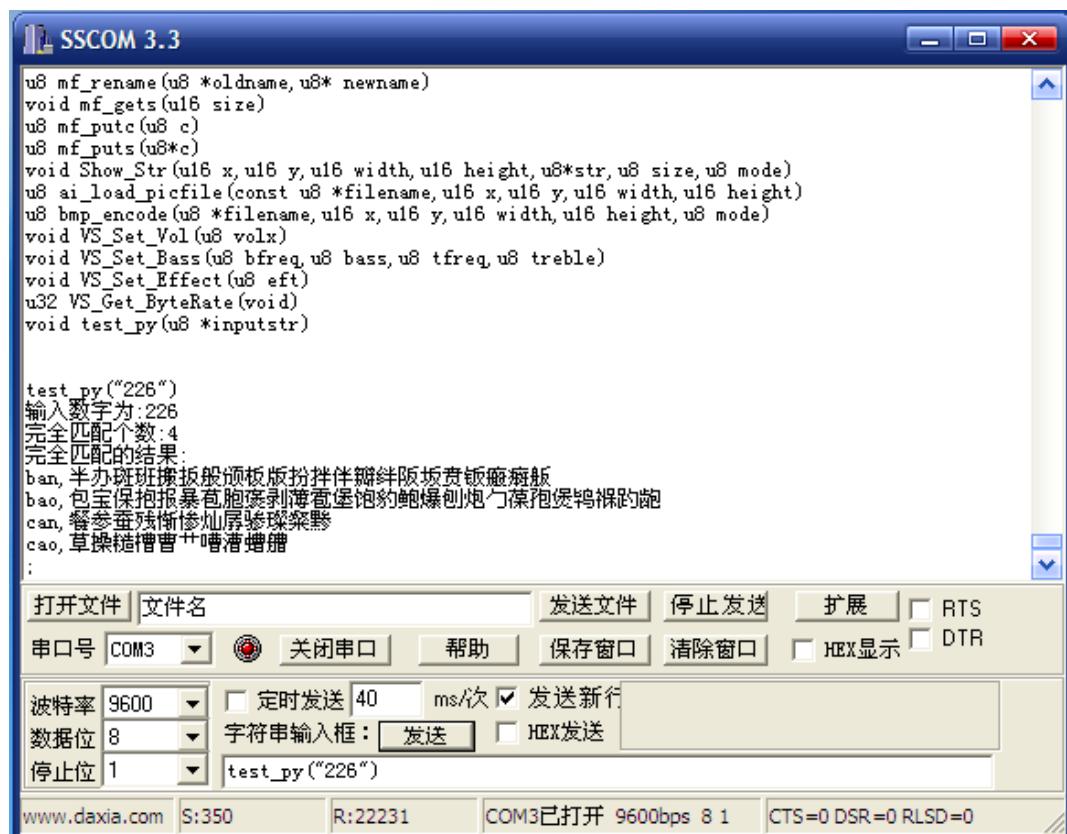


图 52.4.3 USMART 调试 T9 拼音输入法

第五十三章 串口 IAP 实验

IAP，即在应用编程。很多单片机都支持这个功能，STM32 也不例外。在之前的 FLASH 模拟 EEPROM 实验里面，我们学习了 STM32 的 FLASH 自编程，本章我们将结合 FLASH 自编程的知识，通过 STM32 的串口实现一个简单的 IAP 功能。本章分为如下几个部分：

- 53.1 IAP 简介
- 53.2 硬件设计
- 53.3 软件设计
- 53.4 下载验证



53.1 IAP 简介

IAP (In Application Programming) 即在应用编程，IAP 是用户自己的程序在运行过程中对 User Flash 的部分区域进行烧写，目的是为了在产品发布后可以方便地通过预留的通信口对产品中的固件程序进行更新升级。通常实现 IAP 功能时，即用户程序运行中作自身的更新操作，需要在设计固件程序时编写两个项目代码，第一个项目程序不执行正常的功能操作，而只是通过某种通信方式(如 USB、USART)接收程序或数据，执行对第二部分代码的更新；第二个项目代码才是真正的功能代码。这两部分项目代码都同时烧录在 User Flash 中，当芯片上电后，首先是第一个项目代码开始运行，它作如下操作：

- 1) 检查是否需要对第二部分代码进行更新
- 2) 如果不需要更新则转到 4)
- 3) 执行更新操作
- 4) 跳转到第二部分代码执行

第一部分代码必须通过其它手段，如 JTAG 或 ISP 烧入；第二部分代码可以使用第一部分代码 IAP 功能烧入，也可以和第一部分代码一起烧入，以后需要程序更新时再通过第一部分 IAP 代码更新。

我们将第一个项目代码称之为 Bootloader 程序，第二个项目代码称之为 APP 程序，他们存放在 STM32 FLASH 的不同地址范围，一般从最低地址区开始存放 Bootloader，紧跟其后的就是 APP 程序（注意，如果 FLASH 容量足够，是可以设计很多 APP 程序的，本章我们只讨论一个 APP 程序的情况）。这样我们就是要实现 2 个程序：Bootloader 和 APP。

STM32 的 APP 程序不仅可以放到 FLASH 里面运行，也可以放到 SRAM 里面运行，本章，我们将制作两个 APP，一个用于 FLASH 运行，一个用于 SRAM 运行。

我们先来看看 STM32 正常的程序运行流程，如图 53.1.1 所示：

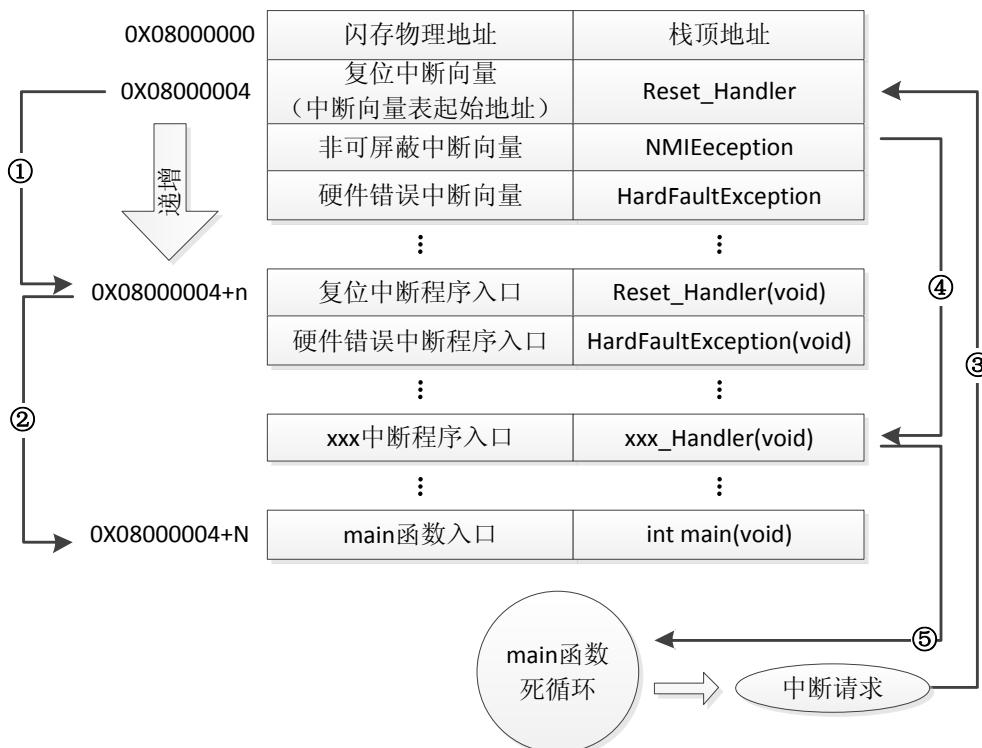


图 53.1.1 STM32 正常运行流程图



STM32 的内部闪存（FLASH）地址起始于 0x08000000，一般情况下，程序文件就从此地址开始写入。此外 STM32 是基于 Cortex-M3 内核的微控制器，其内部通过一张“中断向量表”来响应中断，程序启动后，将首先从“中断向量表”取出复位中断向量执行复位中断程序完成启动，而这张“中断向量表”的起始地址是 0x08000004，当中断来临，STM32 的内部硬件机制亦会自动将 PC 指针定位到“中断向量表”处，并根据中断源取出对应的中断向量执行中断服务程序。

在图 53.1.1 中，STM32 在复位后，先从 0X08000004 地址取出复位中断向量的地址，并跳转到复位中断服务程序，如图标号①所示；在复位中断服务程序执行完之后，会跳转到我们的 main 函数，如图标号②所示；而我们的 main 函数一般都是一个死循环，在 main 函数执行过程中，如果收到中断请求（发生重中断），此时 STM32 强制将 PC 指针指回中断向量表处，如图标号③所示；然后，根据中断源进入相应的中断服务程序，如图标号④所示；在执行完中断服务程序以后，程序再次返回 main 函数执行，如图标号⑤所示。

当加入 IAP 程序之后，程序运行流程如图 53.1.2 所示：

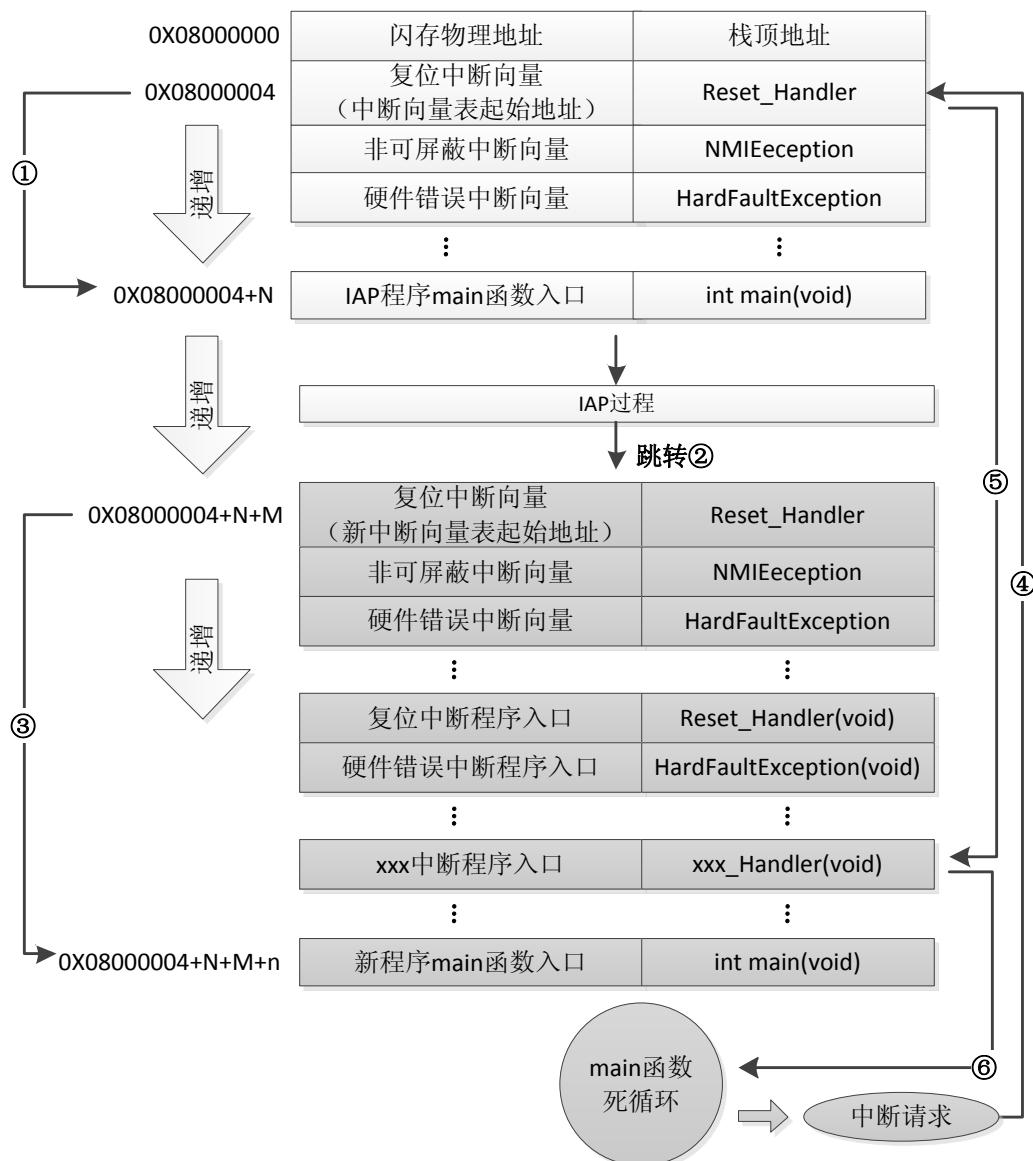


图 53.1.2 加入 IAP 之后程序运行流程图



在图 53.1.2 所示流程中，STM32 复位后，还是从 0X08000004 地址取出复位中断向量的地址，并跳转到复位中断服务程序，在运行完复位中断服务程序之后跳转到 IAP 的 main 函数，如图标号①所示，此部分同图 53.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32 的 FLASH，灰底部分）。新程序的复位中断向量起始地址为 0X08000004+N+M），跳转至新写入程序的复位向量表，取出新程序的复位中断向量的地址，并跳转执行新程序的复位中断服务程序，随后跳转至新程序的 main 函数，如图标号②和③所示，同样 main 函数为一个死循环，并且注意到此时 STM32 的 FLASH，在不同位置上，共有两个中断向量表。

在 main 函数执行过程中，如果 CPU 得到一个中断请求，PC 指针仍强制跳转到地址 0X08000004 中断向量表处，而不是新程序的中断向量表，如图标号④所示；程序再根据我们设置的中断向量表偏移量，跳转到对应中断源新的中断服务程序中，如图标号⑤所示；在执行完中断服务程序后，程序返回 main 函数继续运行，如图标号⑥所示。

通过以上两个过程的分析，我们知道 IAP 程序必须满足两个要求：

- 1) 新程序必须在 IAP 程序之后的某个偏移量为 x 的地址开始；
- 2) 必须将新程序的中断向量表相应的移动，移动的偏移量为 x；

本章，我们有 2 个 APP 程序，一个为 FLASH 的 APP，程序在 FLASH 中运行，另外一个位 SRAM 的 APP，程序运行在 SRAM 中，图 53.1.2 虽然是针对 FLASH APP 来说的，但是在 SRAM 里面运行的过程和 FLASH 基本一致，只是需要设置向量表的地址为 SRAM 的地址。

1.APP 程序起始地址设置方法

随便打开一个之前的实例工程，点击 Options for Target→Target 选项卡，如图 53.1.3 所示：

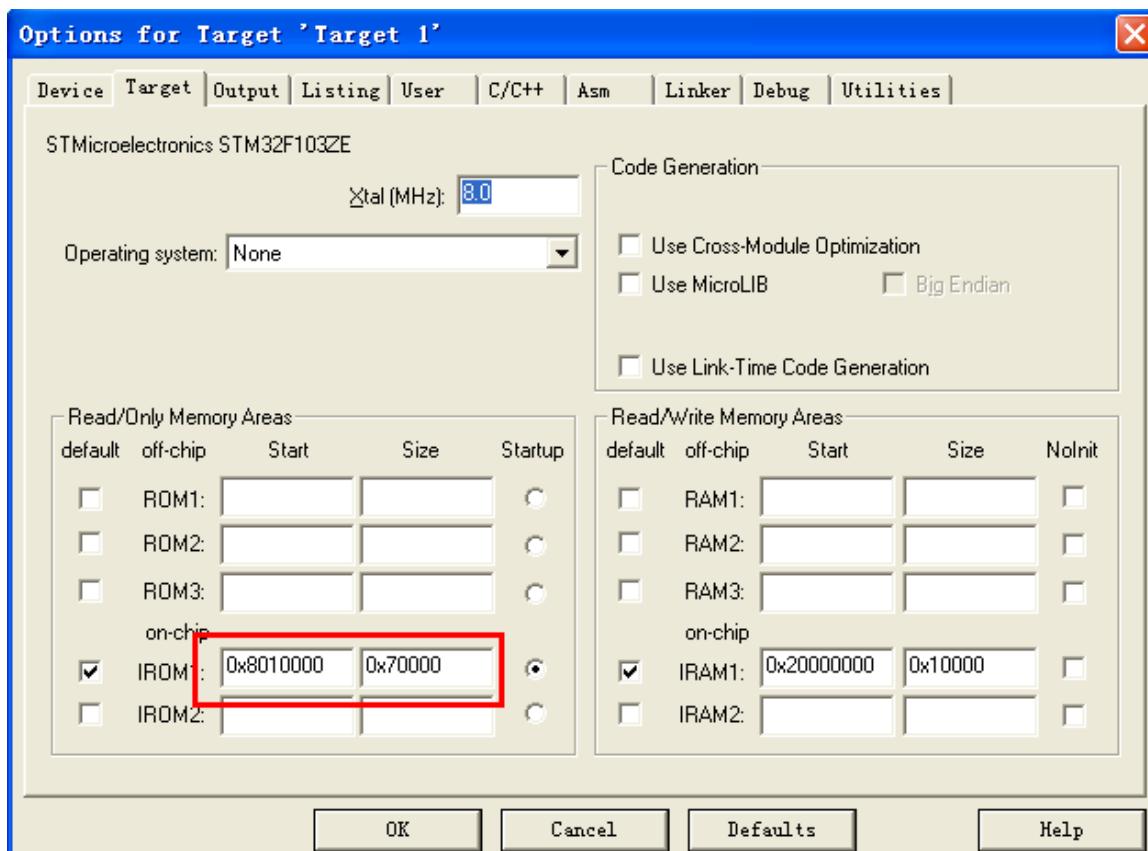


图 53.1.3 FLASH APP Target 选项卡设置

默认的条件下，图中 IROM1 的起始地址(Start)一般为 0X08000000，大小(Size)为 0X80000，即从 0X08000000 开始的 512K 空间为我们的程序存储(因为我们的 STM32F103ZET6 的 FLASH



大小是 512K)。而图中，我们设置起始地址 (Start) 为 0X08010000，即偏移量为 0X10000 (64K 字节)，因而，留给 APP 用的 FLASH 空间 (Size) 只有 0X80000-0X10000=0X70000 (448K 字节) 大小了。设置好 Start 和 Szie，就完成 APP 程序的起始地址设置。

这里的 64K 字节，需要大家根据 Bootloader 程序大小进行选择，比如我们本章的 Bootloader 程序为 22K 左右，理论上我们只需要确保 APP 起始地址在 Bootloader 之后，并且偏移量为 0X200 的倍数即可 (相关知识，请参考：<http://www.openedy.com/posts/list/392.htm>)。这里我们选择 64K (0X10000) 字节，留了一些余量，方便 Bootloader 以后的升级修改。

这是针对 FLASH APP 的起始地址设置，如果是 SRAM APP，那么起始地址设置如图 53.1.4 所示：

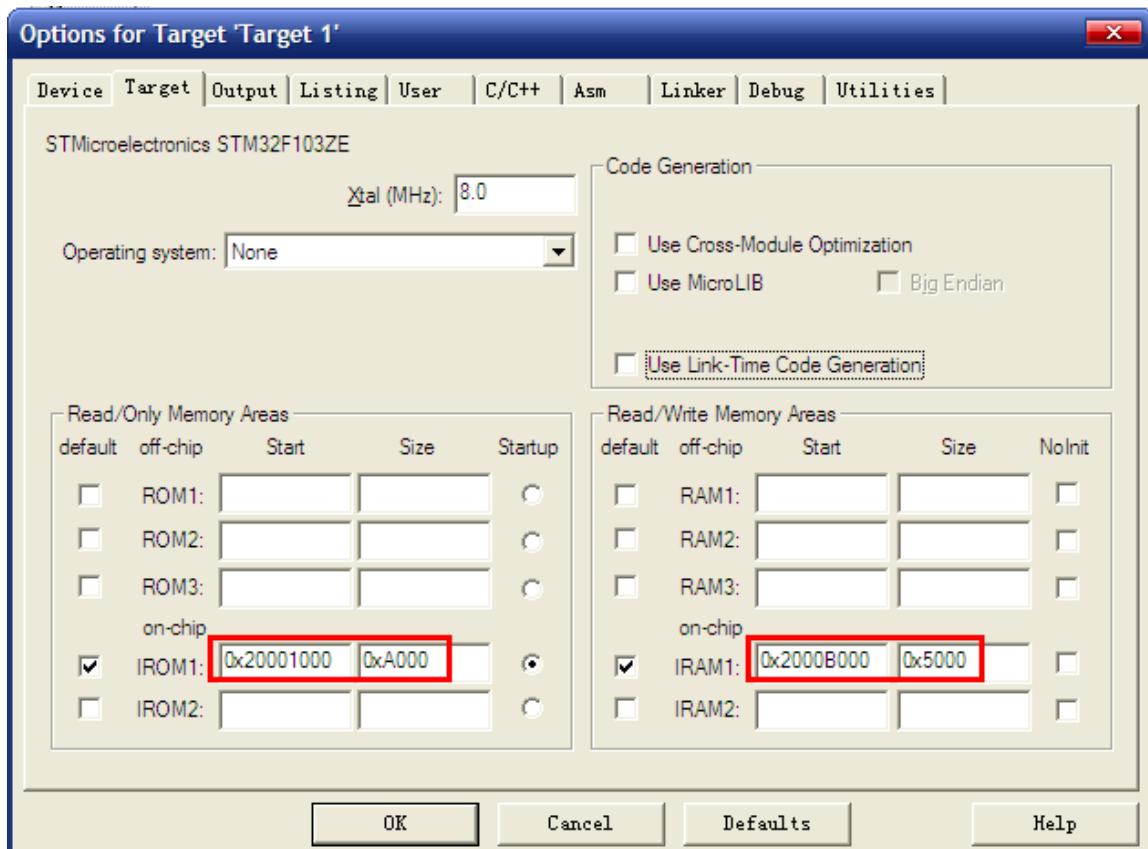


图 53.1.4 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址 (Start) 定义为：0X20001000，大小为 0XA000 (40K 字节)，即从地址 0X20000000 偏移 0X1000 开始，存放 APP 代码。因为整个 STM32F103ZET6 的 SRAM 大小为 64K 字节，所以 IRAM1 (SRAM) 的起始地址变为 0X2000B000 (0X20001000+0XA000=0X2000B000)，大小只有 0X5000 (20K 字节)。这样，整个 STM32F103ZET6 的 SRAM 分配情况为：最开始的 4K 给 Bootloader 程序使用，随后的 40K 存放 APP 程序，最后 20K，用作 APP 程序的内存。这个分配关系大家可以自己的实际情况修改，不一定和我们这里的设置一模一样，不过也需要注意，保证偏移量为 0X200 的倍数 (我们这里为 0X1000)。

2. 中断向量表的偏移量设置方法

之前我们讲解过，在系统启动的时候，会首先调用 systemInit 函数初始化时钟系统，同时 systemInit 还完成了中断向量表的设置，我们可以打开 systemInit 函数，看看函数体的结尾处有这样几行代码：



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

#else
    SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;
                /* Vector Table Relocation in Internal FLASH. */

#endif
```

从代码可以理解，VTOR 寄存器存放的是中断向量表的起始地址。默认的情况 VECT_TAB_SRAM 是没有定义,所以执行 SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; 对于 FLASH APP，我们设置为 FLASH_BASE+偏移量 0x10000,所以我们可以在 FLASH APP 的 main 函数最开头处添加如下代码实现中断向量表的起始地址的重设：

```
SCB->VTOR = FLASH_BASE | 0x10000;
```

以上是 FLASH APP 的情况，当使用 SRAM APP 的时候，我们设置起始地址为：SRAM_base+0x1000,同样的方法，我们在 SRAM APP 的 main 函数最开始处，添加下面代码：

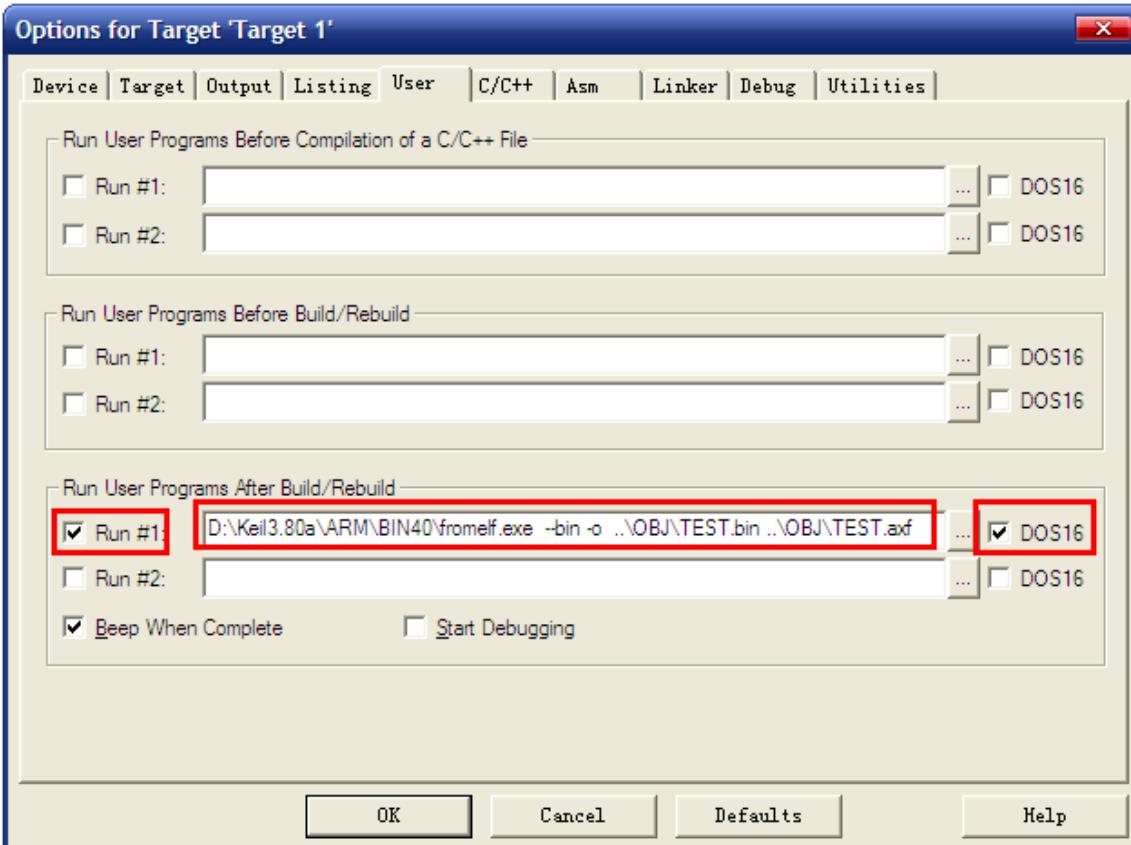
```
SCB->VTOR = SRAM_BASE | 0x1000;
```

这样，我们就完成了中断向量表偏移量的设置。

通过以上两个步骤的设置，我们就可以生成 APP 程序了，只要 APP 程序的 FLASH 和 SRAM 大小不超过我们的设置即可。不过 MDK 默认生成的文件是.hex 文件，并不方便我们用作 IAP 更新，我们希望生成的文件是.bin 文件，这样可以方便进行 IAP 升级（至于为什么，请大家自行百度 HEX 和 BIN 文件的区别！）。这里我们通过 MDK 自带的格式转换工具 fromelf.exe，来实现.axf 文件到.bin 文件的转换。该工具在 MDK 的安装目录\ARM\BIN40 文件夹里面。

fromelf.exe 转换工具的语法格式为：fromelf [options] input_file。其中 options 有很多选项可以设置，详细使用请参考光盘《mdk 如何生成 bin 文件.pdf》。

本章，我们通过在 MDK 点击 Options for Target→User 选项卡，在 Run User Programs After Build/Rebuild 栏，勾选 Run#1 和 DOS16，并写入：D:\Keil3.80a\ARM\BIN40\fromelf.exe --bin -o ..\OBJ\TEST.bin ..\OBJ\TEST.axf，如图 53.1.6 所示：



通过这一步设置，我们就可以在 MDK 编译成功之后，调用 fromelf.exe（注意，我的 MDK 是安装在 D:\Keil3.80A 文件夹下，如果你是安装在其他目录，请根据你自己的目录修改 fromelf.exe 的路径），根据当前工程的 TEST.axf（如果是其他的名字，请记住修改，这个文件存放在 OBJ 目录下面，格式为 xxx.axf），生成一个 TEST.bin 的文件。并存放在 axf 文件相同的目录下，即工程的 OBJ 文件夹里面。在得到.bin 文件之后，我们只需要将这个 bin 文件传送给单片机，即可执行 IAP 升级。

最后再来 APP 程序的生成步骤：

1) 设置 APP 程序的起始地址和存储空间大小

对于在 FLASH 里面运行的 APP 程序，我们可以按照图 53.1.3 的设置。对于 SRAM 里面运行的 APP 程序，我们可以参考图 53.1.4 的设置。

2) 设置中断向量表偏移量

这一步按照上面讲解，重新设置 SCB->VTOR 的值即可。

3) 设置编译后运行 fromelf.exe，生成.bin 文件。

通过在 User 选项卡，设置编译后调用 fromelf.exe，根据.axf 文件生成.bin 文件，用于 IAP 更新。

以上 3 个步骤，我们就可以得到一个.bin 的 APP 程序，通过 Bootlader 程序即可实现更新。大家可以打开我们光盘的两个 APP 工程，熟悉这些设置。

53.2 硬件设计

本章实验（Bootloader 部分）功能简介：开机的时候先显示提示信息，然后等待串口输入接收 APP 程序（无校验，一次性接收），在串口接收到 APP 程序之后，即可执行 IAP。如果

是 SRAM APP，通过按下 KEY0 即可执行这个收到的 SRAM APP 程序。如果是 FLASH APP，则需要先按下 WK_UP 按键，将串口接收到的 APP 程序存放到 STM32 的 FLASH，之后再按 KEY2 既可以执行这个 FLASH APP 程序。通过 KEY1 按键，可以手动清除串口接收到的 APP 程序。DS0 用于指示程序运行状态。

本实验用到的资源如下：

- 1) 指示灯 DS0
- 2) 四个按键 (KEY0/KEY1/KEY2/WK_UP)
- 3) 串口
- 4) TFTLCD 模块

这些用到的硬件，我们在之前都已经介绍过，这里就不再介绍了。

53.3 软件设计

本章，我们总共需要 3 个程序：1，Bootloader；2，FLASH APP；3）SRAM APP；其中，我们选择之前做过的 RTC 实验（在第二十章介绍）来做为 FLASH APP 程序（起始地址为 0X08010000），选择触摸屏实验（在第三十一章介绍）来做 SRAM APP 程序（起始地址为 0X20001000）。Bootloader 则是通过 TFTLCD 显示实验（在第十八章介绍）修改得来。本章，关于 SRAM APP 和 FLASH APP 的生成比较简单，我们就不细说，请大家结合光盘源码，以及 53.1 节的介绍，自行理解。本章软件设计仅针对 Bootloader 程序。

打开本实验工程，可以看到我们增加了 IAP 组，在组下面添加了 iap.c 文件以及其头文件 isp.h。

打开 iap.c，代码如下：

```
#include "sys.h"
#include "delay.h"
#include "uart.h"
#include "stmflash.h"
#include "iap.h"

iapfun jump2app;
u16 iapbuf[1024];
//appxaddr:应用程序的起始地址
//appbuf:应用程序 CODE.
//appsize:应用程序大小(字节).
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 appsize)
{
    u16 t;
    u16 i=0;
    u16 temp;
    u32 fwaddr=appxaddr;//当前写入的地址
    u8 *dfu=appbuf;
    for(t=0;t<appsize;t+=2)
    {
        temp=(u16)dfu[1]<<8;
        temp+=(u16)dfu[0];
```



```

dfu+=2;//偏移 2 个字节
iapbuf[i++]=temp;
if(i==1024)
{
    i=0;
    STMFLASH_Write(fwaddr,iapbuf,1024);
    fwaddr+=2048;//偏移 2048 16=2*8.所以要乘以 2.
}
if(i)STMFLASH_Write(fwaddr,iapbuf,i);//将最后的一些内容字节写进去.
}

//跳转到应用程序段
//appxaddr:用户代码起始地址.
void iap_load_app(u32 appxaddr)
{
    if(((vu32*)appxaddr)&0x2FFE0000)==0x20000000) //检查栈顶地址是否合法.
    {
        jump2app=(iapfun)*(vu32*)(appxaddr+4);
        //用户代码区第二个字为程序开始地址(复位地址)
        MSR_MSP((vu32*)appxaddr);
        //初始化 APP 堆栈指针(用户代码区的第一个字用于存放栈顶地址)
        jump2app(); //跳转到 APP.
    }
}

```

该文件总共只有 2 个函数，其中，`iap_write_appbin` 函数用于将存放在串口接收 `buf` 里面的 APP 程序写入到 FLASH。`iap_load_app` 函数，则用于跳转到 APP 程序运行，其参数 `appxaddr` 为 APP 程序的起始地址，程序先判断栈顶地址是否合法，在得到合法的栈顶地址后，通过 `MSR_MSP` 函数（该函数在 `sys.c` 文件）设置栈顶地址，最后通过一个虚拟的函数（`jump2app`）跳转到 APP 程序执行代码，实现 IAP→APP 的跳转。

打开 `iap.h` 代码如下：

```

#ifndef __IAP_H__
#define __IAP_H__
#include "sys.h"
typedef void (*iapfun)(void); //定义一个函数类型的参数.
#define FLASH_APP1_ADDR 0x080010000
//第一个应用程序起始地址(存放在 FLASH)
//保留 0X08000000~0X0800FFFF 的空间为 Bootloader 使用
void iap_load_app(u32 appxaddr); //跳转到 APP 程序执行
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 applen); //在指定地址开始,写入 bin
#endif

```

这部分代码比较简单，。本章，我们是通过串口接收 APP 程序的，我们将 `uart.c` 和 `uart.h` 做了稍微修改，在 `uart.h` 中，我们定义 `USART_REC_LEN` 为 55K 字节，也就是串口最大一次可以接收 55K 字节的数据，这也是本 Bootloader 程序所能接收的最大 APP 程序大小。然后新增



一个 USART_RX_CNT 的变量，用于记录接收到的文件大小，而 USART_RX_STA 不再使用。打开 usart.c，可以看到我们修改 USART1_IRQHandler 部分代码如下：

```
//串口1中断服务程序
//注意,读取 USARTx->SR 能避免莫名其妙的错误
u8 USART_RX_BUF[USART_REC_LEN] __attribute__ ((at(0X20001000)));
//接收缓冲,最大 USART_REC_LEN 个字节,起始地址为 0X20001000.
//接收状态
//bit15, 接收完成标志
//bit14, 接收到 0xd
//bit13~0, 接收到的有效字节数目
u16 USART_RX_STA=0;           //接收状态标记
u16 USART_RX_CNT=0;           //接收的字节数
void USART1_IRQHandler(void)
{
    u8 res;
#ifdef OS_CRITICAL_METHOD
    //如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
    OSIntEnter();
#endif
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)//接收到数据
    {
        res=USART_ReceiveData(USART1);
        if(USART_RX_CNT<USART_REC_LEN)
        {
            USART_RX_BUF[USART_RX_CNT]=res;
            USART_RX_CNT++;
        }
    }
#endif
    //如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
    OSIntExit();
}
```

这里，我们指定 USART_RX_BUF 的地址是从 0X20001000 开始，该地址也就是 SRAM APP 程序的起始地址！然后在 USART1_IRQHandler 函数里面，将串口发送过来的数据，全部接收到 USART_RX_BUF，并通过 USART_RX_CNT 计数。代码比较简单，我们就不多说了。

最后我们看看 main 函数如下：

```
int main(void)
{
    u8 t;
    u8 key;
    u16 oldcount=0; //老的串口接收数据值
    u16 applenth=0; //接收到的 app 代码长度
```



```
u8 clearflag=0;
uart_init(256000);      //串口初始化为 256000
delay_init();            //延时初始化
LCD_Init();              //液晶初始化
LED_Init();              //初始化与 LED 连接的硬件接口
KEY_Init();              //按键初始化
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,200,16,16,"Warship STM32");
LCD_ShowString(60,70,200,16,16,"IAP TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/24");
LCD_ShowString(60,130,200,16,16,"WK_UP:Copy APP2FLASH");
LCD_ShowString(60,150,200,16,16,"KEY1:Erase SRAM APP");
LCD_ShowString(60,170,200,16,16,"KEY0:Run SRAM APP");
LCD_ShowString(60,190,200,16,16,"KEY2:Run FLASH APP");
POINT_COLOR=BLUE;
//显示提示信息
POINT_COLOR=BLUE;//设置字体为蓝色
while(1)
{
    if(USART_RX_CNT)
    {
        if(oldcount==USART_RX_CNT)
        //新周期内,没有收到任何数据,认为本次数据接收完成.
        {
            applenth=USART_RX_CNT;
            oldcount=0;
            USART_RX_CNT=0;
            printf("用户程序接收完成!\r\n");
            printf("代码长度:%dBytes\r\n",applenth);
        }else oldcount=USART_RX_CNT;
    }
    t++; delay_ms(10);
    if(t==30)
    {
        LED0=!LED0; t=0;
        if(clearflag)
        {
            clearflag--;
            if(clearflag==0)LCD_Fill(60,210,240,210+16,WHITE);//清除显示
        }
    }
    key=KEY_Scan(0);
}
```



```
if(key==KEY_UP)
{
    if(applenth)
    {
        printf("开始更新固件...\r\n");
        LCD_ShowString(60,210,200,16,16,"Copying APP2FLASH... ");
        if(((vu32*)(0X20001000+4))&0xFF000000)==0x08000000)
        //判断是否为 0X08XXXXXX.
        {
            iap_write_appbin(FLASH_APP1_ADDR,USART_RX_BUF,
            applenth); //更新 FLASH 代码
            LCD_ShowString(60,210,200,16,16,"Copy APP Successed!!");
            printf("固件更新完成!\r\n");
        }
        else
        {
            LCD_ShowString(60,210,200,16,16,"Illegal FLASH APP!   ");
            printf("非 FLASH 应用程序!\r\n");
        }
    }
    else
    {
        printf("没有可以更新的固件!\r\n");
        LCD_ShowString(60,210,200,16,16,"No APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}
if(key==KEY_DOWN)
{
    if(applenth)
    {
        printf("固件清除完成!\r\n");
        LCD_ShowString(60,210,200,16,16,"APP Erase Successed!");
        applenth=0;
    }
    else
    {
        printf("没有可以清除的固件!\r\n");
        LCD_ShowString(60,210,200,16,16,"No APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}
if(key==KEY_LEFT)
{
```



```
printf("开始执行 FLASH 用户代码!!\r\n");
if(((*(vu32*)(FLASH_APP1_ADDR+4))&0xFF000000)==0x08000000)
//判断是否为 0X08XXXXXX.
{
    iap_load_app(FLASH_APP1_ADDR);//执行 FLASH APP 代码
}
else
{
    printf("非 FLASH 应用程序,无法执行!\r\n");
    LCD_ShowString(60,210,200,16,16,"Illegal FLASH APP!");
}
clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}

if(key==KEY_RIGHT)
{
    printf("开始执行 SRAM 用户代码!!\r\n");
    if(((*(vu32*)(0X20001000+4))&0xFF000000)==0x20000000)
    //判断是否为 0X20XXXXXX.
    {
        iap_load_app(0X20001000);//SRAM 地址
    }
    else
    {
        printf("非 SRAM 应用程序,无法执行!\r\n");
        LCD_ShowString(60,210,200,16,16,"Illegal SRAM APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}

}
```

该段代码，实现了串口数据处理，以及 IAP 更新和跳转等各项操作。Bootloader 程序就设计完成了，但是一般要求 bootloader 程序越小越好（给 APP 省空间嘛），所以，本章我们把一些不需要用到的.c 文件全部去掉，最后得到工程截图如图 53.3.1 所示：

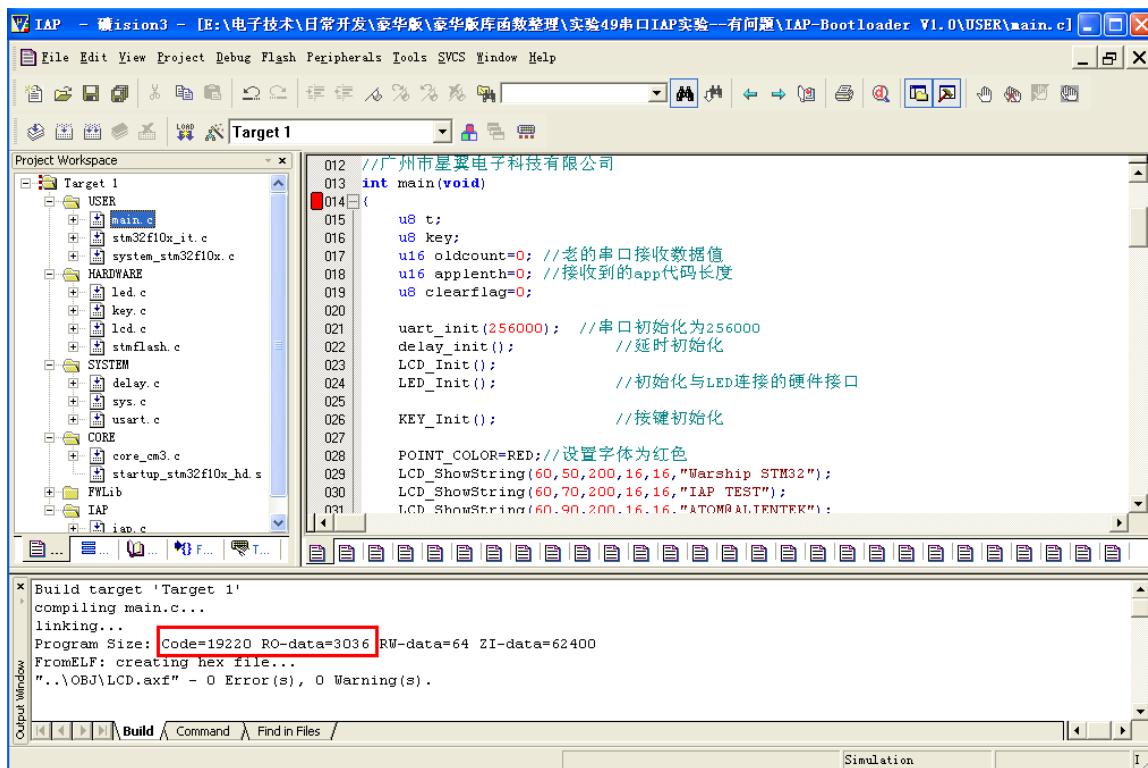


图 53.3.1 Bootloader 工程截图

从上图可以看出，虽然去掉了一些不用的.c 文件，但是 Bootloader 大小还是有 22K 左右，比较大，主要原因是液晶驱动和 printf 占用了比较多的 flash，如果大家想进一步删减，可以去掉 LCD 显示和 printf 等，不过我们在本章为了演示效果，所以保留了这些代码。

至此，本实验的软件设计部分结束。

FLASH APP 和 SRAM APP 两部分代码，我们在实验目录下提供了两个实验供大家参考，不过要提醒大家，根据我们的设置，FLASH APP 的起始地址必须是 0X08010000，而 SRAM APP 的起始地址必须是 0X20001000。

53.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，得到，如图 53.4.1 所示：



图 53.4.1 IAP 程序界面

此时，我们可以通过串口，发送 FLASH APP 或者 SRAM APP 到战舰 STM32 开发板，如图 53.4.2 所示：



图 53.4.2 串口发送 APP 程序界面



先用串口调试助手的打开文件按钮（如图标号 1 所示），找到 APP 程序生成的.bin 文件，然后设置波特率为 256000（为了提高速度，Bootloader 程序将波特率被设置为 256000 了），最后点击发送文件（图中标号 3 所示），将.bin 文件发送给战舰 STM32 开发板。

在收到 APP 程序之后，我们就可以通过 KEY0/KEY2 运行这个 APP 程序了（如果是 FLASH APP，则先需要通过 WK_UP 将其存入对应 FLASH 区域）。

第五十四章 触控 USB 鼠标实验

STM32F103 系列芯片都自带了 USB，不过 STM32F103 的 USB 都只能用来做设备，而不能用作主机。既便如此，对于一般应用来说已经足够了。本章，我们将向大家介绍如何在 ALIENTEK 战舰 STM32 开发板上虚拟一个 USB 鼠标。本章分为如下几个部分：

- 54.1 USB 简介
- 54.2 硬件设计
- 54.3 软件设计
- 54.4 下载验证

54.1 USB 简介

USB ,是英文 Universal Serial BUS (通用串行总线) 的缩写, 而其中文简称为“通串线, 是一个外部总线标准, 用于规范电脑与外部设备的连接和通讯。是应用在 PC 领域的接口技术。USB 接口支持设备的即插即用和热插拔功能。USB 是在 1994 年底由英特尔、康柏、IBM、Microsoft 等多家公司联合提出的。

USB 发展到现在已经有 USB1.0/1.1/2.0/3.0 等多个版本。目前用的最多的就是 USB1.1 和 USB2.0, USB3.0 目前已经开始普及。STM32F103 自带的 USB 符合 USB2.0 规范。

标准 USB 共四根线组成,除 VCC/GND 外,另外为 D+,D-; 这两根数据线采用的是差分电压的方式进行数据传输的。在 USB 主机上, D- 和 D+都是接了 15K 的电阻到低的, 所以在没有设备接入的时候, D+、D-均是低电平。而在 USB 设备中, 如果是高速设备, 则会在 D+上接一个 1.5K 的电阻到 VCC, 而如果是低速设备, 则会在 D-上接一个 1.5K 的电阻到 VCC。这样当设备接入主机的时候, 主机就可以判断是否有设备接入, 并能判断设备是高速设备还是低速设备。接下来, 我们简单介绍一下 STM32 的 USB 控制器。

STM32F103 的 MCU 自带 USB 从控制器, 符合 USB 规范的通信连接; PC 主机和微控制器之间的数据传输是通过共享一专用的数据缓冲区来完成的, 该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定, 每个端点最大可使用 512 字节缓冲区 (专用的 512 字节, 和 CAN 共用), 最多可用于 16 个单向或 8 个双向端点。USB 模块同 PC 主机通信, 根据 USB 规范实现令牌分组的检测, 数据发送/接收的处理, 和握手分组的处理。整个传输的格式由硬件完成, 其中包括 CRC 的生成和校验。

每个端点都有一个缓冲区描述块, 描述该端点使用的缓冲区地址、大小和需要传输的字节数。当 USB 模块识别出一个有效的功能/端点的令牌分组时, (如果需要传输数据并且端点已配置)随之发生相关的数据传输。USB 模块通过一个内部的 16 位寄存器实现端口与专用缓冲区的数据交换。在所有的数据传输完成后, 如果需要, 则根据传输的方向, 发送或接收适当的握手分组。在数据传输结束时, USB 模块将触发与端点相关的中断, 通过读状态寄存器和/或者利用不同的中断来处理。

USB 的中断映射单元: 将可能产生中断的 USB 事件映射到三个不同的 NVIC 请求线上:

1、USB 低优先级中断(通道 20): 可由所有 USB 事件触发(正确传输, USB 复位等)。固件在处理中断前应当首先确定中断源。

2、USB 高优先级中断(通道 19): 仅能由同步和双缓冲批量传输的正确传输事件触发, 目的是保证最大的传输速率。

3、USB 唤醒中断(通道 42): 由 USB 挂起模式的唤醒事件触发。

USB 设备框图如图 54.1.1 所示:

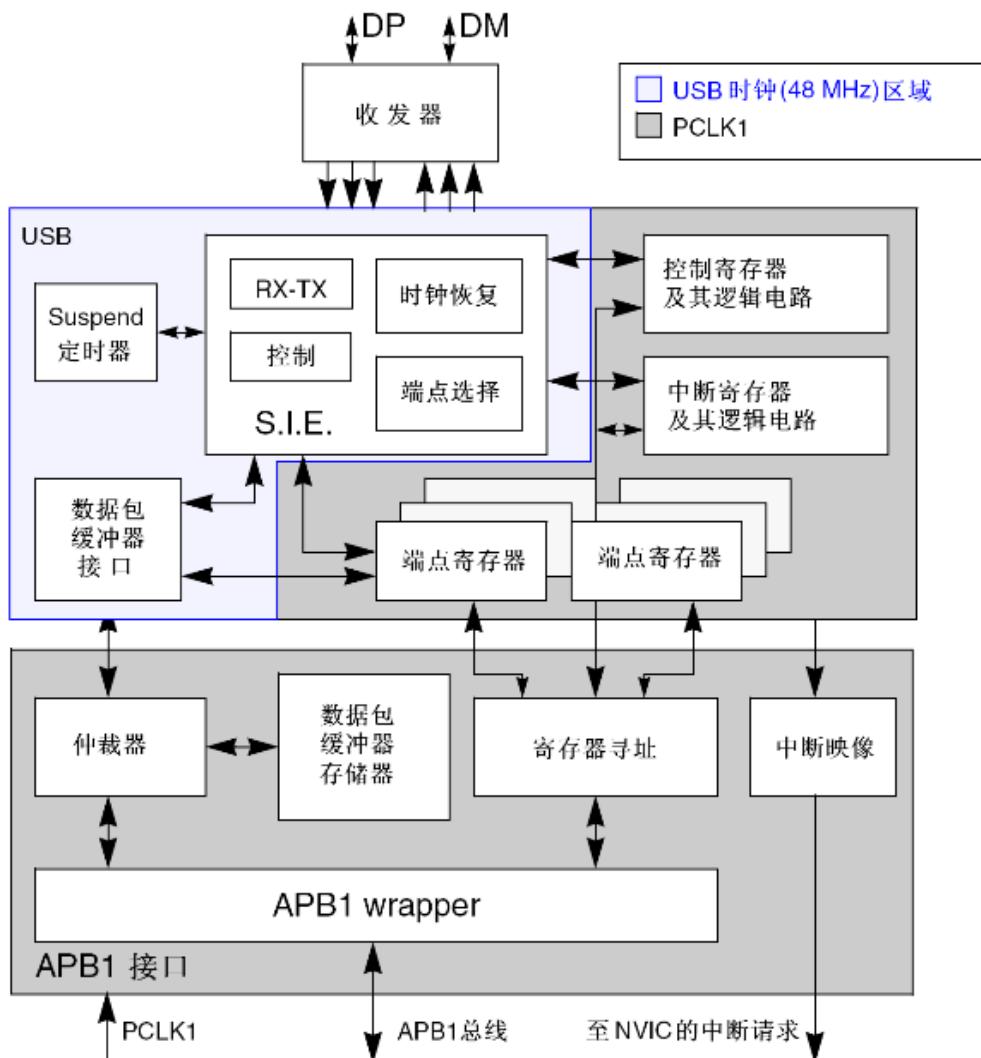


图 54.1.1 USB 设备框图

整个 USB 通信的详细过程是很复杂的，本书篇幅有限，这里我们就不再详细介绍其各个环节，感兴趣的读者，可以去看看电脑圈圈的《圈圈教你玩 USB》这本书，该书对 USB 的讲解是很详细的。USB 部分，ST 提供了几个例程，这些例程对于我们了解 STM32F103 的 USB 会有不少帮助，尤其在不懂的时候，看看 ST 的例程，会有意想不到的收获。本实验的 USB 部分就是移植 ST 的 JoyStickMouse 例程相关部分而来，再加上我们的触摸屏，做成一个触控鼠标。ST 提供的 USB 例程在 X:\Keil3.80\ARM\Examples\ST\STM32F10xUSBLib\Datas 文件夹下（X 是你的安装盘）。

54.2 硬件设计

本章实验功能简介：开机的时候先检测触摸屏是否校准过，如果没有，则校准。如果校准过了，则开始触摸屏画图，然后将我们的坐标数据上传到电脑（假定 USB 已经配置成功了，DS1 亮），这样就可以用触摸屏来控制电脑的鼠标了。我们用按键 KEY0 模拟鼠标右键，用按键 KEY2 模拟鼠标左键，用按键 WK_UP 和 KEY1 模拟鼠标滚轮的上下滚动。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：



- 1) 指示灯 DS0 、 DS1
- 2) 四个按键 (KEY0/KEY1/KEY2/WK_UP)
- 3) 串口
- 4) TFTLCD 模块
- 5) USB 接口

前面 5 部分, 在之前的实例中都介绍过了, 我们在此就不介绍了。接下来看看我们电脑 USB 与 STM32 的 USB 连接口。ALIENTEK 战舰 STM32 采用的是 5PIN 的 MiniUSB 接头, 用来和 STM32 的 USB 相连接, 连接电路如图 54.2.1 所示:

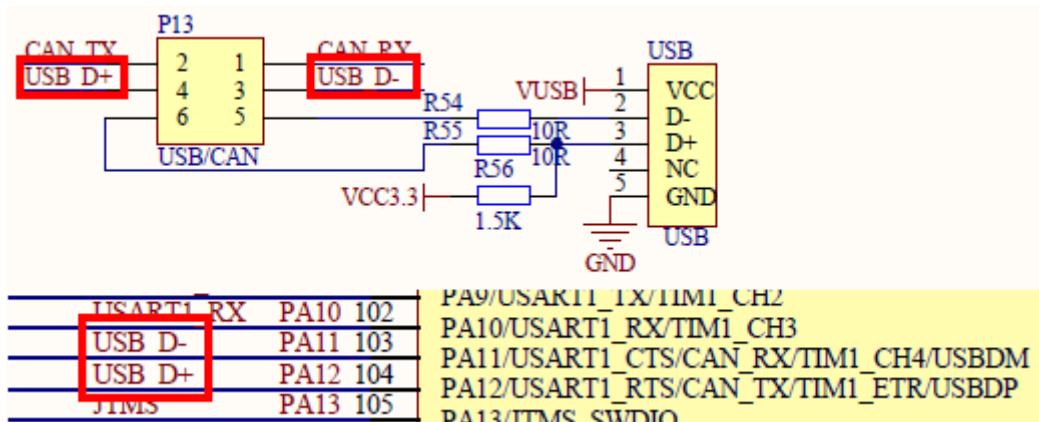


图 54.2.1 MiniUSB 接口与 STM32 的连接电路图

从上图可以看出, USB 座没有直接连接到 STM32 上面, 而是通过 P13 转接, 所以我们需要通过跳线帽将 PA11 和 PA12 分别连接到 D- 和 D+, 如图 54.2.2 所示:

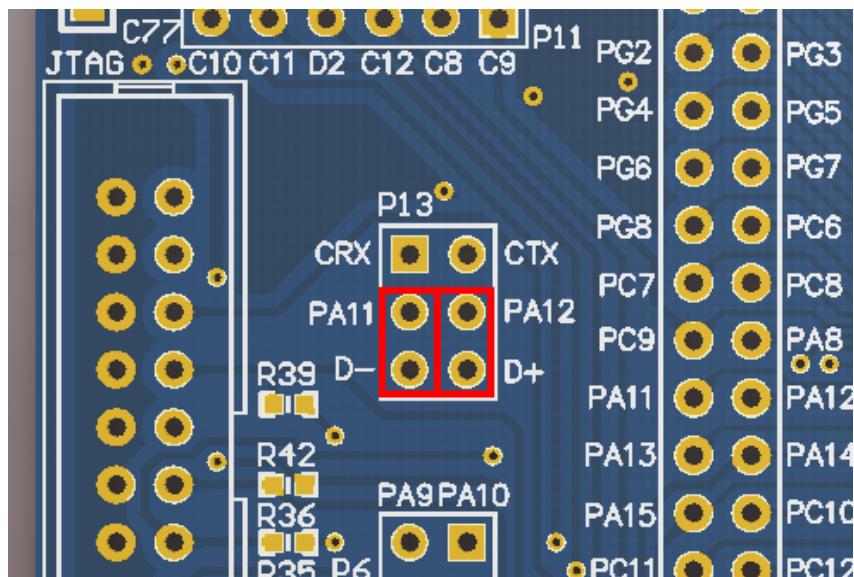


图 54.2.2 硬件连接示意图

54.3 软件设计

打开我们触控 USB 鼠标实验工程目录可以看到, 我们在 USB 文件夹下面新建 LIB 和



CONFIG 两个文件夹，分别用来存放与 USB 核相关的代码以及配置部分代码。这两部分代码我们就不细说了（详见光盘本例程源码），给大家看看在这两个文件夹内的代码都有哪些，如图 54.3.1 所示：

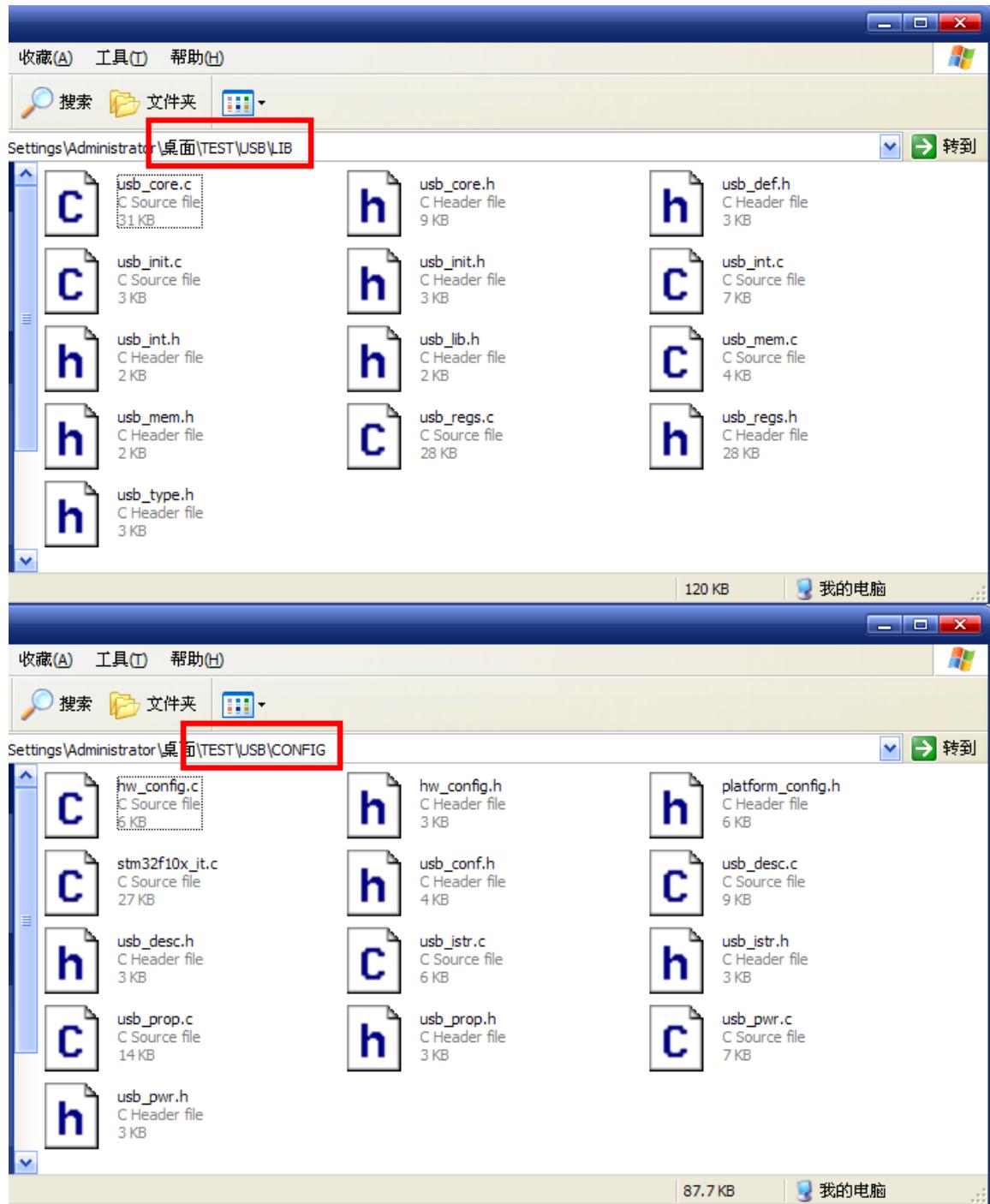


图 54.3.1 USB 相关部分代码

以上代码，就是 ST 提供的 USB 固件库代码，LIB 文件夹内的是固件库文件，而 CONFIG 文件夹内的是一些配置文件。LIB 文件夹下的.c 文件源码来自：X:\Keil3.80A\ARM\RV31\LIB\ST\STM32F10x\USB 目录下，而.h 文件来自 X:\Keil3.80A\ARM\INC\ST\STM32F10x\USB 目录下。CONFIG 文件夹下的.c 和.h 文件源码来自 X:\Keil3.80A\ARM\Examples\ST\STM32F10xUSBLib\Demos\JoyStickMouse 下的 source 和 include 文件夹（X 为你安装 MDK 的磁盘）。



现在，我们先来介绍一下 LIB 文件夹下的几个.c 文件。

usb_regs.c 文件，该文件主要负责 USB 控制寄存器的底层操作，里面有个中 USB 寄存器的底层操作函数。

usb_init.c 文件，该文件里面只有一个函数：USB_Init，用于 USB 控制器的初始化，不过对 USB 控制器的初始化，是 USB_Init 调用用其他文件的函数实现的，USB_Init 只不过是把他们连接一下罢了，这样使得代码比较规范。

usb_int.c 文件，该文件里面只有两个函数 CTR_LP 和 CTR_HP，CTR_LP 负责 USB 低优先级中断的处理。而 CTR_HP 负责 USB 高优先级中断的处理。

usb_mem.c 文件，该文件用于处理 PMA 数据，PMA 全称为 Packet memory area，是 STM32 内部用于 USB/CAN 的专用数据缓冲区，该文件内也只有 2 个函数即：PMAToUserBufferCopy 和 UserToPMABufferCopy，分别用于将 USB 端点的数据传送给主机和主机的数据传送到 USB 端点。

usb_croe.c 文件，该文件用于处理 USB2.0 协议。

以上几个文件具有很强的独立性，除特殊情况，不需要用户修改，直接调用内部的函数即可。接着我们介绍 CONFIG 文件夹里面的几个.c 文件。

usb_pwr.c 文件，该文件用于 USB 控制器的电源管理；

usb_istr.c 文件，该文件用于处理 USB 中断。

usb_prop.c 文件，该文件用于处理 Joystick 的相关事件，包括 Joystick 的初始化、复位等等操作。

usb_desc.c 文件，该文件用于 Joystick 描述符的处理。

hw_config.c 文件，该文件用于硬件的配置，比如初始化 USB 时钟、USB 中断、低功耗模式处理等。

另外 stm32f10x_it.c 就是中断服务函数的集合了，这里面我们只保留了两个函数，第一个函数是：USB_LP_CAN1_RX0_IRQHandler 函数，我们在该函数里面调用 USB_Istr 函数，用于处理 USB 发生的各种中断，注意，这些代码，有些是经过修改了的，具体请看光盘的源码。另外一个函数就是 USBWakeUp_IRQHandler 函数，我们在该函数就做了一件事：清除中断标志。USB 相关代码，就给大家介绍到这里。

接着我们在工程文件里面新建 USB 和 USBCFG 组，分别加入 USB\LIB 下面的代码和 USB\CONFIG 下面的代码。然后把 LIB 和 CONFIG 文件夹加入头文件包含路径。

在 main.c 里面，我们修改 main 函数如下：

```
//装载画图界面
void Load_Draw_Dialog(void)
{
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(lcddev.width-24,0,200,16,16,"RST");//显示清屏区域
    POINT_COLOR=RED;//设置画笔蓝色
}
//计算 x1,x2 的绝对值
u32 usb_abs(u32 x1,u32 x2)
{
    if(x1>x2) return x1-x2;
    else return x2-x1;
```



```
}

//设置 USB 连接/断线
//enable:0,断开
//      1,允许连接
void usb_port_set(u8 enable)
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
    if(enable)_SetCNTR(_GetCNTR()&(~(1<<1))); //退出断电模式
    else
    {
        _SetCNTR(_GetCNTR()|(1<<1)); // 断电模式
        GPIOA->CRH&=0XFFF00FFF;
        GPIOA->CRH|=0X00033000;
        PAout(12)=0;
    }
}

int main(void)
{
    u8 key;
    u8 i=0;
    s8 x0;          //发送到电脑端的坐标值
    s8 y0;
    u8 keysta;      //|[0]:0,左键松开;1,左键按下;
                    //|[1]:0,右键松开;1,右键按下
                    //|[2]:0,中键松开;1,中键按下
    u8 tpsta=0;      //0,触摸屏第一次按下;1,触摸屏滑动
    short xlast;      //最后一次按下的坐标值
    short ylast;
    delay_init();      //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);      //串口初始化波特率为 9600
    LED_Init();      //LED 端口初始化
    LCD_Init();      //LCD 初始化
    KEY_Init();      //按键初始化
    TP_Init();      //初始化触摸屏

    POINT_COLOR=RED;
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"USB Mouse TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/24");
    LCD_ShowString(60,130,200,16,16,"KEY_UP:SCROLL +");
    LCD_ShowString(60,150,200,16,16,"KEY_DOWN:SCROLL -");
```



```
LCD_ShowString(60,170,200,16,16,"KEY_RIGHT:RIGHT BTN");
LCD_ShowString(60,190,200,16,16,"KEY_LEFT:LEFT BTN");
delay_ms(1800); usb_port_set(0); //USB 先断开
delay_ms(300); usb_port_set(1); //USB 再次连接
//USB 配置
USB_Interrupts_Config();
Set_USBClock();
USB_Init();
Load_Draw_Dialog();
while(1)
{
    key=KEY_Scan(1);//支持连接
    if(key)
    {
        if(key==KEY_UP)Joystick_Send(0,0,0,1); //发送滚轮数据到电脑
        else if(key==KEY_DOWN)Joystick_Send(0,0,0,(u8)-1); //发送滚轮数据到电脑
        else
        {
            if(key==KEY_LEFT)keysta|=0X01;           //发送鼠标左键
            if(key==KEY_RIGHT)keysta|=0X02;         //发送鼠标右键
            Joystick_Send(keysta,0,0,0);           //发送给电脑
        }
    }else if(keysta)//之前有按下
    {
        keysta=0;
        Joystick_Send(0,0,0,0); //发送松开命令给电脑
    }
    tp_dev.scan(0);
    if(tp_dev.sta&TP_PRES_DOWN) //触摸屏被按下
    {
        //最少移动 5 个单位,才算滑动
        if(((usb_abs(tp_dev.x,xlast)>4)||(usb_abs(tp_dev.y,ylast)>4))&&tpsta==0)//滑动
        {
            xlast=tp_dev.x;           //记录刚按下的坐标
            ylast=tp_dev.y;
            tpsta=1;
        }
        if(tp_dev.x<lcddev.width&&tp_dev.y<lcddev.height)
        {
            if(tp_dev.x>216&&tp_dev.y<16)Load_Draw_Dialog(); //清除
            else TP_Draw_Big_Point(tp_dev.x,tp_dev.y,RED); //画图
            if(bDeviceState==CONFIGURED)
            {

```



```
if(tpsta)//滑动
{
    x0=(xlast-tp_dev.x)*3; //上次坐标值与新坐标值之差,扩大3倍
    y0=(ylast-tp_dev.y)*3;
    xlast=tp_dev.x;        //记录刚按下的坐标
    ylast=tp_dev.y;
    Joystick_Send(keysta,-x0,-y0,0); //发送数据到电脑
    delay_ms(5);
}
}
}

}elseif (tpsta==0){delay_ms(1);}//清除
if(bDeviceState==CONFIGURED)LED1=0;//USB 配置成功, LED1 亮, 否则, 灭
else LED1=1;
i++;
if(i==200) {i=0; LED0=!LED0;}
}
```

在此部分代码用于实现我们在硬件设计部分提到的功能，USB 的配置通过三个函数完成：USB_Interrupts_Config()、Set_USBClock()和 USB_Init()，第一个函数用于设置 USB 唤醒中断和 USB 低优先级数据处理中断，Set_USBClock 函数用于配置 USB 时钟，也就是从 72M 的主频得到 48M 的 USB 时钟（1.5 分频）。最后 USB_Init()函数用于初始化 USB，最主要的就是调用了 Joystick_init 函数，开启了 USB 部分的电源等。这里需要特别说明的是，USB 配置并没有对 PA11 和 PA12 这两个 IO 口进行设置，是因为，一旦开启了 USB 电源（USB_CNTR 的 PDWN 位清零）PA11 和 PA12 将不再作为其他功能使用，仅供 USB 使用，所以在开启了 USB 电源之后不论你怎么配置这两个 IO 口，都是无效的。要在此获取这两个 IO 口的配置权，则需要关闭 USB 电源，也就是置位 USB_CNTR 的 PDWN 位，我们通过 usb_port_set 函数来禁止/允许 USB 连接，在复位的时候，先禁止，再允许，这样每次我们按复位电脑都可以识别到 USB 鼠标，而不需要我们每次都拔 USB 线。

USB 数据发送，我们采用 Joystick_Send 来实现，我们将得到的鼠标数据，在 Joystick_Send 函数里面打包，并通过 USB 端点 1 发送到电脑。

软件设计部分，就给大家介绍到这里。

54.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，在 USB 没有配置成功的时候，其界面同第三十一章的实验是一模一样的，如图 54.4.1 所示：

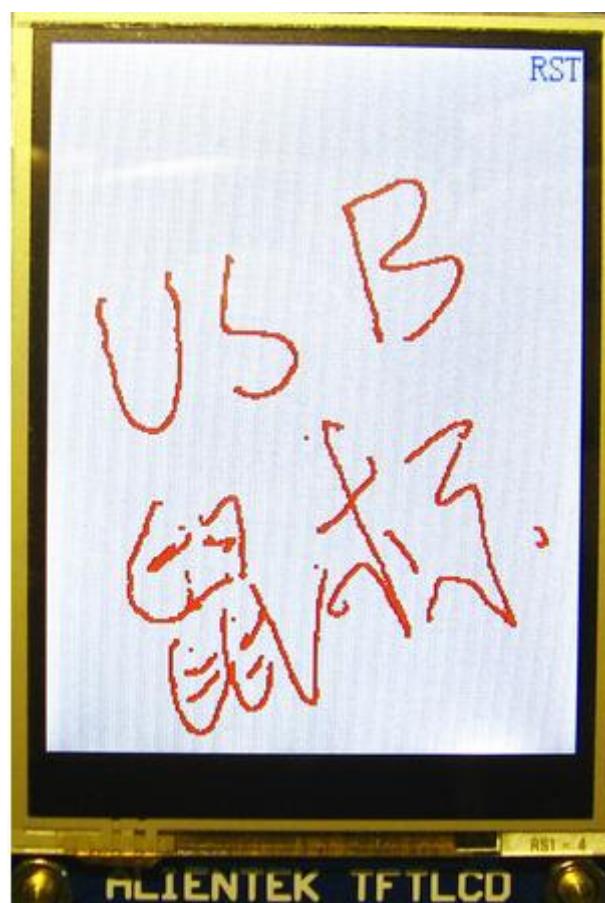


图 54.4.1 USB 无连接时的界面

此时 DS1 不亮，DS0 闪烁，其实就是一个触摸屏画图的功能，而一旦我们将 USB 连接上（将 USB 线接到 USB 接头上，而不是 USB_232 接头上，如果你有两根 USB 线，则可以两个同时都接上，他们不会相互影响），则可以看到 DS1 亮了，而且在电脑上会提示发现新硬件如图 54.4.2 所示：



图 54.4.2 电脑提示找到新硬件

在硬件安装完成之后，我们在设备管理器里面可以发现多出了一个人体学输入设备，如图 54.4.3 所示：

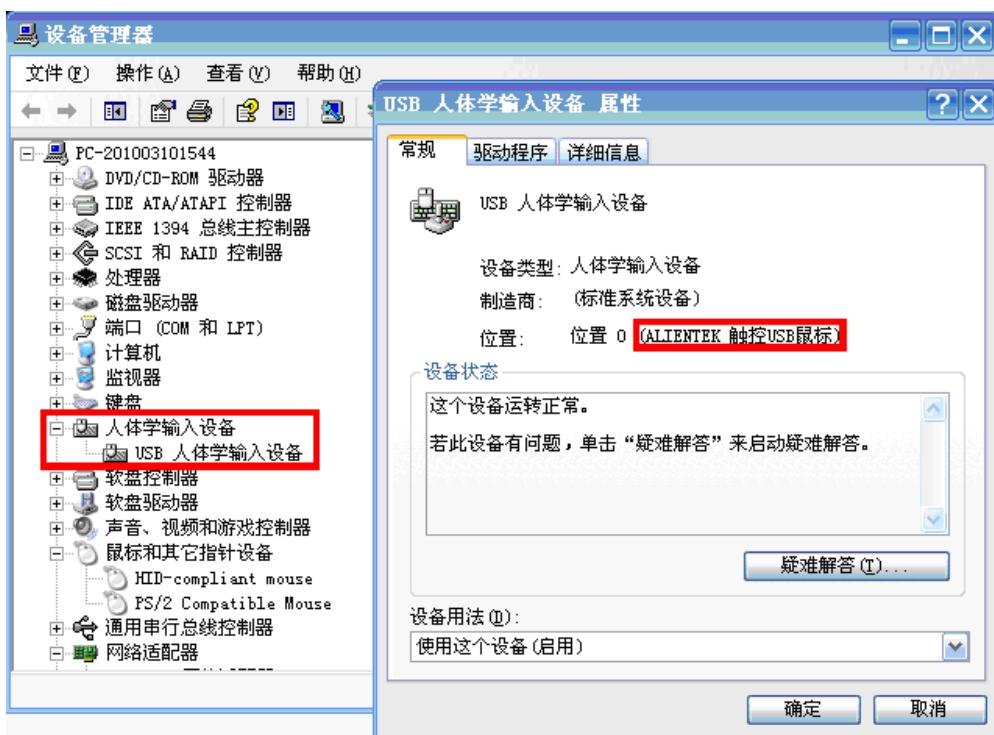


图 54.4.3 USB 人体学输入设备

此时我们按动触摸屏，就可以发现电脑屏幕上的光标随着你在触摸屏上的移动而移动了，同时可以通过按键 KEY0 和 KEY2 模拟鼠标左键和右键，通过按键 WK_UP 和 KEY1 模拟鼠标滚轮。



第五十五章 USB 读卡器实验

上一章我们向大家介绍了如何利用 STM32 的 USB 来做一个触控 USB 鼠标，本章我们将利用 STM32 的 USB 来做一个 USB 读卡器。本节分为如下几个部分：

55.1 USB 读卡器简介

55.2 硬件设计

55.3 软件设计

55.4 下载验证



55.1 USB 读卡器简介

ALIENTEK 战舰 STM32 开发板板载了一个 SD 卡插槽，可以用来接入 SD 卡，另外战舰 STM32 开发板板载了一个 8M 字节的 SPI FLASH 芯片，通过 STM32 的 USB 接口，我们可以实现一个简单的 USB 读卡器，来读写 SD 卡和 SPI FLASH。

本章我们还是通过移植官方的 USB Mass_Storage 例程来实现，该例程在 MDK 的安装目录下可以找到（..\\MDK\\ARM\\Examples\\ST\\STM32F10xUSBLib\\Demos\\Mass_Storage）。

USB Mass Storage 类支持两个传输协议：

- 1) Bulk-Only 传输 (BOT)
- 2) Control/Bulk/Interrupt 传输 (CBI)

Mass Storage 类规范定义了两个类规定的请求：Get_Max_LUN 和 Mass Storage Reset，所有的 Mass Storage 类设备都必须支持这两个请求。

Get_Max_LUN (bmRequestType= 10100001b and bRequest= 11111110b) 用来确认设备支持的逻辑单元数。Max LUN 的值必须是 0~15。注意：LUN 是从 0 开始的。主机不能向不存在的 LUN 发送 CBW，本章我们定义 Max LUN 的值为 1，即代表 2 个逻辑单元。

Mass Storage Reset (bmRequestType=00100001b and bRequest= 11111111b) 用来复位 Mass Storage 设备及其相关接口。

支持 BOT 传输的 Mass Storage 设备接口描述符要求如下：

接口类代码 bInterfaceClass=08h，表示为 Mass Storage 设备；

接口类子代码 bInterfaceSubClass=06h，表示设备支持 SCSI Primary Command-2 (SPC-2)；

协议代码 bInterfaceProtocol 有 3 种：0x00、0x01、0x50，前两种需要使用中断传输，最后一种仅使用批量传输 (BOT)。

支持 BOT 的设备必须支持最少 3 个 endpoint：Control, Bulk-In 和 Bulk-Out。USB2.0 的规范定义了控制端点 0。Bulk-In 端点用来从设备向主机传送数据（本章用端点 1 实现）。Bulk-Out 端点用来从主机向设备传送数据（本章用端点 2 实现）。

ST 官方的例程是通过 USB 来读写 SD 卡 (SDIO 方式) 和 NAND FLASH，支持 2 个逻辑单元，我们在官方例程的基础上，只需要修改 SD 驱动部分代码 (改为 SPI)，并将对 NAND FLASH 的操作修改为对 SPI FLASH 的操作。只要这两步完成了，剩下的就比较简单了，对底层磁盘的读写，都是在 mass_mal.c 文件实现的，所以我们只需要修改该函数的 MAL_Init、MAL_Write、MAL_Read 和 MAL_GetStatus 等 4 个函数，使之与我们的 SD 卡和 SPI FLASH 对应起来即可。

本章我对 SD 卡和 SPI FLASH 的操作都是采用 SPI 方式，所以速度相对 SDIO 和 FSMC 控制的 NAND FLASH 来说，相对会慢一些。

55.2 硬件设计

本节实验功能简介：开机的时候先检测 SD 卡和 SPI FLASH 是否存在，如果存在则获取其容量，并显示在 LCD 上面（如果不存在，则报错）。之后开始 USB 配置，在配置成功之后就可以在电脑上发现两个可移动磁盘。我们用 DS1 来指示 USB 正在读写 SD 卡，并在液晶上显示出来，同样我们还是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 、 DS1
- 2) 串口
- 3) TFTLCD 模块



- 4) SD 卡
- 5) SPI FLASH
- 6) USB 接口

这几个部分，在之前的实例中都已经介绍过了，我们在此就不多说了。

55.3 软件设计

打开本实验的工程文件夹目录可以看到，我们在 USB 文件夹下面新建 LIB 和 CONFIG 文件夹，分别用来存放与 USB 核相关的代码以及配置部分代码。这两部分代码我们也不细说（详见光盘本例程源码），其中 USB 文件夹里面的代码同上一章的一模一样，而 CONFIG 文件夹里面的源码则来自 MDK 自带的 Mass_Storage 例程： X:\Keil3.80A\ARM\Examples\ST\STM32F10xUSBLib\Datas\Mass_Storage 下的 source 和 include 文件夹（X 为你安装 MDK 的磁盘）。

在 main.c 里面，我们修改 main 函数如下：

```
//设置 USB 连接/断线
//enable:0,断开
//      1,允许连接
void usb_port_set(u8 enable)
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE); //使能 PORTA 时钟

    if(enable)_SetCNTR(_GetCNTR()&(~(1<<1))); //退出断电模式
    else
    {
        _SetCNTR(_GetCNTR()|(1<<1)); // 断电模式
        GPIOA->CRH&=0XFFF00FFF;
        GPIOA->CRH|=0X00033000;
        PAout(12)=0;
    }
}

int main(void)
{
    u8 offline_cnt=0;
    u8 tct=0;
    u8 USB_STA;
    u8 Device_STA;
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //LED 端口初始化
    LCD_Init();             //初始化液晶
    KEY_Init();              //按键初始化
    POINT_COLOR=RED; //设置字体为蓝色
```



```
LCD_ShowString(60,50,200,16,16,"WarShip STM32");
LCD_ShowString(60,70,200,16,16,"USB Card Reader TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/25");
SPI_Flash_Init();
if(SD_Initialize())LCD_ShowString(60,130,200,16,16,"SD Card Error!");
//检测 SD 卡错误
else //SD 卡正常
{
    LCD_ShowString(60,130,200,16,16,"SD Card Size:      MB");
    Mass_Memory_Size[0]=(long long)SD_GetSectorCount()*512;
    //得到 SD 卡容量 (字节), 当容量超过 4G 的时候, 需要用到两个 u32 来表示
    Mass_Block_Size[0]=512;
    //因为在 Init 里面设置了 SD 卡的操作字节为 512 个, 所以这里一定是 512 个字节.
    Mass_Block_Count[0]=Mass_Memory_Size[0]/Mass_Block_Size[0];
    LCD_ShowNum(164,130,Mass_Memory_Size[0]>>20,5,16); //显示 SD 卡容量
}
if(SPI_FLASH_TYPE!=W25Q64)LCD_ShowString(60,130,200,16,16,"W25Q64  Error!");
//检测 SD 卡错误
else //SPI FLASH 正常
{
    Mass_Memory_Size[1]=1024*1024*6;//前 6M 字节
    Mass_Block_Size[1]=512;
    //因为在 Init 里面设置了 SD 卡的操作字节为 512 个, 所以这里一定是 512 个字节.
    Mass_Block_Count[1]=Mass_Memory_Size[1]/Mass_Block_Size[1];
    LCD_ShowString(60,150,200,16,16,"SPI FLASH Size:6144KB");
}
delay_ms(1800); usb_port_set(0); //USB 先断开
delay_ms(300); usb_port_set(1); //USB 再次连接
LCD_ShowString(60,170,200,16,16,"USB Connecting...");//提示 SD 卡已经准备了
//USB 配置
USB_Interrupts_Config();
Set_USBClock();
USB_Init();
while(1)
{
    delay_ms(1);
    if(USB_STA!=USB_STATUS_REG)//状态改变了
    {
        LCD_Fill(60,190,240,190+16,WHITE);//清除显示
        if(USB_STATUS_REG&0x01)//正在写
        {
            LCD_ShowString(60,190,200,16,16,"USB Writing...");//USB 正在写入数据
        }
    }
}
```



```
        }
        if(USB_STATUS_REG&0x02)//正在读
        {
            LCD_ShowString(60,190,200,16,16,"USB Reading...");//USB 正在读数据
        }
        if(USB_STATUS_REG&0x04)LCD_ShowString(60,210,200,16,16,"USB Write Err");//提示写入错误
        else LCD_Fill(60,210,240,210+16,WHITE);//清除显示
        if(USB_STATUS_REG&0x08)LCD_ShowString(60,230,200,16,16,"USB Read Err");//提示读出错误
        else LCD_Fill(60,230,240,230+16,WHITE);//清除显示
        USB_STA=USB_STATUS_REG;//记录最后的状态
    }
    if(Divece_STA!=bDeviceState)
    {
        if(bDeviceState==CONFIGURED)LCD_ShowString(60,170,200,16,16,"USB Connected");//提示 USB 连接已经建立
        else LCD_ShowString(60,170,200,16,16,"USB DisConnected");//USB 被拔出了
        Divece_STA=bDeviceState;
    }
    tct++;
    if(tct==200)
    {
        tct=0; LED0=!LED0;//提示系统在运行
        if(USB_STATUS_REG&0x10)
        {
            offline_cnt=0;//USB 连接了,则清除 offline 计数器
            bDeviceState=CONFIGURED;
        }else//没有得到轮询
        {
            offline_cnt++;
            if(offline_cnt>10)bDeviceState=UNCONNECTED;
            //2s 内没收到在线标记,代表 USB 被拔出了
        }
        USB_STATUS_REG=0;
    }
};
```

此部分代码除了 main 函数，还有一个 usb_port_set 函数，usb_port_set 函数我们在上一章已经介绍过了，这里就不多说。我们将 SPI FLASH 的最开始 6M 地址范围用作 SPI FLASH Disk，也就是文件系统管理的范围大小，这个我们在之前的 SPI FLASH 也介绍过。

通过此部分代码就可以实现了我们之前在硬件设计部分描述的功能，这里我们用到了一个全局变量 Usb_Status_Reg，用来标记 USB 的相关状态，这样我们就可以在液晶上显示当前 USB



的状态了。

软件设计部分就为大家介绍到这里。

55.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上，在 USB 配置成功后（假设已经插入 SD 卡，注意：USB 数据线，要插在 USB 口！不是 USB_232 端口！），LCD 显示效果如图 55.4.1 所示：



图 55.4.1 USB 连接成功

此时，电脑提示发现新硬件如图 55.4.2 所示：



图 55.4.2 USB 读卡器被电脑找到

等 USB 配置成功后，DS1 不亮，DS0 闪烁，并且在电脑上可以看到我们的磁盘，如图 55.4.3 所示：



图 55.4.3 电脑找到 USB 读卡器的两个盘符

我们打开设备管理器，在通用串行总线控制器里面可以发现多出了一个 USB Mass Storage Device，同时看到磁盘驱动器里面多了 2 个磁盘，如图 55.4.4 所示：



图 55.4.4 通过设备管理器查看磁盘驱动器

此时，我们就可以通过电脑读写 SD 卡或者 SPI FLASH 里面的内容了。在执行读写操作的时候，就可以看到 DS1 亮，并且会在液晶上显示当前的读写状态。

注意，在对 SPI FLASH 操作的时候，最好不要频繁的往里面写数据，否则很容易将 SPI FLASH 写爆！！



第五十六章 USB 声卡实验

上一章我们向大家介绍了如何利用 STM32 的 USB 来做一个 USB 读卡器，本章我们将利用 STM32 的 USB 来做一个声卡。本节分为如下几个部分：

56.1 USB 读卡器简介

56.2 硬件设计

56.3 软件设计

56.4 下载验证



56.1 USB 声卡简介

ALIENTEK 战舰 STM32 板载了一个 PWM DAC 电路，可以用来做 DAC 输出，通过和 STM32 的 USB 和定时器配合，我们就可以实现一个 USB 声卡。

本章我们还是通过移植官方的 USB Mass_Storage 例程来实现，该例程在 MDK 的安装目录下可以找到（..\\MDK\\ARM\\Examples\\ST\\STM32F10xUSBLib\\Demos\\Audio_Speaker）。

ST 提供的例程实现了一个 22Khz 采样率、8 位的单声道 USB 声卡，通过 STM32 的 PWM 输出，经过 RC 滤波得到音频信号（对 STM3210B-EVAL 平台）。我们在该例程基础上，只需要修改定时器和其输出通道，使之满足战舰 STM32 开发板的硬件即可。

战舰 STM32 的 PWM DAC 部分，我们在第二十五章（PWM DAC 实验）已经有过详细介绍，并且知道我们的 PWM DAC 部分截止频率为 33.8Khz，而本例程的音频采样率为 22Khz，所以使用该 PWM DAC 电路来做音频输出是合适的。

本章，我们采用 TIM4 的通道 1 (PB6) 输出 PWM，因为我们的音频信号是 8 位的，所以设置 ARR 寄存器值为 0XFF，这样，我们可以得到 TIM4_CH1 的输出频率（不分频）为 $72\text{Mhz}/256=281.250\text{Khz}$ ，然后我们通过定时器 7 的中断来更新 TIM4_CH1 的输出，TIM7 的中频率就是我们音频信号的采样率：22Khz。这样，我们就可以使 PWM DAC 输出 22Khz，8 位音频信号了。

56.2 硬件设计

本节实验功能简介：开机的时候先显示一些提示信息，之后开始 **USB** 配置，在配置成功之后就可以在电脑上发现多出一个 **USB** 声卡。我们用 **DS1** 来指示 **USB** 是否连接成功，并在液晶上显示 **USB** 连接状况，如果成功连接，我们可以将耳机插入开发板的 **PHONE** 端口，听到来自电脑的音频信号。同样我们还是用 **DS0** 来指示程序正在运行。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 、 DS1
- 2) 串口
- 3) TFTLCD 模块
- 4) USB 接口
- 5) PWM DAC
- 6) 74HC4052
- 7) TDA1308

这几个部分，在之前的实例中都已经介绍过了，我们在此就不多说了。

56.3 软件设计

打开本实验工程目录可以看见，跟上一个实验一样，我们在 **HARDWARE** 文件夹所在文件夹下新建一个 **USB** 的文件夹，然后在 **USB** 文件夹下面新建 **LIB** 和 **CONFIG** 文件夹，分别用来存放与 **USB** 核相关的代码以及配置部分代码。这两部分代码我们也不细说（详见光盘本例程源码），其中 **USB** 文件夹里面的代码同上一章的一模一样，而 **CONFIG** 文件夹里面的源码则来自 MDK 自带的 **Audio_Speaker** 例程：**X:\\Keil3.80A\\ARM\\Examples\\ST\\STM32F10xUSBLib\\Demos\\Audio_Speaker** 下的 **source** 和 **include** 文件夹（**X** 为你安装 MDK 的磁盘）。



本章，我们还需要通过音频选择电路和耳机驱动电路来推动耳机输出，所以还需要加入音频选择部分的驱动，我们将第四十章实验（实验 35）的 AUDIOSEL 文件夹拷贝到本工程的 HARDWARE 文件夹下，将 audiosel.c 加入 HARDWARE 组下，并将 AUDIOSEL 文件夹加入头文件包含路径。

最后看看 main.c，我们修改之后的 main 函数如下：

```
//设置 USB 连接/断线
//enable:0,断开
//      1,允许连接
void usb_port_set(u8 enable)
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);//使能 PORTA 时钟

    if(enable)_SetCNTR(_GetCNTR()&(~(1<<1))); //退出断电模式
    else
    {
        _SetCNTR(_GetCNTR()|(1<<1)); // 断电模式
        GPIOA->CRH&=0XFFF00FFF;
        GPIOA->CRH|=0X00033000;
        PAout(12)=0;
    }
}

int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);       //串口初始化波特率为 9600
    LED_Init();            //LED 端口初始化
    KEY_Init();             //KEY 初始化
    LCD_Init();             //初始化液晶
    Audiosel_Init();        //初始化声道选择
    POINT_COLOR=RED;//设置字体为蓝色
    LCD_ShowString(60,50,200,16,16,"WarShip STM32");
    LCD_ShowString(60,70,200,16,16,"USB Sound Card TEST");
    LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,200,16,16,"2012/9/25");
    LCD_ShowString(60,130,200,16,16,"USB Connecting...");//提示 SD 卡已经准备了
    delay_ms(1800);
    usb_port_set(0);//USB 先断开一下
    delay_ms(300);
    usb_port_set(1);//USB 再次连接
    //USB 配置
    USB Interrupts_Config();
    Set_USBClock();
```



```
USB_Init();
Speaker_Config();
Audiosel_Set(2); //设置到 PWM 音频通道
LCD_ShowString(60,130,200,16,16,"USB Connecting...");//提示 SD 卡已经准备了
while(1)
{
    if(bDeviceState==CONFIGURED)//USB 连接上了?
    {
        LED1=0;
        LCD_ShowString(60,130,200,16,16,"USB Connected      ");//SD 卡已经准备了
    }else
    {
        LED1=1;
        LCD_ShowString(60,130,200,16,16,"USB DisConnected ");//SD 卡连接失败
    }
    LED0=!LED0; delay_ms(200);
}
}
```

该部分代码同样有 `usb_port_set` 函数，这里我们就不介绍该函数了。在 `main` 函数里面，我们通过调用 `Speaker_Config` 函数，配置 `TIM4_CH1` 为 281.25Khz 的 PWM 输出，配置 `TIM7` 为 22Khz 的定时中断，其他部分我们就不详细介绍了。

软件设计部分就为大家介绍到这里。

56.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上，在 USB 配置成功后（注意：USB 数据线，要插在 USB 口！不是 USB_232 端口！），LCD 显示效果如图 56.4.1 所示：



图 56.4.1 USB 连接成功



此时，电脑提示发现新硬件如图 56.4.2 所示：



图 56.4.2 USB 读卡器被电脑找到

等 USB 配置成功后，DS1 常亮，DS0 闪烁，并且在设备管理器→声音、视频和游戏控制器里面看到多了 USB Audio Device，如图 56.4.3 所示：



图 56.4.3 USB Audio Device

此时，电脑的所有音频输出都被切换到 USB 声卡输出，将耳机插入战舰 STM32 开发板的 PHONE 端口，即可听到来自电脑的声音。

第五十七章 ENC28J60 网络实验

本章，我们将向大家介绍 ALIENTEK ENC28J60 网络模块及其使用。本章，我们将使用 ALIENTEK ENC28J60 网络模块和 uIP 1.0 实现：TCP 服务器、TCP 客服端以及 WEB 服务器等三个功能。本章分为如下几个部分：

57.1 ENC28J60 以及 uIP 简介

57.2 硬件设计

57.3 软件设计

57.4 下载验证



57.1 ENC28J60 以及 uIP 简介

本章我们需要用到 ENC28J60 以太网控制器和 uIP 1.0 以太网协议栈。接下来分别介绍这两个部分。

57.1.1 ENC28J60 简介

ENC28J60 是带有行业标准串行外设接口（Serial Peripheral Interface，SPI）的独立以太网控制器。它可作为任何配备有 SPI 的控制器的以太网接口。ENC28J60 符合 IEEE 802.3 的全部规范，采用了一系列包过滤机制以对传入数据包进行限制。它还提供了一个内部 DMA 模块，以实现快速数据吞吐和硬件支持的 IP 校验和计算。与主控制器的通信通过两个中断引脚和 SPI 实现，数据传输速率高达 10 Mb/s。两个专用的引脚用于连接 LED，进行网络活动状态指示。ENC28J60 总共只有 28 脚，提供 QFN/TF

ENC28J60 的主要特点如下：

- 兼容 IEEE802.3 协议的以太网控制器
- 集成 MAC 和 10 BASE-T 物理层
- 支持全双工和半双工模式
- 数据冲突时可编程自动重发
- SPI 接口速度可达 10Mbps
- 8K 数据接收和发送双端口 RAM
- 提供快速数据移动的内部 DMA 控制器
- 可配置的接收和发送缓冲区大小
- 两个可编程 LED 输出
- 带 7 个中断源的两个中断引脚
- TTL 电平输入
- 提供多种封装：SOIC/SSOP/SPDIP/QFN 等

ENC28J60 的典型应用电路如图 57.1.1.1 所示：

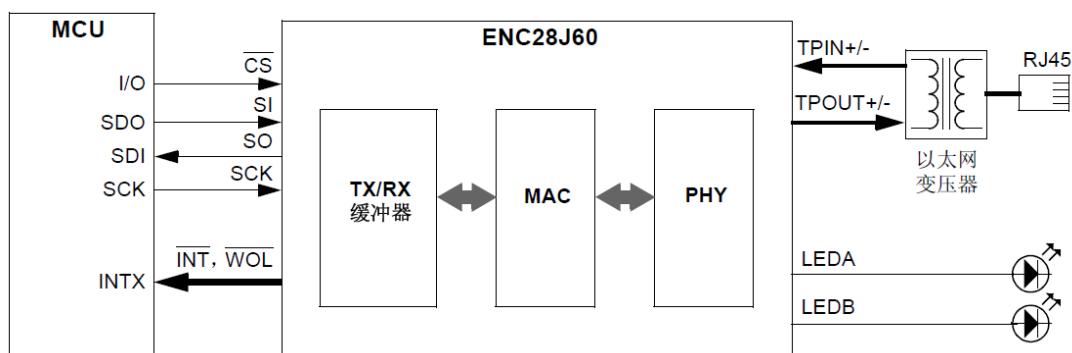


图 57.1.1.1 ENC28J60 典型应用电路

ENC28J60 由七个主要功能模块组成：

- 1) SPI 接口，充当主控制器和 ENC28J60 之间通信通道。
- 2) 控制寄存器，用于控制和监视 ENC28J60。
- 3) 双端口 RAM 缓冲器，用于接收和发送数据包。
- 4) 判优器，当 DMA、发送和接收模块发出请求时对 RAM 缓冲器的访问进行控制。
- 5) 总线接口，对通过 SPI 接收的数据和命令进行解析。
- 6) MAC(Medium Access Control)模块，实现符合 IEEE 802.3 标准的 MAC 逻辑。



7) PHY(物理层)模块, 对双绞线上的模拟数据进行编码和译码。

ENC28J60 还包括其他支持模块, 诸如振荡器、片内稳压器、电平变换器 (提供可以接受 5V 电压的 I/O 引脚) 和系统控制逻辑。

ENC28J60 的功能框图如图 57.1.1.2 所示:

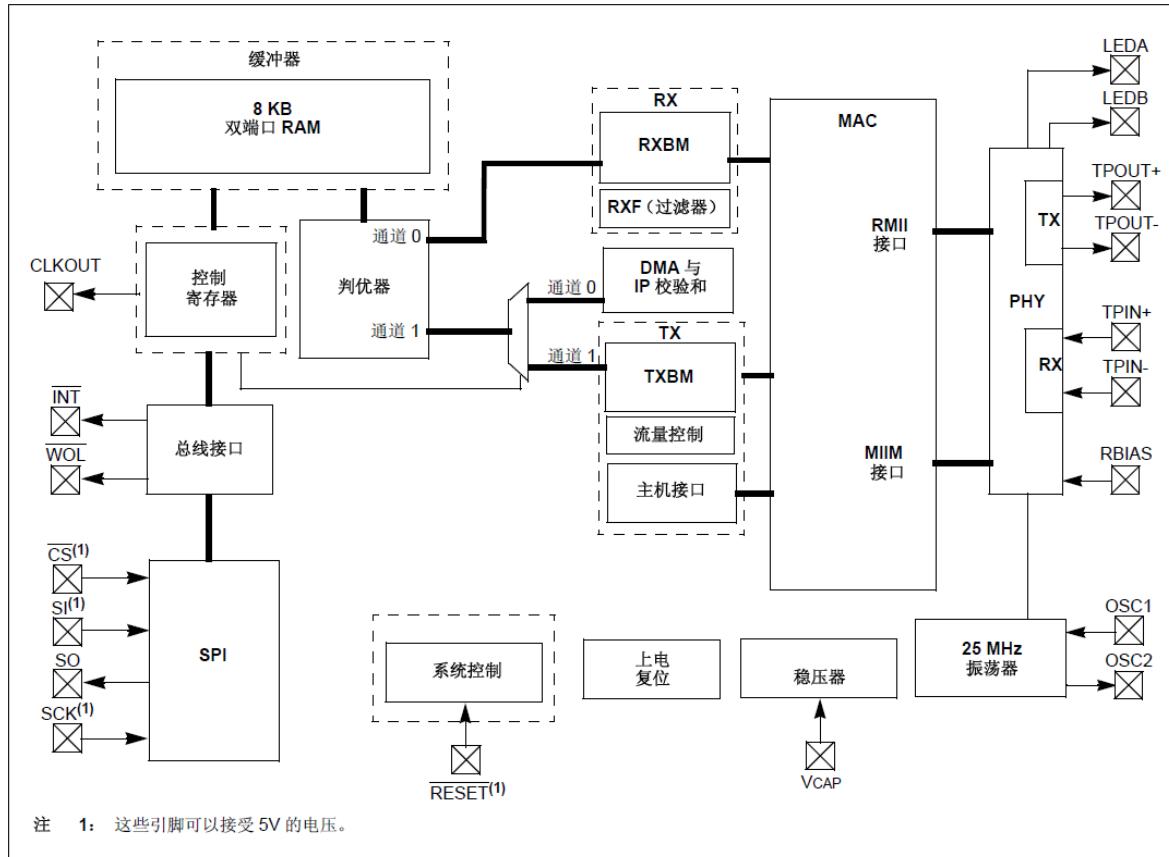


图 57.1.1.2 ENC28J60 功能框图

ALIENTEK ENC28J60 网络模块采用 ENC28J60 作为主芯片, 单芯片即可实现以太网接入, 利用该模块, 基本上只要是个单片机, 就可以实现以太网连接。ALIENTEK ENC28J60 网络模块原理图如图 57.1.1.3 所示:

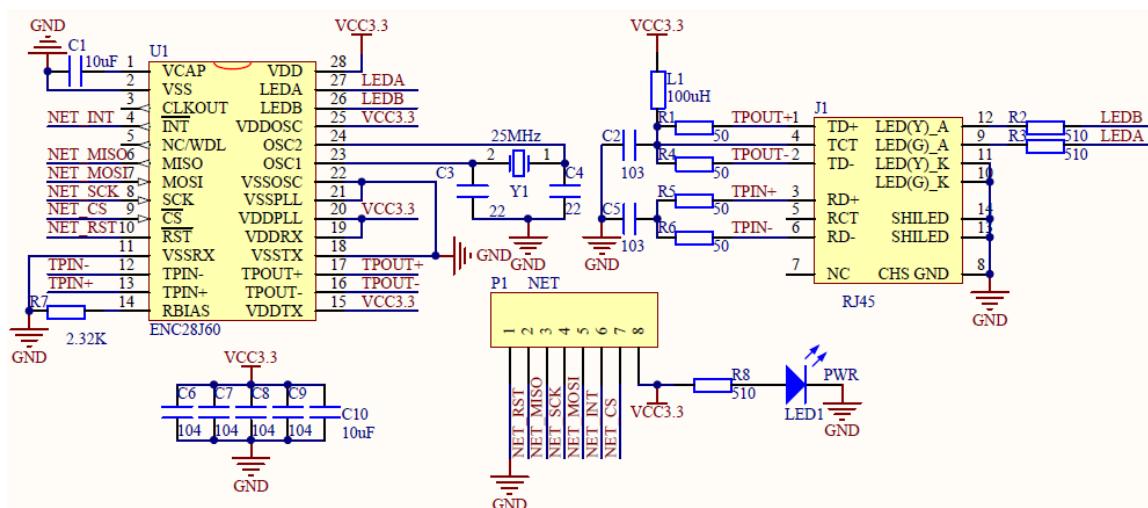


图 57.1.1.3 ALIENTEK ENC28J60 网络模块原理图



ALIENTEK ENC28J60 网络模块外观图如图 57.1.1.4 所示：

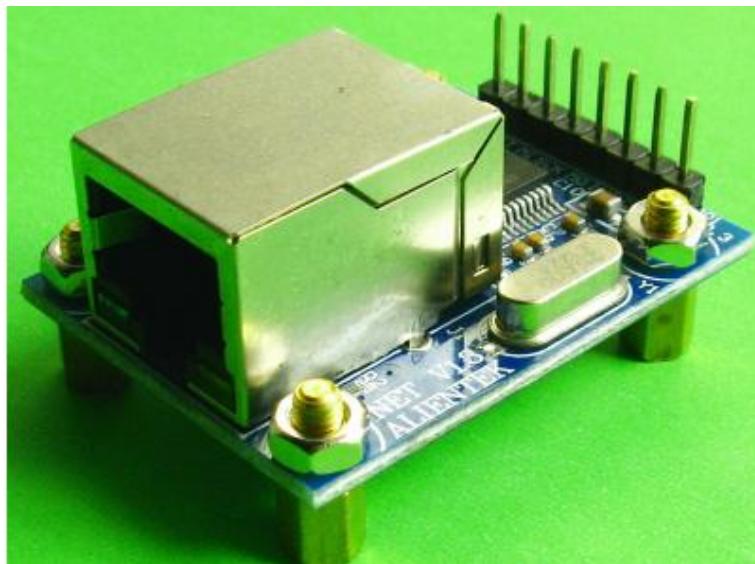


图 57.1.1.4 ALIENTEK ENC28J60 网络模块外观图

该模块通过一个 8 个引脚的排针与外部电路连接，这 8 个引脚分别是：GND、RST、MISO、SCK、MOSI、INT、CS 和 V3.3。其中 GND 和 V3.3 用于给模块供电，MISO/MOSI/SCK 用于 SPI 通信，CS 是片选信号，INT 为中断输出引脚，RST 为模块复位信号。

57.1.2 uIP 简介

uIP 由瑞典计算机科学学院(网络嵌入式系统小组)的 Adam Dunkels 开发。其源代码由 C 语言编写，并完全公开，uIP 的最新版本是 1.0 版本，本指南移植和使用的版本正是此版本。

uIP 协议栈去掉了完整的 TCP/IP 中不常用的功能，简化了通讯流程，但保留了网络通信必须使用的协议，设计重点放在了 IP/TCP/ICMP/UDP/ARP 这些网络层和传输层协议上，保证了其代码的通用性和结构的稳定性。

由于 uIP 协议栈专门为嵌入式系统而设计，因此还具有如下优越功能：

- 1) 代码非常少，其协议栈代码不到 6K，很方便阅读和移植。
- 2) 占用的内存数非常少，RAM 占用仅几百字节。
- 3) 其硬件处理层、协议栈层和应用层共用一个全局缓存区，不存在数据的拷贝，且发送和接收都是依靠这个缓存区，极大的节省空间和时间。
- 4) 支持多个主动连接和被动连接并发。
- 5) 其源代码中提供一套实例程序：web 服务器，web 客户端，电子邮件发送程序(SMTP 客户端)，Telnet 服务器，DNS 主机名解析程序等。通用性强，移植起来基本不用修改就可以通过。
- 6) 对数据的处理采用轮循机制，不需要操作系统的支持。

由于 uIP 对资源的需求少和移植容易，大部分的 8 位微控制器都使用过 uIP 协议栈，而且很多的著名的嵌入式产品和项目(如卫星，Cisco 路由器，无线传感器网络)中都在使用 uIP 协议栈。

uIP 相当于一个代码库，通过一系列的函数实现与底层硬件和高层应用程序的通讯，对于整个系统来说它内部的协议组是透明的，从而增加了协议的通用性。uIP 协议栈与系统底层和高层应用之间的关系如图 57.1.2.1 所示：

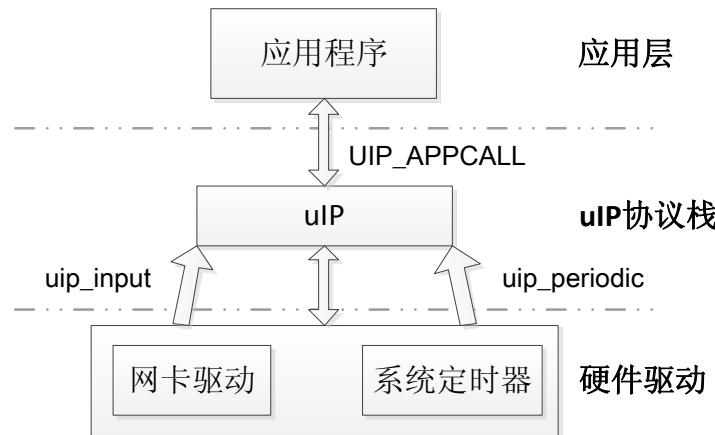


图 57.1.2.1 uIP 在系统中的位置

从上图可以看出, uIP 协议栈主要提供 2 个函数供系统底层调用: `uip_input` 和 `uip_periodic`。另外和应用程序联系主要是通过 `UIP_APPCALL` 函数。

当网卡驱动收到一个输入包时, 将放入全局缓冲区 `uip_buf` 中, 包的大小由全局变量 `uip_len` 约束。同时将调用 `uip_input()` 函数, 这个函数将会根据包首部的协议处理这个包和需要时调用应用程序。当 `uip_input()` 返回时, 一个输出包同样放在全局缓冲区 `uip_buf` 里, 大小赋给 `uip_len`。如果 `uip_len` 是 0, 则说明没有包要发送。否则调用底层系统的发包函数将包发送到网络上。

uIP 周期计时是用于驱动所有的 uIP 内部时钟事件。当周期计时激发, 每一个 TCP 连接都会调用 uIP 函数 `uip_periodic()`。类似于 `uip_input()` 函数。`uip_periodic()` 函数返回时, 输出的 IP 包要放到 `uip_buf` 中, 供底层系统查询 `uip_len` 的大小发送。

由于使用 TCP/IP 的应用场景很多, 因此应用程序作为单独的模块由用户实现。uIP 协议栈提供一系列接口函数供用户程序调用, 其中大部分函数是作为 C 的宏命令实现的, 主要是为了速度、代码大小、效率和堆栈的使用。用户需要将应用层入口程序作为接口提供给 uIP 协议栈, 并将这个函数定义为宏 `UIP_APPCALL()`。这样, uIP 在接受到底层传来的数据包后, 在需要送到上层应用程序处理的地方, 调用 `UIP_APPCALL()`。在不用修改协议栈的情况下可以适配不同的应用程序。

uIP 协议栈提供了我们很多接口函数, 这些函数在 `uip.h` 中定义, 为了减少函数调用造成的额外支出, 大部分接口函数以宏命令实现的, uIP 提供的接口函数有:

1. 初始化 uIP 协议栈: `uip_init()`
2. 处理输入包: `uip_input()`
3. 处理周期计时事件: `uip_periodic()`
4. 开始监听端口: `uip_listen()`
5. 连接到远程主机: `uip_connect()`
6. 接收到连接请求: `uip_connected()`
7. 主动关闭连接: `uip_close()`
8. 连接被关闭: `uip_closed()`
9. 发出去的数据被应答: `uip_acked()`
10. 在当前连接发送数据: `uip_send()`
11. 在当前连接上收到新的数据: `uip_newdata()`
12. 告诉对方要停止连接: `uip_stop()`
13. 连接被意外终止: `uip_aborted()`

接下来, 我们看看 uIP 的移植过程。首先, uIP1.0 的源码包里面有如下内容, 如图 57.1.2.2



所示：

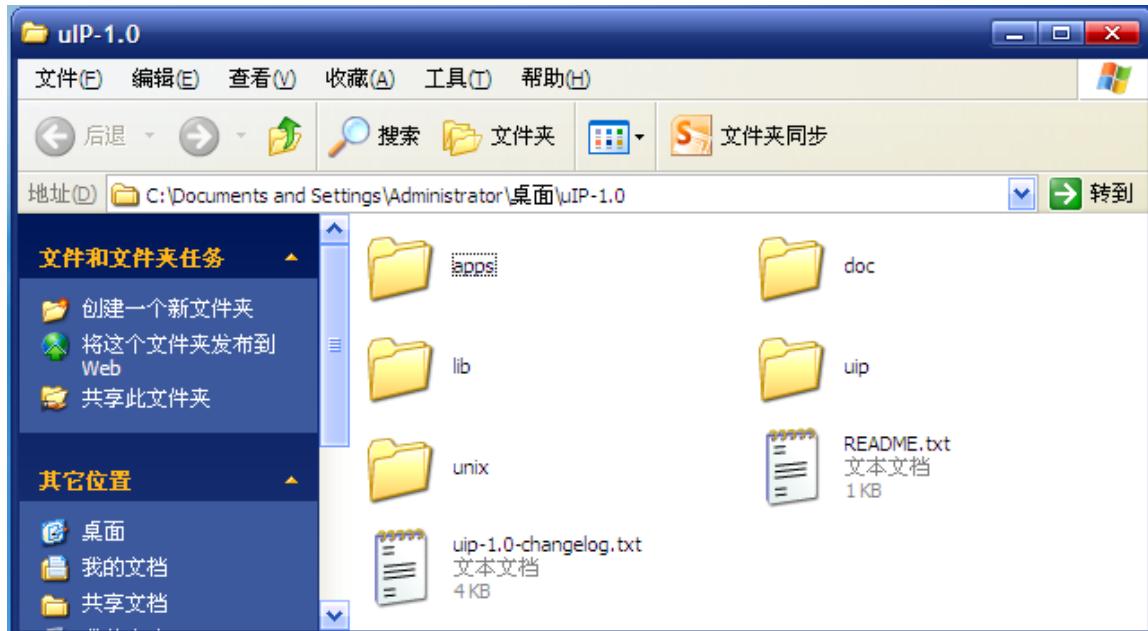


图 57.1.2.2 uIP 1.0 源码包内容

其中 apps 文件夹里面是 uip 提供的各种参考代码，本章我们主要有用到里面的 webserver 部分。doc 文件夹里面是一些 uip 的使用及说明文件，是学习 uip 的官方资料。lib 文件夹里面是用于内存管理的一个代码，本章我们没有用到。uip 里面就是 uip 1.0 的源码了，我们全盘照收。unix 里面提供的是具体的应用实例，我们移植参考主要是依照这个里面的代码。

移植第一步：实现在 unix/tapdev.c 里面的三个函数。首先是 tapdev_init 函数，该函数用于初始化网卡（也就是我们的 ENC28J60），通过这个函数实现网卡初始化。其次，是 tapdev_read 函数，该函数用于从网卡读取一包数据，将读到的数据存放在 uip_buf 里面，数据长度返回给 uip_len。最后，是 tapdev_send 函数，该函数用于向网卡发送一包数据，将全局缓存区 uip_buf 里面的数据发送出去（长度为 uip_len）。其实这三个函数就是实现最底层的网卡操作。

第二步，因为 uIP 协议栈需要使用时钟，为 TCP 和 ARP 的定时器服务，因此我们需要 STM32 提供一个定时器做时钟，提供 10ms 计时（假设 clock-arch.h 里面的 CLOCK_CONF_SECOND 为 100），通过 clock-arch.c 里面的 clock_time 函数返回给 uIP 使用。

第三步，配置 uip-conf.h 里面的宏定义选项。主要用于设置 TCP 最大连接数、TCP 监听端口数、CPU 大小端模式等，这个大家根据自己需要配置即可。

通过以上 3 步的修改，我们基本上就完成了 uIP 的移植。在使用 uIP 的时候，一般通过如下顺序：

1) 实现接口函数（回调函数）UIP_APPCALL。

该函数是我们使用 uIP 最关键的部分，它是 uIP 和应用程序的接口，我们必须根据自己的需要，在该函数做各种处理，而做这些处理的触发条件，就是前面提到的 uIP 提供的那些接口函数，如 uip_newdata、uip_acked、uip_closed 等等。另外，如果是 UDP，那么还需要实现 UIP_UDP_APPCALL 回调函数。

2) 调用 tapdev_init 函数，先初始化网卡。

此步先初始化网卡，配置 MAC 地址，为 uIP 和网络通信做好准备。

3) 调用 uip_init 函数，初始化 uIP 协议栈。

此步主要用于 uip 自身的初始化，我们直接调用就是。



4) 设置 IP 地址、网关以及掩码

这个和电脑上网差不多，只不过我们这里是通过 `uip_ipaddr`、`uip_sethostaddr`、`uip_setdraddr` 和 `uip_setnetmask` 等函数实现。

5) 设置监听端口

uIP 根据你设定的不同监听端口，实现不同的服务，比如我们实现 Web Server 就监听 80 端口(浏览器默认的端口是 80 端口)，凡是发现 80 端口的数据，都通过 Web Server 的 APPCALL 函数处理。根据自己的需要设置不同的监听端口。不过 uIP 有本地端口 (`lport`) 和远程端口 (`rport`) 之分，如果是做服务端，我们通过监听本地端口 (`lport`) 实现；如果是做客户端，则需要去连接远程端口 (`rport`) 。

6) 处理 uIP 事件

最后，uIP 通过 `uip_polling` 函数轮询处理 uIP 事件。该函数必须插入到用户的主循环里面（也就是必须每隔一定时间调用一次）。

57.2 硬件设计

本节实验功能简介：开机检测 ENC28J60，如果检测不成功，则提示报错。在成功检测到 ENC28J60 之后，初始化 uIP，并设置 IP 地址（192.168.1.16）等，然后监听 80 端口和 1200 端口，并尝试连接远程 1400 端口，80 端口用于实现 WEB Server 功能，1200 端口用于实现 TCP Server 功能，连接 1400 端口实现 TCP Client 功能。此时，我们在电脑浏览器输入 <http://192.168.1.16>，就可以登录到一个界面，该界面可以控制开发板上两个 LED 灯的亮灭，还会显示开发板的当前时间以及开发板 STM32 芯片的温度（每 10 秒自动刷新一次）。另外，我们通过网络调试软件（做 TCP Server 时，设置 IP 地址为：192.168.1.103，端口为 1400；做 TCP Client 时，设置 IP 地址为：192.168.1.16，端口为 1200）同开发板连接，即可实现开发板与网络调试软件之间的数据互发。按 KEY0，由开发板的 TCP Server 端发送数据到电脑的 TCP Client 端。按 KEY2，则由开发板的 TCP Client 端发送数据到电脑的 TCP Server 端。LCD 显示当前连接状态。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 、 DS1
- 2) KEY0/KEY2 两个按键
- 3) 串口
- 4) TFTLCD 模块
- 5) ENC28J60 网络模块

前面 4 部分都已经详细介绍过，本章，我们重点看看 ALIENTEK ENC28J60 网络模块同 ALIENTEK 战舰 STM32 开发板的连接，前面我们介绍了 ALIENTEK ENC28J60 网络模块的接口，我们通过杜邦线（或排线）连接网络模块和开发板的 P12 端口，连接关系如表 56.2.1 所示：

编号	1	2	3	4	5	6	7	8
网络模块(P1端子)	GND	NET_RST	NET_MISO	NET_SCK	NET_MOSI	NET_INT	NET_CS	VCC3.3
开发板 (P12端子)	GND	PG6	PB14	PB13	PB15	PD2	PG8	VCC3.3

表 56.2.1 ENC28J60 网络模块同战舰 STM32 开发板连接关系表

上表可以看出，其实网络模块同战舰 STM32 开发板的线序是一一对应的，所以如果你有一个 1*8 的排线，就可以直接对插即可。这里需要注意，本来开发板的 P12 端口是用来连接 SD 卡，实现 SPI 读写 SD 卡的，如果要连接网络模块，我们需要把跳线帽连接到 P10 和 P11，这



样还是可以通过 SDIO 访问 SD 卡。

在开发板连接网络模块以后，我们还需要一根网线（自备），连接网络模块和路由器，这样我们才能实现和电脑的连接。

57.3 软件设计

打开本实验工程可以看到，我们在该工程源码下面加入 uIP-1.0 文件夹，存放 uIP1.0 源码，再新建 uIP-APP 文件夹，存放应用部分代码，因为 uIP 自己有一个 timer.c 和 timer.h 的文件，所以我们还需要修改 HARDWARE 里面的 timer.c 和 timer.h 为不同的名字，本章我们改为 timerx.c 和 timerx.h，我们还需要实现 ENC28J60 的驱动代码，存放在 HARDWARE 文件夹下的 ENC28J60 文件夹里面。详细的步骤我们就不一一阐述了，全部改好之后，工程如图 57.3.1 所示：

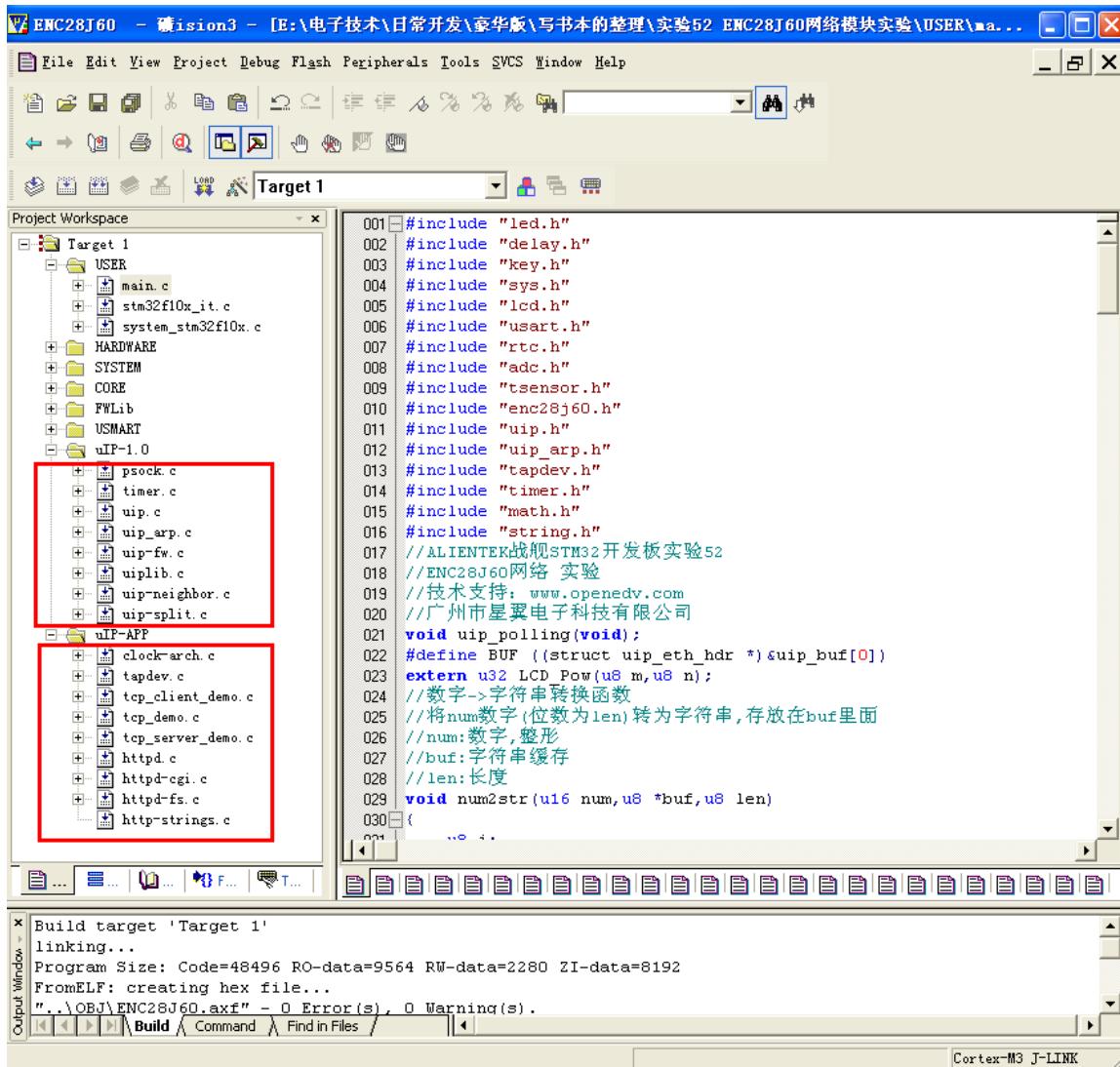


图 57.3.1 移植完后，MDK 工程图

图中 uIP-1.0 文件夹里面的代码全部是 uIP 提供的协议栈源码，而 uIP-APP 里面的代码则部分是我们自己实现的，部分是 uIP 提供的，其中：

clock-arch.c，属于 uIP 协议栈，uIP 通过该代码里面的 clock_time 函数获取时钟节拍。

tapdev.c，同样是 uIP 提供，用来实现 uIP 与网卡的接口，该文件实现 tapdev_init、tapdev_read



和 tapdev_send 三个重要函数。

tcp_demo.c，完成 UIP_APPCALL 函数的实现，即 tcp_demo_appcall 函数。该函数根据端口的不同，分别调用不同的 appcall 函数，实现不同功能。同时该文件还实现了 uip_log 函数，用于打印日志。

tcp_client_demo.c，完成一个简单的 TCP 客户端应用，实现与电脑 TCP 服务端的数据收发。

tcp_server_demo.c，完成一个简单的 TCP 服务端应用，实现与电脑 TCP 客户端的数据收发。

httpd.c、httpd-cgi.c、httpd-fs.c 和 httpd-strings.h，属于 uIP 提供的 WEB 服务器参考代码，我们通过修改部分代码，实现一个简单的 WEB 服务器。

本章代码很多，我们仅挑一些重点和大家介绍。

首先是 enc28j60.c 文件，这个里面存放的是 enc28j60 相关的驱动代码。我们通过 spi2 驱动 enc28j60。所以首先是 enc28j60 初始化函数 ENC28J60_Init()，初始化完成之后，其他的代码就是通过 spi 接口控制 enc28j60 进行相关的操作，这些操作的方法以及指令在 enc28j60 的数据手册可以找到，这里我们不做过多讲解。

Enc28j60 的底层代码写好之后，接着就是接口层封装了。这个是 tapdev.c 里面完成的。打开 tapdev.c 文件可以看到里面有三个函数，这三个函数向下负责与 enc28j60 的接口打交道，向上暴露给 uip 或者用户函数直接调用。代码如下：

```
//MAC 地址,必须唯一
//如果你有两个战舰开发板,想连入路由器,则需要修改 MAC 地址不一样!
const u8 mymac[6]={0x04,0x02,0x35,0x00,0x00,0x01}; //MAC 地址
//配置网卡硬件, 并设置 MAC 地址
//返回值: 0, 正常; 1, 失败;
u8 tapdev_init(void)
{
    u8 i,res=0;
    res=ENC28J60_Init((u8*)mymac); //初始化 ENC28J60
    //把 IP 地址和 MAC 地址写入缓存区
    for (i = 0; i < 6; i++)uip_ethaddr.addr[i]=mymac[i];
    //指示灯状态:0x476 is PHLCON LEDA(绿)=links status, LEDB(红)=receive/transmit
    //PHLCON: PHY 模块 LED 控制寄存器
    ENC28J60_PHY_Write(PHLCON,0x0476);
    return res;
}
//读取一包数据
uint16_t tapdev_read(void)
{
    return  ENC28J60_Packet_Receive(MAX_FRAMELEN,uip_buf);
}
//发送一包数据
void tapdev_send(void)
{
    ENC28J60_Packet_Send(uip_len,uip_buf);
}
```

tapdev_init 函数，该函数用于初始化网卡，即初始化我们的 ENC28J60，初始化工作主要通



通过调用 ENC28J60_Init 函数实现，该函数在 enc28j60.c 里面实现，同时该函数还用于设置 MAC 地址，这里请确保 MAC 地址的唯一性。在初始化 enc28j60 以后，我们设置 enc28j60 的 LED 控制器工作方式，即完成对 ENC28J60 的全部初始化工作。该函数的返回值用于判断网卡初始化是否成功。

tapdev_read 函数，该函数调用 ENC28J60_Packet_Receive 函数，实现从网卡（ENC28J60）读取一包数据，数据被存放在 uip_buf 里面，同时返回读到的包长度（包长度一般是存放在 uip_len 里面的）。

tapdev_send 函数，该函数调用 ENC28J60_Packet_Send 函数，实现从网卡（ENC28J60）发送一包数据到网络，数据内容存放在 uip_buf，数据长度为 uip_len。

再来看看 tcp_demo.c 里面的 tcp_demo_appcall 函数，该函数代码如下：

```
//TCP 应用接口函数(UIP_APPCALL)
//完成 TCP 服务(包括 server 和 client)和 HTTP 服务
void tcp_demo_appcall(void)
{
    switch(uip_conn->lport)//本地监听端口 80 和 1200
    {
        case HTONS(80):
            httpd_appcall();
            break;
        case HTONS(1200):
            tcp_server_demo_appcall();
            break;
        default: break;
    }
    switch(uip_conn->rport)//远程连接 1400 端口
    {
        case HTONS(1400):
            tcp_client_demo_appcall();
            break;
        default: break;
    }
}
```

该函数即 UIP_APPCALL 函数，是 uIP 同应用程序的接口函数，该函数通过端口号选择不同的 appcall 函数，实现不同的服务。其中 80 端口用于实现 WEB 服务，通过调用 httpd_appcall 实现；1200 端口用于实现 TCP 服务器，通过调用 tcp_server_demo_appcall 函数实现；1400 是远程端口，用于实现 TCP 客户端，调用 tcp_client_demo_appcall 函数实现。

接着，我们来看看这 3 个 appcall 函数，首先是 WEB 服务器的 appcall 函数：httpd_appcall，该函数在 httpd.c 里面实现，源码如下：

```
//http 服务（WEB）处理
void httpd_appcall(void)
{
    struct httpd_state *s = (struct httpd_state *)&(uipl_conn->appstate); //读取连接状态
    if(uipl_closed() || uipl_aborted() || uipl_timedout()) //异常处理（这里无任何处理）
}
```



```

else if(uip_connected())//连接成功
{
    PSOCK_INIT(&s->sin, s->inputbuf, sizeof(s->inputbuf) - 1);
    PSOCK_INIT(&s->sout, s->inputbuf, sizeof(s->inputbuf) - 1);
    PT_INIT(&s->outputpt);
    s->state = STATE_WAITING;
    /* timer_set(&s->timer, CLOCK_SECOND * 100); */
    s->timer = 0;
    handle_connection(s);//处理
}else if(s!=NULL)
{
    if(uip_poll())
    {
        ++s->timer;
        if(s->timer >= 20)uip_abort();
        else s->timer = 0;
    }
    handle_connection(s);
}else uip_abort();//
}

```

该函数在连接建立的时候，通 handle_connection 函数处理 http 数据，handle_connection 函数代码如下：

```

//分析 http 数据
static void handle_connection(struct httpd_state *s)
{
    handle_input(s); //处理 http 输入数据
    if(s->state==STATE_OUTPUT)handle_output(s);//输出状态，处理输出数据
}

```

该函数调用 handle_input 处理 http 输入数据，通过调用 handle_output 实现 http 网页输出。对我们来说最重要的是 handle_input 函数，handle_input 函数代码如下：

```

extern unsigned char data_index_html[];//在 httpd-fsdata.c 里面定义,用于存放 html 网页源代码
extern void get_temperature(u8 *temp);//在 main 函数实现,用于获取温度字符串
extern void get_time(u8 *time); //在 main 函数实现,用于获取时间字符串
const u8 * LED0_ON_PIC_ADDR="http://www.openedv.com/upload/2012/9/27/ad65ee9f478ca
11241933beed5b5dbcc_971.gif"; //LED0 亮,图标地址
const u8 * LED1_ON_PIC_ADDR="http://www.openedv.com/upload/2012/9/27/bab5bef0379dc
50129202157c2739c57_775.gif"; //LED1 亮,图标地址
const u8 * LED_OFF_PIC_ADDR="http://www.openedv.com/upload/2012/9/27/ccecf4ebef84b
095545b8feb0cecc671_254.gif"; //LED 灭,图标地址
//处理 HTTP 输入数据
static PT_THREAD(handle_input(struct httpd_state *s))
{
    char *strx;

```



```
u8 dbuf[17];
PSOCK_BEGIN(&s->sin);
PSOCK_READTO(&s->sin, ISO_space);
if(strncmp(s->inputbuf, http_get, 4)!=0)PSOCK_CLOSE_EXIT(&s->sin); //比较客户端
//浏览器输入的指令是否是申请 WEB 指令 “GET ”
PSOCK_READTO(&s->sin, ISO_space); //"
if(s->inputbuf[0] != ISO_slash)PSOCK_CLOSE_EXIT(&s->sin); //判断第一个数据
//(去掉 IP 地址之后),是否是"/"
if(s->inputbuf[1] == ISO_space||s->inputbuf[1] == '?') //第二个数据是空格/问号
{
    if(s->inputbuf[1]=='?'&&s->inputbuf[6]==0x31)//LED1
    {
        LED0=!LED0;
        strx=strstr((const char*)(data_index_html+13),"LED0 状态");
        if(strx)//存在"LED0 状态"这个字符串
        {
            strx=strstr((const char*)strx,"color:#");//找到"color:#"字符串
            if(LED0)//LED0 灭
            {
                strncpy(strx+7,"5B5B5B",6); //灰色
                strncpy(strx+24,"灭",2); //灭
                strx=strstr((const char*)strx,"http://"); //找到"http:"字符串
                strncpy(strx,(const char*)LED_OFF_PIC_ADDR,strlen((const char*)
                LED_OFF_PIC_ADDR)); //LED0 灭图片
            }
            else
            {
                strncpy(strx+7,"FF0000",6); //红色
                strncpy(strx+24,"亮",2); //亮
                strx=strstr((const char*)strx,"http://"); //找到"http:"字符串
                strncpy(strx,(const char*)LED0_ON_PIC_ADDR,strlen((const char*)
                LED0_ON_PIC_ADDR)); //LED0 亮图片
            }
        }
    }
}else if(s->inputbuf[1]=='?'&&s->inputbuf[6]==0x32)//LED2
{
    LED1=!LED1;
    strx=strstr((const char*)(data_index_html+13),"LED1 状态");
    if(strx)//存在"LED1 状态"这个字符串
    {
        strx=strstr((const char*)strx,"color:#");//找到"color:#"字符串
        if(LED1)//LED1 灭
        {
            strncpy(strx+7,"5B5B5B",6); //灰色
```



```
strncpy(strx+24,"灭",2); //灭
strx=strstr((const char*)strx,"http:");//找到"http:"字符串
strncpy(strx,(const char*)LED_OFF_PIC_ADDR,strlen((const char*)
LED_OFF_PIC_ADDR));//LED1 灭图片
}else
{
    strncpy(strx+7,"00FF00",6); //绿色
    strncpy(strx+24,"亮",2); //亮
    strx=strstr((const char*)strx,"http:");//找到"http:"字符串
    strncpy(strx,(const char*)LED1_ON_PIC_ADDR,strlen((const char*)
LED1_ON_PIC_ADDR));//LED1 亮图片
}
}
}

strx=strstr((const char*)(data_index_html+13),"°C");//找到"°C"字符
if(strx)
{
    get_temperature(dbuf);           //得到温度
    strncpy(strx-4,(const char*)dbuf,4); //更新温度
}
strx=strstr((const char*)strx,"RTC 时间:"); //找到"RTC 时间:"字符
if(strx)
{
    get_time(dbuf);                //得到时间
    strncpy(strx+33,(const char*)dbuf,16); //更新时间
}
strncpy(s->filename, http_index_html, sizeof(s->filename));
}else //如果不是'/'?
{
    s->inputbuf[PSOCK_DATALEN(&s->sin)-1] = 0;
    strncpy(s->filename,&s->inputbuf[0],sizeof(s->filename));
}
s->state = STATE_OUTPUT;
while(1)
{
    PSOCK_READTO(&s->sin, ISO_nl);
    if(strncmp(s->inputbuf, http_referer, 8) == 0)
    {
        s->inputbuf[PSOCK_DATALEN(&s->sin) - 2] = 0;
    }
}
PSOCK_END(&s->sin);
}
```



这里，我们需要了解 uIP 是把网页数据（源文件）存放在 `data_index_html`，通过将里面的数据发送给电脑浏览器，浏览器就会显示出我们所设计的界面了。当用户在网页上面操作的时候，浏览器就会发送消息给 WEB 服务器，服务器根据收到的消息内容，判断用户所执行的操作，然后发送新的页面到浏览器，这样用户就可以看到操作结果了。本章，我们实现的 WEB 服界面如图 57.3.2 所示：



图 57.3.2 WEB 服务器界面

图中两个按键分别控制 DS0 和 DS1 的亮灭，然后还显示了 STM32 芯片的温度和 RTC 时间等信息。

控制 DS0, DS1 亮灭我们是通过发送不同的页面请求来实现的，这里我们采用的是 Get 方法（科普找百度），将请求参数放到 URL 里面，然后 WEB 服务器根据 URL 的参数来相应内容，这样实际上 STM32 就是从 URL 获取控制参数，以控制 DS0 和 DS1 的亮灭。uIP 在得到 Get 请求后判断 URL 内容，然后做出相应控制，最后修改 `data_index_html` 里面的部分内容（比如指示灯图标的变化，以及提示文字的变化等），再将 `data_index_html` 发送给浏览器，显示新的界面。

显示 STM32 温度和 RTC 时间是通过刷新实现的，uIP 每次得到来自浏览器的请求就会更新 `data_index_html` 里面的温度和时间等信息，然后将 `data_index_html` 发送给浏览器，这样达到更新温度和时间的目的。但是这样我们需要手动刷新，比较笨，所以我们在网页源码里面加入了自动刷新的控制代码，每 10 秒钟刷新一次，这样就不需要手动刷新了。

`handle_input` 函数实现了我们所说的这一切功能，另外请注意 `data_index_html` 是存放在 `httpd-fsdata.c`（该文件通过 `include` 的方式包含进工程里面）里面的一个数组，并且由于该数组的内容需要不停的刷新，所以我们定义它为 sram 数据，`data_index_html` 里面的数据，则是通过一个工具软件：amo 的编程小工具集合 V1.2.6.exe，将网页源码转换而来，该软件在光盘有提供，如果想自己做网页的朋友，可以通过该软件转换。

WEB 服务器就为大家介绍这么多。



接下来看看 TCP 服务器 appcall 函数: `tcp_server_demo_appcall`, 该函数在 `tcp_server_demo.c` 里面实现, 该函数代码如下:

```
u8 tcp_server_databuf[200];      //发送数据缓存
u8 tcp_server_stata;             //服务端状态
//[7]:0,无连接;1,已经连接;
//[6]:0,无数据;1,收到客户端数据
//[5]:0,无数据;1,有数据需要发送
//这是一个 TCP 服务器应用回调函数。
//该函数通过 UIP_APPCALL(tcp_demo_appcall)调用,实现 Web Server 的功能.
//当 uip 事件发生时, UIP_APPCALL 函数会被调用,根据所属端口(1200),确定是否执行该函数。
//例如 : 当一个 TCP 连接被创建时、有新的数据到达、数据已经被应答、数据需要重发等事件
void tcp_server_demo_appcall(void)
{
    struct tcp_demo_appstate *s = (struct tcp_demo_appstate *)&uip_conn->appstate;
    if(uip_aborted())tcp_server_aborted();      //连接终止
    if(uip_timedout())tcp_server_timedout();    //连接超时
    if(uip_closed())tcp_server_closed();        //连接关闭
    if(uip_connected())tcp_server_connected();  //连接成功
    if(uip_acked())tcp_server_acked();          //发送的数据成功送达
    //接收到一个新的 TCP 数据包
    if (uip_newdata())//收到客户端发过来的数据
    {
        if((tcp_server_stata&(1<<6))==0)//还未收到数据
        {
            if(uip_len>199) ((u8*)uip_appdata)[199]=0;
            strcpy((char*)tcp_server_databuf,uip_appdata);
            tcp_server_stata|=1<<6;//表示收到客户端数据
        }
    }else if(tcp_server_stata&(1<<5))//有数据需要发送
    {
        s->textptr=tcp_server_databuf;
        s->textlen=strlen((const char*)tcp_server_databuf);
        tcp_server_stata&=~(1<<5);//清除标记
    }
    //当需要重发、新数据到达、数据包送达、连接建立时, 通知 uip 发送数据
    if(uip_rexmit()||uip_newdata()||uip_acked()||uip_connected()||uip_poll())
    {
        tcp_server_senddata();
    }
}
```

该函数通过 `uip_newdata()` 判断是否接收到客户端发来的数据, 如果是, 则将数据拷贝到 `tcp_server_databuf` 缓存区, 并标记收到客户端数据。当有数据要发送 (KEY0 按下) 的时候, 将需要发送的数据通过 `tcp_server_senddata` 函数发送出去。



最后，我们看看 TCP 客户端 appcall 函数：tcp_client_demo_appcall，该函数代码同 TCP 服务端代码十分相似，该函数在 tcp_server_demo.c 里面实现，代码如下：

```
u8 tcp_client_databuf[200];      //发送数据缓存
u8 tcp_client_sta;               //客户端状态
//[7]:0,无连接;1,已经连接;
//[6]:0,无数据;1,收到客户端数据
//[5]:0,无数据;1,有数据需要发送
//这是一个 TCP 客户端应用回调函数。
//该函数通过 UIP_APPCALL(tcp_demo_appcall)调用,实现 Web Client 的功能。
//当 uip 事件发生时, UIP_APPCALL 函数会被调用,根据所属端口(1400),确定是否执行该函数。
//例如：当一个 TCP 连接被创建时、有新的数据到达、数据已经被应答、数据需要重发等事件
void tcp_client_demo_appcall(void)
{
    struct tcp_demo_appstate *s = (struct tcp_demo_appstate *)&uip_conn->appstate;
    if(uip_aborted())tcp_client_aborted();          //连接终止
    if(uip_timedout())tcp_client_timedout();        //连接超时
    if(uip_closed())tcp_client_closed();            //连接关闭
    if(uip_connected())tcp_client_connected();       //连接成功
    if(uip_acked())tcp_client_acked();              //发送的数据成功送达
    //接收到一个新的 TCP 数据包
    if (uip_newdata())
    {
        if((tcp_client_sta&(1<<6))==0)//还未收到数据
        {
            if(uip_len>199) ((u8*)uip_appdata)[199]=0;
            strcpy((char*)tcp_client_databuf,uip_appdata);
            tcp_client_sta|=1<<6;//表示收到客户端数据
        }
    }else if(tcp_client_sta&(1<<5))//有数据需要发送
    {
        s->textptr=tcp_client_databuf;
        s->textlen=strlen((const char*)tcp_client_databuf);
        tcp_client_sta&=~(1<<5);//清除标记
    }
    //当需要重发、新数据到达、数据包送达、连接建立时, 通知 uip 发送数据
    if(uip_rexmit()||uip_newdata()||uip_acked()||uip_connected()||uip_poll())
    {
        tcp_client_senddata();
    }
}
```

该函数也是通过 uip_newdata()判断是否接收到服务端发来的数据，如果是，则将数据拷贝到 tcp_client_databuf 缓存区，并标记收到服务端数据。当有数据要发送（KEY2 按下）的时候，将需要发送的数据通过 tcp_client_senddata 函数发送出去。



uIP 通过 clock-arch 里面的 clock_time 获取时间节拍，我们通过在 timerx.c 里面初始化定时器 6，用于提供 clock_time 时钟节拍，每 10ms 加 1，这里代码就不贴出来了，请大家查看光盘源码。

最后在 main.c 里面，我们要实现好几个函数，但是这里仅贴出 main 函数以及 uip_polling 函数，该部分如下：

```
#define BUF ((struct uip_eth_hdr *)&uip_buf[0])  
  
int main(void)  
{  
    u8 key;  
    u8 tcnt=0;  
    u8 tcp_server_tsta=0XFF;  
    u8 tcp_client_tsta=0XFF;  
    uip_ipaddr_t ipaddr;  
  
    delay_init();           //延时函数初始化  
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级  
    uart_init(9600);        //串口初始化波特率为 9600  
    LED_Init();            //LED 端口初始化  
    LCD_Init();            //LCD 初始化  
    KEY_Init();            //初始化按键  
    RTC_Init();            //初始化 RTC  
    Adc_Init();            //初始化 ADC  
    POINT_COLOR=RED;       //设置为红色  
    LCD_ShowString(60,10,200,16,16,"WarShip STM32");  
    LCD_ShowString(60,30,200,16,16,"ENC28J60 TEST");  
    LCD_ShowString(60,50,200,16,16,"ATOM@ALIENTEK");  
    while(tapdev_init())   //初始化 ENC28J60 错误  
    {  
        LCD_ShowString(60,70,200,16,16,"ENC28J60 Init Error!");  
        delay_ms(200);  
        LCD_Fill(60,70,240,86,WHITE);//清除之前显示  
    };  
    uip_init();             //uIP 初始化  
    LCD_ShowString(60,70,200,16,16,"KEY0:Server Send Msg");  
    LCD_ShowString(60,90,200,16,16,"KEY2:Client Send Msg");  
    LCD_ShowString(60,110,200,16,16,"IP:192.168.1.16");  
  
    LCD_ShowString(60,130,200,16,16,"MASK:255.255.255.0");  
  
    LCD_ShowString(60,150,200,16,16,"GATEWAY:192.168.1.1");  
  
    LCD_ShowString(30,200,200,16,16,"TCP RX:");
```



```
LCD_ShowString(30,220,200,16,16,"TCP TX:");

LCD_ShowString(30,270,200,16,16,"TCP RX:");
LCD_ShowString(30,290,200,16,16,"TCP TX:");
POINT_COLOR=BLUE;

uip_ipaddr(ipaddr, 192,168,1,16); //设置本地设置 IP 地址
uip_sethostaddr(ipaddr);
uip_ipaddr(ipaddr, 192,168,1,1); //设置网关 IP 地址(其实就是你路由器的 IP 地址)
uip_setdraddr(ipaddr);
uip_ipaddr(ipaddr, 255,255,255,0); //设置网络掩码
uip_setnetmask(ipaddr);

uip_listen(HTONS(1200)); //监听 1200 端口,用于 TCP Server
uip_listen(HTONS(80)); //监听 80 端口,用于 Web Server
tcp_client_reconnect(); //尝试连接到 TCP Server 端,用于 TCP Client
while (1)
{
    uip_polling(); //处理 uip 事件, 必须插入到用户程序的循环体中
    key=KEY_Scan(0);
    if(tcp_server_tsta!=tcp_server_sto)//TCP Server 状态改变
    {
        if(tcp_server_sto&(1<<7))
            LCD_ShowString(30,180,200,16,16,"TCP Server Connected ");
        else LCD_ShowString(30,180,200,16,16,"TCP Server Disconnected");
        if(tcp_server_sto&(1<<6)) //收到新数据
        {
            LCD_Fill(86,200,240,216,WHITE); //清除之前显示
            LCD_ShowString(86,200,154,16,16,tcp_server_databuf);
            printf("TCP Server RX:%s\r\n",tcp_server_databuf);//打印数据
            tcp_server_sto&=~(1<<6); //标记数据已经被处理
        }
        tcp_server_tsta=tcp_server_sto;
    }
    if(key==KEY_RIGHT)//TCP Server 请求发送数据
    {
        if(tcp_server_sto&(1<<7)) //连接还存在
        {
            sprintf((char*)tcp_server_databuf,"TCP Server OK %d\r\n",tcnt);
            LCD_Fill(86,220,240,236,WHITE);//清除之前显示
            LCD_ShowString(86,220,154,16,16,tcp_server_databuf);//显示当前发送数据
            tcp_server_sto|=1<<5;//标记有数据需要发送
            tcnt++;
        }
    }
}
```



```
        }
    }

    if(tcp_client_tsta!=tcp_client_sta)//TCP Client 状态改变
    {
        if(tcp_client_sta&(1<<7))
            LCD_ShowString(30,250,200,16,16,"TCP Client Connected    ");
        else LCD_ShowString(30,250,200,16,16,"TCP Client Disconnected");
        if(tcp_client_sta&(1<<6)) //收到新数据
        {
            LCD_Fill(86,270,240,286,WHITE); //清除之前显示
            LCD_ShowString(86,270,154,16,16,tcp_client_databuf);
            printf("TCP Client RX:%s\r\n",tcp_client_databuf);//打印数据
            tcp_client_sta&=~(1<<6); //标记数据已经被处理
        }
        tcp_client_tsta=tcp_client_sta;
    }

    if(key==KEY_LEFT)//TCP Client 请求发送数据
    {
        if(tcp_client_sta&(1<<7)) //连接还存在
        {
            sprintf((char*)tcp_client_databuf,"TCP Client OK %d\r\n",tcnt);
            LCD_Fill(86,290,240,306,WHITE); //清除之前显示
            LCD_ShowString(86,290,154,16,16,tcp_client_databuf);//显示当前发送数据
            tcp_client_sta|=1<<5;//标记有数据需要发送
            tcnt++;
        }
    }
    delay_ms(1);
}

//uip 事件处理函数
//必须将该函数插入用户主循环,循环调用.
void uip_polling(void)
{
    u8 i;

    static struct timer periodic_timer, arp_timer;
    static u8 timer_ok=0;
    if(timer_ok==0)//仅初始化一次
    {
        timer_ok = 1;
        timer_set(&periodic_timer,CLOCK_SECOND/2); //创建 1 个 0.5 秒的定时器
        timer_set(&arp_timer,CLOCK_SECOND*10); //创建 1 个 10 秒的定时器
    }
}
```



```
uip_len=tapdev_read(); //从网络设备读取一个 IP 包,得到数据长度.uip_len 在 uip.c 中定义
if(uip_len>0)          //有数据
{
    //处理 IP 数据包(只有校验通过的 IP 包才会被接收)
    if(BUF->type == htons(UIP_ETYPE_IP))//是否是 IP 包?
    {
        uip_arp_ipin();    //去除以太网头结构, 更新 ARP 表
        uip_input();       //IP 包处理
        //当上面的函数执行后, 如果需要发送数据, 则全局变量 uip_len > 0
        //需要发送的数据在 uip_buf, 长度是 uip_len (这是 2 个全局变量)
        if(uip_len>0)//需要回应数据
        {
            uip_arp_out();//加以太网头结构, 在主动连接时可能要构造 ARP 请求
            tapdev_send();//发送数据到以太网
        }
    }
}else if (BUF->type==htons(UIP_ETYPE_ARP))//处理 arp 报文,是否是 ARP 请求包?
{
    uip_arp_arpin();
    //当上面的函数执行后, 如果需要发送数据, 则全局变量 uip_len>0
    //需要发送的数据在 uip_buf, 长度是 uip_len(这是 2 个全局变量)
    if(uip_len>0)tapdev_send();//需要发送数据,则通过 tapdev_send 发送
}
else if(timer_expired(&periodic_timer)) //0.5 秒定时器超时
{
    timer_reset(&periodic_timer);      //复位 0.5 秒定时器
    //轮流处理每个 TCP 连接, UIP_CONNS 缺省是 40 个
    for(i=0;i<UIP_CONNS;i++)
    {
        uip_periodic(i); //处理 TCP 通信事件
        //当上面的函数执行后, 如果需要发送数据, 则全局变量 uip_len>0
        //需要发送的数据在 uip_buf, 长度是 uip_len (这是 2 个全局变量)
        if(uip_len>0)
        {
            uip_arp_out();//加以太网头结构, 在主动连接时可能要构造 ARP 请求
            tapdev_send();//发送数据到以太网
        }
    }
}
#if UIP_UDP //UIP_UDP
//轮流处理每个 UDP 连接, UIP_UDP_CONNS 缺省是 10 个
for(i=0;i<UIP_UDP_CONNS;i++)
{
    uip_udp_periodic(i); //处理 UDP 通信事件
    //当上面的函数执行后, 如果需要发送数据, 则全局变量 uip_len>0
```



```
//需要发送的数据在 uip_buf, 长度是 uip_len (这是 2 个全局变量)
if(uip_len > 0)
{
    uip_arp_out();//加以太网头结构, 在主动连接时可能要构造 ARP 请求
    tapdev_send();//发送数据到以太网
}
#endif

//每隔 10 秒调用 1 次 ARP 定时器函数 用于定期 ARP 处理,
//ARP 表 10 秒更新一次, 旧的条目会被抛弃
if(timer_expired(&arp_timer))
{
    timer_reset(&arp_timer);
    uip_arp_timer();
}

}
```

其中 main 函数相对比较简单，先初始化网卡（ENC28J60）和 uIP 等，然后设置 IP 地址（192.168.1.16）及监听端口（1200 和 80），就开始轮询 uip_polling 函数，实现 uIP 事件处理，同时扫描按键，实现数据发送处理。当有收到数据的时候，将其显示在 LCD 上，同时通过串口发送到电脑。注意，这里 main 函数调用的 tcp_client_reconnect 函数，用于本地（STM32）TCP Client 去连接外部服务端，该函数设置服务端 IP 地址为 192.168.1.103（就是你电脑的 IP 地址），连接端口为 1400，只要没有连上，该函数就会不停的尝试连接。

uip_polling 函数，第一次调用的时候创建两个定时器，当收到包的时候（uip_len>0），先区分是 IP 包还是 ARP 包，针对不同的包做不同处理，对我们来说主要是通过 uip_input 处理 IP 包，实现数据处理。当没有收到包的时候（uip_len=0），通过定时器定时处理各个 TCP/UDP 连接以及 ARP 表处理。

软件设计部分就为大家介绍到这里。

57.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上（假设网络模块已经连接上开发板），LCD 显示如图 57.4.1 所示界面：

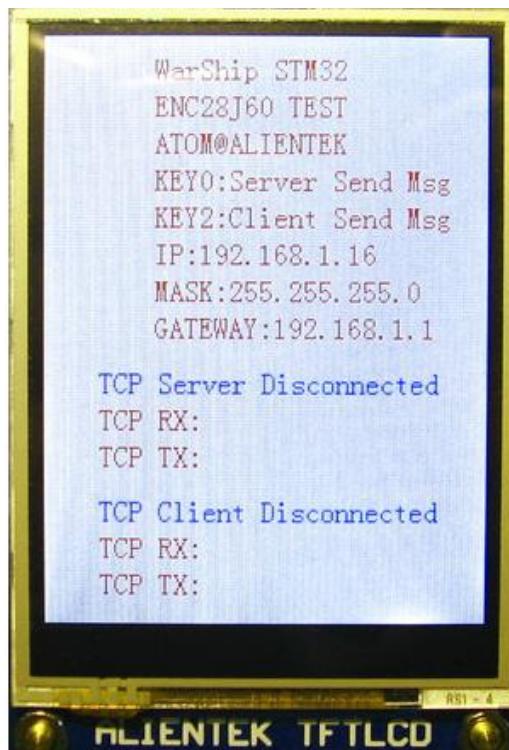


图 57.4.1 初始界面

可以看到,此时 TCP Server 和 TCP Client 都是没有连接的,我们打开:网络调试助手 V3.7.exe 这个软件(该软件在光盘有提供),然后选择 TCP Server, 设置本地 IP 地址为: 192.168.1.103 (默认就是),设置本地端口为 1400,点击连接按钮,就会收到开发板发过来的消息,此时我们按开发板的 KEY2,就会发送数据给网络调试助手,同时也可以通过网络调试助手发送数据到 STM32 开发板。如图 57.4.2 所示:



图 57.4.2 STM32 TCP Client 测试

在连接成功建立的时候，会在战舰 STM32 开发板上面显示 TCP Client 的连接状态，然后如果收到来自电脑 TCP Server 端的数据，也会在 LCD 上面显示，并打印到串口。这是我们实现的 TCP Client 功能。

如果我们在网络调试助手，选择协议类型为 TCP Client，然后设置服务器 IP 地址为 192.168.1.16（就是我们 STM32 开发板设置的 IP 地址），然后设置服务器端口为 1200，点击连接，同样可以收到开发板发过来的消息，此时我们按开发板的 KEY0 按键，就可以发送数据到网络调试助手，同时网络调试助手也可以发送数据到我们的开发板。如图 57.4.3 所示：



图 57.4.3 STM32 TCP Server 测试

在连接成功建立的时候，会在战舰 STM32 开发板上面显示 TCP Server 的连接状态，然后如果收到来自电脑 TCP Client 端的数据，便会在 LCD 上面显示，并打印到串口。这是我们实现的 TCP Server 功能。

最后，我们测试 WEB 服务器功能。打开浏览器，输入 <http://192.168.1.16>，就可以看到如下界面，如图 57.4.4 所示：

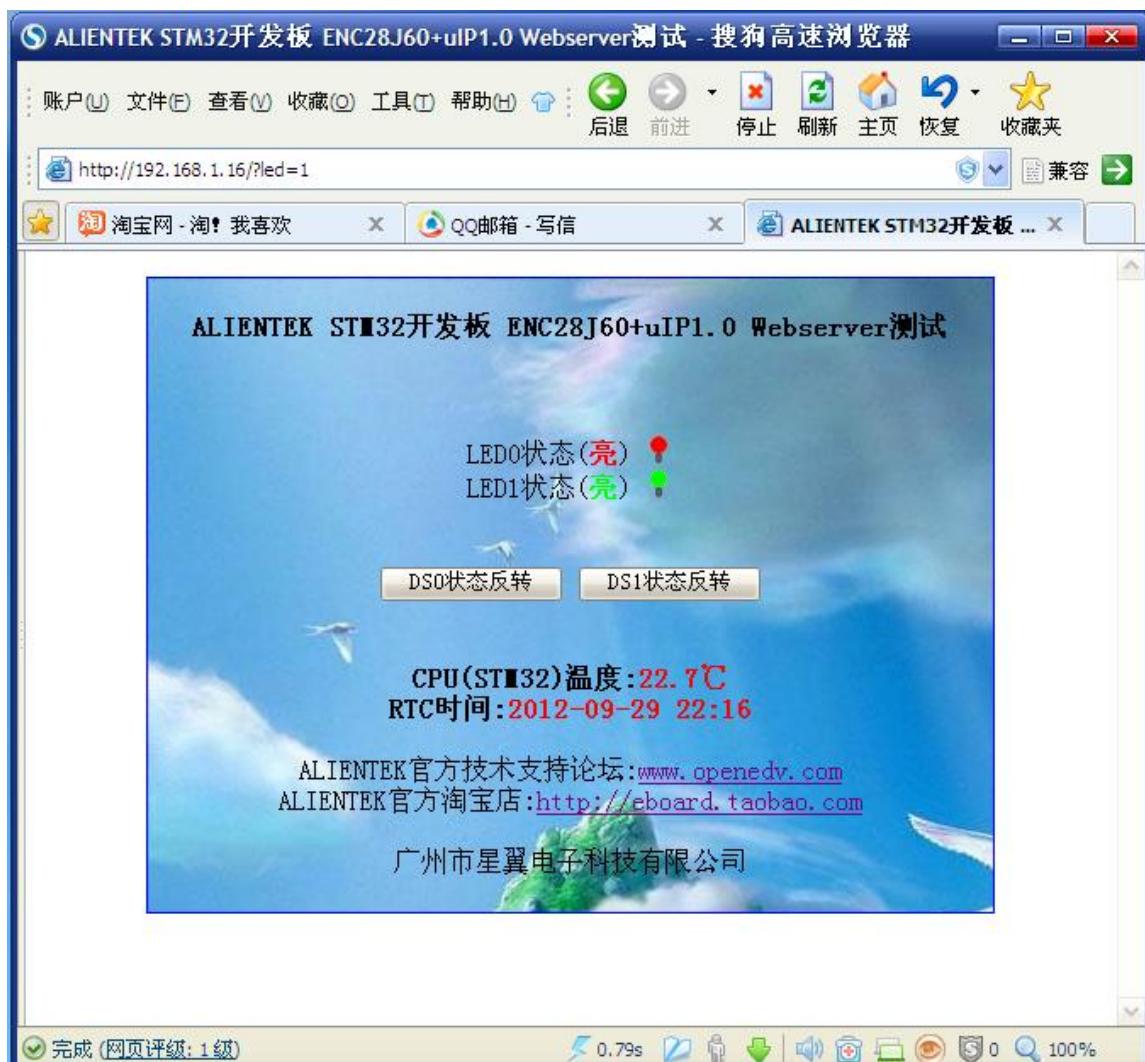


图 57.4.4 STM32 WEB Server 测试

此时，我们点击网页上的 DS0 状态反转和 DS1 状态反转按钮，就可以控制 DS0 和 DS1 的亮灭了。同时在该界面还显示了 STM32 的温度和 RTC 时间，每次刷新的时候，进行数据更新，另外浏览器每 10 秒钟会自动刷新一次，以更新时间和温度信息。



第五十八章 UCOSII 实验 1-任务调度

前面我们所有的例程都是跑的裸机程序（裸奔），从本章开始，我们将分 3 个章节向大家介绍 UCOSII（实时多任务操作系统内核）的使用。本章，我们将向大家介绍 UCOSII 最基本也是最重要的应用：任务调度。本章分为如下几个部分：

58.1 UCOSII 简介

58.2 硬件设计

58.3 软件设计

58.4 下载验证



58.1 UCOSII 简介

UCOSII 的前身是 UCOS，最早出自于 1992 年美国嵌入式系统专家 Jean J.Labrosse 在《嵌入式系统编程》杂志的 5 月和 6 月刊上刊登的文章连载，并把 UCOS 的源码发布在该杂志的 BBS 上。目前最新的版本：UCOSIII 已经出来，但是现在使用最为广泛的还是 UCOSII，本章我们主要针对 UCOSII 进行介绍。在学习本章之前，UCOS 相关的知识比较多，我们实验也知识指点一下大家入门，详细了解建议大家先看看任哲老师的《嵌入式实时操作系统 ucoss II 原理及应用》，这本书的 Pdf 我们光盘有，大家可以翻阅一下。同时我们光盘还提供了一个北航老师的 UCOS 简明讲义，大家也可以翻阅一下。

UCOSII 是一个可以基于 ROM 运行的、可裁减的、抢占式、实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器，是和很多商业操作系统性能相当的实时操作系统 (RTOS)。为了提供最好的移植性能，UCOSII 最大程度上使用 ANSI C 语言进行开发，并且已经移植到近 40 多种处理器体系上，涵盖了从 8 位到 64 位各种 CPU(包括 DSP)。

UCOSII 是专门为计算机的嵌入式应用设计的，绝大部分代码是用 C 语言编写的。CPU 硬件相关部分是用汇编语言编写的、总量约 200 行的汇编语言部分被压缩到最低限度，为的是便于移植到任何一种其它的 CPU 上。用户只要有标准的 ANSI 的 C 交叉编译器，有汇编器、连接器等软件工具，就可以将 UCOSII 嵌入到开发的产品中。UCOSII 具有执行效率高、占用空间小、实时性能优良和可扩展性强等特点，最小内核可编译至 2KB。UCOSII 已经移植到了几乎所有知名的 CPU 上。

UCOSII 构思巧妙。结构简洁精练，可读性强，同时又具备了实时操作系统的全部功能，虽然它只是一个内核，但非常适合初次接触嵌入式实时操作系统的的朋友，可以说是麻雀虽小，五脏俱全。UCOSII (V2.91 版本) 体系结构如图 58.1.1 所示：

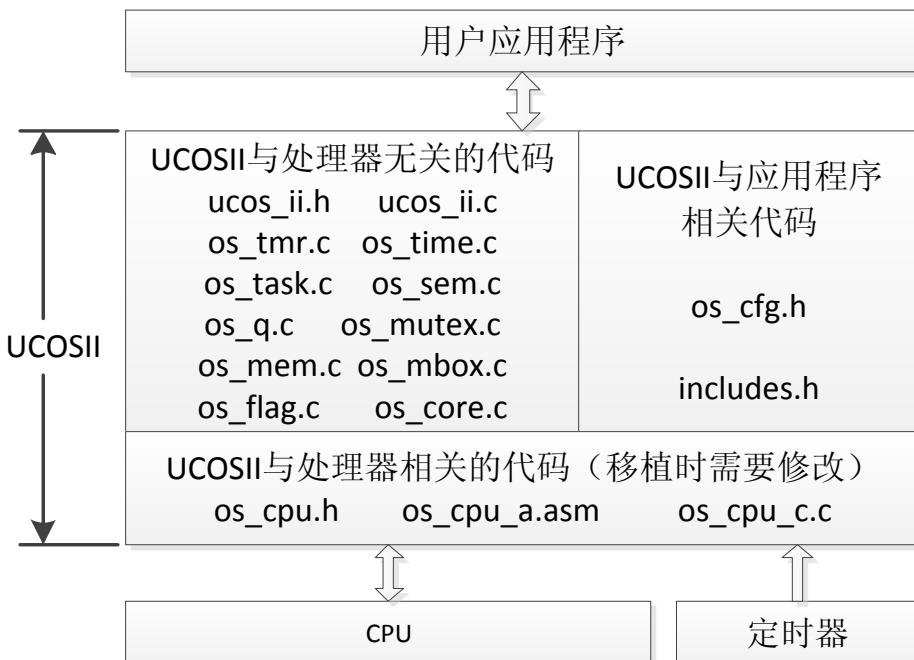


图 58.1.1 UCOSII 体系结构图

注意本章我们使用的是 UCOSII 的最新版本：V2.91 版本，该版本 UCOSII 比早期的 UCOSII (如 V2.52) 多了很多功能（比如多了软件定时器，支持任务数最大达到 255 个等），而且修正了很多已知 BUG。不过，有两个文件：os_dbg_r.c 和 os_dbg.c，我们没有在上图列出，也不



将其加入到我们的工程中，这两个主要用于对 UCOS 内核进行调试支持，比较少用到。

从上图可以看出，UCOSII 的移植，我们只需要修改：os_cpu.h、os_cpu_a.asm 和 os_cpu.c 等三个文件即可，其中：os_cpu.h，进行数据类型的定义，以及处理器相关代码和几个函数原型；os_cpu_a.asm，是移植过程中需要汇编完成的一些函数，主要就是任务切换函数；os_cpu.c，定义一些用户 HOOK 函数。

图中定时器的作用是为 UCOSII 提供系统时钟节拍，实现任务切换和任务延时等功能。这个时钟节拍由 OS_TICKS_PER_SEC（在 os_cfg.h 中定义）设置，一般我们设置 UCOSII 的系统时钟节拍为 1ms~100ms，具体根据你所用处理器和使用需要来设置。本章，我们利用 STM32 的 SYSTICK 定时器来提供 UCOSII 时钟节拍。

关于 UCOSII 在 STM32 的详细移植，请参考光盘资料（《UCOSII 在 STM32 的移植详解.pdf》），这里我们就不详细介绍了。

UCOSII 早期版本只支持 64 个任务，但是从 2.80 版本开始，支持任务数提高到 255 个，不过对我们来说一般 64 个任务都是足够多了，一般很难用到这么多个任务。UCOSII 保留了最高 4 个优先级和最低 4 个优先级的总共 8 个任务，用于拓展使用，单实际上，UCOSII 一般只占用了最低 2 个优先级，分别用于空闲任务（倒数第一）和统计任务（倒数第二），所以剩下给我们使用的任务最多可达 $255-2=253$ 个（V2.91）。

UCOS 是怎样实现多任务并发工作的呢？外部中断相信大家都比较熟悉了。CPU 在执行一段用户代码的时候，如果此时发生了外部中断，那么先进行现场保护，之后转向中断服务程序执行，执行完成后恢复现场，从中断处开始执行原来的用户代码。Ucos 的原理本质上也是这样的，当一个任务 A 正在执行的时候，如果他释放了 cpu 控制权，先对任务 A 进行现场保护，然后从任务就绪表中查找其他就绪任务去执行，等到任务 A 的等待时间到了，它可能重新获得 cpu 控制权，这个时候恢复任务 A 的现场，从而继续执行任务 A，这样看起来就好像两个任务同时执行了。实际上，任何时候，只有一个任务可以获得 cpu 控制权。这个过程很负责，场景也多样，这里只是举个简单的例子说明。

所谓的任务，其实就是一个死循环函数，该函数实现一定的功能，一个工程可以有很多这样的任务（最多 255 个），UCOSII 对这些任务进行调度管理，让这些任务可以并发工作（注意不是同时工作！！，并发只是各任务轮流占用 CPU，而不是同时占用，任何时候还是只有 1 个任务能够占用 CPU），这就是 UCOSII 最基本的功能。Ucos 任务的一般格式为：

```
void MyTask (void *pdata)
{
    任务准备工作...
    While(1)//死循环
    { 任务 MyTask 实体代码;
        OSTimeDlyHMSM(x,x,x); //调用任务延时函数，释放 cpu 控制权,
    }
}
```

假如我们新建了 2 个任务为 MyTask 和 YourTask，这里我们先忽略任务优先级的概念，两个任务死循环中延时时间为 1s。如果某个时刻，任务 MyTask 在执行中，当它执行到延时函数 OSTimeDlyHMSM 的时候，它释放 cpu 控制权，这个时候，任务 YourTask 获得 cpu 控制权开始执行，任务 YourTask 执行过程中，也会调用延时函数延时 1s 释放 CPU 控制权，这个过程中任务 A 延时 1s 到达，重新获得 CPU 控制权，重新开始执行死循环中的任务实体代码。如此循环，现象就是两个任务交替运行，就好像 CPU 在同时做两件事情一样。

疑问来了，如果有很多任务都在等待，那么先执行那个任务呢？如果任务在执行过程中，



想停止之后去执行其他任务是否可行呢？这里就涉及到任务优先级以及任务状态任务控制的一些知识，我们在后面会有所提到。如果要详细的学习，建议看任哲老师的《ucosII 实时操作系统》一书。

前面我们学习的所有实验，都是一个大任务（死循环），这样，有些事情就比较不好处理，比如：MP3 实验，在 MP3 播放的时候，我们还希望显示歌词，如果是 1 个死循环（一个任务），那么很可能在显示歌词的时候，MP3 声音出现停顿（尤其是高码率的时候），这主要是歌词显示占用太长时间，导致 VS1053 由于不能及时得到数据而停顿。而如果用 UCOSII 来处理，那么我们可以分 2 个任务，MP3 播放一个任务（优先级高），歌词显示一个任务（优先级低）。这样，由于 MP3 任务的优先级高于歌词显示任务，MP3 任务可以打断歌词显示任务，从而及时给 VS1053 提供数据，保证音频不断，而显示歌词又能顺利进行。这就是 UCOSII 带来的好处。

这里有几个 UCOSII 相关的概念需要大家了解一下。任务优先级，任务堆栈，任务控制块，任务就绪表和任务调度器。

任务优先级，这个概念比较好理解，ucos 中，每个任务都有唯一的一个优先级。优先级是任务的唯一标识。在 UCOSII 中，使用 CPU 的时候，优先级高（数值小）的任务比优先级低的任务具有优先使用权，即任务就绪表中总是优先级最高的任务获得 CPU 使用权，只有高优先级的任务让出 CPU 使用权（比如延时）时，低优先级的任务才能获得 CPU 使用权。UCOSII 不支持多个任务优先级相同，也就是每个任务的优先级必须不一样。

任务堆栈，就是存储器中的连续存储空间。为了满足任务切换和响应中断时保存 CPU 寄存器中的内容以及任务调用其他函数时的需要，每个任务都有自己的堆栈。在创建任务的时候，任务堆栈是任务创建的一个重要入口参数。

任务控制块 OS_TCB，用来记录任务堆栈指针，任务当前状态以及任务优先级等任务属性。UCOSII 的任何任务都是通过任务控制块（TCB）的东西来控制的，一旦任务创建了，任务控制块 OS_TCB 就会被赋值。每个任务管理块有 3 个最重要的参数：1，任务函数指针；2，任务堆栈指针；3，任务优先级；任务控制块就是任务在系统里面的身份证件（UCOSII 通过优先级识别任务），任务控制块我们就不再详细介绍了，详细介绍请参考任哲老师的《嵌入式实时操作系统 UCOSII 原理及应用》一书第二章。

任务就绪表，简而言之就是用来记录系统中所有处于就绪状态的任务。它是一个位图，系统中每个任务都在这个位图中占据一个进制位，该位置的状态（1 或者 0）就表示任务是否处于就绪状态。

任务调度的作用一是在任务就绪表中查找优先级最高的就绪任务，二是实现任务的切换。比如说，当一个任务释放 cpu 控制权后，进行一次任务调度，这个时候任务调度器首先要去任务就绪表查询优先级最高的就绪任务，查到之后，进行一次任务切换，转而去执行下一个任务。关于任务调度的详细介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》一书第三章相关内容。

UCOSII 的每个任务都是一个死循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是：睡眠状态、就绪状态、运行状态、等待状态(等待某一事件发生)和中断服务状态。

睡眠状态，任务在没有被配备任务控制块或被剥夺了任务控制块时的状态。

就绪状态，系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，任务已经准备好了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行，这时任务的状态叫做就绪状态。

运行状态，该任务获得 CPU 使用权，并正在运行中，此时的任务状态叫做运行状态。

等待状态，正在运行的任务，需要等待一段时间或需要等待一个事件发生再运行时，该任



务就会把 CPU 的使用权让给别的任务而使任务进入等待状态。

中断服务状态，一个正在运行的任务一旦响应中断申请就会中止运行而去执行中断服务程序，这时任务的状态叫做中断服务状态。

UCOSII 任务的 5 个状态转换关系如图 58.1.2 所示：

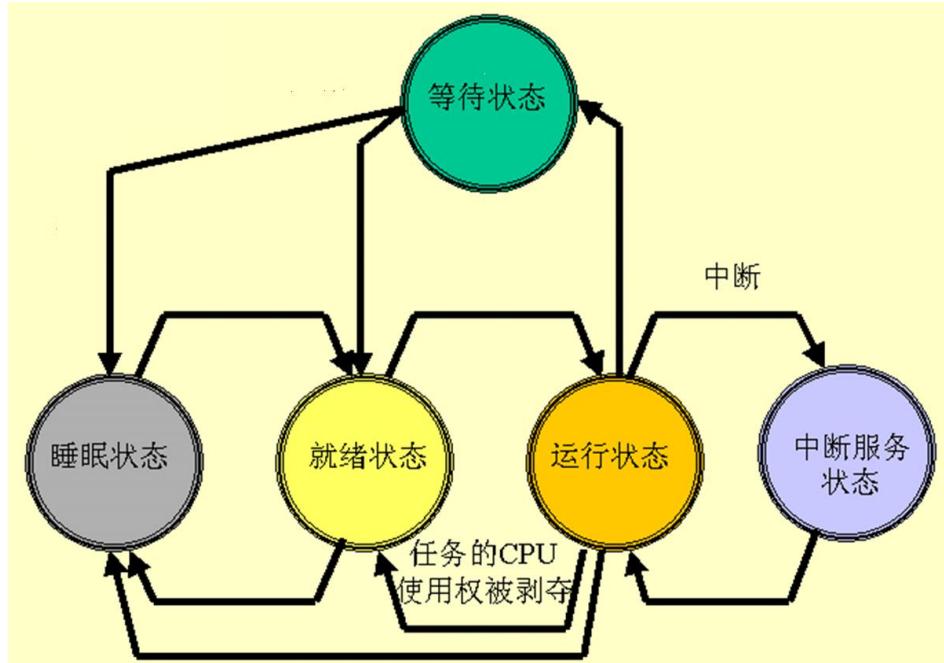


图 58.1.2 UCOSII 任务状态转换关系

接下来，我们看看在 UCOSII 中，与任务相关的几个函数：

1) 建立任务函数

如果想让 UCOSII 管理用户的任务，必须先建立任务。UCOSII 提供了我们 2 个建立任务的函数：OSTaskCreat 和 OSTaskCreateExt，我们一般用 OSTaskCreat 函数来创建任务，该函数原型为：

```
OSTaskCreate(void(*task)(void*pd),void*pdata,OS_STK*ptos,INTU prio);
```

该函数包括 4 个参数：task：是指向任务代码的指针；pdata：是任务开始执行时，传递给任务的参数的指针；ptos：是分配给任务的堆栈的栈顶指针；prio 是分配给任务的优先级。

每个任务都有自己的堆栈，堆栈必须申明为 OS_STK 类型，并且由连续的内存空间组成。可以静态分配堆栈空间，也可以动态分配堆栈空间。

OSTaskCreateExt 也可以用来创建任务，是 OSTaskCreate 的扩展版本，提供一些附件功能。详细介绍请参考《嵌入式实时操作系统 UCOSII 原理及应用》3.5.2 节。

2) 任务删除函数

所谓的任务删除，其实就是把任务置于睡眠状态，并不是把任务代码给删除了。UCOSII 提供的任务删除函数原型为：

```
INT8U OSTaskDel(INT8U prio);
```

其中参数 prio 就是我们要删除的任务的优先级，可见该函数是通过任务优先级来实现任务删除的。

特别注意：任务不能随便删除，必须在确保被删除任务的资源被释放的前提下才能删除！

3) 请求任务删除函数



前面提到，必须确保被删除任务的资源被释放的前提下才能将其删除，所以我们通过向被删除任务发送删除请求，来实现任务释放自身占用资源后再删除。UCOSII 提供的请求删除任务函数原型为：

```
INT8U OSTaskDelReq(INT8U prio);
```

同样还是通过优先级来确定被请求删除任务。

4) 改变任务的优先级函数

UCOSII 在建立任务时，会分配给任务一个优先级，但是这个优先级并不是一成不变的，而是可以通过调用 UCOSII 提供的函数修改。UCOSII 提供的任务优先级修改函数原型为：

```
INT8U OSTaskChangePrio(INT8U oldprio, INT8U newprio);
```

5) 任务挂起函数

任务挂起和任务删除有点类似，但是又有区别，任务挂起只是将被挂起任务的就绪标志删除，并做任务挂起记录，并没有将任务控制块任务控制块链表里面删除，也不需要释放其资源，而任务删除则必须先释放被删除任务的资源，并将被删除任务的任务控制块也给删了。被挂起的任务，在恢复（解挂）后可以继续运行。UCOSII 提供的任务挂起函数原型为：

```
INT8U OSTaskSuspend(INT8U prio);
```

6) 任务恢复函数

有任务挂起函数，就有任务恢复函数，通过该函数将被挂起的任务恢复，让调度器能够重新调度该函数。UCOSII 提供的任务恢复函数原型为：

```
INT8U OSTaskResume(INT8U prio);
```

7) 任务信息查询

在应用程序中我们经常要了解任务信息，查询任务信息函数原型为：

```
INT8U OSTaskQuery(INT8U prio, OS_TCB *pdata);
```

这个函数获得的是对应任务的 OS_TCB 中内容的拷贝。

从上面这些函数我们可以看出，对于每个任务，有一个非常关键的参数就是任务优先级 prio，在 UCOS 中，任务优先级可以用来作为任务的唯一标识，所以任务优先级对任务而言是唯一的，而且是不可重复的。

UCOSII 与任务相关的函数我们就介绍这么多。最后，我们来看看在 STM32 上面运行 UCOSII 的步骤：

1) 移植 UCOSII

要想 UCOSII 在 STM32 正常运行，当然首先是需要移植 UCOSII，这部分我们已经为大家做好了（参考光盘源码，想自己移植的，请参考光盘 UCOSII 资料）。

这里我们要特别注意一个地方，ALIENTEK 提供的 SYSTEM 文件夹里面的系统函数直接支持 UCOSII，只需要在 sys.h 文件里面将：SYSTEM_SUPPORT_UCOS 宏定义改为 1，即可通过 delay_init 函数初始化 UCOSII 的系统时钟节拍，为 UCOSII 提供时钟节拍。

2) 编写任务函数并设置其堆栈大小和优先级等参数。

编写任务函数，以便 UCOSII 调用。

设置函数堆栈大小，这个需要根据函数的需求来设置，如果任务函数的局部变量多，嵌套层数多，那么相应的堆栈就得大一些，如果堆栈设置小了，很可能出现的结果就是 CPU 进入 HardFault，遇到这种情况，你就必须把堆栈设置大一点了。另外，有些地方还需要注意堆栈字节对齐的问题，如果任务运行出现莫名其妙的错误（比如用到 sprintf 出错），请考虑是不是字节对齐的问题。

设置任务优先级，这个需要大家根据任务的重要性和实时性设置，记住高优先级的任务有优先使用 CPU 的权利。

3) 初始化 UCOSII，并在 UCOSII 中创建任务

调用 OSInit，初始化 UCOSII 的所有变量和数据结构，然后通过调用 OSTaskCreate 函数创建我们的任务。

4) 启动 UCOSII

调用 OSSStart，启动 UCOSII。

通过以上 4 个步骤，UCOSII 就开始在 STM32 上面运行了，这里还需要注意我们必须对 os_cfg.h 进行部分配置，以满足我们自己的需要。

58.2 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 3 个任务：开始任务、LED0 任务和 LED1 任务，开始任务用于创建其他（LED0 和 LED1）任务，之后挂起；LED0 任务用于控制 DS0 的亮灭，DS0 每秒钟亮 80ms；LED1 任务用于控制 DS1 的亮灭，DS1 亮 300ms，灭 300ms，依次循环。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 、 DS1

58.3 软件设计

本章，我们在第六章实验（实验 1）的基础上修改，在该工程源码下面加入 UCOSII 文件夹，存放 UCOSII 源码（我们已经将 UCOSII 源码分为三个文件夹：CORE、PORT 和 CONFIG）。

打开工程，新建 UCOSII-CORE、UCOSII-PORT 和 UCOSII-CONFIG 三个分组，分别添加 UCOSII 三个文件夹下的源码，并将这三个文件夹加入头文件包含路径，最后得到工程如图 58.3.1 所示：

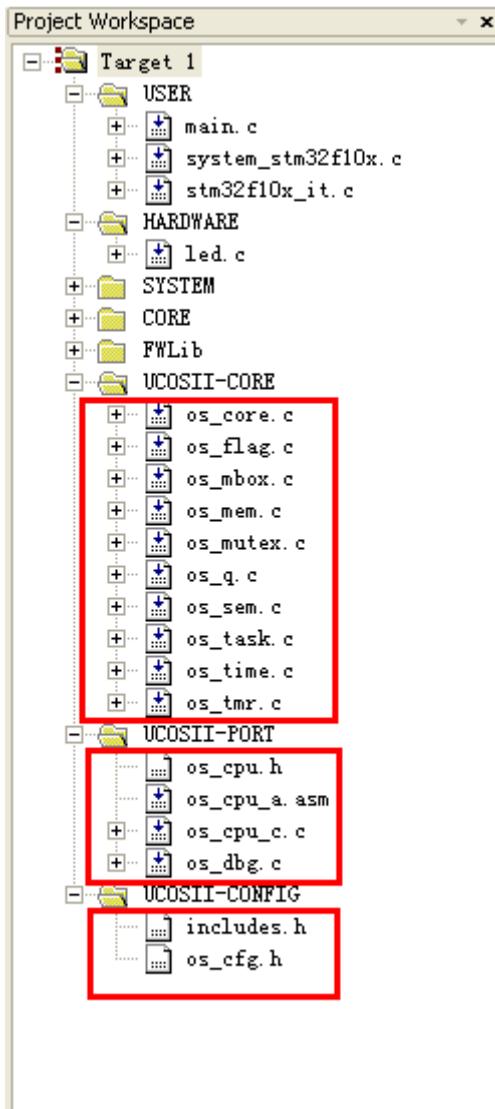


图 58.3.1 添加 UCOSII 源码后的工程

UCOSII-CORE 分组下面是 UCOSII 的核心源码，我们不需要做任何变动。

UCOSII-PORT 分组下面是我们移植 UCOSII 要修改的 3 个代码，这个在移植的时候完成。

UCOSII-CONFIG 分组下面是 UCOSII 的配置部分，主要由用户根据自己的需要对 UCOSII 进行裁剪或其他设置。

本章，我们对 os_cfg.h 里面定义 OS_TICKS_PER_SEC 的值为 200，也就是设置 UCOSII 的时钟节拍为 5ms，同时设置 OS_MAX_TASKS 为 10，也就是最多 10 个任务（包括空闲任务和统计任务在内），其他配置我们就不详细介绍，请参考本实验源码。

前面提到，我们需要在 sys.h 里面设置 SYSTEM_SUPPORT_UCOS 为 1，以支持 UCOSII，通过这个设置，我们不仅可以实现利用 delay_init 来初始化 SYSTICK，产生 UCOSII 的系统时钟节拍，还可以让 delay_us 和 delay_ms 函数在 UCOSII 下能够正常使用（实现原理请参考 5.1 节），这使得我们之前的代码，可以十分方便的移植到 UCOSII 下。虽然 UCOSII 也提供了延时函数：OSTimeDly 和 OSTimeDLyHMSM，但是这两个函数的最少延时单位只能是 1 个 UCOSII 时钟节拍，在本章，即 5ms，显然不能实现 us 级的延时，而 us 级的延时很多时候非常有用：比如 IIC 模拟时序，DS18B20 等单总线器件操作等。而通过我们提供的 delay_us 和 delay_ms，则可以方便的提供 us 和 ms 的延时服务，这比 UCOSII 本身提供的延时函数更好用。



在设置 SYSTEM_SUPPORT_UCOS 为 1 之后，UCOSII 的时钟节拍由 SYSTICK 的中断服务函数提供，该部分代码如下：

```
//systick 中断服务函数,使用 ucos 时用到
void SysTick_Handler(void)
{
    OSIntEnter();           //进入中断
    OSTimeTick();           //调用 ucos 的时钟服务程序
    OSIntExit();            //触发任务切换软中断
}
```

以上代码，其中 OSIntEnter 是进入中断服务函数，用来记录中断嵌套层数（OSIntNesting 增加 1）；OSTimeTick 是系统时钟节拍服务函数，在每个时钟节拍了解每个任务的延时状态，使已经到达延时时限的非挂起任务进入就绪状态；OSIntExit 是退出中断服务函数，该函数可能触发一次任务切换（当 OSIntNesting==0&&调度器未上锁&&就绪表最高优先级任务!=被中断的任务优先级时），否则继续返回原来的任务执行代码（如果 OSIntNesting 不为 0，则减 1）。

事实上，任何中断服务函数，我们都应该加上 OSIntEnter 和 OSIntExit 函数，这是因为 UCOSII 是一个可剥夺型的内核，中断服务子程序运行之后，系统会根据情况进行一次任务调度去运行优先级别最高的就绪任务，而并不一定接着运行被中断的任务！

最后，我们打开 main.c，代码如下：

```
//////////////////UCOSII 任务堆栈设置/////////////////
//START 任务
//设置任务优先级
#define START_TASK_PRIO          10 //开始任务的优先级设置为最低
//设置任务堆栈大小
#define START_STK_SIZE            64
//创建任务堆栈空间
OS_STK START_TASK_STK[START_STK_SIZE];
//任务函数接口
void start_task(void *pdata);

//LED0 任务
//设置任务优先级
#define LED0_TASK_PRIO            7
//设置任务堆栈大小
#define LED0_STK_SIZE              64
//创建任务堆栈空间
OS_STK LED0_TASK_STK[LED0_STK_SIZE];
//任务函数接口
void led0_task(void *pdata);

//LED1 任务
//设置任务优先级
#define LED1_TASK_PRIO            6
//设置任务堆栈大小
```



```
#define LED1_STK_SIZE          64
//创建任务堆栈空间
OS_STK LED1_TASK_STK[LED1_STK_SIZE];
//任务函数接口
void led1_task(void *pdata);
int main(void)
{
    delay_init();           //延时初始化
    NVIC_Configuration();  //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    LED_Init();             //初始化与 LED 连接的硬件接口
    OSInit();               //UCOSII 初始化
    OSTaskCreate(start_task,(void *)0,(OS_STK *)&START_TASK_STK
                 [START_STK_SIZE-1],START_TASK_PRIO );//创建起始任务
    OSStart();
}
//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    pdata = pdata;
    OS_ENTER_CRITICAL();      //进入临界区(无法被中断打断)
    OSTaskCreate(led0_task,(void *)0,(OS_STK*)&LED0_TASK_STK[LED0_STK_SIZE-1],
                LED0_TASK_PRIO);
    OSTaskCreate(led1_task,(void *)0,(OS_STK*)&LED1_TASK_STK[LED1_STK_SIZE-1],
                LED1_TASK_PRIO);
    OSTaskSuspend(START_TASK_PRIO); //挂起起始任务.
    OS_EXIT_CRITICAL();           //退出临界区(可以被中断打断)
}
//LED0 任务
void led0_task(void *pdata)
{
    while(1)
    {
        LED0=0; delay_ms(80);
        LED0=1; delay_ms(920);
    };
}

//LED1 任务
void led1_task(void *pdata)
{
    while(1)
    {
```



```
LED1=0; delay_ms(300);
LED1=1; delay_ms(300);
};

}
```

可以看到，我们在创建 start_task 之前首先调用 ucos 初始化函数 OSInit()，该函数的作用是初始化 ucos 的所有变量和数据结构，该函数必须在调用其他任何 ucos 函数之前调用。在 start_task 创建之后，我们调用 ucos 多任务启动函数 OSStart()，调用这个函数之后，任务才真正开始运行。在这段代码中我们创建了 3 个任务：start_task、led0_task 和 led1_task，优先级分别是 10、7 和 6，堆栈大小都是 64(注意 OS_STK 为 32 位数据)。我们在 main 函数只创建了 start_task 一个任务，然后在 start_task 再创建另外两个任务，在创建之后将自身 (start_task) 挂起。这里，我们单独创建 start_task，是为了提供一个单一任务，实现应用程序开始运行之前的准备工作(比如：外设初始化、创建信号量、创建邮箱、创建消息队列、创建信号量集、创建任务、初始化统计任务等等)。

在应用程序中经常有一些代码段必须不受任何干扰地连续运行，这样的代码段叫做临界段（或临界区）。因此，为了使临界段在运行时不受中断所打断，在临界段代码前必须用关中断指令使 CPU 屏蔽中断请求，而在临界段代码后必须用开中断指令解除屏蔽使得 CPU 可以响应中断请求。UCOSII 提供 OS_ENTER_CRITICAL 和 OS_EXIT_CRITICAL 两个宏来实现，这两个宏需要我们在移植 UCOSII 的时候实现，本章我们采用方法 3 (即 OS_CRITICAL_METHOD 为 3) 来实现这两个宏。因为临界段代码不能被中断打断，将严重影响系统的实时性，所以临界段代码越短越好！

在 start_task 任务中，我们在创建 led0_task 和 led1_task 的时候，不希望中断打断，故使用了临界区。其他两个任务，就十分简单了，我们就不细说了，注意我们这里使用的延时函数还是 delay_ms，而不是直接使用的 OSTimeDly。

另外，一个任务里面一般是必须有延时函数的，以释放 CPU 使用权，否则可能导致低优先级的任务因高优先级的任务不释放 CPU 使用权而一直无法得到 CPU 使用权，从而无法运行。

软件设计部分就为大家介绍到这里。

58.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上，可以看到 DS0 一秒钟闪一次，而 DS1 则以固定的频率闪烁，说明两个任务 (led0_task 和 led1_task) 都已经正常运行了，符合我们预期的设计。

58.5 任务删除，挂起和恢复测试

前面我们简单的建立了两个任务，主要是让大家了解 UCOSII 怎么运行以及怎样创建任务。下面我们在这一节补充一个实验测试任务的删除，挂起和恢复。为了和寄存器版本手册章节保持一致，我们这里不另起一章。实验代码在我们光盘的“实验 53 UCOSII 入门实验 1-2-任务创建删除挂起恢复”中，主函数文件 main.c 源码如下：

#define START_TASK_PRIO	10 //开始任务的优先级设置为最低
//设置任务堆栈大小	
#define START_STK_SIZE	64



```
//创建任务堆栈空间
OS_STK START_TASK_STK[START_STK_SIZE];
//任务函数接口
void start_task(void *pdata);

//LED 任务
//设置任务优先级
#define LED_TASK_PRIO          7
//设置任务堆栈大小
#define LED_STK_SIZE            64
//创建任务堆栈空间
OS_STK LED_TASK_STK[LED_STK_SIZE];
//任务函数接口
void led_task(void *pdata);

//蜂鸣器任务
//设置任务优先级
#define BEEP_TASK_PRIO          5
//设置任务堆栈大小
#define BEEP_STK_SIZE            64
//创建任务堆栈空间
OS_STK BEEP_TASK_STK[BEEP_STK_SIZE];
//任务函数接口
void beep_task(void *pdata);

//按键扫描任务
//设置任务优先级
#define KEY_TASK_PRIO           3
//设置任务堆栈大小
#define KEY_STK_SIZE              64
//创建任务堆栈空间
OS_STK KEY_TASK_STK[KEY_STK_SIZE];
//任务函数接口
void key_task(void *pdata);

int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration();  //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //初始化与 LED 连接的硬件接口
    BEEP_Init();            //蜂鸣器初始化
    KEY_Init();             //按键初始化
```



```
OSInit();           //初始化 UCOSII
OSTaskCreate(start_task,(void *)0,(OS_STK *)&START_TASK_STK
[START_STK_SIZE-1],START_TASK_PRIO );//创建起始任务
OSStart();
}

//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    pdata = pdata;
    OSStatInit();           //初始化统计任务.这里会延时 1 秒钟左右
    OS_ENTER_CRITICAL();    //进入临界区(无法被中断打断)
    OSTaskCreate(led_task,(void *)0,(OS_STK*)&LED_TASK_STK
[LED_STK_SIZE-1],LED_TASK_PRIO);
    OSTaskCreate(beep_task,(void *)0,(OS_STK*)&BEEP_TASK_STK
[BEEP_STK_SIZE-1],BEEP_TASK_PRIO);
    OSTaskCreate(key_task,(void *)0,(OS_STK*)&KEY_TASK_STK
[KEY_STK_SIZE-1],KEY_TASK_PRIO);
    OSTaskSuspend(START_TASK_PRIO);   //挂起起始任务.
    OS_EXIT_CRITICAL();             //退出临界区(可以被中断打断)
}
//LED 任务
void led_task(void *pdata)
{
    while(1)
    {
        LED0=!LED0;
        LED1=!LED1;
        delay_ms(500);
    }
}

//蜂鸣器任务
void beep_task(void *pdata)
{
    while(1)
    {
        if(OSTaskDelReq(OS_PRIO_SELF)==OS_ERR_TASK_DEL_REQ) //判断是否有删除请求
        {
            OSTaskDel(OS_PRIO_SELF);                  //删除任务本身
        }
        BEEP=1;
    }
}
```



```
    delay_ms(60);
    BEEP=0;
    delay_ms(940);
}
}

//按键扫描任务
void key_task(void *pdata)
{
    u8 key;
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY_RIGHT)
        {
            OSTaskSuspend(LED_TASK_PRIO);      //挂起 LED 任务， LED 停止闪烁
        }
        else if (key==KEY_LEFT)
        {
            OSTaskResume(LED_TASK_PRIO);      //恢复 LED 任务， LED 恢复闪烁
        }
        else if (key==KEY_UP)
        {
            OSTaskDelReq(BEEP_TASK_PRIO);     //发送删除 BEEP 任务请求,
        }
        else if(key==KEY_DOWN)
        {
            OSTaskCreate(beep_task,(void *)0,(OS_STK*)&BEEP_TASK_STK
                         [BEEP_STK_SIZE-1],BEEP_TASK_PRIO);//重新创建任务 beep
        }
        delay_ms(10);
    }
}
```

该代码在 start_task 中创建了 3 个任务分别为 led_task, beep_task 和 key_task。led_task 是 LED0 和 LED1 每隔 500ms 翻转一次。beep_task 在没有收到删除请求的时候是隔一段时间蜂鸣器鸣叫一次，key_task 是进行按键扫描。当 KEY_RIGHT 按键按下的时候挂起任务 led_task，这是 LED0 和 LED1 停止闪烁。当 KEY_LEFT 按键按下的时候，如果 led_task 被挂起则恢复之，如果没有挂起则没有影响。当 KEY_UP 按键按下的时候删除任务 beep_task。当 KEY_DOWN 按键按下的时候，重新创建任务 beep_task。

我们的测试顺序为：首先下载代码之后可以看到 LED0 和 LED1 不断闪烁，同时蜂鸣器不断鸣叫。这个时候我们按下 KEY_RIGHT 之后 led_task 任务被挂起，我们可以看到 LED 不再闪烁。接着我们按下 KEY_LEFT， led_task 任务重新恢复，可以看到 LED 恢复闪烁。然后我们按



下 KEY_UP，任务 beep_task 被删除，所以蜂鸣器不再鸣叫。这个时候我们再按下按键 KEY_DOWN，任务 beep_task 被重新创建，所以蜂鸣器恢复鸣叫。



第五十九章 UCOSII 实验 2-信号量和邮箱

上一章，我们学习了如何使用 UCOSII，学习了 UCOSII 的任务调度，但是并没有用到任务间的同步与通信，本章我们将学习两个最基本的任务间通讯方式：信号量和邮箱。本章分为如下几个部分：

- 59.1 UCOSII 信号量和邮箱简介
- 59.2 硬件设计
- 59.3 软件设计
- 59.4 下载验证



59.1 UCOSII 信号量和邮箱简介

系统中的多个任务在运行时，经常需要互相无冲突地访问同一个共享资源，或者需要互相支持和依赖，甚至有时还要互相加以必要的限制和制约，才保证任务的顺利运行。因此，操作系统必须具有对任务的运行进行协调的能力，从而使任务之间可以无冲突、流畅地同步运行，而不致导致灾难性的后果。

例如，任务 A 和任务 B 共享一台打印机，如果系统已经把打印机分配给了任务 A，则任务 B 因不能获得打印机的使用权而应该处于等待状态，只有当任务 A 把打印机释放后，系统才能唤醒任务 B 使其获得打印机的使用权。如果这两个任务不这样做，那么会造成极大的混乱。

任务间的同步依赖于任务间的通信。在 UCOSII 中，是使用信号量、邮箱（消息邮箱）和消息队列这些被称作事件的中间环节来实现任务之间的通信的。本章，我们仅介绍信号量和邮箱，消息队列将会在下一章介绍。

事件

两个任务通过事件进行通讯的示意图如图 59.1.1 所示：



图 59.1.1 两个任务使用事件进行通信的示意图

在图 59.1.1 中任务 1 是发信方，任务 2 是收信方。任务 1 负责把信息发送到事件上，这项操作叫做发送事件。任务 2 通过读取事件操作对事件进行查询：如果有信息则读取，否则等待。读事件操作叫做请求事件。

为了把描述事件的数据结构统一起来，UCOSII 使用叫做事件控制块(ECB)的数据结构来描述诸如信号量、邮箱（消息邮箱）和消息队列这些事件。事件控制块中包含包括等待任务表在内的所有有关事件的数据，事件控制块结构体定义如下：

```

typedef struct
{
    INT8U OSEventType;          //事件的类型
    INT16U OSEventCnt;          //信号量计数器
    void *OSEventPtr;           //消息或消息队列的指针
    INT8U OSEventGrp;           //等待事件的任务组
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE];//任务等待表
#if OS_EVENT_NAME_EN > 0u
    INT8U *OSEventName;         //事件名
#endif
} OS_EVENT;
  
```

信号量

信号量是一类事件。使用信号量的最初目的，是为了给共享资源设立一个标志，该标志表示该共享资源的占用情况。这样，当一个任务在访问共享资源之前，就可以先对这个标志进行查询，从而在了解资源被占用的情况之后，再来决定自己的行为。

信号量可以分为两种：一种是二值型信号量，另外一种是 N 值信号量。

二值型信号量好比家里的座机，任何时候，只能有一个人占用。而 N 值信号量，则好比公



共电话亭，可以同时有多个人（N个）使用。

UCOSII 将二值型信号量称之为也叫互斥型信号量，将 N 值信号量称之为计数型信号量，也就是普通的信号量。本章，我们介绍的是普通信号量，互斥型信号量的介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》5.4 节。

接下来我们看看在 UCOSII 中，与信号量相关的几个函数（未全部列出，下同）。

1) 创建信号量函数

在使用信号量之前，我们必须用函数 OSSemCreate 来创建一个信号量，该函数的原型为：OS_EVENT *OSSemCreate (INT16U cnt)。该函数返回值为已创建的信号量的指针，而参数 cnt 则是信号量计数器 (OSEventCnt) 的初始值。

2) 请求信号量函数

任务通过调用函数 OSSemPend 请求信号量，该函数原型如下：void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)。其中，参数 pevent 是被请求信号量的指针，timeout 为等待时限，err 为错误信息。

为防止任务因得不到信号量而处于长期的等待状态，函数 OSSemPend 允许用参数 timeout 设置一个等待时间的限制，当任务等待的时间超过 timeout 时可以结束等待状态而进入就绪状态。如果参数 timeout 被设置为 0，则表明任务的等待时间为无限长。

3) 发送信号量函数

任务获得信号量，并在访问共享资源结束以后，必须要释放信号量，释放信号量也叫做发送信号量，发送信号通过 OSSemPost 函数实现。OSSemPost 函数在对信号量的计数器操作之前，首先要检查是否还有等待该信号量的任务。如果没有，就把信号量计数器 OSEventCnt 加一；如果有，则调用调度器 OS_Sched() 去运行等待任务中优先级别最高的任务。函数 OSSemPost 的原型为：INT8U OSSemPost(OS_EVENT *pevent)。其中，pevent 为信号量指针，该函数在调用成功后，返回值为 OS_ON_ERR，否则会根据具体错误返回 OS_ERR_EVENT_TYPE、OS_SEM_OVF。

4) 删除信号量函数

应用程序如果不需要某个信号量了，那么可以调用函数 OSSemDel 来删除该信号量，该函数的原型为：OS_EVENT *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err)。其中，pevent 为要删除的信号量指针，opt 为删除条件选项，err 为错误信息。

邮箱

在多任务操作系统中，常常需要在任务与任务之间通过传递一个数据（这种数据叫做“消息”）的方式来进行通信。为了达到这个目的，可以在内存中创建一个存储空间作为该数据的缓冲区。如果把这个缓冲区称之为消息缓冲区，这样在任务间传递数据（消息）的最简单办法就是传递消息缓冲区的指针。我们把用来传递消息缓冲区指针的数据结构叫做邮箱（消息邮箱）。

在 UCOSII 中，我们通过事件控制块的 OSEventPrt 来传递消息缓冲区指针，同时使事件控制块的成员 OSEventType 为常数 OS_EVENT_TYPE_MBOX，则该事件控制块就叫做消息邮箱。

接下来我们看看在 UCOSII 中，与消息邮箱相关的几个函数。

1) 创建邮箱函数

创建邮箱通过函数 OSMboxCreate 实现，该函数原型为：OS_EVENT *OSMboxCreate (void *msg)。函数中的参数 msg 为消息的指针，函数的返回值为消息邮箱的指针。

调用函数 OSMboxCreate 需先定义 msg 的初始值。在一般的情况下，这个初始值为 NULL；但也可以事先定义一个邮箱，然后把这个邮箱的指针作为参数传递到函数 OSMboxCreate 中，使之一开始就指向一个邮箱。

2) 向邮箱发送消息函数



任务可以通过调用函数 OSMboxPost 向消息邮箱发送消息，这个函数的原型为：INT8U OSMboxPost (OS_EVENT *pevent,void *msg)。其中 pevent 为消息邮箱的指针，msg 为消息指针。

3) 请求邮箱函数

当一个任务请求邮箱时需要调用函数 OSMboxPend，这个函数的主要作用就是查看邮箱指针 OSEventPtr 是否为 NULL，如果不是 NULL 就把邮箱中的消息指针返回给调用函数的任务，同时用 OS_NO_ERR 通过函数的参数 err 通知任务获取消息成功；如果邮箱指针 OSEventPtr 是 NULL，则使任务进入等待状态，并引发一次任务调度。

函数 OSMboxPend 的原型为：void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)。其中 pevent 为请求邮箱指针，timeout 为等待时限，err 为错误信息。

4) 查询邮箱状态函数

任务可以通过调用函数 OSMboxQuery 查询邮箱的当前状态。该函数原型为：INT8U OSMboxQuery(OS_EVENT *pevent,OS_MBOX_DATA *pdata)。其中 pevent 为消息邮箱指针，pdata 为存放邮箱信息的结构。

5) 删除邮箱函数

在邮箱不再使用的时候，我们可以通过调用函数 OSMboxDel 来删除一个邮箱，该函数原型为：OS_EVENT *OSMboxDel(OS_EVENT *pevent,INT8U opt,INT8U *err)。其中 pevent 为消息邮箱指针，opt 为删除选项，err 为错误信息。

关于 UCOSII 信号量和邮箱的介绍，就到这里。更详细的介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》第五章。

59.2 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 6 个任务：开始任务、LED 任务、触摸屏任务、蜂鸣器任务、按键扫描任务和主任务，开始任务用于创建信号量、创建邮箱、初始化统计任务以及其他任务的创建，之后挂起；LED 任务用于 DS0 控制，提示程序运行状况；蜂鸣器任务用于测试信号量，是请求信号量函数，每得到一个信号量，蜂鸣器就叫一次；触摸屏任务用于在屏幕上画图，可以用于测试 CPU 使用率；按键扫描任务用于按键扫描，优先级最高，将得到的键值通过消息邮箱发送出去；主任务则通过查询消息邮箱获得键值，并根据键值执行 DS1 控制、信号量发送（蜂鸣器控制）、触摸区域清屏和触摸屏校准等控制。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 、 DS1
- 2) 4 个按键 (KEY0/KEY1/KEY2/WK_UP)
- 3) 蜂鸣器
- 4) TFTLCD 模块

这些，我们在前面的学习中都已经介绍过了。

59.3 软件设计

本章，我们在第三十一章实验（实验 26）的基础上修改，具体方法同上一章一模一样，本章我们就不再详细介绍了，不过本章，我们将 OS_TICKS_PER_SEC 设置为 500，即 UCOSII 的时钟节拍为 2ms。

在加入 UCOSII 代码后，我们只需要修改 main.c 函数了，打开 main.c，输入如下代码：

```
//////////UCOSII 任务堆栈设置//////////
```



```
//START 任务
//设置任务优先级
#define START_TASK_PRIO          10 //开始任务的优先级设置为最低
//设置任务堆栈大小
#define START_STK_SIZE            64
//创建任务堆栈空间
OS_STK START_TASK_STK[START_STK_SIZE];
//任务函数接口
void start_task(void *pdata);

//LED 任务
//设置任务优先级
#define LED_TASK_PRIO              7
//设置任务堆栈大小
#define LED_STK_SIZE                64
//创建任务堆栈空间
OS_STK LED_TASK_STK[LED_STK_SIZE];
//任务函数接口
void led_task(void *pdata);

//触摸屏任务
//设置任务优先级
#define TOUCH_TASK_PRIO             6
//设置任务堆栈大小
#define TOUCH_STK_SIZE               64
//创建任务堆栈空间
OS_STK TOUCH_TASK_STK[TOUCH_STK_SIZE];
//任务函数接口
void touch_task(void *pdata);

//蜂鸣器任务
//设置任务优先级
#define BEEP_TASK_PRIO                 5
//设置任务堆栈大小
#define BEEP_STK_SIZE                  64
//创建任务堆栈空间
OS_STK BEEP_TASK_STK[BEEP_STK_SIZE];
//任务函数接口
void beep_task(void *pdata);

//主任务
//设置任务优先级
#define MAIN_TASK_PRIO                 4
```



```
//设置任务堆栈大小
#define MAIN_STK_SIZE           128
//创建任务堆栈空间
OS_STK MAIN_TASK_STK[MAIN_STK_SIZE];
//任务函数接口
void main_task(void *pdata);

//按键扫描任务
//设置任务优先级
#define KEY_TASK_PRIO            3
//设置任务堆栈大小
#define KEY_STK_SIZE              64
//创建任务堆栈空间
OS_STK KEY_TASK_STK[KEY_STK_SIZE];
//任务函数接口
void key_task(void *pdata);
///////////////////////////////
OS_EVENT * msg_key;          //按键邮箱事件块指针
OS_EVENT * sem_beep;         //蜂鸣器信号量指针
//加载主界面
void ucos_load_main_ui(void)
{
    LCD_Clear(WHITE); //清屏
    POINT_COLOR=RED; //设置字体为红色
    LCD_ShowString(30,10,200,16,16," WarShip STM32");
    LCD_ShowString(30,30,200,16,16,"UCOSII TEST2");
    LCD_ShowString(30,50,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,75,200,16,16,"KEY0:DS0 KEY_UP:ADJUST");
    LCD_ShowString(30,95,200,16,16,"KEY1:BEEP  KEY2:CLEAR");
    LCD_ShowString(80,210,200,16,16,"Touch Area");
    LCD_DrawLine(0,120,lcddev.width,120);
    LCD_DrawLine(0,70,lcddev.width,70);
    LCD_DrawLine(150,0,150,70);
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(160,30,200,16,16,"CPU:   %");
    LCD_ShowString(160,50,200,16,16,"SEM:000");
}

int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化
    LED_Init();             //初始化与 LED 连接的硬件接口
```



```
LCD_Init();           //初始化 LCD
BEEP_Init();          //蜂鸣器初始化
KEY_Init();           //按键初始化
tp_dev.init();
ucos_load_main_ui();
OSInit();             //初始化 UCOSII
OSTaskCreate(start_task,(void *)0,(OS_STK *)&START_TASK_STK
             [START_STK_SIZE-1],START_TASK_PRIO );//创建起始任务
OSStart();
}

//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    pdata = pdata;
    msg_key=OSMboxCreate((void*)0); //创建消息邮箱
    sem_beep=OSSemCreate(0);        //创建信号量
    OSStatInit();                  //初始化统计任务.这里会延时 1 秒钟左右
    OS_ENTER_CRITICAL();           //进入临界区(无法被中断打断)
    OSTaskCreate(led_task,(void *)0,(OS_STK*)&LED_TASK_STK[LED_STK_SIZE-1],
                LED_TASK_PRIO);
    OSTaskCreate(touch_task,(void *)0,(OS_STK*)&TOUCH_TASK_STK
                [TOUCH_STK_SIZE-1],TOUCH_TASK_PRIO);
    OSTaskCreate(beep_task,(void *)0,(OS_STK*)&BEEP_TASK_STK[BEEP_STK_SIZE-1],
                BEEP_TASK_PRIO);
    OSTaskCreate(main_task,(void *)0,(OS_STK*)&MAIN_TASK_STK[MAIN_STK_SIZE
                -1],MAIN_TASK_PRIO);
    OSTaskCreate(key_task,(void *)0,(OS_STK*)&KEY_TASK_STK[KEY_STK_SIZE-1],
                KEY_TASK_PRIO);
    OSTaskSuspend(START_TASK_PRIO);//挂起起始任务.
    OS_EXIT_CRITICAL();           //退出临界区(可以被中断打断)
}

//LED 任务
void led_task(void *pdata)
{
    u8 t;
    while(1)
    {
        t++; delay_ms(10);
        if(t==8)LED0=1;           //LED0 灭
        if(t==100) { t=0;LED0=0;} //LED0 亮
    }
}
```



```
//蜂鸣器任务
void beep_task(void *pdata)
{
    u8 err;
    while(1)
    {
        OSSemPend(sem_beep,0,&err);
        BEEP=1;delay_ms(60);
        BEEP=0;delay_ms(940);
    }
}

//触摸屏任务
void touch_task(void *pdata)
{
    while(1)
    {
        tp_dev.scan(0);
        if(tp_dev.sta&TP_PRES_DOWN)      //触摸屏被按下
        {
            if(tp_dev.x<lcddev.width&&tp_dev.y<lcddev.height&&tp_dev.y>120)
            {
                TP_Draw_Big_Point(tp_dev.x,tp_dev.y,RED);//画图
                delay_ms(2);
            }
            }else delay_ms(10);    //没有按键按下的时候
        }
}

//主任务
void main_task(void *pdata)
{
    u32 key=0;
    u8 err;
    u8 semmask=0;
    u8 tcnt=0;
    while(1)
    {
        key=(u32)OSMboxPend(msg_key,10,&err);
        switch(key)
        {
            case 1://控制 DS1
                LED1=!LED1;
                break;
            case 2://发送信号量
        }
    }
}
```



```
semmask=1;
OSSemPost(sem_beep);
break;
case 3://清除
LCD_Fill(0,121	lcddev.width, lcddev.height,WHITE);
break;
case 4://校准
OSTaskSuspend(TOUCH_TASK_PRIO); //挂起触摸屏任务
TP_Adjust();
OSTaskResume(TOUCH_TASK_PRIO); //解挂
ucos_load_main_ui(); //重新加载主界面
break;
}
if(semmask||sem_beep->OSEventCnt)//需要显示 sem
{
POINT_COLOR=BLUE;
LCD_ShowxNum(192,50,sem_beep->OSEventCnt,3,16,0X80);//显示信号量值
if(sem_beep->OSEventCnt==0)semmask=0;//停止更新
}
if(tcnt==50)//0.5 秒更新一次 CPU 使用率
{
tcnt=0;
POINT_COLOR=BLUE;
LCD_ShowxNum(192,30,OSCPUUsage,3,16,0); //显示 CPU 使用率
}
tcnt++;
delay_ms(10);
}
}
//按键扫描任务
void key_task(void *pdata)
{
u8 key;
while(1)
{
key=KEY_Scan(0);
if(key)OSMboxPost(msg_key,(void*)key);//发送消息
delay_ms(10);
}
}
```

该部分代码我们创建了 6 个任务：start_task、led_task、beep_task、touch_task、main_task 和 key_task，优先级分别是 10 和 7~3，堆栈大小除了 main_task 是 128，其他都是 64。

该程序的运行流程就比上一章复杂了一些，我们创建了消息邮箱 msg_key，用于按键任务



和主任务之间的数据传输（传递键值），另外创建了信号量 sem_beep，用于蜂鸣器任务和主任务之间的通信。

本代码中，我们使用了UCOSII提供的CPU统计任务，通过OSStatInit初始化CPU统计任务，然后在主任务中显示CPU使用率。

另外，在主任务中，我们用到了任务的挂起和恢复函数，在执行触摸屏校准的时候，我们必须先将触摸屏任务挂起，待校准完成之后，再恢复触摸屏任务。这是因为触摸屏校准和触摸屏任务都用到了触摸屏和TFTLCD，而这两个东西是不支持多个任务占用的，所以必须采用独占的方式使用，否则可能导致数据错乱。

软件设计部分就为大家介绍到这里。

59.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上，可以看到 LCD 显示界面如图 59.4.1 所示：



图 59.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率仅为 1%。此时通过在触摸区域画图，可以看到 CPU 使用率飙升(42%)，说明触摸屏任务是一个很占 CPU 的任务；通过按 KEY0，可以控制 DS1 的亮灭；通过按 KEY1 则可以控制蜂鸣器的发声（连续按下多次后，可以看到蜂鸣每隔 1 秒叫一次），同时，可以在 LCD 上面看到信号量的当前值；通过按 KEY2，可以清除触摸屏的输入；通过按 WK_UP 可以进入校准程序，进行触摸屏校准。



第六十章 UCOSII 实验 3-消息队列、信号量集和软件定时器

上一章，我们学习了 UCOSII 的信号量和邮箱的使用，本章，我们将学习消息队列、信号量集和软件定时器的使用。本章分为如下几个部分：

- 60.1 UCOSII 消息队列、信号量集和软件定时器简介
- 60.2 硬件设计
- 60.3 软件设计
- 60.4 下载验证



60.1 UCOSII 消息队列、信号量集和软件定时器简介

上一章，我们介绍了信号量和邮箱的使用，本章我们介绍比较复杂消息队列、信号量集以及软件定时器的使用。

消息队列

使用消息队列可以在任务之间传递多条消息。消息队列由三个部分组成：事件控制块、消息队列和消息。当把事件控制块成员 OSEventType 的值置为 OS_EVENT_TYPE_Q 时，该事件控制块描述的就是一个消息队列。

消息队列的数据结构如图 60.1.1 所示。从图中可以看到，消息队列相当于一个共用一个任务等待列表的消息邮箱数组，事件控制块成员 OSEventPtr 指向了一个叫做队列控制块 (OS_Q) 的结构，该结构管理了一个数组 MsgTbl[]，该数组中的元素都是一些指向消息的指针。

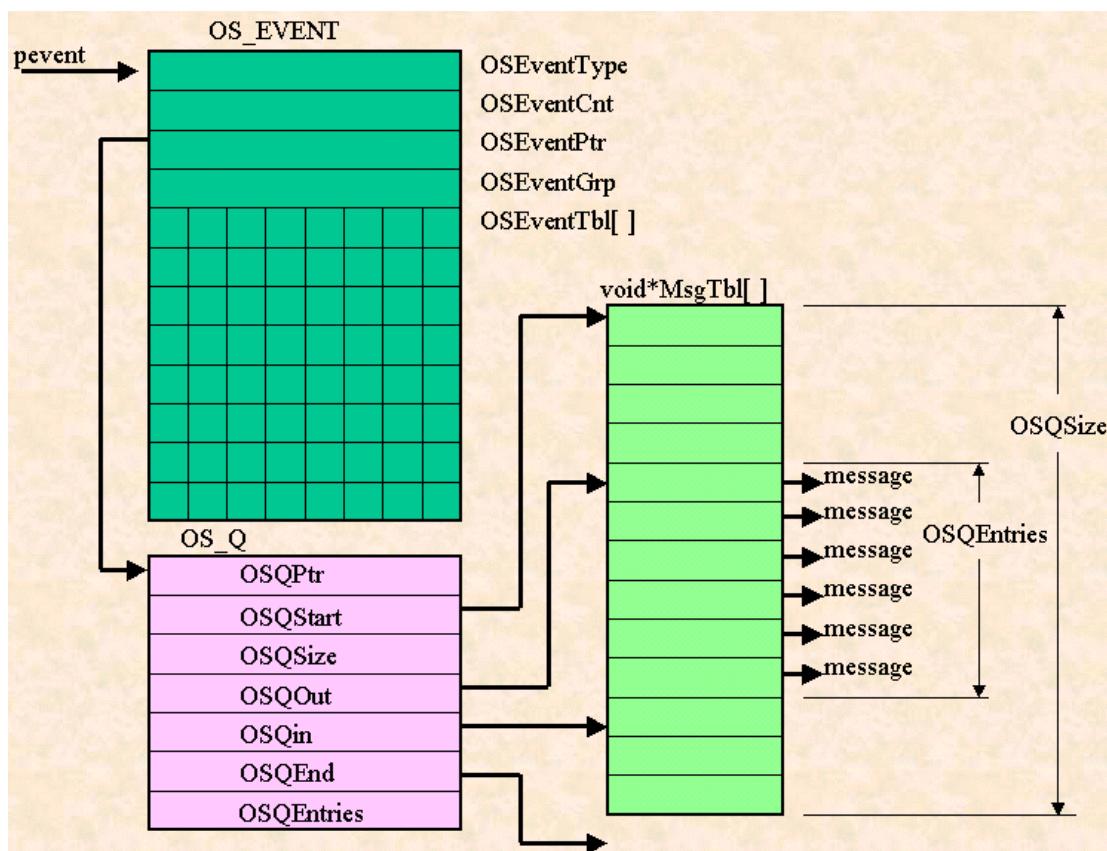


图 60.1.1 消息队列的数据结构

队列控制块 (OS_Q) 的结构定义如下：

```
typedef struct os_q
{
    struct os_q *OSQPtr;
    void **OSQStart;
    void **OSQEnd;
    void **OSQIn;
    void **OSQOut;
    INT16U OSQSize;
    INT16U OSQEntries;
```



} OS_Q;

该结构体中各参数的含义如表 60.1.1 所示:

参数	说明
OSQPtr	指向下一个空的队列控制块
OSQSize	数组的长度
OSQEntres	已存放消息指针的元素数目
OSQStart	指向消息指针数组的起始地址
OSQEnd	指向消息指针数组结束单元的下一个单元。它使得数组构成了一个循环的缓冲区
OSQIn	指向插入一条消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元
OSQOut	指向被取出消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元

表 60.1.1 队列控制块各参数含义

其中，可以移动的指针为 OSQIn 和 OSQOut，而指针 OSQStart 和 OSQEnd 只是一个标志（常指针）。当可移动的指针 OSQIn 或 OSQOut 移动到数组末尾，也就是与 OSQEnd 相等时，可移动的指针将会被调整到数组的起始位置 OSQStart。也就是说，从效果上来看，指针 OSQEnd 与 OSQStart 等值。于是，这个由消息指针构成的数组就头尾衔接起来形成了一个如图 60.1.2 所示的循环的队列。

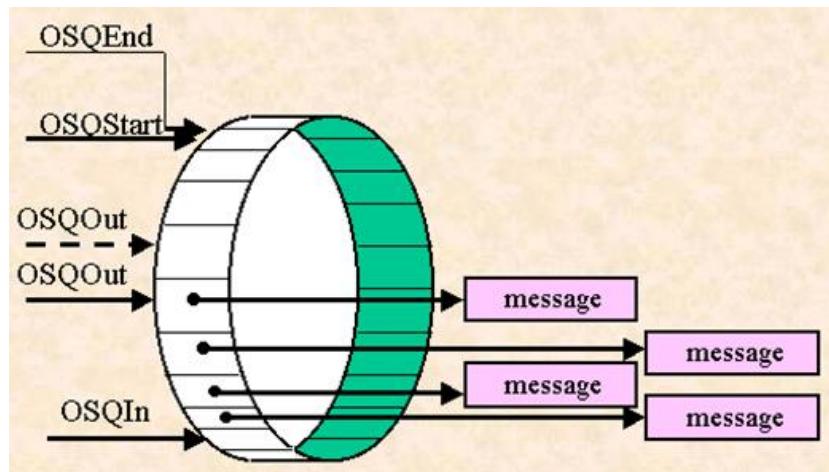


图 60.1.2 消息指针数组构成的环形数据缓冲区

在 UCOSII 初始化时，系统将按文件 os_cfg.h 中的配置常数 OS_MAX_QS 定义 OS_MAX_QS 个队列控制块，并用队列控制块中的指针 OSQPtr 将所有队列控制块链接为链表。由于这时还没有使用它们，故这个链表叫做空队列控制块链表。

接下来我们看看在 UCOSII 中，与消息队列相关的几个函数（未全部列出，下同）。

1) 创建消息队列函数

创建一个消息队列首先需要定义一指针数组，然后把各个消息数据缓冲区的首地址存入这个数组中，然后再调用函数 OSQCreate 来创建消息队列。创建消息队列函数 OSQCreate 的原型为：OS_EVENT *OSQCreate(void**start, INT16U size)。其中，start 为存放消息缓冲区指针数组的地址，size 为该数组大小。该函数的返回值为消息队列指针。

2) 请求消息队列函数



请求消息队列的目的是为了从消息队列中获取消息。任务请求消息队列需要调用函数 OSQPend，该函数原型为：void*OSQPend(OS_EVENT*pEvent, INT16U timeout, INT8U *err)。

其中，pEvent 为所请求的消息队列的指针，timeout 为任务等待时限，err 为错误信息。

3) 向消息队列发送消息函数

任务可以通过调用函数 OSQPost 或 OSQPostFront 两个函数来向消息队列发送消息。函数 OSQPost 以 FIFO（先进先出）的方式组织消息队列，函数 OSQPostFront 以 LIFO（后进先出）的方式组织消息队列。这两个函数的原型分别为：INT8U OSQPost(OS_EVENT *pEvent,void *msg) 和 INT8U OSQPost(OS_EVENT*pEvent,void*msg)。

其中，pEvent 为消息队列的指针，msg 为待发消息的指针。

消息队列还有其他一些函数，这里我们就不介绍了，感兴趣的朋友可以参考《嵌入式实时操作系统 UCOSII 原理及应用》第五章，关于队列更详细的介绍，也请参考该书。

信号量集

在实际应用中，任务常常需要与多个事件同步，即要根据多个信号量组合作用的结果来决定任务的运行方式。UCOSII 为了实现多个信号量组合的功能定义了一种特殊的数据结构——信号量集。

信号量集所能管理的信号量都是一些二值信号，所有信号量集实质上是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如图 60.1.3 所示

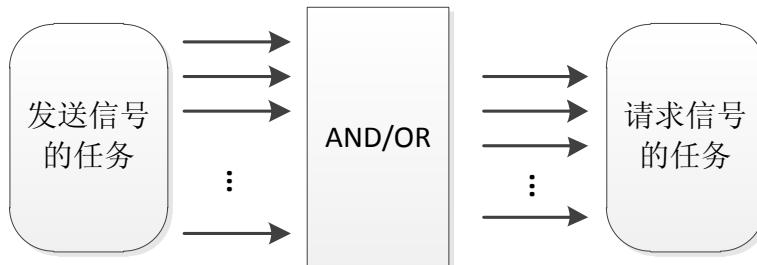


图 60.1.3 信号量集示意图

不同于信号量、消息邮箱、消息队列等事件，UCOSII 不使用事件控制块来描述信号量集，而使用了一个叫做标志组的结构 OS_FLAG_GRP 来描述。OS_FLAG_GRP 结构如下：

```

typedef struct
{
    INT8U    OSFlagType;      //识别是否为信号量集的标志
    void *OSFlagWaitList;    //指向等待任务链表的指针
    OS_FLAGS OSFlagFlags;   //所有信号列表
}OS_FLAG_GRP;

```

成员 OSFlagWaitList 是一个指针，当一个信号量集被创建后，这个指针指向了这个信号量集的等待任务链表。

与其他前面介绍过的事件不同，信号量集用一个双向链表来组织等待任务，每一个等待任务都是该链表中的一个节点（Node）。标志组 OS_FLAG_GRP 的成员 OSFlagWaitList 就指向了信号量集的这个等待任务链表。等待任务链表节点 OS_FLAG_NODE 的结构如下：

```

typedef struct
{
    void    *OSFlagNodeNext;    //指向下一个节点的指针
    void    *OSFlagNodePrev;    //指向前一个节点的指针
    void *OSFlagNodeTCB;       //指向对应任务控制块的指针
}

```



```

void *OSFlagNodeFlagGrp;      //反向指向信号量集的指针
OS_FLAGS OSFlagNodeFlags;    //信号过滤器
INT8U OSFlagNodeWaitType;   //定义逻辑运算关系的数据
} OS_FLAG_NODE;

```

其中 OSFlagNodeWaitType 是定义逻辑运算关系的一个常数（根据需要设置），其可选值和对应的逻辑关系如表 60.1.2 所示：

常数	信号有效状态	等待任务的就绪条件
WAIT_CLR_ALL 或 WAIT_CLR_AND	0	信号全部有效（全 0）
WAIT_CLR_ANY 或 WAIT_CLR_OR	0	信号有一个或一个以上有效（有 0）
WAIT_SET_ALL 或 WAIT_SET_AND	1	信号全部有效（全 1）
WAIT_SET_ANY 或 WAIT_SET_OR	1	信号有一个或一个以上有效（有 1）

表 60.1.2 OSFlagNodeWaitType 可选值及其意义

OSFlagFlags、OSFlagNodeFlags、OSFlagNodeWaitType 三者的关系如图 60.1.4 所示：

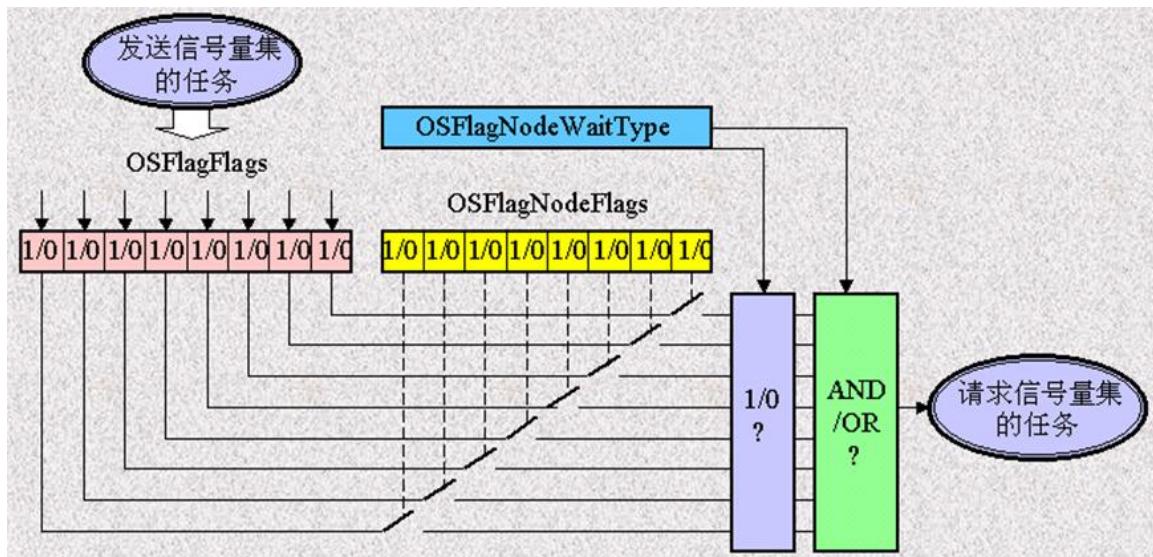


图 60.1.4 标志组与等待任务共同完成信号量集的逻辑运算及控制

图中为了方便说明，我们将 OSFlagFlags 定义为 8 位，但是 UCOSII 支持 8 位/16 位/32 位定义，这个通过修改 OS_FLAGS 的类型来确定（UCOSII 默认设置 OS_FLAGS 为 16 位）。

上图清楚的表达了信号量集各成员的关系：OSFlagFlags 为信号量表，通过发送信号量集的任务设置；OSFlagNodeFlags 为信号滤波器，由请求信号量集的任务设置，用于选择性的挑选 OSFlagFlags 中的部分（或全部）位作为有效信号；OSFlagNodeWaitType 定义有效信号的逻辑运算关系，也是由请求信号量集的任务设置，用于选择有效信号的组合方式（0/1? 与/或？）。

举个简单的例子，假设请求信号量集的任务设置 OSFlagNodeFlags 的值为 0X0F，设置 OSFlagNodeWaitType 的值为 WAIT_SET_ANY，那么只要 OSFlagFlags 的低四位的任何一位为 1，请求信号量集的任务将得到有效的请求，从而执行相关操作，如果低四位都为 0，那么请求信号量集的任务将得到无效的请求。



接下来我们看看在 UCOSII 中，与信号量集相关的几个函数。

1) 创建信号量集函数

任务可以通过调用函数 OSFlagCreate 来创建一个信号量集。函数 OSFlagCreate 的原型为：OS_FLAG_GRP *OSFlagCreate (OS_FLAGS flags, INT8U *err)。其中，flags 为信号量的初始值（即 OSFlagFlags 的值），err 为错误信息，返回值为该信号量集的标志组的指针，应用程序根据这个指针对信号量集进行相应的操作。

2) 请求信号量集函数

任务可以通过调用函数 OSFlagPend 请求一个信号量集，函数 OSFlagPend 的原型为：OS_FLAGS OSFlagPend(OS_FLAG_GRP*pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err)。其中，pgrp 为所请求的信号量集指针，flags 为滤波器（即 OSFlagNodeFlags 的值），wait_type 为逻辑运算类型（即 OSFlagNodeWaitType 的值），timeout 为等待时限，err 为错误信息。

3) 向信号量集发送信号函数

任务可以通过调用函数 OSFlagPost 向信号量集发信号，函数 OSFlagPost 的原型为：OS_FLAGS OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)。其中，pgrp 为所请求的信号量集指针，flags 为选择所要发送的信号，opt 为信号有效选项，err 为错误信息。

所谓任务向信号量集发信号，就是对信号量集标志组中的信号进行置“1”（置位）或置“0”（复位）的操作。至于对信号量集中的哪些信号进行操作，用函数中的参数 flags 来指定；对指定的信号是置“1”还是置“0”，用函数中的参数 opt 来指定（opt = OS_FLAG_SET 为置“1”操作；opt = OS_FLAG_CLR 为置“0”操作）。

信号量集就介绍到这，更详细的介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》第六章。

软件定时器

UCOSII 从 V2.83 版本以后，加入了软件定时器，这使得 UCOSII 的功能更加完善，在其上的应用程序开发与移植也更加方便。在实时操作系统中一个好的软件定时器实现要求有较高的精度、较小的处理器开销，且占用较少的存储器资源。

通过前面的学习，我们知道 UCOSII 通过 OSTimTick 函数对时钟节拍进行加 1 操作，同时遍历任务控制块，以判断任务延时是否到时。软件定时器同样由 OSTimTick 提供时钟，但是软件定时器的时钟还受 OS_TMR_CFG_TICKS_PER_SEC 设置的控制，也就是在 UCOSII 的时钟节拍上面再做了一次“分频”，软件定时器的最快时钟节拍就等于 UCOSII 的系统时钟节拍。这也决定了软件定时器的精度。

软件定时器定义了一个单独的计数器 OSTmrTime，用于软件定时器的计时，UCOSII 并不在 OSTimTick 中进行软件定时器的到时判断与处理，而是创建了一个高于应用程序中所有其他任务优先级的定时器管理任务 OSTmr_Task，在这个任务中进行定时器的到时判断和处理。时钟节拍函数通过信号量给这个高优先级任务发信号。这种方法缩短了中断服务程序的执行时间，但也使得定时器到时处理函数的响应受到中断退出时恢复现场和任务切换的影响。软件定时器功能实现代码存放在 tmr.c 文件中，移植时需只需在 os_cfg.h 文件中使能定时器和设定定时器的相关参数。

UCOSII 中软件定时器的实现方法是，将定时器按定时时间分组，使得每次时钟节拍到来时只对部分定时器进行比较操作，缩短了每次处理的时间。但这就需要动态地维护一个定时器组。定时器组的维护只是在每次定时器到时时才发生，而且定时器从组中移除和再插入操作不需要排序。这是一种比较高效的算法，减少了维护所需的操作时间。



UCOSII 软件定时器实现了 3 类链表的维护:

```
OS_EXT OS_TMR OSTmrTbl[OS_TMR_CFG_MAX]; //定时器控制块数组
OS_EXT OS_TMR *OSTmrFreeList; //空闲定时器控制块链表指针
OS_EXT OS_TMR_WHEEL OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]; //定时器轮
```

其中 OS_TMR 为定时器控制块, 定时器控制块是软件定时器管理的基本单元, 包含软件定时器的名称、定时时间、在链表中的位置、使用状态、使用方式, 以及到时回调函数及其参数等基本信息。

OSTmrTbl[OS_TMR_CFG_MAX]: 以数组的形式静态分配定时器控制块所需的 RAM 空间, 并存储所有已建立的定时器控制块, OS_TMR_CFG_MAX 为最大软件定时器的个数。

OSTmrFreeLiSt: 为空闲定时器控制块链表头指针。空闲态的定时器控制块(OS_TMR)中, OSTmrnext 和 OSTmrPrev 两个指针分别指向空闲控制块的前一个和后一个, 组织了空闲控制块双向链表。建立定时器时, 从这个链表中搜索空闲定时器控制块。

OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]: 该数组的每个元素都是已开启定时器的一个分组, 元素中记录了指向该分组中第一个定时器控制块的指针, 以及定时器控制块的个数。运行态的定时器控制块(OS_TMR)中, OSTmrnext 和 OSTmrPrev 两个指针同样也组织了所在分组中定时器控制块的双向链表。软件定时器管理所需的数据结构示意图如图 60.1.5 所示:

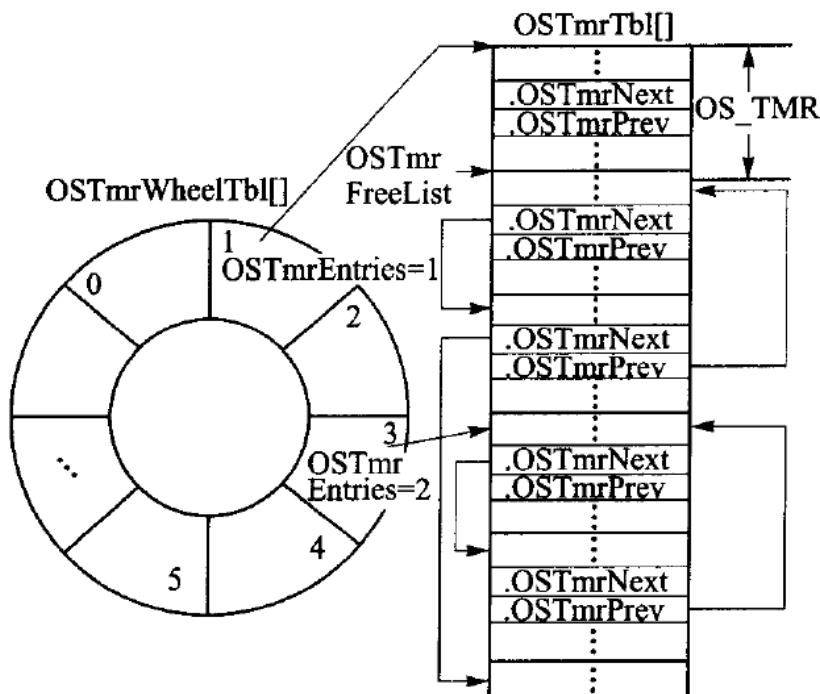


图 60.1.5 软件定时器管理所需的数据结构示意图

OS_TMR_CFG_WHEEL_SIZE 定义了 OSTmrWheelTbl 的大小, 同时这个值也是定时器分组的依据。按照定时器到时值与 OS_TMR_CFG_WHEEL_SIZE 相除的余数进行分组: 不同余数的定时器放在不同分组中; 相同余数的定时器处在同一组中, 由双向链表连接。这样, 余数值为 $0 \sim OS_TMR_CFG_WHEEL_SIZE - 1$ 的不同定时器控制块, 正好分别对应了数组元素 OSTmr-WheelTbl[0]~OSTmrWheelTbl[OS_TMR_CFGWHEEL_SIZE-1]的不同分组。每次时钟节拍到来时, 时钟数 OSTmrTime 值加 1, 然后也进行求余操作, 只有余数相同的那组定时器才有可能到时, 所以只对该组定时器进行判断。这种方法比循环判断所有定时器更高效。随着时钟数的累加, 处理的分组也由 $0 \sim OS_TMR_CFG_WHE_EL_SIZE - 1$ 循环。这里, 我们推荐



OS_TMR_CFG_WHEEL_SIZE 的取值为 2 的 N 次方，以便采用移位操作计算余数，缩短处理时间。

信号量唤醒定时器管理任务，计算出当前所要处理的分组后，程序遍历该分组中的所有控制块，将当前 OSTmrTime 值与定时器控制块中的到时值（OSTmrMatch）相比较。若相等(即到时)，则调用该定时器到时回调函数；若不相等，则判断该组中下一个定时器控制块。如此操作，直到该分组链表的结尾。软件定时器管理任务的流程如图 60.1.6 所示。

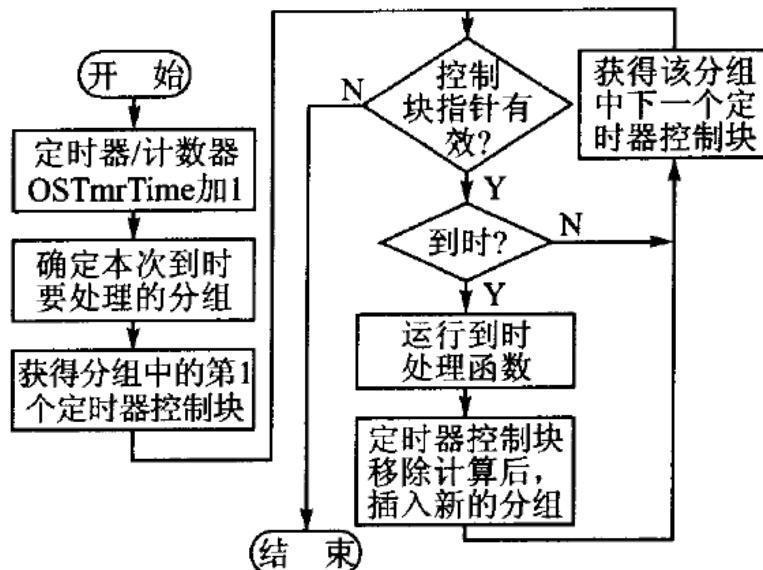


图 60.1.6 软件定时器管理任务流程

当运行完软件定时器的到时处理函数之后，需要进行该定时器控制块在链表中的移除和再插入操作。插入前需要重新计算定时器下次到时时所处的分组。计算公式如下：

定时器下次到时的 OSTmrTime 值(OSTmrMatch)=定时器定时值+当前 OSTmrTime 值
新分组=定时器下次到时的 OSTmrTime 值(OSTmrMatch)%OS_TMR_CFG_WHEEL_SIZE
接下来我们看看在 UCOSII 中，与软件定时器相关的几个函数。

1) 创建软件定时器函数

创建软件定时器通过函数 OSTmrCreate 实现，该函数原型为：OS_TMR *OSTmrCreate (INT32U dly, INT32U period, INT8U opt, OS_TMR_CALLBACK callback,void *callback_arg, INT8U *pname, INT8U *perr)。

dly，用于初始化定时时间，对单次定时（ONE-SHOT 模式）的软件定时器来说，这就是该定时器的定时时间，而对于周期定时（PERIODIC 模式）的软件定时器来说，这是该定时器第一次定时的时间，从第二次开始定时时间变为 period。

period，在周期定时（PERIODIC 模式），该值为软件定时器的周期溢出时间。

opt，用于设置软件定时器工作模式。可以设置的值为：OS_TMR_OPT_ONE_SHOT 或 OS_TMR_OPT_PERIODIC，如果设置为前者，说明是一个单次定时器；设置为后者则表示是周期定时器。

callback，为软件定时器的回调函数，当软件定时器的定时时间到达时，会调用该函数。

callback_arg，回调函数的参数。

pname，为软件定时器的名字。

perr，为错误信息。

软件定时器的回调函数有固定的格式，我们必须按照这个格式编写，软件定时器的回

调函数格式为：void (*OS_TMR_CALLBACK)(void *ptmr, void *parg)。其中，函数名我们可以自己随意设置，而 ptmr 这个参数，软件定时器用来传递当前定时器的控制块指针，所以我们一般设置其类型为 OS_TMR*类型，第二个参数（parg）为回调函数的参数，这个就可以根据自己需要设置了，你也可以不用，但是必须有这个参数。

2) 开启软件定时器函数

任务可以通过调用函数 OSTmrStart 开启某个软件定时器，该函数的原型为：BOOLEAN OSTmrStart (OS_TMR *ptmr, INT8U *perr)。其中 ptmr 为要开启的软件定时器指针，perr 为错误信息。

3) 停止软件定时器函数

任务可以通过调用函数 OSTmrStop 停止某个软件定时器，该函数的原型为：BOOLEAN OSTmrStop (OS_TMR *ptmr, INT8U opt, void *callback_arg, INT8U *perr)。

其中 ptmr 为要停止的软件定时器指针。

opt 为停止选项，可以设置的值及其对应的意义为：

OS_TMR_OPT_NONE，直接停止，不做任何其他处理

OS_TMR_OPT_CALLBACK，停止，用初始化的参数执行一次回调函数

OS_TMR_OPT_CALLBACK_ARG，停止，用新的参数执行一次回调函数

callback_arg，新的回调函数参数。

perr，错误信息。

软件定时器我们就介绍到这。

60.2 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 7 个任务：开始任务、LED 任务、触摸屏任务、队列消息显示任务、信号量集任务、按键扫描任务和主任务，开始任务用于创建邮箱、消息队列、信号量集以及其他任务，之后挂起；触摸屏任务用于在屏幕上画图，测试 CPU 使用率；队列消息显示任务请求消息队列，在得到消息后显示收到的消息数据；信号量集任务用于测试信号量集，采用 OS_FLAG_WAIT_SET_ANY 的方法，任何按键按下（包括 TPAD），该任务都会控制蜂鸣器发出“滴”的一声；按键扫描任务用于按键扫描，优先级最高，将得到的键值通过消息邮箱发送出去；主任务创建 3 个软件定时器（定时器 1，100ms 溢出一次，显示 CPU 和内存使用率；定时 2，200ms 溢出一次，在固定区域不停的显示不同颜色；定时 3，,100ms 溢出一次，用于自动发送消息到消息队列），并通过查询消息邮箱获得键值，根据键值执行 DS1 控制、控制软件定时器 3 的开关、触摸区域清屏、触摸屏校和软件定时器 2 的开关控制等。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 、 DS1
- 2) 4 个机械按键（KEY0/KEY1/KEY2/WK_UP）
- 3) TPAD 触摸按键
- 4) 蜂鸣器
- 5) TFTLCD 模块

这些，我们在前面的学习中都已经介绍过了。

60.3 软件设计

本章，我们在第四十三章实验（实验 38）的基础上修改，首先，是 UCOSII 代码的添加，



具体方法同第五十九章一模一样，本章就不再详细介绍。本章 OS_TICKS_PER_SEC 的设置还是为 500，即 UCOSII 的时钟节拍为 2ms。另外由于我们创建了 7 个任务，加上统计任务、空闲任务和软件定时器任务，总共 10 个任务，如果你还想添加其他任务，请把 OS_MAX_TASKS 的值适当改大。

另外，我们还需要在 os_cfg.h 里面修改软件定时器管理部分的宏定义，修改如下：

```
#define OS_TMR_EN           1u      //使能软件定时器功能
#define OS_TMR_CFG_MAX        16u     //最大软件定时器个数
#define OS_TMR_CFG_NAME_EN    1u      //使能软件定时器命名
#define OS_TMR_CFG_WHEEL_SIZE 8u      //软件定时器轮大小
#define OS_TMR_CFG_TICKS_PER_SEC 100u   //软件定时器的时钟节拍 (10ms)
#define OS_TASK_TMR_PRIO       0u      //软件定时器的优先级,设置为最高
```

这样我们就使能 UCOSII 的软件定时器功能了，并且设置最大软件定时器个数为 16，定时器轮大小为 8，软件定时器时钟节拍为 10ms（即定时器的最少溢出时间为 10ms）。

最后，我们只需要修改 main.c 函数了，打开 main.c，输入如下代码：

```
//////////UCOSII 任务设置//////////
//START 任务
#define START_TASK_PRIO          10 //设置任务优先级
#define START_STK_SIZE            64 //设置任务堆栈大小
OS_STK START_TASK_STK[START_STK_SIZE]; //任务堆栈
void start_task(void *pdata);           //任务函数

//LED 任务
#define LED_TASK_PRIO             7 //设置任务优先级
#define LED_STK_SIZE               64 //设置任务堆栈大小
OS_STK LED_TASK_STK[LED_STK_SIZE]; //任务堆栈
void led_task(void *pdata);           //任务函数

//触摸屏任务
#define TOUCH_TASK_PRIO            6 //设置任务优先级
#define TOUCH_STK_SIZE              64 //设置任务堆栈大小
OS_STK TOUCH_TASK_STK[TOUCH_STK_SIZE]; //任务堆栈
void touch_task(void *pdata);         //任务函数

//队列消息显示任务
#define QMSGSHOW_TASK_PRIO          5 //设置任务优先级
#define QMSGSHOW_STK_SIZE            64 //设置任务堆栈大小
OS_STK QMSGSHOW_TASK_STK[QMSGSHOW_STK_SIZE]; //任务堆栈
void qmsgshow_task(void *pdata);       //任务函数

//主任务
#define MAIN_TASK_PRIO              4 //设置任务优先级
#define MAIN_STK_SIZE                128 //设置任务堆栈大小
OS_STK MAIN_TASK_STK[MAIN_STK_SIZE]; //任务堆栈
```



```
void main_task(void *pdata); //任务函数
///////////////////////////////
//信号量集任务
#define FLAGS_TASK_PRIO 3 //设置任务优先级
#define FLAGS_STK_SIZE 64 //设置任务堆栈大小
OS_STK FLAGS_TASK_STK[FLAGS_STK_SIZE]; //任务堆栈
void flags_task(void *pdata); //任务函数

//按键扫描任务
#define KEY_TASK_PRIO 2 //设置任务优先级
#define KEY_STK_SIZE 64 //设置任务堆栈大小
OS_STK KEY_TASK_STK[KEY_STK_SIZE]; //任务堆栈
void key_task(void *pdata); //任务函数

OS_EVENT * msg_key; //按键邮箱事件块
OS_EVENT * q_msg; //消息队列
OS_TMR * tmr1; //软件定时器 1
OS_TMR * tmr2; //软件定时器 2
OS_TMR * tmr3; //软件定时器 3
OS_FLAG_GRP * flags_key;//按键信号量集
void * MsgGrp[256]; //消息队列存储地址,最大支持 256 个消息
//软件定时器 1 的回调函数
//每 100ms 执行一次,用于显示 CPU 使用率和内存使用率
void tmr1_callback(OS_TMR *ptmr,void *p_arg)
{
    static u16 cpuusage=0; static u8 tcnt=0;
    POINT_COLOR=BLUE;
    if(tcnt==5)
    {
        LCD_ShowxNum(182,10,cpuusage/5,3,16,0); //显示 CPU 使用率
        cpuusage=0; tcnt=0;
    }
    cpuusage+=OSCPUUsage; tcnt++;
    LCD_ShowxNum(182,30,mem_perused(SRAMIN),3,16,0); //显示内存使用率
    LCD_ShowxNum(182,50,((OS_Q*)(q_msg->OSEventPtr))->OSQEntries,3,16,0X80);
    //显示队列当前的大小
}
//软件定时器 2 的回调函数
void tmr2_callback(OS_TMR *ptmr,void *p_arg)
{
    static u8 sta=0;
    switch(sta)
    {
```



```
case 0: LCD_Fill(121,221,lcddev.width,lcddev.height,RED); break;
case 1: LCD_Fill(121,221,lcddev.width,lcddev.height,GREEN); break;
case 2: LCD_Fill(121,221,lcddev.width,lcddev.height,BLUE); break;
case 3: LCD_Fill(121,221,lcddev.width,lcddev.height,MAGENTA); break;
case 4: LCD_Fill(121,221,lcddev.width,lcddev.height,GBLUE); break;
case 5:LCD_Fill(121,221,lcddev.width,lcddev.height,YELLOW); break;
case 6: LCD_Fill(121,221,lcddev.width,lcddev.height,BRRED); break;
}
sta++; if(sta>6)sta=0;
}

//软件定时器 3 的回调函数
void tmr3_callback(OS_TMR *ptmr,void *p_arg)
{
u8* p; u8 err;
static u8 msg_cnt=0; //msg 编号
p=mymalloc(SRAMIN,13); //申请 13 个字节的内存
if(p)
{
sprintf((char*)p,"ALIENTEK %03d",msg_cnt);
msg_cnt++;
err=OSQPost(q_msg,p); //发送队列
if(err!=OS_ERR_NONE) //发送失败
{
myfree(SRAMIN,p); //释放内存
OSTmrStop(tmr3,OS_TMR_OPT_NONE,0,&err); //关闭软件定时器 3
}
}
}

//加载主界面
void ucos_load_main_ui(void)
{
LCD_Clear(WHITE); //清屏
POINT_COLOR=RED; //设置字体为红色
LCD_ShowString(10,10,200,16,16,"WarShip STM32");
LCD_ShowString(10,30,200,16,16,"UCOSII TEST3");
LCD_ShowString(10,50,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(10,75,240,16,16,"TPAD:TMR2 SW KEY_UP:ADJUST");
LCD_ShowString(10,95,240,16,16,"KEY0:DS0 KEY1:Q SW KEY2:CLR");
LCD_DrawLine(0,70,lcddev.width,70);
LCD_DrawLine(120,0,120,70);
LCD_DrawLine(0,120,lcddev.width,120);
LCD_DrawLine(0,220,lcddev.width,220);
LCD_DrawLine(120,120,120,lcddev.height);
```



```
LCD_ShowString(5,125,240,16,16,"QUEUE MSG");//队列消息
LCD_ShowString(5,150,240,16,16,"Message:");
LCD_ShowString(5+120,125,240,16,16,"FLAGS");//信号量集
LCD_ShowString(5,225,240,16,16,"TOUCH"); //触摸屏
LCD_ShowString(5+120,225,240,16,16,"TMR2"); //队列消息
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(150,10,200,16,16,"CPU:    %");
LCD_ShowString(150,30,200,16,16,"MEM:    %");
LCD_ShowString(150,50,200,16,16," Q :000");
delay_ms(300);
}

int main(void)
{
    delay_init();           //延时函数初始化
    NVIC_Configuration(); //设置 NVIC 中断分组 2:2 位抢占优先级, 2 位响应优先级
    uart_init(9600);        //串口初始化波特率为 9600
    LED_Init();             //初始化与 LED 连接的硬件接口
    LCD_Init();             //初始化 LCD
    BEEP_Init();            //蜂鸣器初始化
    KEY_Init();             //按键初始化
    TPAD_Init(72);          //初始化 TPAD
    FSMC_SRAM_Init();       //初始化外部 SRAM
    mem_init(SRAMIN);       //初始化内部内存池
    mem_init(SRAMEX);       //初始化外部内存池
    tp_dev.init();
    ucos_load_main_ui();   OSInit();      //初始化 UCOSII
    OSTaskCreate(start_task,(void *)0,(OS_STK *)&START_TASK_STK[START_STK_SIZE
    -1],START_TASK_PRIO );//创建起始任务
    OSStart();
}

//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0; u8 err;
    pdata = pdata;
    msg_key=OSMboxCreate((void*)0); //创建消息邮箱
    q_msg=OSQCreate(&MsgGrp[0],256); //创建消息队列
    flags_key=OSFlagCreate(0,&err); //创建信号量集
    OSStatInit();                  //初始化统计任务.这里会延时 1 秒钟左右
    OS_ENTER_CRITICAL();           //进入临界区(无法被中断打断)
    OSTaskCreate(led_task,(void *)0,(OS_STK*)&LED_TASK_STK[LED_STK_SIZE-1],
    LED_TASK_PRIO);
    OSTaskCreate(touch_task,(void *)0,(OS_STK*)&TOUCH_TASK_STK
```



```
[TOUCH_STK_SIZE-1],TOUCH_TASK_PRIO);
OSTaskCreate(qmsgshow_task,(void *)0,(OS_STK*)&QMSGSHOW_TASK_STK
[QMSGSHOW_STK_SIZE-1],QMSGSHOW_TASK_PRIO);
OSTaskCreate(main_task,(void *)0,(OS_STK*)&MAIN_TASK_STK[MAIN_STK_SIZE
-1],MAIN_TASK_PRIO);
OSTaskCreate(flags_task,(void *)0,(OS_STK*)&FLAGS_TASK_STK
[FLAGS_STK_SIZE-1],FLAGS_TASK_PRIO);
OSTaskCreate(key_task,(void *)0,(OS_STK*)&KEY_TASK_STK[KEY_STK_SIZE-1]
,KEY_TASK_PRIO);
OSTaskSuspend(START_TASK_PRIO); //挂起起始任务.
OS_EXIT_CRITICAL(); //退出临界区(可以被中断打断)
}
//LED 任务
void led_task(void *pdata)
{
    u8 t;
    while(1)
    {
        t++; delay_ms(10);
        if(t==8)LED0=1; //LED0 灭
        if(t==100) {t=0; LED0=0;} //LED0 亮
    }
}
//触摸屏任务
void touch_task(void *pdata)
{
    while(1)
    {
        tp_dev.scan(0);
        if(tp_dev.sta&&TP_PRES_DOWN) //触摸屏被按下
        {
            if(tp_dev.x<120&&tp_dev.y<lcddev.height&&tp_dev.y>220)
            {
                TP_Draw_Big_Point(tp_dev.x,tp_dev.y,BLUE); //画图
                delay_ms(2);
            }
        }else delay_ms(10); //没有按键按下的时候
    }
}
//队列消息显示任务
void qmsgshow_task(void *pdata)
{
    u8 *p; u8 err;
```



```
while(1)
{
    p=OSQPend(q_msg,0,&err);//请求消息队列
    LCD_ShowString(5,170,240,16,16,p);//显示消息
    myfree(SRAMIN,p); delay_ms(500);
}

//主任务
void main_task(void *pdata)
{
    u32 key=0; u8 err;
    u8 tmr2sta=1; //软件定时器 2 开关状态
    u8 tmr3sta=0; //软件定时器 3 开关状态
    u8 flagsclrt=0;//信号量集显示清零倒计时
    tmr1=OSTmrCreate(10,10,OS_TMR_OPT_PERIODIC,
        (OS_TMR_CALLBACK)tmr1_callback,0,"tmr1",&err);           //100ms 执行一次
    tmr2=OSTmrCreate(10,20,OS_TMR_OPT_PERIODIC,
        (OS_TMR_CALLBACK)tmr2_callback,0,"tmr2",&err);           //200ms 执行一次
    tmr3=OSTmrCreate(10,10,OS_TMR_OPT_PERIODIC,
        (OS_TMR_CALLBACK)tmr3_callback,0,"tmr3",&err);           //100ms 执行一次
    OSTmrStart(tmr1,&err);                                //启动软件定时器 1
    OSTmrStart(tmr2,&err);                                //启动软件定时器 2
    while(1)
    {
        key=(u32)OSMboxPend(msg_key,10,&err);
        if(key)
        {
            flagsclrt=51;//500ms 后清除
            OSFlagPost(flags_key,1<<(key-1),OS_FLAG_SET,&err);//设置信号量为 1
        }
        if(flagsclrt)//倒计时
        {
            flagsclrt--;
            if(flagsclrt==1)LCD_Fill(140,162,239,162+16,WHITE);//清除显示
        }
        switch(key)
        {
            case 1: LED1=!LED1; break;//控制 DS1
            case 2://控制软件定时器 3
                tmr3sta=!tmr3sta;
                if(tmr3sta)OSTmrStart(tmr3,&err);
                else OSTmrStop(tmr3,OS_TMR_OPT_NONE,0,&err);//关闭软件定时器 3
                break;
        }
    }
}
```



```
case 3: LCD_Fill(0,221,119	lcddev.height,WHITE); break;//清除
case 4://校准
    OSTaskSuspend(TOUCH_TASK_PRIO);           //挂起触摸屏任务
    OSTaskSuspend(QMSGSHOW_TASK_PRIO); //挂起队列信息显示任务
    OSTmrStop(tmr1,OS_TMR_OPT_NONE,0,&err); //关闭软件定时器 1
    if(tmr2sta)OSTmrStop(tmr2,OS_TMR_OPT_NONE,0,&err);//关闭定时器 2
    TP_Adjust();
    OSTmrStart(tmr1,&err);                  //重新开启软件定时器 1
    if(tmr2sta)OSTmrStart(tmr2,&err);      //重新开启软件定时器 2
    OSTaskResume(TOUCH_TASK_PRIO); //解挂
    OSTaskResume(QMSGSHOW_TASK_PRIO); //解挂
    ucos_load_main_ui();                //重新加载主界面
    break;
case 5://软件定时器 2 开关
    tmr2sta=!tmr2sta;
    if(tmr2sta)OSTmrStart(tmr2,&err);          //开启软件定时器 2
    else
    {
        OSTmrStop(tmr2,OS_TMR_OPT_NONE,0,&err); //关闭软件定时器 2
        LCD_ShowString(148,262,240,16,16,"TMR2 STOP");
    }
    break;
}
delay_ms(10);
}
}

//信号量集处理任务
void flags_task(void *pdata)
{
    u16 flags;u8 err;
    while(1)
    {
        flags=OSFlagPend(flags_key,0X001F,OS_FLAG_WAIT_SET_ANY,0,&err);
        //等待信号量
        if(flags&0X0001)LCD_ShowString(140,162,240,16,16,"KEY0 DOWN   ");
        if(flags&0X0002)LCD_ShowString(140,162,240,16,16,"KEY1 DOWN   ");
        if(flags&0X0004)LCD_ShowString(140,162,240,16,16,"KEY2 DOWN   ");
        if(flags&0X0008)LCD_ShowString(140,162,240,16,16,"KEY_UP DOWN");
        if(flags&0X0010)LCD_ShowString(140,162,240,16,16,"TPAD DOWN   ");
        BEEP=1; delay_ms(50); BEEP=0;
        OSFlagPost(flags_key,0X001F,OS_FLAG_CLR,&err); //全部信号量清零
    }
}
```



```
//按键扫描任务
void key_task(void *pdata)
{
    u8 key;
    while(1)
    {
        delay_ms(10); key=KEY_Scan(0);
        if(key==0) if(TPAD_Scan(0))key=5;
        if(key)OSMboxPost(msg_key,(void*)key);//发送消息
    }
}
```

本章 test.c 的代码有点多，因为我们创建了 7 个任务，3 个软件定时器及其回调函数，所以，整个代码有点多，我们创建的 7 个任务为：start_task、led_task、touch_task、qmsgshow_task、flags_task、main_task 和 key_task，优先级分别是 10 和 7~2，堆栈大小除了 main_task 是 128，其他都是 64。

我们还创建了 3 个软件定时器 tmr1、tmr2 和 tmr3，tmr1 用于显示 CPU 使用率和内存使用率，每 100ms 执行一次；tmr2 用于在 LCD 的右下角区域不停的显示各种颜色，每 200ms 执行一次；tmr3 用于定时向队列发送消息，每 100ms 发送一次。

本章，我们依旧使用消息邮箱 msg_key 在按键任务和主任务之间传递键值数据，我们创建信号量集 flags_key，在主任务里面将按键键值通过信号量集传递给信号量集处理任务 flags_task，实现按键信息的显示以及发出按键提示音。

本章，我们还创建了一个大小为 256 的消息队列 q_msg，通过软件定时器 tmr3 的回调函数向消息队列发送消息，然后在消息队列显示任务 qmsgshow_task 里面请求消息队列，并在 LCD 上面显示得到的消息。消息队列还用到了动态内存管理。

在主任务 main_task 里面，我们实现了 60.2 节介绍的功能：KEY0 控制 LED1 亮灭；KEY1 控制软件定时器 tmr3 的开关，间接控制队列信息的发送；KEY2 清除触摸屏输入；WK_UP 用于触摸屏校准，在校准的时候，要先挂起触摸屏任务、队列消息显示任务，并停止软件定时器 tmr1 和 tmr2，否则可能对校准时的 LCD 显示造成干扰；TPAD 按键用于控制软件定时器 tmr2 的开关，间接控制屏幕显示。

软件设计部分就为大家介绍到这里。

60.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上，可以看到 LCD 显示界面如图 60.4.1 所示：



图 60.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率为 22% 左右。比上一章多出很多，这主要是 key_task 里面增加了触摸按键 TPAD 的检测，而 TPAD 检测是一个比较耗资源（没有释放 CPU）的过程，另外不停的刷屏（tmr2）也需要一定资源。

通过按 KEY0，可以控制 DS1 的亮灭；

通过按 KEY1 则可以启动 tmr3 控制消息队列发送，可以在 LCD 上面看到 Q 和 MEM 的值慢慢变大（说明队列消息在增多，占用内存也随着消息增多而增大），在 QUEUE MSG 区，开始显示队列消息，再按一次 KEY1 停止 tmr3，此时可以看到 Q 和 MEM 逐渐减小。当 Q 值变为 0 的时候，QUEUE MSG 也停止显示（队列为空）。

通过 KEY2 按键，清除 TOUCH 区域的输入。

通过 WK_UP 按键，可以进行触摸屏校准。

通过 TPAD 按键，可以启动/停止 tmr2，从而控制屏幕的刷新。

在 TOUCH 区域，可以输入手写内容。

任何按键按下，蜂鸣器都会发出“滴”的一声，提示按键被按下，同时在 FLAGS 区域显示按键信息。

第六十一章 战舰 STM32 开发板综合实验

前面已经给大家讲了 55 个实例了，本章将设计一个综合实例，作为本指南的最后一个实验，该实验向大家展示了 STM32 的强大处理能力，并且可以测试开发板的大部分功能。该实验代码非常多，涉及 GUI (ALIENTEK 编写，非 ucGUI)、UCOS、内存管理、图片解码、MP3 播放、文件系统、USB、IAP、NES 模拟器、手写识别、汉字输入等非常多的内容，故本章不讲实现和代码，只讲功能，本章将分为如下几个部分：

61.1 战舰 STM32 开发板综合实验简介

61.2 战舰 STM32 开发板综合实验详解

61.1 战舰 STM32 开发板综合实验简介

战舰 STM32 开发板是 ALIENTEK 的第二款 STM32 开发板(第一款是 MiniSTM32 开发板)，它的出现，主要是为了弥补 Mini 板在一些应用上的缺陷，提供给大家一个更强大的 STM32 开发板平台。

战舰 STM32 开发板的硬件资源在第一章我们已经详细介绍过，是十分强大的，强大的硬件必须配强大的软件才能体现其价值，如果 iPhone 装的是 android 而不是 ios，iPhone 就不是那个 iPhone 了，可能早就被三星打败了。同样，如果开发板只是一堆硬件，那就和一堆废品差不多。

战舰 STM32 开发板的硬件在 V1.0 版本的时候 (2010 年 12 月份)，基本就定型了，之后近两年多的时间，我们一直在编写代码，其中绝大部分时间是在写开发板的综合实验(即本实验)，我们坚持资料不完善，坚决不卖，这样战舰 STM32 开发板的上市时间一推再推，硬件版本也从 1.0 升级到了 1.8，甚至有朋友笑言，我都从大二等到大四了...在此，对那些还在等待我们开发板的朋友说声抱歉，谢谢你们的支持和理解。我想说，用心做产品，真的不容易，战舰开发过程中的点点滴滴，有机会再和大家分享。

在今年 7 月份的时候，终于把战舰 STM32 开发板综合实验的最后一个功能写完了，至此综合实验的开发基本完成，前前后后，耗时近两年。

接下来我们就看看战舰 STM32 开发板综合实验的功能吧。

战舰 STM32 开发板综合实验总共有 18 大功能，分为 2 页，每页 9 个功能，页面的切换采用滑动操作。18 大功能分别为：电子图书、数码相框、音乐播放、应用中心、时钟、系统设置、FC 游戏机、收音机、记事本、运行器、3D、手写画笔、照相机、录音机、USB 连接、TOM 猫、无线传书、计算器。

电子图书，支持.txt/.c/.h/.lrc 等 4 种格式的文件阅读。

数码相框，支持.bmp/.jpeg/.jpg/.gif 等 4 种格式的图片文件播放。

音乐播放，支持.mp3/.wma/.wav/.flac/.ogg/.mid/等常见音频文件的播放。

应用中心，可以扩展 16 个应用程序，我们实现了其中 1 个，其他留给大家自己扩展。

时钟，支持温度、时间、日期、星期的显示，并加入时间 3D 效果显示。

系统设置，整个综合实验的设置。

FC 游戏机，即 NES 模拟器，支持.nes 文件的运行，通过开发板玩 NES 游戏。

收音机，支持全范围 FM (76Mhz~108Mhz) 接收，支持手动/半自动/全自动搜台。

记事本，可以实现文本 (.txt/.c/.h/.lrc) 记录编辑等功能，支持中英文输入，手写识别。

运行器，即 SRAM IAP 功能，支持.bin 文件的运行 (文件大小+SRAM 大小≤60K)。

3D，可以测量角度，并支持 3D 演示。

手写画笔，可以作画/对 bmp 图片进行编辑，支持画笔颜色/尺寸设置。

照相机，可以拍照 (需要摄像头模块支持)，并支持成像效果设置。

录音机，支持 wav 文件格式的录音 (8Khz/16 位单声道录音)。

USB 连接，支持和电脑连接读写 SD 卡/SPI FLASH 的内容。

TOM 猫，和手机的 TOM 猫游戏的功能类似，模仿人声，进行人机对话。

无线传书，通过无线模块，实现两个开发板之间的无线通信。

计算器，一个科学计算器，支持各种运算，精度为 12 位，支持科学计数法表示。

以上，就是综合实验的 18 个功能简介，涉及到的内容包括：GUI (ALIENTEK 编写，非 ucGUI)、UCOS、内存管理、图片解码、MP3 播放、文件系统、USB、IAP、NES 模拟器、手写识别、汉字输入等非常多的内容。下面，我们将详细介绍这 18 个功能。



61.2 战舰 STM32 开发板综合实验详解

要测试战舰 STM32 开发板综合实验的全部功能，大家得自备 1 个 SD 卡和 1 个 ALIENTEK 摄像头模块。不过，就算没有这两个东西，综合实验还是可以正常运行的，只是有些限制而已，比如：不能保存新建的记事本、不能保存新建的画图、不能使用录音机功能、不能使用摄像头功能等。除了这几个，其他功能都可以正常运行。

我们先来看看战舰 STM32 开发板综合实验的启动界面，启动界面如图 61.2.1 所示：

```
ALIENTEK STM32 WARSHIP
Copyright (C) 2010-2020
HARDWARE:V1.8, SOFTWARE:V2.24
LCD ID:9341
CPU:STM32F103ZET6 72Mhz
FLASH:512KB SRAM:64KB
Ex Memory Test:1024KB OK
Ex Flash:8192KB OK
FATFS Check... OK
SD Card: 1882MB OK
Flash Disk:6124KB OK
TPAD Check... OK
RTC Check... OK
ADXL345 Check... OK
24C02 Check... OK
RDA5820 Check... OK
VS1053 Check... OK
Font Check... OK
SYSTEM Files Check... OK
Touch Check... OK
SYSTEM Parameter Load... OK
SYSTEM Starting...
```

图 61.2.1 综合实验启动界面

注意：综合实验支持屏幕截图（通过 USMART 控制，波特率为 115200），本章所有图片均来自屏幕截图！

上图显示了综合实验的详细启动过程，首先显示了版权信息，软硬件版本，接着显示了 LCD 驱动器的型号（LCD ID），然后显示 CPU 和内存信息，之后显示 SPI FLASH 的大小，接着开始初始化文件系统（FATFS），然后显示 SD 卡容量和 FLASH Disk 容量（注意 FLASH Disk 就是指 SPI FLASH，因为我们划分了 6M 空间给 FATFS 管理，所以 FLASH Disk 的容量为 6124KB）。

接着，就是硬件检测，完了之后检测字库和系统文件，再初始化触摸屏，加载系统参数（参数保存在 24C02 里面），最后启动系统。在加载过程中，任何一个地方出错，都会显示相应的提示信息，请在检查无误后，按复位重启。

这里有几个注意的地方：

- ① 如果没插入 SD 卡，会显示 SD Card ERROR，不过系统还是会继续启动，因为没有 SD 卡系统还是可以启动的（前提是 SPI FLASH (W25Q64) 里面的系统文件和字库文件都是正常的）。
- ② 系统文件和字库文件都是存在 SPI FLASH(W25Q64)里面的，如这两个文件被破坏了，在启动的时候，会执行字库和系统文件的更新，此时你得准备一个 SD 卡，并拷贝 SYSTEM 文件夹（注意：这个 SYSTEM 文件夹不是开发板例程里的 SYSTEM 文件夹，而是光盘根目录→SD 卡根目录文件→SYSTEM 文件夹）到 SD 卡根目录，以便系统更新时使用。
- ③ FLASH Disk 是从 SPI FLASH (W25Q64) 里面分割 6M 空间出来实现的，强制将 4K



字节的扇区改为 512 字节使用，所以在写操作的时候擦除次数会明显提升（8 倍以上），因此，如非必要，请不要往 FLASH Disk 里面写文件。频繁的写操作，很容易将 FLASH Disk 写挂掉。

- ④ 在系统启动时，一直按着 KEY0 不放（加载到 Touch Check 的时候），可以进入强制校准。当你发现触摸屏不准的时候，可以使用这个办法强制校准。
- ⑤ 在系统启动时，一直按着 KEY1 不放（加载到 Font Check 的时候），可以强制更新字库。
- ⑥ 本系统用到触摸按键 TPAD 做返回（类似手机的 HOME 键），所以请确保多功能端口 P14 的 ADC 和 TPAD 用跳线帽短接！
- ⑦ 如果插入了 SD 卡，系统在启动的时候，会在 SD 卡的根目录创建 4 个文件夹：TEXT、RECORDER、PAINT 和 PHOTO。其中，TEXT 文件夹用来保存新建的文本文件（记事本功能时使用）；RECORDER 文件夹用来保存录音文件（录音机功能时使用）；PAINT 文件夹用来保存新建的画板文件（手写画笔功能时使用）；PHOTO 文件夹用来保存相片（照相机功能时使用）。

在 SYSTEM Starting...之后，系统启动 UCOSII，并加载 SPB 界面，在加载成功之后，来到主界面，主界面如图 61.2.2 所示：

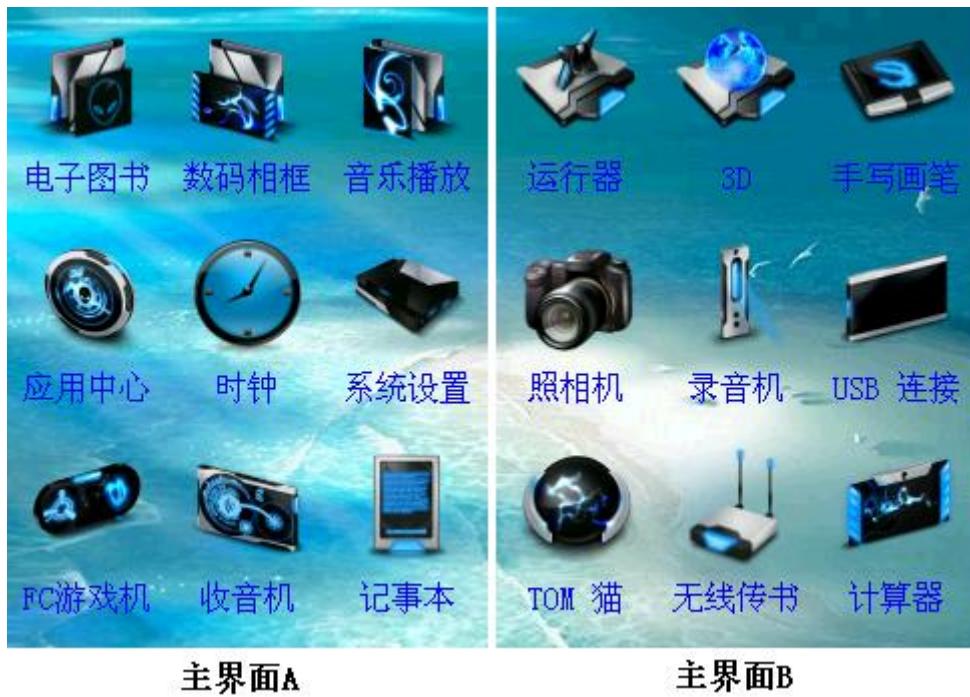


图 61.2.2 综合实验系统主界面

这里主界面默认是简体中文的，我们可以在系统设置里面设置语言，战舰 STM32 开发板综合实验支持 3 种语言选择：简体中文、繁体中文和英文。

在进入主界面之后，开发板上的 DS0 开始有规律的短亮（每 2.5 秒左右亮 100ms），提示系统运行正常，我们可以通过 DS0 判断系统的运行状况。另外，如果运行过程中，出现 HardFault 的情况，系统则会进入 HardFault 中断服务函数，此时 DS0 和 DS1 都会闪烁，提示系统故障。同时在串口打印故障信息。通过串口，系统会打印其他很多信息，最常打印的是内存使用率，然后我们还可以通过 USMART 对系统进行调试。

如图 61.2.2 所示，综合实验的主界面分为 2 页，通过滑动切换，系统刚启动的时候加载的是主界面 A，通过滑动可以切换到主界面 B，类似现在的智能手机。主界面，总共 18 个功能图标，我们可以随便点击一个即可选中，如图 61.2.3 所示：



图 61.2.3 选中电子图书

从上图可以看出，选中之后，图标发生了一点点变化，手机图标也是类似的效果，其实就是一个 alphablend。再次点击该图标，我们就可以进入电子图书功能。

在任何界面下，都可以通过按 TPAD 返回上一级，直至返回到主界面。PS:TPAD 就是战舰 STM32 开发板上的一个触摸按键，即右下角的 ALIENTEK LOGO !!

在介绍完系统启动之后，我们开始介绍各个功能。

61.2.1 电子图书

双击主界面的电子图书图标，进入如图 61.2.1.1 所示的文件浏览界面：

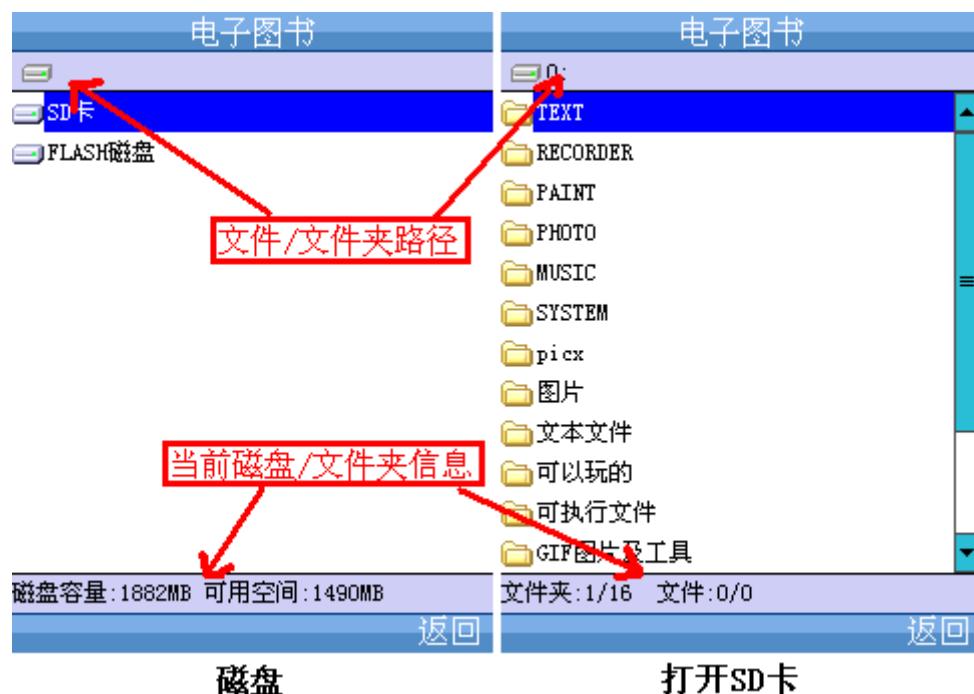




图 61.2.1.1 文件浏览界面

上图中，左侧的图是我们刚刚进入的时候看到的界面（类似在 XP 上打开我的电脑），可以看到我们有 2 个盘：SD 卡和 FLASH 磁盘。我们可以选择任何一个打开，并浏览里面的内容。注意，即使没有插入 SD 卡，还是会出现 SD 卡图标，只是此时不能打开而已！

界面的上方显示文件/文件夹的路径。如果当前路径是磁盘/磁盘根目录则显示磁盘图标，如果是文件夹，则显示文件夹图标，另外，如果路径太深，则只显示部分路径（其余用...代替）。界面的下方显示磁盘/文件夹信息。

界面的下方，显示磁盘信息/当前文件夹信息。对磁盘，则显示当前选中磁盘的总容量和可用空间，对文件夹，则显示当前路径下文件夹总数和文件总数，并显示你当前选中的是第几个文件夹/文件。

双击打开 SD 卡，得到界面如右侧图片所示，此时，因为 SD 卡根目录的文件数目超过了 1 页所能显示的数目，所以在右侧出现了滚动条，我们可以拖动滚动条/按滚动条两端的按钮/直接在屏幕中心区域拖动，来查找你要打开的文件/文件夹。

选中一个文件夹，双击打开得到如图 61.2.1.2 所示界面：

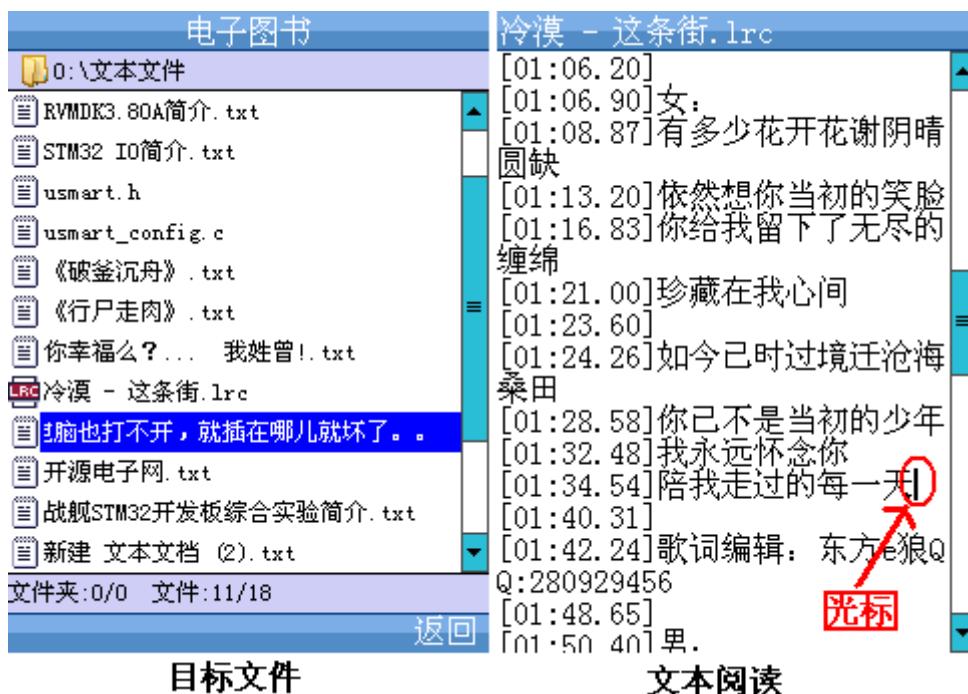


图 61.2.1.2 目标文件和文本阅读

上图左侧显示了当前文件夹下面的目标文件（即电子图书支持的文件，包括.txt/.h/.c/.lrc 等格式，其中.txt/.h/.c 文件共用 1 个图标，.lrc 文件单独一个图标）。另外，如果文件名太长，在我们选中该文件名后，系统会以走字的形式，显示整个文件名。

我们打开一个 lrc 文件，开始文本阅读，如图右侧的图片所示，同样我们可以通过滚动条/拖动的方式来浏览，图中我们还看到有一个光标，触摸屏点到哪，它就在哪里闪烁，可以方便大家阅读。

文本阅读是将整个文本文件加载到外部内存里面来实现的，所以文本文件最大不能超过外部内存总大小，即 680KB（这里仅指受内存管理的部分，不是整个外部 SRAM 的大小）。

当我们想退出文本阅读的时候，通过按 TPAD 触摸按键实现，按一下 TPAD，则又回到查找目标文件状态（左侧图），按返回按钮可以返回上一层目录，如果再按一次 TPAD 则直接返回主界面。



61.2.2 数码相框

双击主界面的数码相框图标，进入文件浏览界面，这个和 61.2.1 节差不多，我们找到存放图片的文件夹，如图 61.2.2.1 所示：



图 61.2.2.1 文件浏览和图片播放

左侧是文件浏览的界面，可以看到在图片文件夹下总共有 18 个文件，包括 gif/jpg/bmp 等，这些都是数码相框功能所支持的格式。右侧图片显示了一个正在播放的 GIF 图片，并在其左上角显示当前图片的名字。当然，我们也可以播放 bmp 和 jpg 文件，如图 61.2.2.2 所示：



图 61.2.2.2 bmp 和 jpg 图片播放



对于 bmp 和 jpg 文件，基本没有尺寸限制（但图片越大，解码时间越久），但是对于 gif 文件，则只支持尺寸在 240*320 以内的文件（因为 gif 图片我们不好做尺寸压缩处理），超过这个尺寸的 gif 图片将无法显示！！

我们可以通过按屏幕的上方（1/3 屏幕）区域切换到上一张图片浏览；通过按屏幕的下方（1/3 屏幕）区域切换到下一章图片；通过单击屏幕的中间（1/3 屏幕）区域可以暂停自动播放，同时 DS1 亮，提示正在暂停状态，双击屏幕的中间区域会弹出返回按钮，如图 61.2.2.3 所示：



图 61.2.2.3 弹出返回按钮

此时，我们可以通过按返回按钮返回文件浏览状态，当然也可以通过按 TPAD 按钮，直接返回文件浏览状态（不需要等返回按钮弹出）。

图片浏览支持两种自动播放模式：循环播放/随即播放。大家可以在系统设置里面设置图片播放模式。系统默认是循环播放模式，在该模式下，每隔 4 秒左右自动播放下一张图片，依次播放所有图片。而随机播放模式，也是每隔 4 秒左右自动播放下一张图片，但是不是顺序播放，而是随机的播放下一张图片。

另外需要注意，不是所有的 jpg 格式图片都可以在我们的开发板上正常播放的（解码程序的问题），只有 JFIF 格式的 jpg 文件才能正常解码显示，对于 EXIF 格式的 jpg 文件，则不能直接显示，大家可以将 EXIF 格式的 jpg 文件用 XP 的画图打开，然后再保存一下，就将 EXIF 格式转为 JFIF 格式了，这样就可以在开发板上正常解码，并显示了。

61.2.3 音乐播放

双击主界面的音乐播放图标，进入文件浏览界面，这个和 61.2.1 节差不多，只是这里我们浏览的文件变为了.mp3/.ogg/.wma/.flac/.wav/.midi 等音频文件，我们找到存放音频文件的文件夹，如图 61.2.3.1 所示：

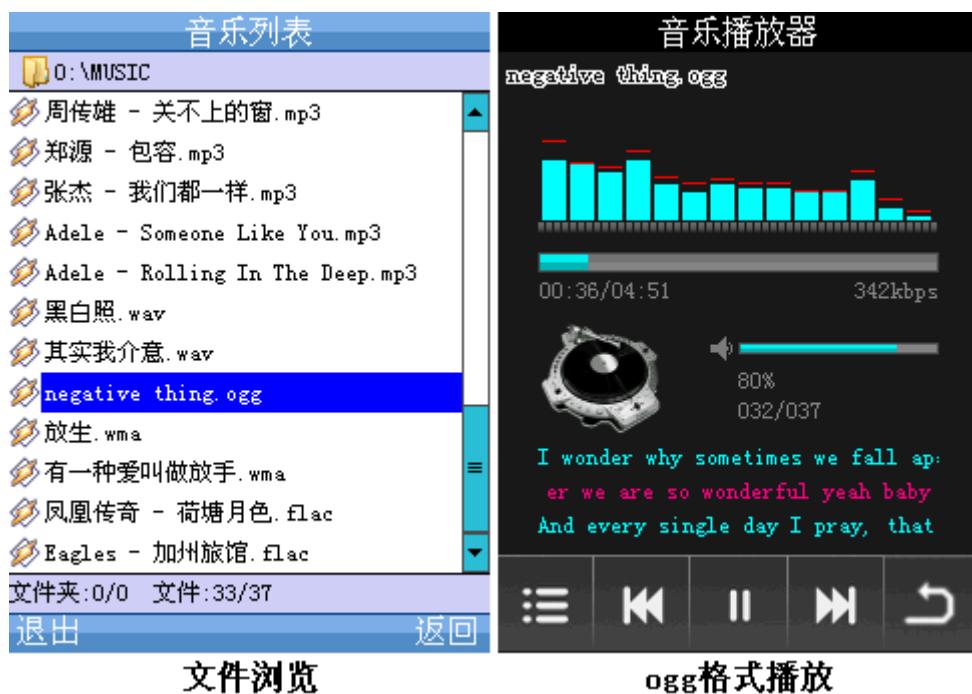


图 61.2.3.1 文件浏览和 ogg 格式播放

左侧是文件浏览的界面，可以看到在 MUSIC 文件夹下总共有 37 个音频文件，包括 mp3/ogg/wma/flac/wav 等格式，这些都是播放器所支持的格式。右侧图片则是我们播放器的主要界面，该界面显示了当前播放歌曲的名字、播放进度、播放时长、总时长、码率、音量、当前文件编号、总文件数、歌词等信息。下方的 5 个按键分别是：目录、上一曲、暂停/播放、下一曲、返回。点击播放进度条，可以直接设置歌曲播放位置，点击声音进度条，可以设置音量。上图为正在播放 ogg 文件，当然我们还可以播放其他音频格式，如图 61.2.3.2 所示：

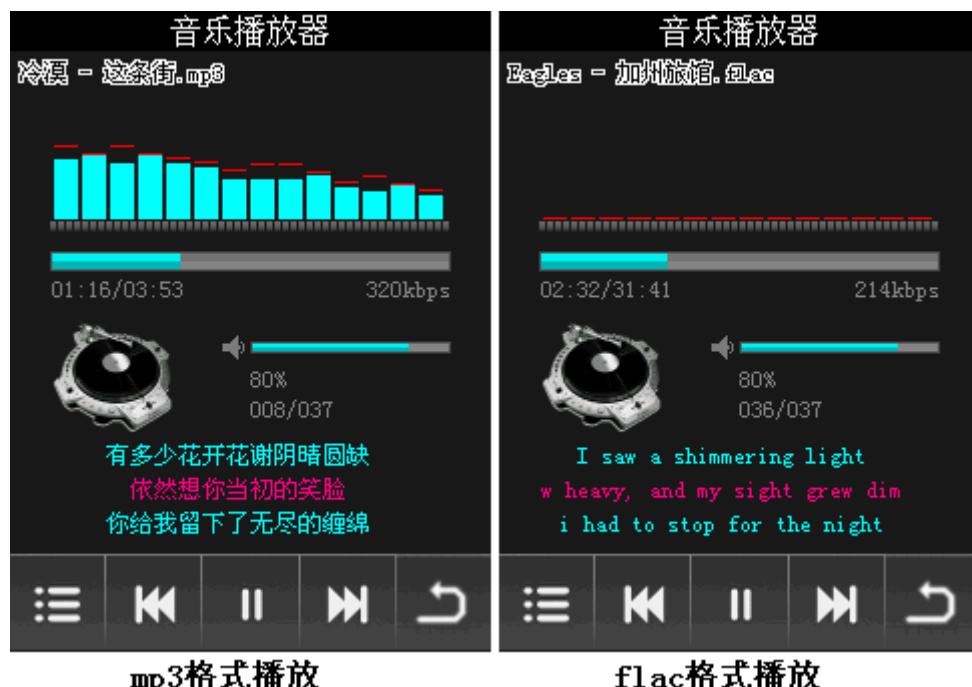


图 61.2.3.2 mp3 格式播放和 flac 格式播放

图 61.2.3.2 中，分别显示了播放 mp3 格式和 flac 格式的音频文件。播放 flac 格式的时候，



由于得不到正确的码率，所以总时间也是不正确的，图中数字仅供参考。另外播放 flac 因为要加载 flac 的 patch，故无法加载频谱分析的 patch，从而无法显示频谱，可以看到在右侧的图片中，没有频谱显示了，除了 flac 不能显示频谱，其他音频文件都是可以正常显示频谱的。

播放器还可以设置音效和播放模式（均在系统设置里面设置）。音效包括高低音调节、空间效果等设置。播放模式有 3 种：全部循环、随机播放、单曲循环，默认为全部循环。

另外，关于歌词显示。歌词必须和歌曲在同一个文件夹里面，且名字必须相同（当后缀是不同的，歌词后缀为.lrc），这样才能正常显示歌词。对于没有歌词文件的歌曲，则直接播放，不显示歌词。歌词分为 3 行，第一行为上一句歌词，第二行为当前正在唱的歌词，第三行为将要唱的歌词。对于第二行歌词，如果太长，则会采用走字的形式来显示，走字时间由系统自动确定。

我们可以通过按目录按钮，来选择其他音频文件；按返回按键（或 TPAD）则可以返回主界面，不过此时正在播放的歌曲还是会继续播放（后台播放），如果想关闭音乐播放器，则需要先按暂停，然后返回主界面，即可关闭音频播放器，否则音频播放器将一直播放音乐。

最后，我们默认是开启了 FM 发射的，在播放 MP3 的时候，音频会通过 RDA5820 发送出去，默认的频率是 93.6Mhz，大家可以打开收音机调到 93.6Mhz，就可以听到来自开发板的歌声了。FM 发射频率和发射开关也都是可以在系统设置里面设置的，具体后面再介绍。

61.2.4 应用中心

双击主界面的应用中心图标，进入应用中心界面，如图 61.2.4.1 所示：



图 61.2.4.1 应用中心和红外遥控

左侧图片是我们刚进入应用中心看到的界面，在该界面下总共有 16 个图标，我们仅实现了第一个：红外遥控功能。其他都没有实现，大家可以自由发挥，添加属于自己的东西。双击第一个图标，会弹出一个红外遥控的小窗口，用于接收红外信号，如图 61.2.4.1 右侧图片所示。

此时，我们将红外遥控对准战舰 STM32 开发板的红外接收头，并按钮，则可以在红外遥控窗体里面显示键值、按键次数、符号等信息。如图 61.2.4.2 所示：



图 61.2.4.2 红外按键解码

图中，我们按下了红外遥控器下的两个按键，分别得到两个按键的键值、次数和符号等信息。其中次数是代表我们持续按下红外遥控某个按键的时长，越长该值越大。

需要注意一点是，如果当前正在播放 MP3，则红外解码成功率大大降低，原因是 MP3 播放任务的优先级最高，严重影响红外信号接收，导致解码成功率降低，当发现无法识别的时候，可以先停止 MP3 的播放再试试。

61.2.5 时钟

双击主界面的时钟图标，进入时钟界面，如图 61.2.5.1 所示：





图 61.2.5.1 时钟界面

图 61.2.5.1 的左侧图片为加载时钟界面时的提示界面，表明没有检测到 18B20，启用内部温度传感器，之后进入时钟主界面，如右侧图片所示。在时钟界面，我们显示了日期、时间、温度、星期等信息，并且在屏幕上方区域，有一个 3D 的时间在显示，3D 时间显示会不停的变换位置，位置变化是无规律的。我们可以在系统设置里面设置时间和日期，并且还可以设置闹钟和闹铃，这个我们后面再介绍。

图中的温度是通过 STM32 自带的温度传感器采集的，所以有点偏高，如果我们在开发板的 U13 处插入 DS18B20，则会采集来自 18B20 的温度，这样就比较准确了。

在进入时间界面以后，要退出该界面有 2 个办法：1，在屏幕向左滑动触摸；2，按 TPAD 返回。

61.2.6 系统设置

双击主界面的系统设置图标，进入系统设置界面，如图 61.2.6.1 所示：



图 61.2.6.1 系统设置主界面和时间设置界面

上图中左侧的图片为系统设置主界面，在系统设置里面，总共有 19 个项目：时间设置、日期设置、闹钟时间设置、闹钟开关设置、闹钟铃声设置、语言设置、数码相框设置、MP3 播放模式设置、MP3 音效设置、FM 发射开关设置、FM 发射频率设置、FM 收音设置、背光设置、屏幕校准、传感器校准、系统文件更新、系统信息、系统状态、关于。通过这 19 个项目，我们可以设置和查看各种系统参数。下面我们将一一介绍这些设置。

首先是时间设置，如图 61.2.6.1 右侧图片所示，双击时间设置，就会弹出一个时间是指对话框，通过这个对话框，我们就可以设置开发板的时间了。设置好之后点击确定回到系统设置主界面，如果想放弃设置，则直接点击取消（或 TPAD）。

再来看看日期设置和闹钟时间设置，如图 61.2.6.2 所示：



图 61.2.6.2 日期设置和闹钟时间设置

上图中，左侧的对话框用来设置系统日期，右侧的对话框用来设置闹钟时间。操作上同前面介绍的时间设置的方法一模一样。关于闹钟，我们等下再详细介绍，先看闹钟开关设置和闹钟铃声设置两个界面，如图 61.2.6.3 所示：



图 61.2.6.3 闹钟开关设置和闹钟铃声设置

上图中，左侧对话框用来设置闹钟开关，右侧对话框用来设置闹钟铃声。这里，我们来介绍一下本系统的闹钟，本系统的闹钟以星期为周期，以时间为点实现闹钟，比如判断一个闹钟是否应该响铃的标准是：先判断星期的条件是否满足，比如上图我们设置是周一到周五闹铃，今天（10月5号）刚好是周五，所以满足星期条件，接着看时间是否相等，如果两个条件都满



足，则闹铃。从前面的时间设置我们知道当前时间是 20:30 分，而上图我们设置的闹钟时间是 20:35，所以时间还不相等，故不闹铃，当时间来到 20:35 的时候，系统将会闹铃。闹铃铃声有 4 种，如上图右侧图片所示，铃声由蜂鸣器产生，铃声 1 对应“滴”，铃声 2 对应“滴、滴”，铃声 3 和 4 依此类推。当闹钟时间到来的时候，产生闹铃，如图 61.2.6.4 所示：



图 61.2.6.4 闹铃和语言设置

上图中，左侧的图片显示正在闹铃。此时会弹出一个闹钟的对话框，并显示当前时间，同时蜂鸣器发出“滴、滴、滴、滴”的闹铃声（铃声 4）。按取消（或 TPAD）可以关闭闹钟，按再响，则 5 分钟后（20:40）继续闹铃。右侧的图片为语言设置界面，系统支持 3 种语言设置，默认为简体中文，设置为繁体中文/English 之后如图 61.2.6.5 所示：





图 61.2.6.5 繁体中文和 English

上图显示了繁体中文和 English 的设置，不过本章我们还是以简体中文为例进行介绍。下面，我们来看看数码相框设置和 MP3 播放模式设置，如图 61.2.6.6 所示：

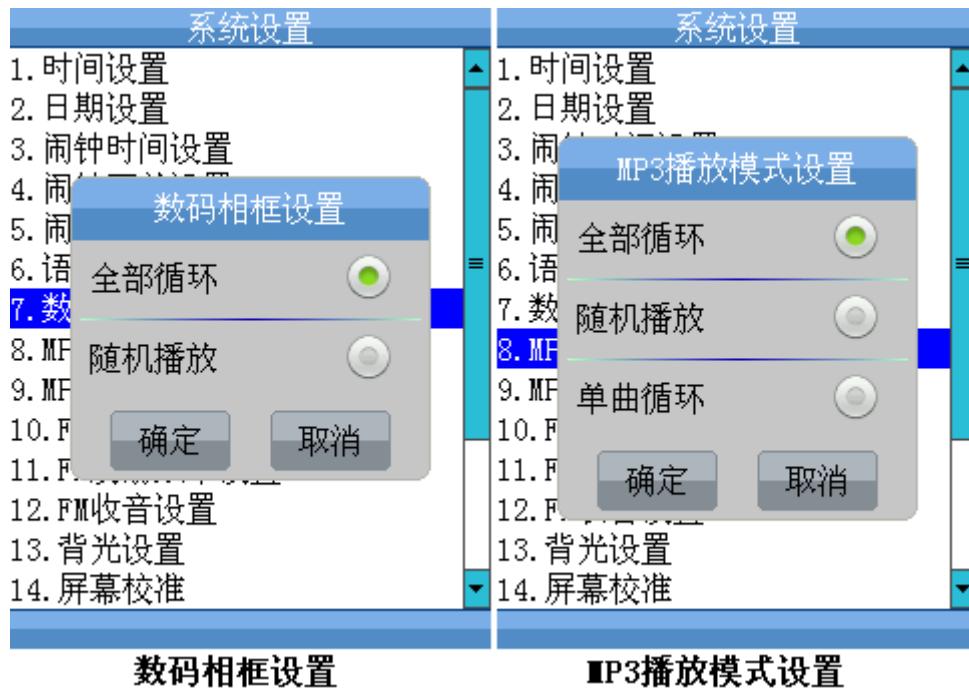


图 61.2.6.6 数码相框设置和 MP3 播放模式设置

前面提到数码相框支持全部循环播放和随机播放两种模式，就是通过上图左侧的界面设置的。而 MP3 的三个播放模式，则通过右侧的界面进行设置。接下来看看 MP3 音效设置和 FM 发射开关设置，如图 61.2.6.7 所示：

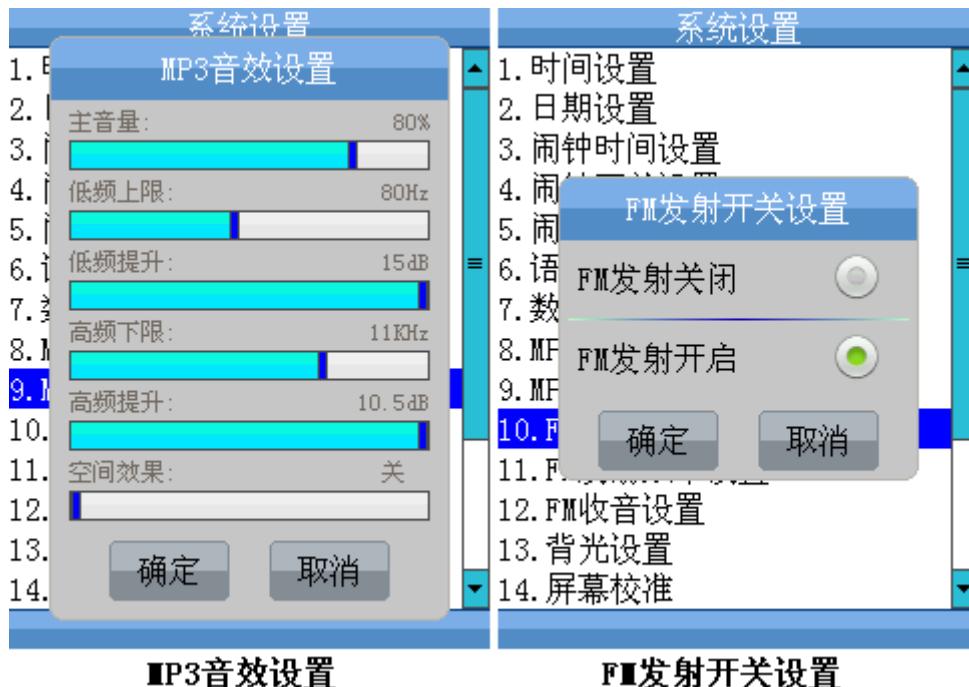


图 61.2.6.7 MP3 音效设置和 FM 发射开关设置

上图中，左侧的界面我们可以设置 MP3 播放的音效（VS1053 的设置），包括音量、高低音



以及空间效果等，大家可以根据自己喜欢设置，以上为默认设置。右侧的 FM 发射开关设置，用来设置是否开启 FM 发射，默认设置为开启，即只要不是收音机模式，其他所有界面 FM 发射都是开启的，这样我们就可以通过收音机来听到来自 STM32 开发板的声音了。

下面我们看看 FM 发射频率设置和 FM 收音设置，如图 61.2.6.8 所示：

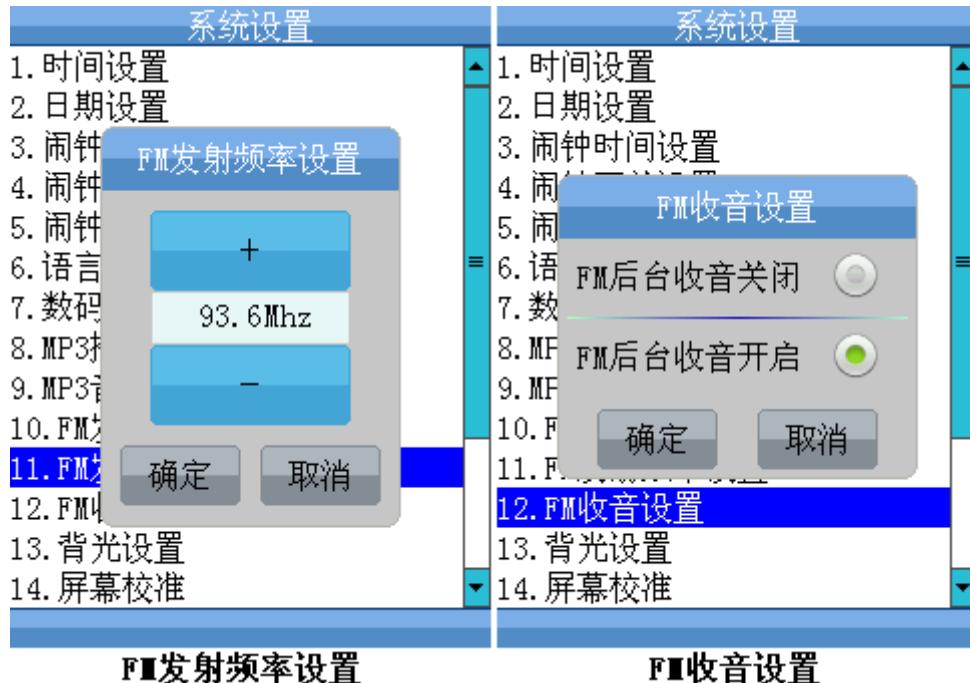


图 61.2.6.8 FM 发射频率设置和 FM 收音设置

上图中，左侧的界面用于设置 FM 发射频率，用于设置 FM 发射频点，我们默认的频率是 93.6Mhz，所以大家的收音机请调到 93.6Mhz（默认频率），以接听来自开发板的声音。右侧的图片用于设置 FM 收音是否开启后台播放的功能。

接下来，我们看看背光设置和屏幕校准，如图 61.2.6.9 所示：

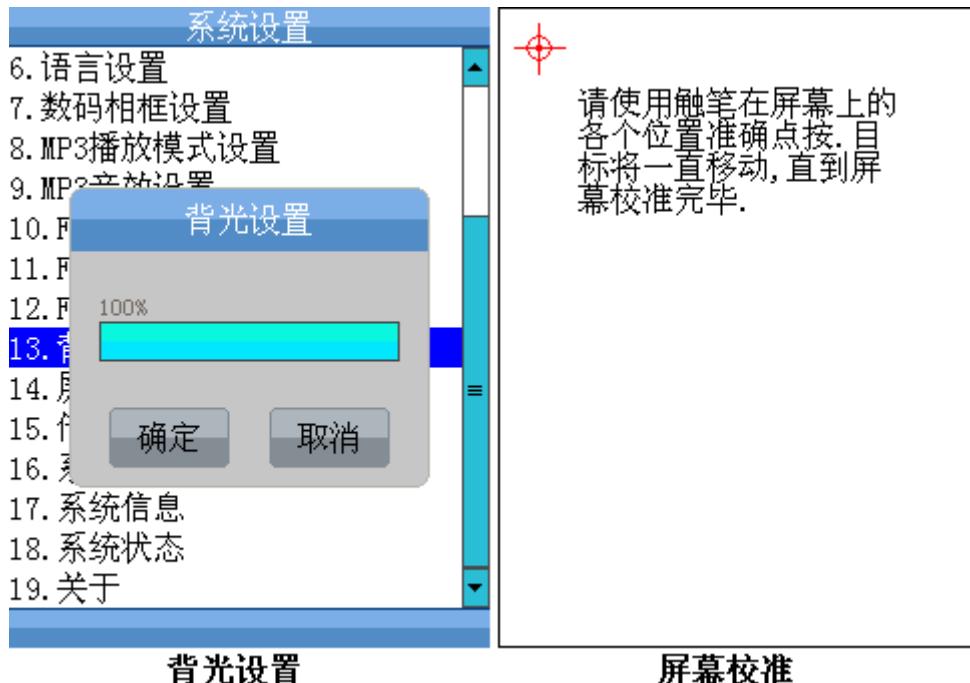




图 61.2.6.9 背光设置和屏幕校准

上图中，左侧的界面用于设置 LCD 背光的亮度，默认我们是设置为最亮的，大家可以根据自己的喜欢设置背光亮度，背光亮度控制是通过 PWM 控制的。

右侧为触摸屏校准界面，这个校准界面和手机校准界面基本类似，校准的时候，请用触笔（或者其他尖一点的东西）依次点击 4 个十字圈的最中心（图中只是第一个，如果点击了第一个会自动弹出第二个，总共 4 个），在 4 个校准点都准确点击之后，系统提示校准成功字符串：Touch Screen Adjust OK!。如果校准失败，则提示失败信息，请重新校准，直到校准成功，如果多次校准都不成功，有可能你的触摸屏有问题了！

另外，在该界面下，如果连续 10 秒没有输入的话，系统会自动退出校准界面，当然，我们也可以按 TPAD 直接退出。

接下来，我们看看传感器校准和系统文件更新，如图 61.2.6.10 所示：



图 61.2.6.10 传感器校准和系统文件更新提示

图中，左侧图片为传感器校准界面，这里的传感器设置 ADXL345 重力加速度传感器，校准的时候，请保持开发板水平并稳定，以得到最好的校准效果。

右侧的界面为系统文件更新提示界面，这里的系统文件是指 SYSTEM 文件夹里面的所有内容。战舰 STM32 开发板综合例程之所以可以没有 SD 卡也能正常运行，主要是将 SYSTEM 文件夹（注意这个不是源码里面的 SYSTEM 文件夹！！）拷贝到了 FLASH Disk（即 W25Q64）里面，这样，我们所有的系统资源都可以从 W25Q64 里面获得，从而正常启动。

SYSTEM 文件夹目前是包含 144 个文件，总大小为 2.6MB，包括 137 个图片/图标，另外包括 5 个字库相关文件以及 2 个 VS1053 的 PATCH 文件。这些文件一般不要修改，如果你想自己 DIY 的话，那可以修改这些文件，以达到你要的效果，不过建议修改之前备份一下，搞坏了还可以还原。

如果在图 61.2.6.10 的系统文件更新提示时选择确定，则会执行系统文件更新，将 SD 卡的 SYSTEM 文件夹，拷贝到 FLASH Disk 里面。这里有个前提，就是你的 SD 卡必须有这个 SYSTEM 文件夹！更新时界面如图 61.2.6.11 所示：



图 61.2.6.11 系统文件更新和系统信息

上图中，左侧的界面显示了系统文件正在更新，该界面显示了当前更新的文件夹以及文件和进度等信息。右侧的界面为系统信息界面，通过该界面，可以看到软硬件的详细信息。

最后，我们来看看系统状态和关于界面，如图 61.2.6.12 所示：



图 61.2.6.12 系统状态和关于界面

上图中，左侧的界面显示了当前系统资源状况，显示了当前 CPU 使用率，CPU 温度以及内存使用率。图为后台正在播放 MP3 的时候资源使用情况，当播放高码率的歌曲的时候，CPU 使用率会大增（如播放 wav，则 CPU 使用率在 60% 左右）。

右侧的图片显示了战舰 STM32 开发板的软硬件版本以及产品序列号，这个序列号是全球



唯一的，每个开发板都不一样。

61.2.7 FC 游戏机

战舰 STM32 开发板综合实验移植了 NES 模拟器，可以运行 nes 游戏，双击主界面的系统设置图标，进入文件浏览界面，如图 61.2.7.1 所示：



图 61.2.7.1 文件浏览和小蜜蜂游戏

左侧为 nes 文件浏览界面，我们随便选择一个打开即可开始游戏了，记得插上手柄哦！右侧的图片为小蜜蜂游戏的界面，当然还可以玩很多其他经典游戏，如下面的图片所示：





图 61.2.7.2 超级玛丽和 90 坦克

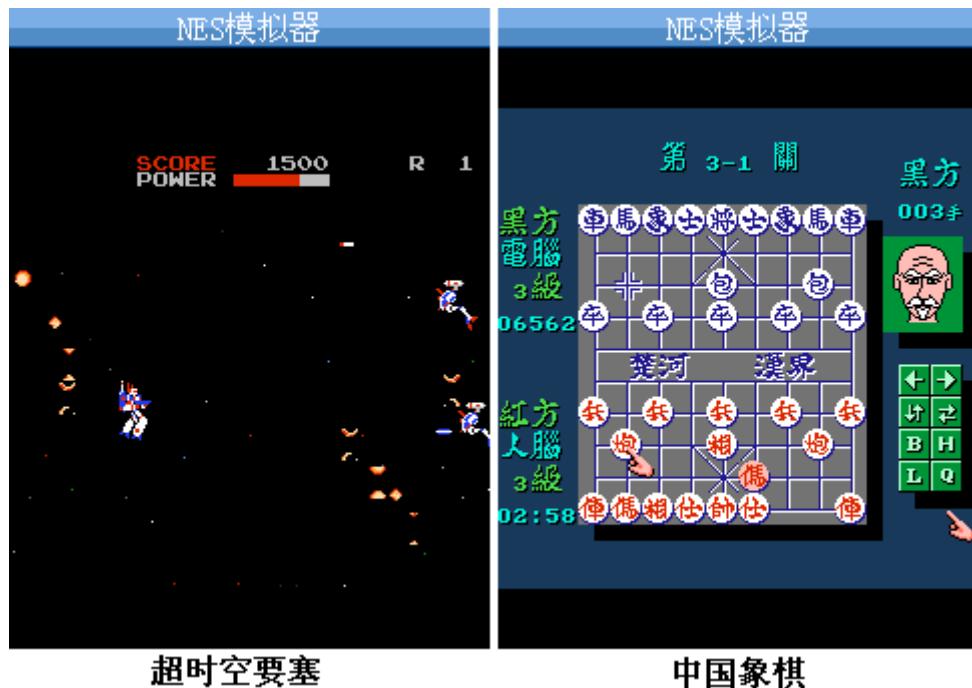


图 61.2.7.3 超时空要塞和中国象棋

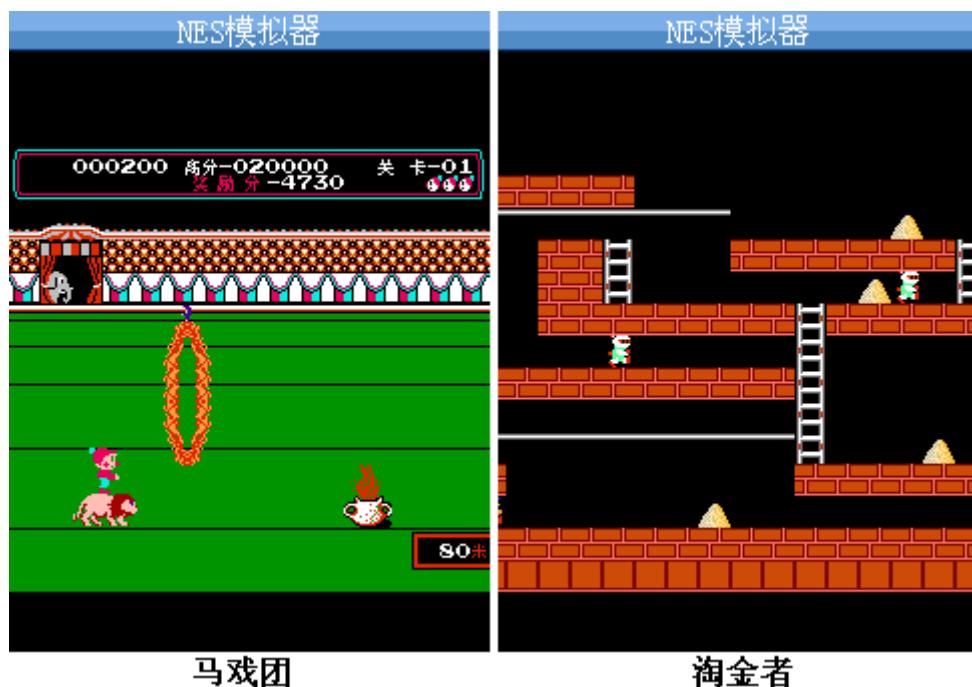


图 61.2.7.4 马戏团和淘金者

这里，我们仅列出了几种游戏，这都是 80 后童年时玩的经典游戏，如今，在战舰 STM32 开发板上，大家可以回味一下当年的经典了。

不过，我们提供的 nes 模拟器，由于代码问题，对大于 50KB 的 nes 文件基本不支持，不过即使这样，还是有很多游戏可玩的。另外也没有加入声音输出。如果对 nes 模拟器有兴趣的朋友可以完善一下这两方面，我们在光盘提供了相关资料可供研究。



61.2.8 收音机

双击主界面的收音机图标，进入收音机界面，如图 61.2.8.1 所示：



图 61.2.8.1 收音机主界面和模式选择界面

上图中，左侧图片为收音机的主界面，显示了当前频率、单/双声道、信号强度、音量、电台编号（自动搜台的时候自动保存）等信息，界面下方的 5 个按钮分别是：模式选择、频率减（或上一个电台）、暂停/继续收音、频率增（或下一个电台）和返回。右侧的图片为按了模式选择后弹出的界面，选择模式设置/频段选择并按确认后，得到如图 61.2.8.2 所示：



图 61.2.8.2 模式设置和频段选择界面



上图中，左侧的图片为模式设置界面，总共有3个模式可以设置：手动搜台、半自动搜台和全自动搜台。

手动搜台：完全手动搜索，通过频率增/减两个按钮调节频率。

半自动搜台：此时频率增/减分别代表查找下一个/上一个电台，只要按一下按钮，收音机自动查找下一个/上一个电台，找到有效电台即停止搜索，并播放这个有效电台。

全自动搜台：选中之后，收音机从最小频率开始找台，一直搜索到最大频率，把整个过程中的有效电台记录下来，搜索完毕，可以从主界面的“CH:”看到总有效电台的个数，可以通过频率增/减按钮来跳转电台。

右侧的图片为频段选择界面，本收音机支持3个频段：日本频段（76Mhz~91Mhz）、欧美频段（87~108Mhz，也是中国电台使用的频段）、扩展频段（76Mhz~108Mhz）。默认设置为欧美频段。

收音机可以后台工作，只要您在系统设置里面开启了后台收音。如果没有开启后台收音，在按返回键之后，收音机将自动关闭。

本收音机使用起来还是比较简单的，使用时，请把天线拉出，如果搜不到台，一般是因为你所处环境干扰太大，建议去空旷地方试试。

61.2.9 记事本

双击主界面的记事本图标，首先弹出模式选择对话框，如图 61.2.9.1 所示：



图 61.2.9.1 模式选择和新建文本文件

记事本支持2种模式：1，新建文本文件，这种方式完全新建一个文本文件（以当前系统时间命名），用来输入信息。2，打开已有文件，这种方式可以对已有的文件进行编辑。

上图中，右侧的界面为我们选择新建文本文件后的界面，此时出现一个空白编辑区和一个闪烁的光标，我们通过下方的键盘输入信息即可，这个输入键盘和我们的手机键盘十分类似，输入方法也是一模一样，支持中文、字母、数字和手写识别输入等几种输入方式。中文输入和标点符号输入，如图 61.2.9.2 所示：

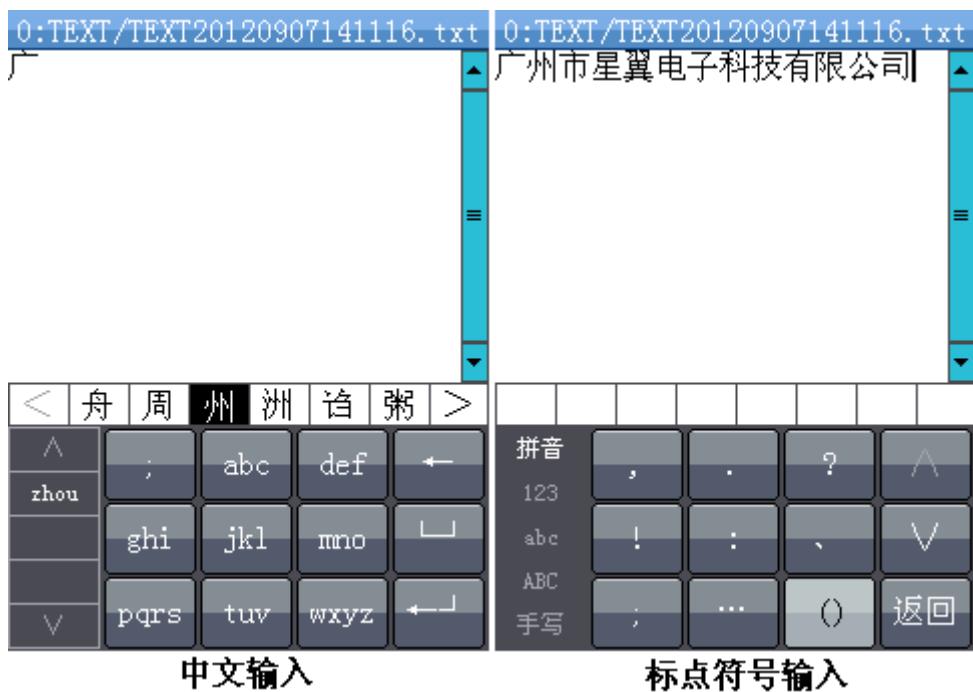


图 61.2.9.2 中文输入和标点符号输入

中文输入就是我们前面 T9 拼音输入法实验的具体运用。该键盘还支持英文输入和手写识别输入，如图 61.2.9.3 所示：

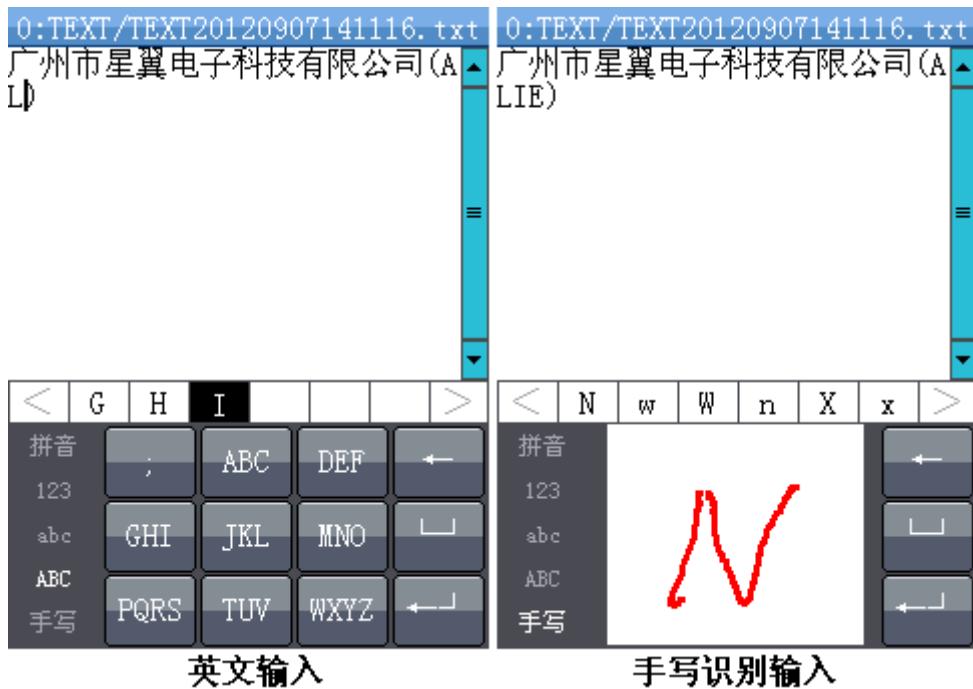


图 61.2.9.3 中文输入和标点符号输入

上图中，左侧的图片为英文输入界面，比较简单；右侧的图片为手写识别的输入界面，这里我们也是用到前面手写识别实验的知识实现的。

只要新建文本文件有被编辑过，那么在返回（按 TPAD 返回）的时候，系统会提示是否保存，如图 61.2.9.4 所示：

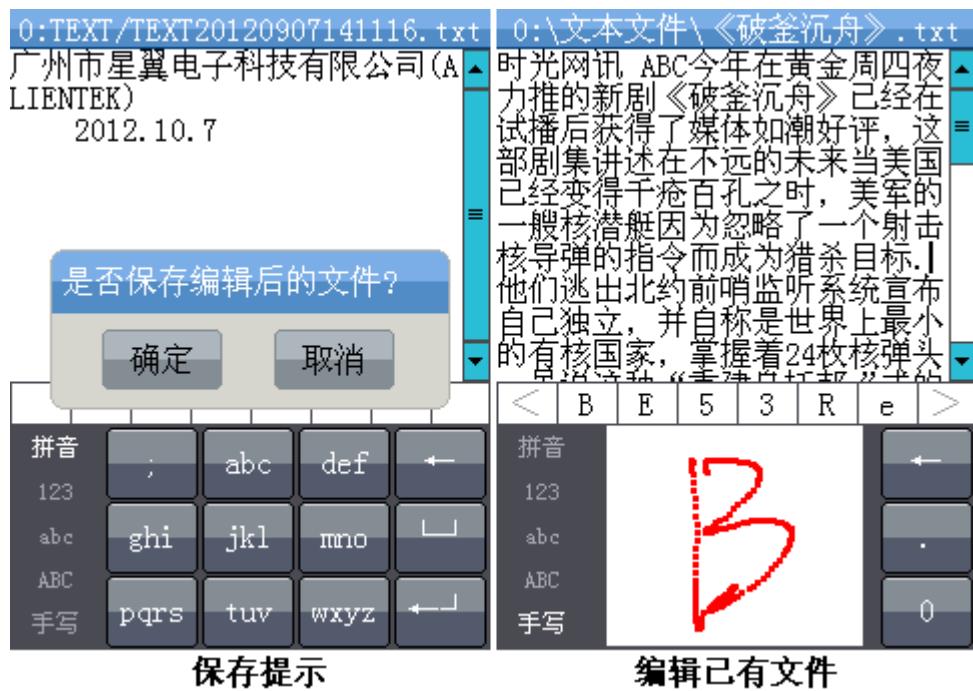


图 61.2.9.4 保存提示和编辑已有文件

上图中，左侧图片为提示保存界面，如果选择确定，该文件将被保存在 SD 卡根目录的 TEXT 文件夹里面。右侧图片为打开已有文件进行编辑的界面，这样我们就可以在战舰 STM32 开发板上编辑.txt/.h/.c/.lrc 文件了。

61.2.10 运行器

双击主界面的运行器图标，首先进入文件浏览界面，如图 61.2.10.1 所示：

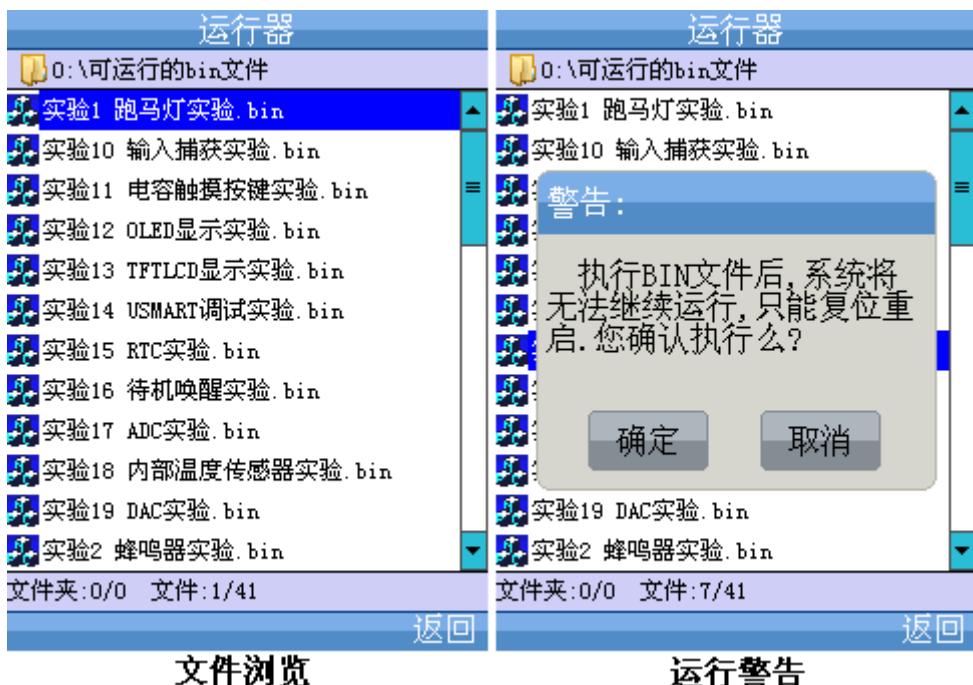


图 61.2.10.1 文件浏览和运行警告

上图中，左侧为文件浏览界面，图中显示了可运行的 bin 文件有 41 个，这些全部来自我们的标准例程。bin 文件的生成办法，请参考串口 IAP 实验这个章节。本运行器支持 60K 字节以内的程序运行（FLASH+SRAM 总共不超过 60K），我们的例程有多达 41 个实验可以直接在运行器里面运行(生成.bin 文件)，我们提供了 SRAM APP 版本的例程，编译后直接生成.bin 文件，拷贝到 SD 卡，即可运行查看实验现象。所有 41 个例程的.bin 文件，我们已单独放到一个文件夹，供大家测试使用。通过运行器，大家可以直接运行我们大部分例程，而不用再去刷代码了，方便大家测试和验证我们的实验。

右侧的图片是运行前的警告界面，因为一旦执行.bin 文件，我们的系统将无法恢复，只能靠复位重启。点击确定之后，STM32 就开始运行你所选择的.bin 文件了，实验现象和对应实验所描述的现象一模一样。之后，

61.2.11 3D

双击主界面的 3D 图标，进入 3D 演示界面，如图 61.2.11.1 所示：

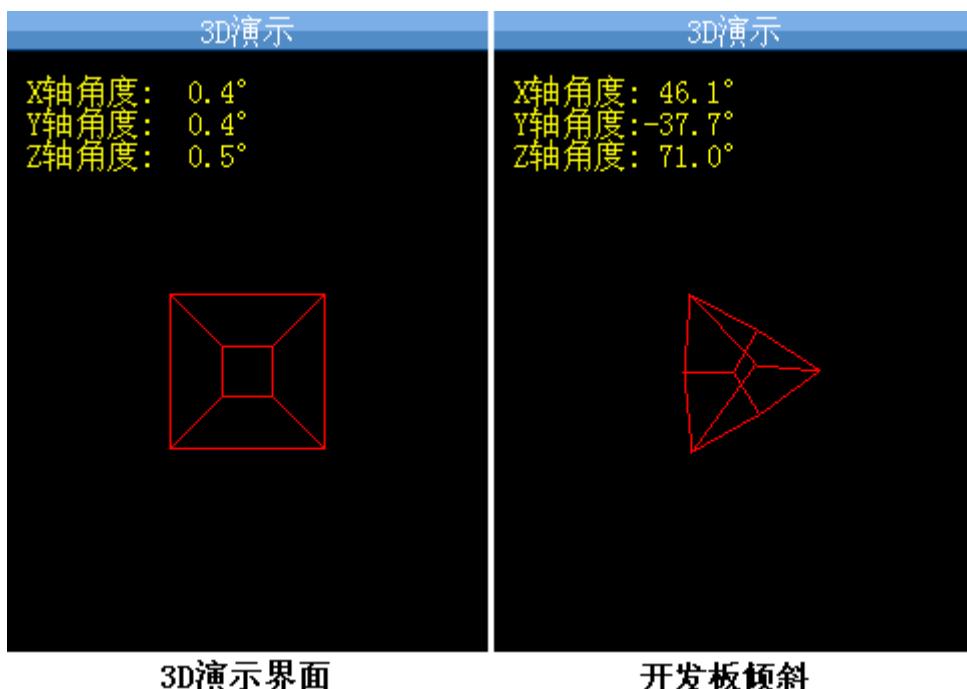


图 61.2.11.1 文件浏览和运行警告

左侧的图片为我们刚进入的是的界面（假设板子是水平放置的），此时可以看到 X/Y/Z 三个方向的角度基本都是 0，屏幕中心为一个立方体图形，该图形会随着角度的变化而变化。右侧的图片，显示了我们后我们把板子倾斜一定角度放置时的情况，可以看到 X/Y/Z 角度都发生了变化，而且立方体图形也产生了变化。

我们还可以通过触摸屏来控制立方体的转动，直接在屏幕滑动即可看到立方体随着我们的滑动而改变方向（视角）。

61.2.12 手写画笔

双击主界面的手写画笔图标，首先弹出模式选择对话框，如图 61.2.12.1 所示：



图 61.2.12.1 模式选择和新建画笔

上图中，左侧图片为我们双击手写画笔后，弹出的模式选择界面，我们可以选择新建画笔，建立一个新的文件；也可以选择打开一个已有的位图进行编辑。右侧的图片为我们新建画笔后输入的内容，默认画笔为最小尺寸，颜色为红色。画笔的颜色和尺寸是可以设置的，按 WK_UP 按键，则弹出画笔设置对话框，如图 61.2.12.2 所示：



图 61.2.12.2 画笔设置和画笔颜色设置

上图中，左侧的图片为按 WK_UP 按键后弹出的画笔设置对话框，我们可以选择对画笔颜色和画笔尺寸进行设置。右侧的图片为画笔颜色设置对话框，在该对话框里面，我们可以直接在颜色条快速输入要设置的颜色，也可以通过下方的三个滚动条进行精确设置，右侧的正方形



区域为预览区。画笔尺寸设置界面如图 61.2.12.3 所示：

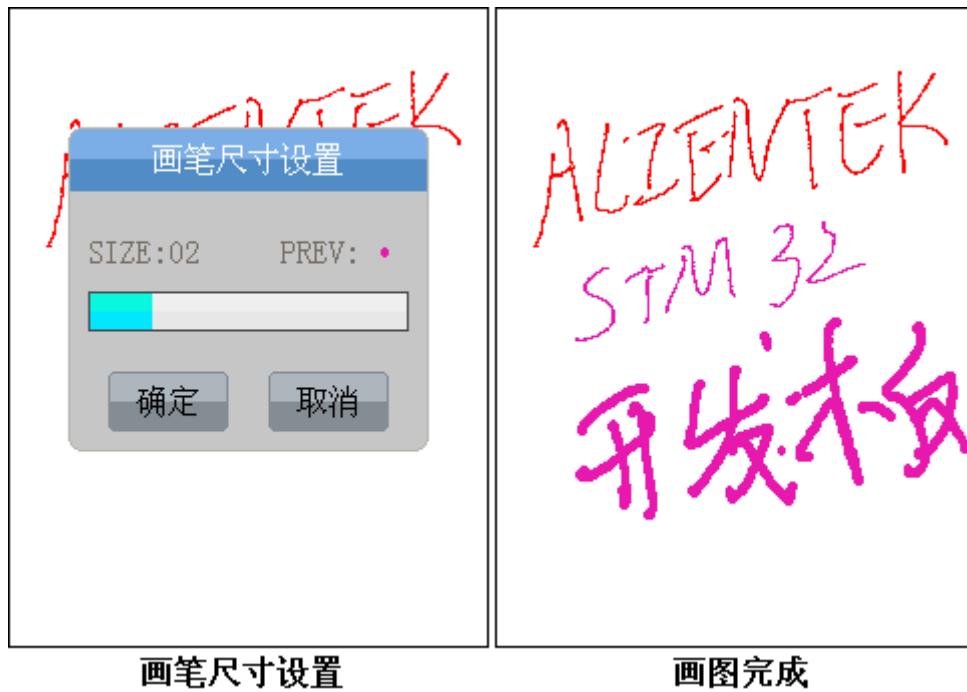


图 61.2.12.3 画笔尺寸设置和完成后的画图

上图中，左侧为画笔尺寸设置界面，我们可以通过滚动条设置画笔尺寸，对话框显示了画笔尺寸和对应的预览图。右侧的图片为我们完成的画图文件，在返回主界面（按 TPAD）的时候，会提示保存，如图 61.2.12.4 所示：



图 61.2.12.4 保存画图和编辑已有位图

上图中，左侧为我们退出时弹出的提示保存对话框，如果选择确定，新的画图文件将会被保存在 SD 卡的 PAINT 文件夹里面，命名方式是以时间命名的，如 PAINT20120907133223.bmp。

右侧的图片为对打开的位图进行编辑的界面，通过这个功能，我们可以在开发板上实现对



一些相片（bmp 格式）进行涂鸦。

61.2.13 照相机

双击主界面的照相机图标，首先初始化 OV7670 摄像头模块，如图 61.2.13.1 所示：



图 61.2.13.1 初始化 OV7670 和等待拍照

在初始化 OV7670 之后，进入等待拍照模式，此时我们可以通过点击屏幕，弹出相机设置对话框，对摄像头的参数进行设置，如图 61.2.13.2：



图 61.2.13.2 相机设置和优先模式设置



在相机设置界面，我们可以对很多参数进行调节。右侧的图片为优先模式设置，支持速度优先和清晰度优先（通过降低帧率实现）两种模式，我们默认的是速度优先模式。

再来看看场景设置和特效设置，如图 61.2.13.3 所示：



图 61.2.13.3 场景设置和特效设置

场景设置支持 5 种常用场景，特效设置支持 6 种特效（不含普通模式），我们可以根据自己的需要选择。

接下来看看亮度设置和色度设置，如图 61.2.13.4 所示：



图 61.2.13.4 亮度设置和色度设置

亮度设置和色度设置各支持 5 个档位调节，我们可以根据自己的需要选择，默认都是 0 的。



最后，看看对比度设置和拍照实现，如图 61.2.13.5 所示：



图 61.2.13.5 对比度设置和拍照

同样，对比度也支持 5 个档位设置，默认为 0。在参数设置好之后，我们按下 WK_UP 按键，就会执行拍照操作，在照片保存期间 DS1 亮，保存完后蜂鸣器发出“滴”的一声，提示拍照成功，同时弹出拍照成功对话框，如上图右侧图片所示。

从上图可以看出，照片文件的命名还是以当前时间为名字命名的。我们将所有的照片都保存在 SD 卡的 PHOTO 文件夹。如果你没有插入 SD 卡，拍照时会提示“创建文件失败，请检查 SD 卡！”的提示信息。

另外，如果你觉得照片模糊，可以手动调节摄像头模块的镜头，进行调焦，以达到最佳效果。

61.2.14 录音机

ALIENTEK 战舰 STM32 开发板综合实验带了录音机功能，可以实现通过 MIC（咪头）录音，并将录音文件保存在 SD 卡。录音文件为 WAV 文件，格式为：单声道、16 位、8Khz 采样率，1 秒钟需要的数据空间为 16K 字节，如果录音 100 秒钟，则需要 1.6M 左右的空间。

双击主界面的录音机图标，进入录音机主界面，如图 61.2.14.1 所示，该界面显示了当前录音时间以及信号电平等，在该界面有两个按钮：左边的按钮用于开始/暂停录音，右边的按钮用于停止录音，并保存当前录音文件。

录音机功能可以设置 MIC（咪头，这里称之为麦克风）增益，通过点击做小脚的选项，系统将弹出麦克风增益设置对话框，增益设置范围为 0~15，0 代表自动增益，默认设置 AGC 为 4，如图右侧图片所示。

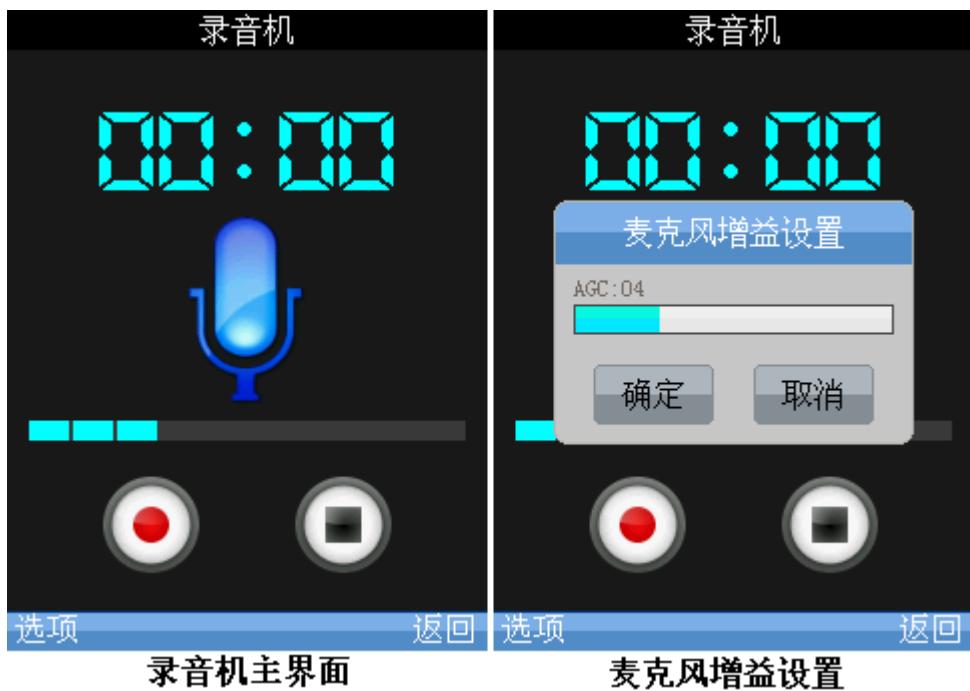


图 61.2.14.1 录音机主界面和麦克风增益设置

我们在录音机主界面点击录音按钮，则开始录音，如图 61.2.14.2 所示：



图 61.2.14.2 录音进行中和提示保存

上图中，左侧的图片为正在录音的界面，此时我们可以按暂停/停止，按停止则自动保存当前录音文件，录音文件同样是以时间命名（见图中上方白字），所有录音文件都是被保存在 RECORDER 文件夹里面的。

在录音的时候，按下 TPAD，会提示是否保存，如上图右侧图片所示，我们可以根据需要选择。



61.2.15 USB 连接

双击主界面的 USB 连接图标，如果开发板的 USB 端口没有连接电脑，则显示无连接，如图 61.2.15.1 所示：

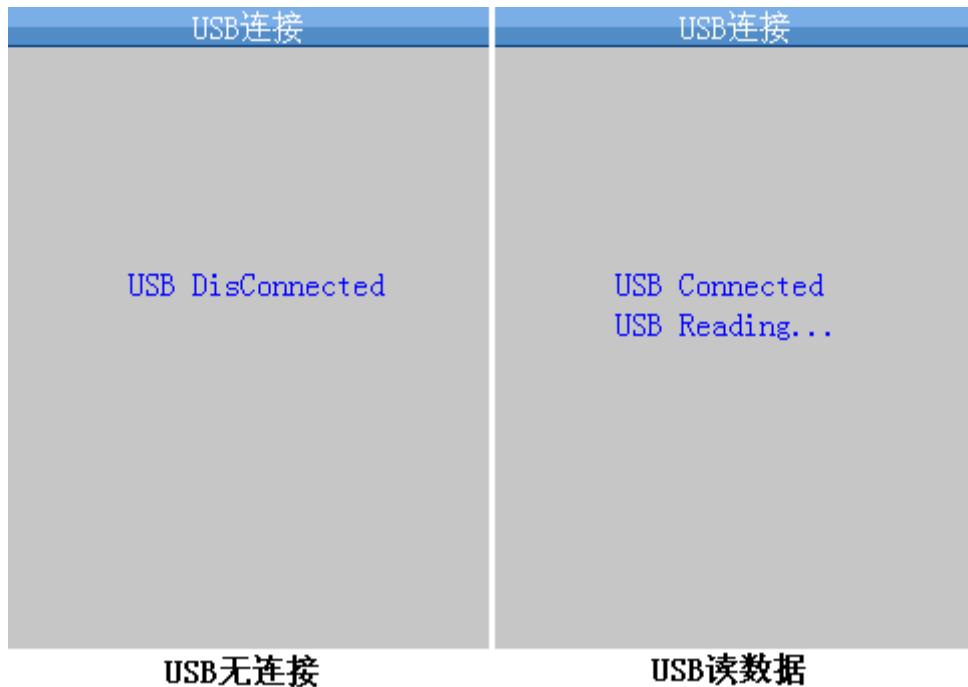


图 61.2.15.1 USB 无连接和 USB 读数据

上图中，左侧的图片显示开发板没有和电脑连接上，此时，我们找一根 USB 线，连接开发板的 USB 端口和电脑的 USB，注意 P13 端口必须设置为：PA11 接 D+，PA12 接 D-。此时，可以看到开发板提示 USB 已连接，并显示 USB 正在读数据，同时我们在电脑上面，可以看到右下角提示发现新硬件（如果是第一次连接的话），如图 61.2.15.2 所示：



图 61.2.15.2 电脑发现新硬件

此时，我们打开我的电脑，即可找到 2 个可移动磁盘，分别为开发板的 SD 卡和 FLASH Disk。这样，我们就实现了开发板和电脑的 USB 连接，可以直接从电脑拷贝文件到开发板的 SD 卡或者 FLASH Disk（即 W25Q64）。

这里再次提醒大家，如非必要，不要往 FLASH Disk 写入数据！否则容易写坏 SPI FLASH。



61.2.16 TOM 猫

这是一个现在在智能手机非常流行的游戏，你说一句话，游戏里的猫也跟着说一句，而且是以怪怪的音调（变调）模仿，十分有意思。

双击主界面的 TOM 猫图标，进入如图 61.2.16.1 所示界面：



图 61.2.16.1 TOM 猫主界面和增益及语速设置界面

上图中，左侧图片为 TOM 猫游戏的主界面，图中显示了一个小猫和信号电平指示，此时我们可以对着 MIC（咪头）说话，你说一句，就可以从耳机（插开发板的耳机接口）或者收音机（开启开发板的 FM 发射，并设置收音机的频率为开发板的 FM 发射频率）里面听到 TOM 猫在重复你的句子，而且是以变调重复的，听起来和手机的 TOM 猫游戏差不多。

我们的 TOM 猫游戏还加入了语速设置，点击左侧图片里的选项按钮，可以弹出增益及语速设置对话框，如右侧图片所示。在这个对话框里面，我们可以设置增益（AGC）和语速（SPEED），增益设置范围为 0~10，建议设置在 4 左右为最佳。

语速设置范围为 4000hz~16000hz，这里我们实现变调的原理很简单，就是人为改变 wav 文件的采样率，我们 wav 录音默认采样率为 8Khz，而如果我们强制修改采样率为其他值，那么语调就肯定发生了变化，我们通过将采样率设置为不同的值得到不同的语调，如果设置为 8Khz，就是正常语调了。默认我们设置语调为 13000Hz，这个语调比较接近手机的 TOM 猫效果，大家可以修改为其他值，比如设置为 4000Hz，听起来就像个老人的声音。

TOM 猫就给大家介绍到这里。

61.2.17 无线传书

该功能用来实现两个开发板之间的无线数据传输，在开发板 A 输入的内容，会在开发板 B 上完整的“复制”一份，该功能需要 2 个开发板（可以战舰板和 Mini 板[实验 28]搭配用）和 2 个 NRF24L01 无线模块。

双击主界面的无线传书图标（假定开发板已插上 NRF24L01 无线模块），会先弹出模式选择对话框，如图 61.2.17.1 所示：



图 61.2.17.1 模式选择和发送模式界面

从左侧的图片可以看出，模式设置，我们可以设置为发送模式或接收模式。右侧的图片则是选择发送模式后进入的界面。我们在另外一块开发板（开发板 B）设置模式为接收模式，然后在本开发板（开发板 A）手写输入一些内容，就可以看到在另外一个开发板也出现了同样的内容，如图 61.2.17.2 所示：



图 61.2.17.2 在开发板 A 输入的内容完整的显示在开发板 B 上

从上图可以看出，在开发板 A 上输入的内容，被完整的复制到开发板 B 上了。这就是无线传书功能。



61.2.18 计算器

战舰 STM32 开发板实现了一个简单的科学计算器，可以计算加减乘除、开方、平方、 M^N 次方、正弦、余弦、正切、对数、倒数、格式转换等一些常见的计算器功能，精度为 12 位，支持科学计数法表示。双击主界面的计算器图标，进入计算器主界面，如图 61.2.18.1 所示：



图 61.2.18.1 计算器主界面和加法计算

上图中，左侧的图片为科学计算器的主界面，和我们手机用的计算器基本一样，使用上非常简单，我们就不详细介绍。右侧的图片为加法计算，支持累加功能。



图 61.2.18.2 计算器主界面和加法计算



上图为乘法计算和倒数计算，可以看到，结果是以科学计数法表示的，最大支持 200 位指数表示，超过范围直接显示错误 (E)。

该计算器还支持格式转换（按 FMT 键），可以将十进制数据（最大为 65535，超过部分将被丢弃）转换为 16 进制/二进制数据表示，如图 61.2.18.3 所示：



图 61.2.18.3 格式转换

上图显示我们将十进制的 65535 转换为 16 进制/二进制后的表示。计算器的其他功能，我们就再列举了，感兴趣的朋友可以慢慢摸索，当然也可以在这个基础上进行改进。通过按 TPAD 可以返回主界面。

至此，整个战舰 STM32 开发板的综合测试实验就介绍完了。这就是我们开发了近两年的东西，其中借鉴了很多网友的代码，在此，对这些网友表示衷心的感谢，同时我也希望我们的这个代码，可以让大家有所受益，能开发出更强更好的产品，如此，我们的努力也就没有白费。

综合实验整个代码编译后大小为 275K 左右，代码量是很大的，希望大家慢慢理解，各个攻破，最后祝大家身体健康、学习进步！